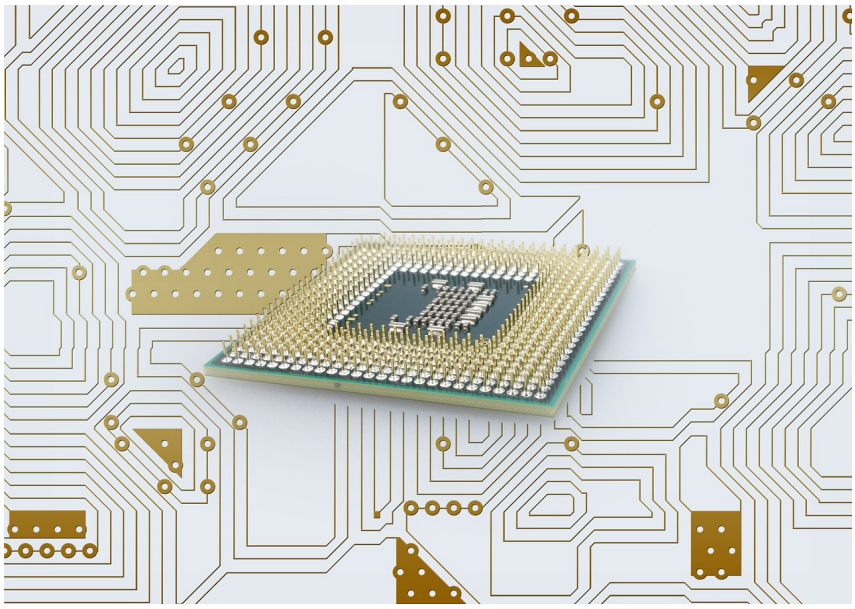
THM - University of Applied Sciences
Faculty IEM



HC1 Central Processing Unit

VHDL Design

Author: Matthias Röll
matthias.roell@iem.thm.de

Date: December 2014

Inhaltsverzeichnis

I Abbildungsverzeichnis

II Tabellenverzeichnis

III Abkürzungsverzeichnis

1 HC1 Architecture & Overview

Die HC1 CPU ist ein simpler 8 Bit Rechner. Er ist fähig 10 verschiedene Operationen durchzuführen und stellt einen 32 Byte Speicher zur Verfügung.

1.1 Block & RTL Diagram

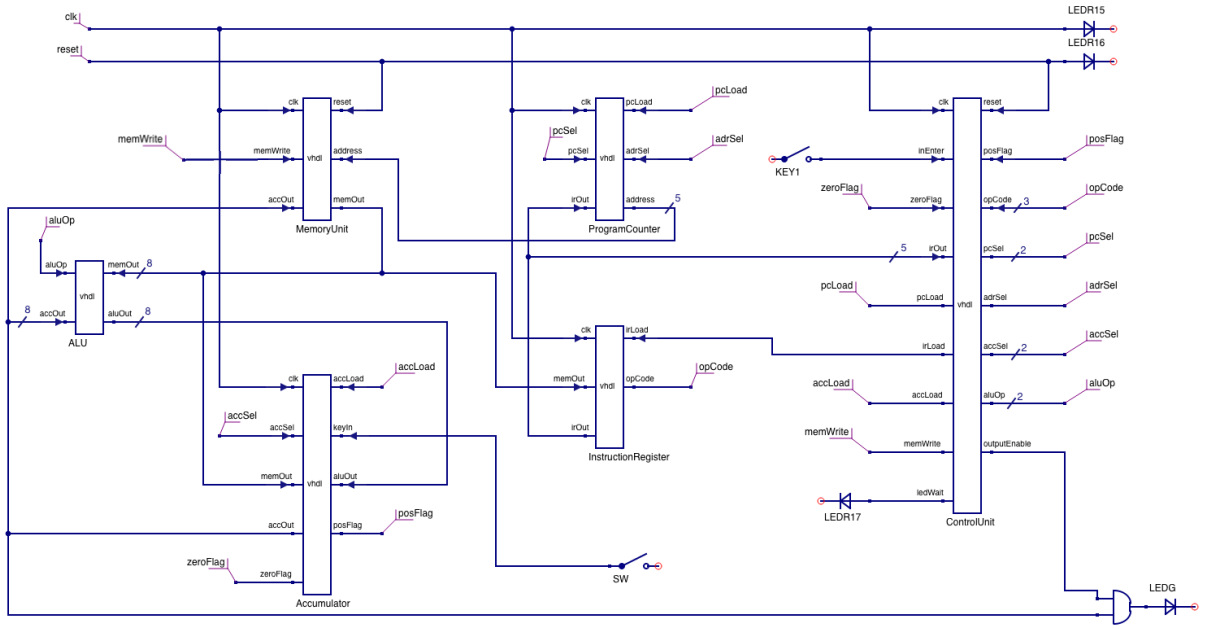


Abbildung 1: HC1 Architektur Diagramm

Das Diagramm zeigt die Bausteine des HC1 Rechners. Diese werden durch eine CPU Entität über Signale verknüpft.

Die CPU besteht aus den folgenden Bausteinen:

- Accumulator
- Arithmetic Logical Unit (ALU)
- Control Unit
- Instruction Register
- Memory Unit
- Program Counter

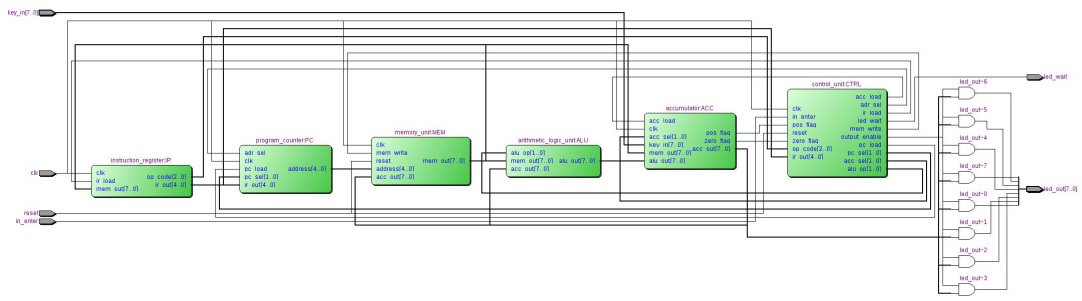


Abbildung 2: RTL Diagramm

1.2 Instruction Set

Der Befehlssatz des Rechner besteht aus 10 verschiedenen Instruktionen.

Jeder Befehl enthält hierbei 3 Op-Code Bits und 5 Adress Bits.

Befehl	OpCode	Data	Taktzyklen
LOAD	000	aaaaa	5
STORE	001	aaaaa	5
ADD	010	aaaaa	5
SUB	011	aaaaa	5
NAND	100	eeeee	5
IN	100	00000	5
OUT	100	00001	5
JZ	101	aaaaa	4
JPOS	110	aaaaa	4
J	111	aaaaa	4

Tabelle 1: Instruction Set

Taktzyklen:

Die dargestellte Takzyklenanzahl der Befehle kann zwischen verschiedenen Implementationen abweichen.

Daten:

Die angegebene Datenkodierung stellt folgende Werte dar:

1. `aaaaa` - Adresse im Bereich 0 - 32
2. `eeeee` - Adresse im Bereich 2 - 32

1.2.1 Befehle

Folgend eine kurze Finktionsbeschreibung der Befehle:

LOAD Lädt einen Wert aus dem Hauptspeicher in den Akkumulator.

STORE Speichert einen Wert aus dem Akkumulator in den Hauptspeicher.

ADD Addiert einen Wert aus dem Hauptspeicher zu dem Wert im Akkumulator.

SUB Subtrahiert einen Wert aus dem Hauptspeicher von dem Wert im Akkumulator.

NAND Führt eine NAND Operation mit dem Wert im Hauptspeicher und dem Akkumulatorwert durch.

IN Speichert eine Eingabe im Akkumulator.

OUT Setzt den aktuellen Akkumulatorwert auf den Ausgangsbus.

JZ Führt eine Sprunganweisung durch, wenn der Akkumulatorinhalt gleich 0 ist (zeroFlag).

JPOS Führt eine Sprunganweisung durch, wenn der Akkumulatorinhalt größer 0 ist (posFlag).

J Führt eine Sprunganweisung durch..

1.3 Default Program & Simulation

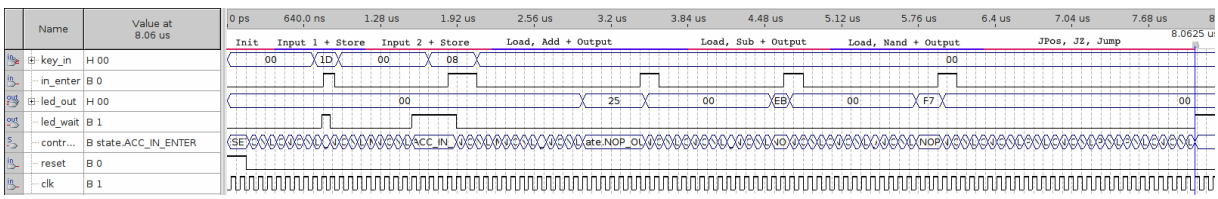


Abbildung 3: Program Simulation

1.4 Modifikationen

Hier werden die Abweichungen zur originalen HC1 Vorlage beschrieben.

ledWait Das im Blockdiagram 1 zu sehende Signal *ledWait* wurde von mir hinzugefügt und ist nicht Bestandteil des ursprünglichen HC1 Rechners. Es dient zur Darstellung einer Eingabeaufforderung.

2 Entities

2.1 Accumulator

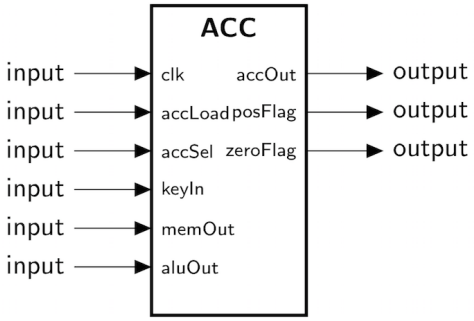


Abbildung 4: Accumulator - Block

2.1.1 Ports

Name	Type	Bitlength	Direction
clk	`std_logic`	1	Input
accLoad	`std_logic`	1	Input
accSel	`std_logic_vector`	2	Input
keyIn	`std_logic_vector`	8	Input
memOut	`std_logic_vector`	8	Input
aluOut	`std_logic_vector`	8	Input
accOut	`std_logic_vector`	8	Output
posFlag	`std_logic`	1	Output
zeroFlag	`std_logic`	1	Output

Tabelle 2: Accumulator Ports

2.2 Arithmetic Logical Unit

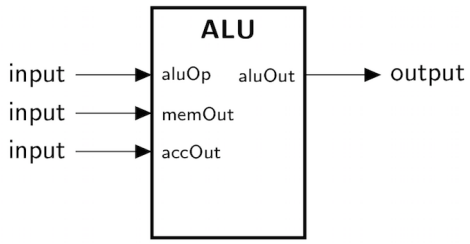


Abbildung 5: Arithmetic Logical Unit - Block

Dies ist der einzige Baustein des HC1 ohne Taktsignal.

2.2.1 Ports

Name	Type	Bitlength	Direction
aluOp	`std_logic_vector`	2	Input
memOut	`std_logic_vector`	8	Input
accOut	`std_logic_vector`	8	Input
aluOut	`std_logic_vector`	8	Output

Tabelle 3: Arithmetic Logical Unit Ports

2.3 Control Unit

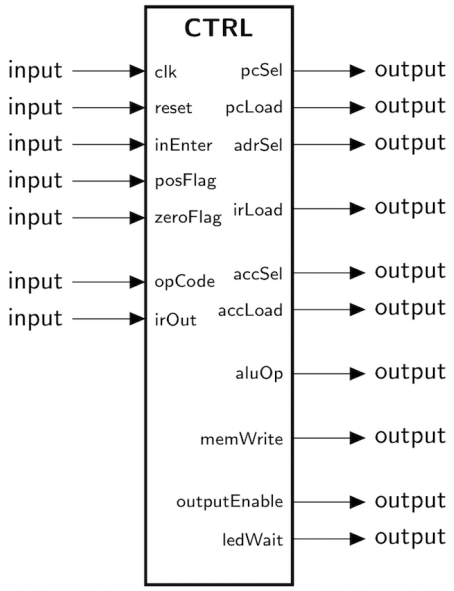


Abbildung 6: Control Unit - Block

Die Control Unit steuert alle Prozesse des Rechners. Sie besteht aus einer State Machine welche die Eingangssignale auswertet und darauf basierend die Ausgänge setzt.

2.3.1 Ports

Name	Type	Bitlength	Direction
clk	`std_logic`	1	Input
reset	`std_logic`	1	Input
inEnter	`std_logic`	1	Input
posFlag	`std_logic`	1	Input
zeroFlag	`std_logic`	1	Input
opCode	`std_logic_vector`	3	Input
irOut	`std_logic_vector`	5	Input

Tabelle 4: Control Unit Input Ports

Wie am Ende des Architekturkapitels 1.4 erklärt ist das *ledWait* Signal nicht Bestandteil des ursprünglichen HC1 Rechners.

2.3.2 State Machine

Im folgenden ein Beispiel wie die State Machine der Control Unit aussehen kann:

Name	Type	Bitlength	Direction
pcSel	`std_logic_vector`	1	Output
pcLoad	`std_logic`	1	Output
adrSel	`std_logic`	1	Output
irLoad	`std_logic`	1	Output
accSel	`std_logic_vector`	2	Output
accLoad	`std_logic`	1	Output
aluOp	`std_logic_vector`	2	Output
memWrite	`std_logic`	1	Output
outputEnable	`std_logic`	1	Output
ledWait (optional)	`std_logic`	1	Output

Tabelle 5: Control Unit Output Ports

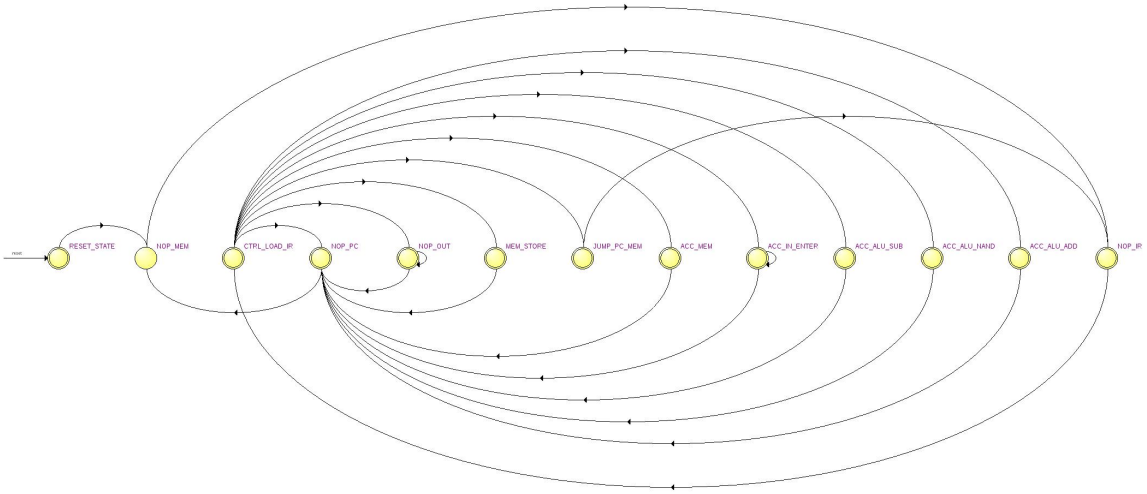


Abbildung 7: Control Unit - State Machine

Da die Abbildung etwas komplex wirkt ist hier nochmals der VHDL Code zum Erstellen eines solchen State Machine Typen abgebildet.

```
1  type state_type is (
2          RESET_STATE,                    -- Reset CPU
3
4          CTRL_LOAD_IR,                   -- New Instruction from IR
5
6          MEM_STORE,                      -- Write Memory
7          ACC_MEM,                        -- Acc load Memory
8
9          ACC_ALU_ADD,                    -- Acc load ALU with ALU-Add Operation
10         ACC_ALU_SUB,                    -- Acc load ALU with ALU-Sub Operation
11         ACC_ALU_NAND,                   -- Acc load ALU with ALU-Nand Operation
12
```

```
13          ACC_inEnter,                    -- Acc load key_in when inEnter is set
14
15          JUMP_PC_MEM,                    -- PC Jump to Address in Memory
16
17          NOP_PC,                         -- Update PC
18          NOP_OUT,                        -- Enable Output while not inEnter
19          NOP_MEM,                        -- Update MEM
20          NOP_IR                          -- Update IR
21  );
```

2.3.3 Implementationshinweis

Es gibt verschiedene Wege das Control Unit Verhalten in VHDL zu implementieren. Das Hauptproblem hierbei ist es einen Weg zu finden die Control Unit taktgesteuert zu impementieren ohne *inferred latches* zu generieren.

Ein Weg dies zu tun ist folgender. Zuerst werden drei verschiedene Prozesse definiert:

1. Eingangssignale State Prozess - **Asynchroner Prozess**
2. Takt Prozess - **Taktsynchroner Prozess**
3. State Ausgangssignal Prozess - **Taktsynchroner Prozess**

Eingangssignale State Prozess Dieser Prozess ist sensitiv gegenüber allen Eingangssignalen mit Ausnahme der *clk* und des *reset* Signals.

Der Prozess überprüft beim Aufrufen den aktuellen State und speichert basierend auf den Eingangssignalen den nächsten State in einem internen Signal.

Um inferred latches zu vermeiden muss der interne nächste State am Anfang des Prozesses auf den aktuellen State gesetzt werden.

Takt Prozess Dieser Prozess aktualisiert den aktuellen State. Hierbei wird der aktuelle State durch den intern gespeicherten State überschrieben.

Zudem wird hier auch das *reset* Signal überprüft.

Die Sensitivitylist enthält das *clk* und das *reset* Signal.

State Ausgangssignal Prozess Hier werden alle Ausgangsignale gesetzt. Das einzige Signal in der Sensitivitylist des Prozesses ist das aktuelle State Signal.

Wichtig ist zu verstehen, dass dieser Prozess nur nach dem Takt Prozess aufgerufen wird, da nur dort das State Signal gesetzt wird.

2.4 Instruction Register

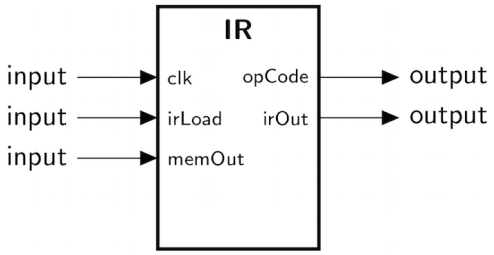


Abbildung 8: Instruction Register - Block

Das Instruction Register splittet den aktuellen Speicherwert (memOut) in einen Befehlscode (opCode) und eine Adresse (irOut).

Das Instruction Register aktualisiert diese Werte nur bei steigender Taktflanke und aktivem irLoad Signal.

2.4.1 Ports

Name	**Type**	**Bitlength**	**Direction**
clk	`std_logic`	1	Input
irLoad	`std_logic`	1	Input
memOut	`std_logic_vector`	8	Input
opCode	`std_logic_vector`	3	Output
irOut	`std_logic_vector`	5	Output

Tabelle 6: Instruction Register Ports

2.5 Memory Unit

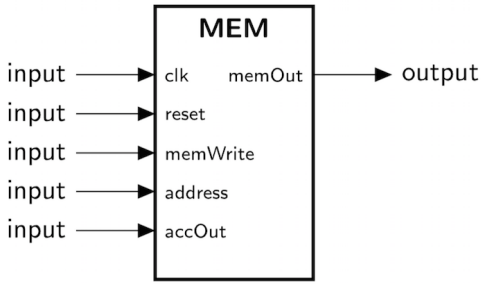


Abbildung 9: Memory Unit - Block

Die Memory Unit stellt den Hauptspeicher des Rechners. Dieser ist 32 Byte groß. Der Speicher wird sowohl für Programmcode als auch für gespeicherte Daten verwendet.

Funktionsweise:

Das address Eingangssignal setzt die aktuell ausgewählte Speicherstelle. Der dort gespeicherte Wert liegt danach am Ausgangsbus memOut an. Falls der Eingang memWrite gesetzt ist wird der Akkumulatorwert accOut in die gewählte Speicheradresse geschrieben.

Alle Daten werden erst bei einer steigenden Taktflanke.

Reset:

Über das reset Signal kann der Hauptspeicher zurückgesetzt werden. Hierbei werden normalerweise alle Speicherstellen auf 0 gesetzt.

Zum Testen des Rechners kann hier jedoch auch ein Programm in den Speicher geladen werden.

2.5.1 Ports

Name	Type	Bitlength	Direction
clk	`std_logic`	1	Input

Tabelle 7: Memory Unit Ports

2.6 Program Counter

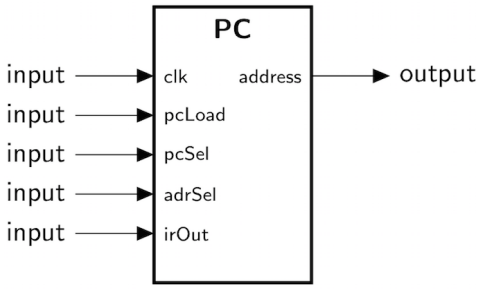


Abbildung 10: Program Counter - Block

Der Program Counter hält die aktuelle Speicheradressennummer. Des weiteren erhöht er diese nach bearbeiten jedes Befehls. Er kann zudem für Sprunganweisungen und das Laden von Daten direkt auf einen Wert gesetzt werden.

2.6.1 Ports

Name	Type	Bitlength	Direction
clk	`std_logic`	1	Input

Tabelle 8: Program Counter Ports

2.7 Central Processing Unit

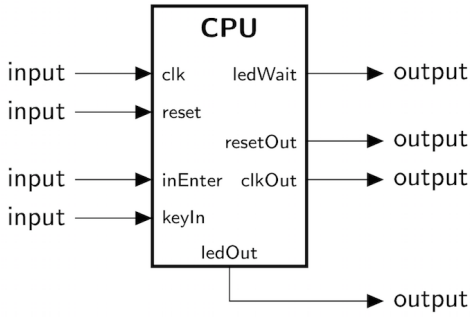


Abbildung 11: Central Processing Unit - Block

Diese VHDL Entität dient zur Verbindung der anderen Bausteine mittels dem sogenannten *mapping*.

2.7.1 Ports

Name	Type	Bitlength	Direction
clk	`std_logic`	1	Input

Tabelle 9: Central Processing Unit Ports

2.7.2 Mapping

2.8 Cyclone II Pin Mapping - HC1

Um nun die HC1 CPU mit dem Quartus Board zu benutzen, müssen die Pins des Boards mit den Ein- und Ausgängen der CPU Entität verbunden werden. Dies geschieht mittels einer Top Level Entität.

A Appendix

A.1 Accumulator - VHDL

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity accumulator is
5      port(
6          -- Control Input --
7          clk         : in  std_logic;
8          accLoad     : in  std_logic;
9          accSel      : in  std_logic_vector(1 downto 0);
10         -- Data Input --
11         keyIn       : in  std_logic_vector(7 downto 0);
12         memOut      : in  std_logic_vector(7 downto 0);
13         aluOut      : in  std_logic_vector(7 downto 0);
14         -- Data Output  --
15         accOut      : out std_logic_vector(7 downto 0);
16         posFlag     : out std_logic;
17         zeroFlag    : out std_logic
18     );
19 end accumulator;
20
21 architecture rtl of accumulator is
22
23     -- Register to hold the output
24     signal holdAccOut : std_logic_vector(7 downto 0);
25
26 begin
27     CLK_PROCESS : process(clk)
28     begin
29         if rising_edge(clk) then
30             -- Load new input
31             if accLoad = '1' then
32                 if accSel = "00" then          -- Load ALU
33                     holdAccOut <= aluOut;
34                 elsif accSel = "01" then       -- Load Memory
35                     holdAccOut <= memOut;
36                 elsif accSel = "10" then       -- Load Keyinput
37                     holdAccOut <= keyIn;
38                 else                        -- Not used Value
39                     holdAccOut <= "00000000";
40                 end if;
41             end if;
42
```

```
43              -- Set Zero Flag
44              if holdAccOut = "00000000" then
45                  zeroFlag <= '1';
46              else
47                  zeroFlag <= '0';
48              end if;
49              -- Set Positive Flag
50              if holdAccOut(7) = '0' then
51                  posFlag <= '1';
52              else
53                  posFlag <= '0';
54              end if;
55
56              -- Set Output
57              accOut <= holdAccOut;
58          end if;
59      end process;
60
61  end;
```

A.2 Arithmetic Logical Unit - VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity arithmetic_logic_unit is
    port(
        -- Control Input --
        aluOp      : in  std_logic_vector(1 downto 0);
        -- Data Input --
        memOut     : in  std_logic_vector(7 downto 0);
        accOut     : in  std_logic_vector(7 downto 0);
        -- Data Output  --
        aluOut     : out std_logic_vector(7 downto 0)
    );
end arithmetic_logic_unit;

architecture rtl of arithmetic_logic_unit is
begin

    -- Set output based on aluOp
    with aluOp select aluOut <=
        std_logic_vector(signed(memOut) + signed(accOut)) when "00",     -- ADD
        std_logic_vector(signed(memOut) - signed(accOut)) when "01",     -- SUB
        not (memOut and accOut) when "10",                               -- NAND
            "00000000" when others;                                      -- Not used operation

end;
```

A.3 Control Unit - VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity control_unit is
    port(
        -- Control Input --
        clk          : in  std_logic;
        reset        : in  std_logic;
        inEnter      : in  std_logic;
        posFlag      : in  std_logic;
        zeroFlag     : in  std_logic;

        -- Data Input --
        opCode       : in  std_logic_vector(2 downto 0);
        irOut        : in  std_logic_vector(4 downto 0);

        -- Program Counter Output  --
        pcSel        : out std_logic_vector(1 downto 0);
        pcLoad       : out std_logic;
        adrSel       : out std_logic;

        -- Instruction Register Output  --
        irLoad       : out std_logic;

        -- Accumulator Output  --
        accSel       : out std_logic_vector(1 downto 0);
        accLoad      : out std_logic;

        -- Arithmetic Logic Unit Output  --
        aluOp        : out std_logic_vector(1 downto 0);

        -- Memory Output  --
        memWrite     : out std_logic;

        -- Generic Output  --
        outputEnable : out std_logic;
        ledWait      : out std_logic
    );
end control_unit;

-- ------------------------------------------- --
-- Control Flow                                --
-- ------------------------------------------- --
```

```
44 --                                                    --
45 --  1. state      <- nextState on clock               --
46 --  2. output     <- based on state                   --
47 --  3.               repeat 1 & 2                     --
48 --                                                    --
49 --  x. nextState  <- when input gets changed          --
50 -- ------------------------------------------------ --
51
52 architecture rtl of control_unit is
53     type state_type is (
54         RESET_STATE,                    -- Reset CPU
55
56         CTRL_LOAD_IR,                   -- New Instruction from IR
57
58         MEM_STORE,                      -- Write Memory
59         ACC_MEM,                        -- Acc load Memory
60
61         ACC_ALU_ADD,                    -- Acc load ALU with ALU-Add Operation
62         ACC_ALU_SUB,                    -- Acc load ALU with ALU-Sub Operation
63         ACC_ALU_NAND,                   -- Acc load ALU with ALU-Nand Operation
64
65         ACC_inEnter,                    -- Acc load key_in when inEnter is set
66
67         JUMP_PC_MEM,                    -- PC Jump to Address in Memory
68
69         NOP_PC,                         -- Update PC
70         NOP_OUT,                        -- Enable Output while not inEnter
71         NOP_MEM,                        -- Update MEM
72         NOP_IR                          -- Update IR
73 );
74     signal state         : state_type := RESET_STATE;
75     signal nextState     : state_type := RESET_STATE;
76
77 begin
78     CLOCK_PROCESS : process(clk, reset)
79     begin
80         if reset = '1' then
81             state <= RESET_STATE;
82         elsif rising_edge(clk) then
83             state <= nextState;
84         end if;
85     end process;
86
87     -- This process updates the next state based on the incoming signals
88     --
```

```
89      INPUT_STATE_PROCESS : process(state, inEnter, opCode, irOut, zeroFlag, posFlag)
90      begin
91          nextState <= state;      -- prevents inferred latches,
92                                   -- because nextState needs to be set even if
93                                   -- the following case is going to set it
94
95          case (state) is
96
97              -- On reset pc is set to address 0x00 so no nop_pc is needed
98              when RESET_STATE =>
99                  nextState <= NOP_MEM;
100
101             -- Next State depends on inputs
102             when CTRL_LOAD_IR =>
103                 -- Going through opCodes
104                 case (opCode) is
105                     when "000" =>       -- LOAD
106                         nextState <= ACC_MEM;
107                     when "001" =>
108                         nextState <= MEM_STORE;
109                     when "010" =>       -- ADD
110                         nextState <= ACC_ALU_ADD;
111                     when "011" =>       -- SUB
112                         nextState <= ACC_ALU_SUB;
113
114                     when "100" =>                       -- NAND, IN, OUT
115                         -- Depends on irOut data
116                         if irOut = "00000" then         -- IN
117                             nextState <= ACC_inEnter;
118                         elsif irOut = "00001" then      -- OUT
119                             nextState <= NOP_OUT;
120                         else                            -- NAND
121                             nextState <= ACC_ALU_NAND;
122                         end if;
123
124                     when "101" =>       -- JUMP ZERO
125                         if zeroFlag = '1' then
126                             nextState <= JUMP_PC_MEM;
127                         else
128                             nextState <= NOP_PC;
129                         end if;
130                     when "110" =>       -- JUMP POSITIVE
131                         if posFlag = '1' then
132                             nextState <= JUMP_PC_MEM;
133                         else
```

```
134                         nextState <= NOP_PC;
135                     end if;
136                 when others =>      -- JUMP ALWAYS
137                     nextState <= JUMP_PC_MEM;
138             end case;
139
140         when ACC_inEnter | NOP_OUT =>
141              if inEnter = '1' then
142                 nextState <= NOP_PC;
143              else
144                 nextState <= state;
145              end if;
146
147         when NOP_PC =>
148             nextState <= NOP_MEM;
149         -- JUMP_PC_MEM skips NOP_PC and NOP_MEM
150         when JUMP_PC_MEM | NOP_MEM =>
151             nextState <= NOP_IR;
152         when NOP_IR =>
153             nextState <= CTRL_LOAD_IR;
154
155         -- Next State is always NOP_PC
156         when others =>
157             nextState <= NOP_PC;
158
159     end case;
160 end process;
161
162 -- This process sets the output based on the current state
163 --
164 STATE_OUTPUT_PROCESS : process(state)
165 begin
166     case (state) is
167         when RESET_STATE =>         -- Reset CPU
168             -- PC
169             pcSel       <= "10";
170             pcLoad      <= '1';
171             adrSel      <= '0';
172             -- IR
173             irLoad      <= '0';
174             -- ACC
175             accSel      <= "00";
176             accLoad     <= '0';
177             -- ALU
178             aluOp       <= "00";
```

```
179                 -- MEM
180                 memWrite     <= '0';
181                 -- GEN
182                 outputEnable <= '0';
183                 ledWait      <= '0';
184
185             when CTRL_LOAD_IR =>         -- New Instruction from IR
186                 -- PC
187                 pcSel        <= "00";
188                 pcLoad       <= '0';
189                 adrSel       <= '1';
190                 -- IR
191                 irLoad       <= '0';
192                 -- ACC
193                 accSel       <= "00";
194                 accLoad      <= '0';
195                 -- ALU
196                 aluOp        <= "00";
197                 -- MEM
198                 memWrite     <= '0';
199                 -- GEN
200                 outputEnable <= '0';
201                 ledWait      <= '0';
202
203             when MEM_STORE =>            -- Write Memory In
204                 -- PC
205                 pcSel        <= "00";
206                 pcLoad       <= '0';
207                 adrSel       <= '1';
208                 -- IR
209                 irLoad       <= '0';
210                 -- ACC
211                 accSel       <= "ZZ";
212                 accLoad      <= '0';
213                 -- ALU
214                 aluOp        <= "00";
215                 -- MEM
216                 memWrite     <= '1';
217                 -- GEN
218                 outputEnable <= '0';
219                 ledWait      <= '0';
220
221             when ACC_MEM =>              -- Acc load Memory
222                 -- PC
223                 pcSel        <= "00";
```

```
224                 pcLoad       <= '0';
225                 adrSel       <= '0';
226                 -- IR
227                 irLoad       <= '0';
228                 -- ACC
229                 accSel       <= "01";
230                 accLoad      <= '1';
231                 -- ALU
232                 aluOp        <= "00";
233                 -- MEM
234                 memWrite     <= '0';
235                 -- GEN
236                 outputEnable <= '0';
237                 ledWait      <= '0';
238
239             when ACC_ALU_ADD =>          -- Acc load ALU with ALU-Add Operation
240                 -- PC
241                 pcSel        <= "00";
242                 pcLoad       <= '0';
243                 adrSel       <= '0';
244                 -- IR
245                 irLoad       <= '0';
246                 -- ACC
247                 accSel       <= "00";
248                 accLoad      <= '1';
249                 -- ALU
250                 aluOp        <= "00";
251                 -- MEM
252                 memWrite     <= '0';
253                 -- GEN
254                 outputEnable <= '0';
255                 ledWait      <= '0';
256
257             when ACC_ALU_SUB =>          -- Acc load ALU with ALU-Sub Operation
258                 -- PC
259                 pcSel        <= "00";
260                 pcLoad       <= '0';
261                 adrSel       <= '0';
262                 -- IR
263                 irLoad       <= '0';
264                 -- ACC
265                 accSel       <= "00";
266                 accLoad      <= '1';
267                 -- ALU
268                 aluOp        <= "01";
```

```
269                 -- MEM
270                 memWrite     <= '0';
271                 -- GEN
272                 outputEnable <= '0';
273                 ledWait      <= '0';
274
275             when ACC_ALU_NAND =>        -- Acc load ALU with ALU-Nand Operation
276                 -- PC
277                 pcSel        <= "00";
278                 pcLoad       <= '0';
279                 adrSel       <= '0';
280                 -- IR
281                 irLoad       <= '0';
282                 -- ACC
283                 accSel       <= "00";
284                 accLoad      <= '1';
285                 -- ALU
286                 aluOp        <= "10";
287                 -- MEM
288                 memWrite     <= '0';
289                 -- GEN
290                 outputEnable <= '0';
291                 ledWait      <= '0';
292
293             when ACC_inEnter =>         -- Acc load key_in when inEnter
294                 -- PC
295                 pcSel        <= "00";
296                 pcLoad       <= '0';
297                 adrSel       <= '0';
298                 -- IR
299                 irLoad       <= '0';
300                 -- ACC
301                 accSel       <= "10";
302                 accLoad      <= '1';
303                 -- ALU
304                 aluOp        <= "00";
305                 -- MEM
306                 memWrite     <= '0';
307                 -- GEN
308                 outputEnable <= '0';
309                 ledWait      <= '1';
310
311             when JUMP_PC_MEM =>         -- PC Jump to Address
312                 -- PC
313                 pcSel        <= "01";
```

```
314                 pcLoad       <= '1';
315                 adrSel       <= '1';
316                 -- IR
317                 irLoad       <= '0';
318                 -- ACC
319                 accSel       <= "00";
320                 accLoad      <= '0';
321                 -- ALU
322                 aluOp        <= "00";
323                 -- MEM
324                 memWrite     <= '0';
325                 -- GEN
326                 outputEnable <= '0';
327                 ledWait      <= '0';
328
329             when NOP_PC =>              -- Update PC
330                 -- PC
331                 pcSel        <= "00";
332                 pcLoad       <= '1';
333                 adrSel       <= '0';
334                 -- IR
335                 irLoad       <= '0';
336                 -- ACC
337                 accSel       <= "00";
338                 accLoad      <= '0';
339                 -- ALU
340                 aluOp        <= "00";
341                 -- MEM
342                 memWrite     <= '0';
343                 -- GEN
344                 outputEnable <= '0';
345                 ledWait      <= '0';
346
347             when NOP_OUT =>             -- Enable Output
348                 -- PC
349                 pcSel        <= "00";
350                 pcLoad       <= '0';
351                 adrSel       <= '0';
352                 -- IR
353                 irLoad       <= '0';
354                 -- ACC
355                 accSel       <= "00";
356                 accLoad      <= '0';
357                 -- ALU
358                 aluOp        <= "00";
```

```
359                 -- MEM
360                 memWrite     <= '0';
361                 -- GEN
362                 -- GEN
363                 outputEnable <= '1';
364                 ledWait      <= '1';
365 
366             when NOP_MEM =>             -- Update MEM
367                 -- PC
368                 pcSel        <= "00";
369                 pcLoad       <= '0';
370                 adrSel       <= '0';
371                 -- IR
372                 irLoad       <= '0';
373                 -- ACC
374                 accSel       <= "00";
375                 accLoad      <= '0';
376                 -- ALU
377                 aluOp        <= "00";
378                 -- MEM
379                 memWrite     <= '0';
380                 -- GEN
381                 outputEnable <= '0';
382                 ledWait      <= '0';
383 
384             when others =>              -- Update IR (NOP_IR)
385                 -- PC
386                 pcSel        <= "00";
387                 pcLoad       <= '0';
388                 adrSel       <= '0';
389                 -- IR
390                 irLoad       <= '1';
391                 -- ACC
392                 accSel       <= "00";
393                 accLoad      <= '0';
394                 -- ALU
395                 aluOp        <= "00";
396                 -- MEM
397                 memWrite     <= '0';
398                 -- GEN
399                 outputEnable <= '0';
400                 ledWait      <= '0';
401         end case;
402     end process;
403 
```

```
404 end;
```

A.4 Instruction Register - VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity instruction_register is
    port(
        clk    : in  std_logic;
        irLoad : in  std_logic;
        memOut : in  std_logic_vector(7 downto 0);
        opCode : out std_logic_vector(2 downto 0);
        irOut  : out std_logic_vector(4 downto 0)
    );
end instruction_register;

architecture rtl of instruction_register is
begin

    -- Simple memory signal splitter. On clock signal set op_code and ir_out
    -- (memory address) from memory output
    --
    CLK_PROCESS : process(clk)
    begin
        if rising_edge(clk) then
            if irLoad = '1' then
                opCode <= memOut(7 downto 5);
                irOut  <= memOut(4 downto 0);
            end if;
        end if;
    end process;

end;
```

A.5 Memory Unit - VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity memory_unit is
    port(
        -- Control Input --
        clk           : in  std_logic;
        reset         : in  std_logic;
        memWrite      : in  std_logic;
        -- Data Input --
        address       : in  std_logic_vector(4 downto 0);
        accOut        : in  std_logic_vector(7 downto 0);
        -- Data Output  --
        memOut        : out std_logic_vector(7 downto 0)
    );
end memory_unit;

architecture rtl of memory_unit is
    type ram_type is array (31 downto 0) of std_logic_vector(7 downto 0);

    signal ram          : ram_type;

    -- holds output address until next clk
    signal readAddress  : std_logic_vector(4 downto 0);

begin
    RAM_PROCESS : process(clk, reset)
    begin
        if reset = '1' then
            -- Default Code

            -- Some Values
            ram(31) <= "00000110";      -- Value: 0x06
            ram(30) <= "00000101";      -- Value: 0x05
            ram(29) <= "00000100";      -- Value: 0x04
            ram(28) <= "00000011";      -- Value: 0x03
            ram(27) <= "00000010";      -- Value: 0x02
            ram(26) <= "00000001";      -- Value: 0x01
            ram(25) <= "00000000";      -- Value: 0x00

            -- Default Program (Instruction Testing)
            --
```

```
44              --          Ins  |  Addr
45              ram(00) <= "000" & "11001"; -- ACC <- 0x00
46              ram(01) <= "100" & "00000"; -- ACC <- IN
47              ram(02) <= "001" & "11000"; -- MEM(24) <- ACC
48              ram(03) <= "100" & "00000"; -- ACC <- IN
49              ram(04) <= "001" & "10111"; -- MEM(23) <- ACC
50              -- ADD
51              ram(05) <= "010" & "11000"; -- ACC <- ACC + MEM(24)
52              ram(06) <= "100" & "00001"; -- OUT <- ACC
53              -- SUB
54              ram(07) <= "000" & "11000"; -- ACC <- MEM(24)
55              ram(08) <= "011" & "10111"; -- ACC <- ACC - MEM(23)
56              ram(09) <= "100" & "00001"; -- OUT <- ACC
57              -- NAND
58              ram(10) <= "000" & "11000"; -- ACC <- MEM(24)
59              ram(11) <= "100" & "10111"; -- ACC <- ACC nand MEM(23)
60              ram(12) <= "100" & "00001"; -- OUT <- ACC
61              -- JPos
62              ram(13) <= "000" & "11111"; -- ACC <- MEM(31)
63              ram(14) <= "110" & "10010"; -- PC <- 18
64              -- JZ
65              ram(18) <= "000" & "11001"; -- ACC <- MEM(25)
66              ram(19) <= "101" & "10000"; -- PC <- 16
67              -- Jump
68              ram(15) <= "100" & "00001"; -- OUT <- ACC
69              ram(16) <= "111" & "00000"; -- PC <- 00
70
71          elsif rising_edge(clk) then
72              -- Write data to memory
73              if memWrite = '1' then
74                  ram(to_integer(unsigned(address))) <= accOut;
75              end if;
76              -- Store output adress
77              readAddress <= address;
78          end if;
79      end process;
80
81      -- Set output (updates on RAM_PROCESS through readAddress signal)
82      memOut <= ram(to_integer(unsigned(readAddress)));
83
84  end;
```

A.6 Program Counter - VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity program_counter is
    port(
        -- Control Input --
        clk         : in  std_logic;
        pcLoad      : in  std_logic;
        pcSel       : in  std_logic_vector(1 downto 0);
        adrSel      : in  std_logic;
        -- Data Input --
        irOut       : in  std_logic_vector(4 downto 0);
        -- Data Output --
        address     : out std_logic_vector(4 downto 0)
    );
end program_counter;

architecture rtl of program_counter is

    -- internal pc address
    signal holdAddress : std_logic_vector(4 downto 0);

begin
    PC_PROCESS : process(clk)
    begin
        if rising_edge(clk) then
            if pcLoad = '1' then
                if pcSel = "00" then        -- increment counter
                    holdAddress <= std_logic_vector(unsigned(holdAddress) + 1);
                elsif pcSel = "01" then     -- set counter to new address from IR
                    holdAddress <= irOut;
                else                        -- reset counter
                    holdAddress <= "00000";
                end if;
            end if;
        end if;
    end process;

    -- Set output based on adrSel
    with adrSel select address <=
        holdAddress when '0',           -- PC address
        irOut when others;              -- IR address
```

```
44
45 end;
```

A.7 CPU - VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.all;

entity cpu is
    port(
        clk             : in  std_logic;                    -- Clock input
        reset           : in  std_logic;                    -- Reset input

        inEnter         : in  std_logic;                    -- Enter key
        keyIn           : in  std_logic_vector(7 downto 0); -- Value input

        ledWait         : out std_logic;                    -- Wait for enter
        ledOut          : out std_logic_vector(7 downto 0); -- Value output

        resetOut        : out std_logic;                    -- Reset indicator
        clkOut          : out std_logic                     -- Clock indicator
    );
end cpu;

architecture rtl of cpu is
    -- Data Wires
    signal address : std_logic_vector(4 downto 0);       -- PC to MEM
    signal memOut : std_logic_vector(7 downto 0);        -- MEM to IR, ALU & ACC
    signal accOut : std_logic_vector(7 downto 0);        -- ACC to MEM, ALU & OUT_LEDS
    signal opCode : std_logic_vector(2 downto 0);        -- IR to CTRL
    signal irOut  : std_logic_vector(4 downto 0);        -- IR to CTRL & PC
    signal aluOut : std_logic_vector(7 downto 0);        -- ALU to ACC

    -- Control Wires
    signal pcSel  : std_logic_vector(1 downto 0);        -- PC input address selection
    signal pcLoad : std_logic;                           -- PC enable
    signal adrSel : std_logic;                           -- PC output address selection

    signal irLoad : std_logic;                           -- IR enable

    signal accSel   : std_logic_vector(1 downto 0);      -- ACC input data selection
    signal accLoad  : std_logic;                         -- ACC enable
    signal posFlag  : std_logic;                         -- ACC data is positive flag
    signal zeroFlag : std_logic;                         -- ACC data is zero flag

    signal aluOp : std_logic_vector(1 downto 0);         -- ALU operation selection
    signal memWrite : std_logic;                         -- MEM write enable
```

```
44      signal outputEnable : std_logic;                    -- LED output enable
45
46  begin
47
48      -- The following commands connecting the signals, inputs & outputs between
49      -- all single components of the cpu
50      --
51      CTRL : control_unit port map(clk, reset, inEnter, posFlag, zeroFlag, opCode,
52                                   irOut, pcSel, pcLoad, adrSel, irLoad, accSel,
53                                   accLoad, aluOp, memWrite, outputEnable, ledWait);
54      MEM  : memory_unit port map(clk, reset, memWrite, address, accOut, memOut);
55      ALU  : arithmetic_logic_unit port map(aluOp, memOut, accOut, aluOut);
56      ACC  : accumulator port map(clk, accLoad, accSel, keyIn, memOut, aluOut,
57                                  accOut, posFlag, zeroFlag);
58      IR   : instruction_register port map(clk, irLoad, memOut, opCode, irOut);
59      PC   : program_counter port map(clk, pcLoad, pcSel, adrSel, irOut, address);
60
61      -- Setting the value output if output is enabled
62      --
63      ledOut(0) <= accOut(0) and outputEnable;
64      ledOut(1) <= accOut(1) and outputEnable;
65      ledOut(2) <= accOut(2) and outputEnable;
66      ledOut(3) <= accOut(3) and outputEnable;
67
68      ledOut(4) <= accOut(4) and outputEnable;
69      ledOut(5) <= accOut(5) and outputEnable;
70      ledOut(6) <= accOut(6) and outputEnable;
71      ledOut(7) <= accOut(7) and outputEnable;
72
73      -- Setting the clock and reset indicators (added for debugging)
74      --
75      resetOut <= reset;
76      clkOut <= clk;
77
78  end;
```