Aufgabe 1: Schaltwerksanalyse (40 Punkte)

Betrachten Sie das nachfolgende Schaltwerk:

```
-- TODO: Entity angeben

architecture Behavioral of Rechner is

    signal x: STD_LOGIC_VECTOR(1 downto 0);
    signal y: STD_LOGIC_VECTOR(1 downto 0);

    type state_type is (A, B, C, D, ERROR);
    signal state      : state_type := A;
    signal next_state : state_type := A;

begin

    x <= x1 & x0;

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

    process_State_Logic : process (state, x)
    begin

        case state is
            when A =>
                case x is
                    when "01"   => next_state <= C;
                    when "10"   => next_state <= B;
                    when others => next_state <= ERROR;
                end case;
            when B =>
                case x is
                    when "00"   => next_state <= C;
                    when "01"   => next_state <= B;
                    when others => next_state <= ERROR;
                end case;
            when C =>
                case x is
                    when "11"   => next_state <= D;
                    when others => next_state <= ERROR;
                end case;
            when D =>
                case x is
                    when "01"   => next_state <= D;
                    when "10"   => next_state <= B;
                    when "11"   => next_state <= B;
                    when others => next_state <= ERROR;
                end case;
            when others => next_state <= ERROR;
        end case;

    end process process_State_Logic;
```

```
    process_State_Output : process (state)
    begin

        case state is
            when A =>
                case x is
                    when "01"   => y <= "10";
                    when "10"   => y <= "11";
                    when others => y <= "00";
                end case;
            when B =>
                case x is
                    when "00"   => y <= "10";
                    when "01"   => y <= "11";
                    when others => y <= "00";
                end case;
            when C =>
                case x is
                    when "11"   => y <= "11";
                    when others => y <= "00";
                end case;
            when D =>
                case x is
                    when "10"   => y <= "10";
                    when "11"   => y <= "01";
                    when others => y <= "00";
                end case;
            when others => y <= "00";
        end case;

    end process process_State_Output;

    y1 <= y(1);
    y0 <= y(0);

end Behavioral;
```

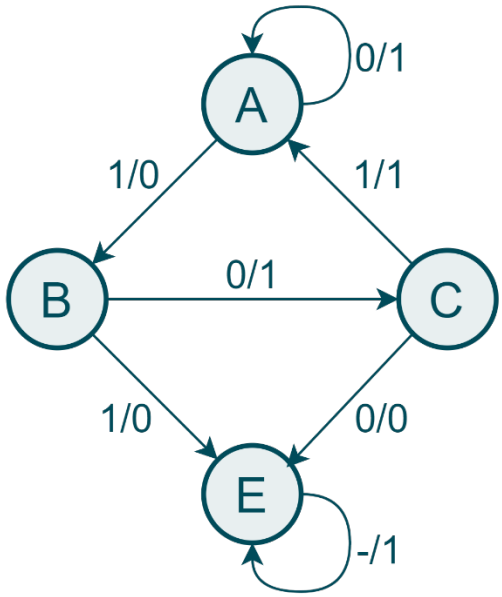
1) Vervollständigen Sie den oben angegebenen VHDL-Code, indem Sie die Entity angeben.

2) Welches Timing weist das Schaltwerk auf (Mealy oder Moore)? Ist das Schaltwerk partiell definiert? Bitte begründen Sie Ihre Antwort kurz.

3) Welche Ausgaben (y_1y_0) erzeugt die taktsequenzielle Eingabe (x_1x_0) von 10 – 01 – 00 – 10 – 11? Starten Sie im Zustand A und geben Sie ebenfalls die Sequenz der Zustände an. Falls es ein Problem mit der Eingabesequenz geben sollte, beschreiben Sie es kurz.

4) Zeichnen Sie einen zum Schaltwerk äquivalenten Zustandsübergangsgraphen. Vernachlässigen Sie dabei den Zustand ERROR und alle Eingaben, die zu diesem Zustand führen. Die Zustände sollen die durch die Typendefinition in VHDL gegebenen Bezeichnungen (A, B, C und D) erhalten.
   a) Um welchen Automatentypen handelt es sich bei Ihrem gezeichneten Zustandsübergangsgraphen? Begründen Sie Ihre Antwort kurz. Geben Sie an, warum Sie sich für diesen Automatentypen entschieden haben.
   b) Ist der von Ihnen erstellte Automat vollständig oder partiell definiert? Begründen Sie Ihre Antwort kurz.

5) Geben Sie für den nachfolgenden Automaten eine Verzögerungskette mit äquivalentem Verhalten an. Ergänzen Sie zudem das Startsignal S zum Initialisieren des Verzögerungselementes des Zustands A.



6) Gegeben sei die nachfolgende Zustandsübergangs- und Ausgabetabelle eines Automaten

$(Z_1Z_0)^n$	Folgezustand $(Z_1Z_0)^{n+1}$ bei Eingabe x_1x_0				Ausgabe y bei Eingabe x_1x_0			
	00	01	10	11	00	01	10	11
00	10	01	00	01	1	1	1	1
01	01	01	01	00	1	1	1	1
10	10	xx	00	10	0	x	0	0
11	10	xx	00	10	0	x	0	0

   a) Bestimmen Sie $Z_{1,\mathbf{DMF}}^{n+1}$ mithilfe des Verfahrens von Quine und McCluskey (QMC). Nutzen Sie dafür die nachfolgende Tabelle bzw. Ihr vorbereitetes Muster. Nutzen Sie die in der VL gegebene Kodierung (nicht negierte Variable – 1, Negierte Variable – 0). Markieren Sie die don't care Terme, indem Sie diese einklammern. Kennzeichnen Sie alle Primimplikanten mit einem Stern.

Klasse	Nr.	Minterme $abcd$	Verschmolzene Terme (Nr.)	Neue Terme	Verschmolzene Terme (Nr.)	Neue Terme
K0	0	0010	0,2	00-0	…	…

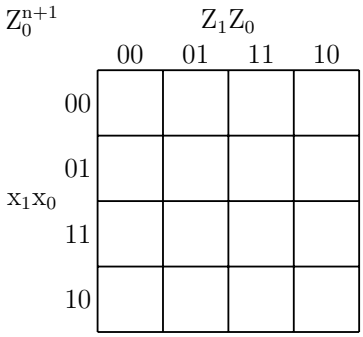
$\bar{a}\bar{b}cd$

Beispielhaft ausgefüllte erste Zeile einer QMC-Tabelle ohne Zusammenhang mit diesem Schaltwerk

Klasse	Nr.	Minterme $Z_1Z_0X_1X_0$	Verschmolzene Terme (Nr.)	Neue Terme	Verschmolzene Terme (Nr.)	Neue Terme

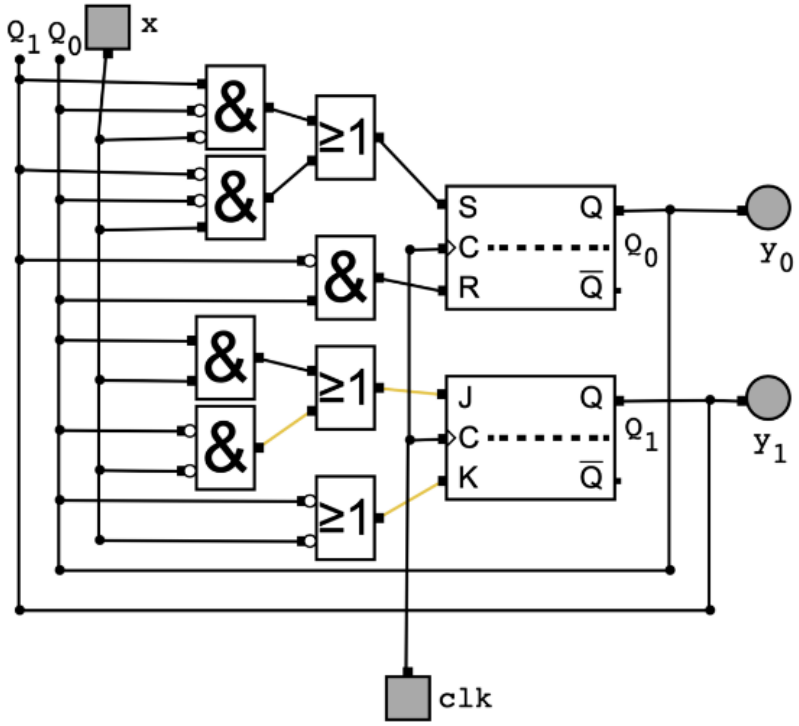
Warum ist eine Primimplikantentafel in diesem Beispiel überflüssig? Begründen Sie kurz und geben Sie $Z^{n+1}_{1,\mathbf{DMF}}$ explizit an.

b) Bestimmen Sie nun sämtliche Formen von $Z^{n+1}_{0,\mathbf{KMF}}$, indem Sie eine Minimierung mit Hilfe eines KV-Diagramms der nachfolgenden Form durchführen und anschließend alle Formen von $Z^{n+1}_{0,\mathbf{KMF}}$ explizit angeben.



c) Geben Sie sowohl $y_{\mathbf{DMF}}$ als auch $y_{\mathbf{KMF}}$ an.

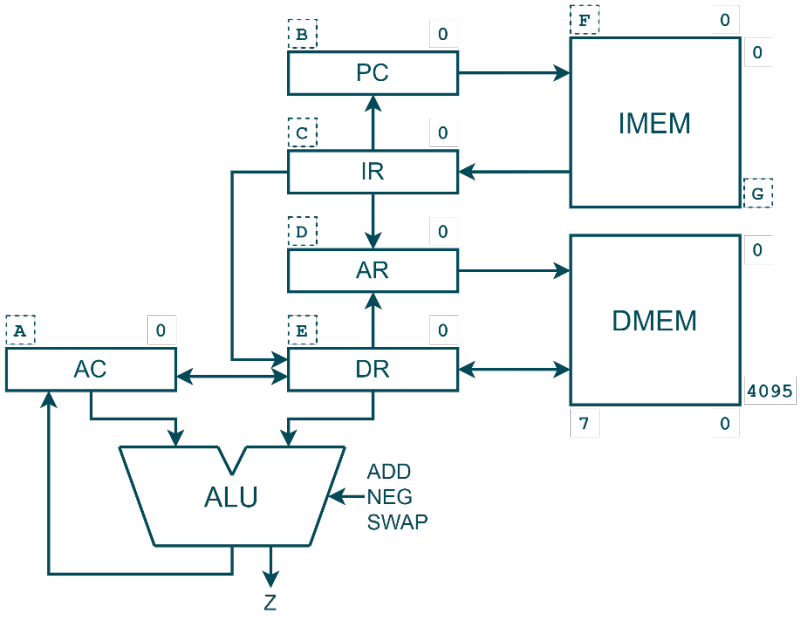
7) Gegeben sei das nachfolgende Schaltwerk



a) Um welchen Automatentypen handelt es sich. Begründen Sie Ihre Antwort kurz.
b) Geben Sie sämtliche Ansteuergleichungen der beiden Flipflops an.
c) Geben Sie die Zustandsübergangsfunktionen für die beiden Zustandsbits Z_1 und Z_0 als DNF an, indem Sie diese aus den Ansteuergleichungen sowie den charakteristischen Funktionen der Flipflops herleiten. Ist im Falle des RS-Flipflops die notwendige Nebenbedingung erfüllt? Weisen Sie Ihre Antwort nach.
d) Liegt die von Ihnen bestimmte Zustandsübergangsfunktion des Zustandsbits Z_0 als DMF vor? Begründen Sie Ihre Antwort.
e) Geben Sie die Ausgabefunktionen des Automaten an.
f) Nehmen Sie an, das Schaltwerk soll nun mit Hilfe von flankengesteuerten D-Flipflops realisiert werden. Wieviel D-Flipflops werden benötigt? Begründen Sie Ihre Antwort.
g) Welche der nachfolgenden Aussagen sind für den mit Hilfe von D-Flipflops realisierten Automaten im Vergleich zur vorherigen Version wahr bzw. falsch? Bitte geben Sie die Antwort (wahr/falsch) für jede Frage explizit an. Falsche Antworten führen zu Punktabzügen, allerdings kann es keine negativen Punkte für diese Teilaufgabe geben.

	Aussage
i	Die Anzahl der Zustände ändert sich nicht.
ii	Es werden mehr Zustandsbits benötigt als zuvor.
iii	Die Ausgabefunktionen sind identisch.
iv	Das zeitliche Verhalten des Automaten ändert sich nicht.
v	Die Ausgabe des Automaten bei gleicher Eingabe ändert sich nicht.
vi	Es gibt jetzt keinen äquivalenten Mealy-Automaten mehr.

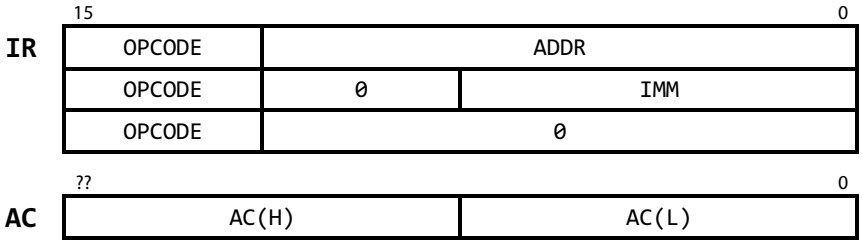
Aufgabe 2: Einfache CPU (34 Punkte)



c_0 – PC <- 0	c_5 – AR <- DR	c_{10} – write DMEM
c_1 – PC <- PC + 1	c_6 – DR <- IR(IMM)	c_{11} – AC <- DR
c_2 – PC <- IR(ADDR)	c_7 – DR <- AC(L)	c_{12} – AC <- AC + DR ≡ *"ADD"*
c_3 – AR <- AR + 1	c_8 – read IMEM	c_{13} – AC <- (not DR) + 1 ≡ *"NEG"*
c_4 – AR <- IR(ADDR)	c_9 – read DMEM	c_{14} – AC <- AC(L).AC(H) ≡ *"SWAP"*

Hinweis: - ADDR, IMM, L, H sind Platzhalter für die tatsächlichen Registerindizierungen
- Bei den Operationen AC <- AC + DR und AC <- DR, AR <- DR werden die Registerbreiten automatisch angeglichen und die fehlenden Bereiche entsprechend mit 0 aufgefüllt, sodass der Zahlenwert der einzelnen Register erhalten bleibt

Steuersignale



Befehls-/Registerformate

Opcode	Befehl		Beschreibung
0	JMP	ADDR	PC = ADDR
1	ADDX	IMM	AC = AC + IMM
2	ADDI	ADDR	AC = AC + MEM[MEM[ADDR]]
3	NEG		AC = -AC
4	BRZ	ADDR	if Z=1 then PC = ADDR
5	DLOAD	ADDR	AC(H) = MEM[ADDR], AC(L) = MEM[ADDR+1]
6	DSTORE	ADDR	MEM[ADDR] = AC(H), MEM[ADDR+1] = AC(L)

Befehlssatz

Gegeben sei das oben dargestellte Operationswerk (OW) mit dem Instruktionsspeicher IMEM und dem Datenspeicher DMEM, die jeweils die gleiche Anzahl an Elementen aufnehmen können. Der Adressbereich eines Befehls (ADDR) muss groß genug sein, um beide Speicher vollständig addresieren können. Das Register AC ist doppelt so groß wie das Datenregister (DR) und wird in die beiden gleich großen Bereiche High (AC(H)) und Low (AC(L)) unterteilt.

Eventuell durch die Operationen der ALU auftretenden Überläufe im AC Register sollen ignoriert werden. Die CPU stellt das Z-Flag bereit, welches angibt, ob das Ergebnis der letzten Operation eine Null (Z=1) war.

1) Ergänzen Sie im OW die fehlenden Breitenangaben der einzelnen Register indem Sie den Buchstabe in den gestrichelten Kästchen jeweils die Breite des entsprechenden Registers zuordnen. Jedes Register soll nur so breit sein, wie es notwendig ist. Sollten Sie einige Werte nicht direkt berechnen können genügt die Angabe als Potenz von 2 (beipielsweise 2^2 statt 4). Definieren Sie zudem die Bereiche OPCODE, ADDR, IMM, H und L.

   A) ___ B) ___ C) ___ D) ___ E) ___ F) ___ G) ___

   OPCODE) __:__ ADDR) __:__ IMM) __:__ H) __:__ L) __:__

2) Was ist bei der Nutzung des ADDI Befehls zu beachten?

3) Deklarieren Sie im nachfolgenden RT-Programm die benötigten Register und den Speicher und notieren Sie die fehlenden Indizierungen (unter Angabe der roten Buchstaben).

   Implementieren Sie zudem die Befehle **DLOAD** sowie **DSTORE** unter Einhaltung des **Moore-Timings**.

   Hinweis: Das Setzen von Z ist im nachfolgenden Quellcode nicht implementiert. Dennoch muss dieses Flag bei Bedarf von Ihnen genutzt werden.

```
1.  -- TODO: Deklaration der Register
2.  -- TODO: Deklaration der Speicher
3.
4.  INIT:   PC <- 0;
5.  FETCH:  read IMEM, PC <- PC + 1 | switch IR( a ) {
6.              case 0:  goto JMP
7.              case 1:  goto ADDX
8.              case 2:  goto ADDI
9.              case 3:  goto NEG
10.             case 4:  goto BRZ
11.             case 5:  goto DLOAD
12.             case 6:  goto DSTORE
13.             default: goto FETCH
14.         };
15.
16. JMP:    PC <- IR( b ) | goto FETCH;
17. NEG:    AC <- (not AC) + 1 | goto FETCH;
18. ADDX:   DR <- IR( c ) | goto ADD;
19. ADDI:   AR <- IR( d );
20.         read DMEM;
21.         AR <- DR;
22.         read DMEM;
23. ADD:    AC <- AC + DR | goto FETCH;
24. BRZ:    if (AC = 0) then
25.             PC <- IR( e )
26.         fi | goto FETCH;
27.
28. DLOAD:  -- TODO: Implementieren
29. DSTORE: -- TODO: Implementieren
```

c_0	PC <- 0
c_1	PC <- PC + 1
c_2	PC <- IR(ADDR)
c_3	AR <- AR + 1
c_4	AR <- IR(ADDR)
c_5	AR <- DR
c_6	DR <- IR(IMM)
c_7	DR <- AC(L)
c_8	read IMEM
c_9	read DMEM
c_{10}	write DMEM
c_{11}	AC <- DR
c_{12}	AC <- AC + DR
c_{13}	AC <- (not DR) + 1
c_{14}	AC <- AC(L).AC(H)

4) Erstellen Sie ein mikroprogrammiertes Steuerwerk, welches das durch den **gegebenen** RT-Code aus Aufgabe 3) spezifizierte Verhalten realisiert. Verwenden Sie hierbei das multiple/mehrfache Befehlsformat sowie eine horizontale Kodierung der Steuersignale. **DLOAD** und **DSTORE** müssen **nicht** realisiert werden! Füllen Sie hierfür die nachfolgende Tabelle aus. Sollten Sie sich diese Tabelle im Vorfeld ausgedruckt oder händisch vorbereitet haben, verwenden Sie diese und übertragen sie alle Informationen aus der Vorlage, anderenfalls müssen Sie die Tabelle abzeichnen. Leere Felder werden hierbei als 0 interpretiert. Sie müssen also nur die 1en eintragen. Ergänzen Sie zudem den für das Mikroprogramm eventuell abzuändernden RT-Code des BRZ-Befehls. Es stehen Ihnen **ausschließlich** die nachfolgenden Condition Select Signale zur Verfügung:

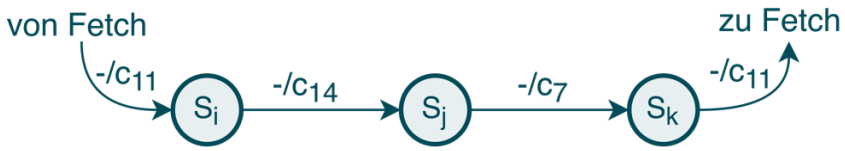
Condition Select	Funktion
00	Aktionsbefehl
01	Springe zu der dekodierten Opcode-Adresse (Mapping-ROM)
10	Springe, falls **Z = 1**
11	Springe unbedingt

Adresse					CS		Horizontale Kodierung Multiples/Mehrfaches Befehlsformat Kontrollsignale c[14:0] / Sprungadresse														RT-Code (NUR ÄNDERUNGEN ZUM URSRPÜNGLICHEN CODE EINTRAGEN)
4	3	2	1	0	1	0	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0																		INIT:
0	0	0	0	1																		FETCH:
0	0	0	1	0																		
0	0	0	1	1																		JMP:
0	0	1	0	0																		
0	0	1	0	1																		NEG:
0	0	1	1	0																		
0	0	1	1	1																		ADDX:
0	1	0	0	0																		
0	1	0	0	1																		ADDI:
0	1	0	1	0																		
0	1	0	1	1																		
0	1	1	0	0																		
0	1	1	0	1																		ADD:
0	1	1	1	0																		
0	1	1	1	1																		BRZ:
1	0	0	0	0																		
1	0	0	0	1																		
1	0	0	1	0																		

5) Geben Sie den Inhalt des Mapping ROM unter Berücksichtigung ALLER Befehle inkl. DLOAD und DSTORE an. Begründen Sie die Einsprungadressen von DLOAD und DSTORE. Sollten Sie DLOAD und DSTORE nicht realisiert haben, denken Sie sich bitte eine Zahl benötigter Takte je Befehl aus und geben diese mit an.

6) Geben Sie den Speicherbedarf Ihres Mikroprogramms unter der Annahme an, dass Sie alle Befehle inklusive Ihres DLOAD und DSTORE realisiert hätten. Sollten Sie DLOAD und DSTORE nicht realisiert haben, nehmen Sie bitte dieselbe Taktzahl wie in 5).

7) Betrachten Sie den gegebenen Automaten, der die Ausführungsphase eines Befehls der CPU dieser Aufgabe beschreibt. Was macht dieser Befehl? Geben Sie die Antwort in einem Satz!



8) Geben Sie beginnend ab Adresse $000 eine Folge von Assemblerdirektiven für die hier erstellte CPU an, um in einer Schleife die Werte eines ab Adresse $000 im Datenspeicher abgelegten Arrays der Länge 7 aufzusummieren, das Ergebnis im Datenspeicher ab Adresse $00E abzulegen und anschließend in einer Endlosschleife zu verharren. Ihnen stehen dabei die folgenden im Datenspeicher abgelegten Hilfsvariablen zur Verfügung. Bis auf die Werte des Arrays dürfen sämtliche Speicherinhalte verändert werden. Sie dürfen alle in dieser Aufgabe realisierten Befehle verwenden.

Adresse	Wert	Erklärung
$000 … $006	…	Werte des Arrays
…		
$00A	0	
$00B	$07	Länge des Arrays / Zähler
$00C	0	
$00D	$00	Anfangsadresse des Arrays / Aktueller Index
$00E	0	
$00F	0	
…		

Inhalt des Datenspeicher

Befehl		Beschreibung
`JMP`	`ADDR`	`PC = ADDR`
`ADDX`	`IMM`	`AC = AC + IMM`
`ADDI`	`ADDR`	`AC = AC + MEM[MEM[ADDR]]`
`NEG`		`AC = -AC`
`BRZ`	`ADDR`	`if Z=1 then PC = ADDR`
`DLOAD`	`ADDR`	`AC(H) = MEM[ADDR], AC(L) = MEM[ADDR+1]`
`DSTORE`	`ADDR`	`MEM[ADDR] = AC(H), MEM[ADDR+1] = AC(L)`

Übersicht: Befehle der CPU

Aufgabe 3: RISC-V und EduCore-V Tiny (21 Punkte)

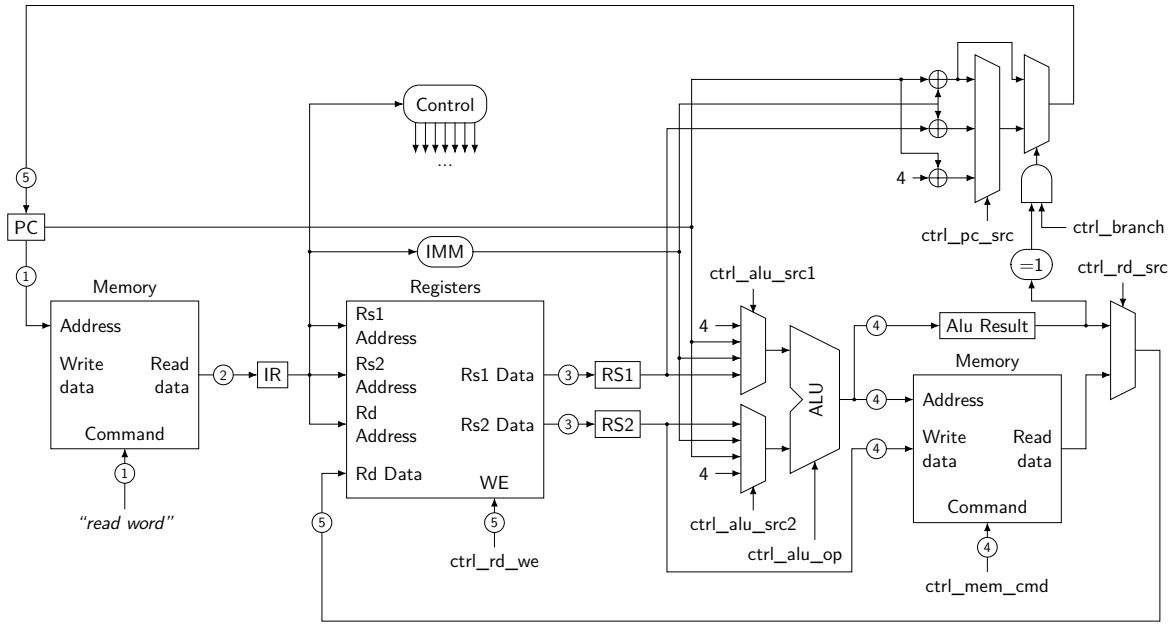


Abbildung 1: Schematische Darstellung des EduCore-V Tiny

31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
imm	rs1	func3	rd	opcode
0 0 0 0 0 0 0 0 0 0 0 0	- - - - -	1 1 1	- - - - -	1 1 0 0 1 1 1

Abbildung 2: Befehlsformat des `rand` 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,` **`RAND`**
`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 3: Kontrollsignale ergänzt um `rand`

In dieser Aufgabe sollen Sie den RV32I Basisbefehlssatz des in Abbildung 1 schematisch dargestellten EduCore-V Tiny um eine Pseudozufallsfunktion `rand` erweitert werden. Der Befehl `rand` nimmt einen Parameter entgegen und nutzt diesen als Seed. Die nächste Zufallszahl wird dann wie folgt berechnet:

$$R = (X \text{ ^ } (X \ll 13)) \text{ ^ } ((X \text{ ^ } (X \ll 13)) \gg 17)$$

wobei X das Eingaberegister ist und R die Ausgabe ist.

Gemäß Abbildung 2 soll der Befehl `rand` zu den Register-Immediate Befehlen gehören. Er bekommt den Seed der Pseudozufallsfunktion im Source-Register (`rs1`). Der Immediate-Bereich des Befehls wird ignoriert und ist immer auf 0 gesetzt. Die neue Zufallszahl wird ins Zielregister `rd` geschrieben. Der Befehl soll den `opcode` 1100111 erhalten. Ferner wird der `func3` Teil auf 111 gesetzt.

1) Kann ein anderer Opcode für die Definition der `rand` Instruktion gewählt werden, ohne den Immediate Decoder zu verändern? Wenn ja, welcher/welche können gewählt werden? Sie können dafür Tabelle auf Seite 16 zurate ziehen.

2) Realisieren Sie die Berechnung der Pseudozufallszahl unter der ausschließlichen Verwendung des RV32I Basisbefehlssatzes sowie der Pseudoinstruktionen als RISC-V Assembler Programm. Laden Sie in das Register `x21` den Seed-Wert `0x17107612` und führen Sie die Berechnung gemäß der oben genannten Formel durch, wobei das Ergebnis nach der Berechnung in Register `x22` abgelegt werden soll.

3) Geben Sie die vom Kontrollsignalgenerator für die Realisierung des `rand` Befehls zu erzeugenden Kontrollsignale an, indem Sie **sämtliche** Signalwerte in der nachfolgenden Tabelle ergänzen. Orientieren Sie sich dafür an Abbildung 1 und an die in Abbildung 3 angegebenen Werte für die Kontrollsignale.

Kontrollsignal	Wert
`ctrl_alu_src1`	A)
`ctrl_alu_src2`	B)
`ctrl_mem_cmd`	C)
`ctrl_alu_op`	D)
`ctrl_rd_src`	E)
`ctrl_rd_we`	F)
`ctrl_pc_src`	G)
`ctrl_branch`	H)

4) Ergänzen Sie nun den nachfolgenden Codeausschnitt der ALU um den `rand` Befehl. Nutzen Sie für die case-Abfrage die Konstante `CTRL_ALU_OP_RAND`. Für die Beschreibung der Operation brauchen Sie keine Konvertierung der Operanden angeben. Nutzen Sie außerdem folgende Operatoren `"A xor B"`, `"A sll num"` und `"A slr num"`. Sie brauchen den gegebene Quellcode nicht abzuschreiben.

```
architecture rtl of m_alu is
begin

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









-- LOESUNG END -
            when others => result <= 32x"0";
        end case;
    end process;

end architecture rtl;
```

5) Realisieren Sie die Berechnung der Pseudozufallszahl unter Verwendung des von Ihnen erweiterten RV32I Basisbefehlssatzes der Pseudoinstruktionen als RISC-V Assembler Programm. Laden Sie in das Register `x21` den Seed-Wert `0x17107612` und führen Sie die Berechnung gemäß der oben genannten Formel, wobei das Ergebnis nach der Berechnung in Register `x22` abgelegt werden soll. Nutzen Sie dafür explizit den von Ihnen realisierten Befehl `rand`. Bedenken Sie, dass der Compiler Ihren Befehl nicht kennt.

6) Analysieren Sie den nachfolgenden Speicherinhalt und geben Sie an, welche Befehle hier implementiert sind. Geben Sie außerdem an, welchen Wert die verwendeten Register nach Ausführung der Befehle haben, wenn sie mit Null initialisiert wurden.

Adr.	00	01	02	03	04	05	06	07
Data	93	00	F0	FF	33	B1	00	00

Format	7	5	5	3	5	7	Instruction
U	imm*				rd	0110111	lui rd, I
U	imm*				rd	0010111	auipc rd, I
J	imm*				rd	1101111	jal rd, I
I	imm*		rs1	000	rd	1100111	jalr rd, rs1, I
B	imm*	rs2	rs1	000	imm*	1100011	beq rs1, rs2, I
B	imm*	rs2	rs1	001	imm*	1100011	bne rs1, rs2, I
B	imm*	rs2	rs1	100	imm*	1100011	blt rs1, rs2, I
B	imm*	rs2	rs1	101	imm*	1100011	bge rs1, rs2, I
B	imm*	rs2	rs1	110	imm*	1100011	bltu rs1, rs2, I
B	imm*	rs2	rs1	111	imm*	1100011	bgeu rs1, rs2, I
I	imm*		rs1	000	rd	0000011	lb rd, I(rs1)
I	imm*		rs1	001	rd	0000011	lh rd, I(rs1)
I	imm*		rs1	010	rd	0000011	lw rd, I(rs1)
I	imm*		rs1	100	rd	0000011	lbu rd, I(rs1)
I	imm*		rs1	101	rd	0000011	lhu rd, I(rs1)
S	imm*	rs2	rs1	000	imm*	0100011	sb rs2, I(rs1)
S	imm*	rs2	rs1	001	imm*	0100011	sh rs2, I(rs1)
S	imm*	rs2	rs1	010	imm*	0100011	sw rs2, I(rs1)
I	imm*		rs1	000	rd	0010011	addi rd, rs1, I
I	0000000	imm*	rs1	001	rd	0010011	slli rd, rs1, I
I	imm*		rs1	010	rd	0010011	slti rd, rs1, I
I	imm*		rs1	011	rd	0010011	sltiu rd, rs1, I
I	imm*		rs1	100	rd	0010011	xori rd, rs1, I
I	0000000	imm*	rs1	101	rd	0010011	srli rd, rs1, I
I	0100000	imm*	rs1	101	rd	0010011	srai rd, rs1, I
I	imm*		rs1	110	rd	0010011	ori rd, rs1, I
I	imm*		rs1	111	rd	0010011	andi rd, rs1, I
R	0000000	rs2	rs1	000	rd	0110011	add rd, rs1, rs2
R	0100000	rs2	rs1	000	rd	0110011	sub rd, rs1, rs2
R	0000000	rs2	rs1	001	rd	0110011	sll rd, rs1, rs2
R	0000000	rs2	rs1	010	rd	0110011	slt rd, rs1, rs2
R	0000000	rs2	rs1	011	rd	0110011	sltu rd, rs1, rs2
R	0000000	rs2	rs1	100	rd	0110011	xor rd, rs1, rs2
R	0000000	rs2	rs1	101	rd	0110011	srl rd, rs1, rs2
R	0100000	rs2	rs1	101	rd	0110011	sra rd, rs1, rs2
R	0000000	rs2	rs1	110	rd	0110011	or rd, rs1, rs2
R	0000000	rs2	rs1	111	rd	0110011	and rd, rs1, rs2

Aufgabe 4: Allgemeine Fragen (5 Punkte)

Hinweis: Eine Punktevergabe in dieser Aufgabe erfolgt nur bei angegebener Begründung.

1) Sind taktzustandsgesteuerte JK-Flipflops im Allgemeinen langsamer als taktzustandsgesteuerte RS-Flipflops. Begründen Sie Ihre Antwort.

2) Ist eine Realisierung mit RISC-V Assembler unter Verwendung von Pseudoinstruktionen im Allgemeinen schneller als eine äquivalente Realisierung mit Hilfe des Basisbefehlssatzes? Begründen Sie Ihre Antwort.

3) Findet man mit dem QMC-Verfahren im Allgemeinen immer genau nur eine Minimalform? Begründen Sie Ihre Antwort.

4) Kann man anhand der Schaltung eines Automaten immer direkt erkennen, ob es sich um einen Moore- oder Mealy-Automat handelt? Begründen Sie Ihre Antwort.

5) Benötig die Realisierung eines Schaltwerkes in Form einer Verzögerungskette immer mehr Flipflops als die Realisierung in Form eines Automaten, bei dem die Zustände fortlaufend binärkodiert sind und die einzelnen Zustandsbits mit Hilfe von Flipflops repräsentiert werden. Begründen Sie Ihre Antwort.