ECE 551
Project Spec

Fall ’12

High Speed Calibratable Angle Resolver
(High Speed CAR)
Grading Criteria: (Project is 30% of final grade)

- Project Grading Criteria:
  - Quantitative Element 25%
    
    (yes this could result in extra credit)

    \[ \text{Quantitative} = \frac{\text{Eric}_\text{Project Area}}{\text{YourSynthesized Area}} \]

  - 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)

- Note: The design has to be functionally correct for this to apply
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
Description: The CAR chip is a high speed, high precision angle resolver. It has a 2-channel 12-bit A2D converter to sample a sine and cosine signal from an angle sensor. The sine channel produces an output that is $A \sin(\alpha) + B$, and the cosine channel produces a signal that is $C \cos(\alpha) + D$. The B and D terms represent undesired offsets that have 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 $\alpha$ 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

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

<table>
<thead>
<tr>
<th>EEPROM Address</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OffsetSin</td>
</tr>
<tr>
<td>1</td>
<td>ScaleSin</td>
</tr>
<tr>
<td>2</td>
<td>OffsetCos</td>
</tr>
<tr>
<td>3</td>
<td>ScaleCos</td>
</tr>
</tbody>
</table>
Calculating $\alpha$

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

However this only gives an answer in the Range of $-\pi/2$ to $\pi/2$

We have to correct for the other quadrants (when CorrectedCos($\alpha$) is negative)

How do you compute ArcTan?

Are you really going to implement a divide?

Remember HW1. Look at HW Solution.
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 carDig.v which is top level of what will be the synthesized DUT.
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

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

- **Reset Over**
  - Wait 3ms
  - CmdMd Command?
    - **Set inCmds flop, write posAck**
    - Issue strt_cnv
    - Wait Conv Cmplt
    - CorrSin = SinSAR+EEP(0)
    - CorrSin = CorrSin*EEP(1)

- **CmdFromMstr?**
  - Yes
    - Decode, Execute, & Write command results to SPI register. Assert RDY
  - No
    - **Perform ArcTan math and write results to SPI register Assert RDY**
    - inCmds?
      - No
        - **Issue strt_cnv**
      - Yes
        - CorrCos = CosSAR+EEP(2)
        - CorrCos = CorrSin*EEP(3)
        - Issue strt_cnv

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
- write EEPROM (Yes)
  - Write if unlocked
  - Assert CP for 3ms
  - Write 0x0A5A to SPI register
- read EEPROM (Yes)
  - Read specified address
  - Write data to SPI register
- Read Corr Sin (Yes)
  - Write CorrSin to SPI register
- Read Corr Cos (Yes)
  - Write CorrSin to SPI register
- Enter CM (Yes)
  - Set inCmds flop
  - Write CorrSin to SPI register

- No
- Unlock EEPROM
- Write EEPROM
- Read EEPROM
- Read Corr Sin
- Read Corr Cos
- Enter CM

Set RDY signal
Back to Main Loop
Command Set

**Command Encoding:** All commands are 16-bits. Bits[15:12] contain the command encoding. 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2'b01</td>
<td>2'b00</td>
<td>12’hxxxx</td>
<td>Read corrected Sin value</td>
</tr>
<tr>
<td>2'b01</td>
<td>2'b01</td>
<td>12’hxxxx</td>
<td>Read corrected Cos value.</td>
</tr>
<tr>
<td>2'b01</td>
<td>2'b10</td>
<td>12’hxxxx</td>
<td>Enter command mode. This sets the inCmds flop</td>
</tr>
<tr>
<td>2'b01</td>
<td>2'b11</td>
<td>12’hxxxx</td>
<td>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.</td>
</tr>
<tr>
<td>2'b10</td>
<td>2'bYY</td>
<td>12’hxxxx</td>
<td>Read EEPROM. The 2-bits (2’bYY) specify which location of EEPROM to read. Data is a don’t care.</td>
</tr>
<tr>
<td>2'b11</td>
<td>2'bYY</td>
<td>12’hDATA</td>
<td>Write EEPROM. The location to write is specified by 2’bYY. The data to write is in the lower 12-bits.</td>
</tr>
</tbody>
</table>
Math

- The A2D values SinSAR and CosSAR are unsigned values that range from 0 to 0xFFFF. 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 $\alpha$. What is your method? Have you considered CORDIC?
CORDIC Algorithm

Lets take example of $\alpha=30$

<table>
<thead>
<tr>
<th>Iter</th>
<th>Rotation Amount</th>
<th>Resulting Angle</th>
<th>Sum of Rotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-45</td>
<td>-15</td>
<td>-45</td>
</tr>
<tr>
<td>2</td>
<td>+22.5</td>
<td>7.5</td>
<td>-22.5</td>
</tr>
<tr>
<td>3</td>
<td>-11.25</td>
<td>-3.75</td>
<td>-33.75</td>
</tr>
<tr>
<td>4</td>
<td>+5.625</td>
<td>1.875</td>
<td>-28.125</td>
</tr>
<tr>
<td>5</td>
<td>-2.8125</td>
<td>-0.9375</td>
<td>-30.9375</td>
</tr>
</tbody>
</table>

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 $\cos(\alpha) > 0$) 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 $\alpha$ 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

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

Normally we think of angles in radians, and having a range of \([-\pi, \pi]\). 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 \([-\pi, \pi]\) with a 12-bit number then \(-\pi \mapsto 0x800\), and \(+\pi = 0x7FF\).

Using that scaling the hex values for our lookup table are as shown in the table to the right.
Implementing CORDIC (to 12-bit precision)

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

Rotate the angle by \( \pi \) or \(-\pi\) by assigning \(\cos(\alpha) = -\cos(\alpha)\) and \(\sin(\alpha) = -\sin(\alpha)\).

If \(\sin(\alpha)\) was positive you will need to start your angle accumulator (used to sum all rotations) at \(\pi\).

If \(\sin(\alpha)\) was negative you will need to start your angle accumulator (used to sum all rotations) at \(-\pi\).

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

Conv_cmplt? Yes

P_reg = SinSAR + EEP(0)

P_reg = P_reg * EEP(1)
CorrSin = MultRes

P_reg = CosSAR + EEP(2)
(issue strt_conv here)

P_reg = P_reg * EEP(3)
CorrCos = MultRes

inCmds? No

Execute Cmd & Write SPI

inCmds? Yes

Compute ArcTan

Write to SPI Register
What is SPI?

- Simple full duplex serial interface (Motorola long long ago)
  - Serial Peripheral Interconnect (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.
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 $\alpha$. This value $\alpha$ is then written to the SPI output register. The whole loop then repeats indefinitely.
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 and Direction

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt_sin</td>
<td>in</td>
</tr>
<tr>
<td>gt_cos</td>
<td>in</td>
</tr>
<tr>
<td>strt_cnv</td>
<td>in</td>
</tr>
</tbody>
</table>

### Dual 12-bit DAC

- **V_Sin**
- **V_Cos**

### A2D Analog

- **gt_sin**
- **gt_cos**

### A2D Digital

- **SAR_i**
- **SAR_v**
- **clk**
- **Rst_n**
- **Cnv_cmplt**
- **Set_cc**
- **Clr_cc**
- **Smpl**

### Notes

- **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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Offset term for SinSAR</td>
</tr>
<tr>
<td>01</td>
<td>Gain term for SinSAR</td>
</tr>
<tr>
<td>10</td>
<td>Offset term for CosSAR</td>
</tr>
<tr>
<td>11</td>
<td>Gain term for CosSAR</td>
</tr>
</tbody>
</table>

### EEPROM Signal Interface

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

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.
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 `carDig_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

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>car_dig_tb.v</td>
<td><strong>Optional</strong> testbench template file. Note currently includes a tasks file in my area. You should change this path.</td>
</tr>
<tr>
<td>car_dig.v</td>
<td><strong>Required</strong> interface skeleton verilog file. <strong>Copy this</strong> and flush it out with your design</td>
</tr>
<tr>
<td>A2D_analog.v</td>
<td>Models DAC &amp; comparator, outputs gt signal.</td>
</tr>
<tr>
<td>A2D_sequencer.v</td>
<td>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. <strong>User must modify file path</strong> to analog_vals.txt</td>
</tr>
<tr>
<td>eep.v</td>
<td>Model of the 4x12bit EEPROM. Reads initial values from eep_init.txt. <strong>User must modify file path</strong>!</td>
</tr>
<tr>
<td>eep_init.txt</td>
<td>Example of file for EEPROM initial values.</td>
</tr>
<tr>
<td>analog_vals.txt</td>
<td>A data set containing data records for testing.</td>
</tr>
</tbody>
</table>