

Kapitel 4

Sequentielle Logik:

1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. **Anwendung: Datenpfade von ReTI**

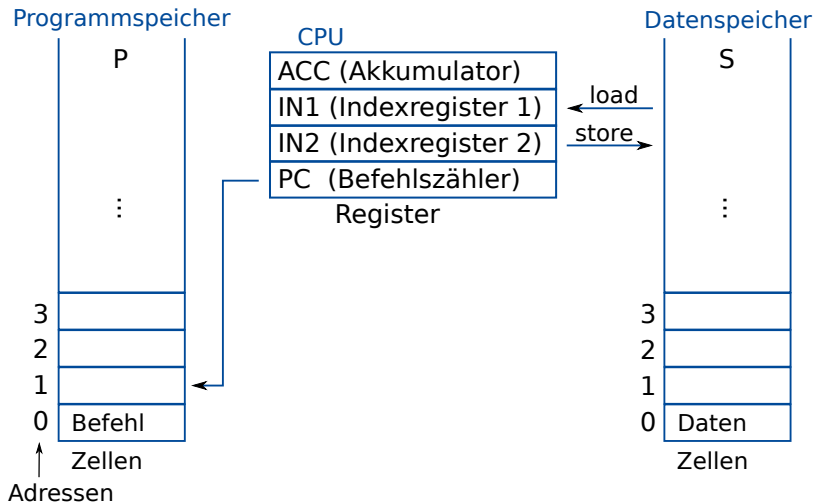
Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik

WS 2015/16

Zur Erinnerung: ReTI bisher



Zur Erinnerung: Datenpfade von ReTI

- ReTI besteht aus
 - 4 benutzersichtbaren Registern *PC*, *ACC*, *IN1*, *IN2*
→ Realisiert durch Zähler (*PC*) bzw. Register (Wortbreite jeweils 32 Bit).
 - Einem 2^{32} -Wort-Speicher, der Daten und Befehle enthält
→ Realisiert durch SRAM.
- ReTI unterstützt Load-/Store-, Compute-, Indexregister- und Sprungbefehle.

31 30	29 ... 24	23 ... 0
Typ	Spezifikation	Parameter <i>i</i>

Zur Erinnerung: ReTI-Befehle im Überblick

Load	31	30	29	28	27	26	25	24	23	...	0
	0	1	M		*		D		i		

Store	31	30	29	28	27	26	25	24	23	...	0
	1	0	M		S		D		i		

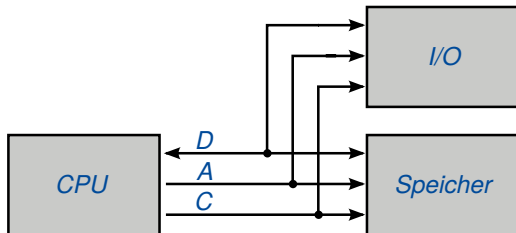
Compute	31	30	29	28	27	26	25	24	23	...	0
	0	0	MI		F		D		i		

Jump	31	30	29	28	27	26	25	24	23	...	0
	1	1	C			*			i		

M - Modus ; S - Source ; D - Destination ; MI - memory/immediate ;
F - Function ; C - Condition

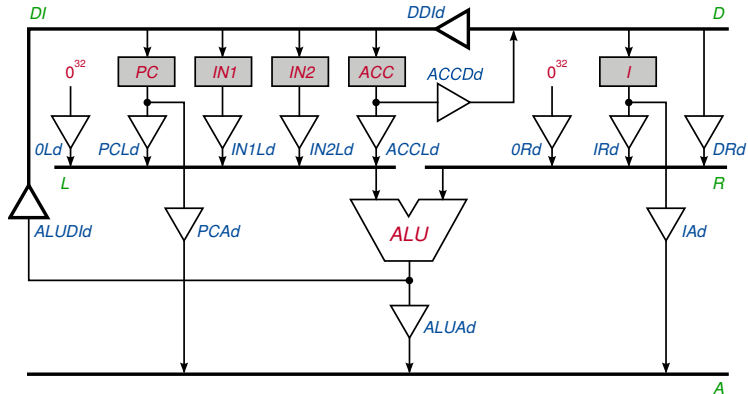
Umsetzung von ReTI: Externe Sicht

- **3-Bus-Architektur** zur Ansteuerung von Speicher und I/O-Geräten:
 - 32 Bit breiter **Datenbus** $D = D[31 : 0]$,
 - 32 Bit breiter **Adressbus** $A = A[31 : 0]$,
 - **Kontrollbus** C (Breite später festgelegt).



- *CPU* besteht aus:
 - Zähler *PC*.
 - 3 für Benutzer sichtbaren Registern *ACC*, *IN1*, *IN2*,
 - Instruktionsregister *I*,
 - *ALU*,
 - *CPU*-internen Bussen:
 - *L*, *R* für linken bzw. rechten Operanden der *ALU*,
 - internem Datenbus *DI*.
- Register, *PC*, *ALU*, Busse und die zugehörigen Treiber sind 32 Bit breit.

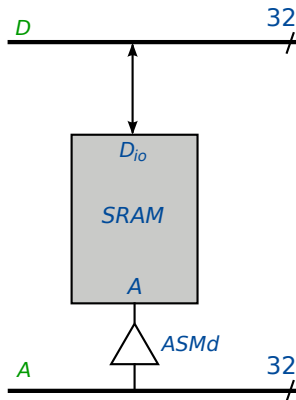
Interner Aufbau der ReTI-CPU



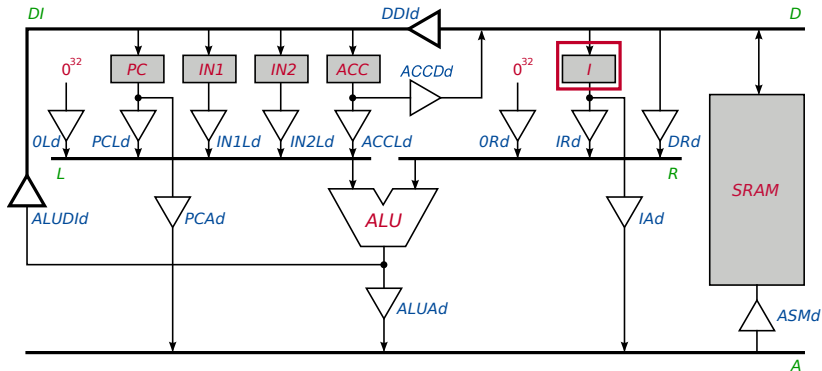
- Namenskonvention für Treiber: Treiber zwischen Bus/Baustein X und Bus/Baustein Y : XYd ;
 - Output-Enable-Signal: $XYdoe$.
 - OLd und ORd können 0^{32} auf L bzw. R legen.
- Busse A und D sind an den Speicher (sowie I/O-Geräte) angeschlossen, s. nächste Folie.
- Das Bild enthält keine Steuerleitungen:
 - Output-Enable-Signale der Treiber,
 - Funktionsauswahl der ALU,
 - Clock-Signale und "Clock-Enable-Signale" der Register.

Speicher SM von ReTI

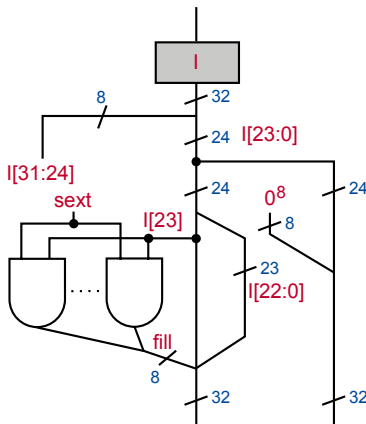
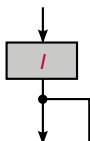
- Datenein- und ausgänge mit Datenbus D der CPU verbunden.
- Adressleitungen mit Adressbus A der CPU verbunden.
- Treiber $ASMd$ immer enabled.



Verfeinerung des Schaltbilds



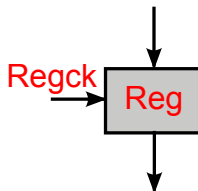
- Im Instruktionsregister *I* steht der gerade verarbeitete Befehl.
 - *I*[31 : 24]: Befehlskodierung (für die im Schaltbild nicht dargestellten Steuersignale benötigt).
 - *I*[23 : 0]: Speicheradresse oder Konstante für die ALU.
 - Speicheradresse wird mit acht Nullen aufgefüllt.
 $0^8 I[23 : 0]$ wird auf Bus *A* gelegt (*IAd* enabled).
 - Natürliche 24-Bit-Konstante wird mit acht Nullen aufgefüllt.
 $0^8 I[23 : 0]$ wird auf *R* gelegt (*IRd* enabled).
 - Ganzzahlige 24-Bit-Konstante wird vorzeichenerweitert.
sext(*I*[23 : 0]) wird auf *R* gelegt (*IRd* enabled).



- Neues Kontrollsignal **sext**:
 - $sext = 0$: Ersetze $I[31:0]$ durch $0^8 I[23:0]$.
 - $sext = 1$: Ersetze $I[31:0]$ durch $sext(I[23:0])$.

Weitere Verfeinerung für Register (1/3)

- Im Buch von Keller / Paul werden die Clockeingänge aller Register **Reg** mit einem Clocksignal **Regck** verbunden, das **durch die Kontrolllogik berechnet wird**.
- ⇒ Datenübernahme zu ausgewählten Zeitpunkten



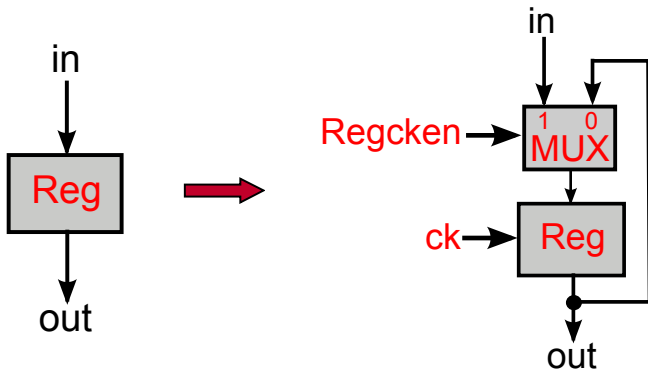
- Vorgehen bei Realisierung der ReTI auf einer Platine mit diskreten Bausteinen ok.!
- Nicht ok. bei heutigen Designs, die Prozessoren auf einem einzigen Chip integrieren.

Weitere Verfeinerung für Register (2/3)

- Gründe für “Designrule”, die es verbietet, Clockeingänge mit berechneten Datensignalen zu verbinden:
 - Spezielle Methoden (“Clock-Tree-Synthese”) gewährleisten, dass die steigende Flanke der globalen Clock an allen Clockeingängen zum gleichen Zeitpunkt ankommt (z.B. durch Ausgleich des Effekts unterschiedlicher Leitungsverzögerungen m.H. von Treibern). Datensignale auf Clockeingängen verhindern Clock-Tree-Synthese.
 - Heutige Werkzeuge zur automatischen Timing-Analyse (und Berechnung der maximalen Clockfrequenz) sind nicht in der Lage, mit berechneten Clocksignalen umzugehen.
 - Spezielle Anforderungen an “Flankensteilheit” der Clocksignale (siehe Kapitel über physikalische Eigenschaften)

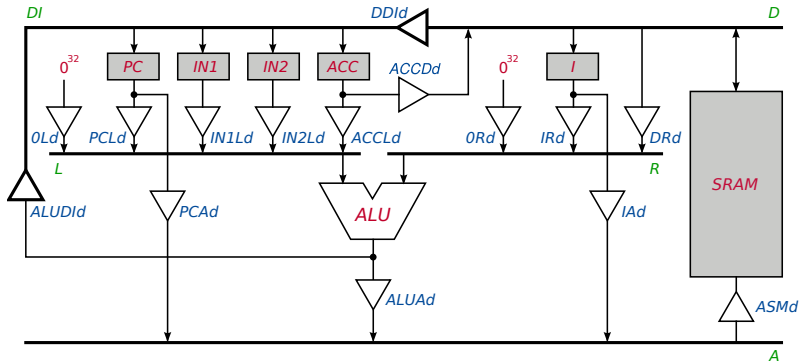
Weitere Verfeinerung für Register (3/3)

- Transformation bzw. Verfeinerung:

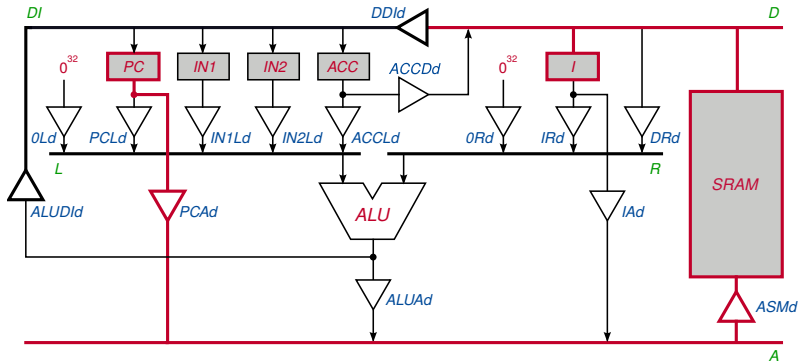


- Zwei sich abwechselnde Phasen der *CPU*:
 - **Fetch-Phase**: Lädt nächsten auszuführenden Befehl aus Memory ins **Instruktionsregister** / der *CPU*.
 - **Execute-Phase**: Befehl, der in / steht, wird ausgeführt.
- Vorgehen:
 - 1 Definition der **Datenpfade**, d.h. der benötigten Datenverbindungen zwischen den Komponenten der *CPU*.
 - 2 Herleitung der **Kontrollsignale** zur Ansteuerung der im Punkt 1 hergeleiteten Datenpfade.
 - Treiber-OE, *ALU*-Funktionsselektion, Enable für Register-Clocks.
 - 3 Sequentielle Synthese.

Datenpfade: Fetch-Phase

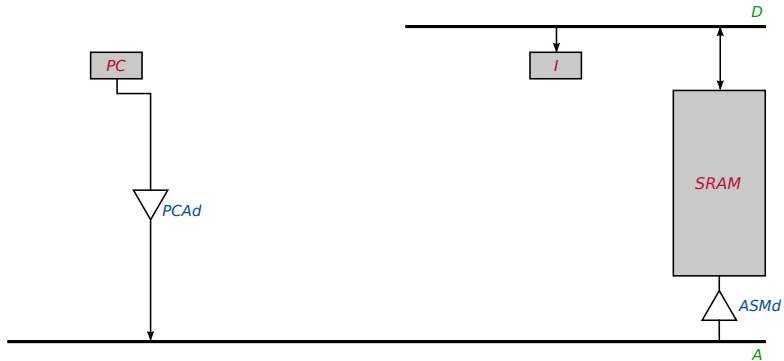


Datenpfade: Fetch-Phase



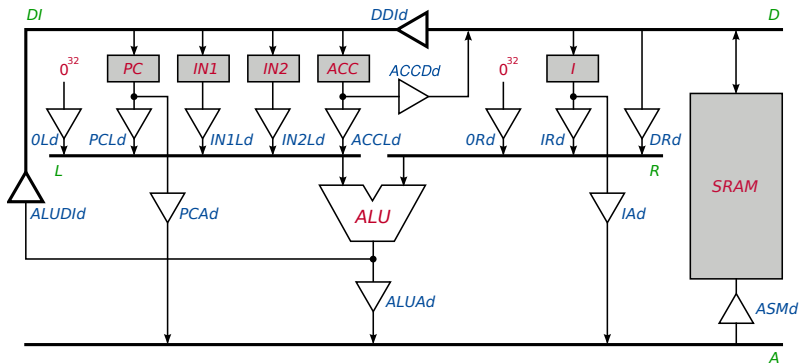
- In der Fetch-Phase muss:
 - Bus *A* mit *PC* verbunden sein, d.h. *PCAd* enabled.
 - Register *I* den Wert von Bus *D* übernehmen, d.h. *Icken* muss enabled werden.
 - Alle anderen Treiber (außer denen, die stets enabled sind) sind disabled, um Bus Contentions zu vermeiden.
- Die Steuersignale müssen in der Fetch-Phase entsprechend gesetzt sein.
 - Z.B. */PCAdoe* = 0, */ALUAdoe* = 1, usw. (active low!)

Fetch: Die durchgeschalteten Pfade

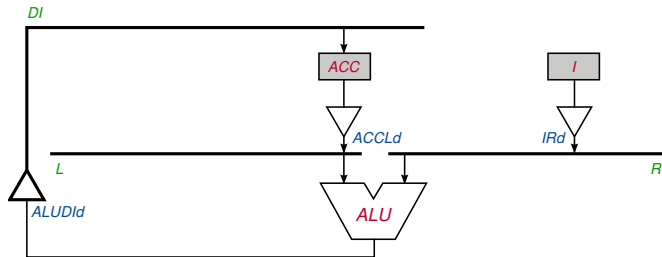


- Betrachte unterschiedliche die Execute-Phase unterschiedlicher Befehlstypen.
 - *Compute Immediate,*
 - *Compute Memory,*
 - *JUMP,*
 - *LOAD,*
 - *LOADIN1 (LOADIN2 analog),*
 - *LOADI,*
 - *STORE,*
 - *STOREIN1,*
 - *MOVE.*

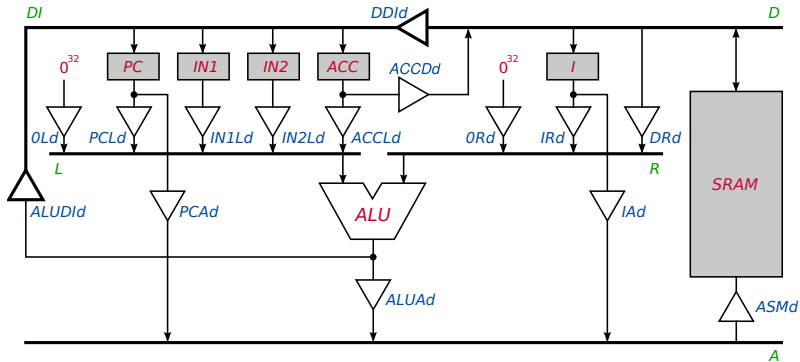
Datenpfade: *Compute Immediate* (1/2)



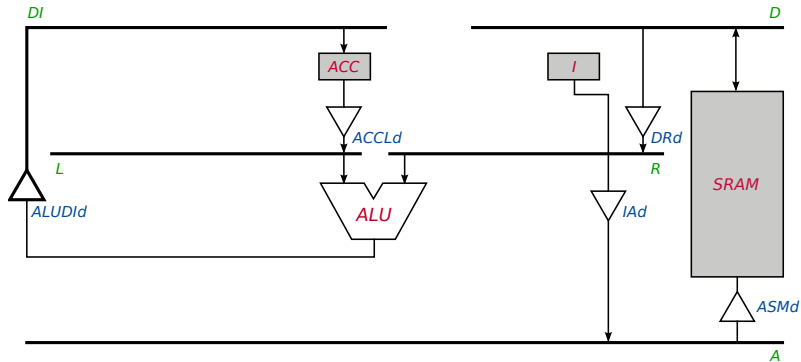
Datenpfade: *Compute Immediate* (2/2)



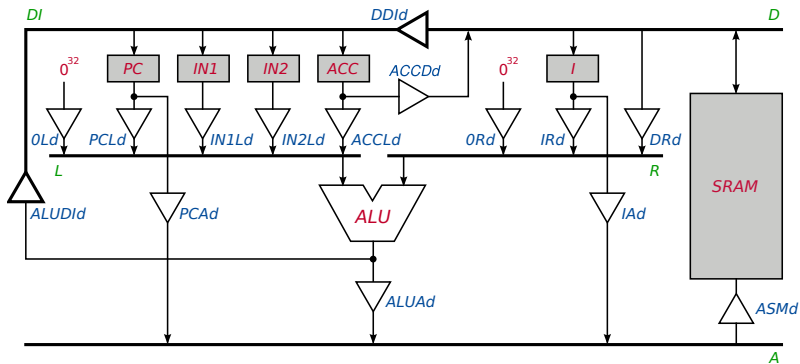
Datenpfade: *Compute Memory* (1/2)



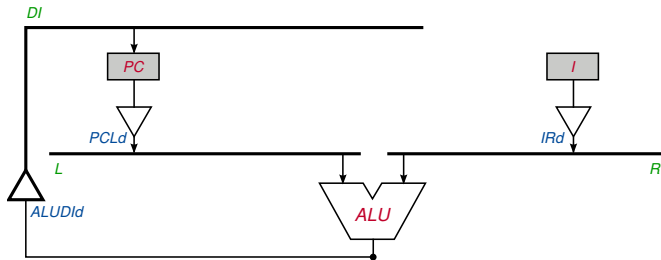
Datenpfade: *Compute Memory* (2/2)



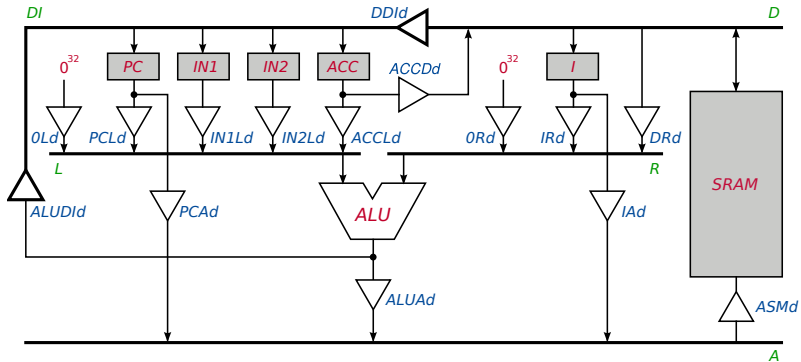
Datenpfade: *JUMP* (1/2)



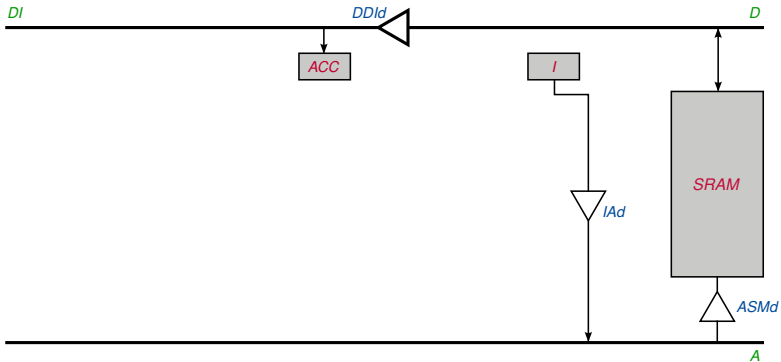
Datenpfade: *JUMP* (2/2)



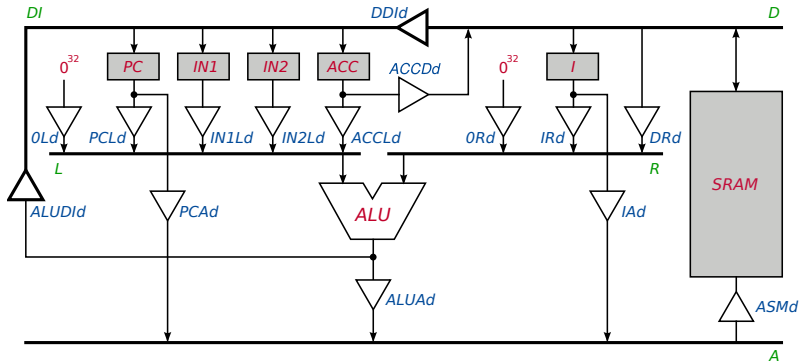
Datenpfade: *LOAD i* (1/2)



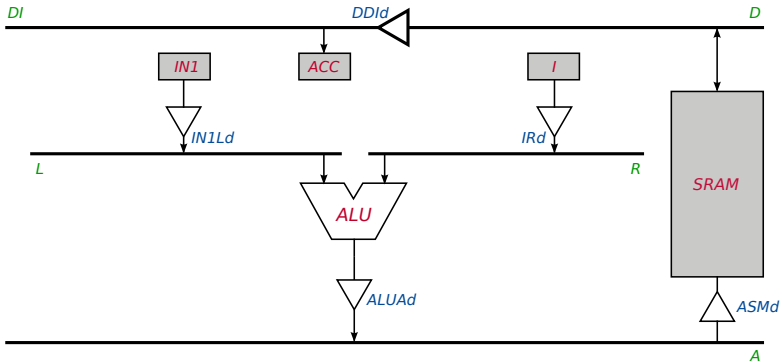
Datenpfade: *LOAD i* (2/2)



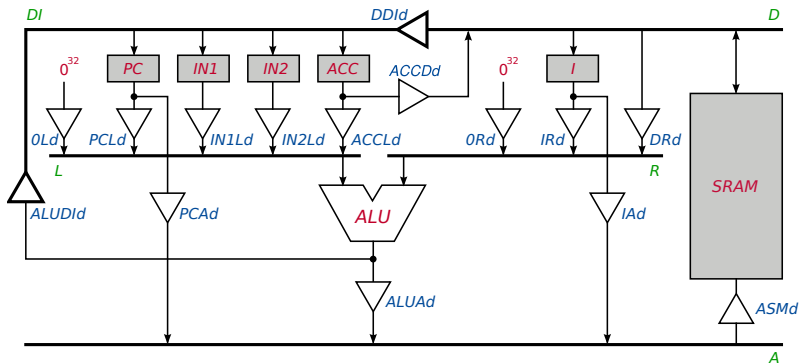
Datenpfade: *LOADIN1 i* (1/2)



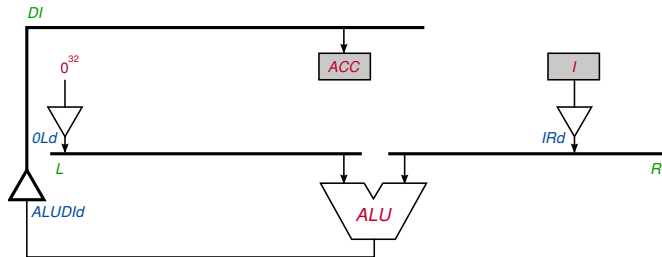
Datenpfade: *LOADIN1 i* (2/2)



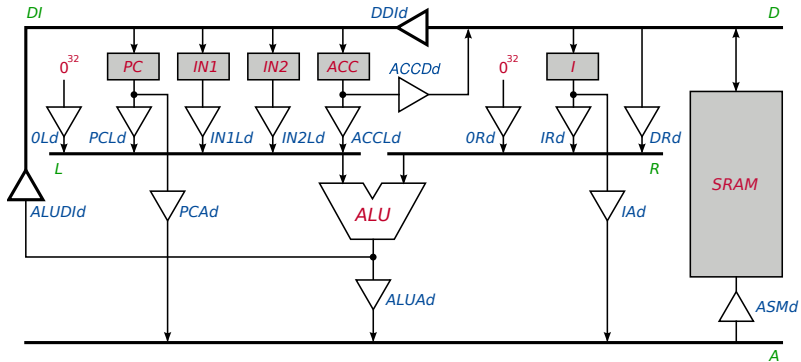
Datenpfade: *LOADI i* (1/2)



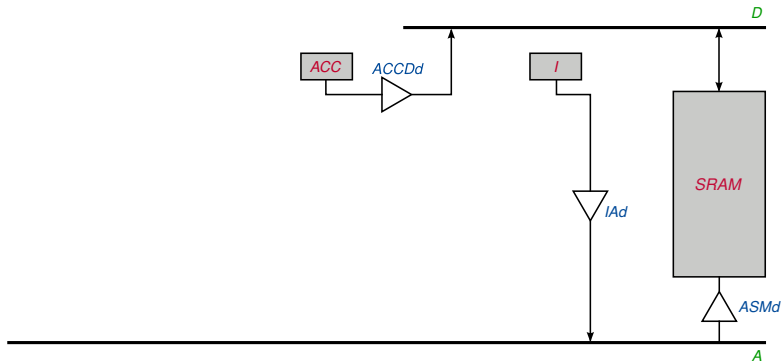
Datenpfade: *LOADI i* (2/2)



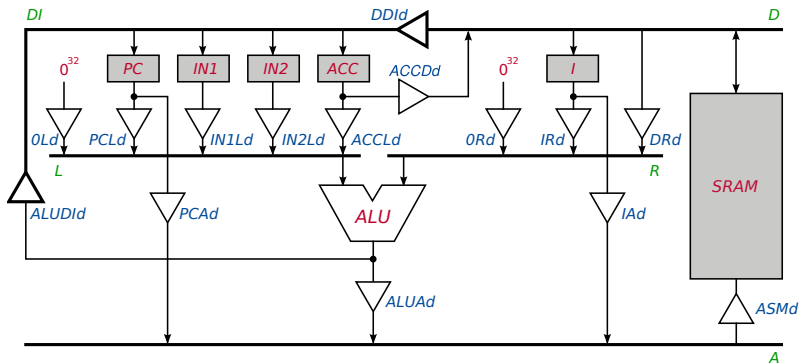
Datenpfade: *STORE i* (1/2)



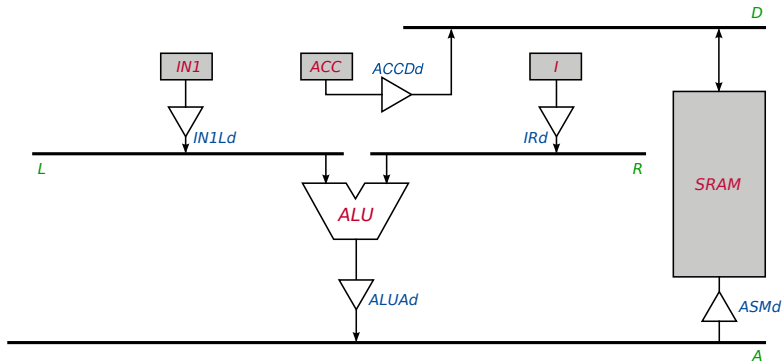
Datenpfade: *STORE i* (2/2)



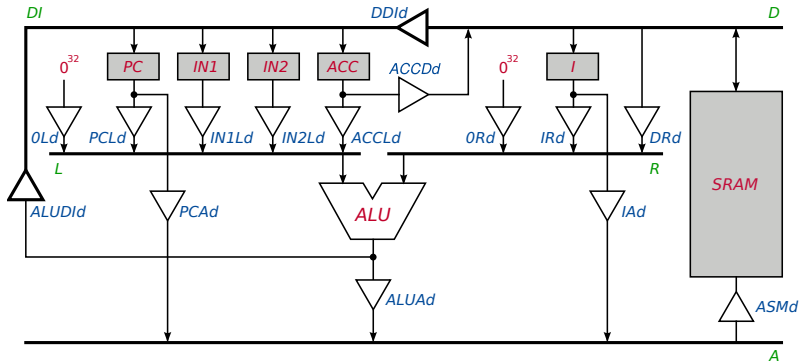
Datenpfade: *STOREIN1 i* (1/2)



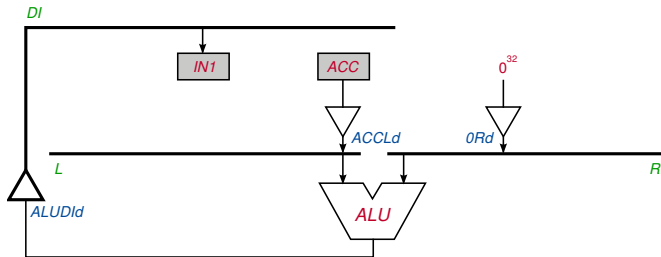
Datenpfade: *STOREIN1 i* (2/2)



Datenpfade: *MOVE ACC IN1* (1/2)



Datenpfade: *MOVE ACC IN1* (2/2)



Zusätzliche Befehle

- Man kann weitere Befehle ohne zusätzliche Hardware realisieren!
- Load- und Compute-Befehle mit beliebigem Zielregister
 $r \in \{PC, IN1, IN2, ACC\}$.
 - Kein $\langle PC \rangle := \langle PC \rangle + 1$, wenn $r = PC$.
- Befehlsformat: *LOAD* r i ; *ADD* r i ; etc.
- Befehlskodierung:

Load	31	30	29	28	27	26	25	24	23	...	0
	0	1	M		*		D		i		

Compute	31	30	29	28	27	26	25	24	23	...	0
	0	0	MI		F		D		i		

Zur Illustration: ReTI-Simulator „Neumi”

- <http://abs.informatik.uni-freiburg.de/teaching/Neumi/>
 - Simulator der ReTI-Maschine.
 - Anleitungen und zwei Beispielprogramme.
- Für die Ausführung wird Java benötigt.
 - Für Windows: <http://www.java.com/de/download/manual.jsp>
 - Für Linux: Nicht den GNU-, sondern den SUN-Compiler verwenden (Neumi funktioniert nicht mit GCJ).
 - Für Solaris und MacOS sollte die richtige Java-Version vorliegen.
- Syntax geringfügig gegenüber Vorlesung abgewandelt.
 - Kommas zwischen Operanden: „**ADDI ACC, 1**” statt „**ADDI ACC 1**”.
 - Jump-Befehle anders geschrieben: „**JUMP ge, 2**” statt „**JUMP_≥ 2**”.
 - Details: Anleitungen auf der Webseite.

- Im Buch von Keller/Paul wird ReTI (bzw. ReSa) mit sogenannten **diskreten FAST-Bausteinen** realisiert.
 - Register, Zähler, ALU, Treiber, Speicher: Bausteine aus der FAST-Bibliothek, teilweise mehrere Bausteine, um **Bitbreite 32** zu erreichen.
 - Kontrollsignale werden durch sog. **PALs** realisiert
 - **Programmable Array Logic** (Bausteine von AMD).
 - Spezielle Beschreibungssprache PALASM.
 - PALs werden heute nur noch selten verwendet.

- Im Gegensatz dazu verwenden wir State-of-the-Art-Bausteine der **NanGate-Bibliothek** (<http://www.si2.org/openeda.si2.org/projects/nangatelib>) im Hinblick auf eine VLSI-Implementierung auf einem einzigen Chip.
- Auf Basis einer solchen Implementierung behandeln wir später wesentliche Konzepte für eine „**Timing-Analyse**“. Wir beginnen zunächst mit der Realisierung der Kontrollsignale (Output-Enable, Clock-Enable ...).
- Kontrollsignale werden durch einen Endlichen Automaten generiert. Wir **skizzieren** hier das Vorgehen.

Zu generierende Kontrollsignale

- **Clock-Enable-Signale** für alle Register r , Bez.: $r\text{cken}$.
- **Output enable Signale** (active low) für alle Treiber XYd , Bez.: $/XY\text{doe}$.
- **Funktions-Select-Signale** $f[2 : 0]$ zum Selektieren der Funktion, die von ALU ausgeführt wird.
- Signale $/PC\text{clear}$, $/PC\text{load}$ für PC .
- sext zur Berechnung der Füllbits bei 24-Bit-Immediate-Konstanten.
- Für den Speicher benötigen wir die **Kontrollsignale** (active low) $/SMD\text{doe}$, $/SMw$.

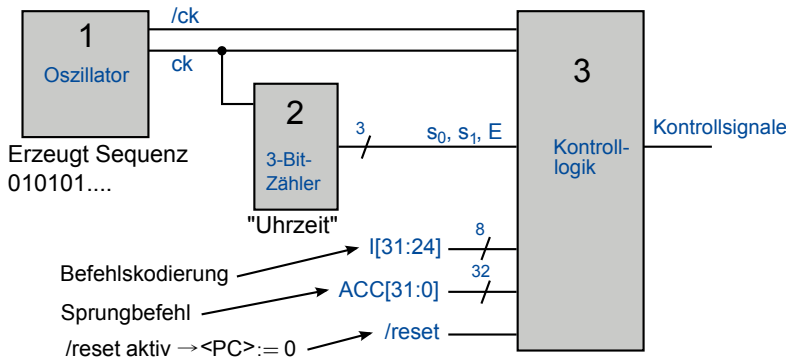
- Grobe Ablaufplanung mit **idealisierten Timing-Diagrammen**.
- Vereinfachende Annahme: Verzögerungszeit aller Bausteine = 0 (exakte Analyse mit Verzögerungszeiten später).
- Befehlsabarbeitung ist unterteilt in **Takte** (= Folge von Taktsignalen *high, low*).
- Fragen:
 - Wie sollen Kontrollsignale zusammenspielen?
 - In welchem Takt sollen welche Treiber aktiviert, welche Registerclock enabled werden?

Befehlsabarbeitung in Takten

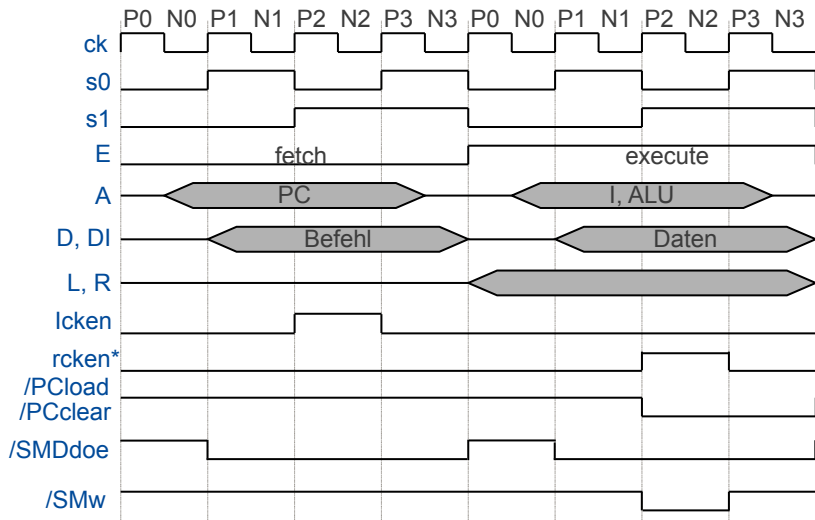
- Sowohl Fetch- als auch Execute-Phase bestehen aus 4 Takten gleicher Länge.
- Kontrollsignal:
 - $E = 0$: Fetch-Phase,
 - $E = 1$: Execute-Phase.
- Signale s_0, s_1 = Binärkodierung der Nummer des Taktes (innerhalb Fetch, Execute), in dem ReTI sich befindet.
- Steigende Flanken (Anfang des Taktes) werden mit P_i , fallende (Mitte des Taktes) mit N_i bezeichnet ($i = 0, \dots, 3$).
- Clock ck , Signale s_0, s_1 und E werden Phasensignale genannt. Weitere (Kontroll-)Signale werden aus den Phasensignalen erzeugt.

Erzeugung der Phasensignale

3-Bit-Zähler, zählt bei jeder positiven Flanke von **ck** um 1 hoch.

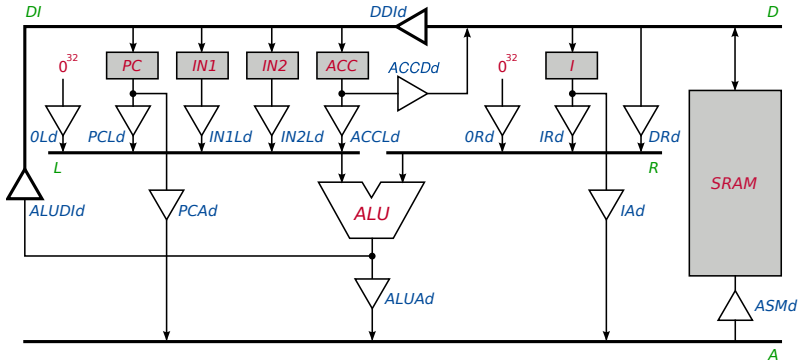


Idealisiertes Timing-Diagramm



(* r = PC, IN1, IN2, ACC)

Zur Erinnerung: Datenpfade

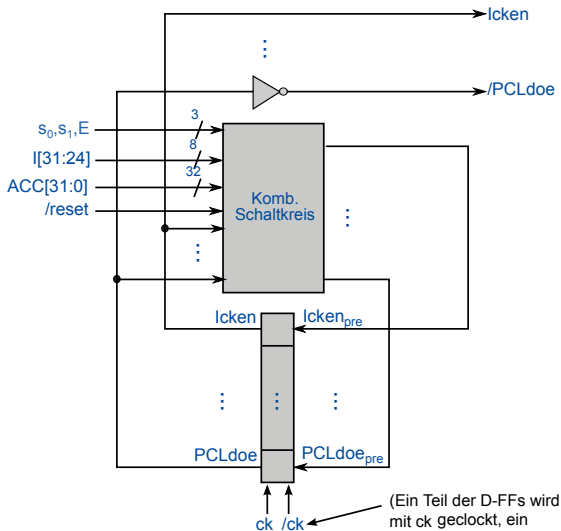


- Nutze **Busse möglichst lange** (unter Vermeidung von **Bus Contention**).
- **Clock-Enable-Signale** der Register **möglichst spät**
→ viel Zeit für Berechnung neuer Daten.
- Nach dem **Entwurfsende** muss das Timing der *CPU* mit den konkreten Werten der eingesetzten Fertigungstechnologie überprüft werden. (“Wie schnell kann man maximal takten, um korrektes Funktionieren zu garantieren?”)
 - Siehe nächstes Kapitel.

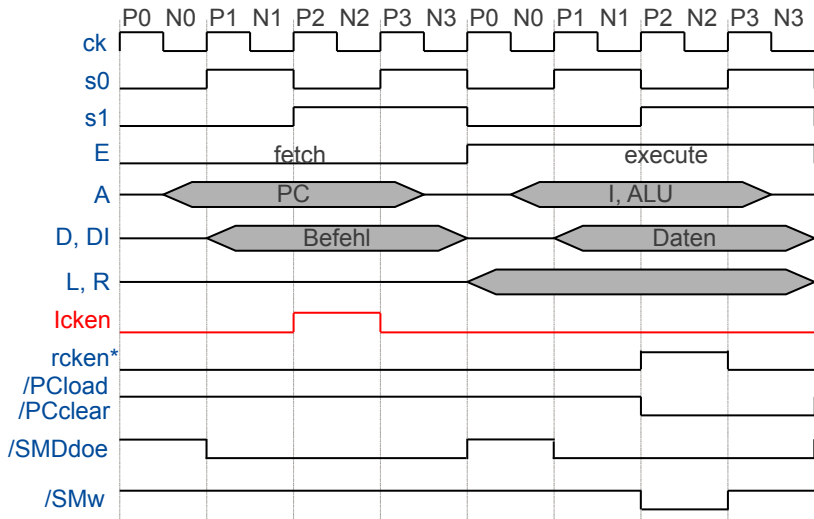
Aufbau der Kontrolllogik (1/2)

- Die eigentliche Kontrolllogik wird realisiert durch einen **Endlichen Automaten**.
- Die Kontrollsignale sind als Ausgangssignale des Endlichen Automaten implementiert.
- Ist ein Kontrollsignal **active low**, dann bezeichnen wir es z.B. mit $/x$. Das Ausgangssignal $/x$ ergibt sich dann durch Negation des Ausgangssignals x eines entsprechenden FFs mit Eingangssignal x_{pre} .
- Ist ein Kontrollsignal **active high**, dann bezeichnen wir es z.B. mit x . Das Ausgangssignal x entspricht dem Ausgangssignal eines FFs mit Eingangssignal x_{pre} .

Aufbau der Kontrolllogik (2/2)

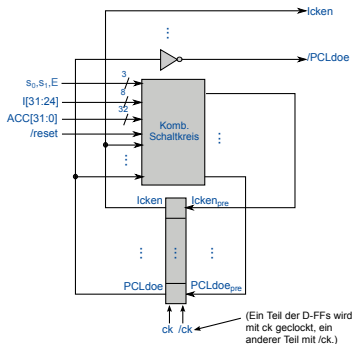
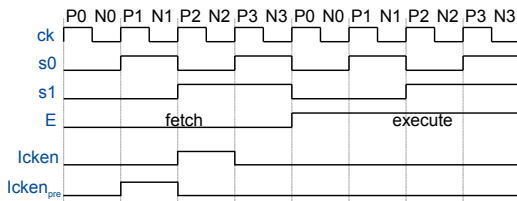


Berechnung von *lcken* als erstes Beispiel (1/2)



(* r = PC, IN1, IN2, ACC)

Berechnung von *lcken* als erstes Beispiel (2/2)

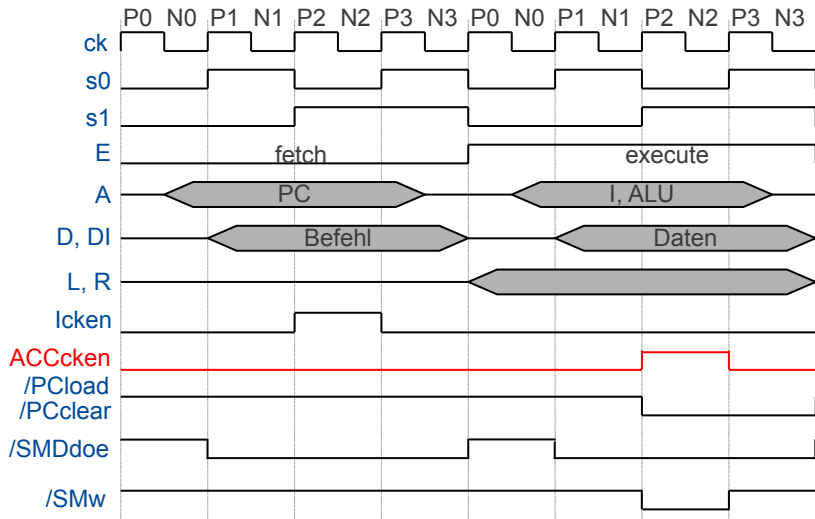


- *lcken_pre* hat steigende Flanke bei P1, fallende bei P2 von Fetch.
- Realisierung: $lck_{pre} = \overline{E} \cdot \overline{s_1} \cdot s_0$.

Berechnung von *ACCcken*, *IN1cken*, *IN2cken*, *PCcken*

- Analog, unter Berücksichtigung der Tatsache, dass Register nur bei bestimmten Befehlen neu beschrieben werden dürfen

Berechnung von *ACCcken* als Beispiel (1/3)



Berechnung von *ACC*cken als Beispiel (2/3)

- ACC_{pre} hat steigende Flanke bei $P1$, fallende bei $P2$ von Execute.
- Aber nur bei folgenden Befehlen:
 - Compute mit $D = ACC$
 - Load mit $D = ACC$
 - Move mit $D = ACC$

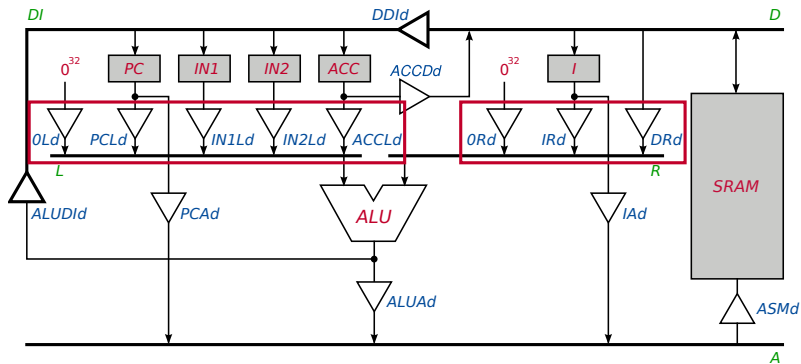
- Compute: $\overline{l_{31}} \cdot \overline{l_{30}}$
- Load: $\overline{l_{31}} \cdot l_{30}$
- Move: $l_{31} \cdot \overline{l_{30}} \cdot l_{29} \cdot l_{28}$
- $D = ACC$: $l_{25} \cdot l_{24}$

Kodierung S, D	
S, D	Register
0 0	PC
0 1	IN1
1 0	IN2
1 1	ACC

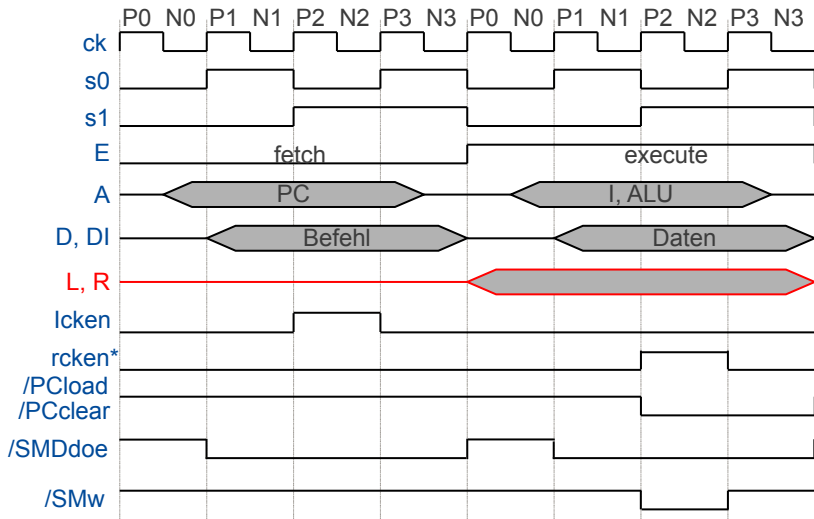
Berechnung von *ACCcken* als Beispiel (3/3)

$$\begin{aligned} ACCcken_{pre} = & E \cdot \overline{s_1} \cdot s_0 \cdot & // \text{ P1 von execute} \\ & l_{25} \cdot l_{24} \cdot & // D = ACC \\ & \left(\overline{l_{31}} \cdot \overline{l_{30}} + \overline{l_{31}} \cdot l_{30} + l_{31} \cdot \overline{l_{30}} \cdot l_{29} \cdot l_{28} \right) & // \text{ Compute, Load oder Move} \end{aligned}$$

Datenpfade und Treiber auf L und R



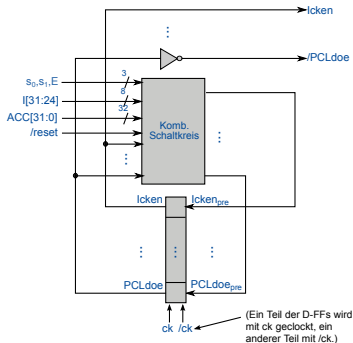
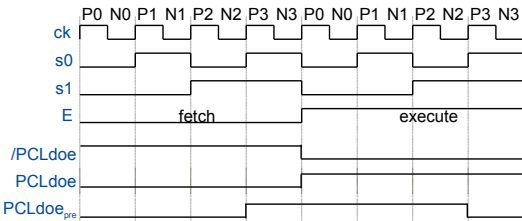
Idealisiertes Timingdiagramm



(* r = PC, IN1, IN2, ACC)

- $0Ld, PCLd, IN1Ld, IN2Ld, ACCLd, 0Rd, IRd, DRd$.
- Enabled in der ganzen Execute-Phase.
- Beispiel für Realisierung: $/PCLdoe$.
 - Enabled für
 - JUMP ($/[31 : 30] = 11$)
 - Compute-Befehle ($/[31 : 30] = 00$) mit $D = PC$ ($/[25 : 24] = 00$)
 - MOVE ($/[31 : 28] = 1011$) mit $S = PC$ ($/[27 : 26] = 00$).

Berechnung von $/PCLdoe$ (1/2)



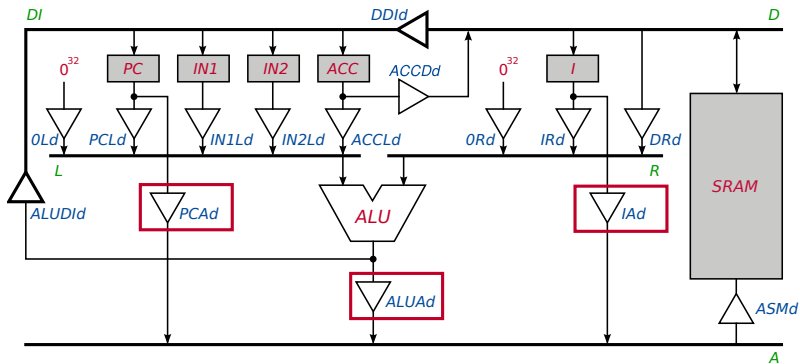
- $PCLdoe_{pre}$ hat steigende Flanke bei P3 von Fetch.

Berechnung von $PCLdoe$ (2/2)

$$\begin{aligned} PCLdoe_{pre} = & \left(\overline{E} \cdot s_1 \cdot s_0 \cdot \right. & // \text{ P3 von fetch} \\ & [l_{31} \cdot l_{30} + & // \text{ JUMP} \\ & \overline{l_{31}} \cdot \overline{l_{30}} \cdot \overline{l_{25}} \cdot \overline{l_{24}} & // \text{ Compute mit } D = PC \\ & l_{31} \cdot \overline{l_{30}} \cdot l_{29} \cdot l_{28} \cdot \overline{l_{27}} \cdot \overline{l_{26}}] \left. \right) & // \text{ MOVE mit } S = PC \\ & + PCLdoe \cdot E \cdot \overline{s_1} \cdot \overline{s_0} & // \text{ Halten in Takt 0 von execute} \\ & + PCLdoe \cdot E \cdot \overline{s_1} \cdot s_0 & // \text{ Halten in Takt 1 von execute} \\ & + PCLdoe \cdot E \cdot s_1 \cdot \overline{s_0} & // \text{ Halten in Takt 2 von execute} \end{aligned}$$

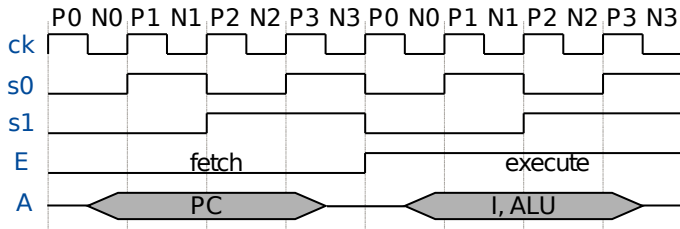
- Output-Enable-Signale für andere Treiber auf L und R analog.

Datenpfade und Treiber auf Adressbus

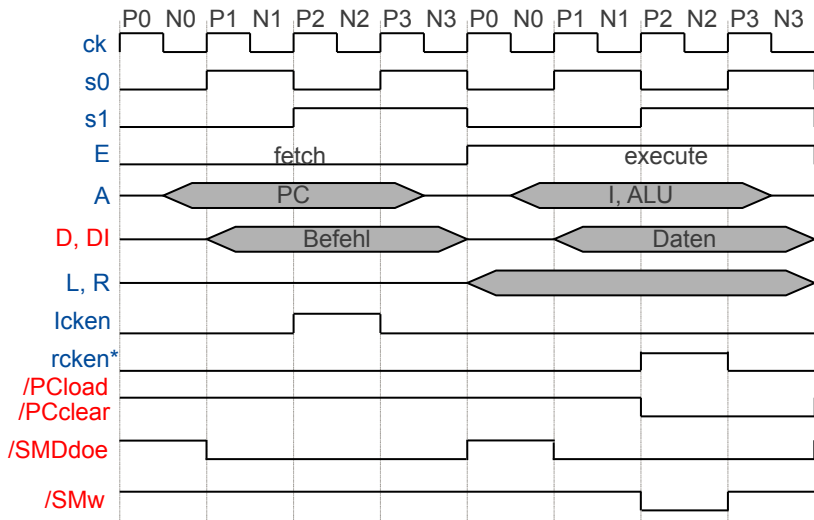


Treiber auf Adressbus

- *PCAd*: enabled (unabhängig vom Befehl) bei *N0* der Fetch-Phase, disabled bei *N3* der Fetch-Phase.
- *IAd*, *ALUAd*: enabled bei *N0*, disabled bei *N3* von Execute (aber nicht bei allen Befehlen).
- Die D-FFs zu Output-Enable-Signalen auf dem Adressbus werden mit der invertierten Clock getaktet (Verschiebung um einen halben Takt!)



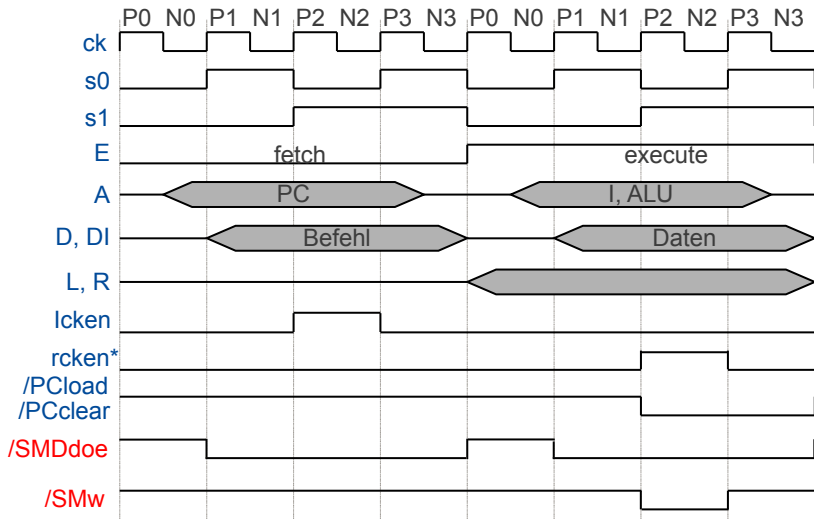
Idealisiertes Timingdiagramm



(* r = PC, IN1, IN2, ACC)

- Treiber auf *D*, *DI*
- Signale */PCload*, */PCclear*
- Speicheransteuerung: s. nächste Folie.
- Kontrolle der ALU und Sign-Extension:
 - Funktions-Select-Signale *f[2:0]* der ALU
 - Eingangsübertrag *C_{in}*
 - *sext* und *fill*
 - All diese Signale werden durch den kombinatorischen Schaltkreis (ohne zusätzliche FFs) berechnet.

Idealisiertes Timing-Diagramm



(* r = PC, IN1, IN2, ACC)

- Output-Enable / *SMD_{doe}* für SMDd (Treiber am Speicherausgang) aktiviert von P1 bis P0 bei Leseoperationen, d.h. bei
 - Fetch
 - Compute Memory
 - LOAD, LOADINj
- Schreibsignal für Speicher / *SM_w* (memory write) aktiviert von P2 bis P3 von execute bei Schreiboperationen, d.h. bei
 - STORE, STOREINj

- Sequentielle Schaltkreise bestehen aus **speichernden Elementen** (Latches, Flipflops) und einem **kombinatorischen Kern**.
- Sie implementieren **endliche Zustandsautomaten**.
- Der Entwurf eines sequentiellen Schaltkreises besteht aus der Aufstellung des **Zustandsdiagramms**, der **Zustandsminimierung**, der **Zustandskodierung** und der **Synthese** der kombinatorischen Logik.
- Nun war es uns möglich, den **Entwurf von ReTi** zu vervollständigen (exakte Timing-Analyse folgt).