Aufgabe 1: Schaltwerksanalyse (40 Punkte)

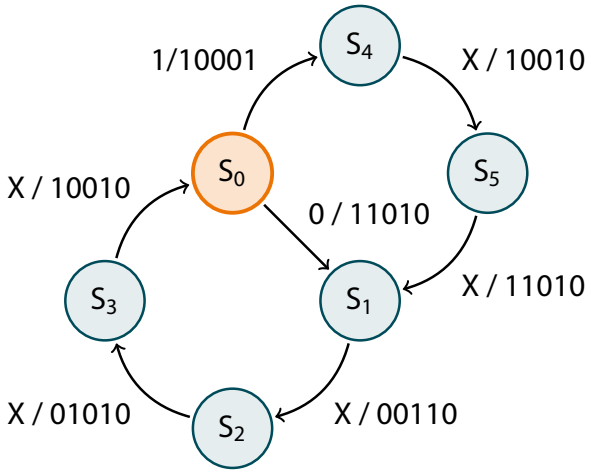


Abbildung 1: Schaltwerk

Betrachten Sie für diese Aufgabe das in Abbildung 1 dargestellte Schaltwerk einer Ampel für den Straßenverkehr und Fußgängerüberweg, dessen Startzustand der Zustand S_0 ist. Die Eingabe x gibt an, ob ein Fußgänger das Signal für die Überquerung angefordert hat $(x = 1)$ oder nicht $(x = 0)$. Der Ausgabevektor $Y = (y_{\mathrm{ro}}, y_{\mathrm{ge}}, y_{\mathrm{gr}}, y_{\mathrm{fr}}, y_{\mathrm{fg}})$ gibt die Beschaltungen der Lampen Autos rot, gelb und grün $(y_{\mathrm{ro}}, y_{\mathrm{ge}}$ und $y_{\mathrm{gr}})$ gefolgt von Fußgänger rot und grün an $(y_{\mathrm{fr}}$ und $y_{\mathrm{fg}})$ an. Beantworten Sie die nachfolgenden Fragen.

a) Um welchen Automatentyp handelt es sich? Bitte begründen Sie Ihre Antwort kurz.

b) Füllen Sie die nachfolgende Zustandsübergangstabelle komplett aus. Nutzen Sie für die Zustandskodierung die Binärdarstellung der angegebenen Zustandsindizes.

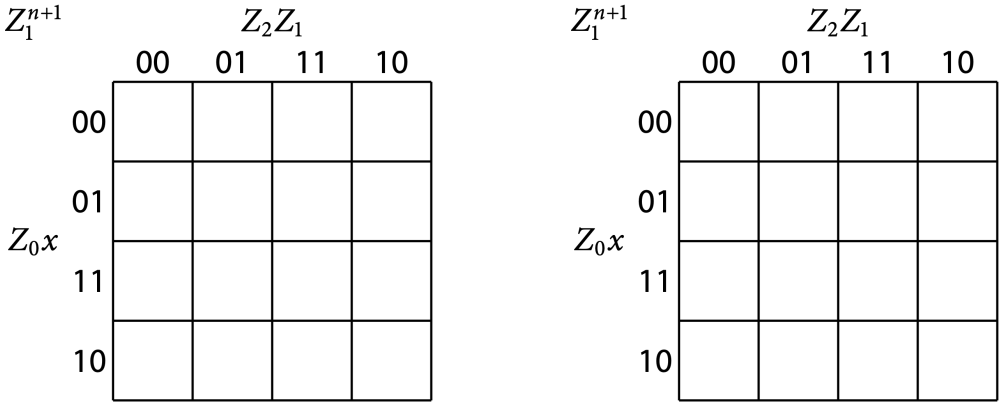
$(Z_2Z_1Z_0)^n$	Folgezustand $(Z_2Z_1Z_0)^{n+1}$		Ausgabe $(y_{\mathrm{ro}}, y_{\mathrm{ge}}, y_{\mathrm{gr}}, y_{\mathrm{fr}}, y_{\mathrm{fg}})$	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000				
001				
010				
011				
100				
101				
110				
111				

c) Die Zustandsübergangsfunktionen des Schaltwerks sind gegeben als:

$$Z_{1,\mathrm{DKN}}^{n+1} = \left(\overline{Z}_2 Z_1 \overline{Z}_0 \overline{x} + \overline{Z}_2 Z_1 \overline{Z}_0 x + \overline{Z}_2 \overline{Z}_1 Z_0 x + \overline{Z}_2 \overline{Z}_1 Z_0 \overline{x}\right)^n$$

$$Z_{0,\mathrm{KMF}}^{n+1} = (Z_2 + Z_1 + \overline{x})^n \left(Z_2 + \overline{Z}_0\right)^n$$

Bestimmen Sie $Z_{1,\mathbf{KMF}}^{n+1}$, indem Sie eine Minimierung mit Hilfe des nachfolgenden KV-Diagramms (1 Diagramm als Ersatz!) durchführen und anschließend $Z_{1,\mathbf{KMF}}^{n+1}$ explizit angeben. Berücksichtigen Sie sich eventuell aus b) ergebende „don't care"-Terme.



d) Bestimmen Sie auf Basis von $Z_{1,\mathrm{DKN}}^{n+1}$ die Ansteuergleichung für ein D-Flipflop unter der Bedingung, dass es in einer Realisierung das Zustandsbit Z_1 repräsentieren soll.

Was ist der Nachteil, wenn man $Z_{1,\mathrm{DKN}}^{n+1}$ statt $Z_{1,\mathrm{DMF}}^{n+1}$ als Ansteuergleichung für das D-Flipflop im Rahmen der Realisierung des Schaltwerks nutzt?

Bestimmen Sie auf Basis von $Z_{0,\mathrm{KMF}}^{n+1}$ die Ansteuergleichung für ein JK-Flipflop unter der Bedingung, dass es in einer Realisierung das Zustandsbit Z_0 repräsentieren soll. Gibt es etwas bei der Verwendung Ihrer Ansteuergleichung zu berücksichtigen?

Hinweis: Ihnen stehen für eine Realisierung unter anderem Inverter zur Verfügung.

e) Geben Sie auf Basis der Zustandsübergangstabelle aus b) die Ausgabefunktionen $y^n_{\mathrm{fg,DMF}}$ und $y^n_{\mathrm{fr,KMF}}$ an.

f) Realisieren Sie das Schaltwerk mit Hilfe von VHDL, indem Sie sowohl die Entity als auch die Architecture (Folgeseite) angeben und den nachfolgend gegebenen VHDL-Code vervollständigen. Die Ausgabe soll als Vektor erfolgen.

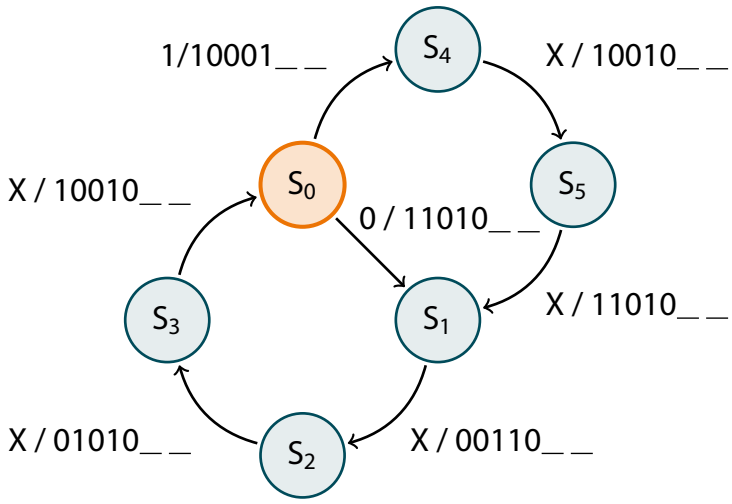
```
entity Automat is

end Automat;
```

```
architecture Behavioral of Automat is









begin

    Zustandsregister : process (clock, reset)
    begin
        if reset = '1' then
            state <= S0;
        elsif rising_edge(clock) then
            state <= next_state;
        end if;
    end process;


    Zustandsuebergangslogik : process (state, x)
    begin
        case state is
            when S0 =>
                if x = '1' then
                    next_state <= S4;
                else
                    next_state <= S1;
                end if;

            when S1 =>
                next_state <= S2;

            when S2 =>
                next_state <= S3;

            when S3 =>
                next_state <= S0;

            when S4 =>
                next_state <= S5;

            when S5 =>
                next_state <= S1;
        end case;
    end process;
```

```
    Ausgabelogik :

end Behavioral;
```

g) Das Schaltwerk soll um die Steuerung einer Ampel für Fahrräder mit den zusätzlichen Phasen Rot und Grün erweitert werden. Die Fahrräder fahren direkt nach den Autos und immer vor den Fußgängern. Der neue Ausgabevektor $Y = (y_{\mathrm{ro}}, y_{\mathrm{ge}}, y_{\mathrm{gr}}, y_{\mathrm{fr}}, y_{\mathrm{fg}}, y_{\mathrm{rr}}, y_{\mathrm{rg}})$ beinhaltet die zusätzlichen Ausgaben y_{rr} für die rote und y_{rg} für die grüne Lampe der Fahrradampel.

Erweitern Sie den Automaten um die eventuell notwendigen Zustände und Ausgaben (mit Notation!). Streichen Sie gegebenenfalls Kanten.



Welche der nachfolgenden Aussagen sind für Ihren abgeänderten Automaten im Vergleich zur vorherigen Version wahr bzw. falsch? Bitte kreuzen Sie entsprechend an. Falsche Kreuze führen zu Punktabzügen, allerdings kann es keine negativen Punkte für diese Teilaufgabe geben.

wahr	falsch	Aussage
☐	☐	Die Anzahl der Zustände ändert sich nicht.
☐	☐	Es werden mehr Zustandsbits benötigt.
☐	☐	Die Ansteuergleichungen aller Flipflops ändern sich.
☐	☐	Eine Realisierung als Moore-Automat hätte weniger Zustände.
☐	☐	Die Ausgabe des Automaten bei gleicher Eingabe ändert sich.
☐	☐	Für die Realisierung werden auf jeden Fall mehr Flipflops benötigt als bei der gegebenen Variante.
☐	☐	Es können keine JK-Flipflops mehr für die Realisierung genutzt werden.

$$\Sigma_{A1} = \_\_\_ \text{ / 40 Punkte}$$

Aufgabe 2: Einfache CPU (30 Punkte)

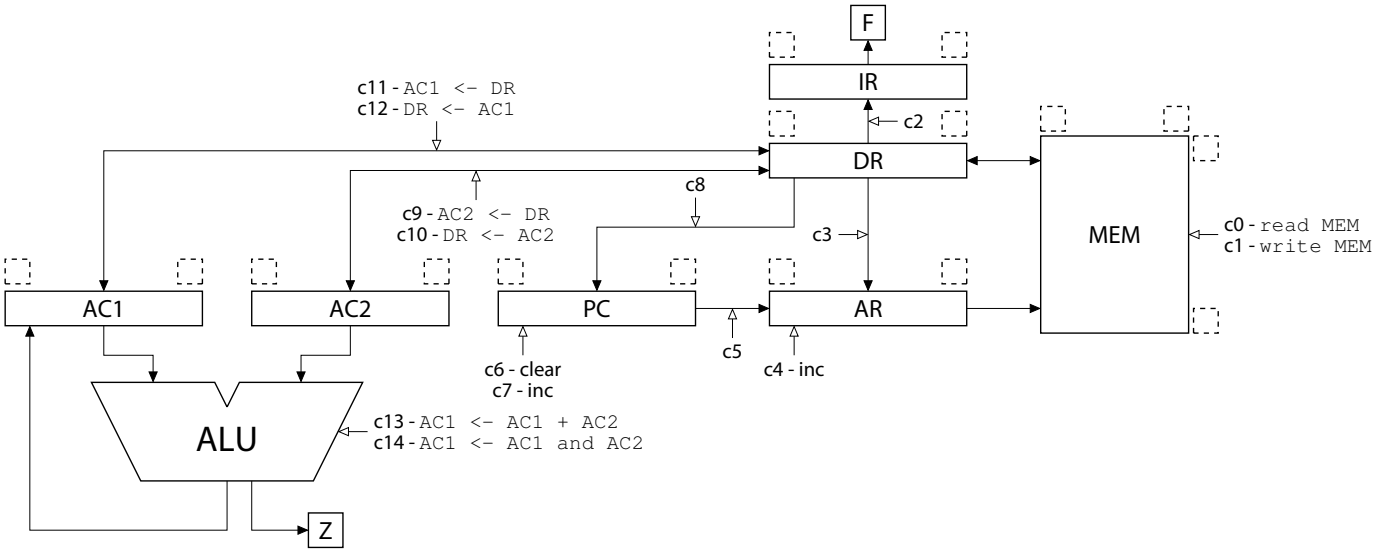


Abbildung 2: Operationswerk

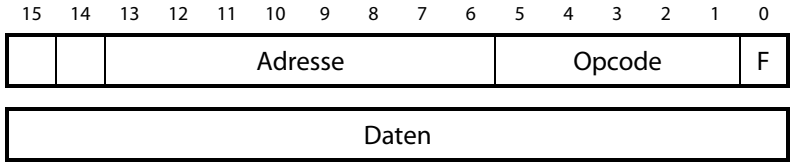


Abbildung 3: Befehlsformat

Gegeben sei das in Abbildung 2 dargestellte Operationswerk (OW) mit dem Speicher MEM, der bis zu 256 Werte aufnehmen kann. Das Befehlsformat der CPU ist in Abbildung 3 dargestellt. Das LSB erlaubt es, einzelnen Befehlen ein Flag (F) als Teil des Opcodes, der in das IR geladen wird, mitzugeben. Sollte es sich bei dem Speicherinhalt nicht um einen Befehl handeln, so werden die kompletten 16 Bit als Daten interpretiert und müssen von der CPU verarbeitbar sein. Die CPU stellt zwei Flags zur Verfügung. Das Z-Flag gibt an, ob das Ergebnis der letzten Operation eine Null (Z=1) war. Das F-Flag gibt an, ob der Inhalt des Datenregisters ungerade ist (F=1).

a) Ergänzen Sie im OW aus Abbildung 2 die fehlenden Breitenangaben der einzelnen Register in den gestrichelten Kästchen. Jedes Register soll nur so breit sein, wie es notwendig ist.

b) Das gegebene Operationswerk soll um ein mikroprogrammiertes Steuerwerk zu einer mikroprogrammierten CPU ergänzt werden, die über den nachfolgenden Befehlssatz verfügt.

Opcode	F	Befehl	Beschreibung
0	0/1	`LOAD X`	Lädt den sich unter Adresse X im Speicher befindenden Wert in das Akkumulator-Register AC1 wenn LSB=0 (F=0) sonst in AC2.
1	0/1	`STORE X`	Speichert den sich in AC1 befindenden Wert unter der Adresse X im Speicher, wenn LSB=0 (F=0) gilt, sonst den Wert aus AC2.
2	0	`ADD`	Addiert den Wert von AC2 auf den von AC1 und legt das Ergebnis in AC1 ab.
3	0	`JMP X`	Setzt den Programmablauf an Adresse X fort
4	0	`JMPZ X`	Setzt den Programmablauf an Adresse X fort, wenn Z = 1 gilt

Ergänzen Sie im nachfolgenden RT-Programm die fehlenden Registerbreiten und Indizes sowie die zu den Registertransferoperationen korrespondierenden Kontrollsignale (rechts neben dem Code. Wo keine Linie ist, soll auch kein Signal angegeben werden).
Implementieren Sie zudem die Befehle LOAD und STORE und geben Sie die entsprechenden Kontrollsignale an.
Hinweis: Das Setzten beider Flags ist im nachfolgenden Quellcode nicht implementiert. Dennoch müssen die Flags bei Bedarf von Ihnen genutzt werden.

```
declare register  AC1(       ), AC2(       ), DR(       ), AR(       ),
                  PC(       ), IR(       ), Z, F

declare memory MEM(        )


INIT:   PC <- 0;                                                  _____

FETCH:  AR <- PC, PC <- PC + 1;                                   _____
        read MEM;                                                 _____
        IR <- DR(      ) | switch IR(       ) {                  _____
        case 0: goto LOAD
        case 1: goto STORE
        case 2: goto ADD
        case 3: goto JMP
        case 4: goto JMPZ
        default: goto FETCH };

ADD:    AC1 <- AC1 + AC2 | goto FETCH;                            _____

JMP:    PC <- DR(     ) | goto FETCH;                             _____

JMPZ:   if Z = 1 then
          PC <- DR(      ), goto FETCH                           _____
        else goto FETCH fi;

LOAD:   # Wird von Ihnen realisiert




STORE:  # Wird von Ihnen realisiert
```

c) Erstellen Sie ein horizontal mikroprogrammiertes Steuerwerk, welches das durch den RT-Code in b) spezifizierte Verhalten inklusive des **LOAD** Befehls realisiert. Füllen Sie hierfür die Tabelle auf der nachfolgenden Seite aus. Leere Felder werden hierbei als 0 interpretiert. Sie müssen also nur die 1en eintragen. Ergänzen Sie zudem den für das Mikroprogramm eventuell abgeänderten RT-Code des LOAD Befehls. Es stehen Ihnen **ausschließlich** die nachfolgenden Condition Select Signale zur Verfügung:

Condition Select	Funktion
000	Nicht springen
001	Springe zu der dekodierten Opcode-Adresse (Mapping-ROM)
010	Springe, falls **Z = 0**
011	Springe, falls F = 1
111	Springe unbedingt

d) Geben Sie das Mapping ROM an.

e) Nennen Sie einen Vor- und einen Nachteil der Realisierung des Steuerwerks als horizontales Mikroprogramm im Vergleich zu einem Mikroprogramm mit mehrfachen Befehlsformat.

f) Nennen Sie einen Vor- und einen Nachteil der Realisierung des Steuerwerks als festverdrahtetes Steuerwerk im Vergleich zu einem mikroprogrammierten Steuerwerk.

Adresse					cs			Sprungadresse					Horizontale Kodierung Kontrollsignale c[14:0]														Entsprechender RT-Befehl
4	3	2	1	0	2	1	0	4	3	2	1	0	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0																								INIT: PC <- 0;
0	0	0	0	1																								FETCH: AR <- PC, PC <- PC + 1;
0	0	0	1	0																								read MEM;
0	0	0	1	1																								IR <- DR( ) | switch IR( )
0	0	1	0	0																								ADD: AC1 <- AC1 + AC2 | goto FETCH;
0	0	1	0	1																								JMP: PC <- DR( ) | goto FETCH
0	0	1	1	0																								JMPZ:
0	0	1	1	1																								
0	1	0	0	0																								LOAD:
0	1	0	0	1																								
0	1	0	1	0																								
0	1	0	1	1																								
0	1	1	0	0																								
0	1	1	0	1																								

$\Sigma_{A2} = \_\_\_ / 30$

Aufgabe 3: RISC-V und EduCore-V Tiny (24 Punkte)

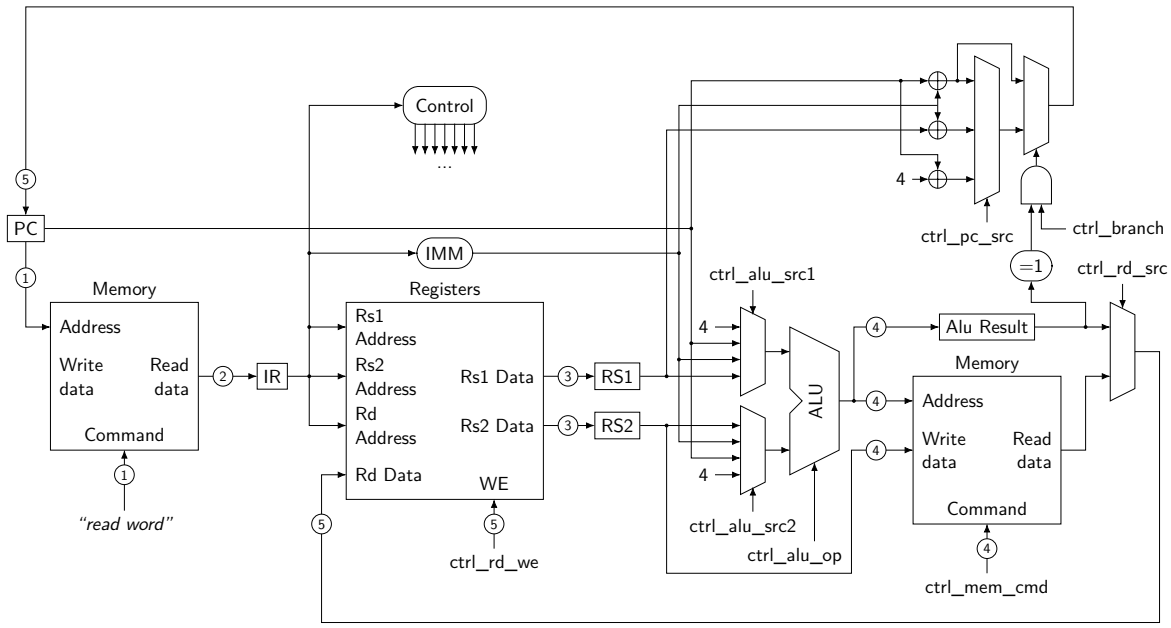


Abbildung 4: Schematische Darstellung des EduCore-V Tiny

31–25	24–20	19–15	14–12	11–7	6–0
func7	rs2	rs1	func3	rd	opcode
0 0 0 0 0 1 0			0 0 0		0 1 1 0 0 1 1

Abbildung 5: Befehlsformat des `vadd` Befehls

`ctrl_alu_src1`	`REG, IMM, PC, 4`
`ctrl_alu_src2`	`REG, IMM, PC, 4`
`ctrl_mem_cmd`	`WRITE_B, WRITE_H, WRITE_W, READ_B, READ_H, READ_W, READ_BU, READ_HU, NOP`
`ctrl_alu_op`	`NOP, OP1, OP2, ADD, SUB, OR, AND, XOR, SLL, SRL, SRA, EQ, NEQ, LT, GE, LTU, GEU,` **`VADD`**
`ctrl_rd_src`	`ALU_RESULT, MEM_DATA`
`ctrl_rd_we`	`false, true`
`ctrl_pc_src`	`PC_NEXT, PC_IMM, RS1_IMM`
`ctrl_branch`	`false, true`

Abbildung 6: Kontrollsignale ergänzt um `vadd`

In dieser Aufgabe sollen Sie den RV32I Basisbefehlssatz des in Abbildung 4 schematisch dargestellten EduCore-V Tiny um die Vektoraddition erweitern. Der Befehl `vadd` soll die Addition zweier vierelementiger Vektoren $u \in \mathbb{N}^4$ und $v \in \mathbb{N}^4$ in der Art vornehmen, dass

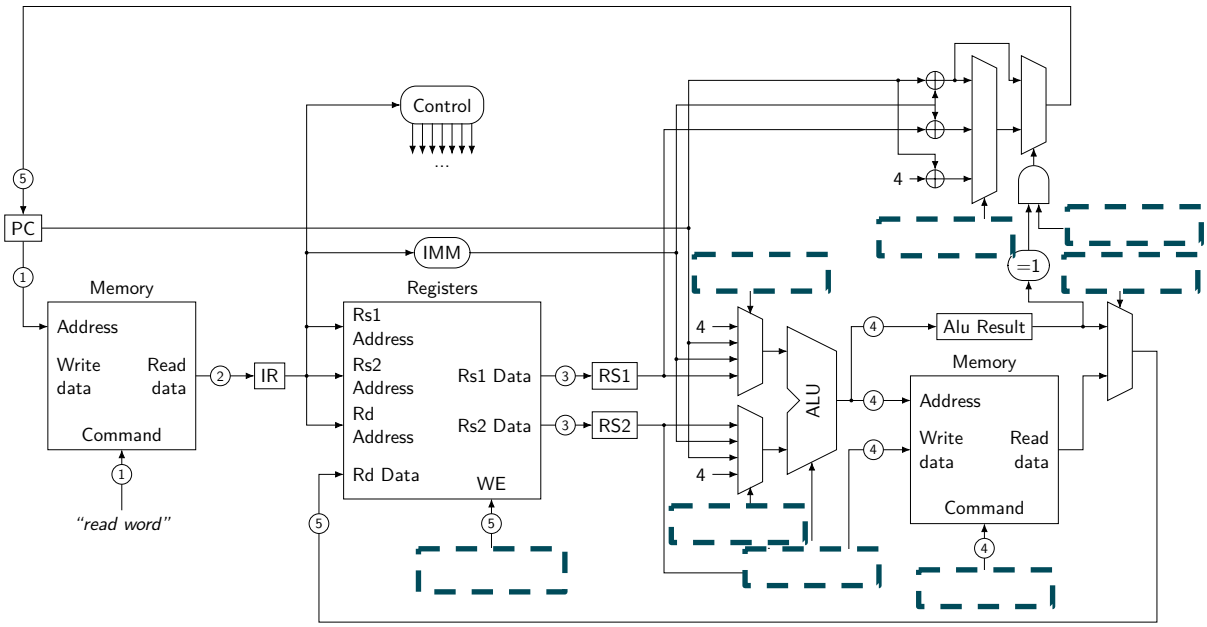
$$x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} u_0 + v_0 \\ u_1 + v_1 \\ u_2 + v_2 \\ u_3 + v_3 \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{pmatrix} + \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} = u + v$$

wobei die einzelnen Elemente der Vektoren $0 \leq u_j, v_i \leq 255 : 0 \leq j, i \leq 3$ gilt.
Sollte bei der Addition der einzelnen Elemente für x_i ein Wert größer 255 herauskommen, so ergibt sich der Endwert als $x_i = x_i \bmod 256$. Ein einzelner Vektor wird als 32 Bit Wert dargestellt.

Gemäß Abbildung 5 soll der Befehl `vadd` zu den Register-Register Befehlen gehören und den Opcode 0110011 erhalten. Um eine Differenzierung zwischen den restlichen Register-Register Befehlen zu gewährleisten wird im funct7 Teil das zweite Bit (Bit 26) gesetzt, wohingegen der funct3 Teil komplett bei Null belassen wird.

a) Realisieren Sie die Vektoraddition unter der ausschließlichen Verwendung des RV32I Basisbefehlssatzes sowie der Pseudoinstruktionen als RISC-V Assembler Programm. Laden Sie in die beiden Register `x10` und `x11` die beiden Vektoren $(88{,}99, AA, BB)^T$ sowie $(CC, DD, EE, FF)^T$ und führen Sie die Addition durch, wobei das Ergebnis nach der Berechnung in Register `x10` abgelegt werden soll.

b) Geben Sie die vom Kontrollsignalgenerator für die Realisierung des `vadd` Befehls zu erzeugenden Kontrollsignale an, indem Sie **sämtliche** Signale in der folgenden schematischen Darstellung des EduCore-V Tiny ergänzen (gestrichelte Kästchen ausfüllen). Sie finden die vorhandenen Signale in Abbildung 6.



c) Ergänzen Sie nun den nachfolgenden Codeausschnitt der ALU um den `vadd` Befehl. Nutzen Sie für die case-Abfrage die Konstante `CTRL_ALU_OP_VADD`.

```
architecture rtl of m_alu is
    constant CONST_ZERO : std_logic_vector(31 downto 0) := (others => '0');
    constant CONST_ONE  : std_logic_vector(31 downto 0) := (0 => '1', others => '0');

begin

    process(reg_op1, reg_op2, operation)
    begin
        case (operation) is
            -- ...
            when CTRL_ALU_OP_ADD
                 => result <= std_logic_vector(unsigned(reg_op1) + unsigned(reg_op2));

            when CTRL_ALU_OP_SUB
                 => result <= std_logic_vector(unsigned(reg_op1) - unsigned(reg_op2));

            when CTRL_ALU_OP_MERGE
                 => result <= reg_op1(31 downto 16) & reg_op2(15 downto 0) ;

            when CTRL_ALU_OP_OR    => result <= reg_op1 or  reg_op2;
            when CTRL_ALU_OP_AND   => result <= reg_op1 and reg_op2;
            when CTRL_ALU_OP_XOR   => result <= reg_op1 xor reg_op2;
-- LOESUNG -











            -- ...
            when others            => result <= CONST_ZERO;
        end case;
    end process;

end architecture rtl;
```

d) Realisieren Sie die Vektoraddition unter Verwendung des von Ihnen erweiterten RV32I Basisbefehlssatzes sowie Pseudoinstruktionen als RISC-V Assembler Programm. Laden Sie in die beiden Register `x10` und `x11` die beiden Vektoren $(88{,}99, AA, BB)^T$ sowie $(CC, DD, EE, FF)^T$ und führen Sie die Addition durch, wobei das Ergebnis nach der Berechnung in Register `x10` abgelegt werden soll.
Nutzen Sie explizit den Aufruf Ihres eben realisierten Befehls `vadd x10, x10, x11`. Bedenke Sie, dass der Compiler Ihren Befehl nicht kennt.

Assemblercode

e) Geben Sie den resultierenden Speicherinhalt (nachfolgende Tabelle) des EduCore-V Tiny direkt nach dem Uploaden Ihres Programms aus d) an. Obwohl das gesamte Programm im Speicher liegt, müssen Sie nur die Zeilen von `vadd` angeben. Bedenken Sie beim Angeben des Speichers, dass jede Speicherstelle ein Byte aufnehmen kann und als Byte-Reihenfolge im Speicher Little Endian genutzt wird. Sie können die Inhalte in der Binär- oder Hexadezimaldarstellung angeben.

Adr.	Speicherinhalt
00	
01	
02	
03	
04	
05	
06	
07	

Adr.	Speicherinhalt
08	
09	
0A	
0B	
0C	
0D	
0E	
0F	

Adr.	Speicherinhalt
10	
11	
12	
13	
14	
15	
16	
17	

$\Sigma_{A3} =$ ___ / 24 Punkte

Aufgabe 4: Allgemeine Fragen (6 Punkte)

a) Welchen Vorteil bietet ein Master-Slave Flipflop?

b) Welchen Vorteil bietet ein JK-Flipflop?

c) Wodurch wird der Unterschied zwischen Moore- und Mealy-Timing im Schaltbild einer Verzögerungskette deutlich?

d) Warum ist es im Moore-Timing sinnvoll, dass Steuer- und Operationswerk auf unterschiedliche Flanken reagieren?

e) Nennen Sie einen Vor- und einen Nachteil der RISC-ISA gegenüber der CISC-ISA.

$$\Sigma_{A4} = \_\_\_ / 6 \text{ Punkte}$$