OST
Ostschweizer
Fachhochschule

VHDL Design Guidelines

Autoren: Prof. Laszlo Arato und Prof. Dr. Urs Graf

Erstellt am: Juli 2014
Letzte Änderung am: 21. Nov. 2024

Zum Teil wurden Beispiele und Illustrationen entnommen aus:

- «VHDL Kompakt», http:// tams-www.informatik.uni-hamburg.de
- «The Student's Guide to VHDL», P. Ashenden
- «VHDL Tutorial», https://www.vhdl-online.de/

Inhaltsverzeichnis

Hinweis zur Benutzung dieses Leitfadens:

In den folgenden Kapiteln zeigt in der Regel die linke Spalte stets Theorie und Erklärung, während in der rechten Spalte dazu passende Beispiele aufgezeigt werden.

Da wir die nächsten 2 Semester sehr viel mit Xilinx «Vivado» als Werkzeug für VHDL und FPGA arbeiten werden, richtet sich die farblichen Markierungen der Code- Beispiele nach dem Xilinx Farbschema:

VHDL Code, wie auch zitierte Schlüsselwörter im Text sind im Font «Courier New» gehalten.

VHDL Schlüsselwörter wie z.B. `ENTITY` und `BEGIN` sind zusätzlich fett und lila markiert.

Boole'sche Ausdrücke wie «`FALSE`» und «`NULL`» sind fett und rosarot markiert.

String Ausdrücke in doppelten Anführungszeichen `"any string"` sind fett und dunkelblau.

Kommentare in VHDL beginnen mit « `--  Remark` » und sind fett und hellgrau markiert.

Label wie z.B Namen der Module und Prozesse wie «`signal_generator : PROCESS`» sind fett und hellgrün. Leider wird diese Farbgebung für Labels in Vivado zurzeit noch nicht wirklich konsequent angewendet … die grüne Färbung ist daher wenig aussagekräftig.

VHDL Schlüsselwörter sind grossgeschrieben, damit sie besser von den Namen für Signale, Variablen und Funktionen unterschieden werden können.

Man könnte auch die Signalnamen grossschreiben, und die VHDL Schlüsselbegriffe klein, den VHDL selbst achten nicht auf Gross/Kleinschreibung …
… aber viele Simulationswerkzeug ändern alle Signalnamen zu Kleinschreibung. Deshalb machen «Camel Case» Namen auch keinen Sinn – was dann im Code noch gut lesbar wäre (z.B. «`myStableSignal`») wird zu einem unübersichtlichen Ausdruck «`mystablesignal`». Besser sind Underscore _ zum Trennen der Namen, wie z.B. «`my_stable_signal`».

Quickstart

Guidelines Stufe I

Zeilenumbruch: Am besten nur 80 Zeichen pro Zeile, inklusive Kommentar. Damit lässt sich der Code in nützlicher Grösse auf A4 ausdrucken.

Gross und Kleinschreibung: Alle Sprachelemente wie `ENTITY, ARCHITECTURE, BEGIN, IF, END, DOWNTO, AND, OR, NOT`, etc. werden grossgeschrieben. Alle benutzerdefinierten Namen von Instanzen und Signalen werden klein geschrieben.
Keine «MixedCase» Schreibweise (wie z.B. in C und Java).
Konstanten und Zustände in FSM dürfen grossgeschrieben werden.
IEEE-Typen, wie `STD_LOGIC, SIGNED` werden gross geschriben, aber Funktionen wie `rising_edge`, werden klein geschrieben.

(Es ist eine gute Sache, durch Gross- und Kleinschreibung Elemente der Sprache und des Designs zu trennen. Da jedoch bei ModelSim und der Synthese alle Signal-Namen sowieso auf Kleinschreibung geändert werden, ist es praktischer, gleich damit zu starten …)

Einrückung: Bei Hierarchiestufen, Schleifen und Bedingungen wird der „innere“ Teil um 4 Leerschläge nach rechts eingerückt. Das geht natürlich auch mit einer Tabulatoreinstellung von 4.

Bibliotheken: Für synthetisierbaren Code verwenden wir **nur** die IEEE Bibliotheken `STD_LOGIC_1164` und `NUMERIC_STD`.

Signal-Namen: Signale und Module sollen starke, aussagekräftige Namen erhalten. Keine VHDL, Verilog oder SystemVerilog Schlüsselwörter als Namen.

File-Pfade & Namen Keine Leerzeichen, Umlaute wie ä, ö, ü oder Sonderzeichen im Datei-Pfad oder Filenamen. VHDL stört das nicht, aber die nachfolgenden oft aus UNIX stammenden Werkzeuge wie ModelSim und Xilinx SDK!

Guidelines Stufe II

PACKAGE: Jedes Modul soll nicht nur die `ENTITY`-Deklaration enthalten, sondern auch ein `PACKAGE` mit der `COMPONENT`-Deklaration. Package-Namen heissen gleich wie die Entity, aber mit Zusatz «_pkg»

Signal-Typen: Für synthetisierbaren Code sollen nur die Signaltypen `STD_LOGIC`, `STD_LOGIC_VECTOR`, `SIGNED` und `UNSIGNED` verwendet werden.

Kein `BOOLEAN`, `INTEGER`, `NATURAL` oder gar `REAL` für synthetisierbaren ASIC oder FPGA Code, nur für Testbenches!

Guidelines Stufe III

Prefix I: I/O-Signale zu einem Modul werden mit einem vorangestellten Buchstaben markiert: `i` für Input, `o` für Output und `b` für Bidirektional. Typen mit `t_`, Records mit `r`, und Variablen mit einem `v` markiert.

Prefix II: Signaltyp und Breite werden mit Kürzel dem Namen vorangestellt:

`sl`	für `STD_LOGIC`,
`slv8`	für `STD_LOGIC_VECTOR(7 DOWNTO 0)`
`sig16`	für `SIGNED(15 DOWNTO 0)`
`usig18`	für `UNSIGNED(17 DOWNTO 0)`

Prefix I und II werden kombiniert: isig16_xxx, vsl_temp, oslv8_data

Records: Registrierte Signale im Modul werden in einem `RECORD` gebündelt. Dieses braucht man als Signale mit den Prefix `r` und `r_next`, und als Variable mit Prefix `v`

Dual Process Coding Style: Pro Modul gibt es am besten jeweils nur einen kombinatorischen und einen registrierten Prozess. Im komb. Prozess wird `r` zu `v` kopiert, verarbeitet, und am Schluss zu `r_next`.
Im registrierten Prozess wird dann `r_next` zu `r`.

Kein `WITH` Befehl
Kein `WHEN` Befehl: Diese Konstrukte, welche Ursache und Wirkung vertauscht haben, soll man meiden, da sie nur Verwirrung stiften.
(Nicht zu verwechseln mit «WHEN» als Teil des «CASE»-Befehls.)

Architektur Namen Je nach implementation heisst die Architektur

`rtl`	für synthetisierbaren VHDL Code
`struct`	für einen Block nur mit Verbindungen
`behavior`	für einen nicht-synthetisierbaren Code
`tb`	für eine Testbench-Architektur

File-Namen:

.vhd	steht für Module mit Architektur, Package und Entity.
_tb.vhd	steht für Testbench.
.p.vhd	steht für nur ein File nur mit Package Definition
.e.vhd	steht für ein File das nur Entity Deklaration enthält
.a_rtl.vhd	steht für ein File das nur die RTL Architektur enthält.

(Die letzten 3 machen nur Sinn bei sehr grossen Projekten).

Guidelines Stufe IV

Dokumentation: Jedes Modul hat ein “Datenblatt” wo seine Funktion sowie seine Eingänge und Ausgänge beschrieben sind.

Self-Checking Testbench: Jedes Modul hat eine (oder mehrere) Testbenches, welche die Funktion vollständig überprüfen.

Einleitung

Die Grenzen zwischen Hardware und Software sind am Verschwinden.

Früher war alles in der Elektronik „Hardware“, bestehend aus einzelnen Transistoren, Operationsverstärkern und Dioden, und dazwischen befinden sich all die passiven Bauelemente wie Widerstände, Kondensatoren und Spulen.

Durch die Erfindung des Mikroprozessors, die Einführung von A/D und D/A Wandlern sowie die Verarbeitung von Informationen und Signalen in Prozessoren kam eine neue Dimension ins Spiel: Software.

Ab sofort galt eine klare Trennung zwischen den Bauteilen die man „anfassen“ konnte als Hardware, und die programmierte Software um die Hardware zum „Leben“ zu erwecken. Unabhängig davon, ob es sich jetzt um einen Mikrocontroller, Mikroprozessor oder DSP handelt, wird die Software sequentiell abgearbeitet, in eine Sprache wie C, C++, C# oder Java geschrieben, im Speicher flüchtig oder nichtflüchtig abgelegt und ausgeführt.

Bald schon zeigte es sich, dass die Software ganz andere Kompetenzen erforderte als das Design der Hardware, und dass sich die Software am besten völlig losgelöst von der Hardware entfalten konnte. Die notwendige Anpassung zwischen der „losgelösten“ Software und der anwendungsspezifischen Hardware wurde „Firmware“ genannt: Firm, weil sie von der Hardware diktiert wurde und relativ starr war.

Mit dem Konzept von FPGAs und ASIC kommt nun eine ganz andere Ware ins Spiel. Bausteine, die von der Produktion her noch unbestimmt sind, und programmiert werden können, von der Struktur her jedoch überhaupt nicht der Logik der sequentiellen Software gehorchen, sondern im Grunde reine Hardware sind: fest verdrahtete Transistor-Schaltungen.

Es haben sich mit den Jahren zwei Sprachen zur Beschreibung dieser konfigurierbaren Logik durchgesetzt: Verilog und VHDL.

Verilog war zuerst da … ein de-facto Industrie-Standard der Firma „Gateway Design Auto-mation“ um das Jahr 1984. Eine Kombination aus der Hardware-Beschreibungssprache HiLo mit Elementen von C. Erst 1995 wurde Verilog von IEEE standardisiert und vereinheitlicht.

Die Entwicklung von VHDL begann zwar bereits 1981 als ein Projekt des amerikanischen Verteidigungsministeriums, um Logik einheitlich zu beschreiben, und wurde 1993 von IEEE standardisiert … jedoch hatte es sich bis dahin in der Industrie nur sehr zögerlich verbreitet.

Trotzdem, oder gerade weil es eine länger durchdachte Entwicklung durchgemacht hat, ist VHDL wesentlich „sauberer“ und klarer, und hat viele Vorteile gegenüber Verilog.

VHDL (wie auch Verilog) sind in vielen Punkten sehr ähnlich wie Software, wie zum Beispiel die Erstellung am Bildschirm, Simulation, Sprachregeln, Code Reviews, etc.
Es macht daher Sinn, Werkzeuge aus der Software-Entwicklung zu nutzen, wie zum Beispiel Syntax-Highlighting, Revision-Control, Linting, Automated Code Metrics, etc.

1. Guidelines Stufe I

Diese Stufe gilt für alle VHDL Projekte und Beispiele an der OST.

1.1. Zeilenlänge

Nur 80 Zeichen (inklusive Leerschlag) pro Zeile

Gründe …

Zusammen mit einer Zeilennummerierung passen bei einem normalen Drucker im A4-Hochformat 80 Zeichen auf eine Zeile. Wenn der Ausdruck dann keine Zeilen-Umbrüche von überlangen Zeilen hat macht es den Ausdruck entsprechend leichter lesbar. Es passen auch je nach Kopf- und Fusszeile ca. 50 Zeilen Code auf eine Seite (z.B. beim Programm Notepad++).

1.2. Gross- und Kleinschreibung bei VHDL

VHDL als Sprache ignoriert Gross- und Kleinschreibung. Es gibt jedoch Programme, welche diese Eigenschaft völlig ignorieren und pauschal alles in Kleinbuchstaben umwandeln. Dazu gehört unter anderem das Simulationsprogramm ModelSim von Mentor Graphics.

Eben weil VHDL Gross- und Kleinschreibung ignoriert sind die folgenden Signal-Definitionen identisch und werden in VHDL nicht unterschieden:

```
SIGNAL   external_clock          : STD_LOGIC;
SIGNAL   EXTERNAL_CLOCK          : STD_LOGIC;
SIGNAL   External_Clock          : STD_LOGIC;
```

Gemischte Gross- und Kleinschreibung wie z.B. bei C und Java ist in VHDL eher unglücklich, weil dann vermeintlich gut lesbare Namen bei der Simulation in ModelSim verwischt und unleserlich werden:

VHDL Editor	Andere VHDL Tools
`ExternalClock`	`externalclock`
`MyNewSyncSignal`	`mynewsyncsignal`
`TrigNextGenState`	`trignextgenstate`

Aus den oben genannten Eigenschaften von VHDL und dessen Werkzeuge ergeben sich Vorteile bei der Einhaltung gewisser Regeln:

- VHDL Sprachelemente alle grossschreiben, wie z.B. `PACKAGE, COMPONENT, ENTITY, ARCHITECTURE, CONSTANT, TYPE, SIGNAL, TO, DOWNTO, PROCESS, FUNCTION, BEGIN, END, IF, THEN, FOR, LOOP`
- Namen von Komponenten, Signalen und Variablen nur mit Kleinbuchstaben schreiben
- Keine «gemischten» Namen aus Gross- und Kleinbuchstaben verwenden
- Zur Trennung von Bezeichnern in Namen das Zeichen «_» (Underscore) verwenden
  ACHTUNG: Zwei «_» (Underscore) hintereinander sind in VHDL **verboten**!
- IEEE Definitionen werden auch gross geschrieben, wie z.B. `STD_LOGIC`, `STD_LOGIC_VECTOR`, `SIGNED`, `UNSIGNED`.

Aber Funktionen wie `rising_edge`, etc. werden klein geschrieben.
(Technisch gesehen sind dies nicht Teile von VHDL, sondern kamen erst später dazu).

- Zustände von FSM (Final-State-Machine) Kodierung kann man grossschreiben, wie z.B.
  `TYPE t_fsm_states IS (INIT, START, CALC, WAIT);`

Gründe …

Übersichtlichkeit und Lesbarkeit sind ganz wichtig um effizienten Code zu schreiben und Code zu lesen.

Durch das Auseinanderhalten von Schlüsselwörtern der Sprache und den eigenen Signal-Namen wird der Code übersichtlicher. Dies wirkt sich vor allem bei Teamwork aus, wo im Rahmen einer Fehlersuche oder eines Design-Reviews „fremde“ Personen den Code lesen und erfassen müssen.

Natürlich hilft auch die Einfärbung der Textstellen mit Editoren wie Notepad++, aber all dies geht wieder verloren, wenn man den Code ausdruckt.

Beispiel:

```
PACKAGE my_design_pkg IS
    COMPONENT my_design
        PORT (
            isl_input_signal    : IN  STD_LOGIC;
            osl_output_signal   : OUT STD_LOGIC;
        );
    END COMPONENT my_design;
END PACKAGE my_design_pkg;
```

1.3. Einrücken von untergeordneten Elementen

Hier gilt es, hierarchische Zusammenhänge offensichtlich zu machen. Vier Leerschläge sind ein guter Kompromiss zwischen Übersichtlichkeit und horizontaler Platzverschwendung.

Als hierarchische Einheit wird alles gesehen, das von Schlüsselwörtern „eingerahmt“ ist.

- Zeilen zwischen den folgenden Schlüsselwörtern sollen eingerückt werden:
  - `PORT ( … );`
  - `PORT MAP ( … );`
  - `GENERIC ( … );`
  - `GENERATE … END;`
  - `BEGIN … END;`
  - `FOR … END;`
  - `IF … END;`
  - `RECORD … END;`
  - `CASE … END;`

- Definition von Konstanten, Typen und Signalen zwischen **ARCHITECTURE** und **BEGIN** sollen eingerückt werden.

- Definition von Variablen zwischen **PROCESS** und **BEGIN**, bzw. **FUNCTION** und **BEGIN** sollen eingerückt werden.

- Ausnahmsweise kann man kurze **IF … THEN** oder eine **ELSE … END;** Anweisung auf einer Zeile stehen lassen, solange es übersichtlich bleibt. Zum Beispiel

```
IF my_input = '1' THEN slv8_my_output <= x"08";
ELSE                   slv8_my_output <= x"1A"; END IF;
```

Gründe …

Übersichtlichkeit und Lesbarkeit sind ganz wichtig um effizienten Code zu schreiben und zu lesen.

Durch das systematische Einrücken werden Fehler offensichtlicher, und der Code lesbarer.

Beispiel 1:

```
main_proc : PROCESS (clock)
BEGIN
    IF rising_edge(clock) THEN
        IF start = '1' AND finished = '1' THEN
            new_start <= '1';
        ELSE
            wait_counter <= wait_counter + 1;
        END IF;
    END IF;
END PROCESS main_proc;
```

- Bei FSM (Final State Machine) Kodierung mit CASE empfiehlt es sich, die Zeilen noch viel weiter einzurücken als nur 4 Leerschläge, damit die CASE-Struktur übersichtlicher wird.

Beispiel 2:

```
…
CASE  fsm_state IS
    WHEN INIT   => --  Init output
                   counter_out <= (OTHERS => '0');
                   fsm_state <= START;

    WHEN WAIT   => --  Wait for start button
                   IF start_button = '1' THEN
                       fsm_state <= START;
                   END IF;

    WHEN START  => --  Count and wait for stop
                   wait_counter <= wait_counter + 1;
                   IF stop_button = '1' THEN
                       fsm_state <= OUTPUT;
```

```
                    END IF;

        WHEN OUTPUT => --  Update counter output value
                       counter_out <= wait_counter;
                       fsm_state <= WAIT;

        WHEN OTHERS => --  Catch all other states
                       fsm_state <= INIT;

    END CASE;
```

1.4. Bibliotheken (Libraries)

Es gibt viele verschiedene Bibliotheken mit unterschiedlichen Definitionen für logische und mathematische Funktionen, Signaltypen, etc. Einige davon stehen im Konflikt zueinander, andere sind veraltet oder grundsätzlich nicht synthetisierbar.

1.4.1. Library IEEE.STD_LOCIG_1164

Die Bibliothek «STD_LOGIC_1164» basiert auf dem Standard IEEE 1164 «Multivalue Logic Model Interoperability» von 1993 ist dies die wichtigste Bibliothek für alle FPGA und ASIC.

Inhalt:

`STD_LOGIC`	resolved `STD_ULOGIC`	Dies ist der Standard-Typ für alle Bit-Signale. `STD_ULOGIC` steht für „unresolved STD_LOGIC“ und hat früher die Designer gewarnt, wenn ein Signal gleichzeitig von 2 Quellen unterschiedlich getrieben wurde. Speziell bei FPGAs kann das heute aber gar nicht mehr vorkommen, und so hat `STD_ULOGIC` an Bedeutung verloren.
`STD_LOGIC_VECTOR`	array of `STD_LOGIC`	Dies ist der Typ für alle Multi-Bit Vektoren wie Enable-Bündel, Interrupt Vektoren, Schalter, etc. Aber NCHT für Zahlen verwenden.
`STD_ULOGIC`	unresolved logic	Hat nur noch in ganz seltenen Fällen bei ASIC Design einen Sinn. Nicht mehr in Gebrauch.
`STD_ULOGIC_VECTOR`	array of `STD_ULOGIC`	Für den Bit-Vektor gilt auch hier dasselbe wie für den Bit-Typ. Veraltet und nicht verwenden.

1.4.2. Library IEEE.NUMERIC_STD

Dies ist die gute, formelle IEEE Bibliothek für neue Projekte.
`NUMERIC_STD` nie zusammen `STD_LOGIC_ARITH` verwenden!

Inhalt:

`UNSIGNED`	array of `STD_LOGIC`	Speziell für rein positive Zahlen, wie z.B. Zähler, Timer, Pointer, etc.
`SIGNED`	array of `STD_LOGIC`	Ein Bit-Feld speziell markiert, dass es als Zahl mit Vorzeichen aufgefasst wird. Kodiert als Zweierkomplement Zahl. Verwendet z.B. für Audio-Signale, etc.

1.4.3. Library IEEE.STD_LOGIC_ARITH

Dies ist keine echte IEEE Bibliothek, aber ein de-facto Industrie-Standard von Synopsys.
Dabei gibt es jedoch Konflikte mit der echten IEEE Bibliothek `NUMERIC_STD`, und verschiedene Hardware-Hersteller haben auch unterschiedliche Interpretationen dieser Bibliotheken.

Nicht für neue Projekte verwenden.
`STD_LOGIC_ARITH` nie zusammen mit `NUMERIC_STD` verwenden!

Inhalt:

unsigned & signed	array of `STD_LOGIC` ist scheinbar dasselbe wie in der Bibliothek `NUMERIC_STD`, aber eben nur scheinbar.

1.5. Namen für Module, Signale und Variablen

Namen bestimmen, ob man sich bei einem Signal sofort etwas darunter vorstellen kann, oder ob man jedes Mal neu rekonstruieren muss, was das Signal oder Modul eigentlich macht …

- Immer starke, sinnvolle, präzise Namen verwenden, die in einem direkten Zusammenhang zur Aufgabe oder Tätigkeit stehen.
- Namen sollten nicht länger als ca. 20 Zeichen lang sein
- Namen sollten nur Kleinbuchstaben enthalten
- Namen dürfen (in VHDL) nicht mit einer Zahl beginnen, und dürfen keine doppelten «_» (Underscore) Zeichen verwenden.
- Gute Namen für **Module** bestehen meistens aus einem Verb und einem Objekt

```
ENTITY  clear_screen
ENTITY  count_transitions
ENTITY  convert_hex_to_bcd
```

- Gute Namen für **Signale** bestehen meistens aus einem Objekt und einem Typ- oder Tätigkeitsbezeichnung, wie z.B.

```
SIGNAL  vga_clock
SIGNAL  camera_start_sync
SIGNAL  lcd_hex_data
```

- Keine profanen, absichtlich irreführenden oder trivialen Namen verwenden, wie z.B.

```
ENTITY  do_something
SIGNAL  bullshit_clock
SIGNAL  foo_bar
SIGNAL  peters_signal
```

- Verwendung der Zahl 2 als Ersatz für _to_ im englischen ist oft nicht eindeutig und daher ungünstig. Z.B. ist **`sync_to_camera`** eindeutiger als **`sync2camera`**.

Gründe …

So wie der Code immer grösser und grösser wächst, wird die Übersichtlichkeit und schnelle Lesbarkeit zu einem kritischen Faktor. Speziell während der Test- und Erweiterungsphase ist es wichtig, Zusammenhänge schnell und richtig zu erfassen.

Als Autor ist man sehr rasch der erste, der von eigenen klaren und starken Namen profitiert und dadurch seine Effizienz steigern kann. Wenn der Code gut ist, wird man ihn selbst wahrscheinlich in ein paar Monaten oder Jahren wieder verwenden … und ist dann dankbar für die Klarheit!

1.6. Keine Schlüsselwörter als Signalnamen verwenden

- Keine VHDL Schlüsselworte als Modul- oder Signalnamen
- Auch keine Verilog oder SystemVerilog Schlüsselworte als Modul- oder Signalnamen

Gründe …

VHDL Schlüsselwörter können keine Signal- oder Modulnamen sein. Das ist klar.

Mixed-Mode Simulatoren (wie z.B. ModelSim) schauen (bei Lizenzierung für mehr als nur eine Sprache) nicht auf den File-Typ im Namen, sondern nur auf Schlüsselwörter. Wenn man jetzt ein Modul entwickelt, das irgendwann später in einer System-Verilog Verifikationsumgebung laufen soll, dann sollte man einfach jetzt schon zukünftige Konflikte vermeiden, und keine Verilog oder System-Verilog Schlüsselwörter verwenden. Mit Prefix I und II ist das sowieso schon gelöst …

1.6.1. Reservierte Schlüsselwörter in VHDL:

```
ABS, ACCESS, AFTER, ALIAS, ALL, AND, ARCHITECTURE, ARRAY, ASSERT,
ATTRIBUTE, BEGIN, BIT, BLOCK, BODY, BOOLEAN, BUFFER, BUS, CASE,
COMPONENT, CONFIGURATION, CONSTANT, DISCONNECT, DOWNTO, ELSE, ELSIF, END,
ENTITY, EXIT, FALSE, FILE, FOR, FUNCTION, GENERATE, GENERIC, GROUP,
GUARDED, IF, IMPURE, IN, INERTIAL, INOUT, IS, INTEGER, LABEL, LIBRARY,
LINKAGE, LITERAL, LOOP, MAP, MOD,  NAND, NATURAL, NEW, NEXT, NOR, NOT,
NULL, OF, ON, OPEN, OR, OTHERS, OUT, PACKAGE, PORT, POSITIVE, POSTPONED,
PROCEDURE, PROCESS, PURE, RANGE, REAL, RECORD, REGISTER, REJECT, RETURN,
ROL, ROR, SELECT, SEVERITY, SIGNAL, SHARED, SLA, SLI, SRA, SRL, SUBTYPE,
THEN, TO, TRANSPORT, TRUE, TYPE, UNAFFECTED, UNITS, UNTIL, USE, VARIABLE,
WAIT, WHEN, WHILE, WITH, XNOR, XOR
```

1.6.2. Reservierte Schlüsselwörter in Verilog:

(Nur Schlüsselwörter aufgelistet, die nicht auch in VHDL vorkommen, wie z.B. AND.
In der Sprache Verilog müssen alle Sprachelemente klein geschrieben sein.)

```
always, assign, buf, bufif0, bufif1, casex, casez, cmos, deassign,
default, defparam, disable, edge, endattribute,
endcase, endfunction, endmodule, endprimitive, endspecify,
endtable, endtask, event, force, forever, fork, highz0, highz1, ifnone,
initial, inout, input, join, large, macromodule, medium, module, negedge,
nmos, notif0, notif1, output, parameter, pmos, posedge, primitive, pull0,
pull1, pulldown, pullup, rcmos, realtime, reg, release, repeat, rnmos,
rpmos, rtran, rtranif0, rtranif1, scalared, signed, small, specify,
specparam, strength, strong0, strong1, supply0, supply1, table, task,
time, tran, tranif0, tranif1, tri, tri0, tri1, triand, trior, trireg,
unsigned, vectored, wand, weak0, weak1, wire, wor
```

1.6.3. Reservierte Schlüsselwörter in SystemVerilog:

(Nur Schlüsselwörter aufgelistet, die nicht schon in VHDL oder Verilog vorkommen.
In der Sprache Verilog müssen alle Sprachelemente klein geschrieben sein.)

```
alias, always_comb, always_ff, always_latch, assert_strobe, automatic,
await, before, bind, break, byte, chandle, class, clocking, const,
constraint, context, continue, cover, dist, do, endclass, endclocking,
endinterface, endprogram, endproperty, endsequence, enum, export,
extends, extern, final, first_match, forkjoin, iff, import, inside, int,
interface, intersect, join_any, join_none, local, logic, longint,
mailbox, modport, packed, priority, program, property, protected, rand,
```

```
randc, ref, resume, semaphore, sequence, shortint, shortreal, solve,
static, string, struct, super, suspend, this, throughout, timeprecision,
timeunit, typedef, union, unique, var, virtual, void, wait_order, with,
within
```

2. Guidelines Stufe II

2.1. PACKAGE, COMPONENT und ENTITY …

In VHDL muss jedes Modul zur Verwendung eine ENTITY und eine COMPONENT Deklaration besitzen. In der Regel wird die ENTITY zusammen mit der Architektur in einem File beschrieben, während die Definition der dazugehörenden COMPONENT im hierarchisch nächst höher gelegenen File als Teil der Architektur im Bereich der Signale definiert wird.

Dies ist sehr unpraktisch, wenn das gleiche Modul in mehreren Projekten und Stellen verwendet werden kann … wie es ja mit dem „Re-Use“ (Wiederverwendung von guten Elementen) eigentlich sein sollte.

Deshalb definiert man viel besser zu jedem Modul gleich von Anfang an

- die **COMPONENT**
- die **ENTITY**
- eine (oder mehrere) **ARCHITECTURE**

Wenn man die **COMPONENT** innerhalb eines **PACKAGE** definiert hat, wird sie für alle zugänglich, und muss nicht mehr in jeder höheren Hierarchiestufe neu geschrieben werden.

⇨ Der Name des **PACKAGE** soll gleich sein wie der Name der **ENTITY**, aber mit dem Post-Fix _pkg

⇨ Natürlich **muss** das **COMPONENT** den gleichen Namen wie die **ENTITY** haben.

⇨ Wenn **PACKAGE** und **ENTITY** Deklaration im gleichen File stehen, muss trotzdem vor der **ENTITY** Deklaration die Liste der notwendigen Bibliotheken (**LIBRARY** und **USE**-Statements) wiederholt werden.

Auf der nächsten Seite sind zwei Beispiele gegeben, jeweils mit einer Top-Level Entity und einem hierarchisch tiefer liegenden Block (my_and_gate).

Auf der linken Seite ist die **COMPONENT** Deklaration Teil eines **PACKAGE** für die **ENTITY** **my_and_gate**, und muss daher nicht nochmals in der nächsten Stufe (**my_logic_top**) definiert werden. Dadurch wird die Struktur von **my_logic_top** „leichter“.

Auf der rechten Seite ist die herkömmliche Art gezeigt, mit der **COMPONENT** Deklaration als Teil von **my_logic_top** .

Beispiel für die Platzierung der „COMPONENT“ Definition

Gutes Beispiel:

File my_logic_top.vhd

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

ENTITY my_triple_and IS
    PORT (
        a, b, c : IN  STD_LOGIC;
        out     : OUT STD_LOGIC
    );
END ENTITY my_triple_and;

ARCHITECTURE struct OF my_triple_and IS
    SIGNAL temp : STD_LOGIC;
BEGIN
    u1 : my_and_gate PORT MAP (
        a   => a,
        b   => b,
        out => temp
    );

    u2 : my_and_gate PORT MAP (
        a   => c,
        b   => temp,
        out => out
    );
END ARCHITECTURE struct;
```

Einbindung des Package der Unter-einheit

File my_and_gate.vhd

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

PACKAGE my_and_gate_pkg IS
    COMPONENT my_and_gate IS
        PORT (
            a, b : IN  std_logic;
            out  : OUT std_logic
        );
    END COMPONENT my_and_gate;
END PACKAGE my_and_gate_pkg;

--------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY my_and_gate IS
    PORT (
        a, b : IN  STD_LOGIC;
        out  : OUT STD_LOGIC
    );
END ENTITY my_and_gate;

--------------------------------
ARCHITECTURE rtl OF my_and_gate IS
BEGIN
    comb_proc : PROCESS (a,b)
    BEGIN
        IF a = '1' AND b = '1' THEN
            out <= '1';
        ELSE
            out <= '0';
        END IF;
    END PROCESS comb_proc;
END ARCHITECTURE rtl;
```

Library Definition nur für das Package

Library Definition für Entity und Package

Schlechtes Beispiel:

File my_logic_top.vhd

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

ENTITY my_triple_and IS
    PORT (
        a, b, c : IN  STD_LOGIC;
        out     : OUT STD_LOGIC
    );
END ENTITY my_triple_and;

ARCHITECTURE struct OF my_triple_and IS

    COMPONENT my_and_gate IS
        PORT (
            a, b : IN  STD_LOGIC;
            out  : OUT STD_LOGIC
        );
    END COMPONENT my_and_gate;

    SIGNAL temp : STD_LOGIC;
BEGIN
    u1 : my_and_gate PORT MAP
        a   => a,
        b   => b,
        out => temp
    );

    u2 : my_and_gate PORT MAP
        a   => c,
        b   => temp,
        out => out
    );
END ARCHITECTURE struct;
```

Einbindung des Package der Unter-einheit

Component Definition muss bei jeder Instanziierung jedes Mal wieder eingegeben werden ...

File my_and_gate.vhd

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

ENTITY my_and_gate IS
    PORT (
        a, b : IN  STD_LOGIC;
        out  : OUT STD_LOGIC
    );
END ENTITY my_and_gate;

--------------------------------
ARCHITECTURE rtl OF my_and_gate IS
BEGIN
    comb_proc : PROCESS (a,b)
    BEGIN
        IF a = '1' AND b = '1' THEN
            out <= '1';
        ELSE
            out <= '0';
        END IF;
    END PROCESS comb_proc;
END ARCHITECTURE rtl;
```

2.2. Einschränkung bei den Signal-Typen

Nicht alle Signal-Typen von VHDL sind gleich effizient und praktisch im Einsatz. Damit der Code synthetisierbar ist, empfehlen sich folgende Einschränkungen:

Nur die folgenden Signal-Typen werden verwendet:

- `STD_LOGIC` Für alle Single-Bit Signale
- `STD_LOGIC_VECTOR` Für alle nicht-mathematischen Vektoren aus Bits, z.B. Interrupt-Signale, Schalter und Knöpfe, etc.
- `UNSIGNED` Für alle rein positiven Zahlen, wie z.B. Zähler, etc.
- `SIGNED` Für alle Zahlen mit Vorzeichen.

2.2.1. Vermeidung vom Typ „REAL“

Der Typ `REAL` stellt eine Floating-Point Zahl dar. VHDL definiert eine Mindestgenauigkeit von 64 Bit, woraus sich ein Mindestwertebereich von -10^{38} bis $+10^{38}$ ergibt. Je nach Implementation kann dies jedoch stark variieren und ist vom Benutzer nicht wirklich kontrollierbar.

Dieser Typ ist sehr gut für Testbench Funktionen und andere Code-Elemente geeignet, welche nicht synthetisiert werden. Für VHDL Code, der aber kompiliert und auf logische Gatter wie FPGA oder ASIC abgebildet werden soll, darf er nicht verwendet werden!

2.2.2. Vermeidung vom Typ „BOOLEAN“ und „BIT“

Die Typen `BOOLEAN` und `BIT` sind vergleichbar mit dem Typ `STD_LOGIC`, haben jedoch nur genau 2 Zustände: `FALSE` und `TRUE`, beziehungsweise `'0'` und `'1'`.

Für die Simulation heisst das, dass Signale von diesem Typ nie „unbekannt“ sind … sie können ja nur zwei Zustände annehmen. Der Simulator löst dieses Problem elegant, indem einfach eine Annahme gemacht wird …

… ganz egal, ob das den Gedanken des Designers entspricht, oder nicht. Dadurch können schwächen im Design verborgen bleiben und erst viel später zu Problemen führen.

Wenn ein `STD_LOGIC` Signal nicht initialisiert wird, dann wird es als **`U`** für «Unknown» dargestellt … alle davon abhängigen Signale sind dann auch unbestimmt, und es fällt auf.
Gerade deshalb kennt `STD_LOGIC` eben neben den Zuständen **`1`** und **`0`** auch **`-`** (Dont Care), **`X`** («Undefined») **`U`** («Unknown»), **`H`** («Weak High»), **`L`** («Weak Low»), **`W`** («Weak»), und **`Z`** («High Impedance»). Nicht so aber bei `BOOLEAN` und `BIT`.

2.2.3. Vermeidung vom Typ «NATURAL», «INTEGER» und «POSITIVE»

Auch hier hat der Designer praktisch keine Kontrolle über die effektive Implementation.

Bei Quartus wird der Typ `INTEGER` immer als 32-Bit Zahl implementiert.
Bei Quartus wird der Typ `NATURAL` immer als 31-Bit Zahl implementiert.

Bei Quartus wird der Typ `POSITIVE` immer als 31-Bit Zahl implementiert.
Ausserdem kennt `POSITIVE` die wichtige Zahl „0“ nicht!

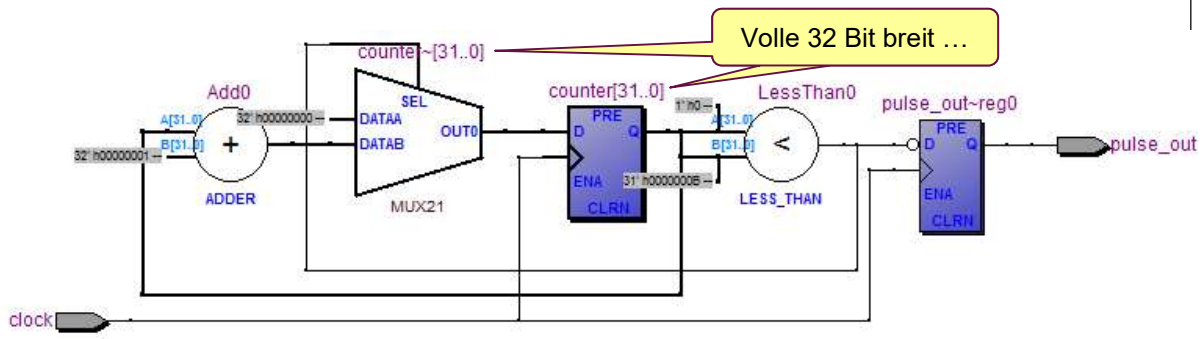
Beispiel mit dem Typ „`INTEGER`“ …

```
ARCHITECTURE demo OF bad_fsm IS
    SIGNAL counter : INTEGER := 0;
BEGIN

    main_proc : PROCESS (clock)
    BEGIN
        IF rising_edge(clock) THEN
            IF counter < 11 THEN
                counter <= counter + 1;
                pulse_out <= '0';

            ELSE
                counter <= 0;  -- Go by default to 0
                pulse_out <= '1';
            END IF;
        END IF;
    END PROCESS main_proc;
END ARCHITECTURE demo;
```

Dies ist ein einfacher Zähler bis 11, der dann wieder bei 0 anfängt. Bei jeder Runde generiert er einen kurzen Puls am Ausgang. Es ist nichts grundsätzlich falsch, es funktioniert, aber …



Im „RTL netlist viewer“ von Quartus erkennt man jedoch, dass „counter“ mit vollen 32 Bits implementiert wurde, obwohl der Zähler nie über 11 (= 4 Bits) hinauskommt. Ausserdem sieht man auch, dass für das Inkrementieren des Zählers entsprechend ein 32-Bit Addierer verwendet wird. Dazu kommt noch, dass die „Less Than“ Funktion auch aus einem 32-Bit Subtrahierer besteht.

Alles in allem wenig effizient … mit 43 Logik-Elementen.
Mit dem Typ `NATURAL` oder `POSITIVE` statt `INTEGER` sind es sogar 72 Logik-Elemente!

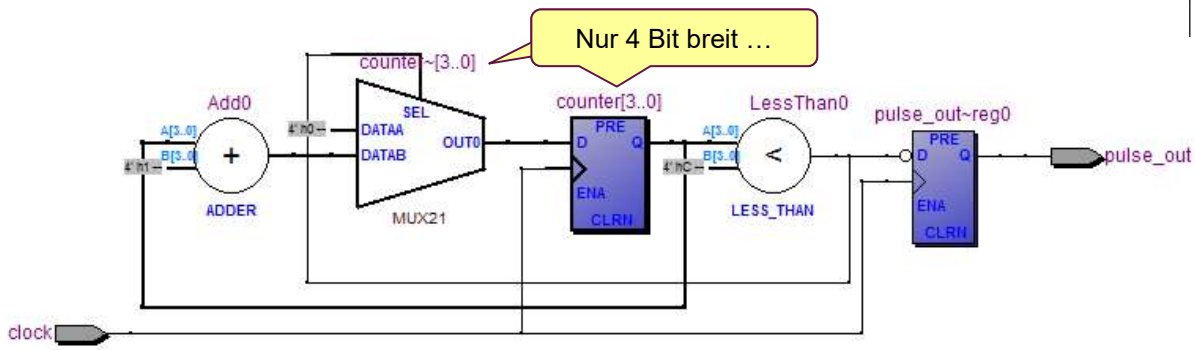
Gleicher Zähler wie vorher, aber mit dem Typ „`UNSIGNED(3 DOWNTO 0)`“ …

```
ARCHITECTURE demo OF good_fsm IS
    SIGNAL counter : UNSIGNED(3 DOWNTO 0) := (OTHERS => '0');
BEGIN

    main_proc : PROCESS (clock)
    BEGIN
        IF rising_edge(clock) THEN
            IF counter < 11 THEN
                counter <= counter + 1;
                pulse_out <= '0';

            ELSE
                counter <= (OTHERS => '0');  -- Go by default to 0
                pulse_out <= '1';
            END IF;
        END IF;
    END PROCESS main_proc;
END ARCHITECTURE demo;
```

Wieder ein einfacher Zähler bis 11, der dann wieder bei 0 anfängt. Bei jeder Runde generiert er einen kurzen Puls am Ausgang. Mit Typ `UNSIGNED` erreicht man eine effiziente Implementation:



Durch die einfache Umstellung von `INTEGER` auf `UNSIGNED(3 DOWNTO 0)` kann man hingegen den Aufwand **von 43 auf nur 5 Logik-Elemente** reduzieren(!!!).

3. Guidelines Stufe III

3.1. Prefix I

Prefix I dient dazu, die Verwendung von Signalen zu markieren.

Bei I/O Signalen ist dies besonders nützlich, da die Richtungen von Schnittstellen-Signalen eines Moduls auf der nächst höheren Ebene bei der Instanziierung von VHDL aus nicht mehr sichtbar sind. Dazu dienen die Buchstaben «i», «o» und «b»

Innerhalb eines Moduls ist es auch nützlich, wenn man Variablen und Konstanten leicht erkennen kann. Dazu dienen die Buchstaben «v» und «c».

Übersicht:

i	für Eingangssignale zu einem Modul, Procedure oder Function	(Input)
o	für Ausgangssignale aus einem Modul oder Procedure	(Output)
b	für Bidirektionale Eingangs- und Ausgangssignale.	(Inout)
c	für Konstanten	
v	für Variablen	

Diese Buchstaben werden dem Prefix-II direkt vorangestellt.

Gründe …

Besonders bei Modulgrenzen hilft es bei der Instanziierung enorm, wenn man weiss, in welche Richtung die Signale laufen, ob sie Ein- oder Ausgänge sind. Das Verbindungs-Symbol « => » hilft hier gar nicht. Wenn man die Richtung von Signalen kennt, weiss man auch bei «quick-n-dirty» provisorischen Einbindungen, welche Signale man z.B. offenlassen kann, und welche man unbedingt anschliessen sollte.

Wenn man den Code gerade schreibt, ist noch alles im Kopf präsent, und solche «offensichtlichen» Hinweise erscheinen als Zeit- und Platzverschwendung. Wenn man jedoch den eigenen Code nach ein paar Wochen, Monaten oder Jahren wieder anschauen muss, helfen diese kleinen Hinweise den Code schneller wieder zu verstehen und zu überblicken …

… so ist es zum Beispiel bei grösseren Projekten und Modulen richtig mühsam, im Nachhinein die Signalrichtungen und Zusammenhänge zu erkennen.

3.2. Prefix II

Prefix II gibt einen Hinweis auf den Typ und die Bit-Breite eines Signals:

sl_	Typ `STD_LOGIC`
usl_	Typ `STD_ULOGIC`
slv8_	Typ `STD_LOGIC_VECTOR`, mit Anzeige der Bitbreite (hier 8-Bits)
uslv16_	Typ `STD_ULOGIC_VECTOR`, mit einer Breite von 16 Bits
sig8_	Typ `SIGNED`, mit 1 Vorzeichen-Bit und 7 Bit Magnitude
usig32_	Typ `UNSIGNED` ohne Vorzeichen-Bit und mi 32 Bit Magnitude
t_	Typen-Definition
r_	Record
a_	Array

Postfix I und II werden mit einem «Underscore» vom Rest des Namens getrennt.

Gründe …

Bei der Umwandlung von Signalen erleichtert es die Arbeit, wenn man den Signaltyp schon mit dabei hat. Werkzeuge wie Eclipse zeigen die Definition von Signalen bereits an, aber haben noch für VHDL andere wesentliche Nachteile …

Studenten die diese Regeln beachtet haben, äusserten sich nach einer kurzen Eingewöhnungsphase in der Regel sehr positiv dazu. Die Anderen erkennen später die Notwendigkeit, das in ihrem Code rückwirkend zu standardisierten …

3.3. Postfix

Postfix markiert gewisse Eigenschaften eines Signals, ähnlich wie bereits Prefix-I und II. Es macht dies jedoch am Ende des Namens. Zurzeit sind nur 2 sinnvolle Arten definiert:

_n	«Active low signal» für Signale wie Chip-Enable-NOT, Read-NOT, Write-NOT, etc.

Gründe …

Es ist sehr sinnvoll, «active low» Signale auch entsprechend zu kennzeichnen. Besonders bei Logik-Verknüpfungen mit AND und OR werden dann die Zusammenhänge offensichtlicher.

_ena	Verwendung von «_ena» als Abkürzung von «Enable». Dabei soll darauf geachtet werden, dass immer «_ena» geschrieben wird, und nicht nur «_en». Die Abkürzung «_en» ist nicht eindeutig, ob das Signal jetzt «acitve high» (enable) oder als «active low» (enable not) gilt. Bei Namen mit «_ena» bzw. «_ena_n» ist es dann klar.

Gründe …

Nehmen wir an, wir finden in einem fremden Code das Signal «memory_en» …
… Ist dieses Signal jetzt «active high», oder «active low» … ?

3.4. Dual Process Coding Style

(Dieser Abschnitt kommt ursprünglich aus dem Dokument «vhdl2proc.pdf» von Jiri Gaisler)

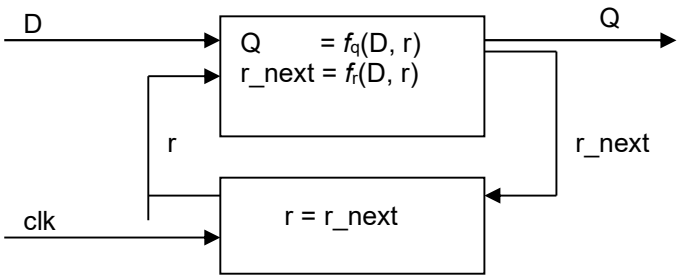
Der grösste Unterschied zwischen VHDL und einer Standard-Programmiersprache wie C ist, dass VHDL gleichzeitig auszuführende Befehle erlaubt, welche über Events ausgelöst werden, und nicht so sequentiell abgearbeitet werden, wie sie geschrieben sind. Dies bildet die tatsächlichen Vorgänge in Hardware besser ab, doch bleibt es eine Herausforderung, dies richtig zu verstehen und zu analysieren wenn die Zahl der gleichzeitig ablaufenden Vorgänge zu gross wird (so ab 50….).

Andererseits sind Ingenieure gewohnt sequentiell geschriebene Programme durchzudenken. Der Fluss von «vorher», «jetzt» und «nachher» ist einfacher und verständlicher.

Die zwei-Prozess-Methode bildet eine einheitliche Art von VHDL Programmierung, welche die Parallelität von VHDL mit den Vorteilen einer sequentiellen Programmierung verbindet.

Jedes Modul oder Einheit hat dabei genau 2 Prozesse: einen kombinatorischen Prozess für alle asynchronen Vorgänge, und einen registrierten Prozess mit allen Registern.

Dabei wird der Algorithmus in sequentieller Art (nicht-gleichzeitig) im kombinatorischen Prozess kodiert, während sich im registrierten Prozess nur Register (Flip-Flops) befinden.



Die Figur zeigt die schematische Darstellung eines Moduls mit 2 Prozessen. Alle Eingänge sind mit „D“ markiert und mit dem kombinatorischen Prozess verbunden. Registrierte Signale kommen als „r“ in den kombinatorischen Prozess, und verlassen diesen als „r_next“.

Die Funktion des kombinatorischen Prozesses kann mit 2 Gleichungen beschrieben werden:

$$Q = fq(D, r)$$
$$r\_next = fr(D, r)$$

Zusammen mit dem registrierten Prozess, welcher praktisch nur aus Flip-Flops besteht, genügen diese beiden Prozesse, um die gesamte Funktionalität des Moduls auszudrücken.

Der Code wird noch wesentlich kompakter und übersichtlicher, wenn man Records verwendet.

3.5. Beispiel für Dual-Coding Style ohne Records

Das Beispiel zeigt einen 8-Bit Zähler mit Wrap-around und mit Start- und Stopp Funktion.

Mit einem Start-Signal kann man den Zähler starten, und er zählt so lange, bis ein Signal auf dem Stopp-Knopf kommt. Wenn beide Signale gleichzeitig kommen, dann „gewinnt“ das Start-Signal.

Durch die Implementation hat das Start-Signal gewollt Priorität über das Stopp Signal.

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

PACKAGE counter8_pkg IS
    COMPONENT counter8 PORT (
        isl_clock             : IN  STD_LOGIC; --  System Signals
        isl_reset             : IN  STD_LOGIC;
        isl_start             : IN  STD_LOGIC; --  counter control
        isl_stop              : IN  STD_LOGIC; --
        ousig8_count_value    : OUT UNSIGNED(7 DOWNTO 0)
    );
    END COMPONENT counter8;
END PACKAGE counter8_pkg;

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

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

ENTITY counter8 IS
    PORT (
        isl_clock             : IN  STD_LOGIC; --  System Signals
        isl_reset             : IN  STD_LOGIC;
        isl_start             : IN  STD_LOGIC; --  counter control
        isl_stop              : IN  STD_LOGIC; --
        ousig8_count_value    : OUT UNSIGNED(7 DOWNTO 0)
    );
END ENTITY counter8;

--------------------------------------------------------------------------
```

```
ARCHITECTURE rtl OF counter8 IS

    SIGNAL usig8_counter_r, usig8_counter_r_next : UNSIGNED(7 DOWNTO 0);
    SIGNAL sl_counting_r, sl_counting_r_next     : STD_LOGIC;

    BEGIN

    --  ##  Combinatorial process
    --  ##############################
    counter8_comb_proc : PROCESS (isl_reset, isl_start, isl_stop,
                                  usig8_counter_r, sl_counting_r)

        VARIABLE vusig8_counter          : UNSIGNED(7 DOWNTO 0);
        VARIABLE vsl_counting            : STD_LOGIC;
    BEGIN
        --  Keep variables stable at all times
        vusig8_counter  := usig8_counter_r;
        vsl_counting    := sl_counting_r;

        --  Counter control
        IF isl_start = '1' THEN
            vsl_counting      := '1';
        ELSIF isl_stop = '1' THEN
            vsl_counting      := '0';
        END IF;

        --  Counter
        IF sl_counting_r = '1' THEN
            vusig8_counter    := usig8_counter_r + 1;
        END IF;

        --  Reset condition last ... to give highest priority
        IF isl_reset = '1' THEN
            vsl_counting      := '0';
            vusig8_counter    := (OTHERS => '0');
        END IF;

        --  Copy variables to signals
        usig8_counter_r_next  <= vusig8_counter;
        sl_counting_r_next    <= vsl_counting;
    END PROCESS counter8_comb_proc;

    --  ##  Register process
    --  ########################
    counter8_reg_proc : PROCESS (isl_clock) BEGIN
        IF rising_edge(isl_clock) THEN
            usig8_counter_r   <= usig8_counter_r_next;
            sl_counting_r     <= sl_counting_r_next;
        END IF;
    END PROCESS counter8_reg_proc;

    --  Generate output signals
    ousig8_count_value        <= usig8_counter_r;

END ARCHITECTURE rtl;
```

3.6. Record Typen

(Dieser Abschnitt basiert ebenfalls auf dem Dokument «vhdl2proc.pdf» von Jiri Gaisler.)

Auch wenn die Methode des «Dual Coding Style» recht elegant ist, wird dies mit realen Signalen und Bussen rasch komplex und unübersichtlich. In der Grafik auf der vorgehenden Seite stehen die Bezeichnungen «r» und «r_next» für ein Bündel von Signalen, das aus dem FSM (Final State Machine) Zustand, registrierten Ausgängen und internen registrierten Variablen besteht, wie zum Beispiel Zähler, Flags, Konfigurations-Bits, Start- und Stopp-Einstellungen und noch vielem mehr.

Entsprechend gibt es eine lange Liste bei der Definition der Signale z.B. mit dem Prefix oder Postfix «r», und dann auch noch dasselbe für «r_next».

Andererseits gibt es in VHDL das Konstrukt «Record», was im Grunde einer Bündel-Definition von Signalen gleichkommt. Man muss dabei nur zwei kleine Details beachten, damit diese Records wirklich synthetisierbar bleiben:

- Alle Elemente des Records müssen statisch und zum Zeitpunkt der Übersetzung bekannt sein. Zum Beispiel kann die Busbreite nicht dynamisch verändert werden.

- Bei der Verwendung für Schnittstellen müssen alle Elemente eines Records die gleiche «Richtung» aufweisen. Man darf also nicht Signale vom Typ «IN» mit Signalen vom Typ «OUT» mischen … sondern muss allenfalls 2 Records definieren.

Wenn man dies berücksichtigt, ist alles sehr einfach und sehr sauber.

Man greift mit dem «·» Operator (Punkt) auf jedes Element eines Records zu.

Beispiel …

```
...

ARCHITECTURE rtl OF demo_1 IS

    TYPE   t_demo_1 IS RECORD
        usig8_sync_counter  : UNSIGNED (7 DOWNTO 0);
        sl_sync_start       : STD_LOGIC;
    END RECORD t_demo_1;

    SIGNAL r, r_next        : t_demo_1;

BEGIN

    r_next                      <= r;        --  Ganzes Record wird kopiert
    r_next.usig8_sync_counter   <= x"02";    --  Nur der Zähler wird gesetzt
    r_next.sl_sync_start        <= '0';      --  Nur ein Bit im Record

...
```

3.7. Variablen statt Signale im kombinatorischen Prozess

(Auch dieser Abschnitt basiert auf dem Dokument «vhdl2proc.pdf» von Jiri Gaisler.)

In einem Prozess kann ein Signal mehrfach zugewiesen werden. Dabei ist dann aber nur letzte Zuweisung gültig ist, und sie wird erst am Ende des Prozesses ausgeführt. Soweit so gut.

Beispiel Binärer Teiler durch 10 (2 x 5) – aber mit 2 schweren Fehlern!

```
ARCHITECTURE rtl OF bad_binary_divide IS

    TYPE t_registers IS RECORD
        usig4_counter       : UNSIGNED (7 DOWNTO 0;
        sl_slow_clock       : STD_LOGIC;
    END RECORD t_registers;

    SIGNAL r, r_next        : t_registers := (
                                  usig4_counter       => (OTHERS => '0')
                                  sl_slow_clock       => '0'
                              );

BEGIN

    --  ##  Combinatorial process
    comb_proc : PROCESS (r)
    BEGIN
        r_next.usig4_counter <= r.usig4_counter + 1;
        IF r_next.usig4_counter = 5 THEN
            r_next.sl_slow_clock <= NOT r.sl_slow_clock;
        END IF;
    END PROCESS binary_divide_proc;

    --  ##  Register process
    comb_proc : PROCESS (isl_clock)
    BEGIN
        IF rising_edge(isl_clock) THEN r <= r_next END IF;
    END PROCESS binary_divide_proc;

    --  ##  Output Assignments
    osl_slow_clock  <= r. sl_slow_clock;

END ARCHITECTURE rtl;
```

Obwohl unmittelbar in der vorherigen Zeile inkrementiert, habem hier alle Signale in «r_next» immer noch den «alten» Wert … bis zum Prozess Ende!

Diese Zeile wird nur bei jedem 5. Durchlauf «getroffen». Während den anderen Durchläufen ist der Wert unbestimmt …

Grundsätzlich ist dies ein gutes Beispiel, wie aus einem schnellen Taktsignal ein Langsames gemacht werden kann. «isl_clock» wird dabei um den Faktor 10 reduziert.

Die Problematik ergibt sich bei der Bedingung «`IF r_next.usig4_counter = 5 THEN ...`», weil in der Zeile zuvor r_next.usig4_counter zwar erhöht wurde, dies aber erst am Ende des Prozesses zur Ausführung kommt. Obwohl man hier in «klassischer» Programmierung den aktuellen Wert erwarten würde, erhält man noch den «alten» Wert.

Aber was noch viel, viel schlimmer ist: «r_next.sl_slow_clock» erhält nur jeden 5. Durchgang überhaupt einen Wert zugewiesen. Das Synthesis-Werkzeug macht daraus entweder ein Latch oder aber eine kombinatorische Schleife – und beides ist nicht was wir eigentlich wollten!

In einem rein kombinatorischen Prozess kann es leicht vorkommen, dass ein Signal nicht in jedem der verschachtelten `IF … THEN … ELSE … END IF;` Pfad gebraucht wird und so nicht immer einen expliziten Wert erhält. In diesem Fall nimmt der VHDL Compiler an, dass der Benutzer den alten Wert über eine Taktflanke hinaus behalten will … und fügt in der Schaltung ein **Latch** ein. Dies ist im Beispiel der Seite 25 geschehen.

Aus dem Verhalten und den Warnungen sind diese Fehler nur sehr schwer zu identifizieren!
Beim nächsten Beispiel sind die Fehler von Seite 25 korrigiert:

Beispiel Binärer Teiler durch 10 (2 x 5) – ohne Fehler!

```
ARCHITECTURE rtl OF good_binary_divide IS

    TYPE t_registers IS RECORD
        usig4_counter       : UNSIGNED (7 DOWNTO 0;
        sl_slow_clock       : STD_LOGIC;
    END RECORD t_registers;

    SIGNAL r, r_next        : t_registers := (
                                usig4_counter       => (OTHERS => '0')
                                sl_slow_clock       => '0'
                              );

BEGIN

    --  ##  Combinatorial process
    comb_proc : PROCESS (r)
    BEGIN
        r_next <= r;

        r_next.usig4_counter <= r.usig4_counter + 1;
        IF r.usig4_counter = 4 THEN  --<= Vergleiche nur auf r statt
r_next
            r_next.sl_slow_clock <= NOT r.sl_slow_clock;
        END IF;
    END PROCESS binary_divide_proc;

    --  ##  Register process
    comb_proc : PROCESS (isl_clock)
    BEGIN
        IF rising_edge(isl_clock) THEN r <= r_next END IF;
    END PROCESS binary_divide_proc;

    --  ##  Output Assignments
    osl_slow_clock  <= r. sl_slow_clock;

END ARCHITECTURE rtl;
```

Die Signale in « r_next» erhallten so immer einen stabilen Wert

Vergleiche immer nur auf stabile Signale in «r», nie auf Signale in «r_next»!

All diese Probleme kann man elegant vermeiden, wenn man im kombinatorischen Prozess nur mit Variablen arbeitet. Am Anfang des Prozesses weist man alle Signale aus «r» Variablen zu, und am Ende des Prozesses werden die Variablen an «r_next» weitergegeben.

Auf diese Weise kombiniert man die Vorteile der beiden Typen zu einem sauberen Stil:

Eigenschaften von Signalen	**Eigenschaften von Variablen**
- Existieren auch ausserhalb von Prozessen	- Existieren nur innerhalb von Prozessen werden bei Prozess-Aufruf erschaffen und am Prozessende zerstört

- Nur die letzte Zuweisung gilt - Werden parallel / gleichzeitig verarbeitet, aber erst am Ende des Prozesses!	- Beliebig Mehrfachzuweisungen möglich - Werden sequentiell abgearbeitet - Werden sofort zugewiesen, und stehen entsprechend sofort weiter zur Verfügung

Zusammen mit der Verwendung von Records wird die Sache ganz einfach und elegant, da auch ein ganzes Record dem anderen zugewiesen werden kann, und nicht jedes Signal einzeln eine Zuweisung braucht. Die Variable „v“ ist vom selben Typ wie „r“ und „r_next“.
Durch die pauschale Zuweisung am Anfang des Prozesses sind alle Teile von „v“ initialisiert.
Am Ende wird dann das unterdessen veränderte „v“ dem „r_next“ zugewiesen, und fertig.

Beispiel …

```
...
        SIGNAL r, r_next            : t_demo_2;

  BEGIN

    --  ##  Combinatorial process
    --  ##############################
    demo_2_comb_proc : PROCESS (isl_reset, r)
        VARIABLE v                  : t_demo_1;
    BEGIN
        v := r;          --  Initialize to keep variables stable at all
times

        --  Process internals ...
        --  ...

        r_next    <= v; --  Copy variables to signals
    END PROCESS demo_2_comb_proc;

...
```

3.8. Beispiel Counter8 (Kapitel 3.5) aber mit Records

```
ARCHITECTURE rtl_with_records OF counter8 IS

    TYPE t_registers IS RECORD
        usig8_counter   : UNSIGNED(7 DOWNTO 0);
        sl_counting     : STD_LOGIC;
    END RECORD t_registers;

    SIGNAL r, r_next    : t_registers;

  BEGIN

    --  ##  Combinatorial process
    --  ##############################
    comb_proc : PROCESS (isl_reset, isl_start, isl_stop, r)
    VARIABLE v          : t_registers;
    BEGIN
        v  := r;  --  Keep variables stable at all times

        --  Counter control
        IF isl_start = '1' THEN
            v.sl_counting     := '1';
        ELSIF isl_stop = '1' THEN
            v.sl_counting     := '0';
        END IF;

        --  Counter
        IF r.sl_counting = '1' THEN
            v.usig8_counter   := r.usig8_counter + 1;
        END IF;

        --  Reset condition last ... to give highest priority
        IF isl_reset = '1' THEN
            v.sl_counting           := '0';
            v.usig8_counter   := (OTHERS => '0');
        END IF;

        r_next <= v;    --  Copy variables to signals
    END PROCESS comb_proc;

    --  ##  Register process
    --  ########################
    reg_proc : PROCESS (isl_clock)
    BEGIN
        IF rising_edge(isl_clock) THEN r <= r_next; END IF;
    END PROCESS reg_proc;

      --  Output Assignments
      ousig8_count_value      <= r.usig8_counter;

END ARCHITECTURE rtl_with_records;
```

3.9. Verwendung von Records für I/O Signale

(Dieser Abschnitt basiert auf dem Dokument “vhdl2proc.pdf” von Jiri Gaisler.)

Genauso wie ein Record-Bündel hilft, die verschiedenen Signale innerhalb eines Moduls zusammenzufassen und übersichtlich zu gestalten, kann man die gleiche Methode anwenden um die Eingänge und Ausgänge eines Moduls übersichtlicher zu gestalten.

Zwingend zu beachten:

- Records müssen nach Signal-Richtung getrennt werden. Es können nicht IN und OUT oder INOUT Signale im gleichen Record nebeneinander existieren.

Praktisch zu beachten:

- Man muss nicht alle I/O Signale in ein Input- und ein Output-Record stecken … man kann diese viel sinnvoller aufteilen
- Man kann Records so gruppieren, dass die Bündel dann auf der nächst höheren Ebene als Bündel zu einem anderen Block verbunden werden können, ohne diese aufzutrennen.
- Man kann die Bündel auch nach Schnittstellen übersichtlich zusammenfassen, wie z.B. DRAM, SRAM, UART, etc.

Jiri Gaisler empfiehlt ausserdem, dass man Takt- und Reset-Signale zur Übersichtlichkeit nicht in ein Bündel integriert, sondern ausdrücklich als „normale“ explizite Verbindungen führt.

3.10. Definition von RECORDs im PACKAGE

Wenn man für Schnittstellen-Signale Records verwendet, dann muss natürlich die hierarchisch nächst höhere Einheit diesen Typen auch kennen. Die elegante Methode dafür ist es, ein solches Record nur einmal im PACKAGE zu einem Modul zu definieren, und dieses dann jeweils mit USE sowohl der nächst höheren Einheit wie auch dem eigenen Modul zur Verfügung zu stellen.

3.11. Beispiel für Dual Coding Style und Records

Das folgende Beispiel zeigt ein reales Modul zur konfigurierbaren Erzeugung von Synchronsignalen für eine VGA Schnittstelle.

Für die Ausgabe der Signale wurden 2 verschiedene Records gewählt:

- Synchronsignale, welche direkt an die VGA Schnittstelle gehen
- Adress-Information, welche die Datengenerierung oder Memory-Zugriff benötigt

Entsprechend wurden die registrierten Signale innerhalb des Moduls gruppiert.

3.11.1. VGA_addr_counter PACKAGE

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

PACKAGE vga_addr_counter_pkg IS

    CONSTANT  ci_vga_coord_width : INTEGER     := 11;  -- number of bits
    CONSTANT  ci_vga_addr_width  : INTEGER     := 2 * ci_vga_coord_width;

    TYPE t_vga_timing_record IS RECORD
         sl_sync                 : STD_LOGIC;
         sl_hsync                : STD_LOGIC;
         sl_vsync                : STD_LOGIC;
         sl_data_valid           : STD_LOGIC;
         sl_image_start          : STD_LOGIC;
    END RECORD t_vga_timing_record;

    TYPE t_vga_addr_record IS RECORD
         usig_addr               : UNSIGNED (ci_vga_addr_width-1 DOWNTO 0);
         usig_x_coord            : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0);
         usig_y_coord            : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0);
         sl_data_valid           : STD_LOGIC;
    END RECORD t_vga_addr_record;

    COMPONENT vga_addr_counter IS
        GENERIC (
            gsl_hsync_polarity   : STD_LOGIC := '1'; -- Active high

            gusig_vga_hsync_width: UNSIGNED (ci_vga_coord_width-1 DOWNTO 0)
                                 := to_unsigned(120, ci_vga_coord_width);

            gusig_h_front_porch  : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0)
                                 := to_unsigned(56, ci_vga_coord_width);

            gusig_vga_img_width  : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0)
                                 := to_unsigned(800, ci_vga_coord_width);

            gusig_h_back_porch   : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0)
                                 := to_unsigned (64, ci_vga_coord_width);

            gsl_vsync_polarity   : STD_LOGIC := '1'; --     Active high

            gusig_vga_vsync_width : UNSIGNED (ci_vga_coord_width--1 DOWNTO 0)
                                 := to_unsigned (6, ci_vga_coord_width);

            gusig_v_front_porch  : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0)
                                 := to_unsigned (37, ci_vga_coord_width);

            gusig_vga_img_height : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0)
                                 := to_unsigned (600, ci_vga_coord_width);

            gusig_v_back_porch   : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0)
                                 := to_unsigned (23, ci_vga_coord_width)
        );

        PORT (
            isl_reset            : IN  STD_LOGIC;
            isl_clock            : IN  STD_LOGIC;
            r_vga_timing_out     : OUT t_vga_timing_record;
            r_vga_addr_out       : OUT t_vga_addr_record
        );
    END COMPONENT vga_addr_counter;
END PACKAGE vga_addr_counter_pkg;
```

Die Bit-Breite ist für alle Zähler und Schnittstellen über die Konstante ci_vga_coord_width parameterisiert

Diese Signale gehen in erster Lin… zur VGA Schnittstelle

Diese Signale gehen in erster Linie zum Bildspeicher

Generische Parameter können bei der Instanziierung über-schrieben werden, oder als Default-Werte

Hier endlich die PORT Definition … sehr kompakt!

3.11.2. VGA_addr_counter ENTITY

Die generischen Parameter zur Konfiguration müssen in der Entity auch deklariert sein, sonst sind sie innerhalb der Entity unbekannt. Je nach Simulations- oder Synthese Werkzeug müssen diese auch einen expliziten Wert haben, auch wenn sie durch das Package überschrieben werden.

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

USE work.vga_addr_counter_pkg.ALL;

ENTITY vga_addr_counter IS

  GENERIC (
    gsl_hsync_polarity     : STD_LOGIC := '1';
    gusig_vga_hsync_width  : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');
    gusig_h_front_porch    : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');
    gusig_vga_img_width    : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');
    gusig_h_back_porch     : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');

    gsl_vsync_polarity     : STD_LOGIC := '1';
    gusig_vga_vsync_width  : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');
    gusig_v_front_porch    : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');
    gusig_vga_img_height   : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');
    gusig_v_back_porch     : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0):=(OTHERS =>'0');
  );

  PORT (
      isl_reset            : IN  STD_LOGIC;
      isl_clock            : IN  STD_LOGIC;
      r_vga_timing_out     : OUT t_vga_timing_record;
      r_vga_addr_out       : OUT t_vga_addr_record
  );
END ENTITY vga_addr_counter;
```

Damit die ENTITY die Definitionen des PACKAGE kennt, muss es hier mit USE Klausel eingeführt

Generische Parameter müssen auch in der ENTITY Werte haben, auch wenn diese immer vom PACKAGE oder der Instanziierung überschrieben

3.11.3. VGA_addr_counter ARCHITECTURE

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

USE work.vga_addr_counter_pkg.ALL;

ARCHITECTURE rtl OF vga_addr_counter IS

    CONSTANT cusig_pixel_per_line       : UNSIGNED (gi_vga_coord_width-1 DOWNTO 0)
                                        := gusig_vga_hsync_width
                                         + gusig_h_front_porch
                                         + gusig_vga_img_width
                                         + gusig_h_back_porch;

   CONSTANT cusig_lines_per_frame       : UNSIGNED (gi_vga_coord_width-1 DOWNTO 0)
                                        := gusig_vga_vsync_width
                                         + gusig_v_front_porch
                                         + gusig_vga_img_height
                                         + gusig_v_back_porch;
```

Dies sind aus den Parametern abgeleitete Konstanten. Da sie statisch sind, ist es einfach praktisch diese im Voraus zu berechnen, statt erst

```
    TYPE vga_addr_count_registers IS RECORD
        usig_vga_h_counter       : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0);
        usig_vga_v_counter       : UNSIGNED (ci_vga_coord_width-1 DOWNTO 0);
        sl_h_pixel_valid         : STD_LOGIC;
        sl_v_pixel_valid         : STD_LOGIC;
    END RECORD vga_addr_count_registers;

    SIGNAL r, r_next                    : vga_addr_count_registers;
    SIGNAL r_timing, r_next_timing      : t_vga_timing_record;
    SIGNAL r_addr, r_next_addr          : t_vga_addr_record;
```

Drei verschiedene Records … einer für die internen Register, und zwei für die Output-Signale zu verschiedenen Blöcken …

```
BEGIN

    --  ##   Combinatorial Process
    --  ##############################
    vga_addr_counter_comb_proc : PROCESS (isl_reset, r, r_timing, r_addr)

        VARIABLE v                      : vga_addr_count_registers;
        VARIABLE v_timing               : t_vga_timing_record;
        VARIABLE v_addr                 : t_vga_addr_record;

    BEGIN
        v                               := r;         -- Keep variables stable
        v_timing                        := r_timing;
        v_addr                          := r_addr;

        v_timing.sl_image_start         := '0';       -- Single-cycle pulse

        --  Horizontal Counter
        --  ==================
        IF r.usig_vga_h_counter < cusig_pixel_per_line - 1 THEN
            v.usig_vga_h_counter        := r.usig_vga_h_counter + 1;

            --      Clear horizontal sync
            IF r.usig_vga_h_counter = gusig_vga_hsync_width - 1 THEN
                v_timing.sl_hsync       := NOT gsl_hsync_polarity;
            END IF;

            --      Calculate horizontal address
            IF  r.usig_vga_h_counter >= gusig_h_back_porch
            AND r.usig_vga_h_counter <  gusig_h_back_porch+gusig_vga_img_width-1 THEN
                v_addr.usig_x_coord             := r_addr.usig_x_coord + 1;
                v.sl_h_pixel_valid              := '1';
            ELSE
                v.sl_h_pixel_valid              := '0';
            END IF;

        ELSE
             -- Horizontal signal wrap-around
             v.usig_vga_h_counter               := (OTHERS => '0');
             v_timing.sl_hsync                  := gsl_hsync_polarity;
             v_addr.usig_x_coord                := (OTHERS => '0');

             --  Vertical Counter
             --  ================
             IF r.usig_vga_v_counter < cusig_lines_per_frame - 1 THEN
                 v.usig_vga_v_counter   := r.usig_vga_v_counter + 1;

                 --  Clear vertical sync
                 IF r.usig_vga_v_counter = gusig_vga_vsync_width - 1 THEN
                     v_timing.sl_vsync  := NOT gsl_vsync_polarity;
                 END IF;
```

So vermeidet man Latches

```
            --  Calculate vertical address
            IF  r.usig_vga_v_counter >= gusig_v_back_porch
            AND r.usig_vga_v_counter <  gusig_v_back_porch
                                     + gusig_vga_img_height - 1 THEN
                v_addr.usig_y_coord      := r_addr.usig_y_coord + 1;
                v.sl_v_pixel_valid       := '1';
            ELSE
                v.sl_v_pixel_valid       := '0';
            END IF;

        ELSE
            --  Vertical wrap-around
            v.usig_vga_v_counter   := (OTHERS => '0');
            v_timing.sl_image_start      := '1';
            v_timing.sl_vsync            := gsl_vsync_polarity;
            v_addr.usig_y_coord          := (OTHERS => '0');
        END IF;
    END IF;

    --  Derived Signals
    --  ===============
    v_addr.usig_addr         := v_addr.usig_y_coord * cusig_pixel_per_line
                              + v_addr.usig_x_coord;

    v_timing.sl_data_valid   := v.sl_h_pixel_valid AND v.sl_v_pixel_valid;
    v_addr.sl_data_valid     := v.sl_h_pixel_valid AND v.sl_v_pixel_valid;

    v_timing.sl_sync         :=  (v_timing.sl_hsync AND gsl_hsync_polarity)
                              OR (v_timing.sl_vsync AND gsl_vsync_polarity);

    --  Reset
    --  =====
    IF isl_reset = '1' THEN
        v.usig_vga_h_counter             := gusig_pixel_per_line - 1;
        v.usig_vga_v_counter             := gusig_lines_per_frame - 1;
        --  Avoid stuck-at warning for higher pins
        v_addr.usig_addr                 := (OTHERS => '1');
    END IF;

    --  Copy variables to signals
    r_next           <= v;
    r_next_timing    <= v_timing;
    r_next_addr      <= v_addr;
END PROCESS vga_addr_counter_comb_proc;

--  ##   Register Process
--  ########################
vga_addr_counter_reg_proc : PROCESS (isl_clock)
BEGIN
    IF rising_edge(isl_clock) THEN
        r            <= r_next;
        r_timing     <= r_next_timing;
        r_addr       <= r_next_addr;
    END IF;
END PROCESS vga_addr_counter_reg_proc;

--  Output assignments
r_vga_timing_out     <= r_timing;
r_vga_addr_out       <= r_addr;

END ARCHITECTURE rtl;
```

Reset kommt ganz am Schluss … so hat es die höchste Priorität.

Alle 3 Register-Records müssen hier umkopiert werden!

Alle 3 Register-Records müssen hier umkopiert werden!

3.11.4. VGA_addr_counter Testbench

Das spezielle bei dieser Testbench ist die Ausnützung der Parameterisierung des Moduls.

Dabei werden relativ kleine Werte für die horizontalen und vertikalen Bildeigenschaften genommen, was die Simulationszeit erheblich verkürzt. Bei gleicher Pixel-Frequenz sind es dann nur noch 150 µs pro Bild statt 13.88 ms.

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

USE work.vga_addr_counter_pkg.ALL;

ENTITY vga_addr_counter_tb IS
END ENTITY vga_addr_counter_tb;

ARCHITECTURE sim OF vga_addr_counter_tb IS

SIGNAL sl_clock            : STD_LOGIC := '0';
SIGNAL sl_reset            : STD_LOGIC := '0';

SIGNAL r_vga_timing        : t_vga_timing_record;
SIGNAL r_vga_addr          : t_vga_addr_record;

BEGIN

    --  ##  Instantiate Device Under Test
    --  ##
    --  ##################################
    my_vga_addr_counter : vga_addr_counter
    GENERIC MAP (
        csl_hsync_polarity       => '1',      --     Active high polarity
        cusig_vga_hsync_width    => to_unsigned (12, ci_vga_coord_width),
        cusig_h_front_porch      => to_unsigned ( 6, ci_vga_coord_width),
        cusig_vga_img_width      => to_unsigned (80, ci_vga_coord_width),
        cusig_h_back_porch       => to_unsigned ( 8, ci_vga_coord_width),

        csl_vsync_polarity       => '1',      --     Active high polarity
        cusig_vga_vsync_width    => to_unsigned ( 6, ci_vga_coord_width),
        cusig_v_front_porch      => to_unsigned ( 7, ci_vga_coord_width),
        cusig_vga_img_height     => to_unsigned (60, ci_vga_coord_width),
        cusig_v_back_porch       => to_unsigned ( 3, ci_vga_coord_width)
    )
    PORT MAP (
        isl_reset         => sl_reset,
        isl_clock         => sl_clock,
        r_vga_timing_out  => r_vga_timing,
        r_vga_addr_out    => r_vga_addr
    );


    --  ##  sl_clock and sl_reset SIGNALs
    --  ##################################
    sl_clock <= NOT sl_clock after 10 ns;            -- 50 MHz
    sl_reset <= '1' AFTER 20 ns,'0' AFTER 60 ns;     -- Active for 2 cycles

END ARCHITECTURE sim;
```

Bekannt aus dem Package «vga_addr_counter_pkg»

Instanziierung des Moduls mit ganz eigenen (kleinen)

3.11.5. Beispiel VGA_addr_counter Instanziierung

Ganz analog zur Instanziierung in der Testbench kann dieses Modul z.B. mit Parametern für eine XGA Ausgabe eingebaut werden. Die verschiedenen Zähler sind bereits für die notwendigen 11-Bit Breite bereits konfiguriert.
Die einzige Herausforderung ist dann nur noch, das Timing für 193.16 MHz Pixelfrequenz zu erfüllen.

Instanziierung des Moduls mit ganz anderen Parametern für einen grossen Bildschirm

```
    --  ##  Instantiate XGA Timing Generator
    --  ##
    --  ####################################
    my_vga_addr_counter : vga_addr_counter
    GENERIC MAP (
        csl_hsync_polarity       => '1',      --     Active high polarity
        cusig_vga_hsync_width    => to_unsigned ( 208, ci_vga_coord_width),
        cusig_h_front_porch      => to_unsigned ( 128, ci_vga_coord_width),
        cusig_vga_img_width      => to_unsigned (1920, ci_vga_coord_width),
        cusig_h_back_porch       => to_unsigned ( 336, ci_vga_coord_width),

        csl_vsync_polarity       => '1',      --     Active high polarity
        cusig_vga_vsync_width    => to_unsigned (   3, ci_vga_coord_width),
        cusig_v_front_porch      => to_unsigned (   1, ci_vga_coord_width),
        cusig_vga_img_height     => to_unsigned (1200, ci_vga_coord_width),
        cusig_v_back_porch       => to_unsigned (  38, ci_vga_coord_width)
    )
    PORT MAP (
        isl_reset          => sl_reset,
        isl_clock          => sl_clock,
        r_vga_timing_out   => r_vga_timing,
        r_vga_addr_out     => r_vga_addr
    );
```

3.12. Vermeidung von WHEN Befehlen

Es gibt in VHDL einen Konstrukt mit Namen «WHEN», welche von der Funktion her einer `IF … THEN … ELSE … END IF;` - Folge sehr ähnlich ist, aber auch ausserhalb von einem Prozess verwendet werden kann. Nur ist dieses Konstrukt absolut nicht intuitiv: Ursache und Wirkung sind vertauscht. Deshalb führt dessen Verwendung leicht zu Fehlern.

Beispiel für den WHEN Befehl

Wirkung *Ursache*

```
ARCHITECTURE when_example OF mux4_1 IS

BEGIN

    output <= in0 WHEN (s1 & s0)="00" ELSE
              in1 WHEN (s1 & s0)="01" ELSE
              in2 WHEN (s1 & s0)="10" ELSE
              in3 WHEN (s1 & s0)="11" ELSE
              'X';

END ARCHITECTURE when_example;
```

Gleiches gilt auch für den «WITH» Befehl, den man ebenfalls meiden sollte:

Beispiel für den WITH Befehl

```
ARCHITECTURE with_example OF mux4_1 IS
```

```
SIGNAL  sel  :  STD_LOGIC_VECTOR(1 DOWNTO 0);
--
================================================================================
=
BEGIN
  sel <= s1 & s0;  -- concatenate s1 and s0
  WITH sel  SELECT
    output <= in0 WHEN "00",
              in1 WHEN "01",
              in2 WHEN "10",
              in3 WHEN "11",
              'X' WHEN OTHERS;

END ARCHITECTURE with_example;
```

Ursache

Wirkung

Es gehört zu einem guten und lesbaren Stil, dass man auf diese Konstrukte verzichtet.

3.13. VHDL Block Namen

ENTITY Name:	Der Name soll ausdrucksstark sein und die Funktion des Moduls klar benennen. Es werden nur Kleinbuchstaben verwendet, mit dem «_» Underscore Character zur Trennung von Wörtern.	
COMPONENT Name:	Ist der gleiche Name wie für die **ENTITY** .	
PACKAGE Name:	Ist der Name der **ENTITY** mit dem Zusatz **_pkg**.	
ARCHITECTURE Name:	«rtl»	Für synthetisierbare RTL Logik welche aus kombinatorischen und registrierten Prozessen besteht.
	«struct»	Für reine Verbindunges-Hierarchien. Dabei hat es in diesem Modul dann keine Prozesse oder Entscheidungen, nur “Verdrahtung”.
	«sim»	Für nicht-synthetisierbare Logik, die nur zur Simulation dient. Das gleiche wie “behavioral”, aber “kürzer”.
	«fpga»	Für FPGA-optimierte Schaltungen.
	«altera» «intel»	Für Altera FPGA-spezifische Schaltungen (z.B. Speicher oder PLL Instanziierungen).
	«xilinx»	Für Xilinx FPGA-spezifische Schaltungen (z.B. Speicher oder PLL Instanziierungen).
	«actel»	Für Actel FPGA-spezifische Schaltungen (z.B. Speicher oder PLL Instanziierungen).
	«asic»	Für ASIC-optimierte und synthetisierbare RTL Logik.
	«net»	Für bereits synthetisierte Netzlisten, die dann aus reinen Gate-Level Modulen und ihren Verbindungen bestehen.
	«mixed»	Für synthetisierbare Logik aus einem Gemisch von RTL, Verbindungen und synthetisierten Elementen.

«dummy»	Für einen leeren Platzhalter, einfach damit die anderen Module in einem Design verbunden und simuliert werden können. Idealerweise hat es keine hängenden Verbindungen, und erzeugt bei der Kompilation keine unnötigen Fehlermeldungen oder Warnungen.

3.14. Auswahl der Architektur mit «Configuration»

VHDL würde es auch erlauben, für eine `ENTITY` mehrere verschiedene `ARCHITECTURE` zu definieren und über `CONFIGURATION` auszuwählen. So könnte man z.B. bereits früh im Projekt eine «Behavioral» Implementation schreiben, damit die anderen Teammitglieder bereits simulieren und ihre Teile verifizieren können, auch wenn das eigene Modul noch nicht in der synthetisierbaren Version fertig ist. Oder man könnte auch z.B. Altera und Xilinx spezifische Implementation realisieren, und diese je nach Ziel-Hardware auswählen.

Leider wird dies von Simulatoren wie ModelSim und Xsim zur Zeit nicht unterstützt!

3.15. RTL Design File Namen

Wie man im Beispiel im Kapitel 3.9 sehen konnte, wird bei entsprechender Typen-Deklaration und Generic Liste das `PACKAGE` und die `ENTITY` Definition recht lang. Um alles trotzdem übersichtlich zu halten, kann es eine gute Idee sein, PACKAGE, ENTITY und ARCHITECTURE in getrennten Files zu speichern. Der einzige zusätzliche Aufwand (ausser dass man dann alle 3 Files für die VHDL Kompilierung angeben muss) ist eine nochmalige Deklaration der verwendeten Bibliotheken am Anfang der `ARCHITECTURE`.

Wenn man ein Modul in mehrere Files aufteilt, ist es eine gute Idee die Namen ähnlich und trotzdem unverwechselbar zu halten. Dabei können einzelnen Buchstaben für verschiedene Teile stehen, wie z.B. mit `.p.vhd` für `PACKAGE`, `.e.vhd` für `ENTITY`, etc.

Object	VHDL code	File name

Package and component	`LIBRARY ...;`
	`USE ...;`
	
	`PACKAGE <block_name>_pgk IS`
	`  COMPONENT <block_name> IS`
	`  GENERIC ( ... );`
	`  PORT ( ... );`
	`  END COMPONENT <block_name>;`
	`END PACKAGE <block_name>_pkg;`
		`<block_name>.p.vhd`

Entity	`LIBRARY ...;`
	`USE ...;`
	
	`ENTITY <block_name> IS`
	`  GENERIC ( ... );`
	`  PORT ( ... );`
	`END ENTITY <block_name>;`
		`<block_name>.e.vhd`

Architecture	`ARCHITECTURE <arch> OF`
	`<block_name> IS`
	`BEGIN`
	`  ...`
	`END ARCHITECTURE <arch>;`
		`<block_name>.a_<arch>.vhd`

Mixed	`PACKAGE, ENTITY und ARCHITECTURE`	`<block_name>.m.vhd`
Configuration	`CONFIGURATION`	`<block_name>.cfg.vhd`