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

Version 1.5
Last revision 2020-09-08

Laboration 3

IL1331 VHDL Design

IL2203 Digital Design and Validation using HDLs

The Microcontroller FSM

Student Name:………………………………………………………..

Personal Number:……………………………………………………

Date of Approval:……………………………………………………..

Assistant:……………………………………………………………….

Laboration 3

The Microcontroller FSM

In this laboration, we build the first version of the control logic of our Microcontroller (a tiny CISC CPU). We complete the Datapath from lab2 to include a Program Counter and a bypass mux for jump/branch instructions. We then create the Microcontroller ROM for issuing microcode instructions for controlling the datapath. We then build a testbench to test it. Finally, we download the Microcontroller component on the prototype FPGA board.

When you design a system, the most common source of errors is systematic errors. Systematic errors are caused mainly by 1) errors in the specification; 2) misinterpreting the specification or 3) bugs in the code. Timing errors, i.e., when things should happen or should be done, and missing signals/registers are common. In this lab, there are two inconsistencies/bugs in the specification, one regarding the bypass mux:es and one regarding the Datapath from lab 2 and the handling of status-flags. You should document what the inconsistencies/faults are and find a remedy.

Preparations

1) Complete the Datapath you designed in Lab2 by adding a bypass mux for jump/branch instructions according to the specification in Appendix A. Also, replace the Comp(A) instruction in the ALU with a Mov(B) instruction.
2) Figure out if/where the specification is wrong and add missing signals/registers.
3) Create the Microcontroller FSM so that it performs the instructions in the instruction list in Appendix B. The most compact form of controller FSMs is to code it as a ROM, but a standard CTRL is also ok.
4) Connect the datapath with the Microcontroller FSM (ROM/Ctrl) according to the specification in Appendix C.
5) Think through carefully which test patterns you need to toggle all wires in the final Microcontroller, together with the instructions you need to run the test. Write them down in a table. This table will serve as your test protocol.

Tasks

Task 1. Complete the Datapath and add bypass Muxes for Jump/Branch instructions

The Datapath has as its task to perform all arithmetic and logic operations in the computer. It contains the RF, the ALU, and input Mux, and an output register. In addition to performing all operations of the Microcontroller/CPU, it also has to perform Jump/Branch instructions. The Program counter is either part of the registers in the register file (typically the last one), but it can also be a separate PC register, external to the register file. In our case, we will not be incrementing the PC externally, but use the last register (reg 7 in our case) for the PC. Branching is performed by adding the offset of the branch instructions to the current Program counter.

1) Add a bypass mux to the input B of the ALU according to the connection diagram in Appendix A. Change the Zero-function of the RTL model of the ALU from Lab2 to an Increment by one function (Incr A).

2) Modify the testbench for the Datapath, and test the new bypass and increment function.

Task 2. CISC CPU/Microcontroller

1) Create a package for creating the microcode instructions to the datapath. Make sure that you can control every bit.
2) Create the microcontroller ROM according to the functionality in Appendix B. Add Register Enable bits for the registers shown in the CPU if it is needed.
3) Connect the microcontroller ROM with the datapath and add the registers and muxes required according to the specification in Appendix B. If you think it is easier, you may remove the output registers from the ALU, but please remember that the Z,N,O-flags needs to be saved in a register before they are used as inputs to the microcontroller ROM.
4) Write a testbench that test the CPU. All instructions should be covered in the testbench.
5) Synthesize the CPU and make sure that the VHDL code is synthesizable and that there are no latches in the design.

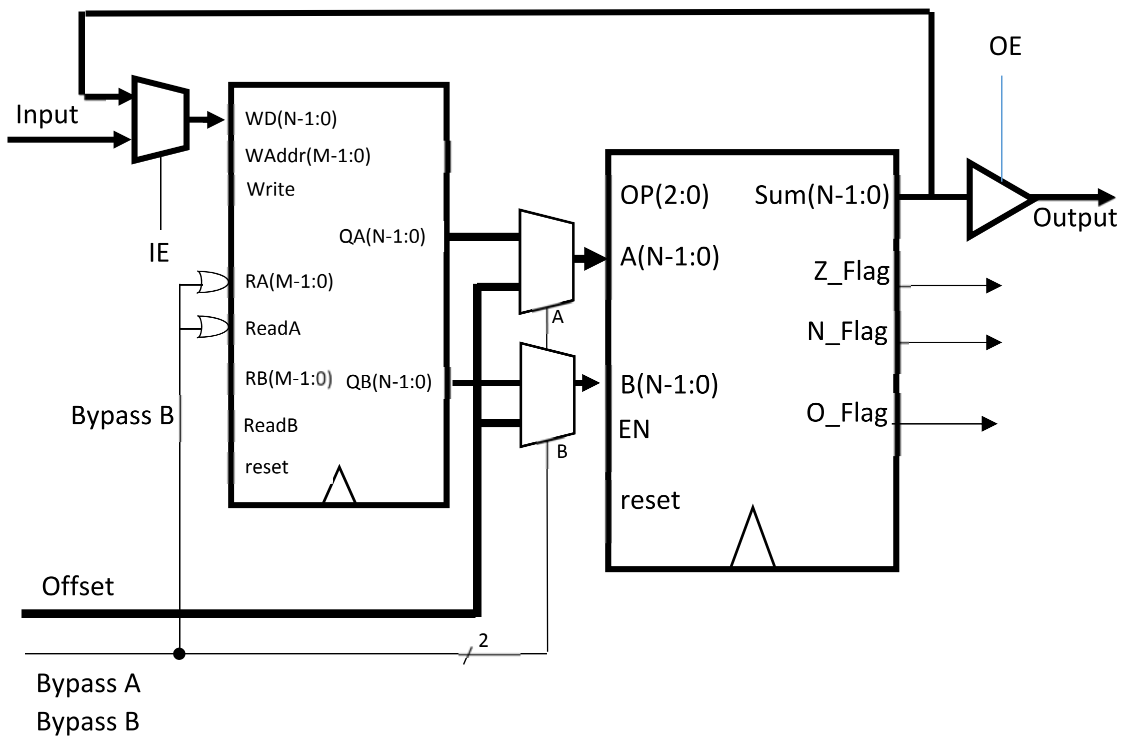
Passing Requirements

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

- A list of missing signals/registers and/or errors in the specification.
- 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.
- When the lab assistant has approved the lab, he/she will sign the front page of it. Keep that document until the course has ended and your grade has been registered in Ladok. It is the only proof you have that you have made the lab.

Appendix A - The Complete Datapath

The Complete Datapath performs all ALU operations from Lab 1 and 2, except for the Zero function. It should be replaced with an Incr A function instead. In addition, a bypass function should be added. The bypass function should take the external input Offset and forward it to the A-input of the ALU. In addition, the bypass B signal should set the ReadA enable to '1' and the Read Address A signal to all ones, to force the corresponding output (A) to become the Program Counter. The LDI instruction should use Bypass A to forward the Data to the A-input of the ALU. Data and Offsets should be sign extended according to their resp. bit lengths.



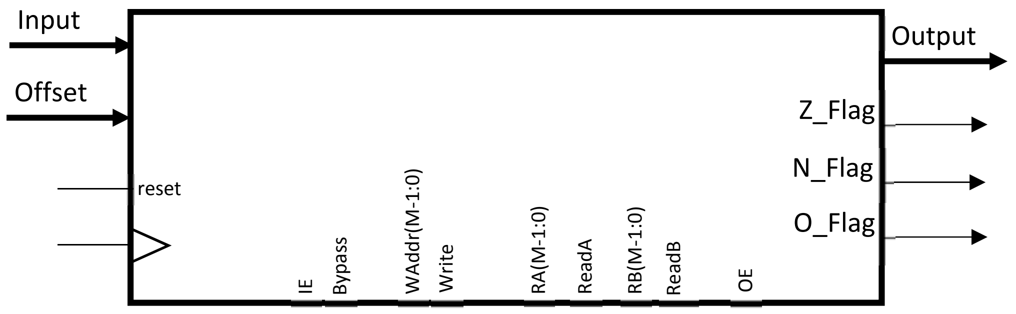


Figure 1. Block diagram – Complete Datapath

Appendix B – 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	N.U.	R1	R2	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

The microcontroller ROM should use the upper four bits of the instruction ($I_{15}..I_{12}$) together with a uPC (counting 0-3), and one input bit for the status-flags, as an address. One MUX should be connected externally for selecting the correct flag. The ROM should output the control signals needed to set the bits in the data path so that all instructions work as specified. Offset/Data should be sign extended before usage. LDI uses a 9-bit offset. Branch-instructions uses a 12-bit offset.

The microcode for the instructions should go through a four stage FSM. The states are shown in Figure 2 below.

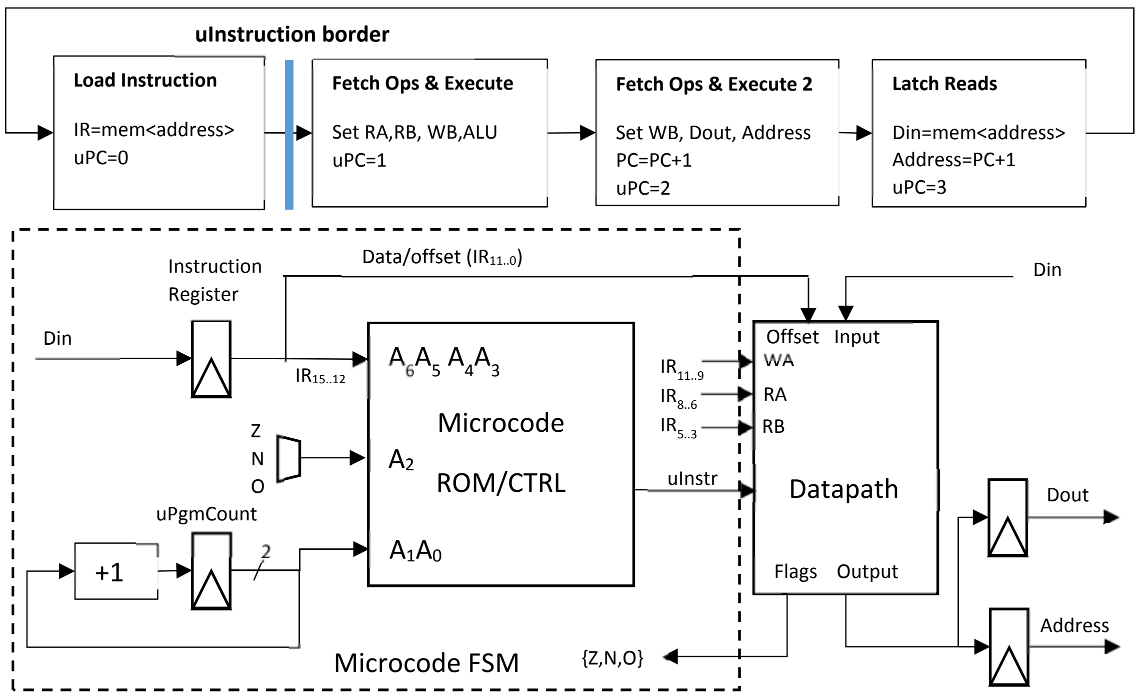


Figure 2. CISC CPU/Microcontroller block diagram

Appendix C - Microcode ROM coding style

The microcode ROM can be created in the style of a 1) a symbolic FSM, decoding the instructions and setting the ctrl signals, 2) decode the instructions and setting the ctrl signals using an EFSM, i.e., a symbolic FSM and a counter, or 3) write it directly using readable uInstructions in a ROM.

Pros & cons:

- A symbolic state machine is easy to write, but is hard to get a grip of if there are many states. It is easy to add more states if required, but it is hard to see if a signal is missing in one state. You do not need a uPC since it is encoded in the states.
- An EFSM is a little bit harder to write than a symbolic FSM. You create a symbolic state for each instruction, and have a counter (the uPC) to differ between the sub-tasks/micro-instructions of the instruction. It is easier to keep track of the symbolic states, but it can sometimes be tricky to control the counter, especially if some instructions require more sub-states than others. Also, it can be hard to see if a signal is missing in one state.
- Coding the ROM directly instead of letting the synthesis tool create it is the hardest to write, but easiest to debug. Since all signals are always present, you cannot miss to assign a value. Removing or adding a state is hard, since the number of sub-states/micro-instructions for each instruction is fixed, so all instructions have the same number of clock cycles.

As an example of coding as a ROM, the instruction protocols for the ADD and BRZ instructions are given in the table below. It should be noted here that the LI-state is actually the last state in an instruction since it registers the next instruction, but that it is on address (0) because this is the value that the uPC is reset to, and the first thing the processor has to do is to load a new instruction.

uPC (A_1A_0)	ADD – Flags (A_2='0')	ADD – Flags (A_2='1')
LI (0)	Enable IR='1'	Enable IR='1'

FO (1)	RA_enable='1'
	RB_enable='1'
	WA_enable='1'
	uInstr=ADD
	Latch_Flags='1'
		RA_enable='1'
		RB_enable='1'
		WA_enable='1'
		uInstr=ADD
		Latch_Flags='1'

EX (2)	(ALU perform ADD)
	uInstr=INCR
	WA_enable='1'
	BypassB='1' (Force PC on A)
	BypassW='1' (Force PC on WA)
	Address=ALU output
		(ALU perform ADD)
		uInstr=INCR
		WA_enable='1'
		BypassB='1' (Force PC on A)
		BypassW='1' (Force PC on WA)
		Address=ALU output

LM (3)	--	--

uPC (A_1A_0)	BRZ – Flags (A_2='0')	BRZ – Flags (A_2='1')
LI (0)	Enable IR='1' Select_Z_Flag='1'	Enable IR='1' Select_Z_Flag='1'

FE1 (1)	RA_enable='1'
	WA_enable='1'
	BypassB='1' (Force PC on A)
	BypassW='1' (Force PC on WA)
	uInstr=INCR A
	Address=ALU output
		RA_enable='1'
		WA_enable='1'
		BypassB='1' (Force PC on A)
		BypassW='1' (Force PC on WA)
		uInstr=ADD
		Address=ALU output

FE2 (2)	--	--
LM (3)	--	--

uPC (A_1A_0)	ST – Flags (A_2='0')	ST – Flags (A_2='1')
LI (0)	Enable IR='1'	Enable IR='1'

FO (1)	RA_enable='1'
	uInstr=MOVA
	Address=ALU output (R1)
		RA_enable='1'
		uInstr=MOVA
		Address=ALU output (R1)

EX (2)	RB_enable='1'
	uInstr=MOVB
	Dout=ALU output (R2)
		RB_enable='1'
		uInstr=MOVB
		Dout=ALU output (R2)

WA (3)	RA_enable='1'
	WA_enable='1'
	Bypass B='1' (Force PC on A)
	BypassW='1' (Force PC on WA)
	uInstr=INCR A
		RA_enable='1'
		WA_enable='1'
		Bypass B='1' (Force PC on A)
		BypassW='1' (Force PC on WA)
		uInstr=INCR A

In the lab, you should chose the FSM coding style that you feel most comfortable with when you design you Microcode ROM.