Kapitel 7

Formale Spezifikation von Hardware:

1. Boolesche Ausdrücke
2. Binäre Entscheidungsdiagramme (BDDs)
3. **Anwendung: Formale Verifikation**

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Dr. Tobias Schubert, Dr. Ralf Wimmer

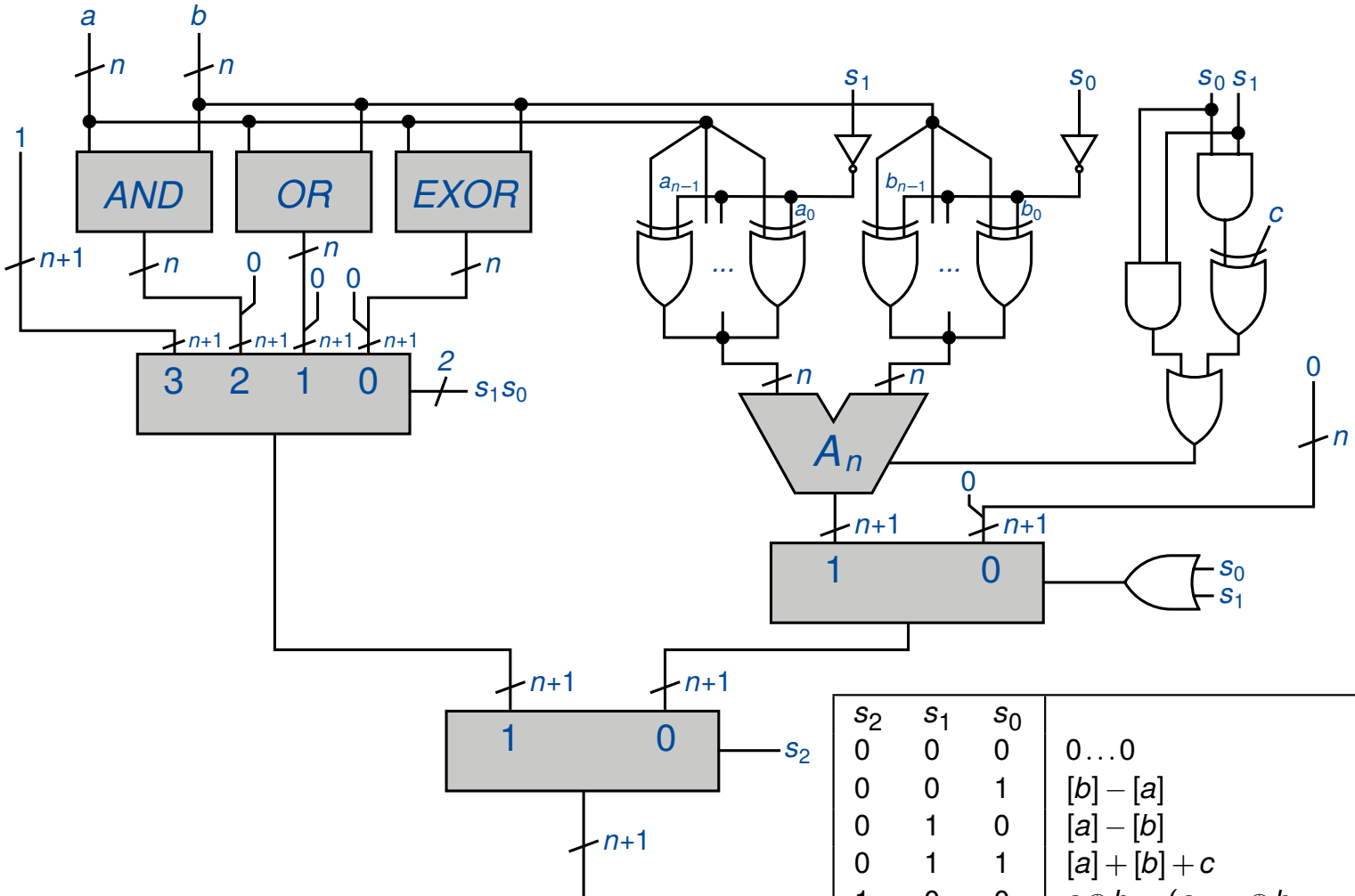
Professur für Rechnerarchitektur

WS 2016/17

Formale Verifikation

- **Ziel**: Mit formalen Methoden weitgehend automatisch feststellen, ob der Entwurf korrekt ist.
  - **Hier**: Nur grobe Prinzipien, es gibt eine Vielzahl von Weiterentwicklungen, die auch im Einsatz sind.
- Äquivalenz-Check: Verleich der Spezifikation mit der Implementierung.
  - **Hier**: Komplexeres Beispiel (ein „Slice" der ReTI-CPU) mit Hilfe von BDDs.
- Eigenschafts-Check: Überprüfung, ob bestimmte Eigenschaften gelten.
  - **Hier**: Abwesenheit von Bus Contention auf einem Bus.

Schaltrealisierung der ALU



s_2	s_1	s_0	
0	0	0	$0\ldots0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1\ldots1$

„0-tes Slice der ALU”

- Das „0-te Slice” ist der Teil der kombinatorischen Logik, der den 0. Ausgang der ALU berechnet.
- Das Vorgehen für die gesamte CPU wäre identisch, die BDDs jedoch (sehr) viel größer.
- **Vorgehen**:
  - Erstelle die Funktionstabelle für den 0. Ausgang der ALU („Spezifikation”) und berechne das BDD dazu.
  - Bestimme die Hardware, welche den 0. Ausgang ansteuert („Implementierung”) und berechne das BDD dazu.
  - Sind die BDDs gleich, ist die Implementierung korrekt!

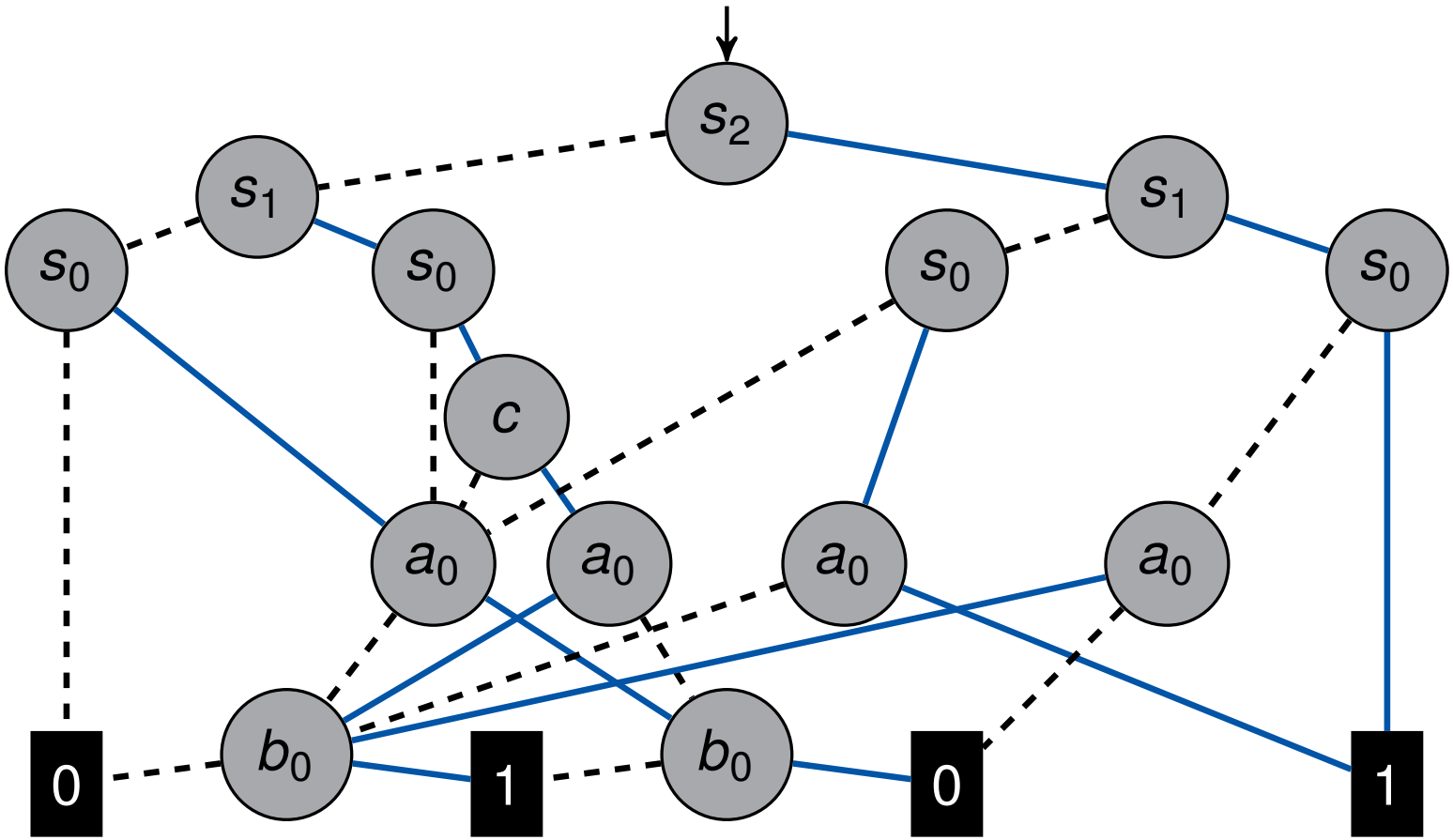
Spezifikation für das 0-te Slice

s_2	s_1	s_0	0. Ausgang
0	0	0	$(0\ldots0)_0 = 0$
0	0	1	$([b]-[a])_0 = b_0 \oplus a'_0 \oplus 1 = a_0 \oplus b_0$
0	1	0	$([a]-[b])_0 = a_0 \oplus b'_0 \oplus 1 = a_0 \oplus b_0$
0	1	1	$([a]+[b]+c)_0 = a_0 \oplus b_0 \oplus c$
1	0	0	$(a \oplus b)_0 = a_0 \oplus b_0$
1	0	1	$(a \vee b)_0 = a_0 \vee b_0$
1	1	0	$(a \wedge b)_0 = a_0 \wedge b_0$
1	1	1	$(1\ldots1)_0 = 1$

s_2	s_1	s_0	
0	0	0	$0\ldots0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1\ldots1$

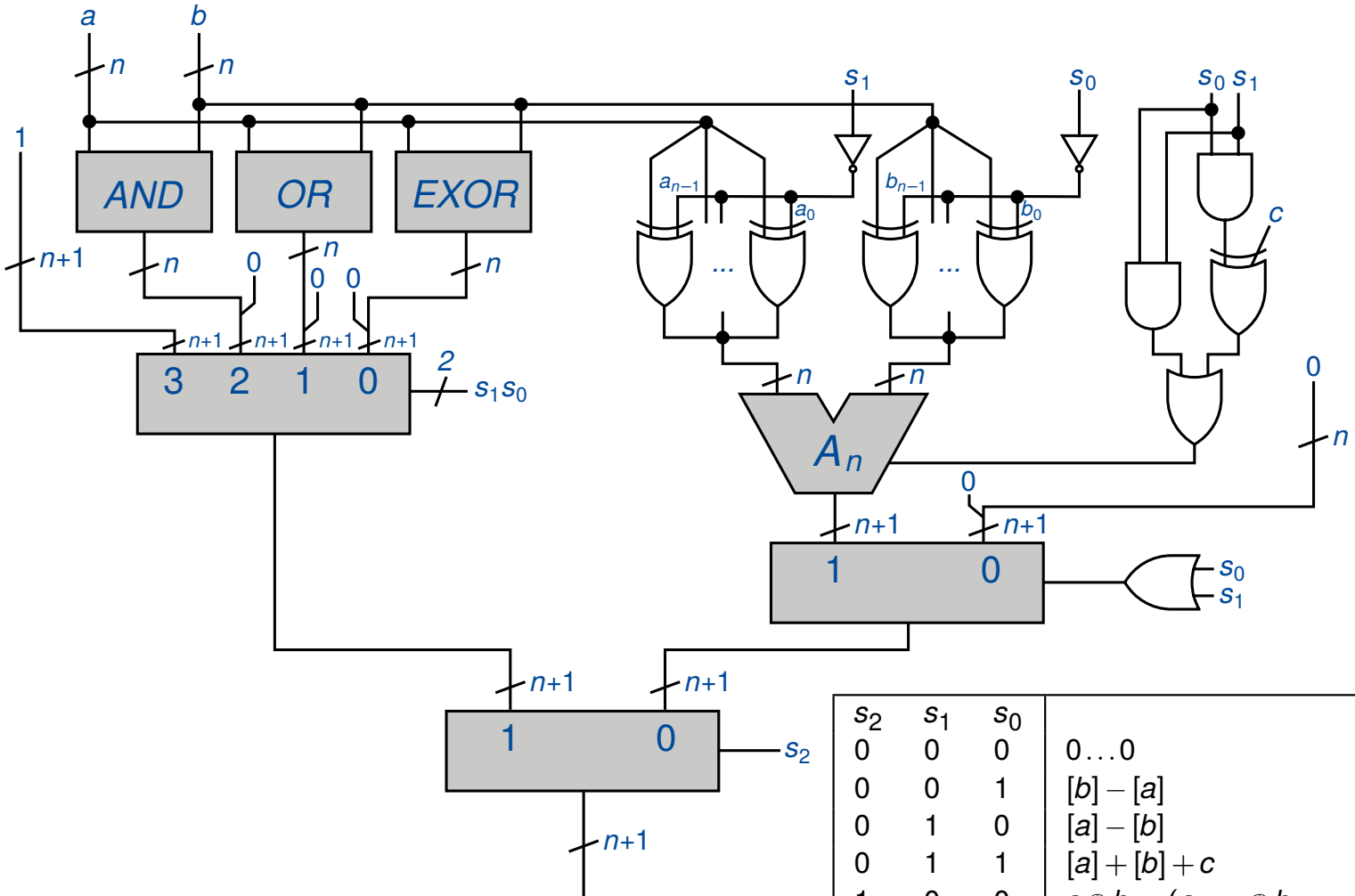
UNI FREIBURG

BDD für die Spezifikation



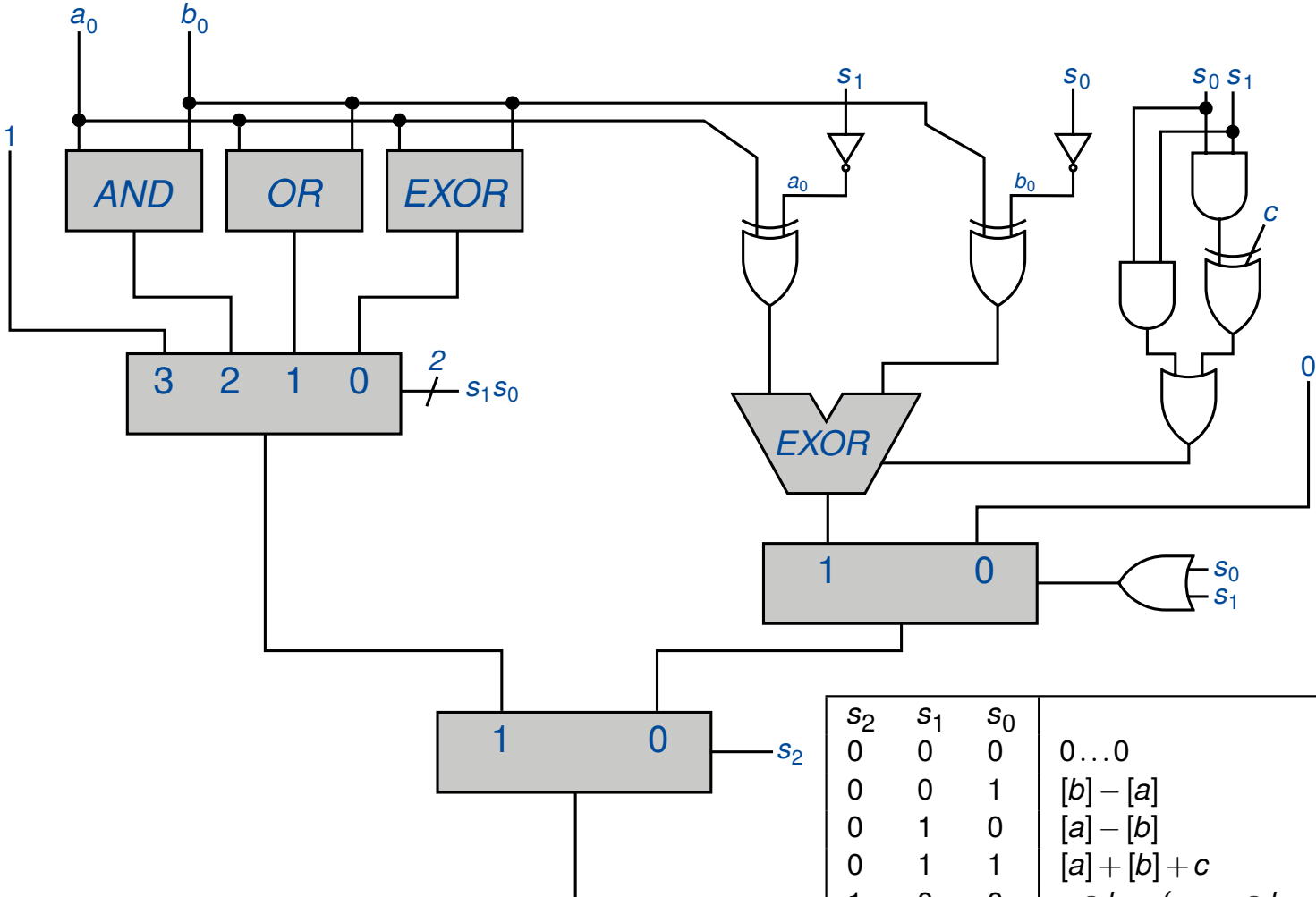
- Variablenordnung: s_2, s_1, s_0, c, a_0, b_0.
- Reduziert bis auf Terminalknoten (wegen Übersichtlichkeit).

Schaltrealisierung der ALU



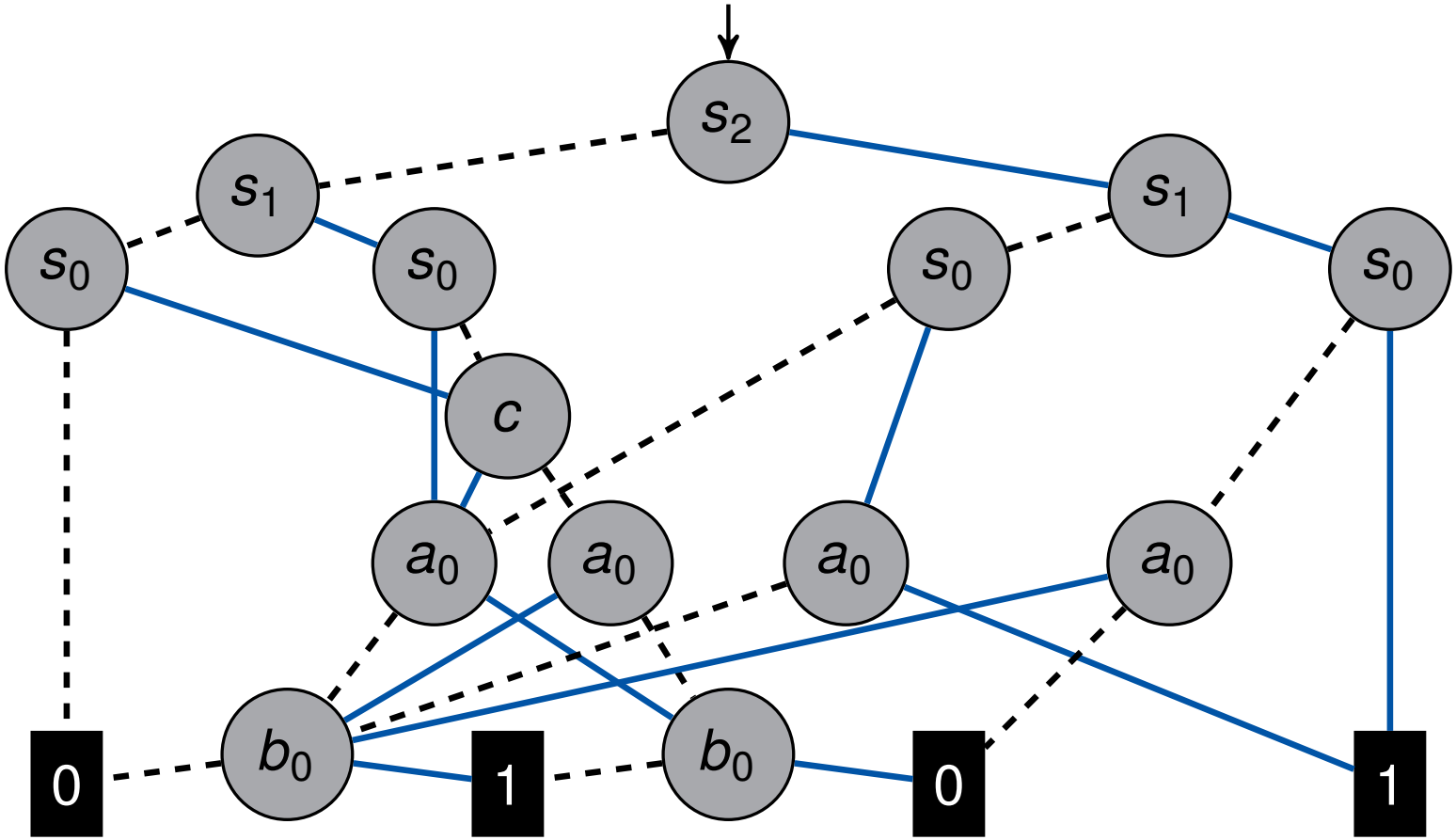
s_2	s_1	s_0	
0	0	0	$0\ldots0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1\ldots1$

Hardware für das „0-te Slice”



s_2	s_1	s_0	
0	0	0	$0 \ldots 0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1 \ldots 1$

BDD für die Schaltrealisierung



- Das BDD entspricht nicht dem für die Spezifikation!
- Irgendwo muss ein Entwurfsfehler sein!

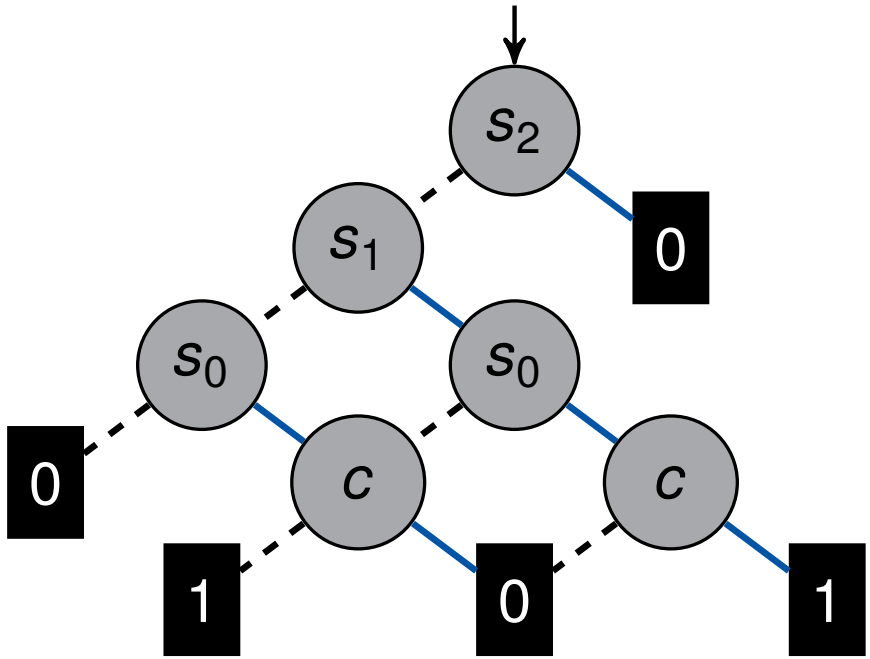
Im Entwurf ist ein Fehler – aber wo?

- Wir wollen nicht nur wissen, dass der Entwurf fehlerhaft ist, sondern den Fehler finden (und dann beheben)!
- Dies wird „Diagnose von Entwurfsfehlern” genannt.
- Ein mögliches Vorgehen: Finde eine Belegung der Eingänge, für die Spezifikation und Implementierung nicht übereinstimmen.
- Wir suchen also eine Belegung von (s_2, s_1, s_0, c, a_0, b_0), für welche gilt:
  $Implem.(s_2, s_1, s_0, c, a_0, b_0) \neq Spez.(s_2, s_1, s_0, c, a_0, b_0)$.
- Mit BDDs ist das ganz einfach: Berechne BDD für
  $Implem.(s_2, s_1, s_0, c, a_0, b_0) \oplus Spez.(s_2, s_1, s_0, c, a_0, b_0)$
  und betrachte einen beliebigen Pfad zu 1-Terminalknoten.

UNI FREIBURG

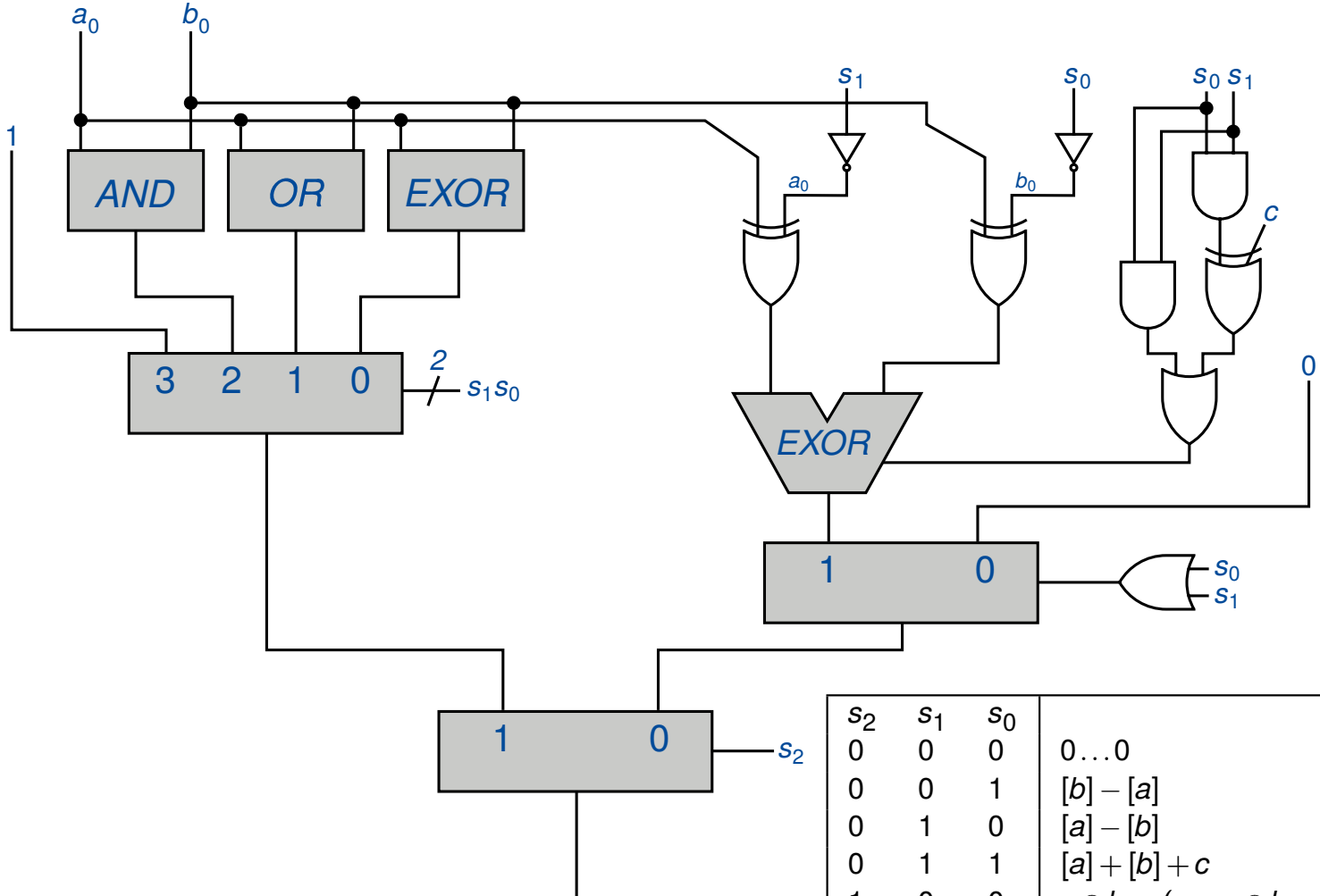
Fehlerdiagnose

- Ergebnis der XOR-Verknüpfung der beiden BDDs:



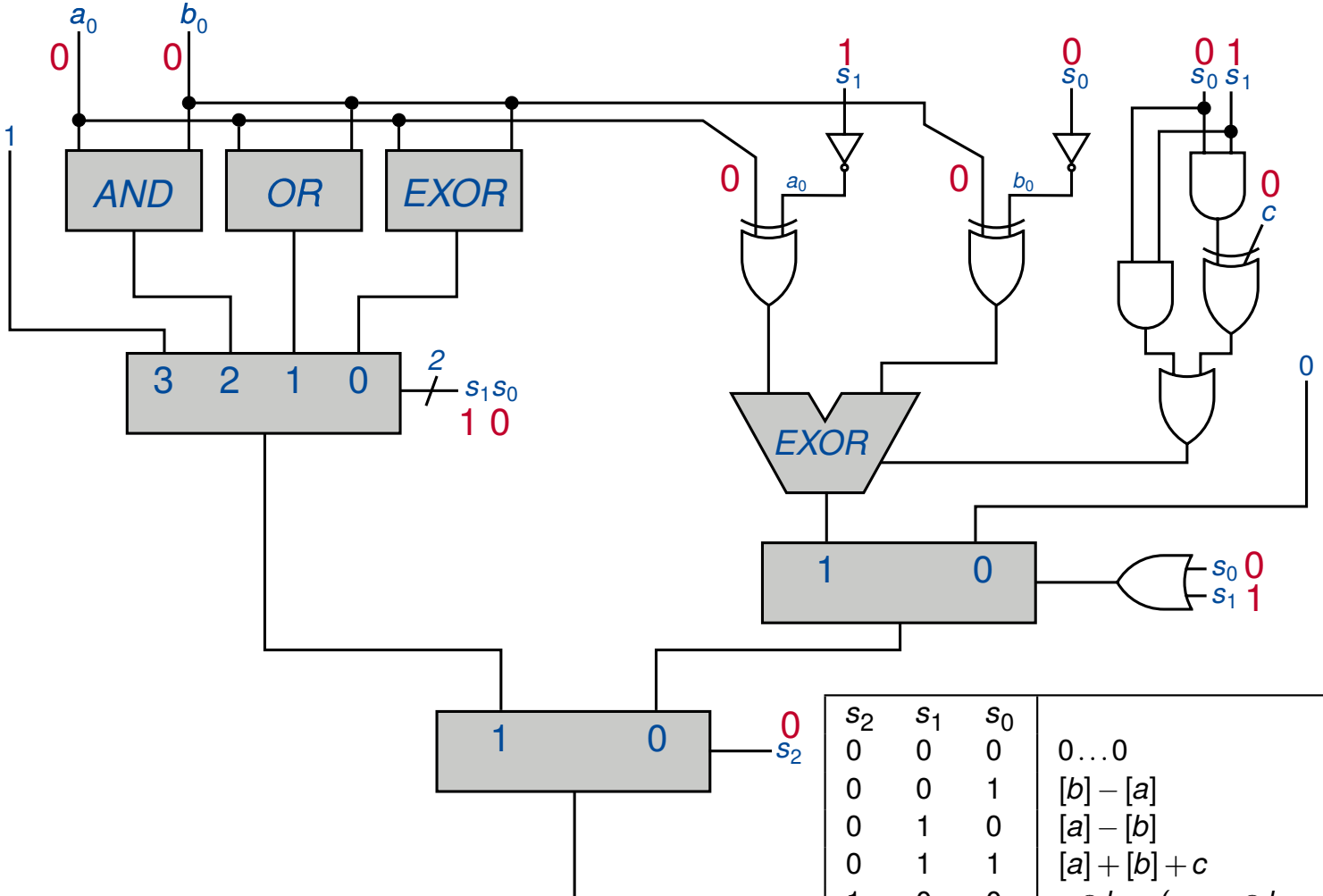
- Ein Pfad zum 1-Terminal: $s_2 = 0$, $s_1 = 1$, $s_0 = 0$, $c = 0$.
- a_0, b_0 sind unbestimmt; setze z.B. $a_0 = 0$, $b_0 = 0$.
- Die ALU soll also $[a] - [b]$ berechnen (da $s_2 s_1 s_0 = 010$).

Simulation der gefundenen Belegung (1/3)



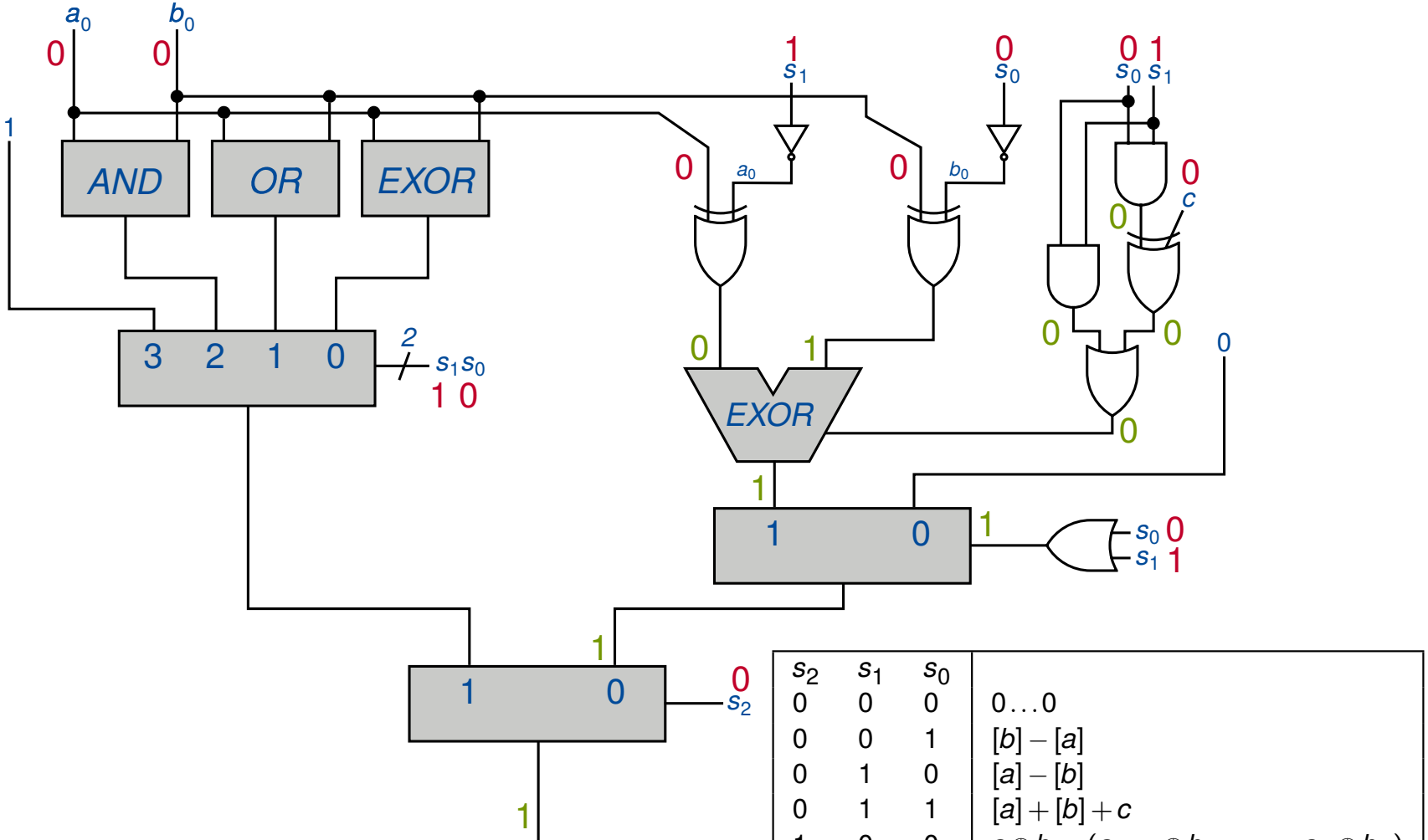
s_2	s_1	s_0	
0	0	0	$0 \ldots 0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1 \ldots 1$

Simulation der gefundenen Belegung (2/3)



s_2	s_1	s_0	
0	0	0	$0 \ldots 0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1 \ldots 1$

Simulation der gefundenen Belegung (3/3)



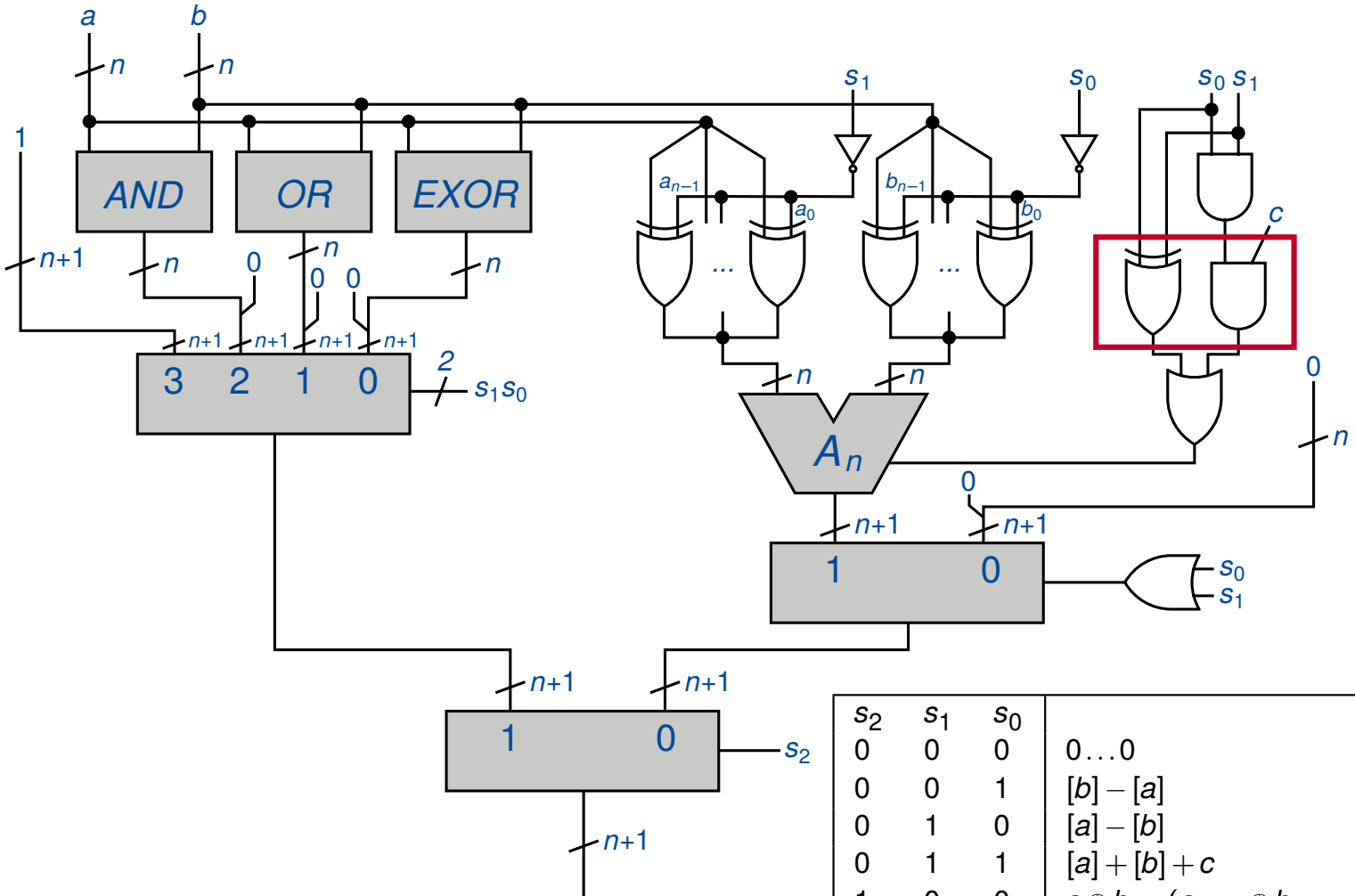
s_2	s_1	s_0	
0	0	0	$0 \ldots 0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1 \ldots 1$

Analyse der Simulationsergebnisse

- Es werden zwei Zahlen subtrahiert, die mit „0” enden. Das Ergebnis soll mit „1” enden.
- Das kann nicht sein, die ALU subtrahiert falsch.
- Die Dateneingänge des Addierers sind richtig.
- Der Übertragseingang des Addierers ist aber falsch, er hätte bei Subtraktion 1 sein müsen!
- Der Fehler muss in der Logik liegen, die den Übertragseingang steuert. Das sind 4 Gatter.
- Tatsächlich sind ein AND- und ein XOR-Gatter vertauscht. Vgl. die (richtige) Folie aus Kap. 3.5.

UNI FREIBURG

Original-Folie aus Kapitel 3



s_2	s_1	s_0	
0	0	0	$0\ldots0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$
1	1	1	$1\ldots1$

Was hat die formale Analyse gebracht?

- Wir mussten den Fehler nach wie vor manuell suchen, ...
- ... allerdings in den 4 Gattern und nicht in der gesamten Schaltung!
- Es gibt (komplexere) Diagnose-Ansätze, die voll-automatisch (oder fast voll-automatisch) arbeiten.
- Außerdem würden wir, wenn wir die Analyse mit der korrigierten Schaltung wiederholen, wissen, dass die Implementierung nun korrekt ist und wir keine weiteren Fehler suchen müssen!

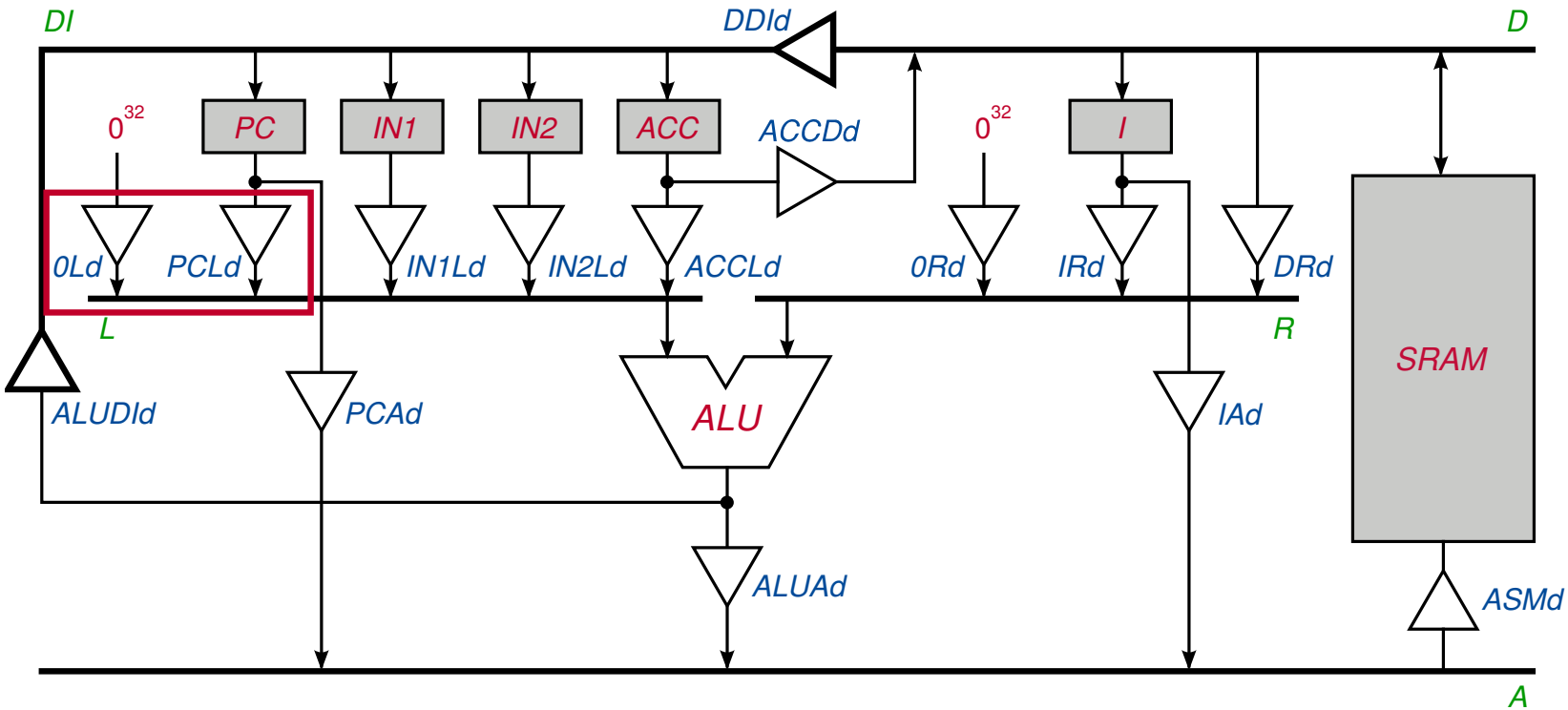
Ist die Analyse den Aufwand wert?

- Es ist sehr kompliziert, BDDs aufzuschreiben und ihre Verknüpfungen auszurechnen.
- Dadurch ist es auch fehleranfällig.
  - Woher weiß man, dass man bei den BDD-Operationen keinen Rechen- oder Schreibfehler gemacht hat?
- **Antwort**: Software-Werkzeuge, die Manipulation von BDDs übernehmen!
  - Diese können BDDs mit Millionen Knoten repräsentieren und in vertretbarer Zeit verarbeiten.
  - Spezifikation und Implementierung werden aus Dateien eingelesen und BDDs automatisch erzeugt/verglichen.
  - Standard in modernen Entwurfsabläufen!

Eigenschaftsprüfung

- **Ziel**: Nachweis, dass eine bestimmte Eigenschaft immer gilt, manchmal gilt oder nie gelten kann.
- Es gibt spezielle Sprachen, um solche Eigenschaften zu formulieren.
- An dieser Stelle soll das Vorgehen am Beispiel von Bus Contention auf Bus L von ReTI illustriert werden.
- Konkret wollen wir zeigen, dass Treiber *PCLd* und *0LD* nie gleichzeitig enabled sind.

Zur Erinnerung: ReTI-Aufbau



Eigenschaftsprüfung: Bus Contention (1/2)

- Weise nach, dass es keine Eingangsbelegung gibt, für die $/PCLdoe$ und $/0Ldoe$ beide aktiv sind.
  - $/PCLdoe$ enabled in der Execute-Phase ($E = 1$) für Befehle JUMP ($I[31:30] = 11$), Compute-Befehle ($I[31:30] = 00$) mit $D = PC$ ($I[25:24] = 00$) und für MOVE-Befehl ($I[31:28] = 1011$) mit $S = PC$ ($I[27:26] = 00$).
  - $/0Ldoe$ enabled in der Execute-Phase ($E = 1$) für Befehl LOADI ($I[31:28] = 0111$).

Eigenschaftsprüfung: Bus Contention (2/2)

- **Eigenschaft**: Für alle Belegungen der relevanten Signale $(E, I_{31}, I_{30}, I_{29}, I_{28}, I_{27}, I_{26}, I_{25}, I_{24})$ gilt stets:

  $(/PCLdoe(E, I_{30}, I_{29}, I_{28}, I_{27}, I_{26}, I_{25}, I_{24})$
  $\vee\ /0Ldoe(E, I_{30}, I_{29}, I_{28}, I_{27}, I_{26}, I_{25}, I_{24})) = 1$.

  - **Bedeutung**: Zu jedem Zeitpunkt muss mindestens eines der Signale 1 sein, sie sind nie gleichzeitig 0 (aktiv).

- **Vorgehen**: Berechne BDD für $/PCLdoe$, BDD für $/0Ldoe$ und die OR-Verknüpfung der beiden BDDs. Diese muss 1 sein, d.h. aus nur einem 1-Terminalknoten bestehen.

  - Falls nicht, liefern die Pfade zum 0-Terminalknoten die Belegungen, für welche die Eigenschaft verletzt ist!

Eigenschaftsprüfung: Diskussion

- Wir haben eine selbstverständliche Aussage bewiesen (Signale bei unterschiedlichen Befehlen aktiv).
  - Der Entwerfer hätte aber, etwa bei der Logik-Optimierung, einen Fehler machen können, die zur Verletzung der Eigenschaft geführt hätte.

- Wir sind noch lange nicht fertig.
  - Auf dem L-Bus und auf anderen Bussen gibt es noch viele Möglichkeiten, wie Bus Contention auftreten kann.
  - Selbst wenn wir sämtliche Quellen für Bus Contention ausschließen, wissen wir noch nicht, dass der Prozessor korrekt implementiert ist. Wir wissen nur, dass keine Bus Contention auftritt.

Formale Verifikation in der Praxis

- Eigene Produktklasse der Entwurfsautomatisierungsindustrie (auch in Deutschland).
- Äquivalenzprüfung ist Routine-Bestandteil beim Entwurf von komplexen Hardware-Produkten.
- Eigenschaftsprüfung spielt im Hardware- und im Software-Bereich eine immer größere Rolle, der Einsatz ist jedoch mit Schwierigkeiten verbunden.
  - Formulierung von Eigenschaften erfordert hochqualifizierte (und teure) Mitarbeiter.
  - Es ist oft nicht klar, wann genug Eigenschaften erstellt wurden und das Produkt als korrekt anzusehen ist.

Zusammenfassung: Formale Spezifikation

- Boolesche Funktionen, boolesche Ausdrücke, kombinatorische Schaltkreise und BDDs sind ineinander transformierbar.
- Formale Verifikation kann Entwurfsqualität enorm steigern und die Entwurfszeiten reduzieren.
- Ein Großteil des Aufwandes kann von einem Rechner geleistet werden.
- Voraussetzung dafür ist eine formal-mathematische Beschreibung, die der Rechner ohne menschliche Hilfe interpretieren kann.