MIPSfpga
by Imagination

Lab 17

Basic Instruction Flow – BEQ Instruction

Lab 17

Basic Instruction Flow – BEQ Instruction

1. Introduction

In this lab we analyze the flow of the branch if equal (`beq`) instruction along the microAptiv pipeline. This I-type instruction (`beq rs, rt, immediate`) reads two registers (*rs* and *rt*) and jumps to a target address (obtained from the constant provided in the instruction, *immediate*) if the value of the two registers are equal. Section 6.4.2 of *DDCA* [1] explains the `beq` instruction in detail, Figure 1 illustrates its format, and the instruction functionality can be expressed as follows:

```
If (Reg[rs]==Reg[rt])          PC ← PC + SignExtension(immediate<<2)
Else                           PC ← PC + 4
```

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

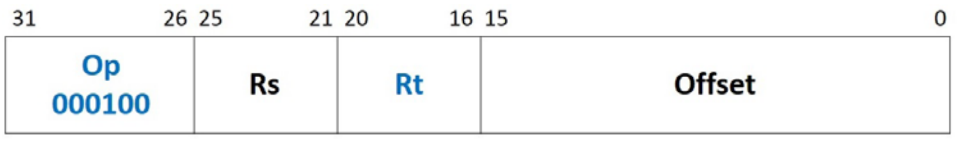

Figure 1. beq machine instruction format

2. Pipeline Stages

In this section we explain the execution of the `beq` instruction through each stage of the pipeline. We start by analyzing the E-Stage for the `beq` instruction and then we explain the Pre-I-Stage and the I-Stage operation (during the cycle when the `beq` instruction is in the E-Stage). Table 1 describes the main hardware modules and signals associated with each stage. The remaining section describes the stages in detail.

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

Pre-I-Stage	
Module/Signal Name	**Description**
m14k_edp	**Execution datapath**
edp_iva_p_n[31:0]	Next Program Counter

preiva_p[31:0]	First source for the multiplexer computing the Next Program Counter, computed as the Current Program Counter + 4
br_targ_e[31:0]	Second source for the multiplexer computing the Next Program Counter, computed at the E-Stage by the `beq` instruction
m14k_mpc_ctl	**Main pipeline control** – Control
mpc_eqcond_e	Control signal for the multiplexer computing the Next Program Counter, computed at the E-Stage by the `beq` instruction
I-Stage	
Module/Signal Name	**Description**
m14k_edp	**Execution datapath**
edp_iva_i[31:0]	Current Program Counter. This is the address of the instruction fetched at the I-Stage
incsum_e[31:0]	Current Program Counter (*edp_iva_i[31:0]*) + 4
E-Stage	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
cnd_eq_en	Condition will be true if *edp_cndeq_e* == 1
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
br_eq	Branch on equal
m14k_edp	**Execution datapath**
pc_hold[31:0]	Program Counter of the subsequent instruction to the `beq`
pcoffset_e [31:0]	Immediate value (*mpc_ir_e[15:0]*) times 4 and sign-extended
br_targ_e[31:0]	Target Address of the `beq` instruction
acmp_e[31:0]	First source for the comparator
edp_bbus_e [31:0]	Second source for the comparator
edp_cndeq_e	Result of the comparison, assigned to signal *mpc_eqcond_e*

a. E-Stage

Figure 2 illustrates the new structures included in the E-Stage for executing a `beq` instruction. A new adder is introduced for computing the target address of the branch (*br_targ_e*), where *pc_hold* is the address of the instruction following the branch, and *pcoffset_e* is the Sign Extension of the constant value provided by the instruction (*mpc_ir_e[15:0]*) multiplied by 4.

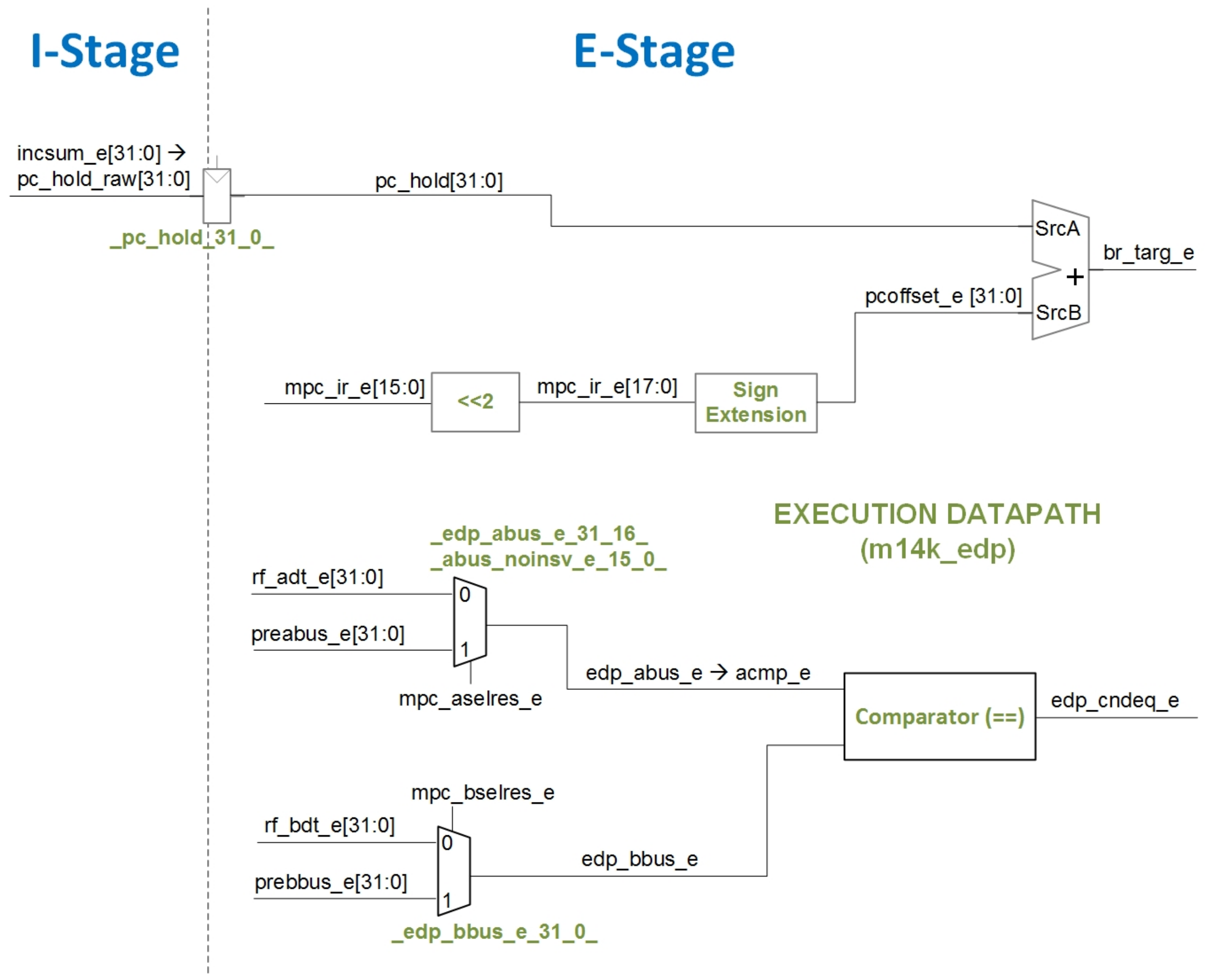

Figure 2. Main structures and signals involved in the E-Stage of a beq instruction. For the sake of simplicity, some elements, such as the register file, are not shown here (you can see them in Figure 3 of Lab 14).

The following code segment, extracted from lines 1139 to 1142 of module **m14k_edp**, implements the Sign Extension unit (shown in the figure as a black-box) that determines the offset for beq instructions, and the new adder, which computes the Branch Target Address:

```
assign pcoffset_e [31:0] =  mpc_br16_e ? { {15{mpc_imsgn_e}}, mpc_ir_e[15:0], 1'b0} :
                                       { {14{mpc_imsgn_e}}, mpc_ir_e[15:0], 2'b0};

assign br_targ_e [31:0] = pc_hold + pcoffset_e;
```

Notice that for 32-bit MIPS instructions, *mpc_br16_e* is 0. In addition to the new adder, a comparator is included in the E-Stage, which compares the two operands (*acmp_e* and *edp_bbus_e*). These operands can come from the Register File (*rf_adt_e* and *rf_bdt_e*) or, in the case of RAW hazard, from an earlier instruction (*pre_abus_e* and *pre_bbus_e*) . The following code segment, extracted from module **m14k_edp**, implements this comparator:

```
assign edp_cndeq_e = acmp_e == edp_bbus_e;
```

When a `beq` instruction is in the E-Stage, the default control flow (next PC = PC+4) must be changed if the condition is met (i.e. if the two operands are equal). This is carried out in the Pre-I-Stage, as explained in the next section.

b. Pre-I-Stage and I-Stage

As explained in Lab 14, during the Pre-I-Stage the next program counter (PC) is computed. By default, the next PC is computed as the current PC plus 4, where the current PC corresponds to the address of the instruction being fetched at the I-Stage (*edp_iva_i*). This default control flow may be altered by a conditional branch instruction executing at the E-Stage. Thus, in the Pre-I-Stage illustrated in Figure 3 (which is highly simplified), the next PC can be computed from two sources: the current PC + 4 (signal *preiva_p*), or the Branch Target Address (*br_targ_e*), computed at the E-Stage as illustrated in Figure 2.

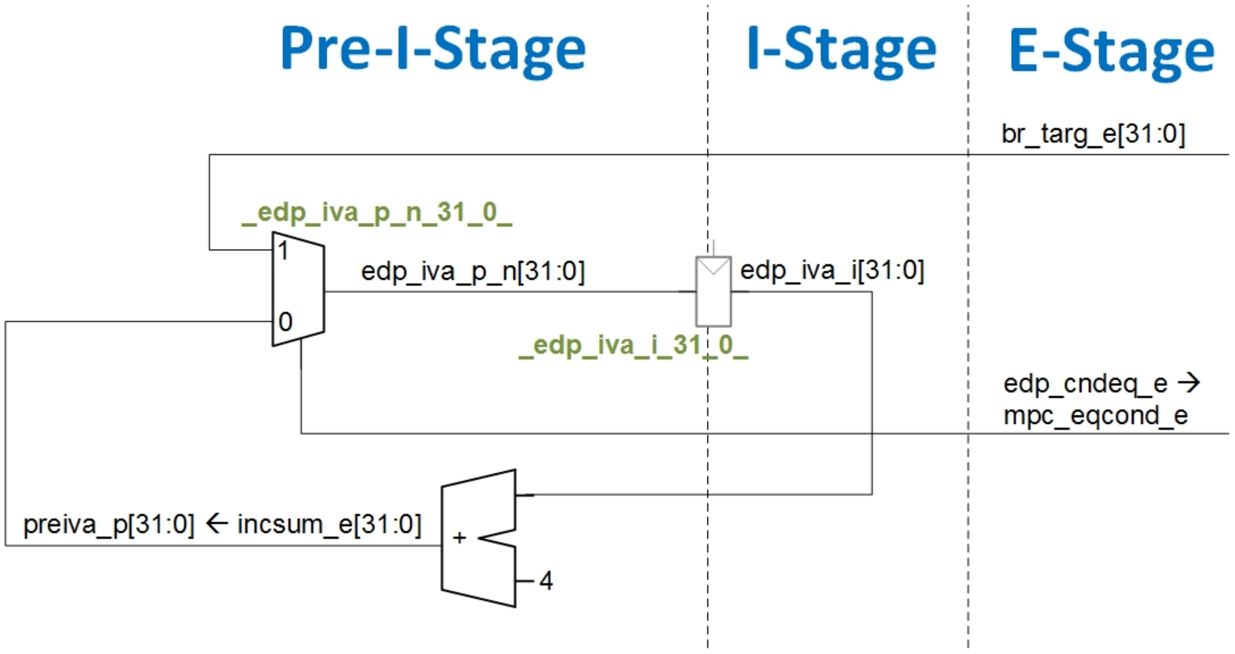

Figure 3. Main structures and signals involved in the Pre-I-Stage.

The output of the multiplexer (*_edp_iva_p_n_31_0_*) shown in Figure 3, which constitutes the next PC (*edp_iva_p_n*), is registered at the end of the Pre-I-Stage in the *_edp_iva_i_31_0_* register. The following code fragment, extracted from module **m14k_edp**, implements the adder, the multiplexer and the register from Figure 3:

```
assign incsum_e [31:0] = edp_iva_i[31:0] + 32'h0000_0004;
…
mvp_mux2 #(32) _edp_iva_p_n_31_0_(edp_iva_p_n[31:0],mpc_eqcond_e, preiva_p, br_targ_e);
…
```

```
mvp_cregister_wide #(32) _edp_iva_i_31_0_(edp_iva_i[31:0],gscanenable, (!icc_umipsmode_i
& mpc_run_ie) || ( icc_umipsmode_i & !(icc_imiss_i & ~(mpc_hw_ls_i | mpc_chain_hold)) &
(mpc_run_ie | mpc_fixupi)), gclk, edp_iva_p_exit);
```

Finally, the control signal for the multiplexer (*mpc_eqcond_e*) is computed at the E-Stage as shown in Figure 2. In our simplified explanation, this signal is 1 when the instruction at the E-Stage is a conditional branch and the condition is met, thus selecting the Branch Target Address as the next PC, and 0 otherwise, thus selecting the current PC + 4 as the next PC. Therefore, if the branch condition is met, the instruction fetched form the Instruction Memory in the next cycle is the one located at the BranchTarget Address. There is a caveat though: in the cycle when the `beq` instruction is in the E-Stage, the instruction located in the program after the branch is fetched in the I-Stage. This might seem a fault in the control flow, but it is actually correct, as MIPS architecture implements *delayed branches*. As such, the instruction after a `beq` (called the *branch delay slot*) is always executed, independent of whether the branch is taken or not.

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* [1] and illustrated in Figure 7.58 of that book, with regards to the `beq` instruction.

- In both designs, a new adder is included for computing the *Branch Target Address*. Note that in the processor from [1], this new adder is placed in the Decode Stage *to* avoid increasing the number of cycles that the pipeline must be stalled when a branch instruction is executed (in microAptiv the *DDCA* Decode and Execute Stages are merged in the E-Stage). Also observe that the sources for this adder are identical in both processors.
- The condition is also computed using analogous logic in both designs.
- Finally, observe that many analogies can be observed between the next PC computation from the processor in *DDCA* [1] (Figure 7.58 in that book) and the simplified Pre-I-Stage of microAptiv illustrated in this lab. For example, in *DDCA* [1], a multiplexer is used for selecting the next PC, where the control signal (*PCSrcD*) has a similar functionality as *mpc_eqcond_e*, and where both the current PC + 4 and the Branch Target Address are provided as inputs to the multiplexer. In that implementation, the output of the multiplexer is registered and used for indexing the Instruction Memory in the next cycle.

3. Example Simulation

In this section we illustrate the simulation of the `beq` instruction as it flows through the stages of the microAptiv pipeline. We first show how to view the flow in simulation and then analyze the results. Viewing the behavior of the different signals related to a beq instruction helps you understanding the theoretical explanations of Section 2.

a. Simulated program

You can create a new project in Vivado following the instructions provided in Section 4 of Lab 14, or you can reuse a project as explained in Section 3 of Lab 15.

In order to simulate a compiled program using Vivado's XSIM you can use a new project, following the instructions provided in Section 4.a of Lab 14, or you can reuse the project created in a previous lab as explained in Exercise 1 of Lab 14. In folder *Lab17_BEQ\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 includes a beq instruction that branches back to a label *a* if the contents of registers t1 and t2 are equal. Note that, by default, the assembler fills the *delay slot* with a nop instruction so you should not manually place a nop after a beq in an assembly program. 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 4).

```
…
80000204 <main>:
80000204:   24090004    li    t1,4
80000208:   240a0004    li    t2,4
8000020c <a>:
8000020c:   25290001    addiu t1,t1,1
80000210:   254a0001    addiu t2,t2,1
80000214:   1149fffd    beq   t2,t1,8000020c <a>
80000218:   00000000    nop
…
```

Figure 4. Example assembly program, with the beq instruction highlighted in blue.

Figure 1 shows the machine format of a beq instruction, which is explained extensively in Section 6.3.2 of DDCA [1]. For the beq instruction used in our program (beq t2, t1, 0xfffd - 0001 0001 0100 1001 1111 1111 1111 1101), each field is as follows:

- **Opcode**: 000100, indicates a beq instruction
- **Rs**: 01010, the first register, t2, which is register 10 (as defined by the MIPS ISA and in file "regdef.h")
- **Rt**: 01001, the second register, t1, which is register 9 (again, see the MIPS ISA or "regdef.h")
- **Offset**: 0xfffd, a constant that determines the Branch Target Address

Now, configure the simulation to run for 100000ns, following the instructions explained in Figure 8 of Lab 14. Moreover, use the waveform configuration provided for this example (*Lab17_BEQ\Simulations\testbench_boot_behav.wcfg*) as explained in Figure 7 of Lab 14, and use the text files provided for this example (*Lab17_BEQ\Simulations\SimulationSources*) as explained in Figure 6 of Lab 14.

b. Analysis of the simulation of the beq instruction

Figure 5 illustrates a timing diagram of the execution stages of the beq instruction.

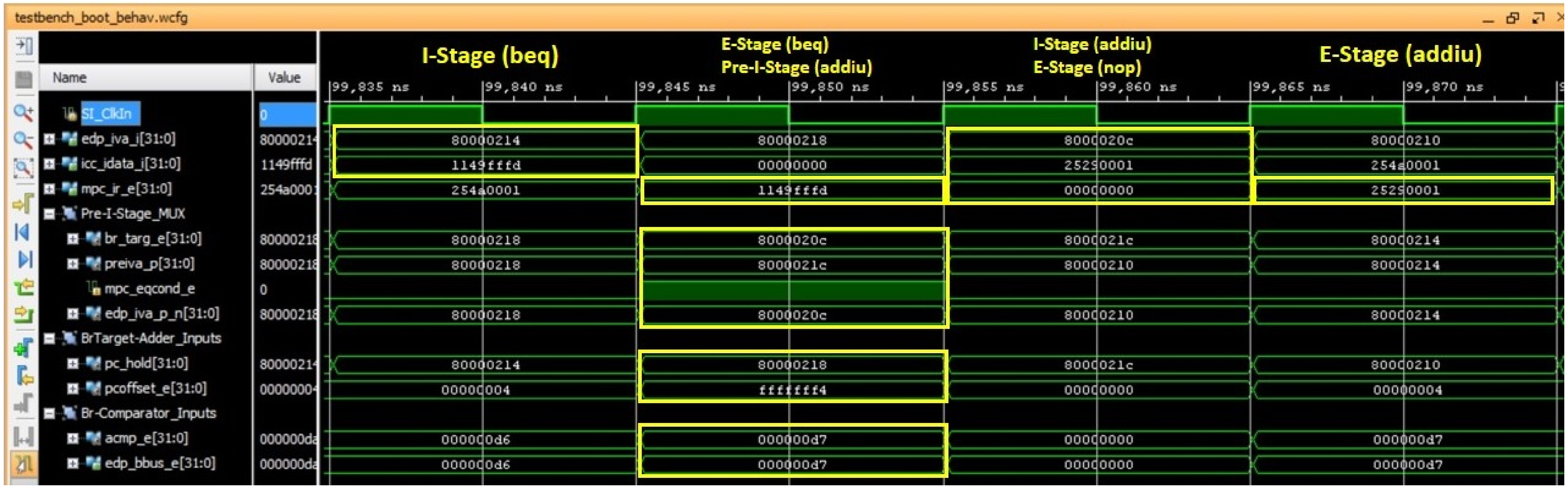

Figure 5. Timing diagram of the execution a beq instruction.

Next we detail the results obtained in the execution of the `beq` instruction.

- 1st cycle, I-Stage of the `beq` instruction:
    - The `beq` is fetched from address *edp_iva_i*=0x80000214 of the I$ (*icc_idata_i*=0x1149fffd).
- 2nd cycle, E-Stage of the `beq`, I-Stage of the `nop` (Delay Slot) and Pre-I-Stage of the `addiu`:
    - E-Stage of the `beq`:
        - The `beq` is stored in the Instruction Register (*mpc_ir_e*=0x1149fffd).
        - The Branch Target Address is computed: (*pc_hold*=0x80000218) + (*pcoffset_e*=0xfffffff4) = (*br_targ_e*=0x8000020c)
        - The branch condition is evaluated: (*acmp_e*=0xd7) == (*edp_bbus_e*=0xd7) → (*mpc_eqcond_e*=1)
    - I-Stage of the `nop` instruction following the `beq` (introduced by the assembler):
        - A `nop` is fetched from the I$ (*icc_idata_i*=0x00000000).
    - Pre-I-Stage of the `addiu`:
        - The Branch Target Address is selected as the next PC (*edp_iva_p_n*=0x8000020c).
- 3rd cycle, I-Stage of the `addiu` instruction (Branch Target) and E-Stage of the `nop`:
    - I-Stage of the `addiu`:
        - The instruction at the Branch Target Address (*edp_iva_i*=0x8000020c) is fetched from the I$ (*icc_idata_i*=0x25290001).
    - E-Stage of the `nop`:
        - The `nop` is stored in the Instruction Register (*mpc_ir_e*=0x00000000).
- 4th cycle, E-Stage of the `addiu` instruction.
    - The `addiu` instruction is executed (*mpc_ir_e*=0x25290001).

4. Exercises

Exercise 1. Analyze other branch and jump instructions

Simulate and analyze unconditional branches such as a `j` instruction or a `jal` instruction, and other conditional branches such as the `bne` instruction, and describe the main differences between those instructions and a `beq` instruction. You can first analyze them theoretically (Table 2 provides the main modules and signals related to each instruction) and then based on a simulation (you can use the Vivado project from Section 3).

Table 2. Exercise 1: main modules and signals

`j` instruction

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
mpc_jimm_e	Control signal computed at module **m14k_mpc_dec**, used for several tasks, which is 1 for a `j` instruction
m14k_edp	**Execution datapath**
jaddr_e	Jump Target Address
`jal` instruction	
Module/Signal Name	**Description**
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
mpc_jimm_e	Control signal computed at module **m14k_mpc_dec**, used for several tasks, which is 1 for a `jal` instruction
lnk31_e	Control signal computed at module **m14k_mpc_dec**, which is 1 for a `jal` instruction and which has an effect on *dest_e* and *maj_vd_e*
m14k_mpc_ctl	**Main pipeline control** – Control
mpc_lnksel_m	Control signal computed at module **m14k_mpc_ctl**, which is 1 for a `jal` instruction and is used in multiplexer *_sp_m_31_0_*
mpc_alusel_m	Control signal computed at module **m14k_mpc_ctl**, which is 0 for a `jal` instruction and is used in multiplexer *_asp_nodsp_m_31_0_*
m14k_edp	**Execution datapath**
jaddr_e	Jump Target Address
x_pc_hold	Address of the second instruction following the branch
`bne` instruction	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
cnd_neq_en	Condition will be true if edp_cndeq_e == 0
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
br_ne	Branch on not equal

Exercise 2. Analyze control signals

Sketch the hardware that feeds the following control signals, used at the Pre-I-Stage for selecting in multiplexer *_edp_iva_p_n_31_0_* the next PC (Figure 3). Table 3 provides the main modules and signals related to these control signals.

- *mpc_eqcond_e / eqcond_e*

Table 3. Exercise 2: main modules and signals

Module/Signal Name	Description
m14k_edp	**Execution datapath**
edp_cndeq_e	Result of the comparison
m14k_mpc_ctl	**Main pipeline control** – Control
cnd_eq_en	Condition will be true if edp_cndeq_e == 1
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
br_eq	Branch on equal

Exercise 3. Adding new instructions to the soft-core: Branch if Equal Compact (beqc)

In ISA Release-6, MIPS included new Branch Compact instructions, and, by contrast, removed Branch Likely instructions. The purpose of this lab is to change the functionality of instruction `beql` (i.e. to remove the *Branch if Equal Likely* instruction) for the functionality of a *Branch if Equal Compact* instruction (`beqc`). The machine code format and functionality of the `beql` instruction is the following:

- Format:

31 … 26	25 … 21	20 … 16	15 … 0
Op 010100	**Rs**	**Rt**	**Offset**

- Description:

```
If (Reg[rs]==Reg[rt])       PC ← PC + SignExtension(immediate<<2)
Else                        PC ← PC + 4 // Annul the Delay Slot
```

The modified `beql` function effectively becomes the `beqc` instruction, but without changing the maching instruction encoding. It functions as follows:

```
If (Reg[rs]==Reg[rt])       PC ← PC + SignExtension(immediate<<2) //
                            Annul the Delay Slot
Else                        PC ← PC + 4
```

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 a new instruction but modifying the functionality of an already available one. Thus, we can use the old mnemonic (i.e. `beql`) to include the new instruction (`beqc`) in our code. The following lines show example code, also provided in the **main.c** file included in folder *Lab17_BEQ\Simulations\SimulationSources_BEQC* where everything is provided (the *.elf* file, the text files for initializing memory, etc.). Note that the sequence repeats indefinitely; you should analyze the second iteration, where the I$ will always hit. Note also that the program uses the assembler directive "*.set noreorder*", which tells the assembler that the programmer is in control and thus it must not move instructions about (in Lab 13 we use this directive for the first time). That way, the branch delay slot is filled with useful instructions, selected by the programmer, rather than `nop` instructions.

```
".set noreorder;"

"   start: li $t1, 1;"
"   li $t2, 1;"
"   li $t3, 1;"
"   li $t4, 2;"

"   beql $t4, $t3, a;"
"   add $t1, $t1, $t1;"
"   add $t2, $t2, $t2;"

"   a: beql $t4, $t4, b;"
"   add $t2, $t2, $t2;"
"   add $t1, $t1, $t1;"

"   b:"
"   b start;"
```

As the first task, analyze the `beql` instruction. For that purpose:

1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_beqc**).
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_beqc**) following the instructions provided in Step 1 - Lab 1, using the files from the new folder (**rtl_up_beqc**).
4. Analyze the `beql` instruction theoretically (Most signals and modules related to this instruction are identical to the `beq` instruction studied above. Table 4 provides the signals not included in Table 1). Simulate the program shown above, and also provided in *Lab17_BEQ\Simulations\SimulationSources_BEQC*. Configure the simulation runtime 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 *Lab17_BEQ\Simulations\SimulationSources_BEQC*, by commenting line "`b start;`". 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 *Lab17_BEQ\Simulations\SimulationSources_BEQC*).
     Then, analyze on your own the code after the commented branch. This code will output, on the 7-segment displays, the value of registers $t1 and $t2.
   - 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 $t1 and $t2 on the 7-segment displays.
   - Step 5 – Debug the program as explained in Step 4 – Section 2 of Lab 2.

Table 4. Exercise 3: main signals related to the beql instruction

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
br_likely_e	1 if a branch likely instruction is decoded
m14k_mpc_ctl	**Main pipeline control** – Control
annul_ds_i	Annulled Delay Slot in I-stage

Below are some hints to help you implement this instruction:

- Instruction `beql` uses a specific signal, called ***annul_ds_i***, which annuls the DS when the condition is not met. You can use the same signal for the `beqc` implementation.

To complete this exercise:

6. In the new folder (**rtl_up_beqc**), expand MIPSfpga to implement a `beqc` instruction, modifying the Verilog files of the soft-core following the instructions provided above. You will only have to change module **m14k_mpc_ctl**.
7. Repeat steps 4 and 5 on the new instruction.

5. References

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