ADSP

Programmstrukturen
Auszüge aus
ADSP-2106x Sharc Users Manual
Analog Devices, Inc.

Programmstrukturen

Der normale Programmfluss ist linear.

Befehle werden sequentiell abgearbeitet.

Abweichungen vom linearen Programmfluss sind:

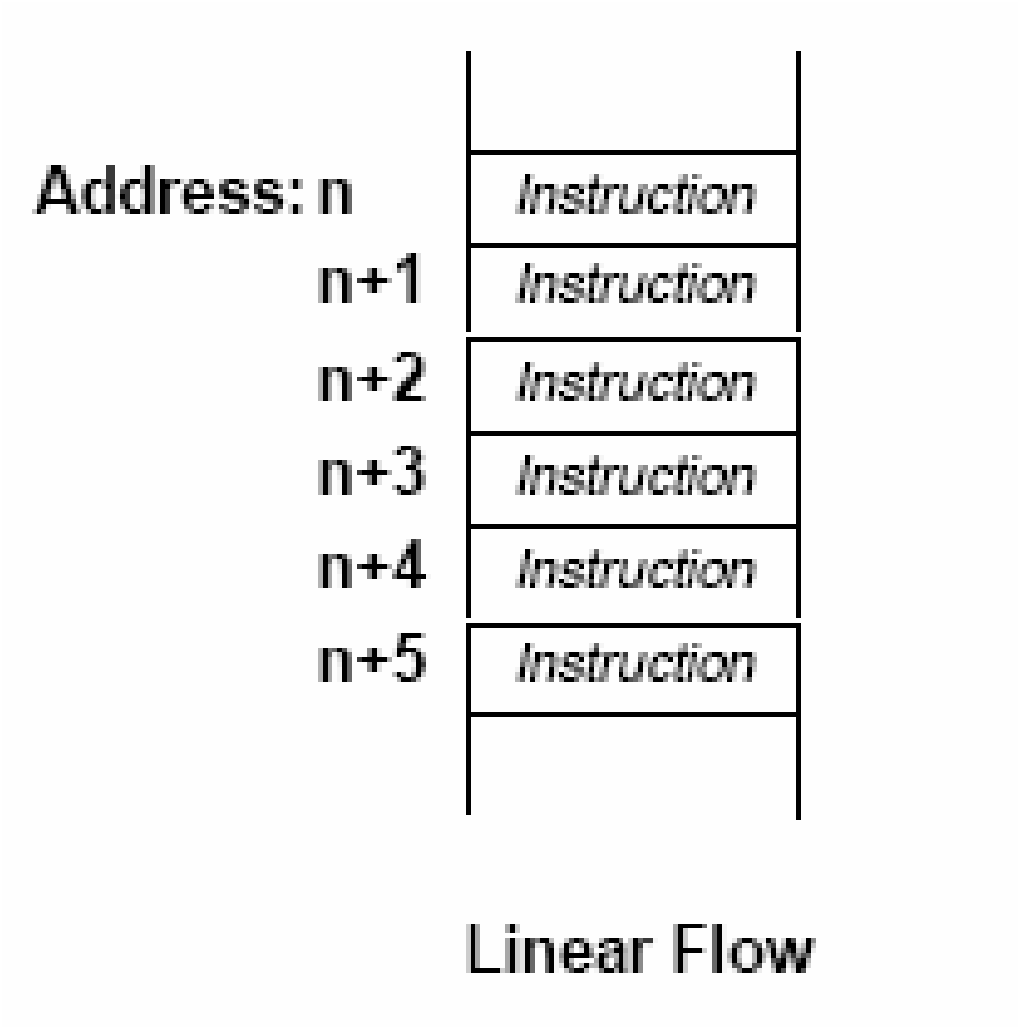
- Schleifen (Loops)
- Programmverzweigungen (bedingte Sprünge)
- Unterprogramme (Subroutines)
- Sprünge (unbedingte Sprünge)
- Programmunterbrechung (Interrupts)
- warten auf Interrupt (Idle)

Programmstrukturen

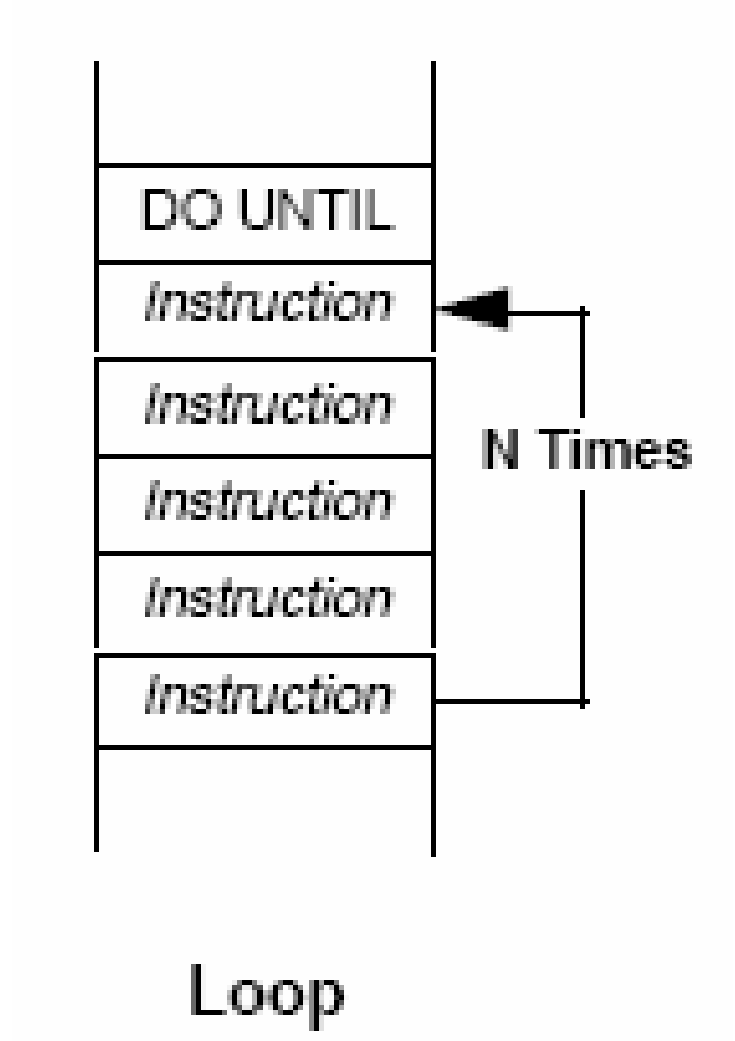
Programmstrukturen realisiert der *Program Sequencer*.

- Ermittlung der Adresse des nächsten Befehls incrementing the fetch address,
- Stackverwaltung,
- Berechnung von Bedingungen (conditions)
- Verwaltung des Schleifenzählers (decrementing the loop counter)
- Interruptbehandlung (handling interrupts)

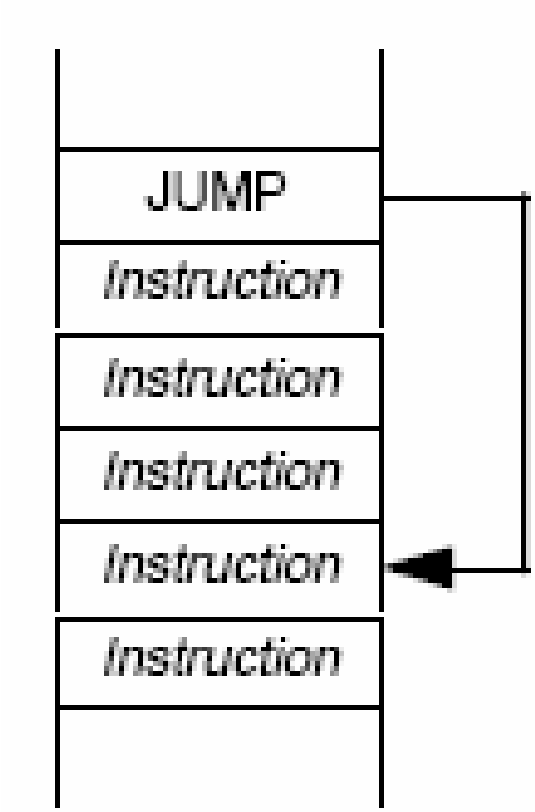
Programmstrukturen



Programmstrukturen

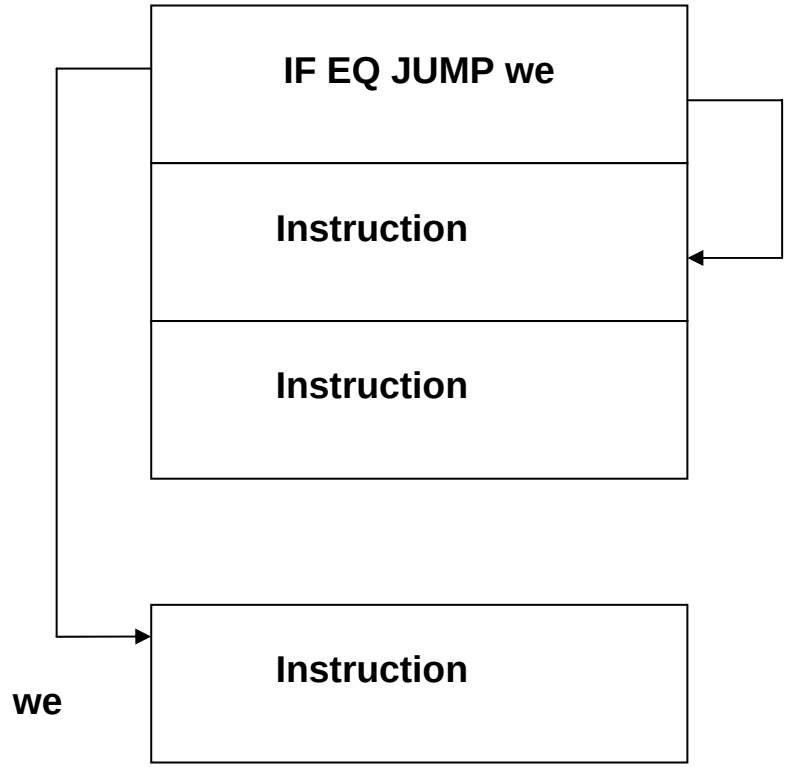


Programmstrukturen



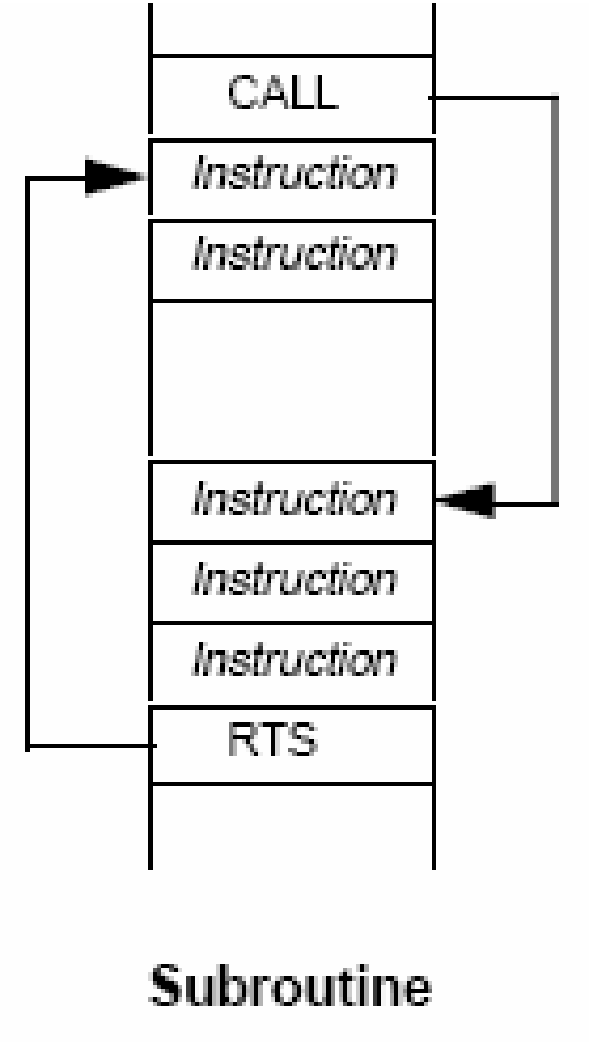
Jump

Programmstrukturen

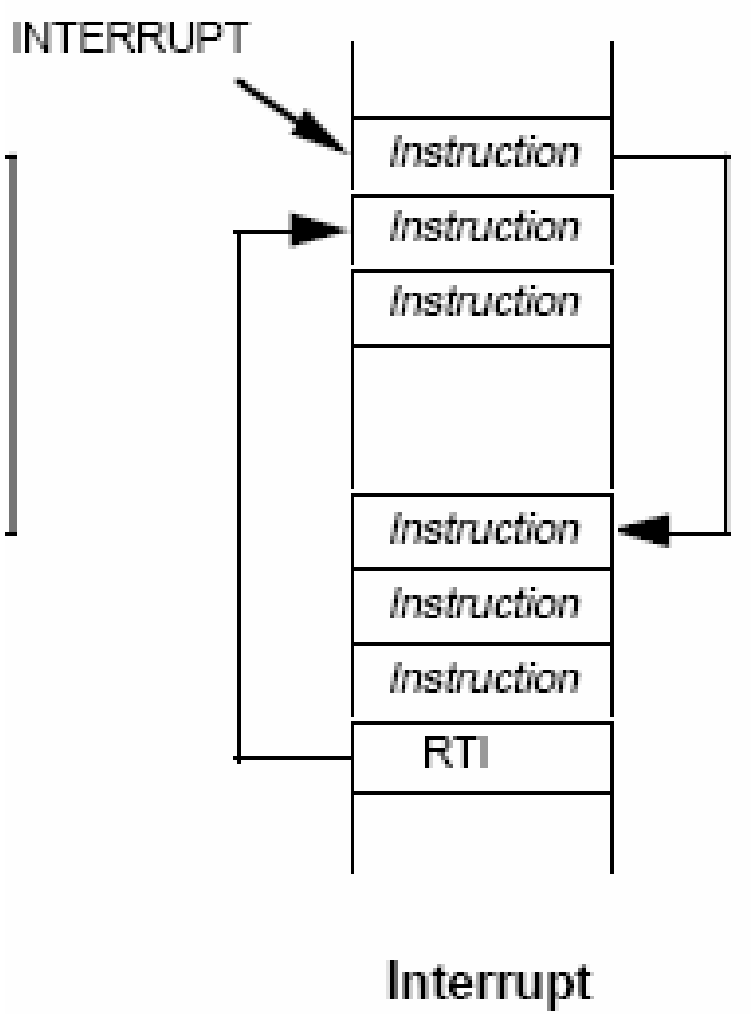


Conditional Jump

Programmstrukturen

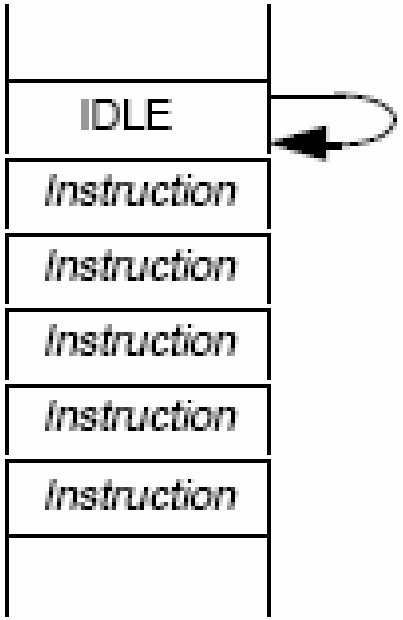


Programmstrukturen



Interrupt

Programmstrukturen



Idle

Befehlszyklen

Befehlszyklen

- Der ADSP -2106 x verarbeitet Anweisungen in drei Taktzyklen:
- Im Befehlslesezyklus liest der ADSP -2106 x die Anweisung entweder aus on chip Befehlscache oder aus dem Programmspeicher
- Während der Dekodierung werden die Steuerbedingungen für die Ausführung eines Befehl gebildet
- Im Ausführungszyklus wir der Befehl ausgeführt

Befehlszyklen

Befehlszyklen

- Der ADSP -2106 x verfügt über eine dreistufige Pipeline
- In jedem Zyklus wird
  - 1 Befehl gelesen
  - 1 Befehl dekodiert
  - 1 Befehl ausgeführt

Befehlszyklen

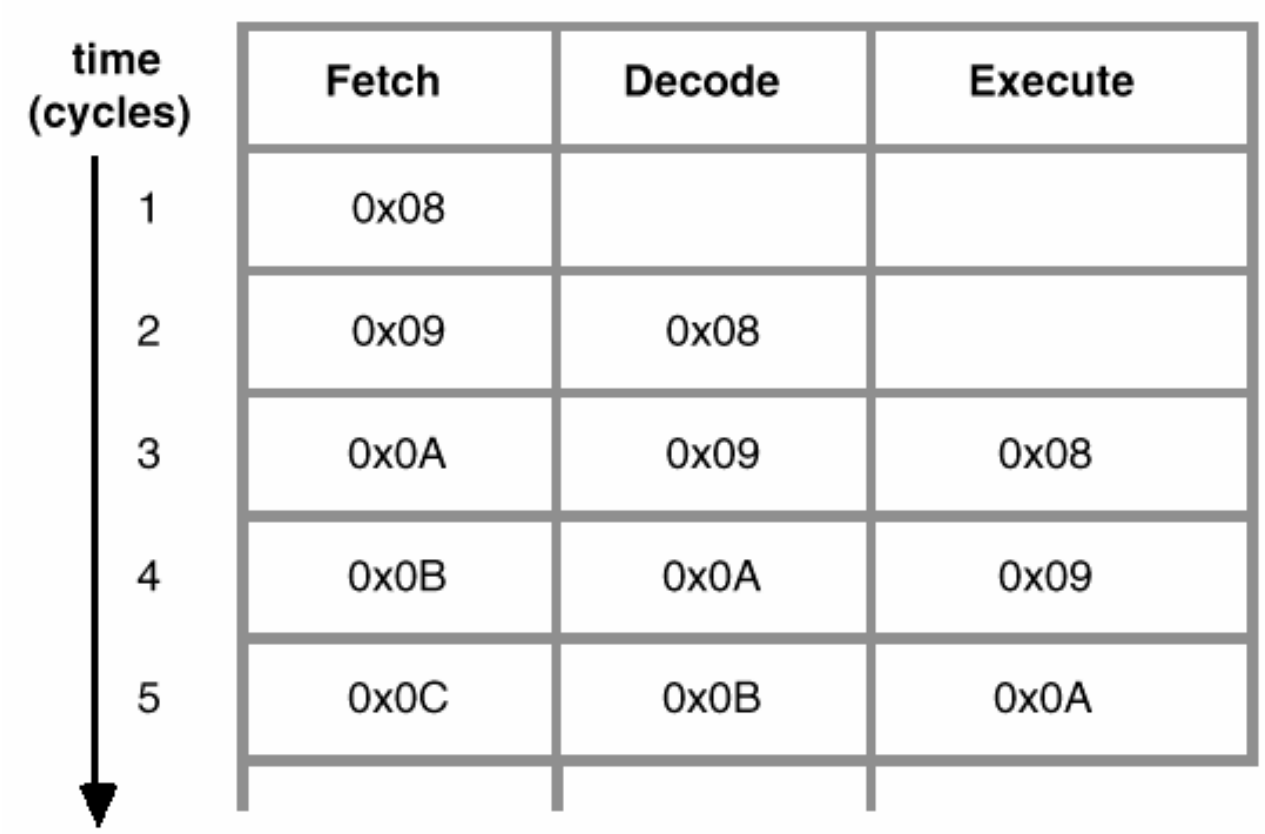
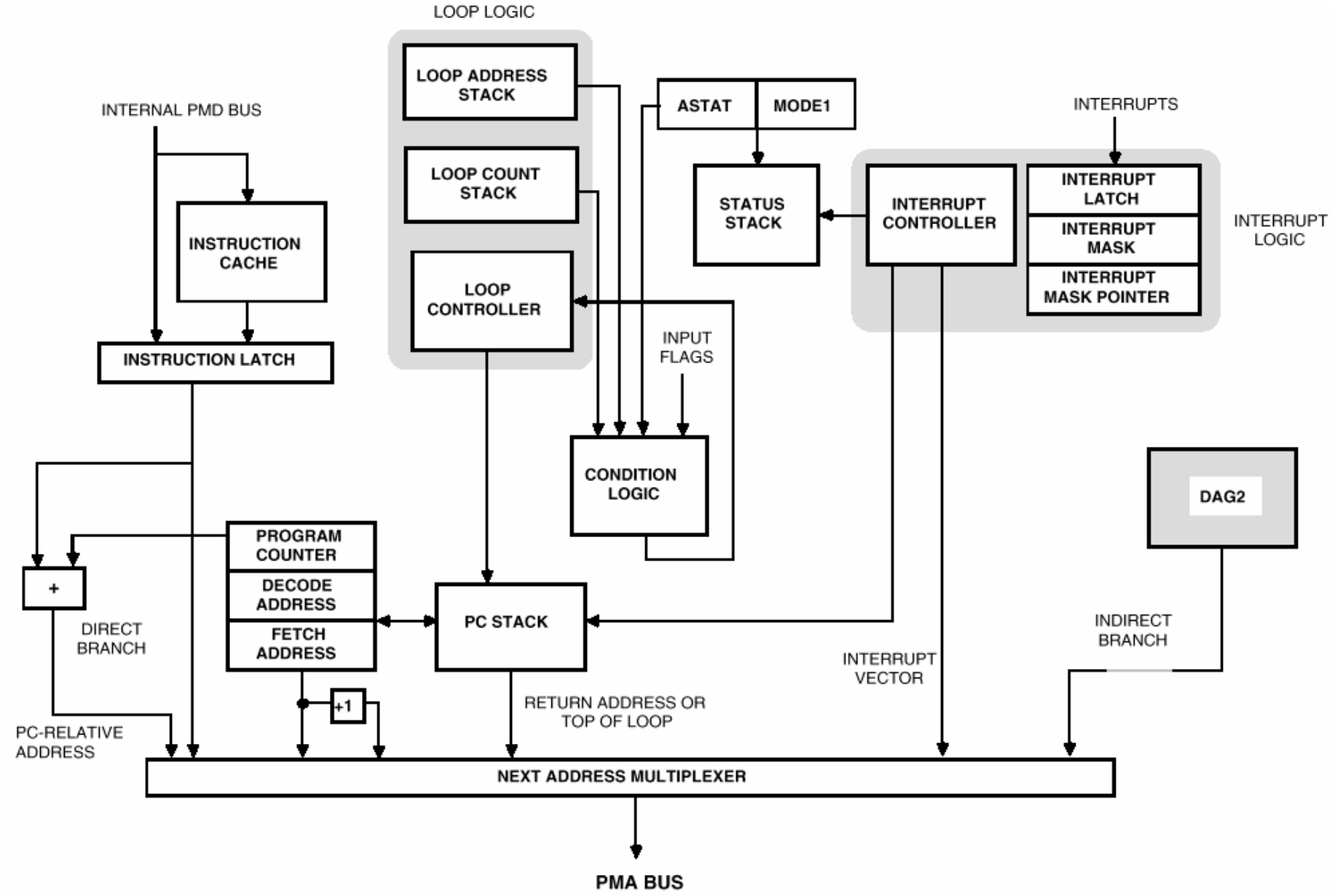


Figure 3.2 Pipelined Execution Cycles

Befehlszyklen

- **Jeder nichtsequentielle Programmfluß kann die Prozessorleistung beeinflussen.**
- **nichtsequentieller Programmfluß kann**
  - **Datentransfer zum Programmspeicher**
  - **Jumps**
  - **Subroutine Calls und Returns**
  - **Interrupts und Returns**
  - **Loops**

Sequenzer


LOOP LOGIC
LOOP ADDRESS STACK
LOOP COUNT STACK
LOOP CONTROLLER
ASTAT
MODE1
STATUS STACK
INTERRUPTS
INTERRUPT CONTROLLER
INTERRUPT LATCH
INTERRUPT MASK
INTERRUPT MASK POINTER
INTERRUPT LOGIC
INTERNAL PMD BUS
INSTRUCTION CACHE
INSTRUCTION LATCH
INPUT FLAGS
CONDITION LOGIC
DAG2
PROGRAM COUNTER
DECODE ADDRESS
FETCH ADDRESS
+
+1
PC STACK
DIRECT BRANCH
PC-RELATIVE ADDRESS
RETURN ADDRESS OR TOP OF LOOP
INTERRUPT VECTOR
INDIRECT BRANCH
NEXT ADDRESS MULTIPLEXER
PMA BUS

Bedingungen

No.	*Mnemonic*	*Description*	*True If*
0	EQ	ALU equal zero	AZ = 1
1	LT	ALU less than zero	
2	LE	ALU less than or equal zero	
3	AC	ALU carry	AC = 1
4	AV	ALU overflow	AV = 1
5	MV	Multiplier overflow	MV = 1
6	MS	Multiplier sign	MN = 1

Bedingungen

No.	*Mnemonic*	*Description*	*True If*
7	SV	Shifter overflow	SV = 1
8	SZ	Shifter zero	SZ = 1
9	FLAG0_IN	Flag 0 input	FI0 = 1
10	FLAG1_IN	Flag 1 input	FI1 = 1
11	FLAG2_IN	Flag 2 input	FI2 = 1
12	FLAG3_IN	Flag 3 input	FI3 = 1
13	TF	Bit test flag	BTF = 1

Bedingungen

No.	*Mnemonic*	*Description*	*True If*
15	LCE	Loop counter expired (DO UNTIL term)	CURLCNTR = 1
15	NOT LCE	Loop counter not expired (IF cond)	CURLCNTR ≠ 1

Bedingungen

No.	*Mnemonic*	*Description*	*True If*
17	GE	ALU greater than or equal zero	
18	GT	ALU greater than zero	
19	NOT AC	Not ALU carry	AC = 0
20	NOT AV	Not ALU overflow	AV = 0
21	NOT MV	Not multiplier overflow	MV = 0
22	NOT MS	Not multiplier sign	MN = 0
23	NOT SV	Not shifter overflow	SV = 0
24	NOT SZ	Not shifter zero	SZ = 0
31	FOREVER	Always False (DO UNTIL) always	
31	TRUE	Always True (IF) always	

BRANCHES (CALL, JUMP, RTS, RTI)

Parameter zur Spezifikation für *branches*:

- JUMP, CALL und RET können bedingt und unbedingt ausgeführt werden
- JUMP, CALL Adressen können indirekt, direkt, or PC-relativ sein
  - indirekte Adresse steht in einem der Programmspeicheradressgeneratoren DAG2
  - Direkte 24 Bit Adresse steht im Befehl
  - PC-relative Adresse im Befehl steht der Offset die Speicheradresse wird gebildet aus PC + Offset

BRANCHES (CALL, JUMP, RTS, RTI)

Parameter zur Spezifikation für *branches*:

- JUMP, CALL und RET verzögert oder nicht verzögert sein
- Bei verzögerter Ausführung werden die beiden Befehle nach dem Sprung noch ausgeführt
- Bei nicht verzögerter Ausführung werden die beiden Befehle nach dem Sprung nicht ausgeführt stattdessen werden NOPs ausgeführt
- JUMP (LA) verursacht einen automatischen Abbruch einer Schleife der PC- und Loop Address stack werden aktualisiert

JUMP (LA) may not be used in the
last three instructions of a loop.)

Delayed & Nondelayed Branches

NON-DELAYED JUMP OR CALL

CLOCK CYCLES ⟶

Execute Instruction	n	nop	nop	j
Decode Instruction	n+1->nop	n+2->nop	j	j+1
Fetch Instruction	n+2	j	j+1	j+2

n+1 suppressed

n+2 suppressed; for call, n+1 pushed on PC stack

Delayed & Nondelayed Branches

NON-DELAYED RETURN

CLOCK CYCLES ⟶

Execute Instruction	n	nop	nop	r
Decode Instruction	n+1->nop	n+2->nop	r	r+1
Fetch Instruction	n+2	r	r+1	r+2

n+1 suppressed

n+2 suppressed; r popped from PC stack

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Delayed & Nondelayed Branches

DELAYED JUMP OR CALL

CLOCK CYCLES →

Execute Instruction	n	n+1	n+2	j
Decode Instruction	n+1	n+2	j	j+1
Fetch Instruction	n+2	j	j+1	j+2

for call, n+3 pushed on PC stack

Delayed & Nondelayed Branches

DELAYED RETURN

CLOCK CYCLES →

Execute Instruction	n	n+1	n+2	r
Decode Instruction	n+1	n+2	r	r+1
Fetch Instruction	n+2	r	r+1	r+2

r popped from PC stack

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Delayed & Nondelayed Branches

- Wegen der Anweisungspipeline müssen ein verzögerterSprung und die zwei folgenden Anweisungen sequentiell ausgeführt werden.

Delayed & Nondelayed Branches

- Nach einem verzögerten Sprung dürfen folgende Anweisungen nicht folgen:
  - Other Jumps, Calls or Returns
  - Pushes or Pops of the PC stack
  - Writes to the PC stack or PC stack pointer
  - DO UNTIL instruction
  - IDLE or IDLE16 instruction

PC Stack

The PC stack holds return addresses for

- subroutines
- interrupt service routines
- top-of-loop addresses for loops.
- The PC stack is 30 locations deep
- A PC stack interrupt occurs when 29 locations of the PC stack are filled

LOOP

```
        LCNTR=30, DO label UNTIL LCE;
        R0=DM(I0,M0), F2=PM(I8,M8);
        R1=R0-R15;
label:  F4=F2+F3;
```

Branches

Program Flow Control Instructions

IF condition JUMP <addr24> *(DB)* ;

(PC, <reladdr24>) *(LA)*

(CI)

(DB,LA)

(DB,CI)

IF condition CALL <addr24> *(DB)* ;

(PC, <reladdr24>)

Branches

Program Flow Control Instructions

IF condition JUMP (Md,Ic) *(DB)*;

(PC, <reladdr6>) *(LA)*

(CI)

(DB,LA)

(DB,CI)

, *compute ELSE compute*

IF condition CALL (Md,Ic) *(DB)* , *compute* ;

(PC, <reladdr6>) *ELSE compute*

Branches

Program Flow Control Instructions

IF condition RTS *(DB)*

(LR)

(DB,LR)

, compute ELSE compute;

IF condition RTI *(DB)* *, compute ELSE compute*;

Branches

Program Flow Control Instructions

LCNTR = <data16> , DO <addr24> UNTIL LCE ;

ureg (PC, <reladdr24>)

DO <addr24> UNTIL termination ;

(PC, <reladdr24>)

Beispiele für Programmstrukturen

Beispielprogramm

- Programmverzweigung
- Programmschleifen
- Unterprogramm

Beispiele für Programmstrukturen

```
.SECTION /dm seg_dmda;
.VAR dminput[2]=0,1;
.VAR dmvek[6];

.SECTION /pm seg_pmda;
.VAR pmvek[6]=1,2,3,4,5,6;

.SECTION /pm seg_rth;
        nop;
        nop;
        nop;
        nop;
        nop;
        jump start;
```

Beispiele für Programmstrukturen

```
.SECTION /pm seg_pmco;
start:      B0=dminput;
            M0=1;
            L0=@dminput;
            B8=pmvek;
            M8=1;
            L8=@pmvek;
/* Programmverzweigung */
            R1=0;
verzw:      R0=DM(I0,M0);
            R0=PASS R0;                 // Flags aktualisieren
            IF NE JUMP zweig2;          // bedingte Verzweigung
            R1=R1+1;
            JUMP verzw;
```

Beispiele für Programmstrukturen

```
/* Programmschleife
mit definierter Anzahl von Umläufen
Kopieren des Vektors pmvek nach dmvek */
zweig2:     B8=pmvek;       // Quell-Vektor adressiern
            M8=1;
            L8=@pmvek;
            B0=dmvek;       // Ziel-Vektor adressiern
            M0=1;
            L0=@dmvek;
cslbeg:     LCNTR = L8, DO cschleife until LCE;
            R0=PM(I8,M8);
cschleife:  DM(I0,M0)=R0; // Schleifenende
```

Beispiele für Programmstrukturen

```
/* Programmschleife mit Abbruchbedingung
in einem Vektor wird ein Element bestimmter Größe
gesucht. Das Programm läuft solange in einer Schleife,
bis der gesuchte Wert gefunden wurde
*/
        B0=dmvek;               // Vektor adressiern
        M0=1;
        L0=@dmvek;
        R0=0;
        R0=PASS R0;             // Flags aktualisieren
        R1=3;                   // zu suchender Wert
```

Beispiele für Programmstrukturen

```
/* Programmschleife mit Abbruchbedingung
doschlbeg:      do doschleife until EQ;
                R0=DM(I0,M0);
                COMP(R0,R1);
                IF EQ JUMP ende;
                R4=I0;          // Test Vektorende erreicht
                R5=B0;
                R0=-1;
                COMP (R4,R5);
ende:
                NOP;            // Verzögerung erforderlich
                NOP;
doschleife:     NOP;
                R3=R0;          // gefundener Wert in R0 und R3
                                // R0=-1 Wert nicht geunden
                idle;
```

Beispiele für Programmstrukturen

```
/*      Programm zur Demonstration der Schleifen
   Schleife mit einer festen Anzahl von Umläufen
   Schleife mit Abbruchbedingung
   Programmentwurf:      H.W.W
   Datum:                11.04.2008
   Tool:                 Visual DSP ++
*/
```

Beispiele für Programmstrukturen

```
.SECTION /dm seg_dmda;
.VAR dmVectori[]= 1,2,3,4,5,6,7,8,9,111;
.VAR dmVectorz[6]= 11,22,33,44,55,66;

.SECTION /pm seg_pmda;
.VAR pmVectori[12];
.VAR pmVectorz[6];
```

Beispiele für Programmstrukturen

```
.SECTION /pm seg_rth;
        nop;
        .align 0x4;
        nop;
        jump start;
```

Beispiele für Programmstrukturen

```
.SECTION /pm seg_pmco;
/*   1.Teil
     Kopieren des Vektors dmVectorz nach pmVectorz  */
start:      B0=dmVectorz;
            L0=@dmVectorz;
            M0=1;
            B8=pmVectorz;
            L8=@pmVectorz;
            M8=1;
            R1=@dmVectorz;
            R1=R1-1;
            R0=DM(I0,M0);
```

Beispiele für Programmstrukturen

```
                R0=DM(I0,M0);
zloopa:         LCNTR = R1, DO zloope UNTIL LCE;
zloope:         R0=DM(I0,M0),PM(I8,M8)=R0;
                PM(I8,M8)=R0;
```

Beispiele für Programmstrukturen

```
Kopieren des Vektors dmVectori nach pmVectori
Kopiert werden alle Elemente, bis von dmVectori
der Wert 0 gelesen wird   */
        B0=dmVectori;
        M0=1;
        L0=@dmVectori;
        B8=pmVectori;
        M8=1;
        L8=@pmVectori;
        R1=0;
        R2=111;
```

Beispiele für Programmstrukturen

```
aloopa:   DO aloope UNTIL EQ;
          R0=DM(I0,M0);
          IF NE PM(I8,M8)=R0;
          IF NE R1=R1+1;
aloope:   COMP (R0,R2);
          L8=R1;
```

Beispiele für Programmstrukturen

/* 3.Teil

Kopieren des Vektors dmVectori nach pmVectori Kopiert werden alle Elemente, bis von dmVectori der Wert 111 gelesen wird. Die Scleife wird abgebrochen mit JUMP *marke* (LA)

*/

Beispiele für Programmstrukturen

```
B0=dmVectori;
M0=1;
L0=@dmVectori;
B8=pmVectori;
M8=1;
L8=@pmVectori;
R1=0;
R2=111;
```

Beispiele für Programmstrukturen

```
abrtla:     DO abrtle UNTIL EQ;
            IF EQ JUMP nachl (LA);
            R0=DM(I0,M0);
            PM(I8,M8)=R0;
            R1=R1+1;
abrtle:     COMP (R0,R2);
nachl:      L8=R1;
            idle;
```