Einführung in die Rechnerarchitektur (1704)
Musterlösung zur Klausur SS98

Aufgabe 1: Multiplizierer (30 Punkte)

1. Der Schaltplan des Multiplizierers M(n,1) ist in Abbildung 1 dargestellt.
2. Der Beweis erfolgt durch vollständige Induktion über i:

$$
\begin{aligned}
i = 0: \quad D_0 &= \langle a \rangle \cdot \sum_{j=n-0}^{n-1} b_j \cdot 2^{j-(n-0)} \\
&= 0 \\
i-1 \to i: \quad D_i &= 2 \cdot D_{i-1} + \langle a \rangle \cdot b_{n-i} \\
&= 2 \cdot \langle a \rangle \cdot \sum_{j=n-(i-1)}^{n-1} b_j \cdot 2^{j-(n-(i-1))} + \langle a \rangle \cdot b_{n-i} \\
&= \langle a \rangle \left(\sum_{j=n-(i-1)}^{n-1} b_j \cdot 2^{j-(n-(i-1))+1} + b_{n-i} \right) \\
&= \langle a \rangle \left(\sum_{j=(n-i)+1}^{n-1} b_j \cdot 2^{j-(n-i)} + b_{n-i} \cdot 2^0 \right) \\
&= \langle a \rangle \cdot \sum_{j=n-i}^{n-1} b_j \cdot 2^{j-(n-i)}
\end{aligned}
$$

3. Der Schaltplan des Multiplizierers M(n,n) ist in Abbildung 2 dargestellt.

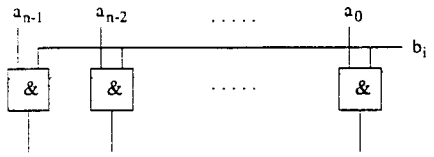

Abbildung 1: Aufbau des Multiplizierers M(1,n)

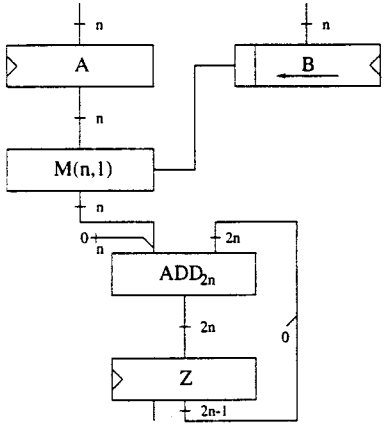

Abbildung 2: Aufbau des Multiplizierers M(n,n)

Aufgabe 2: Steuerwerke (30 Punkte)

1. Der Zustandsautomat der MARK I ist in Abbildung 3 dargestellt. Die Übergangsfunktion sieht wie folgt aus:

m_0: $\overline{op_0} \wedge \overline{op_1} \wedge \overline{op_2}$.

m_1: $\overline{op_0} \wedge \overline{op_1} \wedge op_2$.

m_2: $\overline{op_0} \wedge op_1 \wedge \overline{op_2}$.

m_3: $\overline{op_0} \wedge op_1 \wedge op_2$.

m_4: $op_0 \wedge \overline{op_1} \wedge \overline{op_2}$.

m_5: $op_0 \wedge \overline{op_1} \wedge op_2$.

m_6: $op_0 \wedge op_1 \wedge (ACC < 0)$.

m_7: $op_0 \wedge op_1 \wedge (ACC \geq 0)$.

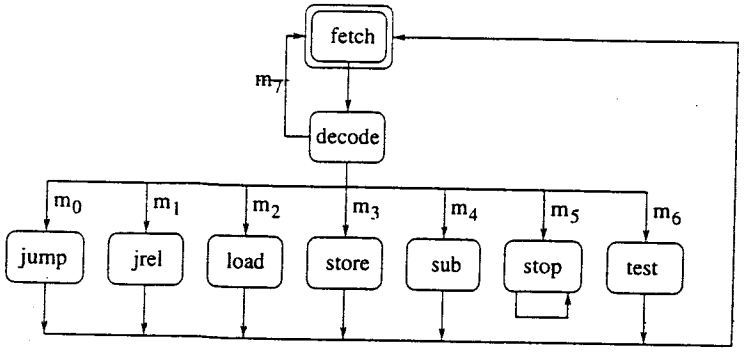

Abbildung 3: Zustandsautomat der MARK I

2. In der folgenden Tabelle sind die jeweils aktiven Kontrollsignale für alle Zustände angegeben. Dabei ist MW das Schreibsignal für den Speicher, das in den Datenpfaden der MARK I leider nicht eingezeichnet war. Der Zustand *store* wurde daher von der Bewertung ausgenommen.

   fetch: CE_IR, PC2MEM, CE_PC

   decode: (none)

   jump: LD_PC, CE_PC

   jrel: CE_PC, MEM2PC

   load: CE_ACC

   store: (MW)

   sub: CE_ACC, LD_ACC

   stop: (none)

   test: CE_PC

3. Zu Beginn befindet sich das Steuerwerk im Zustand *fetch*. Das aktive Kontrollsignal PC2MEM sorgt dafür, daß der Programmzähler benutzt wird, um den Speicher zu adressieren. Der zu ladende Befehl erscheint am Ausgang DataOut des Speichers und wird im Instruktionsregister IR gespeichert, da CE_IR aktiv ist. Gleichzeitig dürfen die Kontrollsignale MEM2PC und LD_PC nicht aktiv sein, um ein Laden des Instruktionswortes in den PC zu verhindern. Das aktive Kontrollsignal CE_PC sorgt dafür, daß der inkrementierte Programmzähler (am Addierer ADD liegen PC und 1 an) in den PC übernommen wird.

   Das Steuerwerk geht nun in den Zustand *decode* über, in dem keine Kontrollsignale aktiv sind. Abhängig vom Opcode des geladenen Befehls geht das Steuerwerk in den nächsten Zustand über. Wir nehmen an, daß es sich um einen LOAD-Befehl handelt, daher befindet sich das Steuerwerk nun im Zustand *load*. Da PC2MEM nicht aktiv ist, wird das Instruktionsregister benutzt, um den Speicher zu adressieren. Das gewünschte Datenwort erscheint am Ausgang DataOut, durchläuft den Subtrahierer, wobei es negiert wird, da am anderen Eingang des Subtrahierers der Wert 0 anliegt (LD_ACC inaktiv). Das Ergebnis wird im Akkumulator gespeichert,da das Kontrollsignal CE_ACC aktiv ist.

4. Da das Laden des Akkumulators auf der Mark I mit einer Negation verbunden ist und nur die Subtraktion zur Verfügung steht, wird die Addition mit einem kleinen Trick realisiert:

$$a + b = (-(-a) - (-b))$$

   Aus dieser Tatsache ergibt sich das folgende Programm:

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

5. Auch bei diesem Programm treten die bereits beschriebenen Probleme auf. In dem unten aufgeführten Assemblerprogramm wird das Laden eines Operanden durch eine LOAD, STORE, LOAD Folge realisiert, um die automatische Negation beim Laden zu umgehen. Das Schieben des Operanden um 1 Bit wird durch die Multiplikation mit 2 bzw. der Addition des Operanden mit sich selbst realisiert.

```
LOAD  0 ; ACC=-A
```

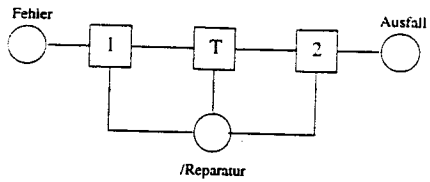

Abbildung 4: Petri-Netz (Aufgabe 3)

```
        STORE 3 ; M(3)=-A
LOOP:   LOAD  1 ; ACC=-k
        STORE 1 ; M(1)=-k
        LOAD  1 ; ACC=k
        SUB   2 ; ACC=K-1
        TEST    ; k-1<0 ?
        JUMP  SHIFT
        JUMP  END
SHIFT:  LOAD  3 ; ACC=-Z
        STORE 3 ; M(3)=-Z
        LOAD  3 ; ACC=Z
        SUB   3 ; ACC=Z-(-Z)=2Z
        STORE 3 ; M(3)=2Z
END:    LOAD  3 ; ACC=-Z
        STORE 3 ; M(3)=-Z
        LOAD  3 ; ACC=Z
        STORE 0 ; M(0)=Z
```

Aufgabe 3: Petri-Netze (10 Punkte)

Ein Petri-Netz, welches die gestellten Anforderungen erfüllt, ist in Abbildung 4 dargestellt.

Aufgabe 4: Speicherverwaltung (10 Punkte)

Siehe Musterlösung zur Einsendeaufgabe E5-2 (Kurseinheit 5).

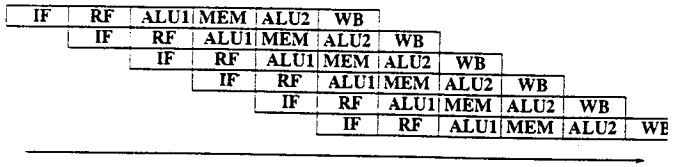

Abbildung 5: Struktur der Pipeline

Aufgabe 5: Pipelining (20 Punkte)

1. Wir identifizieren zunächst die Pipelinestufen, in denen ein Addierer bzw. Inkrementer benötigt wird. Es handelt sich dabei um die Stufen IF (Inkrementieren des PCs), ALU1 (Berechnung von Adressen) und ALU2 (Ausführung von Operationen). Aufgrund der Annahme, daß in jdem Zyklus ein neuer Befehl gestartet wird, ergibt sich die in Abbildung 5 dargestellte Situation. Da alle erwähnten Pipelinestufen in ein und demselben Zyklus vorkommen können, sind insgesamt 3 Addierer bzw. Inkrementer notwendig.

   Eine Belegung der Pipeline, bei der alle 3 Addierer gleichzeitig benutzt werden, ergibt sich z.B. durch die folgende Befehlsfolge:

```
OP RD1,RS1,RS2      ; beliebige Operation (IF)
OP RD1,RS1,RS2      ; beliebige Operation (RF)
OP RD,RS1,#c(RS2)   ; beliebige Operation im (ALU1)
                    ; Register-Speicher-Format
OP RD1,RS1, RS2     ; beliebige Operation (M)
OP RD1,RS2, RS2     ; Compute-Operation (ALU2)
```

2. Wir identifizieren zunächst die Pipelinestufen, in denen ein Zugriff auf den Registersatz erfolgt. Es handelt sich dabei um die Stufen RF (Laden der Operanden (2x lesend)) und WB (Schreiben des Ergebnisses (1x schreibend). Der Registersatz muss daher über 2 Lese- und einen Schreib-Port verfügen, da das Laden der Operanden sowie das Schreiben des Ergebnisses bei allen Instruktionen erforderlich ist.