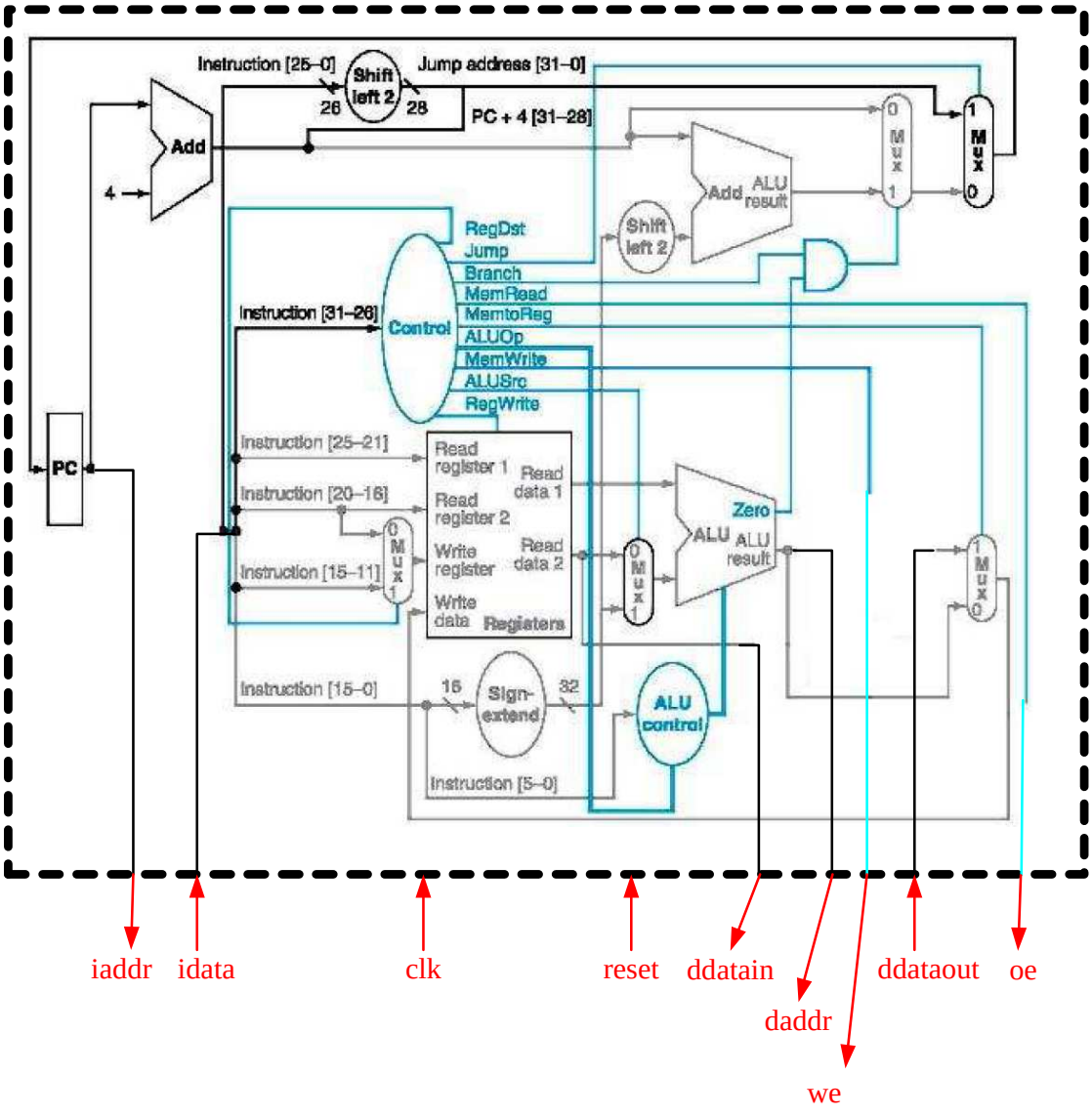
Computer Hardware Experiments

Lab_12

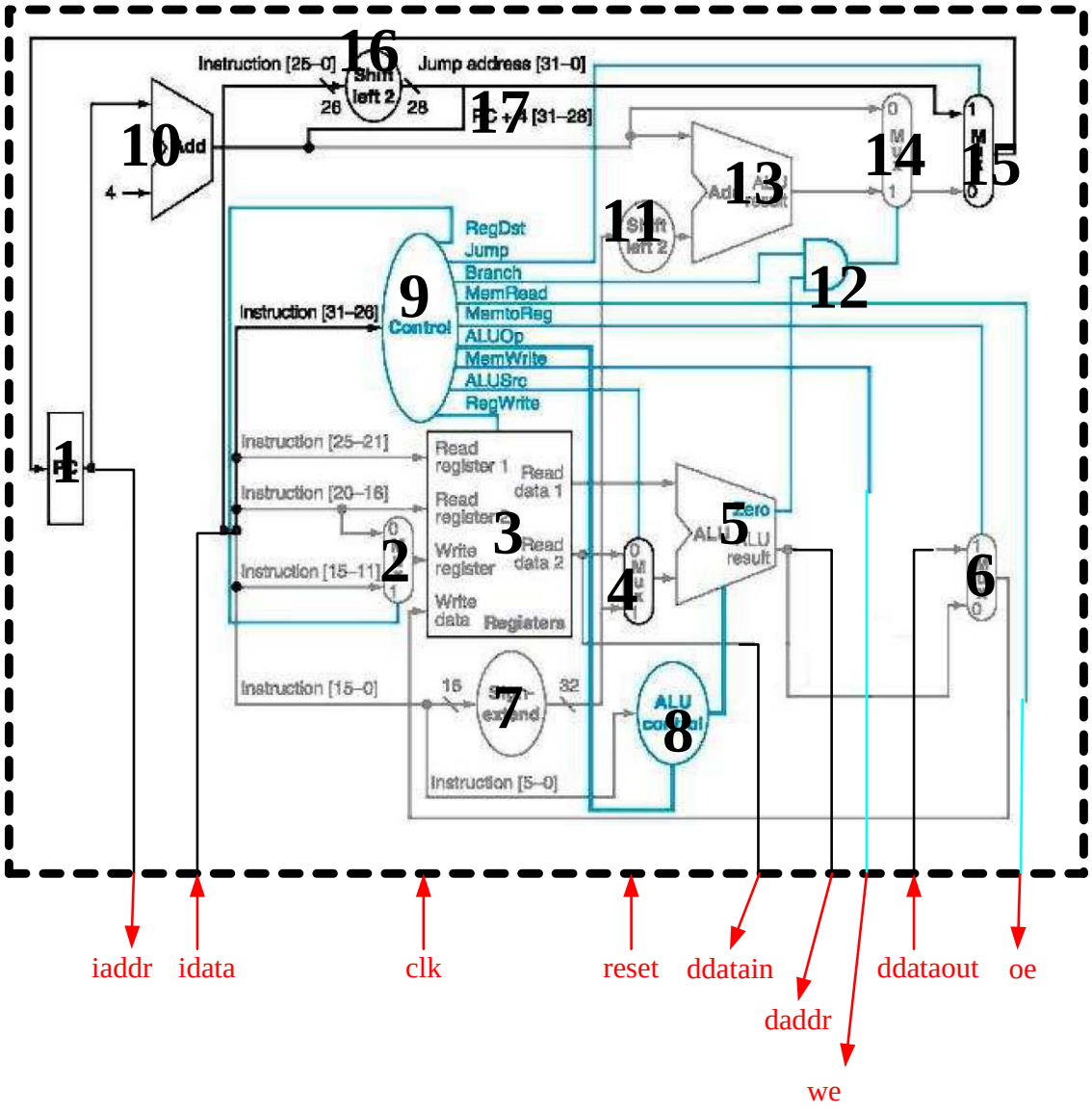
Single Cycle Processor

Deadline: 13/5/2018

In the Lab, you are going to implement a single cycle processor using VHDL, called MiniCPU. The MiniCPU will demonstrate some functions (Table 1) of the MIPS processors as well as the principle of the datapath and the control signals. MiniCPU should read in a file containing MIPS machine codes and simulate what the MIPS processor does cycle-by-cycle. The following is the datapath of the MiniCPU (with input/output port names):



To finish the MiniCPU, you can divide the datapath into small components as shown in the following figure and implement a VHDL for each component. Then you can implement a top level VHDL file, in which you can use port map statement to instantiate all components and connect them together.



Step 1: Implement each component in the above figure:

1. Program counter (PC)

This is a PC register with asynchronous reset. The input/output port is as follows:

```
clk : in STD_LOGIC;
reset : in STD_LOGIC;
in_pc : in STD_LOGIC_VECTOR(31 downto 0);
out_pc : out STD_LOGIC_VECTOR(31 downto 0)
```

When reset is '1', out_pc is set to 0x00004000. Otherwise, out_pc = in_pc at positive edge trigger of clk.

2. Register destination multiplexer

This is a combinational multiplexer which select the destination register. i.e. RS or RD.

3. Register file

This component implements the register file. The VHDL of this component is given, named registers.vhd. Its entity is:

```
entity registers is
port (
     clk: in std_logic;
     reset: in std_logic;
     ra1: in std_logic_vector(4 downto 0);
     ra2: in std_logic_vector(4 downto 0);
     wa: in std_logic_vector(4 downto 0);
     rd1: out std_logic_vector(31 downto 0);
     rd2: out std_logic_vector(31 downto 0);
     wd: in std_logic_vector(31 downto 0);
     we: in std_logic );
end registers;
```

The meaning of each port is:

ra1: read register 1
ra2: read register 2
wa: write register
rd1: read data 1
rd2: read data 2
wd: write data
we: write enable

4. ALU multiplexer

This is a combinational multiplexer which select what data (register or immediate) will be passed to ALU for calculation.

5. ALU

This is a combinational component performs calculation based on input control signal, and assigns “Zero” to 1 if the result is 0; otherwise, assigns 0. The following table shows the operations of the ALU.

Control	Meaning
000	Result = A + B
001	Result = A – B
010	if A < B, Result = 1; otherwise, Result = 0
100	Result = A AND B
101	Result = A OR B
110	Result = Shift B left by 16 bits

It is assumed that the upper input data of the ALU is A, and the lower input data of the ALU is B.

6. Write data multiplexer

This is a combinational multiplexer, control what will be written into register, i.e. data from memory or ALU.

7. Sign extend

This is a combinational extender that extends a 16 bit number to 32 bit.

8. ALU control

This is a combinational component that generates control signal for ALU based on two inputs, “funct” and “ALUOp”.

9. Control signal generation

This is a combinational component that generates control signals based on opcode of instruction (instruction[31-26]). The control signals are shown in the diagram, where the meaning of the values of “ALUOp” is shown as follows:

ALUOp	Meaning
000	ALU will do addition or “don’t care”
001	ALU will do subtraction
010	ALU will do “set less than” operation
100	ALU will do “and” operation
101	ALU will do “or” operation
110	ALU will shift left *extended_value* by 16 bits
111	The instruction is an R-type instruction

10. PC add four

This is a combinational adder, the input is PC, output is PC+4.

11. Shift immediate left

This is a combinational shifter. Input is 32-bit sign extend immediate, output is also a 32-bit signed number which equals to input << 2.

12. Check branch

This is a combinational “AND” gate.

13. Branch address adder

This is a combinational adder calculating branch address, i.e. performs (PC + 4) + signed immediate.

14. Branch multiplexer

This is a combinational multiplexer selecting branch or not.

15. Jump multiplexer

This is a combinational multiplexer selecting jump or not.

16. Jump address shift

This is a combinational shifter. It shift the lower 26 bit of instruction, instruction[25-0], left 2 bits. The input is a 26-bit number, instruction[25-0]. The output is a 28-bit number, which is instruction[25-0] << 2.

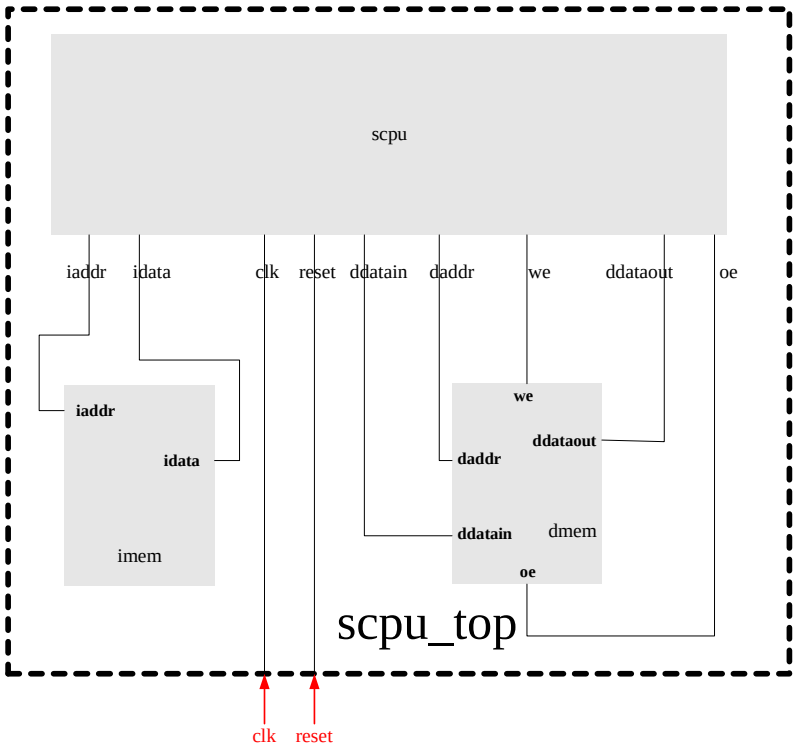
17. Jump address concatenation

This is a combinational component that concatenates the left shifted instruction[25-0] and the first 4 bits of PC. The output is a 32-bit jump address.

Step 2: After you have implemented the above 17 components, you need to implement a top level VHDL file (*scup.vhd*), in which you need to instantiate all the 17 components and use PORT MAP statement to connect them together. You must use the following entity for *scup.vhd*:

```
entity scpu is
    port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        iaddr: out std_logic_vector(31 downto 0);
        idata: in std_logic_vector(31 downto 0);
        daddr: out std_logic_vector(31 downto 0);
        ddatain: out std_logic_vector(31 downto 0);
        ddataout: in std_logic_vector(31 downto 0);
        oe: out std_logic;
        we: out std_logic
    );
end scpu;
```

Step 3: After you finish *scup.vhd*, you can implement another VHDL file (*scup_top.vhd*), in which you can use port map statement to instantiate *scup.vhd* and the other two memory components provided (*imem.vhd* and *dmem.vhd*). Connect the three components as follows:



You must use the following entity for *scup_top.vhd*:

```
entity scpu_top is
    port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC
    );
end scpu_top;
```

The component *imem.vhd* implements the **instruction memory**. The VHDL of this component is given. Its entity is:

```
entity imem is
generic (
     datafile: string := "program.asc" );
port (  clk: in std_logic;
     reset: in std_logic;
     iaddr: in std_logic_vector(31 downto 0);
     idata: out std_logic_vector(31 downto 0));
end imem;
```

The meaning of each input/output port is:

iaddr: input address, you should connect PC to here.
idata: output data, in other word, instruction.

Machine instructions are stored in text file *program.asc*. The instruction memory can output an instruction (idata) every clock cycle based on instruction address (iaddr).

To instantiate this component in your project, you can put the following in your VHDL:

```
component imem
generic (
     datafile: string := "program.asc" );
port (clk: in std_logic;
      reset: in std_logic;
      iaddr: in std_logic_vector(31 downto 0);
      idata: out std_logic_vector(31 downto 0));
end component;
```

The following is an example showing how to port map this component:

```
myimem: imem generic map (
     "program.asc")
port map (
     clk       => clk,
     reset     => reset,
     iaddr     => cur_pc,
     idata     => inst);
```

The component *dmem.vhd* implements the **data memory**, the VHDL has been given for you. Its entity is as follows:

```
entity dmem is
port (
     clk: in std_logic;
     reset: in std_logic;
     daddr: in std_logic_vector(31 downto 0);
     ddatain: in std_logic_vector(31 downto 0);
     ddataout: out std_logic_vector(31 downto 0);
     oe: in std_logic;
     we: in std_logic );
end dmem;
```

The meaning of each input/output is:

daddr: read data address
ddatain: write data
ddataout: read data output
oe: read data enable. If oe is '1', data is read normally; if oe is '0', ddataout is don't care.
we: write data enable.

Step 4: Finally, use the test bench file provided (*scpu_tb.vhd*) to test your design. You can verify your design by checking whether correct result/data is being read from register file/memory or being written into register file/memory. A machine code (*program.asc*) is provided for you to test your program, you must place this file in the directory of your ISE project.

Step 5. Implement your *scpu.vhd*, report the circuit size of your design.

Project report

Write a report to explain your VHDL implementation of the basic MiniCPU, your implementation of the advanced MiniCPU, results, discussions of your implementation, etc.

Notes:

1. You MUST finish your VHDL programs and demonstrate your simulation result on/before Monday, 13/5/2018. You need to submit a **softcopy of your VHDL** programs on/before 13/5 (scup.vhd, scup_top.vhd, the VHDL for the 17 components, and other VHDL files you added). No late submission will be accepted.

2. You MUST submit **a softcopy and a hardcopy of your report** on/before 13/5. EACH STUDENT MUST SUBMIT AN INDIVIDUAL REPORT. No late submission will be accepted.

3. The machine program “program.asc” is provided for you to verify your design. If you want more test cases, you need to convert your assembly instruction into machine format (Hexadecimal) and add them into “program.asc” by yourself.

Table 1: The 12 basic instructions you need to implement.

Category	*Instruction*	*Example*	*Meaning*
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100
Logic	and	and $s1,$s2,$s3	$s1 = $s2 & $s3
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1
	load upper immediate	lui $s1,100	$s1 = 100 * 2^16
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) goto PC + 4 + 100
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1 else $s1 =0
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1 else $s1 =0
Unconditional branch	jump	j label	goto label