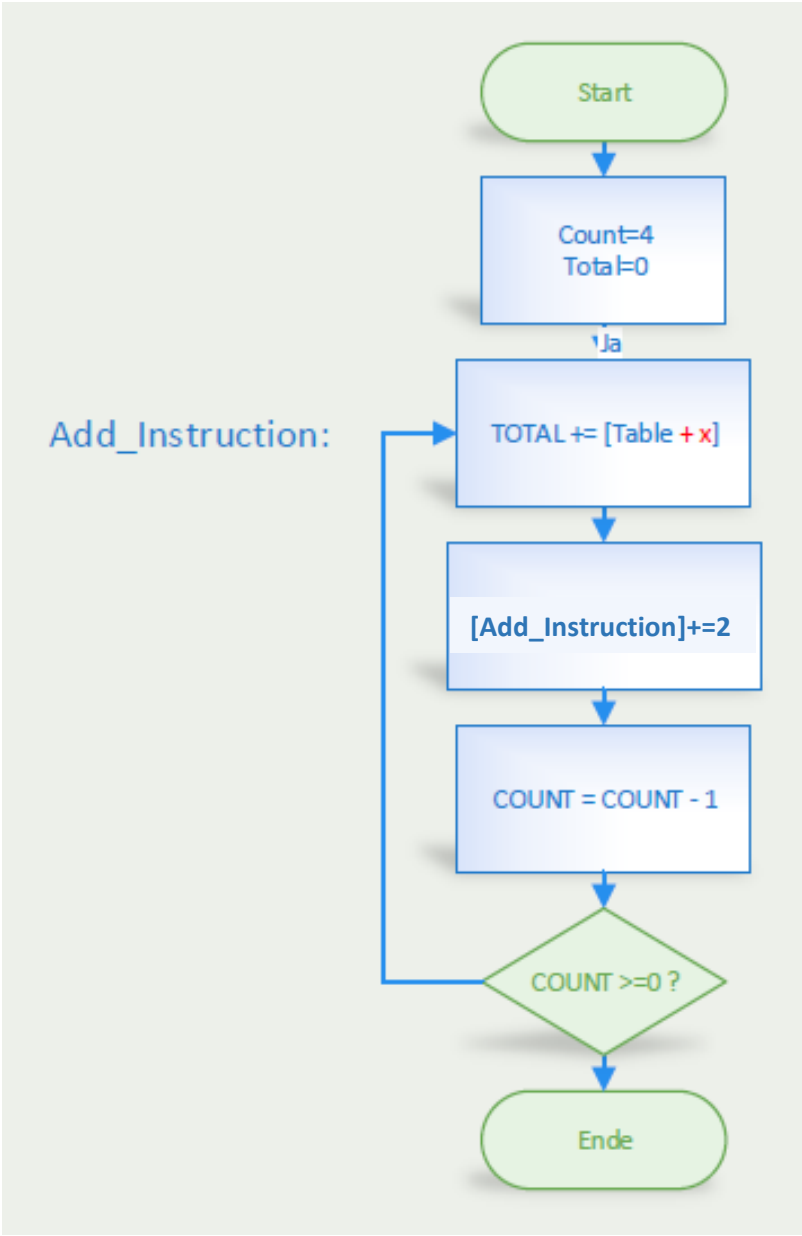
DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Grundlagen von Prozessoren: Review MU0-Rechner

Ein genauer Blick

```
Add_instr: 0010|0000.0001|1100   ADD  Table ; ACC+=<0x1C>
           0001|0000. 0001|0110  STO  Total ; <Total>=ACC
           0000|0000. 0000|0010  LDA  Add_Instr ; ACC=<Add_Instr>
           0010|0000. 0001|1000  ADD  Two ;   ACC+=<0x26>
           0001|0000. 0000|0010  STO  Add_Instr;  <Add_Instr>=ACC
```



Was sollte ein Rechner können – und was kann der MU0?

Rechnen
- **Addition, Subtraktion**, Multiplikation, floating point, etc

Datenbewegung (kopieren)

Strukturierte Lösung eines Problems
- **Abfolge von Rechenoperationen**
- **Verwendung von Daten**, Indirekte Adressierung, Pointer
- **Schleifen, Sprünge**
- Unterprogramme (Strukturen)

Spezielle Befehle (z.B. Interrupts zur Anbindung der Peripherie)

Effizienz
- Optimieren der Datenzugriffe (schneller)
- mehr Daten
- spezielle Algorithmen, z.B. Graphik, Audio-Codecs, etc.

Energieeffizienz

Nächste Generation: MU1

Rechnen
- **Addition, Subtraktion**, Multiplikation, floating point, etc

Datenbewegung (kopieren)

Strukturierte Lösung eines Problems
- **Abfolge von Rechenoperationen**
- **Verwendung von Daten**, **Indirekte Adressierung**, Pointer
- **Schleifen, Sprünge**
- **Unterprogramme (Strukturen)**

Spezielle Befehle (z.B. Interrupts zur Anbindung der Peripherie)

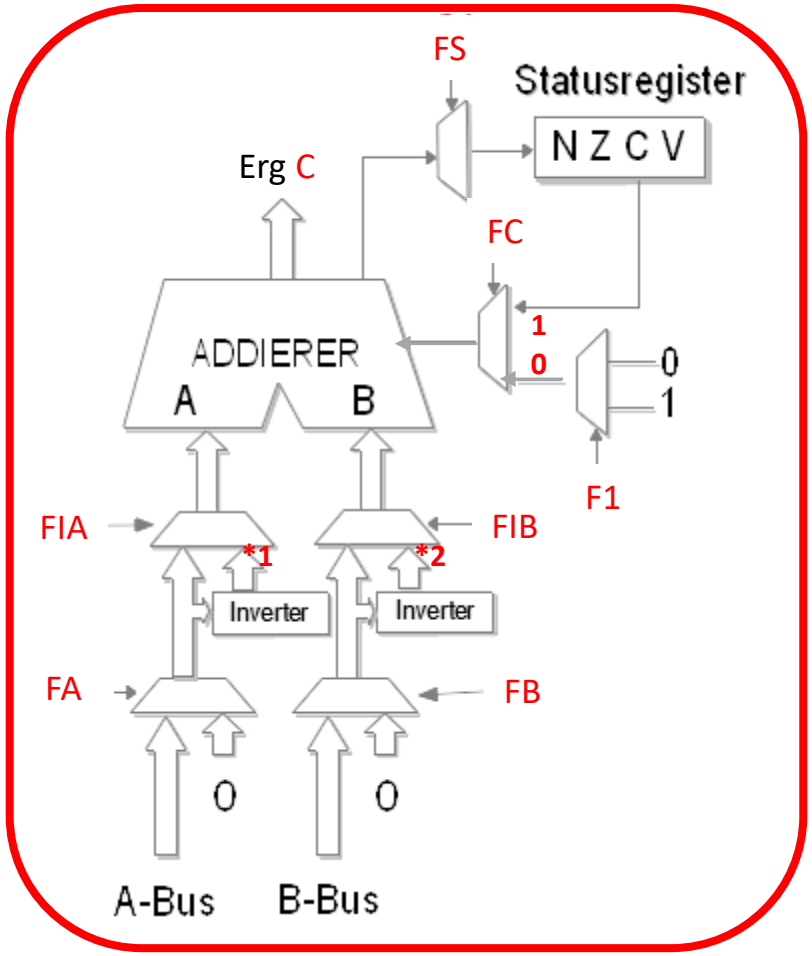
Effizienz
- **Optimieren** der Datenzugriffe (schneller)
- mehr Daten
- spezielle Algorithmen, z.B. Graphik, Audio-Codecs, etc.

Energieeffizienz

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Grundlagen von Prozessoren: Die Rechnergeneration MU1

ALU im Detail



FA: Funktion (0, 1=A) (also eigentlich F0A)

FB: Funktion (0, 1=B) (also eigentlich F0B)

FIA: Funktion (0: A, *1: Inverter, $\overline{A}$)

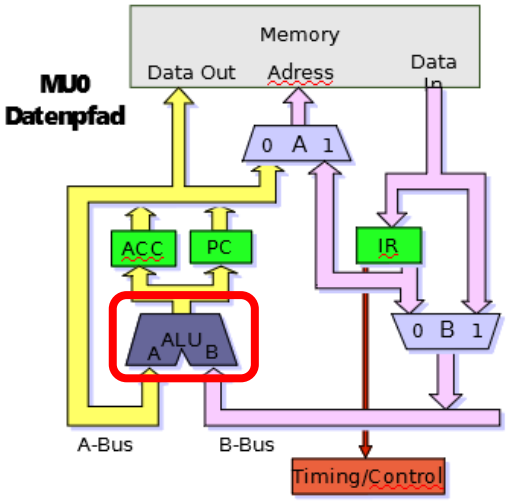
FIB: Funktion (0: B, *2: Inverter, $\overline{B}$)

FC: Funktion (0=F1, 1=Carry-Flag)

F1: Funktion (0/ 1)

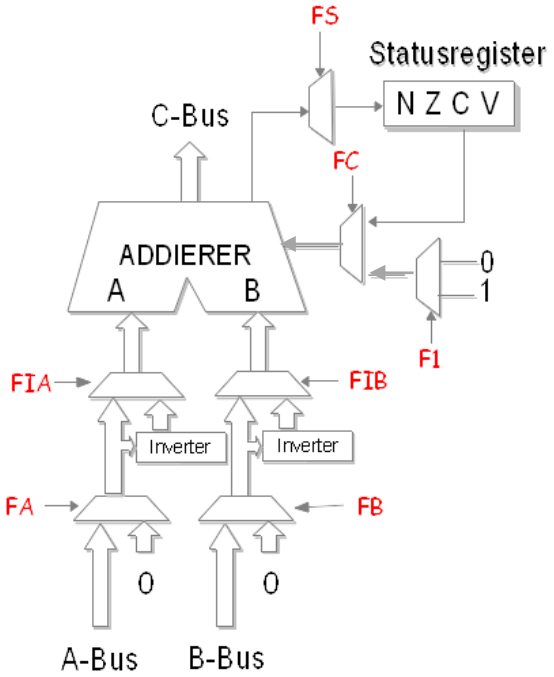
FS: Funktionsergebnis Statusregister

C: Funktionsergebnis C

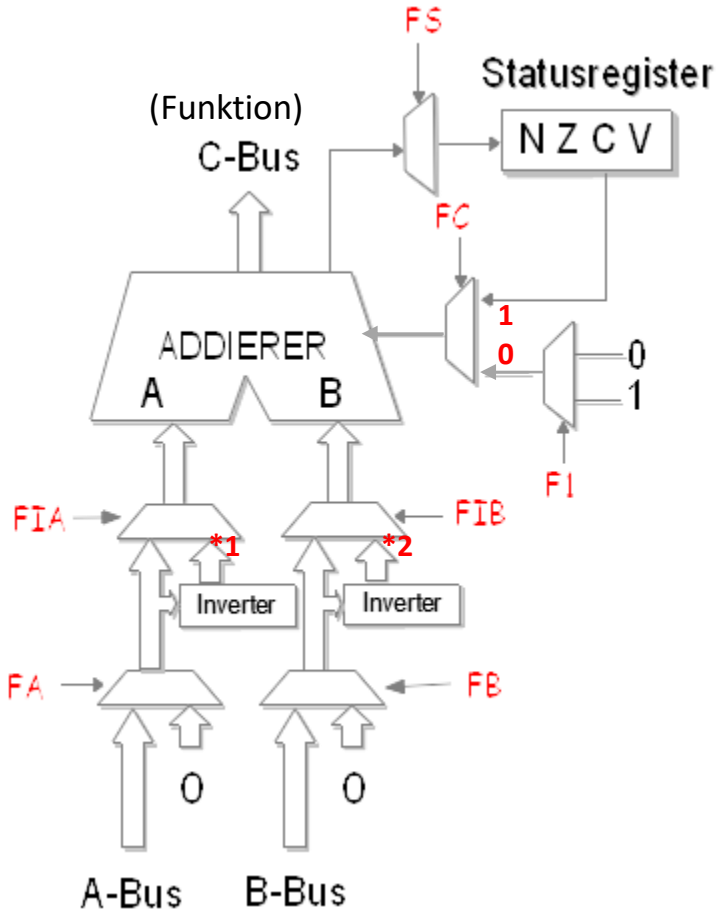


ALU im Detail

- *A+B* ist der Ausgang des Addierers.
- *A-B* wird gebildet als $A+\overline{B}+1$.
- *B* wird implementiert, indem der Eingang A fest auf 0 gesetzt wird
- *A+1* wird implementiert, indem der Eingang B fest auf 0 gesetzt und der F1-Eingang auf 1 gesetzt wird
- Alle Funktionen dieser einfachen ALU können also durch einen Addierer mit Carry-Eingang gebildet werden, dessen Eingang A auf Null gesetzt und dessen Eingang B invertiert werden kann.
- Übung:
  Wie kann man mit dieser ALU *A-1* berechnen?



Die Grundfunktionen der ALU

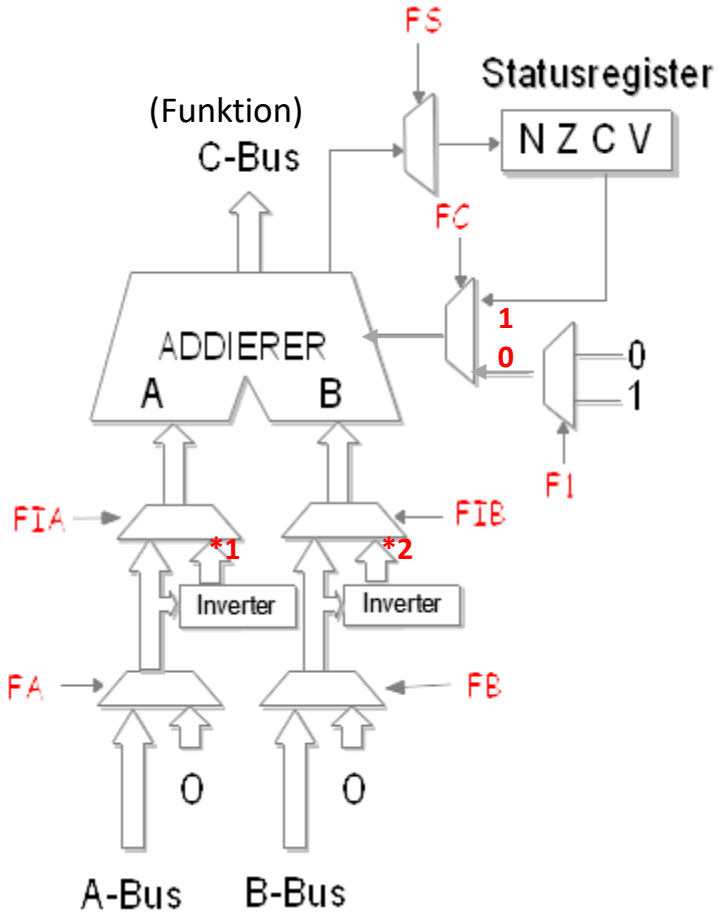


Funktion	FA	FB	FIA	FIB	FC	F1	FS
0							
1							
-1							
A							
B							
A+1							
B+1							
A-1							
A+B							
A-B							
-B							
-A							
B-A							
A+B+C							

?

FS wird gesetzt, wenn die Operation es erfordert und erlaubt (Flags werden geändert)

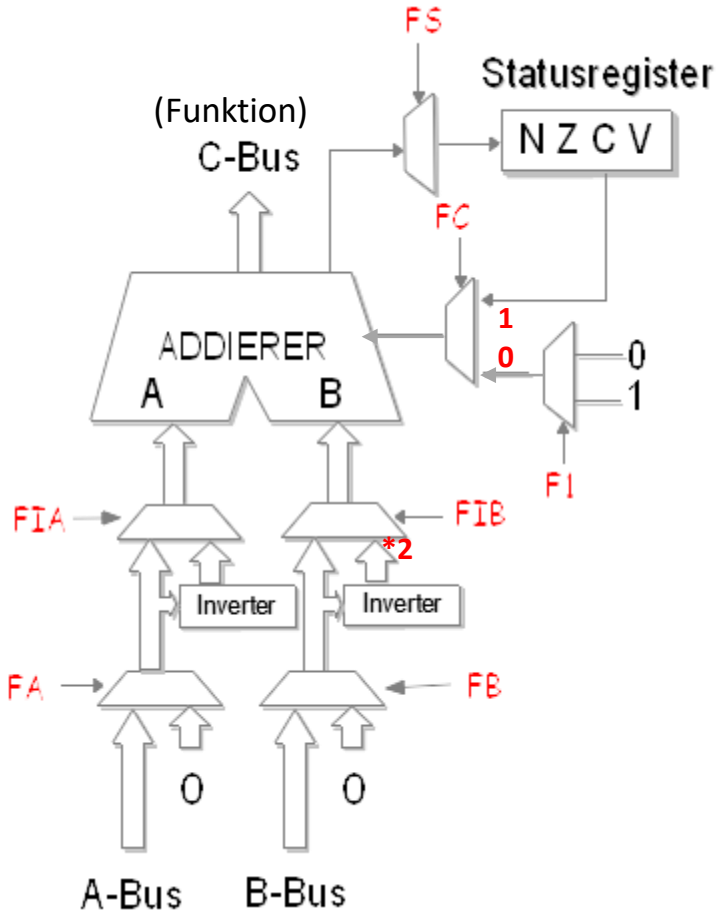
Die Grundfunktionen der ALU



Funktion	FA	FB	FIA	FIB	FC	F1	FS
0							
1							
-1							
A							
B							
A+1							
B+1							
A-1							
A+B							
A-B							
-B							
-A							
B-A							
A+B+C							

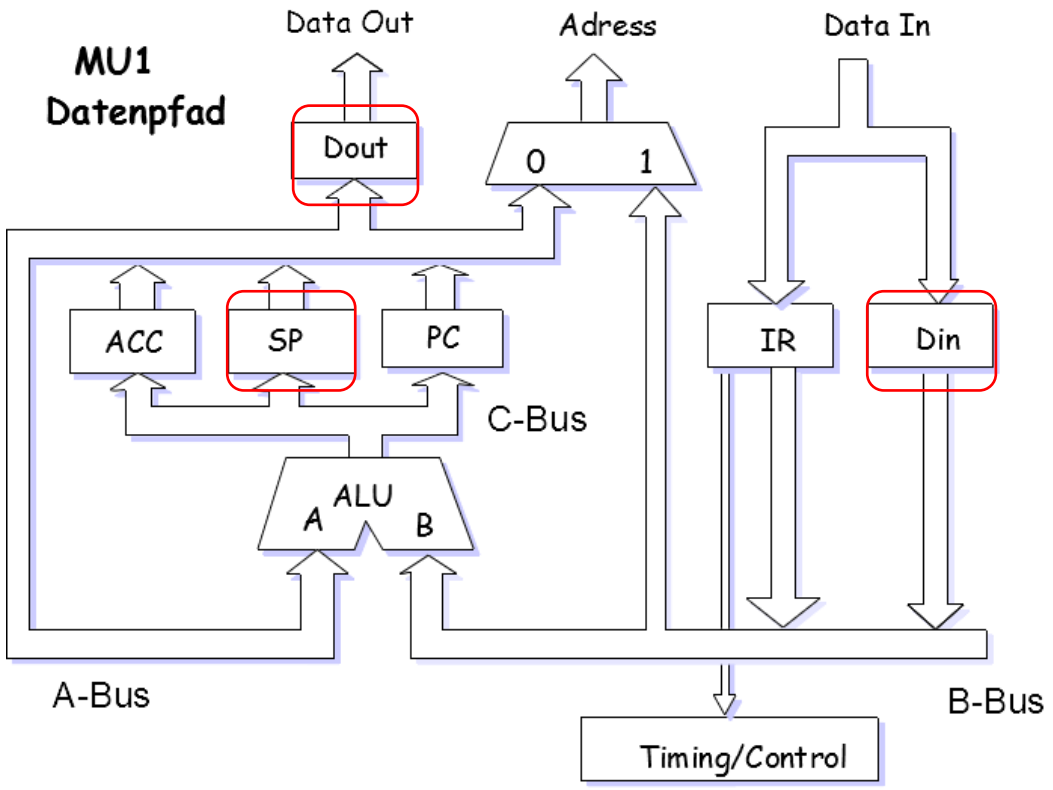
?

Die Grundfunktionen der ALU



Funktion	FA	FB	FIA	FIB	FC	F1	FS
0	0	0	0	0	0	0	x
1	0	0	0	0	0	1	x
-1	0	0	0	1	0	0	x
A	1	0	0	0	0	0	x
B	0	1	0	0	0	0	x
A+1	1	0	0	0	0	1	x
B+1	0	1	0	0	0	1	x
A-1	1	0	0	1	0	0	x
A+B	1	1	0	0	0	0	x
A-B	1	1	0	1	0	1	x
-B	0	1	0	1	0	1	x
-A	1	0	1	0	0	1	x
B-A	1	1	1	0	0	1	x
A+B+C	1	1	0	0	1	0	x

Der MU1-Datenpfad



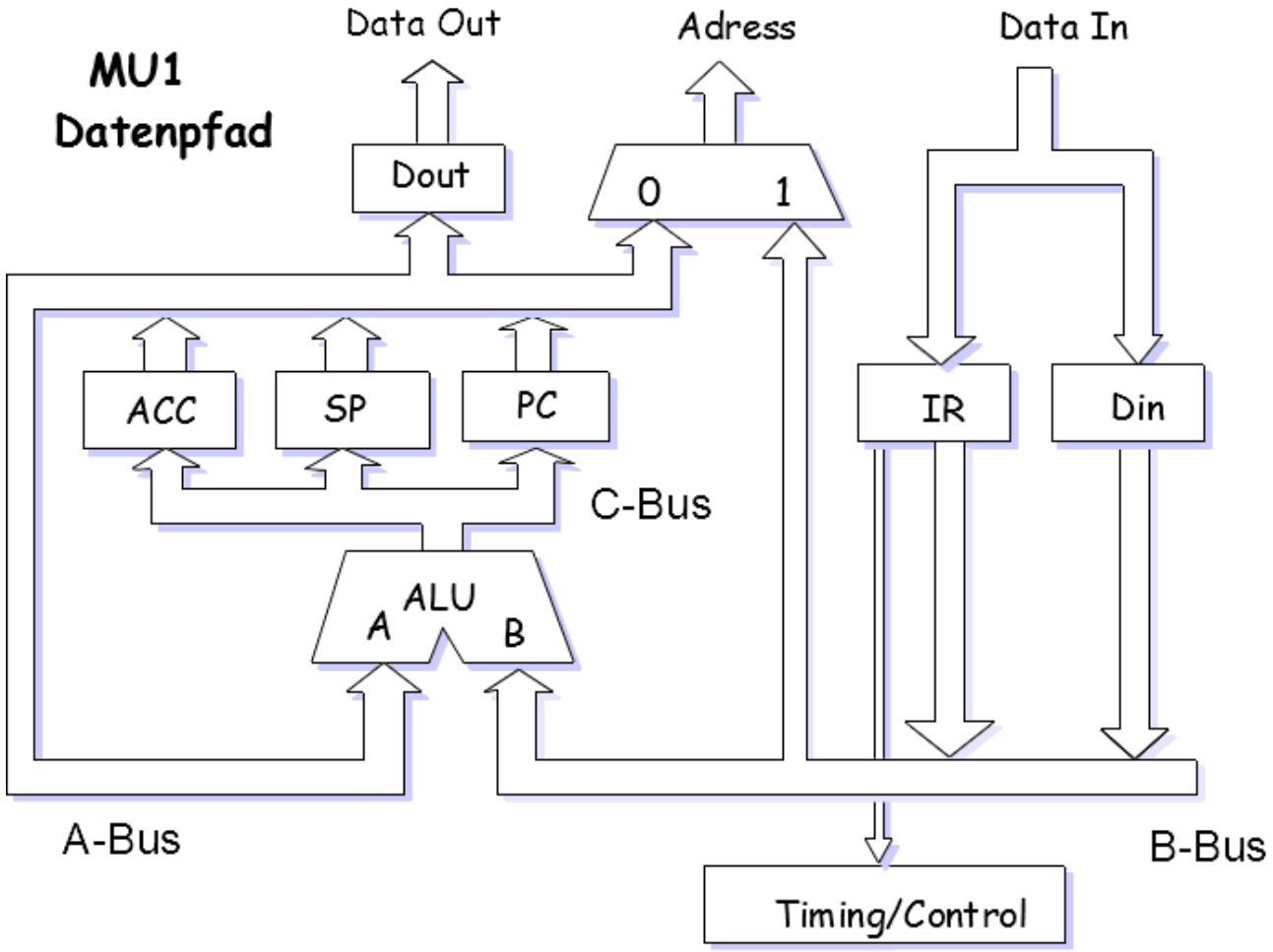
Das erweitere MU0 Modell besitzt

- einen Stackpointer **SP** und
- Register **Din** und **Dout**.

Laufzeiten sind kürzer pro Takt

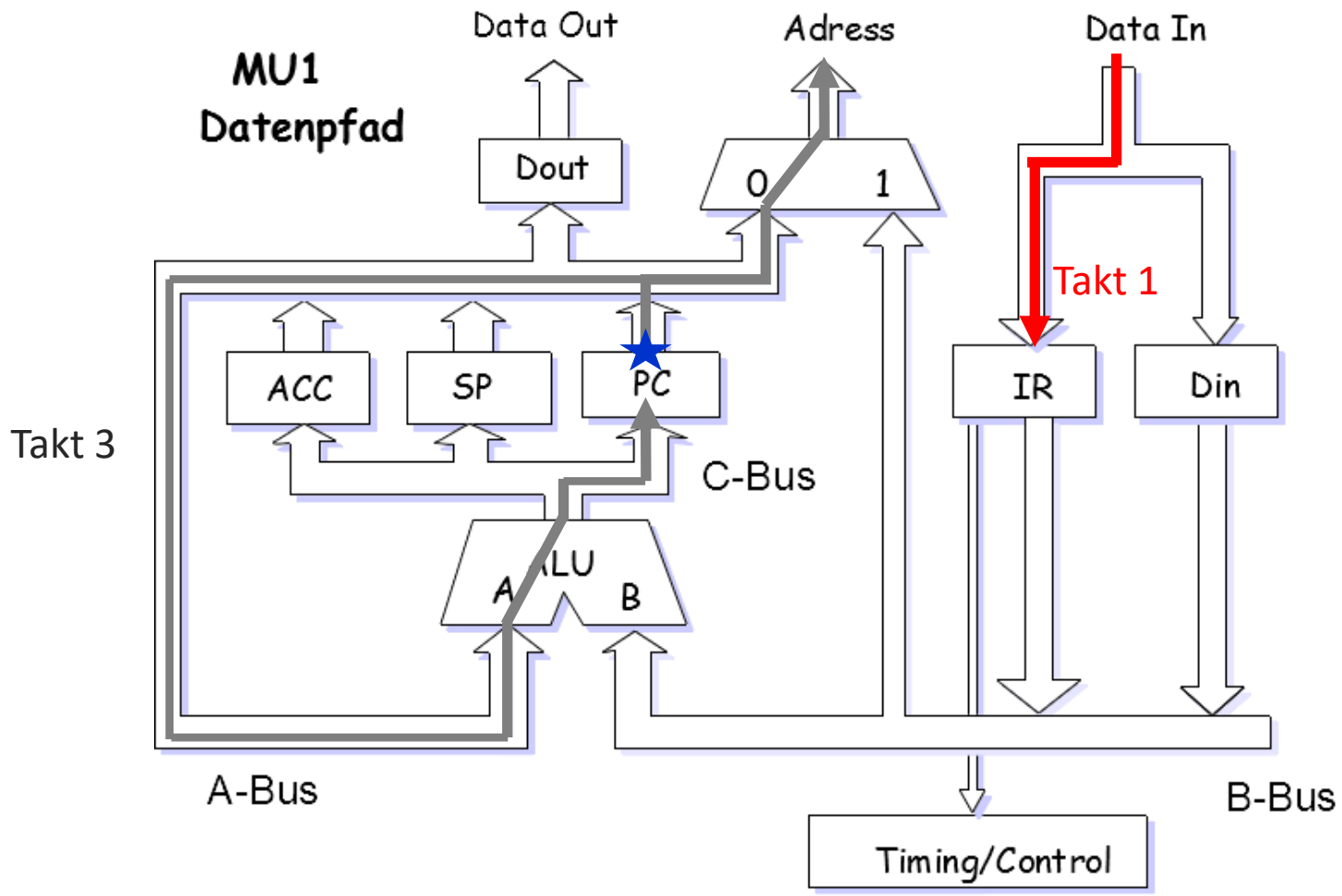
Durch die zusätzlichen Register Din und Dout braucht unser Modell jetzt zwar mehr Taktzyklen, aber diese können schneller durchlaufen werden (teilweise kein Speicherzugriff)
Vorteil: versetzte Parallelität,
Laden und Speichern nur über Register,
Operationen (ALU) greifen nie direkt auf Speicher zu

MU1-Datenpfad



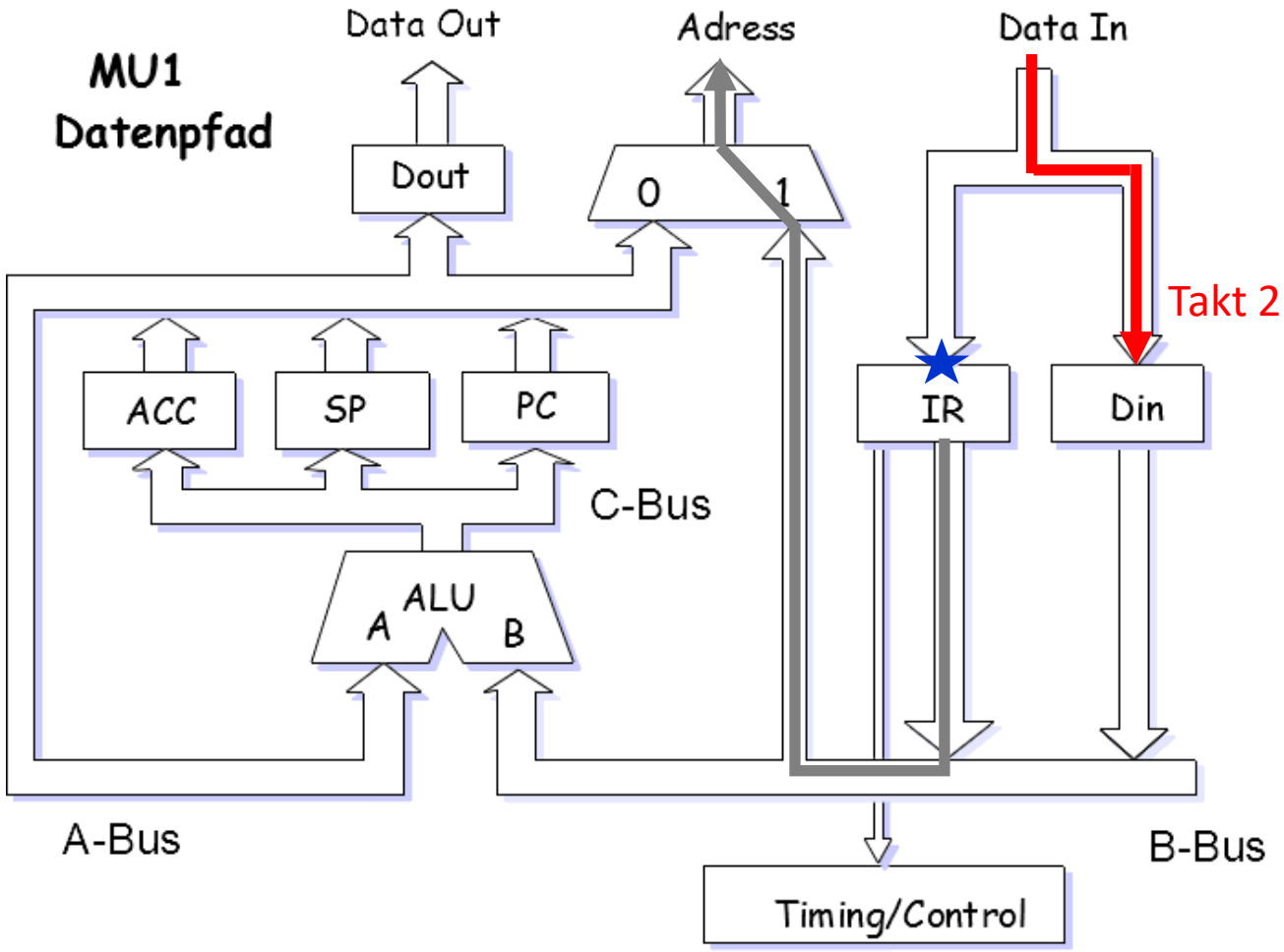
MU1-Datenpfad

LDA S



MU1-Datenpfad

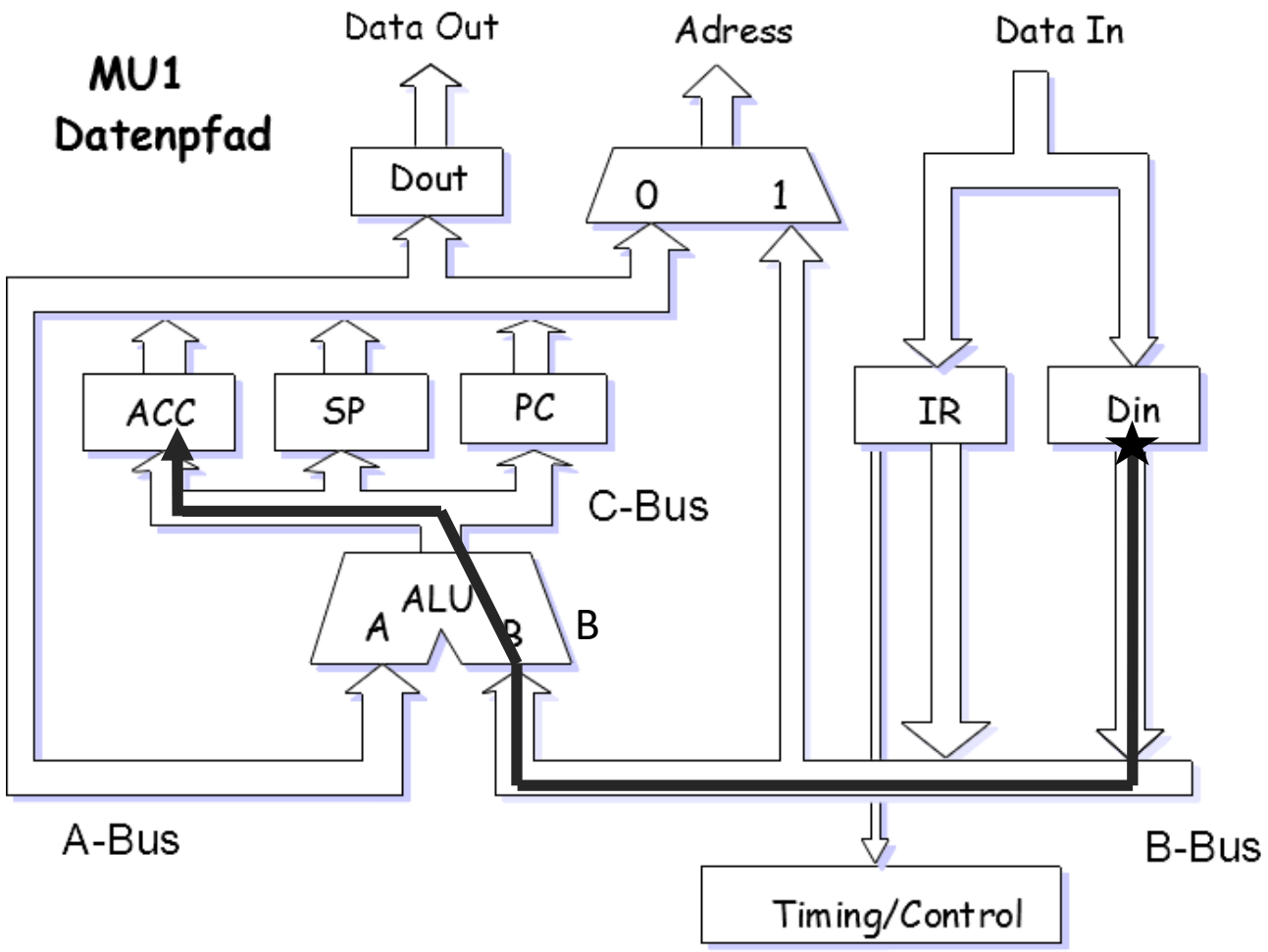
LDA S



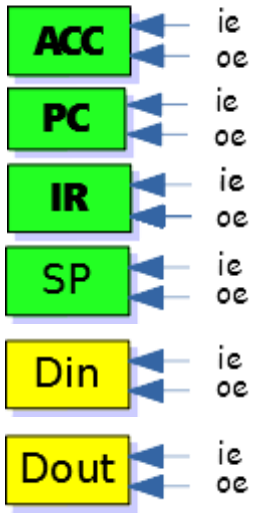
MU1-Datenpfad

LDA S

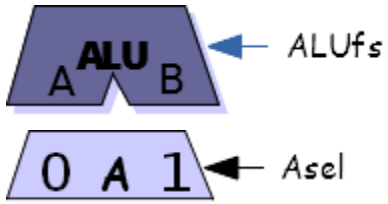
Takt 3



MU1-Elemente ohne Speicher und Steuerwerk



Akkumulator:	ie -> Lesen, oe -> Schreiben
Program Counter:	ie ->Lesen, oe -> Schreiben
Instruction Register:	ie -> Lesen, oe → Schreiben
Stackpointer Register:	ie -> Lesen, oe → Schreiben
DataIn Register:	ie -> Lesen, oe → Schreiben
DataOut Register:	ie -> Lesen, oe → Schreiben



ALU:	Funktionsauswahl über ALUfs (function select)
Adressbusmultiplexer A:	0 -> A-Bus, 1 -> B-Bus

Der Micro-Program-Counter *Step*

- *Step* ist ein Speicher in der Steuereinheit (Timing/Control Unit) unseres Prozessors.
- Die meisten Befehle unseres Prozessors werden nicht in einem Takt ausgeführt, sondern erfordern mehrere Takte. Der Microprogram Counter *Step* gibt der Steuerlogik die Information, welche Aktionen (Fetch/ Execute) jeweils auszuführen sind. Dazu wird *Step* (Takt) als Eingangsinformation für jeden Schritt verarbeitet.
- Zu jedem Schritt steht in der Steuerungstabelle, welcher Wert *Step* in diesem Schritt zugewiesen wird, das heißt: welches der nächste Schritt ist.

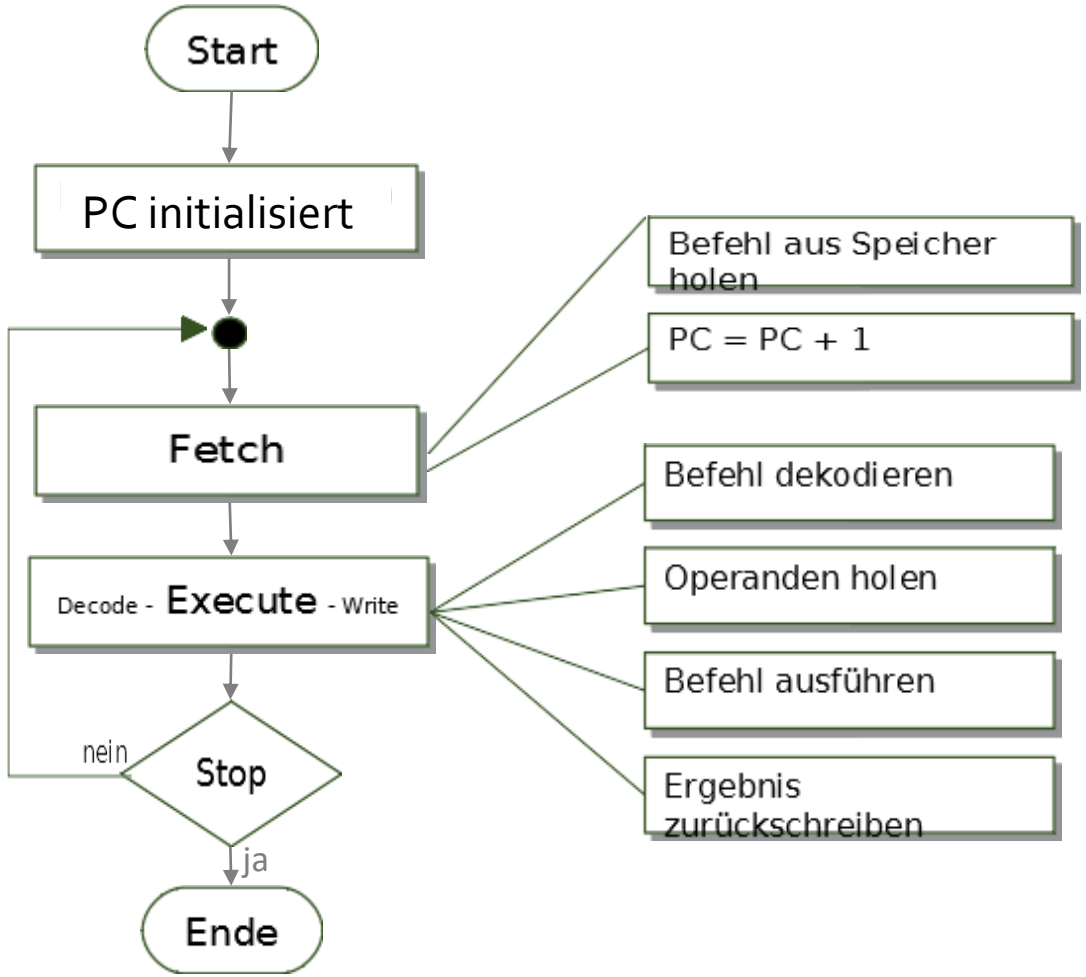
Der Micro Program Counter Step

- Im ersten Schritt eines Befehls (Step ist vorher Null), erfolgt immer der Fetch, das heißt das Lesen des Befehls in das Instruction Register. Der Opcode des Befehls wird dabei ignoriert.

- Im letzten Schritt eines Befehls wird Step wieder auf Null gesetzt, so dass danach ein Fetch ausgeführt wird.

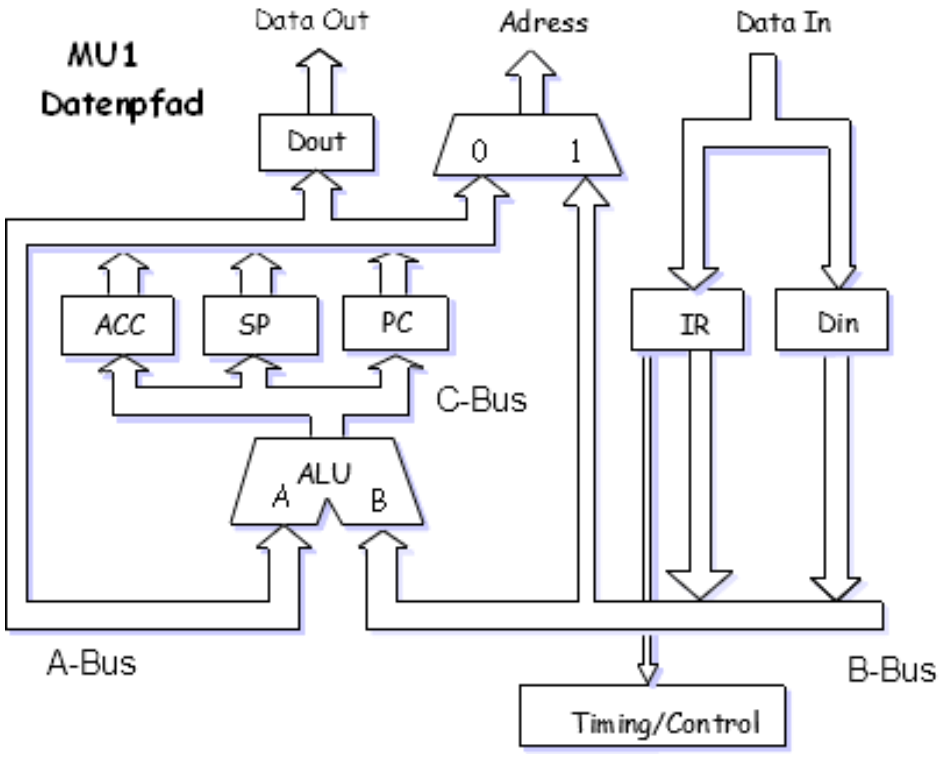
- In den meisten Fällen wird Step in einem Schritt um Eins erhöht.

Befehlsablauf MU1 nach Reset



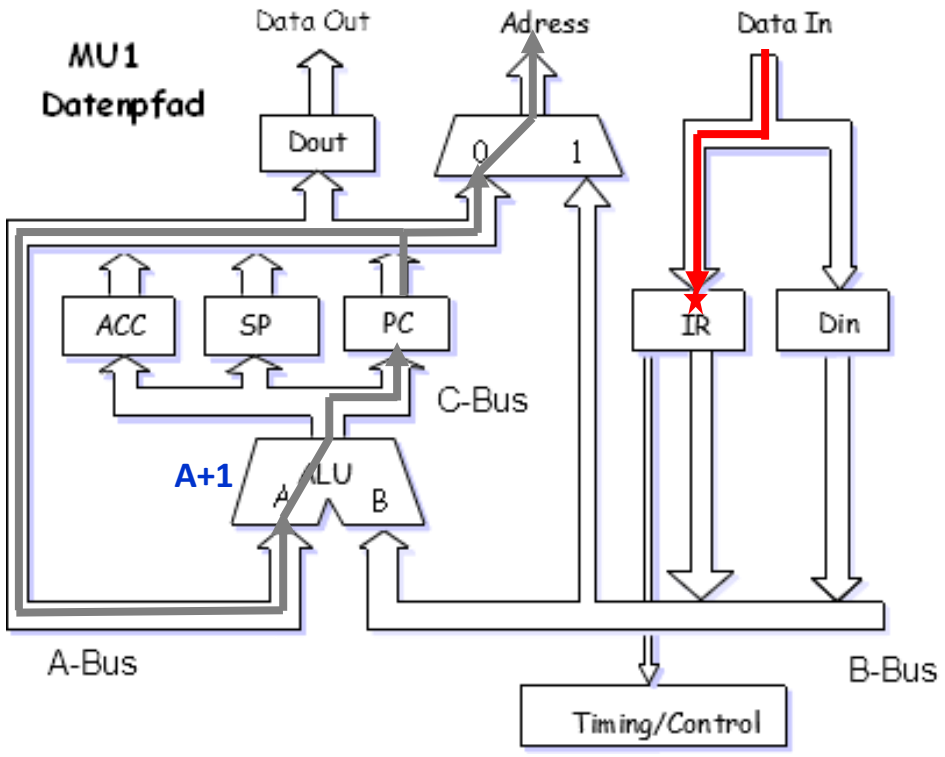
N Mikroschritte

MU1-Befehl



Inputs						Outputs																Description
Instruction	Opcode	/Reset	Step	ACC_Z / Zero	ACC_{15} / Negativ	Step	Adress	ACC_{OE}	ACC_{ie}	PC_{oe}	PC_{ie}	IR_{oe}	IR_{ie}	SP_{oe}	SP_{ie}	DIN_{oe}	DIN_{ie}	$DOUT_{oe}$	$DOUT_{ie}$	ALU Function	MEM_{rq}	RnW	

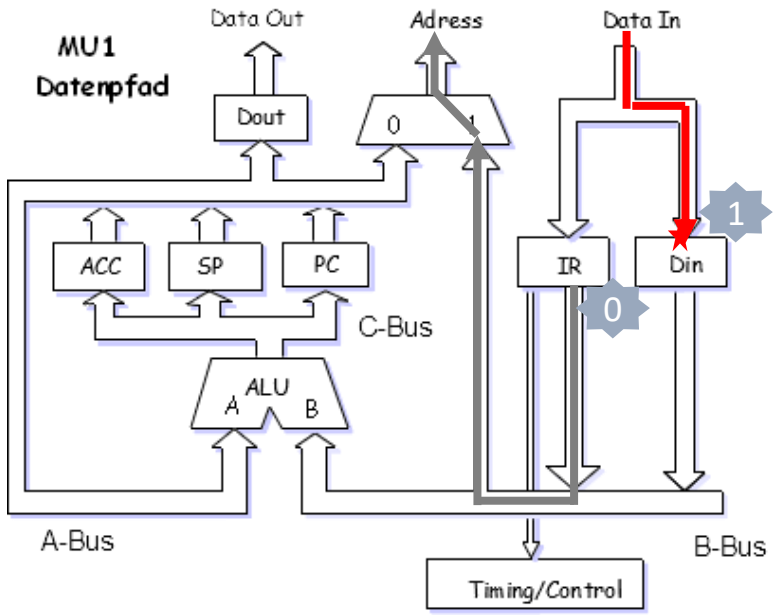
Der Fetch Zyklus im MU1 (Step 0->1, 1 Takt)

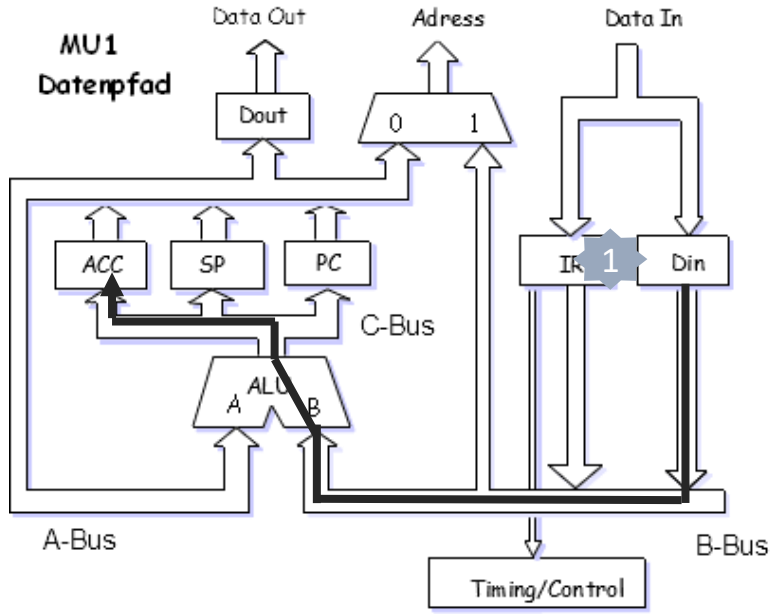


- Der Program Counter (PC) wird auf den Adressbus geschaltet und gleichzeitig an den Eingang A der ALU.
- An der ALU ist der Befehl A+1 ausgewählt, S ist 0.
- In der zweiten Hälfte des Takts wird vom Speicher die nächste Instruktion gelesen und in IR gespeichert.

Inputs						Outputs																Funktion
Instruction	Cpccce	/Reset	step	Z	N	step	Acress	ACCce	ACCie	PCce	PCie	IRce	IRie	SPce	SPie	DINce	DINie	DCLTce	DCLTie	ALU Function	MEMrq	R/rW	
Fetch	xxxx	1	0	x	x	1	0	0	0	1	1	0	1	0	0	0	0	0	0	A+1	1	1	IR=[PC], PC=PC+1

LDA: Das Laden des Akkumulators mit dem Inhalt einer absoluten Adresse

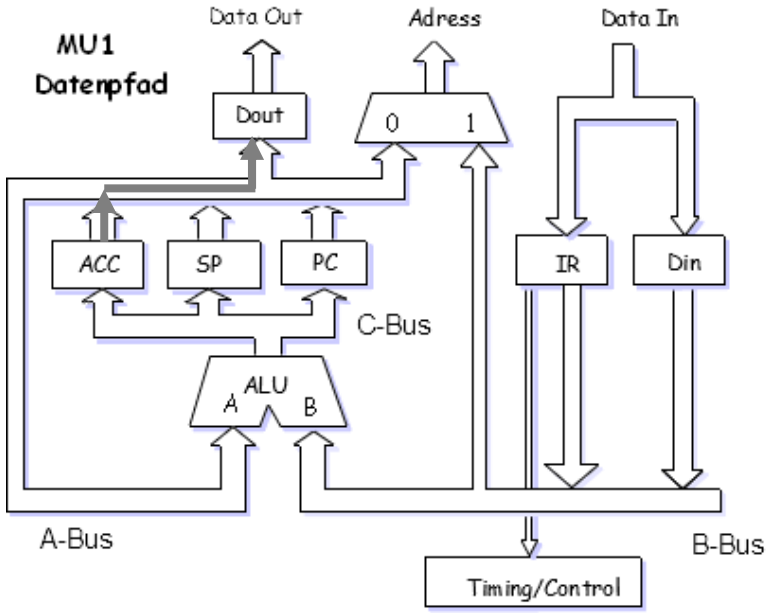


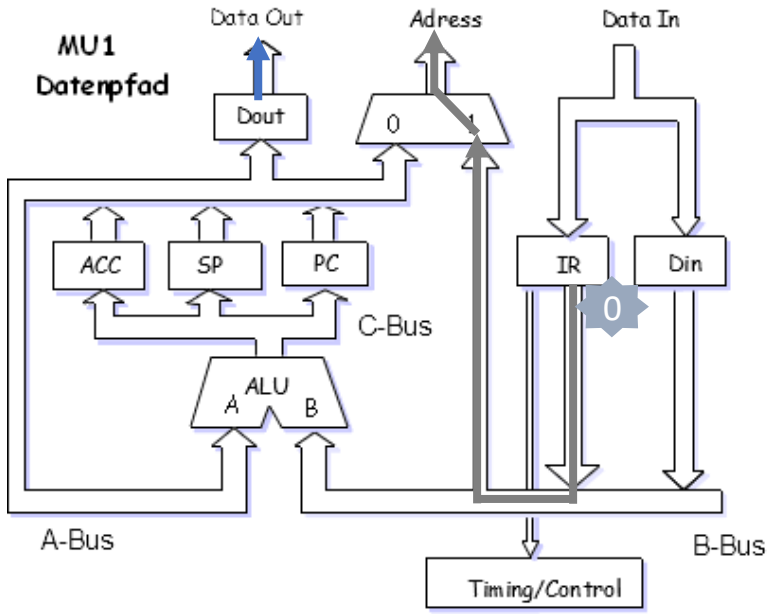


Der Adressteil der Instruktion wird aus dem Instruction Register auf den Adressbus gelegt und der Speicherinhalt nach Din geschrieben, und dann über die ALU in den ACC.

Inputs						Outputs																Funktion
Instruction	Cpcccc	/Reset	step	Z	N	step	Acress	ACCce	ACCie	PCce	PCie	IRce	IRie	SPce	SPie	DINce	DINie	ECLTce	ECLTie	ALU Functi	MEMrq	RrW	
LDA S	0000	1	1	x	x	2	1	0	0	0	0	1	0	0	0	0	1	0	0	x	1	1	Din = [IR]
		1	2	x	x	0	x	0	1	0	0	0	0	0	0	1	0	0	0	B	0	1	ACC = Din

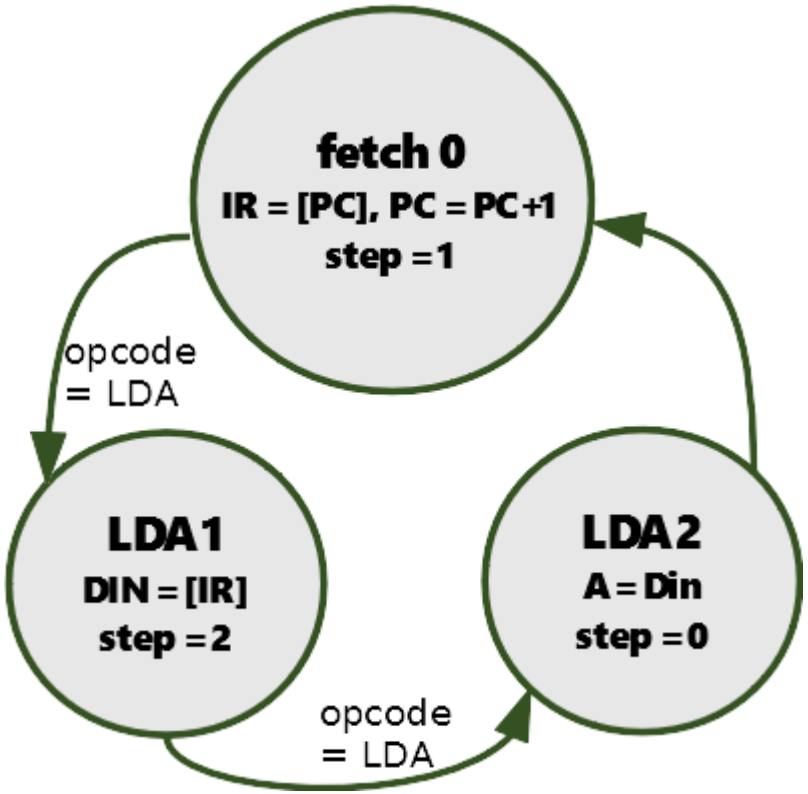
STO s: Das Speichern des Akkumulatorinhalts auf eine absolute Adresse





Der Akkuinhalt wird nach Dout gebracht. Der Adressteil der Instruktion wird aus dem Instruction Register auf den Adressbus gelegt und der Inhalt von Dout in den Speicher geschrieben.

Zustandsbeschreibung des Ladebefehls



- Jeder Befehl beginnt mit einem Fetch-Zyklus.
- Nach dem Fetch-Zyklus entscheidet der Opcode im Instruction Register und die Schrittnummer, welchen Zustand unser Automat als nächstes annimmt.
- Die Schrittnummer (Step) ist unser Programmcounter im Microcode und beschreibt die Abfolge der Befehle im Microcode.
- Zu jedem Schritt gehört eine eindeutige Funktion, die sich wiederum in die Timing-Control-Logik umsetzen läßt.

Adressierungsarten

Unmittelbare Adressierung (immediate)

- Der Befehl enthält eine Konstante
- Beispiel: ADD #1

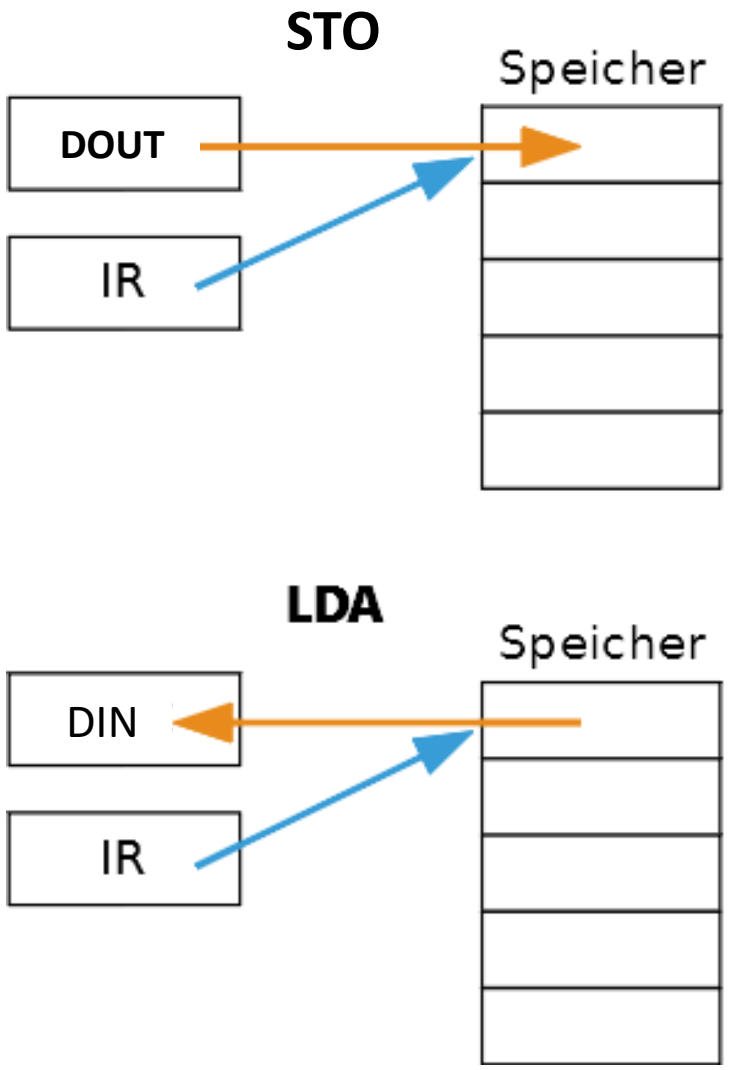
Direkte Adressierung (direct, absolute)

- Der Befehl enhält eine Adresse im Speicher, an der sich der Operand befindet.
- Beispiel: LDA S S gibt ein Adresse im Speicher an

Indirekte Adressierung (indirect)

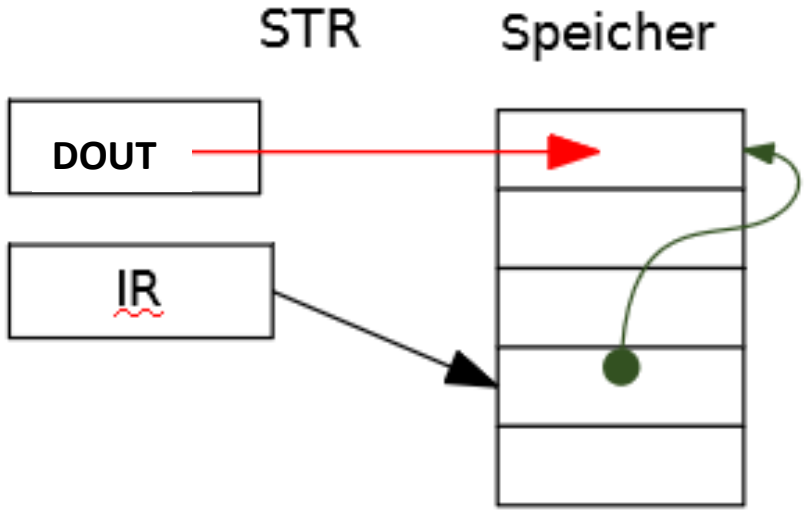
- Der Befehl enthält eine Adresse, an der sich die effektive Adresse des Inhalts befindet.
- Beispiel: LDR S S gibt eine indirekte Adresse an

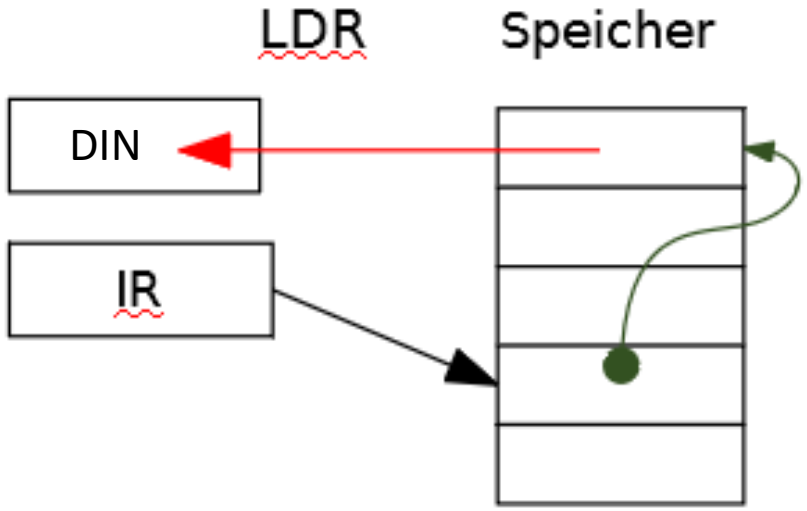
Direkte Adressierung



- Bei der direkten Adresse enthält der Befehl LDA bzw. STA direkt die Speicheradresse, an die gespeichert oder von der gelesen werden soll.
- Die Adresse ist somit im Programm festgeschrieben und kann zur Laufzeit nicht mehr geändert werden.
- (Nur noch mit selbstmodifizierendem Code…)

Indirekte Adressierung





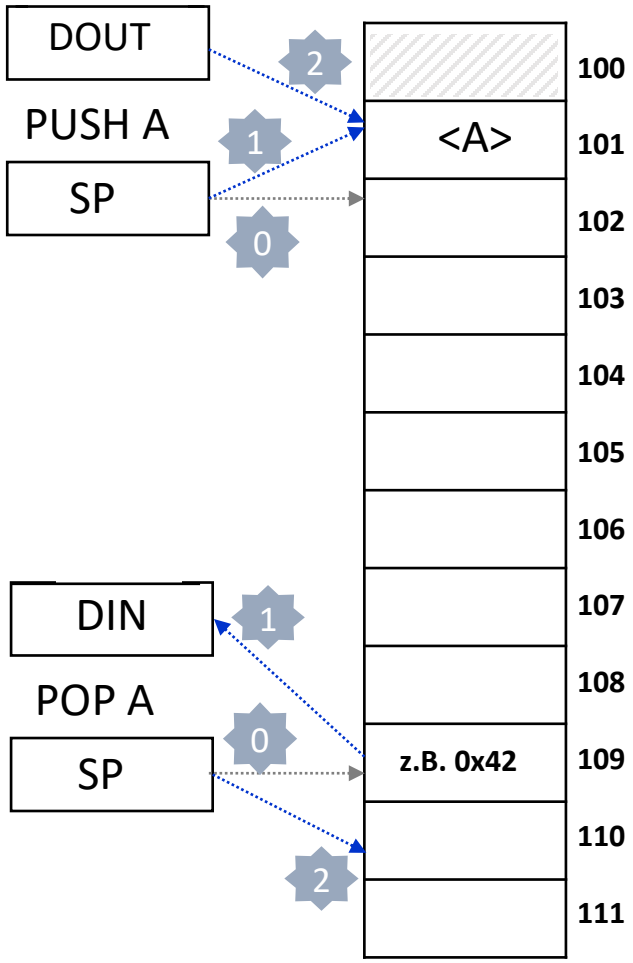
Die Adresse, an die gespeichert oder von der gelesen werden soll, steht nicht mehr direkt im Befehl.
Im Befehl steht nur noch die Speicherstelle, an der sich ein Zeiger auf die Adresse befindet.

Die effektive Adresse kann somit berechnet werden. Mit diesem Befehl ist es möglich, über Arrays zu iterieren, ohne dass dazu selbstmodifiziernder Code nötig ist.

Der Stack

Der Stack

-Adr.- - Werte -
100: DEFW 11



111: STACK_BOTTOM:

Der Stack

-Adr.- - Werte -
1000: DEFW 100

1050:

1100: STACK_BOTTOM

- Der Stack stellt einen dynamischen Speicher dar, auf dem Daten zwischengespeichert werden können.
- Die Grundoperationen eines Stacks sind die Operationen Push und Pop.
- Push erniedrigt den Stackpointer und schiebt dann einen Wert in das Speicherwort, auf das der Stackpointer gerade zeigt („legt ab“) und
- Pop liest ein Speicherwort von der Adresse auf die der Stackpointer zeigt („nimmt runter“) und erhöht den Stackpointer.
- Der Stackpointer zeigt initial auf STACK_BOTTOM (leeres Gabbionengefäss)
- Das unterste Wort eines Stacks bleibt immer ungenutzt und kann deshalb für die „Unterlauf“-Überwachung benutzt werden.

Speicheransicht

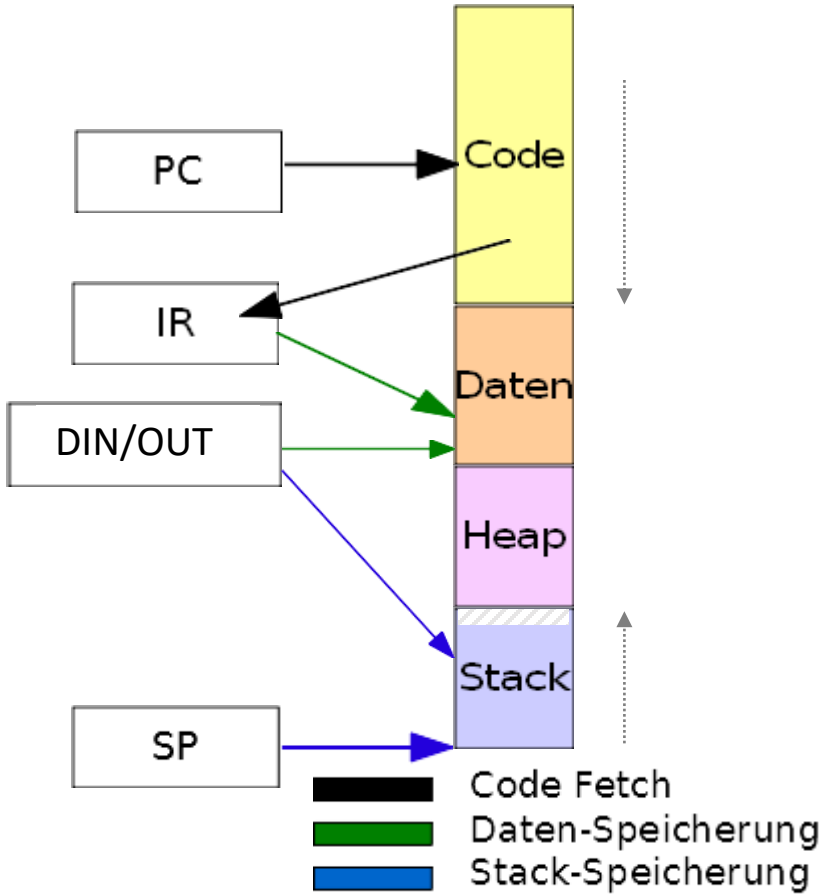
Memory

Addresses

Address 0x00012698 Target is LITTLE endian

	0	4	8	C	ASCII
0x00012698	0x00012980	0x000129e8	0x00012a50	0x00000000	.)...)..P*.....
0x000126a8	0x00000000	0x00000000	0x00000000	0x00000000	.....
0x000126b8	0x00000000	0x00000000	0x00000000	0x00000000	.....
0x000126c8	0x0000a558	0x00000000	0x00000000	0x00000000	X.....
0x000126d8	0x00000000	0x00000000	0x00000000	0x00000000	.....
0x000126e8	0x00000000	0x00000000	0x00000000	0x00000000	.....
0x000126f8	0x00000000	0x00000000	0x00000000	0x00000000	.....
0x00012708	0x00000000	0x00000000	0x00000000	0x00000000	.....

Der Stack im Speicher



- Der Stackpointer wird am Anfang auf eine Adresse am Ende des Speicherbereichs gesetzt und wächst dann langsam „nach oben“ - zu den niederen Adressen.
- Die Programme werden meistens in dem unteren Speicherbereich platziert und der Programmcounter beim Start auf die Adresse 0 gesetzt (s. MU0-Programm).
- An den Programmbereich schließt sich als Nächstes der Datenbereich an.
- Zwischen Daten und Stack findet man meist einen Heapbereich, eine andere Form der dynamischen Speicherverwaltung.

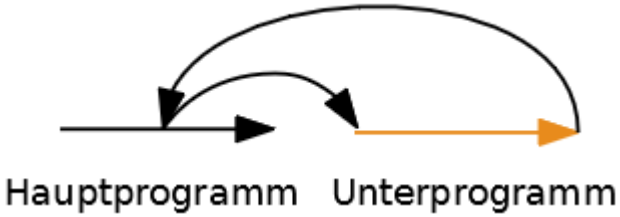
Achtung!

Die Speicheraufteilung kann auch kopfüber – mit der Adresse 0 unten - dargestellt werden!

Achten Sie deshalb bei Darstellungen des Stacks oder der Speicheraufteilung immer darauf, in welche Richtung die Adressen steigen –

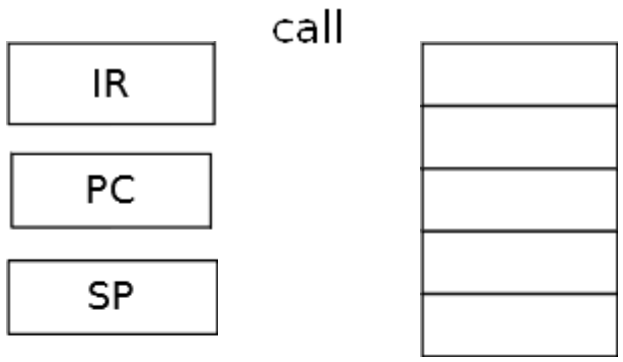
und markieren Sie auch selbst in eigenen Darstellungen mit der 0 immer den Verlauf!

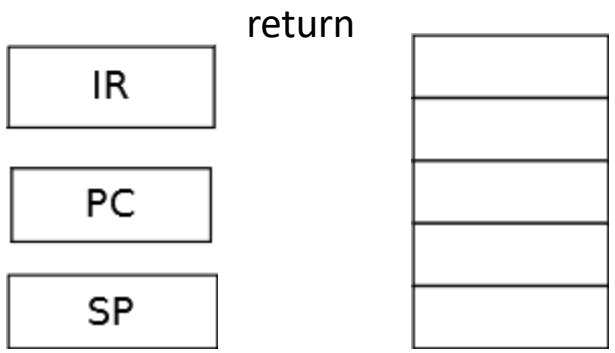
Unterprogramme



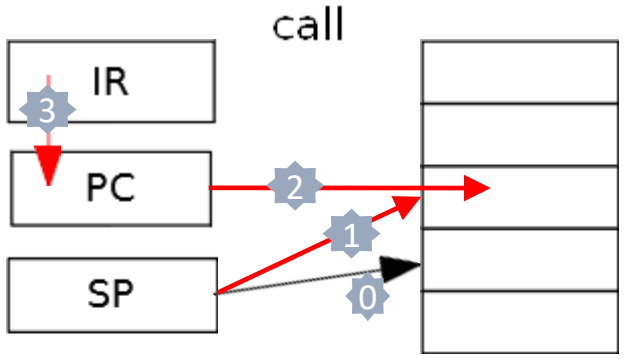
- Unterprogramme sind ein wichtiges Strukturierungsmittel für die Programmierung.
- Unterprogramme werden mit einem Sprungbefehl (call, bl) angesprungen und die *Adresse nach dem Sprungbefehl* wird gespeichert. Nach dem Verlassen des Unterprogramms wird die Programmausführung an der gespeicherten Stelle fortgesetzt.
- Der Rücksprung erfolgt, indem die gespeicherte Adresse wieder in den ProgramCounter übertragen wird.

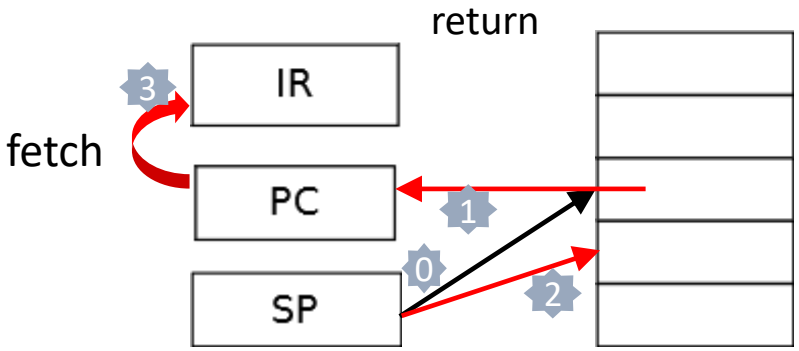
Unterprogramm-Aufrufe (Freeze)





Unterprogramm-Aufrufe





- Unterprogrammaufrufe nutzen häufig (nicht bei jedem Prozessor) den Stack zur Speicherung der Rücksprungadresse.
- Beim Call-Befehl wird die Adresse des nächsten nach der Rückkehr auszuführenden Befehls auf dem Stack gespeichert und die Sprungadresse aus dem Instruction register in den PC geladen.
- Beim Return wird die Rücksprungadresse vom Stack geholt, in den PC geschrieben und in einem fetch-Zyklus die neue Instruktion geladen.

Der Erweiterte MU1 Befehlssatz (16 Befehle)

Load <S>
Store <S>
Add <S>
Sub <S>
Jump
Jump Grt./Equ.
Jump Not Equ.
Stop
Call
Return
Push
Pop
Load indiRect
Store indiRect
Move PC
Move SP

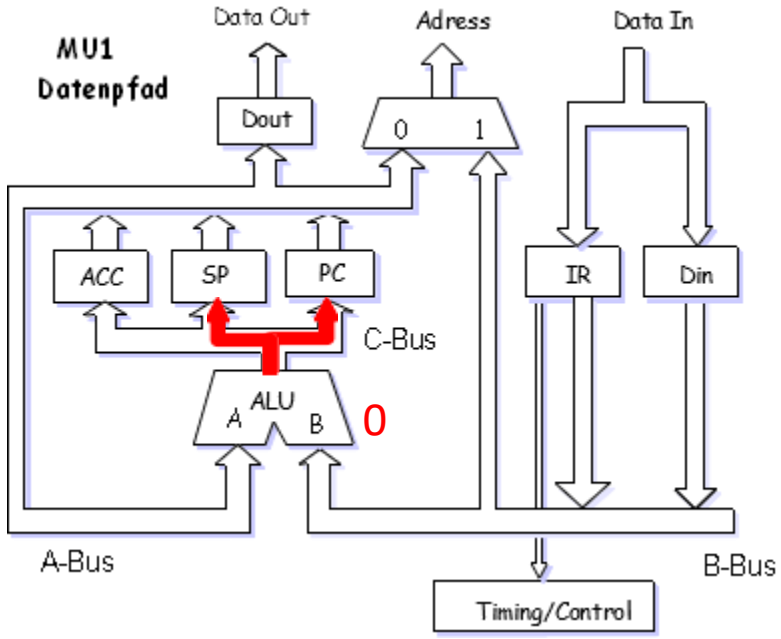
Instruction	Opcode	Effekt
Reset	xxxx	PC =0
LDA S	0000	ACC =[S]
STO S	0001	[S] =ACC
ADD S	0010	ACC =ACC +[S]
SUB S	0011	ACC =ACC - [S]
JMP S	0100	PC =S
JGE S	0101	IF N ==0 PC =S
JNE S	0110	IF Z ==0 PC =S
STOP	0111	stop
CALL S	1000	SP =SP-1; [SP] =PC; PC =S
RETURN	1001	PC =[SP], SP =SP +1
PUSH	1010	SP =SP-1; [SP] =ACC
POP	1011	ACC =[SP], SP =SP +1
LDR S	1100	DIN =[S]; DIN =[DIN]; ACC =DIN
STR S	1101	DIN =[S], DOUT =A; [DIN] =DOUT
MOV PC	1110	PC =ACC
MOV SP	1111	SP =ACC

Der Reset

- Jeder Prozessor hat einen Reset Eingang.
  Normalerweise ist dieser als /Reset also als „not Reset“ ausgeführt, so dass bei Anlegen einer 0 ein Reset ausgeführt wird.

- Liegt am /Reset Eingang eine 0 an, so wird der Prozessor auf einen Grundzustand zurückgesetzt: Program Counter und Stackpointer werden auf 0 gestellt und der Microprogramcounter „Step“ wird ebenfalls auf 0 gesetzt.

Der Reset: Datenpfad + Steuercodes



- Die ALU wird auf die Funktion Null gesetzt und PC und SP mit diesem Wert geladen.
- Der Counter Step in der Steuereinheit wird auf 0 gesetzt, so dass im nächsten Schritt ohne Reset ein Fetch von der Adresse 0 ausgeführt wird.

Steuertabelle (Reset)

Inputs						Outputs										
Instruction	Opcode	/Reset	Step	ACCz	ACC15	Step	Asel	Bsel	SPoe	SPie	PCoe	Pcie	Irie	ALUfs	MEMrq	R/nW
Reset	xxxx	0	x	x	x	0	x	x	0	1	0	1	0	0	0	1
Fetch	xxxx	1	0	x	x	1	0	x	0	0	1	1	1	A+1	1	1

Beschreibung der Microcodenotation durch einfache Sprache (I)

X = Y

Der Inhalt des Registers Y wird nach X verschoben.

X = Y op Z

Der Inhalt von Y (A-Bus der ALU) wird mit Z (B-Bus der ALU) verrechnet und nach X transportiert.

X = [Y]

Y wird als Adressinformation genommen. Der Inhalt der Speicherstelle Y wird nach X transportiert.

[X] = Y

X wird als Adressinformation genommen und Y wird in diese Adresse geschrieben.

Beschreibung des Microcodes durch einfache Sprache (II)

ACC = Din

- Der Inhalt von **Din** wird nach **ACC** transportiert, **ACC** und **Din** sind Registernamen.
- Im Register **Din** muss ***oe*** (Output enable) gesetzt sein, im Register **A** muss ***ie*** (Input enable) gesetzt sein.
- Da der Transport über die ALU erfolgt, muss ALUfunctionSelect **B** sein.
- Da der Speicher nicht angesprochen wird (keine [] enthalten), ist MEMrq 0 und RnW X (beliebig).

Din = [SP]

- Der Inhalt von SP wird als Adressinformation benutzt.
- Dazu muss der Adressmultiplexer auf 0 stehen.
- Im Register SP muss ***oe*** (Output enable) gesetzt sein im Register **Din** muss ***ie*** (Input enable) gesetzt sein.
- Da der Speicher angesprochen wird, muss MEMrq 1 sein, und da Din lesen möchte, muss RnW auch 1 sein.
- Da die ALU nicht benutzt wird, kann ALUfunctionSelect auf X gesetzt werden.

Beschreibung des Microcodes durch einfache Sprache (III)

PC = PC + 1

- ALUfunctionSelect muss **A+1** sein.
- Der Programmcounter wird auf den A-Bus gelegt und in der zweiten Hälfte des Taktes wird das Ergebnis der Berechnung wieder in den Programcounter zurückgeschrieben.
- ***oe*** und ***ie*** von **PC** müssen beide 1 sein.

Din = [SP], SP = SP+1

- Beide Operationen können in einem Takt durchgeführt werden, da die Adressierung des Speichers in der ersten Takthälfte erfolgt. Das Lesen durch Din erfolgt in der zweiten Takthälfte.

Regeln für MU1

[X] = Dout

- Wird ein Datenwert in den Speicher geschrieben, muss die rechte Seite des Befehls Dout sein.
- MEMrq = 1 und RnW = 0
- Der zu schreibende Wert muss vorher nach Dout gebracht werden.

X = [Y]

- Wird vom Speicher gelesen, muss die linke Seite (X) **IR** oder **Din** sein.
- MEMrq = 1 und RnW = 1

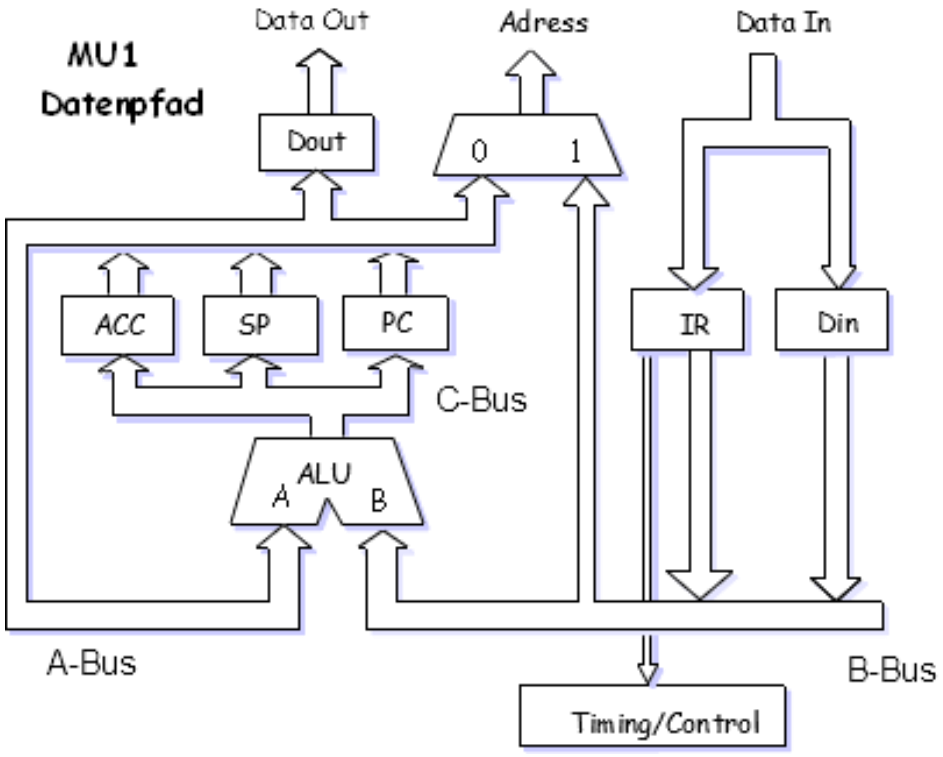
Keine Adresse

Wird in einem Befehl keine Adresse benutzt (keine [] , z.B. ACC=ACC+1), so ist

MEMrq = 0 und RnW = 1

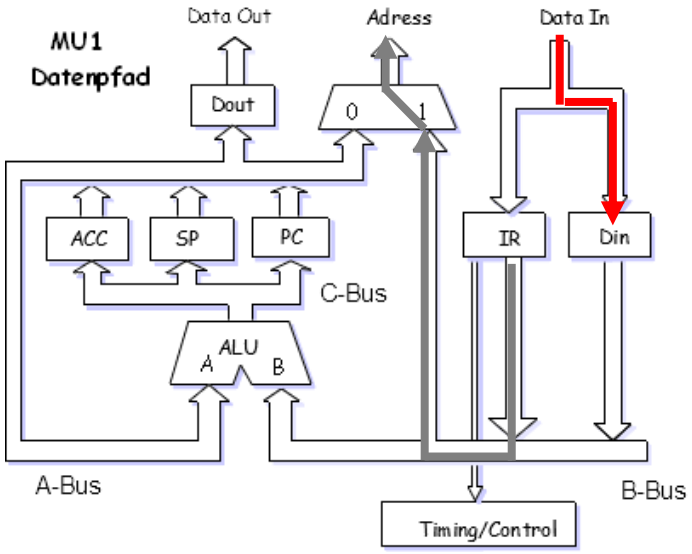
Nur ein Register darf jeweils auf einen Bus schreiben (A- oder B- Bus), aber alle dürfen mitlesen.

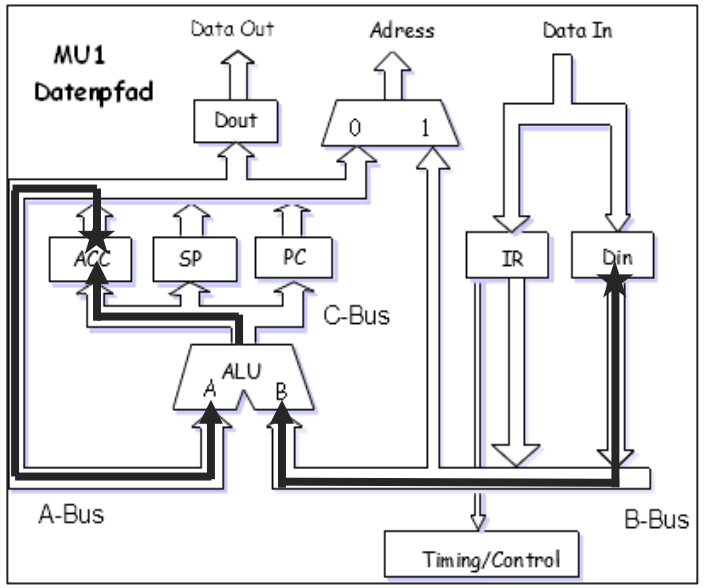
MU1-Befehl



Inputs						Outputs																	Description
Instruction	Opcode	/Reset	Step	ACC_Z / Zero	ACC_{15} / Negativ	Step	Adress	ACC_{OE}	ACC_{ie}	PC_{oe}	PC_{ie}	IR_{oe}	IR_{ie}	SP_{oe}	SP_{ie}	DIN_{oe}	DIN_{ie}	$DOUT_{oe}$	$DOUT_{ie}$	ALU Function	MEM_{rq}	RnW	

Die Addition bei MU1 (nach Fetch)

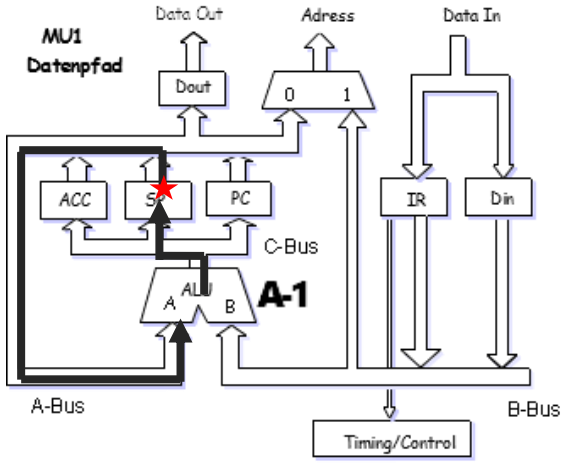


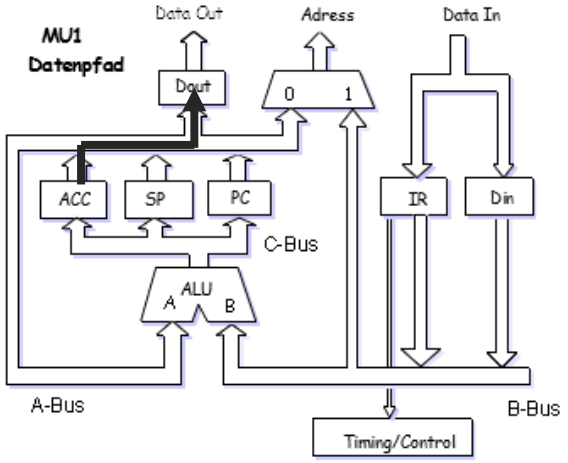


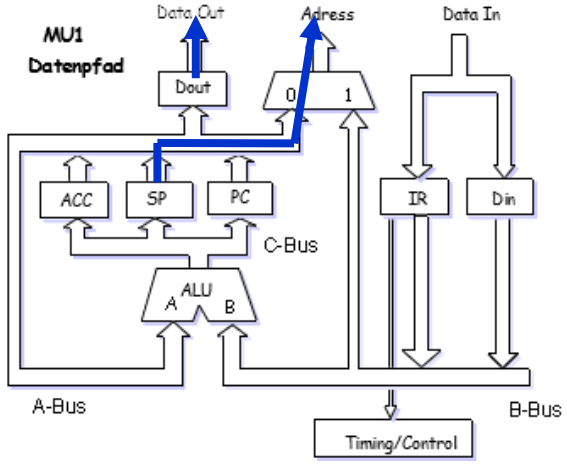
Das Instruction Register wird zur Adressierung benutzt und der Wert nach Din gebracht **Din = [IR].**
Der Inhalt des Akkummulators wird auf den A-Bus gelegt und der Inhalt von Din auf den B-Bus.
ALUfunction ist A+B,S (Summe und Statusflags)

Inputs						***Outputs***																***Funktion***
Instruction	Cycccce	/Reset	step	Z	N	step	Adress	ACCce	ACCle	PCce	PCle	IRce	IRle	SPce	SPle	DINce	DINle	DCLTce	DCLTle	ALU Function	MEMrq	RrW	
ADD S	0010	1	1	x	x	2	1	0	0	0	0	1	0	0	0	0	1	0	0	x	1	1	Din =[IR]
		1	2	x	x	0	x	1	1	0	0	0	0	0	0	1	0	0	0	A+B,S	0	1	ACC =ACC +Din

Die Operation PUSH ACC:
Der Akkumulator wird auf den Stack geschoben







Die Operation *Push* benötigt drei Schritte:

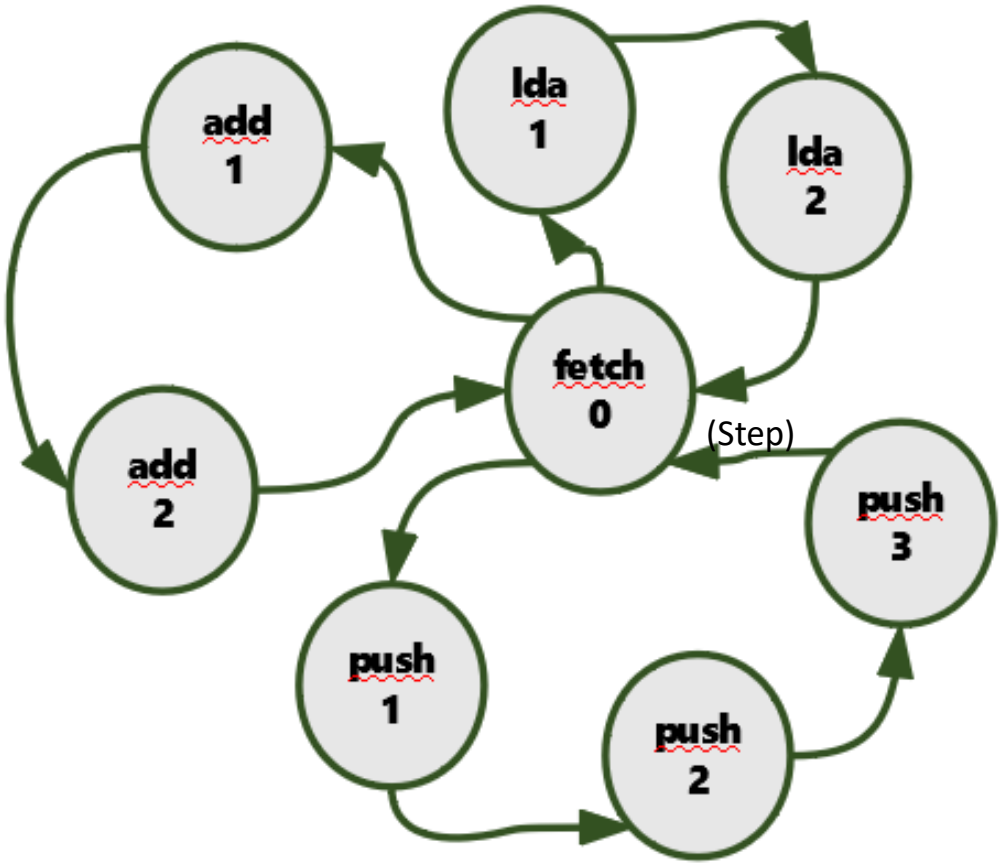
SP = SP -1

Dout = Acc

[SP] = Dout

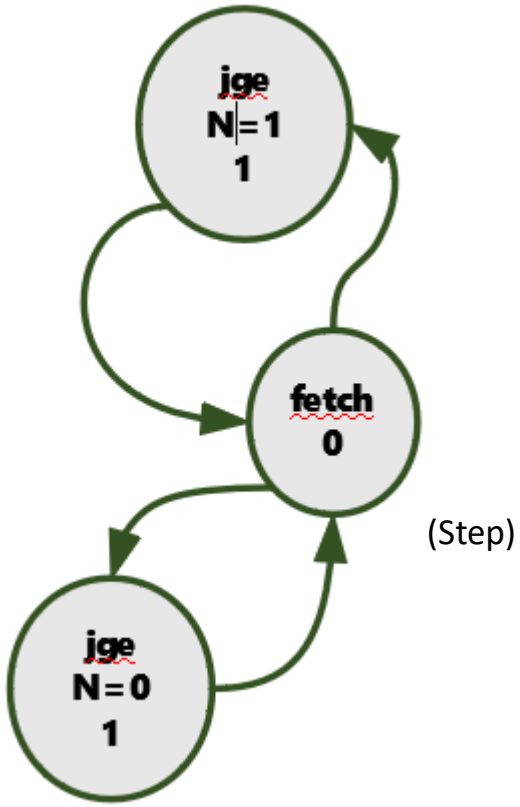
Im letzten Schritt wird der Micro Program Counter Step auf Null gesetzt, so dass sich ein Fetch-Zyklus anschließt.

Der MU1 Zustandsautomat (hier für lda, add und push)



- Nach dem Fetch des Opcodes wird jeweils der durch den Opcode vorgegebene Zustand angesprungen.
- Durch die Step-Variable (Program Counter im Microcode) wird eine Schleife durchlaufen, die wieder bei Schritt 0 im Fetch-Zyklus endet.

Bedingte Sprünge



(Step)

- Der bedingte Sprung **JGE** (Jump on Greater or Equal zero):
  In Abhängigkeit des Negative-Flags im Statusregister (entspricht Bit 15 des Akkumlators, ACC >= 0 oder ACC < 0) wird ein unterschiedlicher Zustand angesprungen:

- Ist N = 0 (ACC >= 0) dann wird die Operation PC = IR ausgeführt.
- Ist N = 1 (ACC < 0) so wird keine Operation ausgeführt (NOP).

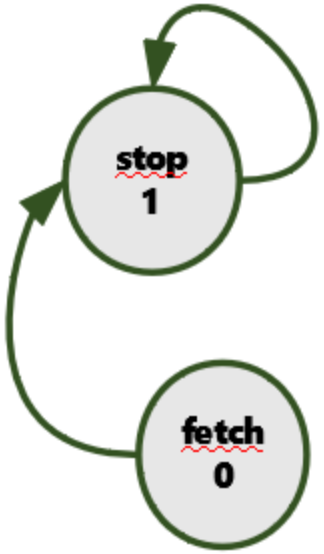
- Der bedingte Sprung **JNE** (jump on not equal) ist strukturell gleich, nur ist hier die Zusatzbedingung, dass der Accumulator Null (Zero-Flag = 1) ist.

MU1 Steuer-Logik für bedingte Sprünge

- Beide Befehle (s.u.) weisen jeweils 2 Zustände auf, die sich nur durch die Statusbits des Akkumulators (Inputs) unterscheiden.
- Beide Zustände haben die Schrittnummer 1 und als Folgeschritt die Schrittnummer 0.
- Die Funktion NOP ist dadurch gekennzeichnet, dass kein Registerinhalt verändert wird:
  - Alle Register haben ***oe*** und ***ie*** auf 0
  - und der Speicher wird nicht angesprochen: MEMrq = 0

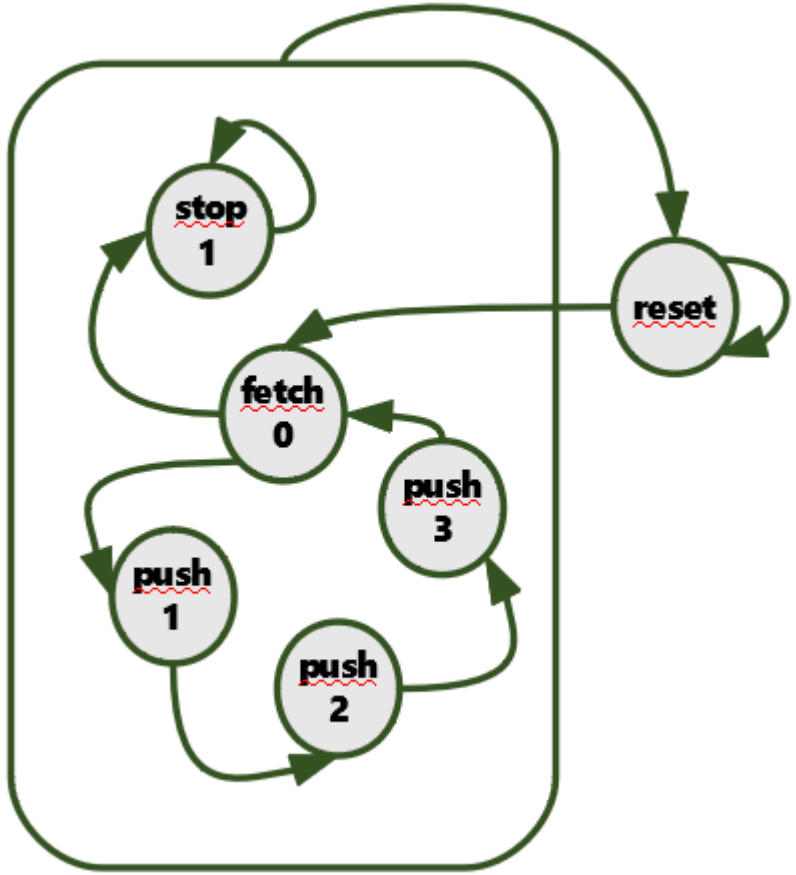
Inputs						Outputs																Funktion
Instruction	Opcode	/Reset	step	Z	N	step	Adress	ACCce	ACCie	PCce	PCie	IRce	IRie	SPce	SPie	DINce	DINie	DCLTce	DCLTie	ALU Function	MEMrq	RnW	
JGE S	0101	1	1	x	0	0	x	0	0	0	1	1	0	0	0	0	0	0	0	B	0	1	if ACC >= 0 then PC = IR
		1	1	x	1	0	x	0	0	0	0	0	0	0	0	0	0	0	0	x	0	1	If ACC < 0 then NOP
JNE S	0110	1	1	0	x	0	x	0	0	0	1	1	0	0	0	0	0	0	0	B	0	1	if ACC != 0 then PC = IR
		1	1	1	x	0	x	0	0	0	0	0	0	0	0	0	0	0	0	x	0	1	If ACC == 0 then NOP

Der Stop-Befehl



- Der Stop-Befehl hat als Folgezustand den Zustand *Stop*.
- Es wird kein Fetch-Zyklus mehr ausgeführt, so dass die Zustandsmaschine in dieser Stellung angehalten wird.

Reset



- Ist das Reset-Signal angelegt, geht unsere Zustandsmaschine immer in den Zustand *Reset*
- Im Reset Zustand:
  - werden Program Counter und SteckPointer auf Null gesetzt: PC = 0, SP=0
  - und die Zustandsvariable Step (der Microprogram Counter) erhält ebenfalls den Wert 0
- Dadurch wird nach Beendigung des Reset Signals mit dem Zustand fetch fortgefahren und der Befehl von der Adresse 0 (PC) geholt