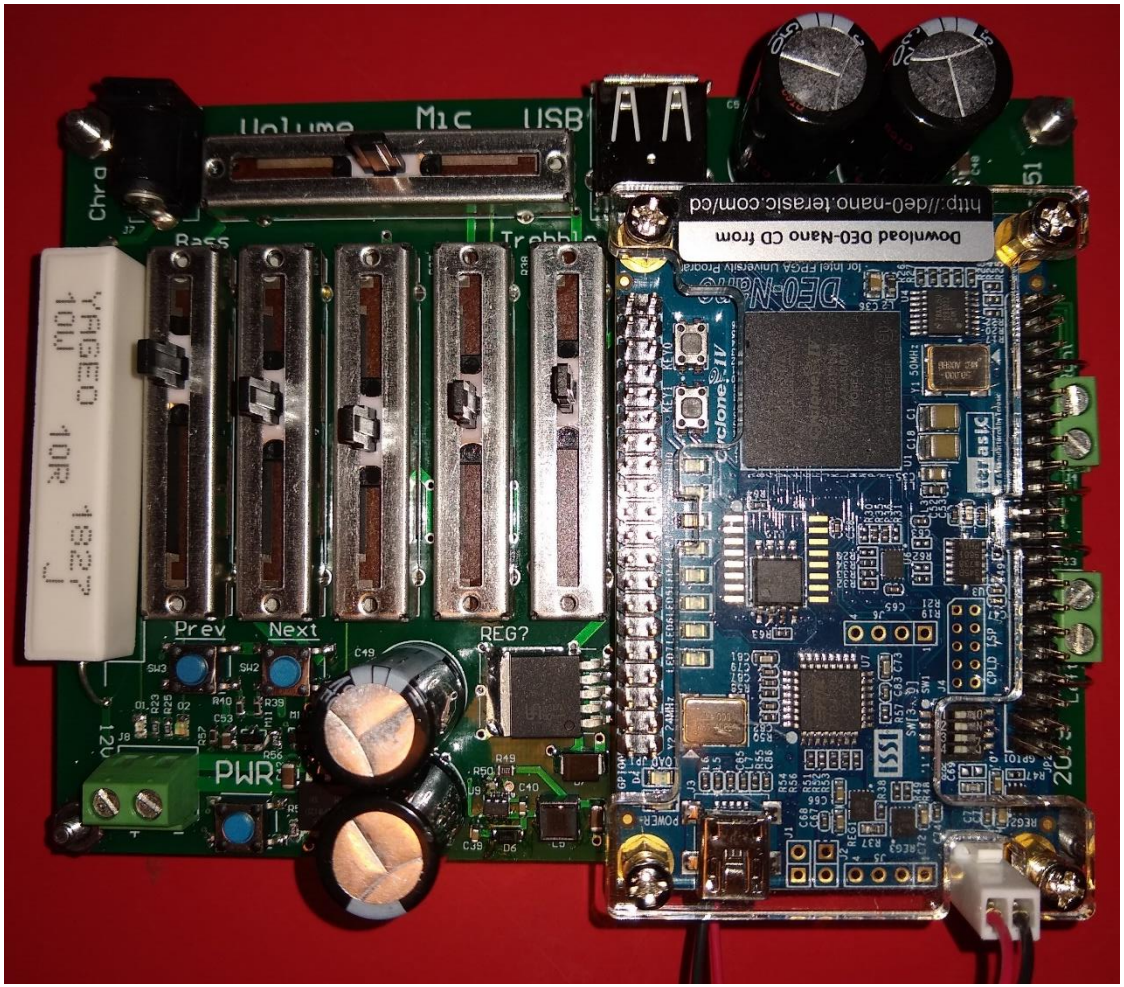
ECE 551 Project Spec



Spring ’23 **Audio Equalizer**

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

- Project Grading Criteria:
  - Quantitative Synthesis Element 15%

  (yes this could result in extra credit)

→ $\text{Quantitative} = \frac{\text{Eric\_ProjectArea}}{\text{YourSynthesizedArea}}$

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

Note: Minimum synthesized frequency to the saed32 library is 333MHz. No benefit to faster than 333MHz.

- Project Demo (85%)
  - ✓ Code Review (12.5%)
  - ✓ Testbench Method/Completeness (15%)
  - ✓ Synthesis Script review (7.5%)
  - ✓ Post-synthesis Test run results (10%)
  - ✓ Results when placed in our Testbench (22.5%)
  - ✓ Test of your code mapped to the actual equalizer and tested. (10%)
  - ✓ Teammates judgement of your contribution (7.5%)

Extra Credit Opportunity:

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

- Run code coverage on a single toplevel test and get 1% extra credit
- Run code coverage across your test suite and get a cumulative coverage number and get another 1% 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 another 1% extra credit.

Project Due Date

- Project Demos will be held in B555:
  - Sat (4/29) from 6:30PM till evening. (2.25% bonus)
  - Sun (4/30) from 6:30PM till evening. (1.75% bonus)
  - Mon (5/1) from 1:30PM till evening. (1.25% bonus)
  - Tues (5/2) from 6:30PM till evening. (0.75%) bonus
  - Weds (5/3) from 1:30PM till evening. (due date)

- Project Demo Involves:
  - ✓ Code Review
  - ✓ Testbench Method/Completeness
  - ✓ Synthesis Script & Results review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in EricHarish testbench
  - ✓ Run of your code on the Equalizer Audio Platform.

Test Platform Board

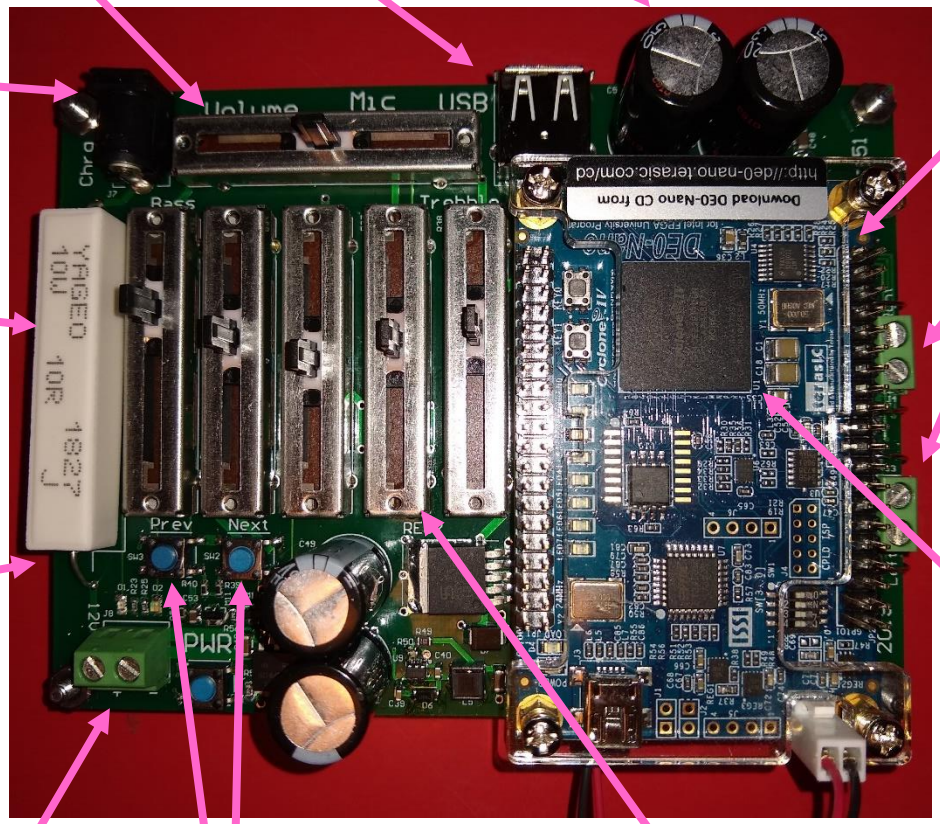
Slide Potentiometer for volume control

2A capable USB charge port for phone

Electrolytic caps on 24V supply for peak demand

40W Class D amplifier circuits "live" underneath the DE0-Nano

DC-barrel connector for external power (re-charge batteries)



Hook your speakers up here

This large power resistor is part of the battery recharge circuit

RN-52 Bluetooth audio module "lives" underneath board on this side.

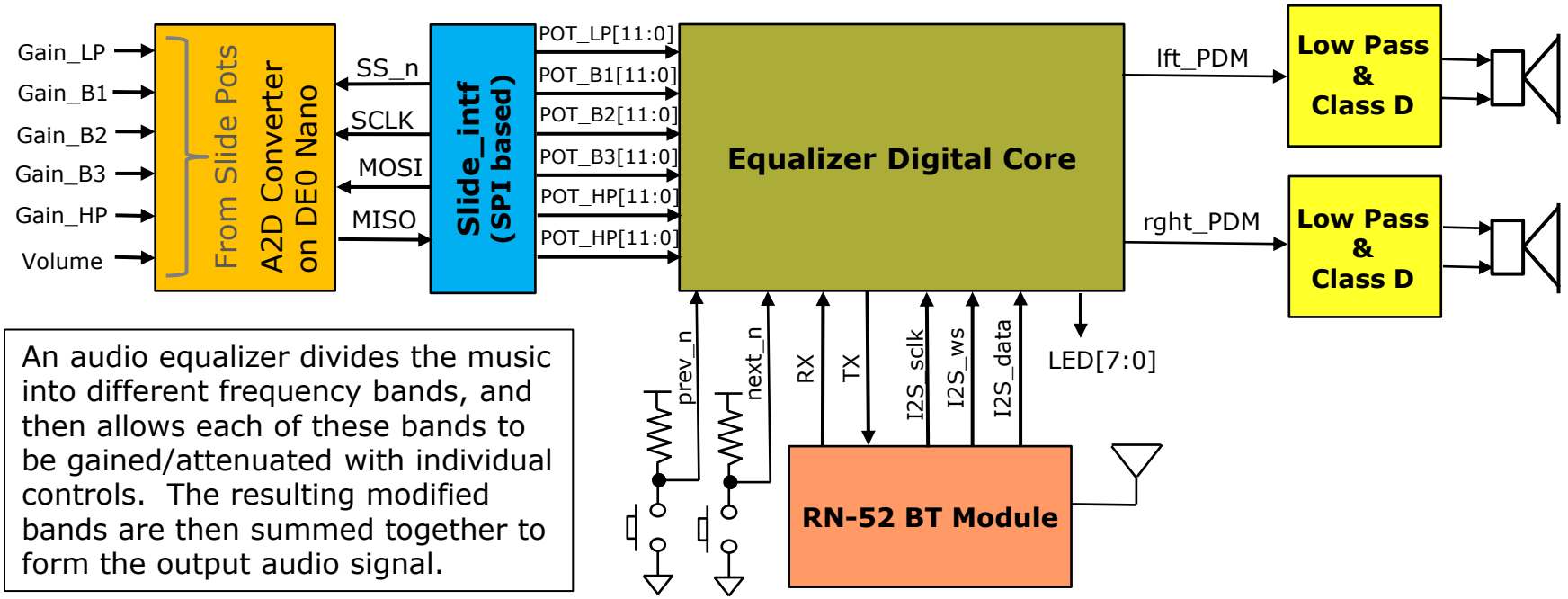
DE0 Nano with Altera Cyclone FPGA. The "intellectual property" of this design.

12V to 24V DC input from batteries

Play previous song or skip to next song via button pushes

Slide pots for scaling of the 5-bands bass thru treble

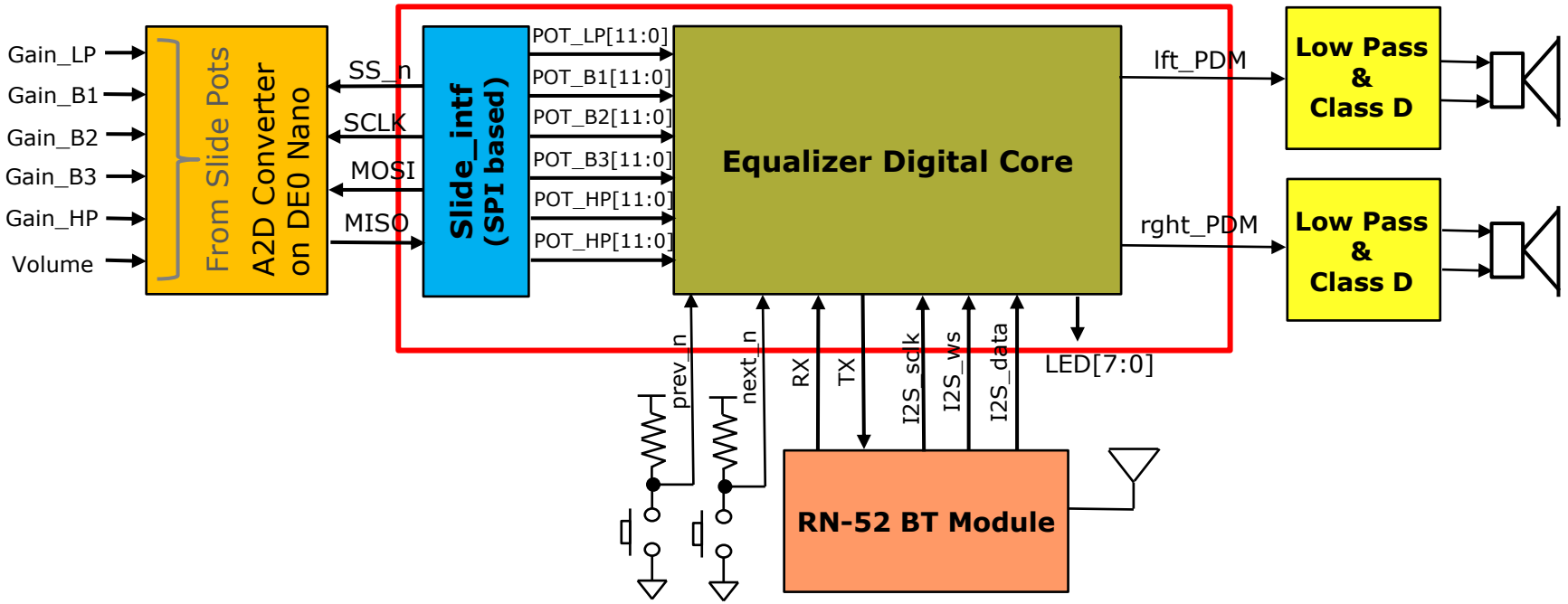
Block Diagram of Digital Portion



An audio equalizer divides the music into different frequency bands, and then allows each of these bands to be gained/attenuated with individual controls. The resulting modified bands are then summed together to form the output audio signal.

The audio processing will be done in the digital domain. The incoming audio stream will come from a Bluetooth module on the I2S protocol. The stream broken into 5 bands via DSP FIR filters. Each band will have a different gain term (as read via an A2D from a slide potentiometer) applied, and the resulting modified digital bands will be summed to form a new left/right audio signal. That resulting audio signal will be converted to PDM bits streams that will in turn go to low pass filters and then a class D amps to drive the speakers. Volume control is also performed by multiplying the summed audio signal for left/right channel by a gain number controlled by a sixth slide potentiometer. Finally there are two buttons on the board that can be used to skip to the next song, or repeat the previous song.

What is synthesized DUT vs modeled?

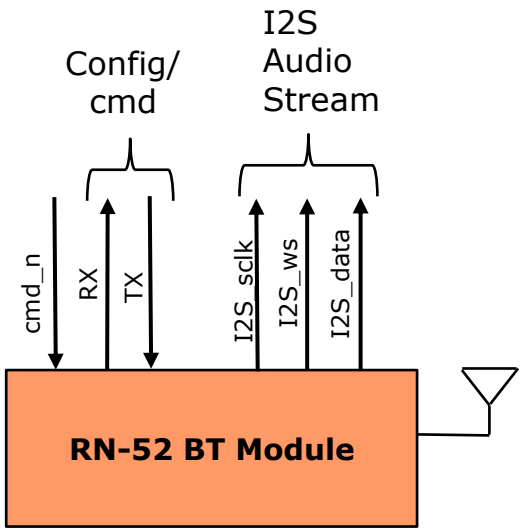


The blocks outlined in red above are pure digital blocks, and will be coded with the intent of being synthesized via Synopsys to our standard cell library. For practical purposes we will also map that logic to FPGA so we can run the demos.

You Must have a block called **Equalizer.sv** which is top level of what will be the synthesized DUT.

RN-52 BT Audio Module

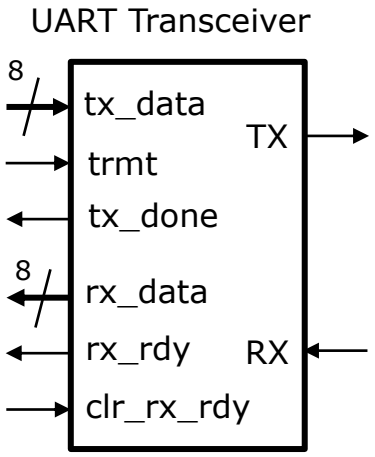
- The RN-52 is a high resolution stereo Bluetooth audio module. It is capable of pairing with phones and ipods using the A2DP profile.
- The RN-52 has a few different output formats for audio, including analog. However we wish to use a serial streaming digital format called I2S (**I**nter **I**C **S**ound)
- The default audio output format is not I2S, so we have to configure the RN-52. This is done via a UART interface, and is covered in the follow slides. There are also a few commands we send to the RN-52 during normal operation to replay, pause, or skip the song.
- When the RN-52 starts steaming audio it is the ruler *(monarch)* of the I2S bus and is driving **sclk**, **ws**, and **data**. We are the serf and have to receive and decode the serial stream into 24-bit left channel and 24-bit right channel data.



ws = **w**ord **s**elect (differentiates left/right data)

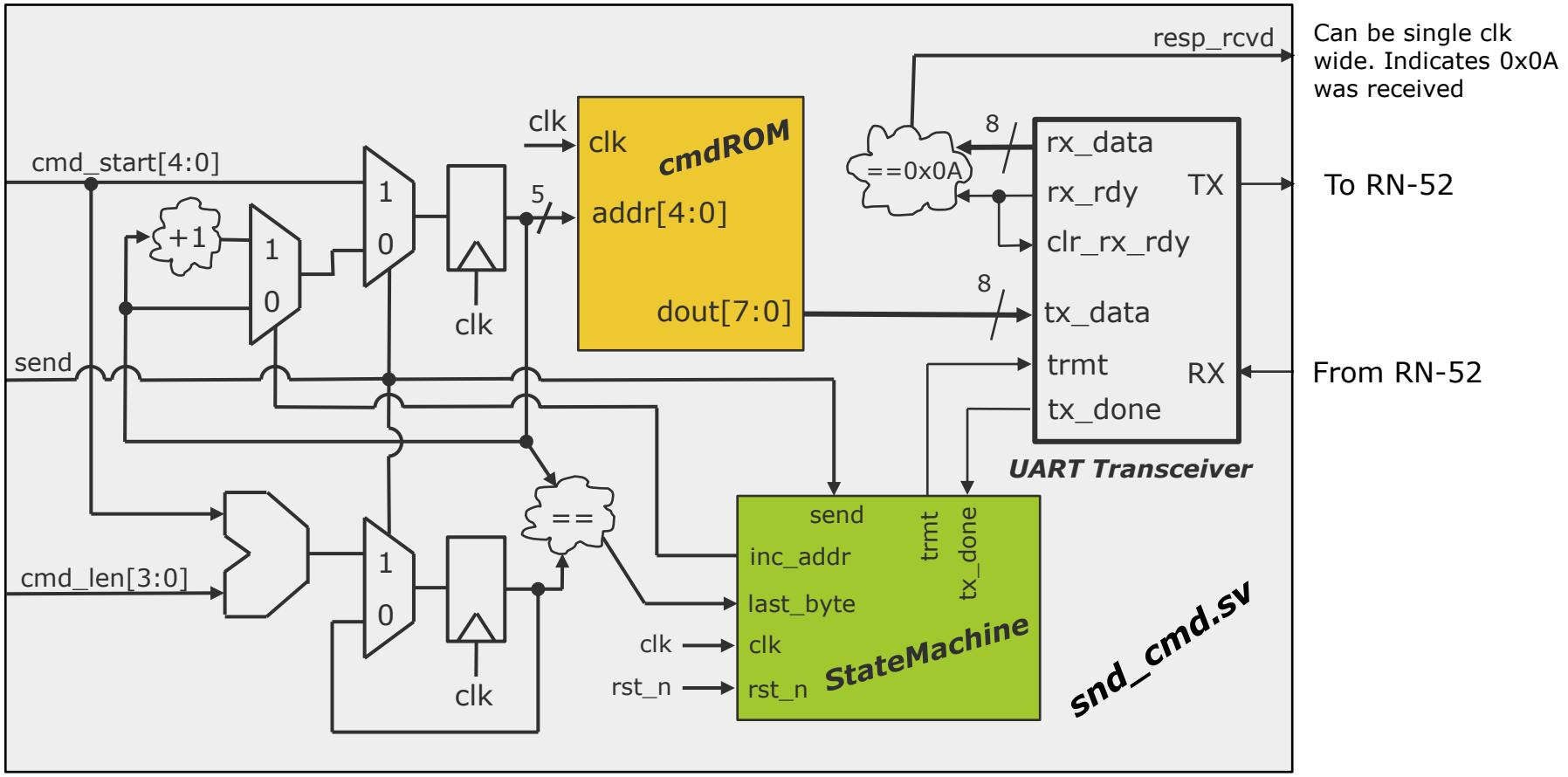
RN-52 BT Audio Module (configuration)

- The RN-52 is configured via a 8-bit serial protocol called UART. Code for an 8-bit UART transceiver is provided on the Canvas page (**UART.sv**)
- We need to send a few ASCII strings to the RN-52 after power up to configure it as we desire. Each string we send the RN-52 will be acknowledged by the RN-52 responding with a 0x0A.



String:	Length:	Description:
First thing is to hold the **cmd_n** pin to the RN-52 high for 2^16 clocks after **rst_n** deasserts. Then raise **cmd_n** and wait for a 0x0A response from the RN-52. Then configuration strings can be sent.		
"S|,01\r"	6	A send of: S|,01 with carriage return switches output mode of RN-52 to I2S. Wait for 0x0A.
"SN,ECE551\r"	10	This sets the advertising name to "ECE551" wait for 0x0A
At this point configuration is done and the SM controlling the UART interface to the RN-52 should go into a loop waiting for either a ***next*** or ***prev*** song from the button interface. If one of these signals is received it will send "AT+\r" or "AT-\r" respectively.		

RN-52 BT Audio Module (configuration)



- A model of a ROM (**cmdROM.v**) is provided. Its contents are read from a file (**cmd_hex.txt**). This file contains the arbitrary characters that make up the commands we need to send to the RN-52.

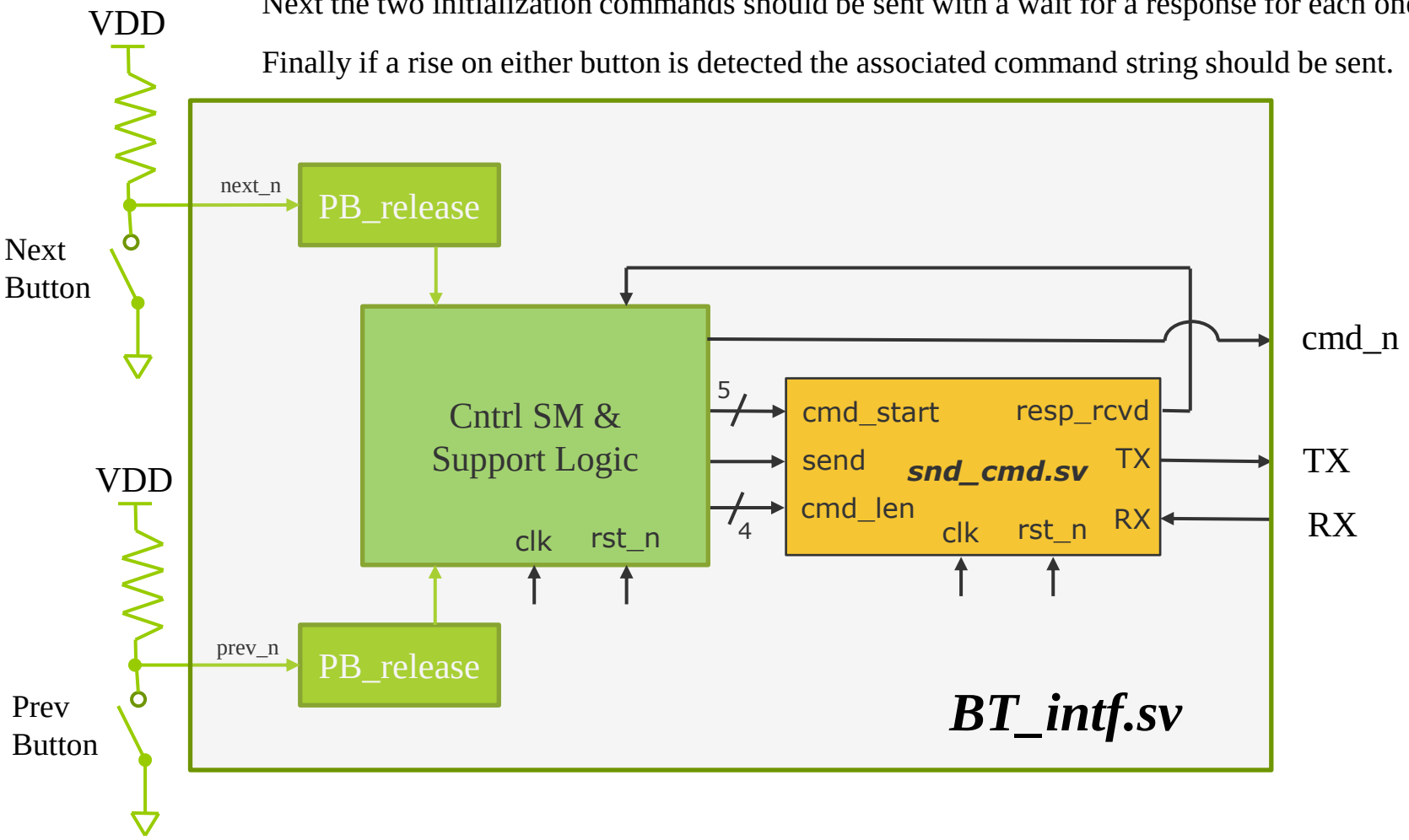
- One asserts **cmd_start** to be the address of the first character of the command to send. At the same time **cmd_len** will hold the length of the command in bytes. The **send** signal is asserted. The SM sends the bytes waiting for **tx_done** between each. When it gets to the **last_byte** it is done. The block provides detection of 0x0A reception.

RN-52 BT Audio Module (next & prev buttons)

After reset **cmd_n** is held high until a 17-bit timer expires. Then **cmd_n** is lowered. The lowering of **cmd_n** will produce a response from the RN-52.

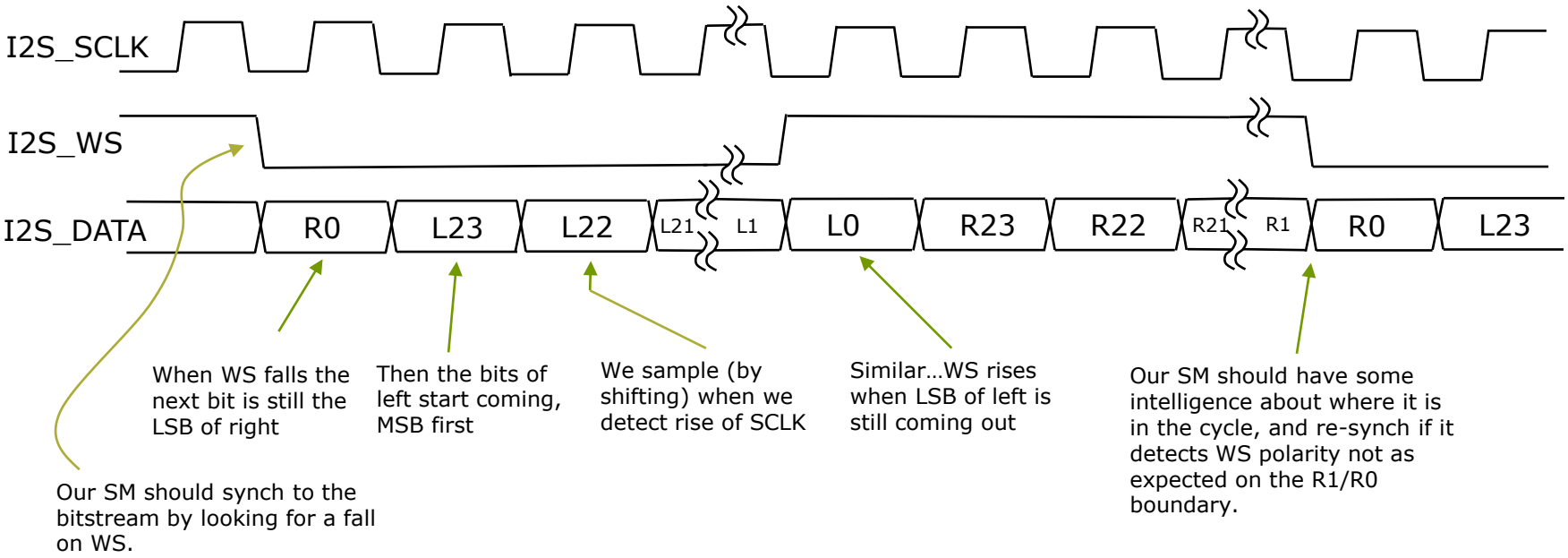
Next the two initialization commands should be sent with a wait for a response for each one.

Finally if a rise on either button is detected the associated command string should be sent.



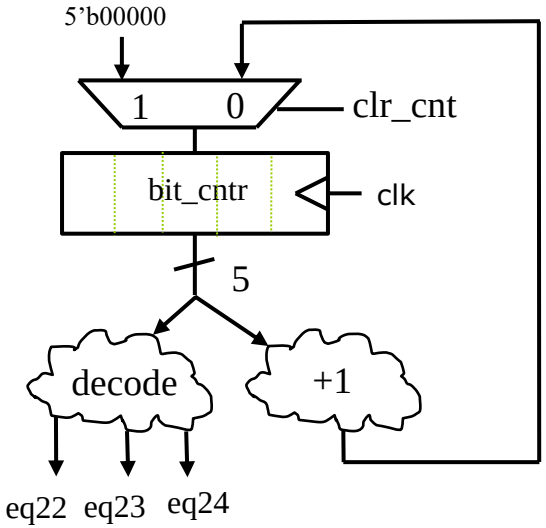
What is I2S (**I**nter **I**C **S**ound)

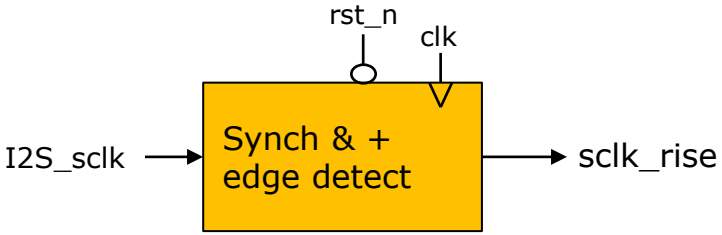
- I2S is the protocol the RN-52 BT Module will use to output audio. It is quite similar to SPI in many ways, except this time we are the serf and the RN-52 is the monarch.

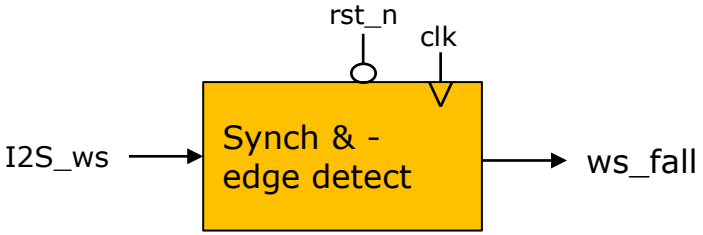


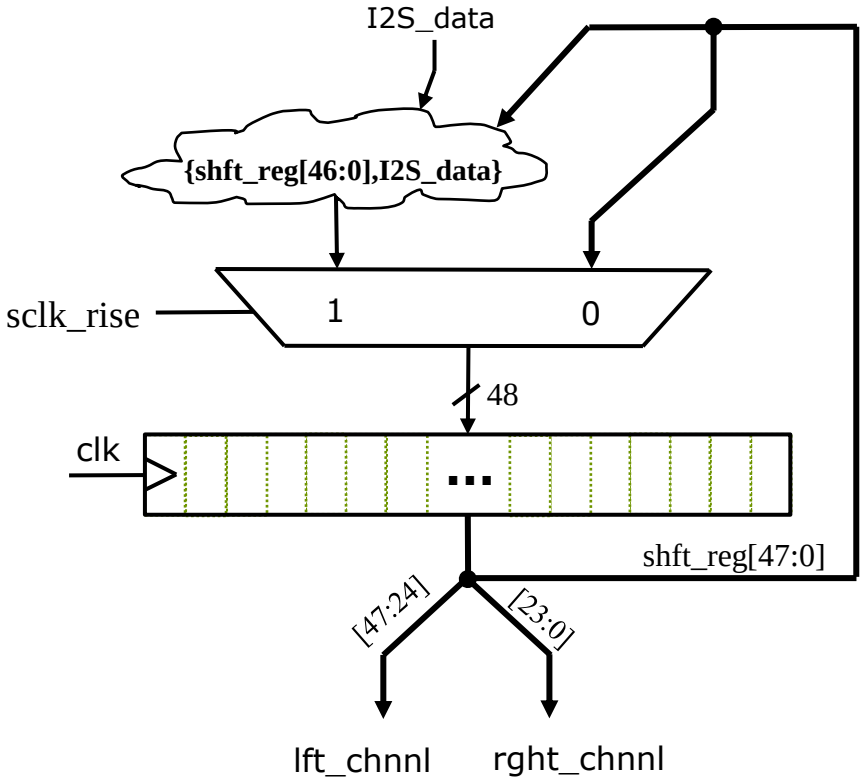
- The protocol is 24-bit left/right audio data at 44.1kHz…**SCLK** freq is: 24*2*44.1k = 2.11MHz
- The **WS** line is generally low during left data, and high during right…except it leads by 1 bit.
- We would sample the data by shifting our shift register(s) when we detect **SCLK** rise.
- We initially synched to the stream by looking for the fall of **WS** then ignoring the next **SCLK** rise, and then start shifting in left data on **SCLK** rise.
- When we know we are at ***R1*** we should check that **WS** is still high. When we know we are at ***R0*** we should check that **SCLK** is low. If this is not true we should re-synch. Otherwise we would go right back to sampling left data after ***R0***.

Possible Architecture of I2S_Serf



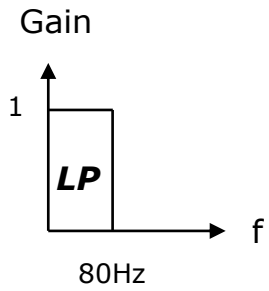




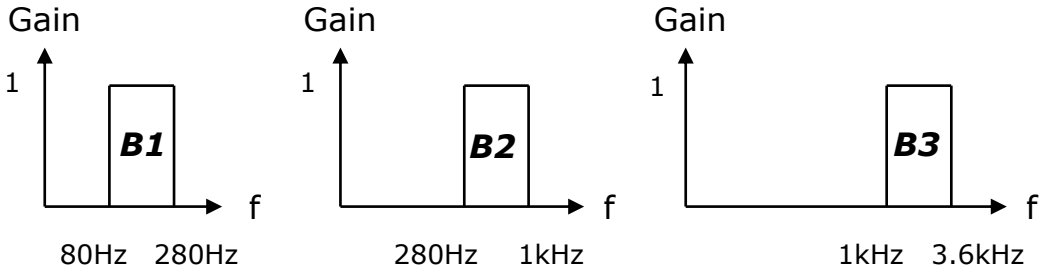


Also need a SM to control it, and assert when the data {**lft_chnnl**,**rght_chnnl}** is **vld**. **vld** can be a single system clock cycle wide. **lft_chnnl** and **rght_chnnl** will get buffered (captured) in registers) elsewhere.

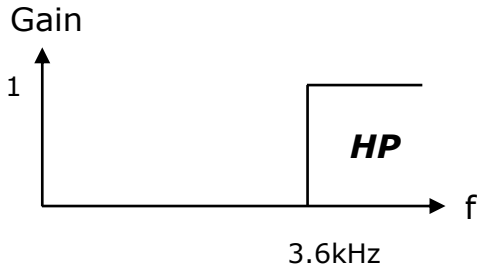
Filtering our Left/Right Audio Signal Into Bands



Bass is a low pass with a 80Hz cutoff. All frequencies below 80Hz are passed with little to no attenuation. All frequencies above are attenuated

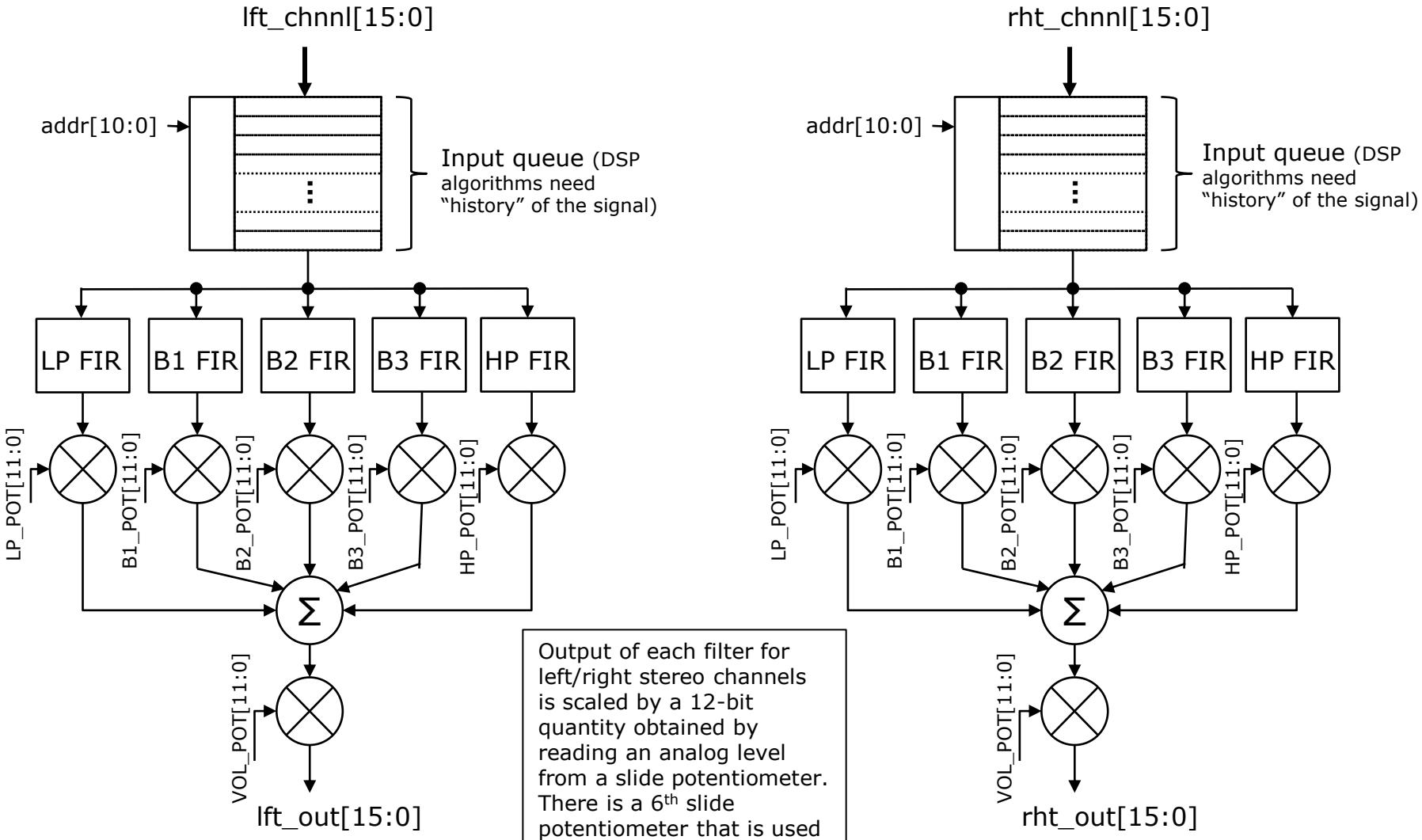


The 3 mid band filters are bandpass filters which pass signals unattenuated in their bands, and attenuate outside their bands.



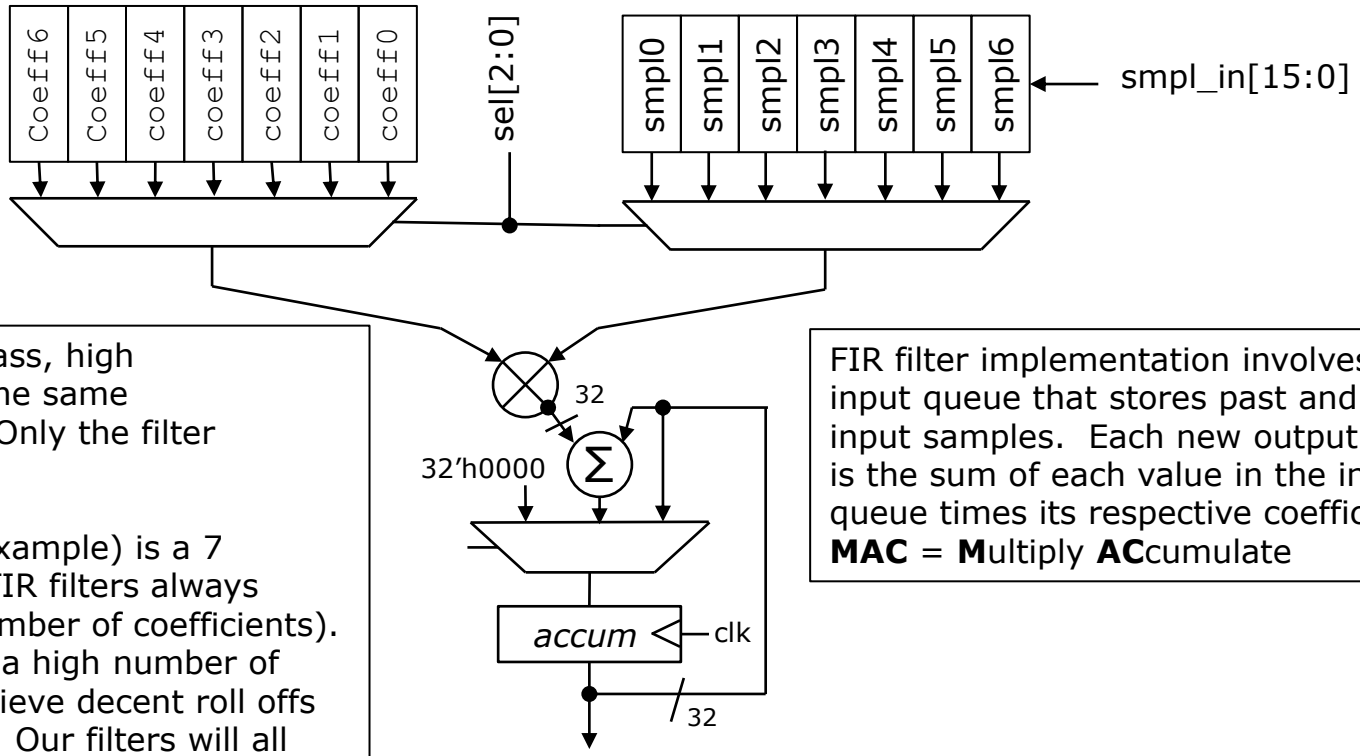
Treble is a high pass with a 3.6kHz cutoff. All frequencies above 3.6kHz are passed with little to no attenuation. All frequencies below are attenuated.

What We Want to Build (Digital Core)



Output of each filter for left/right stereo channels is scaled by a 12-bit quantity obtained by reading an analog level from a slide potentiometer. There is a 6th slide potentiometer that is used for volume control.

DSP FIR Filters (LP, BP,HP…there all the same, just different coefficients are used)



Low pass, band pass, high pass…they're all the same implementation. Only the filter coefficients differ.

Shown here (as example) is a 7 coefficient filter (FIR filters always require an odd number of coefficients). FIR filters require a high number of coefficients to achieve decent roll offs and attenuations. Our filters will all use 1021 coefficients.

FIR filter implementation involves an input queue that stores past and present input samples. Each new output sample is the sum of each value in the input queue times its respective coefficient. **MAC** = **M**ultiply **AC**cumulate

smpl_out[15:0] = bits [30:15] of accum

The multiply is signed 16-bit sample times signed 16-bit coefficient to produce a signed 32-bit product. Results are accumulated in a 32-bit register . Once all coefficients have run through the MAC the result is bits [30:15] of the accumulate register. **NOTE:** The diagram above is for illustrative purposes. The coefficients will be stored in a ROM. A dual port RAM with head and tail pointer will be used to implement a circular queue for the sample queues. Using a series of flops in a shift register with a mux as shown would be extremely wasteful, and not even practical for a FIR filter with 1021 coefficients.

DSP FIR Filters (Sample Rates and Queue Sizes (number of coefficients))

Filter:	Range:	Queue Size:	Sampling Freq:	ROM Model
LP	80Hz & Below	1024	22050Hz	ROM_LP.v
B1	80Hz to 280Hz	1024	22050Hz	ROM_B1.v
B2	280Hz to 1kHz	1024+512	44100Hz	ROM_B2.v
B3	1kHz to 3.6kHz	1024+512	44100Hz	ROM_B3.v
HP	3.6kHz & above	1024+512	44100Hz	ROM_HP.v

The various FIR filters all utilize 1021 coefficients, keeping things on a power of 2 we will use a 1024 entry DP RAM. The lower 2 bands (LP & B1) run at a 22.05kHz sample rate, while the upper 3 bands run at a 44.1kHz sample rate.

ROM models and their associated coefficients are provided for storing the various filter coefficients. These ROM models synthesize and map just fine to the DE0-Nano board.

The next question is how to implement the queues most efficiently. Obviously we need a queue with 1024 entries for both left/right channels. However the upper three filters (B2,B3 & HP) are operating at a higher sample frequency. Hmm…what does this mean?

FIR filters have a group delay. Their output is the filtered input delayed by half the queue depth. So LP & B1 have a group delay of 510/22050 = 23.1ms. B2,B3 & HP have a group delay of 510/44100 = 11.56ms.

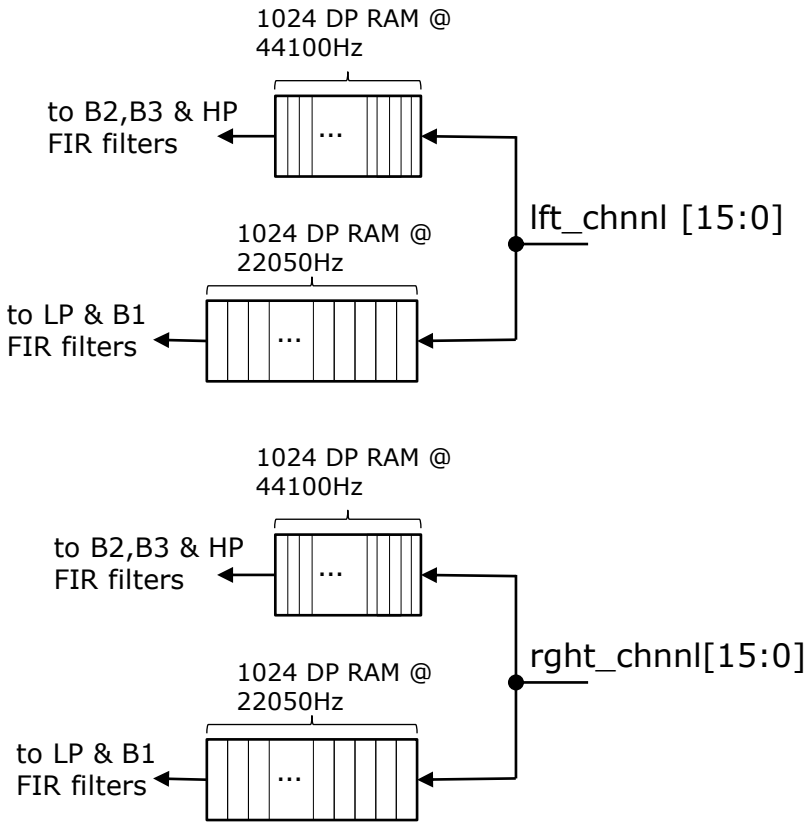
Queue Implementation (Does not consider group delay)

One might be tempted to have two different queues with 1024 entries each. One queue sampling every sample from the RN-52 (at 44.1kHz) and one sampling at half rate (22.05kHz). Each queue (high frequency/low frequency) will contain two dual port RAMs, one for left and one for right.

This approach is close to what we want, however, it does not consider the group delays. Would it sound bad to combine lower frequencies delayed by 23.1ms with upper frequencies that are only delayed by 11.56ms? Would the human ear pick up on that "distortion" of the music? I don't know, and I don't intend to find out. We are doing this right.

NOTE: As mentioned these queues will be implemented using Dual Port RAMS (one read port and one write port). They need to be implemented as circular queues.

NOTE: Everything in our design will run from a 50MHz system clock. You **will not** be generating a 44100Hz clock and using that to control a queue. The high freq queues that sample at 44100Hz will be a queues that write a new entry every time the RN-52 provides a new **vld** lft/rght, and the queues low freq queues will be queues that write a new entry every other **vld** from the RN-52.



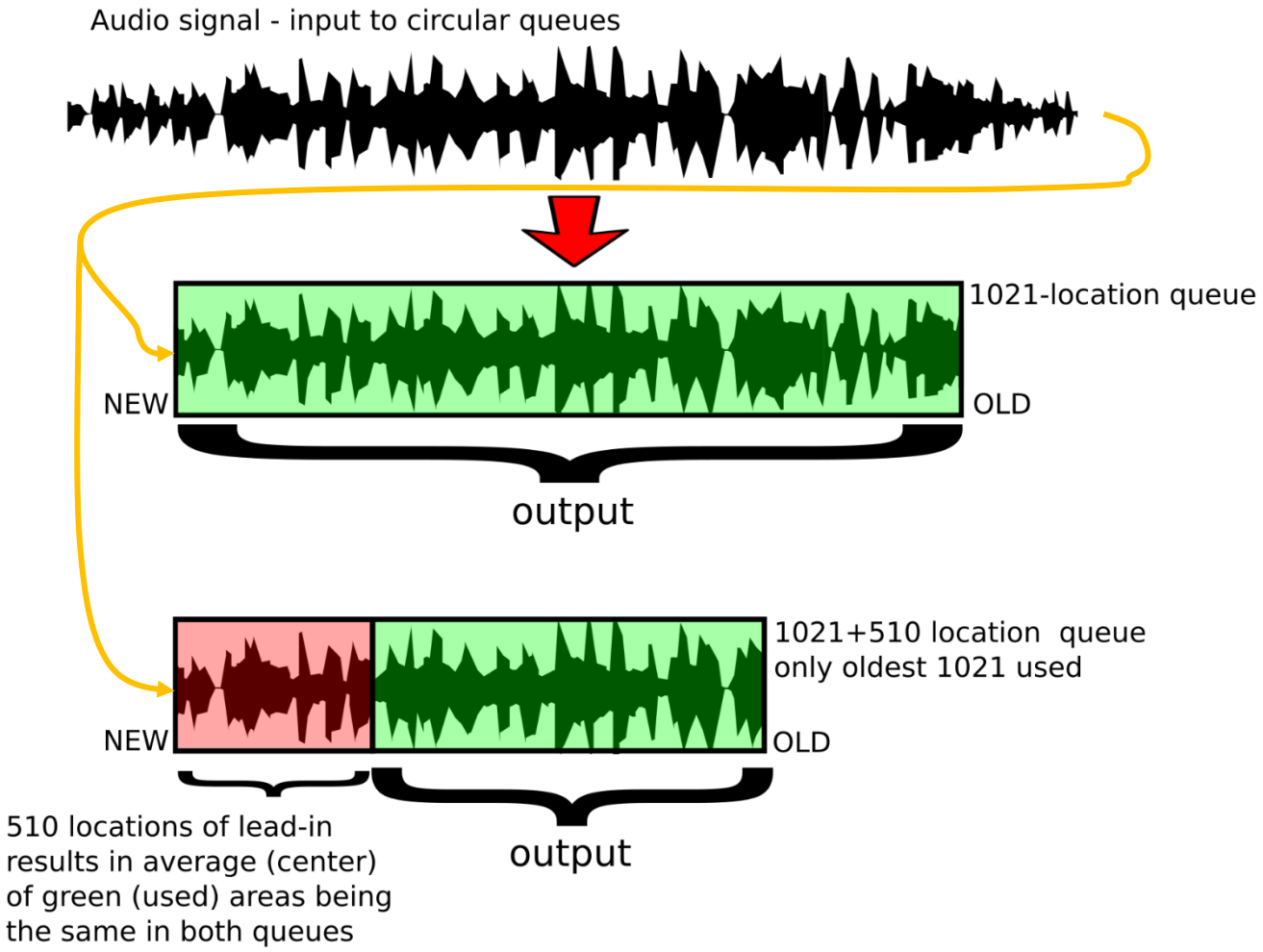
Queue Implementation (why high freq queue needs an extra 510 entries for group delay)

Image courtesy of Christian Zimonick:

low freq queue shown wider since it spans more time (written at half the speed).

Music for both channels generated from the middle of the groups (middle of the used samples).

To line up (in time) the outputs of low freq and high freq queues we need to add 510 entries to the high freq queue that just delay the used window of the signals.



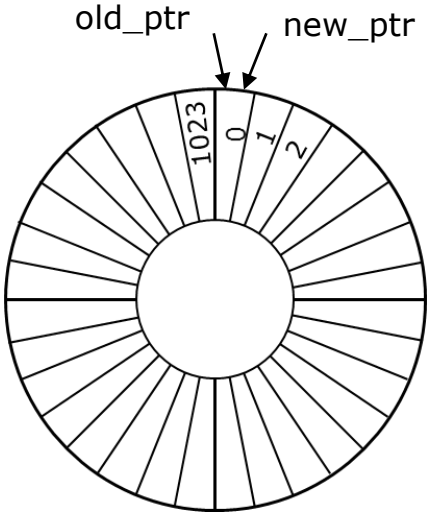
Queue Implementation (matching group delays)

- The group delay of our lower frequencies is 510/22050 = 23.1ms. The group delay of our upper frequencies is 510/44100 = 11.56ms. So we need to increase our B2,B3,HP queue length to add 510 entries of pure delay.

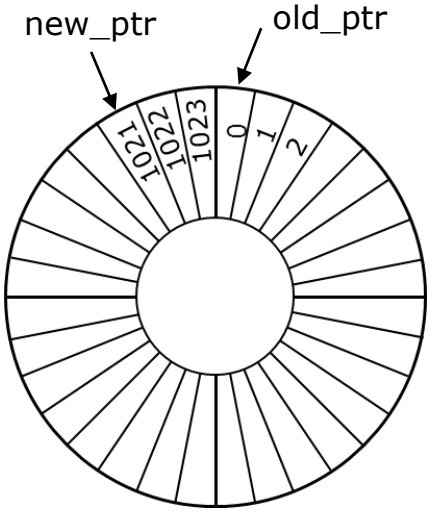
Bands	f_s	Required Size	Actual Size	DP RAM model
B2,B3,HP	44100	1021+510 = 1531	1536	dualPort1536x16.v
LP,B1	22050	1021	1024	dualPort1024x16.v

- Dual Port memories will be used to implement the queues. The queues should be implemented as circular queues.
  - Write port is used for new samples coming in I2S interface (RN-52).
  - Read port is used for reading out when performing the FIR filter algorithm.

LP,B1 Circular Queue Implementation

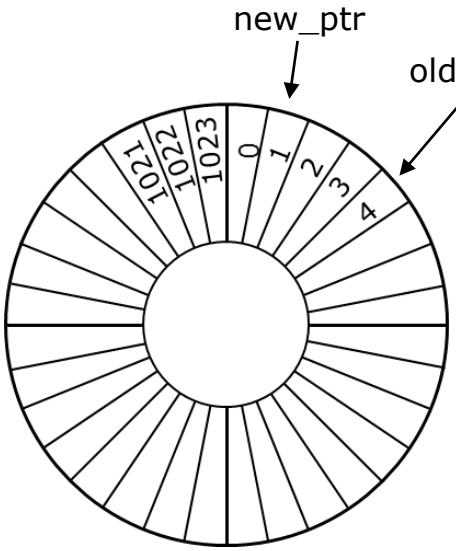


new_ptr points at next available location. ***old_ptr*** points at oldest data location. Circular queue starts like this.



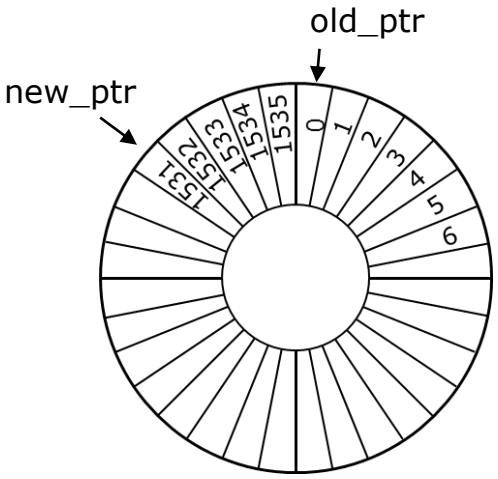
After having written 1021 locations the queue would look like this. Since we are only interested in 1021 locations now ***old_ptr*** can start incrementing on every new write.

After having written 1025 locations the queue for the lower frequencies would look like this. ***old_ptr*** would have advanced to 4 and the first 2 samples would have been overwritten. ***old_ptr*** and ***new_ptr*** keep moving around the circle, with ***old_ptr*** lagging ***new_ptr***.



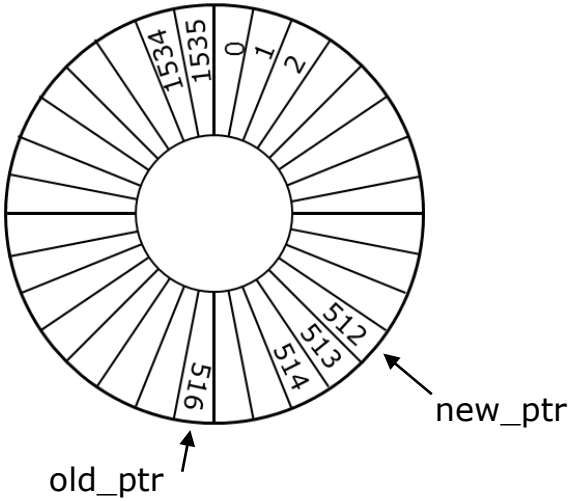
To read out the samples during a FIR filter calculation a 3rd pointer (***read_ptr***) would be initialized to ***old_ptr.*** Locations would then be read, with ***read_ptr*** incrementing with each clk. When ***read_ptr*** advances to ***old_ptr***+1021 then we have read out the samples of interest.

B2,B3,HP Circular Queue Implementation



Case where queue just became full for the first time (we have 1531 entries). For readout during FIR calculation the ***read_ptr*** would start from 0 and reading would be complete when it reaches 1021.

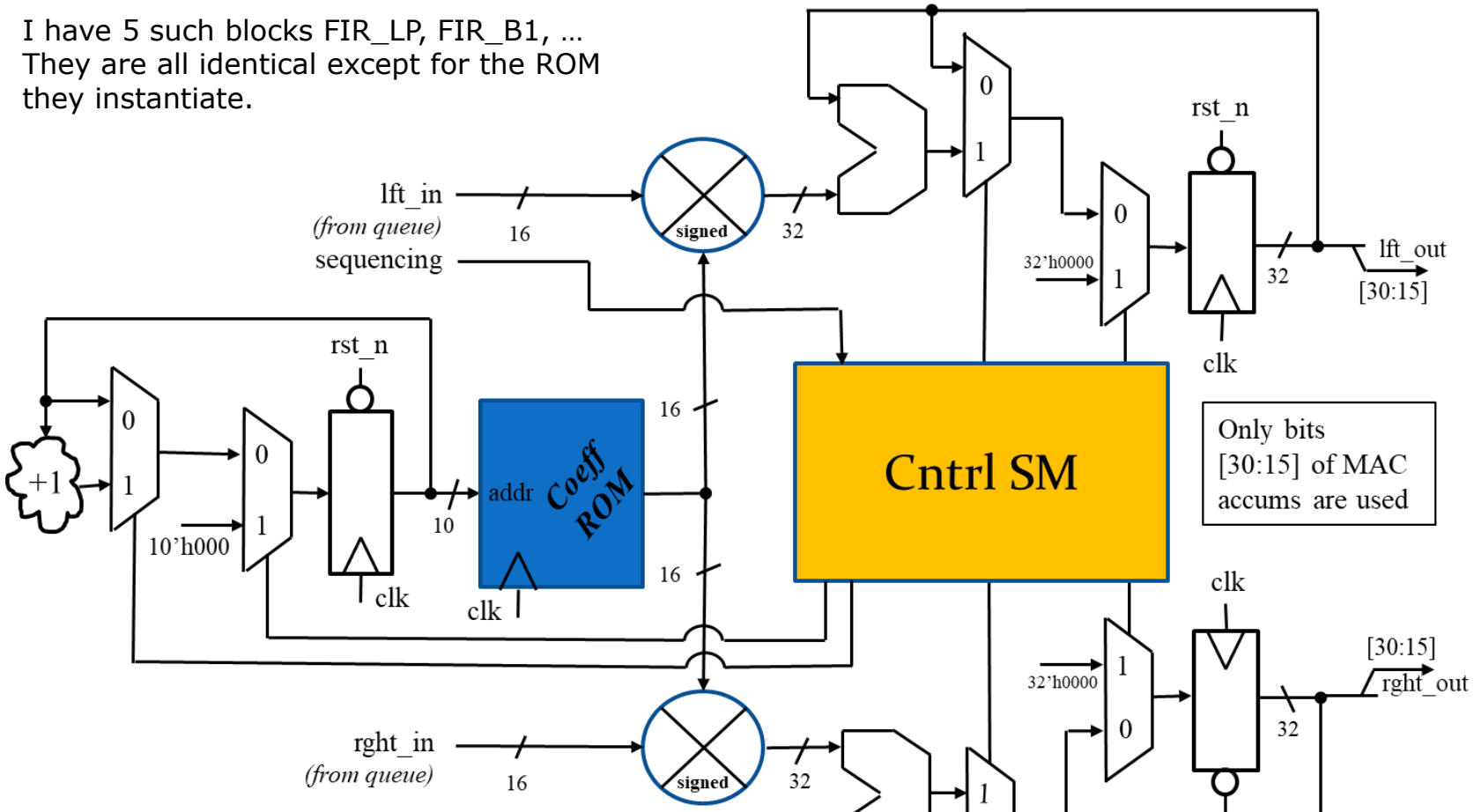
Case where 2048th sample has been written. For readout during FIR calculation the ***read_ptr*** would start from 516 and reading would be complete when it reaches "516+1021". However due to pointer roll over that would be 2



Things are a little trickier for the higher frequencies queue. The queue size is not a power of 2 so we have to watch how pointers roll over from 1535 to 0. When reading out the samples of interest. The ***read_ptr*** would start from ***old_ptr***. When it reached ***old_ptr***+1021 we would be done. Again, one has to watch the calculation for ***old_ptr***+1021 because that could be crossing a 1535 to 0 boundary.

FIRx Block (you can implement FIR functions as you wish. Mine looks like this…)

I have 5 such blocks FIR_LP, FIR_B1, …
They are all identical except for the ROM they instantiate.

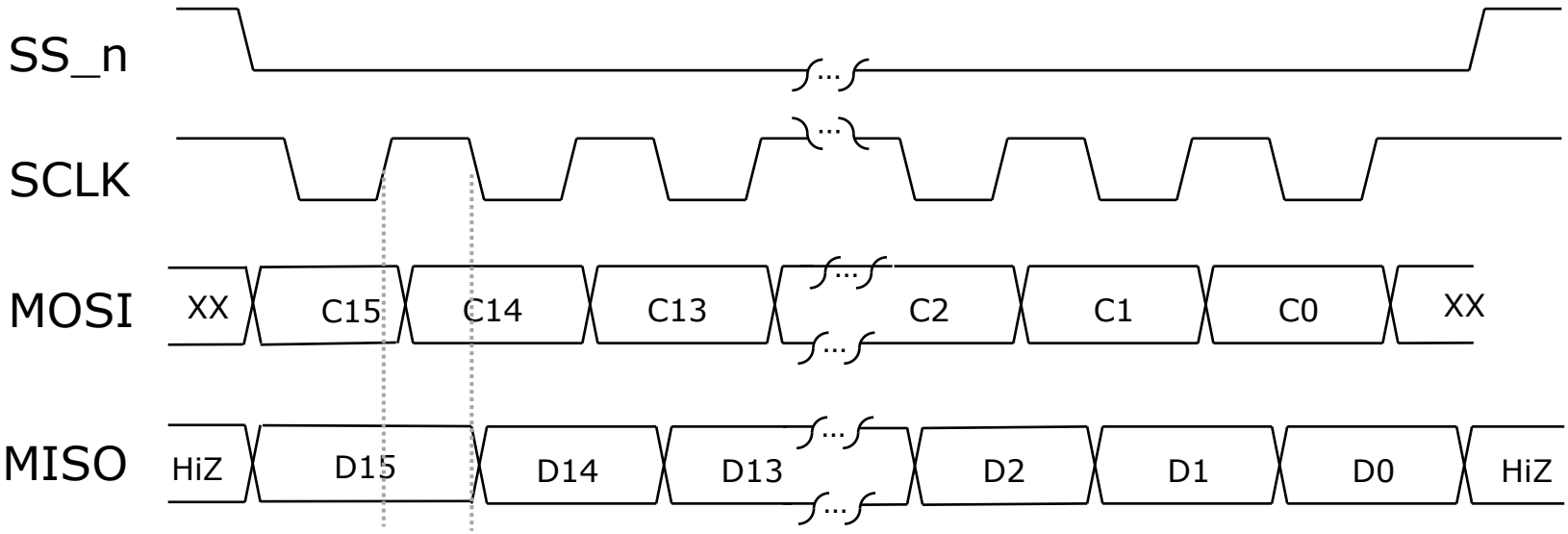


Recall ROM takes one clock cycle to access. ROM addr should be at zero, and incremented as soon as **sequencing** goes high. ROM address should be 10'h001 when MAC of oldest sample from queue and coeff[0] is occurring.

What is SPI?

- Simple Monarch/Serf serial interface (Motorola long long ago)
  - **S**erial **P**eripheral **I**nterconnect (very popular physical interface)
  - 4-wires for full duplex
    - ✓ MOSI (Monarch Out Serf In) (We drive this to 6-axis inertial)
    - ✓ MISO (Monarch In Serf Out) (Inertial sensor drives this back to us)
    - ✓ SCLK (Serial Clock)
    - ✓ SS_n (Active low Serf Select) (For us we only have one SS per SPI channel)
  - There are many different variants
    - ✓ MOSI shifted on SCLK rise vs fall, MISO sampled on SCLK rise vs fall
    - ✓ 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 high, 16-bit packets only
    - ✓ MOSI shifted slightly after SCLK rise (2 system clocks after)
    - ✓ MISO sampled at that same time.

SPI Packets

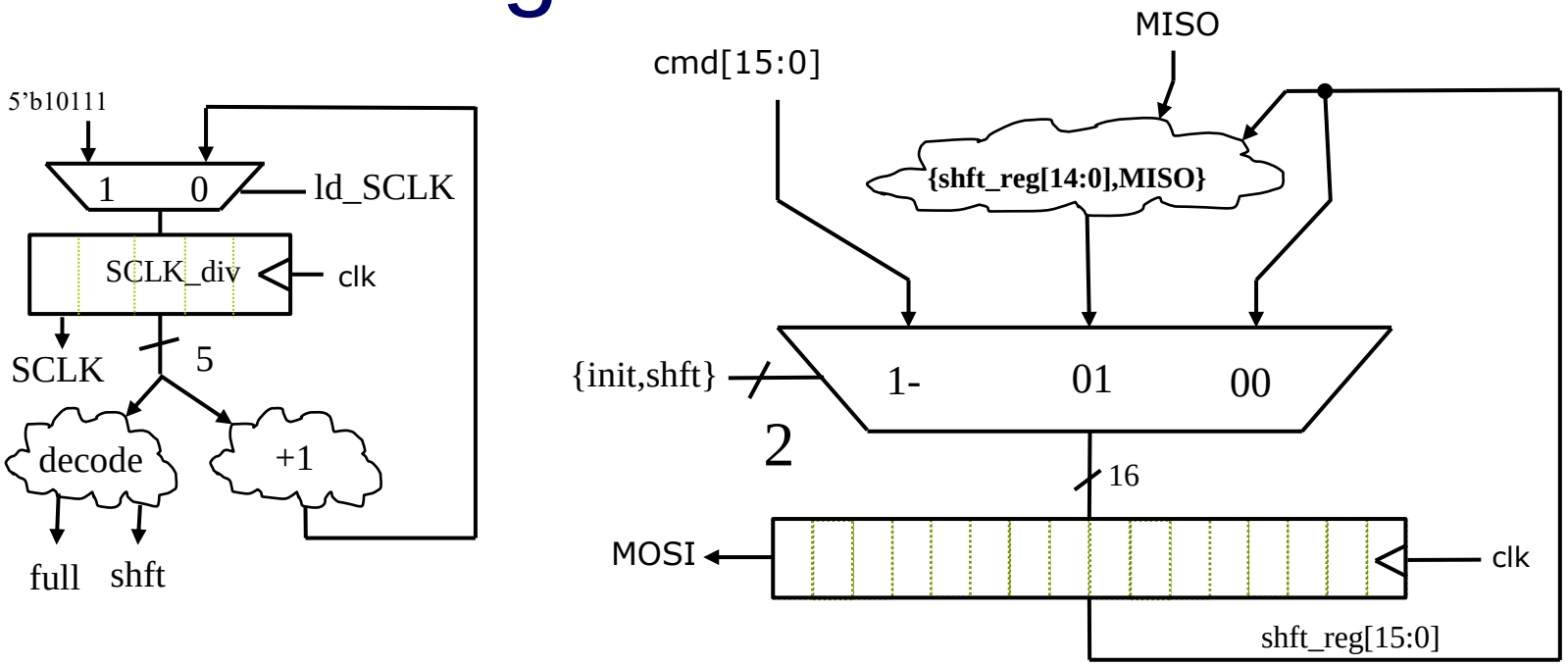


A SPI packet inherently involves a send and receive (full duplex). The full duplex packet is always initiated by the monarch. The monarch controls SCLK, SS_n, and MOSI. The serf drives MISO if it is selected. If the serf is not selected it should leave MISO high impedance. The A2D converter is the SPI peripheral we have in our system.

The SPI monarch will have a 16-bit shift register. The MSB of this shift register is MOSI. MISO will feed into the LSB of this shift register. The shift register should shift **two system clocks after** the rise of SCLK, this eliminates any timing difficulties. The serfs sample MOSI on the positive edge of SCLK, and change MISO on the negative edge of SCLK. Of course all your flops are based purely on clk (system clock (clk)), not SCLK! SCLK is a signal output from your SPI monarch.

SCLK will be 1/32 of our system clock (50MHz/32 = 1.5625MHz

SPI Shift Register



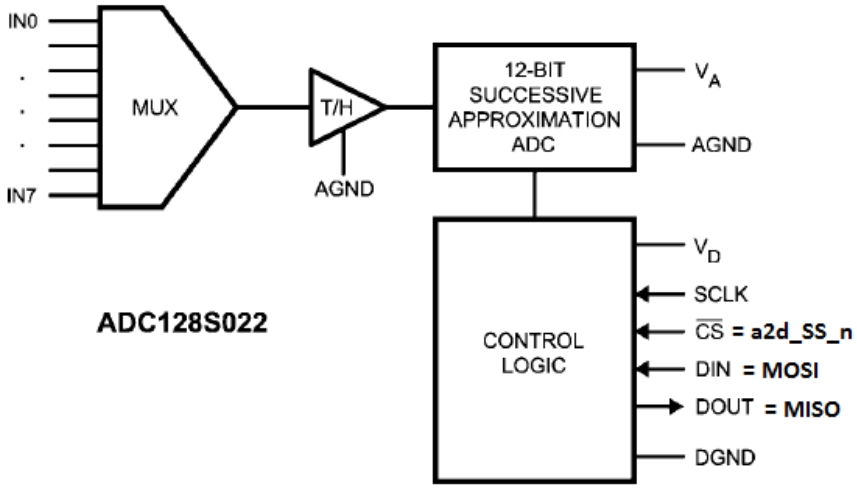
The main datapath of the SPI monarch consists of a 16-bit shift register. The MSB of this shift register provides **MOSI**. The shift register can be parallel loaded with the data to send, or it can left shift one position taking **MISO** as the new LSB, or it can simply maintain *(recirculate)*.

Since the SPI monarch is also generating **SCLK** it can choose to shift this register in any relationship to **SCLK** that it desires. To alleviate timing difficulties it is best that the shift register is shifted two system clocks after **SCLK** rise. Note the value SCLK_div is loaded with (5'b10111). Look back at the waveforms. There is a little time from when **SS_n** falls till the first fall of **SCLK**. Do you get the idea of loading with 5'b10111?

A2D Converter (National Semi ADC128S022)

The ADC128S is a 12-bit eight channel A2D converter. Only one channel can be converted at a time. The A2D is read by via the SPI bus, and is used to convert the values of the six slide potentiometers.

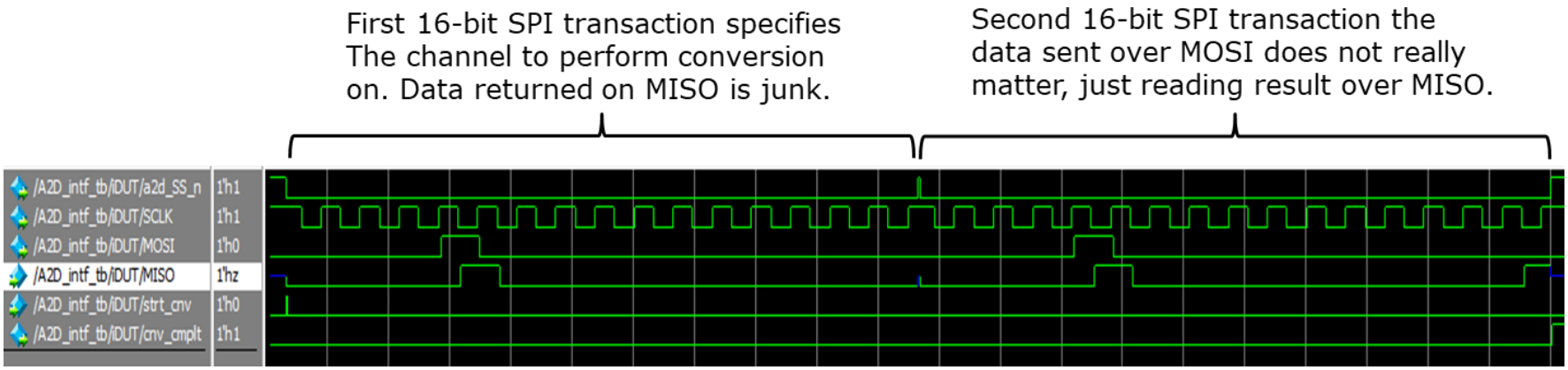
ADC Channel:	IR Sensor:
001 = addr	Gain_LP (gain of Bass)
000 = addr	Gain B1 (gain of 64Hz to 256Hz)
100 = addr	Gain_B2 (gain of 256Hz to 1kHz)
010= addr	Gain_B3 (gain of 1kHz to 4kHz)
011= addr	Gain_HP (gain of treble)
111= addr	Volume (scales both left/right sums)



To read the A2D converter one sends the 16-bit packet {2'b00,addr,11'b000} twice via the SPI. There needs to be a 1 system clock cycle pause between the first SPI transaction completing, and the second one starting

During the first SPI transaction the value returned over MISO will be ignored. The first 16-bits are really setting up the channel we wish the A2D to convert. During the 2nd SPI transaction the data returned on MISO will be the result of the conversion. Only the lower 12-bits are meaningful since it is a 12-bit A2D. The slide potentiometers should be read in a round robin fashion.

A2D Converter (Example SPI Read)



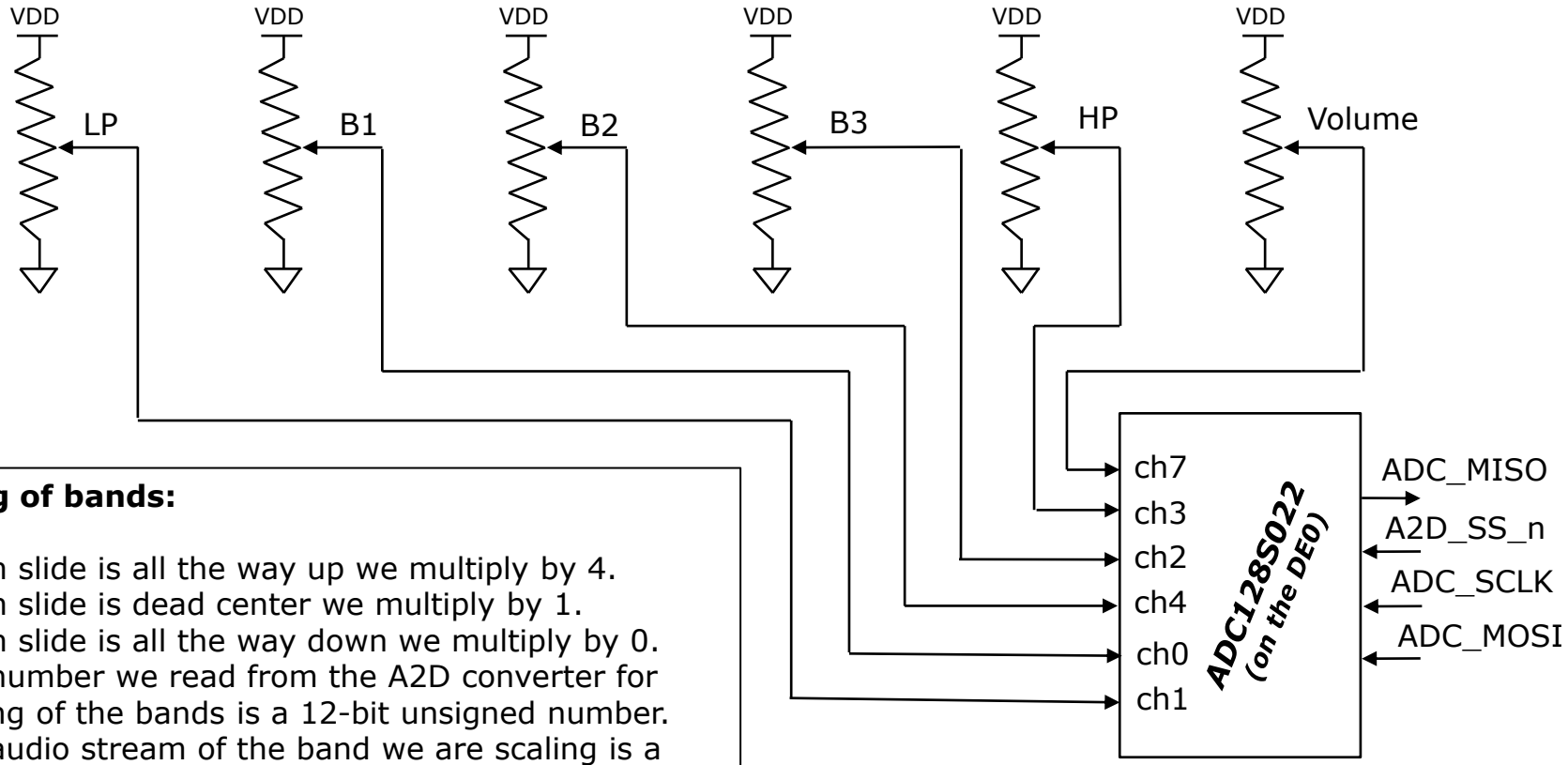
Our use of the A2D converter will involve two 16-bit SPI transactions nearly back to back (separated by 1 system clock cycle).

The first transaction here is sending a 0x0800 to the A2D over MISO. The command to request a conversion is {2'b00,channel[2:0],11'h000}. The upper 2-bits are always zero, the next 3-bits specify 1 of 8 A2D channels to convert, and the lower 11-bits of the command are zero. Therefore, the 0x0800 in this example represents a request for channel 1 conversion.

For the next 16-bit transaction the data sent over MOSI to the A2D does not matter that much. We are really just trying to get the data back from the A2D over the MISO line.

NOTE: you need at least a 1 clock cycle pause between the completion of the first SPI transaction and the initiation of the second.

Scaling of Bands and Volume



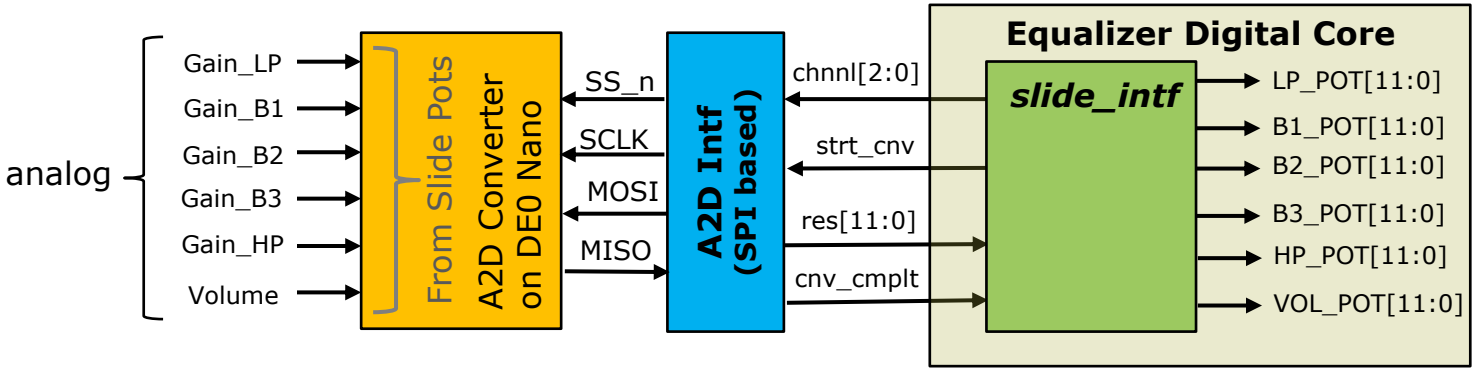
Scaling of bands:

- When slide is all the way up we multiply by 4.
- When slide is dead center we multiply by 1.
- When slide is all the way down we multiply by 0.
- The number we read from the A2D converter for scaling of the bands is a 12-bit unsigned number.
- The audio stream of the band we are scaling is a 16-bit **signed** number.
- Think about this…this is a little tricky.

Scaling of volume:

- The volume of the audio stream (coming from the RN-52 via I2S) can be scaled using the source device.
- The intent of the volume slider is to have a quick easy way to reduce volume. When the slider is all the way up we should be multiplying by unity. When it is all the way down we multiply by zero.

Scaling of Bands and Volume



Given you have a block (***slide_intf***) that performs A2D conversions on the slide potentiometers continuously in a round robin fashion. You now have a 12-bit unsigned number that represents the amount to scale each band (and volume). Now how to make use of those digital scaling numbers?

Volume is a simple linear scaling. VOL_POT[11:0] can be made into a 13-bit signed number {1'b0,VOL_POT} and multiplied by the resulting left/right audio stream. This 29-bit product can then be shifted right 12-bits, and the MSB ignored. So you essentially use bits[27:12] of the 29-bit signed product.

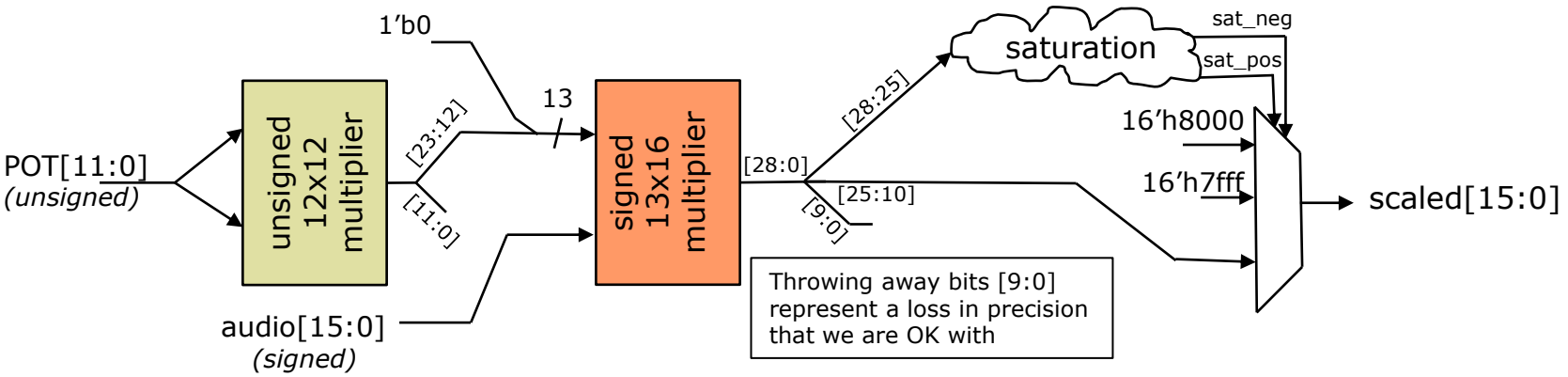
For Band scaling we need to use the square of the potentiometer reading as our scaling factor. This gives us the 4X gain when the pot is at full, 1X gain when it is centered, and zero when it is all the way low. (Human perception of sound is to the square root of amplitude, so we have to use square of the pot value to make the effect seem linear)

Starting with a 12-bit unsigned number we square it to get a 24-bit unsigned product. One can next just use bits [23:12] of this product as the 12-bit number to scale the respective FIR output by. Similar to volume scaling we will want to convert our 12-bit number that represents the square of the potentiometer value into a 13-bit signed number. This signed 13-bit factor will be multiplied by the 16-bit signed result of the respective FIR filter output to produce a 29-bit product. We next shift this product right 10-bits and use bits [25:10] as the result sum with the other bands to form our digital audio output. However, what about the data in bits [28:26] that we are discarding?? Does the result of using bits [25:10] match it in sign?

Band Scaling

For an audio equalizer the gain control determines the scaling of that band of frequencies. When it is set to its maximum the band is multiplied by 4. When it is set to the middle the band is multiplied by 1. When it is set to zero the band is multiplied by zero.

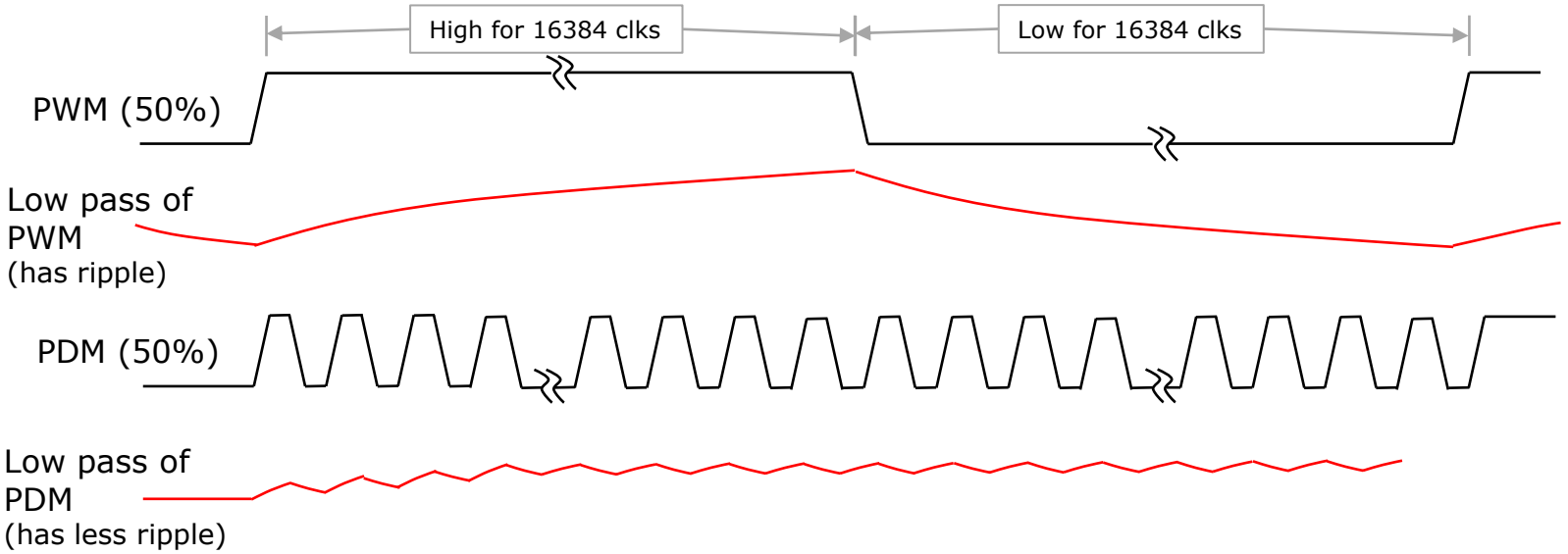
This means we really want to use the square of the potentiometer as our gain term.



After squaring the potentiometer value we can choose to drop the lower 12-bits and use only the upper 12-bits of the squared result for scaling the audio signal. The audio signal is a signed number, so we need to convert our squared value of the pot to a 13-bit signed number by appending a 1'b0 as the MSB. Now we enter a 13x16 multiplier which will produce a 29-bit result. Of this 29-bit result we will use bits [25:10] as our final scaled audio signal for that band. Dropping bits [9:0] represents a truncation. Dropping bits [28:26] however, is not OK. We have to inspect bits [28:25] to see if our result saturated positive or saturated negative. If so we will choose the appropriate saturated number, if not we just pass [25:10] as the result.

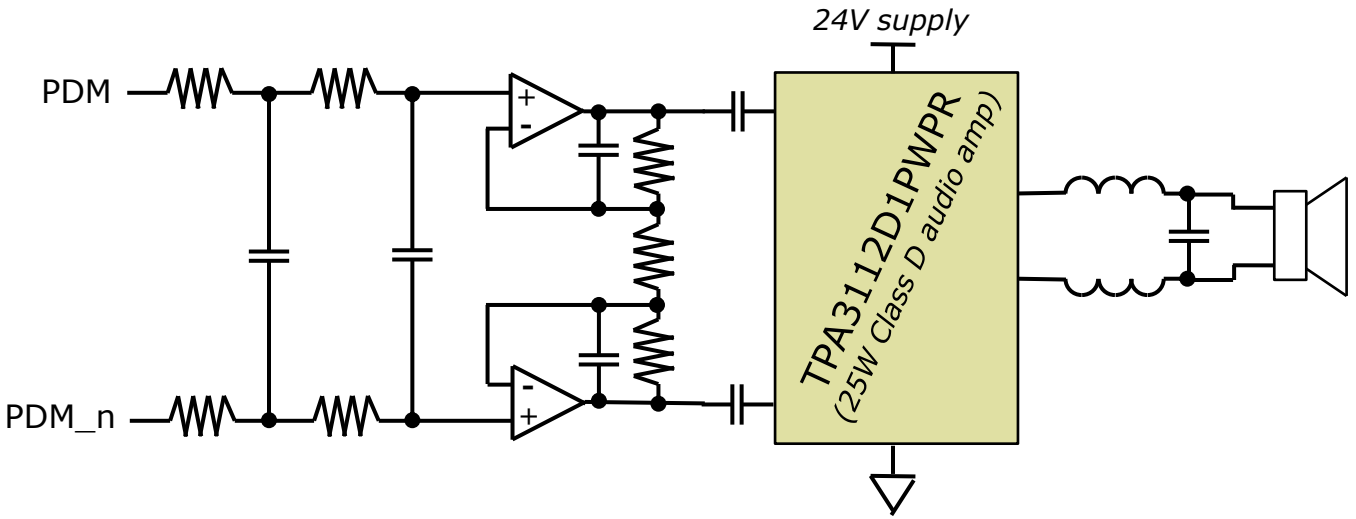
PDM = Pulse Density Modulation (driving the speakers)

- To drive the speakers we eventually need an analog signal. So we need a form of Digital to Analog Conversion (DAC).
- There are many forms of DAC. One of the simplest is Pulse Width Modulation (PWM) followed by low pass filtering.
- A variation of PWM is PDM. PDM and PWM are conceptually similar, but PDM is easier to low pass filter than PWM.
- Consider a 15-bit PWM signal vs a 15-bit PDM signal. Both driving 50% duty cycle.



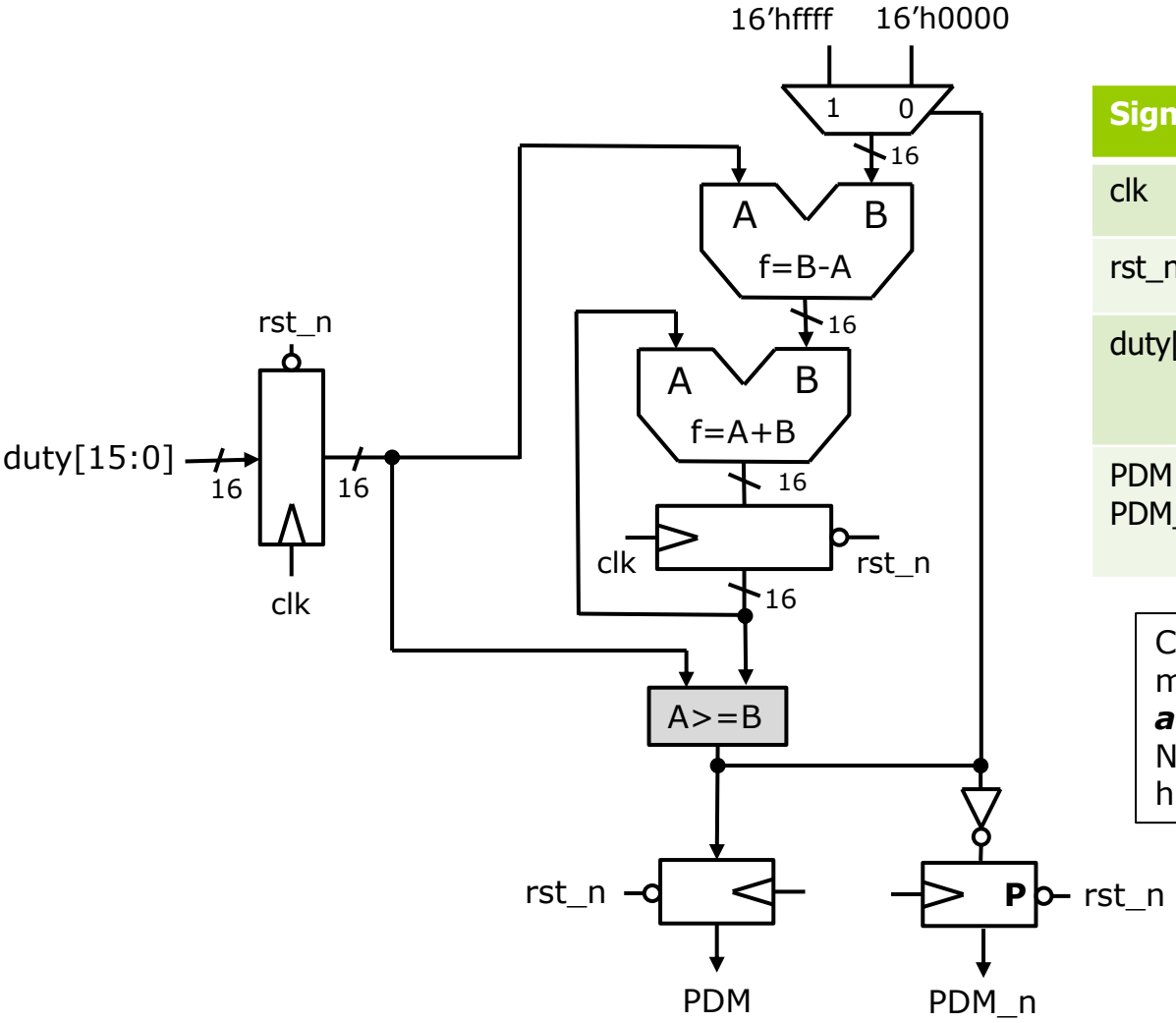
- As is shown above for a low pass filter with a given time constant, the PDM signal filters better and gives a lower ripple. This is an exaggerated view. The time constant of the LPF on the board will result in tiny ripple, and thus very little distortion in music quality.

Low Pass PDM into Class D Amplifier



- Obviously even low passed versions of the PDM signals cannot properly drive a speaker. These are coming from the I/O of our FPGA and only have 3.3V amplitude and relatively high output impedance.
- We will produce a PDM signal and its complement (PDM_n). These will go through a passive 2nd order differential lowpass followed a 3rd stage of active lowpass & gain. This signal then passes through some DC blocking caps into a class D audio amplifier chip capable of 25 watts of power output.

PDM Architecture:



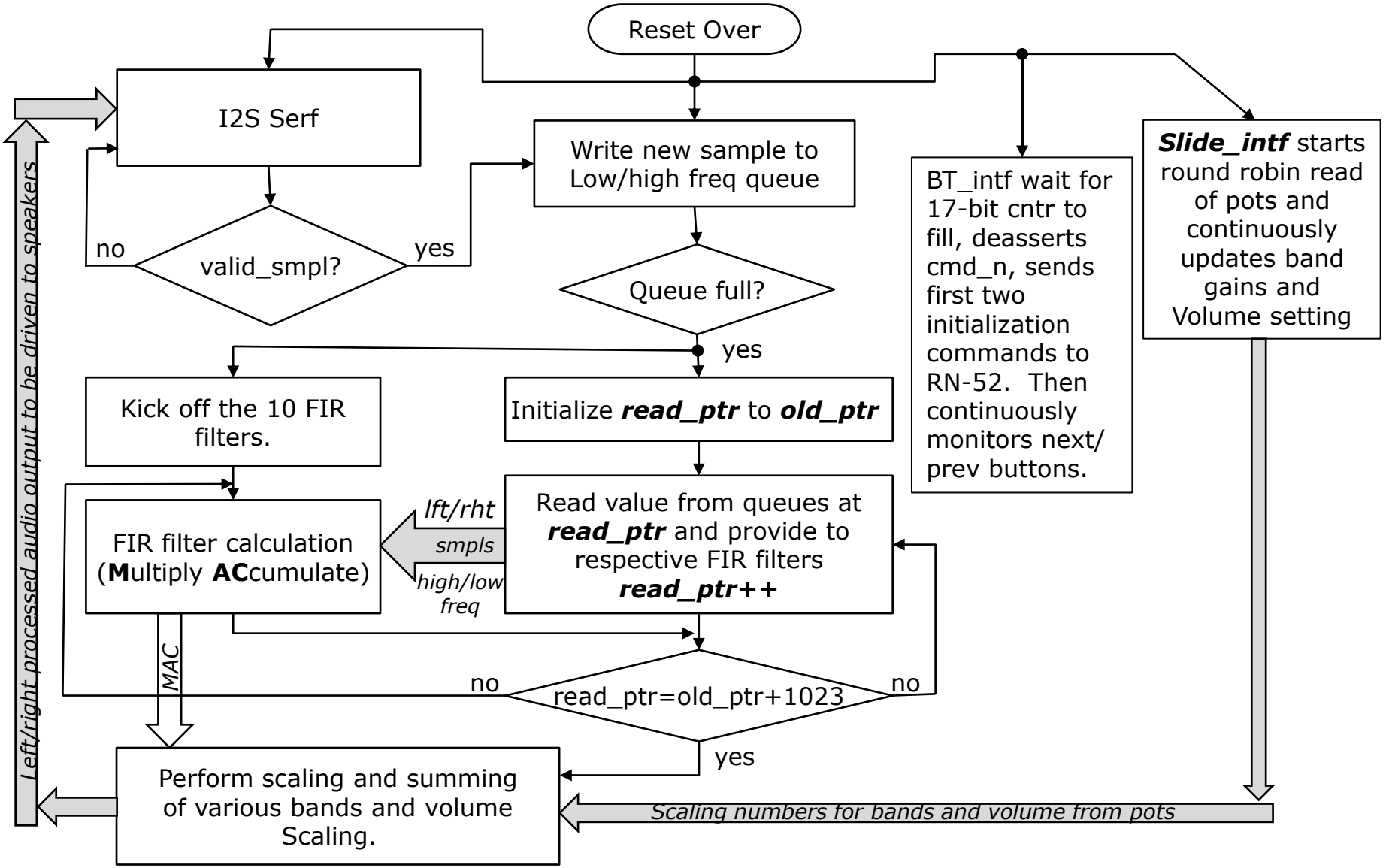
Signal:	Dir:	Description:
clk	in	50MHz system clk
rst_n	in	Asynch active low
duty[15:0]	in	Specifies duty cycle (unsigned 16-bit)
PDM PDM_n	out	PDM signals (glitch free) **NOTE:** PDM_n is preset.

Code what you see using a mixture of dataflow and ***always*** blocks. Code it flat. No need to introduce hierarchy here.

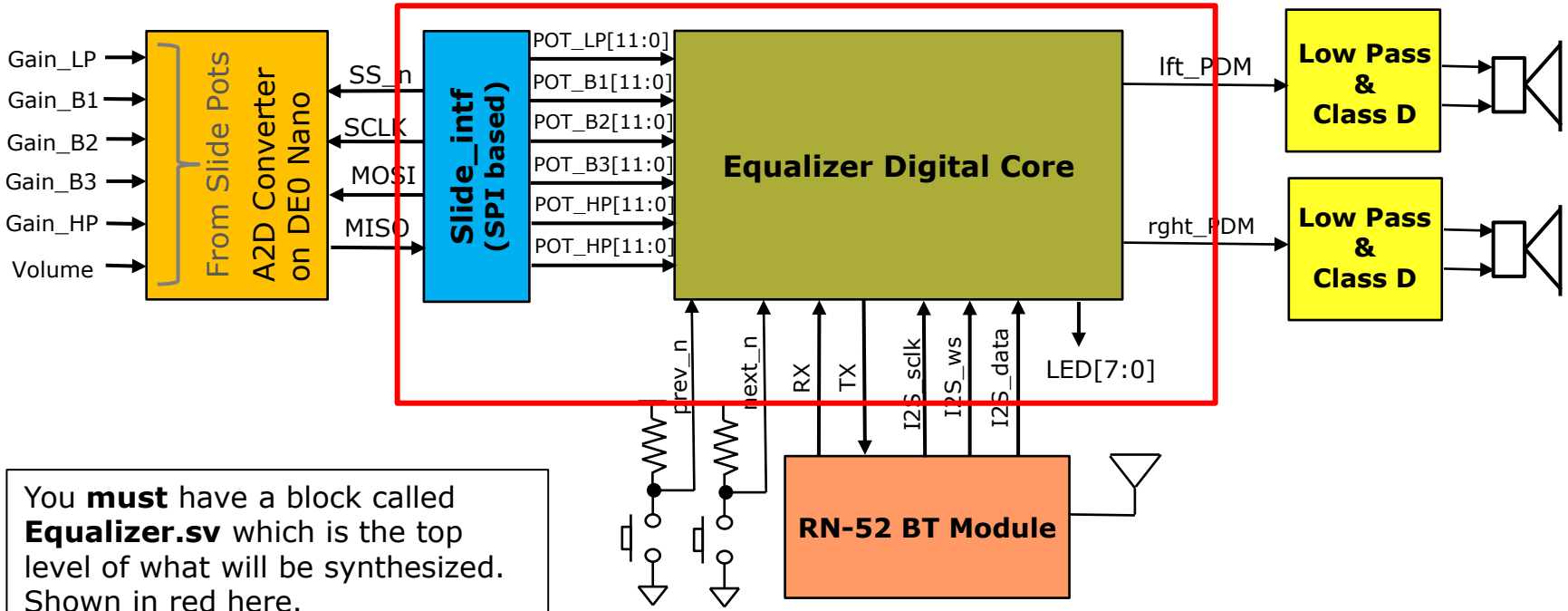
LED[7:0] Outputs

- This is an extra credit opportunity
- There are 8 LEDs on the DE0-Nano board
- Can you do something cool with them?
  - A thermometer type output that represents of running average of music intensity?
    - To make this effective it might have to be sqrt of intensity.
  - Some other cool LED affect that is somehow **tied to the** music.
- If you are going to do this optional part then make it a separate module so you can use a blank version when synthesizing on Synopsys so the area does not count against you.
- Impress me with your LED effects and earn another 1.25% extra credit.

Digital Functionality



Required Hierarchy & Interface

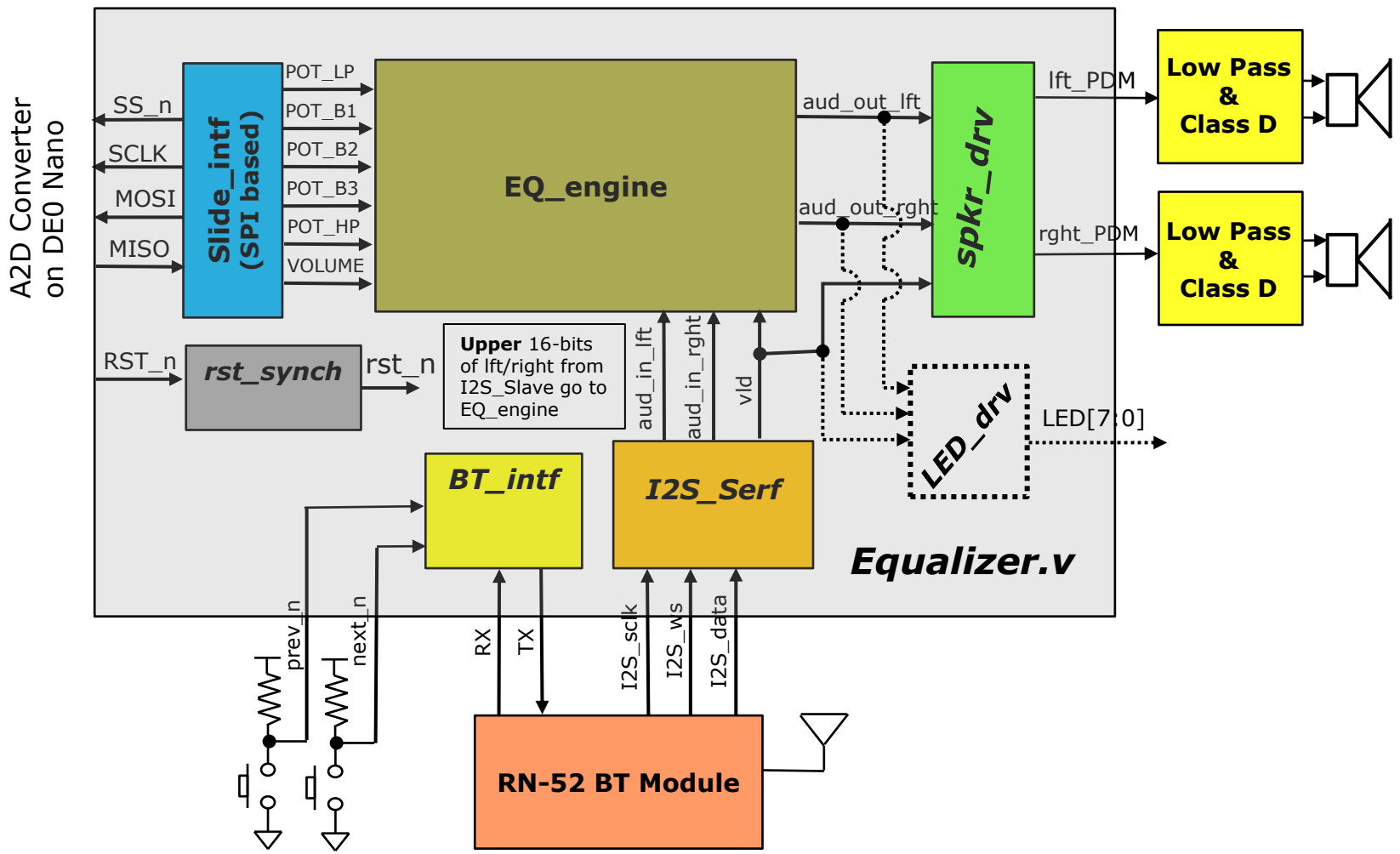


You **must** have a block called **Equalizer.sv** which is the top level of what will be synthesized. Shown in red here.

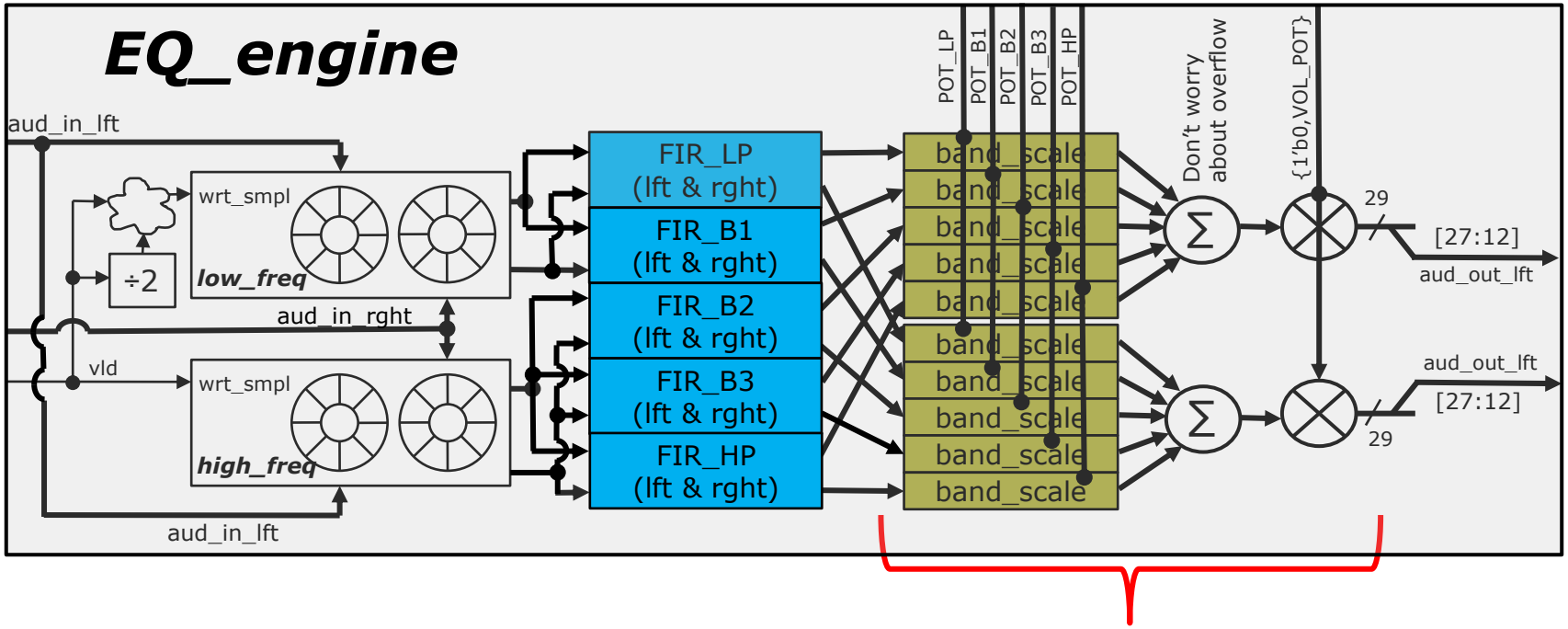
The interface of **Equalizer.sv** must match exactly to the specified **Equalizer.sv** interface. Please download the provided files from the project .zip and use the **Equalizer.sv** provided as your skeleton.

The hierarchy/partitioning of your design below the **Equalizer.sv** level is up to your team. The next slide shows my hierarchy. The hierarchy of your testbench is up to your team.
Your design will also be placed into an EricHarish test bench, which is why it is critical it match at the **Equalizer.sv** level.

My Hierarchy (hierarchy below **Equalizer** up to you…you can match what I did if you like)



My Hierarchy (What is inside EQ_engine)



In the final summation of the bands it is possible there would be overflow. Don't sweat it, just keep everything at 16-bits. If there is overflow it just mean you young whipper snappers are playing your rock-n-roll music too loud and should turn it down. However, I ask that you always keep overflow in mind in case you are working on a mission critical application in the future.

There is a heck of a lot of math going on here kids. Is Synopsys going to be able to fit all this math in a single cycle at a 3ns clock period? You will probably need to insert some flops in this path to pipeline it and ease the timing constraints. Start without flops first and get it functionally working. Add the pipelining flops later, the latency of the flops affects nothing.

After scaling by volume you should have a signed 29-bit product. Just use bits 27:12 as the result for each channel. Don't worry about any saturation. Again, you are playing your music too loud if that happens.

Equalizer Interface

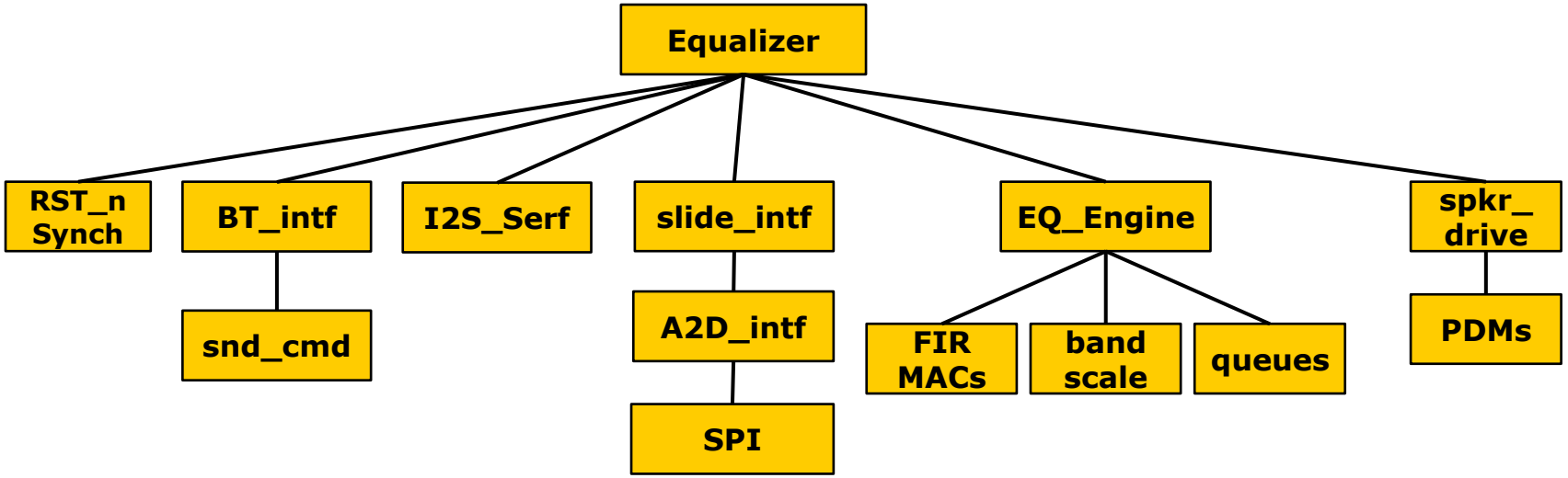
Signal Name:	Dir:	Description:
clk	in	Clock input (50MHz)
RST_n	in	Active low input from push button. Should be synchronized inside equalizer
LED	out	Perhaps nothing (tied low). Perhaps something cool…extra credit opportunity
A2D_SS_n	out	Active low serf select to A2D SPI interface
A2D_SCLK	out	SPI bus clock (to A2D)
A2D_MOSI	out	Serial output data to SPI bus of A2D converter (**M**onarch **O**ut **S**erf **I**n)
A2D_MISO	in	Serial input data from SPI bus of A2D converter (**M**onarch **I**n **S**erf **O**ut)
I2S_sclk	in	Roughly a 2.11MHz signal. Use rise edge to sample I2S_data
I2S_ws	in	I2S Word Select. 0 => Left Chnnl, 1 => Right Chnnl
I2S_data	in	Serial Audio data. 24-bits/channel of which we use the upper 16-bits.
lft_PDM, lft_PDM_n rght_PDM,rght_PDM_n	out	PDM signals that drive H-Bridge that is low passed to drive speakers.
cmd_n	out	To RN-52 Bluetooth Audio module (put in cmd mode after 2.5ms delay)
RX	in	From RN-52 Bluetooth Audio module (115,200 baud)
TX	out	To RN-52 Bluetooth Audio module
prev_n/next_n	in	Couple of push button inputs that allow repeating/skipping of songs
sht_dwn	out	Shuts down class D amp when asserted. Should be high for first 5ms after reset
Flt_n	in	Active low fault signal from class D. Should force **sht_dwn** for 5ms if it occurs

Provided Modules & Files: (available on website under: Project)

File Name:	Description:
Equalizer.sv	**Requried** interface skeleton verilog file. **Copy this** and flush it out with your design
ROM_*.v	ROM models for holding the coefficients for the FIR filters. LP,B1,B2,B3,HP
Coeff_*_hex.txt	FIR coefficients read by the above ROM models.
dualPort*.v	Dual port memories used for queues holding audio samples. 1024 entries for LP & B1 queues and 1536 entries for B2,B3, & HP queues
A2D_with_Pots.sv	Models the A2D converter and the pots it converts. Contains provided module SPI_ADC128S
RN52.sv	Verilog model of RN-52 BT audio module
cmdROM.v & cmd_hex.txt	Contains command strings to be sent to the RN-52 for initialization or for next/prev functionality
resp_ROM.v & resp_hex.txt	Contains responses sent by RN-52 model.
UART.sv	UART module used inside RN_52.sv. You can also use it for your DUT to communicate with RN-52.
I2S_Mnrch.sv	Sub-module of RN-52 model. Master your I2S_Serf is reading from.
tone_ROM_*.v	Sub-module of RN-52 model. Contains “music” of left/right channels
tone_hex_*.txt	Data files with 5 superimposed frequencies in the middle of each band of interest. Read by **tone_ROM_*.v**. Handy for testing at fullchip level
Equalizer_tb.sv	Up to you. This is a skeleton test bench you can start with.

Synthesis:

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



You are **NOT** allowed to use **compile_ultra**

- Your synthesis script should write out a gate level netlist of follower (**Equalizer.vg**).
- You should be able to demonstrate at least one of your tests running on this post synthesis netlist successfully.
- Timing (333MHz) has some challenges. Your main objective is to minimize area.

Synthesis Constraints:

Not allowed to use **compile_ultra**

Contraint:	Value:
Clock frequency	333MHz (yes, I know the project spec speaks of 50MHz, but that is for the FPGA mapped version. The standard cell mapped version needs to hit 333MHz minimum.)
Input delay	0.75ns after clock rise for all inputs
Output delay	0.75ns prior to next clock rise for all outputs
Drive strength of inputs	Equivalent to a **NAND2X1_LVT** gate from our library
Output load	50fF on all outputs
Wireload model	saed32 16000
Max transition time	0.125ns
Clock uncertainty	0.125ns

NOTE: Area should be taken after all hierarchy in the design has been smashed.

My Total Cell Area is: 73226