Moku Cloud Compile
A Getting Started Guide

Moku Cloud Compile is an innovative feature available on Moku:Pro, Moku:Lab, and Moku:Go. The Moku family of test and measurement tools contain FPGA-based instruments and Moku Cloud Compile allows you to deploy custom VHDL code to a Moku. This code can provide custom features and interact with the existing instruments to unlock new and unique instrumentation only possible due Moku Instrument-on-Chip architecture.

This tutorial will guide show you how to create a Moku Cloud Compile account through to coding and deployment of some VHDL examples. By the end of this guide, you will have the fundamental knowledge to compile and deploy custom code to your Moku. This note uses Moku:Pro, though all these examples are also useable on Moku:Lab and Moku:Go.

LIQUID INSTRUMENTS

Table of Contents

Prerequisites

Moku device with MiM and MCC. This application note refers to Moku:Pro; but all the examples can be deployed on Moku:Lab or Moku:Go.

Multi-Instrument-Mode (MiM)

Moku Cloud Compile (MCC)

If your Moku does not have MiM or MCC, please contact us at support@liquidinstruments to enquire about evaluations and upgrades.

Goals and more information

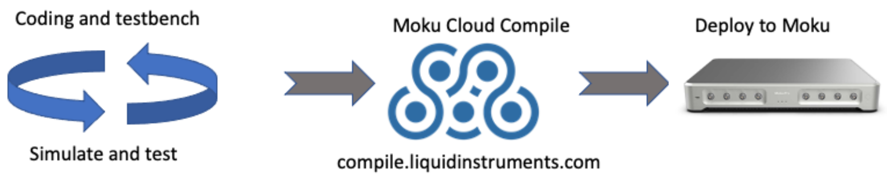
By the end of this tutorial, you will have created a Moku Cloud Compile account, coded and deployed several simple VHDL custom functions, and have the knowledge to take your use of MCC further. There is additional getting started information online: https://www.liquidinstruments.com/blog/2022/09/02/starting-with-moku-cloud-compile/

Overview

The Moku Cloud Compile tool enables you to design custom processing and features for implementation on Moku platforms. Compared to CPU and application specific integrated circuits (ASIC) based DSP approaches, FPGA platforms provide near ASIC-level latency and performance while still being software programmable, more like a traditional CPU.

While there are many widely used software languages that can be employed to write software for CPU based designs, FPGA programming is generally limited to VHDL or Verilog. These typically require a large and complex local toolchain installation. The platforms available for deploying VHDL code are usually limited to evaluation boards from FPGA vendors or a variety of limited functionality, open-source hardware boards. Moku Cloud Compile provides an integrated, cloud based VHDL compiler and combines with the reliability of Moku hardware.

Moku:Pro with Moku Cloud Compile addresses the need for a high-performance laboratory instrument with research-grade hardware and custom processing without the overhead of traditional FPGA design software. MCC compiles your custom VHDL in the cloud and delivers a package ready to deploy to any MCC-enabled Moku.



Multi-instrument Mode and Cloud Compile

Multi-instrument Mode (MiM) allows multiple instruments to be deployed and operate simultaneously. At the highest level, MiM presents four slots representing four partitions of the FPGA on Moku:Pro, and two on Moku:Go and Moku:Lab. You can deploy a flexible arrangement of instruments into these slots. Figure 1 shows the MiM interface with an Oscilloscope deployed in Slot 1 and a Spectrum Analyzer deployed in Slot 2, while Slots 3 and 4 remain to be filled. The available instruments include a Phasemeter, Laser Lock Box, Data Logger, Digital Filter Box, PID Controller, Oscilloscope, Spectrum Analyzer, Lock-in Amplifier, Waveform Generator, Frequency Response Analyzer, FIR Filter Builder, Arbitrary Waveform Generator, Logic Analyzer, and Cloud Compile.

It is the Moku Cloud Compile instrument, which occupies Slot 4 in Figure 2, where you can deploy your compiled designs. MiM thus enables your designs to interact with the Moku instruments in addition to the ADC and DAC inputs and outputs.

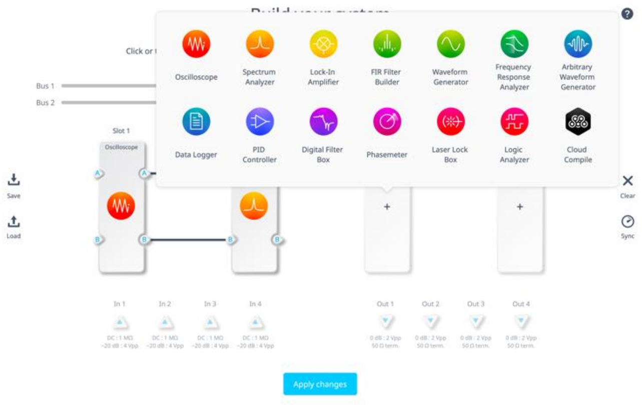


Figure 1: Building a MiM system

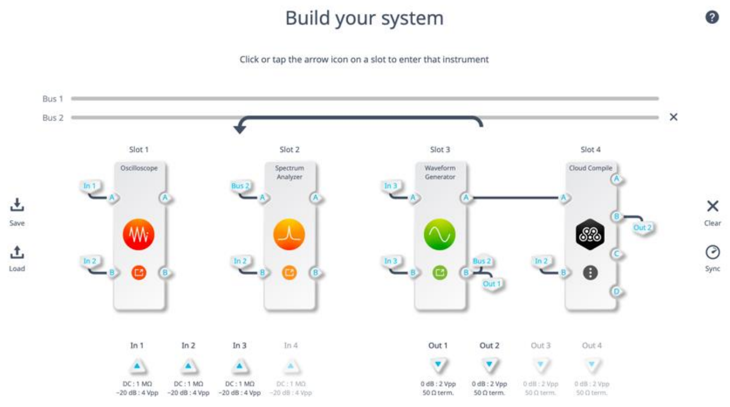
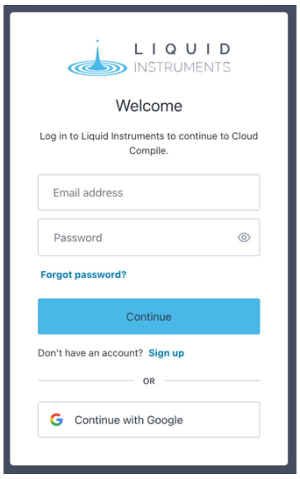


Figure 2: MiM showing available instruments

Setting up a Moku Cloud Compile account

Before you can compile or deploy code to a Moku, you need an online account. This is a simple process:



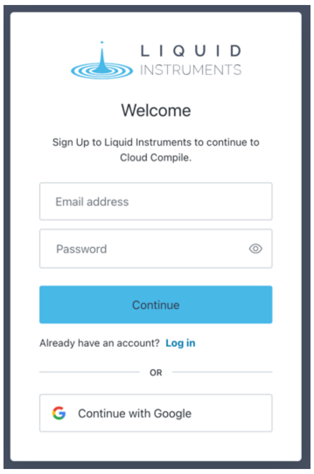
LIQUID INSTRUMENTS

Welcome

Log in to Liquid Instruments to continue to Cloud Compile.

⇒ You can setup an MCC user account at: compile.liquidinstruments.com

If you're a first-time user, you'll need to select “Sign up.” If you're an existing user, log in with your username or email address, then enter your password.



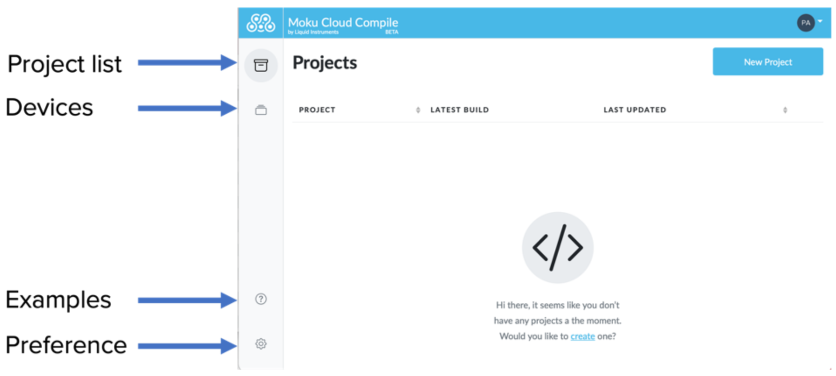
LIQUID INSTRUMENTS

Welcome

Sign Up to Liquid Instruments to continue to Cloud Compile.

The Sign-up page requires only a valid email address and user-defined password.

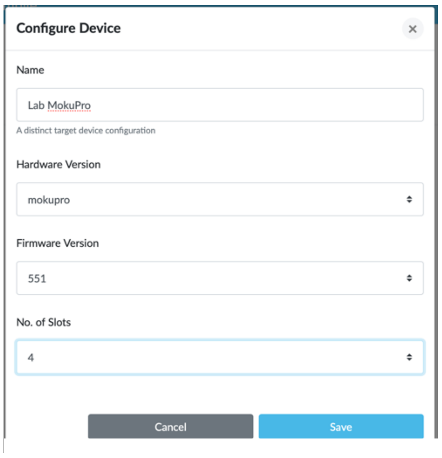
Once signed up and logged in, you will see the Projects page, which initially will be empty, as shown in Figure 3.



Hi there, it seems like you don't have any projects a the moment. Would you like to create one?

Figure 3: New project menu

Before you write your first VHDL example, you should configure a device to target. Select the Devices tab and configure as shown in Figure 4. Choose a convenient name and then select Hardware version, Firmware version and No. of slots as shown. You can determine the firmware version of your Moku via the desktop app (right-click on the Moku icon -> device info) or on the iPad app (touch and hold the Moku icon at the “Select your device” screen).


Configure Device

Name: Lab MokuPro

A distinct target device configuration

Hardware Version: mokupro

Firmware Version: 551

No. of Slots: 4

Figure 4: Configure new Moku device

VHDL example #1: Routing inputs to outputs

Enter the VHDL code

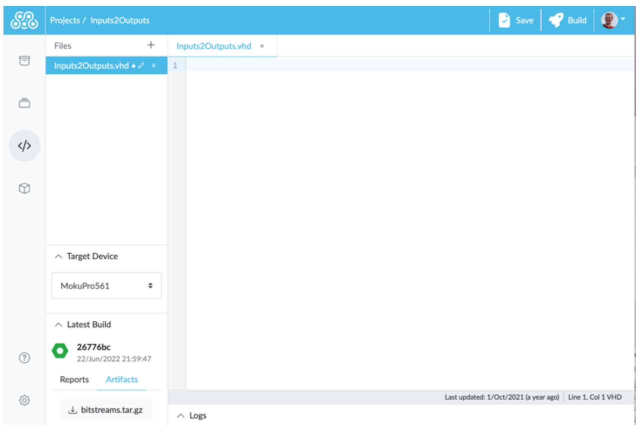
Now that your Moku Cloud Compile (MCC) account is set up and you are familiar with the interface, you will write, compile, and deploy the most basic instrument. Your simple first instrument will simply take the Moku Cloud Compile slot's input signals and connect them directly to the outputs.

To get started, select 'New Project' and enter an appropriate name. This example is "Inputs2Outputs".

```
architecture Behavioural of CustomWrapper is

begin

    OutputA <= InputA;

    OutputB <= InputB;

end architecture;
```

Figure 5: Full code for example #1; Input2Outputs.vhd

The code for this first example is shown in Figure 5**Error! Reference source not found.** This code must define the architecture of the entity '`CustomWrapper`'. '`CustomWrapper`' is the template to which the MCC design must conform. It provides the basic I/O port definitions of the custom MCC instrument. The full definition is given in the online documentation : https://compile.liquidinstruments.com/docs/wrapper.html.



Target Device

MokuPro561

Latest Build

26776bc

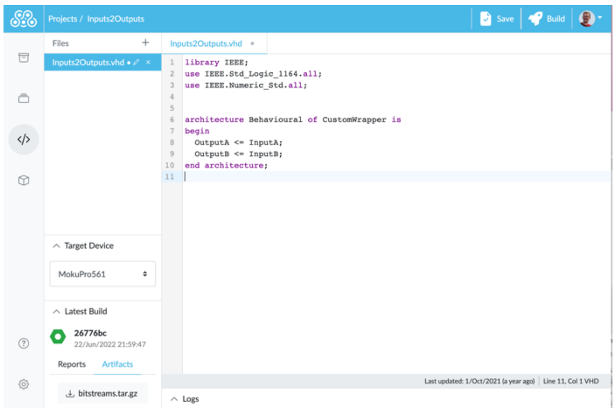
22/Jun/2022 21:59:47

Last updated: 1/Oct/2021 (a year ago) | Line 1, Col 1 VHD

Figure 6: New file in editor

With the "Inputs2Outputs" project open in the editor, create a new file named "Inputs2Outputs.vhd", shown in Figure 6.

Enter the text of Figure 5. It should look like Figure 7.

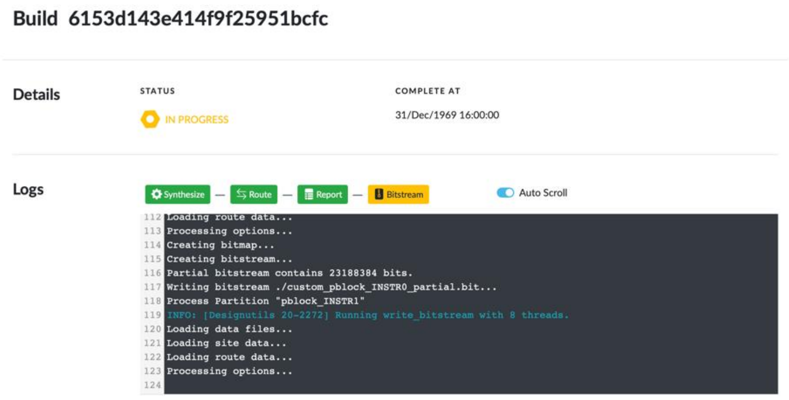


```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;


architecture Behavioural of CustomWrapper is
begin
  OutputA <= InputA;
  OutputB <= InputB;
end architecture;
```

Figure 7: VHDL code in editor

Build the code

Select the “Target device”, ”Save” and then “Build”. The code will then be submitted to the MCC server, and you can select the Build tab to observe the process. This will take several minutes to complete.


Build 6153d143e414f9f25951bcfc

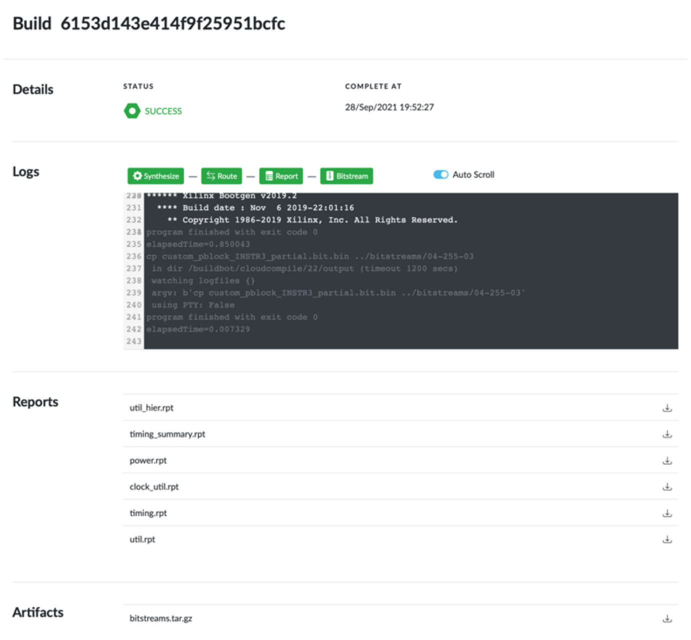
Details STATUS: IN PROGRESS — COMPLETE AT: 31/Dec/1969 16:00:00

```
Loading route data...
Processing options...
Creating bitmap...
Creating bitstream...
Partial bitstream contains 23188384 bits.
Writing bitstream ./custom_pblock_INSTR0_partial.bit...
Process Partition "pblock_INSTR1"
INFO: [Designutils 20-2272] Running write_bitstream with 8 threads.
Loading data files...
Loading site data...
Loading route data...
Processing options...
```

Figure 8: Build underway

Once the code compilation is complete, the “Synthesize”, “Route”, Report” and “Bitstream” should all be green. At the very bottom of this screen is the bitstream, or artifact, ‘bitstream.tar.gz’. The file “bitstream.tar.gz” can then be downloaded to your local PC by clicking on the small download icon. Do not unzip or untar this file.

It is likely you will see many compiler or synthesizer warnings; these can mostly be disregarded for our purpose. However, any errors would need attention as they will halt the build process.



Build 6153d143e414f9f25951bcfc

Details

STATUS	COMPLETE AT
SUCCESS	28/Sep/2021 19:52:27

Logs

```
****** Xilinx Bootgen v2019.2
  **** Build date : Nov  6 2019-22:01:16
    ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
program finished with exit code 0
elapsedTime=0.850043
cp custom_pblock_INSTR3_partial.bit.bin ../bitstreams/04-255-03
 in dir /buildbot/cloudcompile/22/output (timeout 1200 secs)
 watching logfiles {}
 argv: b'cp custom_pblock_INSTR3_partial.bit.bin ../bitstreams/04-255-03'
 using PTY: False
program finished with exit code 0
elapsedTime=0.007329
```

Reports

util_hier.rpt
timing_summary.rpt
power.rpt
clock_util.rpt
timing.rpt
util.rpt

Artifacts

bitstreams.tar.gz

Figure 9: Successful build

Deploy the code

Now using the Moku app, select MiM and then configure the slots and I/O as shown in Figure 10. This will prepare the Moku:Pro for the Moku Cloud Compile bitstream.

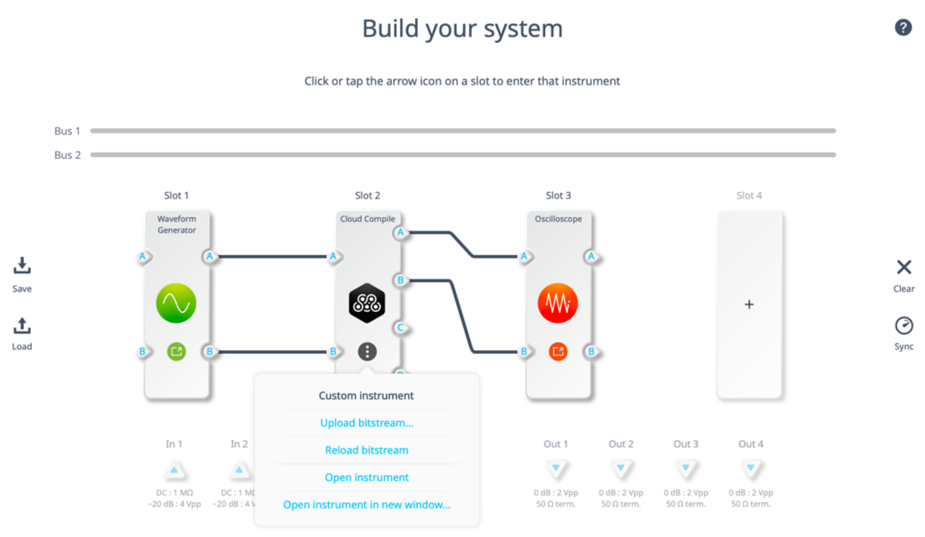
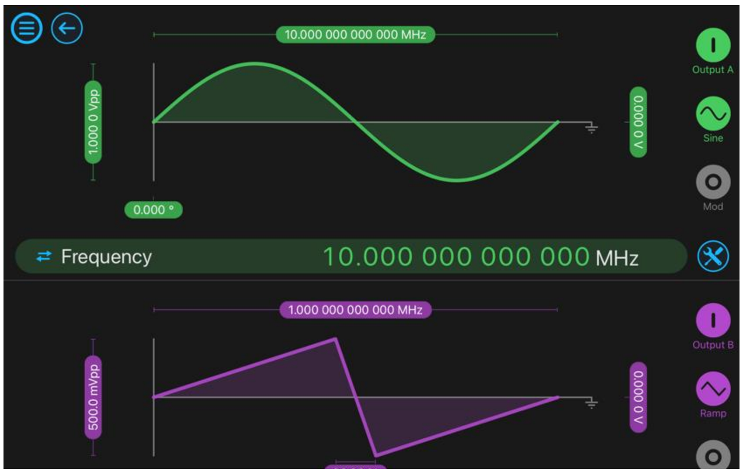


Figure 10: MiM slot configuration for example 1

To deploy the bitstream, tap the Moku Cloud Compile instrument and select “Upload bitstream” as seen in Figure 10. Browse and select the file “bitstream.tar.gz” previously downloaded to your local PC.

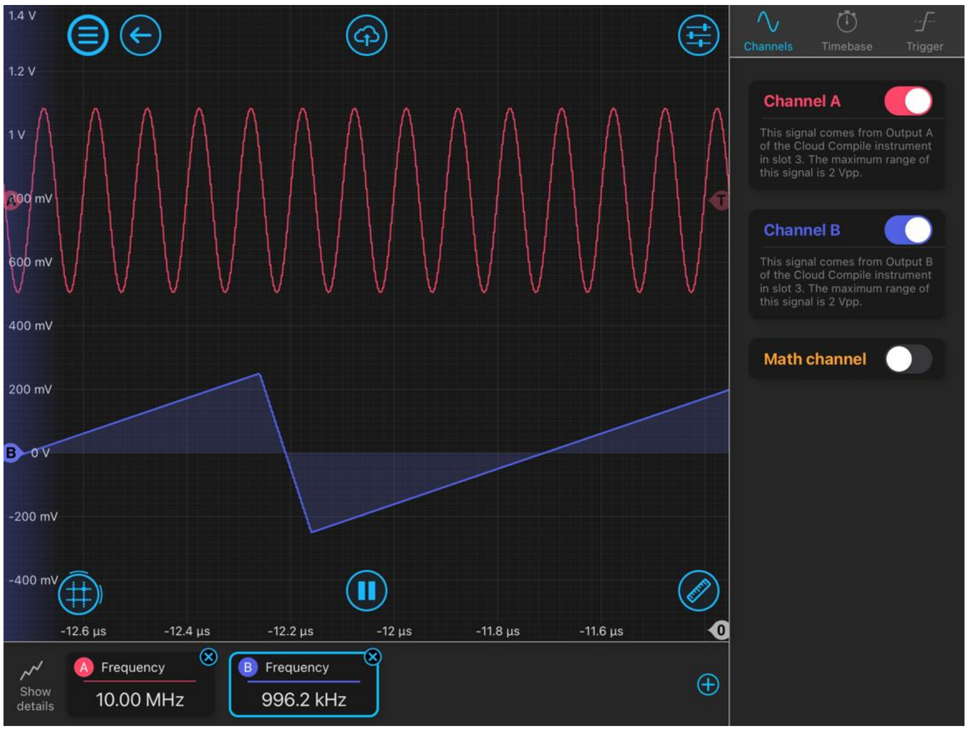
To test your new custom instrument, configure the Slot 1 Waveform Generator with a 10 MHz sine wave on Output A and a 1 MHz ramp wave on Output B. This then passes through our simple instrument. By deploying the



Frequency 10.000 000 000 000 MHz

Figure 11: Slot 2 Waveform Generator setup

Oscilloscope in Slot 3, we can observe the correct “pass through” operation of the MCC instrument deployed between them in Slot 2.



Channel A

This signal comes from Output A of the Cloud Compile instrument in slot 3. The maximum range of this signal is 2 Vpp.

Channel B

This signal comes from Output B of the Cloud Compile instrument in slot 3. The maximum range of this signal is 2 Vpp.

Math channel

A Frequency 10.00 MHz

B Frequency 996.2 kHz

Figure 12: Slot 4 Oscilloscope confirmation of MCC operation

VHDL example #2: subtracting/adding inputs

Example 1 illustrated in detail the steps required to compile and deploy a simple VHDL example.

```
architecture Behavioural of CustomWrapper
is
begin
    OutputA <= InputA + InputB;
    OutputB <= InputA - InputB;
end architecture;
```

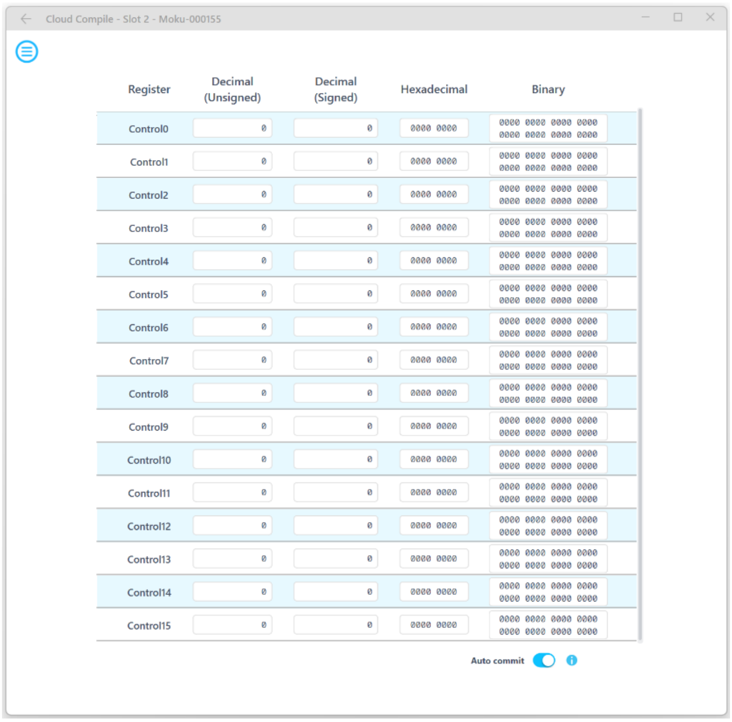
Example 2 shows the VHDL to add and subtract inputs. While this is also a basic example, it has many practical scenarios in the test and measurement world. Summing or subtracting signals in the digital domain without the need for digital to analog conversions and an external analog summing amplifier simplifies many applications.

Users are encouraged to take this example and repeat the compile and deploy steps outlined in example #1. Verification might, like the first example, consist of using MiM with a Waveform Generator in Slot 1, Moku Cloud Compile in Slot 2, and an Oscilloscope in Slot 3. Visual confirmation of correct operation could consist of adding and subtracting sine waves that are alternatively in phase and 180 degrees out of phase.

VHDL example #3: Reading Control Registers

Example 3 introduces the use of the control registers. Access the registers on the Moku: App interface for the MCC instrument once deployed by clicking the MCC instrument and selecting Open Instrument. Users can change these 32-bit control registers in the app and use them in the VHDL code. In example 3, the Least Significant Bit (LSB) of Control1 is used as an enable; when set to '1', InputA and InputB are passed to OutputA and OutputB respectively. When Control1 LSB is set to '0', the outputs are set to 0.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;

architecture Behavioural of CustomWrapper is
begin
    with Control1(0) select
      OutputA <= InputA when '1',
                 (OTHERS => '0') when others;

    with Control1(1) select
      OutputB <= InputB when '1',
               (OTHERS => '0') when others;
end architecture;
```



Register	Decimal (Unsigned)	Decimal (Signed)	Hexadecimal	Binary
Control0	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control1	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control2	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control3	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control4	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control5	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control6	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control7	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control8	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control9	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control10	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control11	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control12	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control13	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control14	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Control15	0	0	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000

Figure 13: Control registers

VHDL example #4: Scaling and offset inputs / DSP slice

This example is slightly more complex than the first three examples.

The code references the VHDL library `Moku.Support`, a Liquid Instruments library. Further information on its contents is available on the Help section of the MCC website. Specifically, the entity `ScaleOffset` is named. This entity wraps a specific hardware block that is dedicated to performing a multiply and add function. This entity is instantiated as block named 'DSP' and used to provide math function of :

Output = (Input * Scale) + Offset

```
library Moku;
use Moku.Support.ScaleOffset;

architecture Behavioural of CustomWrapper is
begin
    -- Z = X * Scale + Offset
    -- Clips Z to min/max (prevents over/underflow)
    -- Includes rounding
    -- One Clock Cycle Delay
    DSP: ScaleOffset
        port map (
            Clk => Clk,
            Reset => Reset,
            X => InputA,
            Scale => signed(Control1(15 downto 0)),
            Offset => signed(Control2(15 downto 0)),
            Z => OutputA,
            Valid => '1',
            OutValid => open
        );
end architecture;
```

In this example, both scale and offset are 16-bit, signed 2's complement numbers. Scale is mapped from -1 to +1. Additionally, the output, Z, is clipped to prevent over or under flowing of the math operation. There is further information on `ScaleOffset`, including how to scale beyond -1 to +1 in the online documentation: https://compile.liquidinstruments.com/docs/support.html#scaleoffset

Another feature of this example is the use of Control1 and Control2 as shown in example 3. Access these control registers from the Moku: application to provide input parameters to the MCC custom instrument, in this case changing the signal scaling and offset without rebuilding the design.

VHDL example #5a: Output limit setting

Example #5a provides a way to clip an output signal to an upper (or lower) limit.

Just as in example #3, this example uses the `Moku.Support` library, this time using the '`clip`' function. The OutputA is now assigned to the clipped, lower 9 bits of InputA. This provides 2^power clipping function, i.e., the signal is clipped to the range 0 - 2^9.

```
library IEEE;
use IEEE.Numeric_Std.all;
library Moku;
use Moku.Support.clip;


architecture Behavioural of CustomWrapper is
begin
    OutputA <= resize(clip(InputA, 8, 0), 16);
end architecture;
```

VHDL example #5b: Flexible output limit

Example #5b provides a way to clip an output signal to any upper and lower limit.

Whereas example #5a would provide a crude but fast clipping to a power of 2, this example provides a more generalized clipping function. The OutputA is assigned to InputA but clipped to the range of +2387 and -7462

```
library IEEE;
use IEEE.Numeric_Std.all;

library Moku;
use Moku.Support.clip_val;

architecture Behavioural of CustomWrapper is
  begin
    OutputA <= clip_val(InputA, -7462, 2387);
end architecture;
```

VHDL example #6: PWM from analog input

Example 6 is more complicated. It generates a pulse width modulation (PWM) signal from the analog InputA. It comprises two files, `Counter.vhdl` and `pwm.vhdl`. Users enter these as separate files in the MCC file editor before building together. This example is specific to Moku:Pro and its 312.5 MHz clock, but can be adapted for Moku:Lab and Moku:Go with 125 MHz and 31.25 MHz clocks, respectively. For more information on the differences between Moku Cloud Compile on Moku:Go, Moku:Lab, and Moku:Pro, read the datasheet here.

```vhdl
-- counter.vhdl

library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

--Output Strobe every 2^EXPONENT / INCREMENT Input Strobes
--Will quantize to round integers but maintains overflow, so
--will average out over time, but will have a +-1 cycle jitter.
entity Counter is
        generic (
                EXPONENT : positive := 8;
                PHASE90 : boolean := false
        );
        port (
                Clk : in std_logic;
                Reset : in std_logic;
                Enable : in std_logic;
                Increment : in unsigned;
                Strobe : out std_logic
        );
end entity;

architecture Behavioural of Counter is
        signal Count : unsigned(EXPONENT downto 0);
begin

        assert Increment'length <= Count'length severity FAILURE;

        process(Clk) is
        begin
                if rising_edge(Clk) then
                        if Reset = '1' then
                                Count <= (others => '0');
                                if PHASE90 then
                                        Count(EXPONENT - 1) <= '1';
                                end if;
                        elsif Enable = '1' then
                                --Trim the MSB but allow overflow into it.
                                --This gives a single Clk cycle output pulse on Strobe.
                                Count <= resize(Count(Count'left-1 downto 0), Count'length) + Increment;
                        else
                                Count(Count'left) <= '0';
                        end if;
                end if;
        end process;

        Strobe <= Count(Count'left);

end architecture;
```

```
-- pwm.vhdl
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;

library Moku;
use Moku.Support.ScaleOffset;
use Moku.Support.clip;

architecture Behavioural of CustomWrapper is
    constant HI_LVL : signed(15 downto 0) := x"7FFF";
    constant LO_LVL : signed(15 downto 0) := x"0000";
    signal Value: signed(12 downto 0);
    signal Count : unsigned(12 downto 0);
    signal Pulse50kHz : std_logic;
    signal Pulse : std_logic;
    signal OutA : std_logic;

begin
    INPUT_SCALE: ScaleOffset
        port map (
            Clk => Clk,
            Reset => Reset,
            X => InputA & "0",
            Scale => signed(Control1(15 downto 0)), -- For internal bus 2Vpp; setting of 0x0200 maps well
            Offset => signed(Control2(15 downto 0)),  -- For internal bus of 2Vpp, offset of 0x0400 works well
for +/-1 v internal bus
            Z => Value,
            Valid => Pulse50kHz,
            OutValid => open
        );

    OSC: entity WORK.Counter
        generic map (22)  -- ~5 kHz from 312.5MHz
        port map (Clk, Reset, '1', to_unsigned(67, 8), Pulse50kHz);

    OSC2: entity WORK.Counter
        generic map (11)  --5kHz/2048 approx 10 MHz
        port map (Clk, Pulse50kHz, '1', to_unsigned(65, 9), Pulse);

    process(Clk) is
    begin
        if rising_edge(Clk) then
            if Pulse50kHz = '1' then
                Count <= resize(unsigned(clip(Value, 11, 0)), Count'length);
            elsif Pulse = '1' and Count /= 0 then
                Count <= Count - 1;
            end if;
        end if;
    end process;


    OutputA <= HI_LVL when Count /= 0 else LO_LVL;


end architecture;
```

Summary

This application note presented some of the advantages and benefits of Moku Cloud Compile when operating as part of Multi-instrument Mode on a Moku:Pro. The entire process was covered, from entering the simplest possible VHDL code in example #1 through building, deploying, and confirming expected operation.

Further example code was presented and briefly explained, inviting you to experiment with the possibilities opened by MCC.

Further MCC designs including more complex math operations, custom trigger modes, and other applications can be developed from these basics.

Questions or comments?

Please contact us at support@liquidinstruments.com