Allgemeine Hinweise zur Klausur

Prüfungsdauer: 120 Minuten Aufgaben: 5 Punktezahl: 100 von 118

Erlaubte Hilfsmittel: beliebiges gedrucktes oder handgeschriebenes Material

Aufgabe 1 - Datenspeicher (20 Punkte)

Ein Signalprozessor soll mit einem Speicherbereich ausgestattet werden, der als Dual-Ported RAM ausgeführt ist. Dieser Speicher soll den Zugriff auf 2 Ports zulassen, die unabhängig von einander zum Lesen bzw. zum Schreiben durch den Prozessor und einen Koprozessor verwendet werden können. Der Speicher soll eine Größe von 4k Worten mit einer Wortbreite von jeweils 16 Bit zur Verfügung stehen.

Frage 1.1 (4 Punkte): Skizzieren Sie ein Blockschaltbild des Speichers mit seinen Eingängen und Ausgängen. Hinweis: Adressleitungen, Datenleitungen und Steuersignale.

Lösung:



Frage 1.2 (4 Punkte): Konzeption und Umsetzung. Beschreiben Sie den wesentlichen Teil der Reali-sierung des Speichers und dessen Umsetzung in HDL.

Lösung: Wesentlicher Teil des Speichers ist ein Array der Größe 4096 (4k) aus Datenworten. Für die Datenworte wird der Typ std\_logic\_vector von 16 Bits verwendet. Als Datentyp für die Adressleitungen der Typ std\_logic\_vector von 12 Bits (zur Adressierung von 4k). Der Speichert ist getaktet, d.h. nur bei steigender Taktflanke werden an den beiden Ports gelesen bzw. geschrieben. Zum Schreiben verfügt jeder Port über ein eigenes Enable-Signal. Details siehe Programmtext in der Lösung der folgenden Aufgabe.

Frage 1.3 (6 Punkte): Skizzieren Sie eine mögliche Realisierung des Speichers in HDL. Erläutern Sie Ihre gewählte Realisierung (z.B. gegenüber anderen Optionen).

Lösung (Beispiel): Getrennte Prozesse für beide Ports (Alternative: einzelner Prozess, ebenfalls machbar, erscheint mir jedoch weniger übersichtlich im HDL-Code).

library ieee;

use ieee.std\_logic\_1164.all;

use IEEE.numeric\_std.all;

entity REGISTER\_BLOCK is

port (Clk : in std\_logic;

DI\_1, DI\_2 : in std\_logic\_vector(15 downto 0);

DO\_1, DO\_2 : out std\_logic\_vector(15 downto 0);

A1, A2 : in std\_logic\_vector(12 downto 0);

WR1\_EN, WR2\_EN: in std\_logic;

end REGISTER\_BLOCK;

architecture RTL of REGISTERBLOCK is

type BLOCK is array(0 to 4096) of std\_logic\_vector(15 downto 0);

signal REG : BLOCK := (others => x"0000");

begin

Port\_1: process (Clk) begin

if rising\_edge(Clk) then

if (WR1\_EN = '1') then

REG(to\_integer(unsigned(A1)) <= DI\_1;

end if;

DO\_1 <= REG(to\_integer(unsigned(A1));

end if;

end process Port\_1;

Port\_2: process (Clk) begin

if rising\_edge(Clk) then

if (WR2\_EN = '1') then

REG(to\_integer(unsigned(A2)) <= DI\_2;

end if;

DO\_2 <= REG(to\_integer(unsigned(A2));

end if;

end process Port\_2;

end RTL;

Frage 1.4 (6 Punkte): Was geschieht, wenn beide Ports von der gleichen Adresse lesen möchten? Was geschieht, wenn ein Port eine Adresse beschreiben möchte, die auf dem anderen Port gerade ausgelesen wird? Was geschieht, wenn beide Ports auf die gleiche Adresse schreiben möchten? Begründen Sie Ihre Aussage. Mit welchen Massnahmen lassen sich Konflikte gege-benenfalls verhindern?

Lösung: Abhängig von der Implementierung des Prozesses (Clock). Im o.g. Beispiel: Lesen von der gleichen Adresse: kein Problem, es wird auf beiden Ports das gleiche Datum ausgegeben. Lesen und Schreiben an der gleichen Adresse: Gelesen wird der alte Wert, geschrieben wird der neue Wert (siehe Funktion Akku als Register in der Aufgabe oben). Doppeltes Schreiben an die gleiche Adresse: Führt zu Konflikten (auch wenn sequentiell geschrieben werden sollte) und sollte durch die Logik der WR\_EN-Signale verhindert werden.

Aufgabe 2 - Signalverarbeitung (20 Punkte)

Zur Signalverarbeitung wird folgender Ausschnitt aus einem Rechenwerk eingesetzt.

A : process (S, H) begin

for i in 0 to 31 loop

P(i) <= S \* H(i);

end loop;

end process A;

B : process (Clk) begin

if rising\_edge(Clk) then

S <= signed(IN(15 downto 0));

end if;

end process B;

process (Clk) begin

if rising\_edge(Clk) then

for i in 0 to 31 loop

Q(i) <= Q(i+1) + P(i);

end loop;

end if;

end process;

OUT <= Q(0)(31 downto 15);

Frage 2.1 (4 Punkte): Die zugehörige Entity hat folgendes Blockschaltbild. Hierbei sind IN und OUT vom Datentyp std\_logic\_vector. Welche internen Signale verwendet das Rechenwerk? Wie lassen sich die Datentypen ineinander konvertieren? Was muss hierbei beachtet werden?



Lösung: (1) Signale vom Typ signed. H(i) und P(i) und Q(i) sind Arrays vom Typ signed mit insgesamt 32 Feldern. (2) Umformung durch Typecasts (z.B. unterdes Format S <= Signed(IN) bzw. Zuweisung Out <= Q(0) obere Bits. (3) Die Wortbreiten der internen Signale P(i) und Q(i) müssen wegen Multiplikation und Additionen größer sein als die der Eingangswerte.



Frage 2.2 (6 Punkte): Rekonstruieren Sie das Blockschaltbild des Rechenwerks mit den beteiligten Prozessen als Skizze. Hinweis: Verwenden Sie nur die Prozesse zusammen mit den zugehörigen Signalen. Zum detaillierten Innenleben der Prozesse folgt eine weitere Frage.

Lösung:



Frage 2.3 (4 Punkte): Das Signal H ist ein Array von 16-Bit Worten vom Datentyp signed, S ist ein 16 Bit-Wert vom Datentyp signed. Die Signale P und Q sind ein Arrays von 32-Bit Worten vom Datentyp signed. Skizzieren Sie das Innenleben der Prozesse (den internen Signalfluss) und rekonstruieren Sie die Struktur des Rechenwerks. Welcher Algorithmus wird hiermit ausgeführt?

Lösung: Faltung in transponierter Form, siehe unten bzw. Manuskript



Frage 2.4 (6 Punkte): Tests. Skizzieren Sie die Struktur von Testfällen in einem Testprogramm für den Algorithmus. Hinweis: Konzept und Aufbau genügen, HDL ist nicht unbedingt erforderlich. In dem Algorithmus werden 16-Bit Festkommazahlen addiert und multipliziert. Hierbei sei angenommen, dass |Σ H(i) | < 1. Wie können Überläufe im Rechenwerk vermieden werden? Wie können Sie auf Überläufe testen?

Lösung: Struktur des Testprogramms: (1) Prozess für Reset, Prozess für Testfälle, hierbei wird am einfachsten mit jedem neuen generierten Takt Clk ein neuer Eingangswert erzeugt. Eine Sprungfunktion am Eingang (d.h. IN = 1) sollte eine Sprungantwort erzeugen, die auf den Wert 1 einschwingt. Ein Impuls am eingang (d.h. IN = 1 für die erste Stützstelle) sollte die Koeffizienten H(i) am Ausgang zeigen. (2) Überläufe: können mit der Bedingung |Σ H(i) | < 1 nicht vermieden werden, da ja z.B. für H(i)=1 für i=0, ..., 5 und H(i) = -1 für i=6, ..., 11 diese Bedingung ebenfalls erfüllt ist. Überläufe können nur beim Summieren auftreten, da Q(i) auf 32 Bit (d.h. doppelte Wortbreite der Summanden) festgelegt ist. Hier muss Reserve eingeplant werden (bei 32 Koeffizienten zusätzlich 4 Bits). Die Zahlen lassen sich am Ausgang dann wieder auf den maximalen Zahlenbereich von 16 Bit skalieren. (3) Test für Überläufe: max. Wertebereich für IN (d.h. IN=1 bei Festkomma direkt nach der Vorzeichenstelle). Koeffizienten so wählen, dass |Σ H(i) | < 1 im Rahmen realistischer Impulsantworten erfüllt wird.

Aufgabe 3 - Steuerwerk eines Prozessors (30 Punkte)

Folgender Programmtext zeigt einen Ausschnitt aus dem Zustandsautomaten eines Prozessors:

update\_state\_register: process(Clk, RS)

begin

if (RS='1') then

current\_state <= Z1;

elsif rising\_edge(Clk) then

current\_state <= next\_state;

end if;

end process update\_state\_register;

logic\_next\_state\_and\_actions: process(current\_state, T\_OPCODE)

begin

case current\_state is

when Z1 =>

case T\_OPCODE is

when NOP => ...;

when ADD | SUB | ANDA | EOR | ORA | LDA => ...;

next\_state<=Z2;

when LSL | LSR | LDI => ...;

when STR => ...;

when JMP => ....; next\_state<=Z2;

when PUSH => ...;

when POP => ...; next\_state<=Z2;

when CALL => ...; next\_state<=Z2;

when RET => ...; next\_state<=Z2;

when BRBC =>

case T\_PSR is

when "10000" => if (T\_Zero = '0') then

...; next\_state<=Z2;

elsif (T\_Zero ='1') then

...;

end if;

...;

end case;

...;

when others => null;

end case; -- T\_OPCODE

when Z2 =>

case T\_OPCODE is

when ADD | SUB | ANDA | EOR | ORA | LDA => ...;

next\_state<=Z1;

when BRBS | BRBC| JMP | CALL => ...;

next\_state<=Z1;

when RET => ...; next\_state<=Z1;

when POP =>...; next\_state<=Z1;

when others => null;

end case; -- T\_OPCODE

when others => null;

end case; -- states

end process logic\_next\_state\_and\_actions;

Frage 3.1 (6 Punkte): Zeichnen Sie ein Blockschaltbild des Automaten ausgehend von der gegebenen HDL Realisierung. Welche Signale liegen an den Eingängen? Welche Signale liegen voraussichtlich an den Ausgängen? Wie viele Leitungen werden für die Kodierung der Zustände benötigt? Wie viele Eingangsleitungen werden für einen RISC-Prozessor mit 32 Befehlen benötigt?

Lösung:



Fragen: Eingänge: OPCODE; Ausgänge: Steuerleitungen zum Rechenwerk und Steuerwerk; Kodierung der 3 Zustände Z1, Z2 und Z3: 4 Bits = 4 Leitungen; Eingänge RISC Prozessor mit 32 Befehlen (Opcode): 5 Leitungen

Frage 3.2 (4 Punkte): Zeichnen Sie das Zustandsdiagramm des Automaten. Hinweis: Verallgemeinern Sie die Bedingungen für die Zustandsübergänge, hierzu folgen noch weitere Fragen.

Lösung:



Frage 3.3 (4 Punkte): Welcher der im HDL-Text aufgeführten Prozesse ist taktabhängig? Welcher der Prozesse ist nicht taktabhängig? Wann genau erfolgen die im Zustandsdiagramm gezeigten Zustandsübergänge? Wann genau erfolgen die (im Programmauszug nicht näher genannten) Zustandsaktionen in Bezug auf das Zustandsdiagramm?

Losung: (1) update\_state\_register: process(Clk, RS) ist taktabhängig. (2) logic\_next\_state\_and \_ actions: process(current\_state, T\_OPCODE) ist ein reines Schaltnetz und nicht taktabhängig (siehe auch Blockdiagramm zu 1.1). (3) Die Zustandsübergänge erfolgen mit jedem Takt. (4) die Zustands-aktionen erfolgen unmittelbar nach dem Zustandswechsel (mit der Laufzeit des Schaltnetzes).

Frage 3.4 (6 Punkte): Im eingangs gezeigten Zustandsautomaten sind die Zustandsaktionen eine un-mittelbare Folge der Eingangssignale sowie des aktuellen Zustands. Folgendes Blockschaltbild zeigt eine alternative Realisierung.



Wie wäre der eingangs gezeigte HDL-Code für diesen Automaten zu ändern? Skizzieren Sie die Struktur des HDL-Codes (grobe Struktur genügt). Hätte diese Variante Nachteile bzgl. des Zeitverhaltens? Hätte diese Variante Vorteile? Halten Sie diese Variante für diesen Zweck überhaupt für praktikabel? Hinweis: Begründen Sie Ihre Aussagen.

Lösung:

update\_state\_register: process(Clk, RS) -- unverändert

...

end process update\_state\_register;

logic\_next\_state: process(current\_state, T\_OPCODE)

begin -- hier nur noch die Zustandsänderungen ohne Zustandsaktionen

case current\_state is

when Z1 => -- Code ist nur zur Verdeutlichung nochmals aufgeführt

-- Kommentar bzgl. Zustandsänderungen ohne - aktionen genügt

case T\_OPCODE is

when ADD | SUB | ANDA | EOR | ORA | LDA |

JMP | POP | CALL | RET | => next\_state<=Z2;

when BRBC =>

case T\_PSR is

when "10000" =>

if (T\_Zero = '0') then next\_state<=Z2; end if;

...;

end case;

...;

when others => null;

end case; -- T\_OPCODE

when Z2 =>

case T\_OPCODE is

when BRBS | BRBC| JMP | CALL | POP | ADD | SUB

| ANDA | EOR | LDA | => next\_state<=Z1;

when RET => next\_state<=Z2;

when others => null;

end case; -- T\_OPCODE

when others => null;

end case; -- states

end process logic\_next\_state;

logic\_state\_actions: process(current\_state)

begin -- hier nur noch die Zustandsaktionen

when Z1 => -- Code ist nur zur Verdeutlichung nochmals aufgeführt

-- Kommentar bzgl. Zustandsaktionen genügt

case T\_OPCODE is

-- pro OPCODE jeweils individuelle Zustandsaktionen

when ADD | SUB | ANDA | EOR | ORA | LDA |

JMP | POP | CALL | RET | => ... ;

when BRBC => -- Bedingungen müssen nochmals abgefragt werden

case T\_PSR is

when "10000" =>

if (T\_Zero = '0') then ...;

elsif (...) ...;

end if;

...;

end case;

...;

when others => null;

end case; -- T\_OPCODE

when Z2 =>

case T\_OPCODE is

-- pro OPCODE jeweils individuelle Zustandsaktionen

when BRBS | BRBC| JMP | CALL | POP | ADD | SUB

| ANDA | EOR | LDA | => ...;

when RET => ...;

when others => null;

end case; -- T\_OPCODE

when Z3 =>

case T\_OPCODE is

-- pro OPCODE jeweils individuelle Zustandsaktionen

when RET => ...;

when others => null;

end case; -- T\_OPCODE

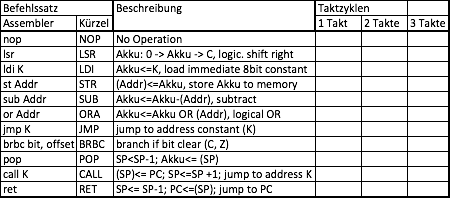
when others => null;

end case; -- states

end process logic\_next\_state;

Nachteile bzgl. des Zeitverhaltens: grundsätzlich keine, da bei mehrtaktigen Befehlen die Pipeline eh angehalten werden muss. Vorteile: keine. Die Übersichtlichkeit erhöht sich leider auch nicht, da alle Zustände und OPCODEs (=Eingänge) erneut abgefragt werden müssen. Aus diesem Grund ist diese Variante auch nicht praktikabel: Die Ausgänge (Steuerleitungen) sind keine reine Funktion der Zustände, sondern ebenfalls abhängig vom Eingangssignal.

Frage 3.5 (4 Punkte): Folgende Tabelle zeigt einen Ausschnitt der Befehle des Prozessors. Welche Befehle benötigen gemäß der eingangs gegebenen Realisierung (HDL) 1, 2 oder 3 Taktzyklen? Hinweis: Verwenden Sie für Ihre Antworten direkt das Aufgabenblatt.



Lösung: siehe HDL-Text eingangs, bzw. Manuskript.

Frage 3.6 (6 Punkte): Erläutern Sie, warum die in Frage 3.5 in der Tabelle aufgeführten Befehle einen, zwei oder drei Takte benötigen. Warum ist die Behandlung der bedingten Sprungbefehle im eingangs gezeigten HDL-Text im Vergleich zum unbedingten Springbefehl so aufwändig? Welche Probleme gibt es in dieser Realisierung mit dem Befehl RET?

Lösung: (1) Eintaktig: No Operation und Befehle auf vorhandenem Operanden im Akku: nop, lsl, lsr, ldi K, st Addr, push. (2) zweitaktig: arithmetisch logische Befehle und Operationen, bei denen ein Operand muss aus dem Arbeitsspeicher geholt werden: ld Addr, add Addr, sub Addr, and Addr, eor Addr, or Addr, pop; beim den Sprungbefehlen bzw. beim Sprung in ein Unterprogramm (jmp K, brbc bit, offset, brbs bit, offset call K) muss der Programmzähler neu geladen werden, was einen extra Arbeitsgang erfordert. (3) dreitaktig: Rückkehr aus einem Unterprogramm: ret. Wegen Kombination aus Operanden aus dem Speicher holen und hiermit den Programmzähler neu laden. Mit zwei Takten funktioniert die gegebene Realisierung nur, wenn zwei NOPs hinter dem Befehl RET programmiert werden. (4) HDL-Aufwand bei bedingtem Sprungbefehl: sowohl die Bitmaske für das Statusbit als auch der Zustand dieses Bits müssen abgefragt werden, um zu entscheiden, ob ein Sprung erfolgt.

Aufgabe 4 - Architektur des Prozessors (24 Punkte)

Folgende Abbildung zeigt den Aufbau eines Mikrocontrollers.



Frage 4.1 (6 Punkte): Erläutern Sie den Ablauf folgender Befehle: (1) NOP, (2) CALL K (Aufruf Unterprogramm), (3) st Addr (Wert an Adresse Adr speichern), (4) AND Addr (mit Wert aus Adresse Addr verknüpfen), (5) LDI K (Konstante laden), (6) BRBC bit, offset (bedingter Sprungbefehl).

Lösung: siehe Skript. Die Befehle beziehen sich jeweils auf den aktuellen Inhalt des Akkus. Die Konstante K bzw. die Sprungadresse K zum Laden des Akkus bzw. Befehlszählers PC werden dem Programmspeicher entnommen. Zum Laden des Programmspeichers und Holen des nächstes Befehls werden 2 Takte benötigt. Bei arithmetisch-logischen Befehlen mit 2 Operanden muss ein Wert von der Adresse Addr aus dem Datenspeicher entnommen werden. Diese Befehle benötigen zwei Takte. Der bedingte Sprung prüft ein mit Hilfe der Maske bit indiziertes Statusbit um zu entscheiden, ob fortgefahren bzw. zur Adresse PC+offset gesprungen wird.

Frage 4.2 (6 Punkte): Pipeline und Sprungbefehle. Folgende Abbildung zeigt die Kette vom Programmzähler (Befehlszähler) bis zum Befehlsregister (Dekoder).



Erläutern Sie den normalen Programmablauf: Wann zählt der Programmzähler weiter? Wie kommt es, dass auf Zeitdiagrammen der Programmzähler dem aktuell dekodierten und vom Steuerwerk verarbeiteten Befehl immer 2 Takte vorauseilt? Sprungbefehle: Was genau geschieht bei einem bedingten bzw. unbedingten Sprungbefehl? Was muss wegen der dargestellten Kaskade (Pipeline) bei Sprungbefehlen beachtet werden?

Lösung: siehe Skript. (1) Normaler Ablauf: Der Befehlszähler zählt nach Ausführung eines Befehls um eins weiter. Bei Befehlen, die zur Ausführung mehr als einen Takt benötigen, wird die Kaskade (Pipeline) vom Steuerwerk angehalten. Sprungbefehle: (2) Es benötigt eine Takt, um einen neuen Wert in den Programmzähler zu laden. Erst im folgenden Takt zeigt der Programmzähler an die korrekte Adresse. Sprungbefehle benötigen also 2 Takte, sofern ein Sprung ausgeführt wird. (3) Bei Ausführung eines Sprungbefehls enthält die Kette (Pipeline) einen ungültigen Befehl, nämlich den Befehl, der im linearen Ablauf auf den Sprungbefehl folgt. Der Assembler (Übersetzungsprogramm von Assemblersprache nach Maschinensprache) sollte also hinter einem Sprungbefehl im Maschinen-programm immer einen NOP (Leerbefehl) einfügen.

Frage 4.3 (6 Punkte): Stapel. Mit dem Befehl PUSH kann der Inhalt des Prozessors im Stapel konserviert werden. Mit dem Befehl POP holt man diesen Inhalt wieder in den Prozessor. Erläutern Sie den Zweck dieser Befehle und die diesbzgl. Abläufe bzgl. Programmzähler (PC) und Stapelzeiger (SP). Beachten Sie hierbei die in der Abbildung zu Frage 2.2 gezeigte Kette (Pipeline).

Lösung: siehe Skript.

Frage 4.4 (6 Punkte): Statusbits. Das Rechenwerk enthält am Ausgang der ALU ein sogenannten Prozessor-Status-Register (PSR, siehe Abbildung zur Architektur eingangs zur Frage). Erläutern Sie die Funktionsweise und den Zweck dieses Registers. Wie wird dieses Register z.B. bei Sprungbefehlen verwendet? Welche Aktionen leitet das Steuerwerk aus diesem Register ab? Wann genau steht dem Steuerwerk diese Information bei der Ausführung eines Befehls zur Verfügung?

Lösung: siehe Skript.

Aufgabe 5 - Rechenwerk des Prozessors (24 Punkte)

Folgender HDL-Text zeigt einen Ausschnitt aus dem Rechenwerk des Prozessors.

process (OPCODE, K, DS\_DO, Akku)

variable RsVar, RdVar, RdoutVar : D\_Type;

variable RVar : signed(D\_Width downto 0); -- 9 bits wide

begin

RsVar := DS\_DO; -- copy Rs from data memory

RdVar := Akku; -- copy Rd from Akku

RdoutVar := Zero; -- avoids latches in synthesis

case OPCODE is

when LSL => CFlag <= RdVar(D\_Width-1); -- msb

RdOutVar(D\_Width-1 downto 1) := RdVar(D\_Width-2 downto 0);

RdOutVar(0):='0'; -- lsb

when LSR => CFlag <= Akku(0); -- lsb

RdOutVar(D\_Width-2 downto 0) := Akku(D\_Width-1 downto 1);

RdOutVar(D\_Width-1):='0'; -- msb

when ADD => RVar := signed('0' & RdVar) + signed('0' & RsVar);

RdOutVar := std\_logic\_vector(RVar(D\_Width-1 downto 0));

NFlag <= RVar(D\_Width);

when SUB => RVar := signed('0' & RdVar) - signed('0' & RsVar);

RdOutVar := std\_logic\_vector(RVar(D\_Width-1 downto 0));

NFlag <= RVar(D\_Width);

when ANDA=> RdOutVar := RdVar AND RsVar;

when EOR => RdOutVar := RdVar XOR RsVar;

when ORA => RdOutVar := RdVar OR RsVar;

when STR | PUSH => DS\_DI <= Akku;

when LDA | POP => RdOutVar := DS\_DO;

when LDI => RdOutVar := K;

when others => null; -- no action

end case;

if (RdOutVar = Zero) then ZFlag <='1'; end if;

RdOut <= RdOutVar;

end process;

load\_akku : process (Clk) begin

if (ACC\_EN = '1') then

if rising\_edge(Clk) then

Akku <= RdOut;

end if;

end if;

end process load\_akku;

end RTL;

Frage 5.1 (6 Punkte): Der HDL-Text lässt sich als Schaltnetz in Kombination mit einem Register inter-pretieren. Skizzieren Sie ein Blockschaltbild mit Eingängen und Ausgängen.

Lösung:



Frage 5.2 (6 Punkte): Skizzieren Sie ein Zeitdiagramm für den folgenden Ablauf mit den Signalen aus dem Blockschaltbild. Hinweis: Verwenden Sie die Vorlage auf dem Aufgabenblatt.

start: ldi 4

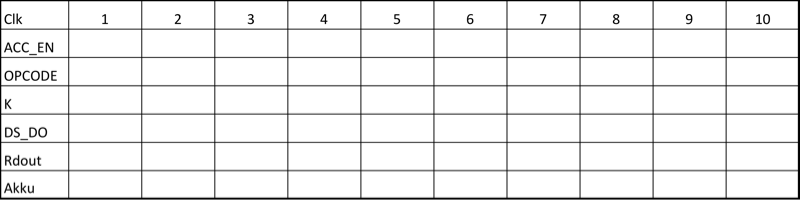
st $0

ldi 2

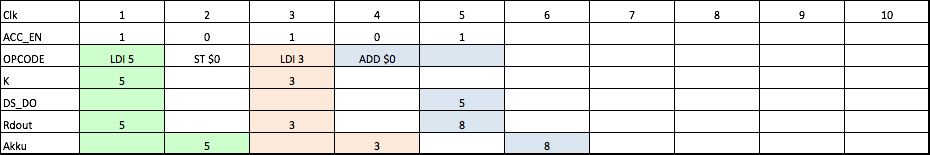
lsl

add $0

end: st $1



Lösung (allgemeines Beispiel, andere Werte, ohne lsl):



Frage 5.3 (6 Punkte): Weshalb ist der Akku als Register ausgeführt (d.h. taktabhängig)? Könnte man den Akku auch als Schaltnetz ausführen um hierdurch das Rechenwerk zu beschleunigen? Begründen Sie Ihre Aussage.

Lösung: Begründung für Register: Für arithmetisch logische Operationen sind 2 Operanden erforderlich. In der vorliegenden Architektur ist einer der Operanden jeweils im Akku gespeichert, der zweite Operand wir aus dem Arbeitsspeicher entnommen. Die eigentliche Rechenoperation findet im Schaltnetz statt (RdOut). Alternative: Beide Operanden aus dem Arbeitsspeicher bzw. aus Registern nehmen. Wenn die Register als Tabellen sofort auslesbar sind, beschleunigt sich hierdurch das Holen der Operanden. Das Speichern der Ergebnisse benötigt jedoch wieder einen Arbeitstakt als definierten Zeitpunkt.

Frage 5.4 (6 Punkte): Durch welche Komponente des Prozessors wird das ACC-EN Signal gesteuert? Woher weiss diese Komponente, welchen Zustand dieses Signal annehmen soll? Begründen Sie Ihre Aussage und nennen Sie ggf. ein Beispiel.

Lösung: Durch das Steuerwerk (Zustandsautomat aus Aufgabe 1). Die Information über den korrekten Zustand der Steuerleitung ACC\_EN (Akku-Enable) ist einerseits im OPCODE unterge-bracht, andererseits im Zustand des jeweiligen Befehls. Arithmetisch logische Befehle, die einen Operanden aus dem Arbeitsspeicher holen müssen, haben z.B. Im ersten Takt keine gültigen Werte für das Rechenwerk. Durch das Steuersignal ACC-EN wird verhindert, dass Operanden im Akku überschrieben werden.