LAB 4 BASIC COMPUTER ORGANIZATION

1. Objectives

In this laboratory, students will analyse the structure of a *basic computer*, will devise, design, implement, simulate in Quartus and, if will be physically present in the lab, will test experimentally its *control unit* on Altera platform; furthermore, students will use *opcodes* to write simple programs in *machine code*. The design must function in simulation, and also on the DE2-115 Altera development board, if physically present in the lab.

2. Equipment and Supplies:

* Quartus II (student edition or web edition)
* Altera DE2-115 board with
  - USB-blaster cable
  - Power supply 12 VDC, 2A

3. References

- You are provided with all the .bdf files, except one that describes the combinational circuits which generates the *output* and *transition functions* of the *Control Unit* and which you are expected to conceive and develop. Their logic diagrams are annexed to this document.
- Chapters 5 and 6 of the textbook: *Computer Systems Structures*, Morris Mano, 3rd edition, 1993, ISBN 0-13-175563-3.
- The course notes
- The user guide of the Altera DE2-115 development kit is provided in the *Laboratories* > *Documentation* section of your CEG2136 Virtual Campus.

4. The Structure of the Basic Computer

4.1 General View

This laboratory implements a computer having a structure that is very close to the one presented in figure 5.4 on page 130 of your textbook. However, there are two major differences:

1. The designed computer's memory (storing both programs and data) has a capacity of 256 words of 8 bits (256 x 8). In the textbook, the BASIC computer has a memory with words of 16 bits, each word being capable of storing one memory-reference instruction (which consists there of a 4-bit *opcode* and a 12-bit *memory address*). In this lab, a *memory-reference instruction* is 2 byte long as well, but the msb byte carries the *opcode*, while the lsb byte contains the *operand address* (8 bits are enough to address a memory space of $2^8 = 256$ memory locations); as such, a 2-byte *memory-reference instruction* is stored in 2 consecutive memory locations (two 1-byte words).
   Consequently, two successive READ cycles are needed to fetch a *memory-reference* instruction: first to get the *opcode*, and the second to get the *address of the data* that the opcode will use.
2. The second major difference consists in the additional circuits which will allow a user to *visualize* the contents of the memory independently of having a program running or not on the DE2-115 board.
   User can preset the DIP switches on the board with the memory address to be visualized. Before the fetch phase of each instruction, the BASIC computer reads the contents of the memory location pointed at by the DIP switches on the board and shows it in hexadecimal format on the 7 segment display of the board.

The block diagram of your computer is presented in Figure 1. The *.bdf* files of all the component blocks, except the *Instruction Decoder* (*lab_controller* of CU), will be provided. The *Control Unit* functions in accord with a time sequence which is generated by the *sequence counter* (SC) that plays the role of *FSM* ***state register***. The SC initial state is 0; it restarts counting from 0 at the beginning of each instruction of a program and it is reset to 0 once that instruction is finished. A decoder converts the 4-bit output of the SC into time-signals, distinct for each possible output (for example, when the SC output is 0010, then the T_2 output of the decoder will go high, if not, it will remain low for any other combination); this combination of the SC and its decoder implement a *One-hot encoded state register*. The *control commands* for the Datapath are synthesized by the *Instruction Decoder* as (FSM output) functions in terms of the contents of the IR, DR and other signals from the Datapath; a set of gates (AND, OR, and NOT) forms the *Instruction Decoder*. As some *CU* outputs are of Mealy type, a *bank of buffer registers* (*Control Register*) is used to insure a duration of one clock period for the *control commands* that are generated by the *CU*, and to synchronize them with the system clock.

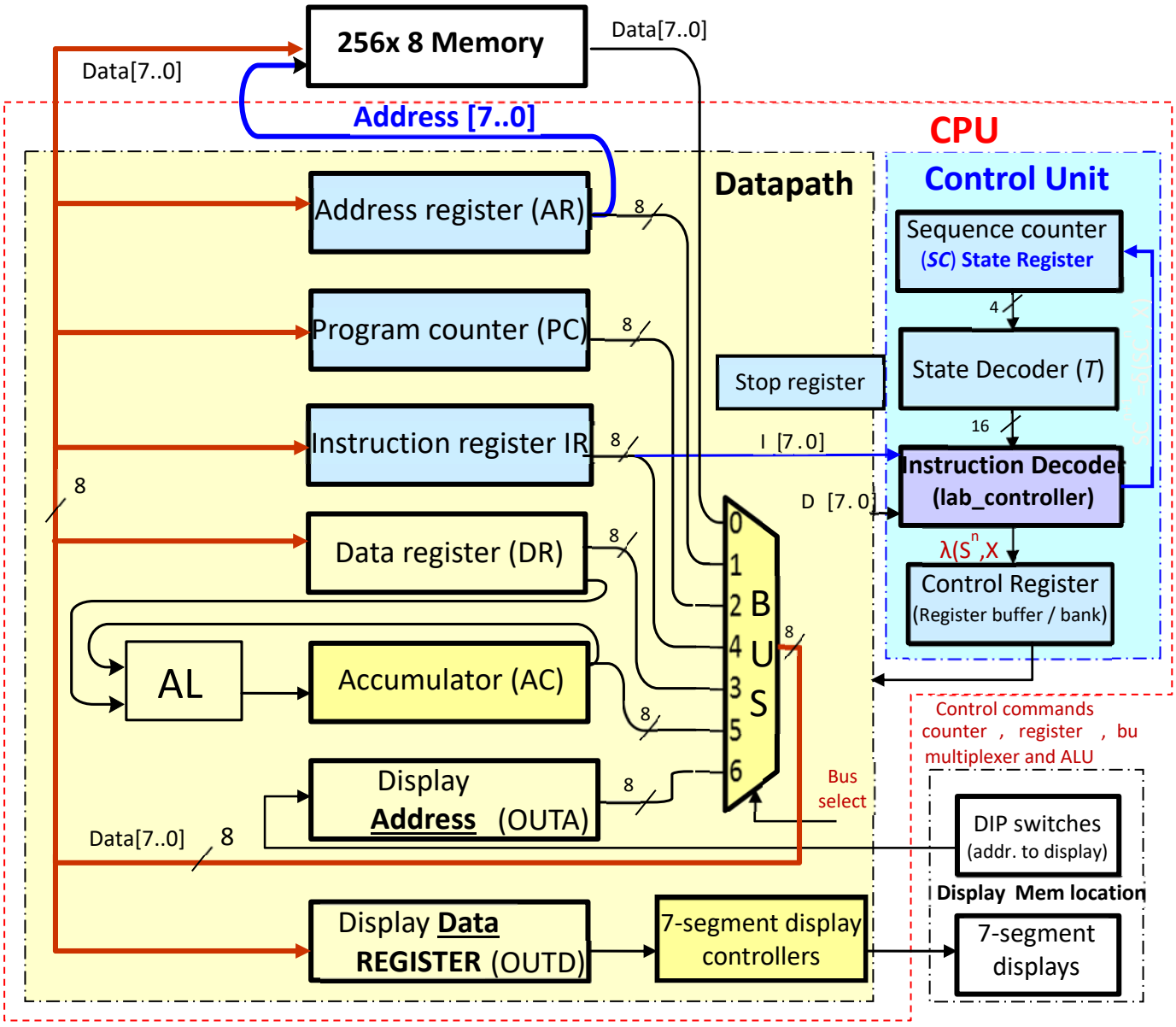

Figure 1: Computer block diagram

The following sequence of steps (grouped in 3 sets – Display, Fetch and Execution) is repeated for each instruction cycle, as long as *Stop Register* = 0:

Display

1. The address of the memory location to being displayed (specified by the DIP switches) is loaded from register OUTA to register AR.
2. The content of memory location at this address is loaded into OUTD whose output is connected to the 7-segment display. Steps 1 and 2 are always executed, even if the system is in *halt*.

Fetch

3. The content of the Program Counter PC (containing at this point the address of the current instruction) is loaded into AR (address register). PC determines the address of the memory location from where the instruction is fetched (read) into the IR of the CPU. Since PC is reset to 0 whenever the UP2 board is programmed, your program's first instruction has to be stored in the first memory location (address 00).
4. PC is incremented to be prepared for getting the next *instruction byte*.
5. The first byte of the instruction (*opcode*) is fetched from memory and is stored in IR (instruction register).
6. The IR content (instruction *opcode*) is decoded by the *Instruction Decoder* (in *Control Unit*).
7. If the instruction refers to a register, then ***skip to step 10***.
8. If the instruction refers to memory, the PC content is transferred to AR, to get the *second byte of the instruction*. This second byte of the instruction may contain
   - either the *address of the operand* - if ***direct*** addressing mode; the instruction's second byte (the *operand address*) is read and directly loaded into AR;
   - or the *pointer to the operand address* (i.e., *address of the operand address*) if ***indirect*** addressing**;** This pointer is read from the memory location pointed by instruction's 2nd byte and is loaded to AR; then a newer read is performed to get the operand address and move it to AR.

   The **FETCH** cycles conclude with AR carrying the *operand address*.

Execution

9. The data (operand) is read from the memory, and the PC is incremented to pointing to the next instruction and being prepared for the next FETCH.
10. The **EXECUTION** cycles of the instruction are implemented, which requires typically several more steps.
11. In the last cycle of the execution the SC is reset to 0, and the procedure begins again at step 1.

NOTE:

- Setting Stop Register = 1=> blocks incrementing PC and stops running further instructions. These steps will be discussed in detail below.
- You will use a new type of file for memory initialization (.mif), to specify the contents of memory; just go to "File" and open a "New" file which will display the list of file formats from where you should select "Memory Initialisation File." For this lab, the file .mif has to be named memorycontents8.mif. To use hex numbers, you have to right-click on the Addr region and then choose Hexadecimal for both Address and Memory (content) Radix. Save Every time you change the contents of this file, you have to save and recompile your project.
- Every time you want to run the program that you loaded in the memory, you have to reprogram your device.

4.3 Syntax of the computer instructions

The instructions that can be executed by the computer are shown in Table 1.

- The second most significant bit (IR_6) specifies if the instruction is a *memory-reference instruction* (IR_6 = **0**) or a *register-reference instruction* (IR_6 = **1**).
- The msb (I = IR_7) specifies the instruction addressing mode (*direct* or *indirect*), while
- the other 6 bits form the *opcode*. For *memory-reference instructions* (IR_6=0) with *direct addressing* mode (IR_7=**0**) another memory access is needed to read the operand, while two more memory read cycles are needed to get the operand in case of *indirect addressing*. In conclusion,

- *register-reference instructions* are encoded with one byte: {*opcode*} with IR_6 = **1**,
- *memory-reference instructions* employ a two-byte format as follows:
  - *memory-reference instructions* with *direct addressing*: {*opcode* with $IR_{7,6}$=**00**, *operand address*}
  - *memory-reference instructions* in *indirect addressing*: {*opcode*, address of the *operand address*}.

Table 1: *Computer Instructions List*

Type of Instruction	Symbol	Binary opcodes = hex opcodes Direct Addressing I = IR_7 = 0	 Indirect Addressing I = IR_7 = 1	Description
Memory Reference (IR_6 = **0**)	AND	**00** 000001=01	**10** 000001=81	AND AC to memory word
	ADD	**00** 000010=02	**10** 000010=82	Add a memory word to AC
	SUB	**00** 000011=03	**10** 000011=83	Subtract a memory word from AC
	LDA	**00** 000100=04	**10** 000100=84	Load AC from a memory location
	STA	**00** 001000=08	**10** 001000=88	Store AC to a memory location
	BUN	**00** 010000=10	**10** 010000=90	Branch unconditionally
	ISZ	**00** 100000=20	**10** 100000=A0	Increment content of memory location and skip the following instruction if the incremented number is 0
Register Reference (IR_6 = **1**)	CLA	01 000001=41		Clear AC
	CMA	01 000010=42		Complement AC
	ASL	01 000100=44		Arithmetic left shift AC
	ASR	01 001000=48		Arithmetic right shift AC
	INC	01 010000=50		Increment AC
	HLT	01 100000=60		Halt. A *Stop* bit is set to 1, which prevents PC from being incremented.

Now let us consider the following **example** file which presents a **simple program** in the memory (in *machine code* and commented with ***assembly language*):**

```
      %  THE PROGRAM IS IN THE ADDRESS ZONE 00 - 7F  %
      00: 04;   %  LDA (direct)  %
      01: 80;   %    from address 80H; AC←M[80], AC contains now the number 1AH%
      02: 44;   %  ASL (direct); AC contains 34H now %
      03: 02;   %  ADD (direct)  %
      04: 81;   %      number of address 81H to AC; AC contains now 34H+2BH=5FH%
      05: 08;   %  STA (direct)  %
      06: a0;   %      AC sent to the address memory A0H; M[a0] ← 5FH %
      07: 83;   %  SUB (indirect)  %
      08: 90;   %      @ address 90H is data pointer (82H); AC =5FH-65H =FAH %
      09: 08;   %  STA (direct)  %
      0a: a1;   %      store AC to the address memory A1H %
      0b: 60;   %  HLT  %
      %  DATA ARE IN THE ADDRESS ZONE 80 to FF  %
      80: 1a;
      81: 2b;
      82: 65;
      90: 82;
      %  address A0H is reserved for the addition result %
      %  address A1H is reserved for the subtraction result %
```

The hex number of the first column indicates the address in the memory, while the second hex number indicates the instruction (opcode or operand's address) represented in *machine code*. The symbol % delimits the comments; these comments will contain the *assembly language* mnemonics of *opcode* of the instructions at hand (*assembly language* representation).

From operating point of view, the memory space of our computer is divided in two: the first half (00-7F) carries the code of your program, while the second half (80-FF) is dedicated to storing data (initial operands and results).

(#00) The program starts at address 00, with memory –reference instruction LDA that loads the register AC with a number which is stored at address 80H; this number is 1AH.
(#02) The second program line contains a register-reference instruction, ASL, which is executed in only one reading cycle of memory. The AC contents are shifted to the left and become 34H.
(#03) This instruction adds the number stored at address 81H to the AC, i.e., performs 34H+2BH= 5FH
(#05) Next instruction stores the sum obtained in AC at address A0H.
(#07) An *indirect* subtraction is performed here. Address 90H contains a pointer to the address 82H, where the number to being subtracted (65H) is stored. The operation is then 5FH– 65H = 0FH;
(#9) The resulted difference 0FH (of AC) is stored at memory address A1H, and the program stops.
Choosing addresses A0H and A1H by DIP, the numbers **5F**H and **FA**H**,** respectively, can be seen on the 7-segment displays.

The .mif file which represents this program in *machine code* (**bold** characters of the above example) is shown in fig. 2 as it is stored in the memory.

Addr	+0	+1	+2	+3	+4	+5	+6	+7
00	04	80	44	02	81	08	A0	83
08	90	08	A1	60	00	00	00	00
10	00	00	00	00	00	00	00	00
18	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00
28	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00
38	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00
58	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00
68	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00
78	00	00	00	00	00	00	00	00
80	1A	2B	65	00	00	00	00	00
88	00	00	00	00	00	00	00	00
90	82	00	00	00	00	00	00	00

Figure 2: .mif file which represents the program in *machine code*

4.4 Detailed Description of the Control Functions (to be generated by the Control Unit)

Starting from Table 1, one can observe that there are three types of distinct instructions that are encoded by the two most significant bits of the instruction register (IR):

1. $X_0 = \overline{IR}_7 \overline{IR}_6$ indicates a direct *memory-reference instruction*;
2. $X_1 = \overline{IR}_7 IR_6$ indicates a *register-reference instruction*
3. $X_2 = IR_7 \overline{IR}_6$ indicates an indirect *memory-reference instruction*.

The seven *memory-reference instructions* can be discriminated by the following control signals (note that $X_0 + X_2 = \overline{IR_6}$), employing the *one-hot encoding* (LDA; STA; BUN; and ISZ) and partially binary encoding (AND, ADD and SUB).

1. $Y_0 = \overline{IR_6}\,\overline{IR_1}\,IR_0$: AND;
2. $Y_1 = \overline{IR_6}\,IR_1\,\overline{IR_0}$: ADD;
3. $Y_2 = \overline{IR_6}\,IR_1\,IR_0$: SUB;
4. $Y_3 = \overline{IR_6}\,IR_2$: LDA;
5. $Y_4 = \overline{IR_6}\,IR_3$: STA;
6. $Y_5 = \overline{IR_6}\,IR_4$: BUN; and
7. $Y_6 = \overline{IR_6}\,IR_5$: ISZ.

These "designations" for X_i and Y_j will be used in the following sections.

The following three tables detail the instruction cycles; you have to read and analyse these tables attentively and make sure that you understand each step. These tables represent the specifications for the design of the CPU Control Unit. The ALU functions are described in TABLE 5. Please note that

- *register-reference instructions* require only one memory READ cycle (to get the *opcode* in T_3),
- *memory-reference instructions* in *direct addressing* mode need three memory READ cycles:
  - 2 READ cycles to **fetch** the instruction (one for the *opcode* in T_3 and another one for the *operand address* in T_6), and
  - a third READ cycle to get the *operand* in T_8 (if needed) to **execute** the instruction,
- *memory-reference instructions* in *indirect addressing* mode need four memory READ cycles (3 READ cycles to **fetch** the instruction – the 1st READ cycle to read the *opcode* in T_3, the 2nd READ cycle to read the *address of the operand address* in T_6, the 3rd READ cycle to get the *operand address* in T_7, and the 4th READ cycle to get the *operand* in T_8 - if needed, to **execute** the instruction.

Table 2: Instruction Fetch Cycle (Initialisation) - common to all types of instruction

State	Description	Notation RTL
T_0	Load the address register AR with the contents of OUTA	T_0: $AR \leftarrow OUTA$
T_1	Read memory location pointed to by AR to the *data output register* OUTD	T_1: $OUTD \leftarrow M[AR]$
T_2	▪ Load AR register with the ADDRESS of the *opcode* of the current instruction (PC) ▪ Increment PC (if the program is running, i.e., if Stop FF S = 0) to point to the address of the next byte to be read, which can be: o either the next instruction, if the current instruction is a *register-reference instruction* o or the 2nd byte of the current instruction, if it is a *memory-reference instruction*.	T_2: $AR \leftarrow PC$ T_2S': $PC \leftarrow PC + 1$
T_3	Read the instruction's first byte (*opcode*) from memory location specified by AR to IR	T_3 : $IR \leftarrow M[AR]$
T_4	This state is a delay that allows the *opcode* to be decoded in the *Control Unit*.	(nothing)
T_5	If X_1, the byte in IR is a *register - reference instruction* and will be executed now	T_5X_1: execute instruction from Table 3 T_5X_1 :$SC \leftarrow 0$
	If X_0 or X_2 (*memory - reference instruction*), ▪ PC is copied to AR, i.e, the address of the instruction's 2nd byte goes from PC to AR => now AR contains the ADDRESS of o the *operand address* if *direct addressing*, or o the **address** *of the operand address* if *indirect addressing* ▪ Increment PC if the program is running (Stop FF S=0). Note that $(X_0 + X_2) = \overline{IR_6}$.	$T_5IR'_6$: $AR \leftarrow PC$ $T_5IR'_6S'$: $PC \leftarrow PC + 1$
T_6	Read from memory location pointed to by AR to AR; the read byte is ▪ the *operand address*, if *direct addressing*, or ▪ the **address** *of the operand address, if indirect addressing*	$T_6IR'_6$: $AR \leftarrow M[AR]$
T_7	If *indirect addressing*, read the *operand address* from memory location pointed to by AR	T_7X_2 : $AR \leftarrow M[AR]$
	If *direct addressing*, don't do anything, as the operand's address is already in AR since T_6	T_7X_0 : (nothing)
T_8 & after	Execute the ***memory - reference instruction*** as described in	(see

	Table 4.	Table 4)

Table 3: Instruction Execution Cycle - Control of the ***register - reference instructions***

Symbol	RTL Notation
CLA	$T_5X_1IR_0 : AC \leftarrow 0$
CMA	$T_5X_1IR_1 : AC \leftarrow \overline{AC}$
ASL	$T_5X_1IR_2 : AC \leftarrow \text{ashl}\, AC$
ASR	$T_5X_1IR_3 : AC \leftarrow \text{ashr}\, AC$
INC	$T_5X_1IR_4 : AC \leftarrow AC + 1$
HLT	$T_5X_1IR_5 : S \leftarrow 1$

Table 4: Instruction Execution Cycle - Control of the ***memory - reference instructions***

Symbol	RTL Notation
AND	$T_8Y_0 : DR \leftarrow M[AR]$ $T_9Y_0 : AC \leftarrow AC \wedge DR,\ SC \leftarrow 0$
ADD	$T_8Y_1 : DR \leftarrow M[AR]$ $T_9Y_1 : AC \leftarrow AC + DR,\ SC \leftarrow 0$
SUB	$T_8Y_2 : DR \leftarrow M[AR]$ $T_9Y_2 : AC \leftarrow AC - DR,\ SC \leftarrow 0$
LDA	$T_8Y_3 : DR \leftarrow M[AR]$ $T_9Y_3 : AC \leftarrow DR,\ SC \leftarrow 0$
STA	$T_8 :$ (cycle not allocated to allow the address bus to stabilize) $T_9Y_4 : M[AR] \leftarrow AC,\ SC \leftarrow 0$
BUN	$T_8Y_5 : PC \leftarrow AR,\ SC \leftarrow 0$
ISZ (assuming that the next instruction is a memory-reference instruction, stored at 2 memory location further down)	$T_8Y_6 : DR \leftarrow M[AR]$ $T_9Y_6 : DR \leftarrow DR + 1$ $T_{10}Y_6 : M[AR] \leftarrow DR$ $T_{11}Y_6 : \text{si}\,(DR = 0)\,\text{alors}\,(\overline{S} : PC \leftarrow PC + 1)$ $T_{12}Y_6 : \text{si}\,(DR = 0)\,\text{alors}\,(\overline{S} : PC \leftarrow PC + 1),\ SC \leftarrow 0$

Table 5: ALU operations table

S2	S1	S0	Operation	Description
0	0	0	*AC + DR*	Addition
0	0	1	*AC + DR' + 1*	Subtraction: *AC - DR*
0	1	0	*ashl AC*	*AC* arithmetic left shift
0	1	1	*ashr AC*	*AC* arithmetic right shift
1	0	0	$AC \wedge DR$	logic AND
1	0	1	$AC \vee DR$	logic OR
1	1	0	*DR*	*DR* transfer to *AC*
1	1	1	*AC'*	Complement *AC*

To illustrate the computer's operation, let's track the registers contents as the following *memory – reference instruction* is run:

00: ADD 80H
02: …

This translates into the *machine code* stored in the computer's memory as follows

Addr	Content
00:	02
01:	80
02:	…

State	PC	AR	IR	DR	AC	SC+	RTL
T_0	00			0	0	1	
T_1	00			0	0	2	
T_2	01 PC++	(**PC**) = **00** = ADDRESS of the ***opcode***		0	0	3	$T_2 : AR \leftarrow PC$ $T_2\overline{S} : PC \leftarrow PC+1$
T_3	01	00 = ADDRESS of the ***opcode***	**M[AR=0] = 02 =*opcode***	0	0	4	$T_3 : IR \leftarrow M[AR]$
T_4	01			0	0	5	
T_5	02 PC++	(**PC**) = **01** = ADDRESS of the 2nd byte of the memory-reference instruction (***operand address***)	***opcode*= 02**	0	0	6	$T_5\overline{IR_6} : AR \leftarrow PC$ $T_5\overline{IR_6}\overline{S} : PC \leftarrow PC+1$
T_6	02	**M[AR=01] = 80** ***operand address*** as *direct addressing* (not the address *of the* ***operand address*** since it is not *indirect addressing*)	***opcode* = 02**	0	0	7	$T_6\overline{IR_6} : AR \leftarrow M[AR]$
T_7	02	* **M[AR]** = ***operand address*** if *indirect addressing* (not here) * still the ***operand address*** as *direct addressing* (read in T_6)	***opcode* = 02**	0	0	8	(nothing)
T_8	02	***operand address***	***opcode* = 02**	**M[AR=80] = *operand***	0	9	*DR <- M[AR]*
T_9	02	***operand address***			**0 + *operand***	**0**	*AC <- AC+DR* *SC <- 0*

5. Prelab - Hardware

5.1 Files Analysis

You will start up by analyzing the files .bdf which you are provided with. You can do it either by using Quartus II, or by "deciphering" the diagrams at the end of this document (Fig. 3 - 11). You don't have to understand in detail the RAM operation (*ram256x8*), nor the controller of the 7-segment display. The diagram of the 4 bit SC counter is not included in the figures below, because the counter has the same architecture like the 8 bit counter, but truncated to 4 bits. Finally, the VHDL code of the bus multiplexer is presented in Table 7. Although you did not learn VHDL yet, you will see that the code is easy to understand, and much simpler to implement than would be a .bdf.

Examining the logic diagrams, answer the following questions and write your answers in your report:

1. Draw a diagram which shows the hierarchy of the files, with *lab3top* at the top. For the files *lab3controller, ram256x8, and sevensegcontroller*, you do not have to identify the subfiles.
2. How can you check by analysing these files that only one register will place its output on the data bus at a time?
3. Are the register *reset* (*clear*) signals synchronous or asynchronous? Explain your answer. Note that all the command signals are active at high (i.e. a "1" will reset a register to 0).
4. What happens if a *load* and a *reset* are simultaneously sent to a register? Why?
5. Why the address register is connected directly to the memory?
6. Why the Program Counter, the Data Register, and the Accumulator are implemented as counters?
7. Of all the three commands of the counters (*reset*, *increment*, and *load*), which one has the highest priority? Which one has the lowest priority? Explain your answer.
8. Is it possible to read a value from memory directly to the accumulator? Explain your answer.
9. Analyze the ALU and determine a truth table which describes the 8 operations which can be selected by the three control lines. Are the shift operations logical or arithmetic?

5.2 Design of the Control Unit

Your main objective is to derive the equations of all the *control signals* which have to be generated by the *Control Unit* in order to control the CPU datapath (registers and ALU), the bus and the memory. To this effect, analyze the RTL expressions of Table 2, Table 3, Table 4, and write the *logic expression* for each of the following *control signal* (λ and δ functions of the *Control Unit*):

Memory ($\lambda_M(In, T)$)
1. memwrite

CPU registers ($\lambda_R(In, T)$)
2. AR_Load
3. PC_Load
4. PC_Inc
5. DR_Load
6. DR_Inc
7. IR_Load
8. AC_Clear
9. AC_Load
10. AC_Inc
11. OUTD_Load

CPU ALU ($\lambda_{ALU}(In, T)$)
12. ALU_Sel2
13. ALU_Sel1
14. ALU_Sel0

Bus (data mux ($\lambda_{Bus}(In, T)$)
15. BusSel2
16. BusSel1
17. BusSel0

Control Unit ($\delta(In, T)$)
18. SC_Clear
19. Halt

The inputs (*In*) of the *Control Unit* are: the *Instruction Register IR* [7..0], the *Data Register DR* [7..0], the *State Register* (*SC*) *T*[12..0], and the stop command (*Stop*) of the *Stop* register. The X_i and Y_j functions (as described in section 4.3) are the core of the *Instruction Decoder* and have to be implemented first, to allow for deriving all the *control signal* (CU's λ and δ functions) from them.

To this effect, examine the tables and determine which signals must be activated to execute each RTL line. For each *control signal*, derive a list of the conditions under which that signal is activated. After making up the list for a particular signal, you can simply do an OR of each condition/term to obtain the final expression of that *control signal*. For example, let consider the bus (multiplexer) for which three select lines (BusSel [2..0]) have to be derived ($\lambda_{Bus}(In, T)$). Let synthesize in a table all the conditions under which each register places its output onto the bus:

Circuit which writes on the bus	Control Conditions (*In*, *T*)	Bus select = $\lambda_{Bus}(In, T)$		
		BusSel2	**BusSel1**	**BusSel1**
Memory	T_1 T_3 $T_6\overline{IR_6}$ T_7X_2 $T_8(Y_0 + Y_1 + Y_2 + Y_3 + Y_6)$	0	0	0
AR	T_8Y_5	0	0	1
PC	T_2 T_5	0	1	0
DR	$T_{10}Y_6$	0	1	1
IR	(never happens here)	1	0	0
AC	T_9Y_4	1	0	1
OUTA	T_0	1	1	0
(not used)	(indifferent)	1	1	1

From this table, one can see that $BusSel2 = T_0 + T_9Y_4$. Actually you have to design an encoder that generates BusSel [2..0] in terms of the *Control Conditions*. Similar Boolean expressions have to be derived for all the other control signals. To eliminate the dependence of the duration of the output functions λ(*In*, *T*) and transition functions δ(*In*, *T*) to the input signals, every function is sampled and stored in a buffer synchronously with the system clock.

It is worth to pay great attention when you derive your lists!

6. Procedure - Hardware

6.1 Build and test the Control Unit

6.1.1 Automatic run

1. Create a new *Quartus II Project* and name it appropriately with *File* → *New Project Wizard* or *File* → *New* → *New Quartus II Project*
   Add *support* files (both *.bdf* and *.vhd*) to project: *Project*→*Add/Remove Files from Project*
   Set *lab3top.bdf* as the top-level entity.
2. Open the file *lab3controller.bdf* (which gives the list of the pins of the inputs and outputs in the order required for generating by default the same symbol as shown in the file *lab3top.bdf*). Use AND, OR and NOT gates to implement the Boolean expressions that you derived in the prelab for the control signals (section 5.3). It is a good idea to make virtual connections by giving names to wires (simply click on a wire and enter the name; the wires which have the same

names are automatically connected by the compiler). If you do not use virtual connections, your file will quickly become an incomprehensible set of spaghetti. Try to reduce the number of logic gates that a signal must go through. The clock period of the DE2-115 board is only 20 ns, and errors can occur if signals are delayed over this period.

3. Create your Memory Initialization File (*memorycontents8.mif*): *File* → *New* → *Memory Initialization File*; Select your radix etc... *Hexadecimal.* You can either put your test program into the *.mif (like the one in Fig. 2) now or later. Save the file.
4. Set *lab3controller.bdf* as *Top-Level Entity* and make sure that it compiles without errors.
5. Assign *lab3top.bdf* to the project (set as *Top-Level Entity*), and choose the device **EP4CE115F29C7**. Assign the pins as shown in Table 6. (right-click on pin → *Locate* → *Locate in Assignment Editor* and then switch in the *Category* panel from *All* to *Locations-Pins* to select the pin in the *Edit* panel, in the *Location* tab, etc…

Table 6: Pins Assignment

Pin Name	Pin Number	Component	Pin Name	Pin Number	Component
clk	PIN_Y2	50MHz clock	A1	PIN_G18	HEX0[0]
Auto	PIN_Y23	SW[17]	B1	PIN_F22	HEX0[1]
DIP7	PIN_AB26	SW[7]	C1	PIN_E17	HEX0[2]
DIP6	PIN_AD26	SW[6]	D1	PIN_L26	HEX0[3]
DIP5	PIN_AC26	SW[5]	E1	PIN_L25	HEX0[4]
DIP4	PIN_AB27	SW[4]	F1	PIN_J22	HEX0[5]
DIP3	PIN_AD27	SW[3]	G1	PIN_H22	HEX0[6]
DIP2	PIN_AC27	SW[2]	A2	PIN_M24	HEX1[0]
DIP1	PIN_AC28	SW[1]	B2	PIN_Y22	HEX1[1]
DIP0	PIN_AB28	SW[0]	C2	PIN_W21	HEX1[2]
1_instruction	PIN_M23	KEY[0]	D2	PIN_W22	HEX1[3]
AR[0]	PIN_G19	LEDR[0]	E2	PIN_W25	HEX1[4]
AR[1]	PIN_F19	LEDR[1]	F2	PIN_U23	HEX1[5]
AR[2]	PIN_E19	LEDR[2]	G2	PIN_U24	HEX1[6]
AR[3]	PIN_F21	LEDR[3]	AC[0]	PIN_E21	LEDG[0]
AR[4]	PIN_F18	LEDR[4]	AC[1]	PIN_E22	LEDG[1]
AR[5]	PIN_E18	LEDR[5]	AC[2]	PIN_E25	LEDG[2]
AR[6]	PIN_J19	LEDR[6]	AC[3]	PIN_E24	LEDG[3]
AR[7]	PIN_H19	LEDR[7]	AC[4]	PIN_H21	LEDG[4]
DR[0]	PIN_J15	LEDR[10]	AC[5]	PIN_G20	LEDG[5]
DR[1]	PIN_H16	LEDR[11]	AC[6]	PIN_G22	LEDG[6]
DR[2]	PIN_J16	LEDR[12]	AC[7]	PIN_G21	LEDG[7]
DR[3]	PIN_H17	LEDR[13]	Stop	PIN_F17	LEDG[8]
DR[4]	PIN_F15	LEDR[14]			
DR[5]	PIN_G15	LEDR[15]			
DR[6]	PIN_G16	LEDR[16]			
DR[7]	PIN_H15	LEDR[17]			

Compile your project.

6. To test and visualise the simulation of your project, create a new *.vwf* file called *lab3top.vwf*. Choose a grid size of 20 ns and an *end of simulation* of 5*μs*. Make a right click and select *"Insert"* then "*Insert Node or Bus…*" and then click on "*node finder*" and on "*List*" selecting *Pins: all* for *Filter,* and *Named* "*". In the panel *Nodes Found* select *clk*, *DIP* [7..0], etc.
7. Set the DIP switches to point to the A0 address. Also make sure the *Auto* signal is set to 1. Start simulation. If your Control Unit functions correctly, the register OUTD would have at the end

the value **5F**, and the *Stop* bit should be activated after approximately 2.7 μs. If you start again your simulation with DIP [7..0] set to A1, the register OUTD should finally contain **FA**. **Show this simulation to your TA.**

6.1.2 Manual run

8. If your simulation does not function, you have to find the source of the problem. Run short test programs (a couple of instructions) to verify the correct generation of the control signals. To help debug the issue, set the *Auto* signal to 0. Setting the *Auto* signal to 0 enables you to step-through the instructions (a push-button press *1_instruction*, permits the user to execute the next instruction) and see where the issue is. In addition, the outputs to the following registers are displayed as well: AR, DR and the Accumulator.
   Check the sequence of the control signals to see whether they correspond to the order described in tables 2 - 4. Keep in mind that the signals arrive one clock cycle after the sequence counter, as they are stored first in the control registers. Another good technique of checking is to observe what is stored in the memory. Examine the signals on the data and address buses each time the signal *memwrite* is requested; this should help you to find out where your program causes an error.
9. Program the DE2-115 board by opening the programmer and choose “Program”. Use the DIP switches to choose the addresses for the data.
10. Enter the program of Section 4.3 in *memorycontents8.mif*. Program the device and use the DIP switches to read the memory content. Was your analysis of the program correct? D**emonstrate to your TA.**

7. Prelab - Software

7.1 Program Analysis

1. Analyze the following program. Examine the instructions one by one, inspecting the values from the AC and the memory to understand what the program does.
2. Write a simple pseudo code to describe the program. Give the following names to the variables stored at the memory addresses A0 - A3 (hex):

Address	Name of variable
A0	Counter
A1	X
A2	Y
A3	Z

Note that X, Y and Z are pointers.

3. What does calculate this program?
4. Why is practical to use in this program *memory - reference instructions* with *indirect addressing* (in other words, instructions which use pointers)?

```
%  PROGRAM IS IN the RANGE Of ADDRESSES 00 TO 7F %
00: 04;   %  LDA (direct)  %
01: a0;   %  from address a0 %
02: 42;   %  CMA  %
03: 08;   %  STA (direct)  %
04: a0;   %  AC to address a0  %
05: 20;   %  ISZ (direct)  %
06: a0;   %  counter stored at a0  %
07: 10;   %  BUN (direct)  %
08: 20;   %  to address 20  %
09: 60;   %  HLT  %
20: 84;   %  LDA (indirect)  %
21: a1;   %  the number pointed to by the memory location a1  %
22: 82;   %  ADD (indirect)  %
23: a2;   %  the number pointed to by the memory location a2  %
24: 88;   %  STA (indirect)  %
25: a3;   %  to the memory location pointed to by a3  %
26: 04;   %  LDA (direct)  %
27: a1;   %  from the address a1  %
28: 50;   %  Inc  %
29: 08;   %  STA (direct)  %
2a: a1;   %  AC to the memory address a1  %
2b: 50;   %  Inc  %
2c: 08;   %  STA (direct)  %
2d: a2;   %  store AC to the memory address a2  %
2e: 50;   %  Inc  %
2f: 08;   %  STA (direct)  %
30: a3;   %  send AC to the memory address a3  %
31: 10;   %  BUN (direct)  %
32: 05;   %  to the memory address 05  %
80: 01;   %  DATA ARE FOUND AT ADDRESSES 80 TO FF  %
81: 01;
a0: 0a;   %  loop counter, which will be done 10 times  %
a1: 80;   %  pointer to the first number to be added  %
a2: 81;   %  pointer to the second number to be added  %
a3: 82;   %  pointer to memory location where result will be stored %
```

7.2 Program Design

1. Write a program which adds consecutively each number of the following sequence of hexadecimal numbers: 21, B5, 37, 08, 5C, 84, A1, 1D, 72, FF, F6, 43, 03, A9, D4, 19, 31, D9, 47, 82, 14, 52, 07, CA, 04. When your current sum becomes equal to zero, your program should store into the memory the last number added, display this number, and eventually stop. **Write the program in machine code in .mif file format, as shown in section 4.2.3.**
2. **(bonus *)** Write a program which can multiply any unsigned number of 4 bits.

It is important to understand how instruction ISZ operates. Let us assume that you want to skip the following instruction if the number of address B0 is equal to FF (hex). Note that FF corresponds to 11111111 (in binary), which is also the 2's complement of - 1 (in decimal). In this case, instruction ISZ B0 will read the number FF, to put it in the data register, and increment to 00. Since the data register is now equal to 0, the next instruction will be skipped. If any other value that FF is at the address B0, then the next instruction will be executed.

Note that if you want to skip the following instruction when the number stored at the address B0 is equal to 00, you need to initially complement it to FF so that condition ISZ is met. This can be done without losing the data of B0 by using the sequence of instructions LDA B0; CMA; STA B1; ISZ B1. It is assumed that B1 is an address which is available for a temporary variable.

8. Procedure - Software

Test your programs

a) Enter the programs which you wrote at section 7 in *memorycontents8.mif* (one at a time). Use the simulator to debug your program; it is possible that you need simulations which extend up to 50 - 100 μs. **Show your simulations to your TA**.
b) Once you are sure that your programs function, configure the DE2-115 with those programs. Use the DIP switches and the 7-segment displays, and make sure that the programs operate as expected. **Show a demonstration to your TA**.

9. Report

Besides the answers to your prelab section, you must include the following in your report:

1. a block diagram of the Control Unit that you conceived; and
2. sample screen snapshots of your simulations or copies of the simulation files, for:
    a. the addition and subtraction program of section 4.3 (as tested in section 6.1.1.6), and
    b. the sums (and multiplication if done) of section 7.2. (as tested in section 8)

A TA will check that your design and your programs function in simulation and on the Altera board if you are physically the lab. Be prepared to answer questions about the lab when you demo it.

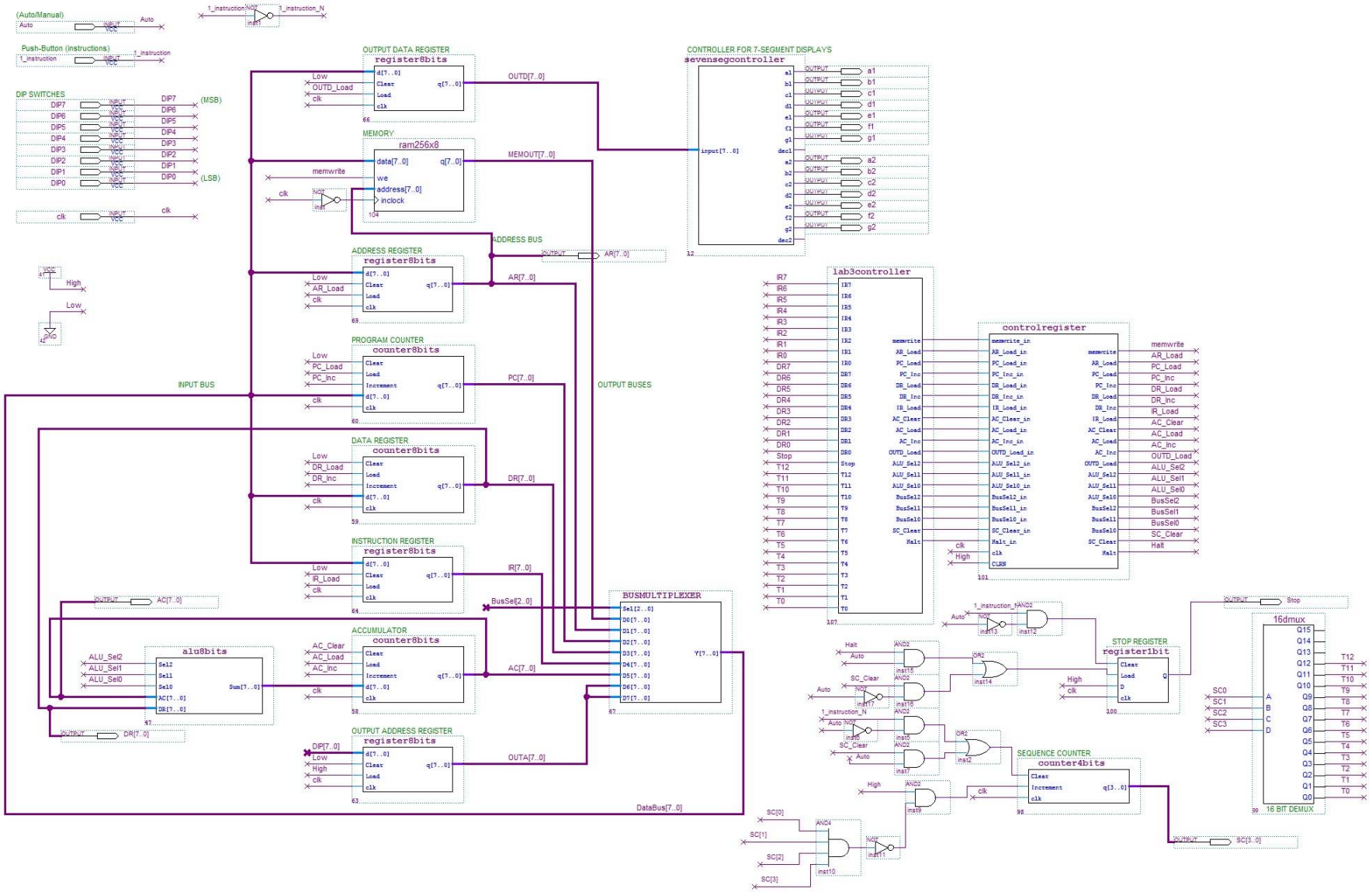

Figure 3: lab3top.bdf

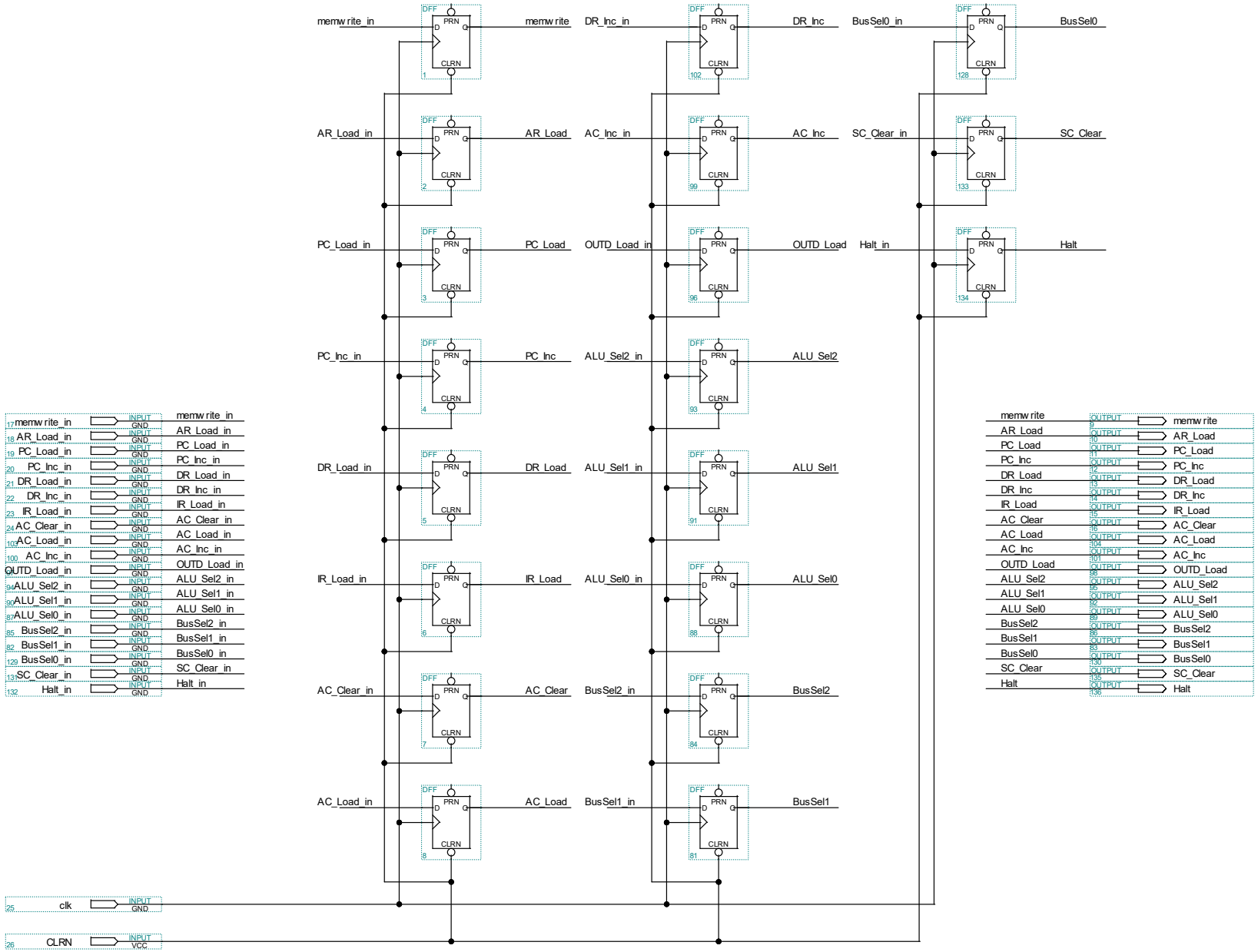

Figure 4: controlregister.bdf

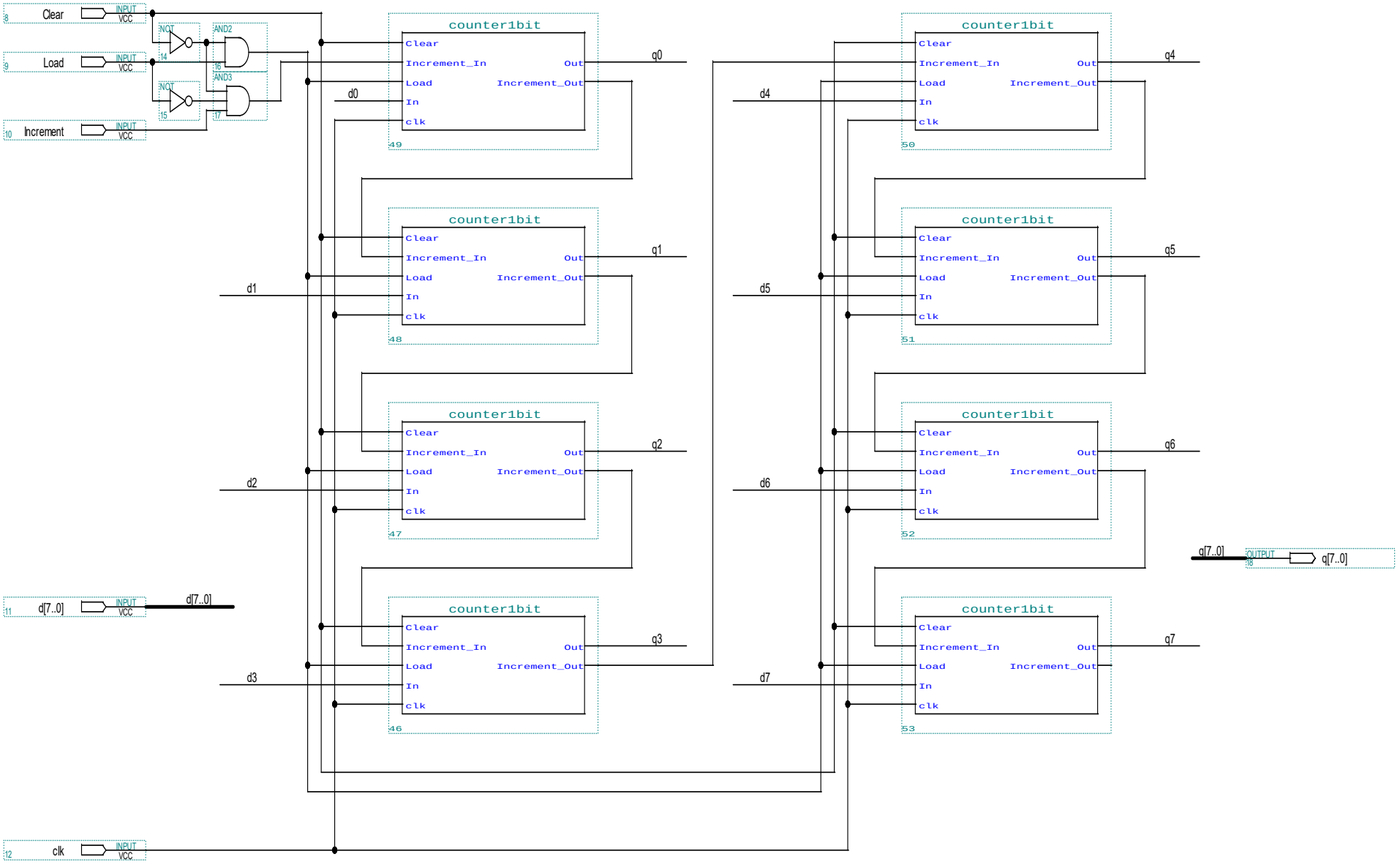

Figure 5: counter8bits.bdf

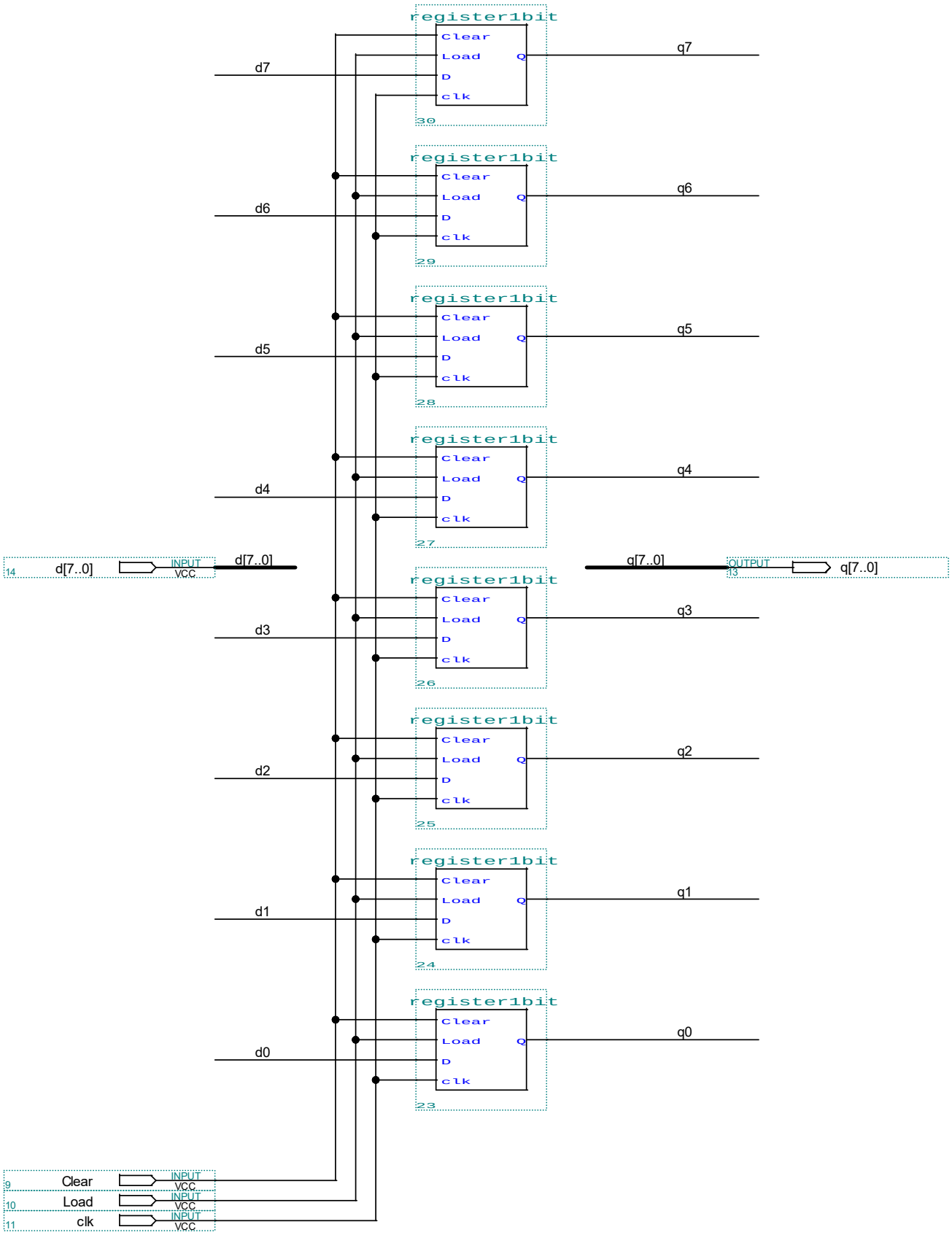

Figure 6: register8bits.bdf

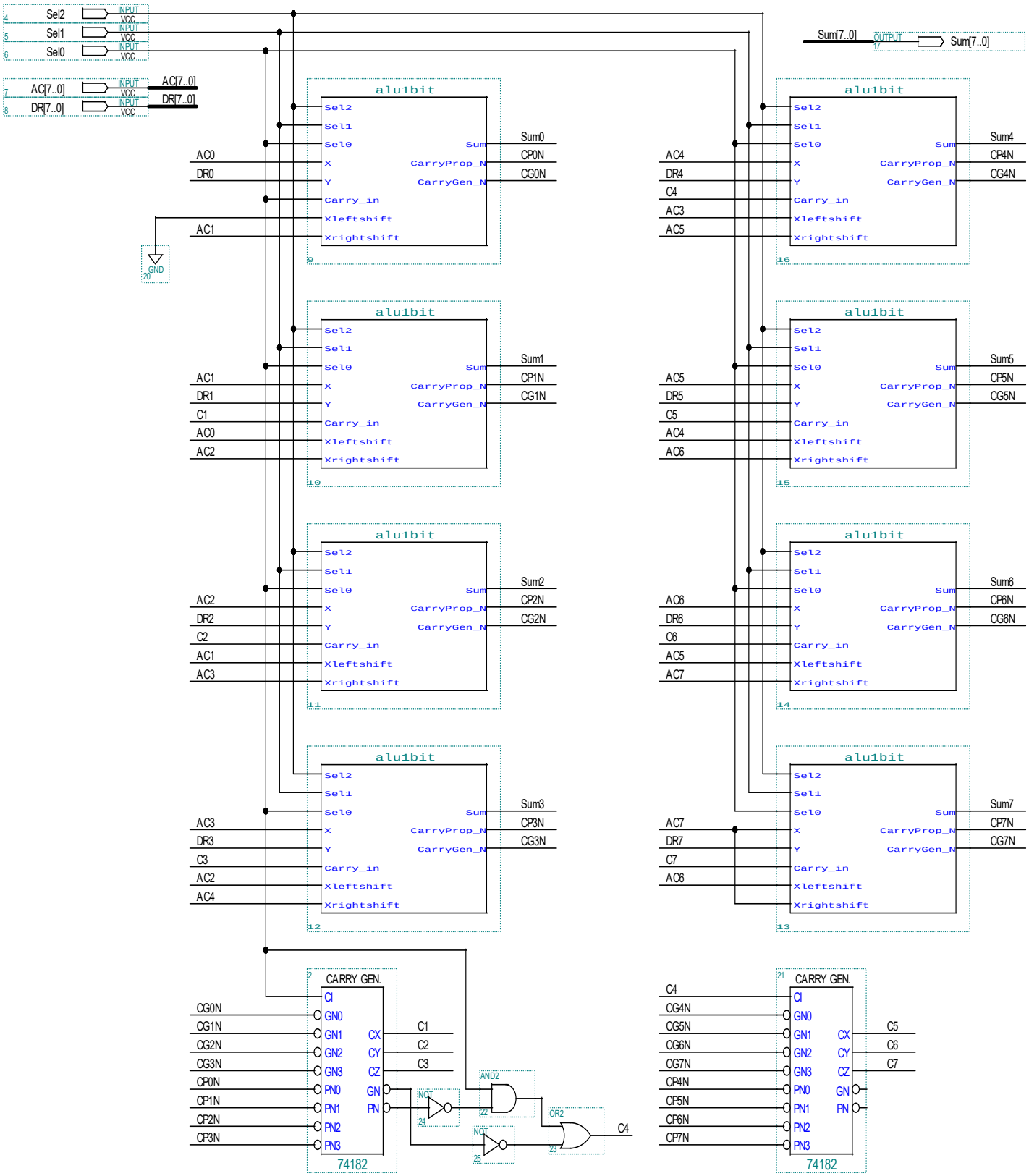

Figure 7: alu8bits.bdf

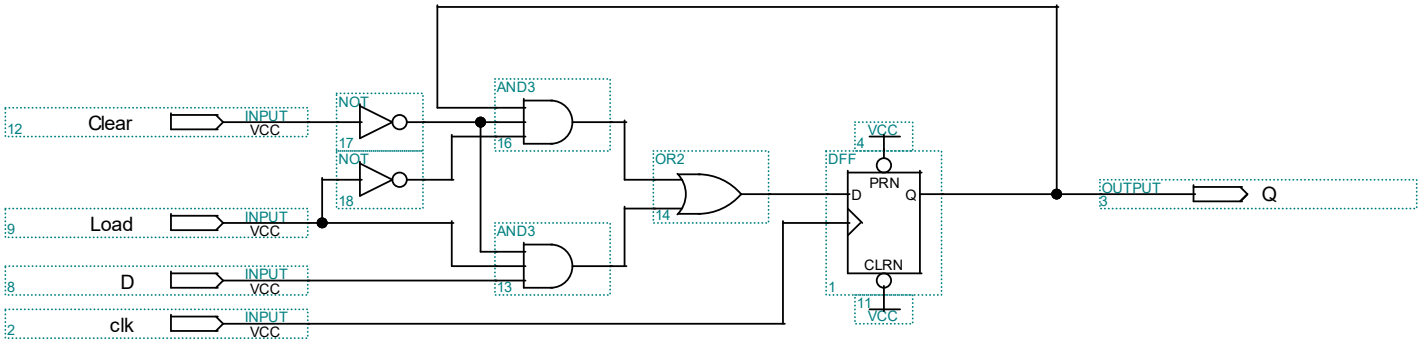

Figure 8: register1bit.bdf

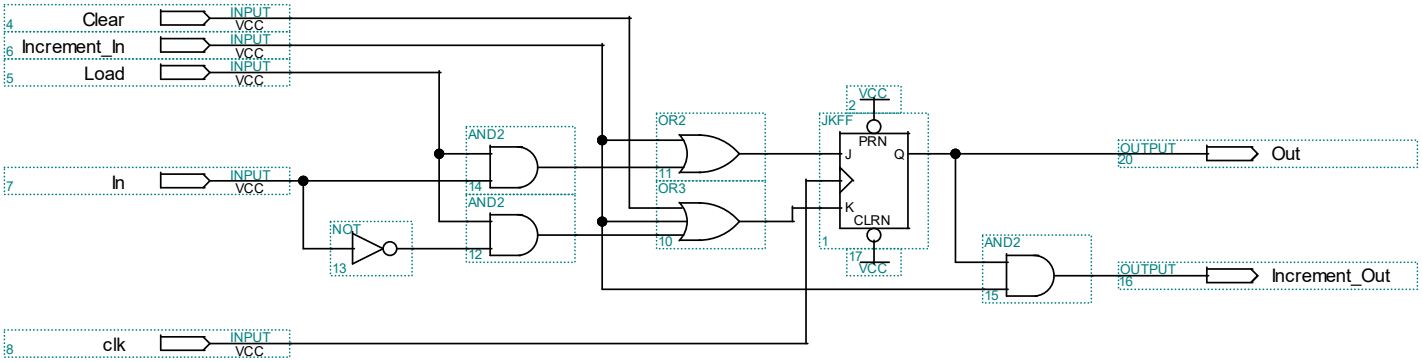

Figure 9: counter1bit.bdf

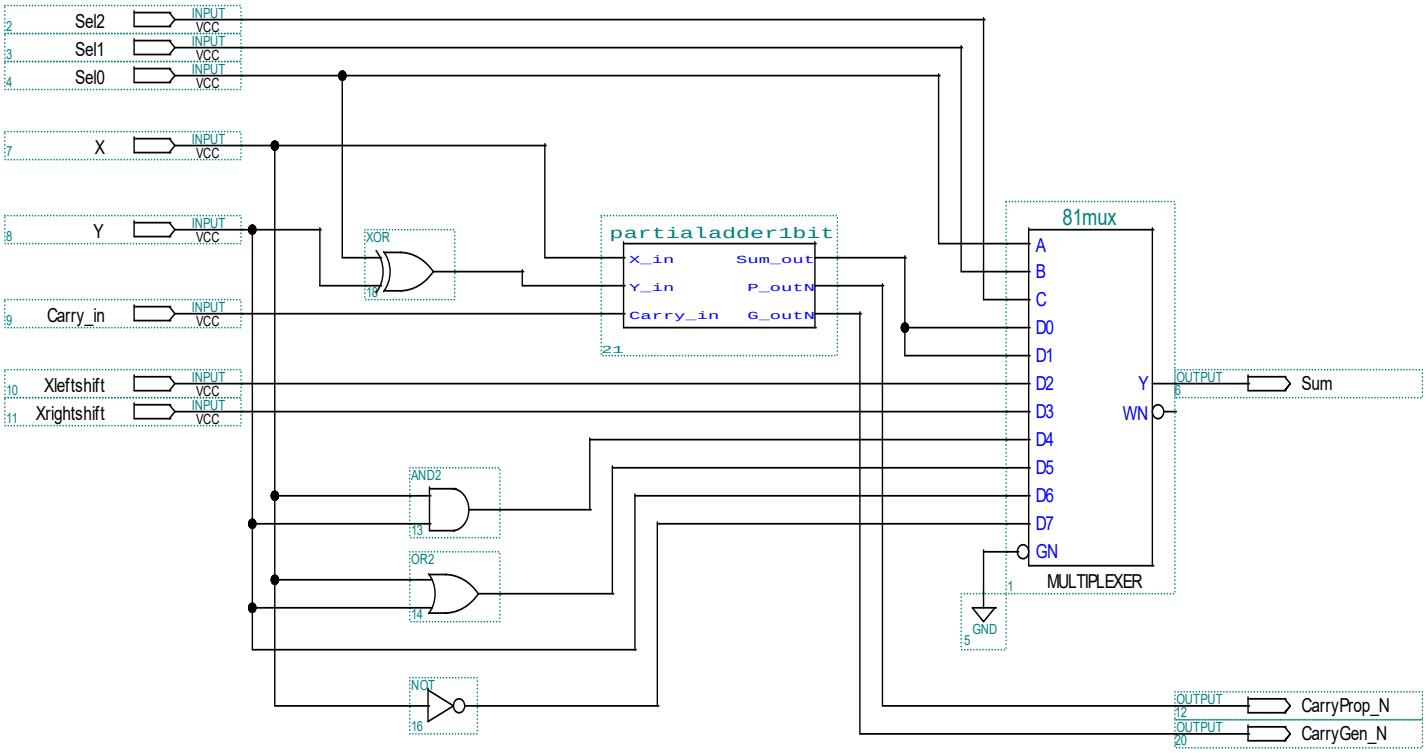

Figure 10: alu1bit.bdf

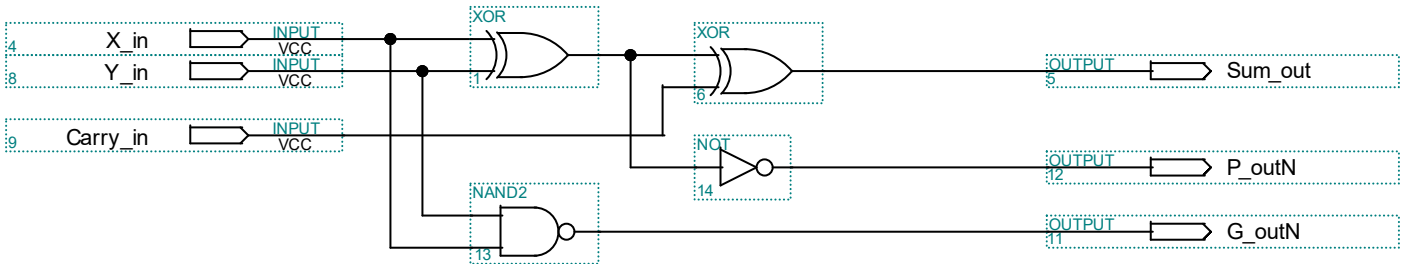

Figure 11: partialadder1bit.bdf

Table 6: VHDL Code for the bus multiplexer

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- Title "Bus multiplexer";
-- File: busmux.vhd
ENTITY busmultiplexer IS
PORT ( Sel        : IN  STD_LOGIC_VECTOR (2 DOWNTO 0);
     D0           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     D1           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     D2           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     D3           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     D4           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     D5           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     D6           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     D7           : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
     Y            : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END busmultiplexer;

ARCHITECTURE busmuxarch OF busmultiplexer IS
BEGIN
Y <=   D0 WHEN Sel = "000" ELSE
     D1 WHEN Sel = "001" ELSE
     D2 WHEN Sel = "010" ELSE
     D3 WHEN Sel = "011" ELSE
     D4 WHEN Sel = "100" ELSE
     D5 WHEN Sel = "101" ELSE
     D6 WHEN Sel = "110" ELSE
     D7;
END busmuxarch;
```