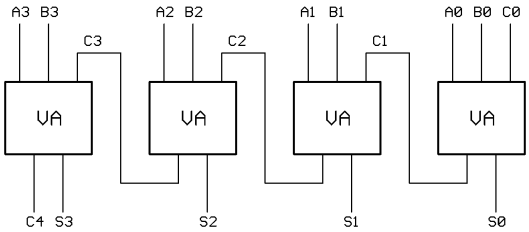
Aufgabe 1:

a) Auf Grund der Klammer hat der in der Aufgabenstellung gegebene Ausdruck ein Zeitbedarf von 3 *ns* (1 *ns* für den Klammerausdruck + 1 *ns* für die beiden UND−Schaltglieder + 1 *ns* für das abschließende ODER−Schaltglied). Multipliziert man die Klammer aus, dann wird der Zeitbedarf auf 2 *ns* reduziert. Dieser Zeitbedarf ist minimal und man erhält die folgende Gleichung:
$c_{i+1} = a_i \wedge b_i \vee c_i \wedge (a_i \vee b_i) = a_i \wedge b_i \vee c_i \wedge a_i \vee c_i \wedge b_i$

b) Ein Ripple−Carry Addierer ist aus einzelnen Volladdierern ausgebaut, deren Carryausgänge kaskadiert werden. Die Struktur eines 4−Bit Ripple−Carry Addierers ist somit:



c) Da nach Teilaufgabe a) ein Übertrag erst nach 2 *ns* gütig ist, benötigt ein 32−Bit Ripple−Carry Addierer für jede Addition $t_c = 32 \cdot 2\,ns = 64\,ns$, bis der auslaufende Übertrag c_{32} gültig ist.

Der Zeitbedarf ist allerdings höher. Der Übertrag c_{31} braucht 62 *ns*, bis er gültig ist. Erst danach kann dieser Übertrag verwendet werden, um das höchstwertigste Bit des Ergebnisses zu bestimmen. Dazu dient ein VA, der noch einmal 3 *ns* braucht, bis er das Ergebnis ausgeben kann. Der Zeitbedarf für eine Addition ist somit 65 *ns*.

d) Aus $c_{i+1} = g_i \vee c_i \wedge p_i = g_i + c_i p_i$ erhält man nacheinander die folgenden Gleichungen:

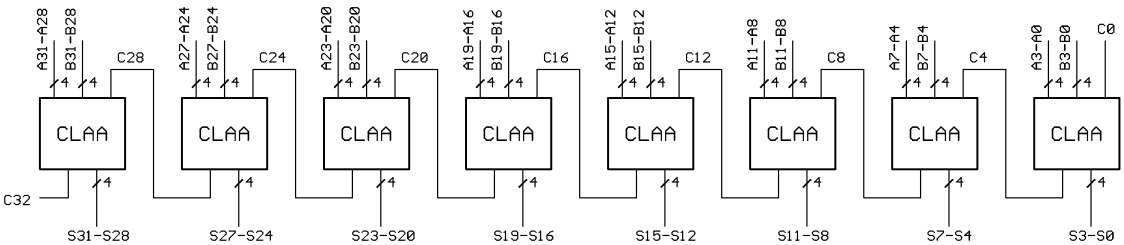
$$c_0 = c_0$$
$$c_1 = g_0 + c_0 p_0$$
$$c_2 = g_1 + c_1 p_1 = g_1 + (g_0 + c_0 p_0) p_1 = g_1 + g_0 p_1 + c_0 p_0 p_1$$
$$c_3 = g_2 + c_2 p_2 = g_2 + (g_1 + g_0 p_1 + c_0 p_0 p_1) p_2 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$
$$c_4 = g_3 + c_3 p_3 = g_3 + (g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2) p_3 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$

Der einlaufende Übertrag c_0 muss in diesen Gleichungen mit berücksichtigt werden, da der *Carry Look Ahead Generator* ansonsten nicht kaskadiert werden kann.

Die Berechnung der g_i und p_i kann in jeweils 1 *ns* erfolgen. Die Berechnung der c_i braucht weitere 2 *ns* (1 *ns* für die UND−Schaltglieder + 1 *ns* für das OR−Schaltglied). Der auslaufende Übertrag c_4 steht also, wie jede andere Übertrag auch, nach 3 *ns* bereit.

Setzt man für g_i und p_i die Definition dieser Signale ein, dann kann man nochmals 1 *ns* sparen.

e) Der gesuchte 32−Bit Addierer aus 4−Bit Carry−Look−Ahead Addierern ist im Prinzip nicht anders aufgebaut, als der 4−Bit Ripple−Carry Addierer aus Teil b). Die folgende Skizze gibt seinen Aufbau an.



Da jeder Übertrag nach 3 *ns* (bzw. 2 *ns*) berechnet wird, dauert es $t_e = 8 \cdot 3ns = 24ns$ (bzw. 16 *ns*) bis der Übertrag c_{32} zur Verfügung steht. Bezieht man die Ausgänge s_{31} bis s_{28} mit in die Abschätzung des Zeitbedarfs mit ein, dann dauert auch bei Verwendung von DNFs (1 *ns* für Negierung der Eingänge + 1 *ns* für die UND−Schaltglieder + 1 *ns* für das OR−Schaltglied) mindestens 3 *ns* bis der jeweilige Ausgang gültig ist.
Betrachtet man nur die Übergänge, dann beträgt die Leistungssteigerung:

$$\frac{t_c}{t_e} = \frac{64ns}{24ns} = \frac{8}{3} = 2,\overline{6}$$

Für die anderen möglichen Zeiten erhält man ähnliche Werte.

f) Wenn genau vier 4−Bit Carry−Look−Ahead Addierer zur Verfügung stehen, um einen 16−Bit Addierer zu konstruieren, dann muss jeder einzelne Addierer vier Bit der beiden Summanden addieren. Auf diese Art und Weise erhält man einen Addierer analog zu Teil e), nur diesmal mit vier Addierern. Der noch verfügbare (4−Bit) CLAG kann nun benutzt werden, um diesen 16 Bit Addierer zu beschleunigen.
Die vier CLAA berechnen intern natürlich die gleichen Signale wie ein CLAG, sonst wären es ja keine Carry−Look−Ahead Addierer. Definiert man nun die folgenden Hilfssignale für jeden CLAA:

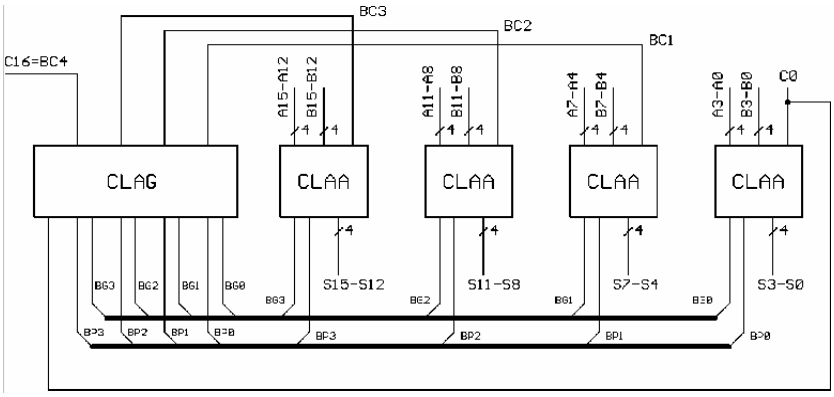
$$BG_i = g_{4i+3} + g_{4i+2}p_{4i+3} + g_{4i+1}p_{4i+2}p_{4i+3} + g_{4i}p_{4i+1}p_{4i+2}p_{4i+3}$$

$$BP_i = p_{4i}p_{4i+1}p_{4i+2}p_{4i+3}$$

dann man die Überträge der vier CLAA darstellen als:

$$c_{4i+4} = BG_i + c_{4i}BP_i$$

Diese Formel ist identisch zu der Formel aus Teil d) für die Überträge. Nun ist klar, wie man den CLAG verwenden muss. Man verwendet anstelle der g_i und p_i die BG_i und BP_i, um die gewünschte Beschleunigung der Überträge zu erreichen. Die vom CLAG berechneten Überträge werden den einzelnen CLAAs zugeführt. Die Schaltung dazu sieht dann wie folgt aus:



Aufgabe 2:

Die Eingänge A_{reg}, B_{reg} und F_{reg} sind Indizes für die 16 Register, darum haben sie auch 4 Leitungen. Das Register `r7` kann so z. B. ausgewählt werden, indem A_{reg} mit dem Wert 7 belegt wird. Steuersignale müssen laut Aufgabenstellungen erst eingetaktet werden, bevor sie gültig sind. Wenn man also `C` löschen oder setzen will, muss danach das Schlüsselwort `CLOCK` verwendet werden.

a) Mikroprogramm für `MOV r0, r1`:

```
Areg ¬ 1,
Freg ¬ 0,
SW ¬ 1111.1,
CLOCK
GOTO FETCH
```

b) Mikroprogramm für `ADD r2, r3, r4`:

```
C ¬ 0,
CLOCK
Freg ¬ 2,
Areg ¬ 3,
Breg ¬ 4,
SW ¬ 1001.0,
CLOCK
GOTO FETCH
```

c) Mikroprogramm für `CMP r0, r1, r2`:

```
r0 ¬ 1,
C ¬ 1,
CLOCK
Areg ¬ 2,
Breg ¬ 1,
SW ¬ 0110.0,              // Freg = r2 − r1
CLOCK
IF (N) GOTO FETCH         // r1 > r2
IF (NOT N) r1 ¬ 0         // r1 £ r2
GOTO FETCH
```

d) Mikroprogramm für `ADD_LONG r2, r4, r6`:

```
C ¬ 0,
CLOCK
Areg ¬ 2,
Breg ¬ 4,
Freg ¬ 6,
SW ¬ 1001.0,
CLOCK                     // niederwertiges Wort addieren
Areg ¬ 3,
Breg ¬ 5,
Freg ¬ 7,
SW ¬ 1001.0
CLOCK                     // höherwertiges Wort addieren
GOTO FETCH
```

Aufgabe 3:

a) Da der LOAD−Befehl immer den Wert, der in den Akkumulator geladen wird, negiert und kein Additionsbefehl zur Verfügung steht wird zuerst die Differenz $-A-B$ gebildet. Diese wird gespeichert, erneut geladen und wieder gespeichert. Man erhält also $-(-A-B)=A+B$. Das dazugehörige Assemblerprogramm lautet:

```
LOAD  0 ; ACC = −A
SUB   1 ; ACC = −A−B = −(A+B)
STORE 0 ; M(0) = −(A+B)
LOAD  0 ; ACC = −(−(A+B)) = A+B
STORE 0 ; M(0) = A+B
```

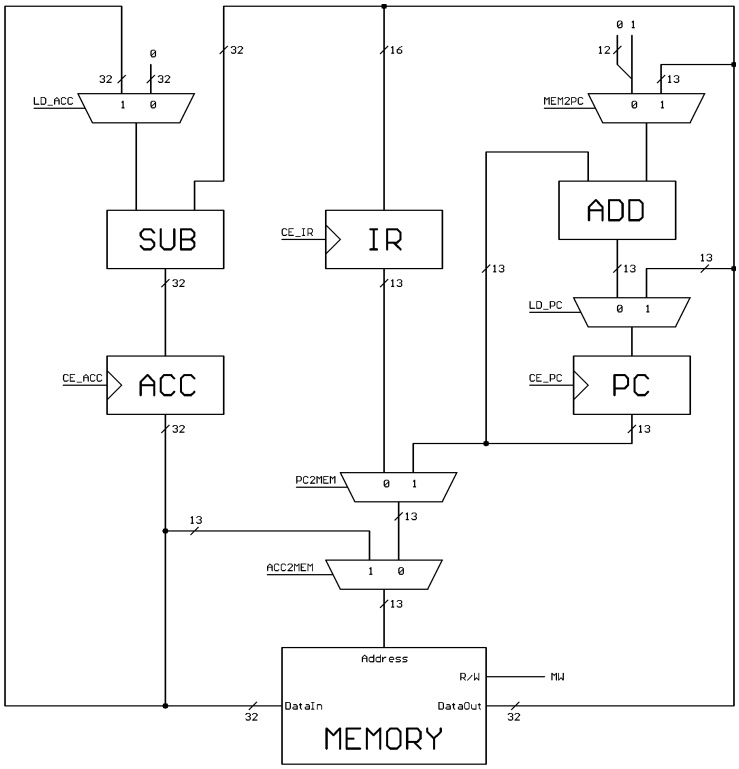
b) Auch hier ergeben sich, wie in Teil a), die Probleme mit der Negation. Sie werden über LOAD, STORE, LOAD Folgen kompensiert. Linksschieben um eine Stelle entspricht einer Multiplikation mit 2 und dies entspricht einer Addition des Wertes zu sich selbst.
Assemblerprogramm:

```
START: LOAD  1 ; ACC = −k
       TEST    ; −k < 0 ? (entspricht Test auf k > 0)
       JMP END

SHIFT: STORE 1 ; M(1) = −k
       LOAD  1 ; ACC = k
       SUB   2 ; ACC = k−1
       STORE 1 ; M(1) = k−1

       LOAD  0 ; ACC = −A
       SUB   0 ; ACC = −A−A = −2A
       STORE 0 ; M(0) = −2A
       LOAD  0 ; ACC = −(−2A) = 2A
       STORE 0 ; M(0) = 2A

       JMP START

ENDE:  STOP    ; Ergebnis in M(0)
```

c) Eine Möglichkeit besteht im Einsatz selbst−modifizierender Programme, in denen die entsprechenden LOAD/STORE−Befehle vor dem Zugriff angepasst werden.

Auf Grund der bekannten Probleme mit selbstmodifizierenden Programmen ist diese Lösung nur als Notlösung zu verstehen. Besser ist es eine Hardwaremodifikation des Steuerwerks vorzunehmen. Die LOAD/STORE−Befehle adressieren den Speicher nur mittels der Konstanten co. Der Zugriff auf den Speicher kann daher nur auf Adressen erfolgen, die bereits bei der Erstellung des Programms bekannt sind. Diese Restriktion kann man beheben, indem man z. B. den Inhalt des Akkumulators als Speicher−Adresse benutzt. Hierzu ist nur ein zusätzlicher Multiplexer vor dem Adress−Eingang des Speicher sowie ein Datenpfad vom Akkumulator zum Eingang des neuen Multiplexers erforderlich.

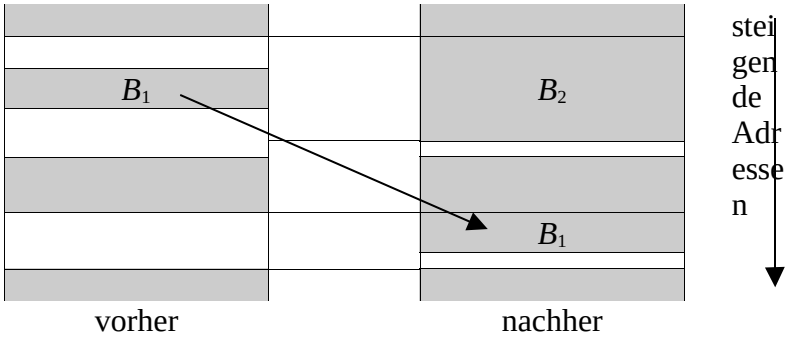
Die modifizierten Datenpfade sind in der folgenden Abbildung dargestellt.



Aufgabe 4:

a) Diese Aufgabe ist identisch mit der Einsendeaufgabe E.5−2. Lösungen mit Bäumen sind zwar speichereffizient, haben aber Zugriffszeiten von $O(\log n)$. Auch in Tabellen kann über eine binäre Suche nur diese Zugriffszeit erreicht werden. Erst bei Tabellen fester Größe ist es möglich als Zugriffszeit $O(1)$ zu erreichen. Wie in der Musterlösung schon gesagt, dient dann ein mehr oder weniger langer Teil der Größe als Index auf diese Tabelle, um einen passenden Speicherbereich zu finden. Im Prinzip ist dies ein Hashingverfahren, wobei als Hashingfunktion einfach die oberen Bits der Größe des gewünschten Speichers dienen.

b) Im folgenden Bild ist zu sehen, wie der Block B_1 mit Hilfe von best−fit umgelagert wird, damit der Block B_2 per first−fit eingelagert werden kann, dabei sind oben kleinere Speicheradressen zu finden..



Aufgabe 5:

a) Die Datenabhängigkeit liegt in der Verwendung von `r3`. Erst nachdem der `ADD`−Befehl den neuen Wert von `r3` geschrieben hat, kann er im `SUB`−Befehl verwendet werden. Ein Operator lädt seine Operanden im zweiten Teilschritt (**ID**) und das Ergebnis des Operators wird im fünften Teilschritt (**WB**) zurückgeschrieben. Es müssen also so viele NOPs eingefügt werden, dass das **ID** vom `SUB` nach dem **WB** vom `ADD` geschieht. Berücksichtigt man dies, ergibt sich das folgende Zeitdiagramm, in dem in der letzten Zeile auch die in der Aufgabenstellung angegebenen Ereignisse markiert sind:

`ADD r3, r1, r2`	IF	ID	EX	MEM	WB				
`NOP`		NOP	NOP	X	X	X			
`NOP`			NOP	NOP	X	X	X		
`NOP`				NOP	NOP	X	X	X	
`SUB r4, r1, r3`					IF	ID	EX	MEM	WB
				A)	B)	C)	D)		

b) Die evtl. Modifikation des PC durch den `BRZ`−Befehl wird auch hier erst im letzten Teilschritt (**WB**) gültig. Mit dem Laden des nächsten Befehls muss man warten, bis der PC gültig ist und somit muss man also so viele NOPs einfügen, bis die Pipeline leergelaufen ist. Der **IF** des nächsten Befehls kann natürlich erst geschehen, wenn feststeht, an welcher Stelle im Speicher der nächste Befehl steht.

`BRZ 20`	IF	ID	EX	MEM	WB					
`NOP`		NOP	NOP	X	X	X				
`NOP`			NOP	NOP	X	X	X			
`NOP`				NOP	NOP	X	X	X		
`NOP`					NOP	NOP	X	X	X	
`ADD r1, r2, r3`						IF	ID	EX	MEM	WB
						A)		B)		

c) Betrachtet man das Programmfragment der Aufgabenstellung, dann muss ein nicht optimierender Compiler die folgende Codesequenz erzeugen, um alle RAW Datenabhängigkeiten zu beseitigen.

```
JR  20
SUB r6, r2, r1
AND r5, r2, r1
ADD r3, r1, r2        ; RAW Abhängigkeit mit dem nächsten Befehl
NOP
NOP
NOP
SUB r4, r1, r3        ; RAW Abhängigkeit mit dem nächsten Befehl
NOP
NOP
NOP
ADD r1, r4, r5
XOR r6, r7, r8
```

Es kommen also zwei Datenabhängigkeiten vor. Die erste ist schon in Teil a) analysiert worden, die zweite ist neu. Durch Umsortieren der Befehle kann man möglicherweise Datenabhängigkeiten beseitigen. Natürlich darf dabei der Sinn des Programms nicht verändert werden. Sinnvollerweise startet man mit einem Befehl, bei dem eine Datenabhängigkeit vorliegt und fügt dahinter Befehle ein, die unabhängig sind und keine neue Datenabhängigkeit erzeugen. Auf diese Art und Weise kann man eine Datenabhängigkeit beseitigen und erhält:

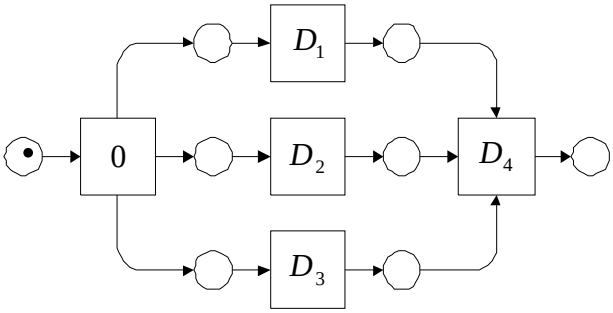
```
JR  20
ADD r3, r1, r2
SUB r6, r2, r1        ; Ersatz für NOP
AND r5, r2, r1        ; Ersatz für NOP
XOR r6, r7, r8        ; Ersatz für NOP
SUB r4, r1, r3        ; RAW Abhängigkeit mit dem nächsten Befehl
NOP
NOP
NOP
ADD r1, r4, r5
```

Durch diese Reihenfolge entsteht nun zwar eine WAW Datenabhängigkeit zwischen dem ersten `SUB`−Befehl und dem `XOR`−Befehl, was aber keine Konsequenzen hat.

Aufgabe 6:

a) Da eine feuernde Transition eine Marke verbraucht, muss die einlaufende Marke vervielfältigt werden, damit auch alle drei CPUs angestossen werden können. Wird die Stelle, an der sich die einlaufende Marke befindet, direkt an die Transitionen, welche die CPUs darstellen, angeschlossen, dann kann nur eine Transition schalten.

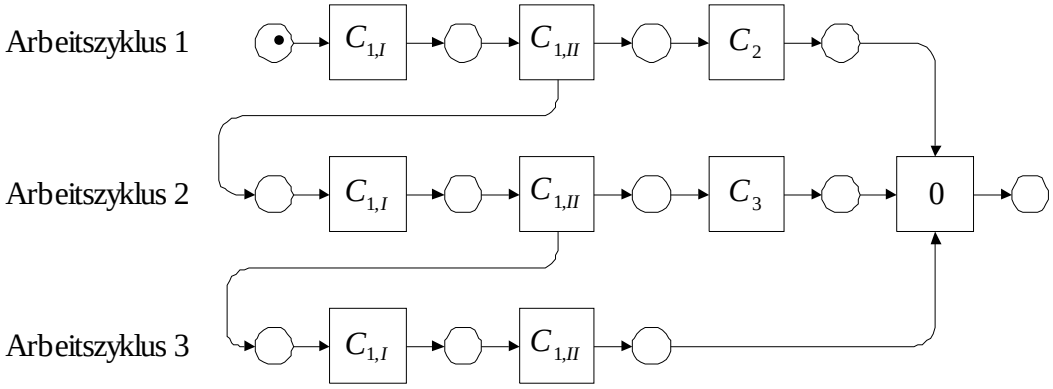
Petrinetz:



D_1, D_2 und D_3 sind die Zeiten in denen die drei CPUs parallel arbeiten. D_4 ist die Zeit, die CPU 1 benötigt, um die Ergebnisse zu verarbeiten.

b) Diese Aufgabe ist sehr ähnlich zu der Aufgabe E.7−4 aus den Einsendeaufgaben, allerdings etwas einfacher. Laut Aufgabenstellung braucht der Ablauf von CPU 2 und 3 nicht mehr aufgespalten werden. Das ein Zyklus von CPU 1 aus zwei Phasen besteht ist im Prinzip nicht wichtig, da CPU 2 nach dem ersten **Zyklus** (nicht Phase) aktiviert werden soll und CPU 3 nach dem zweiten **Zyklus**.

Petrinetz:



$C_{1,I}$ und $C_{1,II}$ sind die zweiten, die CPU 1 für Phase I bzw. Phase II benötigt. C_2 und C_3 sind die Zeiten, die CPU 2 bzw. CPU 3 arbeiten. Das Zusammenfassen wird durch die mit 0 markierte Transition symbolisiert. Wenn das Zusammenfassen einen signifikanten Zeitraum beanspruchen sollte, dann kann man diese Zeit natürlich auch dieser Transition zuordnen.

Kurze Anmerkung zu Petrinetzen:

- Bei Petrinetzen ist immer darauf zu achten, dass nur Stellen und Transitionen über gerichte Pfeile verbunden sind. Es dürfen **niemals** Stellen direkt mit Stellen bzw. Transitionen direkt mit Transitionen verbunden sein.
- Eine Stelle kann mit **mehr** als einer Marke besetzt sein und die Zahl der Marken pro Stelle ist nicht begrenzt.
- Eine Transition feuert **nur**, wenn alle Vorgängerstellen mit **mindestens** einer Marke besetzt sind.
- Wenn eine Transition feuert, dann verschwindet aus allen Vorgängerstellen **genau** eine Marke.