5. VHDL- Einführung II

5.1. Das VHDL- Prozesskonzept

Die bisher verwendeten VHDL- Signalzuweisungen waren nebenläufig, d.h. jede Anweisung wurde unabhängig voneinander sofort ausgeführt, wenn sich ein Signal auf der rechten Seite der Signalzuweisung änderte. Damit lassen sich nur einfache Hardwareblöcke mit einem einzigen Signalausgang realisieren.

Ein allgemeingültiges Modellierungskonzept erhält man durch die Einführung von Prozessen. In diesem Konzept finden sich auch nebenläufige Anweisungen wieder, denn diese lassen sich als implizite Prozesse auffassen.

Prozesse sind durch folgende Eigenschaften gekennzeichnet:

a) Alle Prozesse innerhalb einer Architektur sind nebenläufig; sie laufen aus der Sicht des Simulators also gleichzeitig ab (Simulationstechnik). Jeder Prozess kann daher als individueller Hardwarefunktionsblock aufgefasst werden (Synthesesemantik).

b) In Prozessen werden die Signalwerte festgelegt. Die Kommunikation zwischen Prozessen bzw. zu den Eingangs- und Ausgangsports erfolgen daher über lokale oder Port- Signale.

c) Ein Prozess wird dann aktiviert, wenn sich ein Signal seiner Sensitivity- Liste ändert.

d) Im Ausführungsteil von Prozessen (zwischen `begin` und `end process`) sind nur unbedingte Signalzuweisungen und spezielle sequentielle Anweisungen erlaubt. Diese Anweisungen werden strikt nacheinander ausgeführt. Bedinge und selektive nebenläufige Signalzuweisungen dürfen innerhalb von Prozessen nicht verwendet werden.

e) Die Zuweisung der tatsächlichen Signalwerte eines Prozesses erfolgt, etwas vereinfacht ausgedrückt, am Ende des Prozesses, also durch die `end process` Anweisung. Insbesondere ist es erlaubt, in einem Prozess einen bestimmten Signalwert zuzuweisen, der jedoch durch Überschreiben mit einem anderen Signalwert wieder verworfen wird. Der tatsächlich angenommene Wert ist der zuletzt im Prozess zugewiesene.

Die Syntax von VHDL- Prozessen ist:

```
[<process_label>]:process[ (<sensitivity list>) ]
{ <Variablendeklarationen> }
begin
{ <Sequentielle Anweisungen oder unbedingte Signalzuweisungen> }
```

end process [<process_label>];

Abbildung 5.1 zeigt exemplarisch die Interaktion zweier Prozesse in einer `architecture`, die in Listing 5.1 modelliert wird. Beide Prozesse repräsentieren dabei eine kombinatorische Logikhardware.

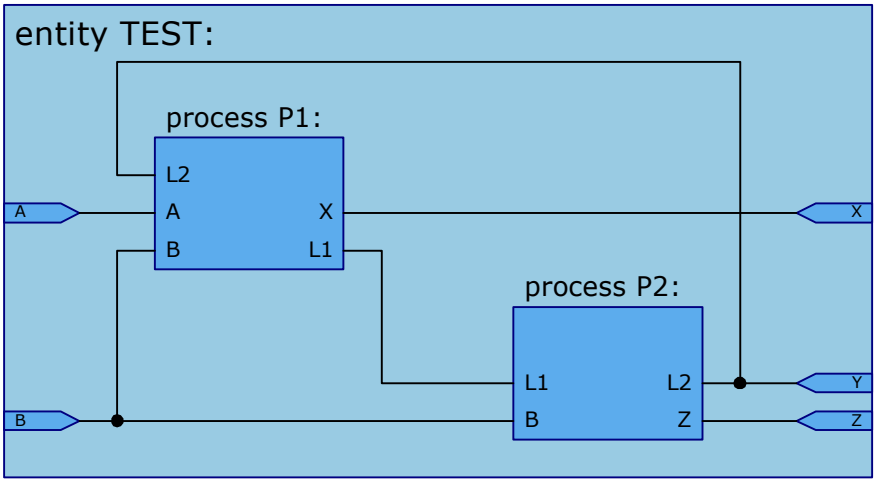

Abbildung 5.1: Hypothetische Architektur mit zwei kombinatorischen Prozessen

Listing 5.1: VHDL- Modell zur Archikektur aus Abbildung 5.1

```
entity TEST is
port( A, B : in bit;                    -- Eingangssignale
      X, Y, Z : out bit);               -- Ausgangssignale
end TEST;

architecture ARCH1 of TEST is
signal L1, L2 : bit;                    -- Deklaration lokaler Signale
begin
P1:   process (L2, A, B)                -- Deklaration P1 mit Sens. Liste
      begin
          …
          X <= …;                       -- Zuweisung an Ausgangssignal X
          L1 <= …;                      -- Zuweisung an lokales Signal L1
          …
      end process P1;
P2:   process (L1, B)                   --Deklaration P2 mit Sens. Liste
      begin
          …
          L2 <= …;                      -- Zuweisung an lokales Signal L1
          Z <= …;                       -- Zuweisung an Ausgangssignal Z
          …
      end process P2;
Y <= L2;                                -- Kopie an Ausgangssignal Y
end ARCH1;
```

In diesem Beispiel werden durch die beiden Prozesse je zwei Signale erzeugt, von denen jeweils nur eines als Ausgangssignal der `entity` dient. Das zweite im Prozess `P1` zugewiesene Signal `L1` wird als Eingangssignal des Prozesses `P2` verwendet, während das zweite Signal `L2` des Prozesses `P2` dem Prozess `P1` als Eingangssignal dient. Da dieses aber auch als Ausgangssignal `Y` verwendet wird, ist die in der `architecture` am Ende stehende nebenläufige Signalzuweisung erforderlich.

Da beide Prozesse der Schaltung in Abbildung 5.1 kombinatorische Logik repräsentieren sollen, muss sichergestellt werden, dass jede Änderung der Prozesseingangssignale auch zu einer potenziellen Änderung der Ausgangssignale führt, also den jeweiligen Prozess aktiviert. Dies ist die Aufgabe der Sensitivity- Liste des Prozesses. Wir können daher festhalten:

Bei VHDL- Prozessen, die kombinatorische Logik beschreiben, müssen sich alle Signale, die auf der rechten Seite einer Seite einer Signalzuweisung stehen oder die sich in Entscheidungsausdrücken sequentieller Anweisungen befinden, in der Sensitivity- Liste des Prozesses befinden.

5.2. Ereignisgesteuerte Simulatoren

In VHDL- Simulatoren, die auf PCs bzw. Workstations ablaufen, muss die Nebenläufigkeit aller Prozesse korrekt wiedergegeben werden, obwohl der Programmcode des Simulators sequenziell abläuft. Grundlage dafür sind die folgenden, in ereignisgesteuerten Simulatoren Mechanismen Bild.

- Ereignis- Warteschleife: Für jedes Signal wird eine Warteschlage geführt, zu welchem Zeitpunkt das Signal welchen Wert annehmen soll.
- Delta- Delay: Jeder Prozess befindet sich in einem von zwei Zuständen: „Warten" bedeutet, dass der Zustand auf eine Signalwechselanforderung (engl. Update Request) eines der in der Sensitivity- Liste befindlichen Signale wartet. Nach einem Update Request geht der Prozess in den Zustand „Ausführen". In diesem Zustand befindet sich der Prozess für einen infinitesimalen kurzen Zeitschritt (Delta- Zyklus). Am Ende dieses Zustands wird in den Signalwarteschleifen aller in diesem Prozess zugewiesenen Signale für den aktuellen Zeitpunkt, bzw. nach der in der Signalzuweisung genannten Verzögerung, eine Anforderung für den Signalwechsel eingetragen.
- Request- Update Prinzip: Auf der physikalischen Simulationszeitskala werden zunächst die zu diesem Zeitpunkt gültigen, in der jeweiligen Warteschlange eingetragenen Signalwechselanforderungen ausgewertet. Jede Prozessausführung erfordert einen Delta- Zyklus. Durch die Prozessausführung kann es zu neuen Signalwechselanforderungen für den gleichen physikalischen Zeitpunkt kommen

(Request), die in die zugehörige Warteschlagen eingetragen werden und einen weiteren Delta- Zyklus erfordern. Erst wenn alle Warteschlagen- Einträge eines physikalischen Zeitpunkts abgearbeitet sind, nehmen die zu ändernden Signale den zuletzt zugewiesenen Wert an (Update). Danach schreitet die physikalische Zeit im Simulator solange voran, bis sich in einer der Warteschlagen eine neue Signalwechselanforderung findet.

Zu Beginn einer Simulation werden zunächst alle Prozesse bzw. alle nebenläufigen Signalzuweisungen in einer vom Anwender nicht zu bestimmenden Reihenfolge einmal durchlaufen. Dabei werden alle Signale auf ihren Anfangswert initialisiert, bzw. es werden die ersten Einträge in die Signalwarteschlangen gemacht.

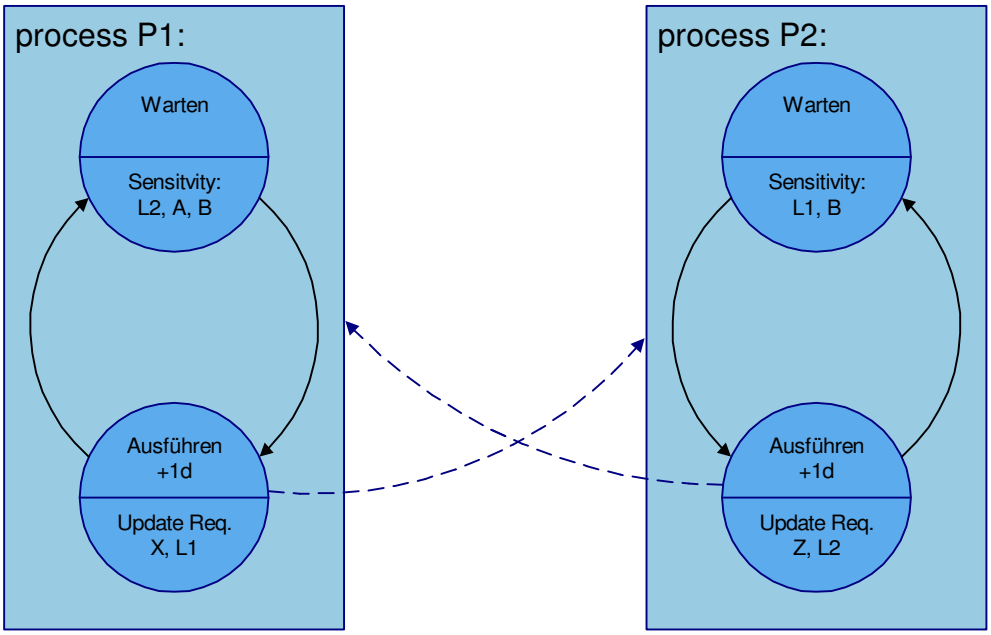

Abbildung 5.2: Abarbeitung von Prozessen in einem ereignisgesteuerten Simulator

Abbildung 5.2 zeigt das Zusammenspiel der in Listing 5.1 formulierten Prozesse `P1` und `P2`. Ausgangspunkt der Erläuterungen sei die Aktivierung des Prozesses `P1` durch einen Signalwechsel eines der Signale `L2` oder `A`.

- Im ersten Delta- Zyklus geht der Prozess `P1` in den Ausführungszustand über und wollen hier annehmen, dass beide Ausgangssignale `X` und `L1` geändert werden.
- Durch den Update-Request für `L1` wird nun der Prozess `P2` aktiviert und in einem zweiten Delta- Zyklus abgearbeitet. Im Moment wollen wir annehmen, dass der Prozess für die aktuellen Signalwerte von `L1` und `B` tatsächlich nur das Signal `Z` ändert.
- Nach Abschluss der beiden Delta- Zyklen schreitet die physikalische Zeit solange voran bis sich ein Signal in einer der beiden Sensitivity- Listen erneut ändert.

- Das so beschriebene Konzept garantiert, dass alle Signale zu einem physikalischen Zeitpunkt nach Ablauf aller Delta- Zyklen unabhängig von der Reihenfolge der Prozessausführung den gleichen Endwert annehmen. Die Aufgabe der Prozessausführungsplanung übernimmt ein im Simulator implementierter Scheduler.
- Abbildung 5.2 zeigt aber auch, dass es prinzipiell möglich ist, dass das System nie stabil wird. Dies wird dann der Fall sein, wenn sich beide Prozesse wechselseitig anregen. Diese Situation tritt auf, wenn im obigen Beispiel durch den Prozess `P2` auch das Signal `L2` geändert werden würde. Das System würde die Prozesse solange abwechselnd aufrufen, bis sich eines der beiden Signale `L1` und `L2` nicht mehr ändern würde.

Eine solche, als kombinatorische Schleife bezeichnete Situation tritt z.B. dann auf wenn der Ausgang eines XOR Gatters auf den Eingang zurückgekoppelt ist. Solange der zweite Eingang gleich 1 ist, werden dafür unendlich viele Delta- Zyklen benötigt und der Simulator scheint zu „hängen". In der Praxis ist die Schaltung instabil und schwingt mit einer Periode, die dem Doppelten der Signallaufzeit durch das Gatter entspricht. Diese einfache kombinatorische Schleife lässt sich jedoch im Quellcode schnell daran erkennen, dass das Ausgangssignal des Prozesses in der Sensitivity- Liste des gleichen Prozesses steht.

Listing 5.2: VHDL-Code für die kombinatorische Schleife

```
entity KOMB_SCHLEIFE is
port( A : in bit;                       -- Eingangssignal
      Y : out bit);                     -- Ausgangssignal
end KOMB_SCHLEIFE;

architecture ARCH1 of KOMB_SCHLEIFE is
signal TEMP : bit;                      -- Deklaration eines lokalen Signals
begin
P1:   process (A, TEMP)                 -- Deklaration von P1 mit Sens. Liste
      begin
          TEMP <= A xor TEMP;           -- Zuweisung an lokales Signal
      end process P1;
Y <= TEMP;                              -- Zuweisung an Ausgangssignal
end ARCH1;
```

Glücklicherweise erkennen VHDL- Simulatoren in der Regel derartige Situationen. So gibt z.B. der ModelSim – Simulator die folgende Fehlermeldung aus:

```
# ** Error: (vsim-3601) Iteration limit reached at time 50ns.
```

Zusammenfassend lässt sich festhalten:

Kombinatorische Schleifen sind unbedingt zu vermeiden. Insbesondere sollten Ausgangssignale eines kombinatorischen Prozesses nicht in der Sensitivity – Liste des gleichen Prozesses stehen.

5.3. Verzögerungsmodelle

Bei der Simulation digitaler Schaltungen sind verschiedene Reaktionen von Gatterausgängen auf Änderungen der Eingangssignale denkbar. Dies soll am Beispiel eines XOR Gatters dargestellt werden, dessen einer Eingang A dauerhaft auf 0 liegt und an dessen Eingangs B ein kurzer (Dauer 5ns) und ein längerer (Dauer 10ns) Eingangsimpuls gelegt werden.

5.3.1. Delta-Delay

Bei diesem Verzögerungsmodell erfolgt die Reaktion eines kombinatorischen Gattereingangs ohne Verzögerung sofort nach Änderung des Eingangssignals. Aus Sicht des Simulators wird jedoch ein infinitesimale Delta- Verzögerung benötigt. Für die VHDL-Signalzuweisung ist dies die Standardeinstellung.

```
Y0 <= A xor B;                    -- Delta Delay
```

5.3.2. Inertial-Delay

Bei diesem Modell wird von einer gewissen Trägheit der Gatter ausgegangen. Die Eingangskapazität, zusammen mit dem Leitungswiderstand lassen den Ausgang verzögert reagieren. Wenn ein Eingangsimpuls kürzer als der in der Signalzuweisung angegebenen Verzögerungszeit ist, so wird der Impuls solange absorbiert weil die Schaltschwelle des Gatters nicht erreicht wird. Das Ausgangssignal bleibt unverändert. Die nachfolgenden Codezeilen zeigen den Unterschied für unterschiedliche Verzögerungszeiten:

```
Y1 <= A xor B after 2ns;          -- Short Inertial Model
Y2 <= A xor B after 8ns;          -- Long Inertial Model
```

Da beide Impulse des Eingangssignals `B` länger als die für `Y1` spezifizierte Verzögerungszeit sind, erscheint das Eingangssignal um 2ns verzögert am `Y1` Ausgang. Hingegen ist die Impulsdauer des ersten Eingangsimpulses mit 5ns kürzer als die für `Y2` angegebene Verzögerungszeit von 8ns. Entsprechend erscheint dieser Impuls am `Y2` Ausgang nicht.

5.3.3. Transport-Delay

Bei diesem Verzögerungsmodell werden alle Eingangsimpulse unabhängig von ihrer Dauer mit der im Modell angegebenen Verzögerung von 4 ns am Ausgang abgebildet. In VHDL muss bei der Signalzuweisung das Schlüsselwort `transport` angegeben werden.

```
Y3 <= transport (A xor B) after 4ns;  -- Transport Model
```

5.3.4. Rejecting Inertial-Delay

Der Nachteil des Inertial-Delay-Modells ist die Tatsache, dass Signalverzögerungen und Mindestimpulsbreiten durch einen einzigen Zeitparameter modelliert werden, was für einige praktische Fälle eine zu starke Vereinfachung bedeutet. Aus diesem Grunde lässt sich ein erweitertes Modell spezifizieren, in dem die Mindestimpulsbreite hinter dem `reject` Schlüsselwort und die Verzögerungszeit wie gewohnt zu spezifizieren sind. Dieses Modell erfordert zusätzlich das Schlüsselwort `inertial`.

```
Y4 <= reject 6ns inertial (A xor B) after 7ns;
```

Da Impulse welche kürzer als die `reject` Zeit z.B. 6ns sind verschluckt werden tritt der kurze Impuls mit 5ns nicht auf. Die Verzögerung des langen Eingangsimpulses kann mit 7 ns davon unabhängig modelliert werden.

Hausübung:

Erstellen Sie ein `entity` mit den Eingang `A` und den Ausgängen `Y0` bis `Y4` und Simulieren sie das Verhalten bei Impulslängen von 5 ns und 10 ns. Zeichnen Sie in den Simulationsergebnissen die Verzögerungswerte ein und überprüfen Sie die Unterdrückung kurzer Impulse. Fügen Sie den VHDL Code und die Simulationsergebnisse ihren Unterlagen bei.

5.4. Sequentielle Anweisungen in Prozessen

Mit den in Abschnitt 1 eingeführten selektiven und bedingten nebenläufigen Signalzuweisungen lassen sich Verhaltenbeschreibungen nur für einfache Digitalschaltungen realisieren. Komplexes Verhalten erfordert den Einsatz von Verzweigungs- und Schleifenanweisungen, so wie sie aus prozeduralen Programmiersprachen bekannt sind. Derartige Konstrukte in VHDL als sequentielle Anweisungen bezeichnet, dürfen nur in Prozessen verwendet werden.

Innerhalb von Prozessen sind nur unbedingte Signalzuweisungen sowie sequentielle Anweisungen erlaubt. Dazu zählen: Die `if`- Anweisung, die `case`- Anweisung, die `for`- und `while`- Schleifenanweisungen sowie die `wait` Anweisung. Wertzuweisungen an dasselbe Signal können in Prozessen an verschiedenen Stellen erfolgen, es wird der Signalwert angenommen, der zuletzt zugewiesen wurde.

5.4.1. `case` - Anweisung

Die VHDL `case`- Anweisung hat eine semantische Ähnlichkeit zu der switch - case Anweisung der Programmiersprache C. Sie ist besonders geeignet, in Prozessen Wahrheitstabellen zu realisieren. Aus Hardwaresicht verbirgt sich hinter der `case`- Anweisung eine Multiplexer- bzw. Demultiplexer - Struktur.

Abhängig von einem einzigen Bedingungsausdruck muss bei der `case` – Anweisung für alle möglichen Werte, die der Bedingungsausdruck annehmen kann, hinter dem Schlüsselwort `when` angegeben werden, welchen Wert ein Signal annehmen soll.

Beispiel 5.1: Modellierung eines 4-zu-1 Multiplexers mit einer case Anweisung

Der 4-zu-1 Multiplexer besitzt einen 4 Bit breiten Eingangssignalvektor E. Abhängig von dem aktuellen Wert eines 2 Bit breiten Selektionssignals S wird entschieden, welchen Wert das Ausgangssignal annimmt.

Listing 5.3: VHDL-Modell eines 4-zu-1 Multiplexers mit case- Anweisung

```
entity MUX4X1_2 is
     port( E : in bit_vector( 3 downto 0);
           S : in bit_vector( 1 downto 0);
           Y : out bit);
end MUX4X1_2;

architecture VERHALTEN of MUX4X1_2 is
begin
MUXPROC: process( S, E)
  begin
     case S is
        when “00” => Y <= E(0);
        when “01” => Y <= E(1);
        when “10” => Y <= E(2);
        when others => Y <= E(3);
     end case;
  end process MUXPROC;
end VERHALTEN;
```

In diesem Beispiel stellt der aktuelle Wert des Signals S den Bedingungsausdruck dar, sodass es für dieses 2-Bit Signal prinzipiell vier Möglichkeiten gibt. Bei der Verwendung der `case`- Anweisung ist zwingend darauf zu achten, dass das Verhalten für alle möglichen Kombinationsmöglichkeiten des Selektionsausdrucks definiert ist (vollständige Spezifikation). Eine vollständige Spezifikation erreicht man immer mit dem in Beispiel 5.1 *verwendeten `others` Konstrukt, welches als letztes in der Auswahlliste stehen muss. Das Syntheseergebnis in Bild zeigt den gewünschten Multiplexer.*

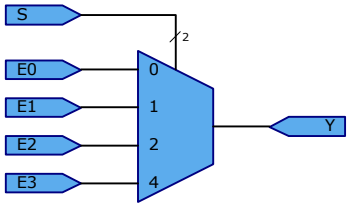

Abbildung 5.3: Syntheseergebnis der `case`- Anweisung in Listing 5.3

In dem Beispiel 5.1 wurde in jeder `when`- Verzweigung genau eine unbedingte Signalzuweisung angegeben. Prinzipiell ist es jedoch möglich, dort mehrere sequentielle Anweisungen insbesondere auch weitere `case`- oder `if`- Anweisungen zu verwenden.

5.4.2. `if` - Anweisung

Erst der sequentielle Charakter der innerhalb von Prozessen erlaubten Anweisungen ergibt für die Verwendung einer `if`- Verzweigungsanweisung, bei der in unterschiedlichen Verzweigungspfaden unterschiedliche Anweisungen stehen, einen Sinn. Durch die Verwendung von `elsif`- Zweigen kann innerhalb einer Anweisung eine Prioritätsreihenfolge mit völlig unterschiedlichen Bedingungen modelliert werden. In der Digitaltechnik spricht man von einem Prioritäts- Encoder.

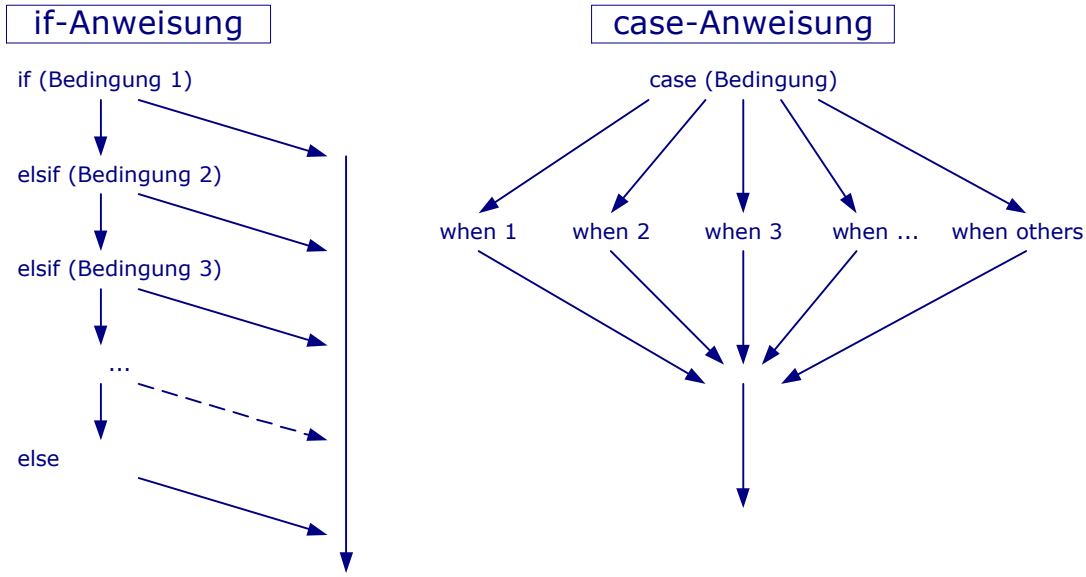

Abbildung 5.4: Vergleich von `if`- und `case`- Anweisung. Durch die Reihenfolge der Bedingungsausdrücke bei der `if`- Anweisung wird eine Priorität vorgegeben.

Beispiel 5.2: Modellierung eines Prioritäts-Encoders mit einer `if`- Anweisung

Gesteuert von zwei Selektionsbits `S(1)` und `S(0)` sollen die Eingangssignale `A` und `B` durch unterschiedliche Boole'sche Funktionen verknüpft werden. Für `S(0)` = 1 soll eine `UND`- und für `S(1)` = 1 eine `ODER` Funktion implementiert werden. Falls beide Selektionssignale nicht aktiv sind, aber das Eingangssignal `X`, soll eine `XOR`- Funktion realisiert werden. Falls keine dieser Bedingungen erfüllt ist, soll das Ausgangssignal `Y` = 0 sein.

Listing 5.4: VHDL-Modell eines Prioritäts-Encoders mit `if`- Anweisung

```
entity PRIORITAETS_ENCODER is
     port( A, B : in bit;
           X : in bit;
```

```
          S : in bit_vector( 1 downto 0);
          Y : out bit);
end PRIORITAETS_ENCODER;

architecture VERHALTEN of PRIRITAETS_ENCODER is
begin
P1: process( S, A, B, X)
    begin
      if S(0) = '1' then
        Y <= A and B;
      elsif S(1) = '1' then
        Y <= A or B;
      elsif ( S = "00" and X = '1') then
        Y <= X xor B;
      else
        Y <= '0';
      end if;
    end process P1;
end VERHALTEN;
```

Zur manuellen Synthese der Schaltung wird diese als Schaltnetz mit Steuersignalen implementiert.

X	S(1)	S(0)	Y
0	0	0	0
0	0	1	$A \wedge B$
0	1	0	$A \vee B$
0	1	1	$A \wedge B$
1	0	0	$A \nleftrightarrow B$
1	0	1	$A \wedge B$
1	1	0	$A \vee B$
1	1	1	$A \wedge B$

	S(0)		
	$A \nleftrightarrow B$	$A \wedge B$	$A \wedge B$
$A \vee B$	$A \vee B$	$A \wedge B$	$A \wedge B$

S(1) — X

Abbildung 5.5: Wahrheitstabelle und KV-Diagramm zum Prioritäts- Encoder aus Listing 5.4

Die Wahrheitstabelle in Abbildung 5.5 *besitzt auf der linken Seite die Steuersignale, die in der* `if`*- Anweisung abgefragt werden und zeigt auf der Rechten Seite die im* Listing 5.4 *spezifizierten Funktionen der Eingangssignale* `A` *und* `B`*. Dabei ist zu beachten, dass die UND- Verknüpfung, die zu der* `S(0)` *Bedingung in der ersten* `if`*- Anweisung gehören, viermal einzutragen ist, denn in dieser Bedingung stellen die Signale* `S(1)` *und* `X` *Don't Care Werte dar. Die ODER- Verknüpfung muss hingegen doppelt auftauchen, da sie in der zugehörigen Bedingung nur das* `X` *Signal als Don't Care Wert hat. Entsprechend tritt die XOR- Verknüpfung nur einmal auf. Dem KV- Diagramm in* Abbildung 5.5 *ist zu entnehmen, dass das Ausgangssignal damit durch drei Primimplikanten definiert ist. Damit ergibt sich die folgende dreistufige Logikfunktion, die aus drei Termen besteht und in Bild auch als Schaltplan dargestellt ist.*

$$Y = (S(0) \wedge (A \wedge B)) \vee (\overline{S(0)} \wedge S(1) \wedge (A \vee B)) \vee (\overline{S(0)} \wedge \overline{S(1)} \wedge X \wedge (A \leftrightarrow B))$$

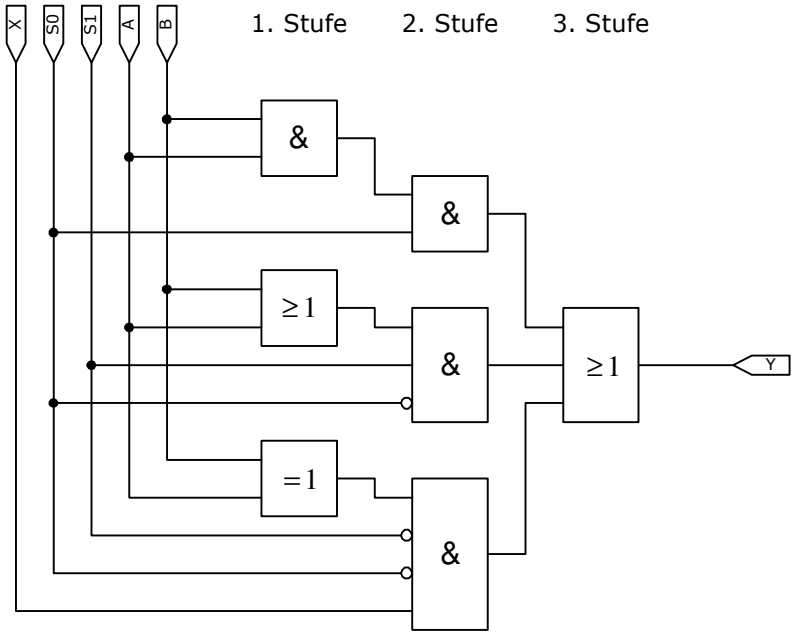

Abbildung 5.6: Syntheseergebnis des Prioritäts- Encoders aus Listing 5.4

In diesem Beispiel einer kombinatorischen `if`*- Anweisung ist auf folgende Punkte besonders hinzuweisen:*

- *Im Unterschied zu der* `case`*- Anweisung können in den einzelnen* `elsif`*- Abfragen völlig unterschiedliche Bedingungsausdrücke geprüft werden.*
- *Die Synthese der* `if`*- /* `elsif`*- Anweisung führt dazu, dass die Bedingung, die im VHDL Code weiter unten abgefragt werden durch immer breiter werdende UND-Gatter realisiert sind. Wenn man davon ausgeht, dass mit steigender Anzahl von Gattereingängen auch die Laufzeit des Gatters anwächst, so bedeutet dies, das die Zeit, die die Hardware zur Decodierung der letzten Bedingung benötigt, am größten ist.*

Die Synthese der `if`- Anweisung erlaubt auch eine unvollständige Spezifikation d.h. im Code werden z.B. durch Weglassen des `else`- Zweiges nicht alle Verbindungsmöglichkeiten explizit aufgeführt. In der Simulationstechnik bedeutet dies, dass die Ausgangssignale ihren alten Wert beibehalten, also ein Speicherverhalten. Die Synthesesemantik sieht in diesem Fall die Verwendung eines D - Latches vor.

Bei der Modellierung kombinatorischer Logik ist ein Speicherverhalten nicht sinnvoll. Leider wird bei unübersichtlichen ineinander geschachtelten `if`- Anweisungen häufig der Fehler gemacht, dass einzelne Verzweigungsmöglichkeiten nicht berücksichtigt wurden und ein unerwünschtes Speicherverhalten auftritt. Dies kann nur dadurch vermieden werden, dass den Ausgangssignalen vor der ersten `if`- Anweisung ein Defaultwert zugewiesen wird, der in den verschiedenen Verzweigungen gegebenenfalls überschrieben

wird. So könnte man z.B. in Beispiel 5.2 auch auf den `else`- Zweig verzichten, wenn nach der `process begin`- und vor der `if`- Anweisung die folgende Zeile eingefügt wäre.

```
Y <= 0;                                -- Defaultzuweisung
```

Wenn durch `if`- Anweisungen in Prozessen kombinatorische Logik modelliert werden soll, so wird für alle Ausgangssignale des Prozesses die Zuweisung eines Defaultwerts vor der ersten `if`- Anweisung empfohlen, um ungewolltes Speicherverhalten zu vermeiden.

5.4.3. Modellierung von Signalflanken mit der `if`- Anweisung

In getakteten Logikkomponenten reagieren deren Ausgänge nach dem Wechsel des Taktsignalpegels (Taktflanke) und zwar nur in einer Richtung, also entweder nach ansteigender oder nach abfallender Flanke. In VHDL lassen sich Pegelwechsel durch das für alle Signale definierte `event`- Attribute modellieren. Wenn sich nun das Taktsignal in der Sensitivity- Liste des Prozesses befindet, so muss durch die `if`- Bedingung festgelegt werden, ob der getaktete Prozess auf eine ansteigende Flanke oder eine abfallende Flanke reagieren muss. Listing 5.5 zeigt das VHDL Modell eines getakteten Prozesses, der ein D- Flipflop modelliert.

Listing 5.5: Signalflankenabfrage mit der if-Anweisung

```
entity DFF is
     port( CLK, D : in bit;
           Q : out bit);
end DFF;

architecture VERHALTEN of DFF is
begin
P1: process( CLK )
     begin
     if CLK = '1' and CLK'event then  -- ansteigende Signalflanke
          Q <= D;
          end if;
     end process P1;
end VERHALTEN;
```

Auf die folgenden Aspekte im Listing 5.5 soll besonders hingewiesen werden:

- Das Taktsignal muss sich in der Sensitvity- Liste befinden.
- Eine ansteigende Flanke wird durch `CLK = '1' and CLK'event` und eine abfallende Flanke durch `CLK = '0' and CLK'event` als Bedingung der `if`- Anweisung modelliert[1].

[1] *Die Reihenfolge der beiden Operatoren des and- Operators in der `if`- Bedingung ist beliebig.*

- Da getaktete Logik insbesondere zur Speicherung von Logikzuständen dient, ist es nicht erforderlich, die `if`- Anweisung vollständig zu spezifizieren, der `else`- Zweig ist also nicht erforderlich.

5.5. Prozesse ohne Sensitivity- Liste

Prozesse, die keine Sensitivity- Liste besitzen sind in der Simulation erlaubt, aber nicht synthesefähig. Sie bieten sich insbesondere für VHDL- Testbenches an.

Prozesse ohne Sensitivity- Liste werden wie alle Prozesse bei Simulationsbeginn automatisch gestartet und bei jeder `wait`- Anweisung an dieser Stelle unterbrochen. Nach Beendigung werden Sie automatisch gestartet.

Beispiel 5.3: Modellierung eines periodischen 10-MHz Taktgeneratorsignals für eine VHDL-Testbench durch einen Prozess ohne Sensitivity- Liste.

```
CLKGEN: process
begin
     CLK <= '1'; wait for 50 ns;
     CLK <= '0'; wait for 50 ns;
end process CLKGEN;
```

Da derartige Prozesse nach Beendigung sofort neu gestartet werden, ist die Existenz einer `wait`- Anweisung zwingend notwendig, da sich der Simulator ohne diese Anweisung in dem Prozess „aufhängen" würde.

5.6. Verwendung von Variablen in Prozessen

Innerhalb von Prozessen ist die Verwendung von VHDL Variablen erlaubt. Sinnvoll kann dies z.B. in kombinatorischer Logik sein, wenn z.B. arithmetische oder logische Ausdrücke in `if`- oder `case`- Anweisungen ausgewertet werden müssen.

Im Gegensatz zu Signalen kann der gerade zugewiesene Wert einer VHDL Variablen sofort abgefragt werden. Der Wertzuweisungsoperator für Variablen ist das Symbol ':='. Die Gültigkeit einer Variablen beschränkt sich auf den Prozess, in dem sie deklariert ist.

Variablen werden zwischen den Schlüsselworten `process` und `begin` deklariert und besitzen potentielles Speicherverhalten. Das bedeutet, dass sie den zuletzt zugewiesenen Wert auch nach Beendigung bis zum nächsten Prozessaufruf beibehalten.

Bei der Verwendung von Variablen in kombinatorischen Prozessen, die also kein Speicherverhalten besitzen, muss darauf geachtet werden, dass ihnen vor ihrer Verwendung immer zuerst ein aktueller Wert zugewiesen wurde. Andernfalls wird ein unerwünschtes D- Latch synthetisiert.

Beispiel 5.4: Verwendung einer Variablen in einem kombinatorischen Prozess

```
entity VAR_TEST is
     port( I1, I2 : in bit_vector(3 downto 0);
           Y : out bit);
end VAR_TEST;

architecture VERHALTEN of VAR_TEST is
begin
COMB: process( I1, I2)
variable TEMP : bit_vector(7 downto 0);   -- Deklaration der Variablen
     begin
     TEMP := I1 & I2;                  -- Variable zur verkettung zweier
                                          Bussignale
     if TEMP = "10101010" then         -- Sofortige Auswertung der Var.
          Y <= '1';
     else
          Y <= '0';
          end if;
     end process COMB;
end VERHALTEN;
```

Die Verwendung eines Signals anstelle einer Variablen in Beispiel 5.4 *würde zu einem unerwünschten Speicherverhalten führen, da das Signal den zugewiesenen Wert erst am Ende des Prozesses annimmt. Bei einer Signalwertabfrage in der* `if`*- Anweisung würde also fehlerhafter Weise auf den Signalwert der letzten Prozessaktivierung zurückgegriffen werden.*

5.7. Modellierungsbeispiel

In diesem Abschnitt soll eine aus mehreren Prozessen bzw. nebenläufigen Anweisung bestehende Schaltung modelliert werden. Dabei handelt es sich um eine konfigurierbare Ausgangszelle eines programmierbaren Logikbausteins (PLD). Diese Schaltung wird von den Implementierungswerkzeugen automatisch instanziiert, um eine als SOP- Ausdruck vorliegende Schaltfunktion nach außen zu führen. Das VHDL Modell in Listing muss bei der Hardwareimplementierung also nicht explizit angegeben werden, es dient hier vielmehr als Modellierungsbeispiel.

Das `entity` des `OLMC` Modells soll dabei das SOP Signal `Y_COMP`

- entweder kombinatorisch oder über ein Register,
- entweder invertiert oder nichtinvertiert,

an einen Ausgangspin des PLD legen. Die ausgewählte Funktion wird durch zwei Programmierbits S(0) und S(1) festgelegt. Die Umschaltung zwischen kombinatorischem und getaktetem Verhalten erfolgt durch einen 2-zu-1 Multiplexer und als programmierbarer Inverter dient ein XOR- Gatter (vgl. Abbildung 5.7). Das im Listing

vorgestellte VHDL Modell enthält neben der Hardwarefunktionalität auch Testbench Prozesse.

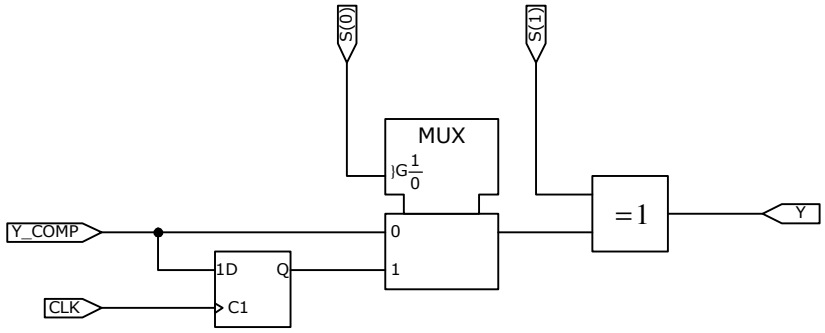

Abbildung 5.7:Aufbau einer Ausgangslogikmakrozelle eines PLDs

Tabelle 5.1: Programmierung der Ausgangszelle

S(1)	S(0)	Ausgangsfunktion
0	0	Nichtinvertiert, kombinatorisch: $Y = Y\_COMP$
0	1	Nichtinvertiert, Registerausgang: $Y = TEMP\_1$
1	0	Invertiert, kombinatorisch: $Y = \overline{Y\_COMP}$
1	1	Invertiert, Registerausgang: $Y = \overline{TEMP\_1}$

Listing 5.6: VHDL-Modell einer Ausgangslogik-Makrozelle mit Testbench Prozessen

```
entity OLMC_TB is
end OLMC_TB;

architecture VERHALTEN of OLMC_TB is
-- port Signale des DUT:
signal CLK, Y_COMP : bit;              -- in ports
signal A : bit_vector(1 downto 0);     -- in port
signal Y : bit;                        -- out port
-- end port Signale des DUT
signal TEMP_1, TEMP_2 : bit;           -- lokale Signale im DUT
signal TEST : integer range 1 to 8;    --  reines Testbench Signal

begin
-- Synthesefaehige Prozesse (Device under Test DUT);
MUX : process(Y_COMP, TEMP_1, S(0) ); -- aktivierende Signale für k. Proz.
     begin
        case S(0) is
          when '0' => TEMP_2 <= Y_COMP;
          when '1' => TEMP_2 <= TEMP_1;
        end case;
```

```
    end process MUX;

D_FF: process( CLK )                    -- nur Taktgesteuert
    begin
      if CLK'event and CLK = '1' then    -- ansteigende Flanke
        TEMP_1 <= Y_COMP;               -- Signaluebernahme
      else
        TEMP_1 <= TEMP_1;               -- gespeichertes Signal
        end if;
      end process D_FF;

Y <= TEMP_2 xor S(1);                   -- gest. Inverter nebenlaeufig
-- end DUT Prozesse

-- Testbench Prozesse:
CLKGEN: process                         -- Taktgenerator 5 MHz
    begin
      CLK <= '0'; wait for 100 ns;
      CLK <= '1'; wait for 100 ns;
      end process CLKGEN;

SIMULI: process                         -- Diskrete Stimuli f. 8 Tests
    begin
      TEST <= 1; Y_COMP <= '1'; S <= "00"; wait for 200ns;
      TEST <= 2; Y_COMP <= '1'; S <= "10"; wait for 200ns;
      TEST <= 3; Y_COMP <= '0'; S <= "00"; wait for 200ns;
      TEST <= 4; Y_COMP <= '0'; S <= "10"; wait for 200ns;
      TEST <= 5; Y_COMP <= '1'; S <= "01"; wait for 200ns;
      TEST <= 6; Y_COMP <= '1'; S <= "11"; wait for 200ns;
      TEST <= 7; Y_COMP <= '0'; S <= "01"; wait for 200ns;
      TEST <= 8; Y_COMP <= '0'; S <= "11"; wait for 200ns;
      end process STIMULI;

RESPONSE_MONTITOR: process              -- Prüfe Testergebnisse
    begin
    wait for 150 ns;                    -- 50 ns nach steigender Flanke
    for TEST in 1 to 8 loop             -- prüfe 8 mal (insgesamt 1600 ns)
      case TEST is
        when 1 => assert Y = '1' report "Error: test 1";
        when 2 => assert Y = '0' report "Error: test 2";
        when 3 => assert Y = '0' report "Error: test 3";
        when 4 => assert Y = '1' report "Error: test 4";
        when 5 => assert Y = '1' report "Error: test 5";
        when 6 => assert Y = '0' report "Error: test 6";
        when 7 => assert Y = '1' report "Error: test 7"; -- Fehler
        when 7 => assert Y = '1' report "Error: test 8";
        end case;
```

```
        wait for 200 ns;                -- naechster Test nach 200 ns
        end loop;
    end process RESPONSE_MONITOR;
-- end Testbench Prozesse
end VERHALTEN;
```

Der Code in Listing 5.6 ist wie folgt zu erläutern:

- Da es sich um eine Testbench handelt, besitzt die `entity` keine Schnittstellensignale. Falls das „Device under Test (DUT)" synthetisiert werden soll, so sind die entsprechend gekennzeichneten, lokal deklarierten Signale als `port`- Signale zu wählen und die Testbench zu entfernen.
- Der synthesefähige Teil des Codes, das DUT, besteht aus zwei Prozessen (`MUX` und `D_FF`) sowie einer nebenläufigen Signalzuweisung für das XOR Gatter. Die Kommunikation zwischen den Prozessen erfolgt durch die beiden lokalen Signale `TEMP_1` und `TEMP_2` (vgl. Abbildung 5.7).
- Drei Prozesse stellen hier die Testumgebung dar. Diese Prozesse arbeiten ohne Sensitivity- Liste. Mit `CLKGEN` wird ein 5-MHz Taktgenerator und mit `STIMULI` werden acht verschiedene Testmuster (Testvektoren, Testpattern) für die Eingänge des DUT definiert. Der Prozess `RESPONSE_MONITOR` dient zum Vergleich mit den für die einzelnen Tests erwarteten Ergebnissen. Die Information über die gerate ausgeführte Testnummer wird dem `RESPONSE_MONITOR` durch das Signal `TEST` übermittelt, welches als ganze Zahl (VHDL Datentyp integer) im Zahlenbereich zwischen 1 und 8 deklariert ist.
- Alle acht Tests sollen für eine Taktperiode, also für 200 ns durchgeführt werden. Entsprechend sind die `wait`- Anweisungen des `STIMULI` Prozesses gewählt. Die gesamte Simulationszeit beträgt also 1600 ns. Die Auswertung der Testergebnisse im `RESPONSE_MONITOR` soll jeweils nach der steigenden Testflanke erfolgen. Da der Taktgenerator mit dem Pegel 0 beginnt, kann der erste Test nach einer Zeit von 150 ns erfolgen. Die Testauswertung erfolgt durch eine `assert`- Anweisung in einer `for- loop-` Schleife innerhalb derer die Zeit in Intervallen von 200 ns voran schreitet.

Hausübung

Erstellen Sie die Testbench OLMC_TB und führen Sie die Simulation durch. Vergleichen Sie Ihre Ergebnisse mit folgenden Aussagen:

Ursache der Fehlermeldung ist der RESPONSE_MONITOR Prozess. Der STIMULI Prozess für Test Nr. 7 sieht vor, dass das SOP- Signal Y_COMP = 0 im Flipflop gespeichert wird und diesen Wert nichtinvertiert, also als 0, ausgegeben wird. Im RESPONSE_MONTOR

wird für diesen Test jedoch fehlerhafterweise eine 1 erwartet. Nach Korrektur dieses Fehlers läuft die Simulation fehlerfrei.

Zum Zeitpunkt t = 400 ns tritt ein Struktur- Hazard im Ausgangssignal Y auf. Ein weiters Zoomen in das Simulationsergebnis zweigt, dass dieser infinitesimal kurz ist, also für genau einen Delta-Zyklus anhält. Ursache dafür ist die Tatsache, dass sich zu diesem Zeitpunkt die beiden Signale Y-COMP und S(1) gleichzeitig ändern. Als Folge des Y_COMP Signalwechsel ändert sich zum gleichen physikalischen Zeitpunkt auch TEMP_2. Damit wird die nebenläufige XOR Signalzuweisung in unterschiedlichen Delta-Zyklen zweimal nacheinander aktiviert. Zunächst mit einem Ausgangssignal Y = 1, aber einen Delta-Zyklus später mit Y = 0. Derartige „symbolische Hazards" geben einen Hinweis darauf, dass für die in Hardware realisierte Schaltung an gleicher Stelle ein Hazard als folge eines kritischen Wettrennens zu erwarten ist.

Dokumentieren Sie die Ergebnisse und für Sie die Dokumente Ihren Unterlagen zu.