User Guide: A Many-Channel FPGA Control System

Daniel Schussheim and Kurt Gibble (kgibble@psu.edu)

Introduction

Here we describe a general-purpose many-channel FPGA system (MCFS) that can implement numerous high- and low-speed PID servos, generate arbitrary waveforms, modulate and lock-in detect and perform digital signal processing.[1] We developed it to frequency lock laser cavities, stabilize temperatures and perform a number of controls and signal processing to laser-cool cadmium atoms. The system's inputs and outputs are:

TABLE 1: Inputs and Outputs

I/O type		# of channels		Sample rate (MS/s)	Range (V)
Fast ADC (LTC2194)	IN	10		100	±4
Fast DAC (MAX5875)	OUT	14		100	±18
Slow ADC (LTC2335-16)	IN	16		Maximum sample rate is 1 channel every μs on each of 2 chips, e.g., 2 ch. @ 1 MS/s or 16 ch. @ 125 kS/s	±10
Slow DAC (LTC2666-16)	OUT	16		0.05	±18
Fast Digital I/O, buffered (TXS0108E-Q1)	IN/OUT	6		100	5V digital
Fast Digital I/O, unbuffered (FPC connector)	IN/OUT	8		100+	3.3V digital
Slow Digital In (74LV165A)	IN	22		2	3.3V digital
Slow Digital Out (74VHC595)	OUT	26	19		
			7	3	
Display connectors (DB9)	N/A	2		N/A	N/A

This MCFS uses an Enclustra Mercury+ KX2 FPGA module that plugs into the baseboard. The baseboard uses all 216 accessible FPGA input/output (I/O). The Xilinx FPGA part number is XC7K160T-2FFG676I, which has 25,350 logic slices, each containing 4 look-up tables and 8 flip-flops; 600 DSP slices, containing a pre-adder, a 25×18 multiplier, a ternary adder and an accumulator; and 325 36 kb RAM blocks. Our baseline FPGA software uses 88,576 (87.4%) of the look-up tables distributed over nearly all of the logic slices. Other pin-compatible FPGA modules with more resources are available.

We use the system to control laser frequencies, optical cavity lengths, and the temperatures of cavities and nonlinear optical crystals. This work builds on a system developed by NIST,[2] adding more I/O channels, using a larger FPGA, and simplifying some filters to allow more servos on a single FPGA. This user guide describes the hardware, and software and firmware developed for this control system, including generating programming files (bitstreams) for the FPGA.

The software is written in Verilog, bitstreams are generated with a Hierarchical Design Flow with Vivado 2020.1, the baseboard was designed with Altium Designer Version 21.3.2 (Build 30), and the power supply board was designed in CircuitMaker.[3]

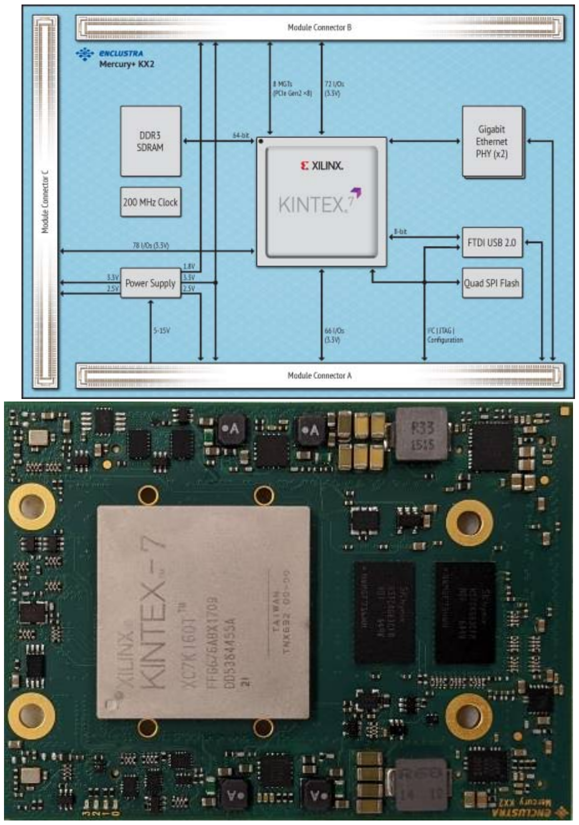

FIG. 1. Enclustra Mercury+ KX2 FPGA module functional diagram (from enclustra.com) and photograph.

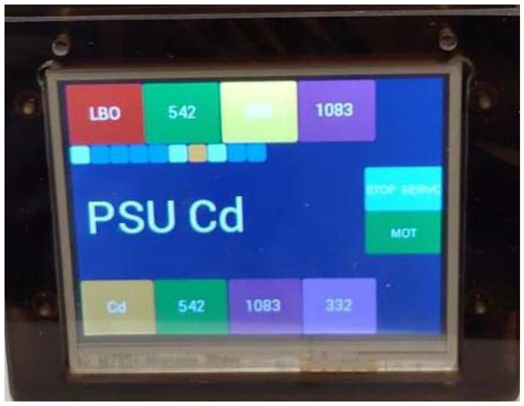

FIG. 2. LCD touch screen display.

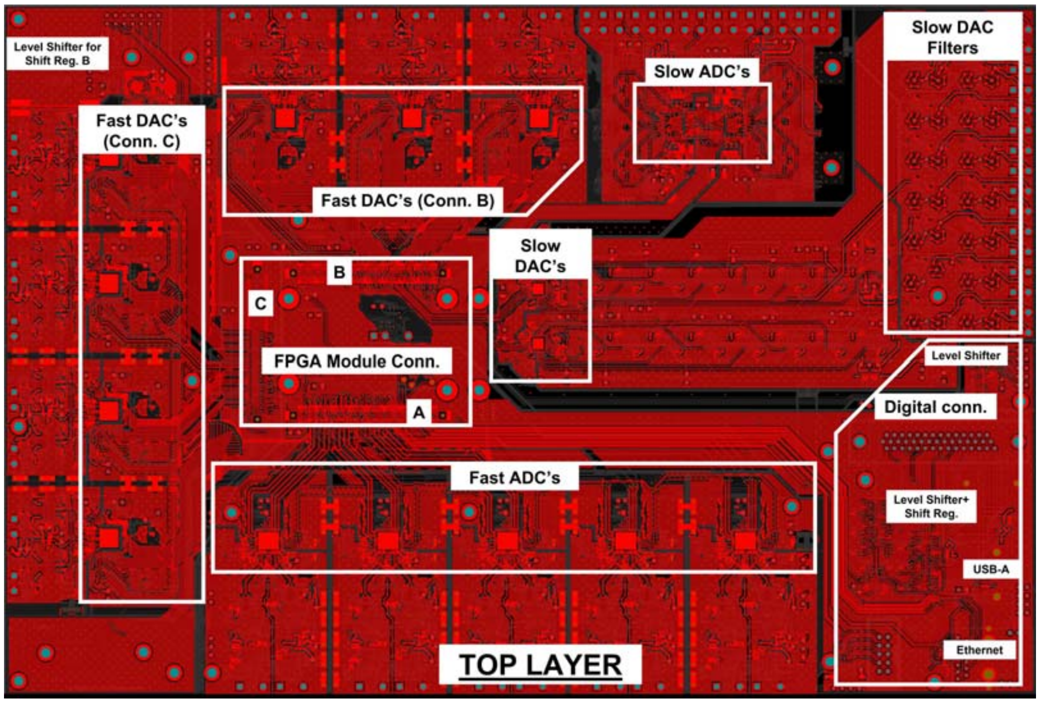

FIG. 3. Top layer of the 6 layer 8" x 12" baseboard. The FPGA module has three 168 pin connectors, labeled A, B and C.

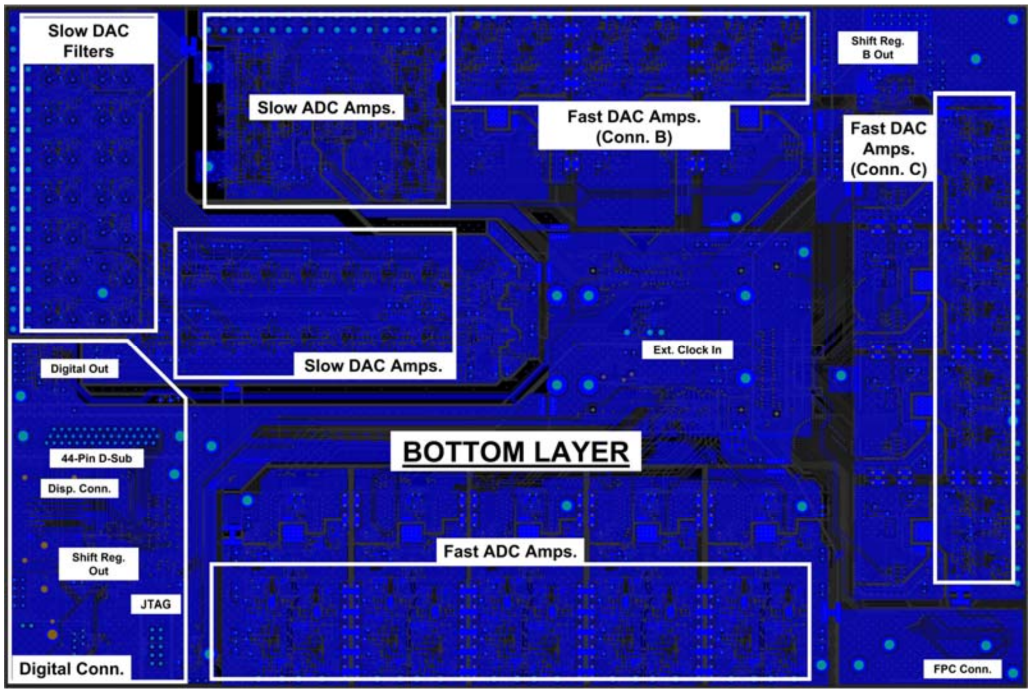

FIG. 4. Bottom layer of the baseboard. We mount the board so that this layer is easily accessible to facilitate changing analog I/O gains and filter components and accessing connectors.

Contents

Hardware

The following is a list of the main hardware components for the MCFS baseboard. A full component list is available in the bill-of-materials, "BOM.xlsx," in the repository.[4]

FPGA module

Mercury+ KX2: FPGA module ME-KX2-160-2I-D11-P from Enclustra GmbH with an XC7K160T-2FFG676I FPGA. This module combines a Xilinx Kintex-7 FPGA with fast DDR3 SDRAM, an FTDI USB 2.0 controller, dual Gigabit Ethernet, multi-gigabit transceivers, and high-speed low-voltage differential signaling (LVDS) I/O. Documentation is available from Enclustra.

Analog-to-digital converters (ADC's)

LTC2194 (×5): 16-bit, 2-channel simultaneous-sampling ADC, with a 105 MS/s maximum sampling rate and serial LVDS digital outputs. They have a ±4 V range in the default baseboard configuration.

LTC2335-16 (×2): 16-bit, 8-channel multiplexed ADC. The channels can be sampled in any order with a specified maximum conversion rate of 1 MS/s. For example, reading channels {1, 2, 1, 3, 1, 5, 1, 6, 1, 7} at 1 MS/s updates channel 1 every 2 µs, channels 2, 3, 5, 6 and 7 every 10 µs, and channels 0 and 4 are never read. The ADC can sample a specified channel on each conversion cycle, or a repeating sequence of up to 16 channels can be programmed. It uses a serial-peripheral interface (SPI) digital I/O bus. The default configuration is reading the channels in order, {0, 1, 2, 3, 4, 5, 6, 7} with a ±10 V input range.

Digital-to-analog converters (DAC's)

MAX5875 (×7): 16-bit, 200 MS/s, dual-channel, current-output DAC. The DAC uses a parallel CMOS I/O bus. The MCFS uses the interleaved mode to reduce the number of FPGA I/O used per chip, updating each channel at 100 MS/s. They have a range of ±18 V in the default baseboard configuration.

LTC2666-16 (×2): 16-bit, 8-channel multiplexed DAC with a specified maximum update rate of 50 kS/s. The DAC's are programmed over an SPI bus and channels can be updated in an arbitrary order. They have a ±18 V range in the default baseboard configuration.

Digital I/O

TXS0108E-Q1 (×2): 8-bit bidirectional, level-shifting, voltage translator, used to buffer FPGA digital I/O. Each chip buffers 8 lines, updating at up to 110 Mb/s (megabits per second).

74LV165A (×3): 8-bit parallel-load serial-in shift register, reading 24 digital inputs at nominally 2 MS/s.

74VHC595 (×5): 8-bit serial-in, parallel-out CMOS shift-register, driving 24 digital outputs at nominally 2 MS/s. A second bank drives 16 digital outputs at nominally 2.94 MS/s, 50/(16+1) MS/s.

External Clock Input

LTC6752: High-speed comparator for an external clock input. This is included on the baseboard but we have not tested or used it. Direct connections on the baseboard for an external digital clock input are also provided.

LCD touch screen

NHD-3.5-320240FT-CTXL-T: 3.5" LCD module with resistive touch sensing. It uses a FTDI/Bridgetek FT812 Embedded Video Engine (EVE2), supports audio, and has 1 MB of internal graphics RAM, built-in scalable fonts, 24-bit color and 320×240 resolution. It is programmed over an SPI bus and supports single-, dual-, or quad-wire modes; the MCFS baseboard is configured for the single-wire mode.

Software

Several software modules implement digital filters, dither modulation and lock-in detection, arbitrary waveform generation, drivers for an LCD touchscreen, and other DSP.[1] The following list describes the software modules for the MCFS, where parent modules instantiate modules listed under them. There is additional discussion of the display control at the end of this section.

`Top.sv`: The top-level module that contains all other module instances. It begins with declaring the port names for the signals connected to the FPGA I/O pins, and ultimately to the baseboard. These connections are specified in the constraints file, `top.xdc`. Five sections follow: `SIGNAL DECLARATIONS`, `SIGNAL ASSIGNMENTS`, `MODULE DECLARATIONS`, `FAST DAC TESTING SECTION`, and `PARAMETER REASSIGNMENT`. `SIGNAL DECLARATIONS` defines the signals used in `top.sv` and that connect to modules instantiated in it. These may include default values, such as default servo parameters. `SIGNAL ASSIGNMENTS` defines signal connections, notably high-level connections of software signals such as a cavity error or servo output to firmware I/O signals, i.e., a particular ADC/DAC channel. Modules are instantiated in `MODULE DECLARATIONS:` cavity servos; temperature servos; FM-MOT modules; display modules; ADC, DAC and shift-register firmware; serial input firmware; and clocking firmware. The `FAST DAC TESTING SECTION` has the signal declarations and module instances of `Sweep.v` and `rmp_sml.v`, which are used to optimize the fast DAC clock phase and measure their bipolar offset errors. Finally, `PARAMETER REASSIGNMENT` updates the parameter values received from the serial input module. Here, parameters can be conveniently fixed to a default value by inhibiting reassignment in this section.

> **`Gated_clk.v`**: Generates low-frequency clock signal using the 7-series FPGA primitive, `BUFGCE`. A counter enables a clock pulse once every `cntMAX` cycles. For example, to make a 1 MHz clock, `cntMAX = 100`. This is the recommended architecture for generating low-frequency clocks that the Vivado design timing tools can interpret as a clock without specifying additional timing constraints.
>
> **`Reset.v`**: A high EN input enables a counter that generates a 1-bit output, high from 100 to 200 µs after the counter starts, for a 100 MHz clock input. Just after the bitstream is loaded, the clock manager is reset, and several other modules (`FastADCsDDR.v`, `SlowADCs.sv`, `TempServos.v`, `SR_CTRL.v` for the display) are reset with a second instance. A third instance allows `SR_CTRL.v` for the display to be reset, which also reinitializes the display, when a reset signal is sent via the serial input.
>
> **`Sig1Dly.v`**: Generates a delayed logic output that is high N clock cycles after a falling trigger. It is used to delay the sleep mode of the fast DAC's and slow ADC's after the servos are turned off.

CavServos.v: Parent module that instantiates up to 10 fast cavity servos. The servos include a PID filter, auto-lock, a dither lock correction (a dither lock for servo 5), and overflow and underflow protection.

AutoLock.v: Generates the auto-lock sweep and enable for the fast cavity servos. It can be configured in several modes: using transmission or reflection as an input, various sweep configurations, and an optional secondary enable, for example to sequentially lock cavity servos.

Sweep.v: This module, from NIST,[2] makes a triangle wave. Two control inputs, on and hold, enable the ramp and hold the current value. A `resetmode` parameter has been added, which sets the off (`on = 0`) state output to be zero or its maximum value.

ErrorScaleShift.v: Bit-shifts an error signal and adds an offset and modulation. `CavPID.v` uses this as an input.

CavPID.v: Implements a fast PID servo, which we use for laser cavity servos.

I1BS.v: A 1st order integral filter with an optional low-frequency gain cap. The filter is implemented using bit-shift-addition operations instead of multipliers, and has a 25% gain and frequency resolution, i.e., the filter coefficients can be: … 0.5, 0.625, 0.75, 0.875 1, 1.25, 1.5, 1.75, 2, ... It has upper and lower output limits, and an input (nominally from the proportional filter output) that is subtracted from the limits, to prevent integrator wind-up. The module has an adjustable input and output signal size and an adjustable number of internal fractional bits. It allows timing margin at 100 MS/s with 25-bit input/output words and 32 additional internal fractional bits.

P1BS.v: A 1st order proportional bit-shift-addition filter with a high-frequency roll-down. The filter has a 25% gain and frequency resolution and upper and lower limits to prevent overflow and underflow. The module has adjustable input and output signal size and an adjustable number of internal fractional bits. It allows timing margin at 100 MS/s with 25-bit input/output words and 32 additional internal fractional bits.

DBS_rolloff.v: A 2nd order differential bit-shift-addition filter with a linewidth and a high-frequency roll-down. It has a 25% gain resolution and the frequency and linewidth coefficients have 2^n precision, which gives a resolution of factors of $\sqrt{2}$ for the high-frequency roll-down, and factors of 2 for the linewidth. Adjustable higher-precision frequency and linewidth coefficients did not have timing margin at 100 MS/s, but may nonetheless work. The filter has upper and lower limits to prevent overflow and underflow. The module has adjustable input and output signal size and an adjustable number of internal fractional bits. It allows timing margin at 100 MS/s with 25-bit input and output words and 16 additional internal fractional bits.

CavPID_rolloff.v: A variant of `CavPID.v` with 2^N resolution for the rolloff frequencies to provide more timing margin.

I1BS_rolloff.v and **P1BS_rolloff.v**: Variants of `I1BS.v` and `P1BS.v`, with 2^N resolution for the cutoff and rolloff frequencies to provide more timing margin.

CavPID_nornd.v: A variant of `CavPID.v`, with `I1BS_rolloff.v` and `P1BS_nornd.v` to provide more timing margin.

P1BS_nornd.v, **P1BS_nornd.v**, and **DBS_nornd.v**: Variants of P1BS.v, I1BS.v and DBS.v that have more timing margin by omitting the LSB rounding of the rolloff and cutoff frequency and damping contributions to the filter outputs while retaining 25% resolution of the frequency terms.

DitherLock.v: Parent module that instantiates a lock-in amplifier and modulation waveform generator, with logic to delay the start of the modulation. The demodulated output connects to an integrator (I1BS.v), whose output corrects an error signal, e.g., a Hänsch-Couillaud error signal, to lock to the maximum of a cavity's transmission. This module outputs the inhtrig signal that is used by ditherInhibit to stop the dither, for example, during a MOT detection cycle. The inhmode input sets the initial and final phase of the gated dither, with 22.5° resolution, so the servo output is nominally settled when the dither ends.

lockIn.v: Lock-in amplifier and modulation waveform synthesizer. The modulation is generated by twice integrating a step-wise waveform,[1] which has no 3rd harmonic and small 5th and 7th harmonics, producing a near sinusoid and further reducing all harmonics. The module outputs both quadrature components of a demodulated input signal at 1f, 2f or 3f. The demodulation has no sensitivity to the 3rd harmonic of the input signal, at 3f, 6f or 9f, respectively.

ditherInhibit.v: Inhibits or enables a dither with a phase set by inhtrig, after an enable signal goes low or high.

dithSwitch.v: Alternately enables two dither modulations: while one dither is enabled, the other is inhibited. It can avoid interference between two dither error signals, as for 361 nm SFG.

SelectableOutput<n>.v: Selects one of 0 to n-1 signals, e.g., to monitor on an analog output. SelectableOutput15.v additionally leaves the output unchanged for selection value of 15.

FM_MOT_PARAM_ASSIGN.v: Assigns one of 4 sets of parameters for different experiment configurations of FM_MOT.sv, selectable with the MOT display button.

FM_MOT.sv: Generates three arbitrary waveforms and instantiates a lock-in detector and a gated integrator for DSP. We use the arbitrary waveforms as triggers and to modulate the frequency and amplitude of a VCO that drives an AOM. An input, from a photomultiplier, is connected to a gated integrator with a background subtraction.

Cd_Oven_PARAM_ASSIGN.v: An input, which is connected to a touchscreen button, selects one of three sets of temperature servo parameters, or disables the servo. Here, it is used to select the temperature setpoint for a Cd oven.

fmMOTmem.v: Parent module with two instances of ram_dual.sv to store two signals and cyclically output the stored values, e.g., to fast DAC outputs for display. As implemented, it stores the gated-integrator output from FM_MOT.sv and the differences of its successive outputs. Debugging probes allow the stored values to be exported.

ram_dual.sv: Stores and reads signals from the FPGA Block RAM. The write-enable, we, triggers the signal capture.

TempServos.v: Parent module with 10 variable duty cycle temperature PID servos. The servos can be connected to analog outputs or variable duty cycle generators. The variable duty cycle outputs can control transistors to pulse heater currents, and they have synchronized delays for load diversity. We use a 125 kHz clock for our temperature servos.

temp_servo_startup_ramp.v: Module to slowly increase the temperature servo duty cycle with a time scale of order minutes, e.g., to slowly warm a non-linear crystal. When the error signal changes sign, or the duty cycle reaches the preset, the ramp stops and holds its value, the PID is enabled, and its output is added to the held ramp output.

tempPID.v: A PID controller like CavServo.v, with DBS.v, the higher resolution variant of the differential filter, which has less timing margin.

DBS.v: Like DBS_rolloff.v, but with 25% resolution on all filter coefficients. It has timing margin for clock frequencies below 80 MHz for a filter word size (SIGNAL_SIZE + FB) of 40 bits.

tempPID_rolloff_bitshift.v: Like tempPID.v, except with 2^N resolution for the filter frequency shifts.

tempPID_dg.v: Like tempPID.v, with debugging outputs: individual P, I and D filter outputs, their sum before and after rounding and after truncating fractional bits.

VDC.v: Generates a 1-bit variable duty cycle output with an adjustable delay for load diversity. It instantiates FastAVG4.v to increase the precision of pulse-width modulation temperature servos.

FastAVG4.v: Generates a sequence of 16 numbers that are added to the servo input to get sub-cycle precision, 1/16 of a cycle resolution, for the variable duty cycle output.[1] The MSB alternates every cycle and the LSB switches every 8 cycles.

RelockLEDs.v: Generates signals to indicate if a servo is locked, unlocked, or unlocked in the past few seconds. It is used in conjunction with the display program to change the color of servo indicator buttons.

err2clrst.v: Sets the 2-bit color state for temperature servo indicators.

display.v: Controls the LCD touchscreen. As described in more detail below, the display has buttons for controlling servos, the experiment state, and the Cd oven temperature, and it displays the status of cavity and temperature servos. The display.v code assembles the display commands into cmdarr and stores inputs from the display in readtemp. This module currently supports one display. The baseboard can support two independent displays, using the CS wires to alternately select each display, since the SDI and SDO wires are shared.

LCD_header.vh: A Verilog header file that contains the signal declarations, tasks and functions for display.v.

clr_str_4_st.v: Converts a 2-bit color state from err2clrst.v into a string command for a display color.

gain_inc_trig.v: Formerly used module to adjust servo parameters using the display. A button selects a servo parameter to adjust and buttons on the display increment or decrement parameters.

Dec2Str1B.v: Changes a signed integer from -9 to 9 into a string for the display. It was previously used in conjunction with gain_inc_trig.v.

Two additional modules are included that are not instantiated in the default designs.

`rmp_sml.v`: Generates a triangle wave with an amplitude of 5 adjustable equal steps (nominally 1 LSB) and an adjustable offset. It is useful for measuring the fast DAC bipolar offset error. In the `FAST DAC TESTING SECTION` of `top.sv`, instances of it for each fast DAC are commented-out.

`PIBS.v`: A 2nd order proportional-integral filter with a low-frequency integral gain cap and a high-frequency roll-down. The filter uses bitshift-addition operations instead of multipliers and has a 25% gain resolution. The filter has upper and lower limits to prevent overflow and underflow. The module has an adjustable input and output signal size and an adjustable number of additional internal fractional bits. It allows timing margin below 70 MS/s for a filter word size of (`SIGNAL_SIZE` + `FB`) 57 bits.

Touchscreen LCD Driver: display.v, LCD_header.v and SR_CTRL.v

The display (see Fig. 2) has 10 touch buttons to control the cavity servos, the Cd oven temperature, and the experiment sequence, which selects different output arbitrary waveforms. The cavity servo buttons change color depending on the cavity's lock state, red if unlocked, yellow if relocked in the last five seconds, and a servo-specific color if locked for more than five seconds. Pushing a cavity servo button cycles between the servo off state `Stp`, the scan state `Scn`, and the lock state, with its servo-specific name. The `STOP SERVO` button on the right side turns off all cavity servos except for the reference cavity lock of the 1083 nm laser. The Cd oven temperature servo has an `OFF` state and three different temperature setpoints: 99C, 119C, or an adjustable temperature setpoint. The Cd temperature servo button changes color depending on the temperature error signal; dark red and blue indicate high and low temperatures, and orange and light blue indicate high and low temperatures near the setpoint. The display also includes 10 temperature indicators (small squares below the top row of cavity servo buttons) for other temperature servos; these use the same color indicators as the Cd oven servo. The two rightmost indicators are used as monitors for the serial parameter reassignment and the fast ADC frames. The FM-MOT button has four states: `OFF`, and `SAT`, `MOT` and `MET`, which have three independent sets of parameters for the VCO FM, laser intensity and MOT coil voltage outputs.

The display is controlled by `display.v`, which contains the sequential logic to output and read the display data. Display signals, functions and tasks are declared in a header file, `LCD_header.vh`. It is added to `display.v` with:

```
`include LCD_header.vh
```

after the module port declarations. The header file includes hex RGB color definitions for the button and temperature indicator colors, temperature indicator range parameters, memory addresses and other parameters for the display, and functions to format, initialize, and assign the serial data that is sent to the display. The parameters, functions, and data formatting are given in the graphics controller programmer's guide.[5] The sequential logic in `display.v` has two states, `active` for display initialization and `data_active` for normal display operation. After initializing in the `active` state, it changes to `data_active`.

The bulk of the sequential logic in `display.v` updates the display and registers buttons that are pushed. The data sent to the display is loaded into `cmdarr` by calling the `assign_displaylist` task in `LCD_header.vh`. Touch buttons and indicators are specified by the `button` task. The touch buttons and indicators each have an identifying "tag" number, `regtouchtag` for the sequential logic. The program structure allows the display to be modified, including adding more control buttons.

The display is programmed and read through an SPI bus. The SPI bus includes a clock, SCK; the serial data input from the display, `SDI;` the serial output data, `cmdarr`, to the display, SDO; two chip select signals, `CS0_OUT` and `CS1_OUT` to select a display, potentially two independent displays; and a power down, PD_OUT. The baseboard has two DB9 connectors to support two independent displays. Additional displays without touch support can be connected in parallel. The display SPI signals are shift register inputs and outputs, which are controlled by the `SR_CTRL.v` firmware module. The 24-bit shift registers are clocked at 50 MHz, which limits the update rate of any shift register I/O to 2 Mb/s, and therefore the SCK clock to less than 1 MHz. The `cmdarr` data is a total of 182, 57-bit words, yielding a display update rate of 96 Hz. A pointer, `bitcount`, in `SR_CTRL.v` cyclically selects up to 24 I/O for the input and output shift registers. If fewer than 24 shift register I/O are used, the display update rate can be increased by decreasing the maximum `bitcount` to less than 24.

On some boards, we have observed occasional problems communicating with the display using the shift register I/O with display cables longer than 13 cm or long cables connected to other shift register outputs. These faults may be due to other shift register loads or marginal cable terminations. Normal function has always resumed after resetting the display via the serial line with `reset.v`. On our most used baseboard, no such problems occurred with a 70 cm cable – the shift registers on this baseboard used 5 V logic outputs, whereas the baseboard with occasional faults used 3.3 V. On the most recent baseboard design, an optional 5 V power connection has been added for the 3V3d.e power plane for the shift registers, with in-series pads for potentially a ferrite bead and two diodes, to reduce the 5V supply voltage. Vias near the display connectors allow them to be powered independently by the usual 3.3 V source, again through ferrite beads.

Firmware

A number of nominally static firmware modules capture incoming data, write data to the outputs, and perform baseline tasks such as generating clocks. After briefly describing each, additional detail is given for the ADC and DAC firmware, and finally the serial interface to reprogram parameters in real-time. The firmware uses several primitives, components native to the target FPGA's architecture, e.g., flip-flops, lookup-tables, shift-registers, RAM, multipliers, and I/O buffers. The UG953 user guide[6] contains the templates for instantiating 7-series FPGA primitives.

`GlobalClocks.v`: Generates the clocks used in the design from a 100 MHz clock on the Enclustra Mercury+ module. It uses a mixed-mode clock manager (MMCM) 7-series FPGA primitive. The clocks' frequencies and phases are set by their `div` and `phs` parameters, and `mmcmloc` selects the MMCM's location.

`FastADCsDDR.v`: Parent module with five instances of `LTC2194DDR.v` for the LTC2194 fast ADC chips. The ADC's are clocked by a shared ENC from the FPGA, generated by a source-synchronous output clock architecture using the ODDR 7-series FPGA primitive. The module sets the bit-slips and fine delay tap values for each data line, which are normally calibrated during the initial testing of a baseboard. It also includes the two instances of the 7-series FPGA primitive `IDELAYCTRL` and specifies their locations, so `IDELAYE2` primitives can be used in `LTC2194DDR.v`.[7]

`LTC2194DDR.v`: Firmware to drive an LTC2194 fast ADC. It deserializes the serial DDR data from the ADC using two phase-shifted 400 MHz clocks. Each ADC uses an LVDS pair for its frame, to synchronize the data, and 2 LVDS data pairs for each channel, 4 wires per channel. One data pair transfers the odd bits and the other the even. These ADC's are programmed via their SPI bus for 2's complement output. This module contains: LVDS buffer instances; deserialization of the output with the 7-series FPGA primitive `ISERDESE2`; five `IDELAYE2` FPGA primitive instances to fine tune deserializer input delays; and the routing of the odd and even bits to form two 16-bit words. The clock phases and line delay calibrations are described below. This interface is demonstrated up to 100 MS/s. UG471: 7 Series FPGAs SelectIO Resources[7] has more details on the primitives in this module.

`BitslipStatic.v`: Enables the bit-slip submodule of ISERDESE2 to apply a specified number of bit-slips to align the incoming serial data.

`BitslipDynamic.v`: Enables the ISERDESE2 submodule to apply bit-slips until the deserialized frame FR is `11110000`.

`FastADCs_SPI.v`: Programs the LTC2194 chips over their SPI bus via `SPI.v`. It generates one CS signal that connects to the five CS wires for the chips on the baseboard, so all chips are programmed simultaneously and identically. (Each ADC could be programmed individually by selecting each CS successively.) SCK is also generated with a source-synchronous architecture. To program the ADC's, this module sends 5 data packets: the first resets the control registers; the second and third specify 2's complement outputs over two LVDS pairs; and the final two data packets set a test pattern output, which can be used to output a known number when calibrating the ADC clock phases and data delays. It also sends a sleep command to all ADC's, 30 s after the `ADCOFF` input goes high, and a wake-up command when `ADCOFF` goes low and the ADC's are asleep.

`SPI.v`: Derived from NIST module[2] that implements a four wire SPI bus.

`FRmon.v`: Monitors the deserialized frames from the fast ADC's and outputs a status for a display indicator color. The displayed color is light blue if all of the frames have been `11110000` for more than 30 seconds, red if any of the five frames is not `11110000`, and orange for 30 seconds after all become `11110000`.

`SlowADCs.sv`: Parent module that includes `LTC2335_16.sv` and the 7-series FPGA input and output buffer primitives, `IBUF` and `OBUF`, for the out-of-context slow ADC implementation (see "Synthesizing and Implementing Bitstreams" below). It also implements a source-synchronous output clock with an enable.

LTC2335_16.sv: Firmware to capture data from two LTC2335-16 ADC's over an SPI bus. It selects the channel to be read on each conversion – both slow ADC's on our baseboard share a single SDI input so they both read their same channel number (if sampling at their maximum rate). The baseline firmware reads the channels in the order {0, 1, 2, 3, 4, 5, 6, 7} as 2's complement numbers.

SPI_LTC2335_16_seq_x2.v: This SPI variant waits an adjustable number of cycles after its chip select signal goes low, so that the timing requirements of the LTC2335-16 are satisfied. The two CS wires to each ADC are connected to the same logic signal so both slow ADC's are programmed simultaneously and identically.

FastDACs.v: Parent module containing seven instances of MAX5875.v, for the two-channel fast DAC's. The data of each DAC is interleaved at 200 MS/s for each of its channels updating at 100 MS/s. It implements a source-synchronous output clock with an adjustable phase-shift to meet the DAC timing requirements. The clock buffer locations are selected for this module instance's specific out-of-context placement.

MAX5875.v: Interleaves the fast DAC data and generates a channel select signal. The default baseboard configuration sets the data to be 2's complement.

SlowDACs.sv: Instantiates LTC2666x2.sv and the signal buffers for out-of-context implementation for the two LTC2666-16 slow DAC's, and includes a source-synchronous output clock with an enable.

LTC2666x2.sv: Addresses two LTC2666-16 multiplexed SPI DAC's that share an SDI. These DAC's have a specified maximum update rate of 50 kS/s. These DAC's use offset binary, so this module inverts the MSB's of the 2's complement data to be sent to the DAC's.

SPI_LTC2666x2.v: This SPI module alternately updates two slow DAC chips.

signFlip.v: Flips the sign of the slow ADC inputs and outputs to the slow DAC's because their inputs/outputs have inverting amplifiers. Since the most negative 16-bit 2's complement number is -2^{15} and the most positive is 2^{15}-1, this module flips the sign, and applies an offset of 1 to yield a 1-to-1 mapping of the inputs to outputs.

serialLine.v: Captures input serial data at 100 kS/s, beginning and ending with 16-bit handshakes. We use it to adjust parameters, which are sent as 27 sequential 35-bit numbers, sandwiched between two handshakes.

serialHSmon.v: Like FRmon.v for the fast ADC frames, it monitors the handshakes from serialLine.v, and outputs a status for a display indicator color, to indicate matching handshakes (light blue), non-matching handshakes (red), and no incoming data (dark blue), with handshakes of all 0's or all 1's.

SR_CTRL.v: Controls the 24-bit shift-register's digital I/O, which includes the display control signals, associated with display.v. Its signals, bitcount and cnt_reset, are connected to display.v for synchronization.

SRB_CTRL.v: Controls the 16-bit shift-register B's digital outputs.

We next discuss aspects of the firmware modules in more detail, including initial timing calibrations.

Fast ADC's: LTC2194.v, LTC2194DDR.v, BitslipDynamic.v and BitslipStatic.v

The baseboard has 10 fast analog inputs connected to five LTC2194 fast ADC's. Their data is transferred over a serial LVDS bus, a high-speed differential signal interface with 100 Ω termination between the two LVDS wires, typically close to the receiver. The LVDS inputs to the FPGA use an on-chip termination, specified in the input buffer primitive `IBUFDS` instances in `LTC2194DDR.v`.

The default design samples each fast ADC channel at 100 MS/s. Each channel uses two LVDS wire pairs; one pair for the odd bits of the 16-bit word and the other for the even. An 8:1 FPGA deserializer receives the serial data from each pair at 800 Mb/s and outputs a parallel 8-bit word at 100 MS/s. The 8-bit words for the even and odd bits are then combined to form the 16-bit sample. The FPGA deserializer primitive, `ISERDESE2`, is used in the double-data rate (DDR) mode and clocked with a 100 MHz clock and two 400 MHz clocks with adjustable phases. The five fast ADC's share a common 100 MHz LVDS encode (ENC) signal from the FPGA. The ADC's sample on the rising edge of ENC+, with 7 ENC cycles of latency, 70 ns at 100 MS/s, from sampling to output. The firmware adds an additional 100 MHz cycle of latency. The ENC is AC-coupled at each chip and the baseboard's default termination is at a central point, with additional pads at each chip.

Calibrating the Fast ADC ISERDESE2 Delays

`LTC2194DDR.v` uses fine delays and bit-slips to synchronize the capture of the serial LTC2194 conversion and frame data. The fine sub-cycle delays are implemented with `IDELAYE2` primitives. They can be incremented with an approximate resolution of 78 ps per tap delay, which is 1/32 of a cycle of the 400 MHz clock so the full range of 32 `IDELAYE2` taps spans two bit-slips. The bit-slips of the deserialized data are applied by the `bitslip` submodule of `ISERDESE2`. Initially, after programming, the `BitslipStatic.v` firmware instances enable `ISERDESE2` to apply a specified number of bit-slips to each conversion data signal. Thereafter, the five instances of the firmware submodule `BitslipDynamic.v` continuously apply bit-slips to the conversion and frame data to maintain each deserialized FR as `11110000`.

To calibrate the ADC's we apply a full-scale triangle wave on each fast ADC input and sample the conversion and frame data with an Integrated Logic Analyzer (ILA), a Vivado IP core.[8] We randomly sample the 3D parameter-space of the phases of the 100 MHz ADC ENC and the two 400 MHz deserializer clocks, relative to the 100 MHz deserializer clock. This yields a volume where the data are captured without glitches, although with likely incorrect bit-slips. We then fix the ENC phase near the center of that volume and map the glitch-free region in the plane of the two 400 MHz clock phases. Next, the fine delays are scanned and centered in their glitch-free range. It may be that the phase and fine delay adjustments have to be iterated. If all of the glitch-free ranges of all of the fine delays are more than two taps, all of the delays can be reduced with a larger ENC phase, two taps for each phase increment of 5.625°. The final step is setting the static bit-slips of each data signal, so that the MSB from the ADC is connected to the MSB of the output word. We found centered phases of (11.25°, -22.5°, -22.5°) for the ENC, 400 MHz, and secondary 400 MHz clocks, and centered taps between 8 and 24, with typical glitch-free ranges of 13 taps. We note that our glitch-free region is well outside of the specified expected phases for the two 400 MHz clocks, 0 and 180°.[9] The three phases are specified in `top.sv` and the taps in `FastADCsDDR.v.`

Finally, we note that we do not use the recommended synchronization for these ADC's, which is to use the DCO clock from the ADC to capture the data with the FPGA. This capture method requires two clock-capable FPGA input pairs per LTC2194 chip, one for FR to align the data and the other for DCO to latch the data, requiring an additional 10 clock-capable pairs for our five ADC's. This method could potentially require fewer phase and delay adjustments.

Fast DAC's: MAX5875.v

The baseboard has seven MAX5875 16-bit DAC's, providing 14 100 MS/s analog outputs. Each fast DAC uses 19 single-ended LVCMOS inputs, 16 for data, a channel select, a clock, and a sleep mode control, which is connected to a shift register B output. On the baseboard, the fast DAC's differential current outputs are terminated with 50 Ω and buffered by AD8421 instrumentation amplifiers with a default gain of 72 and an ±18 V output range. The `MAX5875.v` firmware uses the interleaved mode of the fast DAC's, sending the data for both DAC channels over the same 16 data lines. The three fast DAC's on FPGA connector B share a common 200 MHz clock from the FPGA and the four fast DAC's on FPGA connector C share another 200 MHz clock, yielding 100 MS/s update rates. The latencies of the two channels of each DAC are 8 and 9 200 MHz clock cycles, 40 and 45 ns, and the firmware latency is one 100 MHz cycle.

Slow, Multichannel, Serial ADC's: LTC2335_16.sv

The baseboard has 16 slow inputs connected to two LTC2335-16 16-bit ADC's. Each slow ADC has 8 channels, which can be read in an arbitrary order at a maximum specified update rate of 1 MS/s, e.g., a single input can be updated at up to 1 MS/s or all 8 channels at 125 kS/s. They are controlled with an SPI bus, which uses a clock SCK with an adjustable phase shift, data input `SDI`, chip select CS for each chip, and data output SDO for each chip. The two ADC's have five additional signals, a shared CNV to start conversions, a BUSY for each chip, which triggers the data transfer on the completion of an ADC conversion, and two PD power down's, which are connected to shift-register B outputs. Both slow ADC's share a single `SDI` baseboard wire and the default firmware connects both CS's to the same logic signal. Therefore, both chips read their same channel number when sampling at their maximum rate. The baseline firmware reads the channels in the order, {0, 1, 2, 3, 4, 5, 6, 7}, and sets the format as 2's complement via the SPI bus. Because the slow ADC's share a `SDI`, reading channels in an arbitrary order on each chip requires addressing each with its own CS enabled, and then addressing the other slow ADC. The maximum update rate is a combined 1 MS/s for both chips, e.g., updating both at 0.5 MS/s or just one at 1 MS/s, as opposed to a combined 2MS/s when reading the same channels on both, e.g, updating all 16 channels at 125 kS/s. An alternative mode is programming the internal sequencers of the ADC's to repeatedly read an arbitrary pattern of up to 16 configurations at 1 MS/s on each ADC. Commented out code to program the sequencers is included.

Slow, Multichannel, Serial DAC's: LTC2666x2.sv

Two LTC2666-16 slow DAC's on the baseboard provide 16 analog outputs and are controlled with an SPI bus, with a CS for each chip, and a shared SCK, `SDI` and SDO. The channels can be written in an arbitrary order; the baseline firmware[4] updates the slow outputs in order, {0, 1, 2, 3, 4, 5, 6, 7}. These DAC's use offset binary, so this module inverts the MSB's of the 2's complement data sent to the DAC's. The slow DAC's have a specified maximum update rate of 50 kS/s. They can be updated at higher rates, but overclocking produces a significant zero-crossing offset. We tested update rates of up to 2 MS/s and observed the zero-crossing offset increasing with update rate as 16 LSB/(MS/s) – updating at 500 kS/s has an offset of order 8 LSB. The `LTC2666x2.sv` module can compensate this offset, using the inputs `n0` and `n1`. The offset compensation is commented-out in the baseline firmware.

The slow DAC's also have intrinsic update glitches on their outputs. To reduce the glitch amplitudes, the baseboard has 5th order filters on the slow outputs to suppress frequencies above 50 kHz, with a steeper roll down above 300 kHz, while maintaining less than $\pi/4$ phase lag below 10 kHz. The glitch amplitude is less than 2 LSB, and the glitch impulse is 15 LSB·µs.

Adjusting Parameters with serialLine.v

The `serialLine.v` module captures serial input data used for parameter adjustment. When triggered, `serialLine.v` captures bits at 100 kb/s and partitions them into 27 sequential 35-bit numbers, sandwiched between matching 16-bit handshakes (Fig. 5) that specify the parameters to be updated. `top.sv` specifies a `handshake_default`, to which the "servo number" is added, e.g., our LBO cavity lock is number `20` and our Reference cavity temperature servo is number `33`.

We use a Stanford Research Systems DS345 arbitrary waveform generator to generate the serial data and trigger. It connects to the FPGA through two buffered digital input lines. An Excel spreadsheet and associated visual basic macro encodes parameters in an arbitrary waveform text file that a LabVIEW program uploads to the DS345 via GPIB. The Excel files and LabVIEW program are available in the repository.[4] Implementing USB or ethernet connections with the FPGA are other options, which might use more FPGA resources.

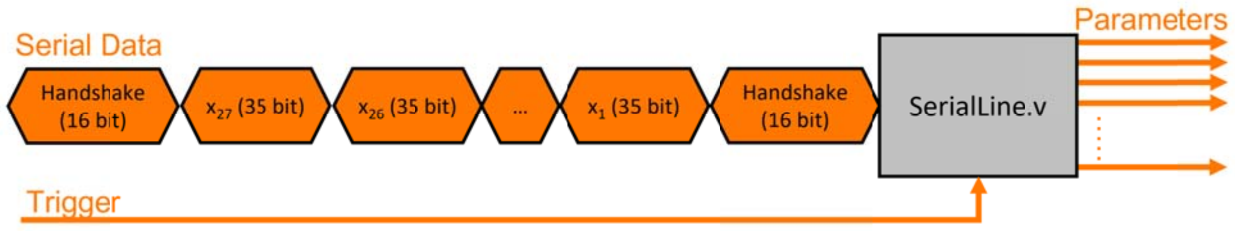

FIG. 5. Serial data transfer. An external trigger initiates the serial upload of adjustable parameters. The serial stream of 27 sequential 35-bit numbers, $\{x_1, x_2, \ldots, x_{27}\}$, begins and ends with identical 16-bit handshakes that specify the parameters to update.

Programming the FPGA

We program the FPGA via its JTAG port with a USB-JTAG adapter, Digilent JTAG-HS3. To load the provided bitstream, (disconnect and then) apply power to the baseboard, connect the adapter to the baseboard and computer, and open the Vivado Hardware Manager. Click the "Auto Connect" shortcut and then the "Program" shortcut in the "Hardware" window, select the bitstream, e.g., `top.bit`, from its file location in the pop-up menu, and click the "Program" button (both of these may appear in the green bar above the Hardware window). The provided design has an ILA debugging probe and Vivado will automatically add a `top.ltx` probe in the same directory when `top.bit` is selected. After the bitstream is loaded, the debugging probes file can be reloaded by right clicking on the FPGA in the Hardware window and choosing `top.ltx` as the probes file. UG908: Vivado Design Suite User Guide Programming and Debugging[10] and UG936: Vivado Design Suite Tutorial Programming and Debugging[11] have further information and a detailed tutorial.

Flash Programming

The Mercury+ KX2 module has on board flash memory that can store a bitstream to automatically load at FPGA power up. To program the flash memory, switch the `BOOT_MODE` control line on the baseboard to logic-high `SLAVE` mode. With the FPGA connected, in the Vivado Hardware Manager right click to "Add New Memory Configuration Device." Select the Spansion `s25fl512s` flash chip of this FPGA module in the menu window. The flash chip should then appear in the device hierarchy in the "Hardware" window of the Vivado Hardware Manager.

To generate the memory configuration files, select "Generate Memory Configuration File" from the "Tools" drop-down menu in the top bar, select the memory part as `s25fl512s`, select an output configuration (`.mcs`) file name, select the SPI×4 interface, check the "Load bitstream files" box and select the bitstream file to load, check the "Overwrite" box, and click OK. This will generate the `.mcs` and associated `.prm` file. To program the memory, right click on the `s25fl512s` flash chip in the device hierarchy in the Hardware window and select the generated `.mcs` and `.prm` files. At this point, the bitstream can be manually loaded from the flash memory with the Hardware Manager. Alternatively, it can be automatically loaded from flash after a power cycle, with the `BOOT_MODE` switch in logic low `FLASH` mode. Note that the bitstream will not load from flash memory if the FPGA's JTAG port is connected to a computer. For additional discussion see UG908,[10] Chapter 5: Programming Configuration Memory Devices.

Synthesizing and Implementing Bitstreams

Quick Start Guide and Summary

This project uses Hierarchical Design (HD) flow to implement a number of firmware modules, and the display and FM-MOT modules out-of-context (OOC), which synthesizes and implements the OOC modules before the full design. This can reduce the time to synthesize and implement the entire design and provides consistent placement and routing of these OOC modules on the FPGA, and thereby likely gives more reproducible timing margins. A detailed discussion of executing each step of synthesis and implementation follows this quick start guide. The project in the repository[4] includes implemented OOC modules.

Generating Tcl scripts. The `Tcl_Script_Generator.tcl` script creates scripts to generate bitstreams from the provided MCFS firmware and software. The generating script uses files from the UG946:

Vivado Hierarchical Design Tutorial version 2014.1, which can be obtained from https://secure.xilinx.com/webreg/clickthrough.do?cid=356076&license=RefDesLicense&filename=ug946-vivado-hierarchical-design-tutorial.zip. This link will prompt the user to sign a Xilinx Design License Agreement before the files can be downloaded. After extracting the 14 Tcl files, replace c:/MCFS_HD/UG946 in `Tcl_Script_Generator.tcl` below with the reference design root directory, which contains the extracted file `design_complete.tcl`. Entering the command below in the 2020.1 Vivado Tcl Shell will generate the Tcl scripts variants for the provided MCFS firmware and software from the downloaded Xilinx reference design.

```
source Tcl_Script_Generator.tcl -notrace
```

Top-level synthesis and implementation. To synthesize and implement the design without changing the OOC modules, e.g., for many changes in `top.sv` or `CavServos.v`, use the following command in the Vivado Tcl Shell:

```
source MakeBitstream.tcl -notrace
```

This runs the `MakeBitstream.tcl` script, which executes the `design.tcl` script and copies useful files to the `Implement/BIT/dump` directory. The `design.tcl` script, whose steps are described in more detail throughout this section, specifies the already implemented OOC modules and the steps to synthesize and implement the full design without resynthesizing or reimplementing the OOC modules. It generates a bitstream to be loaded onto the FPGA in of order 40 minutes.

Top and OOC synthesis and implementation. To synthesize and implement modified OOC modules, the `MakeOOC.tcl` script executes `design_ooc.tcl`, a variant of `design.tcl` with `run.topSynth`, `run.oocSynth`, `run.tdImpl` and `run.oocImpl` set to 1. `MakeOOC.tcl` also backs up previous OOC synthesized and routed design checkpoints (DCP's) before sequentially running the above flow steps to resynthesize and reimplement the OOC modules. The `design_ooc.tcl` script uses a variant of `top.sv` located in `Sources/hdl/ooc` that prevents unwanted optimization that would fix OOC module ports. For example, the `top.sv` variant yields a `SlowADCs.sv` OOC implementation with all of its ports adjustable, including the channel sampling sequence. If `SlowADCs.sv` was synthesized and implemented OOC with the slow ADC channel sequence of the default `top.sv`, the module ports that pass the sampling sequence to `SlowADCs.sv` would be fixed; changing the sequence would then require rerunning the `SlowADCs.sv` OOC synthesis and implementation, as well as the top level `MakeBitstream.tcl`.

The following commands reimplement all OOC modules and generate a bitstream:

```
source MakeOOC.tcl -notrace
```

followed by

```
source MakeBitsream.tcl -notrace
```

The OOC modules in this project are the slow ADC firmware (`SlowADCs.sv`), the fast and slow DAC firmware (`FastDACs.v` and `SlowDACs.sv`), the FM-MOT module (`FM_MOT.sv`), the display module (`display.v`) and the shift-register firmware (`SR_CTRL.v`) for the displays, temperature servo outputs and other digital I/O. Implementing the display module OOC saves considerable time during synthesis and implementation of the full design and the four OOC interface modules are rarely modified. We also tried to implement the fast ADC firmware (`FastADCs.v`) OOC but were unsuccessful. This could be due to the fast ADC firmware using `IDELAYCTRL` groups, which are not fully preserved in HD.[12] We note that generating modules OOC does not always make bitstream generation more efficient – implementing the cavity or temperature servos OOC did not reduce the time for the top level synthesis and implementation.

Hierarchical Design (HD) Flow

We now discuss HD Flow, and then the synthesis and implementation of the OOC modules and the full design using `design_ooc.tcl` and then `design.tcl` Tcl scripts that run in the Vivado Tcl Shell. If there are no changes to the OOC modules, they do not have to be resynthesized and reimplemented, and one can proceed directly to the "Top Implementation" section below for guidance on modifying the default `design.tcl` script. Note that there is no GUI for the HD flow. In contrast to project flow, the HD flow can define the physical locations on the FPGA of selected module instances. More information on HD is available in the user guide, UG905: Vivado Design Suite User Guide Hierarchical Design,[12] and the tutorial, UG946: Vivado Design Suite Tutorial Hierarchical Design.[13] Our HD project for the MCFS is adapted from the tutorial design.

Hierarchical Design `.prj` Files

The Tcl scripts specify the names of `.prj` files that must list all of the modules to be synthesized. For top-level synthesis, `top.prj` lists all of the modules except those that are instantiated in only OOC modules. Modules that are implemented OOC have to be designated in `top.prj` using dummy modules, called black boxes or stub files. A black box only declares the module's ports, which define that OOC module's connections to the rest of the design. During top-level implementation, the placed and routed OOC modules are inserted into the black box locations. The `design_ooc.tcl` script has to similarly specify additional `.prj` files that list all the OOC modules to be synthesized, including their submodules that may not appear in `top.prj`. The `.prj` files are located in the `Sources\prj` directory.

OOC Module Synthesis

Top-level synthesis with black boxes precedes OOC synthesis, which interprets the HDL code for implementation in a specified physical FPGA location. Here, as an example, we specifically describe the OOC module synthesis of the display module, `display.v`. In `design_ooc.tcl`, each OOC module has a subsection with the heading "`OOC Module Definition and OOC Implementation`." The display section begins with the assignment of module name display to the variable module0 and, for the other OOC modules, the sections are similar, albeit with different variables, e.g., module1, module2…. The section also specifies display.prj and a directory Synth\display for this module's OOC synthesis, and designates the instance name disp0 that is in top.sv. There can be multiple instances of an OOC module, and all of these can use the same OOC synthesized design checkpoint (DCP), but each instance requires a unique name and physical FPGA location for OOC implementation. The final few lines of the display section of design_ooc.tcl declare the names of the constraints .xdc files. These will be created during Top-Down implementation and contain the physical optimization and timing constraints for the disp0 instance.

The provided design_ooc.tcl script successively runs topSynth, oocSynth for all OOC modules, tdImpl, and oocImpl for all OOC modules. This script specifies variables to control the HD flow steps to synthesize and implement the OOC modules. For example, modifying display.v without changing its ports only requires running the OOC synthesis and implementation for display.v, without resynthesizing and reimplementing the other OOC modules. To run only OOC synthesis for display.v, the values of the flow control variables in design_ooc.tcl are set as:

- run.oocSynth is set to 1.
- run.topSynth, run.tdImpl, run.oocImpl, run.topImpl and run.flatImpl are set to 0.

Additionally, the synth attributes of the other OOC modules must be changed to 0 in design_ooc.tcl. The line in each OOC Module Definition section ending with ${run.oocSynth} and each OOC Implementation line ending with ${run.oocImpl} can be commented-out and the following line ending in 0 uncommented.

When the MakeOOC.tcl script runs design_ooc.tcl, with the above modifications and the value of the synth attribute for the display module is unchanged, it will synthesize only display.v and preserve the synthesized DCP's for the other OOC modules. MakeOOC.tcl backs up the OOC constraints and the synthesized and routed DCP's before design_ooc.tcl overwrites any.

Top-Down Implementation for OOC Implementation

Top-Down implementation is an intermediate step between OOC synthesis and OOC implementation. It creates the constraints files that define the OOC modules' timing characteristics and the connection points between the rest of the design and the Pblocks containing the OOC module. If a modified OOC module's ports are unchanged, Top-Down implementation does not need to be repeated and flow can proceed to the implementation of the OOC module(s). The Pblock for an OOC module is the physical area reserved on the FPGA for its logic elements and routing, and must be defined for each OOC module before running Top-Down implementation. One method to set a Pblock's parameters is:

1. Implement the full design with the module not OOC.
2. Open the routed DCP, top_route_design.dcp in the Implement\top directory.

3. In the netlist window, right click on the module instance that will be implemented OOC and select "Highlight leaf cells." The cells associated with that instance should be highlighted in the chosen color.
4. In the device view window click "Draw Pblock" and draw a rectangle enclosing the highlighted cells. If the cells are spread too widely, or there is a risk of overlapping another `Pblock`, draw a `Pblock` that encloses most of the cells and will be large enough to contain all of the module's logic.
5. Right click on the `Pblock` in the device view window and click "Pblock properties". In the "Pblock Properties" window, click the "Properties" tab and copy the values in the `DERIVED_RANGES` field.
6. In the `Sources\xdc` directory, open `top_flpn.xdc`. It should contain all the constraints in `top.xdc` and the `Pblock` definitions for the OOC modules.
7. Create the `Pblock` constraints by pasting the `DERIVED_RANGES` values as the argument of the `resize_pblock` command (line 3 below), and substituting the instance name, e.g., `disp0` for `display.v`, as follows:

```
1 create_pblock pblock_disp0
2 add_cells_to_pblock [get_pblocks pblock_disp0] [get_cells -quiet [list
disp0]]
3 resize_pblock [get_pblocks pblock_disp0] -add{RAMB36_X1Y0:RAMB36_X6Y5\
4                                              RAMB18_X1Y0:RAMB18_X6Y11 \
5                                              DSP48_X0Y0:DSP48_X5Y11\
6                                              SLICE_X12Y0:SLICE_X109Y32}
7 set_property CONTAIN_ROUTING 1 [get_pblocks pblock_disp0]
8 set_property HD.PARTPIN_RANGE {} [get_pins {disp0/*}]
```

8. Change the value of `run.tdImpl` to `1` and the values of the other flow control variables to `0` in `design_ooc.tcl`, and then execute `MakeOOC.tcl` to run Top-Down implementation.
9. If an `HD.PARTPIN_RANGE` property was not defined, as in the above template, a suggested `HD.PARTPIN_RANGE` will be displayed in the shell window after the script finishes. Paste this suggested range into the empty brackets `{}` after `set_property HD.PARTPIN_RANGE`, as in line 8 above, for its use in OOC implementation.

As previously mentioned, Top-Down implementation creates several constraints files that define the connection points, timing constraints and optimization for the OOC modules. It can be helpful to inspect these constraints files located in `Sources\xdc`, particularly ones with names like *`<instance_name>`*`_ooc_optimize.xdc`. Those files may contain entries such as:

```
set_logic_zero [get_ports {port_name[3]}]
```

where `port_name[3]` denotes bit 3 of the `port_name` signal of the OOC module, which will always be `0` here. If this port should be adjustable, a possible solution is adding a `(* DONT_TOUCH *)` directive before the declaration of the signal that is being set to zero in the top-level or OOC module – see top.sv for examples. If the incorrect constraint remains after rerunning topSynth, oocSynth for the associated module, and tdImpl, another option is making the signal adjustable in the variant of top.sv in Sources\hdl\ooc. The design_ooc.tcl script targets this modified version of top.sv that avoids fixing any OOC module's ports.

OOC Implementation

With a valid Top-Down implementation, an OOC module can be implemented by setting the value of the run.oocImpl flow control variable to 1 and the others to 0 in design_ooc.tcl. Within the OOC module's subsection, its impl attribute should retain its default value, ${run.oocImpl}. For the modules that are not to be implemented, the ${run.oocImpl} line should be commented-out and the following line should be uncommented so the values of their run.oocImpl flow control variables are set to 0.

When the ooc preservation attribute, which directly follow the specification of impl, is set to routing, the OOC module's placement and routing are preserved when it is added to the full design during the final top-level implementation. If there are routing conflicts between the top-level module and an OOC module, they may be resolved by changing the instance's preservation to placement, which locks the placement and not the routing. When preservation is set to placement, the OOC module's routing is used as a starting point for the final top-level implementation. We have observed routing failures when implementing the full design with OOC SlowDACs.sv with routing preservation; with only placement preservation the top-level implementation reliably routes. Additionally, if the routing conflict is due to congestion, e.g., because the OOC module's elements are restricted to be within a Pblock, shifting the Pblock's boundaries so that there are unconstrained regions between adjacent Pblocks may ameliorate the conflict.

Top Implementation

The final flow step is top-level implementation, which optimizes, places, and routes the non-OOC logic and integrates the implemented OOC modules. The default design.tcl script has default flow control variable values that ultimately yield a bitstream.

- run.topSynth and run.topImpl are set to 1.
- run.oocSynth, run.tdImpl, run.oocImpl and run.flatImpl are set to 0.

An example where topSynth is disabled is discussed below.

The Top Definition section of the default design.tcl defines the top-level module and then specifies the implementation steps to run. The steps executed during top-level implementation are again controlled by attributes, as for OOC synthesis and implementation, with the below defaults.

```
# Attributes that control top-level implementation steps.
set_attribute impl $top      link          1
set_attribute impl $top      opt.pre       "$tclDir/make_debug.tcl"
set_attribute impl $top      opt           1
set_attribute impl $top      place         1
set_attribute impl $top      phys          1
set_attribute impl $top      route         1
set_attribute impl $top      bitstream.pre "$tclDir/write_ltx.tcl"
set_attribute impl $top      bitstream     1
```

With the `link`, `opt.pre`, `opt`, `place`, `phys`, `route`, `bitstream.pre`, and `bitstream` attributes set as above, the `link_design`, `pre_opt_design_script`, `opt_design`, `place_design`, `phys_opt_design`, `route_design`, `pre_write_bitstream_script` and `write_bitstream` flow steps run successively. The `link_design` step links the OOC and the top-level netlists and constraints files to the target XC7K160T FPGA. A netlist file describes the components and connectivity of the FPGA design, which are given in the OOC and top-level DCP files. The `pre_opt_design_script` executes the specified `make_debug.tcl` script, which adds ILA debugging probes to read the stored gated-integrator DSP signals of the FM-MOT module. The `opt_design` step optimizes the netlist for the target FPGA, potentially simplifying the design before placement and routing. The `place_design` step specifies the design ports and logic cells to resources on the target FPGA. The `phys_opt_design` optimizes placements of the post-placement netlist to improve timing margins. The `route_design` step completes the logic by connecting the placed components. The `pre_write_bitstream_script` executes the `write_ltx.tcl` script, which generates the debugging probes file, `top.ltx`, before `write_bitstream` generates the FPGA bitstream `top.bit`. DCP files are created after the `link_design`, `opt_design`, `place_design`, `phys_opt_design`, and `route_design` steps.

Debugging Probes

The Xilinx IP Catalog includes an Integrated Logic Analyzer (ILA) for debugging probes for sampling real-time internal FPGA signals. Chapter 10 of UG908 (page 114) describes several ways to add debugging probes to a design.[10] The highest-level approach adds debugging probes using the ILA wizard, which creates a Verilog Instantiation Template, which may be added to the user's code, like any other module. For HD flow, we instead use Tcl XDC debug commands, in the default `make_debug.tcl` script, to add the debugging probes to a DCP file's netlist with no modifications to the HDL code.

To generate Tcl debug commands, we interactively add the debugging probes in the Vivado GUI. With the values of the flow control variables `run.topSynth` and `run.topImpl` set to `1`, and the attribute settings below, the HD flow is stopped after the `link_design` step so the desired signals for debugging can be selected from the netlist in the created `top_link_design.dcp` file.

```
# Attributes that control top-level implementation steps.
set_attribute impl $top      link          1
# set_attribute impl $top      opt.pre       "$tclDir/make_debug.tcl"
set_attribute impl $top      opt          0
set_attribute impl $top      place        0
set_attribute impl $top      phys          0
set_attribute impl $top      route         0
# set_attribute impl $top      bitstream.pre "$tclDir/write_ltx.tcl"
set_attribute impl $top      bitstream     0
```

The top_link_design.dcp file contains all the signals in the design, including those in OOC module netlists. With top_link_design.dcp open in the Vivado GUI, the design signals are listed in the "Netlist" window and can be selected for debugging by right clicking on the signal and choosing "Mark Debug." After all desired signals are selected, open the "Tools" menu and select "Set Up Debug…." In its pop-up window, the sampling clocks can be specified for each of the signals in the "Clock Domain" field of the "Nets to Debug" section. The number of samples to capture is set in the following "ILA Core Options" window. When "Finish" is clicked, Tcl commands will execute and add the debugging probes to top_link_design.dcp. Those Tcl commands include the sampling parameters and can be copied and pasted from the "Tcl Console" window into a script file, such as make_debug.tcl, so that these debugging probes will be automatically added in the pre_opt_design_script step in future HD flow executions.

Here, after top_link_design.dcp is saved, implementation can resume with the opt_design step and finish with write_bitstream, by setting the values of the design.tcl flow control variable run.topSynth to 1 and Top Definition attributes as follows:

```
# Attributes that control top-level implementation steps.
set_attribute impl $top      link          0
#set_attribute impl $top      opt.pre       "$tclDir/make_debug.tcl"
set_attribute impl $top      opt           1
set_attribute impl $top      place         1
set_attribute impl $top      phys          1
set_attribute impl $top      route         1
set_attribute impl $top      bitstream.pre "$tclDir/write_ltx.tcl"
set_attribute impl $top      bitstream     1
```

HD Flow Examples to Modify **display**

Here we give two examples of script modifications to resynthesize and reimplement a modified display.v, with no changes to other OOC modules. If the ports of display.v are unchanged, setting the values of the run.oocSynth and run.oocImpl flow control variables to 1 and the others to 0 in design_ooc.tcl will rerun only OOC synthesis and OOC implementation. The default attributes for display for synth and impl allow OOC synthesis and OOC implementation to run. To prevent resynthesis and reimplementation of the other unchanged OOC modules, set the values of their synth and impl attributes to 0 (including making their preceding lines comments), as in the display example in "OOC Module Synthesis" above.

If the ports of `display.v` are changed, the top-level synthesis and top-down implementation must be rerun, in addition to OOC synthesis and OOC implementation. Before top-level synthesis, the ports of `display`'s black box module, `Sources\hdl\blackbox\display_bb.v`, will have to change to match those of the modified `display.v`. Note that the `top.sv` variant for `design_ooc.tcl` may also need to be modified to avoid the ports of `display.v` being fixed. The default values of the flow control variables in `design_ooc.tcl` should not change and the values of the `synth` and `impl` attributes, for `display` and the other OOC's, should be set as in the above example for unchanged ports.

After setting the values of the flow control variables and the attributes of either example, running `MakeOOC.tcl` will reimplement `display.v` for this and any future bitstreams, and then `MakeBitsream.tcl` will generate an updated bitstream.

Fast ADC Calibration Scripts

We provide additional Tcl scripts to automate generating bitstreams for calibrating the Fast ADC clock phases and fine delay tap values. The `multiphase.tcl` and `multitap.tcl` scripts execute `design.tcl` multiple times, saving files, including bitstreams and debugging probes files, for each run in `Implement\BIT\dump`. The scripts are in the `FastADCcal` directory and can be executed with:

```
source FastADCcal/multiphase.tcl -notrace
```

and

```
source FastADCcal/multitap.tcl -notrace
```

Here, the top-level module, `top_fADC.sv`, is a minimal version of `top.sv` that contains the firmware for the fast ADC's, fast DAC's (OOC), and shift-register B, which control the fast DAC's power-downs. The `multiphase.tcl` script successively sets the phases of the fast ADC ENC and two 400 MHz clocks in `top_fADC.sv` with the values in `phases.sv`. It will generate a bitstream, with debugging probes for the fast ADC conversion data and deserialized frames, for each set of phases in `phases.sv`. Similarly, the `multitap.tcl` script successively sets the delay taps in `FastADCsDDR.v` to the fine delay tap values in `taps.sv`, generating a bitstream for each set. These scripts can be modified to automate generating other series of bitstreams.

Suggestions on Debugging HD Flow

Here we compile some tips on debugging HD projects. The Xilinx user guide and tutorial have additional information.[12,13] A number of useful checks are:

1. After running `topSynth`, the OOC modules should be black boxes in the synthesized design checkpoint, and have all required signals, e.g., on the black box modules.
2. After running `oocSynth`, the recently synthesized module(s) will have synthesized DCP(s), which should contain all of the input and output signals with their expected number of bits indicated.

3. After running `tdImpl`, there should be four constraints files associated with each OOC module instance in the `Sources\xdc` directory. One of these is `<instance_name>_ooc_optimize.xdc`. It includes constraints that fix the module's I/O that are static or unused in `top.sv`. If any of these signals are to be adjustable, the OOC module or `top.sv` should be modified.
4. After running `oocImpl`, the routed DCP for the OOC module can be checked as in step 2.
5. After running `topImpl`, all logic should be placed and connected in the routed DCP. If an OOC module causes top implementation to fail in the routing stage, relaxing its `preservation` attribute from `routing` to `placement`, as described in the "Top Implementation" section above, may allow the design to route successfully.

Power supply board

The MCFS power supply[3] uses a combination of switching and linear regulators to generate the 1.8 V, 2.5 V, 3.3 V, 5 V, ±15 V and ±18 V supply voltages for the baseboard from a single +15 V input. Table 2 lists the supply voltages shown in Fig. 6, and the approximate current used by each chip or device. To reduce switching frequency harmonics and their impact on the experiment, we use high switching frequencies, separate the power supply board from the baseboard, connect the supply voltages with twisted wire pairs wrapped around toroidal ferrite cores, and have a second stage of linear regulators for analog components. We use an I6A24008A033V-N01-R[14] module to generate –20 V. Its switching frequency is 380 kHz, and we choose frequencies greater than 600 kHz for the other switching chip regulators.

TABLE 2: Chip Voltages and currents

Purpose	**Power Supplies (V)**	**Est. current (mA)**
FPGA module power	15 (switching)	175
FPGA I/O bank 13, 15 and 16 (for fast DAC's, slow ADC's and DAC's, shift register B outputs, and FPC connector)	3.3 (switching)	250
FPGA I/O bank 12 and 14 (for fast ADC's, shift register I/O, and JTAG)	2.5 (switching)	350
Fast ADC's (LTC2194)	1.8 (linear & switching)	(205 & 32) × 5
Voltage reference for fast ADC's (ADR127)	3.3 (linear)	0.1 × 5
Fast ADC SPI pull-up	2.5 (switching)	1
Fast input amps (OPA228)	±18 (linear)	±4.5 × 10
Fast input differential amps (LTC6406)	3.3 (linear)	220
Fast DAC's (MAX5875)	3.3 (linear & switching) 1.8 (linear & switching)	(58 & 3) × 7 (32 & 25) × 7
Fast output amps (AD8241)	±18 (linear)	±36 × 14
Slow ADC's (LTC2335-16)	±15 (linear) 5 (linear) 3.3 (switching)	(+4.5/–5) × 2 14.5 × 2 4.2 × 2
Voltage reference for slow ADC's (LTC6655-4.096)	5 (linear)	7.5 × 2
Slow input quad amps (AD8513)	±18 (linear)	±8.45 × 4
Slow DAC's (LTC2666-16)	±15 (linear) 5 (linear) 3.3 (switching)	±5.5 × 2 6.5 × 2 0.002
Slow output amps (LF356)	±18 (linear)	±10 × 16
Digital buffers (TXS0108E-Q1) for shift-registers	3.3 (switching) 2.5 (switching)	0.006 × 2 0.002 × 2
Digital buffer (TXS0108E-Q1) fast digital I/O	5 (switching) 3.3 (switching)	0.006 0.002
Shift register I/O (74LV165A /74VHC595)	3.3 (switching)	0.02 × 3 / 0.08 × 5
JTAG power	2.5 (switching)	50
Clock buffer digital side power (LTC6752)	2.5 (switching)	5

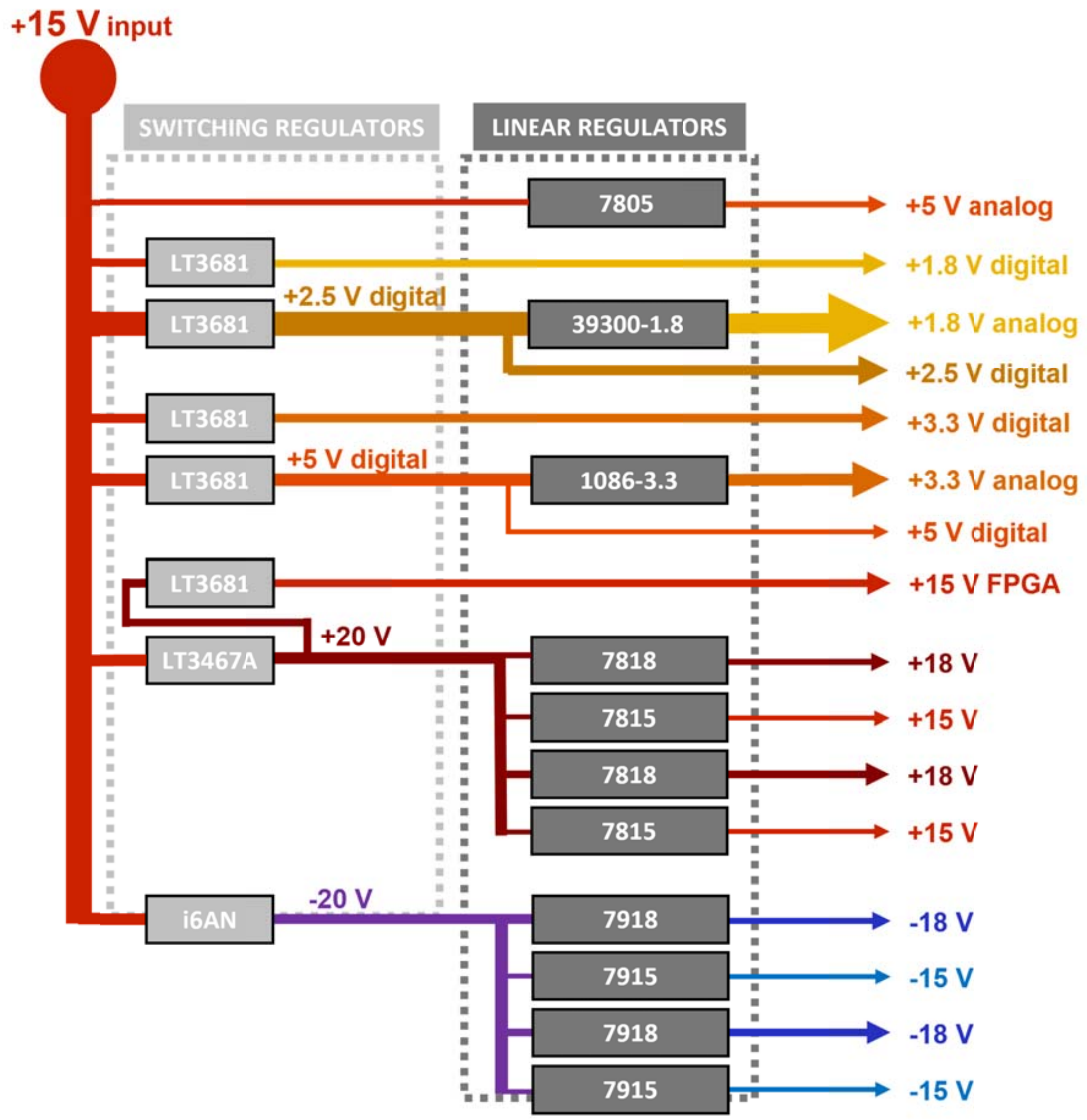

FIG. 6. MCFS power supply overview. The power supply has separate ±15 V for the slow ADC's and DAC's, and the two ±18 V supplies power the input and output amps. The switching frequencies are 600 kHz (1.8 V and 2.5 V digital), 800 kHz (3.3 V, 5 V, and +15 V digital), 2.1 MHz (+20 V), and 380 kHz (-20 V).

Assembly notes

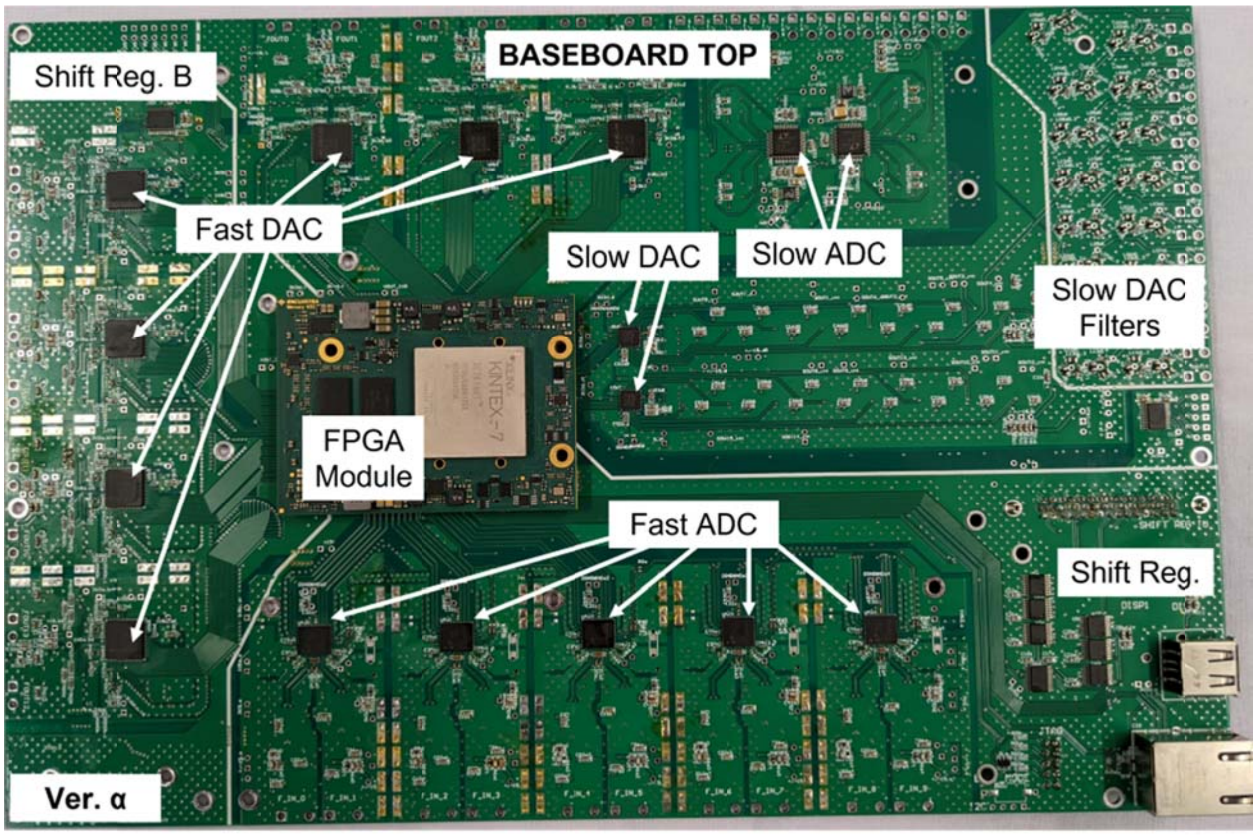

FIG. 7: Top of assembled baseboard. The top layer of the baseboard has most of the digital components, including the ADC's and DAC's, and the connectors for the FPGA module. When mounted in the chassis, this side is close to a vertical heat spreader on the side of the box that passively heatsinks the board, especially the FPGA package.

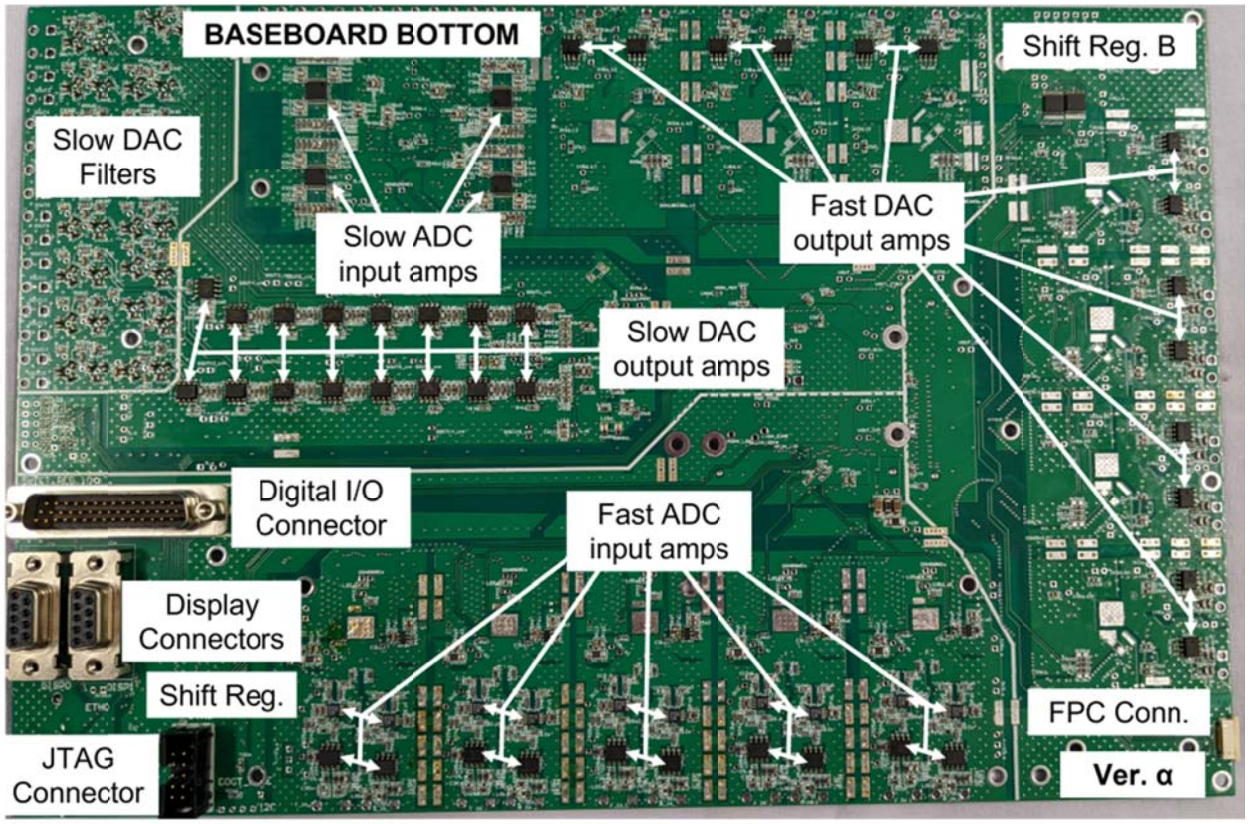

FIG. 8: Bottom of assembled baseboard. The bottom layer of the board has most of the analog components, including many passive components to adjust gains and frequency responses, as well as most of the connectors. These are accessible when the board is mounted in the chassis. The baseboard includes an FPC connector for potential expansion, e.g., to add a future daughterboard with additional slow ADC channels, and the unbuffered FPGA lines that control the shift registers have accessible connections.

We mount the baseboard in a 17” x 10” x 5” aluminum chassis and connect signals with coaxial cable to BNC bulkhead connectors on the front and back of the chassis, and on a side cover. Digital inputs and outputs from the shift-registers are connected to BNC connectors on a side cover via a DB44 connector on the baseboard (Fig. 8). The power supply board and the linear regulators are mounted on a 13” long section of 1.5” angle aluminum. Both circuit boards are mounted vertically in the chassis to dissipate heat and a 2”×2”×1/4” aluminum block with a small air gap thermally connects the FPGA package in Fig. 7 to a 4”×4” chassis-mounted heat spreader. This passive cooling maintains the FPGA internal temperature near 66°C.

USB, Ethernet, Multi-Gigabit Transceivers

The Mercury+ KX2 FPGA module has a USB UART and two Ethernet PHY’s. On the baseboard, we connected the USB UART to a USB-A connector and one Ethernet PHY to an RJ45 connector. This project does not use USB or Ethernet ports. Also unused in this design are several multi-gigabit (PCIe capable) transceivers on connector B, which have 4 reference clock input pairs, 8 receiver pairs and 8 transceiver pairs.

Acknowledgements

We gratefully acknowledge many suggestions from Avrum Warshawsky, contributions of Lam Tran, helpful conversations with Marco Pomponio, and financial support from the National Science Foundation.

References

1. Schussheim D, Gibble K. A many-channel FPGA control system (in preparation).
2. Leibrandt DR, Heidecker J. An open source digital servo for atomic, molecular, and optical physics experiments. Rev Sci Instrum. 2015;86(12):123115. doi:10.1063/1.4938282
3. The power supply board design files are available at https://bit.ly/3LmEj9Y.
4. The MCFS design files are available online at https://github.com/ GibbleLab/FPGA.
5. FT81x Series Programmers Guide. Published online September 25, 2015. https://www.ftdichip.com/Support/Documents/ProgramGuides/FT81X_Series_Programmer_Guide.pdf
6. UG953: Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide. Published online June 3, 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug953-vivado-7series-libraries.pdf
7. UG471: 7 Series FPGAs SelectIO Resources. Published online May 8, 2018. https://www.xilinx.com/support/documentation/user_guides/ug471_7Series_SelectIO.pdf
8. PG261: System Integrated Logic Analyzer v1.1. Published online February 4, 2021. https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/system_ila/v1_1/pg261-system-ila.pdf
9. Page 148 of [7].
10. UG908: Vivado Design Suite User Guide Programming and Debugging. Published online June 3, 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug908-vivado-programming-debugging.pdf
11. UG936: Vivado Design Suite Tutorial Programming and Debugging. Published online June 24, 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug936-vivado-tutorial-programming-debugging.pdf

12. UG905: Vivado Design Suite User Guide Hierarchical Design. Published online June 3, 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug905-vivado-hierarchical-design.pdf

13. UG946: Vivado Design Suite Tutorial Hierarchical Design. Published online April 2, 2014. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug946-vivado-hierarchical-design-tutorial.pdf

14. i6AN Series: 75W, 9 to 40V Input Non-Isolated DC-DC Buck Converter with Negative Output. Accessed September 5, 2021. https://product.tdk.com/en/system/files?file=dam/doc/product/power/switching-power/dc-dc-converter/catalog/i6an_e.pdf