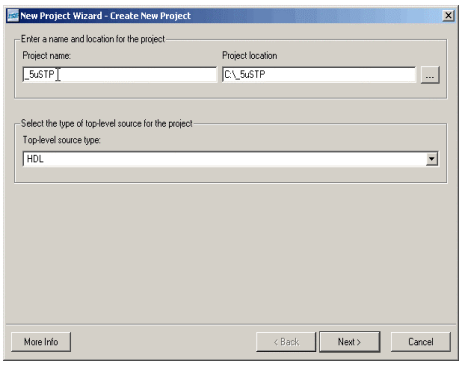
5-MICROSTEP DRIVE CPLD TUTORIAL

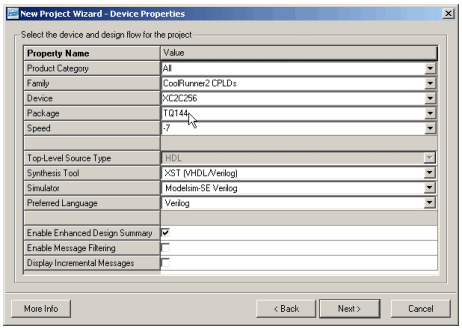
START A NEW ISE 10.1 PROJECT:



Open ISE 10.1 and from “Files” select “New Project…”

In “Project name:” enter “_5uSTP” or any other name you may wish to use. Do not start a project name with a number. For instance, “5uSTP” will not work.

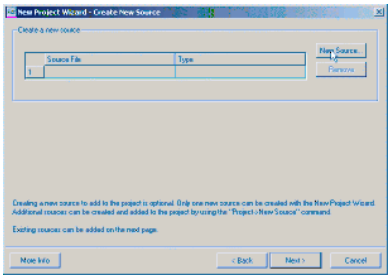
Select “HDL” for the “Top-level source” and hit “Next>”



Now enter your CPLD information. If you are using the Digilent CRII board, select from the pull-downs as shown:

Product Category	All
Family	CoolRunner2 CPLDs
Device	XC2C256
Package	TQ144
Speed	-7
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim-SE Verilog
Preferred Language	Verilog
Enable Summary	“Check”

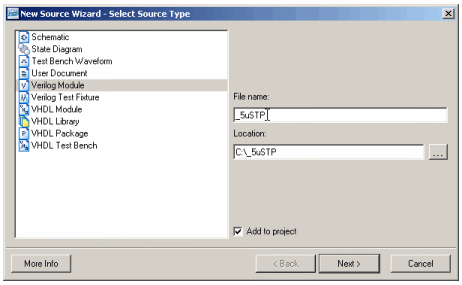
Hit “Next>”, then “New Source…”



Select “Verilog Module” and in “File name” enter the name “_5uSTP”.

Hit “Next>” again. When you see “The directory ‘C:_5uSTP’ doesn’t exist, do you want to create it?” hit “Yes”.

Hit “Next>” repeatedly until you see the following:

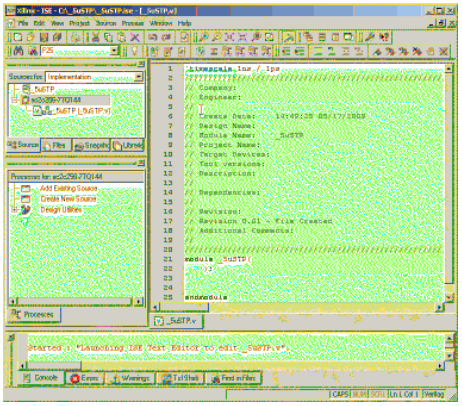


This is the main ISE project screen for the tutorial project. Verilog code is a skeleton starting with the blue text “module” and ending with “endmodule”. We will do our best to fill the space between with code.

There are a bunch of buttons at the top which we will largely ignore and 3 panes which we will not. The big pane on the right displays Verilog code. Green is comments, blue is reserved Verilog names and black is what you write. Pink is I don’t know what. This pane has a tab “_5uSTP.v” and means it is displaying code for that module.

The upper left pane is less useful. It shows “_5uSTP.v” with a little green square. That means it is the “top module”. There will be other modules but only one top module.

The middle left pane is an important one and it will be covered a little later. It kicks-off implementing what’s in the big pane among other things. Implementing means generating the .jed file used to JTAG the CPLD. That file is the end product of all this work.



The bottom one is the pane of pain. It displays implementation progress, shows warnings and issues errors. Errors are showstoppers. The easy ones are syntax errors and the notes displayed are helpful in correcting them. The dreaded ones are “not enough macrocells to fit the CPLD” and “insufficient registers to fit the CPLD” when you squeeze a design into a small (read as inexpensive) CPLD. You will learn to dread the red, round “X”.

I like to pare the green comments header down to a minimum. The big pane works just like Notepad so I erase most of the extra stuff I just don't need.

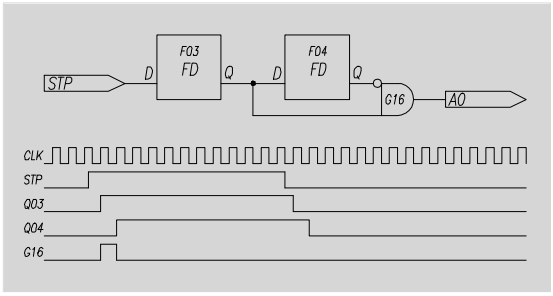
```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  //
4  // Create Date:    14:49:25 05/17/2009
5  // Module Name:    _5uSTP
6  //
7  //////////////////////////////////////////////////////////////////////////////////
8  module _5uSTP(
9      );
10
11
12 endmodule
13
```

Comments in Verilog must be preceded with "//" and then anything you write further on that line is a comment. If you need to block a bunch of stuff out, preceded it with "/*" and end the block-out portion with "*/".

Say you wrote some Verilog code and then you thought of a more efficient way of doing it but you're not sure it will work. You don't want to erase the working code, so you enter "/*" in front of the code you want to disable and "*/" at the end of it.
If the new stuff doesn't work out, erase it and remove the blocks. If it does, erase the old stuff, blocks and all.

CODE EXAMPLE:

Let's start with a very simple but extremely useful circuit.



Good CPLD practice is to make the code fully synchronous. It means if something has a clock input; a D-Flop, counter, shift-register, whatever, its clock is connected to the CPLD system clock. It also means the clock is not available to you in any way. Look at the diagram to the right. It shows two D-Flops and a NAND gate. The diagrammed D-Flops don't even show a clock input because it is assumed all clocks are connected to the system clock input.

What "stands in" for a traditional clock in a synchronous system is the CE input (clock enable). When CE is enabled, the counter, shift-register or other clocked device will 'see' clock edges while enabled and act according to its function. A single clock period CE enable permits only one clock pulse to be seen. This circuit serves the purpose of taking a rising edge signal, (an asynchronous STEP pulse input in this case) and outputting a single clock period wide pulse on the rising edge of the input. This pulse will go to gating that connects to the CE input of a downstream device.

How it works: The D input of the D-Flop named "F03" is connected to a CPLD I/O pin named "STP". On the first rising CLK edge Q03 goes high after its D input goes high. F03 and F04 form a 2-bit shift-register. Q04 goes high one CLK period after Q3. An AND gate named G16 decodes Q03 being high and Q04 being low. Its output goes high for that one CLK period and at no other time.

This circuit gets used again and again, anytime you have to clock off on the edge of a signal.

How it's designed: The circuit uses two simple D-Flops (no CE, no RESET, no SET). You have to design this kind of D-Flop. Good bet this kind of D-Flop will be used elsewhere in the circuit for a different purpose as well so you have to "build" it.

Verilog uses a "top module" and any number of sub-modules. Think of these as subroutines. Write them once, use them often. You don't have a simple D-Flop until you write a module for it so here it is:

```
module DF (input C, D, output Q);
        reg X = 0;
        always @(posedge C)
          X <= D;
assign Q = X;
endmodule
```

Verilog reserved names are in blue. You cannot use reserved names to anywhere in your program to name your stuff. The first line starts with module and ends with endmodule. This sets a beginning and an end for the code that defines stuff, a D-Flop in this instance. The first line declares the module name (I used "DF" for D-flop) and it enumerates all inputs and outputs to the module (I used C for the clock, D for the D-input and Q for the only output).

The second line defines the register name which I call "X" because I have no imagination. The next lines read as "On every positive edge of the C input, register X output becomes what is on its D input. Simple enough, it's what a D-Flop is supposed to do. The assign Q = X line makes the module's Q output become what's on X. That's it, module DF is now a simple D-Flop. This module can be called-up anytime a simple D-Flop is needed somewhere.

Here's how it's called-up:

```
DF F03 (.C(CLK), .D(STP), .Q(Q03));
```

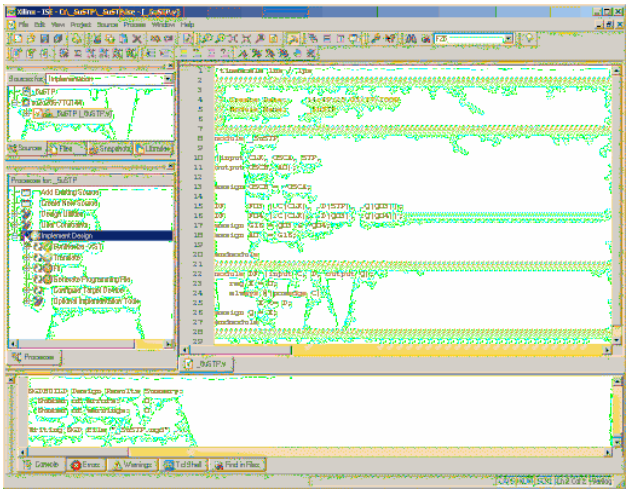
"DF" is the module DF and F03 is its name used where it's needed. .C(CLK) says "Connect its C input to the system clock input called CLK, .D(STP) to connect the D input to an I/O pin named STP and .Q(Q03) says to name its Q output as Q03. Now add the following to the top module:

```
(input CLK, STP,
output A0);

DF     F03 (.C(CLK), .D(STP), .Q(Q03));
DF     F04 (.C(CLK), .D(Q03), .Q(Q04));
assign G16 = Q03 & ~Q04;
assign A0  = G16;
```

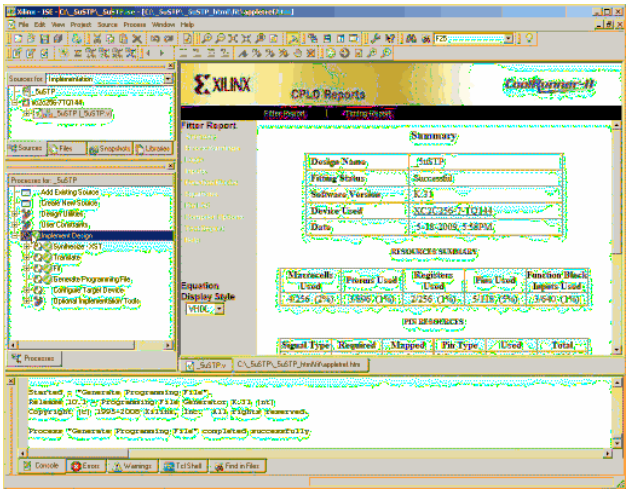
The first two lines declare CLK and STP as inputs and A0 as an output in the top module. The next two lines read as "Connect F03 D-Flop's D input to the I/O pin named STP. Connect its Q output named Q03 to F04's D input. Use an AND gate named G16 whose inputs are connected to Q03 and the inverted Q04. Output G16 to an I/O pin called "A0".

The finished code should look like:

```
module _5uSTP
(input CLK, STP,
output A0);
DF     F03 (.C(CLK), .D(STP), .Q(Q03));
DF     F04 (.C(CLK), .D(Q03), .Q(Q04));
assign G16 = Q03 & ~Q04;
assign A0  = G16;
endmodule
////////////////////////////////////
module DF (input C, D, output Q);
       reg X = 0;
       always @(posedge C)
         X <= D;
assign Q = X;
endmodule
////////////////////////////////////
```



Save the file and then double-click "Implement Design". This starts the compiling of the design. The middle pane shows the progress of the process while the bottom pane shows the details of what's happening. Be watching for Warnings and Errors. If there are no errors, ISE finishes with the following display.



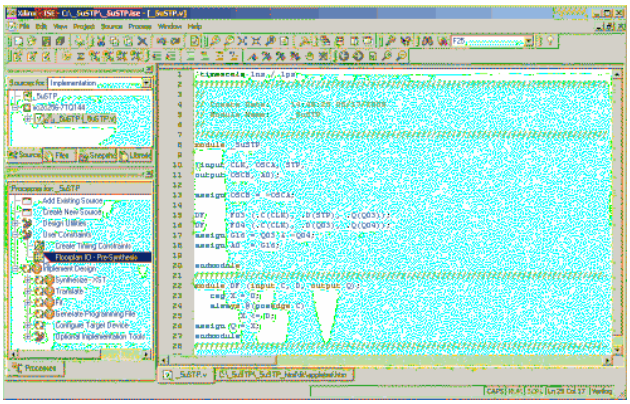
This display shows a RESOURCE SUMMARY table to let you know how many Macrocells, Pterms, Registers, Pins and Function blocks were used to implement the design.

Errors/Warnings in the Fitter Report gives better and more detailed information about the Error and/or Warning cause than the bottom pane does.

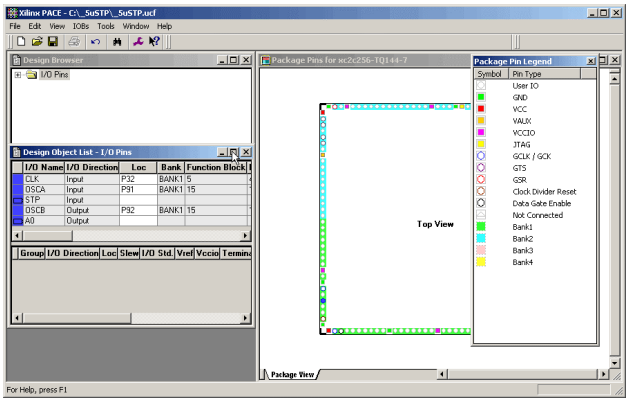
My recommendation is not to assign I/O pins until the entire project is completed. Certain idiosyncrasies of ISE makes it a difficult process to reassign I/O pin numbers and it is particularly difficult to rename I/O pins. So difficult in fact it's sometimes easier to start a new project instead.

Another good idea is to copy the Verilog text in main display and paste it into Notepad as work progresses. This way you have a convenient text file available and you don't have to open ISE every time there is something you want to look at in your code. It works in reverse too; you can copy the Notepad text and paste it into ISE. I do this a lot when I start a new ISE project. I open a Notepad file that has behavioral modules for counters, D-Flops, etc. and paste them into the new project.

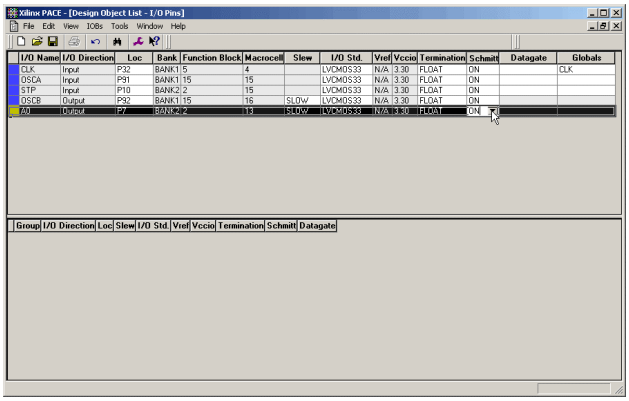
Assigning I/O pins: Open User Constraints and double click the Floorplan IO – Pre-Synthesis button. This brings up a window called Xilinx PACE.



From PACE, expand the Design Object List window.

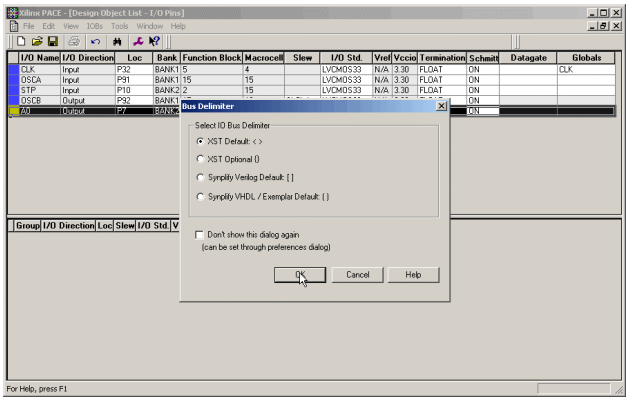


The project's I/O pin names should appear in the left column and their I/O direction is in the next column. Every space with a white background can be filled in. Once a pin number is entered in the Loc column, set the Slew to "SLOW", the I/O Std. To "LVCMOS33", the Termination to "FLOAT" and the Schmitt column to "ON".



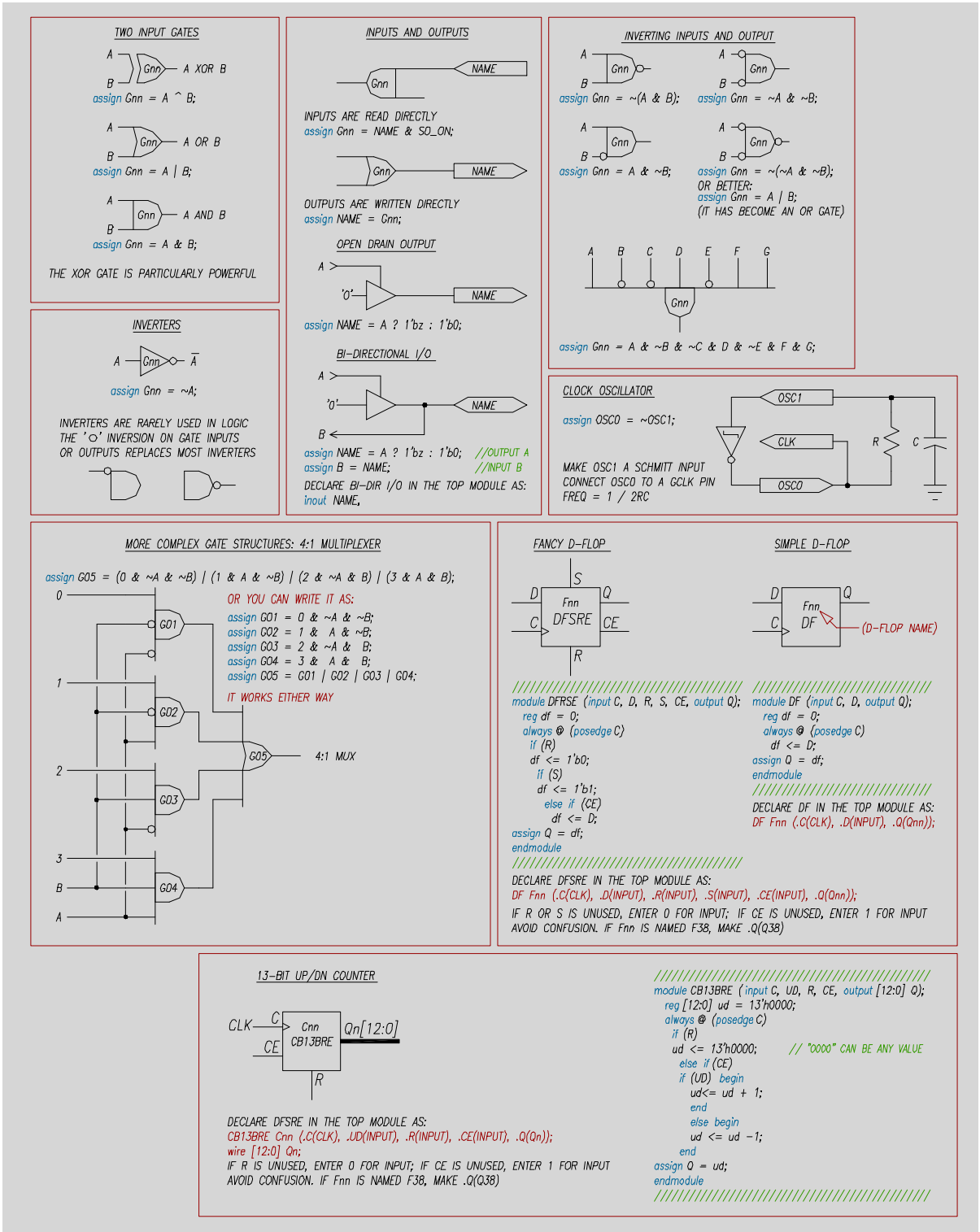
Next, from File, select Save. A window called Bus Delimiter will pop-up. Select "XST Default: <>" and hit "OK". Close Xilinx PACE.

Back in ISE again, double-click "Implement Design" again. When it finishes, the project .jed file is ready for JTAG programming the CPLD.



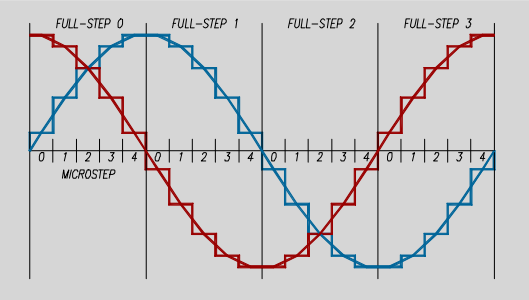
Below is a quick reference for gates, D-Flops and counters. D-Flops and counters are best described behaviorally and written as sub-modules. Once written, the module can be used as many times as needed in the top module. The “Fancy D-Flop” below is very similar to a 7400-series logic 7474 D-Flop. Unlike 7400-series logic, you can create any type of device you need.

For instance, say you need a 13-bit UP/DN counter; you would have to use four 74LS169 packages to form a 16-bit counter and have 3 unused Q outputs. Using a CPLD, you can create a custom-made counter that has only the features you need.



DESIGNING THE 5-MICROSTEP DRIVE:

Step motor winding are driven with sine and cosine currents. Even a full-step sequence can be thought of as a single-bit approximation of a sine and cosine. Only the sign (+/-) is preserved in that approximation that then looks like two square waves having a 90-degree phase relationship and motor's increment of motion is in full steps, usually 200 per revolution. The sine-cosine waveform repeats after 360 degrees during which the motor moves 7.2 degrees. Motor motion will be referred to as "degrees mechanical" while the sine-cosine angle will be called "degrees electrical". In other words, 1.8 degrees mechanical (a full step) is 90 degrees electrical.



If the motor was driven with pure sine-cosine currents, the motor's increment of motion would be infinitesimally small and its motion would have no steps. The blue and red continuous waveforms are sine and cosine. The graph is marked-off in 90 electrical degree and 1.8 mechanical degree intervals. The motor's motion would be continuous over this cycle.

Between the single-bit and the infinite-bit approximation lies microstepping. Microstepping is when the motor currents are a stepwise finite-bit approximations of sine and cosine. In this instance, the 90-degree electrical interval is divided into 5 equal length segments spaced 18 degrees apart and marked "MICROSTEP" 0, 1, 2, 3 and 4. Sine and cosine values are calculated for 9 degrees, 27 degrees, 45 degrees, 63 degrees and 81 degrees, then rescaled to a 0 to 1 range by dividing the results by sine 81 degrees. Analysis shows a 5-bit resolution is sufficient, so these scaled values are multiplied by 2^5 and the results are converted to hexadecimal notation.

Microstep 0 is sine 09 degrees or 0.156, scaled to 0.158, multiplies to 05.06 or 0x05
Microstep 1 is sine 27 degrees or 0.454, scaled to 0.460, multiplies to 14.72 or 0x0F
Microstep 2 is sine 45 degrees or 0.707, scaled to 0.716, multiplies to 22.91 or 0x17
Microstep 3 is sine 63 degrees or 0.891, scaled to 0.902, multiplies to 28.86 or 0x1D
Microstep 4 is sine 81 degrees or 0.988, scaled to 1.000, multiplies to 32.00 or 0x20

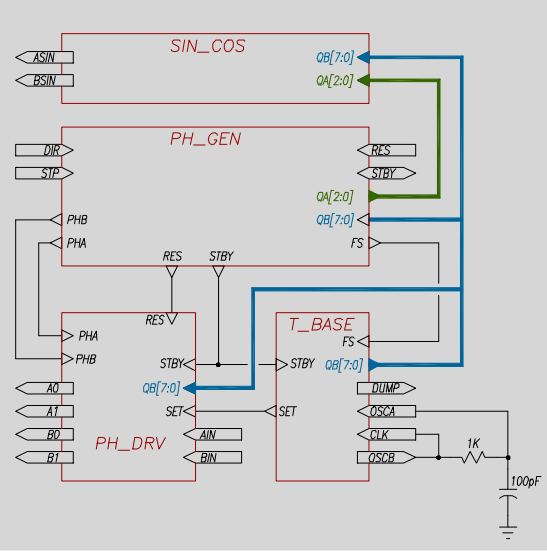
The drive logic can be separated into four function blocks:

SIN_COS takes the microstep counter bus QA[2:0], and the time-base bus QB[7:0] to generate the sine and cosine weighed values. These are converted to a pair of PWM signals that are output on I/O pins named ASIN and BSIN.

PH_GEN inputs I/O pins STP and DIR to its step counters. These counters generate internal signals PHA, PHB and FS. It also generates the microstep counter bus QA[2:0]. I/O input RES resets the counters while I/O pin STBY is used to generate a 1-second standby timer.

PH_DRV generates synchronous PMW non-recirculating mode switching while the motor is moving and changes to recirculating mode when the motor is stopped. It outputs the bridge control signals on I/O pins A0, A1, B0 and B1. Its I/O pin inputs are AIN and BIN.

T_BASE is the circuit's time-base generator. I/O pins OSCA and OSB form a 5MHz RC oscillator whose output goes to I/O input CLK. The clock is divided by 256 to generate the 20kHz drive switching frequency. This counter is the QB[7:0] bus. I/O pin DUMP generates a 20kHz ramp function need for feed-forward compensation.

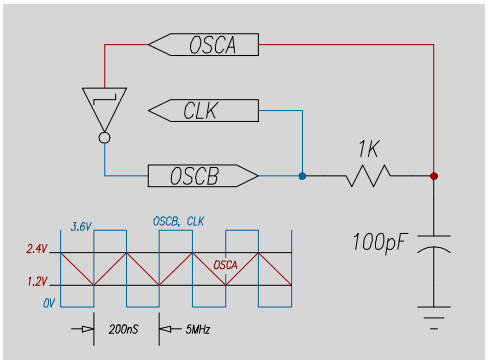


T_BASE (TIME BASE):

CPLD designs are assumed to be fully synchronous. The CPLD has a GCLK (global clock) I/O input pin for this reason. Fully synchronous means every clocked device, D-Flops and counters, have all their CLK inputs tied together and they go to a single clock input. Nothing changes in any register inside the CPLD unless its on a CLK edge.

The 5 microstep drive design is therefore fully synchronous. Every synchronous circuit requires an oscillator (a circuit that autonomously generates a '0', '1', '0', '1',... logic sequence at a continuous and fixed rate). Usually oscillators use a crystal to set the frequency of this sequence and they are required if the frequency has to be exact. In this design the frequency can be inexact (+/-5% instead of +/-0.0005%). This allows for a much less expensive RC (resistor, capacitor) oscillator instead of a crystal-controlled one.

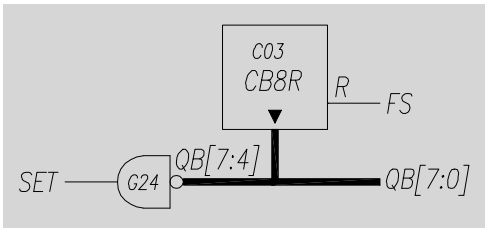
The CoolRunnerII series CPLD I/O is perfect for creating such an oscillator. All it takes is making an inverter in Verilog having an I/O pin named OSCA and an I/O pin named OSCB. It must be described in PACE as OSCA having a SCHMITT, FLOAT input and a LVCMOS33 standard interface on OSCA and OSCB.



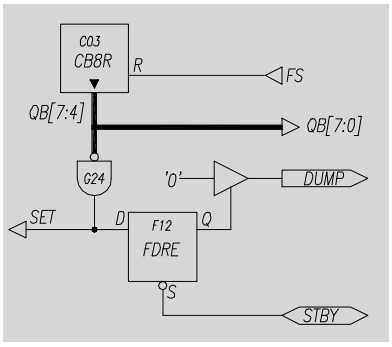
OSCA will look like the red trace in the graph. OSCB will look like the blue trace. OSCB connects to the CPLD global CLK I/O pin. The oscillator frequency will be 1/2RC

The second picture shows the main time-base counter C03. It is an 8-bit counter with a reset input R. The counter runs continuously and is reset by an internal signal FS generated in the PH_GEN circuit to synchronize the counter to the full step rate. The FS pulse is 1 CLK cycle wide.

G24 generates an internal signal SET that initiates the switching cycle for the motor windings in the PH_DRV circuit diagram. G24 goes true (logic 1) for 16 CLK cycles (3.2 uS) and this signal repeats every time counter C03 rolls over (every 51.2 uS or about 20 kHz). It is important to start a new switching cycle on every full-step location, which is why FS resets C03.



The complete T_BASE circuit is shown on the right. Counter C03 outputs its QB[7:0] bus to the all the other function blocks.



Q12 operates the output enable on the output driver going to DUMP. This causes DUMP to go low for 3.2uS and be an open circuit the rest of the time.

Anytime a counter a counter is used, it's output bus must be named and declared as a "wire". Counter C03 is an 8-bit counter whose 8-bit output bus is named QB[7:0]. It must be declared as:

```
wire [7:0] QB;
```

A module named DF (D-Flop) is used having an identifier named F12. This same module may be used in many other locations so each use must have a unique identifier. The module DF is written behaviorally as:

```
module DF (input D, C, CE, R, S, output Q);
reg df = 0;
always @(posedge C)
if (R)
df <= 1'h0;
else if (S)
df <= 1'b1;
else if (CE)
df <= D;
assign Q = df;
endmodule
```

DF is written as a complete D-Flop having a clock input C, a D input, a clock enable CE, a reset input R and a set input S. Its only output is Q.

The module must start with the reserved word `module` followed by the module name (I used DF). This most be followed by `(input D, C, CE, R, S, output Q);` where every input and output is named (my choices were D, C, CE, R, S and Q).

`reg df = 0;` A register name must be picked and you cannot use the output name (Q here). I used “df”.
`always @(posedge C)` This means everything that follows happens on the rising edge of the clock.
`if (R)` This means **if** the reset input is logical ‘1’,
`df <= 1'h0;` make the register ‘df’ a logical ‘0’. In effect, reset the register on the next clock if R is ‘1’.
`else if (S)` **Otherwise if** the set input is a logical ‘1’,
`df <= 1'b1;` make the register ‘df’ a logical ‘1’. In effect, set the register on the next clock if S is ‘1’.
`else if (CE)` **Otherwise if** clock enable is a logical ‘1’,
`df <= D;` make register ‘df’ become the same as what is on the D input.
`assign Q = df;` Make ‘Q’ have the same value as register ‘df’.
`endmodule` This reserved word (everything in blue is reserved) says you are done describing module DF

Similarly, a module named CB8R (Counter, Binary, 8-bit, Reset input) with the identifier C03 is used. CB8R is behaviorally written as:

```
module CB8R (input C, R, output [7:0] Q);
 reg [7:0] count = 0;
 always @(posedge C)
if (R)
count <= 0;
else
count <= count + 1;
assign Q = count;
endmodule
```

This pretty much follows module DF: Name the module, name the inputs and outputs. Since the output is an 8-bit bus, it must be named as output [7:0] Q. Then name the register (I chose ‘count’ because of a lack of imagination) and describe what you want to have happen.

Always on the rising edge of the clock, **if** reset is ‘1’, make the ‘count’ register equal to zero. **Otherwise** increment the ‘count’ register (count becomes count plus one). Make output ‘Q’ have the same value as ‘count’ and you are done. Simple enough.

After the sub-modules DF and CB8R are written, they need to be called-up in the top module. Here’s DF being called in the top module:

```
DF     F12 (.C(CLK), .D(G24), .R(0), .CE(1), .S(~STBY), .Q(Q12));
```

First, the module name DF and its identifier F12 are entered. Next, `.C(CLK)`, means “connect system CLK to DF module input .C, `.D(G24)`, means connect AND gate G24 to the D input, `.R(0)`, means reset isn’t used, `.S(~STBY)`, means connect the set input to an inverted (~) signal called STBY, `.CE(1)` means we want this D-Flop to accept every clock cycle and `.Q(Q12)` means we want to name this DF output “Q12”. You don’t have to but it’s smart to name Q the same number as the DF identifier.

The same goes for the CB8R sub-module.

```
CB8R   C03 (.C(CLK), .R(G14), .Q(QB));
```

Name the module, its identifier (C03 in this instance) and describe how the module inputs and outputs connect. In this case C connects to CLK, R connects to a gate named G14 and its output is named ‘QB’. Don’t forget to declare it as a wire `wire [7:0] QB;` in the main module.

The CoolRunner II CPLDs don’t have open-drain outputs per se. They do have 2-state and 3-state outputs however. A tri-state output buffer has an input, an output and an output enable. When the output enable is low, the buffer output ‘floats’ (has no connection). An open-drain output is effected by using a tri-state buffer with its input tied to ground (0). When enabled, the output is ‘0’, when disabled it floats. This exactly emulates an open-drain output. The DUMP output line serves this purpose:

```
assign DUMP = ~Q12 ? 1'bz : 1'b0;
```

It just as easily been set to have the buffer input be a logical ‘1’ which would have made the output act as an open-source output to Vdd (a single-pole, single-throw switch to +3.3V). The code for that is:

```
assign DUMP = ~Q12 ? 1'bz : 1'b1;
```

This is kind of cool if you need to switch to logic ‘1’ only.

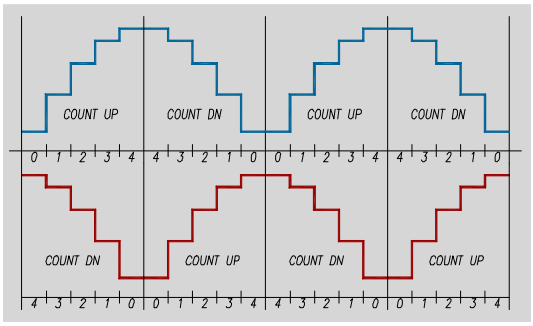
This tutorial design uses a D-Flops (DF module), an 8-bit main time-base counter (CB8R module), a 2-bit up-down counter (CB2BRE) and a 3-bit up-down counter named (CB3BRE). The project main module skeleton and the sub-module code for the above modules is on the following page: It includes the first function (T_BASE) in the main module.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Module Name:    DRV_5uS
//
//////////////////////////////////////////////////////////////////////////////////
module DRV_5uSTP

(input CLK, AIN, BIN, DIR, STP, OSCA, RES,
 inout STBY,
output ASIN, BSIN, DUMP, A0, A1, B0, B1, OSCB);

wire [2:0] QA;
wire [7:0] QB;
wire [1:0] QC;
wire [2:0] QD;

//T_BASE-----------------------------------------------------------------------
CB8R   C03 (.C(CLK), .R(G14), .Q(QB));
DF     F12 (.C(CLK), .D(G24), .R(0), .CE(1), .S(~STBY), .Q(Q12));
assign DUMP = ~Q12 ? 1'bz : 1'b0;
assign G24  = QB[7] & QB[6] & QB[5] & QB[4];
assign OSCB = ~OSCA;

//the rest of the main module code here:

endmodule
//////////////////////////////////////////////////////////////////////////////////-
module DF (input D, C, CE, R, S, output Q);
reg df = 0;
always @(posedge C)
if (R)
df <= 1'h0;
else if (S)
df <= 1'b1;
else if (CE)
df <= D;
assign Q = df;
endmodule
//////////////////////////////////////////////////////////////////////////////////-
module CB8R (input C, R, output [7:0] Q);
reg [7:0] count = 0;
always @(posedge C)
if (R)
count <= 0;
else
count <= count + 1;
assign Q = count;
endmodule
//////////////////////////////////////////////////////////////////////////////////-
module CB2BRE (input UD, R, CE, C, output [1:0] Q);
reg [1:0] u_d = 0;
always @(posedge C)
if (R)
u_d <= 0;
else if (CE)
if (UD) begin
u_d <= u_d + 1;
end
else begin
u_d <= u_d - 1;
end
assign Q = u_d;
endmodule
//////////////////////////////////////////////////////////////////////////////////-
module CB3RE (input C, R, CE, output [7:0] Q);
reg [2:0] count = 0;
always @(posedge C)
if (R)
count <= 0;
else if (CE)
count <= count + 1;
assign Q = count;
endmodule
//////////////////////////////////////////////////////////////////////////////////-
```

PH_GEN (PHASE GENERATORS):

5 microsteps per full step requires a count of 20 step pulses for a 360 electrical degree cycle. This count requires a 5-bit counter. Because the 5 microstep count must be repeated 4 times for a full cycle, it makes sense to split this 5-bit counter two separate counters; a 3-bit counter and a 2-bit counter. The 3-bit counter must be made to have a modulus of 5, meaning it's minimum down count is 0 and its maximum up count is 4.

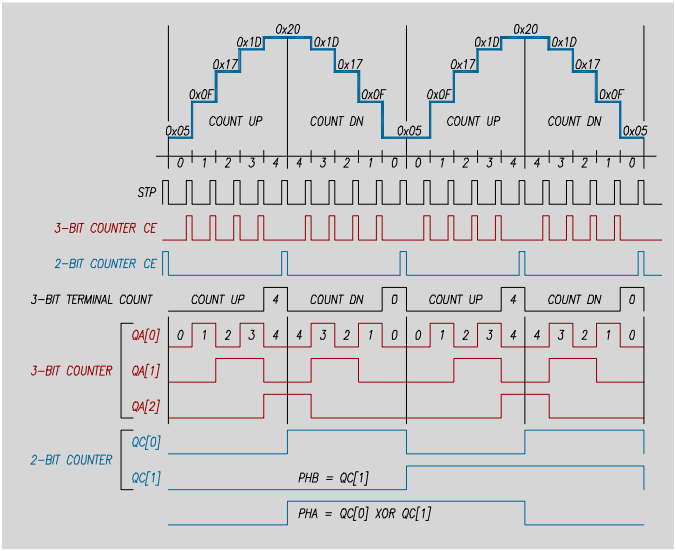
Normally the sine-cosine lookup table in a microstep drive only contains absolute values for sine and cosine. This means the data is stripped of its sign information. The sign is restored in PH_DRV.



Note that the absolute sine and cosine values are only unique in the first 90 electrical degrees. After that they mirror or repeat. This means the lookup table needs at most 5 entries.

Inspection shows the third quadrant is a repeat of the first while the second and forth quadrants are mirror images of the first quadrant.

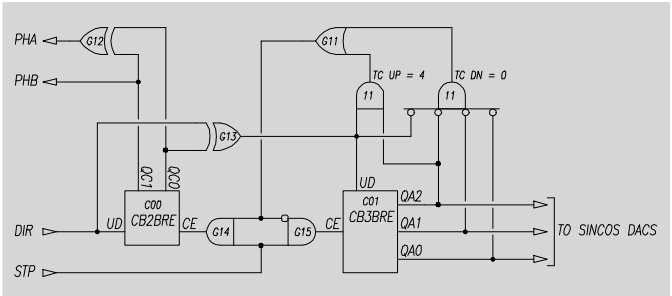
In the first quadrant, the lookup table entries are retrieved in an ascending order by the 3-bit microstep counter counting up from '0' to '4'. All that's needed to mirror the entries in the second quadrant is to have the counter now count down' from '4' to '0. Repeat for third and fourth quadrants.



The 3-bit UD counter is shown in red and the 2-bit UD counter is shown in blue. The STP signal is sent to red counter CE (clock enable) so long as QA[2:0] is not '4' while counting up and not '0' while counting down. This is the TERMINAL COUNT signal. If TC is true, the STP signal is switched to the blue counter's CE instead.

The blue QC[0] determines the count direction for the red counter. If QC[0] is low, red counts up, if QC[0] is high, red counts down.

QC[1] is the sign for the sine lookup table value. If QC[1] is low, the sign is '+', otherwise it's '-'. QC[1] XOR QC[0] is the sign for the cosine lookup table value. Works the same way too.

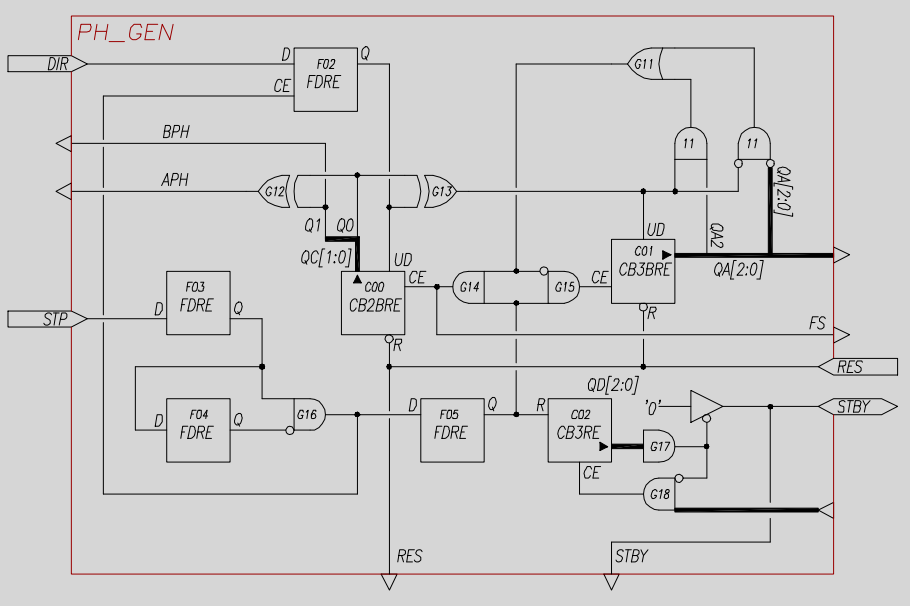


Here is the first cut at a counter schematic. The 4 inverting input AND gate G11 goes true when QA[2:0] is zero and UD (up/down) is zero. The 2 input AND gate G11 goes true when QA[2:0] > 3 and UD is true. The 2-input OR gate G11 goes true on the terminal counts.

G15 goes true when STP is true (1 CLK period) and G11 is low. The 3-bit counter's CE is true then and the count direction is set by its UD input.

G14 goes true when STP and G11 are true. The 2-bit counter's CE is true then and its count direction is set by DIR on its UD input. G13 inverts QC[0] to C01's UD input when DIR is true, otherwise QC[0] is not inverted. G12 generates a quadrature signal of QC[1].

The result is a logic diagram that does what is shown in the middle picture. Being a first cut, other circuits have to be added to it to make it practical.



This is the complete logic diagram for the PH_GEN circuit block. F03 clocks in the I/O pin STP to synchronize it to the system CLK. F04 and G16 generates a 1 CLK wide true pulse on the leading edge of F03. This is the same circuit as the one in Chapter 1. G16 also causes F02 to clock in I/O pin DIR on the trailing edge of the G16 pulse. F05 delays the G16 pulse by one CLK period to synchronize it with the DIR data on Q02.

C00 and C01 are reset while I/O input RES is low. It forms a power-on reset if a resistor to Vdd and a capacitor to Vss is wired to RES. The time duration is RC; a 1MEG resistor and a 1uF capacitor results in a 1-second power-on reset time.

The same RC circuit is replicated on the I/O pin STBY except its function is to detect when more than 1 second has elapsed since the last STP pulse. It works by discharging the 1uF on the STBY pin on every STP pulse. A 1MEG resistor to Vdd will charge this capacitor to the logic '1' threshold in one second if the capacitor is not discharged. The problem is a 1uF capacitor takes 360uS to discharge at 10mA; the current is limited to that value externally so as not to damage the CPLD with a very large discharge current.

It requires 11 registers to provide a 360uS period using a 5MHz CLK so that isn't practical. C02 is reset to zero on every STP pulse. G17 goes true when QD[2:0] reaches 0x7. G18 inverts G17 and stops C02 at its terminal count. G18 ands the time-base bus QD[7:0] to enable C02 on every 256th CLK period. This makes G17 low for 2,048 CLK cycles (400uS) on every STP pulse. G17 forces I/O pin STBY low during this time to dump the STBY capacitor. The rest of the time STBY is an input I/O pin.

G14 is also an internal signal FS. It goes true for 1 CLK cycle at every full-step (sine 0, 90, 180 and 270) transition. QC[1] and G12 form internal signals PHA and PHB that are quadrature signals used to restore the sine and cosine sign.

VERILOG CODE:

The code for the above logic diagram starts below the `//PH_GEN-----` line. The D-Flops and counters connections are defined first. The modules below the top module describe the D-Flops and counters used in the top module. The gate, counter and bus names are the same as in the logic diagram. Don't bother implementing this design as written so far; it will crash. This function block has no I/O outputs; it will be stitched to the other 3 function blocks via internal signals when the project is finished.

Let's add the PH_GEN code to the main module:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Module Name:    DRV_5uS
//
//////////////////////////////////////////////////////////////////////////////////
module DRV_5uSTP

(input CLK, AIN, BIN, DIR, STP, OSCA, RES,
 inout STBY,
output ASIN, BSIN, DUMP, A0, A1, B0, B1, OSCB);

wire [2:0] QA;
wire [7:0] QB;
wire [1:0] QC;
wire [2:0] QD;

//T_BASE-------------------------------------------------------------
CB8R   C03 (.C(CLK), .R(G14), .Q(QB));
DF     F12 (.C(CLK), .D(G24), .R(0), .CE(1), .S(~STBY), .Q(Q12));
assign DUMP = ~Q12 ? 1'bz : 1'b0;
assign G24  = QB[7] & QB[6] & QB[5] & QB[4];
assign OSCB = ~OSCA;
//PH_GEN-------------------------------------------------------------------------
DF     F02 (.C(CLK), .D(DIR), .R(0), .S(0), .CE(G16), .Q(Q02));
DF     F03 (.C(CLK), .D(STP), .R(0), .S(0), .CE(1), .Q(Q03));
DF     F04 (.C(CLK), .D(Q03), .R(0), .S(0), .CE(1), .Q(Q04));
DF     F05 (.C(CLK), .D(G16), .R(0), .S(0), .CE(1), .Q(Q05));
CB2BRE C00 (.C(CLK), .UD(Q02), .R(~RES), .CE(G14), .Q(QC));
CB3BRE C01 (.C(CLK), .UD(G13), .R(~RES), .CE(G15), .Q(QA));
CB3RE  C02 (.C(CLK), .R(Q05), .CE(G18), .Q(QD));
assign G12 = QC[1] ^ QC[0];
assign G13 = QC[0] ^ Q02;
assign G11 = ( G13 &  QA[2]) | (~G13 & ~QA[2] & ~QA[1] & ~QA[0]);
assign G14 =  G11 & Q05;
assign G15 = ~G11 & Q05;
assign G16 =  Q03 & ~Q04;
assign G17 =  QD[2] & QD[1] & QD[0];
assign STBY = ~G17 ? 1'bz : 1'b0;
assign G18 =  ~G17 & QB[7] & QB[6] & QB[5] & QB[4] & QB[3] &QB[2] & QB[1] & QB[0];

//the rest of the main module code here:

endmodule
```

SIN_COS (SINE – COSINE GENERATORS):

The outputs of the 3-bit microstep counter C01 would normally point to a sine-cosine lookup table. The table contents would then be ouput from the CPLD to a pair of multiplying DACs (digital to analog converters) to generate the reference voltages for the two current-loop comparators. The analog multiplier inputs to the DACs would set the amplitude of the DAC output voltages and function as a current set for the drive.

It has already been established the DACs need a 5-bit resolution to get the necessary precision. This requires 10 I/O pins for a pair of parallel loading DACs or a complex interface for serial DACs. It is an unimaginative and expensive solution. Instead, let's see if we can use some of the advantages of a true CMOS CPLD and do the following:

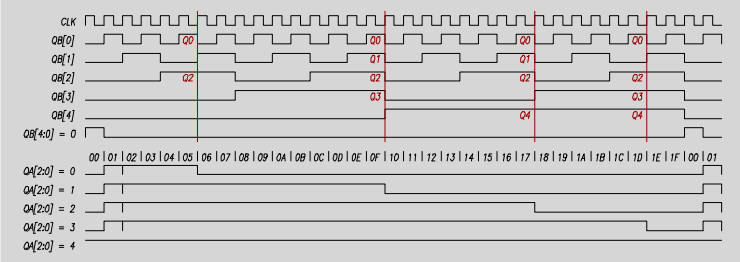
1) Dispense with using DACs altogether. They are expensive when compared to "for free".
2) Combine the lookup table and DAC functions into one.
3) Use only 2 I/O pins.

Seems like tall order but it's not. Let's look at what's need and how these properties can be exploited. The sine and cosine functions are monotonic. Over a 0 to 90 degree span, each lookup entry for sine is larger than the previous one and each cosine value is smaller than the previous one. The bandwidth required for the sine and cosine values is low; 2kHz is more than plenty. Finally, CPLD outputs are nearly perfect analog switches to GND; less than 20 Ohms when 'on', many Meg-Ohms when 'off'. This suggests a PWM type digital to analog conversion.

What is needed:

QA[2:0] = 0, PWM duty cycle = 05 / 32 or 0x05/0x20.
QA[2:0] = 0, PWM duty cycle = 15 / 32 or 0x0F/0x20.
QA[2:0] = 0, PWM duty cycle = 23 / 32 or 0x17/0x20.
QA[2:0] = 0, PWM duty cycle = 29 / 32 or 0x1D/0x20.
QA[2:0] = 0, PWM duty cycle = 32 / 32 or 0x20/0x20.

5-bit DAC resolution in a PWM based DAC requires 32 time cells (32 = 2^5). We start by describing what we would like to see from a logic diagram.



The time-base QD[7:0] bus divides the 5MHz CLK down to the 20kHz switching frequency of the drive. The lower 5 bits of this bus has a repetition frequency of 156kHz. This will be the PWM frequency. A single-pole low-pass passive filter is adequate to pass the needed 2kHz while attenuating the 156kHz PWM carrier frequency. Here's how it's done:

1) Reset a D-Flop when time-base QB[4:0] = 0
2) When QA[2:0] = 0 AND QB[4:0] = 0x05, set the D-Flop. Once set, it cannot be cleared until (1) again.

OR:
1) Reset a D-Flop when time-base QB[4:0] =0
2) When QA[2:0] = 1 AND QB[4:0] = 0x1F, set the D-Flop. Once set, it cannot be cleared until (1) again.

OR
1) Reset a D-Flop when time-base QB[4:0] = 0
2) When QA[2:0] = 2 AND QB[4:0] = 0x17, set the D-Flop. Once set, it cannot be cleared until (1) again.

OR:
1) Reset a D-Flop when time-base QB[4:0] = 0
2) When QA[2:0] = 3 AND QB[4:0] = 0x1D, set the D-Flop. Once set, it cannot be cleared until (1) again.

OR:
1) Reset a D-Flop when time-base QB[4:0] = 0
2) When QA[2:0] = 4 AND QB[4:0] = 0x20, set the D-Flop. Once set, it cannot be cleared until (1) again.

The above description suggests using an AND-OR gate array.

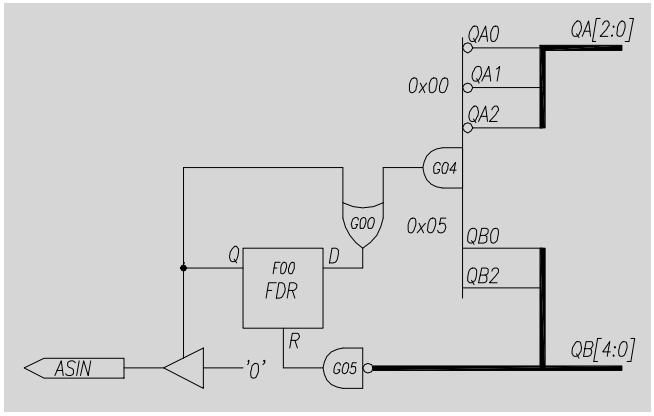
The sine of 9 degrees approximation is 5/32. The PWM output is '1' for the first 5 CLK periods of the 32 period cycle and '0' for the remaining 27 CLK periods. Once '0', it cannot go high again until QB[4:0] = 0 is true again. 0x05 is decoded

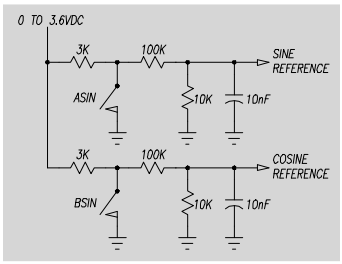
from the QB bus; QB[4] = 0, QB[3] = 0, QB[2] = 1, QB[1] = 0 and QB[0] =1. A simplification is only the '1' terms need to be decoded, QB[2] AND QB[0]. The first time this combination appears is in time-cell 5; it appears 3 more times, 0x0D, 0x15 and 0x1D but that doesn't matter because 0x05 is the first occurrence.

In this figure G05 decodes QB[4:0] = 0 and resets FD00 to start the PWM cycle. G04 decodes mocrostep 0 (QA[2:0] = 0) and when QB[2] and QB[0] are both true (PWM time-cell 0x05).

G00 ORs Q00 with G04 to D00. The first time G04 goes true, it clocks Q00 true also. Thereafter Q00 stays at a '1' because G00 keeps D00 true. Q00 cannot change until the next time G05 goes true and resets Q00.

Q00 is output on I/O pin ASIN as open-drain output. While Q00 is '0' the tri-state buffer to ASIN is disabled and ASIN is an open circuit. When Q00 is true, the tri-state buffer is enabled and it puts '0' on the ASIN pin.





This figure shows ASIN and BSIN as analog switches to ground. When ASIN is closed, the 3K to 100K node is very near 0VDC. When ASIN is open, the 0 to 3.6VDC reference voltage (CURRENT SET) is applied to what effectively becomes a 103K resistor.

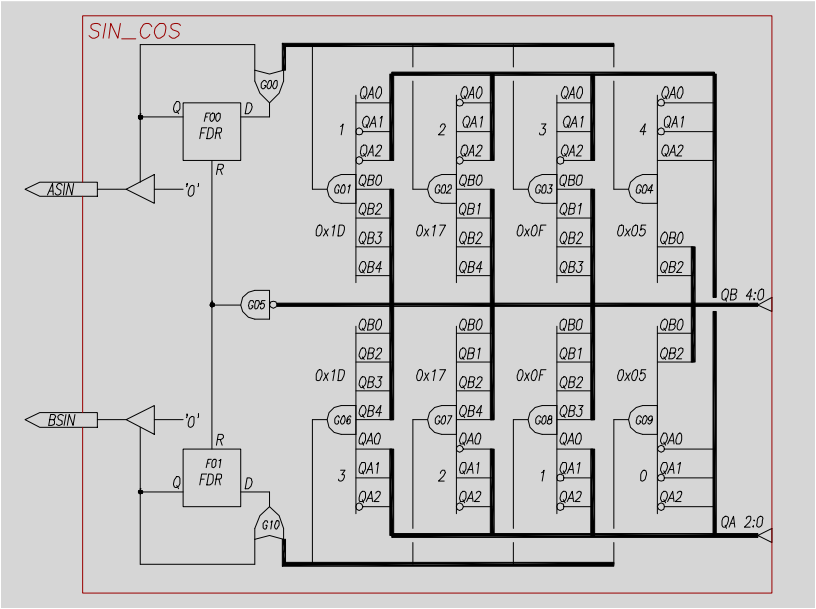
The 10K resistor and paralleled 10nF capacitor low-pass filters the PWM signal on ASIN to its analog value, multiplies it by the reference voltage and attenuates its magnitude to what appears across the current sense resistor in the power section of the drive.

Putting it all together: This logic diagram shows the QA[2:0] bus microstep decoder, the QB[4:0] bus sine-cosine encoder and the PWM modulators for the ASIN and BSIN I/O output pins.

The hexadecimal sine-cosine values and the microstep count are marked next to the AND gates.

The bottom row of gates are the sine encoders. Note microstep '4' is not encoded; its value is 0x20 which means ASIN is an open-circuit for the entire PWM period (100% modulation).

The same is true for BSIN except its microstep '0' doesn't need decoding. This saves two gates in this circuit.



The code for this section is short and sweet. Let's add this to what we have already and add the this code to the main module.

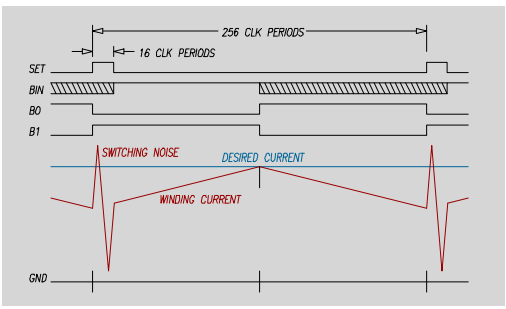
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Module Name:    DRV_5uS
//
//////////////////////////////////////////////////////////////////////////////////
module DRV_5uSTP

(input CLK, AIN, BIN, DIR, STP, OSCA, RES,
 inout STBY,
output ASIN, BSIN, DUMP, A0, A1, B0, B1, OSCB);

wire [2:0] QA;
wire [7:0] QB;
wire [1:0] QC;
wire [2:0] QD;

//T_BASE-------------------------------------------------------------
CB8R   C03 (.C(CLK), .R(G14), .Q(QB));
DF     F12 (.C(CLK), .D(G24), .R(0), .CE(1), .S(~STBY), .Q(Q12));
assign DUMP = ~Q12 ? 1'bz : 1'b0;
assign G24  = QB[7] & QB[6] & QB[5] & QB[4];
assign OSCB = ~OSCA;

//PH_GEN-------------------------------------------------------------------------
DF     F02 (.C(CLK), .D(DIR), .R(0), .S(0), .CE(G16), .Q(Q02));
DF     F03 (.C(CLK), .D(STP), .R(0), .S(0), .CE(1), .Q(Q03));
DF     F04 (.C(CLK), .D(Q03), .R(0), .S(0), .CE(1), .Q(Q04));
DF     F05 (.C(CLK), .D(G16), .R(0), .S(0), .CE(1), .Q(Q05));
CB2BRE C00 (.C(CLK), .UD(Q02), .R(~RES), .CE(G14), .Q(QC));
CB3BRE C01 (.C(CLK), .UD(G13), .R(~RES), .CE(G15), .Q(QA));
CB3RE  C02 (.C(CLK), .R(Q05), .CE(G18), .Q(QD));
assign G12 = QC[1] ^ QC[0];
assign G13 = QC[0] ^ Q02;
assign G11 = ( G13 &  QA[2]) | (~G13 & ~QA[2] & ~QA[1] & ~QA[0]);
assign G14 =  G11 & Q05;
assign G15 = ~G11 & Q05;
assign G16 =  Q03 & ~Q04;
assign G17 =  QD[2] & QD[1] & QD[0];
assign STBY = ~G17 ? 1'bz : 1'b0;
assign G18 =  ~G17 & QB[7] & QB[6] & QB[5] & QB[4] & QB[3] &QB[2] & QB[1] & QB[0];

//SIN_COS------------------------------------------------------------------------
DF    F00 (.C(CLK), .D(G00), .R(G05), .S(0), .CE(1), .Q(Q00));
DF    F01 (.C(CLK), .D(G10), .R(G05), .S(0), .CE(1), .Q(Q01));
assign G01 =  QA[2] & ~QA[1] & ~QA[0] & QB[2] & QB[0];
assign G02 = ~QA[2] &  QA[1] &  QA[0] & QB[3] & QB[2] & QB[1] & QB[0];
assign G03 = ~QA[2] &  QA[1] & ~QA[0] & QB[4] & QB[2] & QB[1] & QB[0];
assign G04 = ~QA[2] & ~QA[1] &  QA[0] & QB[4] & QB[3] & QB[2] & QB[0];
assign G06 = ~QA[2] & ~QA[1] & ~QA[0] & QB[2] & QB[0];
assign G07 = ~QA[2] & ~QA[1] &  QA[0] & QB[3] & QB[2] & QB[1] & QB[0];
assign G08 = ~QA[2] &  QA[1] & ~QA[0] & QB[4] & QB[2] & QB[1] & QB[0];
assign G09 = ~QA[2] &  QA[1] &  QA[0] & QB[4] & QB[3] & QB[2] & QB[0];
assign G00 = Q00 | G01 | G02 | G03 | G04;
assign G10 = Q01 | G06 | G07 | G08 | G09;
assign G05 = ~QB[4] & ~QB[3] & ~QB[2] & ~QB[1] & ~QB[0];
assign BSIN = ~Q01 ? 1'bz : 1'b0;
assign ASIN = ~Q00 ? 1'bz : 1'b0;

//the rest of the main module code here:

endmodule
```

PH_DRV

The design uses a non-recirculating switching mode while the motor is turning. The motor current control loop is a fixed-frequency design to avoid the drawbacks of a chopper design. A 20kHz pulse starts the switching cycle by causing the bridge to increase current in the winding. The current level comparator is ignored for the first few microseconds to allow switching transient noise to die away. This is called the blanking period.

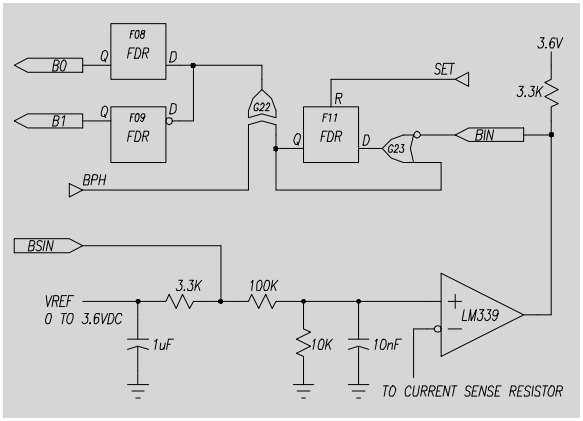
After this time, the comparator output is monitored. While motor winding current is less the required level, the comparator output is '1'; when the current reaches or exceeds the required level, the comparator output is '0'. When this occurs, the bridge is caused to decrease the winding current by reversing the polarity of the voltage across the winding for the remainder of the switching period. The process repeats with the next 20kHz pulse.

This picture shows a SET pulse generated by the time-base. It is 16 CLK cycles wide and it has a 256 CLK cycle period. A 5MHz results in SET being 3.3uS wide and having a 51.2 uS period (about 20kHz).



The width of the SET pulse is the blanking time. The red curve shows the motor current having switching noise as the current slope goes from decreasing to increasing current. The blue line is

the desired current. The comparator output to BSIN goes low when the desired current is reached, causing the bridge control outputs B0 and B1 to change state. BIN is then ignored for the rest of the switching period.



The SET pulse resets F11. F11 ignores gate G23 for the duration of the SET pulse width, which is the blanking period.

Afterward, when the LM339 comparator output goes low, Q11 feeds back to D11 through G23 and Q11 stays a '1' until the next SET pulse.

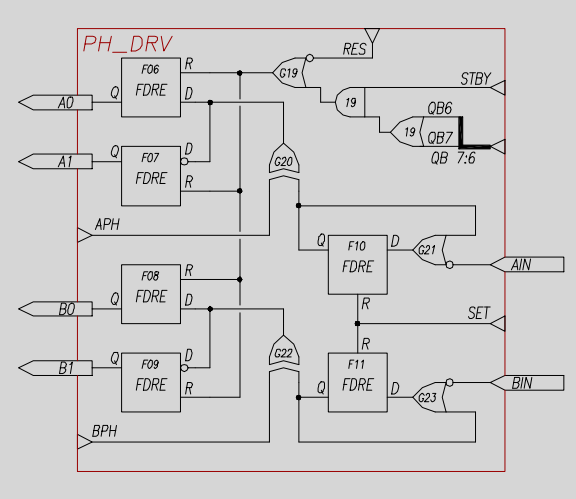
BPH is B-phase winding current direction command from PH_GEN. G22 inverts Q11 if BPH is a '1'.

F08 and F09 drive the I/O pins B0 and B1 that go to the bridge driver. D09 is an inverting input so Q09 is inverted while Q08 is not.

The comparator's non-inverting input connects to the low-pass filter driven by BSIN and its inverting input goes to the B-winding bridge current sense resistor.

This is the complete logic diagram for PH_DRV. RES resets F06, F07, F08 and F09 through G19 forcing all bridge outputs low.

STBY changes the drive to recirculating mode switching when true. QB[7] and QB[6] are ORed, then ANDed with STBY in G19. This resets the I/O pins A0, A1, B0 and B1 and shorts the motor windings for the last 75% of the period. The motor current control loop operates normally during the first 25% of the switching period. Current decays slowly in the windings while they are shorted and require less time to restore the current to the required level. That leads to reduced ripple current, eddy current losses and motor heating while the motor is stopped.



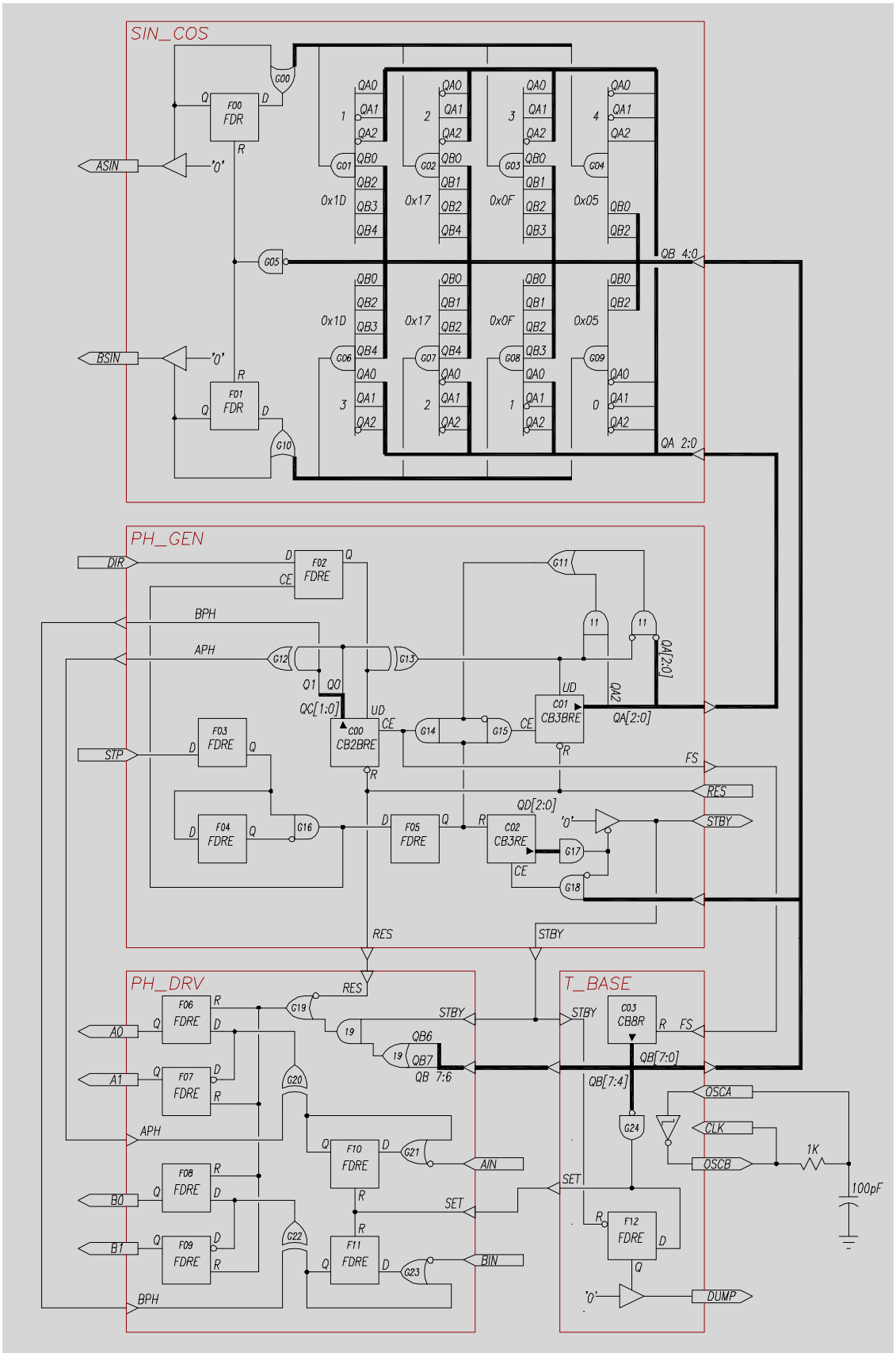
Recirculating mode isn't suitable for a moving motor because of the slow current decay rate. It introduces a distortion call "current tailing" where sine 0 to 90 degrees reference is tracked accurately but sine 90 to 180 tracking is limited by the much lower decay slope.

Let's add the code to this last for this section to the top module and the sub-modules: The design is finished!

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Module Name:    DRV_5uS
//
//////////////////////////////////////////////////////////////////////////////////
module DRV_5uSTP

(input CLK, AIN, BIN, DIR, STP, OSCA, RES,
 inout STBY,
output ASIN, BSIN, DUMP, A0, A1, B0, B1, OSCB);

wire [2:0] QA;
wire [7:0] QB;
wire [1:0] QC;
wire [2:0] QD;

//T_BASE--------------------------------------------------------
CB8R   C03 (.C(CLK), .R(G14), .Q(QB));
DF     F12 (.C(CLK), .D(G24), .R(0), .CE(1), .S(~STBY), .Q(Q12));
assign DUMP = ~Q12 ? 1'bz : 1'b0;
assign G24  = QB[7] & QB[6] & QB[5] & QB[4];
assign OSCB = ~OSCA;

//PH_GEN--------------------------------------------------------------------------
DF     F02 (.C(CLK), .D(DIR), .R(0), .S(0), .CE(G16), .Q(Q02));
DF     F03 (.C(CLK), .D(STP), .R(0), .S(0), .CE(1), .Q(Q03));
DF     F04 (.C(CLK), .D(Q03), .R(0), .S(0), .CE(1), .Q(Q04));
DF     F05 (.C(CLK), .D(G16), .R(0), .S(0), .CE(1), .Q(Q05));
CB2BRE C00 (.C(CLK), .UD(Q02), .R(~RES), .CE(G14), .Q(QC));
CB3BRE C01 (.C(CLK), .UD(G13), .R(~RES), .CE(G15), .Q(QA));
```

```
CB3RE  C02 (.C(CLK), .R(Q05), .CE(G18), .Q(QD));
assign G12 = QC[1] ^ QC[0];
assign G13 = QC[0] ^ Q02;
assign G11 = ( G13 &  QA[2]) | (~G13 & ~QA[2] & ~QA[1] & ~QA[0]);
assign G14 =  G11 & Q05;
assign G15 = ~G11 & Q05;
assign G16 =  Q03 & ~Q04;
assign G17 =  QD[2] & QD[1] & QD[0];
assign STBY = ~G17 ? 1'bz : 1'b0;
assign G18 =  ~G17 & QB[7] & QB[6] & QB[5] & QB[4] & QB[3] &QB[2] & QB[1] & QB[0];

//SIN_COS-----------------------------------------------------------------------
DF    F00 (.C(CLK), .D(G00), .R(G05), .S(0), .CE(1), .Q(Q00));
DF    F01 (.C(CLK), .D(G10), .R(G05), .S(0), .CE(1), .Q(Q01));
assign G01 =  QA[2] & ~QA[1] & ~QA[0] & QB[2] & QB[0];
assign G02 = ~QA[2] &  QA[1] &  QA[0] & QB[3] & QB[2] & QB[1] & QB[0];
assign G03 = ~QA[2] &  QA[1] & ~QA[0] & QB[4] & QB[2] & QB[1] & QB[0];
assign G04 = ~QA[2] & ~QA[1] &  QA[0] & QB[4] & QB[3] & QB[2] & QB[0];
assign G06 = ~QA[2] & ~QA[1] & ~QA[0] & QB[2] & QB[0];
assign G07 = ~QA[2] & ~QA[1] &  QA[0] & QB[3] & QB[2] & QB[1] & QB[0];
assign G08 = ~QA[2] &  QA[1] & ~QA[0] & QB[4] & QB[2] & QB[1] & QB[0];
assign G09 = ~QA[2] &  QA[1] &  QA[0] & QB[4] & QB[3] & QB[2] & QB[0];
assign G00 = Q00 | G01 | G02 | G03 | G04;
assign G10 = Q01 | G06 | G07 | G08 | G09;
assign G05 = ~QB[4] & ~QB[3] & ~QB[2] & ~QB[1] & ~QB[0];
assign BSIN = ~Q01 ? 1'bz : 1'b0;
assign ASIN = ~Q00 ? 1'bz : 1'b0;

//PH_DRV------------------------------------
DF     F06 (.C(CLK), .D(G20),  .R(G19), .S(0), .CE(1), .Q(A0));
DF     F07 (.C(CLK), .D(~G20), .R(G19), .S(0), .CE(1), .Q(A1));
DF     F08 (.C(CLK), .D(G22),  .R(G19), .S(0), .CE(1), .Q(B0));
DF     F09 (.C(CLK), .D(~G22), .R(G19), .S(0), .CE(1), .Q(B1));
DF     F10 (.C(CLK), .D(G21),  .R(G24), .S(0), .CE(1), .Q(Q10));
DF     F11 (.C(CLK), .D(G23),  .R(G24), .S(0), .CE(1), .Q(Q11));
assign G19 = ~RES | (STBY & (QB[7] | QB[6]));
assign G22 = Q11 ^ QD[1];
assign G23 = Q11 | ~BIN;
assign G20 = Q10 ^ G12;
assign G21 = Q10 | ~AIN;

endmodule
////////////////////////////////////////////////////////////////////////////////
module DF (input D, C, CE, R, S, output Q);
reg df = 0;
always @(posedge C)
if (R)
df <= 1'h0;
else if (S)
df <= 1'b1;
else if (CE)
df <= D;
assign Q = df;
endmodule

////////////////////////////////////////////////////////////////////////////////
module CB8R (input C, R, output [7:0] Q);
reg [7:0] count = 0;
always @(posedge C)
if (R)
count <= 0;
else
count <= count + 1;
assign Q = count;
endmodule
////////////////////////////////////////////////////////////////////////////////
module CB2BRE (input UD, R, CE, C, output [1:0] Q);
reg [1:0] u_d = 0;
always @(posedge C)
if (R)
u_d <= 0;
else if (CE)
if (UD) begin
u_d <= u_d + 1;
end
else begin
u_d <= u_d - 1;
end
assign Q = u_d;
endmodule
////////////////////////////////////////////////////////////////////////////////
module CB3RE (input C, R, CE, output [7:0] Q);
reg [2:0] count = 0;
always @(posedge C)
if (R)
count <= 0;
else if (CE)
count <= count + 1;
assign Q = count;
endmodule
////////////////////////////////////////////////////////////////////////////////
```

THE COMPLETE CIRCUIT:



IT'S AN ANALOG WORLD:

The circuit in this CPLD is a digital circuit. It must interface with the real world, which is analog. Here is the circuit that connects the CPLD to the external circuitry to make it a step motor drive. Below is the external schematic needed to form a complete motor drive.