ECE 551
Project Spec

Fall ’12

High Speed **C**alibratable **A**ngle **R**esolver

(High Speed CAR)

Grading Criteria: (Project is 30% of final grade)

- Project Grading Criteria:
  - Quantitative Element 25%

    (yes this could result in extra credit)

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

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

  - Project Demo (65%)
    - ✓ Code Review (12.5%)
    - ✓ Testbench Method/Completeness (15%)
    - ✓ Synthesis Script review (10%)
    - ✓ Post-synthesis Test run results (12.5%)
    - ✓ Results when placed in Eric Testbench (15%)

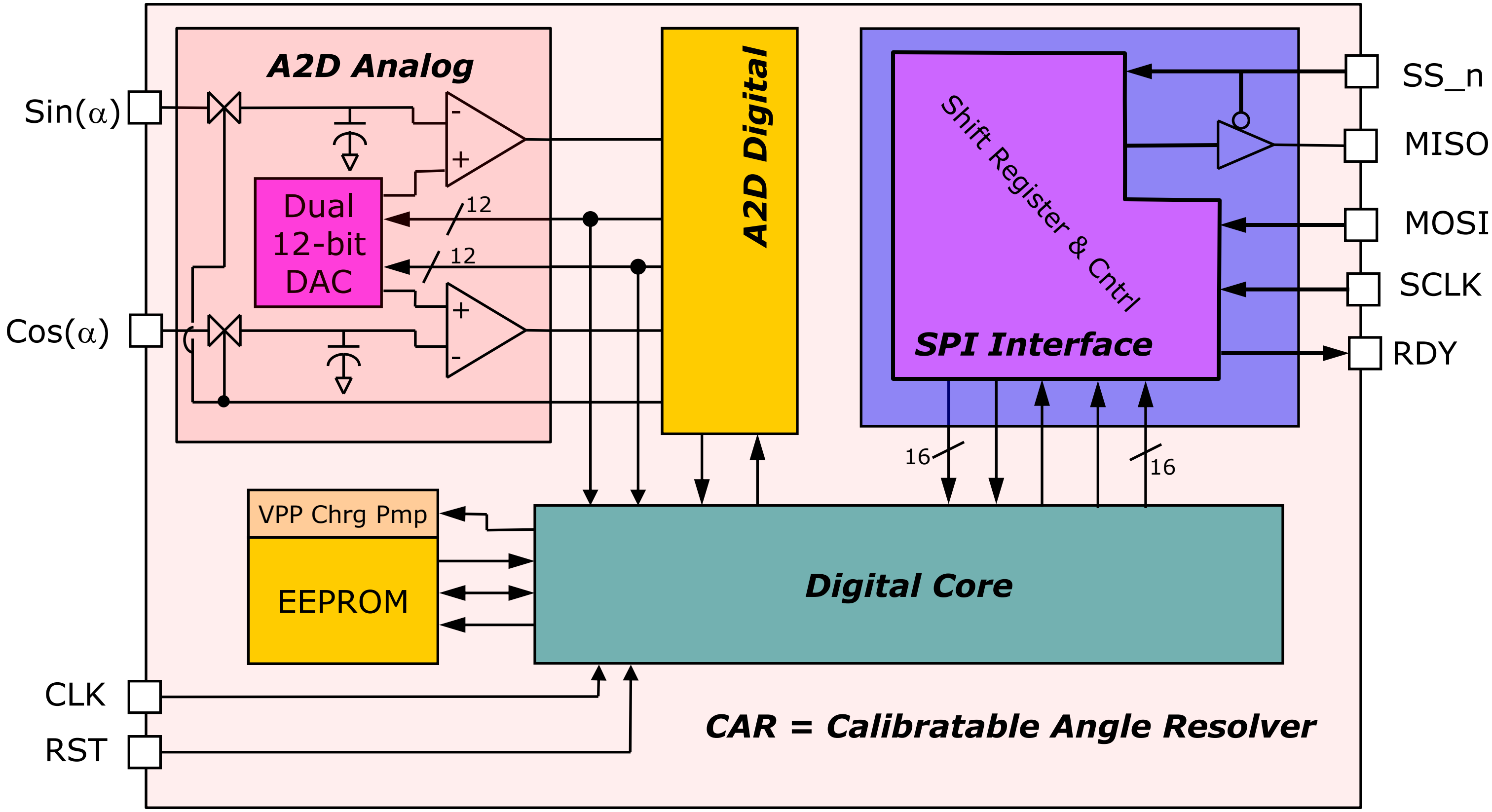
  - Project Report 10%
    - ✓ Expected to be around 5-8 pages (don't go crazy, I don't want to grade a really long report)
    - ✓ Show partitioning of digital core (block diagram, few words)
    - ✓ Report on datapath implementation (algorithm, support HW needed in datapath)
    - ✓ Report on each team members contributions
    - ✓ Section on lessons learned (focus on the hard leaned lessons…the heartache)

Project Due Date

- Project Demos will be held in B555:
  - Date **12/12/12** from 1:00PM till evening.
  - Other Date **12/14/12** from 1:00PM till evening.

- Project Demo Involves:
  - ✓Code Review
  - ✓ Testbench Method/Completeness
  - ✓Synthesis Script & Results review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in Eric testbench
  - ✓ Hand in of short report

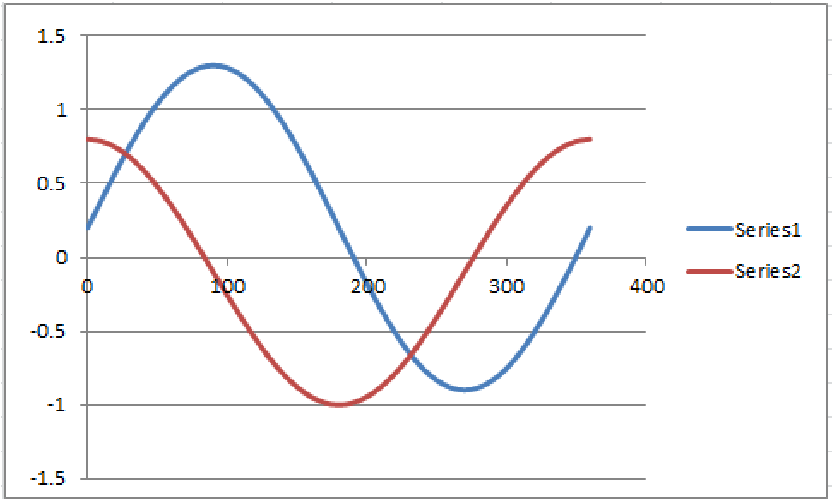
Chip Block Diagram



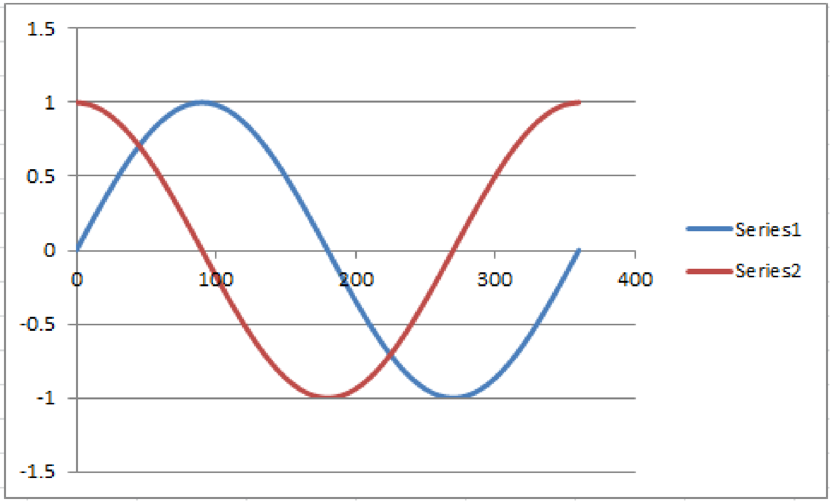
CAR = Calibratable Angle Resolver

Description: The **CAR** chip is a high speed, high precision angle resolver. It has a 2-chanel 12-bit A2D converter to sample a sine and cosine signal from an angle sensor. The sine channel produces and output that is A*sin(α)+B, and the cosine channel produces a signal that is C*cos(α)+D. The B and D terms represent undesired offsets that has to be cancelled through calibration. A and C represent unknown scaling terms that also need normalization through calibration. Calibration coefficients are stored in EEPROM. After each channel is digitally corrected for offset and gain the angle α is calculated by the digital core, and available on the SPI interface.

Calibration…what needs to be performed



Raw Sin/Cos signals into A2D



Need to correct for Gain & Offset errors

CorrectedSin = (SinSAR+OffsetSin)*ScaleSin
CorrectedCos = (CosSAR+OffsetCos)*ScaleCos

EEPROM Address:	Coefficient:
0	OffsetSin
1	ScaleSin
2	OffsetCos
3	ScaleCos

Perform correction in digital domain using scaling and offset coefficients stored in EEPROM

Calculating α

$$\alpha = ArcTan\left(\frac{CorrectedSin(\alpha)}{CorrectedCos(\alpha)}\right)$$

However this only gives an answer in the Range of −π/2 to π/2

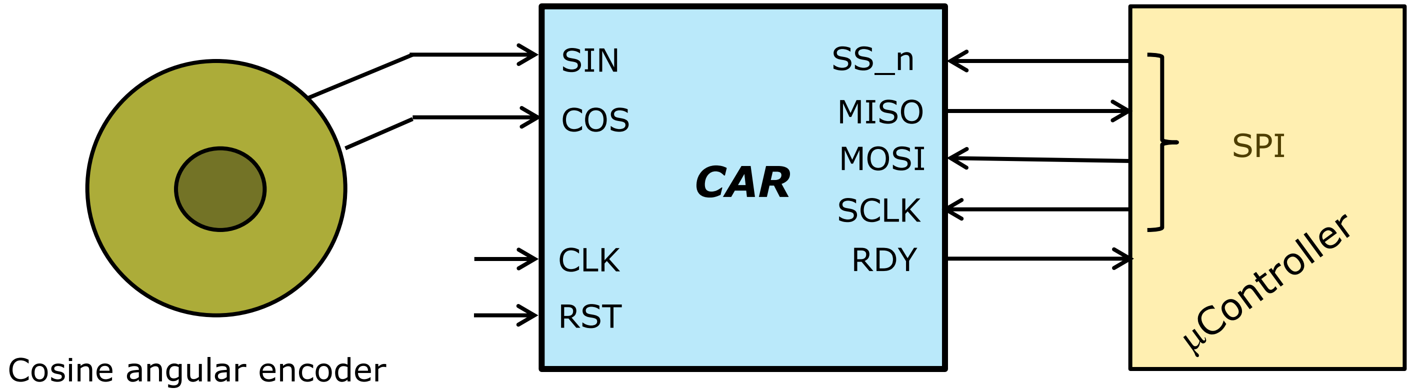
We have to correct for the other quadrants (when CorrectedCos(α) is negative)

How do you compute ArcTan?

Are you really going to implement a divide?

Remember HW1. Look at HW Solution.

CAR in a System



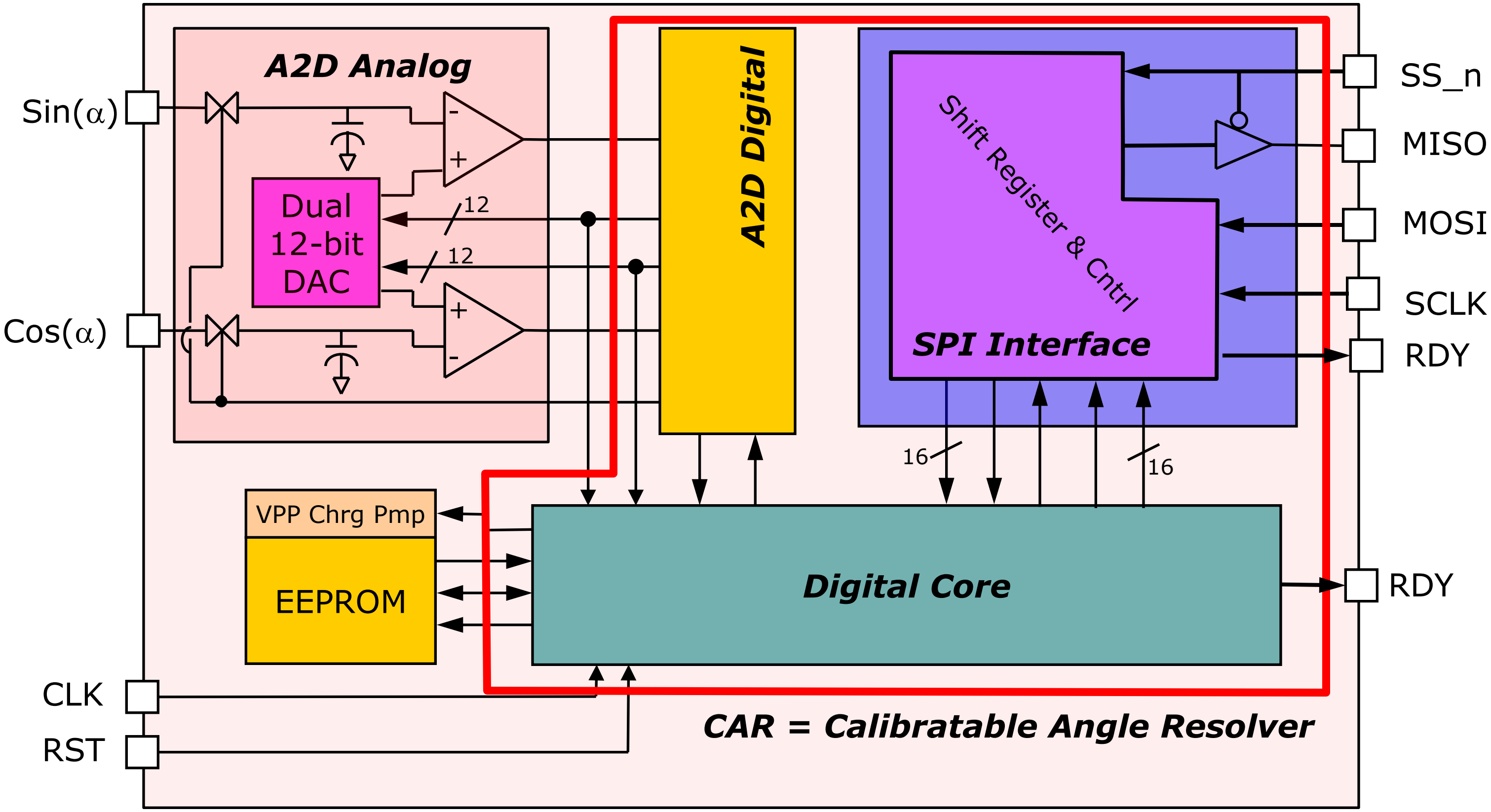
An angle resolver could be used in multiple different systems. It could be part of a VFD drive system for an AC motor. It could be part of a power steering angular sensor. It could be part of a antenna sweep mechanism for an AWACs plane. It could be part of some high speed industrial robotics.

In most any system the output of our chip would be processed by an embedded micro-controller which would in turn control the process of interest. For the project we will take the output to be a SPI bus. It could however be any number of serial protocols. (I2C, UART, SENT, ModBus, …)

Cosine based angular encoders are most commonly constructed from optical means or magnetic means. Light shining through slots on a disk to an optical sensor, or rotating magnetic field penetrating a hall effect sensor.

In the project area of the website take a look at the TLE5012 spec from Infineon. We are essentially making a slightly simplified version of this chip. *(This spec is self contained…look only if interested)*

What is synthesized DUT vs modeled?

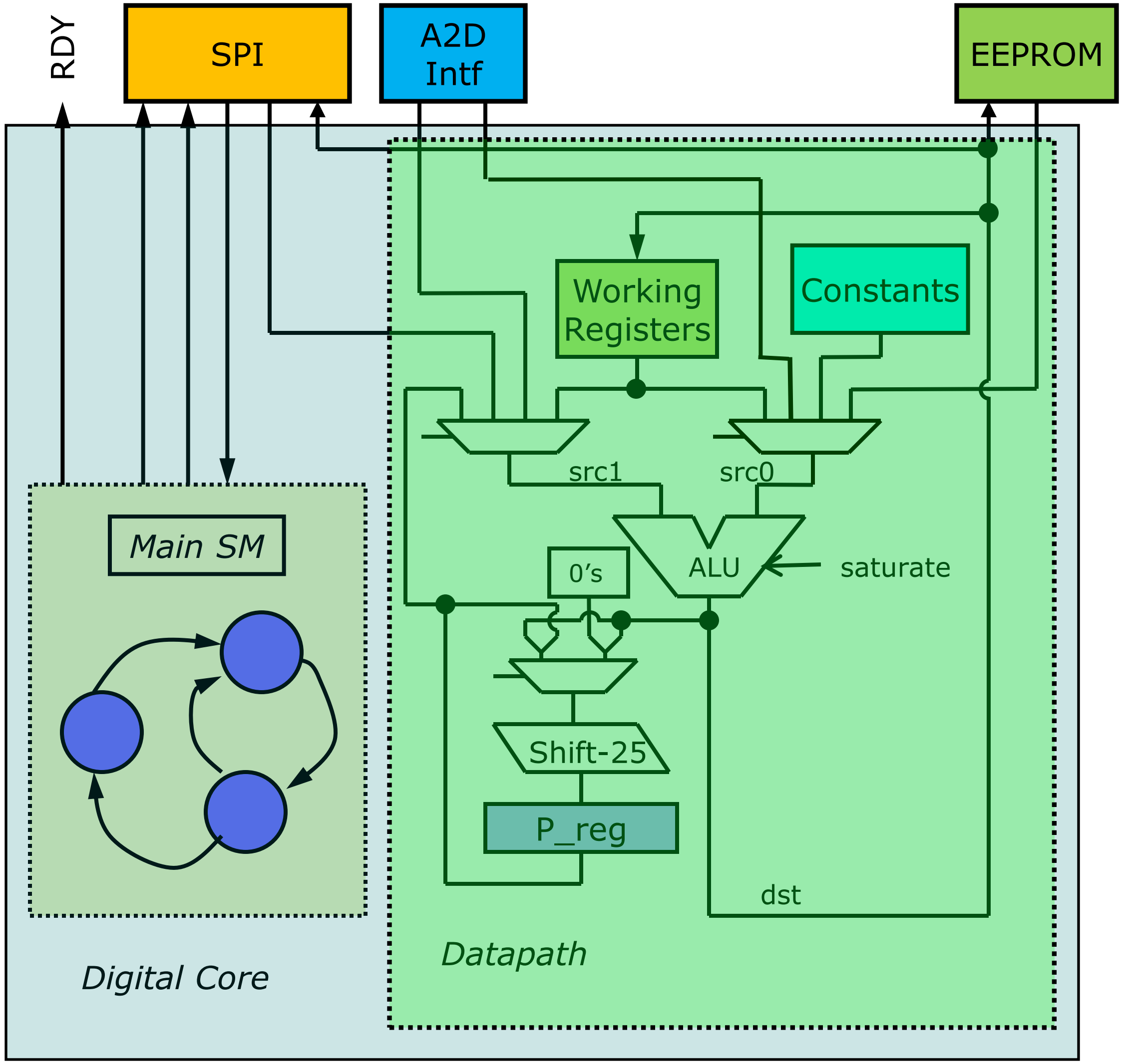


The blocks outlined in red above are pure digital blocks, and will be coded with the intent of being synthesized. All other blocks will be modeled in Verilog for fullchip simulation purposes.

For the class project Verilog models of (EEPROM, A2D Analog) will be provided.

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

Digital Core Block Diagram



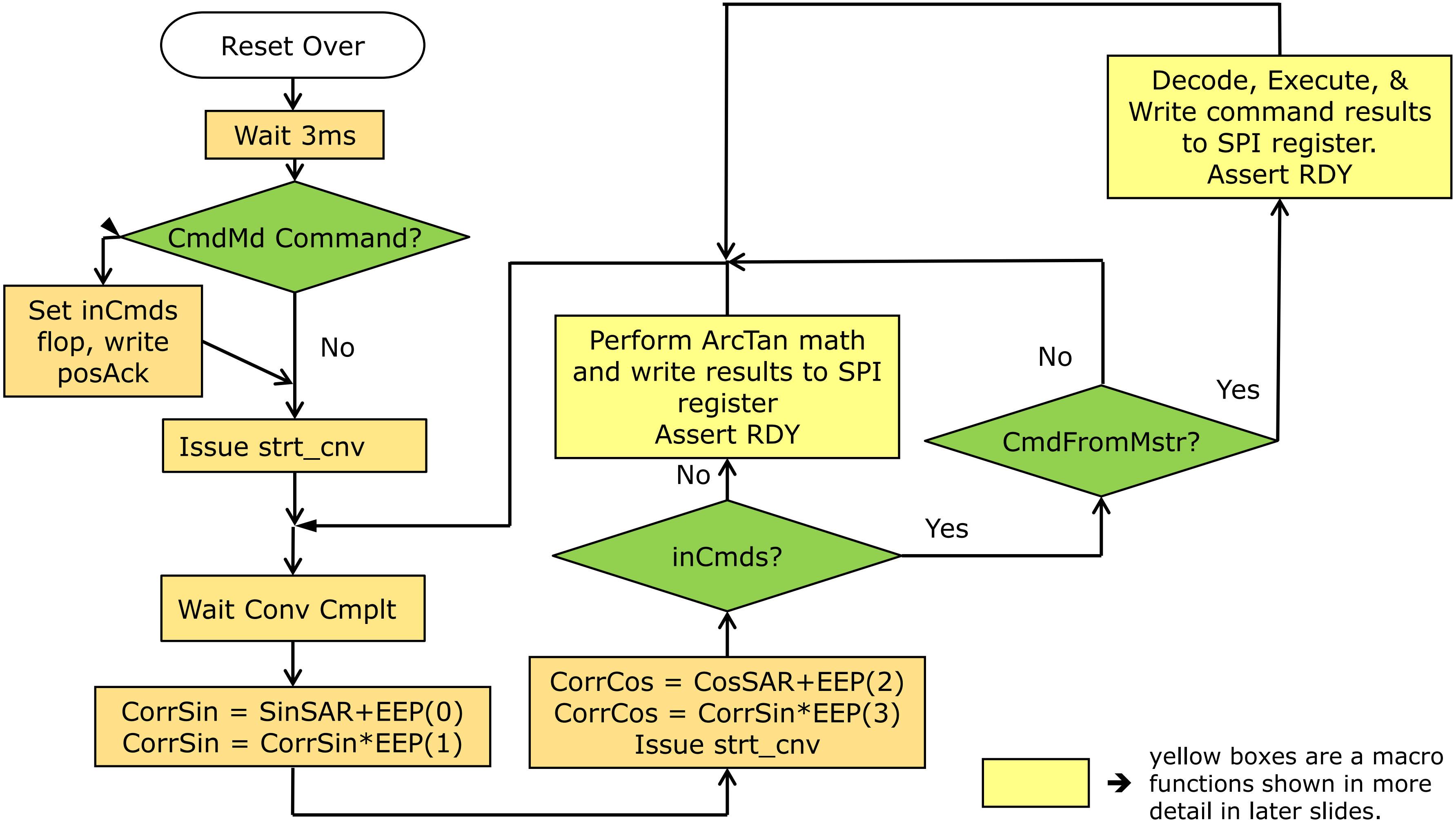
It is recommended to partition the digital core into 2 blocks. A datapath block will perform the computation and data movement. A control block (implemented as a FSM) will orchestrate all the data movement, and interface to the SPI bus, A2D, and EEPROM.

Signal interface of the digital core will be specified in the following slides. Signal interface between the datapath and the main state machine are left to the project team.

Digital Core Interface

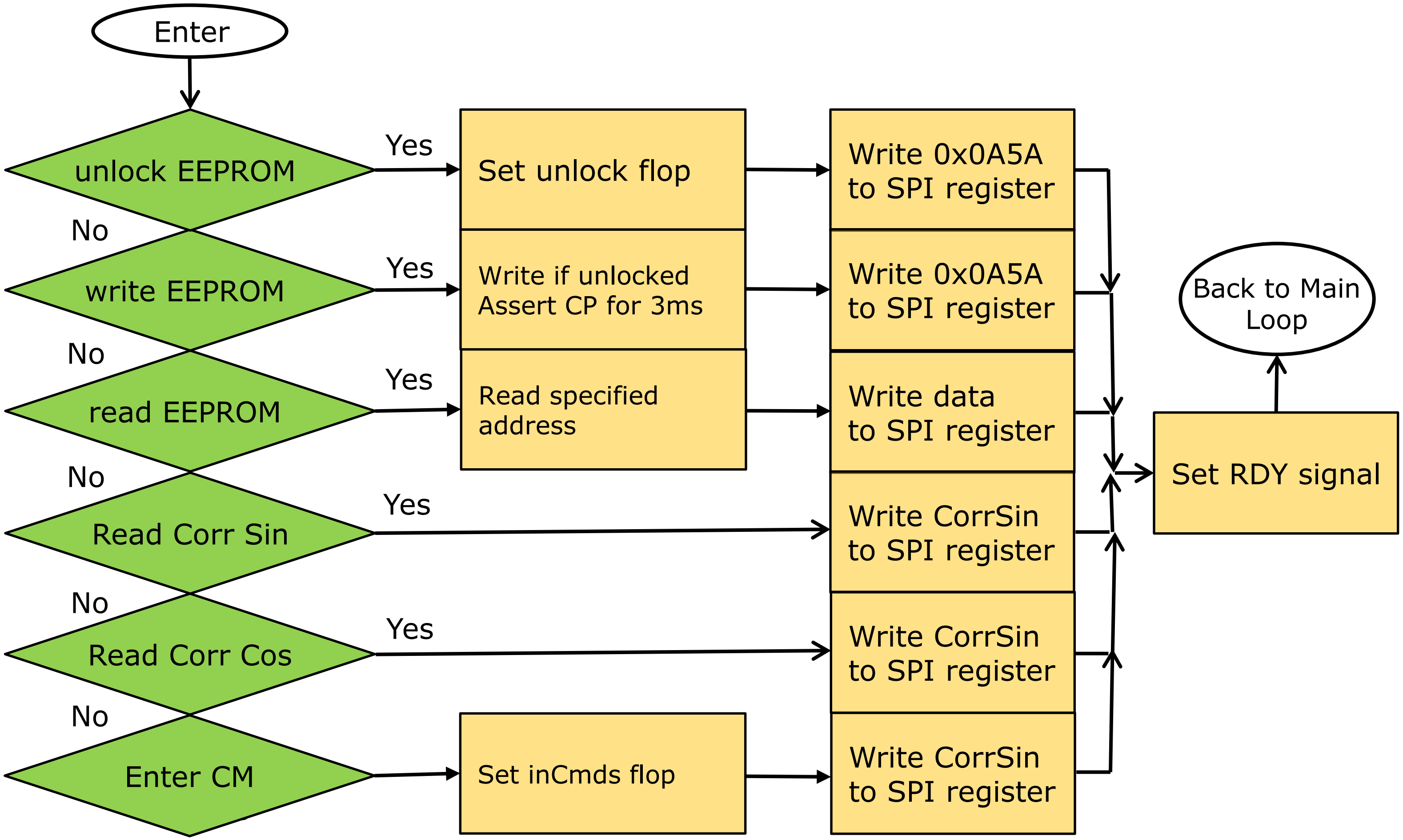
Signal Name:	Dir.	Description:
cmd_rdy	In	Signal from “modbus” interface is ready
cmd_rcvd[**15**:0]	In	**16**-bit data from “modbus” interface.
wrt_SPI	Out	Signal to SPI interface asserted when digital core writes a new output to SPI.
dst[11:0]	Out	Output data bus from core. Used for writes to the EEPROM, wraps internally to working registers. Lower 12-bits used for **16**-bit writes to modbus interface
eep_addr[1:0]	Out	Address to EEPROM
eep_rd_data[11:0]	In	Data from the EEPROM (returned on a read)
eep_cs_n	Out	Chip select (enable) to EEPROM block (active low)
eep_r_w_n	Out	Read/write control to EEPROM (active low for write)
chrg_pmp_en	Out	Signal enables on chip VPP charge pump (maintain for 3ms)
strt_cnv	Out	Signal to A2D interface instructs it to start a conversion on both channels
cnv_cmplt	In	Signal from A2D interface. Indicates both channels are ready
sinSAR	In	Angle sin value(result of A2D conversion)(to 12-bit DAC of A2D analog).
cosSAR	In	Angle cos value (result of A2D conversion)(to 12-bit DAC of A2D analog).
clk	In	Clock input to core (471.85MHz)
rst_n	In	Reset signal (active low)

Digital Core Main Flow



🟨 ➔ yellow boxes are a macro functions shown in more detail in later slides.

Decode & Execute Cmd


Enter
unlock EEPROM
Yes
Set unlock flop
Write 0x0A5A to SPI register
No
write EEPROM
Yes
Write if unlocked Assert CP for 3ms
Write 0x0A5A to SPI register
Back to Main Loop
No
read EEPROM
Yes
Read specified address
Write data to SPI register
Set RDY signal
No
Read Corr Sin
Yes
Write CorrSin to SPI register
No
Read Corr Cos
Yes
Write CorrSin to SPI register
No
Enter CM
Set inCmds flop
Write CorrSin to SPI register

Command Set

Command Encoding: All commands are 16-bits. Bits[15:12] contain the command endoding. Bits[13:12] contain the address if the command is EEPROM related. The lower 12-bits (Bits[11:0]) contain data in the case of EEPROM write, and are don't care otherwise.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits[15:14]	Bits[13:12]	Bits[11:0]	Description:
2'b01	2'b00	12'hxxx	Read corrected Sin value
2'b01	2'b01	12'hxxx	Read corrected Cos value.
2'b01	2'b10	12'hxxx	Enter command mode. This sets the inCmds flop
2'b01	2'b11	12'hxxx	Unlock EEPROM. This upper nibble [15:12] must be sent prior to any EEPROM write. It is used as an interlock to reduce probability of inadvertent writes.
2'b10	2'bYY	12'hxxx	Read EEPROM. The 2-bits (2'bYY) specify which location of EEPROM to read. Data is a don't care.
2'b11	2'bYY	12'hDATA	Write EEPROM. The location to write is specified by 2'bYY. The data to write is in the lower 12-bits.

Math

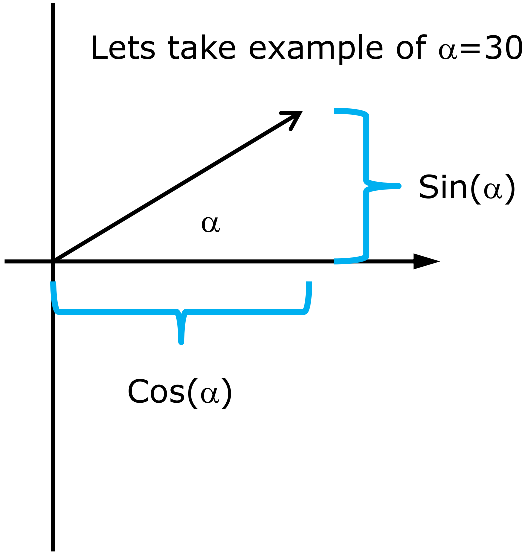
- The A2D values SinSAR and CosSAR are unsigned values that range from 0 to 0xFFF. They need to be made into signed numbers in the range from 0x800 to 0x7FF. This is done by simply inverting the MSB.

- Offset correction is performed on the rawA2D readings. EEPROM locations 0 and 2 contain signed offsets for SinSAR and CosSAR respectively.

- Next each of the offset corrected A2D readings have a gain term. The offset corrected SinSAR & CosSAR reading are multiplied by an attenuation term stored in EEPROM locations 1 and 3 respectively.

- Now it is time to compute the arctangent of (Sin/Cos) to derive α. What is your method? Have you considered CORDIC?

CORDIC Algorithm



Iter:	Rotation Amount	Resulting Angle	Sum of Rotations
1	-45	-15	-45
2	+22.5	7.5	-22.5
3	-11.25	-3.75	-33.75
4	+5.625	1.875	-28.125
5	-2.8125	-0.9375	-30.9375

If one had a convenient way of rotating a vector by binary weighted amounts starting at 45°. For instances ±45°,±22.5°,±11.25°,±5.625°, … Then any vector in quadrants I or IV (vectors with cos(α) >0) could be rotated to zero (so resulting vector lies along the x-axis). If one keeps track of the sum of all the rotations necessary to get the vector to zero then they know what the original angle α was.

CORDIC is an algorithm that can do such vector rotations, and with very simple digital blocks required. Adder, shifter, small lookup table, …

Implementing CORDIC (to 12-bit precision)

Need a lookup table with 12 entries of 12-bit vectors

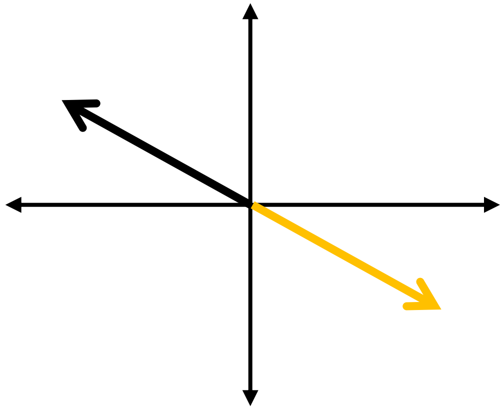
Normally we think of angles in radians, and having a range of [-π,π). However, real digital designers do not work with real numbers, they scale things to be integers.

We will define our own results. We will scale our number system to suite our needs. If we are representing an angle in the range [-π,π)with a 12-bit number then −π ➔ 0x800, and +π = 0x7FF.

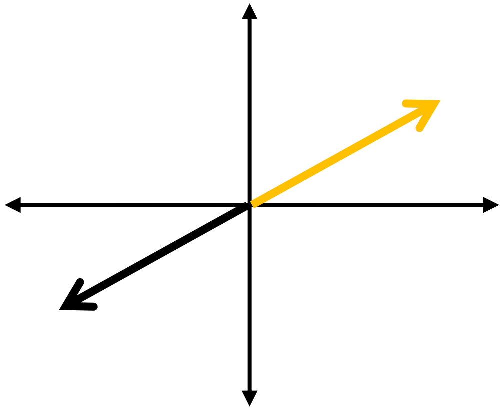
Using that scaling the hex values for our lookup table are as shown in the table to the right.

Indx:	Value:	Hex:
0	Arctan(1)	0x200
1	Arctan(1/2)	0x12E
2	Arctan(1/4)	0x0A0
3	Arctan(1/8)	0x051
4	Arctan(1/16)	0x029
5	Arctan(1/32)	0x014
6	Arctan(1/64)	0x00A
7	Arctan(1/128)	0x005
8	Arctan(1/256)	0x003
9	Arctan(1/512)	0x001
10	Arctan(1/1024)	0x001
11	Arctan(1/2048)	0x000

Implementing CORDIC (to 12-bit precision)



Rotate initial vector by +π, so we start CORDIC iterations with a vector in Quadrant IV



Rotate initial vector by -π, so we start CORDIC iterations with a vector in Quadrant I

CORDIC works well when the initial vector is in quadrant I or IV, but what about when cos(α) is negative?

Rotate the angle by π or –π by assigning cos(α) = -cos(α) and sin(α) = -sin(α).

If sin(α) was positive you will need to start your angle accumulator (used to sum all rotatons) at π.

If sin(α) was negative you will need to start your angle accumulator (used to sum all rotatons) at -π.

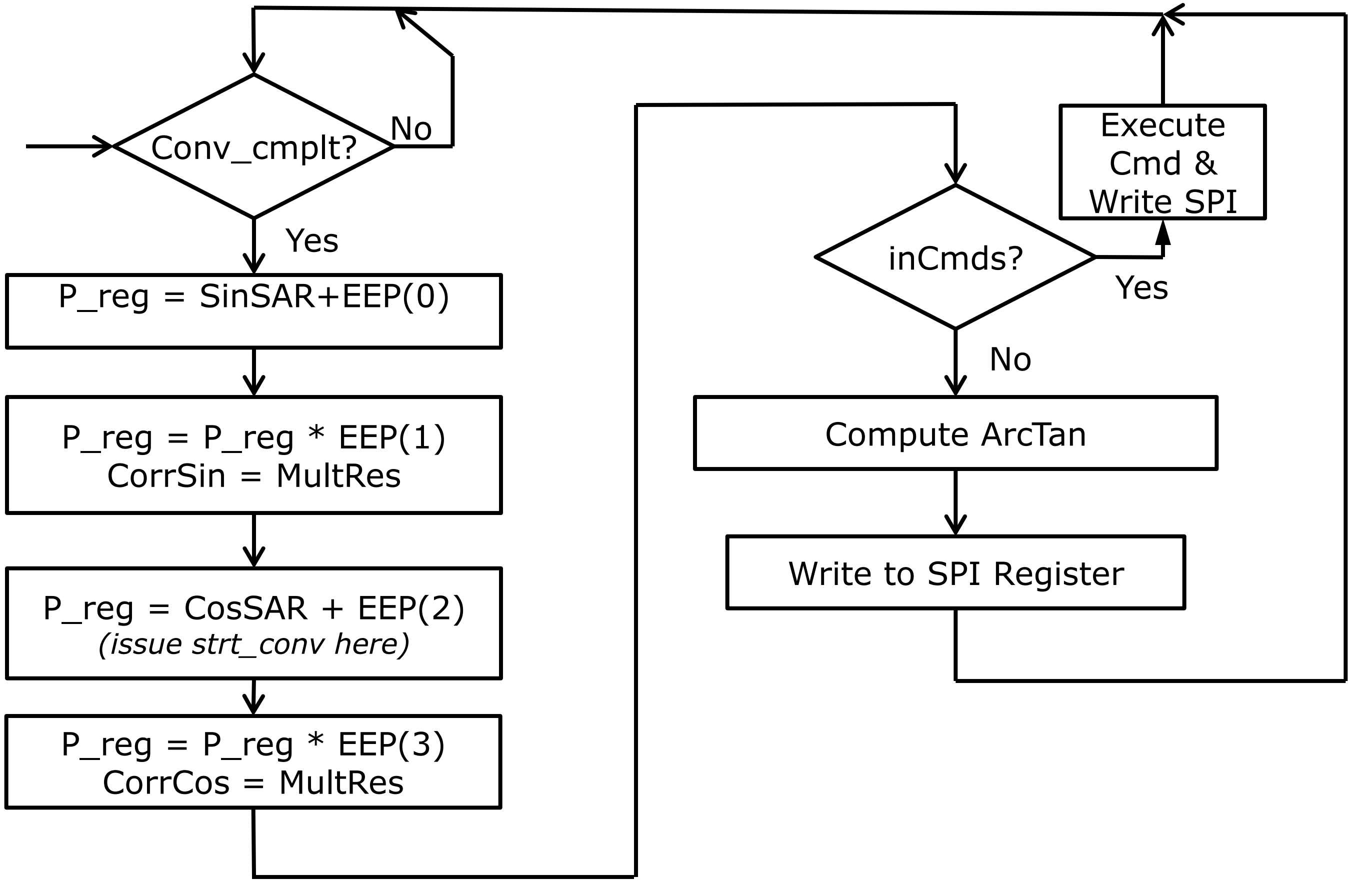
```
if (cos(a)<0) {
  if (sin(a)<0) {              // quad III case
    angle_accum = 0x800;       // negative π
  }
  else {                       // quad II case
    angle_accum = 0x7FF;       // positive π
  }
  cos(a) = -cos(a);
  sin(a) = -sin(a);
}
else {                         // Quad I & IV case
  angle_accum = 0;             // no initial rotation
}
```

Implementing CORDIC (the iterations)

```
for (iter=0; iter<12; iter++) {
  if (sin(a)>0) {                // clockwise rotation needed
    angle_accum = angle_accum + table(iter);
    nxt_cos(a) = cos(a) + (sin(a)>>>iter);
    nxt_sin(a) = sin(a) - (cos(a)>>>iter);
}
  else {                         // counter-clockwise rotation
    angle_accum = angle_accum - table(iter);
    nxt_cos(a) = cos(a) - (sin(a)>>>iter);
    nxt_sin(a) = sin(a) + (cos(a)>>>iter);
  }
}
```

Of course loop is not done as a “for” loop in verilog. It is done as a state machine that repeats its operations 12 times, and utilizes a 4-bit counter to know when it has done it 12 times. Do we have to iterate 12 times? Look at the last entry in the table.

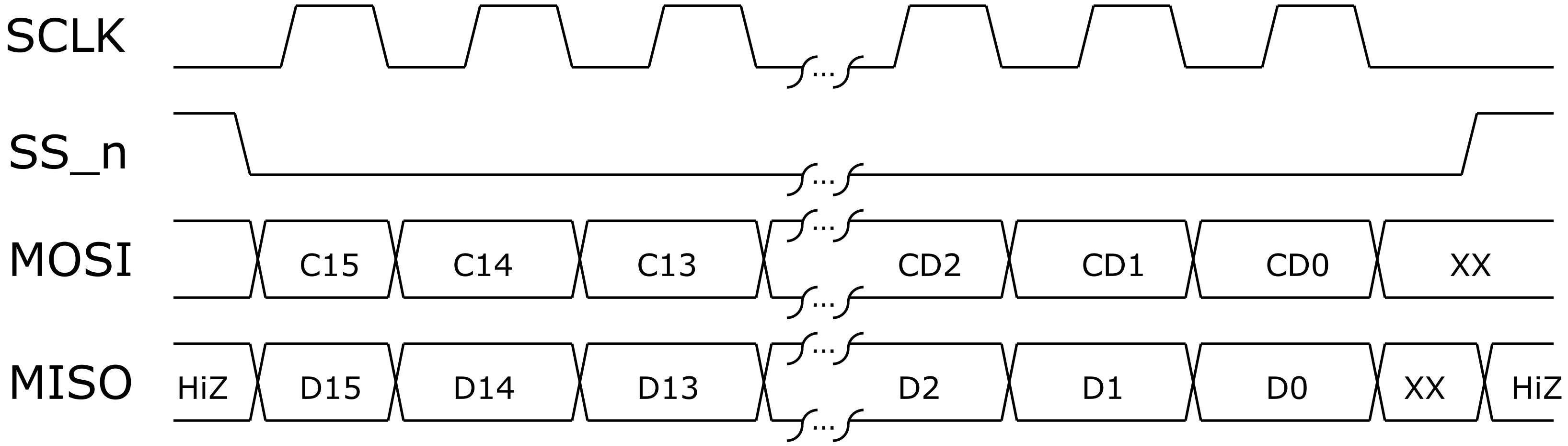
Normal Operation Loop



What is SPI?

- Simple full duplex serial interface (Motorola long long ago)
  - **S**erial **P**eripheral **I**nterconnect (very popular physical interface)
  - 4-wires
    - ✓ MOSI (Master Out Slave In) (CAR is a slave, your testbench will be a master)
    - ✓ MISO (Master In Slave Out) (CAR will drive this) (but only when SS_n is low)
    - ✓ SCLK (Serial Clock)
    - ✓ SS_n (Active low Slave Select) (If it is low then CAR is being selected)
  - There are many different variants
    - ✓ MISO changes 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 to the most commonly used variant
    - ✓ MISO changes on SCLK low
    - ✓ SCLK normally low
    - ✓ 16-bit packets (both directions will be 16-bit packets)

SPI Packets

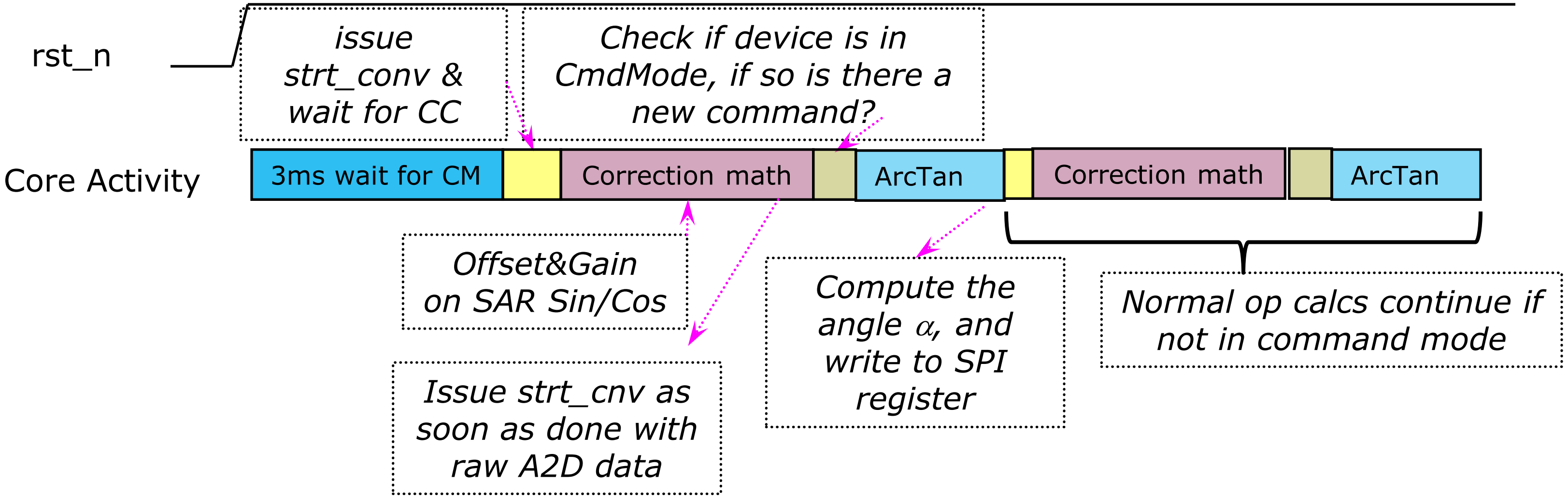


A SPI packet inherently involves a send and receive (full duplex). The full duplex packet is always initiated by the master. Master controls SCLK, SS_n, and MOSI. The slave drives MISO if it is selected. If the slave is not selected it should leave MISO high impedance.

MOSI will change on the falling edge of SCLK with the understanding that the slave will flop it on the falling edge of SCLK.

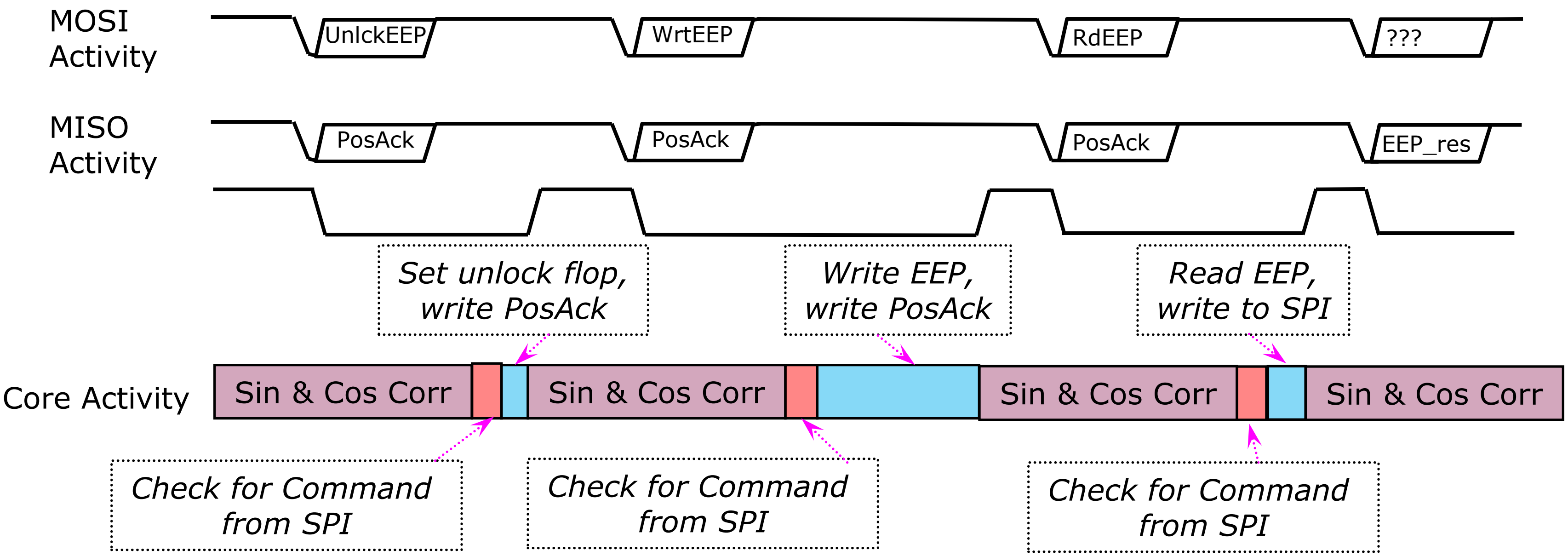
MISO will change on the falling edge of SCLK with the understanding that the master will flop it on the rising or falling edge of SCLK.

General Flow (coming out of reset)


rst_n
Core Activity
issue strt_conv & wait for CC
Check if device is in CmdMode, if so is there a new command?
3ms wait for CM
Correction math
ArcTan
Correction math
ArcTan
Offset&Gain on SAR Sin/Cos
Issue strt_cnv as soon as done with raw A2D data
Compute the angle α, and write to SPI register
Normal op calcs continue if not in command mode

Out of reset the core will wait for 3ms for a SPI command to come in. If a Enter Command Mode (EnterCM) command comes in then the core will enter a functional loop that is used for sensor calibration. If no EnterCM command comes in then the core will enter its normal operation loop (as shown here). In this loop it performs A2D measurements on the Sin & Cos inputs, corrects them for offset/gain errors, and computes the arctan to get the sensor angle α. This value α is then written to the SPI output register. The whole loop then repeats indefinitely.

Flow When in Command Mode



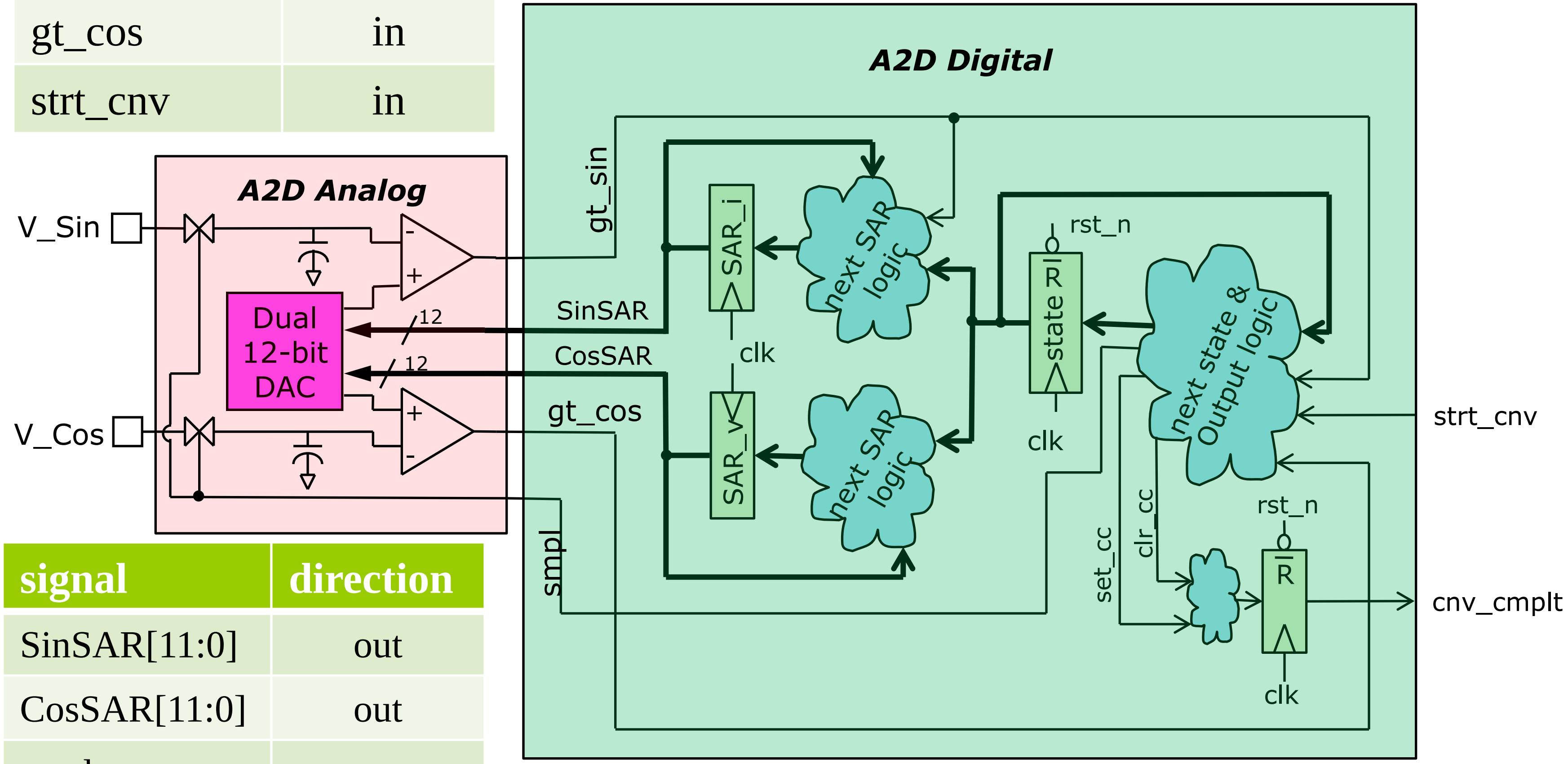
When in command mode the digital core still calculates offset and scaling correction on the sine and cosine channels. It then checks for an incoming command from the SPI bus. If a command has come in it will decode and execute that command and write a response to the SPI output register. The first command it sees in this example is an unlock EEPROM command. It sets the unlock bit, writes a positive acknowledge (16’h0A5A) to the SPI bus. The core then returns for another round of sine/cosine measurement and correction. When completed it sees the next command is a write to EEPROM (the output on MISO during this time is a posAck to the previous UnlckEEP command). Since the unlock bit is set from the previous command it writes EEPROM, the core then writes a positive acknowledge to SPI and returns to the sine/cosine correction.

A2D Interface

signal	direction
gt_sin	in
gt_cos	in
strt_cnv	in

NOTE: Each value of SAR_i/SAR_v should be held for **128** clocks to give time for the analog comparators to settle.

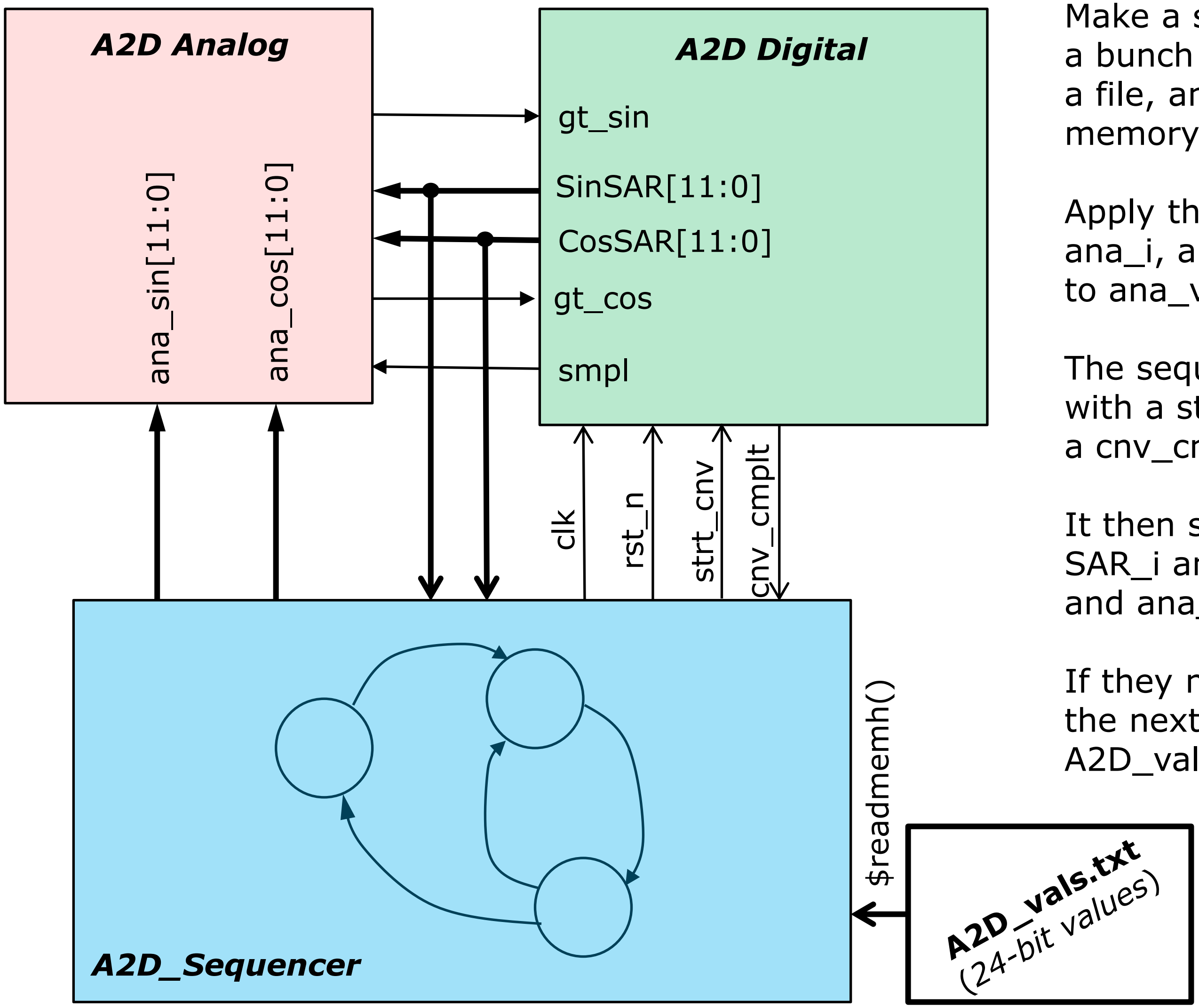
NOTE: ***smpl*** signal asserted for 128 clocks after ***strt_cnv***



signal	direction
SinSAR[11:0]	out
CosSAR[11:0]	out
smpl	out
cnv_cmplt	out

- A2D_Analog verilog will be provided
- You create A2D_Digital

Possible Testbench for A2D Digital



Make a sequencer that reads a bunch of 24-bit values from a file, and packs them in a memory (use $readmemh()).

Apply the upper 12-bit to ana_i, and the lower 12-bits to ana_v.

The sequencer then hits it with a strt_cnv, and waits for a cnv_cmplt.

It then self checks the values SAR_i and SAR_v vs ana_i, and ana_v respectively.

If they match it moves on to the next value specified in A2D_vals.txt

EEPROM

- The EEPROM is an 4-entry 12-bit word EEPROM. There is a on-chip charge pump used to provide the programming voltage from the nominal 1.8V supply.

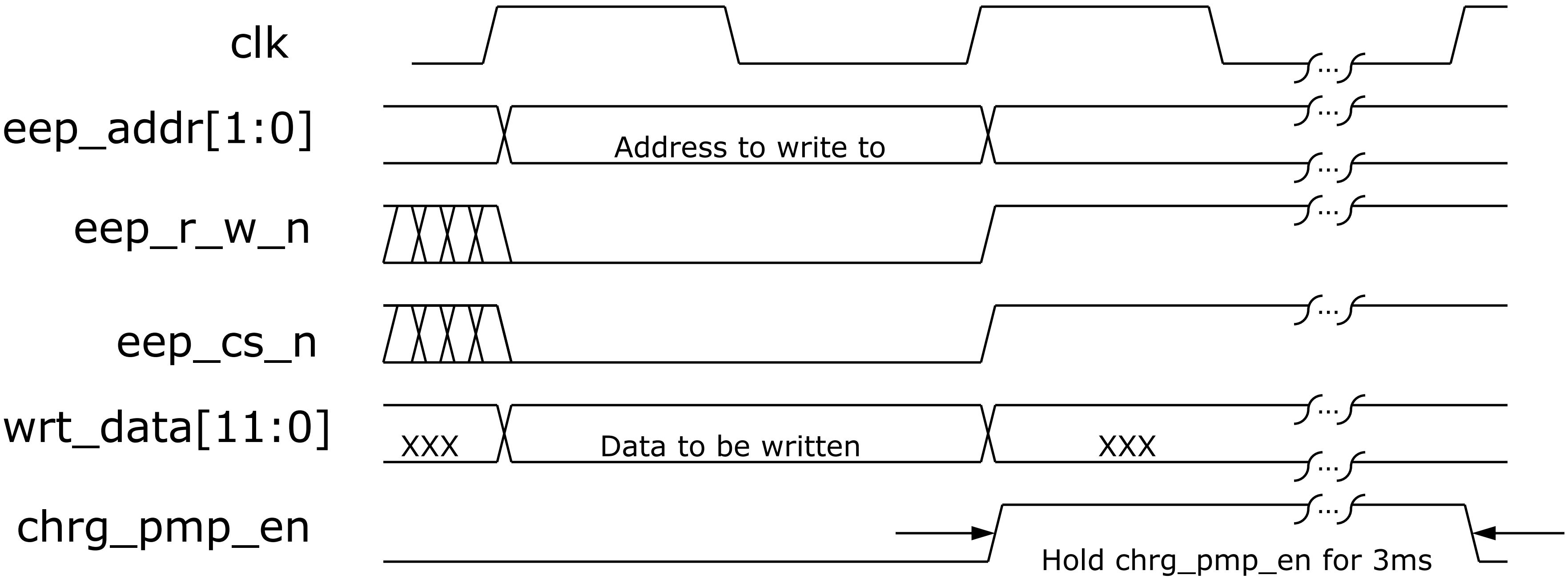
EEPROM Map

Address:	Description:
00	Offset term for SinSAR
01	Gain term for SinSAR
10	Offset term for CosSAR
11	Gain term for CosSAR

EEPROM Signal Interface

Signal Name:	Direction	Description:
clk / rst_n	In	Hook to main system clock & reset
eep_addr[1:0]	in	Address to EEPROM
wrt_data[11:0]	in	Data to be written to EEPROM
rd_data[11:0]	out	Data from EEPROM when read.
eep_cs_n	in	Active low chip select. Bus operation is for EEPROM
eep_r_w_n	in	Bus operation is write when 0, and read when 1
chrg_pmp_en	in	Enables Vpp charge pump (hold for at least 3ms)

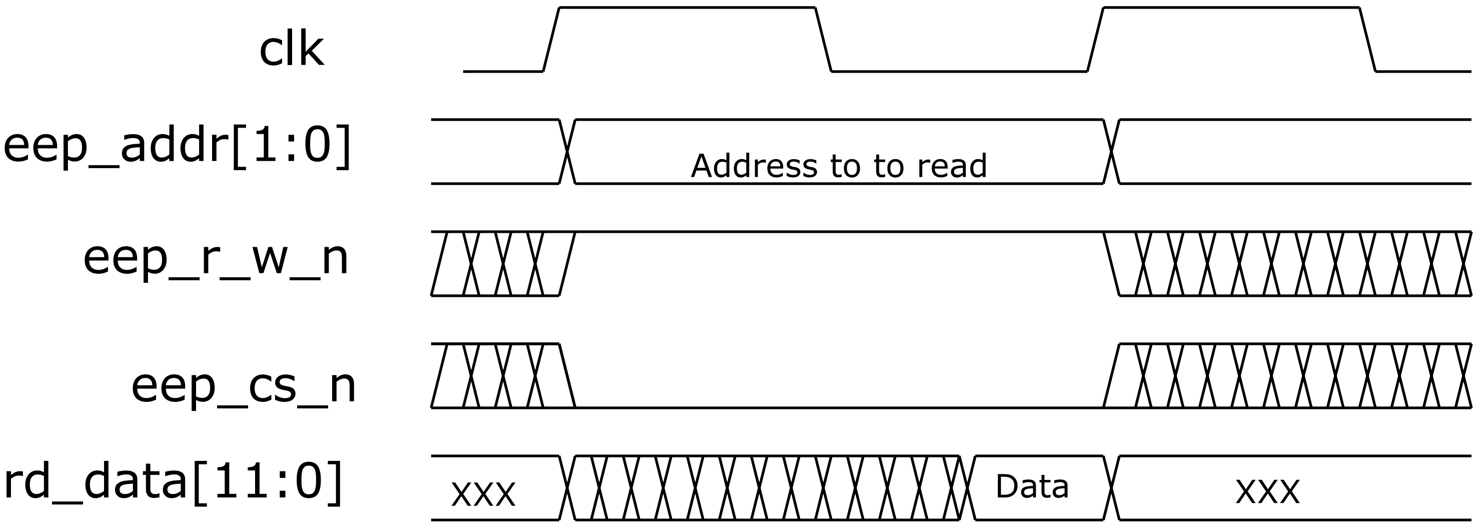
EEPROM Timings (write)



Address and data to be written only have to be held for 1 clock cycle.

chrg_pmp_en, however has to be maintained for 3ms during a write.

EEPROM Timings (read)

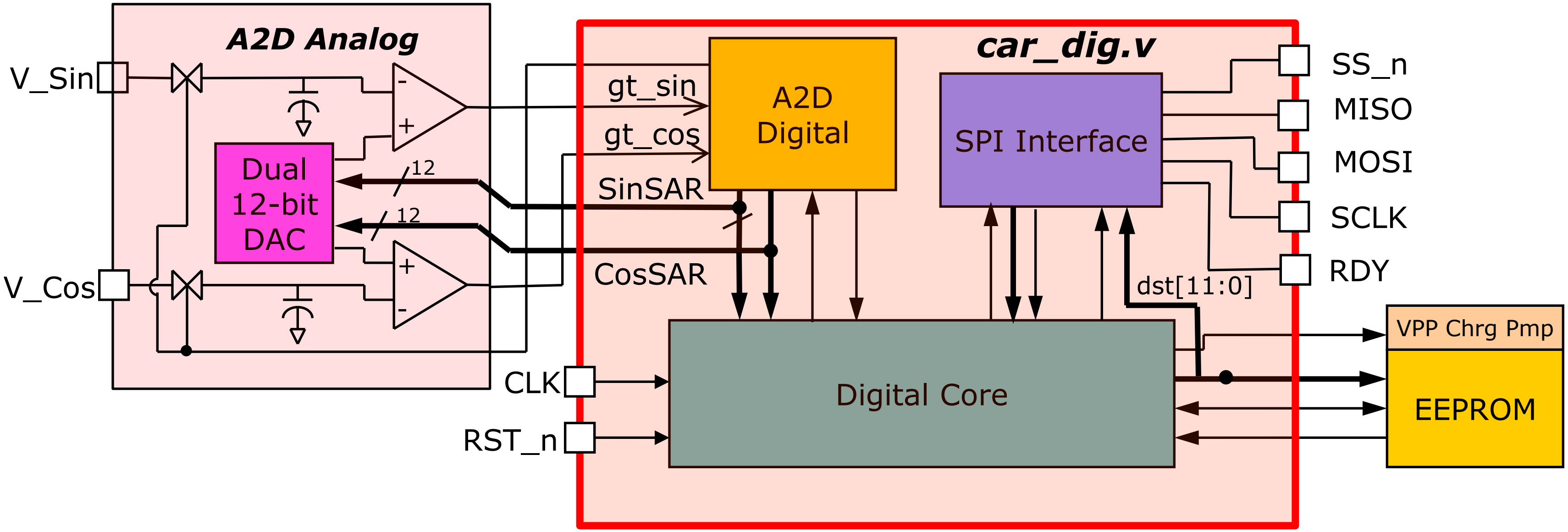


The EEPROM is single cycle read, but the data out will be valid late in the cycle.

chrg_pmp_en should be low the entire time.
chrg_pmp_en is used to fire up the on chip charge pump that provides the programming voltage. Therefore it is only used during EEPROM writes.

Required Hierarchy & Interface

- Your Design will be placed in an “Eric” Testbench to validate its functionality. Therefore, it must be pin for pin compatible with our testbench.



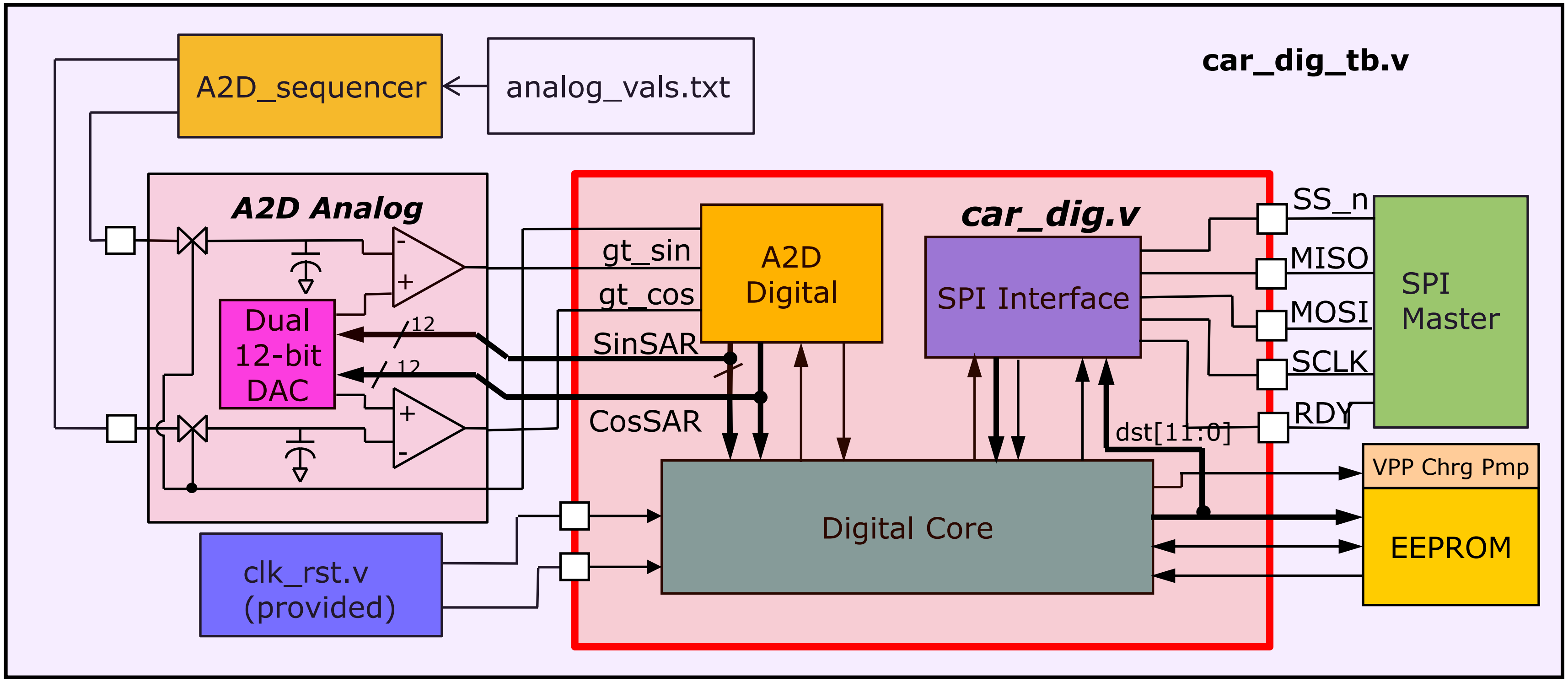
CAR = **C**alibratable **A**ngle **R**esolver

You Must have a block called **car_dig.v** which is top level of what will be the synthesized DUT. The interface of car_dig.v **must match exactly** to our specified car_dig.v interface **Please copy car_dig.v** (interface skeleton) from **~ece551/public**.

The hierarchy of your testbench above car_dig.v is up to your team.

Recommended Testbench Hierarchy

Below is the recommended hierarchy for the full chip testbench. You are not required to do it this way, but if you do there is a template file available in ~ece551/public



A file called **car_dig_tb.v** can be copied from **~ece551/public**. It is a template of a testbench with this hierarchy.

Available Models & Hierarchy

- Interface &Model files available at:
  - ~ejhoffman/ece551/public

File Name:	Description:
car_dig_tb.v	**Optional** testbench template file. Note currently includes a tasks file in my area. You should change this path.
car_dig.v	**Requried** interface skeleton verilog file. **Copy this** and flush it out with your design
A2D_analog.v	Models DAC & comparator, outputs gt signal.
A2D_sequencer.v	Sequencer that reads values from a file (analog_vals.txt) and applies them as the "analog" stimulus to the FB input of A2D_analog.v. **User must modify file path** to analog_vals.txt
eep.v	Model of the 4x12bit EEPROM. Reads initial values from eep_init.txt. **User must modify file path!**
eep_init.txt	Example of file for EEPROM initial values.
analog_vals.txt	A data set containing data records for testing.