3.5 Realisierung eines einfachen Prozessors

Zur Vorlesung Rechenanlagen

SS 2019

Wir wollen in diesem Abschnitt alles, was wir bisher gelernt haben, zusammenführen, und unsere eingangs betrachtete, abstrakte Maschine nun realisieren.

Dazu lassen wir zunächst nochmal alle Befehle Revue passieren, um Änderungen oder Ergänzungen an unserem anfänglichen Konzept anzusprechen.

Ferner werden wir uns das Leben nicht zu schwer machen, und uns zunächst auf ein Minimum an notwendigen Befehlen beschränken.

Die Arithmetik

Wir haben Arithmetik auf ganz verschiedenen Objekten. Der Einfachheit halber wollen wir Gleitkommabefehle ganz ausschliessen. Allerdings sollen sowohl unsigned als auch signed Operationen auf Worten möglich sein. Alle Operanden beziehen sich dabei auf GP-Register der Registerbank. Befehle für Teilworte klammern wir aus. Für

$$\texttt{OP} \in \{\, \texttt{ADD,SUB,MUL,DIV,SXX} \,\}$$

erhalten wir bei einer Zahlendarstellung $d \in \{\, bv, b_1, b_2 \,\}$, für die man sich entscheiden muss, folgende Befehle:

OP Ri,Rj,Rk; **OPU** Ri,Rj,Rk;

OPI Ri,Rj,#k; **OPUI** Ri,Rj,#k;

Kein Suffix besagt, dass es sich um eine Operation auf einem Maschinenwort unter Darstellung *d* handelt. Das Suffix U steht für unsigned, I für immediate.

Arithmetik ff

Beispiel: **`ADDUI Ri,Rj,#k`** bezeichnet demnach eine Unsigned Addition mit (unsigned!) immediate Operand *k*.

Die Wirkung sei stets die folgende:

$$reg[i] = u_n^{-1}((u_n(reg[j]) \; op \; u_n(reg[k])) mod \, 2^n)$$

$$\text{bzw.} \quad reg[i] = u_n^{-1}((u_n(reg[j]) \; op \; k) mod \, 2^n)$$

für **`OPU Ri,Rj,Rk;`** bzw. **`OPUI Ri,Rj,#k;`** Befehle, wobei zusätzlich ein möglicher Überlauf angezeigt wird.

Wir lassen bei unserem einfachen Beispielprozessor Befehle, die sich auf Unsigned Zahlen beziehen, der Einfachheit halber weg.

Arithmetik ff

Für **`OP Ri,Rj,Rk;`** bzw. **`OPI Ri,Rj,#k;`** Befehle sei die Wirkung folgende:

$$reg[i] = d_n^{-1}(d_n(reg[j])\ op\ d_n(reg[k]))$$

$$\text{bzw.}\quad reg[i] = d_n^{-1}(d_n(reg[j])\ op\ k)$$

wobei bei Überlauf, dessen Anzeige ebenfalls erfolgt, der Inhalt vom Zielregister nicht näher festgelegt ist.

In unserem einfachen Beispielprozessor lassen wir Befehle für Multiplikation und Division der Einfachheit wegen weg, und gehen davon aus, dass sie durch Unterprogramme realisiert sind.

Verzweigungen

Verzweigungen haben von nun an die Form

```
BEQZ Ri,#j; BNEZ Ri,#j;
```

und als neue Befehle

```
BOV #j; BNOV #j;
```

wobei **`BOV, BNOV`** Verzweigungen nach dem Overflow Flag sind. Damit können wir einen Überlauf zumindest explizit abfangen. Wir werden später sehen, dass es dazu bessere Techniken gibt (vgl. Ausnahmebehandlungen).

Transporte

Als Transportbefehle hatten wir

`LOAD Ri,Rj,#k; STORE Ri,Rj,#k;`

`i,j` sind Registeradressen, und **`k`** ein immediate Operand. Die Wirkung ist:

`LOAD Ri,Rj,#k:` *reg[i] := mem[u_n(reg[j]) + k]*

`STORE Ri,Rj,#k:` *mem[u_n(reg[j]) + k] := reg[i]*

Transporte von Halbworten oder Bytes klammern wir aus. Wir benutzen einen wortorientierten Speicher!

Sprünge

Wir hatten **`JMP #k; JREG Ri;`** und **`JAL Ri,#k;`**
Diese Befehle werden wir unverändert übernehmen.

Nun können wir beginnen, die Befehle zu kodieren:

Kodierung der Befehle

Ziele:

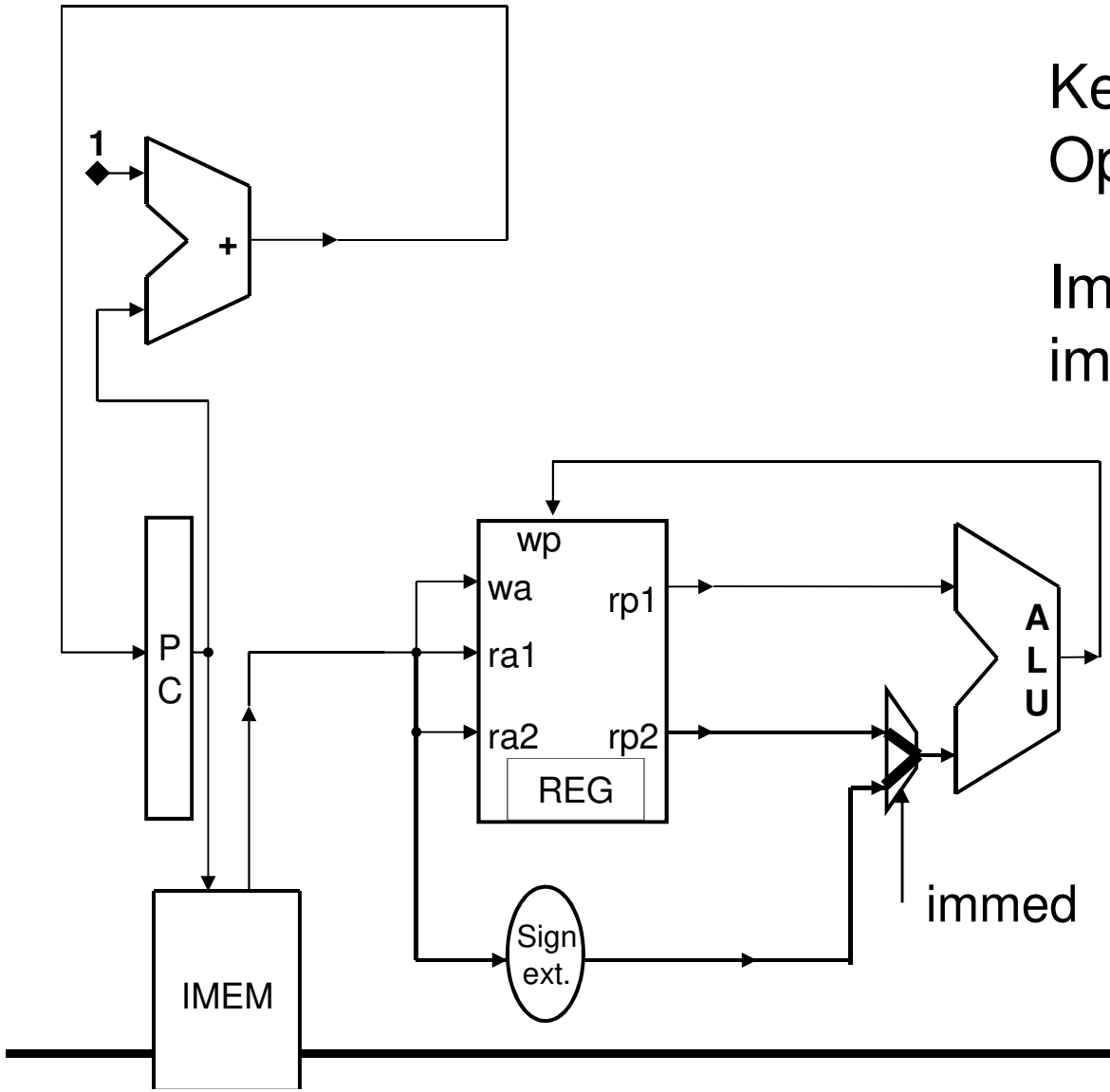
- Kodiere Instruktionen in (ggf. Vielfachen von) Maschinenworten
- Gestalte die Kodierung leicht dekodierbar.

Der Datenfluss im Prozessor hängt sehr stark davon ab, ob es sich um

- **Arithmetik auf Registern (A)**
- **Verzweigungen (B)**
- **Transporte (T)**
- **Sprünge (J)**

handelt, ist aber bei Befehlen einer dieser Gruppen immer sehr ähnlich:

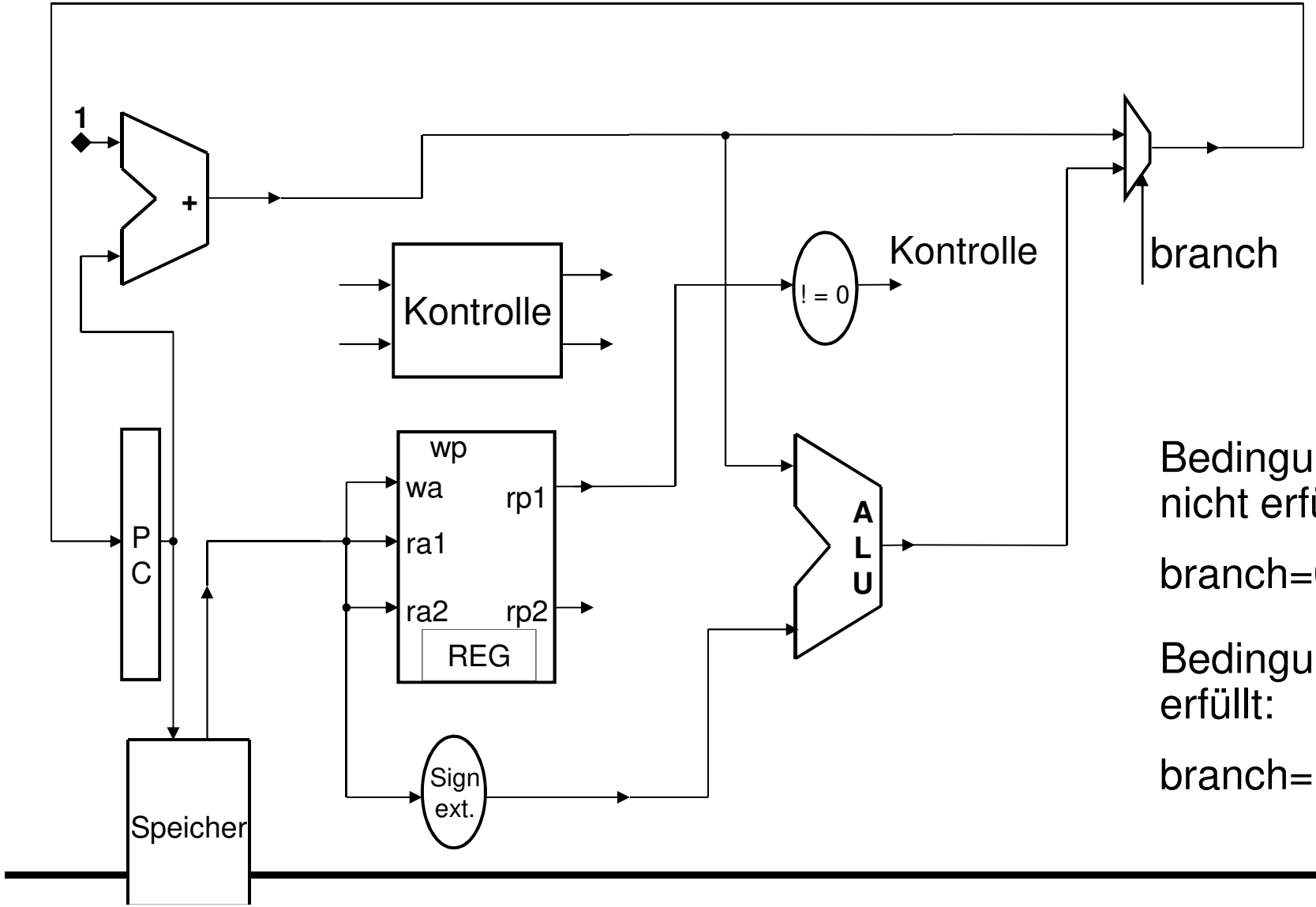
Beispiel: Datenfluss Arithmetik



Kein Immediate Operand: immed = 0

Immediate Operand: immed = 1

Beispiel: Datenfluss Verzweigung



Beispiel: Datenfluss Transporte

Laden **`LOAD Ri,Rj,#k`**

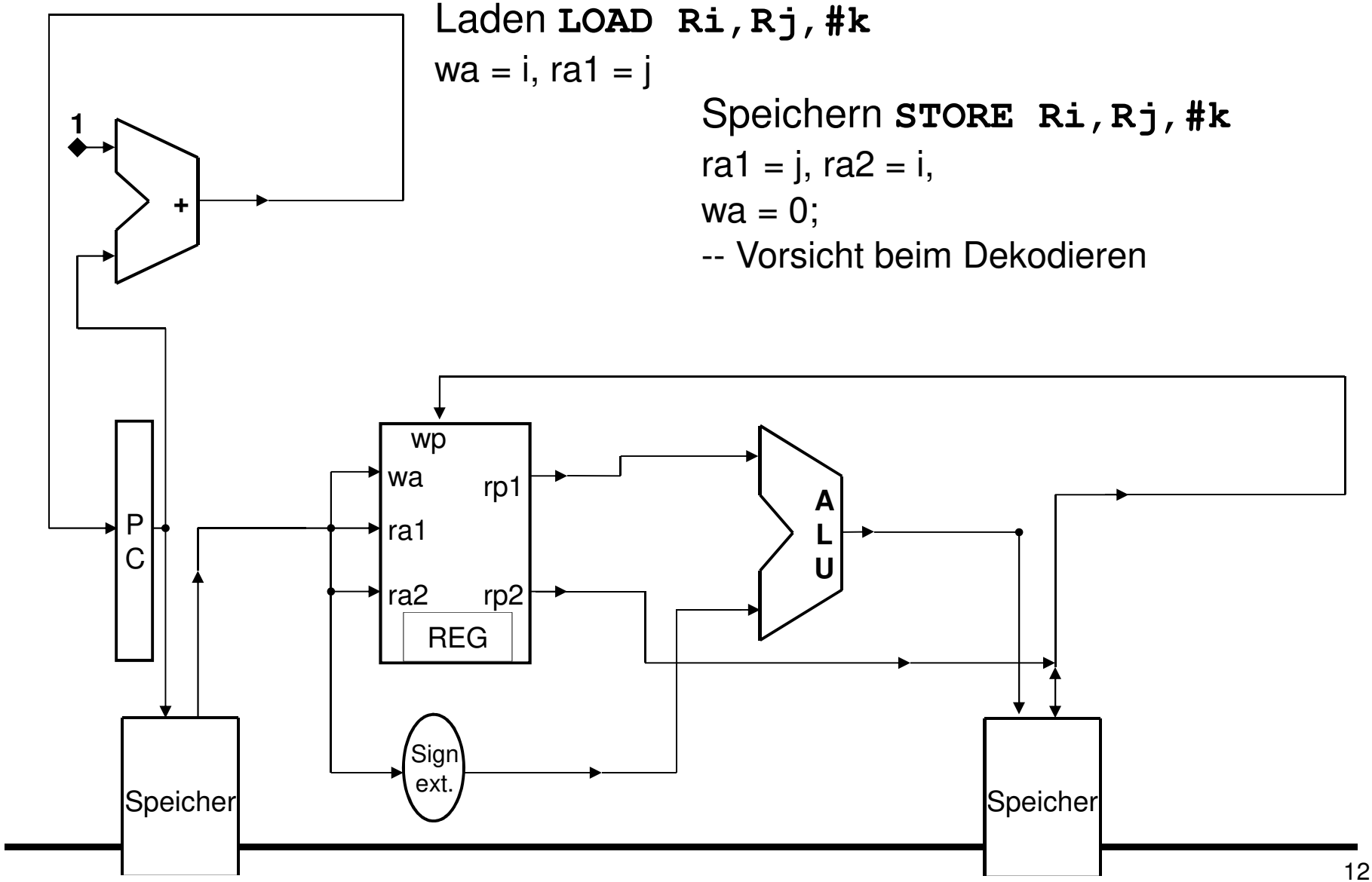
wa = i, ra1 = j

Speichern **`STORE Ri,Rj,#k`**

ra1 = j, ra2 = i,

wa = 0;

-- Vorsicht beim Dekodieren



Instruktionsklassen

Die Idee besteht nun darin, alle diese Datenflüsse „übereinander“ zu legen und von der Kontrolle, die den Befehl dekodiert, Kontrollsignale zu berechnen, die durch Multiplexer den korrekten Datenfluss für den jeweiligen Befehl steuern.

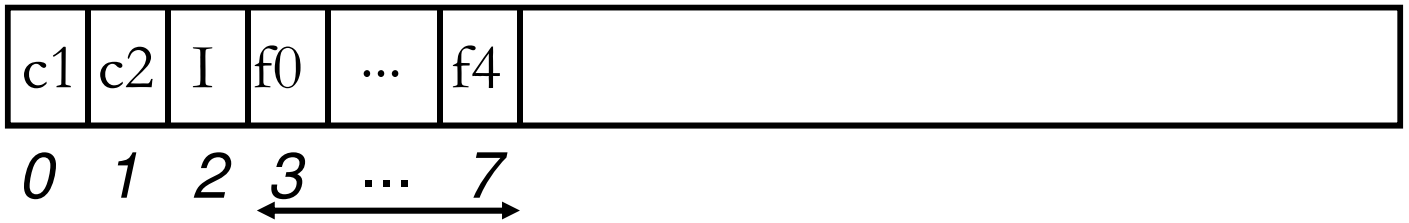
Da sich die Datenflüsse ganz grundsätzlich unterscheiden, ist es sinnvoll in einem Feld des Befehlswortes zunächst Klassen mit sehr ähnlichem Datenfluss zu definieren:

Instruktionsklassen:

(c1c2): (00) - Arithmetik; (01) - Verzweigung; I: Immediate Bit
(10) - Transport; (11) - Sprung;

Function Codes

Da es nun immer noch mehrere Befehle in einer Klasse gibt, sollte man als nächstes die genaue Funktion des Befehls im Maschinenwort kodieren. Je nach Klasse kann dieses Feld unterschiedliche Länge haben - wir haben etwa sehr viel mehr Arithmetikbefehle als Sprünge. Bei unserer Maschine sollten dazu 5 Bit ausreichen.



Beispiele: Wir wollen die Kodes hier nicht einzeln angeben. Folgende Beispiele wären denkbar:

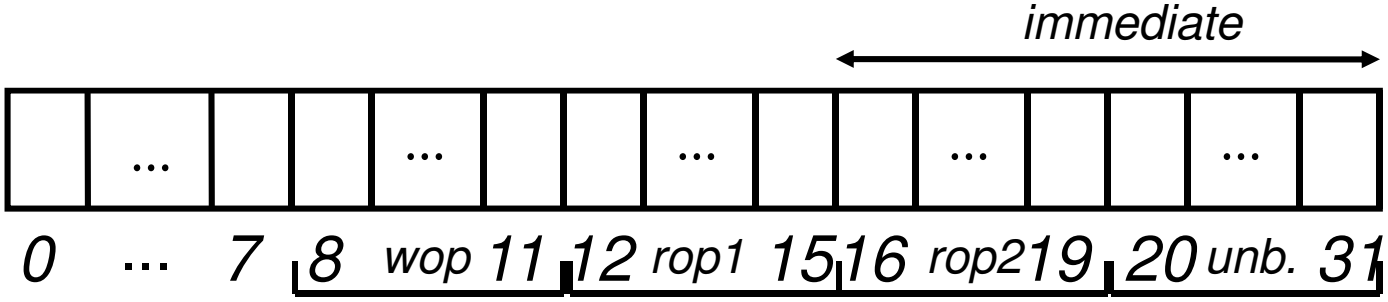
ADDI	00 1 00000	**ADD**	00 0 00000
BEQZ	01 1 00010	**STORE**	10 1 00001

Registeradressen

Wir haben nun ein Byte verheizt, könnten also bei 3 Adressbefehlen noch jeweils ein Byte für Registeradressen vorsehen, d.h. unsere Maschine hätte das Potential für bis zu 256 Register.

Problem: Bei 2 Registern und einem Immediate Operand sieht es aber schlecht aus, weil für diesen Operanden dann nur noch ein Byte bleibt, oder wir auf 64-Bit Instruktionscodes übergehen müssen.

Spendieren wir der Maschine nur bis zu 16 Register, können wir 16-Bit Immediate Operanden verkraften, was in den meisten Fällen bei Integer Arithmetik ausreicht.



Verzweigungsbefehle:

Problem: Jetzt wirds eng!

Wir haben 1 Leseoperanden und einen immediate Operanden. Selbst wenn wir nur 16 Register benutzen, bleiben nur 16-Bit für das Sprungziel, wenn der Leseoperand stets auf Bit 12…15 liegt.

Auswege:

• dekodiere den Schreiboperanden als Leseoperanden (mehr Aufwand)

• benutze längere Instruktionen (z.B. 64-Bit)

• benutze relative Sprünge in Verzweigungen

Verzweigungsbefehle ff:

Wir gehen letzteren Weg, d.h. wir definieren die Wirkung von Verzweigungen wie folgt

$$\mathbf{BXX}\ \mathbf{R}i,\#k : pc := \begin{cases} pc + k + 1 & \text{falls } reg(i)\ \mathrm{XX}\ 0 \\ pc + 1 & \text{sonst} \end{cases}$$

Nun können wir in einem Programm immerhin in einer Umgebung von ca. *[-32000:32000]* beliebig verzweigen. Das ist in den zeitfressenden Fällen ausreichend (Rücksprünge in inneren Programmschleifen).

Weite Distanzen kann man dann nur noch über Sprünge erreichen, d.h. man ersetzt für zu große (kleine) *k*:

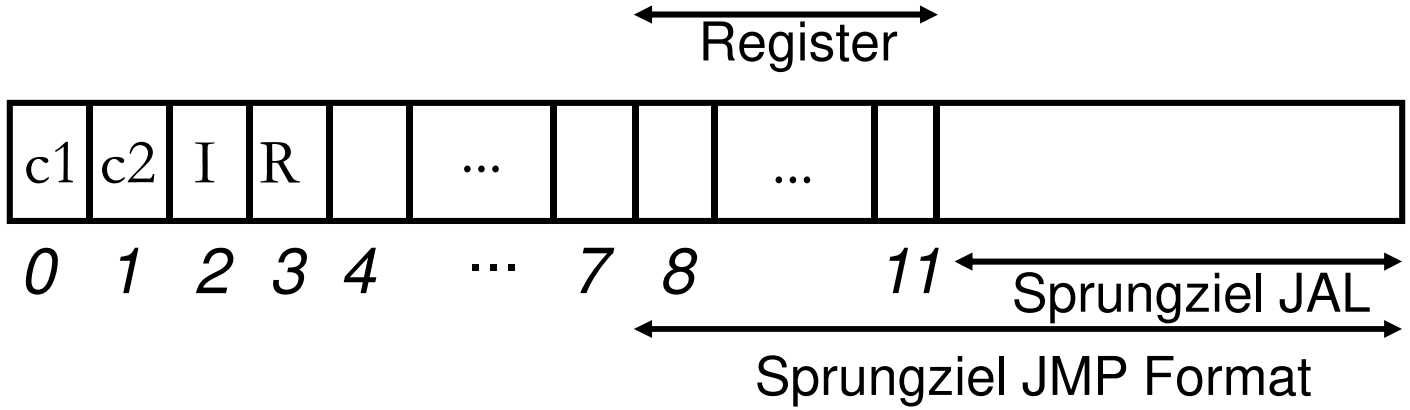
r: **`BXX Ri,#k;`** durch die Sequenz (YY = not XX)

r: **`BYY Ri,#1; JMP #k;`** -- falls k ≥ 0

r: **`BYY Ri,#1; JMP #k-1;`** -- falls k < 0

Sprünge

Hier sind sehr große immediate Operanden wichtig, um auch sehr lange Programme bearbeiten zu können. Deshalb kodieren wir diese Befehle etwas anders:



(I,R): (1,0) - **`JMP;`** (1,1) - **`JAL;`** (0,1) - **`JREG`**

Damit schaffen wir bis *24* Bit Sprungziele, können also in Programmcode von ca. *64* MB Länge (*4*16* MB) springen.

Es ist auch hier geschickt, mit relativen, statt absoluten Sprüngen zu arbeiten: Programme werden verschiebbar!

Eine simple 1 Zyklus Maschine

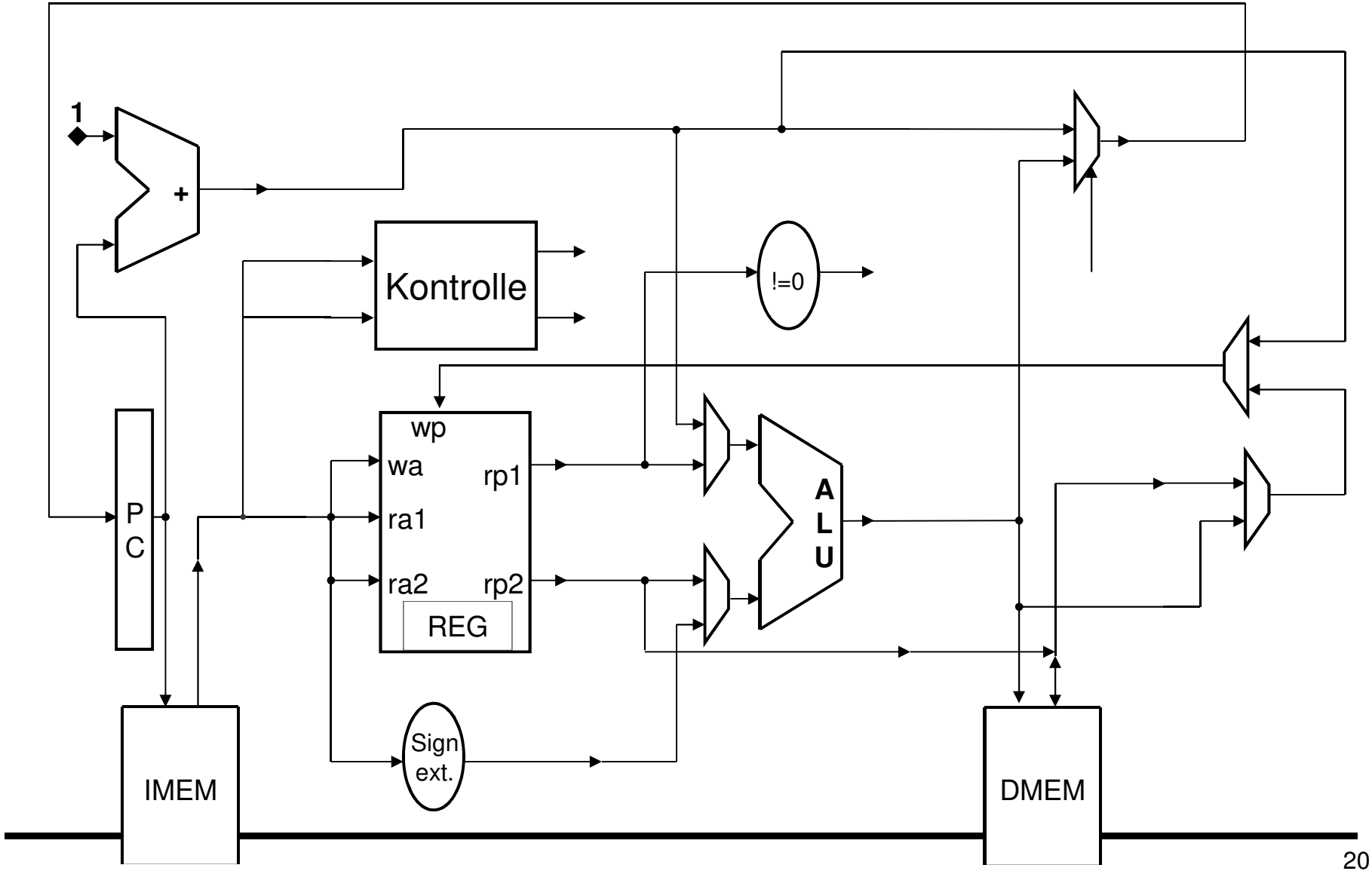
Ideen:

Lege alle Datenflüsse, wie sie bei den einzelnen Befehlstypen vorkommen, „übereinander“ und wähle in Abhängigkeit vom Befehlstyp den entsprechenden Fluss aus. Die **Kontrolle** erzeuge dazu aus dem Befehlscode

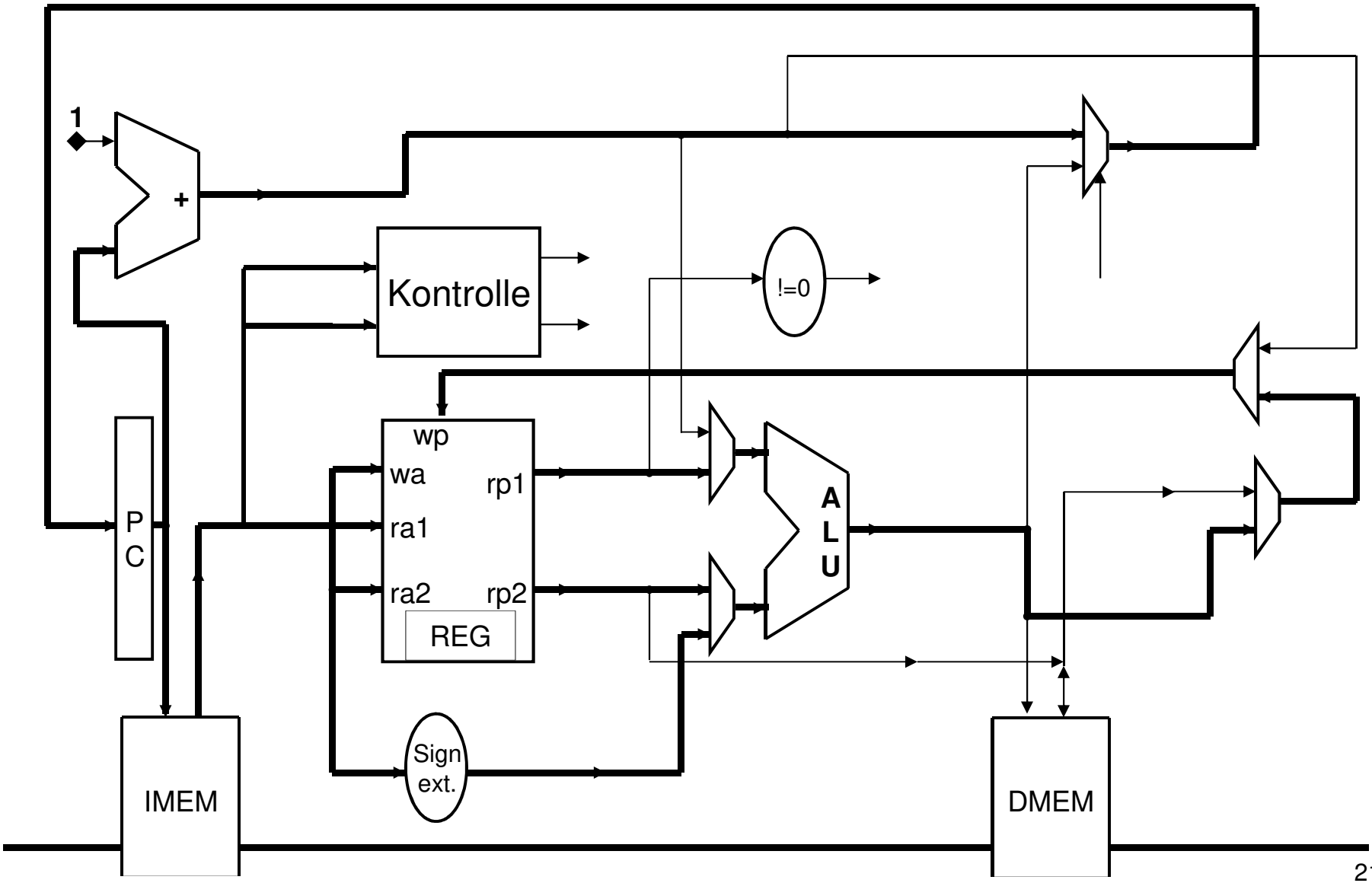
- Steuerleitungen für Multiplexer zur Kontrolle des Datenflusses
- Operationcodes, Adressen und Kontrollsignale für die beteiligten Speicher und ALUs.

Da die Maschine in jedem Zyklus des Taktes einen Befehl bearbeiten soll, benötigen wir zwei RAM Speicher, einen für Programmcode und einen für Daten. **Harvard Architektur**

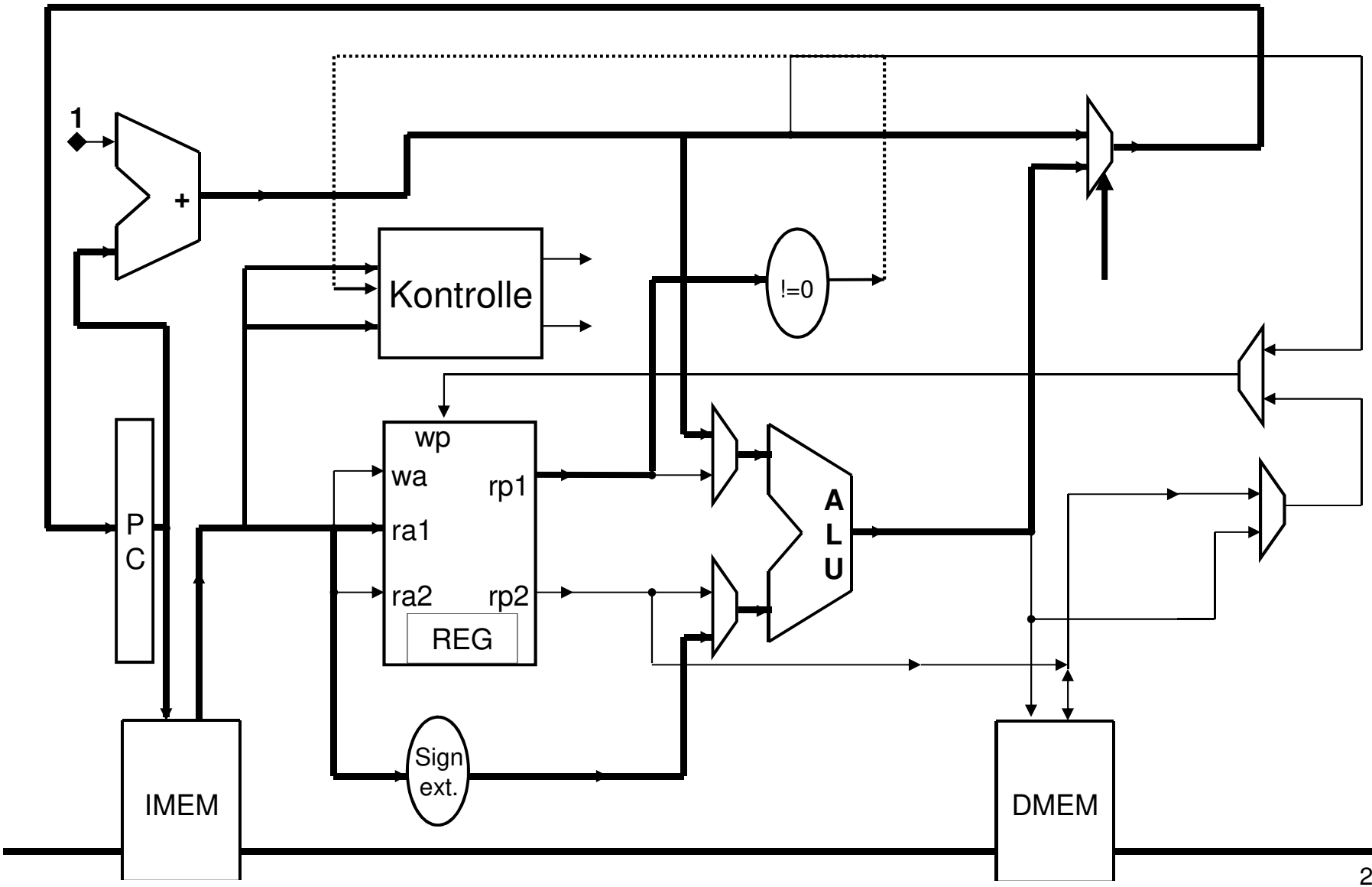
1- Zyklus WüRC



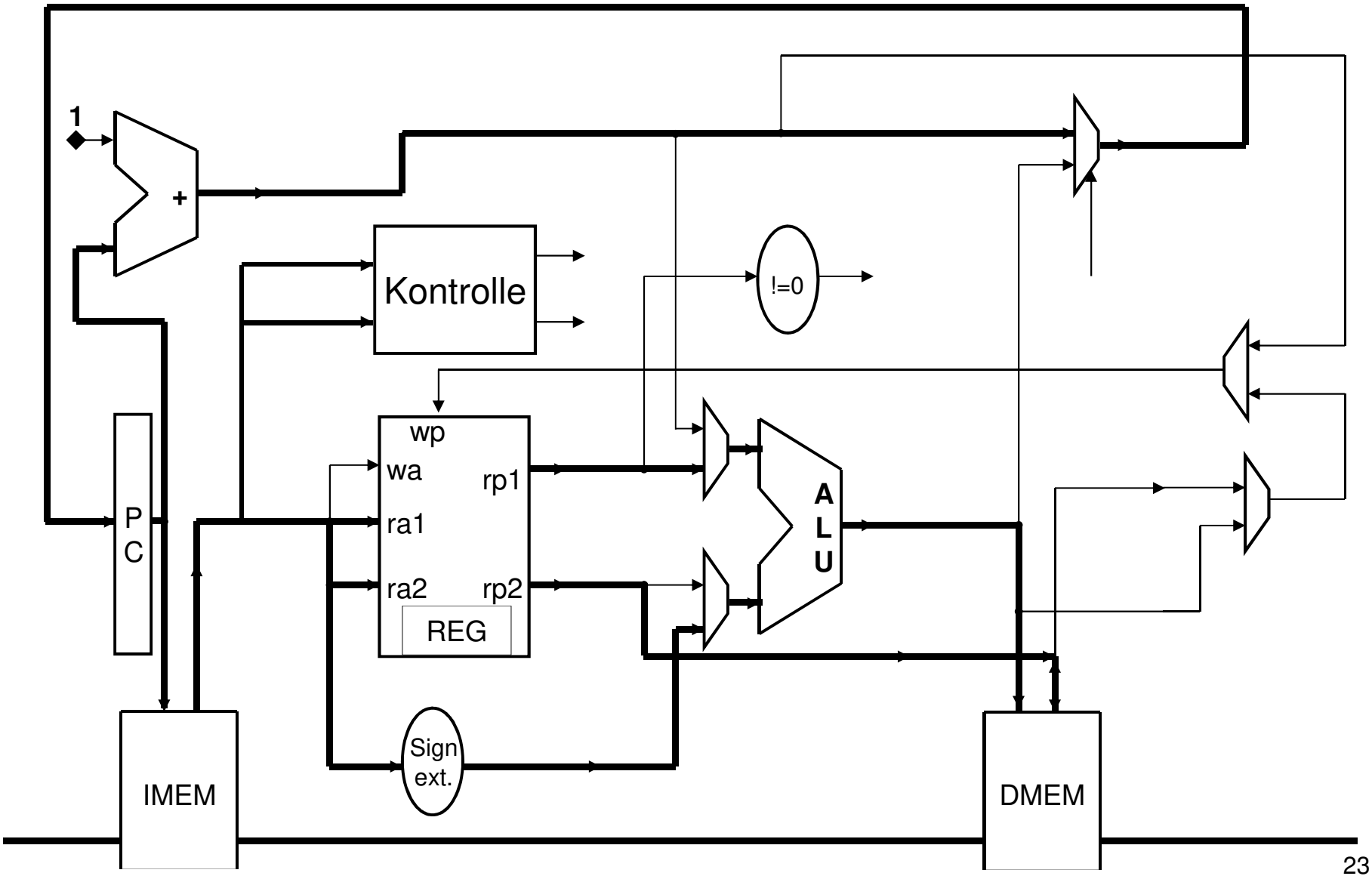
Bearbeitungsbeispiel: Arithmetik



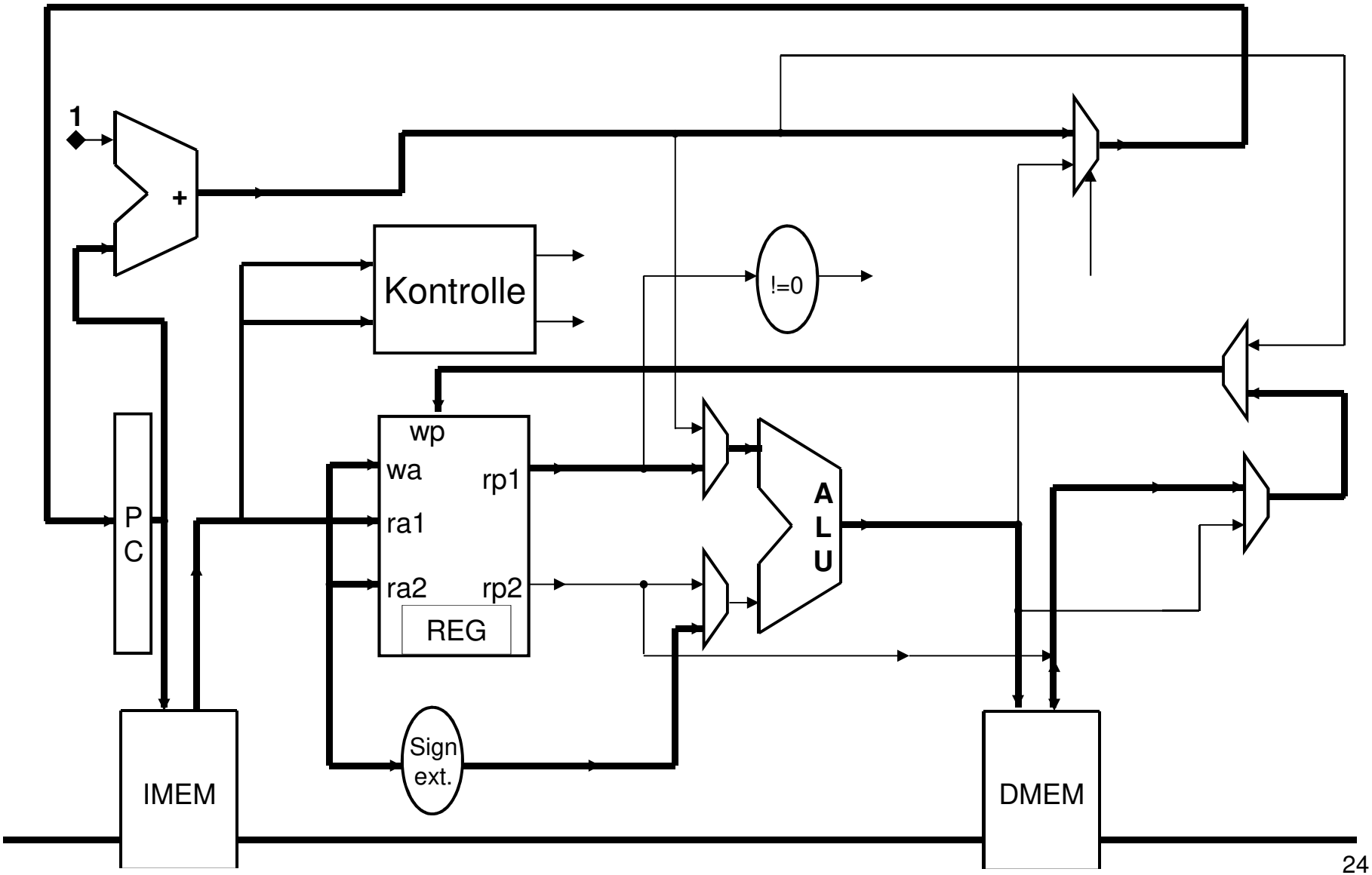
Bearbeitungsbeispiel: Verzweigung



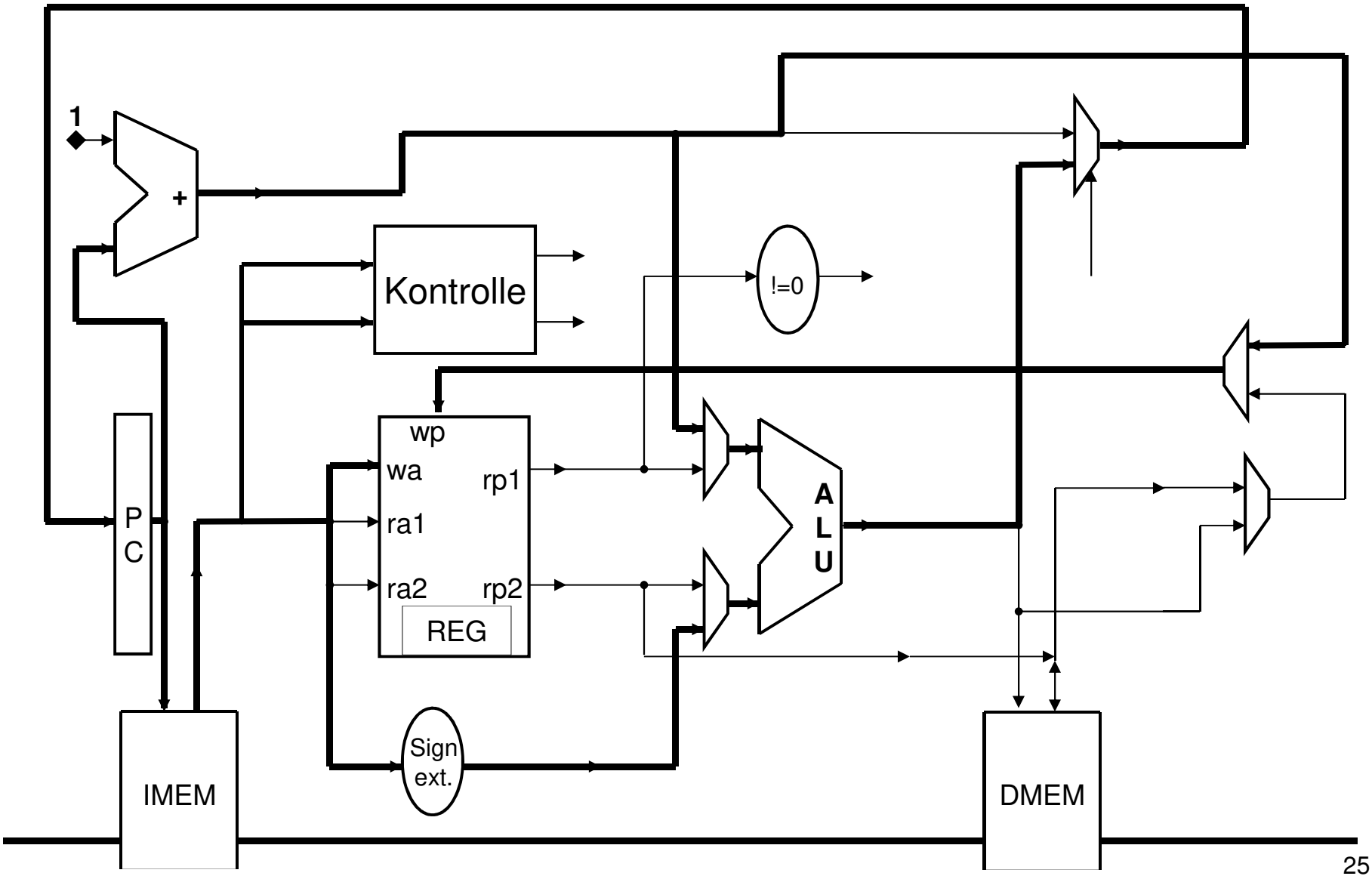
Bearbeitungsbeispiel: STORE



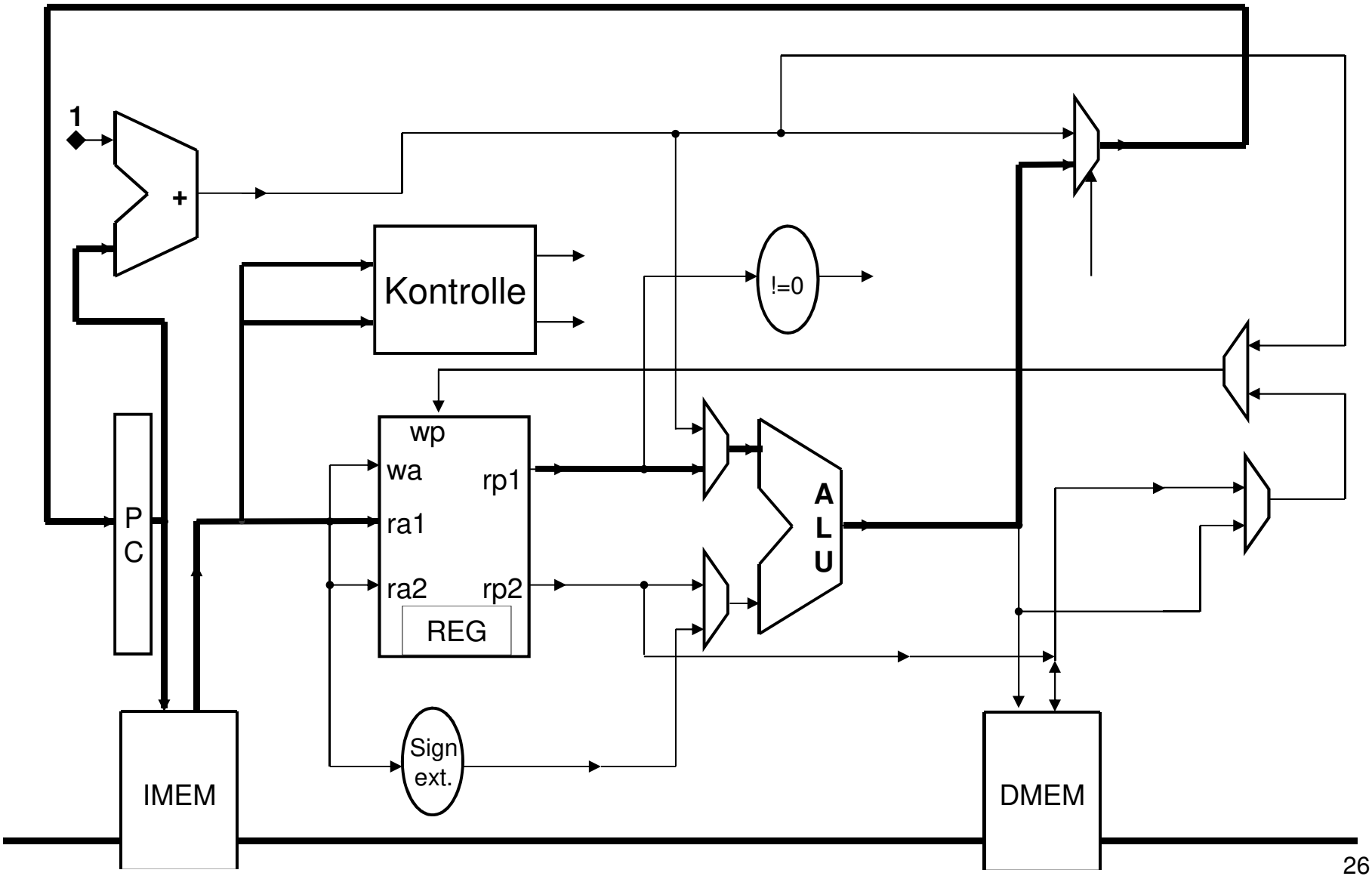
Bearbeitungsbeispiel: LOAD



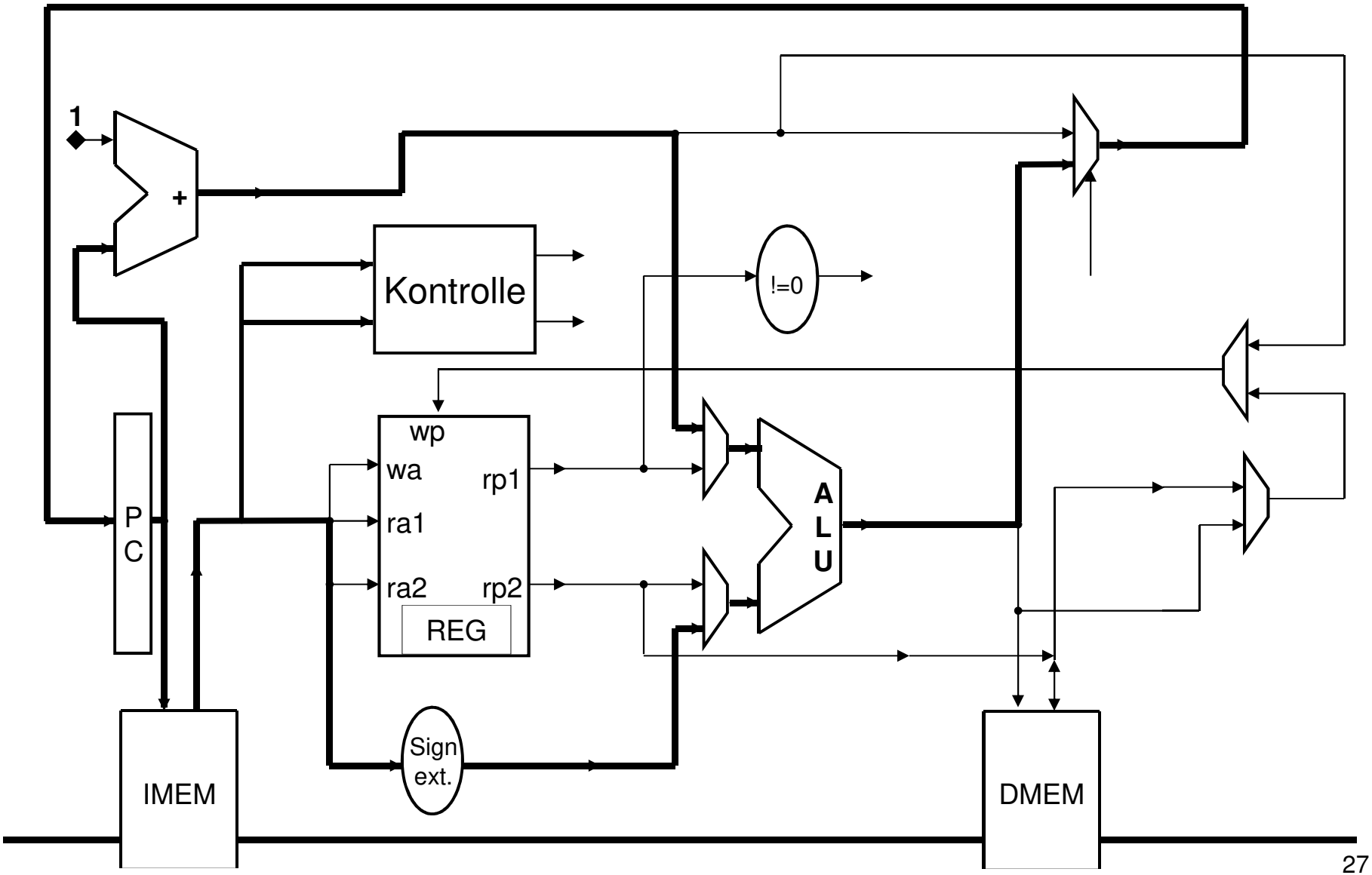
Bearbeitungsbeispiel: JAL



Bearbeitungsbeispiel: JREG



Bearbeitungsbeispiel: JMP



Kritik

Diese simple Realisierung bearbeitet pro Taktzyklus einen Befehl, die Zykluszeit ist aber mit

$$\Gamma_{clk} \geq 2 \cdot t_{access\_cache} + 2 \cdot t_{access\_reg} + t_{ALU} + t_0$$

enorm hoch. t_0 sei dabei die Laufzeit der Multiplexer und der Busse.

Dafür ist die Kontrolle sehr einfach zu bauen: sie besteht im wesentlichen nur aus Dekodierschaltungen, deren Kosten von der gewählten Kodierung der Befehle abhängen.

Geht es besser?

Eine Mehr-Zyklus Implementierung

Man kann die Abarbeitung der Befehle auch auf mehrere Taktzyklen abbilden.

Vorteile

- Laufen Speicherzugriffe für Daten und Befehle in unterschiedlichen Zyklen ab, kommt man wieder mit nur einem Speicher aus. **Von Neumann Architektur**
- Befehle, die nicht alle Resourcen nutzen (z.B. Arithmetik greift nicht auf den Datencache zu) können in weniger Zyklen laufen.
- Man kann die Befehle im Fließbandverfahren bearbeiten, d.h. mehrere Befehle in verschiedenen Zyklen parallel (mit Abstrichen bei den ersten beiden Vorteilen). Dieses **Pipelining** Prinzip wird in allen modernen Prozessoren angewandt.

Beispiel: 5-Zyklus WüRC

Wir erkennen in unserer 1-Zyklus Maschine 5 zeitintensive Schritte:

- Befehl holen **(instruction fetch)** (IF)
- Befehl dekodieren und Register auslesen
  (instruction decode and register fetch) (ID)
- Berechnung ausführen **(execute)** (EX)
- Speicherzugriff **(memory access)** (MEM)
- Ergebnis wegschreiben **(write back)** (WB)

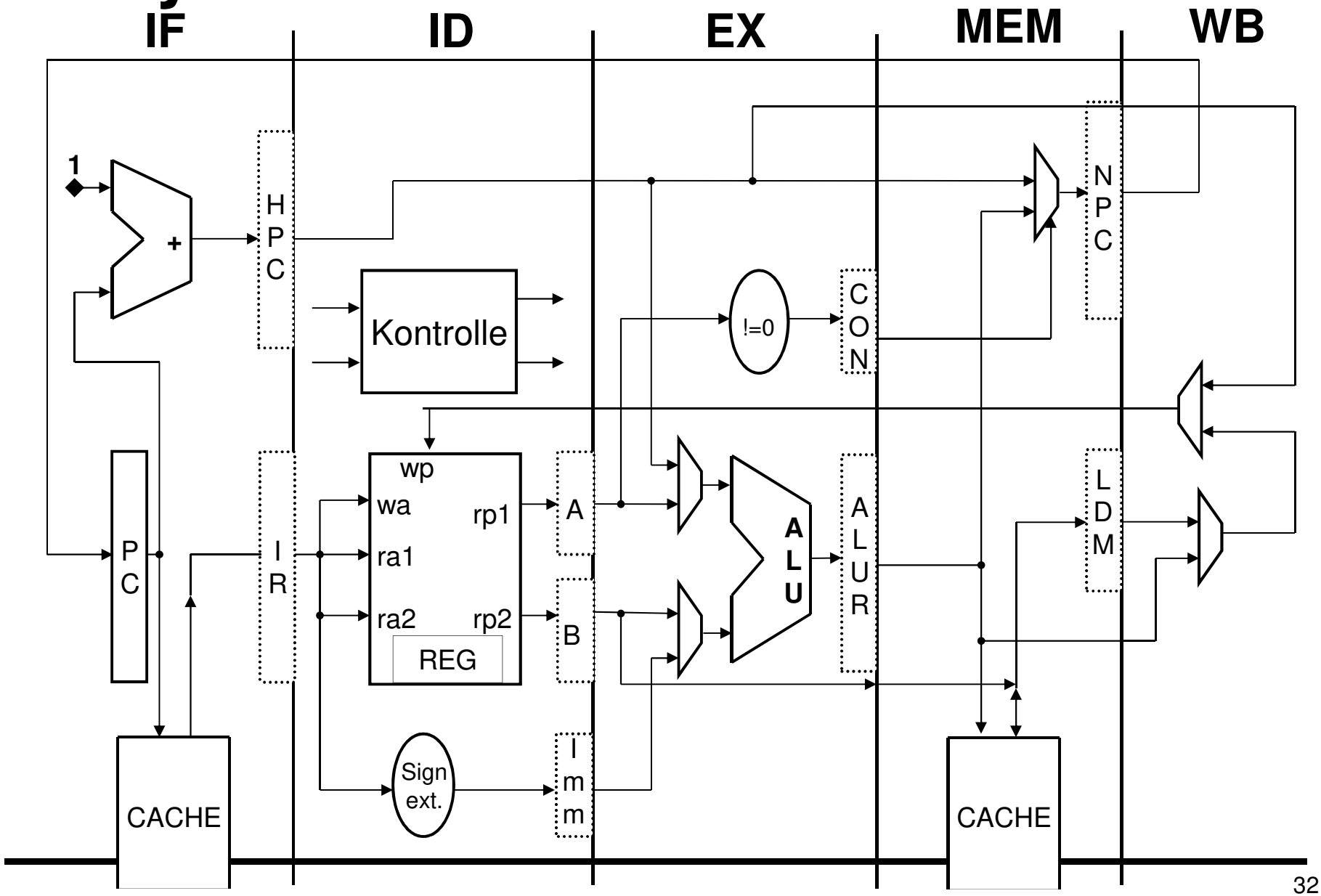
Nicht jede dieser Phasen wird von jedem Befehl echt benutzt.

5-Zyklus WüRC ff

Um diese 5 Phasen voneinander zu trennen, müssen wir

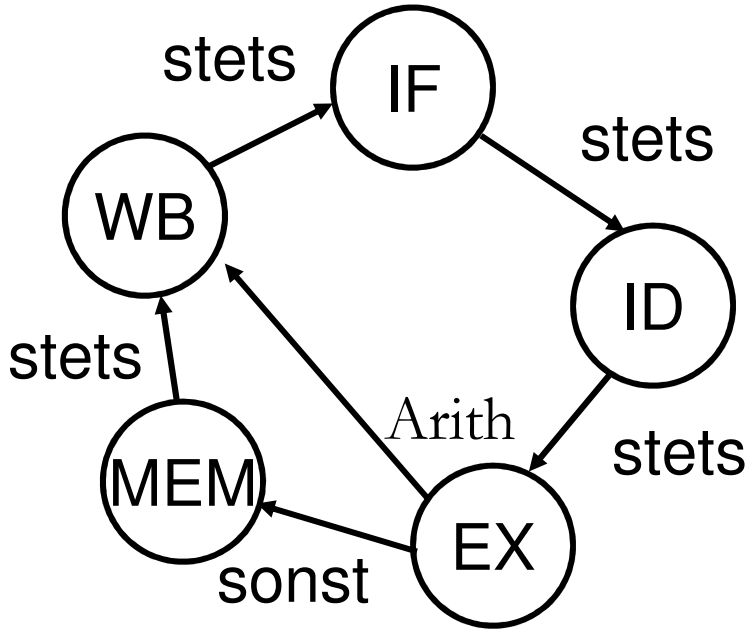
- zusätzliche Register einbauen, nämlich
  - Ein Instruktionsregister (IR), beschrieben in IF
  - Ein HilfsPC Register (HPC), beschrieben in IF
  - Operandenregister (A,B,Imm) beschrieben in ID
  - Ein Bedingungsregister (CON), beschrieben in EX
  - Ergebnisregister der ALU (ALUR) beschrieben in EX
  - Speicherdatenregister (LDM) beschrieben in MEM
  - Ein weiteres PC Register (NPC), beschrieben in MEM
- die Kontrolle in einen Automaten umwandeln, der die Kontrollsignale zu den richtigen Zeiten erzeugt.

5-Zyklus WüRC ff



5-Zyklus Kontrolle

Die Kontrolle unserer 5-Zyklus Version wird nun zur Finite State Machine, die in den jeweiligen Zyklen die entsprechenden Kontrollsignale für Register und Multiplexer erzeugt. Je nach Befehl kann sie auch Zyklen überspringen.



5-Zyklus WüRC -- Performanz

Die Taktperiode dieser Implementierung ist im wesentlichen bestimmt durch

$$\Gamma_{clk} \geq max\{t_{access\_cache}, t_{access\_reg}, t_{ALU}\} + t_0$$

worin t_0 wieder die Laufzeit von Leitungen, Multiplexern und Dekodierlogik ist. Wir würden also etwa $4t_0$ gegenüber der Einzyklus Maschine verlieren, wenn Caches, Register und ALU etwa gleich schnell sind, **aber**

- wir benötigen eigentlich nur noch einen Cache und
- nicht alle Befehle dauern 5 Zyklen!

Die durchschnittliche Laufzeit hängt nun entscheidend vom **Instruktionsmix** ab!

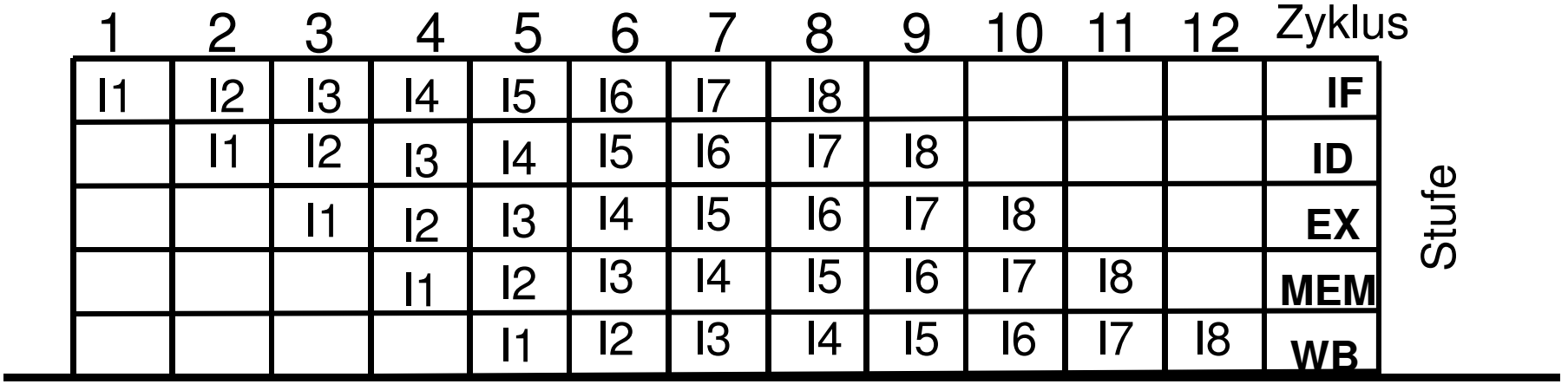
Pipeline WüRC

Wenn wir darauf bestehen, dass

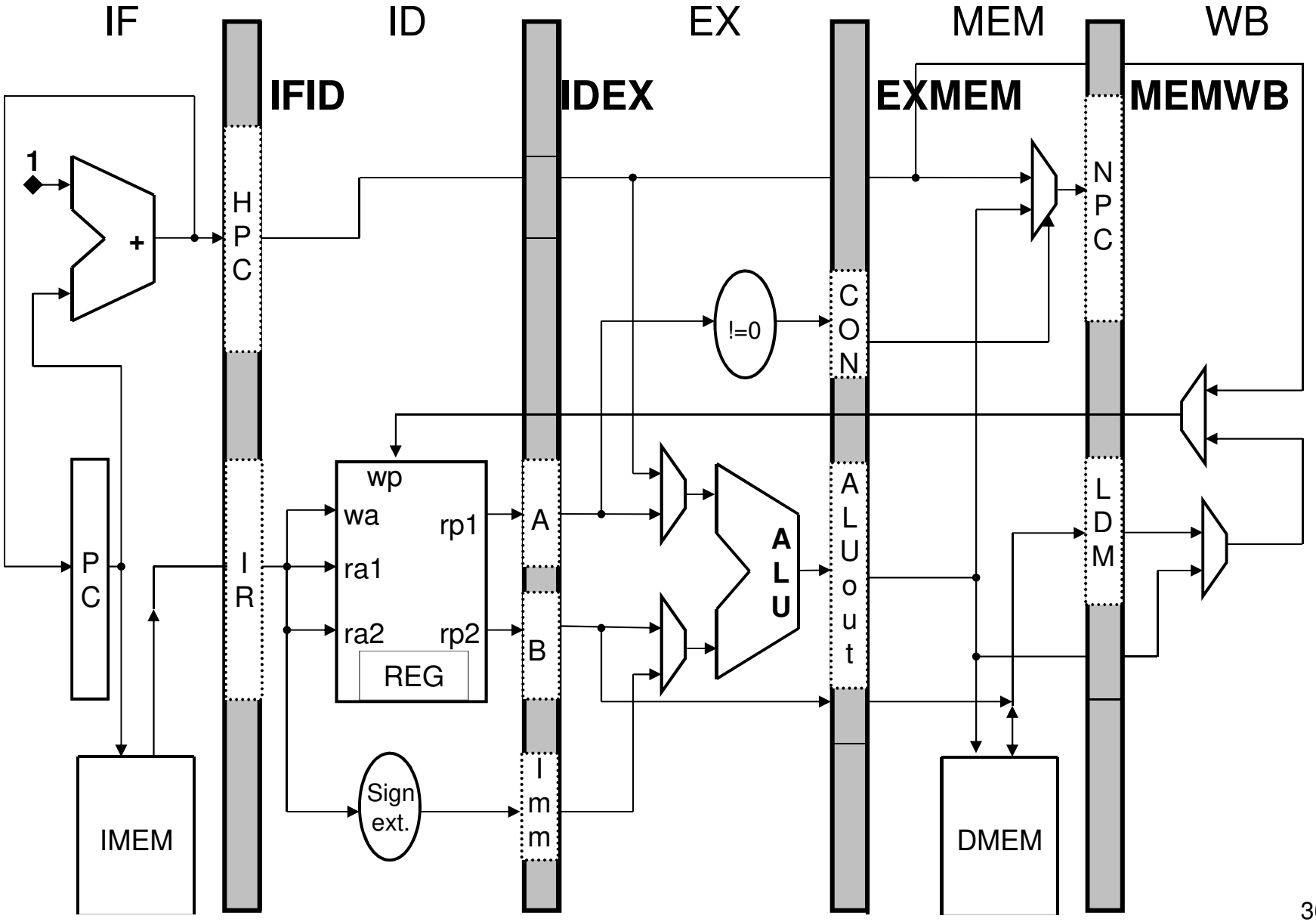
- Befehls- und Datencaches benutzt werden und
- jeder Befehl in 5 Zyklen bearbeitet wird,

dann kann man die Befehle parallel im **Fließbandprinzip** bearbeiten, d.h. zu einem Zeitpunkt in jedem Zyklus einen anderen Befehl.

Veranschaulichung für 8 Befehle: I1,…,I8



Realisierung einer Pipeline: 1. Versuch



Realisierung: 1.Versuch

Wir nehmen zunächst an, dass wir eine Befehlsfolge ohne Sprünge verarbeiten. Dann liegt der PC + 1 schon direkt im nächsten Zyklus vor, und wir können in jedem Zyklus einen Befehl holen.

Damit uns Zwischenzustände der Maschine nicht verlorengehen, trennen wir jedes Paar von benachbarten Phasen durch ein Pipelineregister. Diese Register nennen wir nach den Phasen, die sie trennen: IFID, IDEX, EXMEM, MEMWB.

Es müssen nun sämtliche Kontrollsignale und Zwischenergebnisse, die in einer Stufe entstehen, durch alle nachfolgenden Pipelineregister weitergereicht werden. Wir benennen die Teile des Pipelineregisters entsprechend mit den Namen der Zwischenergebnisse:

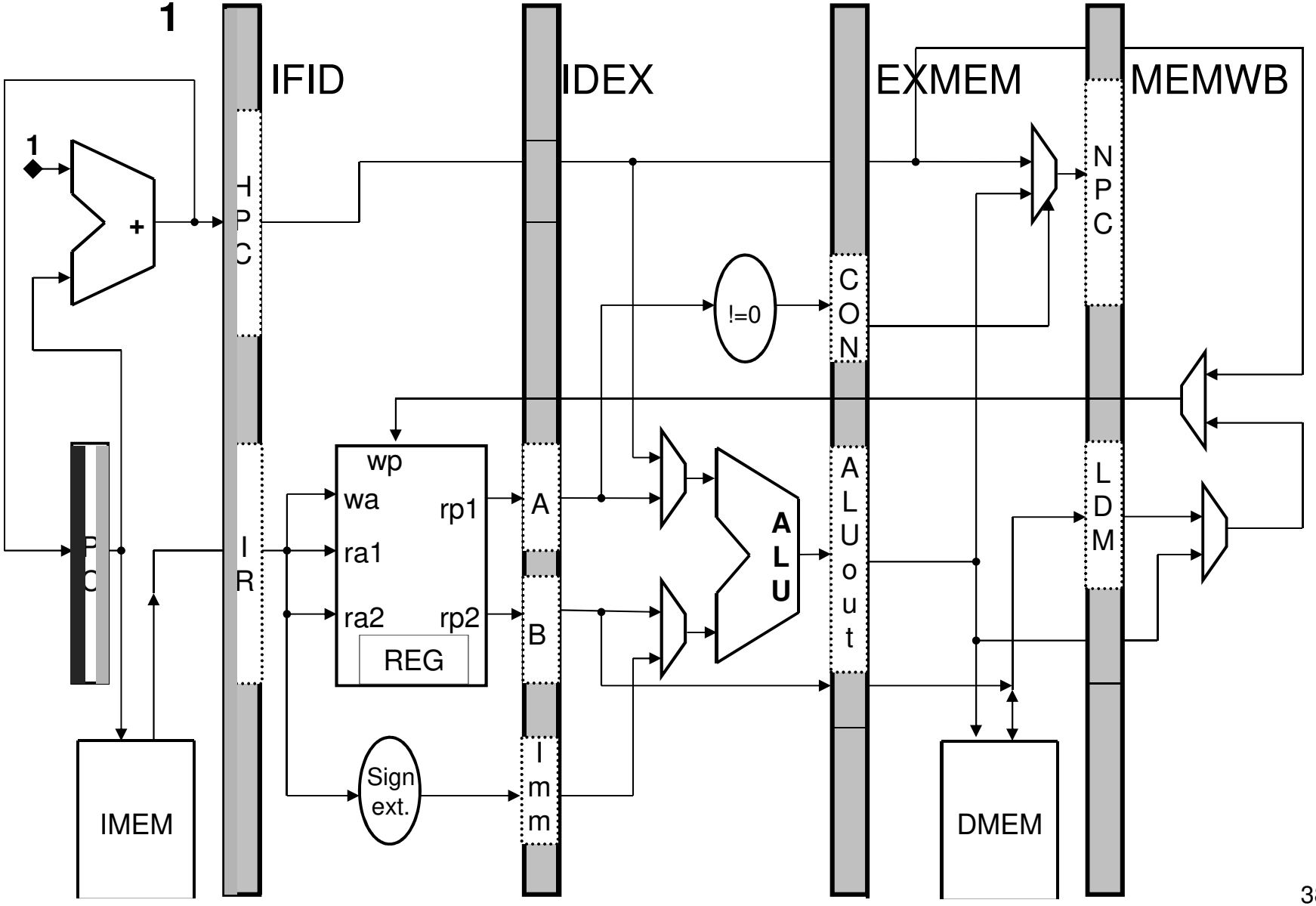
IR wird zu IFID.IR und findet sich ggf. dekodiert wieder in IDEX.IR, EXMEM.IR, MEMWB.IR

Ebenso wird HPC zu IFID.HPC, IDEX.HPC,...

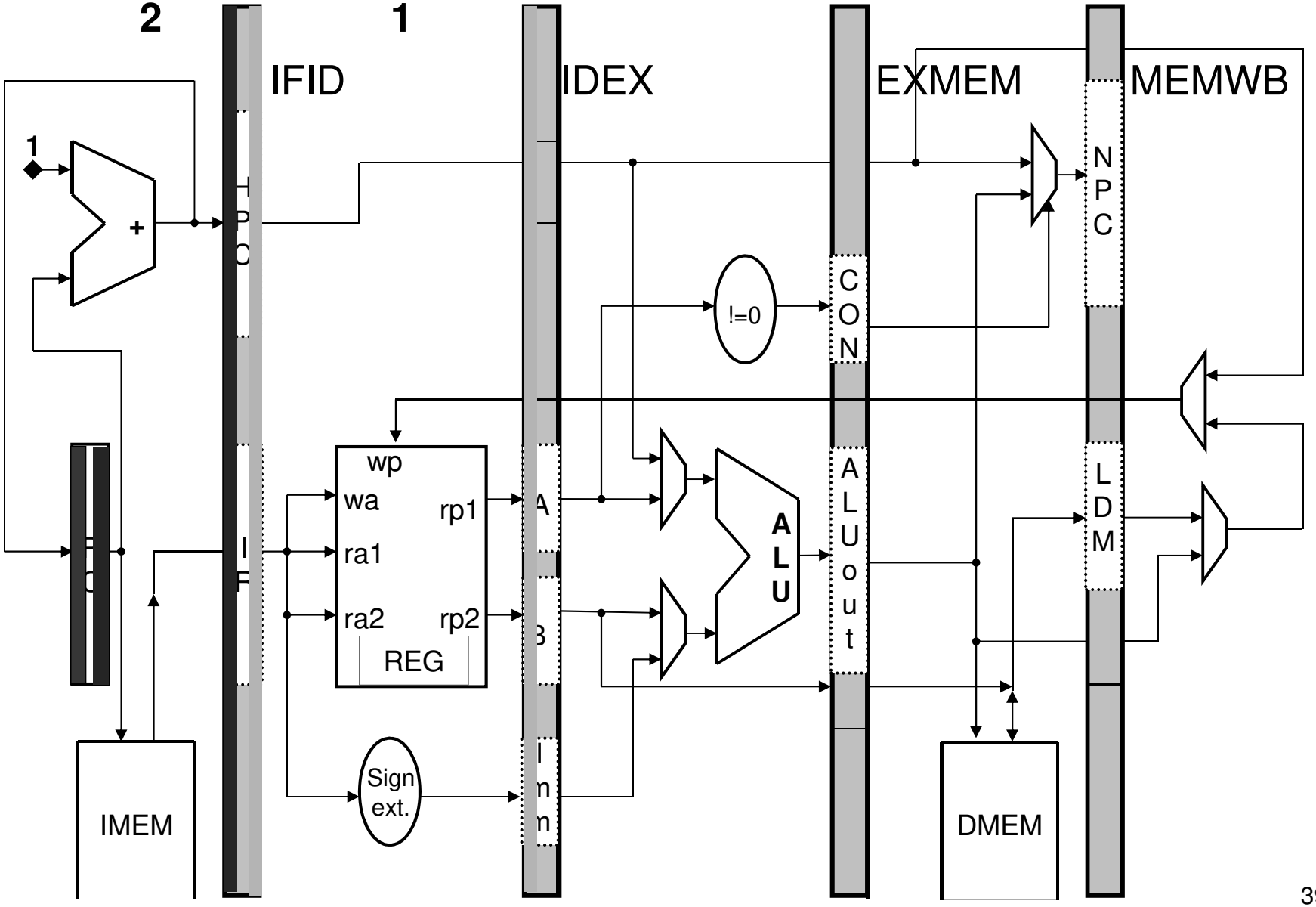
oder ALU_out zu EXMEM.ALU_out, MEMWB.ALU_out,...

oder A zu IDEX.A, EXMEM.A, ...

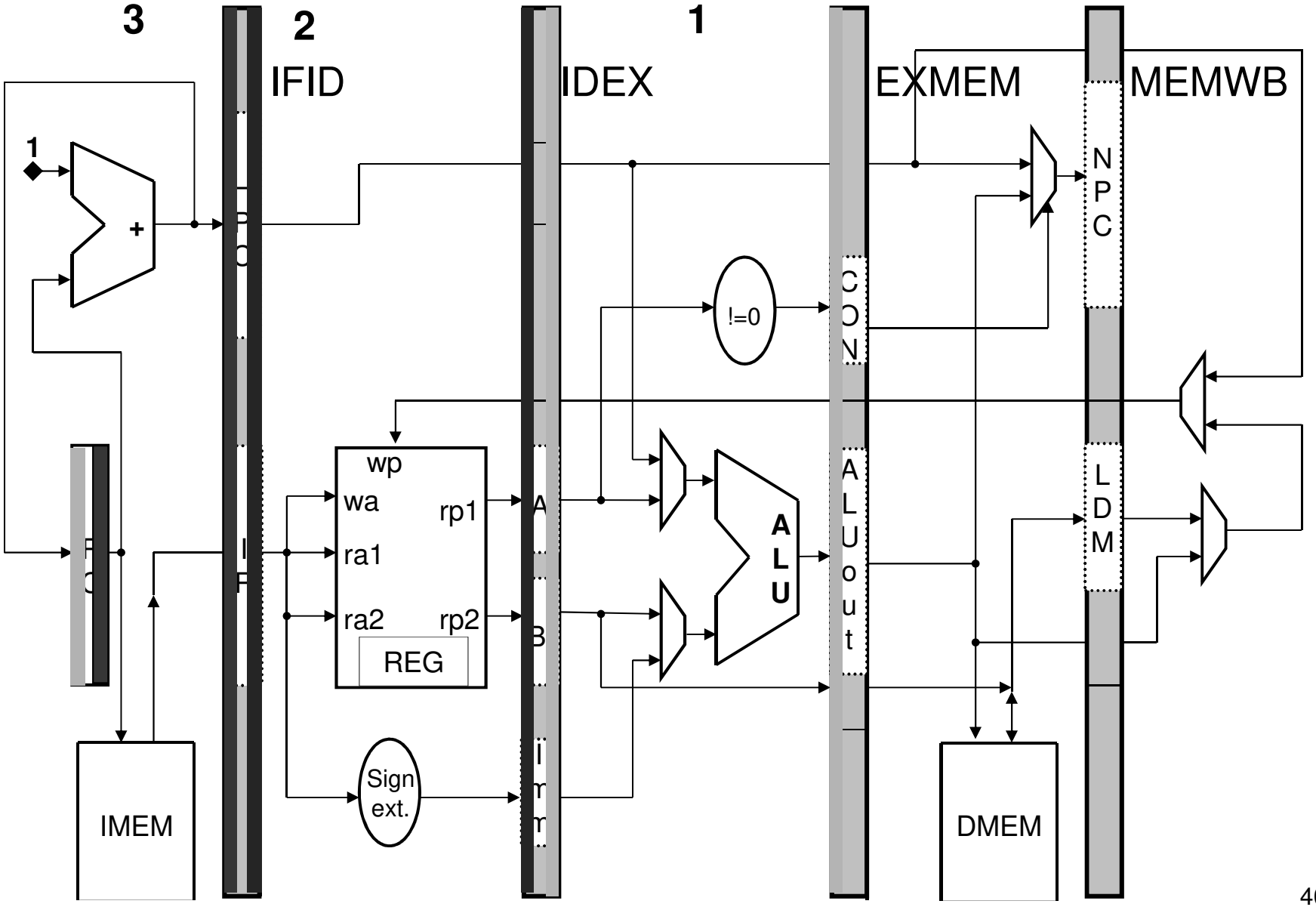
Realisierung einer Pipeline: 1. Versuch -- ff



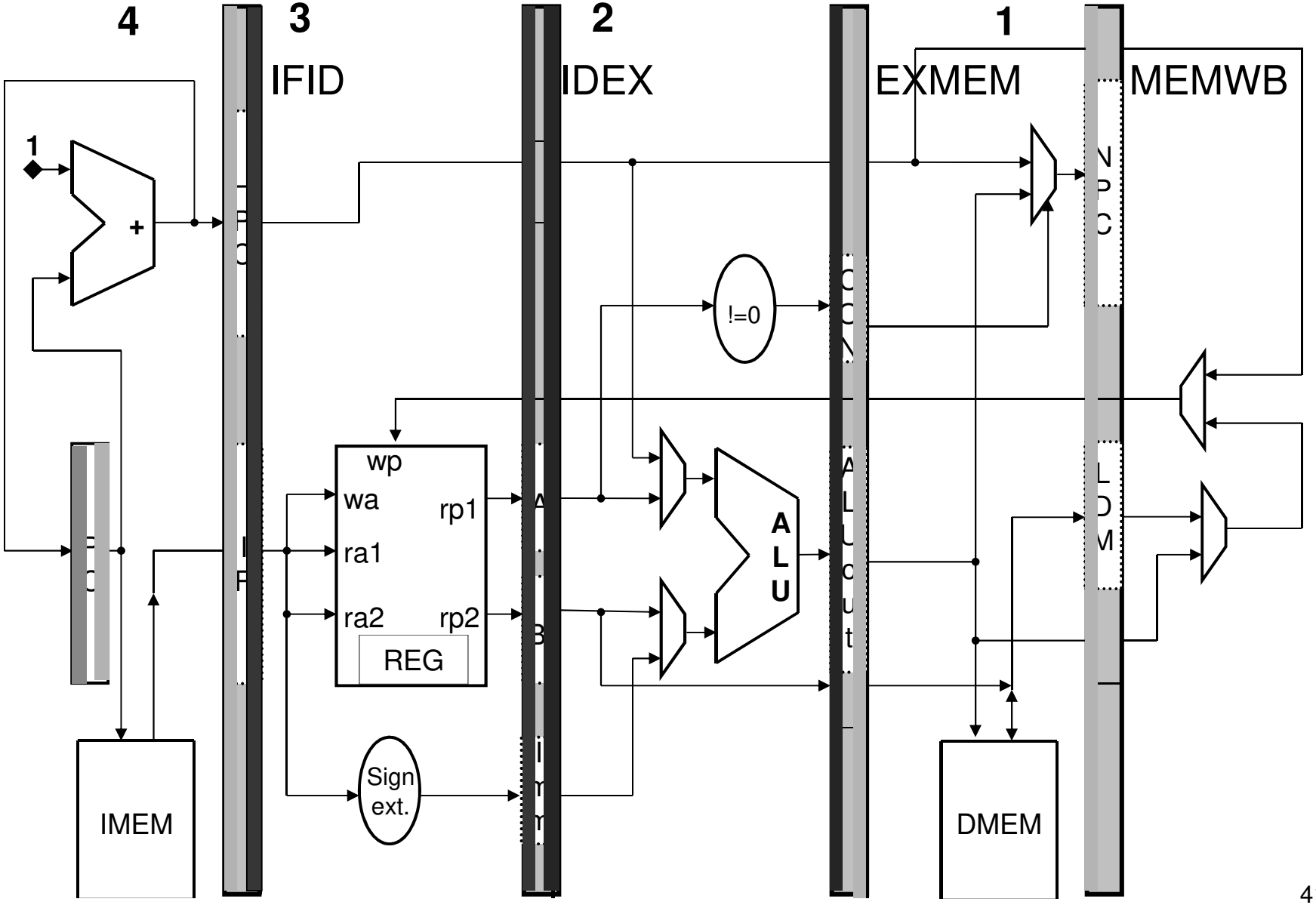
Realisierung einer Pipeline: 1. Versuch -- ff



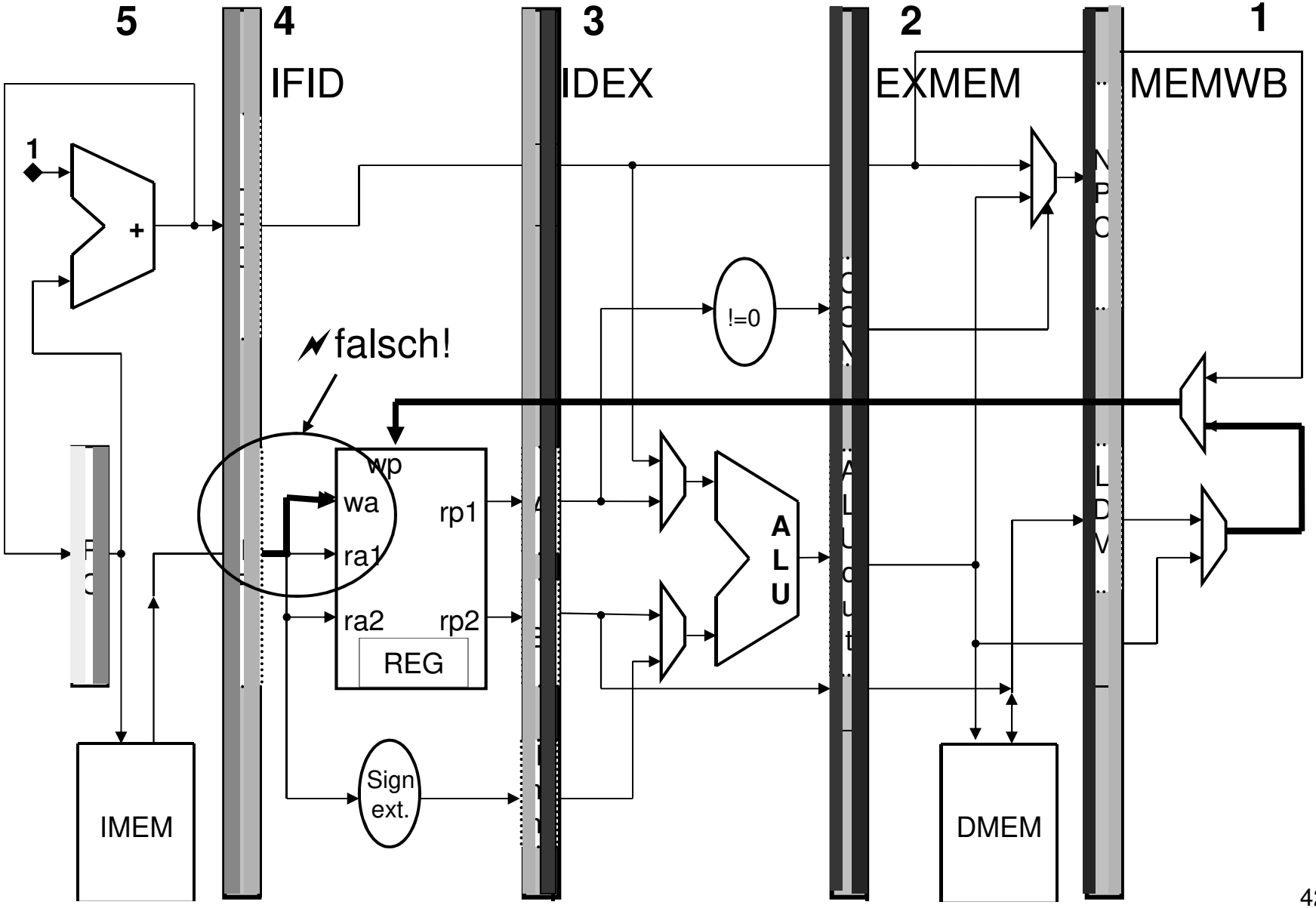
Realisierung einer Pipeline: 1. Versuch -- ff



Realisierung einer Pipeline: 1. Versuch -- ff



Realisierung einer Pipeline: 1. Versuch -- ff

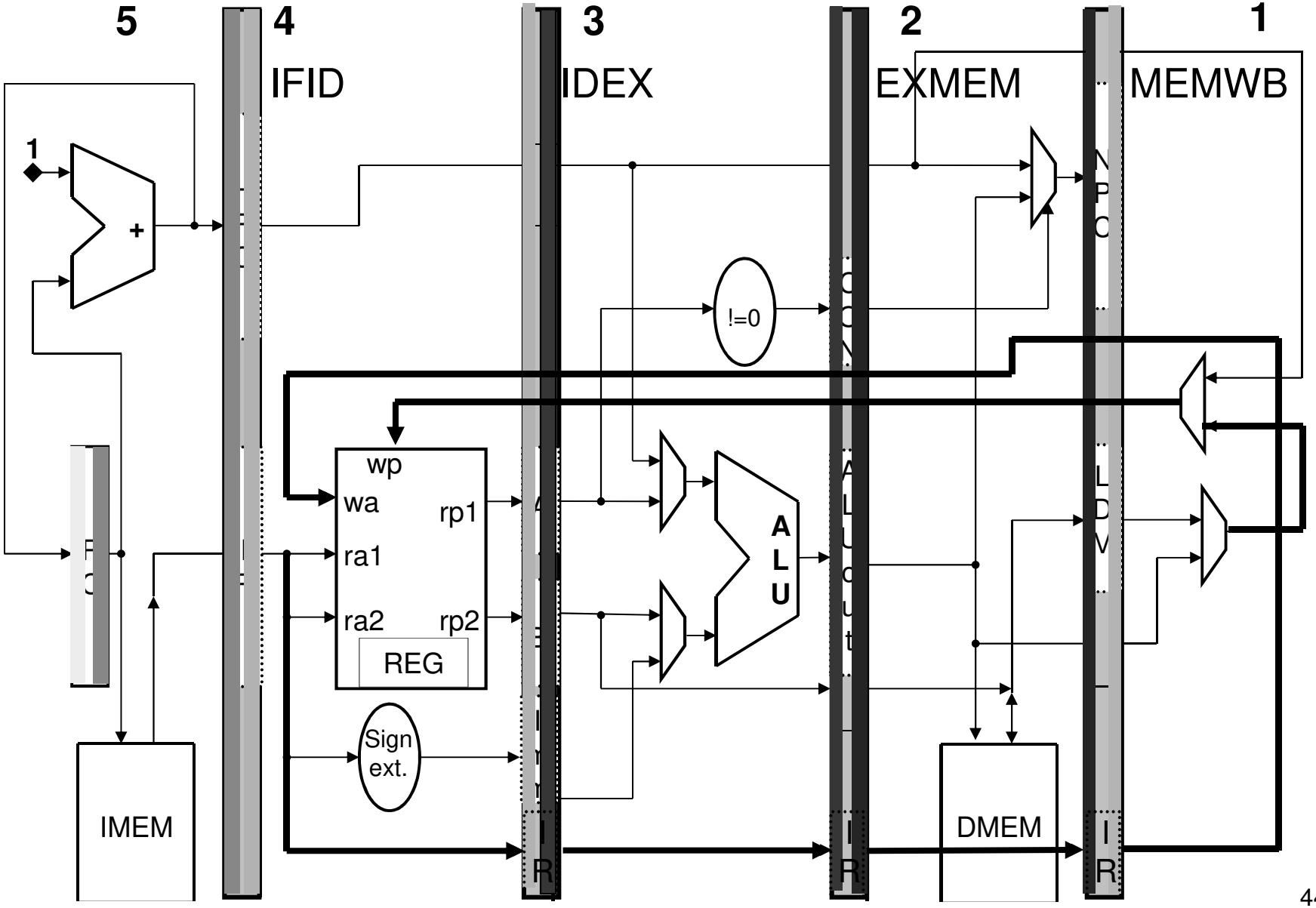


Fehler in der Hardware

Wir erkennen einen kapitalen Fehler in unserer Pipeline:
Die Adresse des Zieloperandens darf nicht direkt angelegt werden, sondern muss durch die Pipeline hindurchgeschleift werden und erst im WB als MEMWB.IR Anteil an die Registerbank gelegt werden.
Wir würden sonst den Zieloperand des Befehls *4* mit dem Ergebnis des Befehls *1* überschreiben!

Alle Kontrollinformation eines Befehls, die im *ID* Zyklus entsteht, muss bis zu der Pipelinestufe, in der sie wirkt, über Pipelineregister weitergereicht werden!

Realisierung einer Pipeline: 1. Versuch -- ff



Pipeline WüRC

Wäre es wirklich so einfach, dann könnten wir beliebige Befehlsfolgen der Länge *n* in *n+4* Zyklen bearbeiten, was für große *n* einer Rate von ca. 1 Zyklus pro Befehl entspricht. Wir wären also fast 5 mal schneller als die ein Zyklus Maschine, wenn Cache- und ALU-Laufzeiten etwa gleich sind.

Pferdefüße:

- Datenhazards: Befehle in der Pipeline verändern Operanden anderer Befehle in der Pipeline.
- Kontrollhazards: Verzweigungen in der Pipeline machen Nachfolgebefehle ungültig.

Der prinzipelle Aufbau erfordert allerdings nur ein paar Register mehr!

Ausblick

Pipelining ist nicht so leicht umzusetzen wie hier skizziert, neben den schon erwähnten Problemen kommen noch Dinge wie

- Rechenwerke (z.B. Gleitkomma, Division, Quadratwurzel, ...) mit unterschiedlich tiefen Pipeline-stufen,
- Unterbrechungen und Ausnahmebedingungen,
- Code Scheduling, …

Will man noch höhere Raten als einen Befehl pro Zyklus, muss man die Datenpfade breiter auslegen und versuchen, mehrere Befehle parallel zu holen und zu bearbeiten. → **Superskalare Prozessoren**

Dann wirds noch interessanter (schwieriger).