Einführung in Xilinx Vivado Simulator

Inhaltsverzeichnis

Versionen

Version	Status	Datum	Verantwortlicher	Änderungsgrund
2019.a	Start	30.7.2019	L. Arato	Start des Dokumentes auf der Basis von «Einführung ModelSim Simulator»
2019.b	Release	3.10.2019	L. Arato	Review bis und mit Kapitel 6. Überarbeitung von Kapitel 7 fehlt noch!
2020.a	Release	14.09.2020	L. Arato	Anpassung an OST
2020 b	Release	7.10.2020	L. Arato	Referenz zu EuR-III entfernet

1 Einleitung

1.1 Zweck des Dokuments

Diese Einführung soll Studenten und anderen interessierten Personen helfen, möglichst schnell und effizient den «Behavioral Simulator» als Teil der Vivado Entwicklungsumgebung von Xilinx zu nutzen.

1.2 Gültigkeit des Dokuments

Die Ausführungen gelten sowohl für die kostenlose Webedition Ausführung, wie auch für die Lizenzierte Vollversion. Dort wo Unterschiede bestehen, wird darauf explizit hingewiesen.

1.3 Begriffsbestimmungen und Abkürzungen

FPGA	Field Programmable Gate Array, ein programmierbarer Logikbaustein.
Xilinx	Xilinx Corporation als Hersteller von FPGAs
VHDL	«Very High-Speed Hardware Description Language»
ZYBO	Marketing Name von Digilent für ein Board mit Xilinx Zynq XC7010 FPGA
Zynq	Marketing Name von Xilinx für die Virtex-7000 Serie FPGAs

1.4 Zusammenhang mit anderen Dokumenten

Dieses Dokument ist die Ergänzung zum Dokument «Einführung in Xilinx Vivado» und das «VHDL Skript».

Folgende begleitende Dokumente sind geplant oder bereits in Arbeit:

- «VHDL Skript»
- «Einführung in Xilinx Vivado»
- «Smart Testbench Design»
- «VHDL Design Guidelines»

Weitere unterstützende Literatur:

- «The Zynq Book»
- «ZYBO Reference Manual»

2 Xilinx Vivado Installation

Für das Download der Vivado Software, Installation und Lizenzierung wird auf das Dokument «Vivado Installation und Lizenzierung.pdf» verwiesen.

3 Erste Simulation: Three_Bit_Counter

Anhand eines sehr einfachen Beispiels werden die Funktionen aufgezeigt und erklärt.

3.1 VHDL Source Code

Die Target-Funktion ist ein sehr einfacher 3-Bit Zähler mit einem Enable-Signal (Three_Bit_Counter.vhd):

```
35    LIBRARY IEEE;
36    USE IEEE.STD_LOGIC_1164.ALL;
37    USE IEEE.NUMERIC_STD.ALL;
28
39    PACKAGE three_bit_counter_pkg IS
40        COMPONENT three_bit_counter IS
41            PORT (
42                clk          : IN  STD_LOGIC;
43                enable       : IN  STD_LOGIC;
44                count        : OUT UNSIGNED (2 DOWNTO 0)
45            );
46        END COMPONENT three_bit_counter;
47    END PACKAGE three_bit_counter_pkg;
49    ----------------------------------------------------------------------
50
51    LIBRARY IEEE;
52    USE IEEE.STD_LOGIC_1164.ALL;
53    USE IEEE.NUMERIC_STD.ALL;
54
55    ENTITY three_bit_counter IS
56            PORT (
57                clk          : IN  STD_LOGIC;
58                enable       : IN  STD_LOGIC;
59                count        : OUT UNSIGNED (2 DOWNTO 0)
60            );
61    END ENTITY three_bit_counter;
63    ----------------------------------------------------------------------
64
65    ARCHITECTURE behavior OF three_bit_counter IS
66
67    SIGNAL curr_count : UNSIGNED (2 DOWNTO 0) := (OTHERS => '1');
68
69    BEGIN
70
71        reg_proc : PROCESS (clk)
72        BEGIN
73            IF rising_edge(clk) THEN
74                IF enable = '1' THEN
75                    curr_count <= curr_count + 1 AFTER 1 ns;
76                END IF;
77            END IF;
78        END PROCESS reg_proc;
79
80        --  Output Function
81        count <= curr_count;
82
83    END ARCHITECTURE behavior;
```

3.2 Erklärungen zum «Three_Bit_Counter» VHDL Code

Zeilen 35 – 37: Package Deklaration verwendeten Bibliotheken
Wir verwenden für die Schnittstellen nach aussen Signale vom Typ «`STD_LOGIC`» und «`UNSIGNED`».
Diese Typen sind in der Bibliothek (`LIBRARY`) des IEEE Standards definiert, und zwar in den Paketen «`STD_LOGIC_1164`» und «`NUMERIC_STD`».

Zeilen 39 – 47: Definition der Komponente
Die Komponenten-Definition (`COMPONENT`) könnte auch in der nächst höheren Hierarchiestufe stehen, denn erst dort wird sie zur Instanziierung dieses Moduls benötigt. Da jedoch dieses Modul (three_bit_counter) in mindestens zwei höheren Modulen verwendet wird (VHDL Design und Testbench) ist es immer von Vorteil, wenn man die Komponenten-Definition beim Modul selbst behält, und über ein «`PACKAGE`» den anderen Modulen zur Verfügung stellt.

Zeilen 51 – 53: Entity und Architecture Deklaration verwendeter Bibliotheken
Dies muss für die `ENTITY` und `ARCHITECTURE` an dieser Stelle wiederholt werden … die Anweisungen in Zeilen 35 bis 37 gelten nur für das `PACKAGE`.

Zeilen 56 – 62: Definition der ENTITY
Es mag zwar wenig sinnvoll erscheinen, dass man jedes Mal praktisch identisch die `COMPONENT` und die `ENTITY` definieren muss, aber in VHDL ist es halt so. Man kann es zum Teil mit der Definition einer Funktion in Software wie C vergleichen, wenn der Funktions-Aufruf mit allen notwendigen Parametern nicht nur im .c File definiert ist, sondern nochmals identisch (aber ohne Funktionsinhalt) im .h File.

Zeile 66: Definition der Architektur
Die Architektur kann fast jeden beliebigen Namen tragen. Da man aber zu jeder Entity mehrere Architekturen definieren kann, ist es sinnvoll der Architektur immer einen aussagekräftigen, sinnvollen Namen zu geben.
In diesem Zusammenhang bedeutet der Name «`behavior`» dass es sich um eine Architektur handelt die sich in allen Aspekten so verhält wie (später) die «richtige» Implementation, aber dass diese Architektur NICHT dafür vorgesehen ist, synthetisiert und implementiert zu werden (z.B. wegen dem «`AFTER 1 ns`»).
Eine Architektur die Synthetisiert werden kann, nennt man z.B. «`rtl`» oder «`struct`» oder «`ALTERA`».

Zeile 67: Signal-Definition
Das interne Signal wird benötigt, weil in VHDL ein Ausgangs-Signal nicht innerhalb des Moduls selbst wieder gelesen werden kann. Deshalb verwenden wir ein internes Signal für den Zähler, und kopieren dessen Wert kontinuierlich (in Zeile 81) auf den Ausgang.

Zeilen 69 – 83: Die Architektur

Zeile 71: Prozess-Deklaration und Sensitivity Liste
Der Prozess hat einen Namen («reg_proc») vor dem Doppelpunkt, und dann das Schlüsselwort «**PROCESS**».
Die Sensitivity-Liste in den Klammern nach dem Schlüsselwort «**PROCESS**» ist extrem wichtig für die Simulation. Der Prozess wird **nur dann** neu durchlaufen, wenn sich ein Wert in der Liste ändert. Wenn hier z.B. «clk» fehlen würde, würde während der Simulation **nichts** geschehen, und so würde auch «curr_count» nie aktualisiert … obwohl «clk» getaktet und «enable» aktiv wären!
Zeilen 72 – 78: Der registrierte Prozess
Bedingungen (`IF`) oder Schleifen (`LOOP`) können nur innerhalb eines Prozesses verwendet werden.

Zeile 75: Die eigentliche Zuweisung
Hier wird tatsächlich gezählt … allerdings jeweils erst mit einer Nanosekunde Verzögerung. Dies kann eine langsame Schaltung annähern, aber sie verhindert auch, dass diese Architektur exakt synthetisiert werden und in einem FPGA verwendet werden kann.

Zeilen 81: Kopieren des internen Zählerstandes auf den Ausgang
Da das Ausgangssignal «count» beim Zählen in Zeile 75 zur Verfügung steht, müssen wir ein internes Signal verwenden und dieses hier auf den Ausgang kopieren.

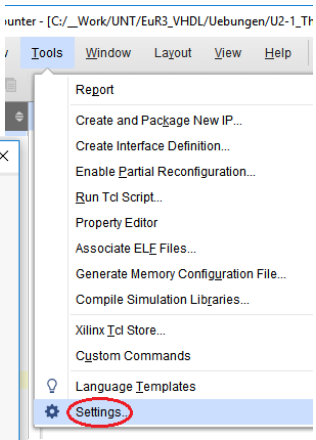
3.3 Start der Simulation in Vivado

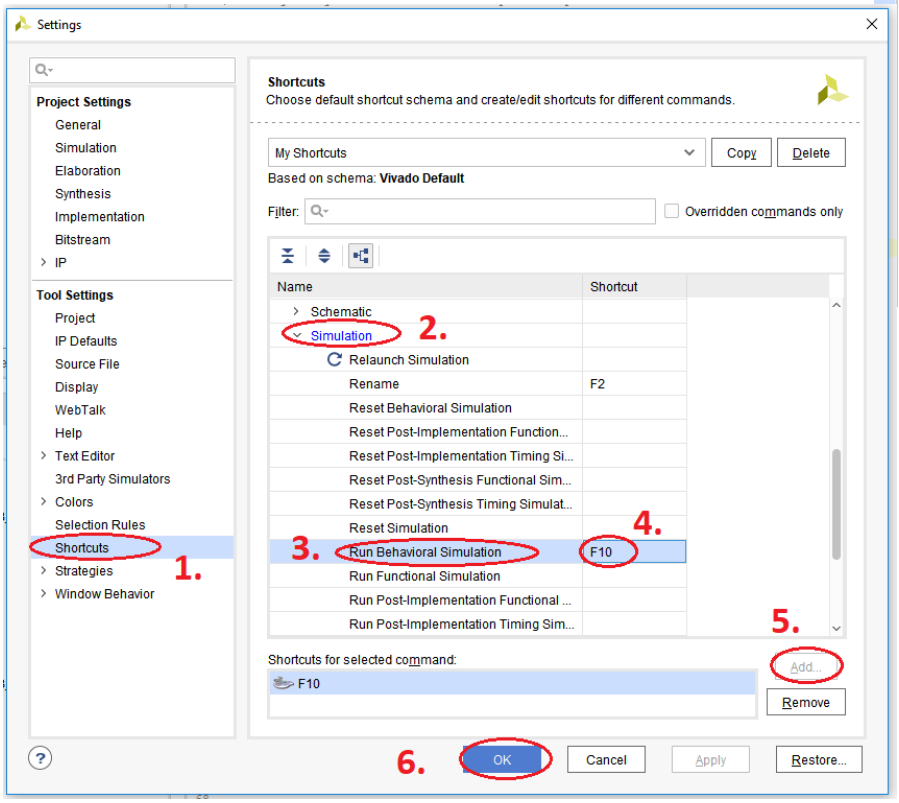
Man kann den Simulator in Vivado auf verschiedene Arten starten:

- Aus dem Menu am oberen Bildrand heraus: «Flow» - «Run Simulation» - «Run Behavioral Simulation»

- Im «Flow Navigator» (linker Bildrand): «Simulation» - «Run Simulation» - «Run Behavioral Simulation»

Da wir den Simulator sehr oft verwenden werden, lohnt sich die Konfiguration eines «Hot-Key» wie z.B. die Taste «F10». Dazu öffnen Sie das «Settings» Fenster aus dem Menu heraus mit «Tools» - «Settings …»





1. Scrollen Sie links zum Eintrag «Shortcuts»

2. Scrollen Sie im zentralen Fenster zu Simulation

3. Expandieren Sie «Simulation» damit Sie «Run Behav… Sim…» sehen

4. Klicken Sie auf das noch leere Feld des Shortcuts
5. Fügen Sie mit «Add…» eine neue Taste (z.B. Taste «F10») als Shortcut dazu
6. Schliessen Sie die Einstellungen mit «OK» ab.

3.4 Vorbereitung der Simulation (Compile, Analyze, Elaborate & Launch)

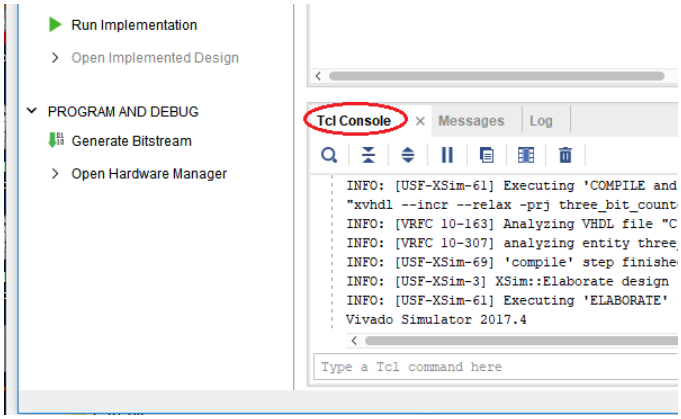
Nach dem Startbefehl geschieht sehr vieles automatisch im Hintergrund:

- Der Simulator inspiziert das zur Simulation ausgewählte Top-Level File und sucht sich die notwendigen «Include» Files selber zusammen
- Alle notwendigen Files werden kompiliert und analysiert (VHDL Code verstanden?)
- Das Design wird «elaboriert» … also in ein für die Simulation effizientes Format umgesetzt
- Als letztes wird die eigentliche Simulation gestartet, das Waveform Fenster geöffnet, und die Simulation für eine vorbestimmte Zeit laufen gelassen.

Dieser Ablauf kann im «Tcl Console» Fenster am Bildrand unten rechts beobachtet werden.

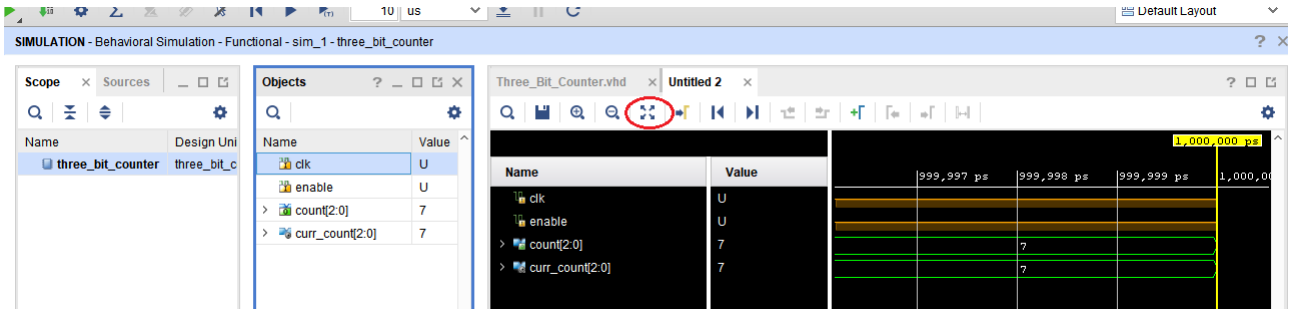
Wenn es während diesem Ablauf Fehler gegeben hat, z.B. wegen falscher Syntax, dann erscheint das auch in diesem Fenster.

Scrollen Sie dann einfach die Meldungen hinauf bis zum ersten Eintrag mit der Markierung «ERROR» am linken Rand … meist hilft die Meldung und Positionsangabe (File, Zeile) um den Fehler zu finden …



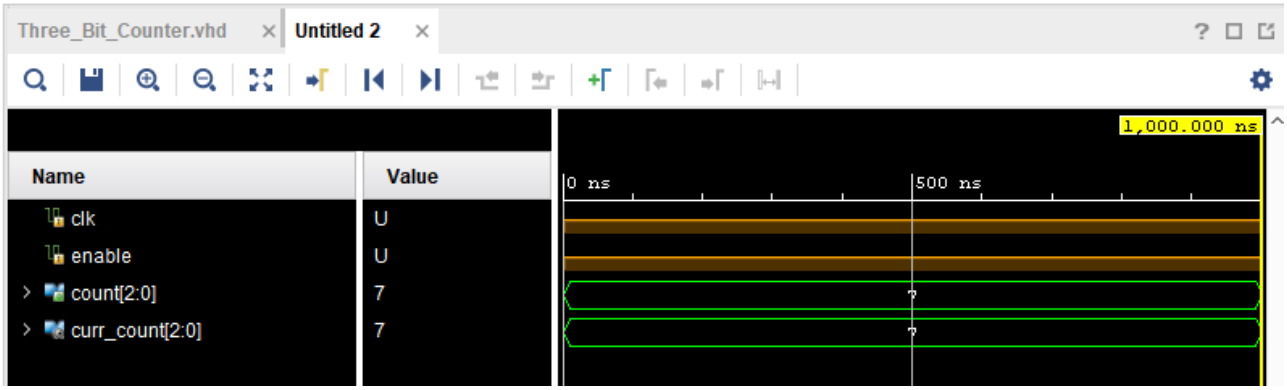
3.5 Simulation ohne Testbench

Wenn es bei der Vorbereitung der Simulation keinen Fehler gegeben hat, dann erscheint jetzt ein Fenster mit viel Schwarz und ein paar farbigen Linien … das ist das «Waveform» Fenster:



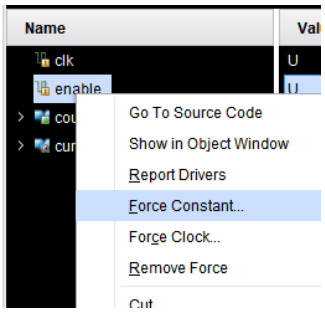
Sie sehen dabei die Signalverläufe der Top-Level Signale während den letzten paar Pico-Sekunden (!) der Simulation.

Klicken Sie auf den Knopf am oberen Rand der Simulation mit den 4 Pfeilen in alle Richtungen, um das Fenster zeitlich über die ganze Simulationsdauer zu strecken.



Ok … jetzt erscheint die ganze Simulationszeit, von links 0 ns bis rechts 1'000.000 ns.
Aber es läuft noch nichts.

Es läuft nichts, weil wir keine Stimuli bereitgestellt haben. Die Eingangssignale zu unserem Block sind nicht definiert und zeigen das mit einem «U» für «Unknown» an.

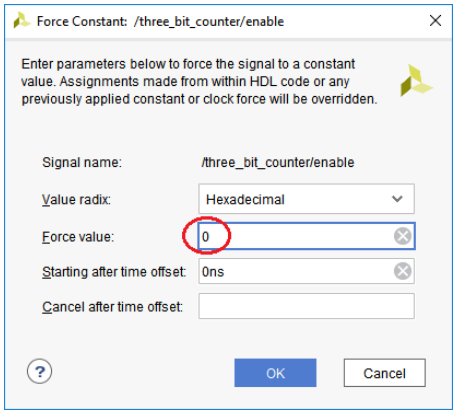


Kontrolle des Enable Signals

Als erstes wollen wir das «enable» Signal auf 0 setzen.

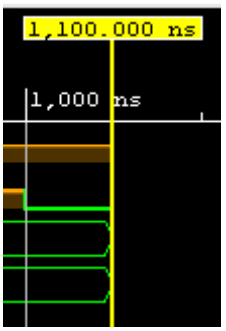
Im «Wave» Fenster erscheint nach einem mit Rechts-Klick auf den Signalnamen «enable» das rechtsstehende Menu.

Dort klicken Sie dann auf «Force Constant…» und zwingen («Force») im
neu auftauchenden Fenster den Wert (Value) auf «0».

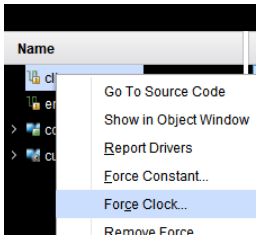


Lassen Sie die Simulation für 100 ns weiterlaufen, indem sie in der Zeile ganz unten (wo «Type a Tcl command here» steht) den Befehl «run 100 ns» eingeben, gefolgt von einem «Carriage Return» Enter.

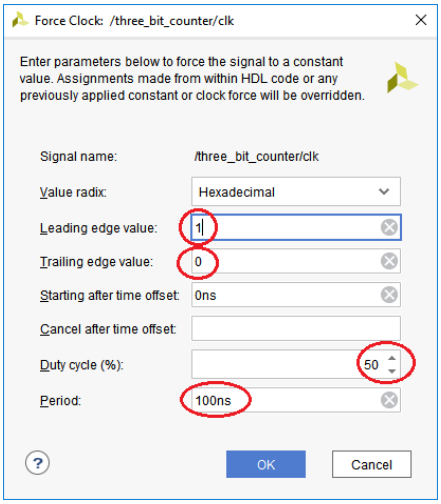
Das Ende der Simulation ist jetzt bei 1'100 ns, und während den letzten 100 ns ist das «enable» Signal jetzt grün auf null.



Anlegen des Taktes



Wie bereits beim «enable» Signal klicken wir mit der rechten Maustaste auf den Namen «clk» im «Wave» Fenster, und wählen jetzt im Menu aber «Force Clock».
Es erscheint das Menu für die Takt-Definition.



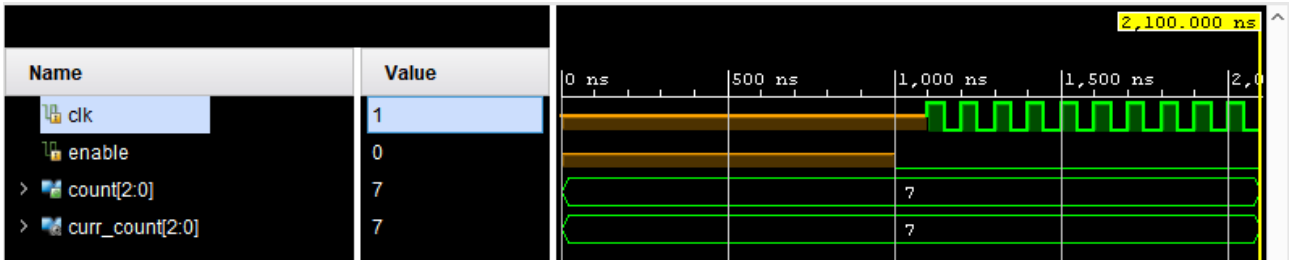
Das Taktsignal soll jeweils zuerst auf 1 gehen («Leading edge value»), und dann nach einiger Zeit auf 0 gehen («Trailing edge value»).

Belassen Sie die anderen Grundeinstellungen auf 0 ns Offset,
50 % Duty Cycle und 100 ns Periode auf «100 ns».

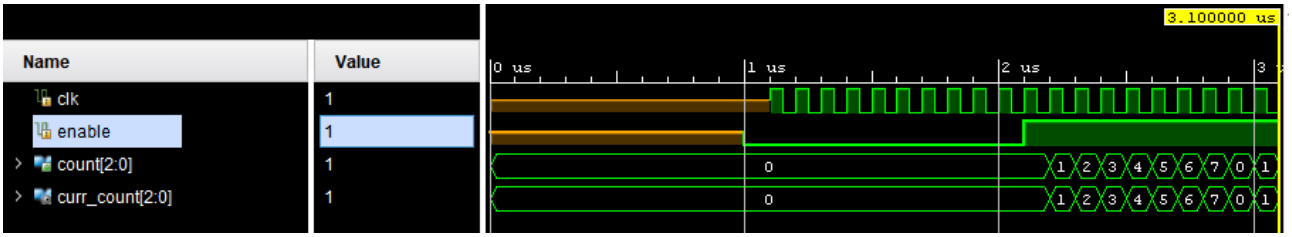
Dies ergibt einen schönen, symmetrischen 10 MHz Takt am Signal «clk».

Klicken Sie «OK», und lassen Sie dann die Simulation für 1000 ns laufen.

Die Simulation wird praktisch keine Zeit benötigen (das Design ist ja noch sehr einfach), und nach einem weiteren «Zoom Fit» mit dem Knopf  sollte es so aussehen:



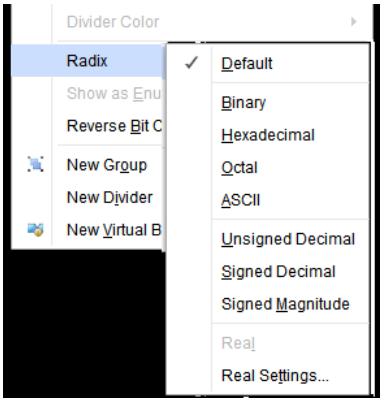
Damit der Zähler auch wirklich zählt, müssen Sie jetzt noch das «enable» Signal auf «1» setzen, und die Simulation weitere z.B. 1000 ns laufen lassen. Dann sollten Sie den Zähler zählen sehen …



3.6 Änderung der Bit-Auflösung der Anzeige (Radix)

Wählen Sie im «Wave» Fenster die beiden 3-Bit «count» Signale aus. Deren Darstellung ist zurzeit als «Default» der Typ «Unsigned Decimal».

Mit Rechts-Klick auf das Signal «count» können Sie dann im Menu unter «Radix» ein anderes Format auswählen, z.B. binär oder hexadezimal …
… ganz so, wie es für Ihre Anwendung am besten passt.



3.7 Schlussfolgerungen

Es ist grundsätzlich möglich, ein Design durch manuelle Kontrolle der Signale zu simulieren …
… aber …

- es ist mühsam und aufwendig
- die Verwendung von Eingabe-Fenstern ist intuitiv aber nicht sehr effizient
- auch nach kleinen Änderungen oder bei jedem Neustart der Simulation muss man alles wiederholen

4 Steuerung der Simulation durch ein Skript-File

Wie Sie vielleicht bemerkt haben, erscheint im «Tcl Console» Fenster nach jeder Signal-Konfiguration per Maus eine Kommando-Zeile. Wie Sie richtig vermuten, genügen auch diese Befehle um dasselbe zu erreichen. Man kann alle benötigten Befehle in ein geeignetes Text-File kopieren, und dann nacheinander automatisch ausführen lassen.

- Erstellen Sie ein neues Text-File in Ihrem Three_Bit_Counter Verzeichnis mit dem Namen «counter_stimulus.txt» und öffnen Sie es in einem Text-Editor, z.B. Notepad++.
- Kopieren Sie die bisher verwendeten und unten aufgeführten Befehle in das File
- Speichern sie diese Text-Datei

Dieses File sollte jetzt etwa folgenden Inhalt haben:

```
add_force {/three_bit_counter/enable} -radix hex {0 0ns}
run 100 ns
add_force {/three_bit_counter/clk} -radix hex {1 0ns} {0 50000ps} -repeat_every 100000ps
run 1000 ns
add_force {/three_bit_counter/enable} -radix hex {1 0ns}
run 1000 ns
```

Im «Tcl Console» Fenster von Vivado können sie jetzt folgende Befehle ausführen:

Restart // Die Simulation wird zurückgesetzt, und alle Zuweisungen werden gelöscht

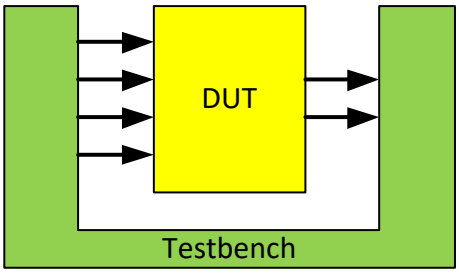
Source <Project-Pfad>/counter_stimulus.txt // Führt die Befehle im Skript-File aus. Als Pfad müssen Sie leider den ganzen Verzeichnis-Pfad zu Ihrem File eingeben.

Schlussfolgerungen
Es ist möglich, die Simulation mit einem Skript-File durchzuführen. Vor allem wenn man die Simulation wiederholen will, ist das sehr praktisch … aber …

- es ist mühsam und aufwendig das Simulations-File zu erstellen
- die Verwendung von Eingabe-Fenstern ist intuitiv aber immer noch nicht sehr effizient
- die direkte Verwendung von Tcl Befehlen ist etwas kryptisch und gewöhnungsbedürftig

5 Three_bit_counter mit einer Testbench

5.1 Die Testbench



Eine Testbench ist ein Modul, welches die zu prüfende Einheit umschliesst, und dadurch sowohl die Signal-Eingänge wie Ausgänge kontrollieren und überwachen kann.

Die zu prüfende Einheit nennt man in der Regel DUT (Device Under Test).

Die Testbench wird in der Regel in der gleichen Sprache geschrieben wie das zu simulierende Objekt, weil dann der Simulator nur eine Sprache unterstützen muss und deshalb günstiger und schneller ist. Aber es ist durchaus möglich, die Testbench in Verilog, oder sogar in SystemC oder Java zu schreiben. Ein anderer Vorteil der gleichen Sprache ist natürlich auch, dass dann der Entwickler nur eine Sprache wirklich beherrschen muss.

5.2 Regeln für die Testbench

Im Gegensatz zum DUT muss die Testbench nicht synthetisierbar sein, sie wird immer nur simuliert und wird nicht auf dem FPGA implementiert. Deshalb kann man

- Ohne Bedenken „WAIT FOR“ Befehle mit Zeiten bis Milli- oder Pico-Sekundenbereich benützen
- Nach Belieben „LOOP“ Schleifen verwenden
- Signale vom Typ Integer und Real verwenden
- Synchrone und asynchrone Prozesse nach Bedarf mischen

5.3 VHDL Testbench für den ThreeBitCounter

Diese Testbench ist spezifisch für den Baustein ThreeBitCounter aus Kapitel 5.1 geschrieben.

```
35    LIBRARY IEEE;
36    USE IEEE.STD_LOGIC_1164.ALL;
37    USE IEEE.NUMERIC_STD.ALL;
38
39    USE work.three_bit_counter_pkg.ALL;
40
41    ENTITY three_bit_counter_tb IS
42    END ENTITY three_bit_counter_tb;
44    ------------------------------------------------------------
45
46    ARCHITECTURE sim OF three_bit_counter_tb IS
47
48        SIGNAL  sl_clock, sl_enable     : STD_LOGIC := '0';
49        SIGNAL  usig3_count       : UNSIGNED(2 DOWNTO 0);
50
51    BEGIN
52
53        --      ##    Unit Under Test Instantiation
54        u_three_bit_counter : three_bit_counter PORT MAP (
55            clk       => sl_clock,
56            enable          => sl_enable,
57            count           => usig3_count
58        );
59
60
61        --      ##    TestBench Clock Process
62        clock_generator : PROCESS
63        BEGIN
64            sl_clock <= NOT sl_clock;
65            WAIT FOR 5 ns;  --     = 50 MHz clock
66        END PROCESS clock_generator;
67
68
69        --  ##    TestBench Enable Signal Generation
70        stimulus_generator : PROCESS
71        BEGIN
72            sl_enable    <= '0';  WAIT FOR 100 ns;
73            sl_enable    <= '1';  WAIT FOR 200 ns;
74
75            --  Stop Simulation
76            REPORT "End of simulation" SEVERITY FAILURE;
77        END PROCESS stimulus_generator;
78
79    END ARCHITECTURE sim;
```

5.4 Erklärungen zur Testbench für den ThreeBitCounter

Diese Testbench besteht aus folgenden Teilen:

Zeilen 42 – 43: Deklaration der Entity
Wie jedes VHDL Modul, braucht auch dieses eine Deklaration der «`ENTITY`». Da es aber keine Signale nach aussen gibt, ist die «`PORT LIST`» leer und kann weggelassen werden.

Zeile 45: Architecture
Wie jedes VHDL Modul hat auch die Testbench eine Architektur, eine innere Struktur und Inhalt.

Zeilen 47 – 48: Definition der Signale
Wir benötigen innerhalb der Testbench auch Signale, welche hier definiert werden. In unserem einfachen Fall sind dies nur die Schnittstellen-Signale. Bei komplexeren Testbenches können dies auch die inneren Zustandssignale, Flags, Zähler und ähnliches sein.

Zeile 50: BEGIN
Hier fängt nun endlich die Architektur der Testbench wirklich an.

Zeilen 52 – 57: Instanziierung der DUT
Das zu testende Modul wird hier als hierarchisch tiefer gelegene Komponente instanziiert, und die Signale der Testbench den Signalen des Moduls zugewiesen.

Zeilen 60 – 65: Clock Prozess
Hier wird ein fortwährendes Taktsignal mit 100 MHz erzeugt. Jeweils 5 ns hoch, dann 5 ns tief, und es fängt endlos wieder von vorne an. Erreicht wird dies, indem alle 5 ns das Taktsignal invertiert wird.

Zeilen 68 – 76: Erzeugung der Stimulus Signals
Je nach Anwendung kann dies sehr umfangreich werden – in diesem Beispiel gibt es lediglich das «enable» Signal. Dieses soll wie schon bei der manuellen Simulation zuerst 100 ns ausgeschaltet gehalten werden (0), und dann für 200 ns aktiv (1).

Zeile 76: Abbruch der Simulation
Wenn die Simulation zu Ende ist, wird sie mit «`ASSERT FALSE … SEVERETY FAILURE;` » beendet.
Durch einen **ASSERT** Befehl der immer getriggert wird (Bedingung ist immer «`FALSE`») und der die Stufe «FAILURE» hat, wird die Simulation an dieser Stelle unterbrochen. Ein kleiner Nachteil dieser Art des Beendens der Simulation ist die Fehlermeldung im Report-Fenster, was aber in der Tat keinen Fehler meldet. Dafür hat man den Vorteil, dass man keine feste Simulationslänge eingeben muss, sondern die Simulation jederzeit wieder automatisch angepasst wird, wenn neue Funktionen getestet werden.
Bei einem immer aktiven «`ASSERT`» kann man die Bedingung weglassen, und nur `REPORT` … schreiben.

6 Self-checking Testbench : Adder4

Eine gute Testbench kann nicht nur Signale generieren, sondern kann das Resultat auch selbst überprüfen.
Für dieses Beispiel verwenden wir statt «high-level Code» absichtlich einen einfachen 4-Bit Addierer mit Carry, der aus 4 einzelnen Volladdierer aufgebaut ist, welche wiederum aus einzelnen Gattern bestehen.
Damit haben wir eine «low-level» Implementation, welche wir gegen eine «high-level» Funktion testen.

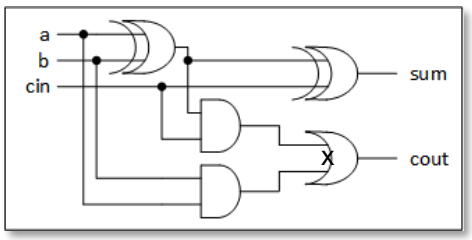
6.1 VHDL Source Code

6.1.1 full_add.vhd

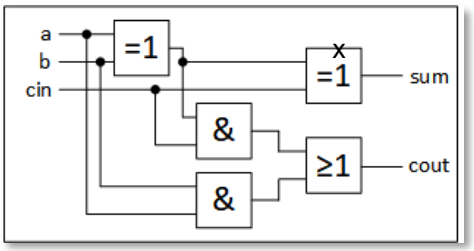
Dies ist ein einfacher Volladdierer (Full-Adder) mit 3 Eingängen und 2 Ausgängen.

Um die Instanziierung zu vereinfachen besitzt dieses Modul sein eigenes `PACKAGE` und `COMPONENT` Deklaration … damit man diese nicht bei jeder Verwendung wieder neu schreiben muss.

```
35    LIBRARY IEEE;
36    USE IEEE.STD_LOGIC_1164.ALL;
37
38    PACKAGE full_add_pkg IS
39        COMPONENT full_add IS
40            PORT (
41                a, b, cin : IN  STD_LOGIC;
42                sum, cout : OUT STD_LOGIC
43            );
44        END COMPONENT full_add;
45    END PACKAGE full_add_pkg;
46
47    --------------------------
49    LIBRARY IEEE;
50    USE IEEE.STD_LOGIC_1164.ALL;
51
52    ENTITY full_add IS
53            PORT (
54                a, b, cin : IN  STD_LOGIC;
55                sum, cout : OUT STD_LOGIC
56            );
57    END ENTITY full_add;
59    ----------------------------
61    ARCHITECTURE rtl OF full_add IS
63        SIGNAL x : STD_LOGIC;
65    BEGIN
67        x       <= a XOR b;
68        sum     <= x XOR cin;
69        cout    <= (a AND b) OR (x AND cin);
71    END ARCHITECTURE rtl;
```



Darstellung der Logik mit U.S. Symbolen



Darstellung der Logik mit DIN Symbolen

6.2 adder4

Dieser 4-Bit Volladdierer verwendet 4-mal das Modul **`full_add`**, und verbindet diese nur. Deshalb nennt man diese Architektur «struct» oder «structure», da auf dieser Ebene keine logischen Verknüpfungen oder Bedingungen existieren.

```
35    LIBRARY IEEE;
36    USE IEEE.STD_LOGIC_1164.ALL;
37    USE IEEE.numeric_std.ALL;
38
39    PACKAGE adder4_pkg IS
40        COMPONENT adder4 IS
41            PORT (
42                a, b   : IN  UNSGNED(3 DOWNTO 0);
44                cin    : IN  STD_LOGIC;
43                sum    : OUT UNSIGED(3 DOWNTO 0);
45                cout   : OUT STD_LOGIC
46            );
47        END COMPONENT adder4;
48    END PACKAGE adder4_pkg;
49
50    ----------------------------------------------------------------------
51
52    LIBRARY IEEE;
53    USE IEEE.STD_LOGIC_1164.ALL;
54    USE IEEE.numeric_std.ALL;
55    USE work.full_add_pkg.ALL;
56
57    ENTITY adder4 IS
58            PORT (
59                a, b   : IN  UNSIGNED(3 DOWNTO 0);
60                cin    : IN  STD_LOGIC;
61                sum    : OUT UNSIGNED(3 DOWNTO 0);
62                cout   : OUT STD_LOGIC
63            );
64    END ENTITY adder4;
65
66    ----------------------------------------------------------------------
67
68    ARCHITECTURE struct OF adder4 IS
69        SIGNAL   c      : STD_LOGIC_VECTOR(2 DOWNTO 0);
70    BEGIN
71        u_adder_0 : full_add PORT MAP (a(0), b(0), cin,  s(0), c(0));
72        u_adder_1 : full_add PORT MAP (a(1), b(1), c(0), s(1), c(1));
73        u_adder_2 : full_add PORT MAP (a(2), b(2), c(1), s(2), c(2));
74        u_adder_3 : full_add PORT MAP (a(3), b(3), c(2), s(3), cout);
75    END ARCHITECTURE struct;
```

6.3 Einfache selbst-checkende Testbench für adder4

```
36  LIBRARY IEEE;
37  USE IEEE.STD_LOGIC_1164.ALL;
38  USE IEEE.NUMERIC_STD.ALL;
39  USE work.adder4_pkg.ALL;
40
41  ENTITY adder4_tb IS
42  END ENTITY adder4_tb;
43
44  ARCHITECTURE sim OF adder4_tb IS
45
46      SIGNAL  a, b, sum    : UNSIGNED(3 DOWNTO 0);
47      SIGNAL  cin, cout    : STD_LOGIC;
48      SIGNAL  num_errors   : INTEGER;
49      SIGNAL  int_results  : INTEGER;
50
51  BEGIN
53      --##    Unit Under Test Instantiation
54      u_dut : adder4 PORT MAP (
55          a    => a,
56          b    => b,
57          cin  => cin,
58          sum  => sum,
59          cout => cout
60      );
61
62      --##    TB Main Process
63      signal_generator : PROCESS
64          VARIABLE v_result     : INTEGER;
65      BEGIN
66          num_errors <= 0;
67          FOR count_a IN 0 TO 15 LOOP
68              FOR count_b IN 0 TO 15 LOOP
69                  FOR count_c IN 0 TO 1 LOOP
70                      a <= to_unsigned(count_a, 4);
71                      b <= to_unsigned(count_b, 4);
72                     IF count_c > 0 THEN cin <= '1';
73                     ELSE                cin <= '0'; END IF;
74
75                     --  Assemble result
76                     WAIT FOR 1 ns;
77                     v_result := to_integer(sum);
78                     IF cout = '1' THEN
79                         v_result := v_result + 16;
80                     END IF;
81                     int_results <= v_result; -- For visibility only
82
83                     --  Verify result
84                     WAIT FOR 1 ns;
85                     IF v_result /= count_a + count_b + count_c THEN
86                         num_errors <= num_errors + 1;
87                         REPORT "Result Mismatch" SEVERITY WARNING;
88                     END IF;
89                     WAIT FOR 18 ns;  --    Results are stable ...
90                 END LOOP;
91             END LOOP;
```

```
92              END LOOP;
93
94              IF num_errors = 0 THEN
95                  REPORT "Simulation without errors" SEVERITY NOTE;
96              ELSE
97                  REPORT "Simulation with errors" SEVERITY ERROR;
98              END IF;
99
100              --  Stop Simulation
101              ASSERT FALSE REPORT "End of simulation" SEVERITY FAILURE;
102          END PROCESS signal_generator;
103
104      END ARCHITECTURE sim;
```

6.4 Erklärungen zur Testbench

Zeilen 54 – 57: Instanziierung des zu testenden Modules
Traditionell heisst das zu testende Objekt «DUT» = «Device Under Test».
Alle Eingänge müssen von der Testbench getrieben werden.
Es müssen aber nicht alle Ausgänge verbunden sein, man kann Signale auch mit `OPEN` offen lassen.

Zeilen 62 – 102: Prozess zur Signalerzeugung und Überprüfung
Da das Target in diesem Fall ein rein kombinatorischer Block ist, genügt ein einzelner Prozess und es wird keine Takterzeugung benötigt. In dieser Testbench übernimmt der Prozess «signal_generator» alle Aufgaben.

Zeile 64: Variable `v_result`
Das Resultat des Adders ist über 4 Summen-Bits und ein Carry-Bit verteilt, und muss in seiner Bedeutung erst noch zusammengesetzt werden. Wenn wir dazu ein SIGNAL verwenden, dann wird die Zuweisung erst am Ende des Prozesses getätigt, was für eine sofortige Kontrolle zu spät ist. Bei Verwendung einer Variablen können wir das Resultat zusammensetzen (Zeilen 77 bis 80) und dann sofort kontrollieren (Zeile 85).

Zeile 66: Signal `num_errors`
Dies ist der Zähler für Fehler. Er wird am Anfang auf null gesetzt, und dann bei jeder Abweichung zwischen der Testbench und der zu testenden Logik erhöht. Wenn der Zähler bis zum Schluss auf null bleibt, ist die Simulation fehlerfrei.

Zeilen 67 – 69: Erzeugung der Eingangssignale
Hier geht es darum, auf möglichst einfache Art möglichst viele mögliche Zustände zu erzeugen. Mit den drei verschachtelten `FOR` Schleifen werden nacheinander alle Möglichkeiten für den 4 Bit Eingang a und b, sowie die zwei Möglichkeiten für das Carry-In abgedeckt.

Dies ist also ein «Exhaustive Check» aller Möglichkeiten. Für einen einfachen 4-Bit Addierer gibt es «nur»
16 x 16 x 2 = 512 Möglichkeiten – ein noch durchaus überschaubare Zahl von Kombinationen.

In der Praxis wird die Zahl der Möglichkeiten sehr rasch unüberschaubar gross, was dann eine vollständige Abdeckung verunmöglicht – in diesen Fällen ist man dann gezwungen auf Stichproben, Zufallswerte und Teilabdeckung der Möglichkeiten auszuweichen.

Zeilen 70 – 73: Erstellen der Eingangssignale (Stimuli)
Mit «count_a», «count_b» und «count_c» werden alle Möglichkeiten durchgespielt. Doch da diese Werte jeweils in **FOR** Schleifen automatisch als Zählervariablen angesetzt werden, lassen sich diese nicht direkt verwenden. Mit dem «**to_unsigned**» Befehl werden diese Integer Werte zu 4-Bit «STD_LOGIC_VECTOR» Format umgeformt. Im Falle des Carry-In setzt eine «**IF** – **ELSE**» Anweisung das Bit auf «0» oder «1».

Zeilen 75 – 81: Zusammensetzen des Resultats
1 Nanosekunde nach Anlegen der Stimuli wird das Resultat genommen. Dabei setzt es sich aus dem «sum» und dem «cout» Signal zusammen. Wenn «cout», also Carry-out gesetzt ist, kommt noch eine zusätzliche Stelle dazu – also wird 16 zum Resultat addiert.
Zur Sichtbarkeit wird die Variable **v_result** auf das Signal **int_result** kopiert, welches dann in der Simulation auch sichtbar und beobachtbar ist. Variablen sind es leider nicht direkt.

Zeilen 83 – 88: Kontrolle der Resultate
Wieder 1 Nanosekunde später wird das zusammengesetzte Resultat **v_result** gegen die Summe der Werte aus den drei **FOR** Schleifen verglichen. Wenn dies nicht identisch ist, wird die Zahl der Fehler erhöht.

Zeile 87: Ausgabe einer Warnung während der Simulation
Wenn eine Abweichung festgestellt wurde, wird nicht nur der Zähler «num_errors» erhöht, sondern mit **REPORT** auch eine Warnung in der Simulation ausgegeben. Zusammen mit dem Text wird auch die Schwere des Fehlers (**SEVERITY**) Ausgegeben und farblich im Wave-Fenster markiert.
Möglichkeiten hier sind «NOTE», «WARNING», «ERROR» und «FAILURE».

Zeile 89: Warten auf 18 ns
Die Simulation soll alle 20 ns einen Wert testen. Mit 512 Möglichkeiten läuft die Simulation also ca. 1 µs.
Damit die Signale besser beobachtbar sind, bleiben sie nach der Auswertung 2 x 1ns noch für 18 ns stabil.

Zeile 94 bis 101: Abbruch der Simulation
Wenn alle Möglichkeiten der **FOR** Schleifen durchprobiert sind, bricht die Simulation ab. Vor dem Ende des Xilinx Simulator wird noch eine Zusammenfassung ausgegeben, ob die Simulation fehlerfrei war, oder nicht.

7 Anspruchsvolle Testbench : Arcus Tangens CORDIC

Der hier folgende Teil ist noch in Arbeit und nicht fertig !!!

Hier ist noch ein anspruchsvolles Beispiel, welches die Stärken einer Self-Checking Testbench durch alternative Berechnung der Resultate im Simulations-Bereich zeigt.

7.1 VHDL Source Code

Dieser auf dem CORDIC Algorithmus basierende Block zur Berechnung des Arcus Tangens eines Winkels aus Ankathete und Gegenkathete verwendet eine iterative Näherung und benötigt praktisch nur so viele Takt-Zyklen wie das Resultat dann Bit-Genauigkeit haben soll. Dies ist die Grundlage des CORDIC, und kann auf dem Internet nachgelesen werden.

7.1.1 arctan_cordic.m.vhd

```
55  LIBRARY IEEE;
56  USE IEEE.std_logic_1164.ALL;
57  USE IEEE.numeric_std.ALL;
58
59  PACKAGE arctan_cordic_pkg IS
60      COMPONENT arctan_cordic IS
61          PORT (
62              isl_clock              : IN  std_logic;
63              isl_start              : IN  std_logic;
64              isig12_input_x         : IN  signed (11 DOWNTO 0);
65              isig12_input_y         : IN  signed (11 DOWNTO 0);
66              osl_output_valid     : OUT std_logic;
67              osig12_arctan_output : OUT signed (11 DOWNTO 0)
68          );
69      END COMPONENT arctan_cordic;
70  END PACKAGE arctan_cordic_pkg;
71
72  ------------------------------------------------------------------
73
74  LIBRARY IEEE;
75  USE IEEE.std_logic_1164.ALL;
76  USE IEEE.numeric_std.ALL;
77
78  USE work.cordic_rom_pkg.ALL;
79  USE work.barrel_shifter_pkg.ALL;
80
81  ENTITY arctan_cordic IS
82          PORT (
83              isl_clock              : IN  std_logic;
84              isl_start              : IN  std_logic;
85              isig12_input_x         : IN  signed (11 DOWNTO 0);
86              isig12_input_y         : IN  signed (11 DOWNTO 0);
87              osl_output_valid     : OUT std_logic;
88              osig12_arctan_output : OUT signed (11 DOWNTO 0)
```

```
 89          );
 90  END ENTITY arctan_cordic;
 91
 92  ------------------------------------------------------------------
-----

 94  ARCHITECTURE rtl OF arctan_cordic IS
 95
 96      TYPE t_cordic_reg IS RECORD
 97          sl_start_cordic_d1          : std_logic;
 98          sl_finished                 : std_logic;
 99          usig4_iteration_count : unsigned(3 DOWNTO 0);
100
101          sig19_cordic_adder_1  : signed(18 DOWNTO 0);
102          sig19_cordic_adder_2  : signed(18 DOWNTO 0);
103          sig19_cordic_adder_3  : signed(18 DOWNTO 0);
104          sl_carry_1                  : std_logic;
105          sl_carry_2                  : std_logic;
106          sl_carry_3                  : std_logic;
107
108          sig12_cordic_output         : signed(11 DOWNTO 0);
109      END RECORD;
110
111      TYPE t_adder_result IS RECORD
112          sig18_sum             : signed(17 DOWNTO 0);
113          sl_carry                    : std_logic;
114      END RECORD;
115
116      SIGNAL r, r_next                : t_cordic_reg;
117
118      SIGNAL sig12_rom_inv5           : signed(11 DOWNTO 0);
119
120      SIGNAL sig18_input_x            : signed(17 DOWNTO 0);
121      SIGNAL sig18_input_y            : signed(17 DOWNTO 0);
122
123      SIGNAL sig18_barrel_shift_1_out : signed(17 DOWNTO 0);
124      SIGNAL sig18_barrel_shift_2_out : signed(17 DOWNTO 0);
125
126  ------------------------------------------------------------------
```

```
127
128         -- Calculate bit sum using carry from previous step,
129         -- and then the carry out
130         FUNCTION add_w_carry (
131                         isig18_a, isig18_b : signed(17 DOWNTO 0);
132                         isl_carry_in : std_logic
133                 ) RETURN t_adder_result IS
134             VARIABLE sl_carry           : std_logic;
135             VARIABLE r_result           : t_adder_result;
136         BEGIN
137             sl_carry  := isl_carry_in;
138             FOR i IN 0 TO 17 LOOP
139                 r_result.sig18_sum(i)   :=  isig18_a(i)
140                                         XOR isig18_b(i)
141                                         XOR sl_carry;
142                 sl_carry       := (isig18_a(i) AND isig18_b(i))
143                                 OR (isig18_a(i) AND sl _carry)
144                                 OR (isig18_b(i) AND sl_carry);
145             END LOOP;
146             r_result.sl_carry           := sl_carry;
147             RETURN r_result;
148         END FUNCTION add_w_carry;
149
150     ------------------------------------------------------------------
```

```
151
152   BEGIN
153
154       --  Resize input from Q12.0 to Q14.4 for better accuracy
155      sig18_input_x <= (17 DOWNTO 16 => isig12_input_x(11))
156                     & isig12_input_x
157                     & (3 DOWNTO 0 => isig12_input_x(11));
158
159      sig18_input_y <= (17 DOWNTO 16 => isig12_input_y(11)
160                     & isig12_input_y
161                     & (3 DOWNTO 0 => isig12_input_y(11));
162
163       cordic_comb_proc : PROCESS (isl_clock, isl_start, r, r_next,
164                                   isig12_input_x, isig12_input_y,
165                                   sig18_input_x, sig18_input_y,
166                                   sig18_barrel_shift_1_out,
167                                   sig18_barrel_shift_2_out,
168                                   sig12_rom_inv5
169                                  )
170
171           VARIABLE  v : t_cordic_reg;
172
173           VARIABLE  vsig18_adder_1_in_a  : signed(17 DOWNTO 0);
174          VARIABLE   vsig18_adder_1_in_b  : signed(17 DOWNTO 0);
175           VARIABLE  vsig18_adder_2_in_a  : signed(17 DOWNTO 0);
176           VARIABLE  vsig18_adder_2_in_b  : signed(17 DOWNTO 0);
177           VARIABLE  vsig18_adder_3_in_a  : signed(17 DOWNTO 0);
178           VARIABLE  vsig18_adder_3_in_b  : signed(17 DOWNTO 0);
179
180           VARIABLE  vr_adder_result   : t_adder_result;
181
182       BEGIN
183           v := r;               --    Keep variables stable
184
185           vsig18_adder_1_in_a   := r.sig19_cordic_adder_1(17 DOWNTO
186   0);
187           vsig18_adder_1_in_b   := r.sig19_cordic_adder_1(17 DOWNTO
188   0);
189           vsig18_adder_2_in_a   := r.sig19_cordic_adder_2(17 DOWNTO
190   0);
191           vsig18_adder_2_in_b   := r.sig19_cordic_adder_2(17 DOWNTO
192   0);
193           vsig18_adder_3_in_a   := r.sig19_cordic_adder_3(17 DOWNTO
194   0);
195           vsig18_adder_3_in_b   := r.sig19_cordic_adder_3(17 DOWNTO
196   0);
197
198           v.sl_start_cordic_d1  := isl_start; -- Start on rising edge
199
200
```

```
195           --  First cycle in computation
196           IF r.sl_start_cordic_d1 = '0' AND isl_start = '1' THEN
197
198               -- if numerator is equal to +/-0 then return directly 0
199               IF (isig12_input_y = B"000000000000"
200             OR isig12_input_y = B"111111111111") THEN
201                    v.sl_finished := '1';
202                    v.sig12_cordic_output  := (OTHERS => '0');
203
204               -- if donumerator is equal to +/-0 then return directly
205               -- +90 or -90 according to the sign of numerator
206               ELSIF (isig12_input_x = B"000000000000"
207             OR     isig12_input_x = B"111111111111") THEN
208                    v.sl_finished := '1';
209                   IF (isig12_input_y(11) = '0') THEN  -- 1440 = +90
deg
210                       v.sig12_cordic_output   := B"010110100000";
211                    ELSE                              -- -1440 == -90
deg
212                       v.sig12_cordic_output   := B"101001011111";
213                    END IF;
214
215     -- if no extreme case, start 1st cycle of regular cordic process
216                ELSE
217                   v.sl_finished  := '0';
218                   v.usig4_iteration_count    := (OTHERS => '0');
219
220                   -- Process first round
221                   IF isig12_input_x(11) = '1' THEN       -- sign of x
222                       vsig18_adder_1_in_a     := NOT sig18_input_x + 1;
223                        vsig18_adder_2_in_a    := NOT sig18_input_y + 1;
224                    ELSE
225                        vsig18_adder_1_in_a         := sig18_input_x;
226                        vsig18_adder_2_in_a         := sig18_input_y;
227                    END IF;
228                    v.sl_carry_1 := '0';
229                    v.sl_carry_2 := '0';
230                    v.sl_carry_3 := '0';
231
232                    vsig18_adder_1_in_b := (OTHERS => '0');
233                    vsig18_adder_2_in_b := (OTHERS => '0');
234
235                    vsig18_adder_3_in_a := (OTHERS => '0');
236                    vsig18_adder_3_in_b := (OTHERS => '0');
237
238               END IF;
239
240
```

```
241           --   Regular cordic processing for 2nd and subsequent steps
242           ELSIF r.usig4_iteration_count < 8 THEN
243             vsig18_adder_1_in_a := r.sig19_cordic_adder_1(17 DOWNTO
0);
244             vsig18_adder_2_in_a := r.sig19_cordic_adder_2(17 DOWNTO
0);
245             vsig18_adder_3_in_a := r.sig19_cordic_adder_3(17 DOWNTO
0);
```

Zum überarbeiten …

```
228
229
230           IF r.sig19_cordic_adder_2(17) = '1' THEN
231               vsig18_adder_1_in_b     := NOT sig18_barrel_shift_1_out;
232               vsig18_adder_2_in_b     := sig18_barrel_shift_2_out;
233               vsig18_adder_3_in_b     := NOT ("000000" &
sig12_rom_inv5);
234           ELSE
235               vsig18_adder_1_in_b     := sig18_barrel_shift_1_out;
236               vsig18_adder_2_in_b     := NOT sig18_barrel_shift_2_out;
237               vsig18_adder_3_in_b     := "000000" & sig12_rom_inv5;
238           END IF;
239
240           v.usig4_iteration_count     := r.usig4_iteration_count + 1;
241
242           -- Check if the result has been found (num == 0)
243           IF (r.sig19_cordic_adder_2 = -1 OR
r.sig19_cordic_adder_2=0) THEN
244               v.usig4_iteration_count := x"8";
245           END IF;


248       --      Reached the result
249       ELSE
250
251           v.sig12_cordic_output     := r.sig19_cordic_adder_3(13
DOWNTO 2);
              -- Override sign bit
252           v.sig12_cordic_output(11) := r.sig19_cordic_adder_3(17);
253           v.usig4_iteration_count   := x"0";
254           v.sl_finished              := '1';
255
256       END IF;
257
258       --      Adder instatiation
259       vr_adder_result     := add_w_carry(vsig18_adder_1_in_a,
                                 vsig18_adder_1_in_b, v.sl_carry_1);
260       v.sig19_cordic_adder_1    := vr_adder_result.sig18_sum(17)
                                     & vr_adder_result.sig18_sum;
261       v.sl_carry_1              := vr_adder_result.sl_carry;
262
263       vr_adder_result           := add_w_carry(vsig18_adder_2_in_a,
                                 vsig18_adder_2_in_b, v.sl_carry_2);
264       v.sig19_cordic_adder_2    := vr_adder_result.sig18_sum(17)
                                     & vr_adder_result.sig18_sum;
```

```
265       v.sl_carry_2              := vr_adder_result.sl_carry;
266
267       vr_adder_result           := add_w_carry(vsig18_adder_3_in_a,
                                       vsig18_adder_3_in_b, v.sl_carry_3);
268       v.sig19_cordic_adder_3    := vr_adder_result.sig18_sum(17)
                                        & vr_adder_result.sig18_sum;
269       v.sl_carry_3              := vr_adder_result.sl_carry;
270
271       r_next <= v;                      --    Copy variables to signals
272   END PROCESS cordic_comb_proc;
273
274   ------------------------------------------------------------------
-----
275
276       cordic_reg_proc : PROCESS (isl_clock)
277       BEGIN
278           IF rising_edge(isl_clock) THEN r <= r_next; END IF;
279       END PROCESS cordic_reg_proc;
280
281   ------------------------------------------------------------------
-----
282
283       --  Output assignments
284       osl_output_valid          <= r.sl_finished;
285       osig12_arctan_output      <= r.sig12_cordic_output;
286
287       u_shifter_1 : barrel_shifter PORT MAP (
288           usig4_shift_value     => r.usig4_iteration_count,
289           isig_input            => r.sig19_cordic_adder_2(17 DOWNTO
0),
290           osig_output           => sig18_barrel_shift_1_out
291       );
292
293       u_shifter_2 : barrel_shifter PORT MAP (
294           usig4_shift_value     => r.usig4_iteration_count,
295           isig_input            => r.sig19_cordic_adder_1(17 DOWNTO
0),
296           osig_output           => sig18_barrel_shift_2_out
297       );
298
299       u_cordic_rom : cordic_rom PORT MAP (
300           usig4_addr            => r.usig4_iteration_count,
301           sig12_data_out        => sig12_rom_inv5
302       );
303
304   END ARCHITECTURE rtl;
```

7.1.2 barrel_shifter.m.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

PACKAGE barrel_shifter_pkg IS
      COMPONENT barrel_shifter IS
            GENERIC (N : integer := 18);
```

```
            PORT (
                  usig4_shift_value : IN  unsigned(3 DOWNTO 0);
                  isig_input              : IN  signed (N-1 DOWNTO 0);
                  osig_output             : OUT signed (N-1 DOWNTO 0)
            );
      END COMPONENT barrel_shifter;
END PACKAGE barrel_shifter_pkg;


------------------------------------------------------------------------
------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY barrel_shifter is
    GENERIC (N : integer := 18);
    PORT (
                  usig4_shift_value : IN  unsigned(3 DOWNTO 0);
                  isig_input              : IN  signed (N-1 DOWNTO 0);
                  osig_output             : OUT signed (N-1 DOWNTO 0)
    );
END barrel_shifter;

------------------------------------------------------------------------
------

ARCHITECTURE rtl OF barrel_shifter IS

    SIGNAL sig_temp_1, sig_temp_2: signed (N-1 DOWNTO 0);

BEGIN
      shift_one : PROCESS (isig_input, usig4_shift_value(0))
      BEGIN
            IF usig4_shift_value(0)='1' THEN      -- shift-by-one
                  sig_temp_1(N-1) <= isig_input(N-1);
                  sig_temp_1(N-2 DOWNTO 0) <= isig_input(N-1 DOWNTO 1);
            ELSE
                  sig_temp_1(N-1 DOWNTO 0) <= isig_input(N-1 DOWNTO 0);
            END IF;
      END PROCESS;

    shift_two : PROCESS (sig_temp_1, usig4_shift_value(1))
    BEGIN
        IF usig4_shift_value(1)='1' THEN    -- shift-by-two
            sig_temp_2(N-1) <= sig_temp_1(N-1);
            sig_temp_2(N-2) <= sig_temp_1(N-1);
            sig_temp_2(N-3 DOWNTO 0) <= sig_temp_1(N-1 DOWNTO 2);
        ELSE
            sig_temp_2(N-1 DOWNTO 0) <= sig_temp_1(N-1 DOWNTO 0);
        END IF;
    END PROCESS;

    shift_four : PROCESS (sig_temp_2, usig4_shift_value(2))
    BEGIN
        IF usig4_shift_value(2)='1' THEN    -- shift-by-four
            osig_output(N-1) <= sig_temp_2(N-1);
```

```
            osig_output(N-2) <= sig_temp_2(N-1);
            osig_output(N-3) <= sig_temp_2(N-1);
            osig_output(N-4) <= sig_temp_2(N-1);
            osig_output(N-5 DOWNTO 0) <= sig_temp_2(N-1 DOWNTO 4);
        ELSE
            osig_output(N-1 DOWNTO 0) <= sig_temp_2(N-1 DOWNTO 0);
        END IF;
    END PROCESS;

END ARCHITECTURE rtl;
```

7.1.3 cordic_rom.m.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE IEEE.numeric_std.ALL;

PACKAGE cordic_rom_pkg IS
      COMPONENT cordic_rom IS
            PORT (
                  usig4_addr        : IN  unsigned(3 downto 0);
                  sig12_data_out    : OUT signed(11 DOWNTO 0)
        );
      END COMPONENT cordic_rom;
END PACKAGE cordic_rom_pkg;

--------------------------------------------------------------------------
-------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE IEEE.numeric_std.ALL;

ENTITY cordic_rom IS
            PORT (
                  usig4_addr        : IN  unsigned(3 downto 0);
                  sig12_data_out    : OUT signed(11 DOWNTO 0)
            );
END ENTITY cordic_rom;

--------------------------------------------------------------------------
-------------------------

ARCHITECTURE structural OF cordic_rom IS

      CONSTANT DIMROM: natural := 8;
      CONSTANT DIMWORD: natural := 12;
      TYPE   ROM_IMAGE IS ARRAY (integer RANGE 0 TO DIMROM-1) OF
signed(DIMWORD-1 DOWNTO 0);

      CONSTANT ROM : ROM_IMAGE := ( -- INTEGER value -- FRACTIONAL value
            0 => x"B40",                  -- 2880              -- 45
            1 => x"6A4",                  -- 1700                  --
26,562
```

```
            2 => x"382",                        -- 898                  -
- 14,031
            3 => x"1C8",                        -- 456                --
7,125
            4 => x"0E5",                        -- 229                  -
- 3,578
            5 => x"073",                        -- 115                  -
-     1,796
            6 => x"039",                        -- 57                   -
- 0,890
            7 => x"01D"                         -- 29           -- 0,453
      );

BEGIN
    sig12_data_out <= ROM (to_integer(usig4_addr(2 DOWNTO 0)));
END ARCHITECTURE structural;
```

7.2 arctan_cordic.tb.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.math_real.ALL; -- for UNIFORM, TRUNC

USE work.arctan_cordic_pkg.ALL;

ENTITY arctan_cordic_tb IS
END ENTITY arctan_cordic_tb;

ARCHITECTURE behavioral OF arctan_cordic_tb IS


      TYPE   t_tb_result IS (GOOD, ERROR);

      SIGNAL tb_result                  : t_tb_result := GOOD;

      SIGNAL i_denumerator              : integer := 5;
      SIGNAL i_numerator                     : integer := 5;

      SIGNAL real_arc_tan                    : real;
      SIGNAL real_arc_tan_rtl_out   : real;
      SIGNAL real_diff_of_rtl           : real;

      SIGNAL sig12_input_x              : signed (11 downto 0); -- WRITE
HERE THE VALUE FOR DEN
      SIGNAL sig12_input_y              : signed (11 downto 0); -- WRITE
HERE THE VALUE FOR NUM
      SIGNAL sig12_output_z             : signed (11 downto 0);

      SIGNAL sl_output_valid            : std_logic;
      SIGNAL sl_output_valid_d1         : std_logic;
      SIGNAL sl_req_sample              : std_logic := '0';

      SIGNAL sl_clock                        : std_logic := '0';
      SIGNAL sl_reset, sl_reset_d1  : std_logic := '0';
```

```
BEGIN

      --    ##    Instantiate Device Under Test
      --    ##
      --    ################################
    my_rtl_cordic : arctan_cordic PORT MAP (
            isl_clock                   => sl_clock,
            isl_start                   => sl_req_sample,
            isig12_input_x              => sig12_input_x,
            isig12_input_y              => sig12_input_y,
            osl_output_valid        => sl_output_valid,
            osig12_arctan_output    => sig12_output_z
      );
      real_arc_tan_rtl_out          <=
real(to_integer(signed(sig12_output_z)))/16.0;


      --    ##    sl_clock and sl_reset SIGNALs
      --    ################################
      sl_clock    <= NOT sl_clock after 100 ns; -- 50 MHz
      sl_reset    <= '1' after 200 ns,'0' after 600 ns; --, '1' after
9us, '0' after 9.3us ;


      --    ##    Random Stimulus Generation
      --    ##
      --    ################################
      random_stim_gen_proc : PROCESS (sl_clock) --sl_reset, egress_valid)
            VARIABLE seed1    : positive := 2564;           -- Seed values
for random generator
            VARIABLE seed2    : positive := 6542;           -- Seed values
for random generator
            VARIABLE rand: real;                            -- Random
real-number value in range 0 to 1.0
            VARIABLE int_rand_x : integer;                  -- Initialise
seed1, seed2 if you want -
            VARIABLE int_rand_y : integer;                  -- otherwise
they're initialised to 1 by default

      BEGIN
            IF rising_edge(sl_clock) THEN
                  sl_reset_d1                       <= sl_reset;
                  sl_output_valid_d1                <= sl_output_valid;

                  --    Act upon falling edge of sl_reset, or rising edge
of output_valid
                  IF (sl_reset = '0' AND sl_reset_d1 = '1')
                  OR (sl_output_valid = '1' AND sl_output_valid_d1 = '0')
THEN

                        UNIFORM(seed1, seed2, rand);
            -- generate random value, range 0 .. 1
                        int_rand_x := INTEGER(TRUNC(rand*4096.0-2048.0));
      -- convert to integer, range 0 - 4095
                        i_denumerator     <= int_rand_x;
                        sig12_input_x           <= to_signed(int_rand_x,
12);  -- convert integer to std_logic_vector
```

```
                        UNIFORM(seed1, seed2, rand);
            -- generate random value, range 0 .. 1
                        int_rand_y := INTEGER(TRUNC(rand*4096.0-2048.0));
      -- convert to integer, range 0 - 4095
                        i_numerator <= int_rand_y;
                        sig12_input_y          <= to_signed(int_rand_y,
12);  -- convert integer to std_logic_vector

                        sl_req_sample <= '1';
                  ELSE
                        real_arc_tan      <=
arctan(real(i_numerator)/real(i_denumerator)) * 180.0 / 3.1415;
                        sl_req_sample <= '0';
                  END IF;

            END IF;
      END PROCESS random_stim_gen_proc;


      --    ##    Self-Testing the results
      --    ##
      --    ################################
      check_result_proc : PROCESS (sl_clock)
      BEGIN
            IF rising_edge(sl_clock) THEN
                  IF (sl_output_valid = '1' AND sl_output_valid_d1 =
'0')THEN
                        --    Check for correct result
                        --    Even though the output is 12 bits, the
current cordic implementation does only 8 rounds.
                        --    Therefore the accuracy is only +/- 90°/128 =
+/- 0.352
                        real_diff_of_rtl <= abs(real_arc_tan -
real_arc_tan_rtl_out);
                        IF abs(real_arc_tan - real_arc_tan_rtl_out) > 0.52
THEN
                              tb_result   <= ERROR;
                        END IF;
                  END IF;
            END IF;
      END PROCESS check_result_proc;

END ARCHITECTURE behavioral;
```

7.3 ModelSim Command File arctan_cordic_rtl_vhdl.do

```
transcript on
if {[file exists rtl_work]} {
      vdel -lib rtl_work -all
}
vlib rtl_work
vmap work rtl_work

vcom -93 -work work {../../../src/barrel_shifter.m.vhd}
vcom -93 -work work {../../../src/cordic_rom.m.vhd}
```

```
vcom -93 -work work {../../../src/arctan_cordic.m.vhd}

vcom -93 -work work {../../../sim/arctan_cordic.tb.vhd}

vsim -t 1ps -L altera -L lpm -L sgate -L altera_mf -L altera_lnsim
        -L cycloneive -L rtl_work -L work -voptargs="+acc"
arctan_cordic_tb

do ../../../sim/wave.do

view structure
view signals
run 40 us
wave zoom full
```

7.4 ModelSim Wave Command File wave.do

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate -divider Input
add wave -noupdate /arctan_cordic_tb/i_denumerator
add wave -noupdate /arctan_cordic_tb/i_numerator
add wave -noupdate -divider Output
add wave -noupdate /arctan_cordic_tb/real_arc_tan
add wave -noupdate /arctan_cordic_tb/real_arc_tan_rtl_out
add wave -noupdate /arctan_cordic_tb/real_diff_of_rtl
add wave -noupdate -divider Result
add wave -noupdate /arctan_cordic_tb/tb_result
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {3177267 ps} 0}
configure wave -namecolwidth 321
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 0
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ps
update
WaveRestoreZoom {0 ps} {42 us}
```