Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

HARDWARE-BESCHREIBUNGSSPRACHEN

Hardwareentwurf mit VHDL

21. Oktober 2021
Revision: b941727 (2021-01-16 01:57:51 +0100)

Steffen Reith

Theoretische Informatik
Studienbereich Angewandte Informatik
Hochschule **RheinMain**

PIPELINING

Notizen

Notizen

GRUNDLAGEN

Pipelining ist eine grundlegende Technik, um die **Performanz** eines Systemes zu **verbessern**.

- → Idee: Tasks (wenn möglich) zeitlich **überlappend auszuführen**
- → Idee: kombinatorische Schaltkreise in **Teilschritte aufteilen**
- → Idee: verwende **Register** um **Zwischenergebnisse** für den nächsten Schritt zu **speichern**

Hier gibt es zwei wichtige Kenngrößen:

- → Delay: **Zeit** die die **Bearbeitung eines Tasks** benötigt
- → Durchsatz: **Anzahl** der bearbeiteten **Tasks pro Zeiteinheit**

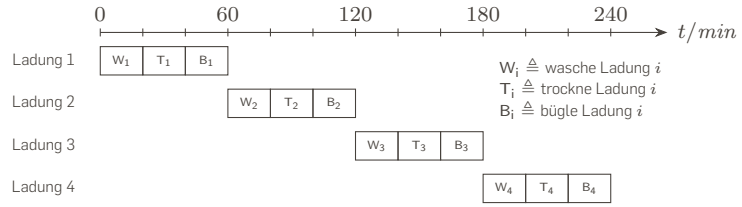
Pipelining **verbessert** den **Delay nicht** (ehr schlechter) und vergrößert (evtl.) den **Durchsatz**.

Notizen

EIN BEISPIEL: WASCHEN OHNE PIPELINING

In einer Wäscherei sind die Tasks „waschen“, „trocknen“ und „bügeln“ durchzuführen:

Ohne Pipeline



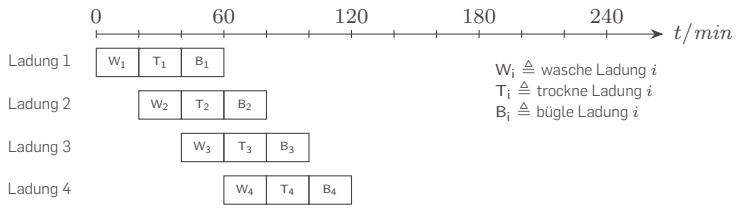
Damit ergibt sich:

- → **Delay**: 60 **Minuten** (Zeit für die Bearbeitung einer Ladung Wäsche)
- → **Durchsatz**: $4/(4 \cdot 3 \cdot 20) = 1/60$ **Ladungen pro Minute**

Notizen

EIN BEISPIEL: WASCHEN MIT PIPELINING

Mit Pipeline



Damit ergibt sich (idealisiert):

- → **Delay**: unverändert 60 **Minuten** (Zeit für die Bearbeitung einer Ladung Wäsche)
- → **Durchsatz**: Für k Ladungen wird die Zeit $40 + 20k$ benötigt. In diesem Beispiel ergibt sich $4/(40 + 20 \cdot 4) = 1/30$ **Ladungen pro Minute**

Notizen

EIN BEISPIEL: WASCHEN MIT PIPELINING (II)

Werden sehr **viele Ladungen Wäsche gewaschen**, so ergibt sich sogar

$$\lim_{k\to\infty} \frac{k}{40 + 20 \cdot k} = \frac{1}{20}$$

der dreifache Durchsatz.

Die dargestellte Situation ist stark idealisiert, denn:

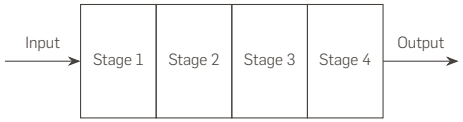
- → Die drei Teilaufgaben „waschen“, „bügeln“ und „trocknen“ haben den **identischen Zeitbedarf**.
- → Der **zusätzliche Zeitbedarf** (z.B. Ablage in einem Zwischenspeicher) für die **Überlappung** der Aufgaben wurde **vernachlässigt**.

Notizen

PIPELINING MIT SCHALTKREISEN

Der gleiche Ansatz kann auf Schaltkreise angewendet werden[3].

Ziel: Teile den kombinatorischen Schaltkreis in möglichst **identisch lang** arbeitende Teile auf (**Stages**).



Seien T_1, T_2, T_3 und T_4 die Delays der Stages 1-4, dann ist $T_{\max}$ der **Delay der „langsamsten“ Stage**:

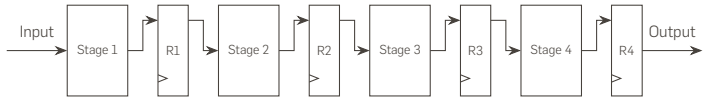
$$T_{\max} = \max\{T_1, T_2, T_3, T_4\}.$$

[3] Das Prinzip des Pipelinings in CPUs wird in D. Patterson und J. Hennessy, Computer Organization and Design, Morgan Kaufmann, 2012 in den Kapiteln 4.5 und 4.6 beschrieben.

Notizen

PIPELINING MIT SCHALTKREISEN (II)

Zur Regulierung des Signalflusses werden die (synchronen) Register R_1, R_2 und R_3 eingeführt. Register R_4 dient als **Ausgabebuffer**:



Sei T_{cq} die **Verzögerung eines Registers** R_i mit der das Clocksignal am Q-Ausgang ankommt, dann beträgt die minimale Periodendauer T_c

$$T_c = T_{\max} + T_{\text{set}} + T_{\text{cq}}$$

T_c gibt dabei also die Dauer eines **Teilschritts** an.

Notizen

PIPELINING MIT SCHALTKREISEN (III)

Der ursprüngliche kombinatorische Schaltkreis braucht für die **gesamte** Aufgabe (Delay):

$$T_{\text{comb}} = T_1 + T_2 + T_3 + T_4$$

Für die Version mit Pipeline ergibt sich der schlechtere Delay

$$T_{\text{pipe}} = 4T_c = 4T_{\max} + 4(T_{\text{set}} + T_{\text{cq}})$$

Hier zeigt sich, dass die **einzelnen Stages** möglichst den **gleichen Delay** haben sollten. Dies muss man evtl. durch mehrere Implementierungsversuche „ausprobieren“ und dann die Stages anpassen (Retiming).

Notizen

PIPELINING MIT SCHALTKREISEN (IV)

Der **Durchsatz des kombinatorischen Schaltkreises** beträgt

$$\frac{1}{T_{\text{comb}}}$$

Um k Tasks zu bearbeiten, benötigt die **Variante mit Pipeline**

$$3T_c + kT_c$$

Zeit und liefert den Durchsatz (für **große** k):

$$\lim_{k\to\infty} \frac{k}{3T_c + kT_c} = \frac{1}{T_c}$$

Unter den Annahmen, dass $T_{\text{set}} + T_{\text{cq}}$ vernachlässigbar **klein** und $T_{\max} = T_{\text{comb}}/4$ (perfekt balancierte Stages) gilt:

$$T_{\text{pipe}} = 4T_c \approx 4T_{\max} = T_{\text{comb}}$$

Notizen

PIPELINING MIT SCHALTKREISEN (V)

Damit ergibt sich

$$\frac{1}{T_c} \approx \frac{1}{T_{\max}} = \frac{4}{T_{\text{comb}}},$$

d.h. der **vierfache Durchsatz**. Diese Betrachtung kann auch auf eine Pipeline mit n **Stufen übertragen** werden.

Achtung: Verwendet man sehr viele Pipelinestufen, so wird T_c klein, aber $T_{\text{set}} + T_{\text{cq}}$ ist nicht mehr vernachlässigbar!

Für den **effektiven Einsatz** von Pipelining sollte ein Schaltkreis

- → kontinuierlich große Mengen an Daten verarbeiten müssen,
- → den Durchsatz als wichtigstes Designkriterium haben,
- → in möglichst gleich „große“ Stages aufteilbar sein und
- → der Delay einer Stage ist groß gegenüber $T_{\text{set}} + T_{\text{cq}}$.

Notizen

GENERELLES VORGEHEN

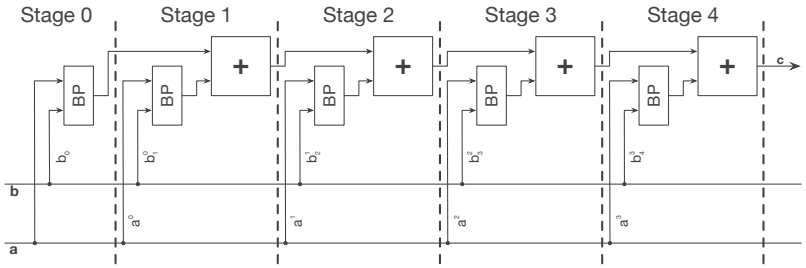
Liegt ein Schaltkreis als VHDL-Beschreibung vor, so kann mit Hilfe der folgenden Schritte eine gepipelinte Version gewonnen werden:

- → **Bringe** die graphische Darstellung der **Schaltung** in eine **„Kettenform“**
- → **Identifiziere größere Grundbausteine** in der Kette und **schätze** deren **Verzögerung**
- → **Teile** die Kette in gleich große **Stücke / Stages**. Die **Verzögerungszeit** jeder Stage solle deutlich **größer als** $T_{\text{set}} + T_{cq}$ sein.
- → **Identifiziere** alle **Signale**, die **die Grenze** zwischen zwei Stages **überqueren**
- → **Füge Register** für diese Signale **an den Grenzen** der Stages ein.

Notizen

EIN EINFACHER 5-BIT MULTIPLIZIERER

Nun soll ein 5-Bit Multiplizierer vorgestellt werden, der dann mit einer Pipeline ausgestattet wird:



Es gilt $a = a_0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + \cdots + a_4 \cdot 2^4$ und $b = b_0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \cdots + b_4 \cdot 2^4$.

Notizen

EIN EINFACHER 5-BIT MULTIPLIZIERER (II)

Das **Produkt** $a \cdot b$ kann dann geschrieben werden als:

$$a \cdot b = \sum_{i=0}^{4} b_i 2^i \sum_{j=0}^{4} a_j 2^j = \sum_{i=0}^{4} b_i \sum_{j=0}^{4} a_j 2^{i+j} = \sum_{i=0}^{4} \sum_{j=0}^{4} a_j b_i 2^{i+j}$$

Die Summanden $b_i \sum_{j=0}^{4} a_j 2^{i+j} = b_i a 2^i$ heißen **Bitprodukt** (von b_i und a) und können durch eine and-Verknüpfung bestimmt werden.

Damit ergibt sich:

- → Genau dann, wenn $b_i = 1$ trägt das **Bitprodukt** von a und b_i den **Wert** $a2^i$ zum Produkt bei.
- → Die Zahl $a2^i$ **kann leicht** durch (mehrfaches) anhängen von 0 **erzeugt werden**.

Notizen

AUSSCHNITT DER VHDL-IMPLEMENTIERUNG

```
1  entity BMult is
2    port (a : in  std_logic_vector(4 downto 0);
3          b : in  std_logic_vector(4 downto 0);
4          c : out std_logic_vector(9 downto 0));
5  end entity;
6
7  architecture Behavioral of BMult is
8
9    -- Vector version of ith bit of b
10   signal b0Vect, b1Vect, b2Vect : std_logic_vector(4 downto 0);
11
12   -- The ith bit-product
13   signal bitProd0, bitProd1, bitProd2 : unsigned(9 downto 0);
14
15   -- The ith partial product
16   signal partProd0, partProd1, partProd2 : unsigned(9 downto 0);
17
18   -- Copies of inputs for each stage
19   signal a0, a1, a2 : std_logic_vector(4 downto 0);
20   signal b0, b1, b2 : std_logic_vector(4 downto 0);
```


Notizen

AUSSCHNITT DER VHDL-IMPLEMENTIERUNG (II)

```
1  begin
2
3    -- Stage 0
4    b0Vect <= (others => b(0));
5    bitProd0 <= unsigned("00000" & (b0Vect and a));
6    partProd0 <= bitProd0;
7    a0 <= a;
8    b0 <= b;
9
10   -- Stage 1
11   b1Vect <= (others => b0(1));
12   bitProd1 <= unsigned("0000" & (b1Vect and a0) & "0");
13   partProd1 <= bitProd1 + partProd0;
14   a1 <= a0;
15   b1 <= b0;
16
17   -- Stage 2
18   b2Vect <= (others => b1(2));
19   bitProd2 <= unsigned("000" & (b2Vect and a1) & "00");
20   partProd2 <= bitProd2 + partProd1;
```


Notizen

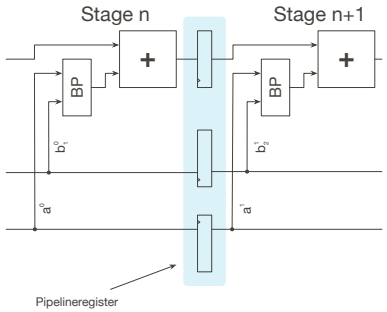
AUSSCHNITT DER VHDL-IMPLEMENTIERUNG (III)

```
1    a2 <= a1;
2    b2 <= b1;
3  ...
4
5    -- Output
6    c <= std_logic_vector(partProd4);
7
8  end;
```


Notizen

EIN MULTIPLIZIERER MIT PIPELINE

Nun werden an den **Grenzen** der einzelnen Stages **Pipelineregister** für alle Signale **eingefügt**, die die Grenze einer Stage überqueren:



Ausnahme: An der **Grenze zwischen Stage 0 und Stage 1** werden **keine Pipelineregister** eingefügt, da die ersten beiden Bitprodukte parallel ausgerechnet werden.

Notizen

AUSSCHNITT DER VHDL-IMPLEMENTIERUNG

```
1  entity PMult is
2    port (clk   : in std_logic;
3          reset : in std_logic;
4          a     : in  std_logic_vector(4 downto 0);
5          b     : in  std_logic_vector(4 downto 0);
6          c     : out std_logic_vector(9 downto 0));
7  end entity;
8
9  architecture Behavioral of PMult is
10
11 -- Registers for the copies of the input
12 signal a1_reg, a2_reg, a3_reg : std_logic_vector(4 downto 0);
13 signal b1_reg, b2_reg, b3_reg : std_logic_vector(4 downto 0);
14 signal partProd1_reg, partProd2_reg, partProd3_reg,
15        partProd4_reg : unsigned(9 downto 0);
16
17 -- Inputs for the pipeline registers
18 signal a1_next, a2_next, a3_next : std_logic_vector(4 downto 0);
19 signal b1_next, b2_next, b3_next : std_logic_vector(4 downto 0);
20 signal partProd1_next, partProd2_next, partProd3_next,
21        partProd4_next : unsigned(9 downto 0);
```


Notizen

AUSSCHNITT DER VHDL-IMPLEMENTIERUNG (II)

```
1  pipeline : process (clk)
2  begin
3
4    if (rising_edge(clk)) then
5
6      if (reset = '1') then -- Init all registers
7        a1_reg <= (others => '0');
8        a2_reg <= (others => '0');
9        a3_reg <= (others => '0');
10       b1_reg <= (others => '0');
11       b2_reg <= (others => '0');
12       b3_reg <= (others => '0');
13       partProd2_reg <= (others => '0');
14       partProd3_reg <= (others => '0');
15     else -- Handle the pipeline registers
16       a1_reg <= a1_next;
17       a2_reg <= a2_next;
18       a3_reg <= a3_next;
19       b1_reg <= b1_next;
20       b2_reg <= b2_next;
```


Notizen

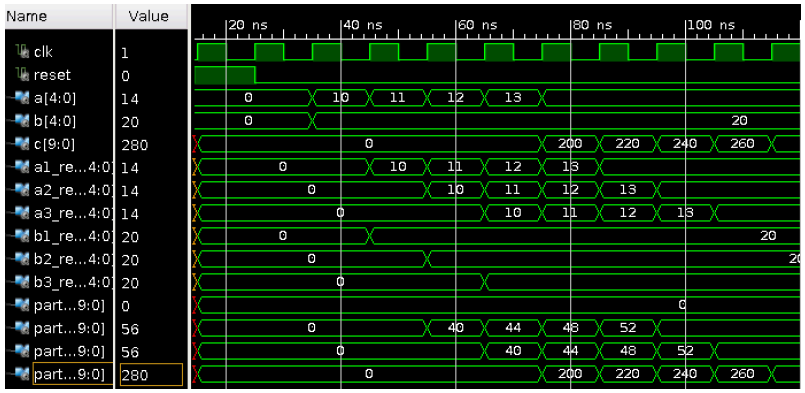
AUSSCHNITT DER VHDL-IMPLEMENTIERUNG (III)

```
      b3_reg <= b3_next;
      partProd2_reg <= partProd2_next;
      partProd3_reg <= partProd3_next;
    end if;
  end if;
end process;

-- Stage 2
b2Vect <= (others => b1_reg(2));
bitProd2 <= unsigned("000" & (b2Vect and a1_reg) & "00");
partProd2_next <= bitProd2 + partProd1_reg;
a2_next <= a1_reg;
b2_next <= b1_reg;

-- Stage 3
b3Vect <= (others => b2_reg(3));
bitProd3 <= unsigned("00" & (b3Vect and a2_reg) & "000");
partProd3_next <= bitProd3 + partProd2_reg;
a3_next <= a2_reg;
b3_next <= b2_reg;
```


Notizen

TIMINGDIAGRAMM

Mit Hilfe einer Testbench werden nacheinander die Werte 10, 11, 12, 13 und 14 für a und 20 für b in den Multiplizierer eingegeben.



Notizen

Vielen Dank!

Notizen

Notizen