

Digitaltechnik

Wintersemester 2017/2018

13. Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

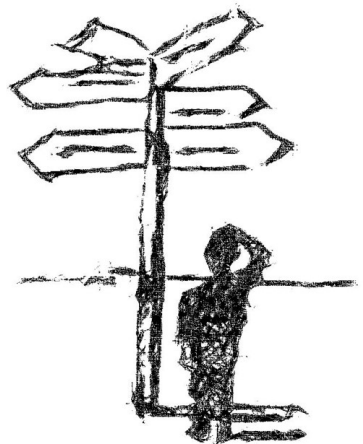




1. Einleitung
2. Weitere arithmetische Grundsaltungen
3. Ausblick Rechnerorganisation
4. Zusammenfassung

Agenda

1. Einleitung
2. Weitere arithmetische Grundsaltungen
3. Ausblick Rechnerorganisation
4. Zusammenfassung





- ▶ Evaluation der Veranstaltung durch die Fachschaft
 - ▶ in V14 (31.01.18)
 - ▶ keine nachträgliche Teilnahme möglich



- ▶ Evaluation der Veranstaltung durch die Fachschaft
 - ▶ in V14 (31.01.18)
 - ▶ keine nachträgliche Teilnahme möglich
- ▶ Klausurvorbereitung
 - ▶ Deck- und Hilfblatt im Moodle verfügbar
 - ▶ Wiederholungsthemen können weiter vorgeschlagen werden

Rückblick auf die letzte Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Mehr SystemVerilog für Testumgebungen
- ▶ SystemVerilog Abschluss
- ▶ **Arithmetische Grundsaltungen**



Harris 2013
Kap. 4.4-5.2

Wiederholung: Linearisierung von Arrays



TECHNISCHE
UNIVERSITÄT
DARMSTADT



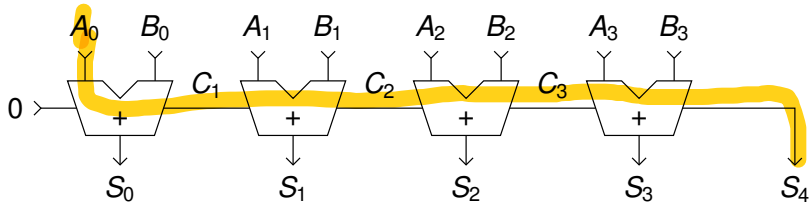
- ▶ Arrays von Vektoren als I/O Ports von Modulen (idR.) nicht unterstützt
- ⇒ Linearisierung zu langem Bitvektor notwendig („flattening“)

```
1 module flattening
2   (input logic [7:0] inarr [0:3], input logic [8*4-1:0] invec,
3    output logic [9:0] outarr [0:1], output logic [10*2-1:0] outvec);
4
5   // Eingabe aufspalten
6   logic [9:0] s [0:1];
7   assign s[0] = inarr[0] + inarr[1] + inarr[2] + inarr[3];
8   assign s[1] = invec[0+:8] + invec[8+:8] + invec[23+:8] + invec[31+:8];
9               // invec[7+:0] + invec[15+:8] + invec[23+:16] + invec[31+:24];
10
11  // Ausgabe zusammensetzen
12  outarr[1] = s[1];
13  outarr[0] = s[0]; // {outarr[1], outarr[0]} = {s[1], s[0]}
14
15  outvec[10+:10] = s[1]; // outvec[19:10] = s[1];
16  outvec[9 -:10] = s[0]; // outvec[9:0] = s[0];
17                       // outvec = {s[1], s[0]}
18 endmodule
```

Wiederholung: Ripple-Carry-Adder



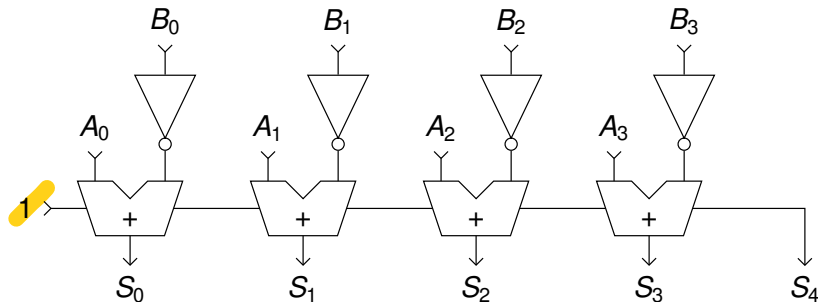
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Wiederholung: Subtraktion



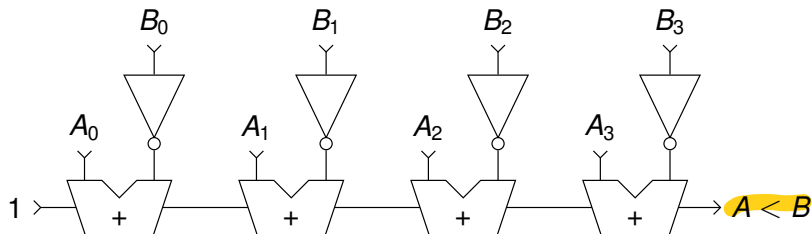
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Wiederholung: „Kleiner als“ Vergleich



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Überblick der heutigen Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Weitere arithmetische Grundsaltungen
 - ▶ Schnelle Additionen
 - ▶ kombinatorische und sequentielle Multiplikation
- ▶ Ausblick Rechnerorganisation
 - ▶ Von-Neumann- und Harvard-Architektur
 - ▶ Speicher
 - ▶ Arithmetisch-Logische Einheit
 - ▶ Steuerwerk



Harris 2013

Kap. 5.2

Seite 233 - 248

Kap. 7.3

Seite 368 - 381

Agenda



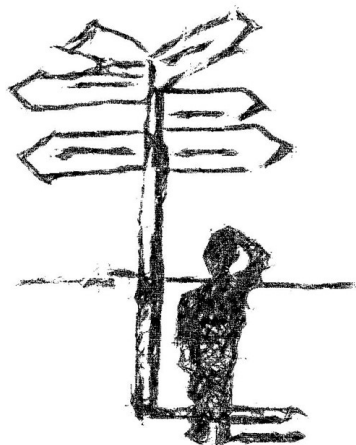
TECHNISCHE
UNIVERSITÄT
DARMSTADT

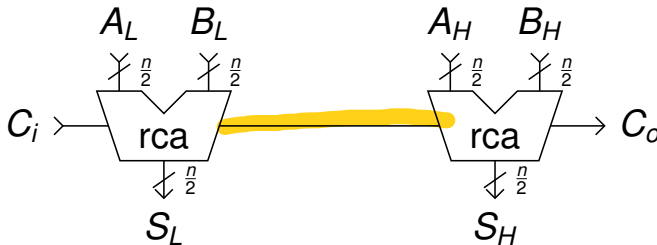
1. Einleitung

2. Weitere arithmetische Grundsaltungen

3. Ausblick Rechnerorganisation

4. Zusammenfassung





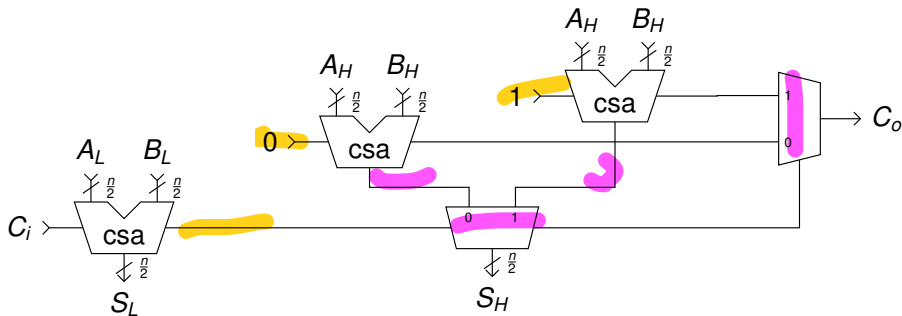
- ▶ Aufteilen in unteres und oberes Halbwort („Divide and Conquer“)
- ▶ zweiter Addierer muss auf Übertrag aus erstem Addierer „warten“
- ⇒ kritische Pfade beider Teiladdierer werden addiert
- ▶ für schnellen Addierer müssen *oberes und unteres Halbwort* gleichzeitig bearbeitet werden

Conditional Sum Adder (CSA)

siehe Ü12.3



TECHNISCHE
UNIVERSITÄT
DARMSTADT



- ▶ Übertrag vom unteren in oberes Halbwort kann nur zwei Werte annehmen
 - ▶ für beide Optionen kann das obere Halbwort schon mal vorberechnet werden
 - ▶ Auswahl des richtigen Ergebnisses, sobald tatsächlicher Übertrag bekannt
- ⇒ nach halbem CSA folgt nur noch ein MUX auf kritischem Pfad

Carry Lookahead Adder (CLA)

Motivation



TECHNISCHE
UNIVERSITÄT
DARMSTADT

	1	0	1	1	0	Übertrag
		1	0	0	1	Summand
+		1	0	1	1	Summand
<hr/>						
=	1	0	1	0	0	Summe

Carry Lookahead Adder (CLA)

Motivation



	1	0	1	1	0	Übertrag
		1	0	0	1	Summand
+		1	0	1	1	Summand
<hr/>						
=	1	0	1	0	0	Summe

- ▶ für $A_i + B_i = 1$ ist $C_i = 1$ unabhängig von C_{i-1}
- ⇒ Spalte i *generiert* einen Übertrag („generate“)
- ▶ für $A_i + B_i = 1$ ist $C_i = 1$ falls auch $C_{i-1} = 1$
- ⇒ Spalte i *leitet* Übertrag *weiter* („propagate“)
- ▶ für $A_i + B_i = 0$ ist $C_i = 0$ unabhängig von C_{i-1}
- ⇒ Spalte i *leitet* Übertrag *nicht weiter*

Carry Lookahead Adder (CLA)

Generate und Propagate pro Spalte



TECHNISCHE
UNIVERSITÄT
DARMSTADT

► Generate-Flag für Spalte i : $G_i = A_i B_i$

► Propagate-Flag für Spalte i : $P_i = A_i + B_i$

⇒ Übertrag aus Spalte i : $C_i = G_i + C_{i-1} P_i$

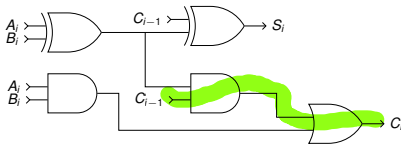
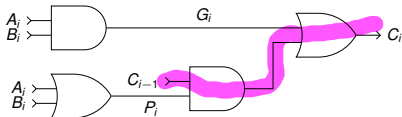
Carry Lookahead Adder (CLA)

Generate und Propagate pro Spalte



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Generate-Flag für Spalte i : $G_i = A_i B_i$
- ▶ Propagate-Flag für Spalte i : $P_i = A_i + B_i$
- ⇒ Übertrag aus Spalte i : $C_i = G_i + C_{i-1} P_i$
- ▶ erst mal kein Vorteil im Vergleich zu Volladdier:
(AND und OR auf kritischem Pfad zwischen C_{i-1} und C_i)



Carry Lookahead Adder (CLA)

Generate und Propagate über k Spalten

- ▶ Generate- und Propagate-Flags können über mehrere Spalten kombiniert werden (hier gezeigt für $k = 4$)
- ▶ k -Spalten Block *propagiert* Übertrag, wenn jede einzelne Spalte propagiert
⇒ $P_{3:0} = P_3 P_2 P_1 P_0$
- ▶ k -Spalten Block *generiert* Übertrag, wenn eine der Spalten generiert, und alle anderen Spalten darüber propagieren
⇒ $G_{3:0} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$

Carry Lookahead Adder (CLA)

Generate und Propagate über k Spalten

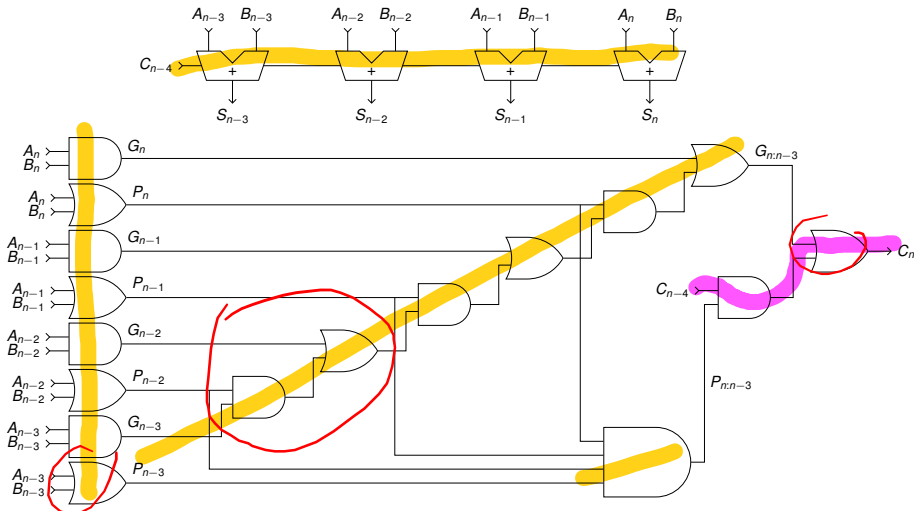
- ▶ Generate- und Propagate-Flags können über mehrere Spalten kombiniert werden (hier gezeigt für $k = 4$)
- ▶ k -Spalten Block *propagiert* Übertrag, wenn jede einzelne Spalte propagiert
 $\Rightarrow P_{3:0} = P_3 P_2 P_1 P_0$
- ▶ k -Spalten Block *generiert* Übertrag, wenn eine der Spalten generiert, und alle anderen Spalten darüber propagieren
 $\Rightarrow G_{3:0} = G_3 + \underline{P_3} G_2 + \underline{P_3} \underline{P_2} G_1 + \underline{P_3} \underline{P_2} \underline{P_1} G_0$
- ▶ Übertrag überspringt k Spalten auf einmal:

$$C_n = G_{n:n-k+1} + C_{n-k} \cdot P_{n:n-k+1}$$

$$= (G_n + P_n (G_{n-1} + \dots + (P_{n-k+2} G_{n-k+1}))) + C_{n-k} \cdot \prod_{i=n-k+1}^n P_i$$

Carry Lookahead Adder (CLA)

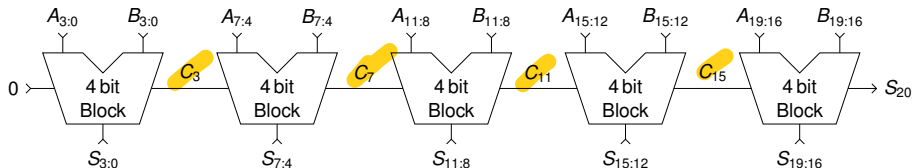
Block für $k = 4$ Spalten



Carry Lookahead Adder (CLA)

kritischer Pfad

- ▶ Propagate und Generate Signale können in allen Blöcken gleichzeitig berechnet werden
 - ▶ für große Bitbreiten N dominiert $\frac{N}{k} \cdot (t_{pd,AND} + t_{pd,OR})$
- ⇒ Blöcke möglichst groß wählen (kostet aber mehr Ressourcen)
- ▶ CLA ab etwa 16 bit schneller als RCA





- ▶ Parallel Prefix Adder
 - ▶ alle C_i per **Generate und Propagate** möglichst schnell bestimmen
- ▶ **Carry-Save Adder**
 - ▶ verwendet spezielles Datenformat, um C_i und S_i zusammen abzuspeichern



Harris 2013
S. 237



- ▶ Produkt von n und m Bit breiten Faktoren ist $n + m$ bit breit
- ▶ Teilprodukte aus einzelnen Ziffer des Multiplikators mit dem Multiplikanden
- ▶ verschobene Teilprodukte danach addiert

- ▶ Produkt von n und m Bit breiten Faktoren ist $n + m$ bit breit
- ▶ Teilprodukte aus einzelnen Ziffer des Multiplikators mit dem Multiplikanden
- ▶ verschobene Teilprodukte danach addiert

Decimal

$$\begin{array}{r} 230 \\ \times 42 \\ \hline 460 \\ + 920 \\ \hline 9660 \end{array}$$

Multiplikand
Multiplikator

Teilprodukte

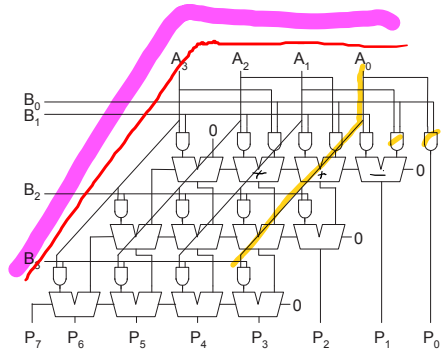
Ergebnis

$$230 \times 42 = 9660$$

Binary

$$\begin{array}{r} 0101 \\ \times 0111 \\ \hline 0101 \\ 01010 \\ 010100 \\ + 000000 \\ \hline 0100011 \end{array}$$

$$5 \times 7 = 35$$

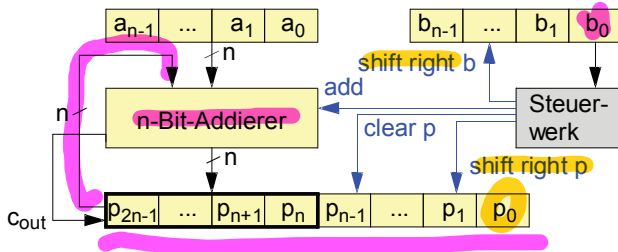


Sequentielle 4×4 Multiplikation



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Lösche Ergebnisregister p
- ▶ Addiere a auf oberes Halbwort von p falls $b_0 = 1$
- ▶ Verschiebe p einschließlich c_{out} der vorherigen Addition um eine Stelle nach rechts
- ▶ Verschiebe b um eine Stelle nach rechts



Sequentielle 4×4 Multiplikation (Implementierung aus Übung 11.2)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 module mul (input logic CLK, RST, START, input logic [3:0] A, B,  
2           output logic DONE,           output logic [7:0] Y);  
3  
4     logic [2:0] n;  
5     logic [3:0] b;  
6     logic [7:0] a, p;  
7  
8     always_ff @(posedge CLK) begin  
9       if (RST) begin  
10        {n, a, b, p, DONE, Y} <= 0;  
11      end else if (START) begin  
12        r <= 0; a <= A; b <= B; n <= 4; DONE <= 0;  
13      end else if (n > 1) begin  
14        if (b[0]) p <= p + a;  
15        a <= a << 1; b <= b >> 1; n <= n-1;  
16      end else if (n == 1) begin  
17        Y <= b[0] ? p + a : p; n <= 0; DONE <= 1;  
18      end else begin  
19        {DONE, Y} <= 0;  
20      end  
21    end  
22  endmodule
```

Weitere wichtige arithmetische Algorithmen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Division, Wurzel
- ▶ Pipelining
- ▶ Gleitkomma- / Fließkomma-Arithmetik



Harris 2013
Kap 5.2.7, 5.3

2 min Murmelphase



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Warum ist 4 bit RCA schneller als 4 bit CLA
- ▶ Wie hängt der kritische Pfad der kombinatorischen Multiplikation von den Bitbreiten der Eingänge ab?
- ▶ Wieviele 4×4 Operationen schafft der vor sequentielle Multiplizierer?



Wiederholung: Schichtenmodell eines Computers

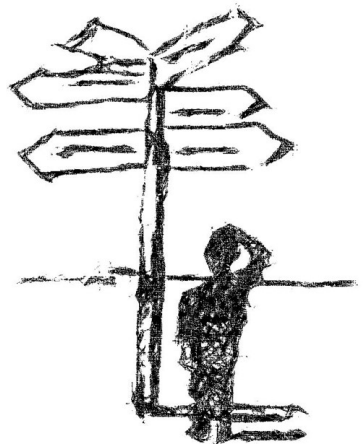


TECHNISCHE
UNIVERSITÄT
DARMSTADT

Anwendungs- software	Programme
Betriebs- systeme	Gerätetreiber
Architektur	Befehle Register
Mikro- architektur	Datenpfade Steuerung
Logik	Addierer Speicher
Digital- schaltungen	UND Gatter Inverter
Analog- schaltungen	Verstärker Filter
Bauteile	Transistoren Dioden
Physik	Elektronen

Agenda

1. Einleitung
2. Weitere arithmetische Grundsaltungen
3. Ausblick Rechnerorganisation
4. Zusammenfassung



John von Neumann, 1903 - 1957



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ wichtige Beiträge im Bereich Mathematik, Physik und Informatik
- ▶ Mitglied im Manhattan-Projekt

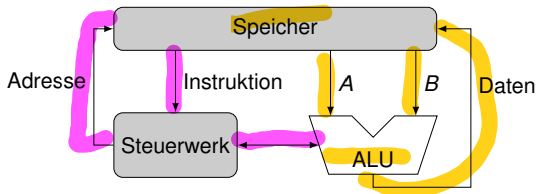


John von Neumann, 1903 - 1957



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ wichtige Beiträge im Bereich Mathematik, Physik und Informatik
- ▶ Mitglied im Manhattan-Projekt
- ▶ **Von-Neumann Rechnerarchitektur:**
 - ▶ Arithmetisch-Logische Einheit (ALU) für grundlegende Operationen
 - ▶ **gemeinsamer Speicher für Instruktionen und Daten**
 - ▶ Steuerwerk interpretiert Instruktionen



- ▶ 2D Logik-Array
 - ▶ Breite: Bit pro Speicherzelle
 - ▶ Tiefe: Anzahl der Speicherzellen
- ▶ typische Ports:
 - ▶ Adressen: identifizieren zu lesende/schreibende Speicherzelle
 - ▶ Daten: gelesene bzw. zu schreibende Daten
 - ▶ Steuersignale: Aktivierung von Schreib-/Lesezugriff
 - ▶ Takt, aber idR. kein Reset
- ▶ meist nur wenige (ein oder zwei) Schreib-/Lesezugriffe gleichzeitig
- ▶ hierarchisch organisiert (bspw. Register, Hauptspeicher, Festplatte)

SystemVerilog Beschreibung eines einfachen Speichers

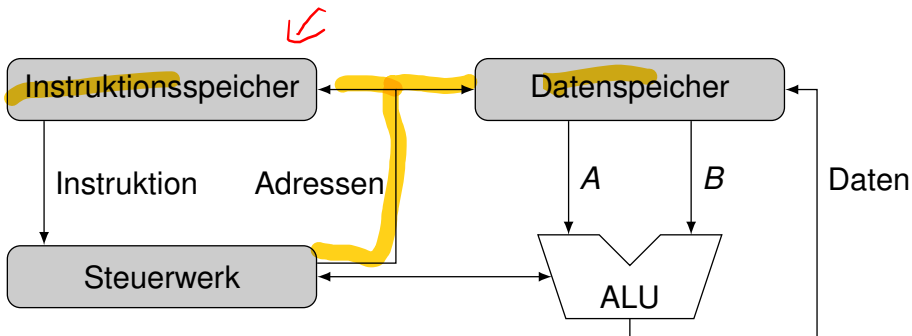


TECHNISCHE
UNIVERSITÄT
DARMSTADT

memory.sv

```
1 module memory
2     #(parameter WIDTH = 8,
3       parameter DEPTH = 16)
4     (input logic CLK, // Takt
5      input logic [$clog2(DEPTH)-1:0] ADDR, // Schreib/Lese Adresse
6      input logic [WIDTH-1:0] DI, // Dateneingang
7      input logic WEN, // Schreibzugriff aktivieren
8      output logic [WIDTH-1:0] DO); // Datenausgang
9
10    logic [WIDTH-1:0] m [0:DEPTH-1]; // 2D Speicherstruktur
11    assign DO <= m[ADDR]; // asynchrones Lesen
12    always_ff @(posedge CLK)
13        if (WEN) m[ADDR] <= DI; // synchrones Schreiben
14
15 endmodule
```

- ▶ verwendet **getrennte Instruktions- und Datenspeicher**
 - ▶ **vermeidet Engpass**, da einzelner Speicher idR. nicht Instruktion und zwei ALU-Operanden gleichzeitig lesen kann
 - ▶ **verhindert selbstüberschreibende Programme**

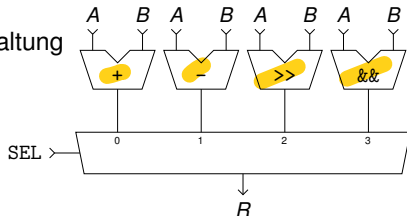


Arithmetische-/Logische Einheit (ALU)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ verknüpft zwei Operanden zu einem Ergebnis
- ▶ ausgeführte Operation kann über Steuersignal (SEL) gewählt werden
- ▶ zusätzliche Status-Ausgänge, bspw.
 - OV Überlauf (bspw. bei Addition/Subtraktion)
 - C Übertrag (bspw. bei Addition/Subtraktion)
 - Z Ergebnis ist Null
 - N Ergebnis ist negativ
- ▶ im einfachsten Fall kombinatorische Schaltung
 - ▶ kein Operator-Pipelining
 - ▶ keine sequentiellen Operatoren



SystemVerilog Beschreibung einer einfachen ALU



TECHNISCHE
UNIVERSITÄT
DARMSTADT

alu.sv

```
1  module alu #(parameter WIDTH = 8)
2      (input  logic [WIDTH-1:0] A,B, // Operanden
3       input  logic [1:0] SEL,      // Auswahlsignal
4       output logic [WIDTH-1:0] R,   // Ergebnis
5       output logic OV,Z,N);        // Statussignale
6
7      logic [WIDTH-1:0] r [0:3];
8      logic [1:0] ov;
9      add #(WIDTH) op0 (A, B, {ov[0],r[0]});
10     sub #(WIDTH) op1 (A, B, {ov[1],r[1]});
11     assign r[2] = A >> B;
12     assign r[3] = A && B;
13
14     assign R = r[SEL];
15     assign OV = SEL < 2 ? |ov : 0;
16     assign Z = r[SEL] < 0;
17     assign N = r[SEL] == 0;
18
19 endmodule
```

- ▶ lädt **Instruktionen** aus (Instruktions-)Speicher
- ▶ setzt **Steuersignale** abhängig von der aktuellen Instruktion
 - ▶ Auswahl von Operand A, B (**Leseadressen, Register-Enable, MUX-Select**)
 - ▶ **Konstanten** („**Immediates**“) als ALU-Eingang
 - ▶ **SEL** für ALU
 - ▶ Schreibadresse und Enable für **Ergebnis R**
 - ▶ **Leseadresse** für nächste Instruktion
- ▶ idR. als **endlicher Automat** realisiert
- ▶ **Status-Register** (bspw. für ALU Statussignale)
- ▶ **Programm-Zähler (PC)**
 - ▶ normalerweise nur **inkrementiert**: $PC += \text{Instruktionsbreite}$
 - ▶ **unbedingte Sprünge**: $PC += \text{Immediate}$
 - ▶ **bedingte Sprünge**: $PC += Z ? \text{Instruktionsbreite} : \text{Immediate}$



- ▶ Instruktionssatz beschreibt die Menge der ausführbaren Instruktionen
- ▶ idR Verschiedene Instruktionsformate innerhalb einer ISA:
 - ▶ [OPCODE, R_ADDR, A_ADDR, B_ADDR] für Arithmetik auf Registern
 - ▶ [OPCODE, R_ADDR, Immediate] für Laden von Konstanten
 - ▶ [OPCODE, Immediate] für Sprünge
- ▶ Speicheradressierungen, bspw.
 - ▶ absolut: Adresse = Konstante
 - ▶ indirekt: Adresse = MEM[Konstante] + Offset
- ▶ Reduced Instruction Set Computer (RISC)
 - ▶ nur die wichtigsten Operationen in der ALU
 - ▶ nicht alle Adressierungsarten für alle Operationen
- ▶ CISC Instruction Set Computer (CISC)
 - ▶ auch komplexe Operationen, jeweils für alle Adressierungsarten

- ▶ Beschreibt nicht die Unterstützten Operationen (ISA), sondern **Details der Implementierung** von Steuerwerk und Datenpfad
- ▶ **Single-Cycle** Architektur
 - ⇒ Eine Instruktion pro Takt, Ergebnis im selben Takt fertig
- ▶ **Multi-Cycle** Architektur
 - ⇒ mehrere Takte zwischen Laden von Instruktionen
- ▶ **Pipeline** Architektur
 - ⇒ Eine Instruktion pro Takt, Ergebnis mehrere Takte später fertig
- ▶ **Very Large Instruction Word** Architektur
 - ⇒ Sehr breite Instruktionen, die mehrere parallele Operationen beschreiben
- ▶ **Superskalare** Architektur (Vektorprozessoren)
 - ⇒ Sehr breite Datenwörter (mehrere Operanden auf einmal)

2 min Murmelphase



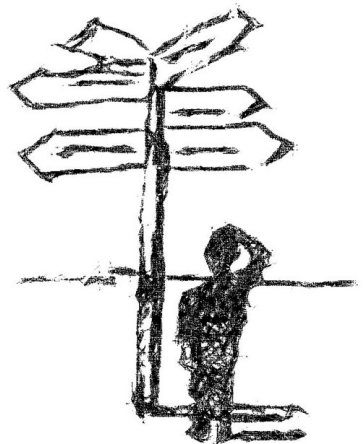
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Wieviele gleichzeitige Speicher-Zugriffe braucht Von-Neumann Rechner maximal?
- ▶ Welchen Vorteil haben RISC gegenüber CISC Prozessoren



Agenda

1. Einleitung
2. Weitere arithmetische Grundsaltungen
3. Ausblick Rechnerorganisation
4. Zusammenfassung





- ▶ Weitere arithmetische Grundsaltungen
- ▶ Ausblick Rechnerorganisation

- ▶ Nächste Vorlesung behandelt
 - ▶ HDL Zielarchitekturen (FPGA, ASIC)
 - ▶ Abschluss der Veranstaltung