CEG2136 LAB4

Control Unit (CU)

Objectives

In this lab you will design and simulate a CPU that you designed its control unit, using Quartus II as a development environment.

CPU Features:

- ✓ Reduced Instruction Set (RISC) Architecture
- ✓ Hardwired Control Unit
- ✓ Full featured instruction set

CPU Architecture

Overview

This laboratory implements a computer having a structure that is very close to the one presented the textbook. However, there are a major difference:

- 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 of a 4-bit opcode and a 12-bit memory address). In this lab, a memory-reference instruction is 2 byte (16-bit) long as well, but the first byte carries the opcode, while the second 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.
- The block diagram of your computer is presented in Figure 1. The schematics diagram (.bdf) files of all the component blocks, except the Instruction Decoder (lab_controller.bdf), will be provided.
- The Control Unit functions according to a time sequence, which is generated by the sequence counter (SC). The time sequence plays the role of FSM state register. SC initial state is 0; it restarts counting from 0 at the beginning of each instruction and it is reset to 0 once that instruction execution is finished.
- A decoder converts the 4bit output of the SC into time-signals, distinct for each possible output (for example, when the SC output is 0010, then the T2 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 **Instruction Decoder** is a combinational circuit that takes IR, DR, T_i as input, and output the data-path control signals, e.g registers' load/increment/reset/bus-selec/etc.
- 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.

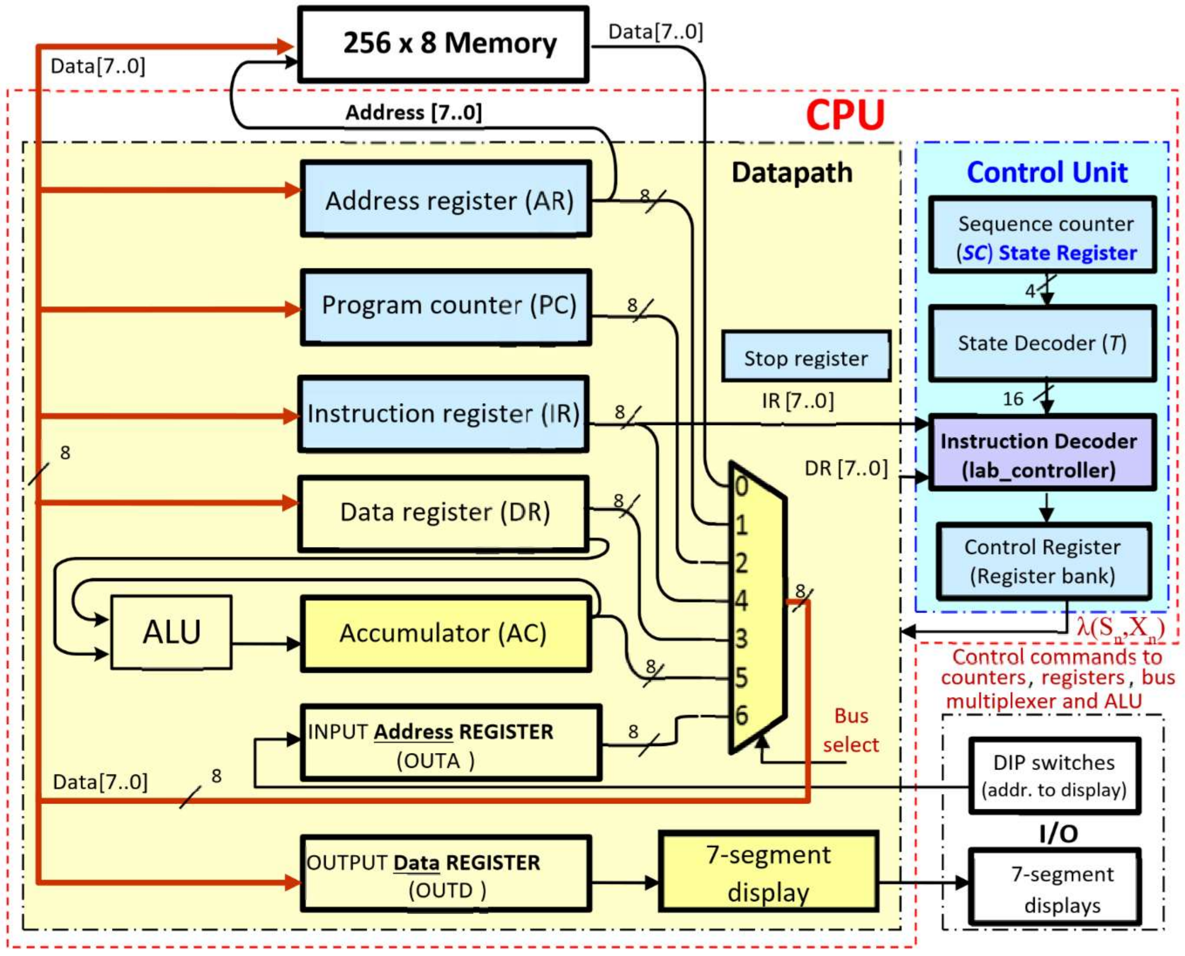

256 x 8 Memory
Data[7..0]
Data[7..0]
Address [7..0]
CPU
Datapath
Control Unit
Address register (AR)
Program counter (PC)
Instruction register (IR)
Data register (DR)
ALU
Accumulator (AC)
INPUT Address REGISTER (OUTA)
OUTPUT Data REGISTER (OUTD)
7-segment display
Stop register
IR [7..0]
DR [7..0]
Sequence counter (SC) State Register
State Decoder (T)
Instruction Decoder (lab_controller)
Control Register (Register bank)
λ(Sn,Xn)
Control commands to counters, registers, bus multiplexer and ALU
Bus select
Data[7..0]
DIP switches (addr. to display)
I/O
7-segment displays
8
4
16

Figure 1: Computer block diagram

The following sequence of steps (grouped in 3 sets – **Display**, **Fetch** and **Execute**) is repeated for each instruction cycle, as long as Stop Register is not set:

Display

1. The address of the memory location to be 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

1. The content of the Program Counter PC (containing at this point the address of the current instruction) is loaded into AR (address register). PC defines 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 you start simulation, your program's first instruction has to be stored in the first memory location at address 0.
2. PC is incremented to be prepared for getting the next instruction byte.
3. The first byte of the instruction (opcode) is fetched from memory and is stored in IR (instruction register).
4. The IR content (instruction opcode) is decoded by the Instruction Decoder (in Control Unit).
5. If the instruction is register access, then skip to step 8.
6. If the instruction memory access, the PC content is transferred to AR, to get the second byte of the instruction. This second byte of the instruction may contain:
    a. 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;
    b. 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 an extra read is performed to get the operand address and move it to AR.
    c. The FETCH cycles conclude with AR carrying the operand address.

Execution

7. 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.
8. The EXECUTION cycles of the instruction are implemented, which requires typically several more steps.
9. In the last cycle of the execution the Sequence Counter (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.
- The processor memory is initialized using a file called **memorycontents8.mif**. It defines the memory size and width, and the initial contents of the memory.
    - The memory will be initialized with the machine code instructions of the lab.

Syntax of the computer instructions

The instruction set is shown in Table 1.

Table 1: CPU Instruction Set

Type of Instruction	**Symbol**	**Binary/hex opcodes**		**Description**
		Direct Addressing I = IR6 = 0	Indirect Addressing I = IR6 = 1	
	LDD	**00000000=0x00**	**01000000=0x40**	Load AC directly from instruction
Memory Reference **(IR7=0)**	AND	**00000001=0x01**	**01000001=0x41**	AND AC to memory word
	ADD	**00000010=0x02**	**01000010=0x42**	Add AC to memory word
	SUB	**00000011=0x03**	**01000011=0x43**	Subtract a memory word from AC
	LDA	**00000100=0x04**	**01000100=0x44**	Load AC from a memory word
	STA	**00001000=0x08**	**01001000=0x48**	Store AC to a memory word
	BUN	**00010000=0x10**	**01010000=0x50**	Branch unconditionally
	ISZ	**00100000=0x20**	**01100000=0x60**	Increment memory word and skip the following instruction if the result is zero
Register Reference **(IR7=1)**	CLA	**10000001=0x81**		Clear AC
	CMA	**10000010=0x82**		Complement AC
	ASL	**10000100=0x84**		Arithmetic shift left AC
	ASR	**10001000=0x88**		Arithmetic shift right AC
	INC	**10010000=0x90**		Increment AC
	HLT	**10100000=0xA0**		Halt

- **Bit7** of the instruction register, IR[7], specifies if the instruction is a memory-reference instruction, where IR[7]=0, or a register reference instruction, where IR[7]=1.
- **Bit6** of the instruction register, IR[6], specifies the instruction addressing mode, direct addressing, where IR[6]=0, or indirect addressing, where IR[6]=1.
- Register reference instructions are 8-bits long. It requires one memory read to read the instruction's OpCode.
- Memory refernce instructions are 16-bits. It requires two memory reads to fetch the instruction. The first memory read is to get the instruction's OpCode, and the second read is to get the source/destination <u>address</u>.
- Direct load instruction (LDD) is 16-bits. It requires two memory reads to fetch the instruction. The first memory read is to get the instruction's OpCode, and the second read is to get the <u>value</u> to be stored in the AC register.
- In case of indirect addressing mode, IR[6]==1, the target address is considered a pointer to the address. This means it requires an extra memory read to get the actual address of the target.

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). Signals $\mathbf{X_0}$, $\mathbf{X_1}$, $\mathbf{X_2}$ signifies this event.

1. Direct memory reference instructions, identified by control signal $\mathbf{X_0 = IR_7'IR_6'}$
2. Indirect memory reference instructions, identified by Control signal $\mathbf{X_1 = IR_7'IR_6}$
3. Register reference instructions, identified by Control signal $\mathbf{X_2 = IR_7IR_6'}$

The seven memory-reference instructions can be discriminated by the following control signals, employing the one-hot encoding (LDA; STA; BUN; and ISZ) and partially binary encoding (AND, ADD and SUB).

1. **AND** (direct/indirect) identified by control signal $\mathbf{Y0 = IR_7'IR_1'IR_0}$
2. **ADD** (direct/indirect) identified by control signal $\mathbf{Y1 = IR_7'IR_1IR_0'}$
3. **SUB** (direct/indirect) identified by control signal $\mathbf{Y2 = IR_7'IR_1IR_0}$
4. **LDA** (direct/indirect) identified by control signal $\mathbf{Y3 = IR_7'IR_2}$
5. **STA** (direct/indirect) identified by control signal $\mathbf{Y4 = IR_7'IR_3}$
6. **BUN** (direct/indirect) identified by control signal $\mathbf{Y5 = IR_7'IR_4}$
7. **ISZ** (direct/indirect) identified by control signal $\mathbf{Y6 = IR_7'IR_5}$
8. **LDD** identified by control signal $\mathbf{Y7 = (IR_5+IR_4+IR_3+IR_2+IR_1+IR_0)'}$

These designations for $\mathbf{X_i}$ and $\mathbf{Y_j}$ 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 one memory read cycle (to get the opcode in T1),
- memory-reference instructions in direct addressing mode need three memory read cycles:
    - 2 read cycles to fetch the instruction (one for the opcode in T1 and another one for the operand address in T4), and
    - a third cycle to get the operand in T6 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 cycle to read the opcode in T1, the 2nd cycle to read the address of the operand address in T4, the 3rd to get the operand address in T5, and the 4th cycle to get the operand in T6).

Table 2: Instruction Fetch Cycle

State	Description	RTL Notation
T_0	Load the address register AR with contents of register OutA	**T_0**: AR←OutA
T_1	Read the memory location of address in AR, store the value in OutD	**T_1**: OutD←M[AR]
T_2	- Load AR register with content of program counter register PC - Increment PC if program is still running, i.e., stop bit S is zero. PC now points to either: - the next instruction, if the current instruction is only 8b - the second byte of the current instruction, if it's a 16b	**T_2**: AR←PC **T_2S'**: PC←PC+1
T_3	Read instruction's opcode	**T_3**: IR←M[AR]
T_4	This state is a delay that allows the opcode to be decoded by CU and stored in X and Y registers	(nothing)
T_5	If X_2, the byte in IR is a register reference instruction - execute the instruction according to Table3 - Reset sequence counter SC	**T_5X_2**: execute instruction - Table3 **T_5X_2**: SC←0
	If X_0 or X_1, the byte in IR is a memory reference instruction. Read the second byte of the instruction. Firstly, move the address from PC to AR.	**T_5IR_7'**: AR←PC
	Increment PC if program is still running, i.e., stop bit S is zero. PC now points to the opcode of the next instruction.	**$T_5IR_7'S'$**: PC←PC+1
T_6	Read the second byte of the instruction from memory location at address AR, M[AR], and move the value to **AR**	**T_6 IR_7'**: AR←M[AR]
T_7	If direct load instruction, LDD, move **AR** to **DR** and reset SC	**T_7Y_7**: DR←AR
	If indirect addressing mode, and not LDD command, read the operand address from M[AR]	**T_7 X_1**: AR←M[AR]
	If direct addressing mode, and not LDD command, do nothing, to align with indirect addressing mode.	**T_7 X_0**: nothing
T_8	If direct load instruction, LDD, move **DR** to **AC** and reset **SC**	**T_7Y_7**: AC←DR **T_7Y_7**: SC←0
T_8-T_{12}	Execute the memory reference instruction according to Table 4	(see Table 4)

Table 3: Register Reference instructions Execution Cycle

Instruction Symbol	RTL Notation
CLA	**T_5 X_2IR_0**: AC←0
CMA	**T_5 X_2IR_1**: AC←AC'
ASL	**T_5 X_2IR_2**: AC← ASL(AC)
ASR	**T_5 X_2IR_3**: AC← ASR(AC)
INC	**T_5 X_2IR_4**: AC← AC+1
HLT	**T_5 X_2IR_5**: S← 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

Lab Preparation

Files Analysis

You will start up by analyzing the schematic files (.bdf) which you are provided with. You can do it either by using Quartus II. You don't have to understand in detail the RAM operation (ram256x8). The diagram of the 4 bit SC counter has the same architecture like the 8 bit counter, but truncated to 4 bits. Finally, the VHDL code of the bus multiplexer. 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).

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 data-path (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):

<u>Memory ($\lambda_M(In, T)$)</u>

1. memwrite

<u>CPU registers ($\lambda_R(In, T)$)</u>

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

<u>CPU ALU ($\lambda_{ALU}(In, T)$)</u>

12. ALU_Sel2
13. ALU_Sel1
14. ALU_Sel0

<u>Bus (data mux ($\lambda_{Bus}(In, T)$)</u>

15. BusSel2
16. BusSel1
17. BusSel0

<u>Control Unit ($\delta(In, T)$)</u>

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 **Xi** and **Yj** signals (as described earlier) 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 activate. 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 (λ Bus(In, T)).

The Program

To test your design we are going to use a small program and check it is working properly!

The program is evaluating 5 elements of the Fibonacci sequence and storing all elements in an array, and keeping the sum of all elements. The first two elements of Fibonacci sequence are 1, 1. The following elements are the sum of the previous two elements. That means out sequence is 1,1,2,3,5,8,13.

The Java-like program code is shown below.

```
byte a = 1;
byte b = 1;
byte c;
byte sum = 0;
byte cnt;
byte indx = 0;
byte F[] = new [8];
F[indx] = a;
indx++;
F[indx] = b;
indx++;
sum = a + b;
for (cnt=0; cnt<5; cnt++) {
   c = a+b;
   a = b;
   b = c;
   F[indx] = c;
   indx++;
   sum = sum + c;
}
```

Since we don't have compiler or OS, we have to define the variables addresses manually. Like OS, we leave the top of the memory for program code, and we allocate at the bottom of the memory. Here's the list of variables and their addresses:

Variable	Address	Address in binary
a	0XF0	**11110000**
b	0XF1	**11110001**
c	0XF2	**11110010**
sum	0XF3	**11110011**
cnt	0XF4	**11110100**
indx	0XF5	**11110101**
F	0XF8	**11111000**

The processor assembly code and its corresponding machine code is shown below.

Instruction	I	Arg	Comment	Addr	opcode	Arg
CLA			AC = 0	0	10000001	
STA		sum	sum = 0	1	00001000	11110011
INC			AC = 1	3	10010000	
STA		a	a = 1	4	00001000	11110000
STA		b	b = 1	6	00001000	11110001
LDD		11111011	AC = -5	8	00000000	11111011
STA		cnt	cnt = -5	10	00001000	11110100
LDD		11111000	AC = &F[0]	12	00000000	11111000
STA		indx	indx = & F[0]	14	00001000	11110101
LDA		a	AC = a	16	00000100	11110000
STA	I	indx	F[0] = a	18	**01001000**	11110101
ADD		b	AC = a+b	20	00000010	11110001
STA		sum	sum = a+b	22	00001000	11110011
LDA		indx	AC = indx	24	00000100	11110101
INC			AC++	26	10010000	
STA		indx	Indx++ === &F[1]	27	00001000	11110101
LDA		b	AC = b	29	00000100	11110001
STA	I	indx	F[1] = b	31	**01001000**	11110101
LDA		indx	AC = indx	33	00000100	11110101
INC			AC++	35	10010000	
STA		indx	Indx++ === &F[2]	36	00001000	11110101
LDA		a	AC = a	38	00000100	11110000
ADD		b	AC = a+b	40	00000010	11110001
STA		c	c = a + b	42	00001000	11110010
STA	I	indx	F[indx] = c	44	**01001000**	11110101
LDA		indx	AC = indx	46	00000100	11110101
INC			AC++	48	10010000	
STA		indx	Indx++	49	00001000	11110101
LDA		b	AC = b	51	00000100	11110001
STA		a	a = b	53	00001000	11110000
LDA		c	AC = c	55	00000100	11110010
STA		b	b = c	57	00001000	11110001
LDA		sum	AC = sum	59	00000100	11110011
ADD		c	AC = sum + c	61	00000010	11110010
STA		sum	sum = sum + c	63	00001000	11110011
ISZ		cnt	cnt++	65	00100000	11110100
BUN		loop		67	00010000	00100110
HLT				69	10100000	

- The program starts with variable initialization.
  - **Sum** = 0,
  - **a** =1,
  - **b** =1,
  - **cnt** =-5,
  - **F** = 0xF8, start address of output array
- Elements **a** and **b** are stored in F[0] and F[1].
- Sum is updated to **sum = a + b**.
- Loop of 5 iterations to calculate the next 5 elements of Fibonacci

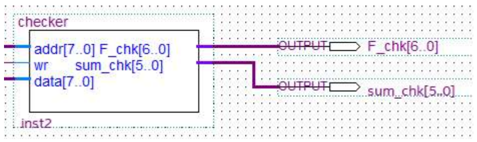
Browse the assembly code and get familiar with it. The program will take around 400 clock cycles to execute!

The Checker

A Verilog checker is monitoring the memory access and detects:

1. When the Fibonacci elements are stored correctly into the array
2. When the Sum is updated to a new value.

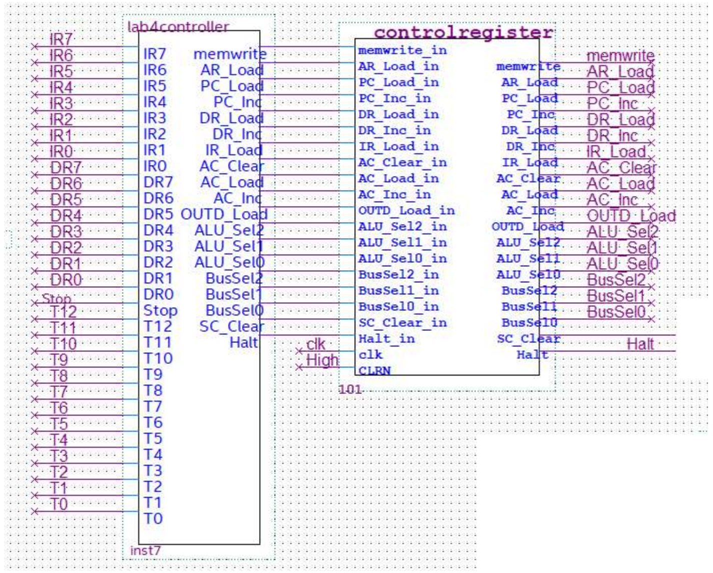
Signals F_chk[6..0] and sum_chk[5..0] are used to check these events



Your Task

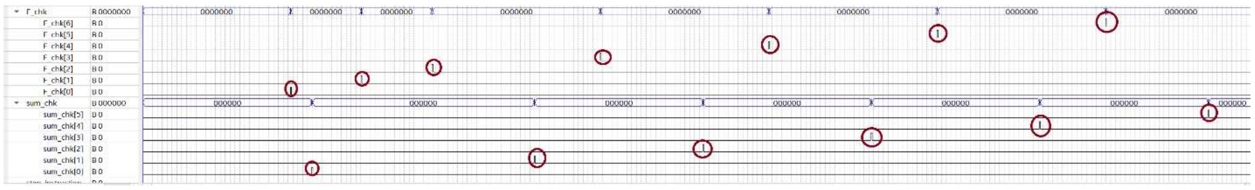
A complete project of the CPU is provided to you that implements the CPU design of Figure1.

The control part of the CU, namely the lab controller, has only input and output pins, but no logic. This combinational circuit provide control to the CPU.


lab4controller
controlregister
IR7 IR6 IR5 IR4 IR3 IR2 IR1 IR0
DR7 DR6 DR5 DR4 DR3 DR2 DR1 DR0
Stop
T12 T11 T10 T9 T8 T7 T6 T5 T4 T3 T2 T1 T0
memwrite AR_Load PC_Load PC_Inc DR_Load DR_Inc IR_Load AC_Clear AC_Load AC_Inc OUTD_Load ALU_Sel2 ALU_Sel1 ALU_Sel0 BusSel2 BusSel1 BusSel0 SC_Clear Halt
memwrite_in AR_Load_in PC_Load_in PC_Inc_in DR_Load_in DR_Inc_in IR_Load_in AC_Clear_in AC_Load_in AC_Inc_in OUTD_Load_in ALU_Sel2_in ALU_Sel1_in ALU_Sel0_in BusSel2_in BusSel1_in BusSel0_in SC_Clear_in Halt_in clk CLRN
clk
High
inst7
101

Your task is as follows:

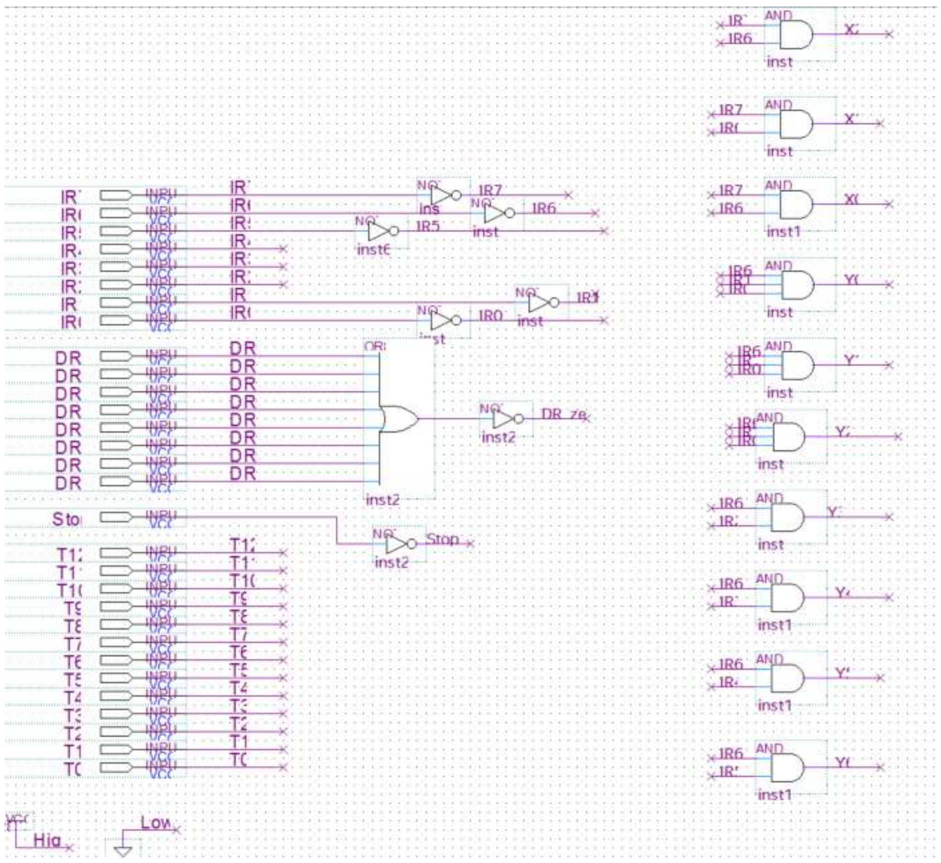
1. Fill the spreadsheet of control signals to be derived by the control unit as function of the trigger variable X_i, Y_j, T_k, i=0..2, j=0..7, k=0..12
2. List the Boolean expression of each of these functions, namely:
    a. BusSel0..2
    b. ALU_Sel0..2
    c. Memwrite
    d. AR Load
    e. PC Load
    f. PC INC
    g. DR Load
    h. DR INC
    i. IR Load
    j. AC Clear
    k. AC Load
    l. AC INC
    m. OUTD Load
    n. SC Clear
    o. HALT
3. Complete the schematic file for lab4_controller.bdf based on your derivation of the Boolean expressions in step 2.
4. Compile your project, fix all compile issues
5. Run simulation to demonstrate correct operation of the control unit
    a. A waveform file Test1.vwf is provided for demonstration.
    b. You need to show all checker signals triggered as shown in figure below.
6. Clean your project and submit the report and the design project source in one zip file to Brightspace.
7. Your Lab report should include the following:
    a. The spreadsheet containing all control signal triggers and Boolean expressions
    b. The Boolean expressions derived for all the control signals in step 2
    c. Screenshot of your Lab4_controller.bdf schematic
    d. Simulation waves showing correct behavior of the CPU
        i. You must show the checker signals triggered for all events



Notes:

1. This is a long assignment, so start early and give yourself time to finish in time
2. 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. Here's a peek of my solution: all connected through the wire names! (Note: this is old image, may not apply here)



Bonus Task

Demonstrate your design is working on the test board available in the lab. *Note that you need to modify the program to end with an endless loop, e.g., CLA, BUN; instead of ending with HALT. Show that all the expected values in F[0]..F[4] are as expected, namely, 1, 1, 2, 3, 5, 8, 13*

- 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
- Program the DE2-115 board by opening the programmer and choose "Program". Use the DIP switches to choose the addresses for the data.
- Enter the program in memorycontents8.mif. Program the device and use the DIP switches to read the memory content. Was your analysis of the program correct? Demonstrate to your TA.

The expected Address/Value are as follows:

```
(addr == 8'hf8) && (data == 8'h01)
(addr == 8'hf9) && (data == 8'h01)
(addr == 8'hfA) && (data == 8'h02)
(addr == 8'hfB) && (data == 8'h03)
(addr == 8'hfC) && (data == 8'h05)
(addr == 8'hfD) && (data == 8'h08)
(addr == 8'hfE) && (data == 8'h0D)
```