Lab 5 (RISC Processor) Report

Task:
Using Spartan-3 Starter Kit Board create a RISC processor. Minimum instruction set includes NOP, ADD, SUB, AND, NOT, LOAD, STORE, JMP, JMG (jump if greater), JMZ (jump if zero) commands. More instructions may be added if needed. Write a test program to verify processor's correctness. It may be useful to add additional circuitry to enable processor to output data on Seven-Segment LED displays or get it from Push Button or Slide Switches.

Theory and FPGA pin information:
RISC stands for Reduced Instruction Set Computer, it is a name given to a certain design philosophy. RISC does not stand for instruction set count reduction; it is about reducing the complexity. Cutting the number of instructions is one way to pull some complexity. Eliminating rarely-used instructions and making sure that remaining commands run really fast is only one technique among many.

CISC processors have MOV instruction to move data around. Data was moved either from register to memory, from memory to register, from register to register and from memory to memory. MOV instruction also had many different addressing modes and worked on different data sizes. Breaking down the MOV instruction became one of the core concepts of RISC. Instead of MOV, RISC uses two separate instructions - LOAD and STORE. One instruction is to load a register from memory, and another - to store a register to memory. All other operations are register to register. There are fewer addressing modes and the data size of the load and store is fixed (the entire register).
This concept is called the Load/Store architecture and it is common for all RISC machines.

Making the instruction set simpler conserves space, so it can be used to increase the performance further. One thing to help with performance is to add more general purpose registers. Most CISC instructions write the result back into one of their sources, due to the register shortage. This means that it is replaced with the new value. If you need the old one, then you should save a copy of it beforehand, thus loosing in performance. RISC usually has enough registers, three operand instructions are used. The result is stored in a separate register, so that later operands can be reused for something else. In many ways this makes things more efficient.

Another way to make things go faster is to allow more things to happen at once. Two execution units working at the same time (on independent things, of course) can get nearly twice as much work done. But this also puts a lot of pressure on the compiler, since now it must also be a "project manager" (to keep both sides busy).

It was learned that processors could be sped up if we brake down the instructions into stages. On each stage a certain part of that instruction is done and then it would advance to the next stage. This technique is called pipelining and it is widely used in RISC machines.

Imagine a processors instruction broken into four parts - fetch, decode, execute and write. In the first stage, the fetch unit does the fetch of the instruction. In the second stage, instruction advances to the decode unit, but now the fetch unit is available. We can fetch the next instruction at the same time. The processor is doing parts of two instructions at once now. In the third stage, instruction gets executed. Also the

instruction behind is getting decoded, and a new instruction is getting fetched. In the final stage any data that needs to be written back is stored. It takes four cycles to get the instruction done. But there are other instructions stacked in the pipeline. It takes only one additional cycle to complete the next instruction, instead of four, now. And there are two partially done instructions in the pipeline as well. Note that pipeline does not reduce the time for one instruction to get completed, but the overall time to get four instructions done is much faster now, compared to non-pipelined processor doing the same amount of work.

For the pipeline to work, each phase needs its own set of registers to hold the instruction and operands. This is necessary because each stage will be working on a different instruction. Inserting registers in between the different phases has a huge advantage: the clock period can be much shorter. We only need to allow for an instruction to make it through one stage, not all of them. So, at least in theory, the processor can be clocked faster.

Pipeline approach has its tradeoffs, though. These come in the form of stalls. For example, what if the next instruction requires the result of the previous one? This situation is called a data hazard. One way to overcome this problem is to stall the pipeline to wait for previous instruction to complete its execution. NOP instructions are inserted between these dependent commands. The downside is obvious – an increase in overall execution time. You can also try to execute instructions out of order. If further instructions do not rely on the awaited result and must be executed anyways, you can simply fill the stall with them, instead of NOP commands. This is not always possible, though. ALU forwarding is also an option in this situation, although it requires additional circuitry. The trick is to forward the ALU output back into the ALU.

When using Jump instructions, before it is executed and the program counter is updated with a new address, some instructions, which come directly after Jump command, have already entered the pipeline. These instructions will be executed as well. This problem is called a control hazard. As with the data hazards, pipeline stall may solve the problem. Adding NOP instruction after the Jump command corrects the error. If the following instruction is to be executed anyway, it may be left as it is and no pipeline stall is needed. This was taken into account while writing test program for the processor.

Table 1: LED Connections

LED	LD7	LD6	LD5	LD4	LD3	LD2	LD1	LD0
FPGA Pin	P11	P12	N12	P13	N14	L12	P14	K12

To light a LED the associate FPGA pin (Table 1) must be driven High (equals logic 1).

Table 2: Push Button Switches Connections

Push Button	BTN3 (User Reset)	BTN2	BTN1	BTN0
FPGA Pin	L14	L13	M14	M13

Pressing a Push Button Switch generates logic High (equals logic 1) on the associated FPGA pin (Table 2).

Table 3: Slide Switches Connections

Switch	SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0
FPGA Pin	K13	K14	J13	J14	H13	H14	G12	F12

When in the UP position, switch outputs logic High (equals logic 1) on the associated FPGA pin (Table 3).

A Low (equals logic 0) value lights the individual segment, a High (equals logic 1) turns off the segment (Table 4). Each individual character has a separate select control input: AN3, AN2, AN1 and AN0 (Table 5). The LED control signals are time-multiplexed to display data on all four characters. Present the value to be displayed on the segment control inputs and select the specified character by driving the associated select control signal Low (equals logic 0).

Table 4: Seven-Segment Display Control Signals Connections

Segment	FPGA Pin
A	E14
B	G13
C	N15
D	P15
E	R16
F	F13
G	N16
DP	P16

Table 5: Seven-Segment Display Select Signals Connections

Anode Control	AN3	AN2	AN1	AN0
FPGA Pin	E13	F14	G14	D14

Table 6: 3-bit Display Color Codes

Red (R)	Green (G)	Blue (B)	Resulting Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Table 7: VGA Port Connections

Signal	FPGA Pin
Red (R)	R12
Green (G)	T12
Blue (B)	R11
Horizontal Sync (HS)	R9
Vertical Sync (VS)	T10

Drive the R, G, and B signals High or Low to generate the eight possible colors (Table 6).

Note, that Sync outputs (Table 7) are inverted, they are normally tided to High (equal logic 1), and change to Low (equal logic 0) during the synchronization pulse.

Table 8: PS/2 Connections

PS/2 DIN Pin	Signal	FPGA Pin
1	DATA (PS2D)	M15
2	Reserved	—
3	GND	GND
4	Voltage Supply	—
5	CLK (PS2C)	M16
6	Reserved	—

The clock and data signals (Table 8) are only driven when data transfers occur, otherwise they are held in the idle state at logic High (equal logic 1). Both clock line and data line are bi-directional.

The Spartan-3 Starter Kit Board has a megabyte of fast asynchronous SRAM. The memory array includes two SRAM devices. The SRAM array forms either a single 256Kx32 SRAM memory or two independent 256Kx16 arrays.

Table 9: SRAM Control Signal

Signal	FPGA Pin
OE#	K4
WE#	G3

Both SRAM devices share common write-enable, output-enable and address lines (Tables 9, 13). However, each device has a separate chip enable, data IO lines and individual byte-enables (Tables 11, 12). Working modes are shown in Table 10.

Table 10: Truth Table of SRAM Control Signals

Mode	$\overline{WE}$	$\overline{CE}$	$\overline{OE}$	$\overline{LB}$	$\overline{UB}$	I/O0-I/O7	I/O8-I/O15
Not Selected	X	H	X	X	X	High-Z	High-Z
Output Disabled	H	L	H	X	X	High-Z	High-Z
	X	L	X	H	H	High-Z	High-Z
Read	H	L	L	L	H	DOUT	High-Z
	H	L	L	H	L	High-Z	DOUT
	H	L	L	L	L	DOUT	DOUT
Write	L	L	X	L	H	DIN	High-Z
	L	L	X	H	L	High-Z	DIN
	L	L	X	L	L	DIN	DIN

Table 11: SRAM IC10 Connections

Signal	FPGA Pin
IO15	R1
IO14	P1
IO13	L2
IO12	J2
IO11	H1
IO10	F2
IO9	P8
IO8	D3
IO7	B1
IO6	C1
IO5	C2
IO4	R5
IO3	T5
IO2	R6
IO1	T8
IO0	N7
CE	P7
UB	T4
LB	P6

Table 12: SRAM IC11 Connections

Signal	FPGA Pin
IO15	N1
IO14	M1
IO13	K2
IO12	C3
IO11	F5
IO10	G1
IO9	E2
IO8	D2
IO7	D1
IO6	E1
IO5	G2
IO4	J1
IO3	K1
IO2	M2
IO1	N2
IO0	P2
CE	N5
UB	R4
LB	P5

Table 13: Address Bus Connections

Address Bit	FPGA Pin
A17	L3
A16	K5
A15	K3
A14	J3
A13	J4
A12	H4
A11	H3
A10	G5
A9	E4
A8	E3
A7	F4
A6	F3
A5	G4
A4	L4
A3	M3
A2	M4
A1	N3
A0	L5

Spartan-3 FPGA device also features Block RAM. VHDL code to instantiate a Block RAM can be found in Language Templates.

50MHz clock oscillator source FPGA pin number is “T9”.

Work Description:
The purpose of this exercise is to introduce students to RISC design philosophy. The task does not feature a curtain requirements concerning realization of this circuit. Students are free to choose the ways and methods for completing this task. The final result of this exercise is a report (this document), VHDL source files with commentaries, test program code listing, plus a working demonstration of the hardware implementation.

Solution (VHDL):
For this RISC processor implementation the author will use VHDL. The processor described was inspired by the MIPS processor, a RISC (Reduced Instruction Set Computer) machine designed in 1984 at Stanford University. Processor incorporates a pipeline. The execution of an instruction is split into five stages:

IF Instruction Fetch stage fetches the next instruction from memory using the address in the PC (Program Counter) register and stores it in the first IR (Instruction Register). This stage takes two clock cycles to execute. First of all, content of the PC register is sent to the instruction memory. The instruction memory responds by sending instruction to the first IR register on the next clock cycle. Thus the Instruction Fetch step consists of two stages – IF 1 and IF 2, forming six in total.

ID Instruction Decode stage decodes the instruction in the first IR, sends it to the second IR, updates PC with next instruction’s address and reads any operands required from the register file.

EX The Execute stage executes the instruction. All Execution Block operations are done in this stage (addition, subtraction, compare, shift left or right, etc.). The instruction is passed to the third IR.

MA The Memory Access stage performs any memory or port access required by the current instruction. For LOAD or INPUT it would load an operand from memory or port. For STORE or OUTPUT it would store an operand into memory or port. All other instructions do nothing during this stage. The instruction is passed to the fourth IR.

WB For instructions that have a destination register the Write Back stage writes this result back to the register file. NOP, STORE and OUTPUT instructions do nothing during this stage.

It takes six clock cycles for one instruction to get through the pipeline (except for jump instructions, which are executed during the ID phase). The instruction itself has a fixed length and is 32-bit wide (Figure 1). The instruction format is also fixed. It consists of five fields – operation code field, two operand fields, result write back field and address field:

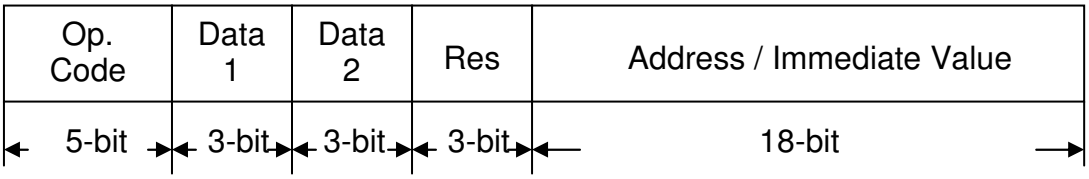

Figure 1: RISC Processor Instruction Format

In the Op. Code field an operation code is stored, this field is forwarded to every IR, because it determines Control Block outputs during ID, EX, MA and WB stages. Data 1 field contains the first operand address in register file; it is used only during ID stage and is not passed to the second IR. Data 2 field content depends on instruction. It is either the second operand address in register file, or shift operation power, or Port ID for INPUT and OUTPUT instructions. Field value is not required for WB stage, thus it is not passed to the fourth IR register. Res field contains the result register address in register file. Field value is used only during WB stage, thus it must be passed all the way through all IRs. Address / Immediate Value field content depends on instruction. It is either an address for MA stage or contains immediate value. Field value is not required for WB stage, thus it is not passed to the fourth IR register.

The block scheme in Figure 2 shows basic architecture of the processor, although some details are omitted (like Control Block outputs):

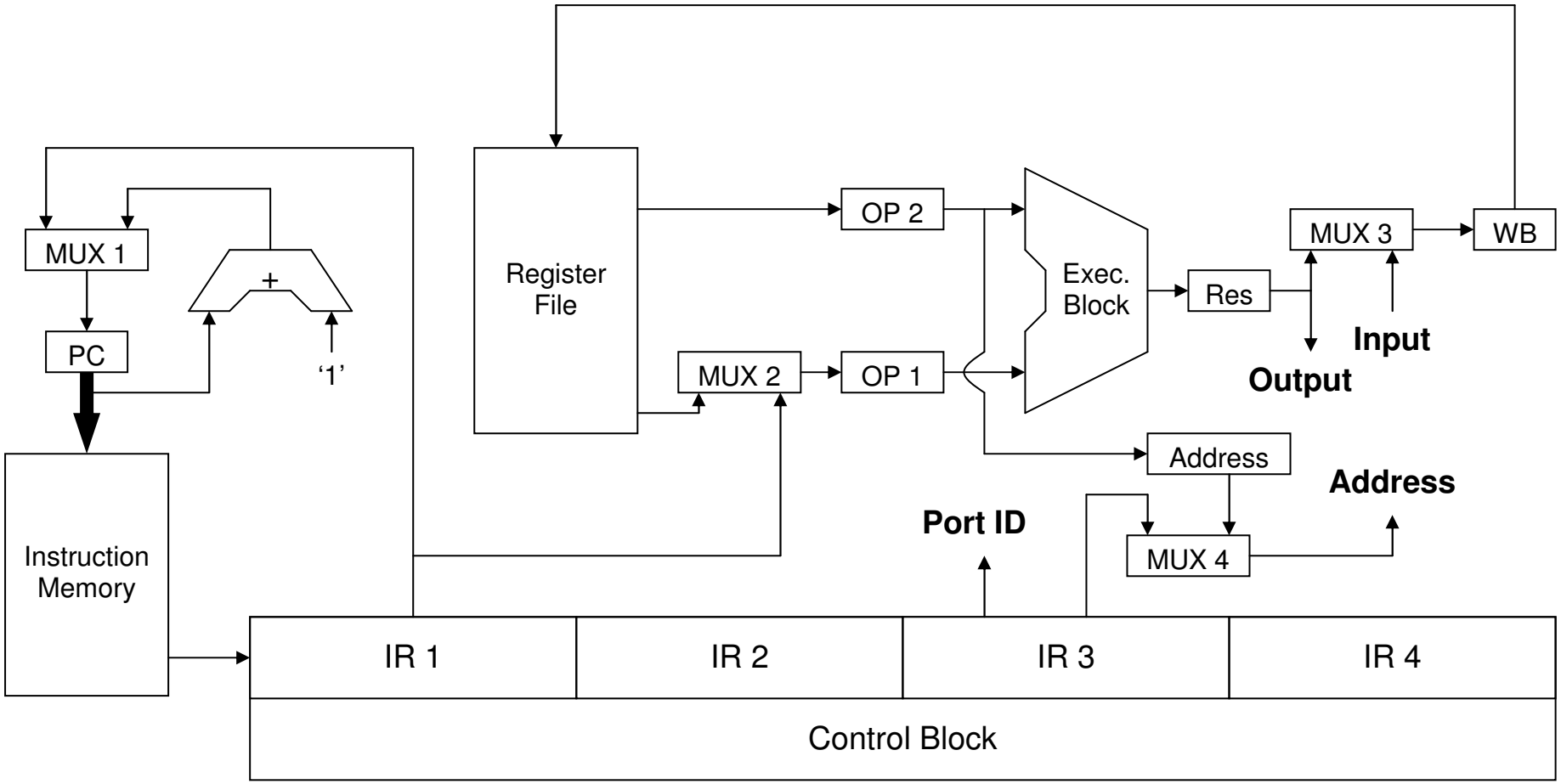

Figure 2: Hardware Implementation of RISC Processor

The program code is stored in the Instruction Memory. Instruction Memory is actually a ROM, containing 2048 32-bit instructions. PC register is 11-bit wide, it contains next instruction address. When PC register content is updated, it is sent to the instruction memory address line. Instruction memory responds by sending instruction to the first IR register on the next clock cycle. Instruction Memory is implemented using Block RAM. Accessing data from the Block RAM is a synchronous operation, thus it takes two clock cycles it total to load next instruction into first IR register. This process forms the IF stage.

Control Block consists of four IRs. These registers represent ID, EX, MA and WB stages of the pipeline. Their content is used to generate control signals. Every instruction must sequentially pass through all of IRs in order to get executed (except jump instructions). First IR is 32-bit wide and store whole instruction. Second and third IRs are 29-bit wide, Data 1 field is omitted in these registers. The fourth IR is 8-bit wide; it contains only the Op. Code and Res fields.

To fetch the next instruction, PC register must be updated with its address. Usually next instruction is situated right after the current one. By adding '1' to the PC content a new address is calculated. It is done with a dedicated adder. During jumps, however, PC register is updated with the address contained in the instruction itself. Multiplexer MUX1 is used to select PC register input source.

Register File is an array of eight 16-bit registers. It is used to store intermediate data. OP1 and OP2 registers are used to store operands needed for EX phase. They are loaded from the Register File; Data 1 and Data 2 fields of IR1 are used to select the operands. OP1 register can be also loaded with the immediate value stored in the instruction. Multiplexer MUX2 is used to select OP1 register input source.

This forms the ID stage. Control signals are generated by the Control Block depending on the Op. Code field in IR1.

Execution Block consists of three parts – ALU (Arithmetic-Logic Unit), Shifter and Comparator. It performs addition, subtraction, shift (Sifter power is selected with Data 2 field in IR2), compare, logic AND, OR, XOR and NOT operations. The result is stored in Res register.

There are two Flag Registers (Figure 3). One is affected by ALU and Shifter, second – by Comparator. Each Flag Register is 3-bit wide. Whenever Execution Block performs an operation, it also updates the corresponding Flag Register.

Figure 3: ALU and Shifter (on the left) and Comparator (on the right) Flag Registers

Sign - indicates whether result is positive or negative ('0' – positive, '1' – negative)
Zero - indicates whether result equals zero or not
Carry - indicates a carry (equals last shifted out bit for shift operations)
Equal - indicates whether OP1 equals OP2 or not
Great - indicates whether OP1 is greater then OP2 or not
Less - indicates whether OP1 is less then OP2 or not

This forms the EX stage. Control signals are generated by the Control Block depending on the Op. Code field in IR2.

OP2 register value is stored in Address Register. It is used for Register Addressing. A value stored in Register File can be used to access memory. However, register is only 16-bit wide, but memory address must be 18-bit long. Thus, Address Register defines sixteen least significant bits; two most significant bits are still taken from Address / Immediate Value field of IR3 to form a total of 18-bit memory address.

Processor supports Direct Addressing as well. The complete address is taken from the Address / Immediate Value field of IR3. Two most significant bits are common for both addressing modes. The rest is selected via multiplexer MUX4.

Res register output also serves as Data Output of the processor. Data can be forwarded either to memory or to port. Processor can obtain data as well, either from memory or port. Data Input is fed into multiplexer MUX3. Port is selected by Data 2 field of IR3.

This forms the MA stage. Control signals are generated by the Control Block depending on the Op. Code field in IR3.

WB register is used to store a value, which is to be written back to Register File. This value may come either from Res register or from outside of processor (from memory or port) via Data Input. Multiplexer MUX3 is used to select WB register input source. Then WB content is forwarded to Register File. Destination register is selected with Res field of IR4.

This forms the WB stage. Control signals are generated by the Control Block depending on the Op. Code field in IR4.

The Spartan-3 Starter Kit Board has a megabyte of fast asynchronous SRAM. The memory array includes two SRAM devices. One of them is used as Data Memory for storing data. It can store 256K of 16-bit words. SRAM sixteen I/O lines are bi-directional. We need to incorporate tri-state buffers. When reading, processor must disconnect from memory and let it drive the I/O lines. VHDL code in Listing 1 describes a tri-state buffer:

Listing 1: VHDL Description of Tri-State Buffer

```
MemIO <= IO_Out when WE = '0' else (others => 'Z');
```

The memory Write-Enable signal, generated by the processor, serves as a control signal for the tri-state buffer. When the signal is Low (equals logic 0), it is possible to write to memory. Thus the processor must drive bi-directional I/O pins. There is no need for processor to do so otherwise. Tri-state buffer's output is then set to 'Z' (High Impedance), enabling memory to drive I/O pins.

In this design processor has access to eight external ports. These are LEDs, Push Button and Slide Switches, Seven-Segment LED indicators, Keyboard controller, VGA controller (Address Register, Data Input and Data Output lines).

Push Button Switch and Slide Switch controllers just pass the momentary state of the switches, adding zeroes to form a 16-bit word, accepted by the processor. LED controller is a bit more complicated. It consists of an 8-bit register, which stores LED control signals. It uses the lower byte of processor's Data Output as an input. However,

a new value is assigned only if *Enable* signal goes High (equals logic 1). VHDL code in Listing 2 describes *Enable* signal generation:

Listing 2: Port Register Enable Signal Generation

```
process (Port_ID, Port_Write)
begin

if Port_ID = ID then
  Enable <= Port_Write;
else
  Enable <= '0';
end if;

end process;
```

Controller checks whether *Port ID* output of the processor matches its internal ID, then *Enable* signal is assigned a value of *Port_Write*, which is generated by the processor to indicate its intention of writing to the specified port.

A similar technique is used by Seven-Segment LED Indicators Controller. It consists of four registers (one for each LED display), which store Control Signals. It uses twelve lower bits of processor's Data Output as an input. Eight lower bits of the input are Control Signals; these are fed to all register. However, a new value is assigned only if corresponding *Enable* signal goes High (equals logic 1). VHDL code in Listing 3 describes *Enable* signals generation:

Listing 3: Seven-Segment Port Registers Enable Signal Generation

```
process (Port_ID, Port_Write, Input (11 downto 8))
begin

if Port_ID = ID then
  Enable1 <= Input (8) and Port_Write;
  Enable2 <= Input (9) and Port_Write;
  Enable3 <= Input (10) and Port_Write;
  Enable4 <= Input (11) and Port_Write;
else
  Enable1 <= '0';
  Enable2 <= '0';
  Enable3 <= '0';
  Enable4 <= '0';
end if;

end process;
```

Four higher bits of the input serve as select signals. They point to the LED indicators for which the Control Signals are intended. Controller checks whether *Port ID* output of the processor matches its internal ID, then *Enable* signals are assigned a value of *Port_Write* anded with corresponding select bit. Thus, a register can be overwritten only when both are High (equal logic 1).

Keyboard Controller is very similar to the one described in Lab 4 report, except now it only receives a scan code and lets the processor decide how to treat it. The output is formed from the acquired scan code and *NewScan* flag (the least significant bit), the

higher bits are filled with zeroes to form a 16-bit word, accepted by the processor. *NewScan* flag is used to distinguish newly acquired scan codes from the already read ones. When the new scan code is received, flag goes High (equals logic 1) and remains so until port is read by the processor, then it goes Low (equals logic 0).

The Video Controller is more complicated, than the one described in Lab 3 report. Figure 4 shows a block scheme, explaining the hardware implementation of Video Controller:

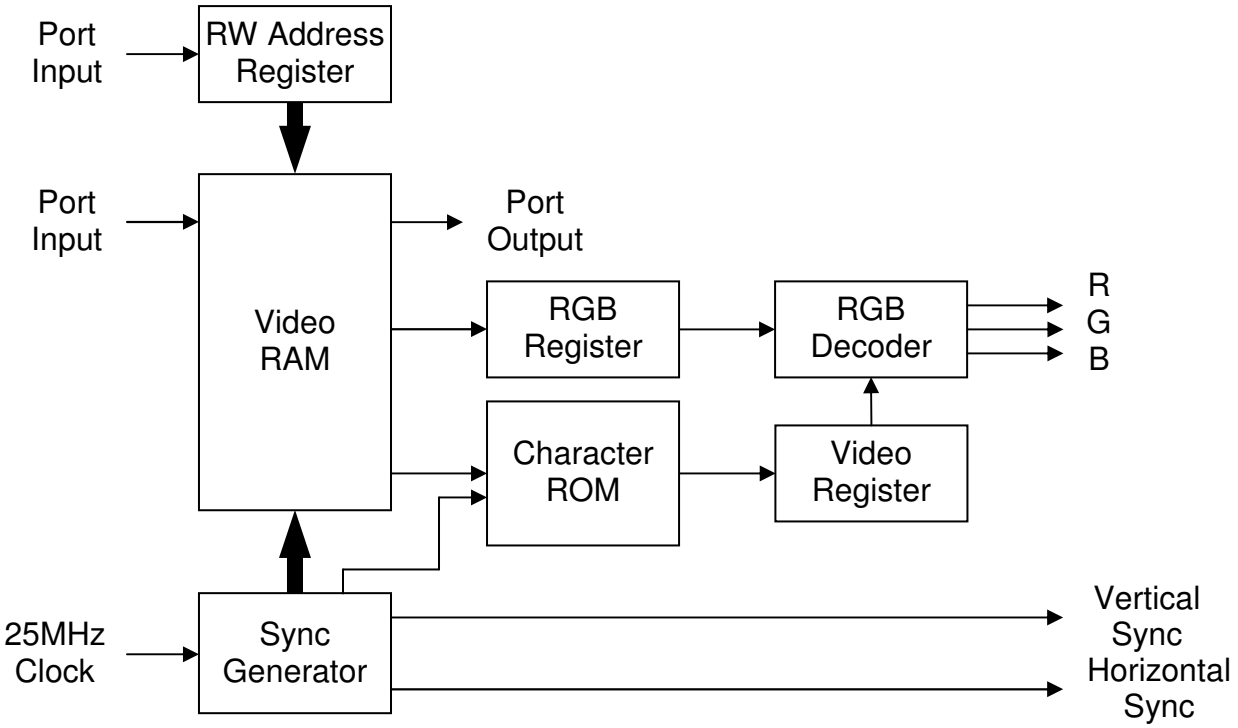

Figure 4: Hardware Implementation of Video Signal Generation Circuit

This video controller also works in character mode, although Character ROM is much bigger this time. It incorporates all the characters from the keyboard; they are placed according to their ASCII code values. Symbols are 8x8 pixels large. Video RAM consists of 16-bit words. Lower byte defines Character ROM address, upper byte – symbol attributes (font color, background color, shifted or non-shifted character). Video RAM has one input and two outputs. Port Input and Port Output are controlled by address stored in RW Address register. Processor may read Video RAM word from Port Output or write one through Port Input by placing its address in RW Address register first. The second Video RAM output is used for video signal generation. Address is produced by Sync Generator. Color concerning information from the upper byte of Video RAM word is stored in RGB register. The lower byte is fed to Character ROM address input. Three lower bits of Vertical Counter from Sync Generator are also fed to Character ROM address input to form a complete ROM word address. Shift bit value is also forwarded to Character ROM to select either upper or lower byte of ROM word. Selected byte is then stored in Video Register. The Video Register's most significant bit defines the currently displayed pixel color. High (equals logic 1) denotes a background, Low (equals logic 0) indicates font. To display the next bit in the loaded character word, register shifts its content one bit left, filling empty least significant bit with zero. RGB decoder produces RGB output based on Video Register's most significant bit value; color information is taken from RGB register.

The instruction set of the processor is a bit larger then suggested in the task (Table 14):

Table 14: Instruction Set of RISC Processor

Mnemonic	Instruction Format					Description
	Op.Code	Data1	Data2	Res	Addr/Imm	
NOP	00000	---	---	---	---	does nothing
JUMP	00001	---	---	---	Addr/Imm	loads Addr/Imm to PC
JUMP E	00010	---	---	---	Addr/Imm	loads Addr/Imm to PC if Equal flag = '1'
JUMP G	00011	---	---	---	Addr/Imm	loads Addr/Imm to PC if Great flag = '1'
JUMP L	00100	---	---	---	Addr/Imm	loads Addr/Imm to PC if Less flag = '1'
JUMP S	00101	---	---	---	Addr/Imm	loads Addr/Imm to PC if Sign flag = '1'
JUMP C	00110	---	---	---	Addr/Imm	loads Addr/Imm to PC if Carry flag = '1'
JUMP Z	00111	---	---	---	Addr/Imm	loads Addr/Imm to PC if Zero flag = '1'
ADD	01000	Data1	Data2	Res	---	adds Data1 to Data2, stores result in Res, affects Sign, Carry, Zero flags.
ADD I	01001	---	Data2	Res	Addr/Imm	adds Addr/Imm to Data2, stores result in Res, affects Sign, Carry, Zero flags.
SUB	01010	Data1	Data2	Res	---	subtracts Data2 from Data1, stores result in Res, affects Sign, Carry, Zero flags.
SUB I	01011	---	Data2	Res	Addr/Imm	subtracts Data2 from Addr/Imm, stores result in Res, affects Sign, Carry, Zero flags.
SHIFT L	01100	Data1	Data2	Res	---	shifts Data1 left according to Data2, stores result in Res, affects Sign, Carry, Zero flags.
SHIFT R	01101	Data1	Data2	Res	---	shifts Data1 right according to Data2, stores result in Res, affects Sign, Carry, Zero flags.
COMP	01110	Data1	Data2	---	---	compares Data1 to Data2, affects Equal, Great, Less flags.
COMP I	01111	---	Data2	---	Addr/Imm	compares Addr/Imm to Data2, affects Equal, Great, Less flags.
AND	10000	Data1	Data2	Res	---	bitwise logic AND of Data1 and Data2, stores result in Res, affects Sign, Carry,

						Zero flags.
AND I	10001	---	Data2	Res	Addr/Imm	bitwise logic AND of Addr/Imm and Data2, stores result in Res, affects Sign, Carry, Zero flags.
OR	10010	Data1	Data2	Res	---	bitwise logic OR of Data1 and Data2, stores result in Res, affects Sign, Carry, Zero flags.
OR I	10011	---	Data2	Res	Addr/Imm	bitwise logic OR of Addr/Imm and Data2, stores result in Res, affects Sign, Carry, Zero flags.
XOR	10100	Data1	Data2	Res	---	bitwise logic XOR of Data1 and Data2, stores result in Res, affects Sign, Carry, Zero flags.
XOR I	10101	---	Data2	Res	Addr/Imm	bitwise logic XOR of Addr/Imm and Data2, stores result in Res, affects Sign, Carry, Zero flags.
NOT	10110	Data1	---	Res	---	bitwise logic NOT of Data1, stores result in Res, affects Sign, Carry, Zero flags.
NOT I	10111	---	---	Res	Addr/Imm	bitwise logic NOT of Addr/Imm, stores result in Res, affects Sign, Carry, Zero flags.
LOAD R	11000	---	Data2	Res	---	loads Res with memory word, addressed by Data2.
STORE R	11001	Data1	Data2	---	---	stores Data1 to memory word, addressed by Data2.
LOAD	11010	---	---	Res	Addr/Imm	loads Res with memory word, addressed by Addr/Imm.
LOSD I	11011	---	---	Res	Addr/Imm	loads Res with Addr/Imm.
STORE	11100	Data1	---	---	Addr/Imm	stores Data1 to memory word, addressed by Addr/Imm.
INPUT	11101	---	Data2	Res	---	loads Res with port value, addressed by Data2.
OUTPUT	11110	Data1	Data2	---	---	stores Data1 to port, addressed by Data2.
OUTPUT I	11111	---	Data2	---	Addr/Imm	stores Addr/Imm to port, addressed by Data2.

Data1, Data2, Res, Addr/Imm denote values from corresponding instruction fields; "---" means that instruction field is ignored, thus its value does not matter.

In order to test the hardware implementation of the RISC processor the program that is shown in Listing 4 has been stored in the Instruction Memory:

Listing 4: Test Program for the RISC Processor

```
00.  LOAD      ---    ---    Rg3     Addr = 0 // Initial Seven-Segment display values
01.  LOAD      ---    ---    Rg4     Addr = 1 // are loaded (letters S, P, A, R).
02.  LOAD      ---    ---    Rg5     Addr = 2
03.  LOAD      ---    ---    Rg6     Addr = 3
04.  LOAD I    ---    ---    Rg2      Imm = 4 // Initial base value is loaded.
05.  JUMP      ---    ---    ---        PC = 9 // Jumps past base reset.
06.  NOP // NOPs are added to correct control hazard.
07.  NOP
08.  LOAD I    ---    ---    Rg2   Imm = 0 // Resets base value.
09.  ADD I     ---    Rg3    Rg7   Imm = X"800" // Adds select signals, which are used
10.  ADD I     ---    Rg4    Rg7   Imm = X"400" // by Seven-Segment controller.
11.  ADD I     ---    Rg5    Rg7   Imm = X"200"
12.  ADD I     ---    Rg6    Rg7   Imm = X"100"
13.  OUTPUT Rg7 "010"   --- // Sends Control Signals to Seven-Segment controller
14.  OUTPUT Rg7 "010"   ---
15.  OUTPUT Rg7 "010"   ---
16.  OUTPUT Rg7 "010"   ---
17.  COMP I    ---    Rg0    ---     Imm = X"FFFF" // Checks whether counter has reached
18.  ADD I     ---    Rg0    Rg0   Imm = 1 // its maximum value and adds one.
19.  JUMPE     ---    ---    ---       PC = 25 // Jumps to second counter.
20.  NOP // NOPs are added to correct control hazard.
21.  NOP
22.  JUMP      ---    ---    ---       PC = 17 // Jumps back to the first counter's start.
23.  NOP // NOPs are added to correct control hazard.
24.  NOP
25.  LOAD I    ---    ---    Rg0     Imm = 0 // Resets first counter.
26.  COMP I    ---    Rg1    ---     Imm = X"40" // Checks whether counter has reached
27.  ADD I     ---    Rg1    Rg1   Imm = 1 // its maximum value and adds one.
28.  JUMPE     ---    ---    ---       PC = 34 // Jumps to shift phase.
29.  NOP // NOPs are added to correct control hazard.
30.  NOP
31.  JUMP      ---    ---    ---       PC = 17 // Jumps back to the first counter's start.
32.  NOP // NOPs are added to correct control hazard.
33.  NOP
34.  LOAD I    ---    ---    Rg1     Imm = 0 // Resets second counter.
35.  SHIFT R Rg4 "000" Rg3 // Moves Rg4 content to Rg3
36.  SHIFT L  Rg5 "000" Rg4 // Moves Rg5 content to Rg4
37.  SHIFT R Rg6 "000" Rg5 // Moves Rg6 content to Rg5
38.  LOAD R    ---    Rg2   Rg6 // Loads a new symbol from memory, addressed by base.
39.  COMP I    ---    Rg2    ---     Imm = 15 // Checks if base has reached maximum.
40.  ADD I     ---    Rg2    Rg2     Imm = 1 // Increments base value
41.  JUMPE     ---    ---    ---       PC = 8 // Jumps to the start, base is reset.
42.  NOP // NOPs are added to correct control hazard.
43.  NOP
44.  JUMP      ---    ---    ---       PC = 9 // Jumps to the start, base is not reset.
```

The rest of the Instruction Memory is filled with NOP commands. The program creates the same creeping line from the Lab 2 on four Seven-Segment LED displays of Spartan-

3 Starter Kit Board. This program assumes that Seven-Segment display values are already stored in Data Memory. One of the two SRAM devices is used for data storage. When FPGA is reconfigured SRAM is not reset. Thus we must implement a memory initialization prior to loading the RISC processor design. Two VHDL processes that are used to write Seven-Segment display values to SRAM device are shown in Listing 5:

Listing 5: SRAM Initialization Processes

```
process (Clock)
begin

if Clock'event and Clock = '1' then

if Counter = "1111111111111111" then
  Counter <= Counter;
else
  Counter <= Counter + 1;
end if;

end if;

end process;

process (Counter)
begin

case Counter is
        when "0000000000000000" => MemData <= X"0049";
        when "0000000000000001" => MemData <= X"0031";
        when "0000000000000010" => MemData <= X"0011";
        when "0000000000000011" => MemData <= X"00F5";
        when "0000000000000100" => MemData <= X"00E1";
        when "0000000000000101" => MemData <= X"0011";
        when "0000000000000110" => MemData <= X"00D5";
        when "0000000000000111" => MemData <= X"00FD";
        when "0000000000001000" => MemData <= X"000D";
        when "0000000000001001" => MemData <= X"00FF";
        when "0000000000001010" => MemData <= X"00C1";
        when "0000000000001011" => MemData <= X"0003";
        when "0000000000001100" => MemData <= X"0011";
        when "0000000000001101" => MemData <= X"00F5";
        when "0000000000001110" => MemData <= X"0085";
        when "0000000000001111" => MemData <= X"00FF";
        when others             => MemData <= X"0000";
end case;

end process;
```

The first process implements an 18-bit counter, which is used to generate SRAM address signal. It sequentially scans through all memory words. The second process is used to assign values to certain memory locations.

The described design has been successfully implemented, verified and demonstrated on the Spartan-3 Starter Kit FPGA Board.