Aufgabe 5

PUSH(α):

Die Routine PUSH(α) kann nur dann ein Element auf den Stack legen, wenn die maximale Kapazität noch nicht ausgeschöpft ist. Die möglichen Statusmeldungen in ρ(0) sind demzufolge ρ(0)=0, wenn das Element abgelegt wurde und ρ(0)=1, wenn der Stack zu voll ist, um noch ein weiteres Element abzulegen.
Gerade letztere Meldung ist wichtig, da sie erst erscheint, wenn bei bereits vollem Stack ein PUSH(α) versucht wird. Füllt dagegen ein PUSH(α) den Stack, so muss ein ρ(0)=0 erscheinen, da dies ja eine erfolgreiche Operation war, obwohl jetzt natürlich der Stack voll ist. Ich halte es daher für sinnvoll, wenn man in einer "richtigen" Stack-Bibliothek noch die Routine IsEmpty und IsFilled implementiert. Zusätzlich ist darauf zu achten, dass der Stack initialisiert wird, insbesondere ρ(1)=1 (z.B. durch do ρ(1):=1).
Laut Aufgabenstellung sind 20 Speicherzellen für den Stack reserviert, davon werden jedoch 2 für Verwaltungszwecke benötigt. Die erste Speicherzelle, die für Stackelemente benutzt werden kann, ist ρ(2), die letzte ist ρ(19). Ich konstruiere meine Stackroutinen derart, dass ρ(1) die Position der letzten belegten Speicherzelle im Stack enthält, also Werte zwischen 1 (leer) und 19 (voll) annehmen kann. α ist nach der Routine unverändert.

```
01: do ρ(0):=α;          # rette den Akkumulator
02: do α:=ρ(1);          # bisherige Stackgrösse auslesen
03: if α=19 goto 10;     # ist Stack schon voll ?
04: do α:=α+1;           # noch Platz, neue Grösse berechnen
05: do ρ(1):=α;          # neue Grösse abspeichern
06: do α:=ρ(0);          # den ursprünglichen Akkumulator
                           wiederherstellen
07: do ρ(ρ(1)):=α;       # und im Stack ablegen
08: do ρ(0):=0;          # Operation war erfolgreich
09: goto 11;             # fertig !
10: do ρ(0):=1;          # Fehler anzeigen: Stack ist voll
11: ...                  # hier folgt dann das Programm, in dem der
                           Stack verwendet wird
```

POP(α):

Diese Routine ähnelt sehr stark PUSH(α). Wesentliche Unterschiede liegen darin, dass der Akkumulator α stets geändert wird (auch wenn der Stack leer ist, also POP(α) scheitert !) und dass als Fehlerbedingung nun ρ(1)=1 gilt.

```
01: α:=ρ(1);             # Stackgrösse auslesen
02: if α=1 goto 11;      # ist der Stack leer ?
03: α:=ρ(ρ(1));          # Element auslesen
04: ρ(0):=α;             # temporär sichern, um mit α die neue
                           Stackgrösse zu berechnen
05: α:=ρ(1);             # Stackgrösse auslesen
06: α:=α-1;              # und neu berechnen
07: ρ(1):=α;             # sowie abspeichern
08: α:=ρ(0);             # das ausgelesens Stackelement holen
09: ρ(0):=0;             # Operation war erfolgreich
10: goto 12;             # fertig !
11: ρ(0):=2;             # Fehler anzeigen: Stack ist leer
12: ...                  # hier folgt dann das Programm, in dem der
                           Stack verwendet wird
```

Ich habe keinerlei weiteren Zwischenspeicher benötigt, da die jeweiligen Routinen den Status selbst erkennen können und er im Verlaufe nicht benötigt wird, erst am Ende der Routinen (Zeilen 8 bis 10 bzw. 9 bis 11) wird er neu gesetzt. In den davor liegenden Zeilen dient die Statuszelle als temporärer Speicher für den Akkumulator.

Aufgabe 6

a) Die Anzahl der Zustände des Schaltwerkes berechnet sich wie folgt:
$\#Z_s = \#\alpha * \#\beta * \#\gamma * \#\gamma_1 * \#\eta * \#\omega_1 * \#\omega_2$
$\#Z_s = (2Q+1) * (\#Op*K) * (M+1) * (M+1) * (N+1) * 2 * 2$
mit $K = \max(N,M)$
Die Einführung eines zweiten Indexregisters erhöht die Anzahl der Zustände des Schaltwerkes um den Faktor M+1.

b) `do` $\gamma:=\gamma_1$`:`
$k' = (\pi'=\pi, \rho'=\rho, \alpha'=\alpha, \beta'=\pi(\eta'), \eta'=\eta+1, \gamma'=\gamma, \gamma_1'=\gamma, \omega_1'=\omega_1=1, \omega_2'=\omega_2=0)$
mit $\eta<N$
`if` $\gamma==\gamma_1$ `goto j:`
Bedingung $\gamma==\gamma_1$ erfüllt:
$k_1' = (\pi'=\pi, \rho'=\rho, \alpha'=\alpha, \beta'=\pi(\eta'), \eta'=j, \gamma'=\gamma, \gamma_1'=\gamma_1, \omega_1'=\omega_1=1, \omega_2'=\omega_2=0)$
mit $j<=N$
Bedingung $\gamma==\gamma_1$ nicht erfüllt:
$k_2' = (\pi'=\pi, \rho'=\rho, \alpha'=\alpha, \beta'=\pi(\eta'), \eta'=\eta+1, \gamma'=\gamma, \gamma_1'=\gamma_1, \omega_1'=\omega_1=1, \omega_2'=\omega_2=0)$
mit $\eta<N$

Aufgabe 7

Ich benötigte 11 Zeilen, um den Horner-Algorithmus auf einem Rechner mit 2 Indexregistern zu implementieren. Die Anzahl der Rechenschritte beträgt dabei 4n+9. Die verwendete Speicheranordnung der Koeffizienten ist kompatibel zu den in der Vorlesung vorgestellten Programmen Horner 1 bis 3.
Die Grundidee funktioniert wie folgt:
Das zweite Indexregister γ_1 zeigt auf die Speicherzelle, die den letzten Koeffizienten enthält.
In den Kernzeilen 6 bis 9 wird nun verglichen, ob γ bereits alle Koeffizienten durchlaufen hat. Ist dies nicht der Fall, so ist das Polynom noch nicht vollständig berechnet. Demzufolge muss auf den Akkumulator α, der stets das Zwischenergebnis speichert und nicht für andere Zwecke benötigt wird, die Operation $\alpha:=\alpha*x_0+a_{k+1}$ angewendet werden. Als Seiteneffekt wandert der Zeiger γ um eine Speicherzelle weiter.
Am Schluss muss nur noch die "Ergebnis"-Speicherzelle $\rho(2)$ mit α beschrieben werden.

```
00: begin: goto 1;
01: α:=3;              # α:=3
02: γ1:=α;
03: γ:=γ1;             # γ=γ1=α=3;
04: α:=ρ(γ1+ρ(0));     # Dummyoperation, nur der Seiteneffekt auf γ1
                         interessiert: γ1:=n+3
05: α:=ρ(γ);           # α:=a0
06: if γ==γ1 goto 10;  # Polynom berechnet ?
07: α:=α*ρ(1);         # α:=fk*x0
08: α:=α+ρ(γ+1);       # α:=fk*x0+ak+1, Seiteneffekt γ:=γ+1
09: goto 6;
10: ρ(2):=α;
11: end.
```

Aufgabe 8

a)

Bestandteil	Einfache Rechenmaschine	v.-Neumann-Rechner
Akkumulator	α	A
Befehlsregister (instruction register)	β, aber nur Operation	IR, aber nur Operation
memory address register	γ	MAR
Befehlszähler (program counter)	η	PC

b)

Register	Anzahl Bits
Akkumulator A	ld(2Q+1)
Befehlsregister IR	ld(#Op)
memory address register MAR	ld(N+1)
Befehlszähler PC	ld(N+1)
memory buffer register	ld(N+1)

Da für Daten- und Programmspeicher ein Unified-Memory-Konzept verwendet werden soll, ist N=M. Die mathematische Funktion ld(x) steht für den dualen Logarithmus und die sich ergebende Anzahl Bits ist bei einem gebrochenzahligen Ergebnis von ld(x) als die nächstgrößere natürliche Zahl zu verstehen.