Royal Institute of Technology (KTH)
School of Electrical Engineering and
Computer Science (EECS)
Department of Electronics (ELE)

Version 1.5
Last revision 2020-09-15

Laboration 4

IL1331 VHDL Design

IL2203 Digital Design and Validation using HDLs

The One-Chip Computer

Student Name:……………………………………………………….

Personal Number:………………………………………………….

Date of Approval:………………………………………………….

Assistant:…………………………………………………………….

Laboration 4

The One Chip Computer

In this laboration, we finish the control logic of our Microcontroller (a tiny CISC CPU) and connect it to its surrounding environment. We complete the Microcontroller ROM and include R/W_n signals to control reading/writing from/to external memories. We also build a General Purpose IO unit for writing data to LEDS, and write the program (assembly code) that should be stored in the external memory. We then build a testbench to run the system with the assembly code in it to test it. Finally, we download the Microcontroller component on the prototype FPGA board and verify that it works.

Preparations

1) Create an external memory using the Quartus Megawizard plug-in manager. It should have 256 positions (8-bit address), 16 bits data, one read enable and one write enable inport. The outport Q should not be registered. The steps are shown on the slides in lecture F7.
2) Write down the Hex-code for the program in Appendix B.
3) Think through carefully which test patterns you need to toggle all wires in the final design. After testing that the CPU is properly connected, you should store the hex-code of the program from Appendix B.

Tasks

One of the crucial design steps is integration. Once all sub-components are finished, we have to put them together into a working design. However, every time we add another level of hierarchy in the design, we have to adapt the lower level components somewhat so that they can be reused at the next level. This also leads also to new constraints on testing, and we often have to rewrite the test benches and redo the tests we have already done on the lower level components.

Task 1. Complete the Microcontroller FSM and add a R_Wn signal (or use two separate signals, RDEN and WREN) for controlling accesses to an external external memory.

1) Complete the CPU by adding the signal R_Wn to the Microcontroller FSM for controlling external components. It should indicate when external components are being read (active high) or written (active low).
2) Modify the testbench for the Microcontroller/CPU, and test that the new signal is working properly for the LD and ST-instructions.

Task 2. Create the external memory.

1) Create the external memory according to the preparation instructions. You should specify that you use a memory initialization file called memory.mif or memory.hex. Example of the two formats are given in appending C. Also, allow the In-System Memory Content Editor to capture and update content independently of the system clock. The latter is important during debugging. Name the memory instance ID RAM.

Task 3. Create the assembly code program

1) Create a package for coding the assembly instructions from lab4 so that you can program the CPU. The assembly instructions are shown again in Appendix A.

Declaring constants for the mnemonics and register names, and concatenating them together to instructions is a good way to create readable instructions. Make sure that you also declare constants for the unused bits, e.g.:

ADD & R1 & R2 & R3 & nu3 -- not_used_3_bits (e.g. “000”)
ST & nu3 & R1 & R2 & nu3
etc.

2) One nice thing that Modelsim can provide us with are the hex-codes for the instructions. Make a fake memory architecture. This fake memory architecture should only be used during initial simulations, to derive the bit patterns that should be stored in the real memory. Write down the assembly program outlined in Appendix B and store it in the fake memory architecture at the addresses 0-14.

```
architecture fake of memory is
   signal RAM:program(0 to 255):=(
      (LDI & R5 & B"1_0000_0000"),
      (ADD & R5 & R5 & R5 & nu3),
      …
      (BRA & X"000"),
      others=>(NOP & R0 & R0 & R0 & nu3));

   signal ...
begin

   ...

end fake;
```

3) Start the simulation. Select radix hexadecimal for the RAM signal. Modelsim will convert your assembly program to hex-codes. Look on the RAM signal in the simulation window of Modelsim when you test the fake memory architecture, and write down the hex-codes in the memory.mif file.
4) From now on, you should use the generated memory in your simulations so you get the memory latency right.

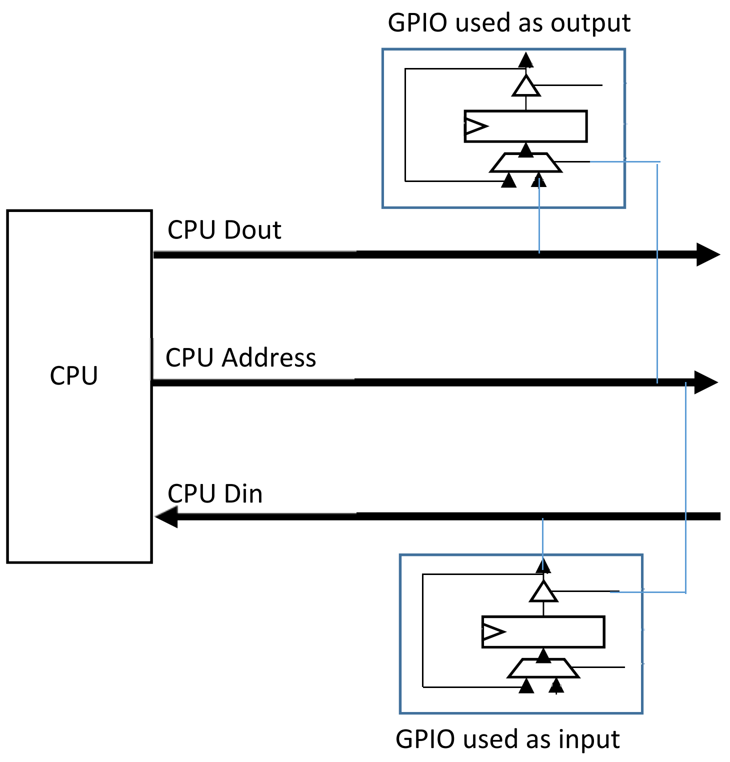
Task 4. Create a General Purpose IO component.

1) Create a General Purpose IO component according to the specification in Appendix D. The component should be possible to use both as an input and/or an output component, depending on which way it is connected. Input components connect their IO side to the IO switches and the output to the CPU DIN bus. Output components should be connected with their Inputs side to the CPU Dout bus, and the IO side to the LEDs.

Task 5. Create the Computer according to the specification in Appendix E.

1) Connect the external memory to the CPU at addresses H’0000-H’00FF. Use the full address from the CPU and generate the wren and rden signals required by the memory and make them react accordingly to the signal R_Wn, when the address is in the given range.
2) Connect the external GPIO as an output for the CPU at address H’F000. Code the wren and rden signals to react accordingly.
3) Simulate the Computer using the fake memory architecture and make sure that the program works as expected.

4) Connect the GPIO to the LEDs and synthesize the Computer using Quartus. Use the generated memory from Quartus instead of the fake one used during simulations.
5) Download the design to the FPGA board. Reset and run it. Make sure that the LED output matches the GPIO output that you got during simulations.
6) Make sure that the memory content has been properly initiated during synthesis by checking the contents using the Memory content editor. The addresses 0-12 should contain your program. If not included properly, you can download the file into the memory using the memory content editor. You can also manipulate memory positions manually.
7) Open the SignalTapII Logic Analyzer tool in Quartus. Connect the Quartus Logic Analyzer to display the data going on the CPU databus to and from the memory. Set the trigger to when the reset signal is released (i.e., falling edge). Re-synthesize the design. Record the databus traffic to/from the memory. Make sure that the timing of the recorded memory traffic corresponds with the simulated one. The steps are also shown on the slides from lecture F7.



Passing Requirements

To pass the lab, the student should be able to show the assistants:

- Well-documented, well-commented and proper indented VHDL-models.
- Simulation results and waveforms in Modelsim.
- A functioning prototype on the FPGA board.
- Answer any questions that the assistants may have during the lab examination.

Appendix A – CPU Instruction list

Instruction	I_{15}	I_{14}	I_{13}	I_{12}	$I_{11}I_{10}I_9$	$I_8I_7I_6$	$I_5I_4I_3$	$I_2I_1I_0$	Explanation
ADD r1,r2,r3	0	0	0	0	R1	R2	R3	N.U.	r1=r2+r3 pc=pc+1
SUB r1,r2,r3	0	0	0	1	R1	R2	R3	N.U.	r1=r2-r3 pc=pc+1
AND r1,r2,r3	0	0	1	0	R1	R2	R3	N.U.	r1=r2 AND r3 pc=pc+1
OR r1,r2,r3	0	0	1	1	R1	R2	R3	N.U.	r1=r2 OR r3 pc=pc+1
XOR r1,r2,r3	0	1	0	0	R1	R2	R3	N.U.	r1=r2 XOR r3 pc=pc+1
NOT r1,r2	0	1	0	1	R1	R2	N.U.	N.U.	r1=NOT(r2) pc=pc+1
MOV r1,r2	0	1	1	0	R1	R2	N.U.	N.U.	r1=r2 pc=pc+1
NOP	0	1	1	1	N.U.	N.U.	N.U.	N.U.	pc=pc+1
LD r1,<r2>	1	0	0	0	R1	R2	N.U.	N.U.	r1=mem<r2> pc=pc+1
ST <r1>,r2	1	0	0	1	R1	R2	N.U.	N.U.	mem<r1>=r2 pc=pc+1
LDI r1,data	1	0	1	0	R1	Data			r1=Sign Extended Data pc=pc+1
Not Used	1	0	1	1	N.U.	N.U.	N.U.	N.U.	Reserved for future use pc=pc+1
BRZ <offset>	1	1	0	0	Offset				Z=1=> pc=pc+S.E.offset Z=0=> pc=pc+1
BRN <offset>	1	1	0	1	Offset				N=1=> pc=pc+S.E.offset N=0=> pc=pc+1
BRO <offset>	1	1	1	0	Offset				O=1=> pc=pc+S.E.offset O=0=> pc=pc+1
BRA <offset>	1	1	1	1	Offset				pc=pc+offset

N.U. = Not Used
S.E. = Sign Extended

Appendix B – Test program

The program memory should have the following content:

Address	Content
0	LDI R5,256 ; R5=FF00
1	ADD R5,R5,R5 ; R5=FE00
2	ADD R5,R5,R5 ; R5=FC00
3	ADD R5,R5,R5 ; R5=F800
4	ADD R5,R5,R5 ; R5=F000
5	LDI R6,H'20
6	LDI R3,3
7	ST R6,R3
8	LDI R1,1
9	LDI R0,H'E
A	MOV R2,R0
B	ADD R2,R2,R1
C	SUB R0,R0,R1
D	BRZ H'03 ; PC+3
E	NOP
F	BRA H'FC ; PC-4
10	ST R6,R2
11	ST R5,R2 ; PIO=R2
12	BRA H'00 ; PC+0
13	NOP
14	NOP

Appendix C - Memory Interchange Format (.mif)

```
-- First comes a lot of header lines containing the Altera disclaimer information
-- etc.
-- Quartus II generated Memory Initialization File (.mif)

WIDTH=8;
DEPTH=8192;

ADDRESS_RADIX=UNS;
DATA_RADIX=UNS;

CONTENT BEGIN
           0    :  0;
           1    :  1;
           2    :  2;
           [3..8191]  :  0;
END;
```

Hex-file format

The format of each line in a .hexfile is as follows: : AABBBBCCDD…DDEE

`AA is the number of bytes in the data field DD…DD`

`BBBB is the starting address`

`CC is the` type (00 = data, 01 = end of file, 02 = address offset)

`DD…DD is the data field`

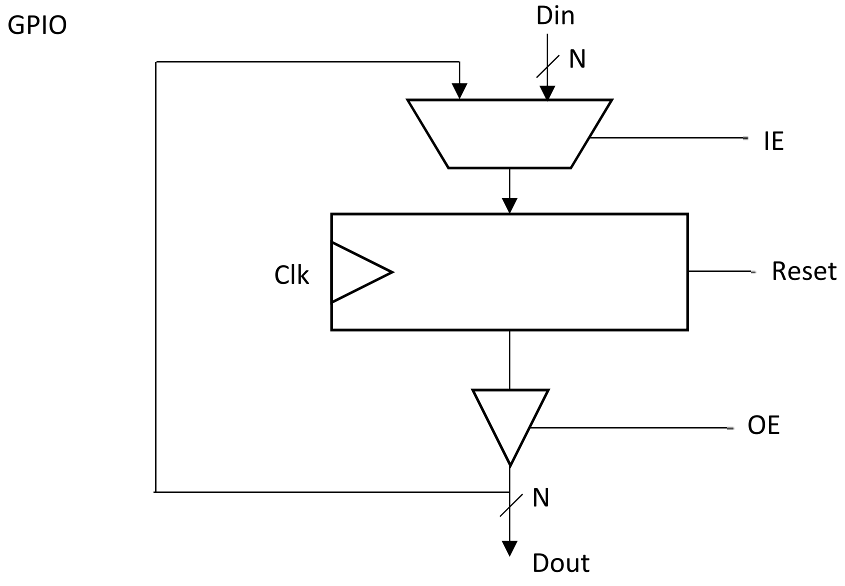
`EE is a checksum value`

Due to the complexity of the Hexadecimal (Intel-Format) File type, Intel recommends that you use the Quartus® Prime Memory Editor to create .hex files:

https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_hexfile.htm

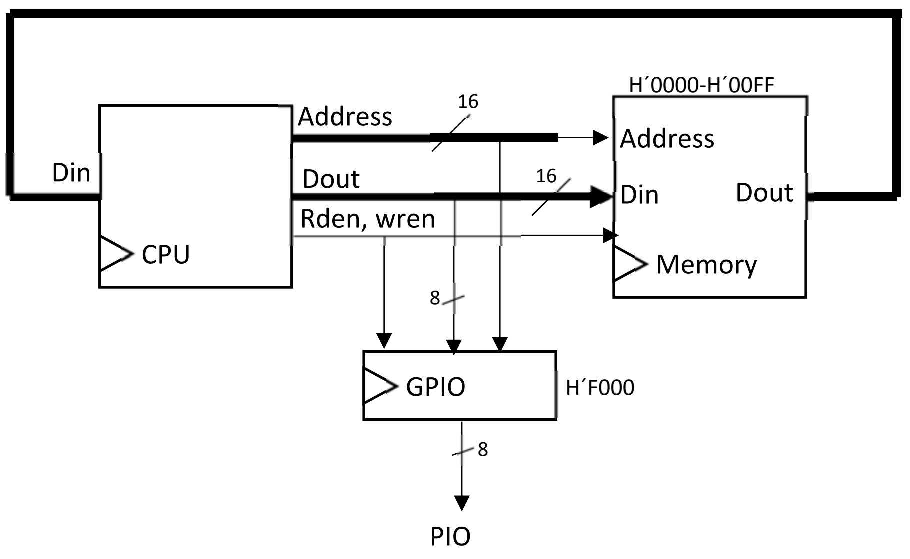
Appendix D – GPIO Specification

The GPIO is an addressable register. A truly general HPIO has one configuration register that decides the direction of the inputs. However, in this lab we will only use it as an output, so the configuration register is not needed.

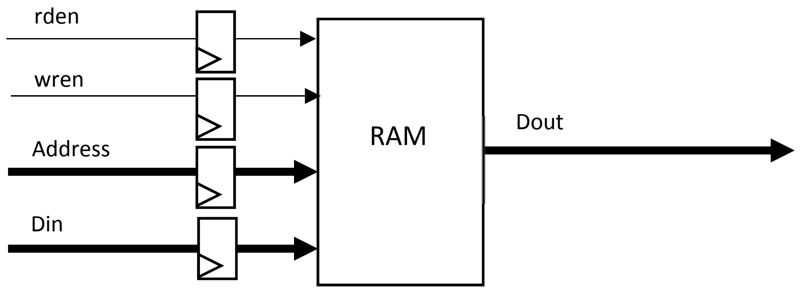


Appendix E – One-Chip Computer Specification

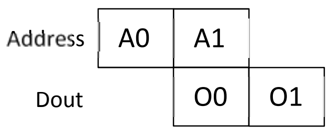
One-Chip Computer



Generated RAM



Timing


Address
A0
A1
Dout
O0
O1