MIPSfpga
by Imagination

Lab 15

Basic Instruction Flow – AND Instruction

Lab 15

Basic Instruction Flow – AND Instruction

1. Introduction

In this lab we analyze the flow of the `and` instruction along the microAptiv pipeline. This R-type instruction (format: `and rd, rs, rt`) reads the values stored in two registers (the *source* registers, *rs* and *rt*), performs the logic *AND* operation between the two values (*Reg[rs] & Reg[rt]*), and writes the result to a third register (the *destination* register, *rd*). Section 6.4.1 of [1] explains the `and` instruction in detail, Figure 1 illustrates its format, and the instruction functionality can be expressed as follows:

Reg[rd] = Reg[rs] & Reg[rt]

We begin this lab by explaining the main tasks carried out by an `and` instruction in each stage of the pipeline (Section 2). Section 3 walks the students through a detailed simulation of the `and` instruction through the pipeline and Section 4 provides exercises that guide the students in exploring and expanding the pipeline using various logic instructions.

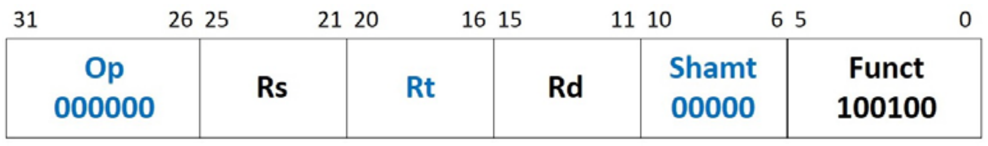


Figure 1. `and` machine instruction format.

2. Pipeline Stages

In this section we explain the execution of the `and` instruction through each stage of the pipeline. Much of the process is identical to the `add` instruction, so we present only those stages, modules, and signals that differ. These differences are summarize in Table 1, and the remaining section describes the stages in detail.

Table 1. MIPSfpga pipeline: main modules and signals related to the and instruction

E-Stage	
Module/Signal Name	**Description**
m14k_edp	**Execution datapath**
logic_ain_e[31:0]	Logic Unit Source A
logic_bin_e[31:0]	Logic Unit Source B
logic_out_e[31:0]	Result of the logic operation at the E-Stage

mpc_alufunc_e	Control signal of multiplexer *_logic_out_e_31_0_*, computed at **m14k_mpc_dec**
M-Stage	
Module/Signal Name	**Description**
m14k_edp	**Execution datapath**
prealu_m[31:0]	Result of the logic operation at the M-Stage
mpc_sellogic_m	Control signal of multiplexer *_edp_alu_m_31_0_*, computed at **m14k_mpc_dec**
edp_alu_m[31:0]	Result of the ALU operation at the M-Stage

a. E-Stage

The Execute (E) Stage *executes*, or performs, the desired operation. The three main functions of this stage for an and instruction are to: (1) fetch two registers from the register file (RF), (2) perform the *AND* operation of the two source operands, and (3) decode the instruction – i.e., generate the control signals. The first and third functions are analogous to the add instruction, thus, below we only explain the second function in detail.

Figure 2 illustrates the main structures and signals involved in the E-Stage and in the M-Stage of an and instruction. The Logic Unit, implemented in module **m14k_edp**, performs the *AND* operation, the *OR* operation, the *XOR* operation (using 2 *AND* gates, 1 *OR* gate and one inverter), and the *NOR* operation (using an inverter after the *OR* gate). A 4-1 multiplexer (*_logic_out_e_31_0_*) selects one of those operations, and a new register (*_prealu_m_31_0_*) registers the result to the M-Stage.

The following code segment, extracted from lines 1350-1366 of module **m14k_edp**, includes the Logic Unit, the 4-1 multiplexer and the register from Figure 2:

```
mvp_mux2 #(32) _logic_bin_e_31_0_(logic_bin_e[31:0],mpc_alubsrc_e, edp_abus_e,
bbus_imm_e);

mvp_mux2 #(32) _logic_ain_e_31_0_(logic_ain_e[31:0],mpc_aluasrc_e, edp_abus_e,
bbus_imm_e);

assign a_and_b_e [31:0] = logic_ain_e & logic_bin_e;
assign a_or_b_e [31:0] = logic_ain_e | logic_bin_e;
assign a_xor_b_e [31:0] = a_or_b_e & ~a_and_b_e;
assign a_nor_b_e [31:0] = ~a_or_b_e;

mvp_mux4 #(32) _logic_out_e_31_0_(logic_out_e[31:0],mpc_alufunc_e, a_and_b_e, a_or_b_e,
a_xor_b_e, a_nor_b_e);

mvp_mux2 #(32) _alu_e_31_0_(alu_e[31:0],mpc_clsel_e, logic_out_e, cnt_lead_e);

mvp_cregister_wide #(32) _prealu_m_31_0_(prealu_m[31:0],gscanenable, mpc_prealu_cond_e,
gclk, alu_e);
```

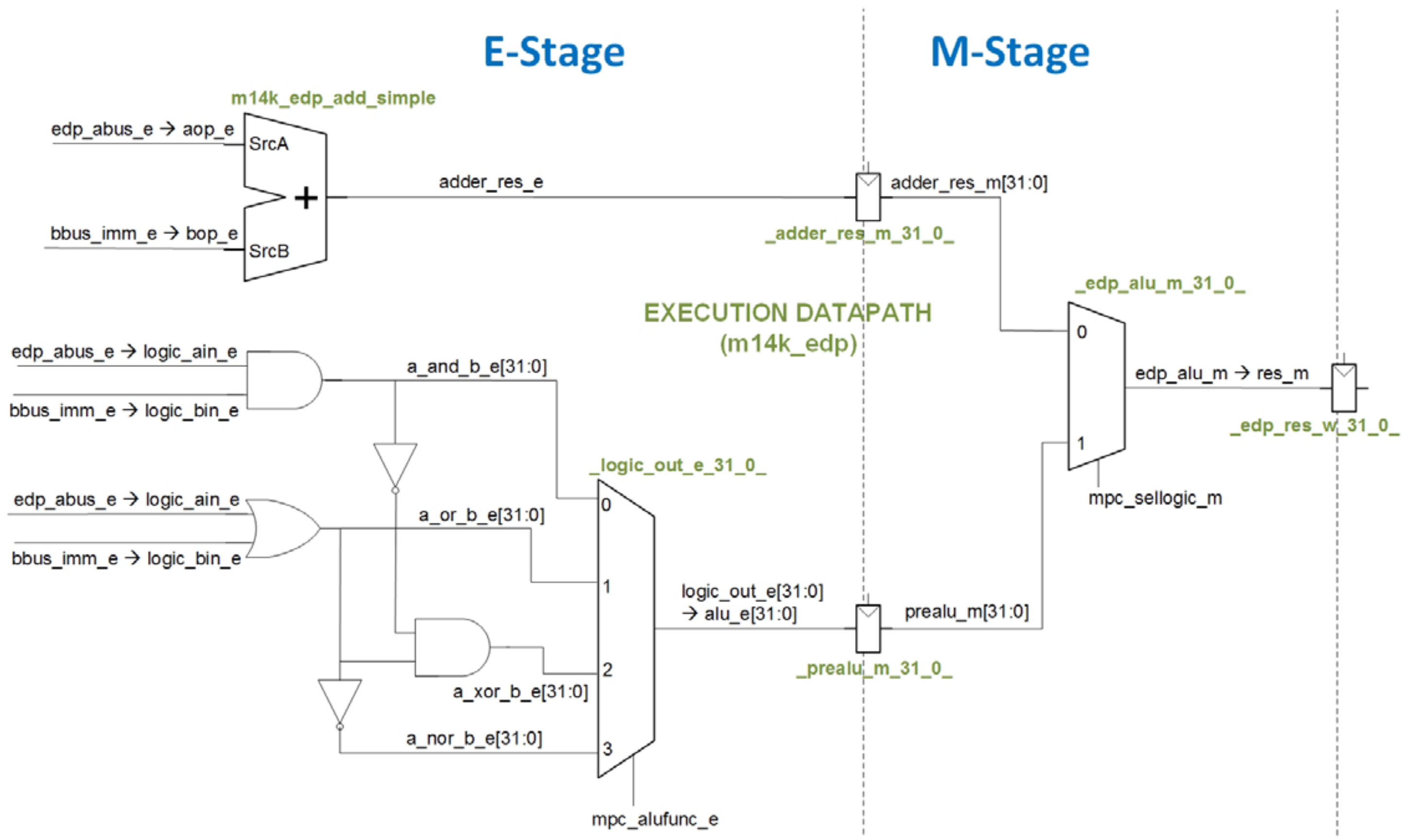


Figure 2. Main structures and signals involved in the E-Stage and M-Stage of the and instruction. For the sake of simplicity, some elements, such as the register file and the forwarding multiplexers, are not shown here (you can see them in Figure 3 of Lab 14).

The logic unit computes the *AND* operation of signals *logic_ain_e* and *logic_bin_e* and outputs it on signal *a_and_b_e[31:0]*. The *_logic_out_e_31_0_* multiplexer then selects input 0, which is registered to the M-Stage (*prealu_m[31:0]*).

b. M-Stage

In the M-Stage (Figure 2) a new multiplexer is included (for the sake of simplicity, in Figure 3 of Lab 14 we did not include this multiplexer) for selecting the value to write to the RF in the W-Stage, which can come from the arithmetic unit or from the logic unit. The following code segment, extracted from line 1369 of module **m14k_edp**, implements this multiplexer:

```
        mvp_mux2 #(32) _edp_alu_m_31_0_(edp_alu_m[31:0],mpc_sellogic_m, adder_res_m, prealu_m);
```

c. Comparision of microAptiv with the processor from *DDCA* [1]

This section gives a brief comparison between the microAptiv pipeline and the pipelined processor introduced in *DDCA* [2] and illustrated in Figure 7.58 of that book, with regards to the and instruction.

- The ALU in *DDCA* performs operations needed by the implemented instructions: addition, subtraction, *AND*, *OR*, and set if less than, as described in Sections 5.2.4 and 7.3 of the book. The ALU in microAptiv performs operations needed by the full set of MIPS R3 instructions. Specifically, the logic unit performs *AND*, *OR*, *NOR* and *XOR* operations.
- In *DDCA* [1], the ALU chooses between performing an arithmetic or logic function using a multiplexer that is internal to the ALU. On the other hand, microAptiv uses two multiplexers external to the ALU(see Figure 2, 4:1 multiplexer at the E-Stage and 2:1 multiplexer at the M-Stage), to choose between logic or arithmetic functions.

3. Example Simulation

In this section we illustrate the simulation of the and instruction as it flows through the microAptiv pipeline. We first guide you in the simulation process and then analyze the results in detail. Viewing the behavior of the different signals related to an and instruction helps you understanding the theoretical explanations of Section 2.

a. Simulation Process

In order to simulate a compiled program using Vivado's XSIM you can create a new project, following the instructions provided in Section 4.a of Lab 14, or you can reuse the project created in Lab 14 as explained in Exercise 1 of that lab. In folder *Lab15_AND\Simulations* we provide both the new waveform configuration file (*testbench_boot_behav.wcfg*) and the source files (*SimulationSources*). Before you start to use these source files, make a copy of the whole folder. You can view the source program in file **main.c**. This simple program initializes registers t3 and t4, *ands* them, and places the result in t5. You may also be interested in viewing file **program.dis** which shows the disassembled executable interspersed with the assembly or C source code. If you look for "<main>:" in that file, you can see the assembly instructions, as well as the memory address and machine code of each instruction (Figure 3).

```
…
80000204 <main>:
80000204:   240b0002    li    t3,2
80000208:   240c0003    li    t4,3
8000020c:   240d0000    li    t5,0
80000210:   00000000    nop
80000214:   018b6824    and   t5,t4,t3
80000218:   1000ffff    b     80000218 <main+0x14>
8000021c:   00000000    nop
…
```

Figure 3. Example assembly program, with the and instruction highlighted in blue.

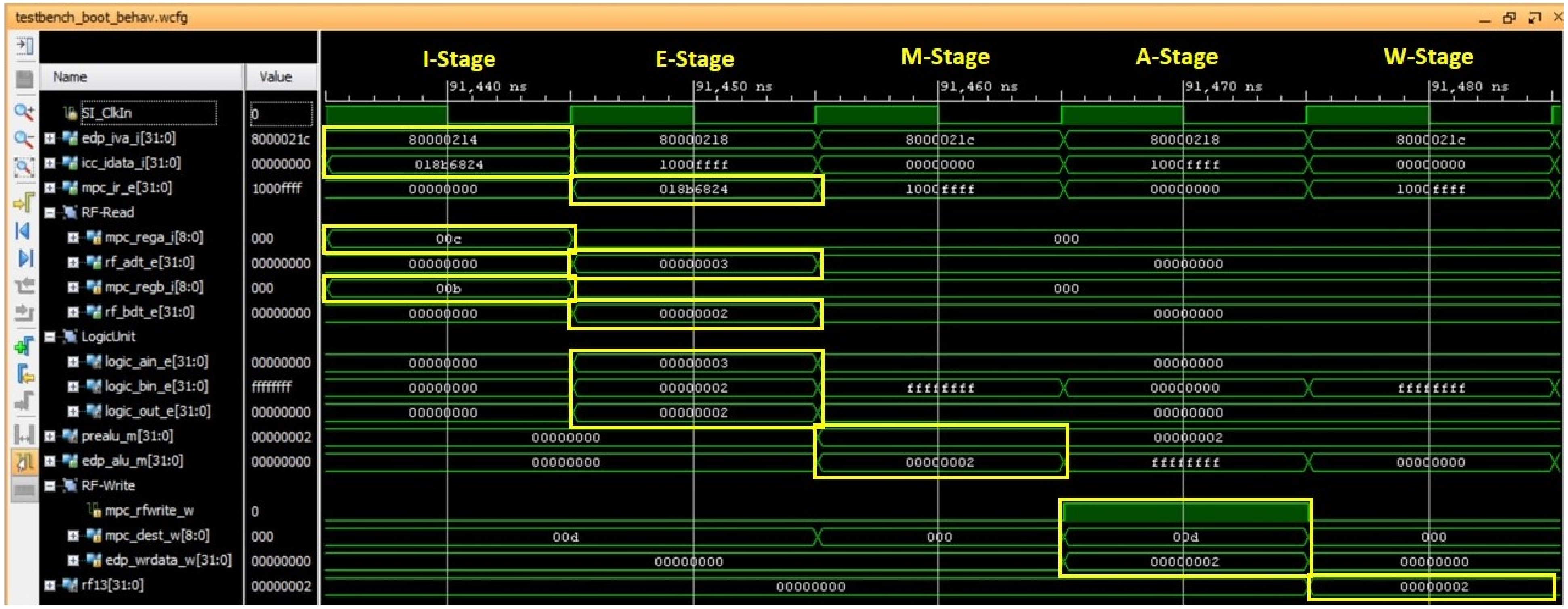
Figure 1 shows the machine format of an `and` instruction, which is explained extensively in Section 6.3.1 of *DDCA* [2]. Specifically, for the `and` instruction used in our program (`and t5, t4, t3` - 0x018b6824 - 0000 0001 1000 1011 0110 1000 0010 0100), each field has the following meaning:

- **Opcode**: 000000, indicates an R-type instruction (*DDCA* terminology) or SPECIAL instruction (microAptiv terminology)
- **rs**: 01100, the first source operand, t4, which is register 12 (as defined by the MIPS ISA and in file "regdef.h")
- **rt**: 01011, the second source operand, t3, which is register 11 (again, see the MIPS ISA or "regdef.h")
- **rd**: 01101, the register destination, t5, which is register 13 (again, see the MIPS ISA or "regdef.h")
- **shamt**: 00000, the amount to shift (not applicable for an `add` operation, so it is 0)
- **funct field**: 100100, indicates the function to perform, in this case *AND*

Now, configure the simulation: The `and` instruction is fetched around time 91440ns, thus configure the simulation to run for 92000ns, as explained in Figure 8 of Lab 14. Moreover, use the waveform configuration provided for this exercise (*Lab15_AND\Simulations\testbench_boot_behav.wcfg*) as explained in Figure 7 of Lab 14, and use the text files for initializing the memories (*Lab15_AND\Simulations\SimulationSources*), as explained in Figure 6 of Lab 14.

b. Analysis of the simulation of the and instruction

Figure 4 illustrates a timing diagram showing the stages of execution of an `and` instruction.



Name	Value
SI_ClkIn	0
edp_iva_i[31:0]	8000021c
icc_idata_i[31:0]	00000000
mpc_ir_e[31:0]	1000ffff
RF-Read	
mpc_rega_i[8:0]	000
rf_adt_e[31:0]	00000000
mpc_regb_i[8:0]	000
rf_bdt_e[31:0]	00000000
LogicUnit	
logic_ain_e[31:0]	00000000
logic_bin_e[31:0]	ffffffff
logic_out_e[31:0]	00000000
prealu_m[31:0]	00000002
edp_alu_m[31:0]	00000000
RF-Write	
mpc_rfwrite_w	0
mpc_dest_w[8:0]	000
edp_wrdata_w[31:0]	00000000
rf13[31:0]	00000002

Figure 4. Timing diagram of the execution of an and instruction.

Next we detail the results obtained in the execution of the and instruction, shown in Figure 4. In this analysis we skip those things that are identical to the add instruction.

- 2nd cycle, E-Stage.
  - The operands read from the Register File are provided as inputs to the Logic Unit:
    - ***logic_ain_e***=0x00000003.
    - ***logic_bin_e***=0x00000002.
  - The and operation is performed in the Logic Unit and result (given by signal *logic_out_e*) is selected by the *_logic_out_e_31_0_* multiplexer:
    - ***logic_out_e***=0x00000002.
  - The result of the *AND* operation is registered in *_prealu_m_31_0_*.
- 3rd cycle, M-Stage.
  - The result of the Logic Unit, stored in register *_prealu_m_31_0_*, is selected by the *_edp_alu_m_31_0_* multiplexer:
    - ***prealu_m***=0x00000002.
    - ***edp_alu_m***=0x00000002.

4. Exercises

Exercise 1. Analyze control signals

Sketch the hardware that feeds the following select signals the two multiplexers of Figure 2. Table 2 provides the main modules and signals related to each control (select) signal. Focus on the main signals and do not go into many details, as it could get too complex.

- *mpc_alufunc_e*. This signal controls the 4-1 *_logic_out_e_31_0_* multiplexer.
- *mpc_sellogic_m*. This signal controls the 2-1 *_edp_alu_m_31_0_* multiplexer.

Table 2. Exercise 1: main modules and signals

`mpc_alufunc_e` **control signal**	
Module/Signal Name	**Description**
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
`mpc_sellogic_m` **control signal**	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
sel_logic_e	Selection signal for the *_edp_alu_m_31_0_* multiplexer at the E-Stage

Exercise 2. Adding new instructions to the soft-core: nand instruction

In this exercise you will expand the MIPSfpga (microAptiv) core to perform the nand instruction. This instruction reads two registers (identified by *rs* and *rt*), computes the NAND operation among them, and stores the result in the *rd* register of the register file. The format and description of a nand instruction are as follows:

- Format:

31 26	25 21	20 16	15 11	10 6	5 0
Op 000000	**Rs**	**Rt**	**Rd**	**Shamt** 00000	**Funct** 001110

- Description: *Reg[Rd] = (Reg[Rs] NAND Reg[Rs])*

Note that we are using a *special* Opcode and an unused Funct field (001110). Because the assembler does not support the new mnemonic, you must write the nand instruction in machine code in the assembly program. The following lines show how to include a nand instruction. You can find this program in the **main.c** file included in folder *Lab15_AND\Simulations\SimulationSources_NAND*, where everything is provided (the *.elf* file, the text files for initializing memory, etc.).

```
"   li $t3, 2;"         // Initialize first source operand
"   li $t4, 3;"         // Initialize second source operand
"   li $t6, 0;"         // Initialize destination operand
"   nop;"               // Avoid RAW dependencies
"   .word 0x018b700e;"  // nand $t6, $t4, $t3
"   b .;"               // Stay here
```

Table 3 includes the signals that we must modify for including a nand instruction. Below the table are hints to help you implement this instruction.

Table 3. Exercise 2: main signals related to the nand instruction

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
nand_instr	New signal, which is 1 when a nand instruction is decoded
spec_ri_e	Trigger *reserved instruction* trap for *special* instruction
sel_logic_e	Control signal selecting the Logic or the Arithmetic Unit result
alu_sel_e	Control signal selecting the ALU result
m14k_edp	**Execution datapath**
a_nand_b_e	New signal providing the result of the nand operation

- **Create a new signal *nand_instr***: Define a new signal in module **m14k_mpc_dec** that is 1 when a nand instruction is detected at decoding, and 0 otherwise. Provide this signal to all the modules that use it.
- **Disable exceptions:** For encoding the nand instruction, we use a *funct* (function) field that is not defined in the microAptiv (MIPS R3) ISA. Thus, this encoding typically triggers a *Reserved Instruction* exception. We must therefore disable this exception for this encoding (op = *000000* and *funct = 001110*). Signals *ri_e* and *ri_g_e* are set to 1 in module **m14k_mpc_dec** when a *Reserved Instruction* exception must be triggered. These two signals depend on several other signals. Specifically, signal *spec_ri_e* handles *SPECIAL* instructions. Change this signal for inhibiting the *Reserved Instruction* exception for a nand instruction.
- **Compute the result of the nand instruction (rs *NAND* rt) at the E-Stage:** Include a *nand* gate in Figure 2 and insert its output (signal *a_nand_b_e*) in the pipeline by including a new multiplexer that selects between the output of the 4:1 multiplexer (*logic_out_e*) or the result of the *nand* operation (*a_nand_b_e*).
- **Select the result of the Logic Unit to write to the register file (*edp_wrdata_w*):** We need to modify two signals:
    - *sel_logic_e*: Modify this signal so that in the M-Stage (*mpc_sellogic_m*, see Figure 2) the result of the Logic Unit is selected in the case of a nand instruction.
    - *alu_sel_e*: Modify this signal so that the result of the ALU is selected (in the M-Stage: *mpc_alusel_m*), for a nand instruction.
- **Set register file write strobe (*mpc_rfwrite_w*) and compute destination register (*mpc_dest_w*) for nand instruction**: Signal *mpc_rfwrite_w* depends on *vd_e*, whereas *mpc_dest_w* depends on *dest_e*. Both signals (*vd_e* and *dest_e*) are computed in module **m14k_mpc_dec**, and they need no changes for including the nand instruction.

To complete these task, do the following:

1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_nand**).
2. In the new folder, expand the capability of the MIPSfpga system so that it can write to the 7-segment displays on the Nexys4 DDR board, as explained in Lab 5.
3. In the new folder, expand MIPSfpga to implement nand, modifying the Verilog files containing the soft-core, following the instructions provided above. You will have to change the two Verilog files included in Table 3 (**m14k_mpc_dec** and **m14k_edp**) as well as the interface of the two top-modules **m14k_mpc** and **m14k_core** for communicating signals between modules.
4. Create a new Vivado project (**project_nand**) following the instructions provided in Step 1 - Lab 1, using the files from the new folder (**rtl_up_nand**).

5. Using the program shown above, and provided in folder *Lab15_AND\Simulations\SimulationSources_NAND*, debug your implementation with Vivado's XSIM simulator. Follow the steps explained in Section 4 of Lab 14 for configuring the simulator. Specifically, the fetch of the `nand` instruction is done around time 91440ns, thus configure the simulation runtime as explained in Lab 14. As for the waveform configuration file, you can use the file used for the `and` instruction as a starting point (*testbench_boot_behav.wcfg*), and then add the necessary signals depending on your implementation as explained in Exercise 1 of Lab 14 (Figures 13 and 14).
6. Finally, execute the program on the FPGA board. Follow the next steps:
   - Step 1 – Prepare the source files for execution on the board: Modify and analyze the program shown above for this exercise, provided in folder *Lab15_AND\Simulations\SimulationSources_NAND*, by commenting line "`b .;`" (as shown in Figure 15 of Lab 14). Then, re-generate the executable files, as explained in Section 7.2 of the Getting Started Guide, by using the *make* command (the **Makefile** is also provided in *Lab15_AND\Simulations\SimulationSources_NAND*).
     Then, analyze on your own the code after the commented branch. This code will output, on the 7-segment displays, the value of register $t4, which contains the result of the `nand` instruction.
   - Step 2 – Synthesize the new processor, as explained in Step 3 – Lab 1.
   - Step 3 – Program the FPGA board, as explained in Step 4 – Lab 1.
   - Step 4 – Download the program to the board, as explained in Step 3 – Section 2 of Lab 2: The program will start running on the board, and you will be able to see the value of register $t4 on the 7-segment displays.
   - Step 5 – Debug the program as explained in Step 4 – Section 2 of Lab 2: You can use the following sequence of commands in the debugger:

▪ `monitor reset halt`	(reset the processor)
▪ `b *0x80000214`	(set breakpoint before `nand` instruction)
▪ `c`	(the processor executes until the breakpoint)
▪ `i r`	(list the register file contents)
▪ `stepi`	(execute 1 instruction)
▪ `i r`	(list the register file contents)

Exercise 3. Adding new instructions to the soft-core: `seleqz/selnez` instructions

In ISA Release-6, MIPS replaced instructions `movz/movn` with the new instructions `seleqz/selnez`. MicroAptiv implements ISA Release-3, which includes the `movz/movn` instructions, thus in this exercise you will modify MIPSfpga to change the functionality of these latter two instructions.

Machine code format and functionality of `movz`:

- Format:

31 26	25 21	20 16	15 11	10 6	5 0
Op 000000	**Rs**	**Rt**	**Rd**	**Shamt 00000**	**Funct 001010**

- Description: *if GPR[Rt] = 0 then GPR[Rd] ← GPR[Rs]*

Expanded functionality so that `movz` emulates `seleqz`:

- Format: Same as `movz` instruction.
- Description: *if GPR[Rt] = 0 then GPR[Rd] ← GPR[Rs]*
  else then GPR[Rd] ← 0

Machine code format and functionality of `movn`:

- Format:

31 26	25 21	20 16	15 11	10 6	5 0
Op 000000	**Rs**	**Rt**	**Rd**	**Shamt 00000**	**Funct 001011**

- Description: *if GPR[Rt] ≠ 0 then GPR[Rd] ← GPR[Rs]*

Expanded functionality so that `movn` emulates `selnez` functionality:

- Format: Same as `movn` instruction.
- Description: *if GPR[Rt] ≠ 0 then GPR[Rd] ← GPR[Rs]*
  else then GPR[Rd] ← 0

Note that, as opposed to previous exercises, we are using a *special* Opcode and Funct field which are already in use, so we are not incorporating new instructions but modifying the functionality of already available instructions. Thus, we can use the old mnemonics (i.e. `movz` and `movn`) to include the new instructions (`seleqz` and `selnez`) in our code. The following lines show example code, also provided in the **main.c** file included in folder *Lab15_AND\Simulations\SimulationSources_SeleqzSelnez* where everything is provided (the *.elf* file, the text files for initializing memory, etc.).

```
"   li $t4, 6;"
"   li $t3, 0;"
"   li $t5, 0;"
"   li $t6, 0;"
"   li $t7, 0;"
"   li $t8, 0;"
"   nop;"
"   movz $t5,$t4,$t3;"
"   movz $t6,$t4,$t4;"
"   movn $t7,$t4,$t3;"
"   movn $t8,$t4,$t4;"
"   b .;"               // Stay here
```

As a first task, analyze the `movz/movn` instructions. For that purpose:

1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_SeleqzSelnez**).
2. In the new folder, expand the capability of the MIPSfpga system so that it can write to the 7-segment displays on the Nexys4 DDR board, as explained in Lab 5.
3. Create a new Vivado project (**project_SeleqzSelnez**) following the instructions provided in Step 1 - Lab 1, using the files from the new folder (**rtl_up_SeleqzSelnez**).
4. Analyze the `movz/movn` instructions theoretically (Table 4 provides the main modules and signals related to these instructions) and then simulate the program shown above and provided in *Lab15_AND\Simulations\SimulationSources_SeleqzSelnez*. Note that, in that program, the fetch of the first `movz` instruction is done around time 91520ns, thus configure the simulation to run for that time, as explained in Lab 14. Moreover, add the new text files and a waveform configuration file (add the necessary signals depending on your implementation).
5. Execute the program on the FPGA board. Follow the next steps:
   - Step 1 – Prepare the source files for execution on the board: Modify and analyze the program shown above for this exercise, provided in folder *Lab15_AND\Simulations\SimulationSources_SeleqzSelnez*, by commenting line "`b .;`" (as shown in Figure 15 of Lab 14). Then, re-generate the executable files, as explained in Section 7.2 of the Getting Started Guide, by using the *make* command (the **Makefile** is also provided in *Lab15_AND\Simulations\SimulationSources_SeleqzSelnez*).
     Then, analyze on your own the code after the commented branch. This code will output, on the 7-segment displays, the value of registers $t5-$t8, which contain the result of several `movz/movn` instructions.
   - Step 2 – Synthesize the new processor, as explained in Step 3 – Lab 1.
   - Step 3 – Program the FPGA board, as explained in Step 4 – Lab 1.
   - Step 4 – Download the program to the board, as explained in Step 3 – Section 2 of Lab 2: The program will start running on the board, and you will be able to see the value of registers $t5-$t8 on the 7-segment displays.

- Step 5 – Debug the program as explained in Step 4 – Section 2 of Lab 2: You can use the following sequence of commands in the debugger:

`monitor reset halt`	(reset the processor)
`b *0x80000220`	(set breakpoint before the first `movz` instruction)
`c`	(the processor executes until the breakpoint)
`i r`	(list the register file contents)
`stepi`	(execute 1 instruction)
`stepi`	(execute 1 instruction)
`stepi`	(execute 1 instruction)
`stepi`	(execute 1 instruction)
`i r`	(list the register file contents)

Table 4. Exercise 3: main modules and signals related to movz/movn instructions

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
mpc_cmov_e	Control signal: 1 for `movz/movn` and 0 otherwise
cmov_type_e	Control signal: 1 for `movz` and 0 for `movn`
m14k_edp	**Execution datapath**
a_and_b_e	Output of the *AND* operation between signals *logic_ain_e* and *logic_bin_e*. For `movz/movn`, this signal is selected as the value to write to the register file (*edp_wrdata_w*) at the W-Stage. Besides, for `movz/movn`: *logic_bin_e=logic_ain_e=rs*
kill_cmov_e	For `movz/movn`, this signal is used as the register file write strobe (*mpc_rfwrite_w*) at the W-Stage. It depends on signal *edp_cndeq_e*, which is the output of a comparator between *rt* and 0

Below are some hints to help you implement this instruction:

- **Modify signal *kill_cmov_e***: This signal enables/disables the write to the register file. For `movz/movn` instructions, the write to the register file depends on the result of the comparison. Instructions `seleqz/selnez` always write to the register file, thus this signal must be modified.
- **Select the value to write to the register file:** The value to write for a `movz/movn` instruction is provided from the *AND* gate output (*a_and_b_e*). For `seleqz/selnez` instructions, modify that logic for including a value of 0 when the condition is not met.

Now, complete the following tasks:

6. In the new folder (**rtl_up_SeleqzSelnez**), expand MIPSfpga to implement `seleqz/selnez` instructions, modifying the Verilog files of the soft-core, following the instructions provided above. You need to change **m14k_mpc_ctl** and **m14k_edp,** as well as the interface of the two top-modules **m14k_mpc** and **m14k_core** for communicating signals between modules.

7. Repeat steps 4 and 5 on the new instructions.
8. Finally, as another exercise, change the functionality of the new instructions, so that instead of writing a 0 when the condition is not met, *~rs* is written:

```
SELEQZ:
        if GPR[Rt] = 0          then GPR[Rd] ← GPR[Rs]
        else                    then GPR[Rd] ← ~GPR[Rs]
SELNEZ:
        if GPR[Rt] !=0          then GPR[Rd] ← GPR[Rs]
        else                    then GPR[Rd] ← ~GPR[Rs]
```

5. References

[1] "Digital Design and Computer Architecture", 2nd Edition. David Money Harris and Sarah L. Harris. Morgan Kaufmann, 2012.