Digitaltechnik

7 Ein einfacher CISC-Prozessor

Revision 2.1

Die Y86 Instruction Set Architecture

Die Y86-ISA in C++

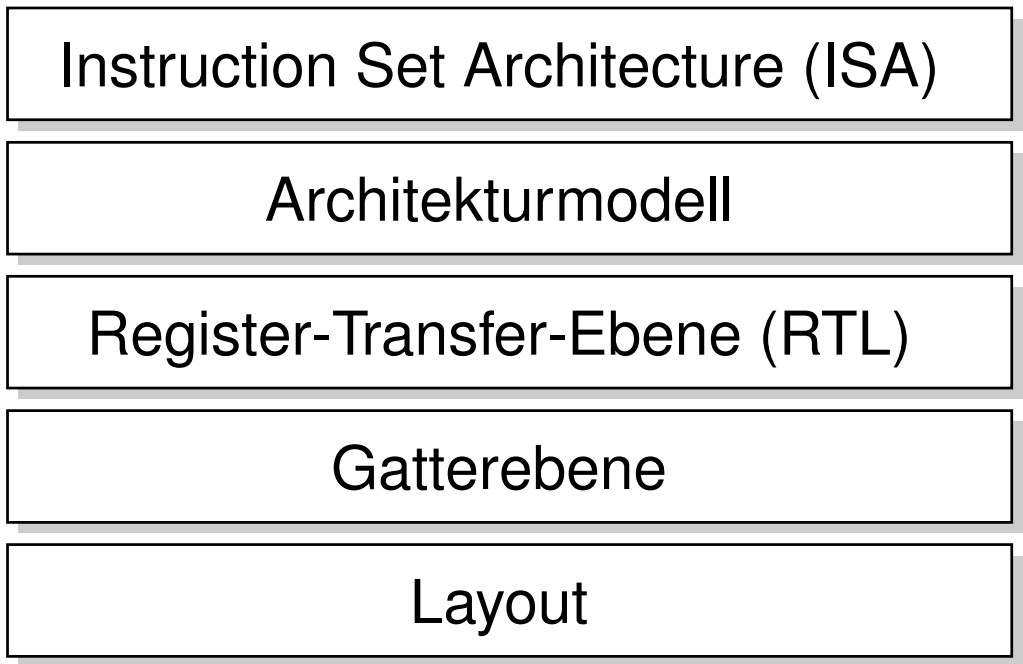
Eine sequenzielle Y86-Implementierung

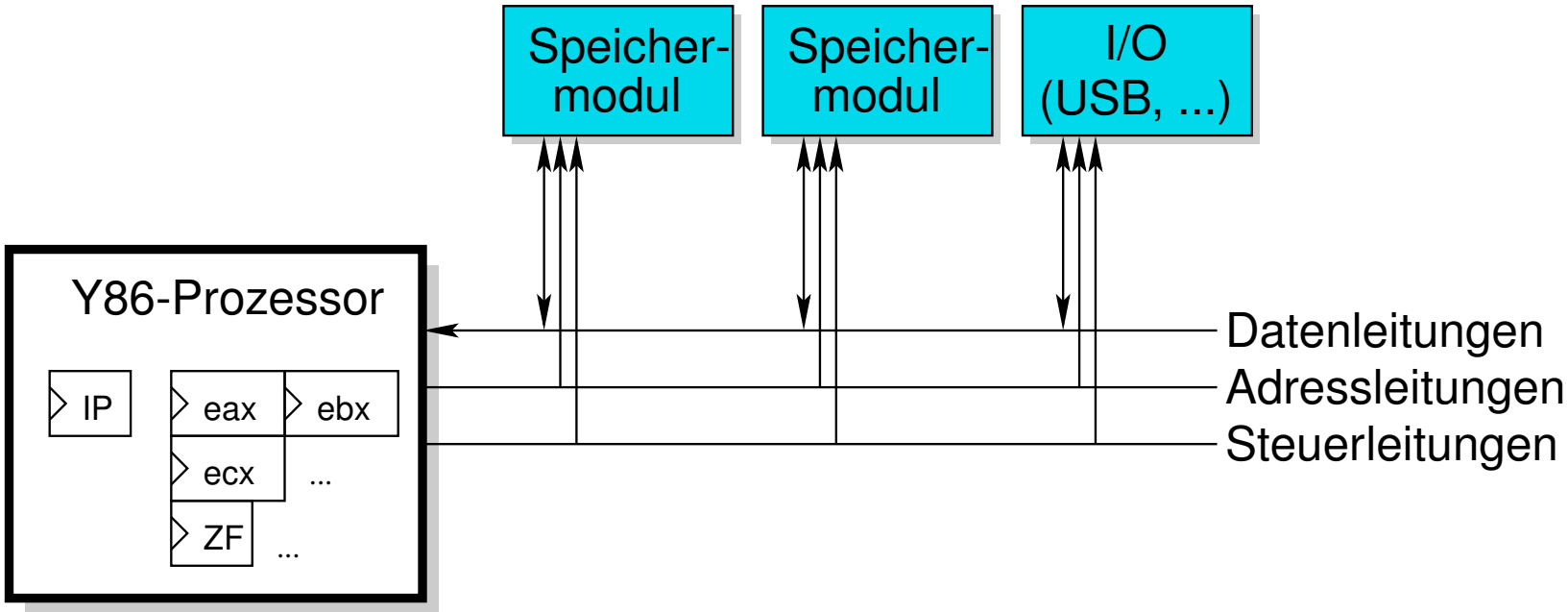
Erweiterung: Vergleiche

Eine Y86-Implementierung mit Pipeline

I/O

Beschreibungs-Ebenen





- Programme und Daten im selben Speicher (von Neumann Architektur)
- I/O ebenfalls über Speicherbus (memory-mapped I/O)

Sichtbare Register

RAM

- Enthält Daten und Programme

Datenregister

Index	0	1	2	3	4	5	6	7
Name	**eax**	**ecx**	**edx**	**ebx**	**esp**	**ebp**	**esi**	**edi**

Instruction Pointer (**IP**)

- Zeigt auf die nächste auszuführende Instruktion

Flag-Register (**ZF**, ...)

- Speichern Bedingungen für Sprungbefehle

Y86-Assembler

- Untermenge von Intels X86 Assembler

✔ Y86-Programme können auf X86-Maschinen ausgeführt werden (siehe Buch)

✘ I.A. nicht umgekehrt, da zu viele Befehle fehlen (aber: selbst ist der Student!)

Befehle

- $\mathbf{add/sub}$: Addition/Subtraktion der Werte in zwei Registern; $\mathbf{ZF}$ wird gesetzt

- $\mathbf{RRmov}$: Kopiert Wert von einem Register in ein anderes
- $\mathbf{RMmov}$: Kopiert Wert von einem Register in das RAM
- $\mathbf{MRmov}$: Kopiert Wert vom RAM in ein Register

- $\mathbf{jnz}$: Springt zur angegebenen Adresse wenn $\mathbf{ZF} = 0$

- $\mathbf{hlt}$: Die Ausführung wird angehalten

Zugriffe auf das RAM

- Zugriffe auf das RAM erfolgen mit einem Displacement:

$$\mathbf{ea} = \mathbf{esi} + \text{Displacement}$$

- Das Displacement ist in der Instruktion einkodiert

- Der $\mathbf{esi}$-Offset dient zur Implementierung von Arrays

Befehlsformate I

Mnemonic	Bedeutung	Opcode
add	RD←RD+RS	01 | 11 RS RD
sub	RD←RD-RS	29 | 11 RS RD
jnz	`if`(¬ZF) IP←IP+Distance	75 | Distance
RRmov	RD←RS	89 | 11 RS RD
RMmov	`MEM[ea]`←RS	89 | 01 RS 110 | Displacement
MRmov	RS←`MEM[ea]`	8b | 01 RS 110 | Displacement
hlt		f4

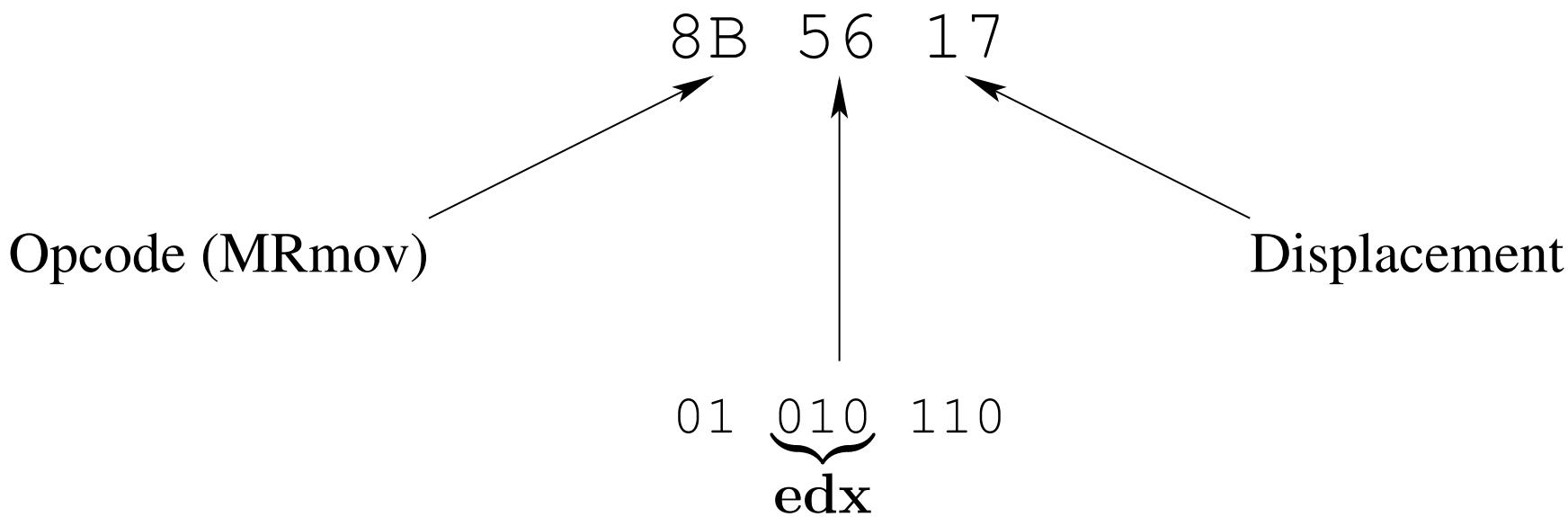
Beispiel 1

$$\mathbf{add\ eax,\ edx}$$

- Intel-Konvention: Das Zielregister steht immer links
- Das Zielregister ist auch ein Quellregister
- Semantik:

$$\mathbf{eax} \leftarrow \mathbf{eax} + \mathbf{edx}$$

mov edx, [**BYTE** one+**esi**]



Semantik:

$$\mathbf{edx} \leftarrow \mathsf{MEM}[\mathbf{esi}+17]$$

Wie funktionieren Sprungbefehle?

```
if(a==b) {
    T;
}
else {
    F;
}
```

Wie funktionieren Sprungbefehle?

```
if(a==b) {
   T;
}
else {
   F;
}
```

→

```
    mov eax, [BYTE a+esi]
    mov ebx, [BYTE b+esi]
    sub eax, ebx
    jnz f
    ;
    ;   Code für 'T'
    ;
    mov eax, [BYTE one+esi]
    add eax, eax
    jnz e
f   ;
    ;   Code für 'F'
    ;

e   ;   ...
```

Beispiel

Adresse	Maschinencode	Assembler mit Mnemonics	
00	29 F6		**sub esi**, **esi**
02	29 C0		**sub eax**, **eax**
04	29 DB		**sub ebx**, **ebx**
06	8B 56 17	l	**mov edx**, [**BYTE** one+**esi**]
09	01 D0		**add eax**, **edx**
0B	01 C3		**add ebx**, **eax**
0D	89 C1		**mov ecx**, **eax**
0F	8B 56 1B		**mov edx**, [**BYTE** ten+**esi**]
12	29 D1		**sub ecx**, **edx**
14	75 F0		**jnz** l
16	F4		**hlt**
17	01 00 00 00	one	**dd** 1
1B	0A 00 00 00	ten	**dd** 10

Der NASM Assembler

- Windows:

```
nasm -f win32 my_test.asm
link /subsystem:console /entry:start my_test.obj
```

- Linux:

```
nasm -f elf my_test.asm
ld -s -o my_test my_test.o
```

- MacOS:

```
nasm -f macho my_test.asm
ld -arch i386 -o my_test my_test.o
```

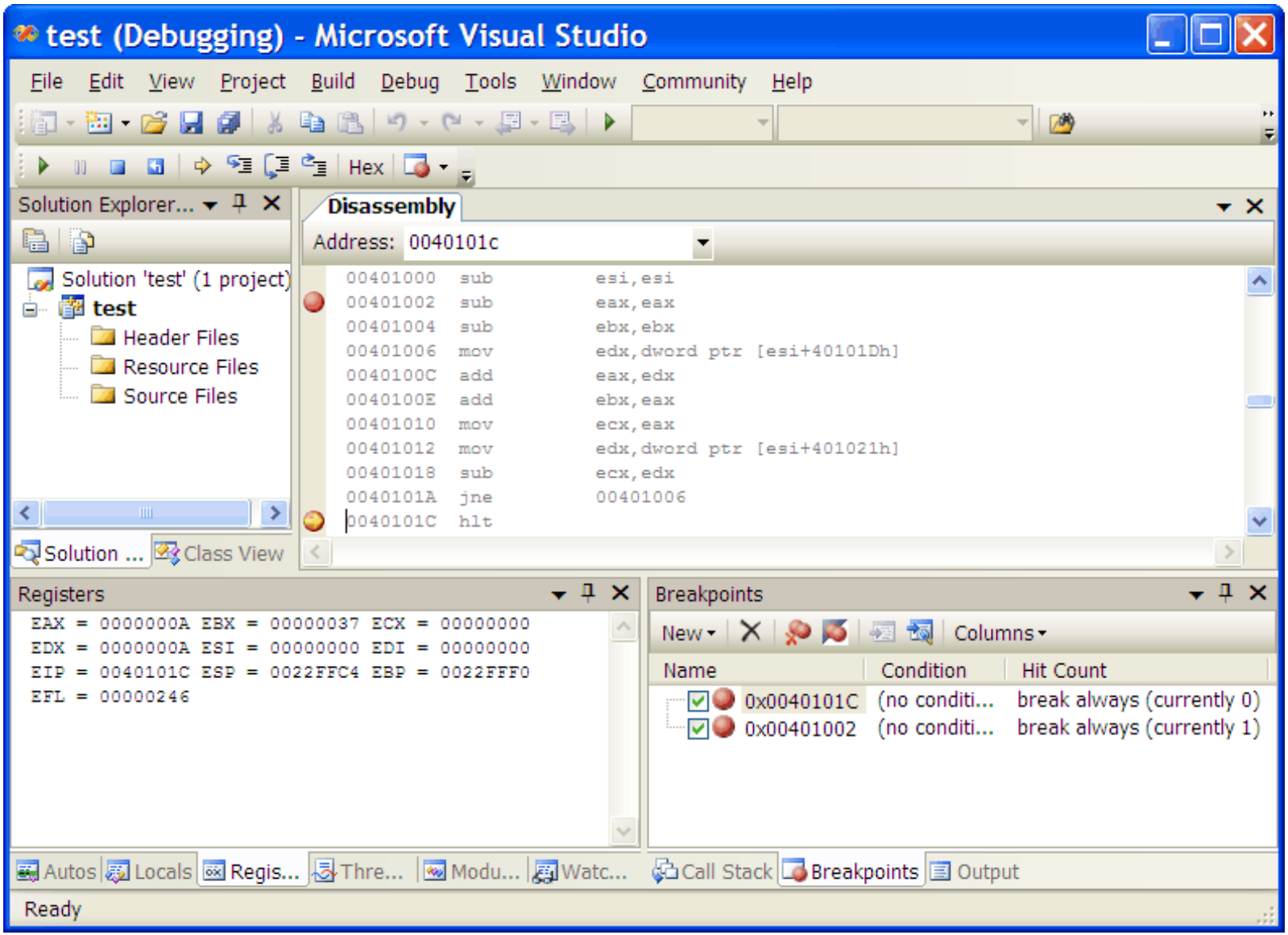
Debugging mit GDB (Teil 1)

- `run`
  Ausführung beginnen
- `x/[`*Anzahl*`]` *Label*
  Speicherbereich ausgeben
- `x/[`*Anzahl*`i]` *Label*
  Speicherbereich disassemblieren, z. B. `x/5i $pc`
- `info registers`
  Werte der Register ausgeben
- `step`
  Eine Instruktion ausführen

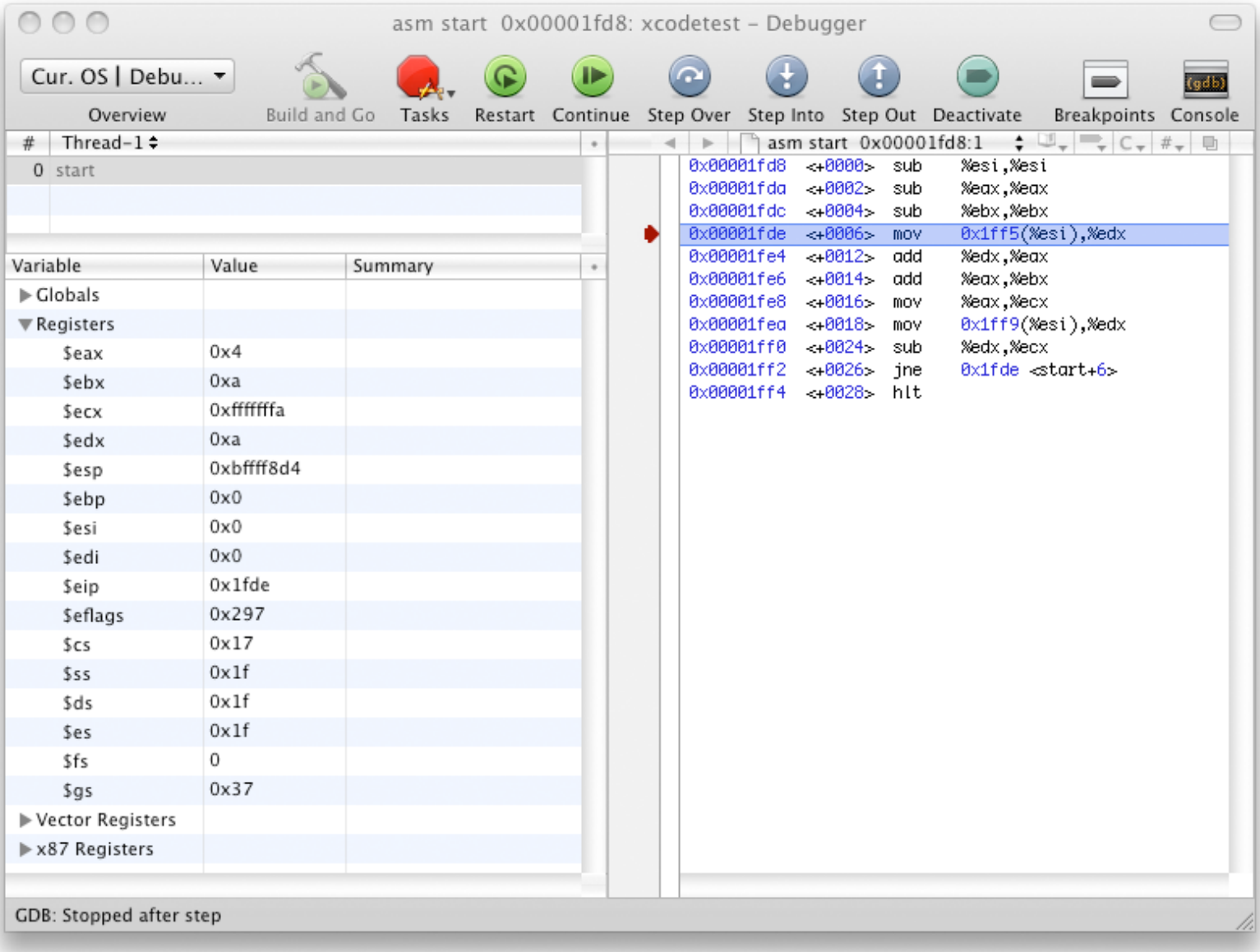
Debugging mit GDB (Teil 2)

- `break` *Label*
  Breakpoint setzen
- `info break`
  Breakpoints anzeigen
- `delete breakpoints` *Nummer*
  Breakpoint löschen
- `continue` Die Ausführung fortsetzen

Debugging mit Visual Studio



Debugging mit XCode



Die Y86-ISA in C++

- Noch keine Implementierungsdetails
- Dient als Spezifikation und für Simulationen
- Sehr kompakt
- Sehr beliebte Modellierungstechnik in der Industrie!
- Unterschiedliche Modelle für Performance, Power, ...

ISA in C++ (1. Teil)

```
#define RD(I)        ((I)&0x07)
#define RS(I)        (((I)&0x38)>>3)
#define mod(I)       (((I)&0xc0)>>6)
#define EXTEND8(x)   ((signed int)(signed char)(x))
```

Die Operatoren `&` und `>>` funktionieren wie in Verilog

ISA in C++ (2. Teil)

```
class cpu_Y86t {
public:
  cpu_Y86t():IP(0), ZF(false) { // Konstruktor: alles auf 0
    for(unsigned i=0; i<100; i++) MEM[i]=0;
    for(unsigned j=0; j<8; j++) R[j]=0;
  }

  unsigned IP;              // Der Instruction Pointer
  unsigned char MEM[100]; // Hauptspeicher: 100 Adressen
  unsigned R[8];            // Die Werte der 8 Register
  bool ZF;                  // Flags

  void show(); // Zustand anzeigen
  void step(); // eine Instruktion ausführen
  void run();  // Programm ausführen
};
```

ISA in C++ (3. Teil)

```
void cpu_Y86t::step() {
  unsigned ea;
  unsigned I0=MEM[IP], I1=MEM[IP+1], I2=MEM[IP+2];
  IP=IP+1;

  switch(I0) {
  case MRmov:
    if(mod(I1)==1) { // Memory zu Register
      IP+=2;
      ea=R[6]+EXTEND8(I2);
      R[RS(I1)]=mem32(ea);
    }
    break;
```

ISA in C++ (4. Teil)

```
case RMmov:
  if(mod(I1)==1) // Register zu Memory
  {
    IP+=2;
    ea=R[6]+EXTEND8(I2);
    MEM[ea+0]=(R[RS(I1)]&0xff);
    MEM[ea+1]=(R[RS(I1)]&0xff00)>>8;
    MEM[ea+2]=(R[RS(I1)]&0xff0000)>>16;
    MEM[ea+3]=(R[RS(I1)]&0xff000000)>>24;
  }
  else if(mod(I1)==3) { // Register zu Register
    IP+=1;
    R[RD(I1)]=R[RS(I1)];
  }
  break;
```

ISA in C++ (5. Teil)

```
  case add:
    IP+=1;
    R[RD(I1)]+=R[RS(I1)];
    ZF=(R[RD(I1)]==0);
    break;

  case sub:
    IP+=1;
    R[RD(I1)]-=R[RS(I1)];
    ZF=(R[RD(I1)]==0);
    break;

  case jnz:
    IP+=1;
    if(!ZF) IP+=EXTEND8(I1);
    break;

  default:;
  } }
```

ISA in C++: Beispielausgabe

```
MRmov edx, [ESI+0x17]

IP=9

eax=         9 ecx= ffffffff edx=         1 ebx=        2d
esp=         0 ebp=         0 esi=         0 edi=         0 ZF=0

MEM:  29 f6 29 c0 29 db 8b 56 17  1 d0  1 c3 89 c1 8b
      56 1b 29 d1 75 f0 f4  1  0  0  0  a  0  0  0  0
```

Eine sequenzielle Y86-Implementierung

- Brücke zwischen Digitaltechnik und Computerarchitektur
- einfachste, unrealistische Variante in Verilog
- zeigt nur prinzipielle Vorgehensweise
  (Separierung von Controller/Datenpfad, Pipeline)
- ✘ moderne Konzepte wie Out-of-Order Execution fehlen ganz

  mehr dazu in der Vorlesung *Computer Architecture*

Aufteilung der Ausführung in 5 Phasen (Stufen)

1. **Instruction Fetch (IF)**
   Lädt das Instruktionswort aus dem RAM in das Register IR

Aufteilung der Ausführung in 5 Phasen (Stufen)

1. **Instruction Fetch (IF)**
   Lädt das Instruktionswort aus dem RAM in das Register IR

2. **Instruction Decode (ID)**
   Lädt die Operanden aus dem Register File in die Register A und B, Inkrementierung des IPs

Aufteilung der Ausführung in 5 Phasen (Stufen)

1. **Instruction Fetch (IF)**
   Lädt das Instruktionswort aus dem RAM in das Register IR

2. **Instruction Decode (ID)**
   Lädt die Operanden aus dem Register File in die Register A und B,
   Inkrementierung des IPs

3. **Execute (EX)**
   Ausführung einer evtl. **add/sub** Instruktion,
   Addressarithmetik für RAM-Zugriffe

Aufteilung der Ausführung in 5 Phasen (Stufen)

1. **Instruction Fetch (IF)**
   Lädt das Instruktionswort aus dem RAM in das Register IR

2. **Instruction Decode (ID)**
   Lädt die Operanden aus dem Register File in die Register A und B,
   Inkrementierung des IPs

3. **Execute (EX)**
   Ausführung einer evtl. **add/sub** Instruktion,
   Addressarithmetik für RAM-Zugriffe

4. **Memory (M)**
   Zugriff auf das RAM

Aufteilung der Ausführung in 5 Phasen (Stufen)

1. **Instruction Fetch (IF)**
   Lädt das Instruktionswort aus dem RAM in das Register IR

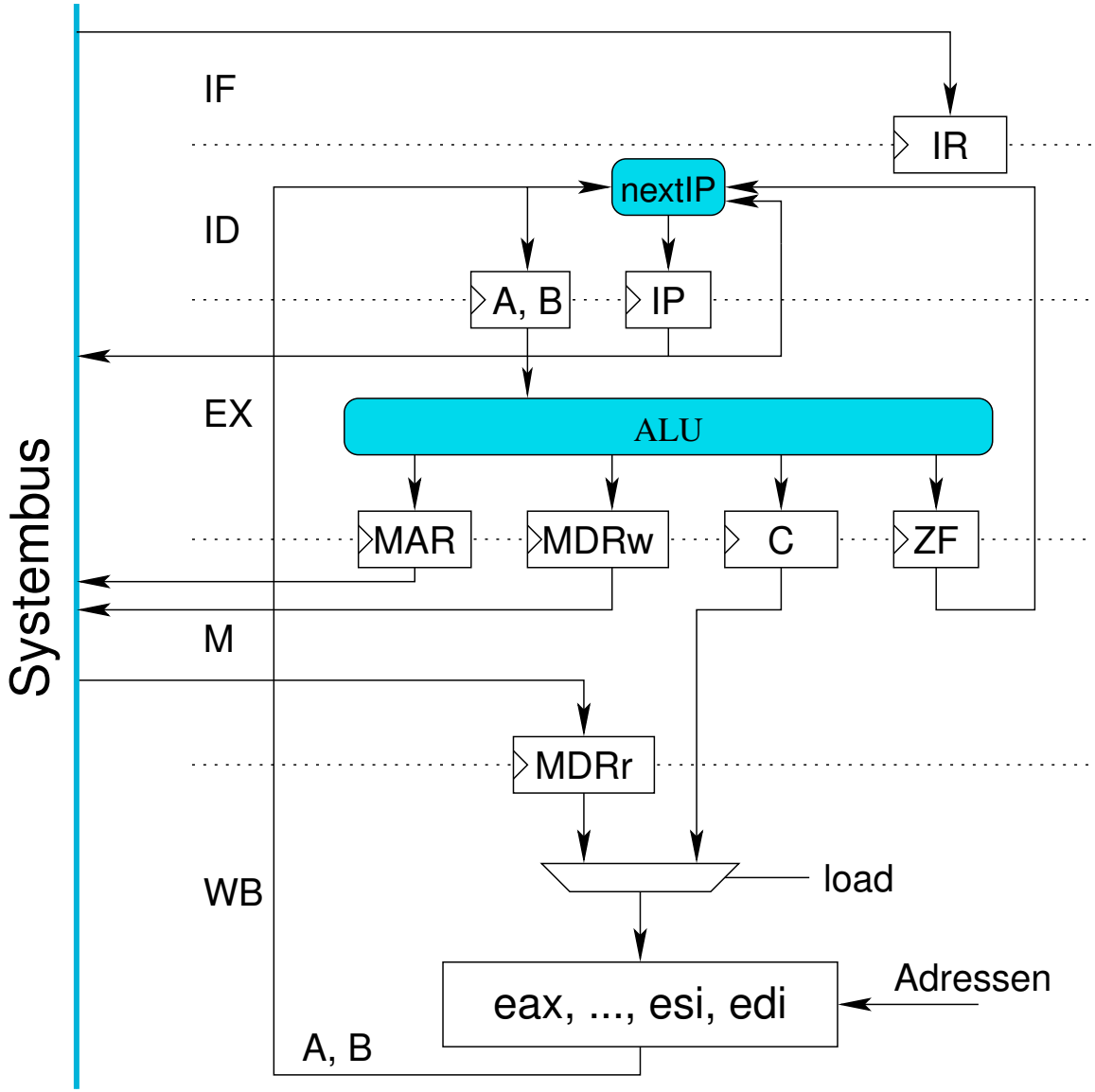
2. **Instruction Decode (ID)**
   Lädt die Operanden aus dem Register File in die Register A und B, Inkrementierung des IPs

3. **Execute (EX)**
   Ausführung einer evtl. **add/sub** Instruktion, Addressarithmetik für RAM-Zugriffe

4. **Memory (M)**
   Zugriff auf das RAM

5. **Write-Back (WB)**
   Speicherung des Ergebnisses von **add/sub** und **MRmov** im Register File

Strukturelle Implementierung



Sequenzielle Ausführung

- Wir implementieren zunächst eine sequenzielle Maschine:
  Die Stufen werden in der Reihenfolge IF – ID – EX – M – WB abgearbeitet

- Nicht alle Stufen werden von allen Instruktionen verwendet – so wird die Stufe M nur von `RMmov` bzw. `MRmov` verwendet

- Wir führen die Phase trotzdem aus

Sequenzielle Ausführung

Es sei $I_1, I_2, \ldots$ die Folge der Instruktionen in Programmreihenfolge.

Zeit	0	1	2	3	4	5	6	7	8
IF	I_1								
ID									
EX									
MEM									
WB									

Sequenzielle Ausführung

Es sei $I_1, I_2, \ldots$ die Folge der Instruktionen in Programmreihenfolge.

Zeit	0	1	2	3	4	5	6	7	8
IF									
ID		I_1							
EX									
MEM									
WB									

Sequenzielle Ausführung

Es sei $I_1, I_2, \ldots$ die Folge der Instruktionen in Programmreihenfolge.

Zeit	0	1	2	3	4	5	6	7	8
IF									
ID									
EX			I_1						
MEM									
WB									

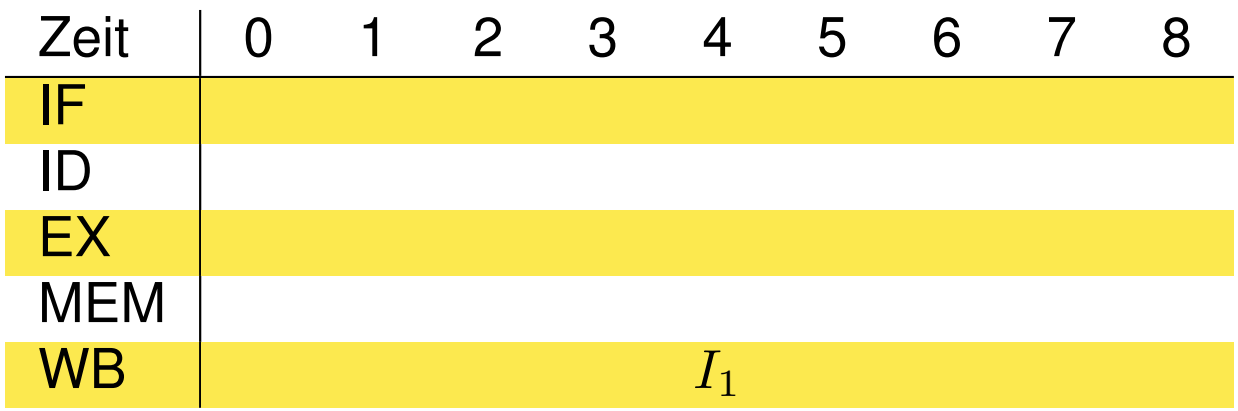
Sequenzielle Ausführung

Es sei $I_1, I_2, \ldots$ die Folge der Instruktionen in Programmreihenfolge.

Zeit	0	1	2	3	4	5	6	7	8
IF									
ID									
EX									
MEM				I_1					
WB									

Sequenzielle Ausführung

Es sei $I_1, I_2, \ldots$ die Folge der Instruktionen in Programmreihenfolge.



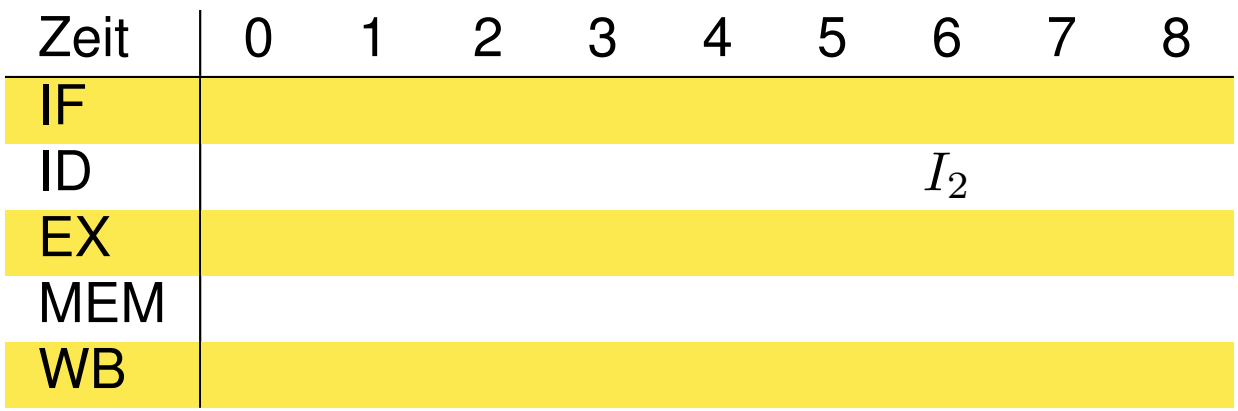
Sequenzielle Ausführung

Es sei I_1, I_2, ... die Folge der Instruktionen in Programmreihenfolge.

Zeit	0	1	2	3	4	5	6	7	8
IF						I_2			
ID									
EX									
MEM									
WB									

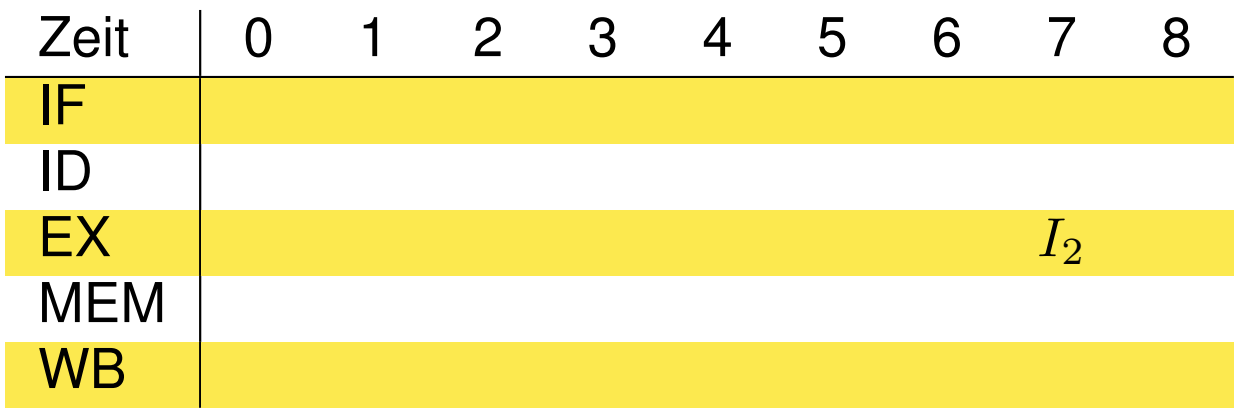
Sequenzielle Ausführung

Es sei I_1, I_2, ... die Folge der Instruktionen in Programmreihenfolge.



Sequenzielle Ausführung

Es sei $I_1, I_2, \ldots$ die Folge der Instruktionen in Programmreihenfolge.



Sequenzielle Ausführung

Es sei I_1, I_2, ... die Folge der Instruktionen in Programmreihenfolge.

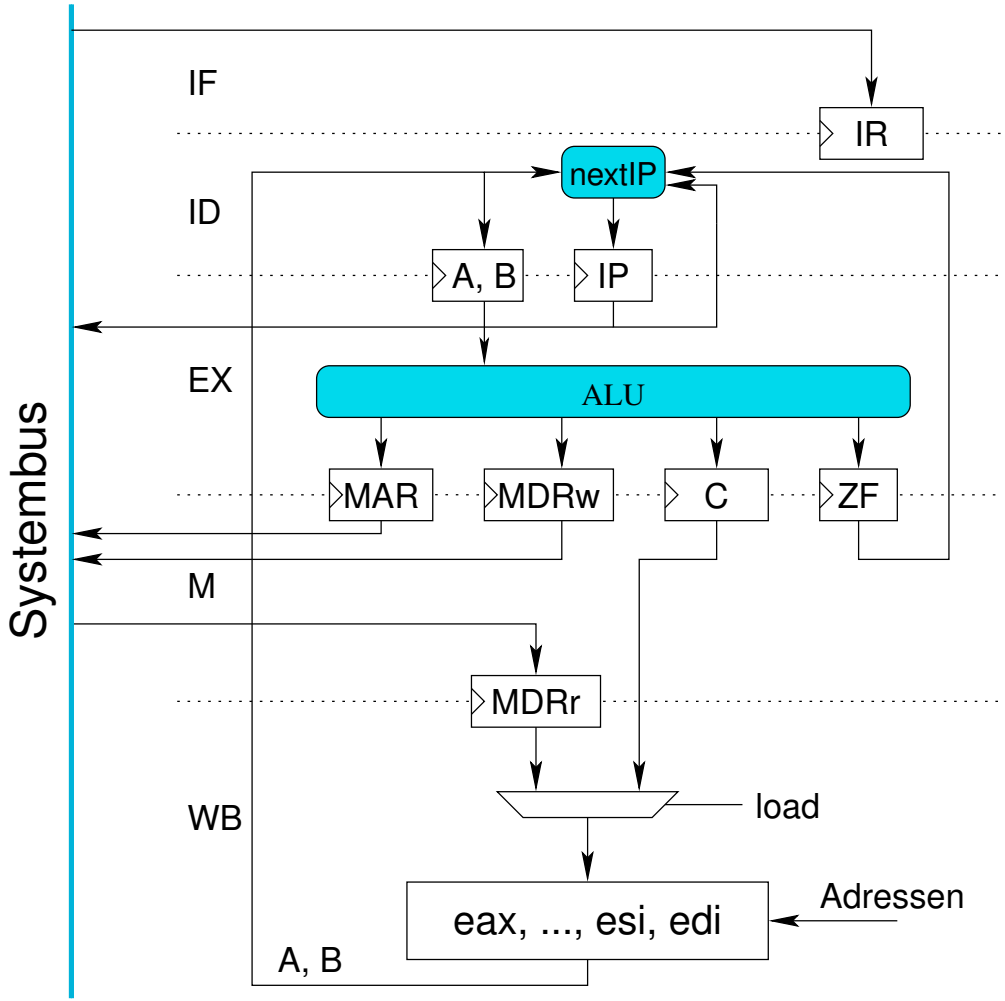
Zeit	0	1	2	3	4	5	6	7	8
IF									
ID									
EX									
MEM									I_2
WB									

Beispiel: Ausführung von **add**

Takt: ⓪

Programm:

```
add edx, ebx
mov [100+esi], edx
```

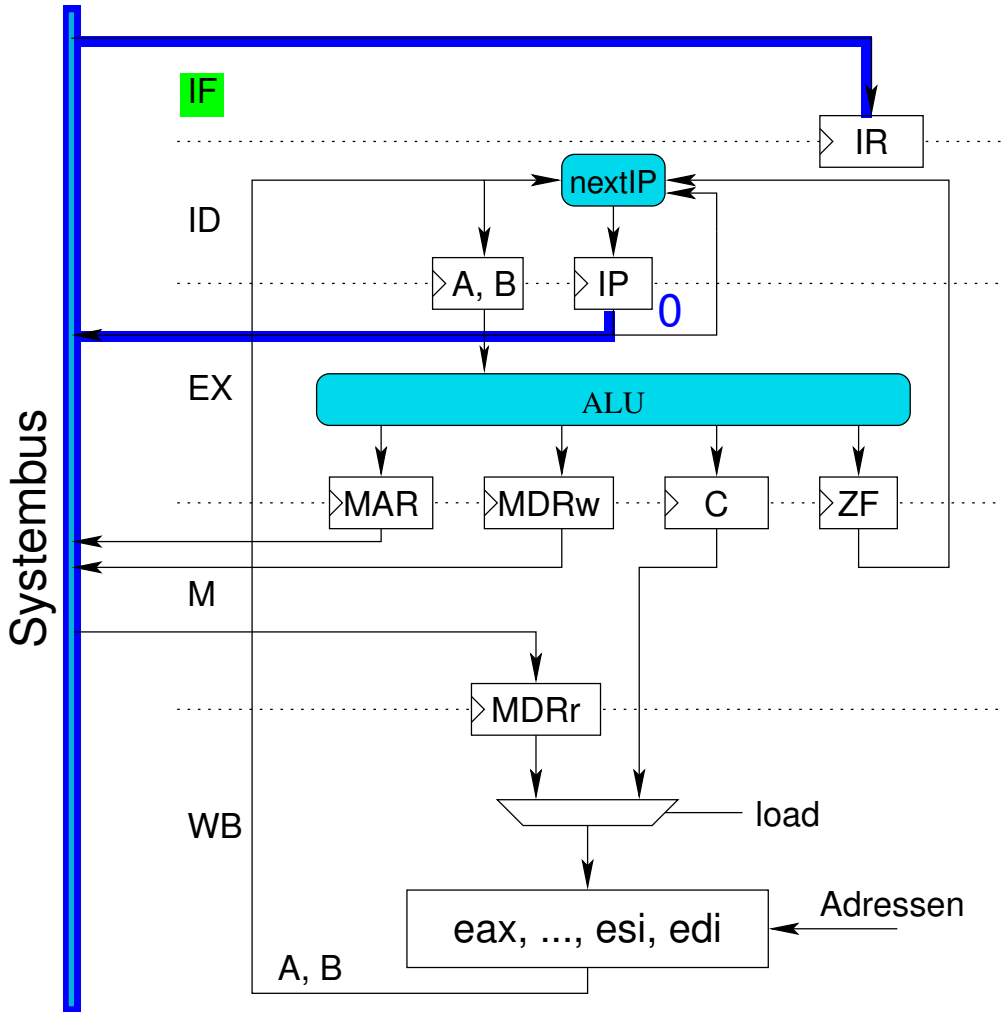


Beispiel: Ausführung von **add** (1)

Takt: ⓪

Programm:

```
add edx, ebx
mov [100+esi], edx
```

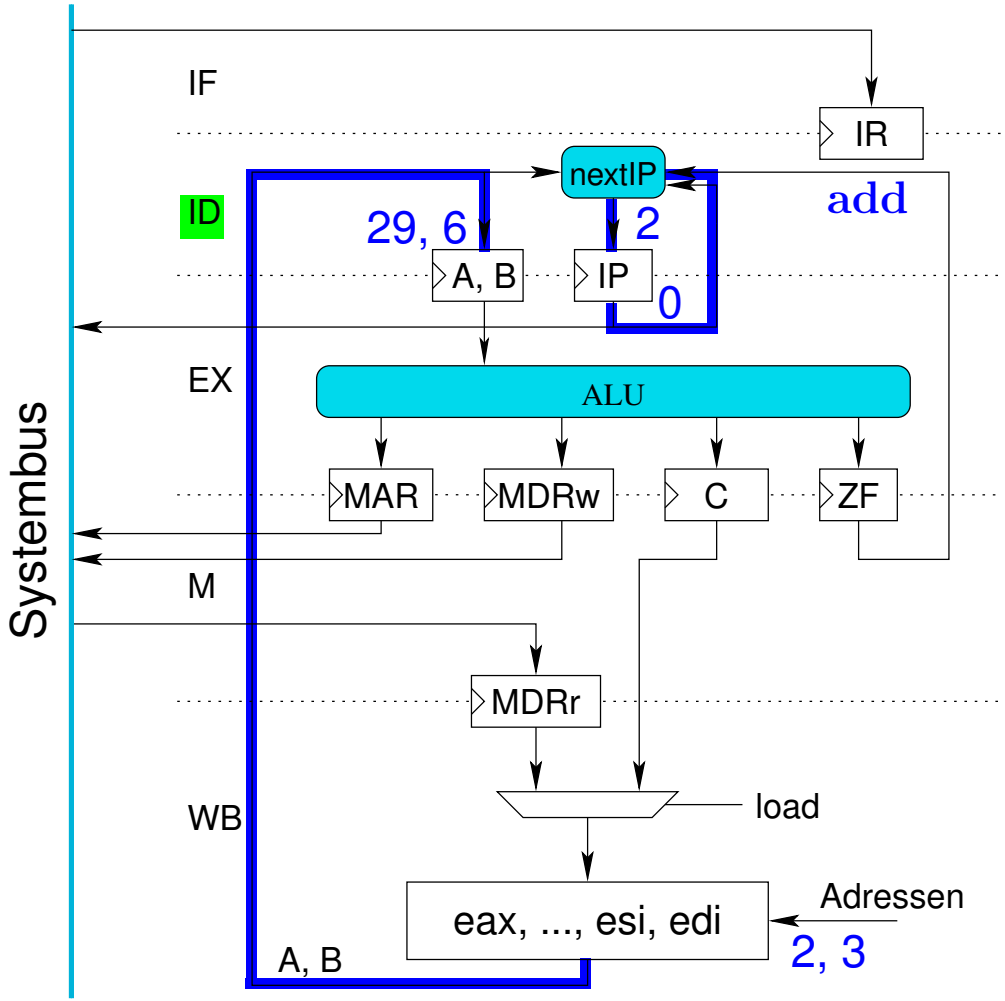


Beispiel: Ausführung von **add** (2)

Takt: ①

Programm:

```
add edx, ebx
mov [100+esi], edx
```

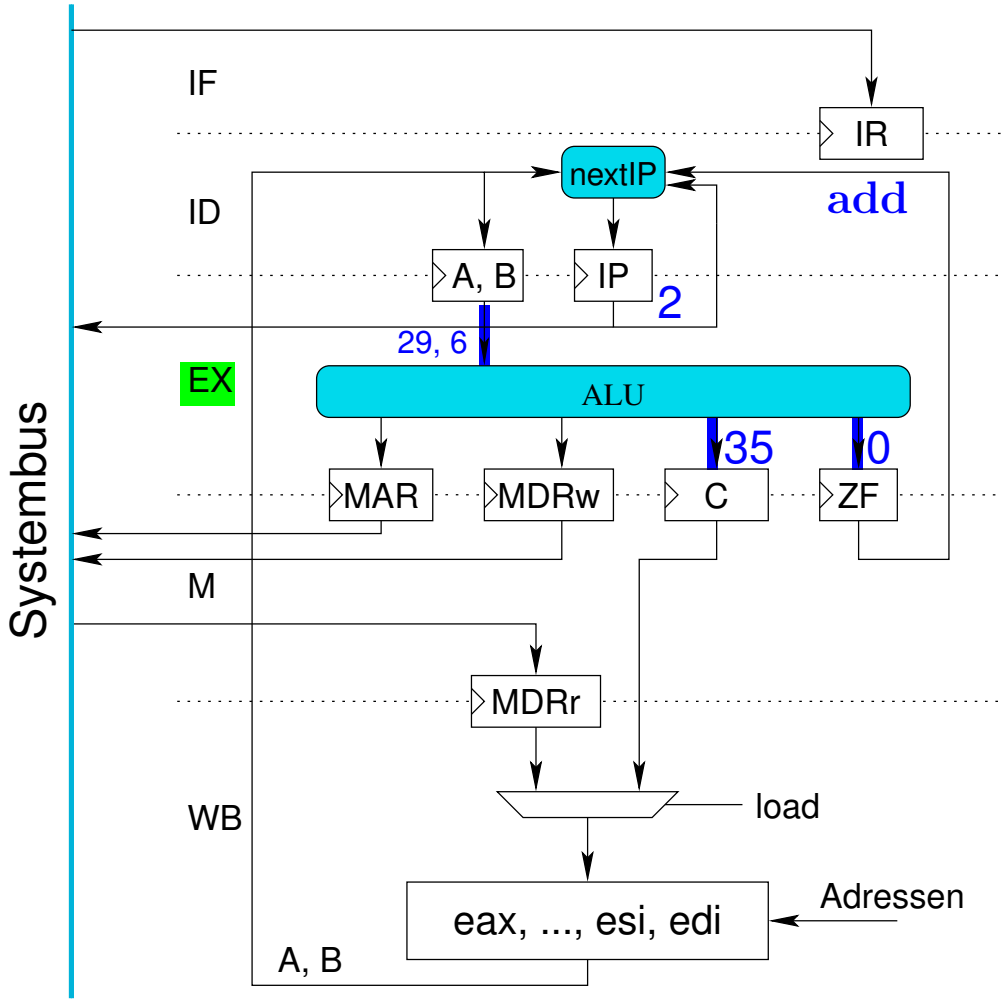


Beispiel: Ausführung von **add** (3)

Takt: ②

Programm:

```
add edx, ebx
mov [100+esi], edx
```

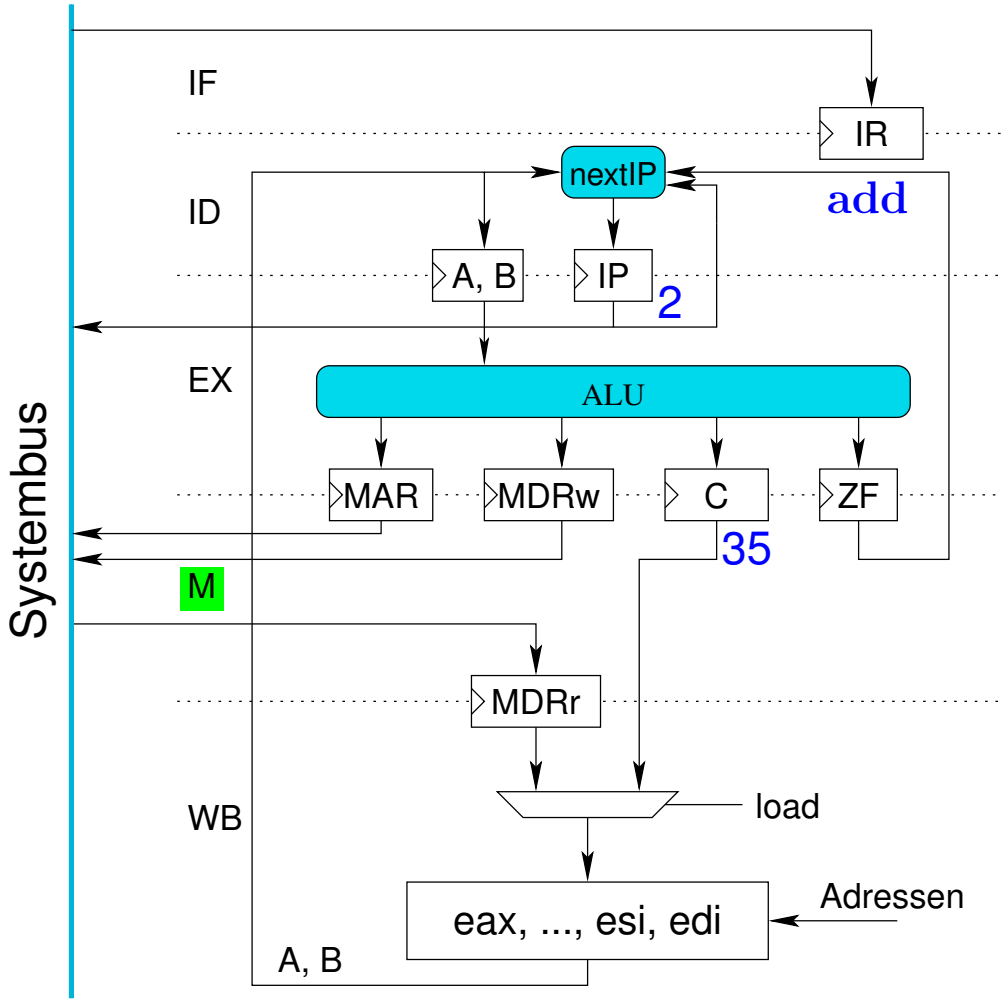


Beispiel: Ausführung von **add** (4)

Takt: ③

Programm:

```
add edx, ebx
mov [100+esi], edx
```

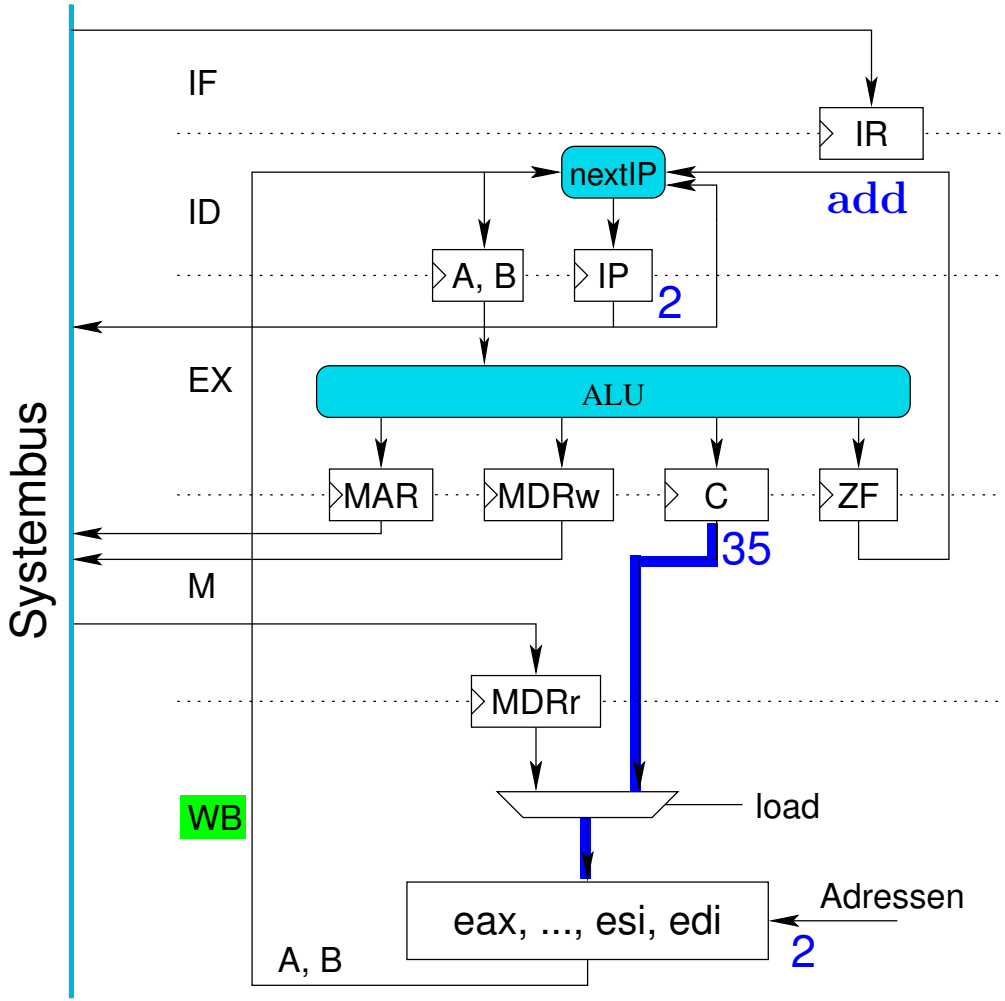


Beispiel: Ausführung von **add** (5)

Takt: ④

Programm:

```
add edx, ebx
mov [100+esi], edx
```

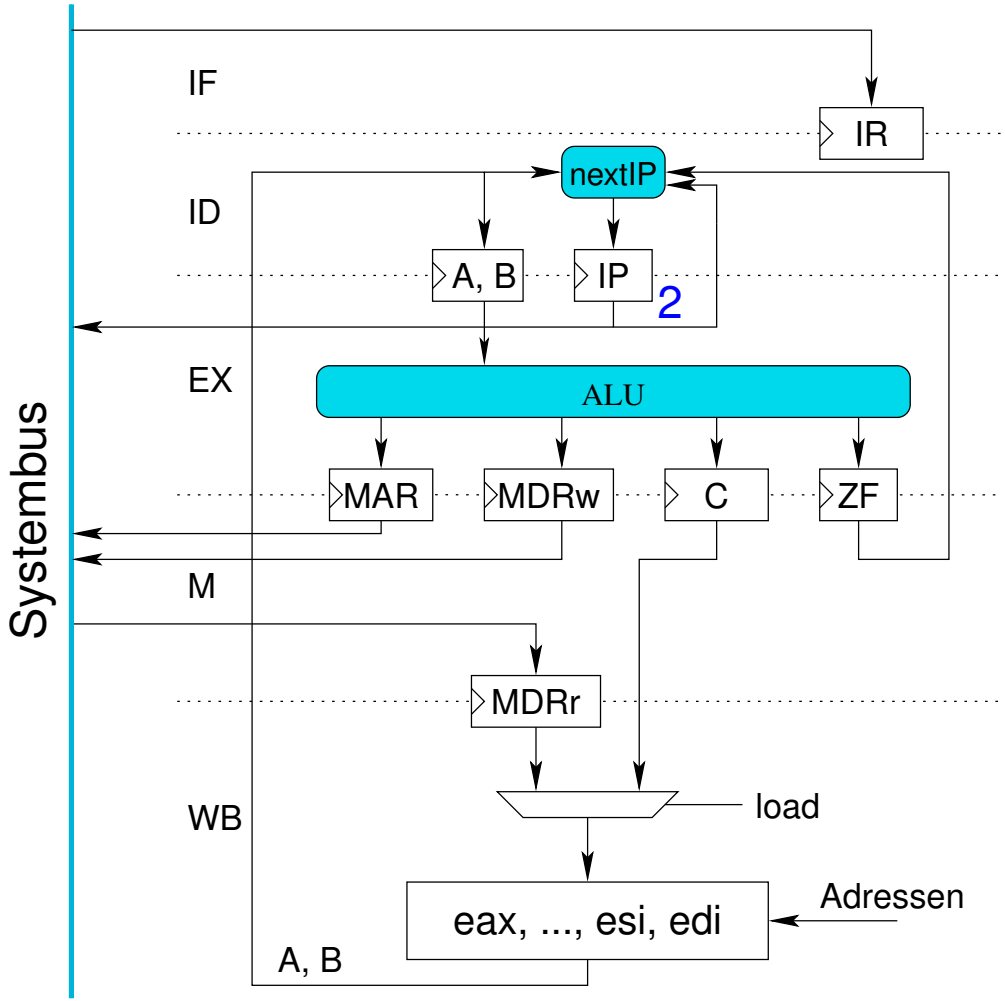


Beispiel: Ausführung von **RMmov**

Takt: ⑤

Programm:

```
add edx, ebx
mov [100+esi], edx
```

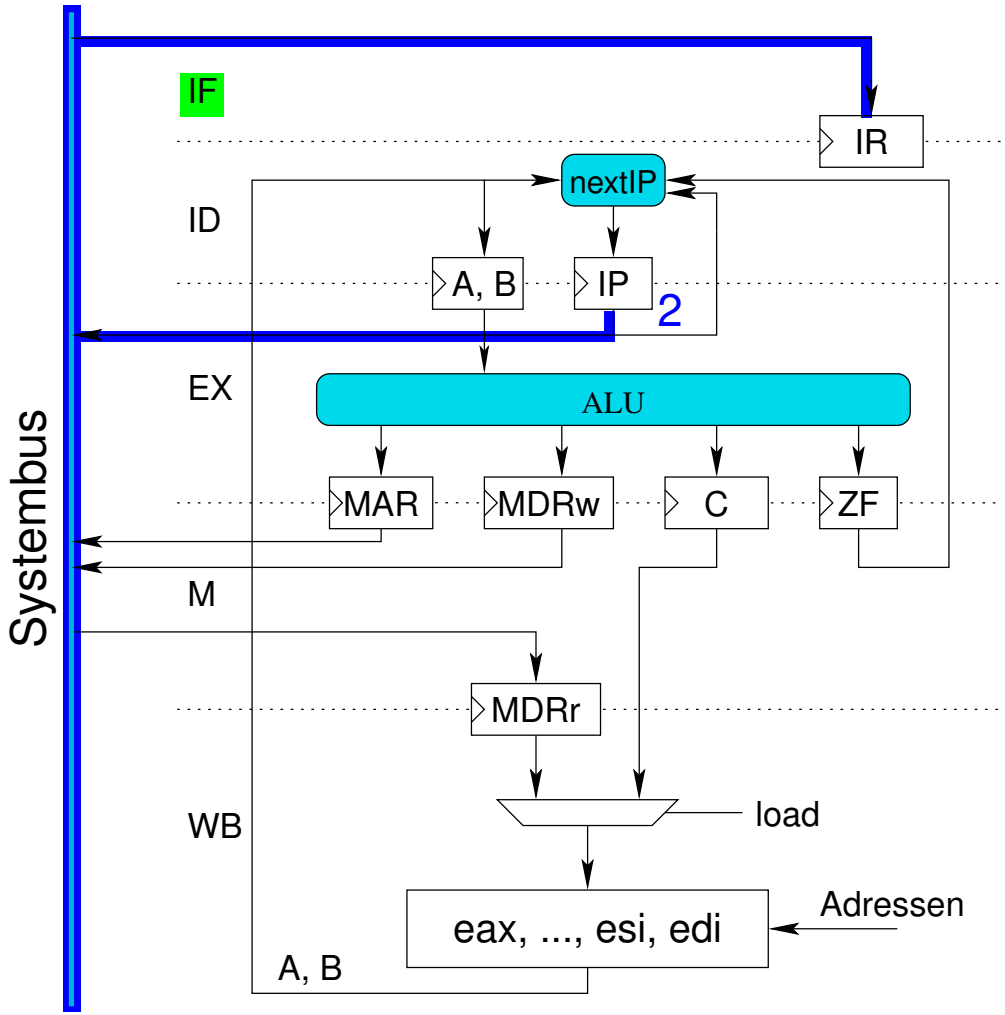


Beispiel: Ausführung von **RMmov** (1)

Takt: ⑤

Programm:

```
add edx, ebx
mov [100+esi], edx
```

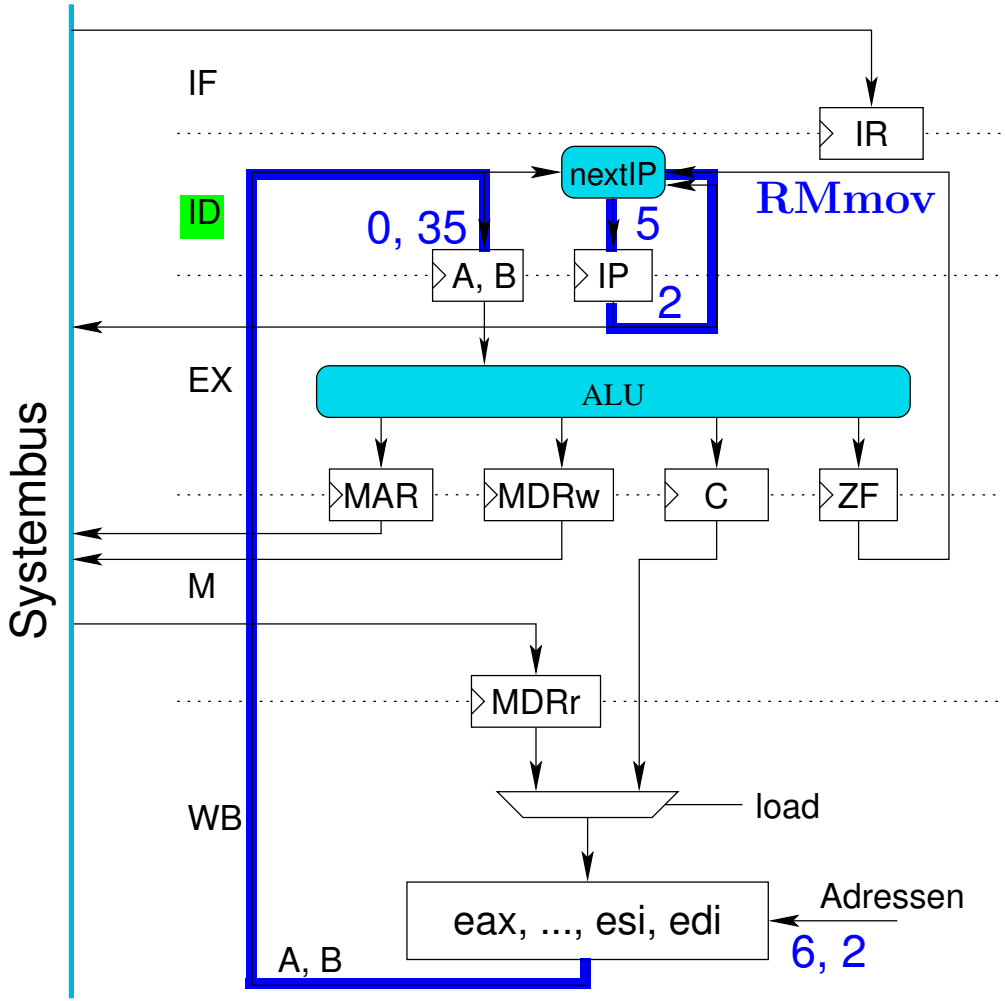


Beispiel: Ausführung von **RMmov** (2)

Takt: ⑥

Programm:

```
add edx, ebx
mov [100+esi], edx
```

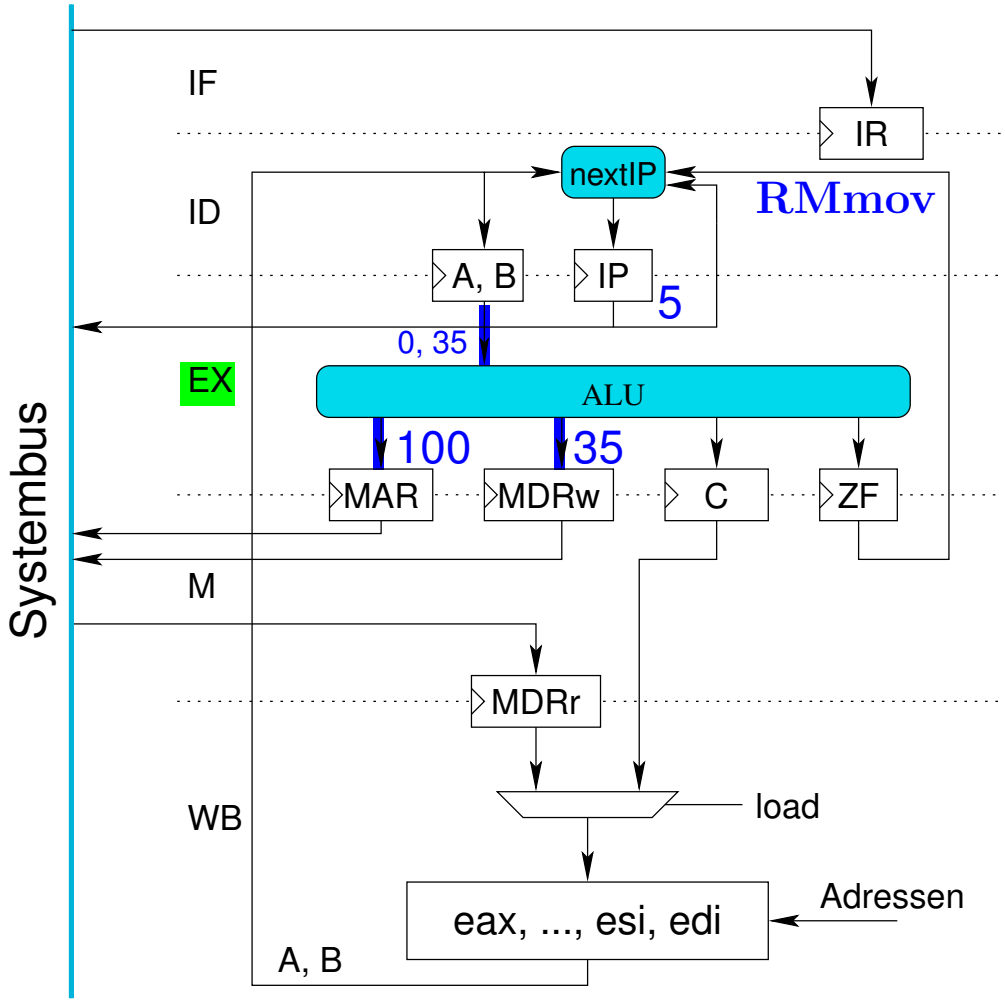


Beispiel: Ausführung von **RMmov** (3)

Takt: ⑦

Programm:

```
add edx, ebx
mov [100+esi], edx
```

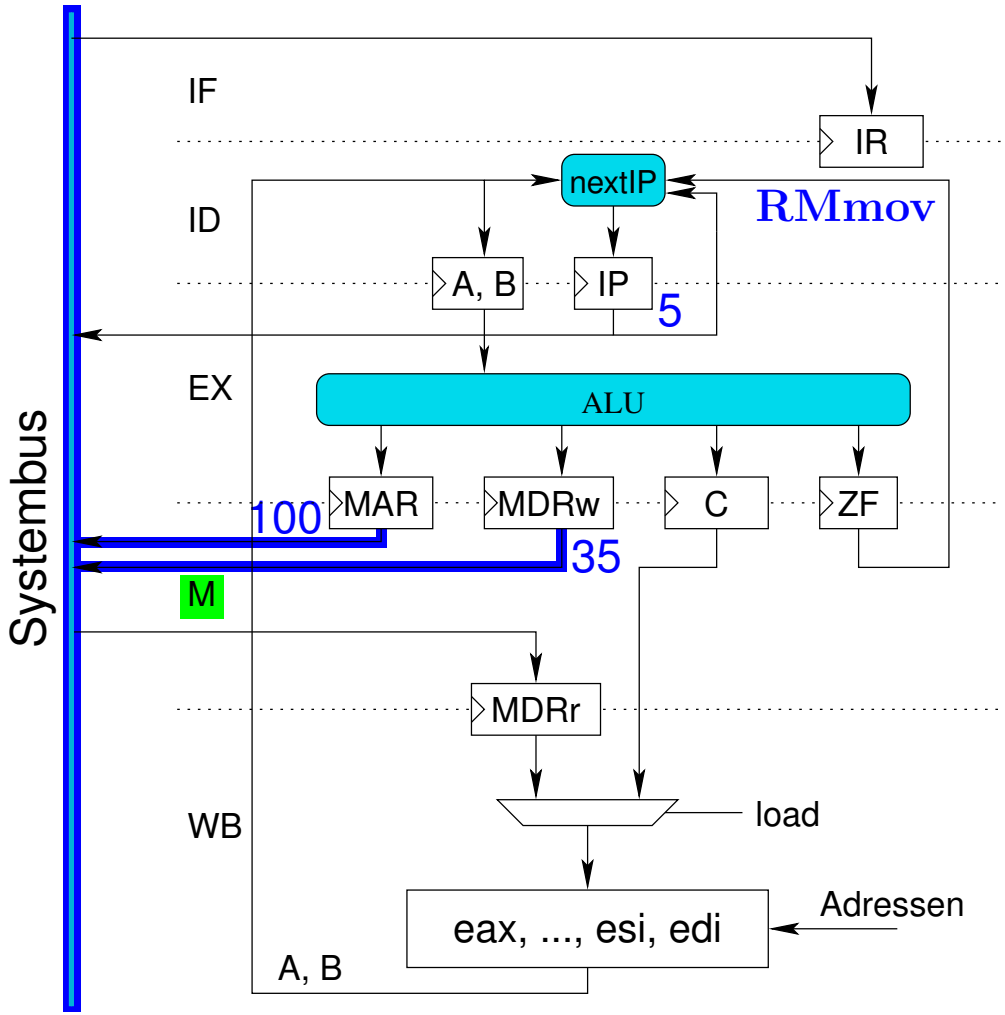


Beispiel: Ausführung von **RMmov** (4)

Takt: ⑧

Programm:

```
add edx, ebx
mov [100+esi], edx
```

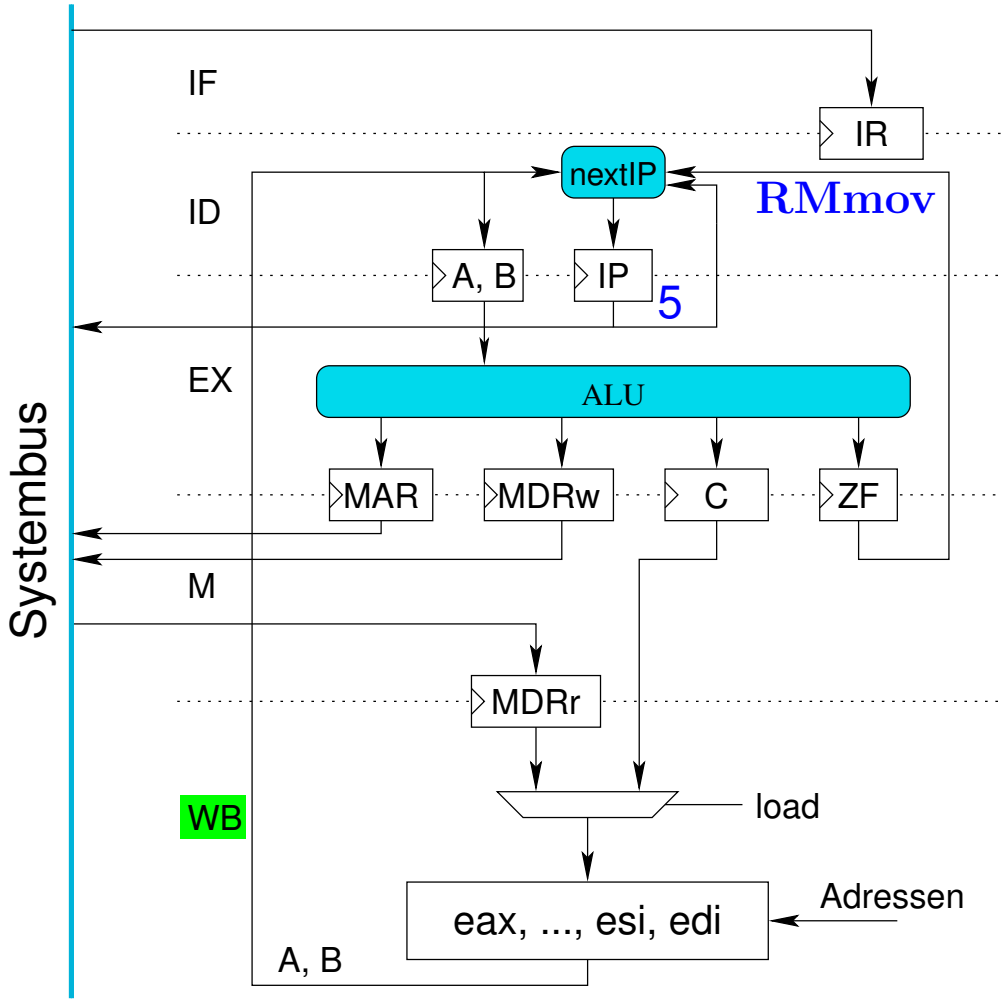


Beispiel: Ausführung von **RMmov** (5)

Takt: ⑨

Programm:

```
add edx, ebx
mov [100+esi], edx
```



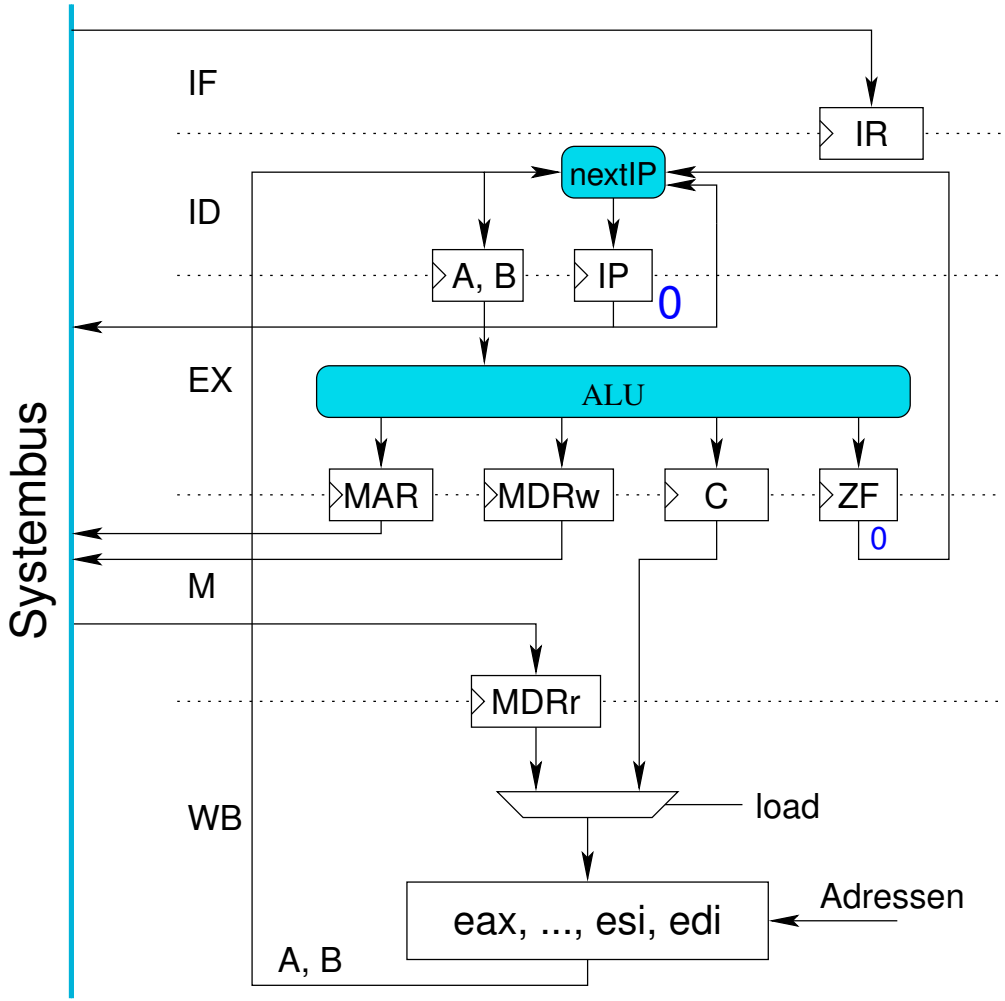
Beispiel: Ausführung von **jnz**

Takt: ⓪

Programm:

```
jnz l
```

Die Distanz sei 10



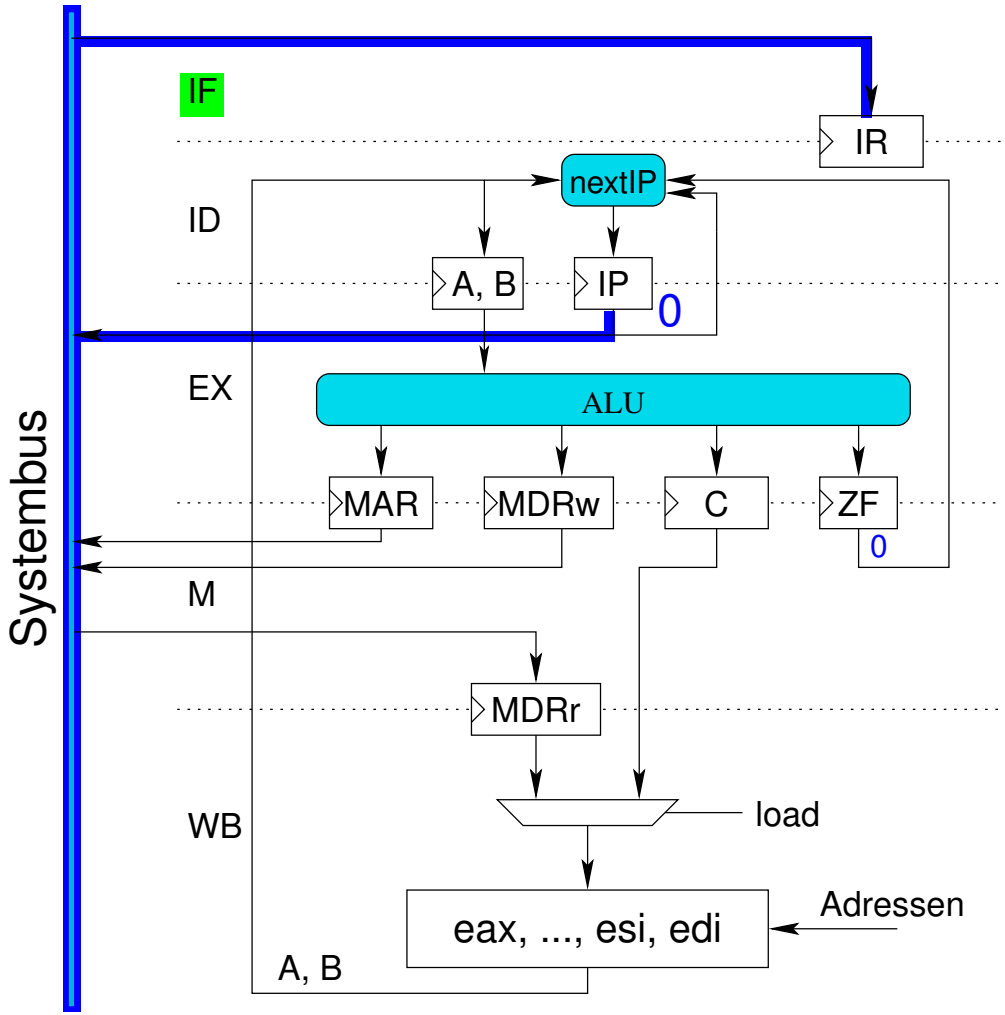
Beispiel: Ausführung von **jnz** (1)

Takt: ⓪

Programm:

```
jnz l
```

Die Distanz sei 10



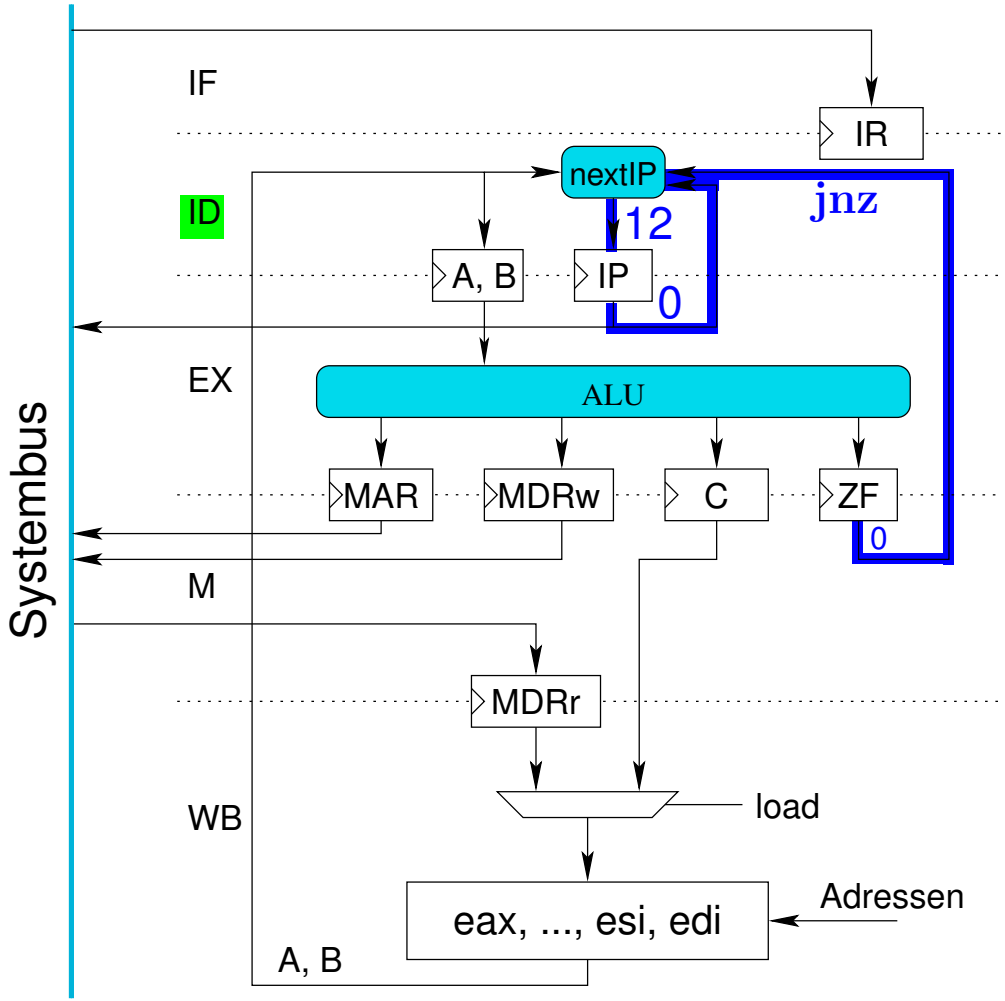
Beispiel: Ausführung von **jnz** (2)

Takt: ①

Programm:

```
jnz l
```

Die Distanz sei 10



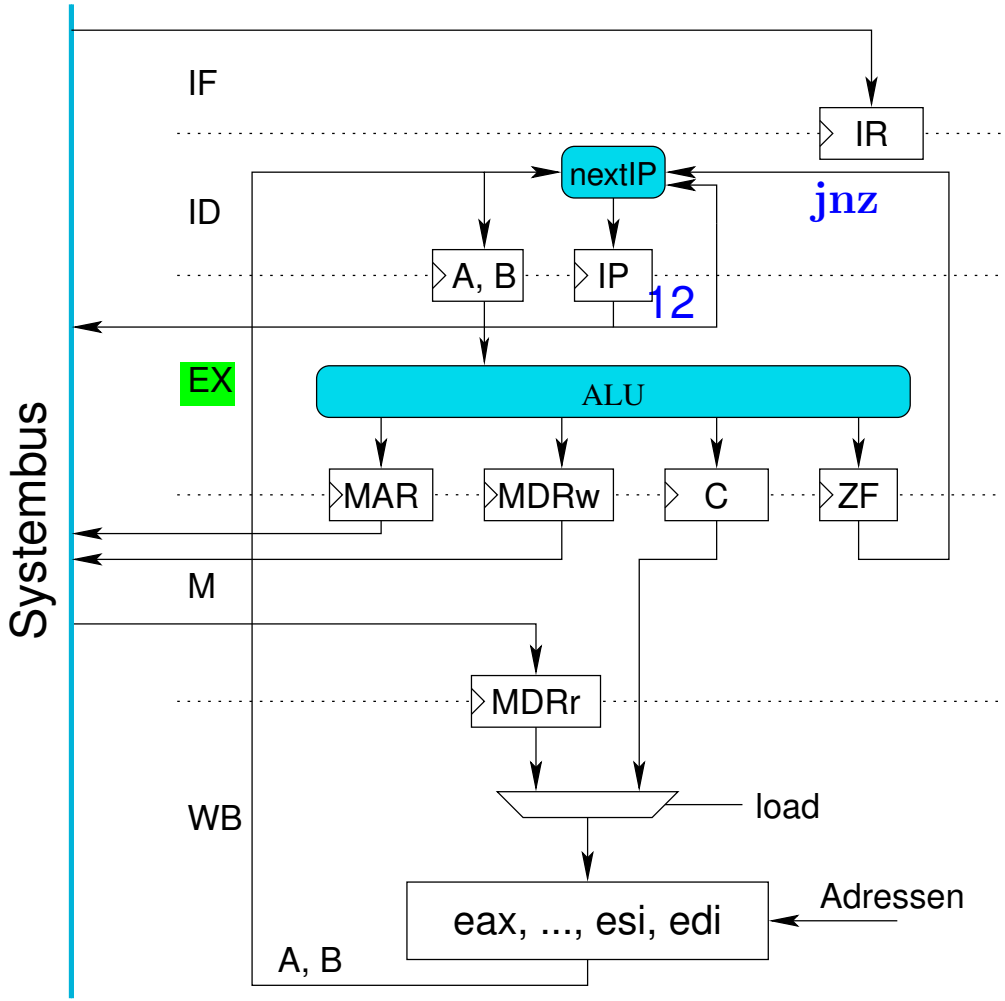
Beispiel: Ausführung von **jnz** (3)

Takt: ②

Programm:

```
jnz l
```

Die Distanz sei 10



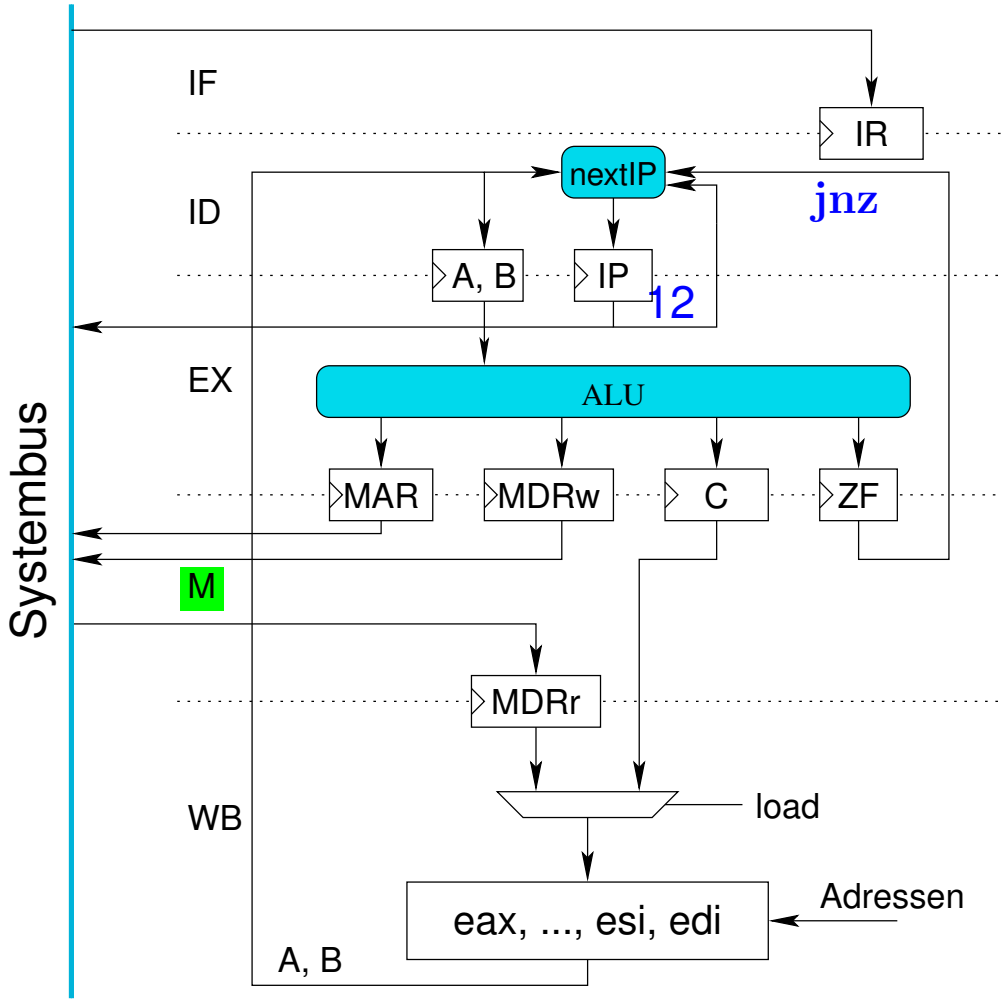
Beispiel: Ausführung von **jnz** (4)

Takt: ③

Programm:

```
jnz l
```

Die Distanz sei 10



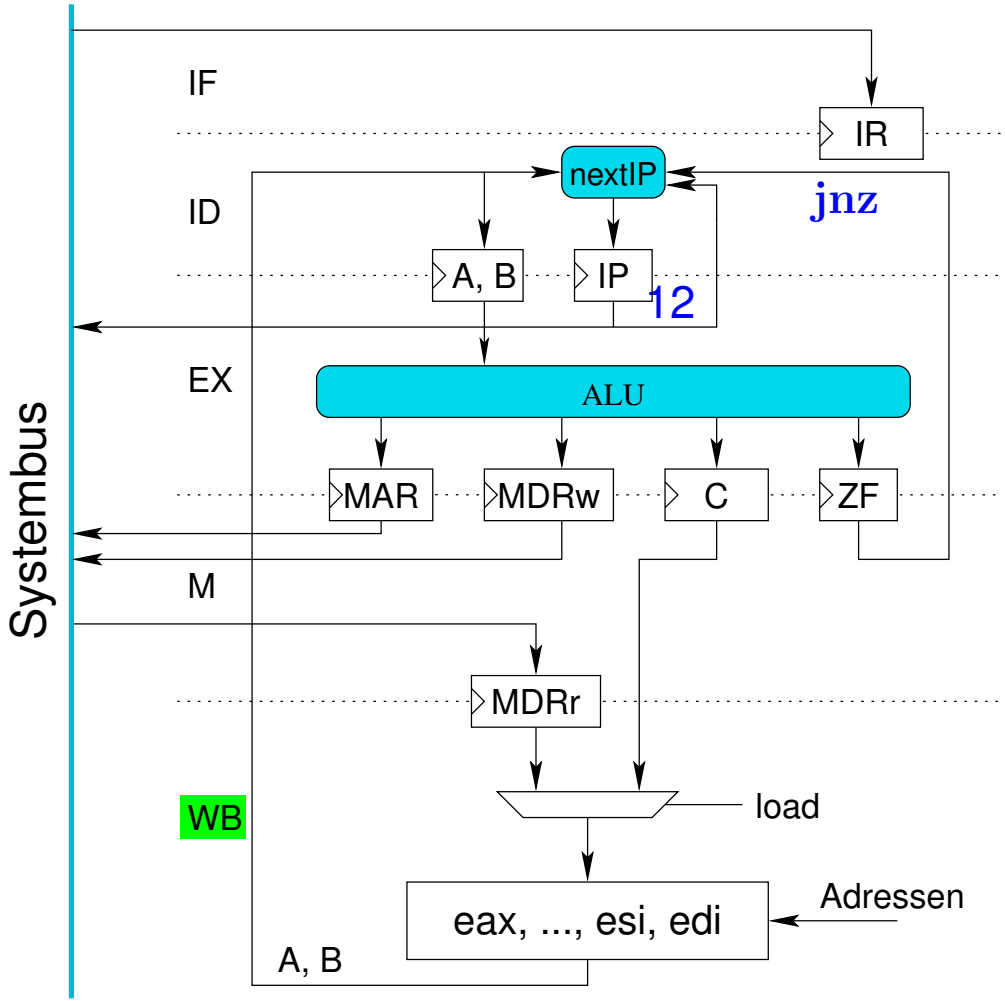
Beispiel: Ausführung von **jnz** (5)

Takt: ④

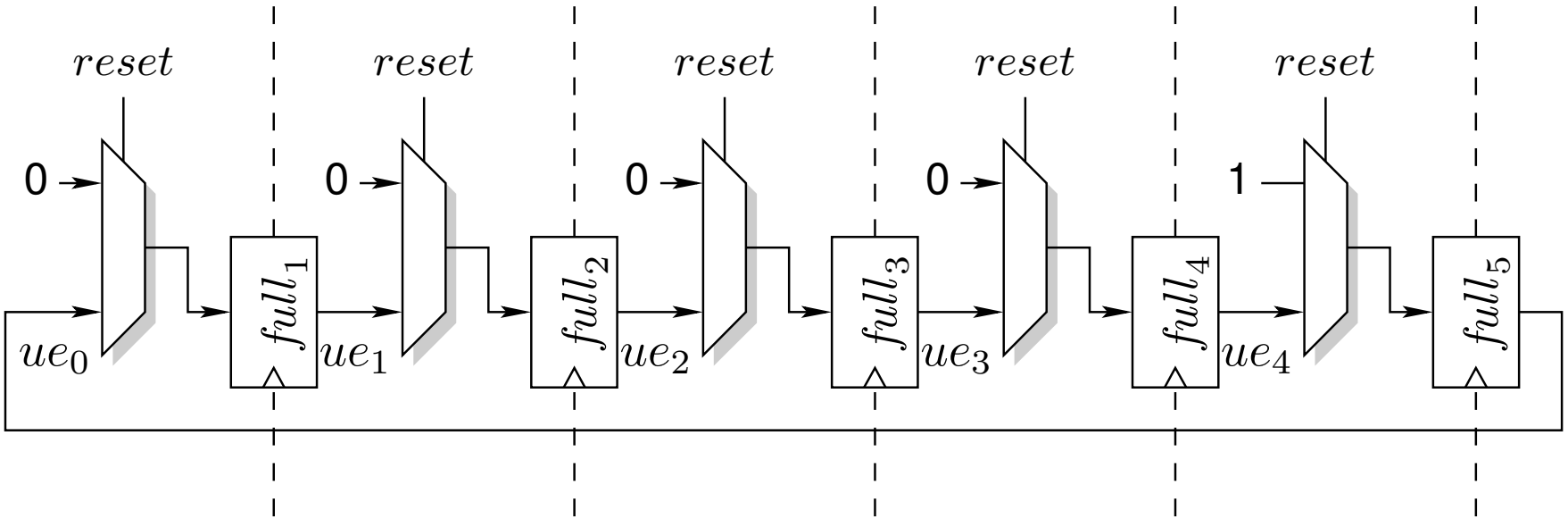
Programm:

```
jnz l
```

Die Distanz sei 10

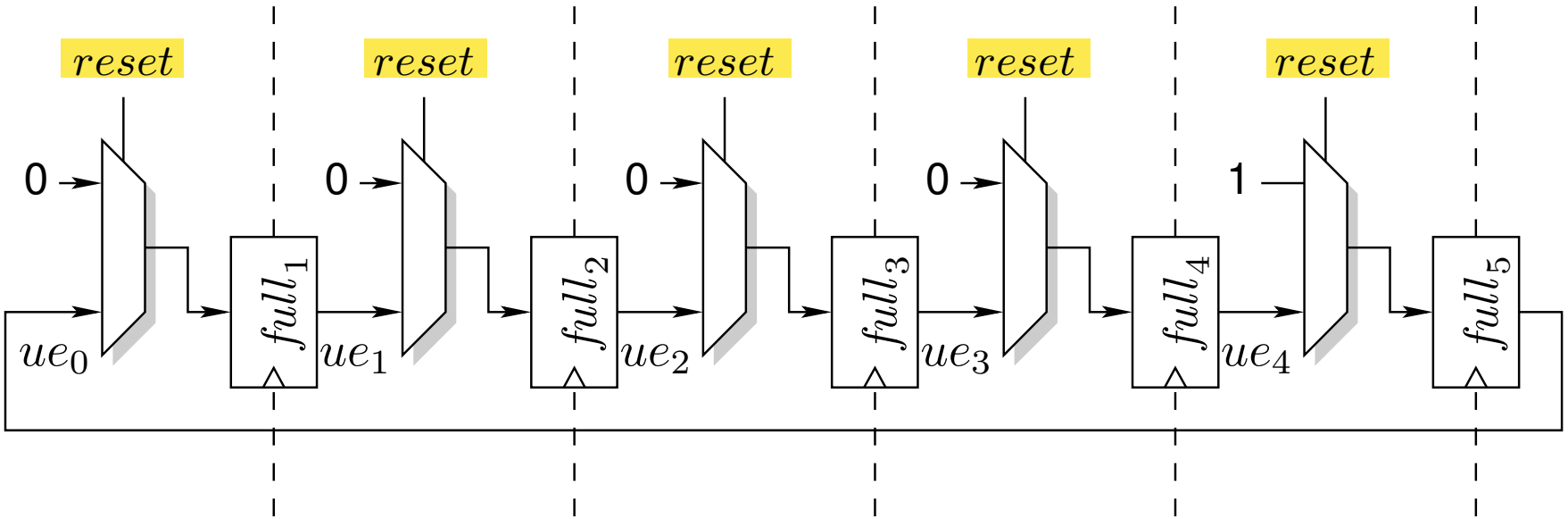


Kontrolle für die sequentielle Implementierung



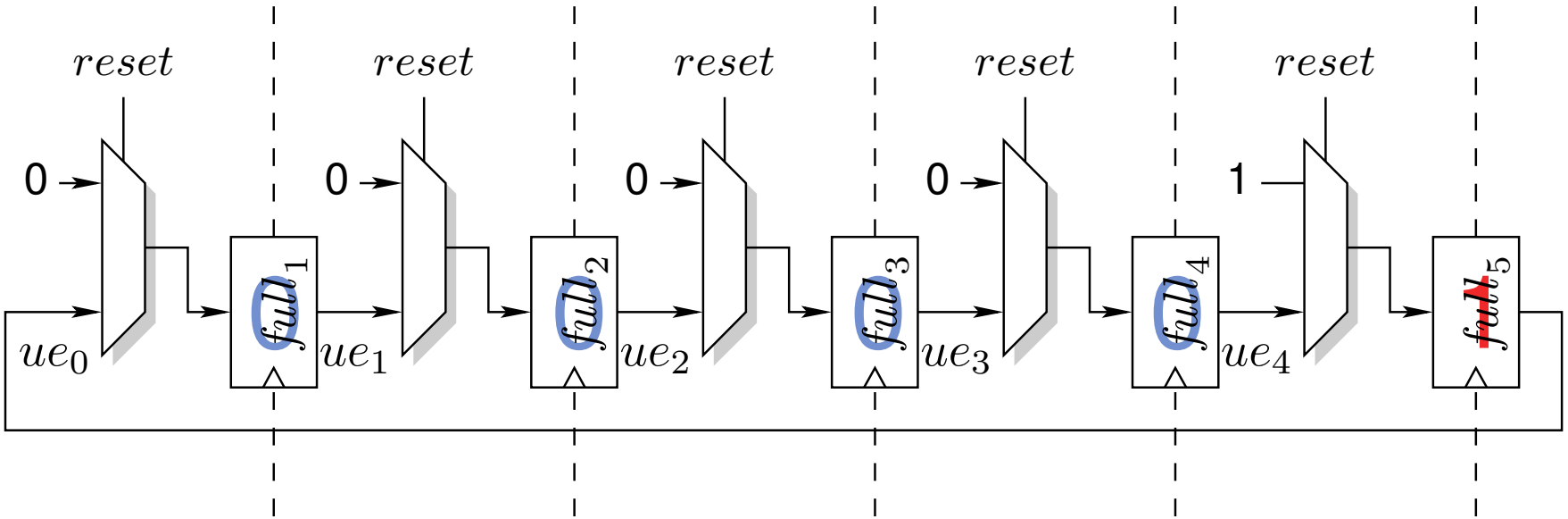
- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



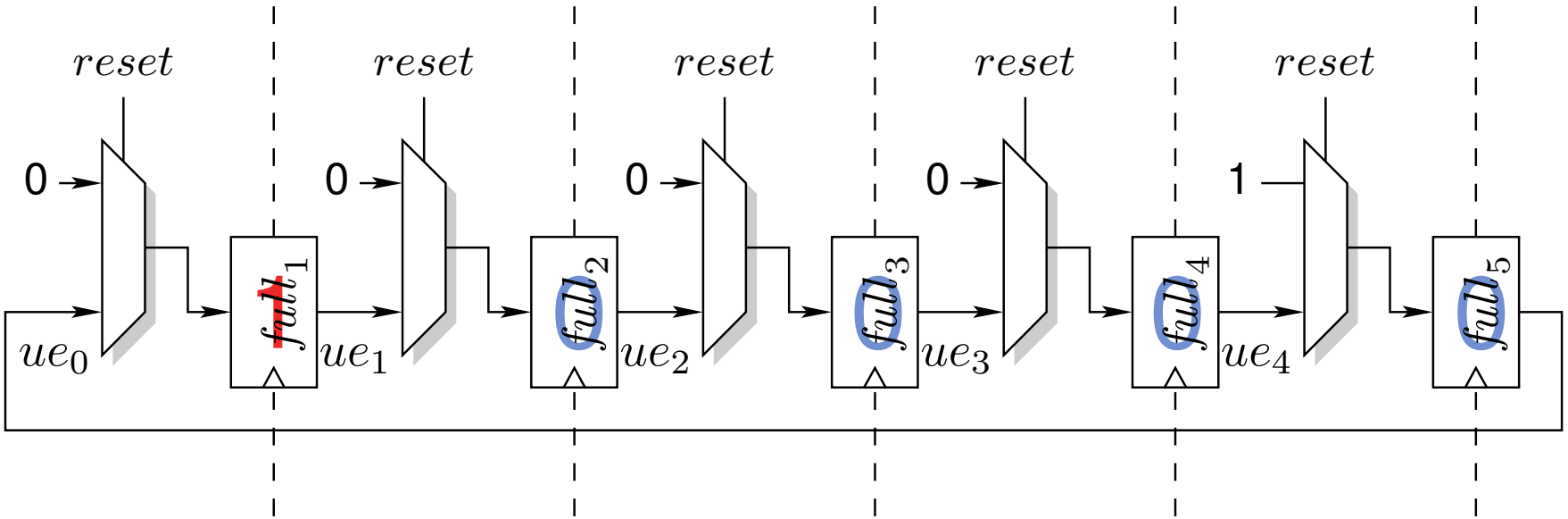
- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



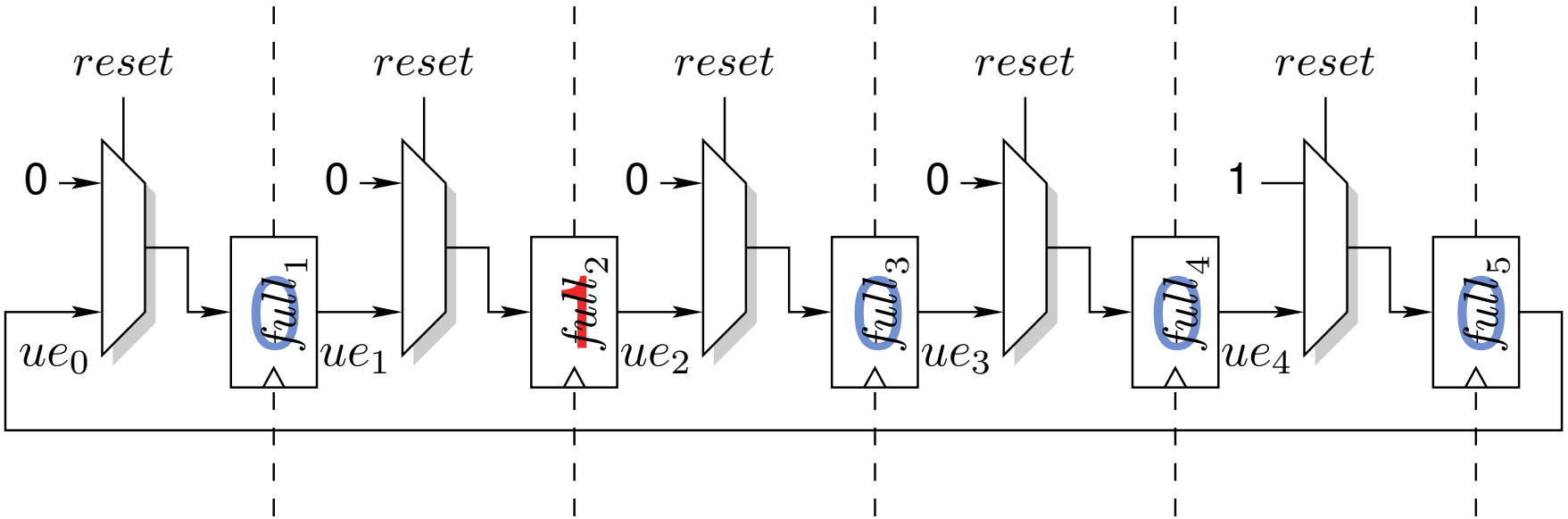
- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



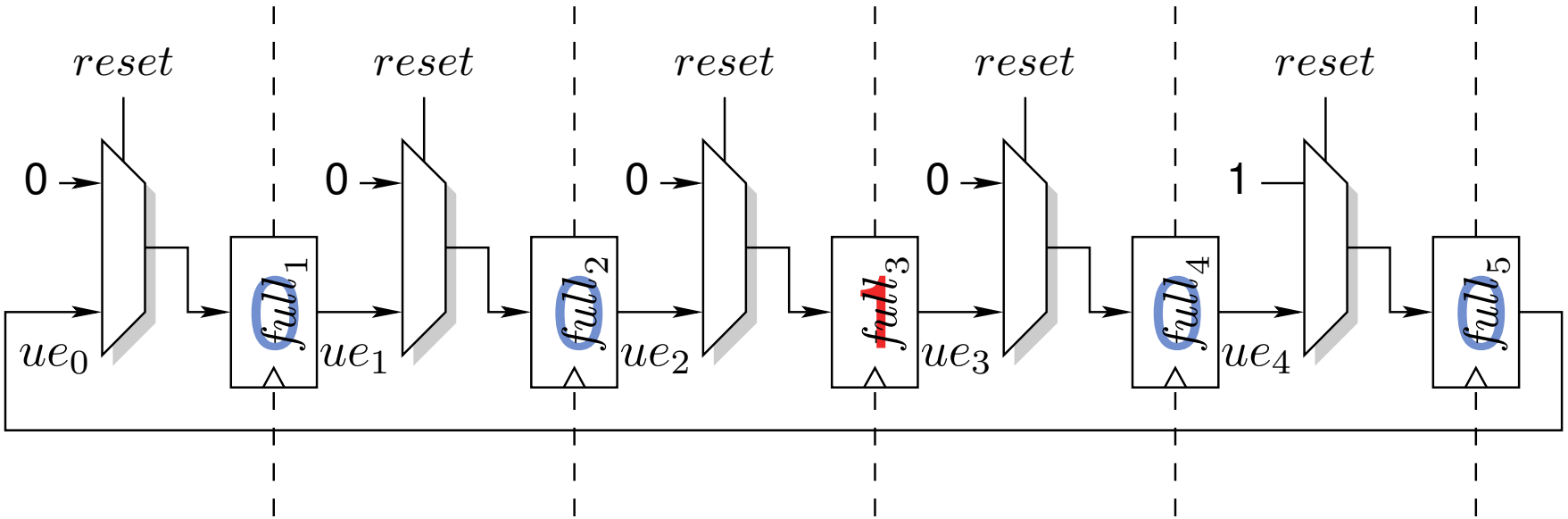
- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



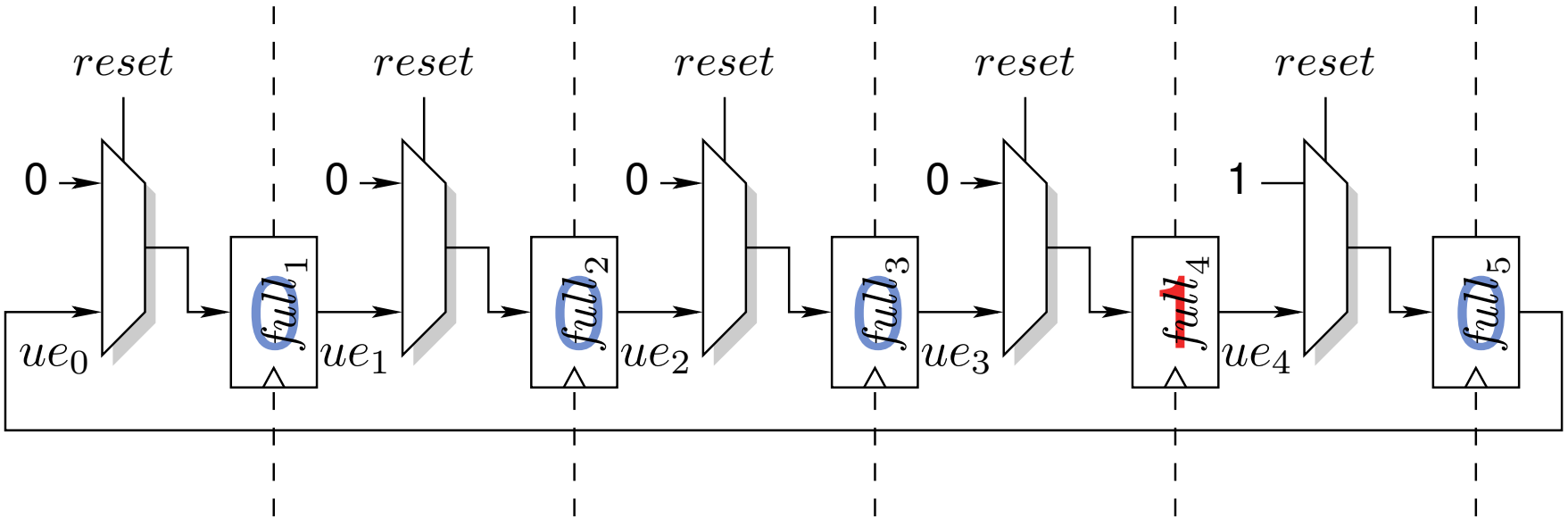
- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



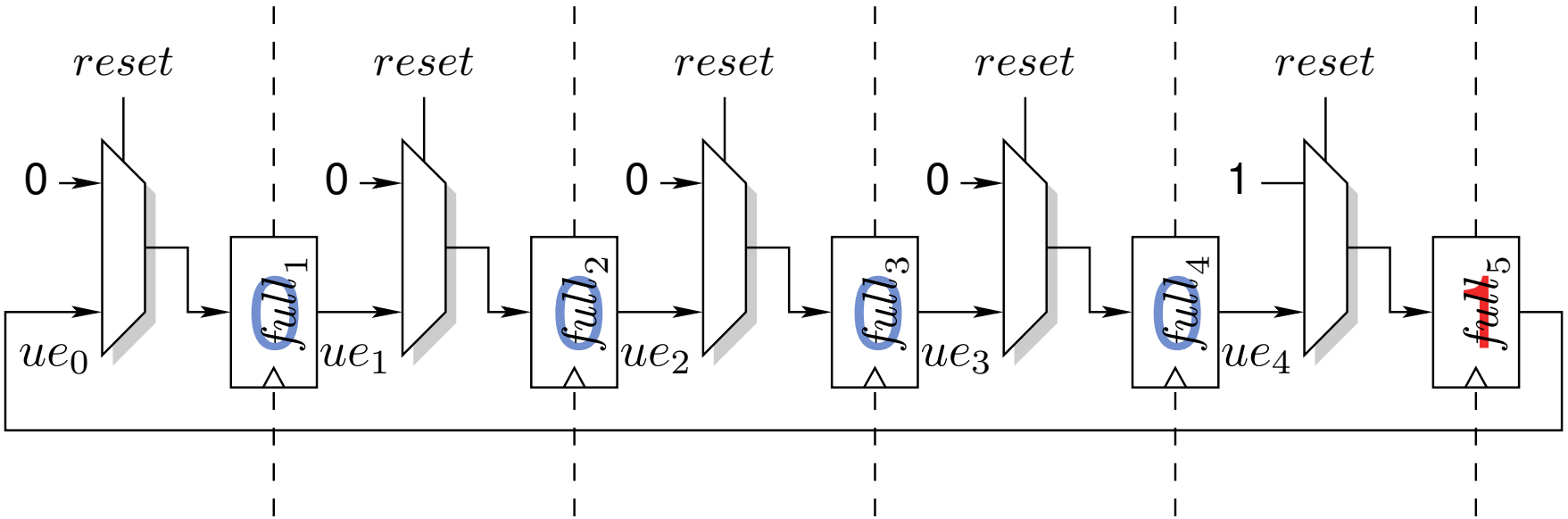
- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



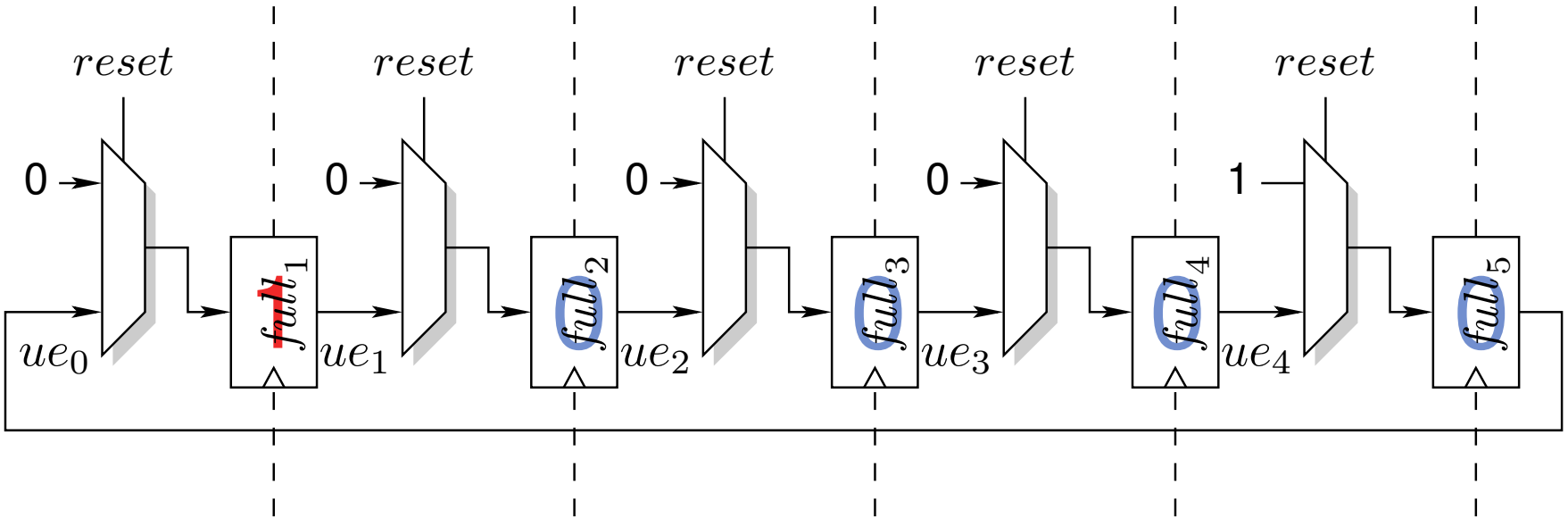
- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle für die sequentielle Implementierung



- Ähnlich wie Shift-Register
- Ein “Full-Bit” pro Stufe
  $full_1$: Instruction Fetch, $full_2$: Decode, $full_3$: Execute, usw.
- Aktiviert die Clock-Enables der Register der jeweils *nächsten* Stufe

Kontrolle in Verilog

```
reg [5:1]   full ;
wire [4:0]  ue={ full [4:1],   full [5]  };

always @(posedge clk)
    full ={ ue[4],  ue[3],  ue[2],  ue[1],  ue[0]  };

 initial    full ='b10000;
```

Die Hardware der einzelnen Stufen

- IF: Instruktion vom Bus lesen
- ID: Dekodierung, nextIP
- EX: ALU, Berechnung der Flags
- MEM: Interface zum RAM
- WB: Ein Mux und das Register File

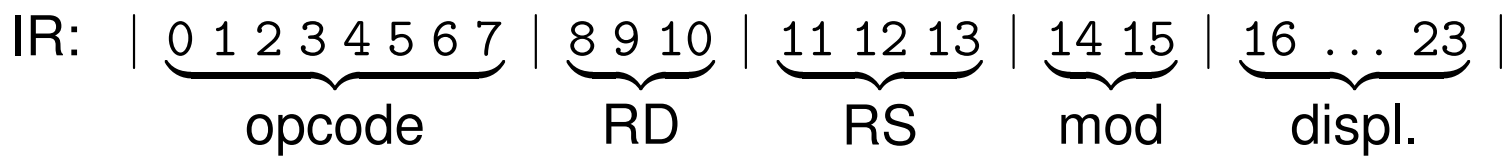
Instruction Fetch

```
reg [31:0] IR;

always @(posedge clk)
   if (ue[0]) IR=bus_in;
```

Dekodierung (Teil 1)

Wir verwenden maximal 24 Bits des Instruktionsworts:




```
wire [7:0]  opcode=IR[7:0];
wire [1:0]  mod=IR[15:14];
wire [4:0]  RD=IR[10:8];
wire [4:0]  RS=IR[13:11];

wire [31:0]  distance={ { 24 { IR[15] } }, IR[15:8] };
wire [31:0]  displacement={ { 24 { IR[23] } }, IR[23:16] };
```

Dekodierung (Teil 2)

```
wire load=opcode==’h8b && mod==1;
wire move=opcode==’h89 && mod==3;
wire store=opcode==’h89 && mod==1;
wire add=opcode==’h01;
wire sub=opcode==’h29;
wire jnz=opcode==’h75;

wire memory=load || store;
wire aluop=add || sub;

wire [4:0]  Aad=memory?6:RD,
            Bad=RS;
```

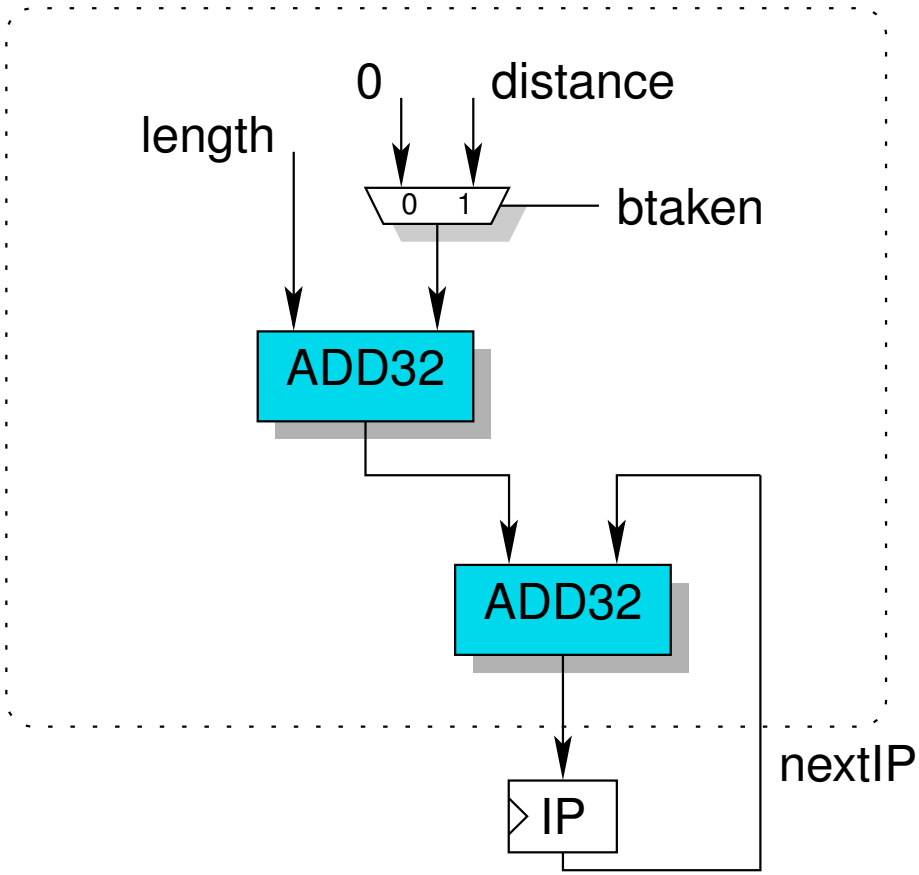
Zwei Fälle:

1. Der **jnz**-Sprungbefehl wird abgearbeitet und **ZF** ist *nicht* gesetzt („Branch taken“).

   In diesem Fall wird der IP um die Länge der Instruktion und zusätzlich um den Wert des Distance-Feldes erhöht.

2. Sonst: IP um die Länge der Instruktion erhöhen

Implementierung von nextIP



nextIP in Verilog

```
wire btaken=jnz && !ZF;

wire [1:0] length=                    memory?3:
                  (aluop || move || jnz)?2:
                                          1;

always @(posedge clk)
  if(ue[1]) begin
    IP = IP+length;
    if(btaken) IP = IP+distance;
  end
```

Implementierung der ALU

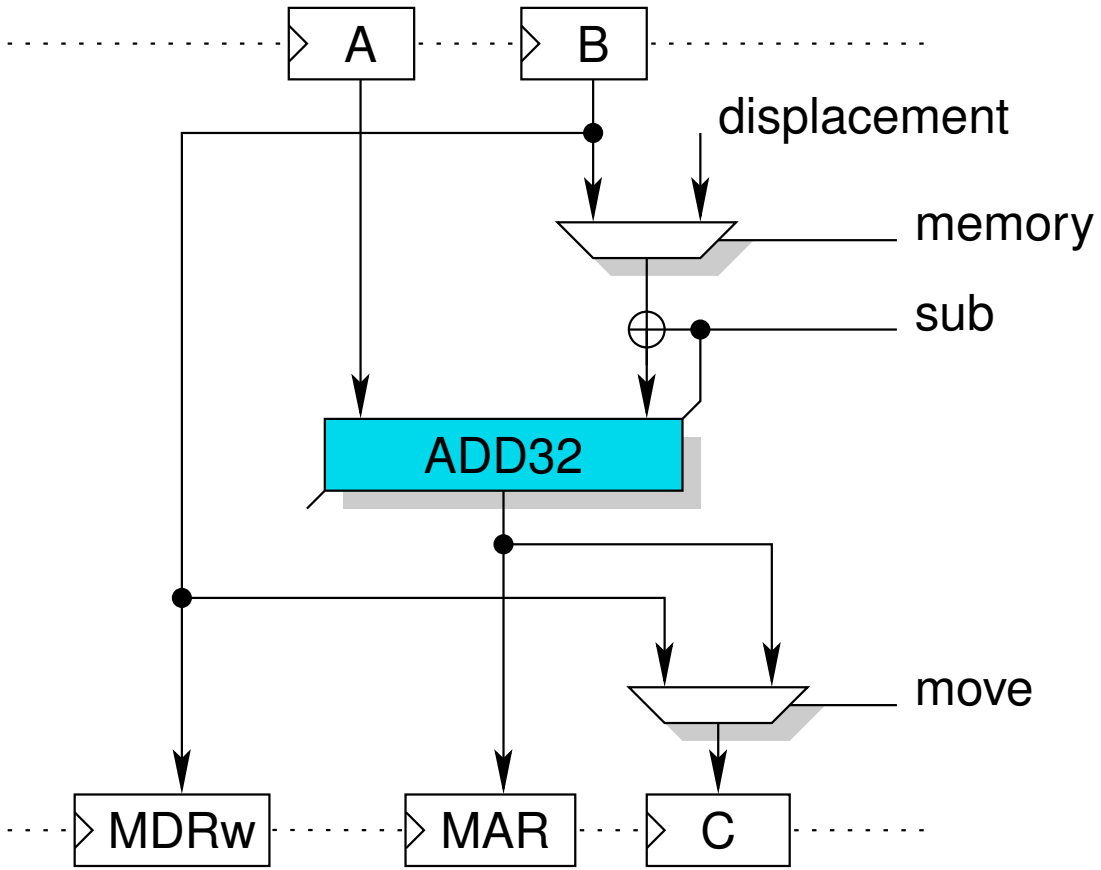
Das ALU-Modul wird auf zweierlei Weise verwendet:

1. Berechnet das Ergebnis der ALU-Befehle **add** und **sub**. Das Ergebnis wird im Register C abgelegt.

2. Berechnet die Effective Address (ea) der Speicher-Befehle **MRmov** und **RMmov**. Die Adresse wird im Register MAR (Memory Address Register) abgelegt.

✔ Das spart einen Addierer.

Implementierung der ALU



Die ALU in Verilog

```
wire [31:0]  ALU_op2=memory?displacement:sub?~B:B;
wire [31:0]  ALUout=A+ALU_op2+sub;

always @(posedge clk)
  if (ue[2])  begin
    MAR = ALUout;
    C = move?B:ALUout;
    MDRw = B;
    if (aluop) ZF = (ALUout==0);
  end
```

Die Memory-Stufe

```
reg [31:0] MDRr;

always @(posedge clk)
   if (ue[3] && load) MDRr=bus_in;

assign bus_WE=ue[3] && store;
assign bus_A=ue[3]?MAR:ue[0]?IP:0;
assign bus_RE=ue[0] || (ue[3] && load);
```

Die Writeback-Stufe

```
reg [31:0]  R[7:0];

assign Aop=R[Aad];
assign Bop=R[Bad];

always @(posedge clk)
   if (ue[4])
      if (aluop  ||  move || load)
        R[load?RS:RD]=load?MDRr:C;
```

Erweiterung: Vergleiche

Wir hätten gerne Befehle für

```
if(a<b) { ... }
```

Das hängt offensichtlich vom Vorzeichen ab:

mit Vorzeichen	ohne Vorzeichen
$0 > -7$	$0 < 9$
$[\![0000]\!] > [\![1001]\!]$	$\langle 0000 \rangle < \langle 1001 \rangle$

Vergleiche von Zahlen ohne Vorzeichen

Vorzeichenlose Binärzahlen:

$$\langle a \rangle < \langle b \rangle \iff \langle a \rangle - \langle b \rangle < 0$$

Erinnerung: $-b = (\neg b) + 1$
Wir bekommen das „$+1$“ indem wir $c_0 = 1$ setzen.

Vergleiche von Zahlen ohne Vorzeichen

Vorzeichenlose Binärzahlen:

$$\langle a \rangle < \langle b \rangle \iff \langle a \rangle - \langle b \rangle < 0$$

Erinnerung: $-b = (\neg b) + 1$
Wir bekommen das „$+1$“ indem wir $c_0 = 1$ setzen.

Wir rechnen mit einem zusätzlichen Bit („zero extension“):

$$\begin{array}{rccccc} & 0 & a_{n-1} & \dots & a_1 & a_0 \\ + & 1 & \neg b_{n-1} & \dots & \neg b_1 & \neg b_0 \\ & c_n & c_{n-1} & \dots & c_1 & 1 \\ \hline = & s_n & s_{n-1} & \dots & s_1 & s_0 \end{array}$$

(Übertragsbits)

(Summe)

Vergleiche von Zahlen ohne Vorzeichen

Vorzeichenlose Binärzahlen:

$$\langle a \rangle < \langle b \rangle \iff \langle a \rangle - \langle b \rangle < 0$$

Erinnerung: $-b = (\neg b) + 1$
Wir bekommen das „$+1$“ indem wir $c_0 = 1$ setzen.

Wir rechnen mit einem zusätzlichen Bit („zero extension“):

$$\begin{array}{rccccc} & 0 & a_{n-1} & \dots & a_1 & a_0 \\ + & 1 & \neg b_{n-1} & \dots & \neg b_1 & \neg b_0 \\ & c_n & c_{n-1} & \dots & c_1 & 1 \\ \hline = & s_n & s_{n-1} & \dots & s_1 & s_0 \end{array}$$

(Übertragsbits)
(Summe)

Also: $\langle a \rangle - \langle b \rangle < 0 \iff s_n$

Vergleiche von Zahlen ohne Vorzeichen

Vorzeichenlose Binärzahlen:

$$\langle a \rangle < \langle b \rangle \iff \langle a \rangle - \langle b \rangle < 0$$

Erinnerung: $-b = (\neg b) + 1$
Wir bekommen das „$+1$“ indem wir $c_0 = 1$ setzen.

Wir rechnen mit einem zusätzlichen Bit („zero extension“):

$$\begin{array}{rccccc} & 0 & a_{n-1} & \dots & a_1 & a_0 \\ + & 1 & \neg b_{n-1} & \dots & \neg b_1 & \neg b_0 \\ & c_n & c_{n-1} & \dots & c_1 & 1 \\ \hline = & s_n & s_{n-1} & \dots & s_1 & s_0 \end{array}$$

(Übertragsbits)
(Summe)

Also: $\langle a \rangle - \langle b \rangle < 0 \iff s_n \iff \neg c_n$

Zweierkomplement:

$$[\![a]\!] < [\![b]\!] \iff [\![a]\!] - [\![b]\!] < 0$$

Wir rechnen mit einem zusätzlichen Bit („sign extension“):

$$\begin{array}{rccccc} & a_{n-1} & a_{n-1} & \dots & a_1 & a_0 \\ + & \neg b_{n-1} & \neg b_{n-1} & \dots & \neg b_1 & \neg b_0 \\ & c_n & c_{n-1} & \dots & c_1 & 1 \\ \hline = & s_n & s_{n-1} & \dots & s_1 & s_0 \end{array}$$

(Übertragsbits)

(Summe)

Vergleiche von Zahlen mit Vorzeichen

Zweierkomplement:

$$[\![a]\!] < [\![b]\!] \iff [\![a]\!] - [\![b]\!] < 0$$

Wir rechnen mit einem zusätzlichen Bit („sign extension“):

$$\begin{array}{rrrrrrl} & a_{n-1} & a_{n-1} & \dots & a_1 & a_0 & \\ + & \neg b_{n-1} & \neg b_{n-1} & \dots & \neg b_1 & \neg b_0 & \\ & c_n & c_{n-1} & \dots & c_1 & 1 & \text{(Übertragsbits)} \\ \hline = & s_n & s_{n-1} & \dots & s_1 & s_0 & \text{(Summe)} \end{array}$$

Also:

$[\![a]\!] - [\![b]\!] < 0 \iff s_n$

Zweierkomplement:

$$[\![a]\!] < [\![b]\!] \iff [\![a]\!] - [\![b]\!] < 0$$

Wir rechnen mit einem zusätzlichen Bit („sign extension“):

$$\begin{array}{rccccc} & a_{n-1} & a_{n-1} & \dots & a_1 & a_0 \\ + & \neg b_{n-1} & \neg b_{n-1} & \dots & \neg b_1 & \neg b_0 \\ & c_n & c_{n-1} & \dots & c_1 & 1 \\ \hline = & s_n & s_{n-1} & \dots & s_1 & s_0 \end{array}$$

(Übertragsbits)

(Summe)

Also:

$$[\![a]\!] - [\![b]\!] < 0 \iff s_n \iff a_{n-1} \oplus \neg b_{n-1} \oplus c_n$$

Zweierkomplement:

$$[\![a]\!] < [\![b]\!] \iff [\![a]\!] - [\![b]\!] < 0$$

Wir rechnen mit einem zusätzlichen Bit („sign extension“):

$$\begin{array}{rccccc} & a_{n-1} & a_{n-1} & \dots & a_1 & a_0 \\ + & \neg b_{n-1} & \neg b_{n-1} & \dots & \neg b_1 & \neg b_0 \\ & c_n & c_{n-1} & \dots & c_1 & 1 \\ \hline = & s_n & s_{n-1} & \dots & s_1 & s_0 \end{array}$$

(Übertragsbits)

(Summe)

Also:

$$[\![a]\!] - [\![b]\!] < 0 \iff s_n \iff a_{n-1} \oplus \neg b_{n-1} \oplus c_n \iff s_{n-1} \oplus c_{n-1} \oplus c_n$$

Zusätzliche Flags: **CF**, **SF**, **OF**

Wir[1] führen drei zusätzliche Flags für arithmetische Operationen ein:

- $\mathbf{CF}$: Das Carry-Flag (c_n bei Addition, $\neg c_n$ bei Subtraktion)
- $\mathbf{SF}$: Das Sign-Flag (s_{n-1})
- $\mathbf{OF}$: Das Overflow-Flag ($c_n \oplus c_{n-1}$)

[1] d.h. Intel

Beispiele (Teil 1)

$$
\begin{array}{rrl}
 & 000\ldots000 & = 0 \\
+ & 000\ldots001 & = 1 \\
 & 0000\ldots000 & \\
\hline
= & 000\ldots001 & = 1
\end{array}
$$

$$\mathbf{ZF} = 0, \mathbf{CF} = 0, \mathbf{SF} = 0, \mathbf{OF} = 0$$

Beispiele (Teil 1)

$$
\begin{array}{rrl}
 & 000\ldots000 & = 0 \\
+ & 000\ldots001 & = 1 \\
 & 0000\ldots000 & \\
\hline
= & 000\ldots001 & = 1
\end{array}
$$

$$\mathbf{ZF} = 0, \mathbf{CF} = 0, \mathbf{SF} = 0, \mathbf{OF} = 0$$

$$
\begin{array}{rrl}
 & 000\ldots001 & = 1 \\
- & 000\ldots001 & = 1 \\
 & 1111\ldots111 & \\
\hline
= & 000\ldots000 & = 0
\end{array}
$$

$$\mathbf{ZF} = 1, \mathbf{CF} = 0, \mathbf{SF} = 0, \mathbf{OF} = 0$$

Beispiele (Teil 1)

$$
\begin{array}{rrl}
 & 000\ldots000 & = 0 \\
+ & 000\ldots001 & = 1 \\
 & 0000\ldots000 & \\
\hline
= & 000\ldots001 & = 1
\end{array}
\qquad \mathbf{ZF} = 0, \mathbf{CF} = 0, \mathbf{SF} = 0, \mathbf{OF} = 0
$$

$$
\begin{array}{rrl}
 & 000\ldots001 & = 1 \\
- & 000\ldots001 & = 1 \\
 & 1111\ldots111 & \\
\hline
= & 000\ldots000 & = 0
\end{array}
\qquad \mathbf{ZF} = 1, \mathbf{CF} = 0, \mathbf{SF} = 0, \mathbf{OF} = 0
$$

$$
\begin{array}{rrl}
 & 111\ldots111 & = -1 \\
+ & 000\ldots010 & = 2 \\
 & 1111\ldots110 & \\
\hline
= & 000\ldots001 & = 1
\end{array}
\qquad \mathbf{ZF} = 0, \mathbf{CF} = 1, \mathbf{SF} = 0, \mathbf{OF} = 0
$$

Beispiele (Teil 2)

$$
\begin{array}{rrl}
 & 011\ldots111 & = 2^{n-1}-1 \\
+ & 000\ldots001 & = 1 \\
 & 0111\ldots110 & \\
\hline
= & 100\ldots000 & = 2^{n-1}
\end{array}
$$

$$\mathbf{ZF} = 0, \mathbf{CF} = 0, \mathbf{SF} = 1, \mathbf{OF} = 1$$

Beispiele (Teil 2)

$$
\begin{array}{rrl}
 & 011\ldots111 & = 2^{n-1}-1 \\
+ & 000\ldots001 & = 1 \\
 & 0111\ldots110 & \\
\hline
= & 100\ldots000 & = 2^{n-1}
\end{array}
$$

$$\mathbf{ZF} = 0, \mathbf{CF} = 0, \mathbf{SF} = 1, \mathbf{OF} = 1$$

$$
\begin{array}{rrl}
 & 100\ldots000 & = -2^{n-1} \\
- & 000\ldots001 & = 1 \\
 & 1000\ldots001 & \\
\hline
= & 011\ldots111 & = 2^{n-1}-1
\end{array}
$$

$$\mathbf{ZF} = 0, \mathbf{CF} = 0, \mathbf{SF} = 0, \mathbf{OF} = 1$$

Sprungbefehle für Vergleiche

Befehl	Flags
jz, je	$\mathbf{ZF}$
jnz, jne	$\neg\mathbf{ZF}$
jnae, jb	$\mathbf{CF}$
jae, jnb	$\neg\mathbf{CF}$
jna, jbe	$\mathbf{CF} \vee \mathbf{ZF}$
ja, jnbe	$\neg(\mathbf{CF} \vee \mathbf{ZF})$
jnge, jl	$\mathbf{SF} \oplus \mathbf{OF}$
jge, jnl	$\neg(\mathbf{SF} \oplus \mathbf{OF})$
jng, jle	$((\mathbf{SF} \oplus \mathbf{OF}) \vee \mathbf{ZF})$
jg, jnle	$\neg((\mathbf{SF} \oplus \mathbf{OF}) \vee \mathbf{ZF})$
jmp short	unbedingt

n = not, z = zero, e = equal,
g = greater, l = less, a = above, b = below

d.h. **jnbe** = jump if not (below or equal)

Sprungbefehle für Vergleiche

```
        sub   ax, bx
        Jxxx  Ziel
        ...
Ziel:
```

Springen bei	mit Vorzeichen	ohne Vorzeichen
$\mathbf{ax = bx}$	**je**	**je**
$\mathbf{ax \neq bx}$	**jne**	**jne**
$\mathbf{ax > bx}$	**jg**	**ja**
$\mathbf{ax \geq bx}$	**jge**	**jae**
$\mathbf{ax < bx}$	**jl**	**jb**
$\mathbf{ax \leq bx}$	**jle**	**jbe**

Beispiel Sprungbefehle

```
    start   sub esi, esi                      ; Arrayindex
            mov edx, [BYTE Intmax+esi]        ; Minimum
            mov ecx, [BYTE Top+esi]           ; Oberster Index
            sub ebx, ebx                      ; Zaehler
5
    L       mov eax, ebx
            sub eax, ecx
            jae end                           ; Zaehler>=Top?

10          mov esi, ebx
            mov edi, [BYTE Array+esi]         ; edi:=array[ebx]

            mov eax, edi
            sub eax, edx
15          jge skip                          ; array[ebx]>=Minimum?

            mov edx, edi                      ; Minimum:=array[ebx]

    skip    sub esi, esi
20          mov eax, [BYTE Four+esi]
            add ebx, eax                      ; Zaehler+=4

            jmp near L

25  end     hlt
```

Beispiel Sprungbefehle (Teil 2)

```
Four    dd 4
Top     dd 40
Array   dd 1, 2, 3, 4, 5, 6, -7, 8, 9, 10
Intmax  dd 0x7fffffff
```

Opcodes der Vergleichsbefehle

Befehl	Opcode		Flags
jz, je	74	01110100	$\mathbf{ZF}$
jnz, jne	75	01110101	$\neg\mathbf{ZF}$
jnae, jb	72	01110010	$\mathbf{CF}$
jae, jnb	73	01110011	$\neg\mathbf{CF}$
jna, jbe	76	01110110	$\mathbf{CF} \vee \mathbf{ZF}$
ja, jnbe	77	01110111	$\neg(\mathbf{CF} \vee \mathbf{ZF})$
jnge, jl	7c	01111100	$\mathbf{SF} \oplus \mathbf{OF}$
jge, jnl	7d	01111101	$\neg(\mathbf{SF} \oplus \mathbf{OF})$
jng, jle	7e	01111110	$((\mathbf{SF} \oplus \mathbf{OF}) \vee \mathbf{ZF})$
jg, jnle	7f	01111111	$\neg((\mathbf{SF} \oplus \mathbf{OF}) \vee \mathbf{ZF})$
jmp short	eb	11101011	unbedingt

Erweiterung Simulator (Teil 1)

```
class cpu_Y86t {
public:
  cpu_Y86t():IP(0)
  {
    ...
    ZF=SF=OF=CF=false;
  }

  bool ZF, SF, OF, CF;          // Flags

  ...
};
```

Erweiterung Simulator (Teil 2)

```
void cpu_Y86t::step() {
  unsigned ea;
  unsigned I0=MEM[IP], I1=MEM[IP+1], I2=MEM[IP+2];
  IP=IP+1;

  switch(I0) {
  ...
  case jg:
    IP+=1; if(!((SF^OF) || ZF)) IP+=EXTEND8(I1); break;

  case jz:
    IP+=1; if(ZF) IP+=EXTEND8(I1); break;

  case jnz:
    IP+=1; if(!ZF) IP+=EXTEND8(I1); break;
  ...
  }
}
```

Erweiterung Simulator (Teil 3)

```
void cpu_Y86t::add_sub(
  unsigned &dest, unsigned src, bool sub)
{
  unsigned long long result;

  if(sub)
    result=(unsigned long long)dest-(unsigned long long)src;
  else
    result=(unsigned long long)dest+(unsigned long long)src;

  ZF=((unsigned long)result==0);
  SF=sign32(result);
  CF=result&0x100000000ULL;
  OF=CF^SF^sign32(dest)^sign32(src);

  dest=result;
}
```

Erweiterung Simulator (Teil 4)

```
    case add:
      IP+=1;
      add_sub(R[RD(I1)], R[RS(I1)], false);
      break;
5
    case sub:
      IP+=1;
      add_sub(R[RD(I1)], R[RS(I1)], true);
      break;
```

Erweiterung Hardware (Teil 1)

```
wire jb  =opcode=='h72;
wire jae=opcode=='h73;
wire jbe=opcode=='h76;
wire ja  =opcode=='h77;
wire  jl  =opcode=='h7c;
wire jge=opcode=='h7d;
wire  jle =opcode=='h7e;
wire jg  =opcode=='h7f;
wire jz  =opcode=='h74;
wire jnz=opcode=='h75;
wire jmp_short=opcode=='heb;
wire branch=jb  ||  jae  ||  jbe  ||  ja    ||   jl   ||  jge  ||
                jle  ||  jg   ||  jz   ||  jnz  ||  jmp_short;
```

Erweiterung Hardware (Teil 2)

```
wire btaken=
        jb  ? CF:
        jae? !CF:
        jbe? CF||ZF:
        ja  ? !(CF||ZF):
        jl  ? SFˆOF:
        jge? !(SFˆOF):
        jle ? (SFˆOF)||ZF:
        jg  ? !((SFˆOF)||ZF):
        jz  ? ZF:
        jnz? !ZF:
        jmp_short;
```

Erweiterung Hardware (Teil 3)

```
reg ZF, CF, OF, SF;

wire [31:0]  ALU_op2=memory?displacement:sub?~B:B;
wire [32:0]  ALUout=A+ALU_op2+sub;

always @(posedge clk)
   if (ue[2])  begin
      ...
      if (aluop) begin
        ZF=(ALUout[31:0]==0);
        CF=ALUout[32]^sub;
        SF=ALUout[31];
        OF=ALUout[32]^ALUout[31]^A[31]^ALU_op2[31];
      end
   end
```

Ausblick

Weitere mögliche Erweiterungen:

- Stack: **push**, **pop**
- Funktionen: **call**, **ret**
- Indirekte Sprünge
  (Zieladresse kommt aus einem Register oder aus dem RAM)
- Weitere Adressierungsmodi:

```
mov eax, [esi]
mov eax, [4*esi]
mov eax, [Konstante]
```

- Konstanten in ALU-Befehlen:

```
mov eax, 1
add eax, 1
```

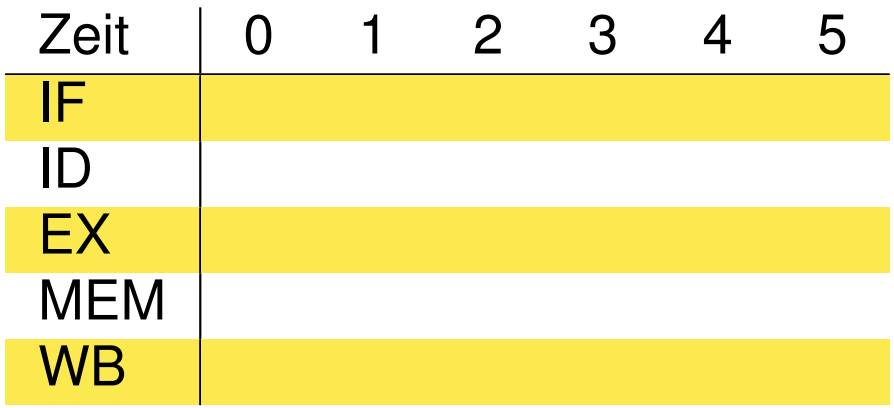
- Abkürzungen: **inc**, **dec**

Pipelining

- Erhöhung des Instruktionsdurchsatzes mit der Fliessband-Idee

- Standard-Technik in allen modernen Schaltungen
  (auch GPUs, Video, ...)

Pipelining



Zeit	0	1	2	3	4	5
IF	I_1					
ID						
EX						
MEM						
WB						

Zeit	0	1	2	3	4	5
IF	I_1	I_2				
ID		I_1				
EX						
MEM						
WB						

Zeit	0	1	2	3	4	5
IF	I_1	I_2	I_3			
ID		I_1	I_2			
EX			I_1			
MEM						
WB						

Zeit	0	1	2	3	4	5
IF	I_1	I_2	I_3	I_4		
ID		I_1	I_2	I_3		
EX			I_1	I_2		
MEM				I_1		
WB						

Zeit	0	1	2	3	4	5
IF	I_1	I_2	I_3	I_4	I_5	
ID		I_1	I_2	I_3	I_4	
EX			I_1	I_2	I_3	
MEM				I_1	I_2	
WB					I_1	

Zeit	0	1	2	3	4	5
IF	I_1	I_2	I_3	I_4	I_5	I_6
ID		I_1	I_2	I_3	I_4	I_5
EX			I_1	I_2	I_3	I_4
MEM				I_1	I_2	I_3
WB					I_1	I_2

Performance:

$$IPC \cdot \frac{1}{\tau}$$

$$\begin{aligned} IPC &\approx 1 \\ \tau &\approx D_{FF} + \frac{D}{n} \end{aligned}$$

wobei:

- IPC: Instructions per Cycle
- τ: Zykluszeit
- n: Stufen
- D: Kombinatorisches Delay ohne Pipeline-Register

Pipelining

✘ Pipelining kann verhindert werden durch:

- Ressourcenkonflikte
- Daten- und Kontrollabhängigkeiten

Ressourcenkonflikte werden zunächst durch Replikation behoben;
falls dies nicht möglich ist, müssen Stall-Cycles eingefügt werden.

Ressourcenkonflikte

Q: Welche Ressourcen teilen sich die Stufen unserer Pipeline?

Ressourcenkonflikte

Q: Welche Ressourcen teilen sich die Stufen unserer Pipeline?

A: Den Systembus, bzw. das RAM (in den Stufen IF und MEM)!

Ressourcenkonflikte

Q: Welche Ressourcen teilen sich die Stufen unserer Pipeline?

A: Den Systembus, bzw. das RAM (in den Stufen IF und MEM)!

Q: Was tun?

Ressourcenkonflikte

Q: Welche Ressourcen teilen sich die Stufen unserer Pipeline?

A: Den Systembus, bzw. das RAM (in den Stufen IF und MEM)!

Q: Was tun?

A: Die meissten Prozessoren haben einen L1-Cache der zwei (Lese-)Zugriffe gleichzeitig erlaubt!

(Oder gleich einen getrennten I- und D-Cache)

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	0
IF	**mov ebx, [esi]**
ID	
EX	
MEM	
WB	

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	1
IF	**add eax, ebx**
ID	**mov ebx, [esi]**
EX	
MEM	
WB	

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	2
IF	...
ID	**add eax, ebx**
EX	**mov ebx, [esi]**
MEM	
WB	

DATENABHÄNGIGKEIT!
Wir würden jetzt den falschen (alten) Wert aus ebx lesen!

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	3
IF	...
ID	**add eax, ebx**
EX	BUBBLE
MEM	**mov ebx, [esi]**
WB	

DATENABHÄNGIGKEIT!
Wir würden jetzt den falschen (alten) Wert aus ebx lesen!

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	4
IF	...
ID	**add eax, ebx**
EX	BUBBLE
MEM	BUBBLE
WB	**mov ebx, [esi]**

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	5
IF	...
ID	...
EX	**add eax, ebx**
MEM	BUBBLE
WB	BUBBLE

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	6
IF	...
ID	...
EX	...
MEM	**add eax, ebx**
WB	BUBBLE

Beispiel-Programm:

```
mov ebx, [esi]
add eax, ebx
```

Ausführung in der Pipeline:

Zeit	7
IF	...
ID	...
EX	...
MEM	...
WB	**add eax, ebx**

Forwarding

- ✘ Die „Pipeline-Bubbles“ verlangsamen offensichtlich die Ausführung
- Ähnliches Problem: ALU-Operation direkt gefolgt von **jz**/**jnz**
- Beobachtung: Tritt gdw. auf, wenn Instruktion in der Phase ID Daten braucht, die noch nicht geschrieben worden sind
- Lösung: Forwarding

Forwarding

1. Vergleiche Quellregister der Instruktion in Stufe 2 mit den Zielregistern der Instruktionen in den Stufen 3, 4, und 5

2. Falls mehrere Übereinstimmungen:
   nimm die kleinste Stufe mit Übereinstimmung

3. Falls Ergebnis noch nicht verfügbar, füge Bubble ein

   Beispiel: MRmov in Stufe 2

- Viele moderne I/O Geräte sind über einen *seriellen Bus* mit dem System verbunden (USB, S-ATA)
- Implementierung: Shiftregister

Memory-Mapped I/O

Anbindung an die CPU:

- Das/die Register kann einfach als „RAM-Stück“ an die CPU angebunden werden
- Die Adresse wird idR automatisch vergeben (PCI)

- Man liesst eine 0, falls keine Daten vorliegen.

Wie auslesen? Etwa so? (*polled-I/O*)

```
...
unsigned char ch=*((unsigned char *)0x10000);
if(ch!=0) process_Key(ch);
...
```

Wie auslesen? Etwa so? (*polled-I/O*)

```
...
unsigned char ch=*((unsigned char *)0x10000);
if(ch!=0) process_Key(ch);
...
```

✘ verheizt Strom

Interrupts

- Interrupts: (vorübergehende) Unterbrechung des Programms

- Anwendungen:
  - I/O ohne Polling
  - Multitasking/multithreading
  - auch: Exceptions

- Implementierung:
  - Interrupt Request (IRQ) Signal an der CPU
  - Interrupt Controller zur Verwaltung mehrerer Quellen

Effekt:

1. Sichert die aktuellen Werte des IP und der Flags auf einem Stack

2. Springt zum Programm das den Interrupt behandelt ($\rightarrow$ Tabelle)

3. Rücksprung mit Spezialbefehl (IRET)