c Microelectronic Systems

Lab 4

Control units

May 4, 2022

Introduction

During this laboratory experience you will:

1. Analyse a generic datapath, a simplified version of the *DLX microprocessor*.
2. Understand how a Control Unit is organized and how it works.
3. Design and test different versions of the Control Unit for the given simplified DLX datapath:
   - Hardwired
   - Finite State Machine
   - Micro Programmed

For all these exercises, you are not requested to design any datapath but only to understand the composition and the behavior of the one described inside the next section. In particular, the discussed structure represents a simplified version of the DLX datapath, therefore, this Lab will be the starting point for the definition of the Control Unit of your final project.

As usual, copy the files from the following directory:

/home/repository/ms/cap4/

4.1 Datapath structure

The following sections detail the reference structure of a DLX implementation, from its architecture to its behavior.

4.1.1 Circuit architecture

The DLX processor *(see cap.2 and 3 of Hennessy-Patterson)* is a simple *load-store* architecture with a set of 32 General Purpose Registers *(GPRs)*, each with a parallelism of 32-bits. A slightly simplified version of its datapath is shown in Figure 4.1.

This datapath is composed of **three pipeline stages**:

- A first stage which consists of a *Register File* followed by *5 pipeline registers*. The Register File has *two read ports* and *one write port*. The addresses are provided to the Register File through external inputs (**RS1**, **RS2**, **RD**), while the input data arrive directly from the output of the ALU. *Four control signals* are necessary within this pipe stage:
  - **EN1** enables the *Register File* and the *pipeline registers*;
  - **RF1** enables the *read port 1* of the Register File;
  - **RF2** enables the *read port 2* of the Register File;
  - **WF1** enables the *write port* of the Register File.
- In the second stage, *two multiplexers* are considered to select the operands at the input of the ALU, while *three registers* are used as pipeline. The Arithmetic-Logic Unit is a simple circuit that allows four types of operation: *addition, subtraction, logic AND, logic OR*. As a consequence, only *two control bits* are necessary for its proper functioning. Each input multiplexer allows the selection between one of the output ports of the Register File, belonging to the previous stage, and external inputs (**INP1**, **INP2**). *Five control signals* are necessary in this pipeline stage:
  - **EN2** enables the pipe registers;

  - **S1** allows the input selection of the first multiplexer;
  - **S2** allows the input selection of the second multiplexer;
  - **ALU1**, **ALU2** control the ALU operations:
    * **00** for addition;
    * **01** for subtraction;
    * **10** for AND;
    * **11** for OR.

- In the third pipeline stage, a *Memory* is used to load/store data. It is a simplified Register File with one port for reading and one for writing. The address is generated by the ALU, while the input data come from the delayed output of the register B *(see Figure 4.1)*. A *multiplexer* is considered to select the data that will be written into the Register File belonging to the first stage, and which is located between the output of the Memory and the DLX output. *One register* is used for pipelining. *Four 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** allows the *input selection* of the multiplexer.

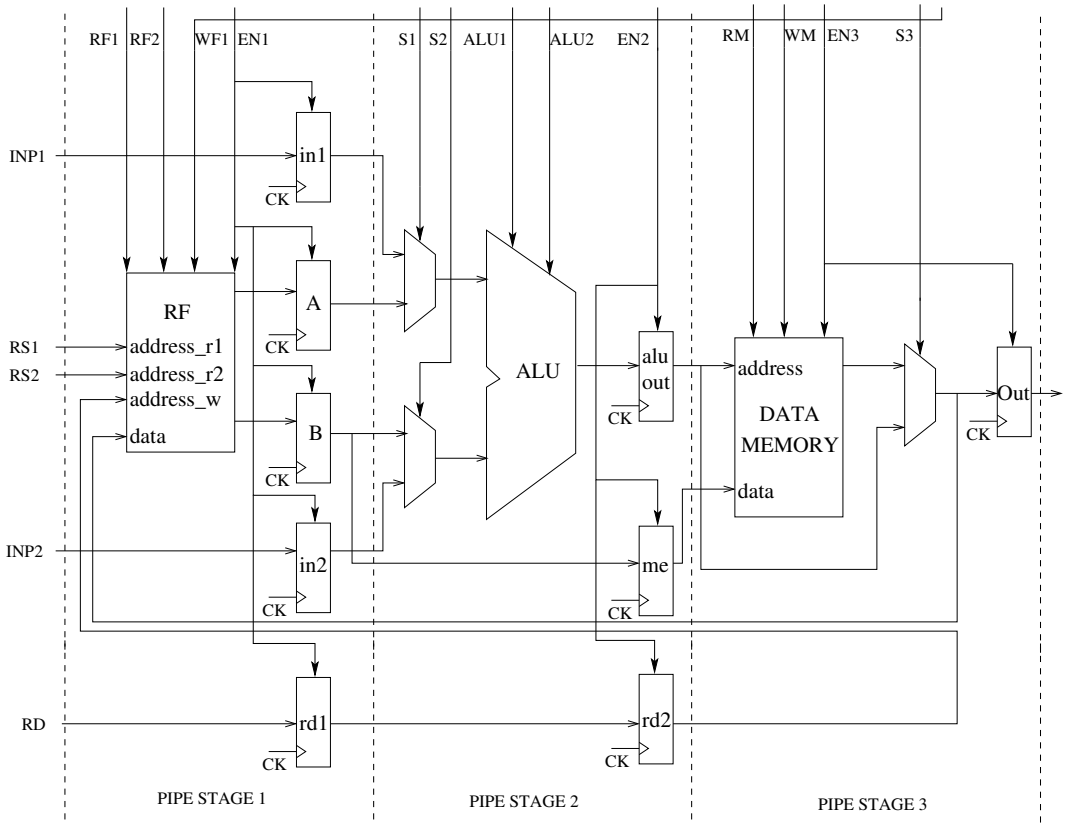

Figure 4.1: Schematic of a simplified implementation of a DLX processor.

4.1.2 Instructions

The accepted Instructions by the original implementation of the DLX are grouped into *three main types*. Since you are considering a simplified data-path, only two of them will be supported. However, the structure and the aims of these instructions remain faithful to the original ones:

- *I-type*: Loads/Stores and ALU operations between a register and a data provided by the command itself.
- *R-type*: Register-register ALU operations.

More in detail, the control word is composed of many fields:

- Control unit inputs:
  - **OPCODE** *(6 bits)*: It identifies the type of the operation.
  - **FUNC** *(11 bits, 2 used and 9 unused in this implementation)*: In case of R-Type instructions the OPCODE indicates only that a mathematical operation must be executed, but the type of the operation is specified by the FUNC field.
- Datapath inputs:
  - R-Type control word.
    - **RA** *(5 bit)*: It indicates the address of the first operand register.
    - **RB** *(5 bit)*: It indicates the address of the second operand register.
    - **RC** *(5 bit)*: It indicates the address of the register where the results of the ALU operation has to be stored.
  - I-Type control word.
    - **RA** *(5 bit)*: It has different meanings. It could represent one of the operands of an arithmetic operation, or the address of the register that must be moved into another location, or even the address of a register that is used to calculate the address for a load/store in memory.
    - **RB** *(5 bit)*: It indicates the address of the register where the results of an operation must be stored, or from which to derive a data that must be saved in memory.
    - **IMMEDIATE** *(16 bit)*: It indicates an immediate operand that can be added to a number stored inside a register. The result is then considered to calculate the address for a load/store in memory.

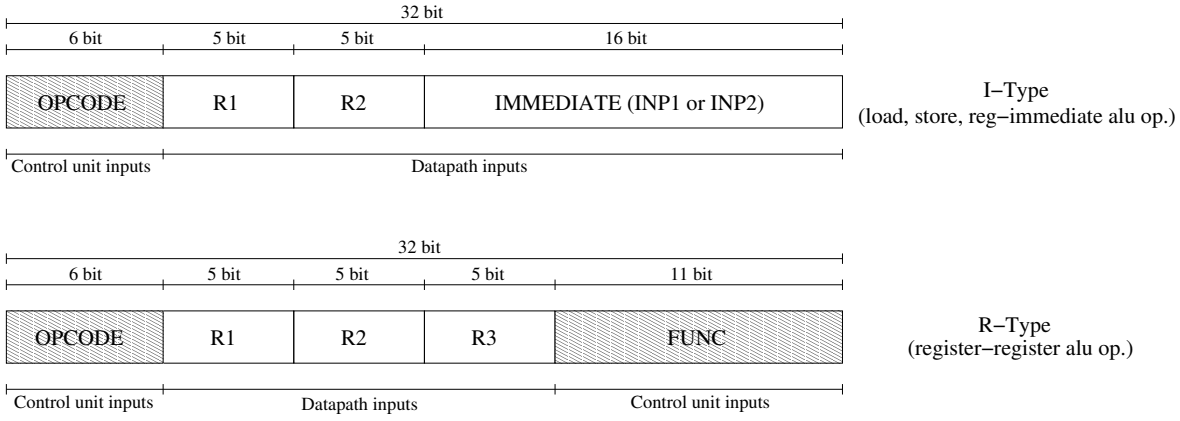

Figure 4.2: Structure of the implemented instructions.

4.1.3 Instruction set

The Instruction Set is the complete set of instructions that a circuit can perform. The one characterizing the considered datapath is described in the following.

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

4.1.4 Instruction execution example

Let's suppose that an **addition** must be performed:

ADD RA RB RC

The instruction requires three clock-cycles to be executed and, in particular:

- FIRST CLOCK CYCLE
  - **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 CYCLE
  - **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 CYCLE

- **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 schematic of the Control Unit is shown in Figure 4.3. As you can observe, the circuit receives in input two signals, **OPCODE** *(6 bits)* and **FUNC** *(11 bits)*, and generates the corresponding 13 control signals. The Control Unit can be seen as a simple Look-up table.

The control signals are delayed matching the delay of the datapath and the execution of each instruction is fully pipelined, i.e. at each clock cycle a new instruction can be sent to the circuit.

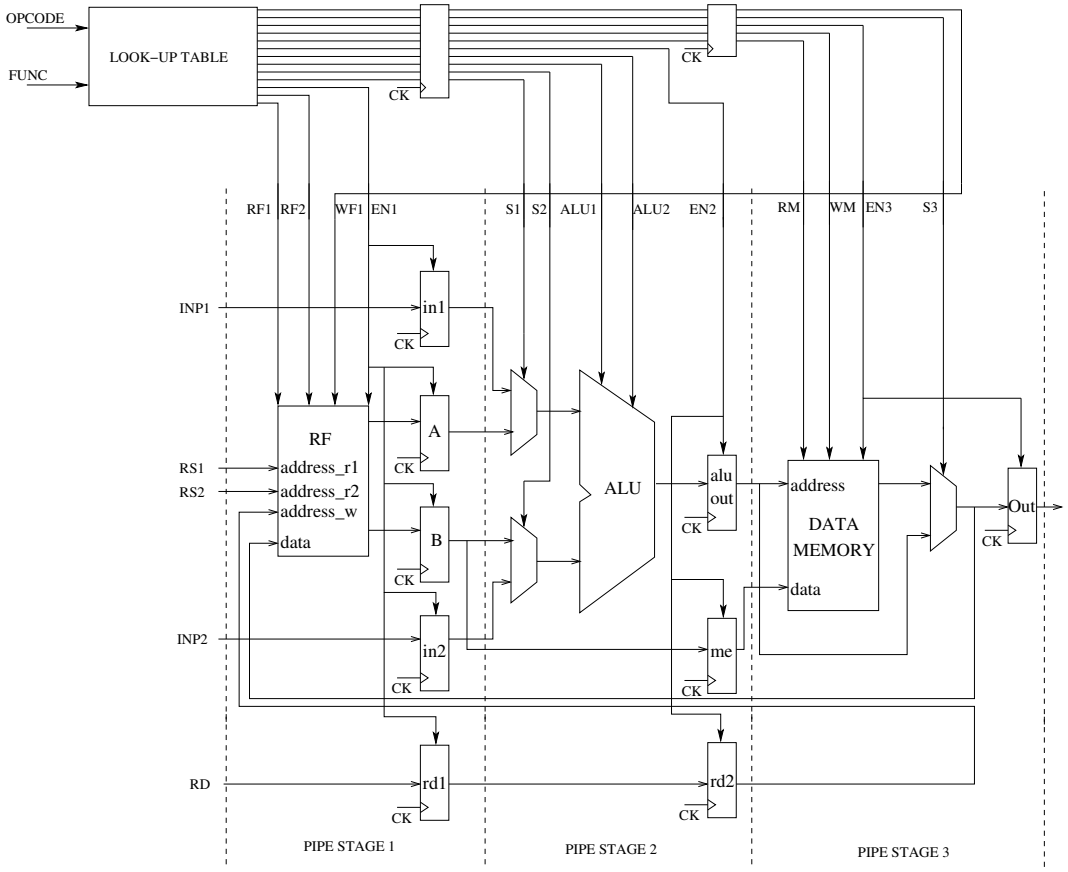

Figure 4.3: Schematic of the complete structure of the simplified DLX architecture, where a *hardwired Control Unit* has been implemented.

Please, remember that the purpose of this laboratory experience is to design of a Control Unit, not the associated datapath.

Start from the provided files **myTypes.vhd** and **CU_Test.vhd** and complete them, considering that:

- ➔ **myTypes.vhd** is a package containing the coding of the instructions to simplify the writing of the testbench.
- ➔ **CU_Test.vhd** contains a piece of a testbench for the Control Unit, where the instance of the relative component is declared. From this description, you have to design the Control Unit in a separate file.

Take a look also at the file **CU_HW.vhd**, which contains an example of a DLX Hardwired Control Unit, to get some hints on how to make your own implementation.

Create a **NEW** file for your Control Unit, **DON'T** directly modify the file **CU_HW**. This serves only as a reference structure for such a Control Unit.

When your design is complete, you are requested to finalise the provided testbench *(CU_Test.vhd)*, to prove the correct behaviour of the circuit.

Hints&Tips: Please, remember the following instructions to set up the simulation environment:

- Set-up the Modelsim environment variable **setmentor**.
- Create the work library, **vlib work**.

Summary of what is requested

Hardwired control unit executing all the instructions: VHDL netlist and testbench, both well commented, and meaningful waveforms.

4.3 FSM Control Unit

Considering the same datapath, you can now work with a *Finite State Machine (FSM)* version of the Control Unit. Its reference structure is depicted in Figure 4.4. Clearly, to complete this task, you are assumed to know how to describe a FSM in VHDL. If you are not, a small example of a two state FSM is given, named **fsm1.vhd**, together with the related testbench **tb_fsm.vhd**. Moreover, a few additional details are reported in Appendix A.

It is important to underline that, differently from the Hardwired Control Unit, for the FSM the execution of the instructions is NOT PIPELINED. A new input must arrive at the input of the Control Unit every three clock cycles.

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 own Control Unit.

Create a **NEW** file for your Control Unit, **DON'T** modify the file **CU_FSM.vhd**. This serves only to help you understand the structure of such a Control Unit.

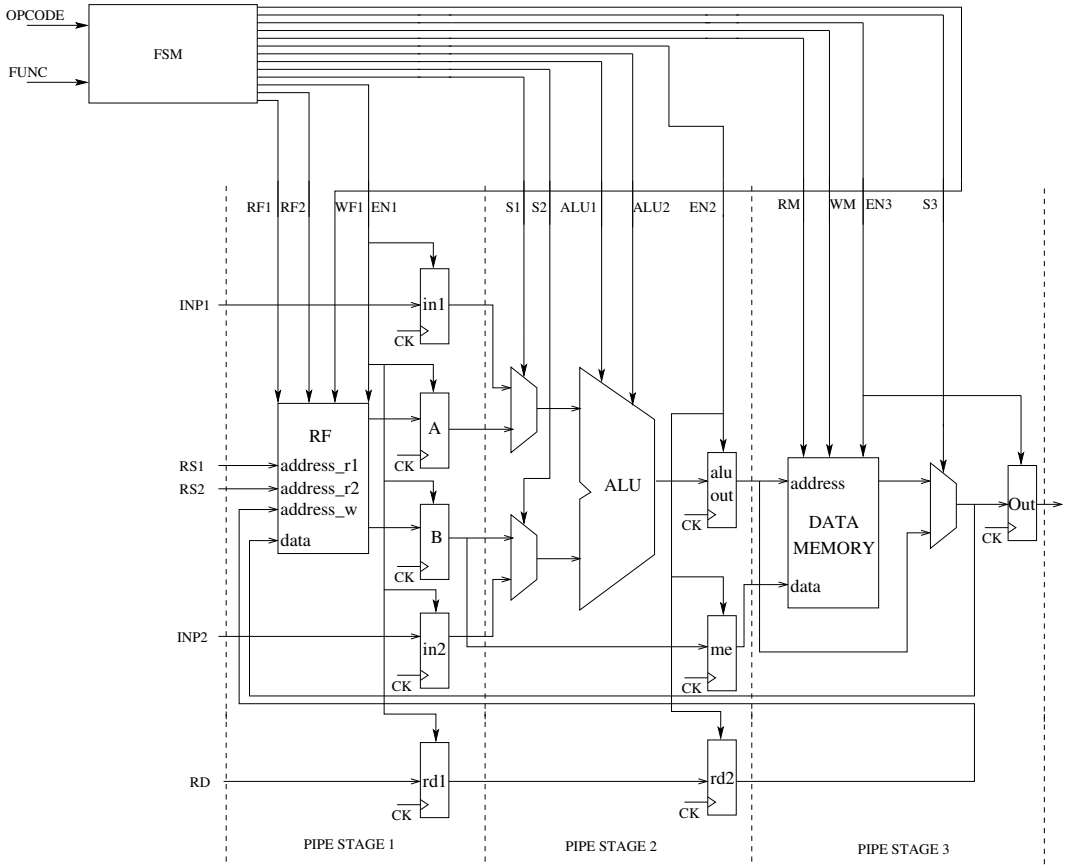

Figure 4.4: Schematic of the complete structure of the simplified DLX architecture, where a *FSM Control Unit* has been implemented.

Hints&Tips: In the DLX microprocessor, the states of the FSM Control Unit correspond to the various steps of the execution of the instructions *(i.e. fetch, decode, execute, ...)*. Each of the five states enables the control signals of the associated pipeline stages. Note that, the "fetch" state corresponds to the instruction load. The state that follows is one among the possible "Decode" states, one for any executable instruction. The described strategy can be applied also to the simplified datapath you are considering for this Lab.

Summary of what is requested

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

4.4 Micro Programmed Control Unit

The third and last version of the Control Unit you are required to implement is the *Micro Programmed* one. Its schematic is shown in Figure 4.5, while additional details are discussed in the following:

➔ The CU integrates a **microcode memory** which stores the appropriate control signals for each instruction and the related pipe stage, thus implementing the FSM for the given instruction.

➔ The **OPCODE** and **FUNC** fields of the instructions, stored together in the uPC register, are used as a *starting-address* for the microcode memory.

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 fields are considered as an address to map the control in the five different stages, the value of each OPCODE must differ of at least three values.

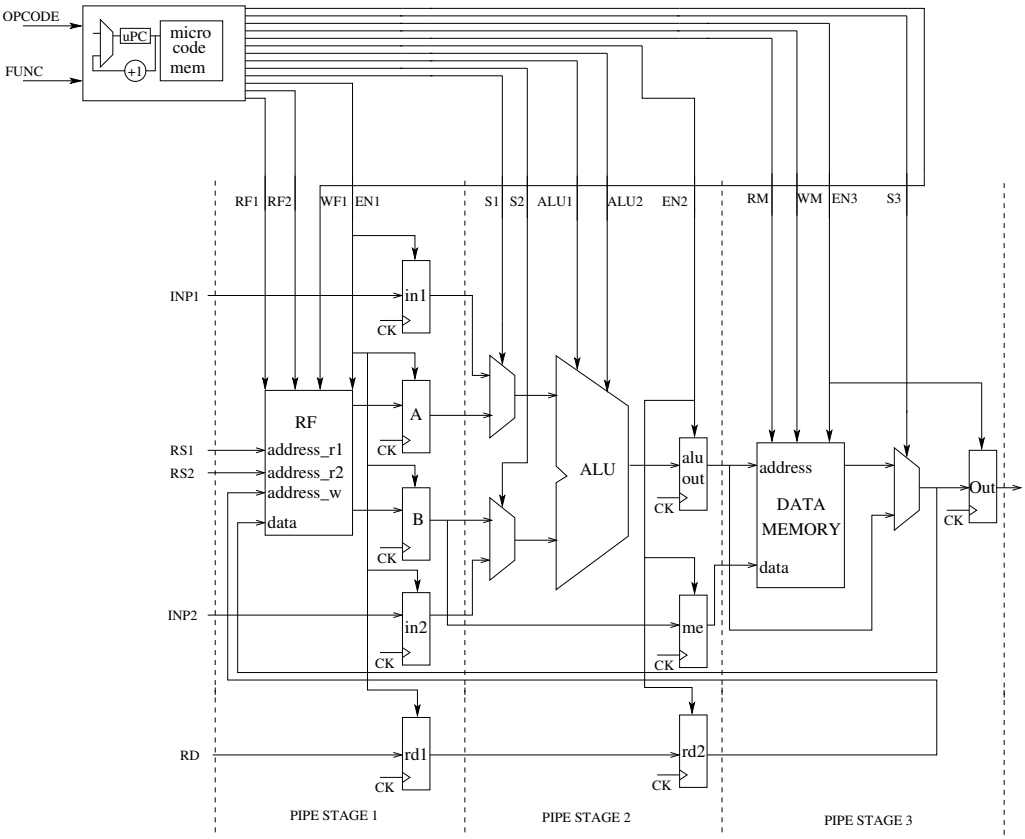

Figure 4.5: Schematic of the complete structure of the simplified DLX architecture, where a *Micro Programmed Control Unit* has been implemented.

Hints&Tips: An example on how the micro programmed Control Unit works, with particular attention to how the microcode memory must be filled, is reported in Figure 4.6. The union of the OPCODE and FUNC fields represents the address of the microcode memory, which is placed at the input of a Program Counter. Therefore, you will probably need to change the coding of each instruction inside the file **myTypes.vhd**, to optimize the memory occupation. The behavior of the circuit is described in the following:

1. At the first clock cycle the Program Counter takes a new address which points to a proper location of the microcode memory.
2. For each of the next 2 clock cycles the previous address is incremented by 1.
3. Then, a new instruction, so a new address, is fetched from the input.

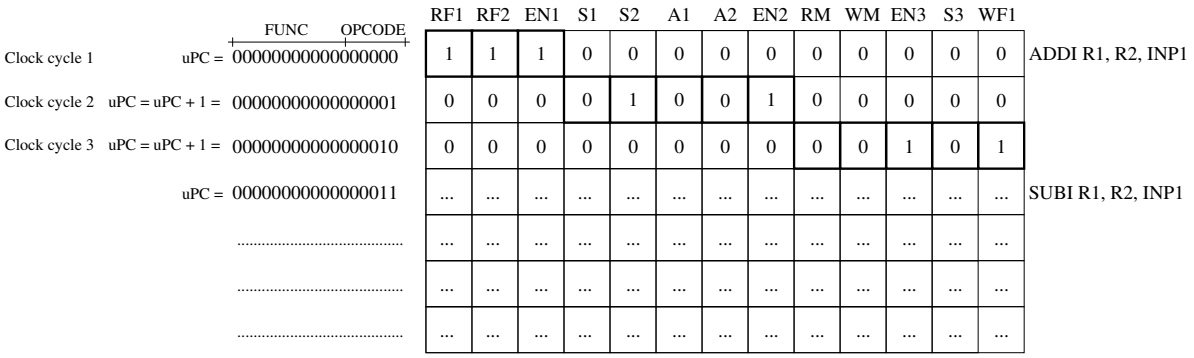

Figure 4.6: Working principle of the Microcode Memory within the Micro Programmed Control Unit.

The provided file **CU_UP.vhd** contains the Micro Programmed Control Unit of the DLX. Have a look at it to get some hints on how to make your own Control Unit.

Create a **NEW** file for your Control Unit, **DON'T** modify the file **CU_UP.vhd**. This serves only to help you understand the structure of such a Control Unit.

Summary of what is requested

Micro Programmed Control Unit executing all the instructions: *well commented* VHDL netlist and testbench, meaningful waveforms.

Appendix A

Odd parity checker

Here, a simple "odd parity checker" implemented using a Finite State Machine is discussed. The input of the FSM is a bit serial stream, while the output is an odd/even count flag *('0' is even, '1' is odd)*. The *"State Transition Diagram (STG)"* of this FSM is depicted in Figure A.1.

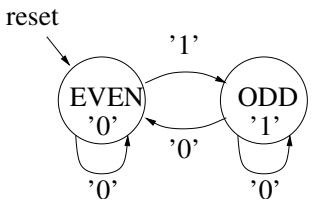

Figure A.1: State Transition Diagram of an odd parity checker circuit.

Among the provided files for this lab, you will find **fsm1.vhd** and **tb_fsm1.vhd**. Open the former and see how the next state and the output processes look like. Then, simulate it through the test bench.