OST
Ostschweizer
Fachhochschule

Einführung in AMD Vivado Simulator

Inhaltsverzeichnis

1. Einfache Simulation 2
1.1. VHDL Source Code 2
1.2. Start der Simulation in Vivado 3
1.3. Vorbereitung der Simulation (Compile, Analyze, Elaborate & Launch) 3
1.4. Simulation ohne Testbench 3
1.4.1. Kontrolle des Enable Signals 4
1.4.2. Anlegen des Taktes 4
1.4.3. Änderung der Bit-Auflösung der Anzeige (Radix) 5
1.5. Schlussfolgerungen 5
2. Steuerung der Simulation durch ein Skript-File 6
2.1. Beispiel 6
2.2. Schlussfolgerungen 6
3. Simulation mit einer Testbench 7
3.1. Die Testbench 7
3.2. Regeln für die Testbench 7
3.3. VHDL Testbench für den Three_Bit_Counter 7
4. Self-Checking Testbench 10
4.1. VHDL Source Code 10
4.1.1. full_add.vhd 10
4.1.2. adder4.vhd 10
4.2. Einfache Self-Checking Testbench für adder4 11

OST
Ostschweizer
Fachhochschule

1. Einfache Simulation

Anhand eines einfachen Beispiels werden die Funktionen aufgezeigt und erklärt.

1.1. VHDL Source Code

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

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

PACKAGE three_bit_counter_pkg IS
    COMPONENT three_bit_counter IS
        PORT (
            clk     : IN  STD_LOGIC;
            enable  : IN  STD_LOGIC;
            count   : OUT UNSIGNED (2 DOWNTO 0)
        );
    END COMPONENT three_bit_counter;
END PACKAGE three_bit_counter_pkg;

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

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY three_bit_counter IS
    PORT (
        clk     : IN  STD_LOGIC;
        enable  : IN  STD_LOGIC;
        count   : OUT UNSIGNED (2 DOWNTO 0)
    );
END ENTITY three_bit_counter;

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

ARCHITECTURE behavior OF three_bit_counter IS

    SIGNAL curr_count    : UNSIGNED (2 DOWNTO 0) := (OTHERS => '1');

BEGIN
    reg_proc : PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            IF enable = '1' THEN
                curr_count <= curr_count + 1 AFTER 1 ns;
            END IF;
        END IF;
    END PROCESS reg_proc;

    count <= curr_count;

END ARCHITECTURE behavior;
```

1.2. Start der Simulation in Vivado

Man kann den Simulator in Vivado auf verschiedene Arten starten:

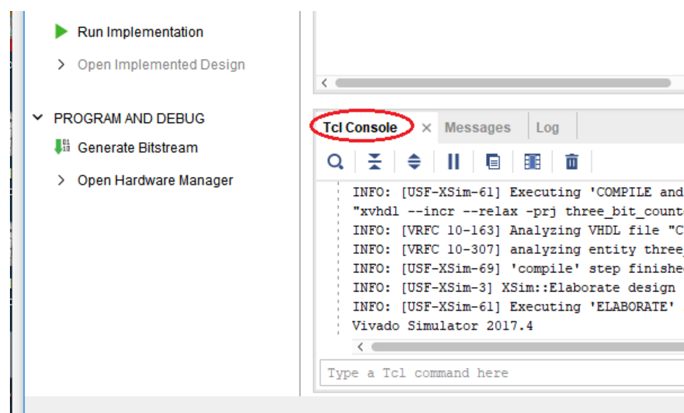
- Aus dem Menu heraus: *Flow - Run Simulation - Run Behavioral Simulation*
- Im *Flow Navigator* (linker Fensterbereich): *Simulation - Run Simulation - Run Behavioral Simulation*

1.3. Vorbereitung der Simulation (Compile, Analyze, Elaborate & Launch)

Nach dem Startbefehl geschieht sehr vieles automatisch im Hintergrund:

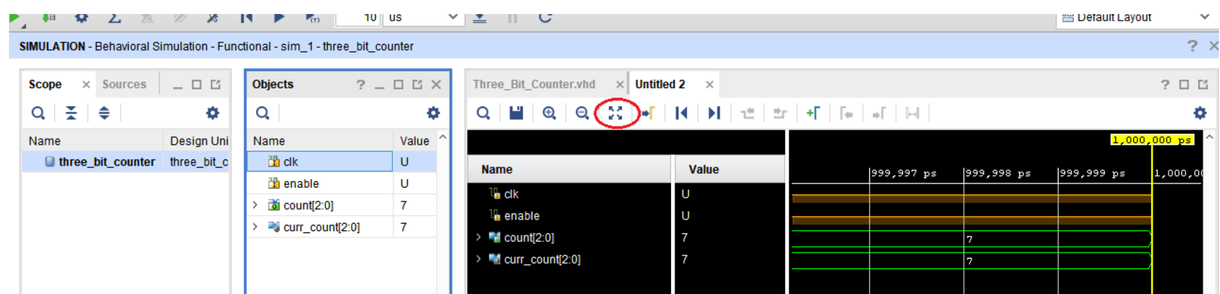
- Der Simulator inspiziert das zur Simulation ausgewählte Top-Level File und sucht sich die notwendigen «Include» Files selber zusammen
- Alle notwendigen Files werden kompiliert und analysiert (VHDL Code ok?)
- Das Design wird «elaboriert», also in ein für die Simulation effizientes Format umgesetzt
- 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 dieses Ablaufs 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.

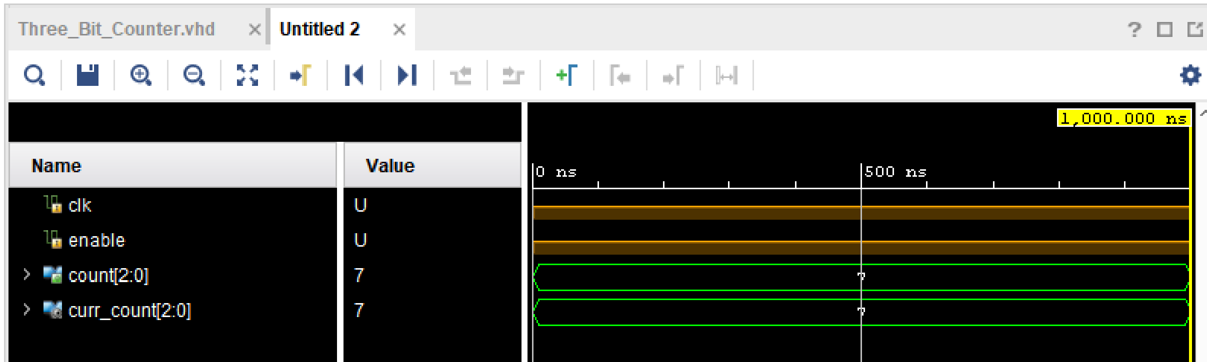


1.4. Simulation ohne Testbench

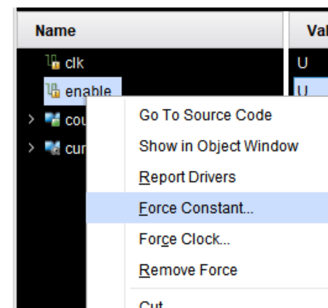
Wenn es bei der Vorbereitung der Simulation keinen Fehler gegeben hat, dann erscheint jetzt das *Waveform* Fenster:



Sie sehen dabei die Signalverläufe der Top-Level Signale während den letzten paar Picosekunden der Simulation. Klicken Sie auf den Knopf am oberen Rand der Simulation mit den 4 Pfeilen in alle Richtungen (*Zoom Fit*), um das Fenster zeitlich über die ganze Simulationsdauer zu strecken.

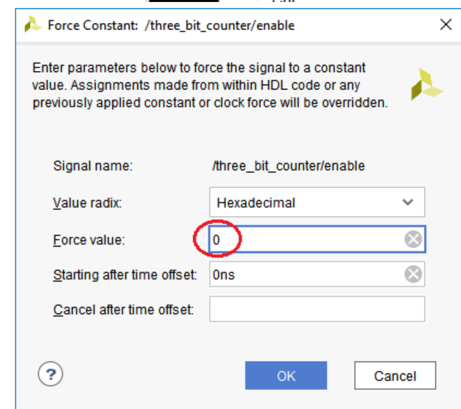


Jetzt erscheint die ganze Simulationszeit, von links 0 ns bis rechts 1'000.000 ns. Aber es läuft noch nichts.



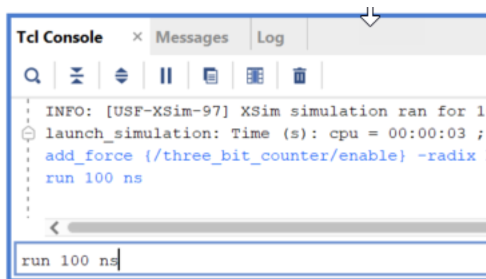
Es läuft nichts, weil wir keine Stimuli bereitgestellt haben. Die Eingangssignale zu unserem Block sind nicht definiert und zeigen das mit einem ‘U’ für *Unknown* an.

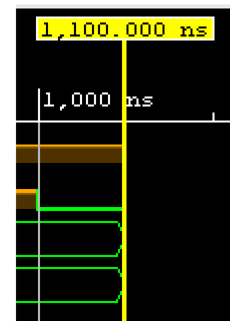
1.4.1. Kontrolle des Enable Signals



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 im neuen Fenster den Wert auf ‘0’

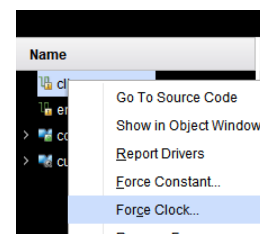
Lassen Sie die Simulation für 100 ns weiterlaufen, indem sie in der Zeile ganz unten (wo *Type a Tcl command here* steht) den Befehl *run 100 ns* eingeben, gefolgt von einem *Enter*.



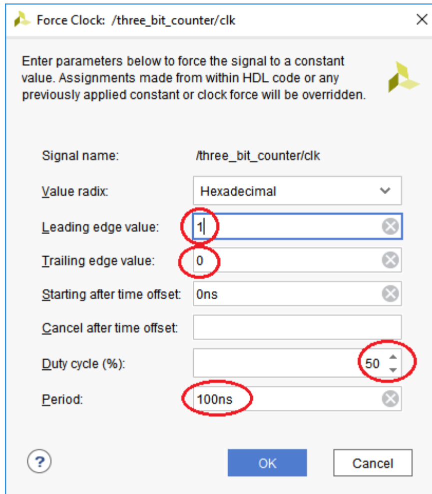


Das Ende der Simulation ist jetzt bei 1'100 ns, und während den letzten 100 ns ist das Enable-Signal jetzt grün und auf null.

1.4.2. Anlegen des Taktes

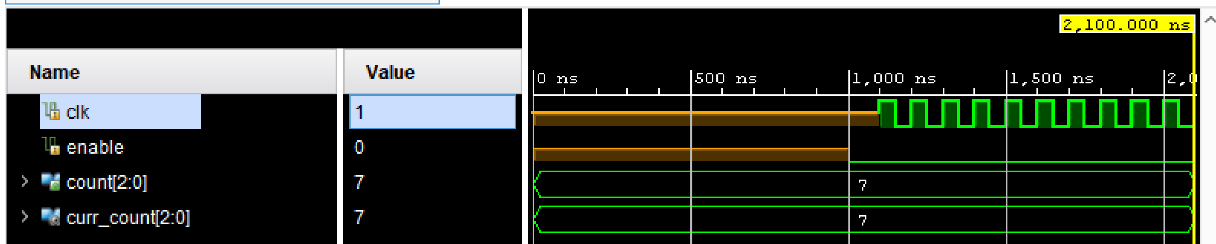


Wie bereits beim Enable-Signal klicken wir mit der rechten Maustaste auf den Namen *clk* im *Wave* Fenster, und wählen jetzt im Menu aber *Force Clock*. Es erscheint das Menu für die Takt-Definition.

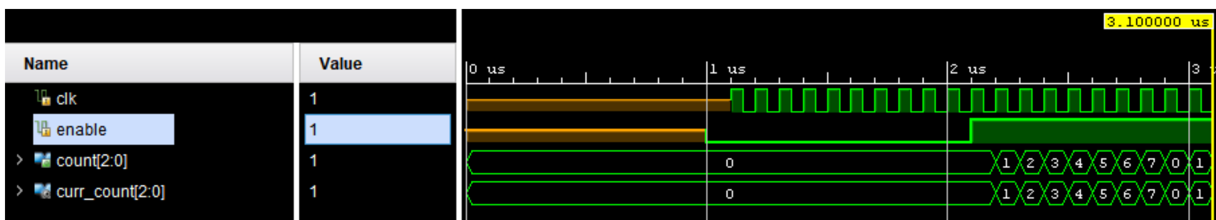


Das Taktsignal soll jeweils zuerst auf ‘1’ gehen (*Leading edge value*) und dann nach einiger Zeit auf ‘0’ gehen (*Trailing edge value*).
Belassen Sie die anderen Grundeinstellungen auf 0 ns Offset, 50 % Duty Cycle und 100 ns Periode auf 100 ns. Dies ergibt einen symmetrischen 10 MHz Takt am Signal *clk*.
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*

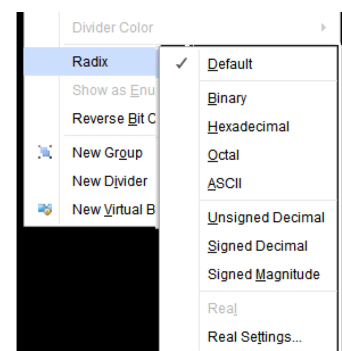
mit dem Knopf sollte es wie folgt aussehen:



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



1.4.3. Änderung der Bit-Auflösung der Anzeige (Radix)



Wählen Sie im *Wave* Fenster die beiden 3-Bit *count* Signale aus. Deren Standarddarstellung ist zurzeit der Typ *Unsigned Decimal*. Mit Rechts-Klick auf das Signal können Sie dann im Menu unter *Radix* ein anderes Format auswählen, z.B. binär oder hexadezimal.

1.5. 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 zwar intuitiv, aber nicht sehr effizient
- auch nach kleinen Änderungen oder bei jedem Neustart der Simulation muss man alles wiederholen

2. 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. Diese Befehle genügen, 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
- Führen Sie die Simulation aus

2.1. Beispiel

Dieses File sollte jetzt z.B. 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 scriptfile // Führt die Befehle im Skript-File aus. File mit vollständigem Pfad. Z.B.
Source ''C:/user/myself/OneDrive - OST/Dokumente/DT/Three_Bit_Counter/counter_stimulus.txt"

2.2. Schlussfolgerungen

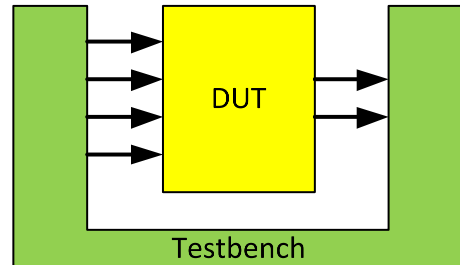
Es ist möglich, die Simulation mit einem Skript-File durchzuführen. Wenn man die Simulation wiederholen will, ist das sehr praktisch. Trotzdem gibt es folgende Nachteile:

- 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

3. Simulation mit einer Testbench

3.1. 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.



3.2. Regeln für die Testbench

Im Gegensatz zum DUT muss die Testbench nicht synthetisierbar sein, sie wird immer nur simuliert und wird nie auf dem FPGA implementiert. Deshalb kann man alle Möglichkeiten der Sprache VHDL benutzen wie:

- *WAIT FOR* Befehle mit Zeiten von Milli- oder Pico-Sekundenbereich benützen
- *LOOP* Schleifen verwenden
- Signale vom Typ Integer und Real verwenden
- Synchrone und asynchrone Prozesse nach Bedarf mischen

3.3. VHDL Testbench für den Three_Bit_Counter

Diese Testbench passt auf den Baustein Three_Bit_Counter aus dem ersten Kapitel.

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

USE work.three_bit_counter_pkg.ALL;

ENTITY three_bit_counter_tb IS
END ENTITY three_bit_counter_tb;
----------------------------------------------------------

ARCHITECTURE sim OF three_bit_counter_tb IS

    SIGNAL sl_clock, sl_enable       : STD_LOGIC := '0';
    SIGNAL usig3_count               : UNSIGNED(2 DOWNTO 0);

BEGIN
    --  ##  Unit Under Test Instantiation
    u_three_bit_counter : three_bit_counter PORT MAP (
        clk     => sl_clock,
        enable  => sl_enable,
        count   => usig3_count
    );

    --  ##  TestBench Clock Process
    clock_generator : PROCESS
    BEGIN
        sl_clock <= NOT sl_clock;
        WAIT FOR 5 ns;   --     = 50 MHz clock
    END PROCESS clock_generator;
```

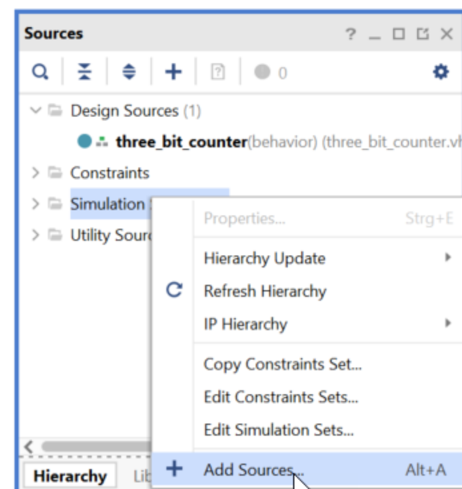
```
    --  ##    TestBench Enable Signal Generation
    stimulus_generator : PROCESS
    BEGIN
        sl_enable <= '0';
        WAIT FOR 100 ns;
        sl_enable <= '1';
        WAIT FOR 200 ns;
        --  Stop Simulation
        REPORT "End of simulation" SEVERITY FAILURE;
    END PROCESS stimulus_generator;

END ARCHITECTURE sim;
```

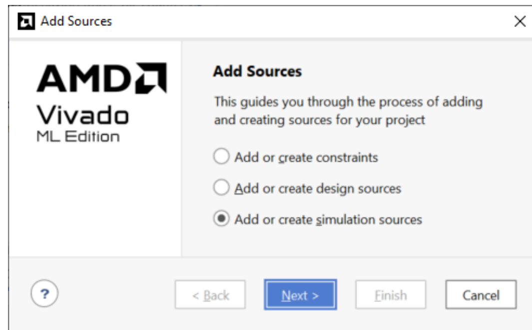
Erklärungen

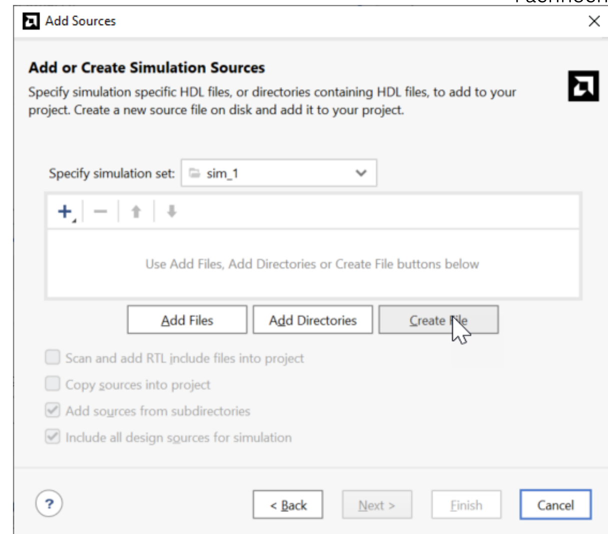
- Wie jedes VHDL Modul, braucht auch dieses eine Deklaration der «`ENTITY`». Da es aber keine Signale nach aussen gibt, ist die «`PORT LIST`» leer und kann weggelassen werden.
- Wir benötigen innerhalb der Testbench auch Signale, welche hier definiert werden. In unserem einfachen Fall sind dies nur die Schnittstellen-Signale. Bei komplexeren Testbenches können dies auch die inneren Zustandssignale, Flags, Zähler und ähnliches sein.
- Das zu testende Modul (DUT) wird hier als hierarchisch tiefer gelegene Komponente instanziiert, und die Signale der Testbench den Signalen des Moduls zugewiesen.
- 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.
- 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 und dann für 200 ns aktiv sein.
- 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 abgebrochen. Ein kleiner Nachteil dieser Art des Beendens der Simulation ist die Fehlermeldung im Report-Fenster. 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.

3.4. Testbench erzeugen und simulieren

Erstellen Sie ein neues Simulations-File mit dem Namen *three_bit_counter_tb.vhd*







Die Umgebung sollte das neue File direkt als top-level Modul erkannt haben. Falls das nicht der Fall sein sollte, können Sie das über das Kontextmenu veranlassen mit *Set as Top*.
Jetzt können Sie die Simulation wie im ersten Kapitel starten und das Wave-Fenster beobachten.

4. Self-Checking Testbench

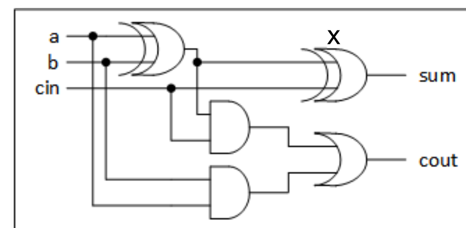
Eine sogenannte Self-Checking Testbench kann nicht nur Signale generieren, sondern kann das Resultat auch selbst überprüfen. Für dieses Beispiel verwenden wir statt «high-level Code» absichtlich einen einfachen 4-Bit Addierer mit Carry, der aus 4 einzelnen Volladdierer aufgebaut ist, welche wiederum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

4.1. VHDL Source Code

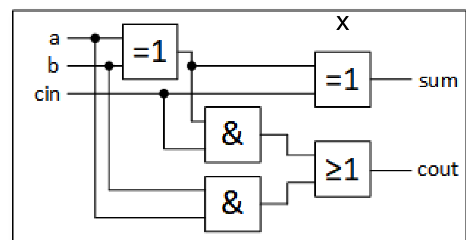
4.1.1. full_add.vhd

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.

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

PACKAGE full_add_pkg IS
    COMPONENT full_add IS
        PORT (
            a, b, cin : IN  STD_LOGIC;
            sum, cout : OUT STD_LOGIC
        );
    END COMPONENT full_add;
END PACKAGE full_add_pkg;

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

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY full_add IS
    PORT (
        a, b, cin : IN  STD_LOGIC;
        sum, cout : OUT STD_LOGIC
    );
END ENTITY full_add;

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

ARCHITECTURE rtl OF full_add IS
    SIGNAL x : STD_LOGIC;
BEGIN
    x   <= a XOR b;
    sum     <= x XOR cin;
    cout    <= (a AND b) OR (x AND cin);
END ARCHITECTURE rtl;
```



Darstellung der Logik mit U.S. Symbolen



Darstellung der Logik mit DIN Symbolen

4.1.2. adder4.vhd

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.

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

PACKAGE adder4_pkg IS
    COMPONENT adder4 IS
        PORT (
            a, b : IN  UNSIGNED(3 DOWNTO 0);
            cin  : IN  STD_LOGIC;
            sum  : OUT UNSIGNED(3 DOWNTO 0);
            cout : OUT STD_LOGIC
        );
    END COMPONENT adder4;
END PACKAGE adder4_pkg;

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

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

ENTITY adder4 IS
    PORT (
        a, b : IN  UNSIGNED(3 DOWNTO 0);
        cin  : IN  STD_LOGIC;
        sum  : OUT UNSIGNED(3 DOWNTO 0);
        cout : OUT STD_LOGIC
    );
END ENTITY adder4;

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

ARCHITECTURE struct OF adder4 IS
    SIGNAL c : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
    u_adder_0 : full_add PORT MAP (a(0), b(0), cin,  sum(0), c(0));
    u_adder_1 : full_add PORT MAP (a(1), b(1), c(0), sum(1), c(1));
    u_adder_2 : full_add PORT MAP (a(2), b(2), c(1), sum(2), c(2));
    u_adder_3 : full_add PORT MAP (a(3), b(3), c(2), sum(3), cout);
END ARCHITECTURE struct;
```

4.2. Einfache Self-Checking Testbench für adder4

Die nachfolgende Testbench instanziert einen 4 bit Addierer und prüft seine Resultate auf unterschiedliche Eingangssignale indem das zu erwartende Resultat direkt in VHDL berechnet wird.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE work.adder4_pkg.ALL;

ENTITY adder4_tb IS
END ENTITY adder4_tb;
```

OST
Ostschweizer
Fachhochschule

```
ARCHITECTURE sim OF adder4_tb IS
    SIGNAL a, b, sum    : UNSIGNED(3 DOWNTO 0);
    SIGNAL cin, cout    : STD_LOGIC;
    SIGNAL num_errors   : INTEGER;
    SIGNAL int_results : INTEGER;

BEGIN
    --##   Unit Under Test Instantiation
    u_dut : adder4 PORT MAP (
        a     => a,
        b     => b,
        cin   => cin,
        sum   => sum,
        cout => cout
    );

    --##   TB Main Process
    signal_generator : PROCESS
        VARIABLE v_result : INTEGER;
    BEGIN
        num_errors <= 0;
        FOR count_a IN 0 TO 15 LOOP
            FOR count_b IN 0 TO 15 LOOP
                FOR count_c IN 0 TO 1 LOOP
                    a <= to_unsigned(count_a, 4);
                    b <= to_unsigned(count_b, 4);
                    IF count_c > 0 THEN cin <= '1';
                    ELSE cin <= '0';
                    END IF;

                    --  Assemble result
                    WAIT FOR 1 ns;
                    v_result := to_integer(sum);
                    IF cout = '1' THEN
                        v_result := v_result + 16;
                    END IF;
                    int_results <= v_result; -- for visibility only

                    --  Verify result
                    WAIT FOR 1 ns;
                    IF v_result /= count_a + count_b + count_c THEN
                        num_errors <= num_errors + 1;
                        REPORT "Result Mismatch" SEVERITY WARNING;
                    END IF;
                    WAIT FOR 18 ns;   --    Results are stable
                END LOOP;
            END LOOP;
        END LOOP;

        IF num_errors = 0 THEN
            REPORT "Simulation without errors" SEVERITY NOTE;
        ELSE
            REPORT "Simulation with errors" SEVERITY ERROR;
        END IF;

        --  Stop Simulation
        ASSERT FALSE REPORT "End of simulation" SEVERITY FAILURE;
    END PROCESS signal_generator;

END ARCHITECTURE sim;
```

Erklärungen

- 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.
- Das Resultat des Adders `v_result` 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 und dann sofort kontrollieren.
- Der Zähler für Fehler `num_errors` 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.
- 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 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.
- 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’.
- 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.
- 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.
- 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`».
- 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.
- Wenn alle Möglichkeiten der **FOR** Schleifen durchprobiert sind, bricht die Simulation ab. Vor dem Ende des Simulator wird noch eine Zusammenfassung ausgegeben, ob die Simulation fehlerfrei war, oder nicht.

4.3. Testbench erzeugen und simulieren

Erstellen Sie das File für die Testbench wie im letzten Kapitel und starten Sie die Simulation. Die maximale Simulationszeit ist hier wesentlich länger als im letzten Kapitel, weil viel mehr Möglichkeiten getestet werden müssen. Im *Project Manager* unter *Settings* können Sie unter *Simulation* diese maximale Simulationszeit anpassen.