CHAPTER 4

Control units

In this chapter you will:

1. Analise a generic datapath that is a simplified version of the DLX microprocessor (only analyze).
2. Understand how a control unit it is organized and how it works.
3. Design different versions of the control unit, WITHOUT writing the code for the Datapath.
4. Design and test an hardwired control unit for the datapath, WITHOUT writing the code for the Datapath.
5. Design and test a FSM control unit for the datapath, WITHOUT writing the code for the Datapath.
   - A few details on how to implement a FSM are given in the last section (4.5).
6. Design and test a microcontrolled control unit for the datapath, WITHOUT writing the code for the Datapath.

The given structure is a simplified version of the DLX datapath, so this lab will be the starting point for the control unit of your final project. Copy the files from the usual directory: **/home-/repository/ms/cap4/**

In case it was not clear from previous instructions... you don't have to write the code for the data path, but ONLY THE CONTROL UNIT for it.

4.1 Datapath structure

4.1.1 Circuit architecture

The DLX processor (see cap.2 and 3 of Hennessy-Patterson) is a simple *load-store* architecture with 32 32-bit general-purpose registers (GPRs). In figure 7.1 a slightly simplified version of its datapath is shown.

This datapath (not to be implemented here, but to be controlled only.... repetita iuvant) is composed of 3 pipeline stages.

- A first stage, with a register file and 5 pipeline registers. The register file has 2 read port and 1 write port. The addresses of the register file are external inputs (**RS1**, **RS2**, **RD**), while the input data arrive from the output of the alu. 4 control signals are necessary in this pipe stage:
  - **EN1** enables the register file and the pipeline registers;
  - **RF1** enables the read port 1 of the register file;

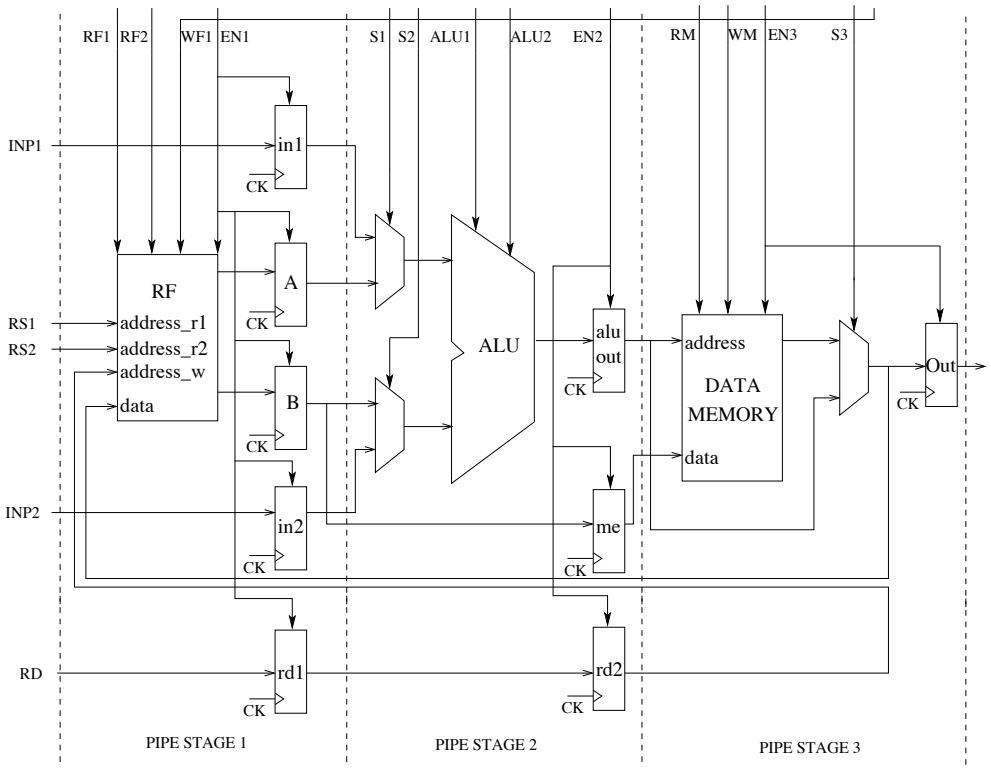

Figure 4.1:

- **RF2** enables the read port 2 of the register file;
- **WF1** enables the write port of the register file.

- In the second stage, 2 multiplexers are used to select the operands of an alu, while 3 registers are used for pipelining. The arithmetic-logic unit is a simple circuit that allows 4 type of operation: addition, subtraction, logic AND, logic OR. As a consequence only two control bits are necessary for the alu. Each multiplexer allows the selection between one of the output port of the register file and an external input (**INP1**, **INP2**). 5 control signals are necessary in this pipe stage:
  - **EN2** enables the pipe registers;
  - **S1** input selection of the first multiplexer;
  - **S2** input selection of the second multiplexer;
  - **ALU1**, **ALU2** alu control bit;
    - **00** addition;
    - **01** subtraction;
    - **10** AND;
    - **11** OR.
- In the third pipe stage a memory is used to load/store data. The memory is a simplified register file with 1 port for read and writing. The address of the memory is generated by the alu, while the data in input is the delayed output of the register B. A multiplexer is used to select the data that will be written in the register file, between the output of the memory and the alu output. 2 registers are used for pipelining. 4 control signals are used in this pipe stage:
  - **EN3** enables the memory and the pipeline register;
  - **RM** enables the read-out of the memory;
  - **WM** enables the write-in of the memory;
  - **S3** input selection of the multiplexer.

4.1.2 Instructions

DLX Instructions are grouped into 3 main types. As a consequence in this simplified data-path the control words follow the DLX instructions structure to help you in your project:

- *I-type*: Loads/Stores and ALU operations between a register and an immediate.
- *R-type*: Register-register ALU operations.

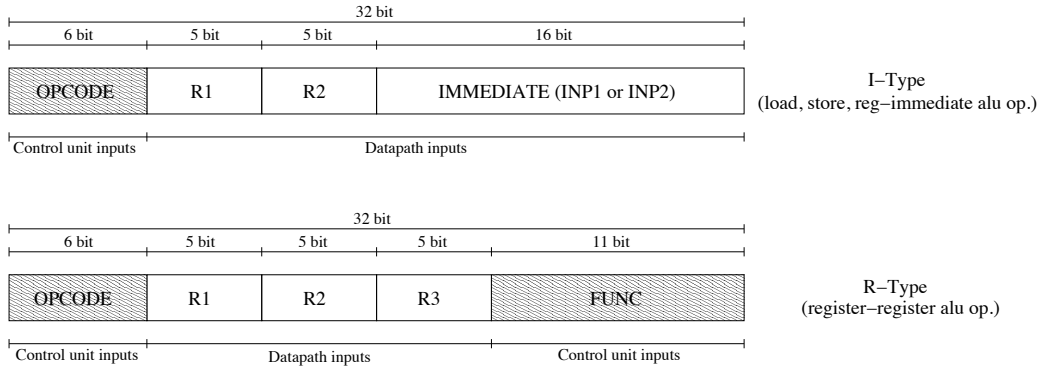

Figure 4.2:

The control word is compose by many field.

- Control unit inputs.
  - **OPCODE** (6 bit) It identifies the type of the operation.
  - **FUNC** (11 bit, 2 used and 9 unused in this datapath) In case of R-Type instructions the OPCODE indicates only that a mathematical operation must be executed, but the type of operation is indicated in the FUNC field.
- Datapath inputs:
  - R-Type control word.
    - **R1** (5 bit) It indicates the address of the first register.
    - **R2** (5 bit) It indicates the address of the second register.
    - **R3** (5 bit) It indicates the address of the register where the results of the alu operation must be stored.
  - I-Type control word.
    - **R1** (5 bit) It has different meanings: it's one of the operand of arithmetic operations or it's the address of the register that must be moved into another location, is the address of a register that is used to calculate the address for a load/store in memory.
    - **R2** (5 bit) It indicates the address of the register where the results of the operation must be stored and it stores the number that must be saved in memory.
    - **IMMEDIATE** (16 bit) It indicates an immediate value that can be added to a number stored in a register and it is used to calculate the address for a load/store in memory.

4.1.3 Instruction set

The instruction set is the complete set of instruction that a circuit can perform. The instruction set of this datapath is the following.

- R-Type instructions:
  - **ADD R1,R2,R3** (meaning R[R3] = R[R1] + R[R2]);

  - **SUB R1,R2,R3** (meaning R[R3] = R[R1] - R[R2]);
  - **AND R1,R2,R3** (meaning R[R3] = R[R1] AND R[R2]);
  - **OR R1,R2,R3** (meaning R[R3] = R[R1] OR R[R2]);

- I-Type instructions:
  - **ADDI1 R1,R2,INP1** (meaning R[R2] = R[R1] + INP1);
  - **SUBI1 R1,R2,INP1** (meaning R[R2] = R[R1] - INP1);
  - **ANDI1 R1,R2,INP1** (meaning R[R2] = R[R1] AND INP1);
  - **ORI1 R1,R2,INP1** (meaning R[R2] = R[R1] OR INP1);
  - **ADDI2 R1,R2,INP2** (meaning R[R2] = R[R1] + INP2);
  - **SUBI2 R1,R2,INP2** (meaning R[R2] = R[R1] - INP2);
  - **ANDI2 R1,R2,INP2** (meaning R[R2] = R[R1] AND INP2);
  - **ORI2 R1,R2,INP2** (meaning R[R2] = R[R1] OR INP2);
  - **MOV R1,R2** (meaning R[R2] = R[R1]) - The value of the immediate must be equal to 0;
  - **S_REG1 R2,INP1** (meaning R[RD] = INP1) - Save the value INP1 in the register file, R1 field is not used;
  - **S_REG2 R2,INP2** (meaning R[RD] = INP2) - Save the value INP2 in the register file, R1 field is not used;
  - **S_MEM2 R1,R2,INP2** (meaning MEM[R[R1]+INP2] = R[R2]) - The content of the register R2 is saved in a memory cell, which address is calculated adding the content of the register R1 to the value INP2;
  - **L_MEM1 R1,R2,INP1** (meaning R[R2] = MEM[R[R1]+INP1]) - The content of the memory cell, which address is calculated adding the content of the register R1 to the value INP1, is saved in the register R2;
  - **L_MEM2 R1,R2,INP2** (meaning R[R2] = MEM[R[R1]+INP2]) - The content of the memory cell, which address is calculated adding the content of the register R1 to the value INP2, is saved in the register R2;

4.1.4 Instruction execution example

Let's suppose that an addition must be performed: ADD R1,R2,R3. The instruction requires three clock cycles to be executed.

- FIRST CLOCK CICLE
  - **EN1 = 1** The register file and all pipe registers are enabled.
  - **RF1 = 1** The first read port of the register file is enabled.
  - **RF2 = 1** The second read port of the register file is enabled.
- SECOND CLOCK CICLE
  - **EN2 = 1** All pipe registers are enabled.
  - **S1 = 0** The first register file output is selected (the setting of S1 can change and depends on how the inputs are connected to the multiplexer).
  - **S2 = 1** The second register file output is selected (the setting of S2 can change and depends on how the inputs are connected to the multiplexer).
  - **ALU1 = 0, ALU2 = 0** The alu is prepared for the addition.
- THIRD CLOCK CICLE

- **EN3 = 1** The memory and all pipe registers are enabled.
- **RM = 0, WM = 0** Both port of the memory are disabled.
- **S3 = 0** The output of the alu is selected (the setting of S3 can change and depends on how the inputs are connected to the multiplexer).
- **WF1 = 1** The write port of the register file is enabled.

4.2 HARDWIRED control unit

The schematics of the control unit is shown in figure 4.3. The control unit receives the OPCODE (6 bit) and the FUNC (11 bit) signals and generates the control signals. The control unit can be seen as a simple Look-up table. Control signals are delayed to match the delay of the Datapath. The execution of each instruction is fully pipelined, that means at each clock cycle a new instruction can be sent to the circuit.

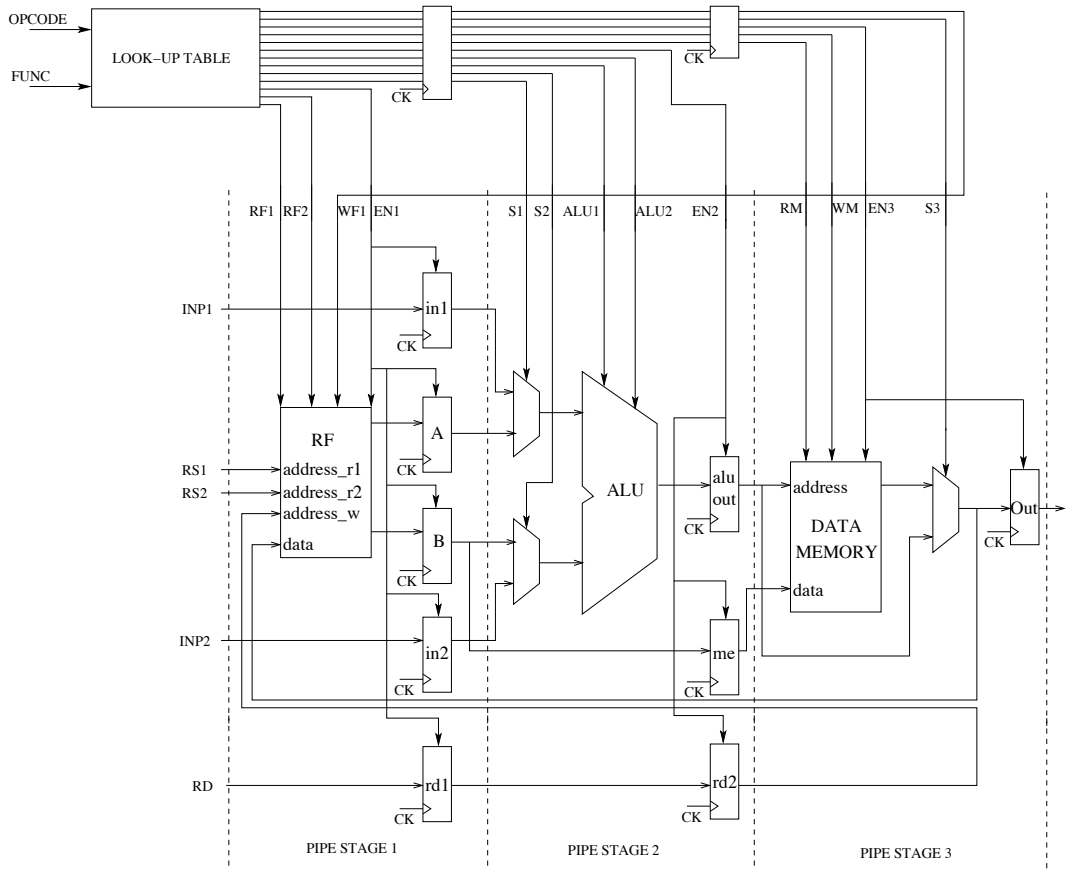

Figure 4.3:

IMPORTANT: Your aim in this lab is to create the control unit and not the Datapath itself (maybe still not clear?). The control unit is a device which has two input signals OPCODE (6 bit) and FUNC (11 bit) and 13 output control signals. Start from files myTypes.vhd and CU_Test.vhd, and complete them. The first one is a package containing the coding of the instructions to simplify the testbench writing. The file CU_Test.vhd contains the testbench for the control unit, where you can see the component of the control unit itself that you have to describe in a separate file. The file CU_HW.vhd contains the hardwired control unit of the DLX, have a look at it to get some hints on how to make your control unit. Create a NEW file for your control unit, DON'T create your control unit modifying the file CU_HW, this file serves only to help you understand the structure of an harwired control unit.

1. Set up the simulation.

- Set-up the modelsim environment variable **setmentor**.
- Create the work library (vlib work).
- Create the hardwired control unit that executes all the Datapath instructions according to the component that you find in the file CU_Test.vhd.
- Create the testbench for the hardwired control unit starting from the file CU_Test.vhd.

2. Simulate all the instructions.

Summary of what is requested

Hardwired control unit executing all the instructions: VHDL netlist (well well commented!) and testbench (commemted), meaningful waveforms. Not required but suggest is a flow diagram of your control unit.

4.3 FSM control unit

Using the same environment you can now work with the FSM version. Clearly, to complete it, we assume you know how to describe a FSM in vhdl. If not, a small example of a two state FSM is given (fsm1.vhd and test bench tb_fsm.vhd to simulate it, a few details are in section 4.5). It is important to underline that, differently from the hardwired control unit, the execution of the instructions is NOT PIPELINED. A new input must arrives at the control unit inputs every three clock cycles.

HINTS & TIPS. In the DLX microprocessor the states of the FSM control unit correspond to the various step of the instructions execution (fetch, decode, execute, ...). Each of the five states activate the control signals of the correspondent pipeline stages. The "fetch" state correspond to the loading of the instruction. The next state is one of the "Decode" state, one for any instruction executable. This strategy can be applied also to this Datapath. The file CU_FSM.vhd contains the FSM control unit of the DLX, have a look at it to get some hints on how to make your control unit. Create a NEW file for your control unit, DON'T create your control unit modifying the file CU_FSM.vhd, this file serves only to help you understand the structure of a FSM control unit.

Summary of what is requested

FSM control unit executing all the instructions: VHDL netlist (well commented!) and testbench, meaningful waveforms.

4.4 Micro programmed control unit

The schematics of the control unit is shown in figure 4.5.

The microcode memory stores the appropriate control signals depending on the instruction and on the pipe stage of the instruction, thus implementing the FSM for the given instruction. The OPCODE and FUNC of the instruction are used as a *starting-address* for the microcode memory and it is stored in the uPC register. In a normal three clock-cycle instruction, the basic address stored in the uPC is incremented three times in order to generate the subsequent control signals for all the datapath stages.

Since the OPCODE and FUNC are used as an address to map the control in the five different stages, the value of each OPCODE must differ of at least three values. It is important to underline that, differently from the hardwired control unit, the execution of the instructions is NOT PIPELINED. A new input must arrives at the control unit inputs every three clock cycles.

HINTS & TIPS. An example on how the microcode memory must be filled and how the micro-controlled control unit works is reported in figure 4.6. The OPCODE and the FUNC field are the address of the microcode memory, so probably you will need to change the coding of each instruction in the file myTypes.vhd to optimize the memory occupation. This address is the input of a program counter, which schematics is shown in figure 4.6. At the first clock cycle the program counter takes

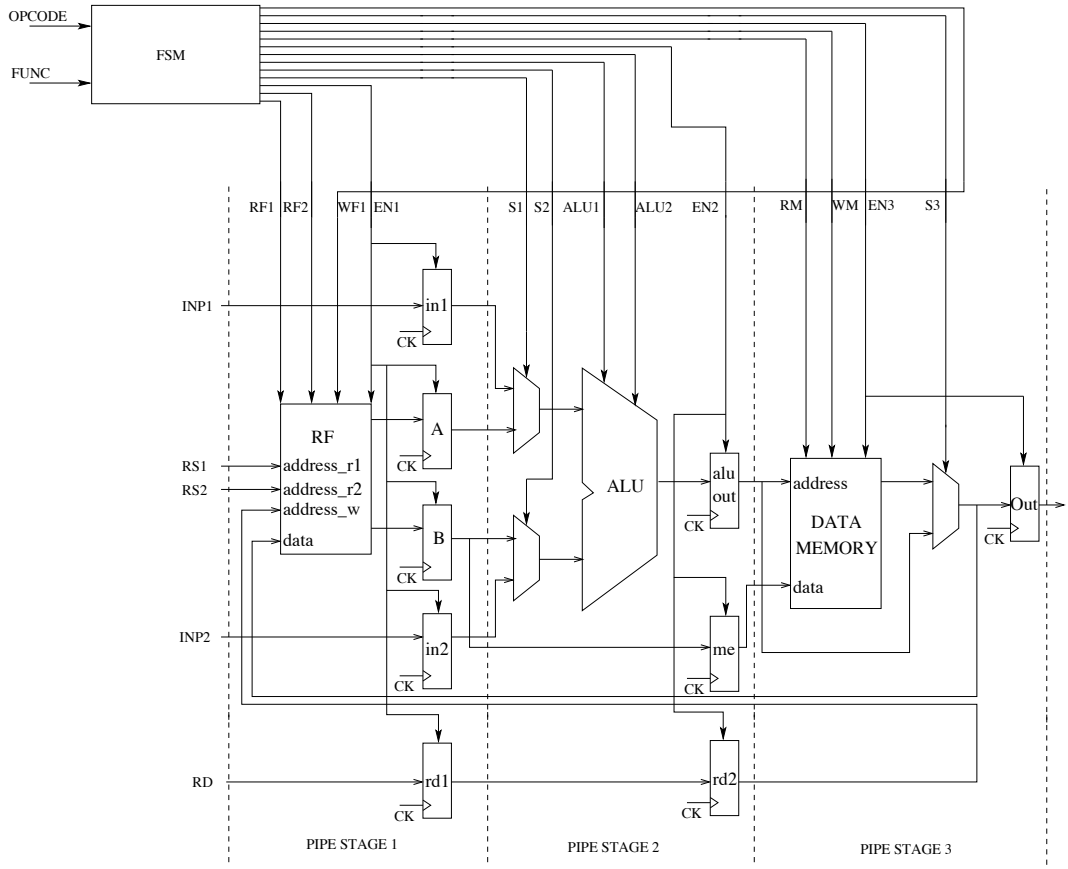

Figure 4.4:

a new address which correspond to the location in the microcode memory of the selected instruction. For the next 2 clock cycles the previous adder is incremented by 1 at each clock cycle. Then a new address is taken from the inputs, corresponding to a new instruction. The file CU_UP.vhd contains the microprogrammed control unit of the DLX, have a look at it to get some hints on how to make your control unit. Create a NEW file for your control unit, DON'T create your control unit modifying the file CU_UP.vhd, this file serves only to help you understand the structure of a microprogrammed control unit.

Summary of what is requested

Microcontrolled control unit executing all the instructions: VHDL netlist (well commented!) and testbench, meaningful waveforms.

4.5 Odd parity checker

Just to remember, let's start with a simple "odd parity checker" implemented using a FSM. The FSM's input is a bit serial stream and the output is a odd/even count flag ('0' is even, '1' is odd).

The "State Transition Diagram (STG)" of this FSM is the following

You will find the file **fsm1.vhd** and test bench **tb_fsm1.vhd** in the usual directory. Open it and see how the next state and output processes look like. Simulate the test bench.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity odd_parity_checker is
port(   A:      in std_logic;
        clock:  in std_logic;
```

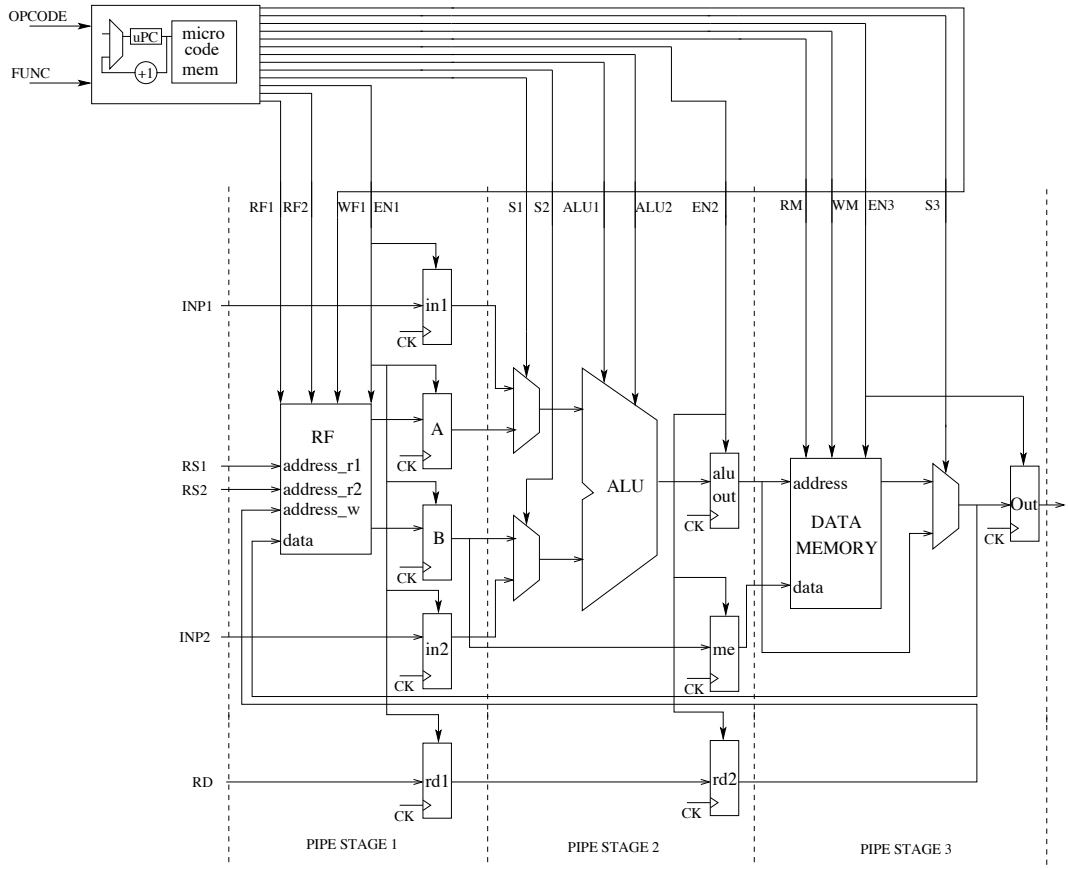

Figure 4.5:

		FUNC OPCODE	RF1	RF2	EN1	S1	S2	A1	A2	EN2	RM	WM	EN3	S3	WF1	
Clock cycle 1	uPC =	00000000000000000	1	1	1	0	0	0	0	0	0	0	0	0	0	ADDI R1, R2, INP1
Clock cycle 2	uPC = uPC + 1 =	00000000000000001	0	0	0	0	1	0	0	1	0	0	0	0	0	
Clock cycle 3	uPC = uPC + 1 =	00000000000000010	0	0	0	0	0	0	0	0	0	0	1	0	1	
	uPC =	00000000000000011	...	...	...	...	...	...	...	...	...	...	...	...	...	SUBI R1, R2, INP1
		.....	...	...	...	...	...	...	...	...	...	...	...	...	...	
		.....	...	...	...	...	...	...	...	...	...	...	...	...	...	
		.....	...	...	...	...	...	...	...	...	...	...	...	...	...	

Figure 4.6:

```
        reset:   in std_logic;
        O:       out std_logic
);
end     odd_parity_checker;

architecture FSM_OPC of odd_parity_checker is

        type TYPE_STATE is (S0, S1);
        signal CURRENT_STATE : TYPE_STATE;
        signal NEXT_STATE : TYPE_STATE;

begin
        -- STATE TRANSITION PROCESS
        P_OPC : process(CLOCK, RESET)
        begin
                if reset='1' then
                        CURRENT_STATE <= S0;
                elsif (CLOCK ='1' and CLOCK'EVENT) then
                        CURRENT_STATE <= NEXT_STATE;
```

```
            end if;
        end process P_OPC;

        -- NEXT STATE ASSIGNEMENT (FUNCTION OF INPUT AND PREVIOUS STATE)
        P_NEXT_STATE : process(CURRENT_STATE,A)
        begin
            NEXT_STATE <= CURRENT_STATE;
            case CURRENT_STATE is
                when S0 => if A ='0' then
                               NEXT_STATE <= S0;
                           elsif A ='1' then
                               NEXT_STATE <= S1;
                           end if;
                when S1 => if A ='0' then
                               NEXT_STATE <= S1;
                           elsif A ='1' then
                               NEXT_STATE <= S0;
                           end if;
            end case;
        end process P_NEXT_STATE;

        -- OUTPUT DEFINITION PROCESS (FUNCTION OF CURRENT STATE ONLY -- MOORE)
        P_OUTPUTS: process(CURRENT_STATE)
        begin
            --O <= '0';
            case CURRENT_STATE is
                when S0 => O <= '0';
                when S1 => O <= '1';
            end case;
        end process P_OUTPUTS;
end FSM_OPC;
```