MIPSfpga by Imagination

Lab 14

Basic Instruction Flow – ADD Instruction

These materials produced in association with Imagination.
Join our University community for more resources.
community.imgtec.com/university

Lab 14
Basic Instruction Flow – ADD Instruction

1. Overview

This is the first of five labs describing the flow of instructions through the MIPSfpga (microAptiv) pipeline. We analyze a breadth of instructions, including an arithmetic operation (`add`, Lab 14), logical operation (`and`, Lab 15), memory access instruction (`lw`, Lab 16), and branch instruction (`beq`, Lab 17). Moreover, in Lab 18, we analyze the Hazard Unit provided by microAptiv. The labs are intended for students who have a basic understanding of MIPS instructions and microarchitecture – i.e., chapters 6 and 7 of the text *Digital Design and Computer Architecture* [2] or equivalent material. Before beginning these labs, it is recommended to study chapter 2 of document [1], and chapters 6 and 7 of [2], which we will refer to as *DDCA* – we frequently reference this book in the explanation of some concepts.

During these five labs, we aim to explain the microAptiv pipeline while still recognizing the inherent complexity and sophistication of a commercial processor. We balance clarity with understanding by abstracting away some details. For example, we may abstract away the exact combinational path of a given signal assignment. Interested readers may themselves dig into the details of the exact path of the assignment, as desired. We use the following notation when talking about signal assignment:

- ***Signal*$_2$ = *Signal*$_1$**: Signal$_2$ is assigned the value of Signal$_1$, possibly through a combination of logic gates, multiplexers, or other signals.
- ***Signal*$_2$ <= *Signal*$_1$**: Signal$_2$ is assigned the value of Signal$_1$ after the next clock edge.

The lab discussions indicate which MIPSfpga modules are used. It is strongly recommended that, while reading and working through the lab, you open MIPSfpga in Vivado and view the accompanying hardware modules and signals being discussed. MIPSfpga module names are in bold (i.e., **m14k_top**), MIPSfpga signal names are italicized (i.e., *SI_ClkIn*), and instructions are in code font (i.e., `add`).

It is also strongly recommended that students preserve a copy of the original package, as there are some files that may change during the simulations and exercises.

Before starting the description of specific instructions, Figure 1 shows the simplified top-level block diagram for the microAptiv UP processor, and Figure 2 shows the simplified module hierarchy.

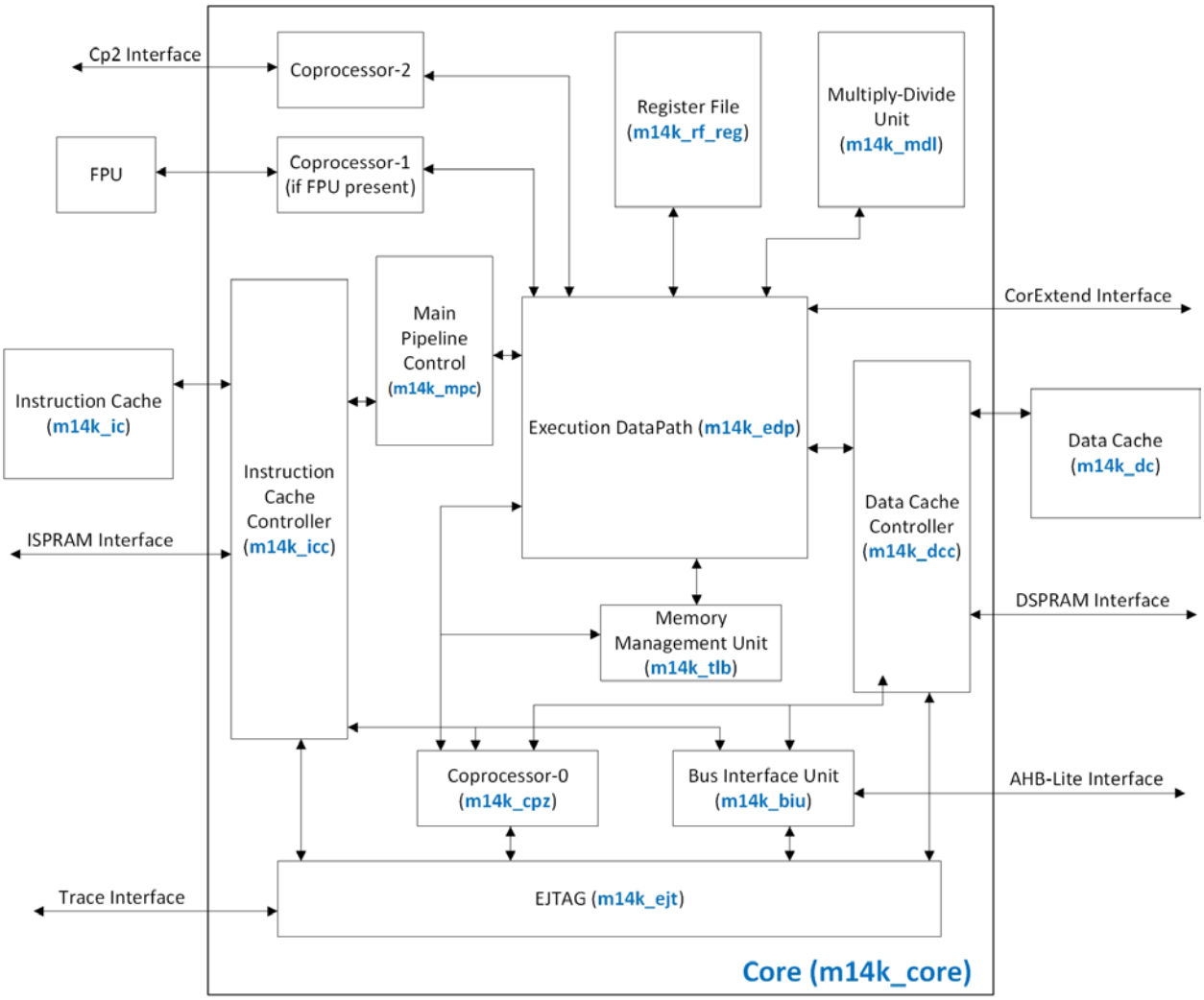


Figure 1. MicroAptiv UP CPU-level simplified block diagram [3].

After being fetched from the instruction cache (I$), implemented within the Instruction Cache module (**m14k_ic**) and controlled by the Instruction Cache Controller (**m14k_icc**), instructions are decoded within the Main Pipeline Control module (**m14k_mpc**) and executed within the Execution DataPath module (**m14k_edp**). The operands are obtained from the Register File module (**m14k_rf_reg**). After execution, arithmetic-logic instructions write their result back to the register file; load/store instructions access the data cache (D$), implemented within the Data Cache module (**m14k_dc**) and controlled by the Data Cache Controller (**m14k_dcc**); and branch/jump instructions modify the program counter.

We next provide a short description of the main modules of the block diagram used in this and the following labs. We list the module name shown in the block diagram followed by the Verilog module name in parentheses.

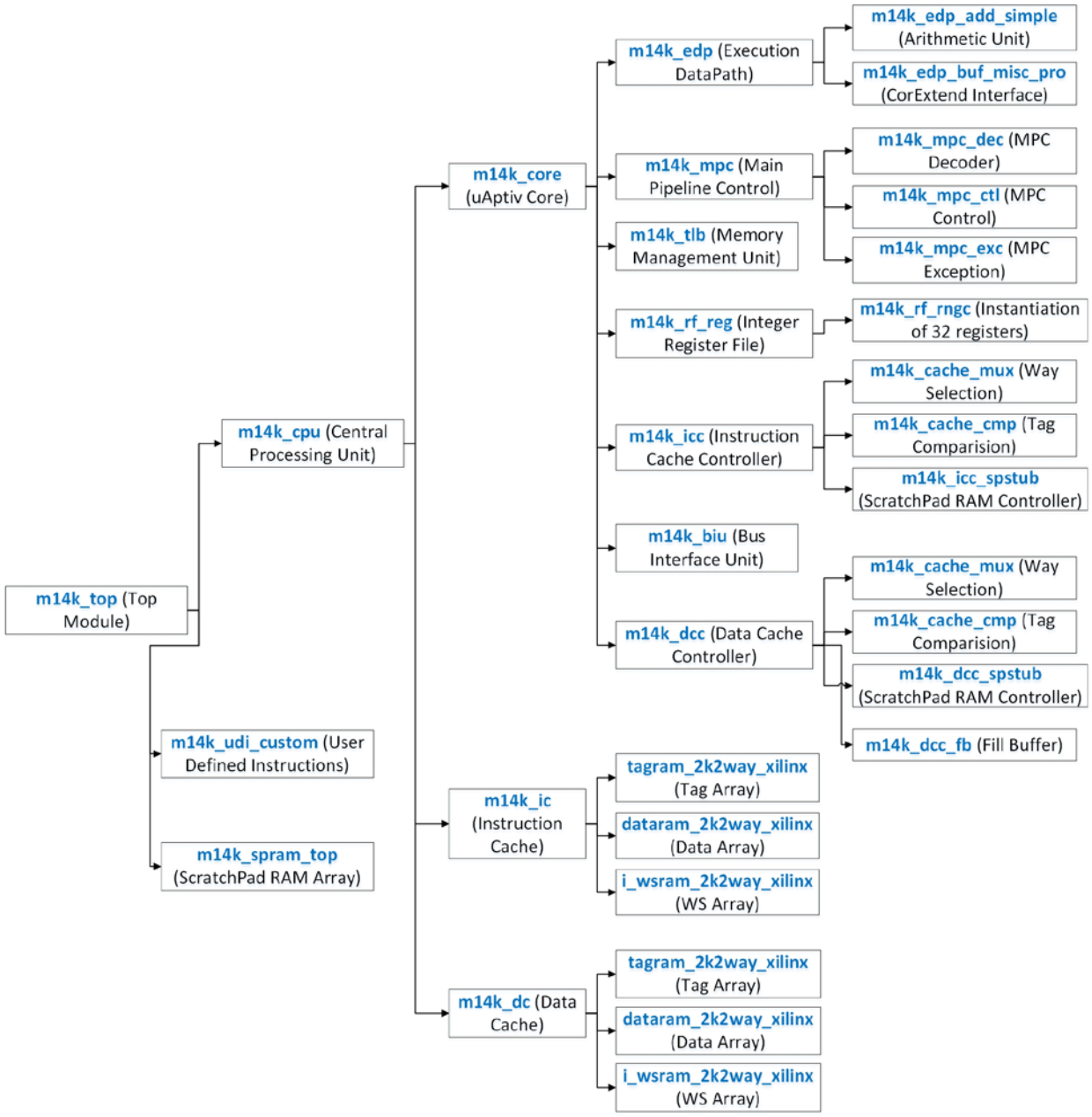


Figure 2. MicroAptiv UP simplified module hierarchy.

- **Main Pipeline Control** (**m14k_mpc**): The main pipeline control module (MPC) recognizes and manages dependencies in the pipeline. It decodes each instruction and checks if any interlocks are needed to block the issue of the next instruction. MPC also generates control signals for the pipeline and processes exceptions. The MPC block also receives the stall conditions from the other functional blocks and manages the progression of the pipeline.

- **Execution Data Path** (**m14k_edp**): The execution data path (EDP) implements the fundamental 32-bit integer data operations of the microAptiv UP core. In addition to data calculations, it contains logic for branch determination, branch target calculation, and load alignment. The integer functional units contained within the EDP for performing integer operations include:
    - **Arithmetic Logic Unit (ALU)** for performing arithmetic and bitwise logical operations. It also performs load/store address calculation and branch target calculation.
    - **Address unit** for calculating the next PC and select signals for the next PC multiplexers.
    - **Load Aligner** for aligning non-word-aligned loads.
    - **Shifter and Store Aligner** for shifting or aligning non-word-aligned loads.
    - **Branch condition comparator** to compare the two operands in conditional branches.
    - **Bypass multiplexers** to forward result to prior pipeline stages.
- **Register File** (**m14k_rf_reg**): The RF block contains the integer register file. The block consists of a 31-entry by 32-bit register file with 2 read ports and 1 write port for implementing the register file defined by the MIPS32 architecture. Only 31 of the 32 registers are included because register 0 always reads as zero and writes to it are ignored. The register file is read and written on the rising edge of the clock.
- **Memory Management Unit** (**m14k_tlb**): The memory management unit (MMU) contains a translation lookaside buffer (TLB), smaller instruction and data micro TLBs, and additional control logic. The TLBs convert virtual addresses (program, process or thread-relative) and the ASID (process or thread ID) into physical memory addresses. In place of the TLB-based memory management unit, an optional fixed mapping translation (FMT) module is also provided.
- **Instruction Cache Controller** (**m14k_icc**): The ICC block contains the logic to control accesses to the instruction cache (I$). The cache arrays are located in the I$ module while the tag comparison logic and cache data multiplexers are located in the ICC module. The ICC module generates the control signals needed to determine when to perform lookups, indexed reads, and fills to the I$ block.
- **Instruction Cache** (**m14k_ic**): This block contains the I$ arrays: data, tag, and way-select arrays.
- **Data Cache Controller** (**m14k_dcc**): Like the ICC, the DCC module manages the reading and writing of the data cache (D$). It supports the same cache characteristics as the ICC, although the Instruction and Data caches can be independently configured. In addition, the DCC block manages stores to the data cache. Since store data cannot be written to

the cache until cache lookup is done, the DCC block maintains a buffer for the store data.

- **Data Cache** (**m14k_dc**): This block contains the D$ arrays: data, tag, and way-select arrays.
- **Bus Interface Unit** (**m14k_biu**): The BIU contains the logic to drive the AHB-Lite interface signals. Additionally, it contains the implementation of the 32-byte collapsing write buffer. The purpose of this buffer is to store and combine write transactions before issuing them at the external interface. When using a write-through caching policy, the write buffer can significantly reduce the number of write transactions on the external interface as well as reduce the amount of stalling in the core due to multiple writes in a short period of time. When a write-back caching policy is used, the write buffer gathers the contents of a dirty cache line when it is evicted.
- **Coprocessor Zero** (**m14k_cpz**): CPZ is the system control coprocessor (also referred to as COP0), which supports the virtual memory system and exception handling. CPZ also contains registers that hold information about the state of the processor and that are used for handling exceptions.

Figure 3 shows the processor that you will obtain after completing these five labs (Labs 14 to 18). Being such a big and detailed figure, it may be difficult to observe and analyze the details in a printed version of the document. Obviously, the reader can easily zoom into the figure in the Word document. Moreover, source files for all the figures used in the explanations are provided. For example, Figure 3 is available in folder *Part3_Core**Lab14_ADD**Figures* in several formats, such as Visio, PDF, JPEG and PNG. As an interesting first exercise, you can broadly compare Figure 3 with the processor from Figure 7.58 of [2], trying to detect the main coincidences and differences among both implementations.

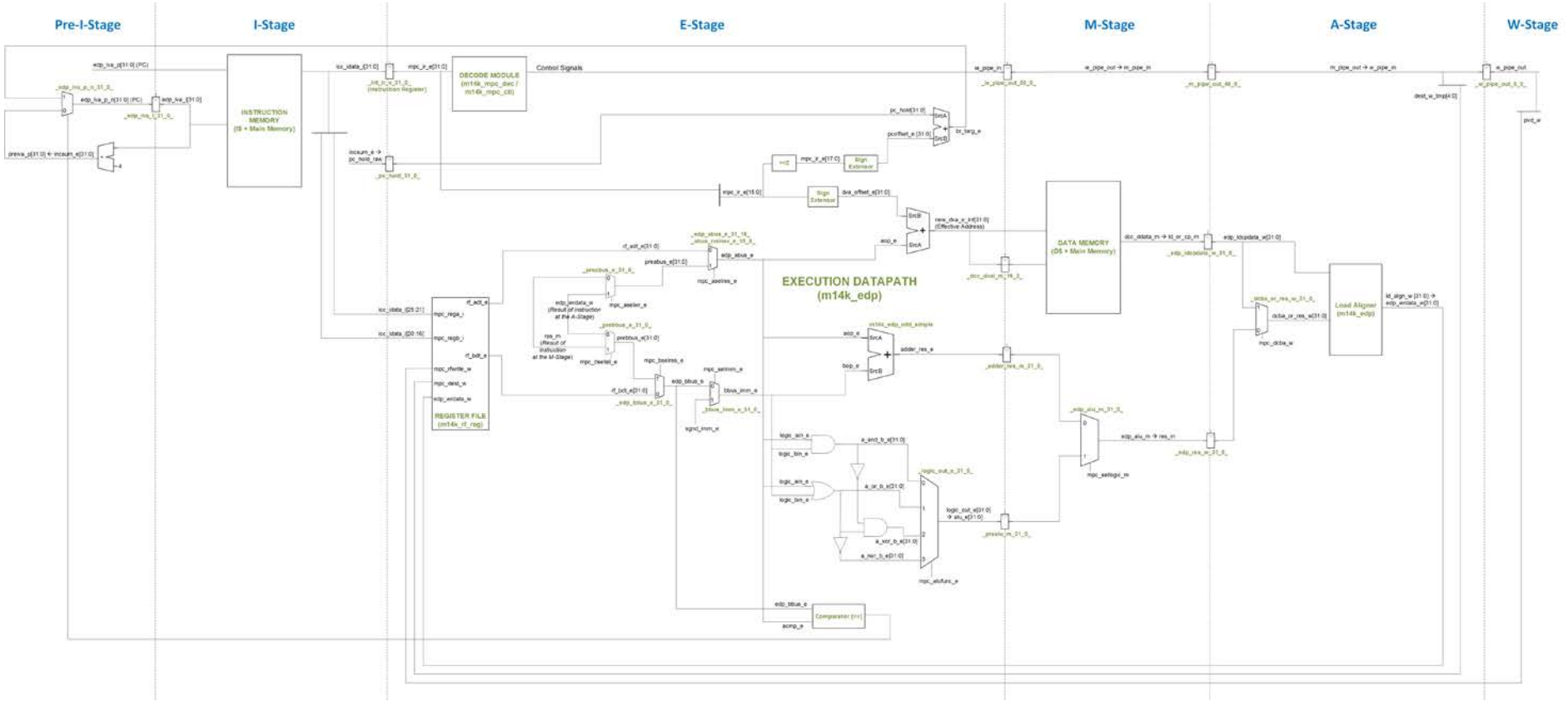


Figure 3. Resulting pipeline after Labs 14-18.

2. Introduction

In this lab we analyze the flow of the `add` instruction along the microAptiv pipeline. This R-type instruction (format: `add rd, rs, rt`) reads the values stored in two registers (the *source* registers, *Rs* and *Rt*), adds the two values (*Reg[Rs] + Reg[Rt]*), and writes the result to a third register (the *destination* register, *Rd*). Section 6.2 of [2] explains the `add` instruction in detail, Figure 4 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 `add` instruction in each stage of the pipeline (Section 3). Section 4 walks through a detailed simulation of the `add` instruction through the pipeline, and Section 5 provides exercises that explore and expand the pipeline using various arithmetic instructions.

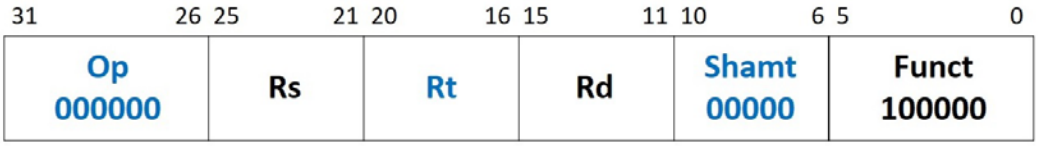


Figure 4. Machine instruction format of an `add` instruction.

3. Pipeline Stages

In this section we explain the execution of the `add` instruction through each stage of the pipeline. As explained in Chapter 2 of [1], microAptiv is divided into 5 stages: Instruction (I), Execution (E), Memory (M), Align (A), and Writeback (W) Stages. It also includes one background stage, the Pre-I-Stage. Table 1 briefly describes each stage, and Table 2 describes the main hardware modules and signals associated with each stage. The remaining section describes the stages in detail from the point of view of the `add` instruction.

Table 1. MIPSfpga pipeline

Number	Stage	Name	Description
0	Pre-I	Pre-Instruction	The processor computes the address of the next instruction
1	I	Instruction	The processor fetches an instruction
2	E	Execution	The processor decodes the instruction, fetches operands from the register file and performs an ALU operation (for example, addition, subtraction, or memory address calculation)
3	M	Memory	If applicable, the processor accesses a memory operand
4	A	Align	If applicable, loaded data is aligned to its word boundary
5	W	Writeback	If applicable, the processor writes the result to the register

			file

Table 2. MIPSfpga pipeline: main modules and signals related to the add instruction

Pre-I-Stage	
Module/Signal Name	**Description**
m14k_edp	**Execution datapath**
edp_iva_p[31:0]	Next PC, which contains the address of the instruction to fetch in the next cycle at the I-Stage
m14k_icc	**Instruction cache controller**
icc_dataaddr[13:2]	Index to the instruction memory
I-Stage	
Module/Signal Name	**Description**
m14k_edp	**Execution datapath**
edp_iva_i[31:0]	Current PC, which contains the address of the instruction being fetched at the I-Stage
m14k_icc	**Instruction cache controller**
icc_idata_i[31:0]	Fetched instruction
m14k_rf_reg	**Register file**
rf_adt_e[31:0]	First read data
rf_bdt_e[31:0]	Second read data
E-Stage	
Module/Signal Name	**Description**
m14k_rf_reg	**Register file**
rf_adt_e[31:0]	First data read from the register file
rf_bdt_e[31:0]	Second data read from the register file
m14k_mpc_ctl	**Main pipeline control** - Control
ie_pipe_in[50:0]	Control signals
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
mpc_ir_e[31:0]	Instruction register
m14k_edp	**Execution datapath**
aop_e[31:0]	Adder Source A
bop_e[31:0]	Adder Source B
adder_res_e[31:0]	Result of the addition
mpc_aselres_e	Control signal of multiplexers *_edp_abus_e_31_16_*//*_abus_noinsv_e_15_0_*, computed at **m14k_mpc_dec**
mpc_bselres_e	Control signal of multiplexer *_edp_bbus_e_31_0_*, computed at **m14k_mpc_dec**
mpc_selimm_e	Control signal of multiplexer *_bbus_imm_e_31_0_*, computed at **m14k_mpc_dec**
M-Stage	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
ie_pipe_out[50:0]	Control signals

m_pipe_in[46:0]	
m14k_edp	**Execution datapath**
adder_res_m[31:0]	Result of the addition from the E-Stage
res_m[31:0]	Result of the addition to the A-Stage
A-Stage	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
m_pipe_out[46:0]	Control signals
w_pipe_in[5:0]	
m14k_rf_reg	**Execution datapath**
mpc_dest_w[8:0]	Register number to write
edp_wrdata_w[31:0]	Data to write
mpc_rfwrite_w	Write enable

a. Pre-I-Stage

The Pre-Instruction (Pre-I) Stage is a background stage that computes the next program counter (signal *edp_iva_p[31:0]*, computed within **m14k_edp**), used for indexing the instruction memory (signal *icc_dataaddr[13:2]*, computed within **m14k_icc**). The inputs to the instruction memory must be ready at the clock edge, thus, during the Pre-I-Stage, the next program counter (next PC) is directly provided to the instruction memory so that it can be read along the I-Stage, as indicated by the following simplified signal assignments (available in several places within modules **m14k_edp** and **m14k_icc**): *icc_dataaddr[13:2] = ival_munge [13:2] = edp_ival_p[13:2] = edp_cacheiva_p[13:2] = edp_iva_p[13:2]*.

The hardware associated with the Pre-I-Stage is in the **m14k_edp** module. By default, the next PC is computed as the current PC (*edp_iva_i[31:0]*) plus 4 (line 1192 of **m14k_edp** module):

```
assign incsum_e [31:0] = edp_iva_i[31:0] + 32'h0000_0004;
```

Note that the current PC (*edp_iva_i[31:0]*) contains the address of the instruction at the I-Stage. In Lab 17, where we analyze the branch-if-equal (beq) instruction, we discuss the Pre-I-Stage in detail and show other possible sources for the next PC.

b. I-Stage

The Instruction (I) Stage (Figure 5) fetches the instruction from instruction memory based on the next PC, generated in the previous cycle within module **m14k_edp**. The instruction memory of microAptiv is made up of an instruction cache (implemented within module **m14k_ic**) and a main memory, accessed in case of a miss to the instruction cache (I$). The fetched instruction is found in signal *icc_idata_i[31:0]*, with a latency of 1 cycle upon an I$ hit or *n* cycles upon an I$ miss, from where it is registered to the E-Stage (*mpc_ir_e[31:0] <= icc_idata_i[31:0]*). Part 4 of these labs analyzes microAptiv's memory system in detail, including cache memory accesses.

During the I-Stage, the two source register numbers to read along the E-Stage are provided to the register file (implemented within module **m14k_rf_reg**) as indicated by the following simplified signal assignments: *mpc_rega_i[4:0] = icc_idata_i[25:21]* (line 1709 of module **m14k_mpc_dec**) and *mpc_regb_i[4:0] = icc_idata_i[20:16]* (line 1715 of module **m14k_mpc_dec**). Note that, similarly to the instruction memory, all register file (RF) read inputs must be ready at the clock edge.

c. E-Stage

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

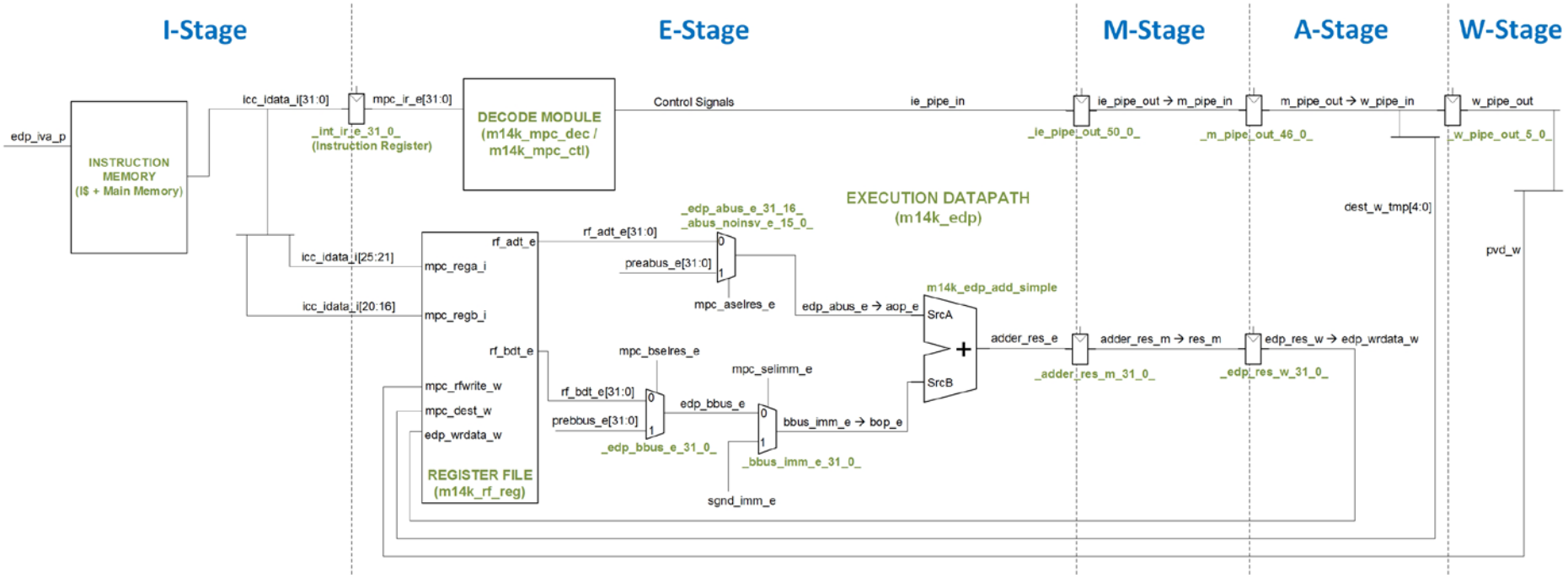


Figure 5. Main structures and signals involved in the E-Stage of an add instruction.

Figure 5 illustrates the main structures and signals involved in the E-Stage of an `add` instruction. The main hardware modules used are: **m14k_rf_reg** and **m14k_rf_rngc** (instantiated within module **m14k_rf_reg** as M14K_RF_REG_MODULE0), which implement the RF, **m14k_edp** and **m14k_edp_add_simple**, the execution datapath, and **m14k_mpc_ctl** and **m14k_mpc_dec**, decoding hardware. As stated earlier, the instruction register (IR) holds the current instruction (signal *mpc_ir_e[31:0]*), as shown in Figure 5.

i. Fetch Two Registers

Before performing the add operation, the RF, implemented in modules **m14k_rf_reg** and **m14k_rf_rngc**, produces two operands using the register numbers encoded in the instruction and provided by signal *icc_idata_i* from the I-Stage, as described in Section 3.b. The two registers read from the register file are in signals *rf_adt_e* and *rf_bdt_e***.** Note that two registers are always read from the RF, regardless of the instruction being executed.

The two signals *rf_adt_e* and *rf_bdt_e* are provided to the adder (module **m14k_edp_add_simple**) through several multiplexers (muxes). Figure 5 shows three of these multiplexers: muxes *_edp_abus_e_31_16_/_abus_noinsv_e_15_0_* and *_edp_bbus_e_31_0_* are used to forward an operand from a subsequent stage, and mux *_bbus_imm_e_31_0_* chooses between a register or immediate operand for the second source (srcB) – for example, `addi` requires a constant, or *immediate*, value. The following code segments, extracted from module **m14k_edp** (starting at lines 927, 936 and 1107), correspond to the multiplexers shown in Figure 5:

```
mvp_mux2 #(16) _edp_abus_e_31_16_(edp_abus_e[31:16],mpc_aselres_e, rf_adt_e[31:16],
preabus_e[31:16]);
mvp_mux2 #(16) _abus_noinsv_e_15_0_(abus_noinsv_e[15:0],mpc_aselres_e, rf_adt_e[15:0],
preabus_e[15:0]);
mvp_mux2 #(5) _abus_noinsv_imm_e_4_0_(abus_noinsv_imm_e[4:0],mpc_dec_imm_rsimm_e,
abus_noinsv_e[4:0], mpc_dec_imm_apipe_sh_e[4:0]);
mvp_mux2 #(5) _edp_abus_e_4_0_(edp_abus_e[4:0],mpc_dec_insv_e, abus_noinsv_imm_e[4:0],
dsp_dspc_pos_e[4:0]);
mvp_mux2 #(11) _edp_abus_e_15_5_(edp_abus_e[15:5],mpc_dec_imm_rsimm_e,
abus_noinsv_e[15:5], mpc_dec_imm_apipe_sh_e[15:5]);
…
mvp_mux2 #(32) _edp_bbus_e_31_0_(edp_bbus_e[31:0],mpc_bselres_e, rf_bdt_e, prebbus_e);
…
mvp_mux2 #(32) _bbus_imm_e_31_0_(bbus_imm_e[31:0],mpc_selimm_e, edp_bbus_e, sgnd_imm_e);
```

ii. Perform the Addition

After obtaining the two source operands, the E-Stage performs the addition using module **m14k_edp_add_simple**. The adder receives its two operands via signals *aop_e* and *bop_e* and returns the result of the addition via signal *adder_res_e*, which is then registered to the next stage (M-Stage): *adder_res_m* **<=** *adder_res_e*. The following code segment, extracted from module **m14k_edp** (line 1130), corresponds to the adder instantiation:

```
`M14K_EDP_ADD edp_add( .a(aop_e),
```

```
.b(bop_e),
.ci(mpc_subtract_e),
.s(adder_res_e),
.co(car31_e),
.c31(c31_e));
```

iii. Decode the Instruction

The final task of the E-Stage is to decode the instruction. Instruction (signal *mpc_ir_e*) is *decoded* by the decoder module **m14k_mpc_dec**, which generates control signals used by the rest of the pipeline. These control signals include multiplexer select signals, register write signals, etc. The control signals are registered by the pipeline registers as shown in Figure 6.

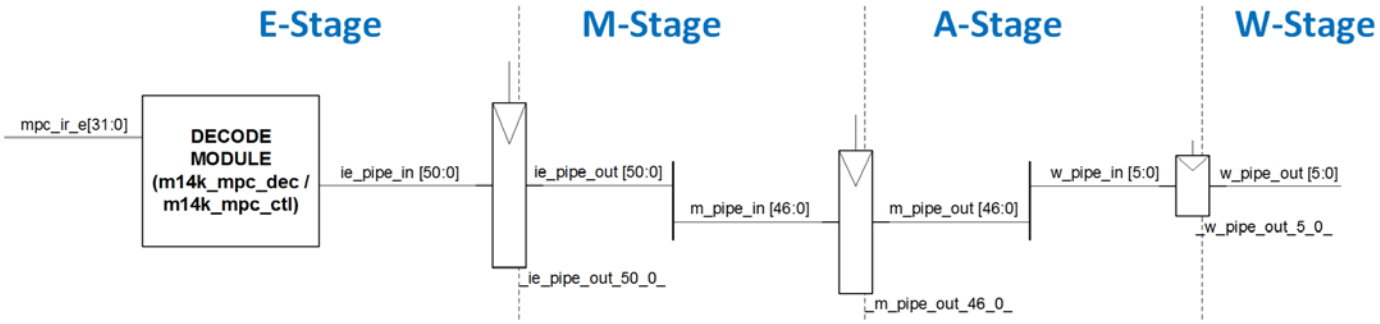


Figure 6. Control Pipeline Registers

These registers can be found at the beginning of the **m14k_mpc_ctl** module (line 1751), as shown in the next code fragments:

```
mvp_cregister_wide #(51) _ie_pipe_out_50_0_(ie_pipe_out[`M14K_IE_RANGE],gscanenable,
mpc_run_ie, gclk, ie_pipe_in);
…
mvp_cregister_wide #(47) _m_pipe_out_46_0_(m_pipe_out[`M14K_M_RANGE],gscanenable,
mpc_run_m, gclk, m_pipe_in);
…
mvp_cregister_wide #(6) _w_pipe_out_5_0_(w_pipe_out[`M14K_W_RANGE],gscanenable,
mpc_run_w, gclk, w_pipe_in);
```

Some control signals are directly used in the E-Stage, such as *mpc_aselres_e*, *mpc_bselres_e*, and *mpc_selimm_e*. The remaining control signals, which are needed in subsequent pipeline stages, are assigned to signal *ie_pipe_in*, within module **m14k_mpc_ctl**, and then registered to *ie_pipe_out* at the end of the cycle. Note that *ie_pipe_in* is composed of 51 bits ("*define M14K_IE_RANGE 50:0*" in file **m14k_const.vh**), divided into different fields that are defined in lines 558 to 590 within file **m14k_const.vh**. We discuss some of the fields of this register, *ie_pipe_in* and you can examine other fields of this signal as desired.

d. M-Stage

The Memory (M) Stage performs the data memory access, which consists of a data cache (D$), implemented within module **m14k_dc**, and a main memory accessed in the case of a cache

miss. Thus, for an add instruction, this stage does nothing. The control signals that are used in this stage are obtained from signal *ie_pipe_out*, whereas those that will be needed in the A-Stage or in the W-Stage are registered from *m_pipe_in* to *m_pipe_out* (2), within module **m14k_mpc_ctl**, as shown in the next code fragment:

```
mvp_cregister_wide #(47) _m_pipe_out_46_0_(m_pipe_out[`M14K_M_RANGE],gscanenable,
mpc_run_m, gclk, m_pipe_in);
```

Also, the result from the adder in the E-Stage is registered to the next cycle after traversing some multiplexers (*res_m = adder_res_m*, and *edp_res_w <= res_m*) not detailed here.

e. A-Stage

The Alignment (A) Stage aligns the data read from the D$, if needed. It also prepares the register file write in the next (W) stage (similarly to the RF read, all the write inputs must be ready at the clock edge). Obviously, for an add instruction this stage does not align data but only prepares for the RF write.

The control signals that are needed in this stage are obtained from signal *m_pipe_out.* Signals needed in the next stage, the W-Stage, are registered: *w_pipe_in = m_pipe_out*, and *w_pipe_out* <= w_pipe_in (2), within **module m14k_mpc_ctl**, as shown in the next code fragment:

```
mvp_cregister_wide #(6) _w_pipe_out_5_0_(w_pipe_out[`M14K_W_RANGE],gscanenable,
mpc_run_w, gclk, w_pipe_in);
```

The inputs for writing the RF in the next (W) stage are computed as described below (also see Figure 5):

- *edp_wrdata_w*: is assigned from the adder result (*edp_wrdata_w = edp_res_w*), after traversing several multiplexers (in this lab we do not detail any of them; we discuss some of them in Lab 16).
- *mpc_dest_w*: contains the destination register number, obtained from the instruction in the E-Stage and now available in the pipeline registers (*m_pipe_out*). In Section 5 you will analyze this signal on your own.
- *mpc_rfwrite_w*: enables the write to the Register File when it is set to 1. In Section 5 you will analyze this signal on your own.

f. W-Stage

The final stage of the pipeline is the Write (W) Stage, where the register file is actually written, using the signals computed in the previous stage, as just explained.

g. Comparision of microAptiv with the processor from *DDCA* [2]

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, concerning the `add` instruction. Table 3 gives an overview of the equivalent stages in each processor's pipeline.

Table 3. MicoAptiv and *DDCA* Pipeline Comparison

MicroAptiv Pipeline Stage	*DDCA* [2] Pipeline Stage	Description from *DDCA* [2]
Pre-I (Pre-Instruction Fetch)	F (Instruction Fetch) D (Instruction Fetch)	The next PC is calculated in the Fetch stage for non-branch instructions and in the Decode stage for Branch instructions.
I (Instruction Fetch)	F (Instruction Fetch)	Fetch instruction from memory (essentially same as microAptiv)
E	D (Decode) and E (Execute)	The D stage reads the source operands from the RF and generates control signals, and the E stage performs the operation
M	M (Memory access)	Data access (essentially same as microAptiv)
A	not required	The *DDCA* MIPS processor allows only aligned accesses, so alignment is not required
W	W (Writeback)	Write result to RF (essentially same as microAptiv)

- In the processor from [2], the analogous stage to the I Stage is called Fetch (F) Stage. In the Fetch Stage, the processor retrieves the instruction from the instruction memory, a black box capable of always providing the requested instruction in one cycle (i.e. an ideal memory). Note that the instruction memory in [2] receives the instruction address at the beginning of the Fetch cycle and produces the instruction at the end of the cycle. In microAptiv, the instruction address is provided in the cycle prior to the Fetch cycle.
- In the processor from [2] the instruction register (IR) is part of the pipeline register between stages instead of the individual IR found in microAptiv.
- A major difference between the *DDCA* pipeline register and microAptiv is that the E Stage is divided into two stages, the Decode (D) and Execute (E) Stages (see Figure 7.58 in [2]). During the D Stage, the instruction is decoded (i.e., control signals are generated) and the register file (RF) is read; and during the E Stage, the operation (in this case, addition) is performed by the Arithmetic Logic Unit (ALU).

- The Control Unit of the processor from *DDCA* (see Figure 7.58 in [2]), which decodes the instruction, is obviously much simpler than the Main Pipeline Control from microAptiv. The *DDCA* pipeline only implements a small subset of instructions whereas microAptiv implements the entire Release-3 (R3) MIPS instruction set.
- The high-level behavior of the RF is analogous in both processors. The interested readers can study modules **m14k_rf_reg** and **m14k_rf_rngc**, for understanding the RF internal implementation for microAptiv, and Verilog is provided in *DDCA* [2] for its RF. Similarly to the instruction memory, the RF inputs in [2] are provided in the cycle it is read, not in the cycle before as in microAptiv.
- The multiplexers used for accessing the ALU are similar as well. In both cases, the inputs to the ALU come from the RF, the Forwarding Logic, or the Immediate.
- 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. This lab and Lab 15 analyze the structures composing the ALU available in microAptiv.
- The *DDCA* processor supports only aligned memory accesses, so microAptiv's A-Stage does not exist.

4. Example Simulation

In this section we illustrate the simulation of an `add` instruction as it flows through the different stages of the microAptiv pipeline. We first guide you in the simulation process and then we analyze the results in detail. Viewing the behavior of the different signals related to an `add` instruction helps you understanding the theoretical explanations of Section 3. **Remember to keep a backup (clean) copy of the rtl as we are going to edit some files in this and the next section.**

a. Simulation Process

This section describes how to simulate a compiled program using Vivado's XSIM. Although this is already explained in Lab 1, there are some differences here, so we opt for explaining it again from the beginning. Follow these steps, described in detail below:

Step 1. Create a Vivado project and select *testbench_boot* as top-module for simulation
Step 2. Analyze the source files provided for simulation and add the program files
Step 3. Configure and run the simulation

Step 1. Create a Vivado project and select *testbench_boot* as top-module for simulation

Follow the instructions provided in Step 1 - Lab 1 for creating a new Vivado project. Once created, make file *testbench_boot.v* the top-level module for simulation. For that purpose, in the **Project Manager** panel, scroll down to **Simulation Sources** and expand it and then expand the **sim_1** folder. Right-click on *testbench_boot.v* and set it as the top-level module for simulation. Notice that the *testbench_boot* entry is now bold.

Step 2. Analyze the source files provided for simulation and add the program files

Folder *Lab14_ADD\Simulations\SimulationSources* includes the simulation source files that we use in this section. Make a copy of this folder before starting to work with it. You can view the source program in file **main.c**. This simple program initializes registers t3 and t4, adds 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 7).

```
…
80000204 <main>:
80000204:   240b0002    li    t3,2
80000208:   240c0003    li    t4,3
8000020c:   240d0000    li    t5,0
80000210:   00000000    nop
80000214:   018b6820    add   t5,t4,t3
80000218:   1000ffff    b     80000218 <main+0x14>
8000021c:   00000000    nop
…
```

Figure 7. Example assembly program, with the `add` instruction highlighted in blue.

Figure 4 shows the machine format of an `add` instruction, which is explained extensively in Section 6.3.1 of *DDCA* [2]. Specifically, for the `add` instruction used in our program (`add t5, t4, t3` - 0x018b6820 - 0000 0001 1000 1011 0110 1000 0010 0000), 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" within the Imagination toolchain)
- **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**: 100000, indicates the function to perform, in this case addition

The two sets of text files for initializing MIPSfpga memories (**ram_p0.txt**, **ram_p1.txt**, **ram_p2.txt**, **ram_p3** and **ram_b0.txt**, **ram_b1.txt**, **ram_b2.txt**, **ram_b3**) contain the instructions that will be loaded into MIPSfpga's code and boot memories respectively. These files are provided in folder *Lab14_ADD\Simulations\SimulationSources*. In Vivado, click on **Add Sources** in the **Flow Navigator** window on the left, select the **Add or create simulation sources** option, and then click **Next**. Click on **Add Files…**, select **All Files** in the *Files of type* filter, browse to folder *Lab14_ADD\Simulations\SimulationSources*, select the eight text files, and click **OK** (Figure 8). Leave the *Copy sources into project box* unselected, but leave the *Include all design sources for simulation* box checked. Then click **Finish**. The eight text files should be added to the project hierarchy.

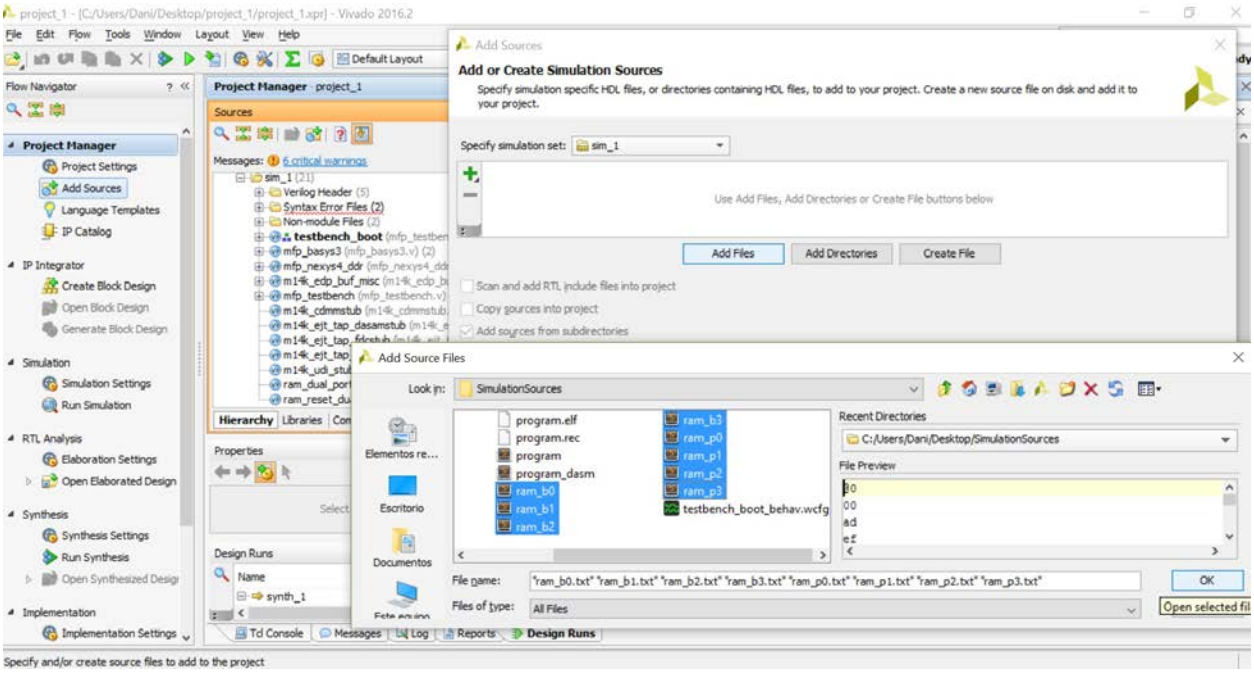


Figure 8. Add *ram_program_init.txt* and *ram_reset_init.txt* text files.

Recall that you can 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 *Lab14_ADD\Simulations\SimulationSources*), and you can also re-generate the text files (**ram_p0.txt**, **ram_p1.txt**, **ram_p2.txt**, **ram_p3** and **ram_b0.txt**, **ram_b1.txt**, **ram_b2.txt**, **ram_b3**) using the **CreateMemFiles.exe** script as explained in Section 7.3 of the Getting Started Guide.

Step 3. Configure and run the simulation

Now we are ready to perform the simulation. Before launching it, we have to configure several things.

We are providing file **testbench_boot_behav.wcfg** in folder *Lab14_ADD\Simulations*, which configures the resulting waveform according to our interests. Specifically, in this first simulation, we configure the waveform to show many of the signals from Table 2, organized in descriptive groups. For using that waveform configuration file, first make a copy of it, and then, similarly to what we did with the text files in the previous step, click on **Add Sources** in the **Flow Navigator** window on the left, select the **Add or create simulation sources** option, and then click **Next**. Click on **Add Files…**, browse to the *testbench_boot_behav.wcfg* file in the *Lab14_ADD\Simulations* folder, select it, and click **OK** (Figure 9). Leave the *Copy sources into project box* unselected, but leave the *Include all design sources for simulation* box checked. Then click **Finish**. The waveform configuration file should be added to the project hierarchy.

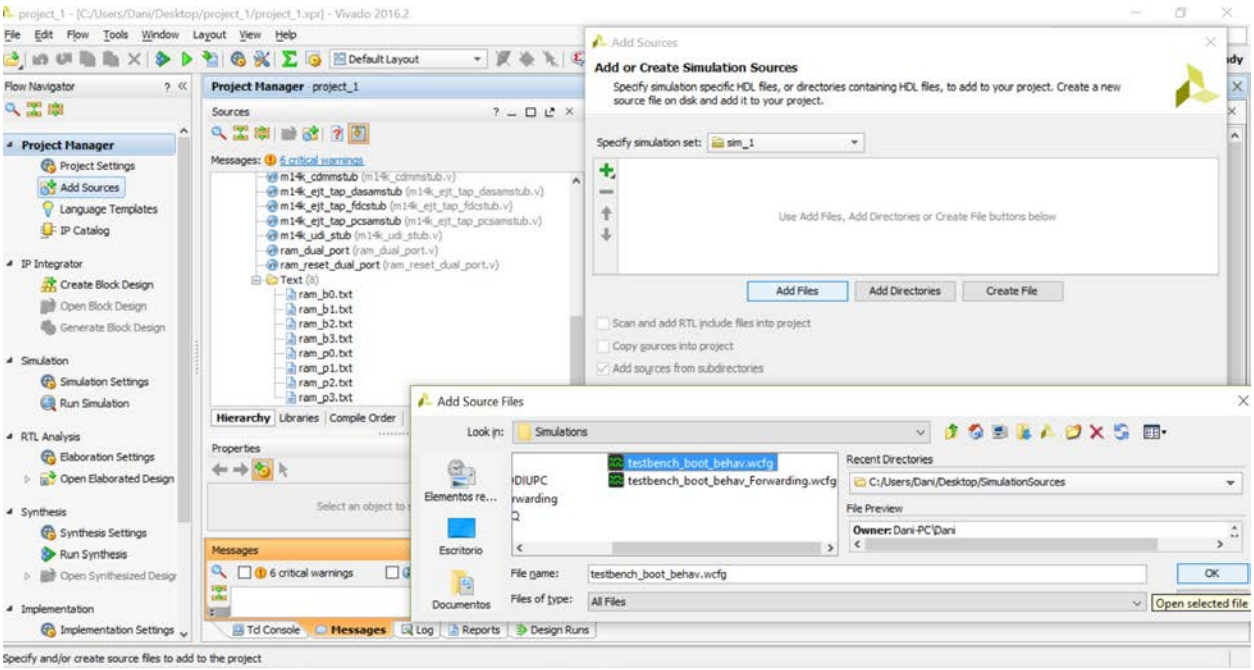


Figure 9. Add waveform configuration file.

We also want to configure the simulation runtime. The `add` instruction is fetched around time 91440ns. In the **Project Manager – Simulation Settings – Simulation**, you can configure the simulation to stop at time 92000ns by setting: *xsim.simulate.runtime* = 92000ns (Figure 10).

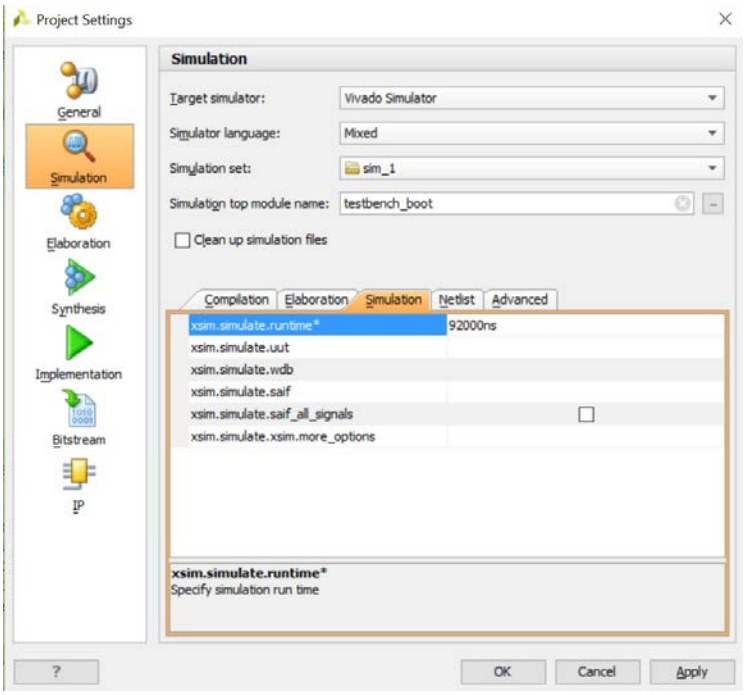


Figure 10. Configuration of simulation runtime.

In order for the simulation to run until this time without stopping, we recommend you to comment lines 78 and 83 of file *rtl_up\testbench\testbench_boot.v*, as shown in Figure 11.

```
75      always @(negedge SI_ClkIn)
76        if (HADDR == 32'h1fc001f4) begin
77         $display("Data cache initialized. About to make kseg0 cacheable.");
78  //       $stop;
79      end
80      else if (HADDR > 32'h00000758 & HADDR < 32'h1fc00000 & ~program_started) begin
81       program_started = 1;
82       $display("Beginning of program code.");
83  //     $stop;
84      end
85  endmodule
```

Figure 11. Remove $stop command from file testbench_boot.v.

Finally, add "-cc gcc" as an elaboration option in *xsim.elaborate.xelab.more_option*s. Figure 12 illustrates this step.

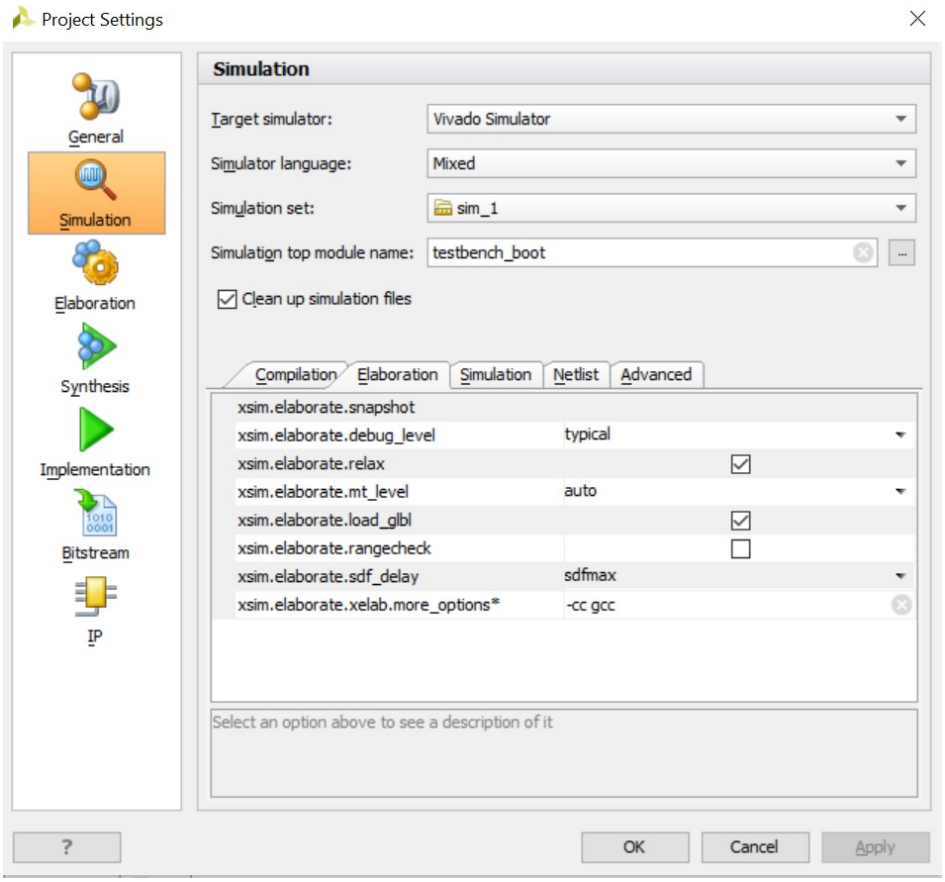


Figure 12. More elaboration options: "-cc gcc".

At this point, the simulation is configured. Click on **Run Simulation → Run behavioral simulation** in the Flow Navigator window. The *testbench_boot* and lower-level modules will compile, the simulation window will open, and the simulation results will be displayed. You can float the waveform window by clicking on the float button ( ) and then maximize it by clicking on the full size button (left one). You can also use Zoom In ( ), Zoom Out ( ), and Zoom to Cursor ( ) buttons to view a desired section of the waveform. In our case, you have to zoom out and to expand the groups (such as *RF-Read*, *Adder* or *RF-Write*). Then, given that we have configured the simulation to stop right before the `add` instruction fetch, change the run time to 10ns (one cycle) 

and continue simulation for some cycles by clicking on button several times.

b. Analysis of the simulation of the `add` instruction

Figure 13 illustrates a timing diagram of the different stages of the execution of the `add` instruction.

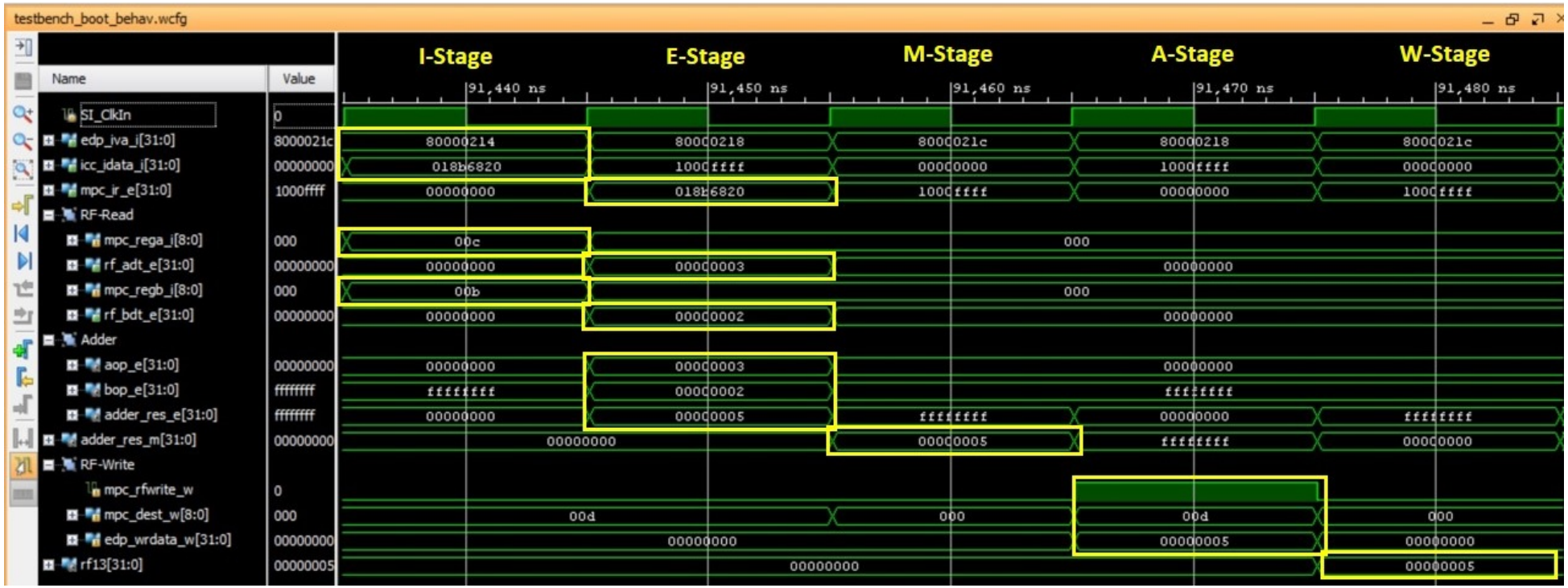


Figure 13. Timing diagram of the execution of an *add* instruction.

Next we detail the results obtained in the execution of the `add` instruction, shown in Figure 7.

- 1st cycle, I-Stage:
  - The Current PC is: *edp_iva_i[31:0]*=0x80000214.
  - The Instruction read from the I$ is: *icc_idata_i[31:0]*=0x018b6820.
  - The register identifiers to read from the RF in the next cycle are provided to the RF: *mpc_rega_i[4:0]*=01100 and *mpc_regb_i[4:0]*=01011.
  - The instruction read from the I$ is registered for the next cycle (*mpc_ir_e* <= *icc_idata_i*).
- 2nd cycle, E-Stage:
  - The instruction is decoded, generating control signals for the following cycles (*ie_pipe_out*, *m_pipe_out* and *w_pipe_out*). These signals are passed to the next stages via the Pipeline Registers (*_ie_pipe_out_50_0_*, *_m_pipe_out_46_0_* and *_w_pipe_out_5_0_*).
  - The Register File is read based on the signals provided in the previous cycle:
    - *rf_adt_e*=0x00000003.
    - *rf_bdt_e*=0x00000002.
  - The addition is performed by the ALU module, i.e. **m14k_edp_add_simple**:
    - *adder_res_e*=0x00000005.
- 3rd cycle, M-Stage.
  - No memory accesss occurs, so only the ALU result is passed to the next stage: *adder_res_m*=0x00000005.
- 4th cycle, A-Stage.
  - The inputs for writing the RF in the next stage are generated:
    - *mpc_rfwrite_w*=1. Write enabled.
    - *mpc_dest_w*=01101.
    - *edp_wrdata_w*=0x00000005.
- 5th cycle, W-Stage.
  - The RF is written. The destination register, t5, which corresponds to register 13, is written: *rf13*=0x00000005.

5. Exercises

Exercise 1. Analyze related instructions

Simulate and analyze a subtract (`sub`) instruction, an add immediate unsigned (`addiu`) instruction, and a set-if-less-than (`slt`) instruction, and describe the main differences between those instructions and an `add` instruction. You can first analyze them theoretically (Table 4

provides the main modules and signals related to each instruction) and then based on simulation (you can use the Vivado project used in Section 4).

Table 4. Exercise 1: main modules and signals

`sub` instruction	
Module/Signal Name	**Description**
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
m14k_edp	**Execution datapath**
mpc_subtract_e	Control signal computed at module **m14k_mpc_dec**, used for several instructions, such as `sub`
`addiu` instruction	
Module/Signal Name	**Description**
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
mpc_addui_e	Control signals related to the `addiu` instruction
mpc_imsgn_e	
mpc_selimm_e	
m14k_edp	**Execution datapath**
sgnd_imm_e[31:0]	Sign extended immediate value and upper immediate value
icc_pcrel_e	Always set to 0 in this configuration of the core
mpc_ir_e[31:0]	Instruction register assigned at module **m14k_mpc_dec**
`slt` instruction	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
mpc_udisel_m	Control signals related to the `slt` instruction
mpc_udislt_sel_m	
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
m14k_edp	**Execution datapath**
bit0_m	Result of *set less than* operation. Depends on signals *mpc_signed_m* (computed at **m14k_mpc_ctl**), and *pro31_m* and *car31_m*
m14k_edp_buf_misc_pro	**Execution datapath submodule**
udislt_m[31:0]	Result of *set less than* operation
res_m[31:0]	

You can generate your own source files for simulating these other instructions by following the next steps: (1) Make a copy of folder *Lab14_ADD**Simulations**SimulationSources*, (2) in the new folder, edit file **main.c**, and substitute the `add` instruction with the new instruction that you wish to simulate, (3) follow the instructions provided in Sections 7.2 and 7.3 of the Getting Started Guide for compiling the source files and creating the text files that initialize the MIPSfpga memories.

Once the eight new text files have been created, you are ready to perform the simulation. The easiest option in this case is to reuse the Vivado project created in Section 4.a for simulating the add instruction. For that purpose, you must reconfigure that project as follows.

Step 1. Substitute the old text files for the new ones
Step 2. Run the simulation
Step 3. Add new signals to the timing diagram, such as *mpc_subtract_e*
Step 4. Re-run the simulation

Step 1. Substitute the old text files for the new ones

In order to add the new text files for initializing MIPSfpga memories we must first remove the old ones. For that purpose, select the eight files on the **Project Manager**, right-click on them, and select the option **Remove File from Project …** (Figure 14). Then, follow the procedure explained above (Figure 8) for adding the new text files.

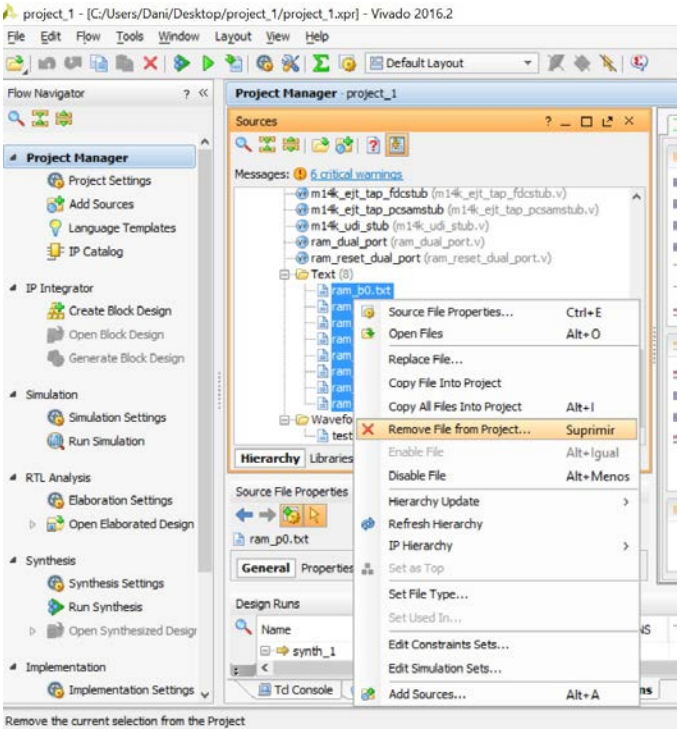


Figure 14. Remove the eight text files.

Note that, if you had to replace the waveform configuration file (*testbench_boot_behav.wcfg*) for a new one (in this case this is not necessary, as we are analyzing the same signals as before), you would follow the same procedure explained in Figure 14 for removing the old file, and then you would follow the procedure explained in Figure 9 for adding the new waveform configuration file.

Step 2. Run the simulation

Click on **Run Simulation → Run behavioral simulation** in the Flow Navigator window. The *testbench_boot* and lower-level modules will compile, the simulation window will open, and the simulation results will be displayed.

Step 3. Add signal *mpc_subtract_e* to the timing diagram

In the **Scopes** window, expand the *testbench_boot* hierarchy to *testbench_boot → sys → top → cpu → core → mpc*, and left-click on *mpc_dec* (Figure 15).

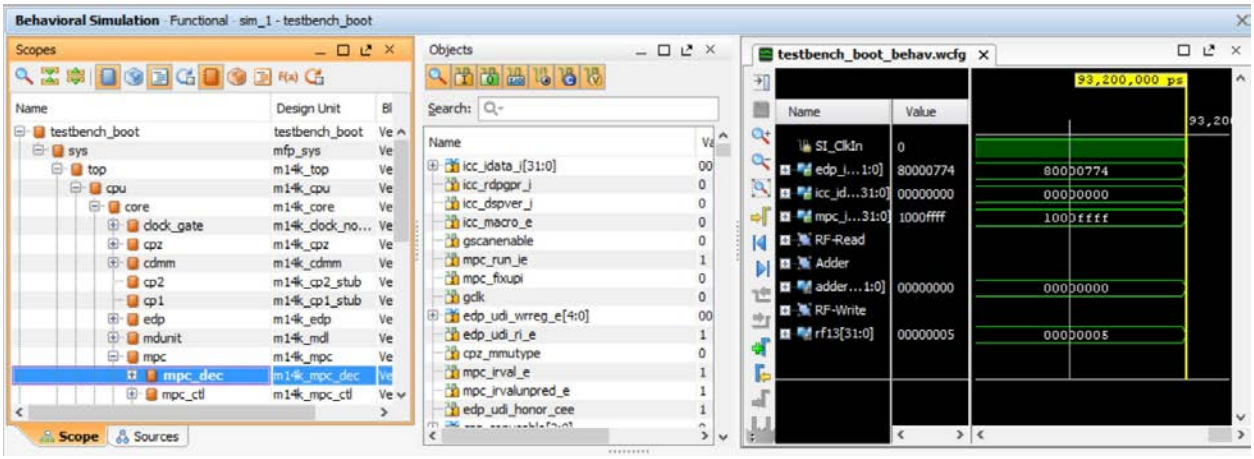


Figure 15. Scopes and Objects windows.

In the **Objects** window (Figure 15), all the signals belonging to the interface of module **m14k_mpc_dec** are shown. Look for signal *mpc_subtract_e*, right-click on it, and select *Add to Wave Window* (Figure 16). The new signal will be added to the timing diagram.

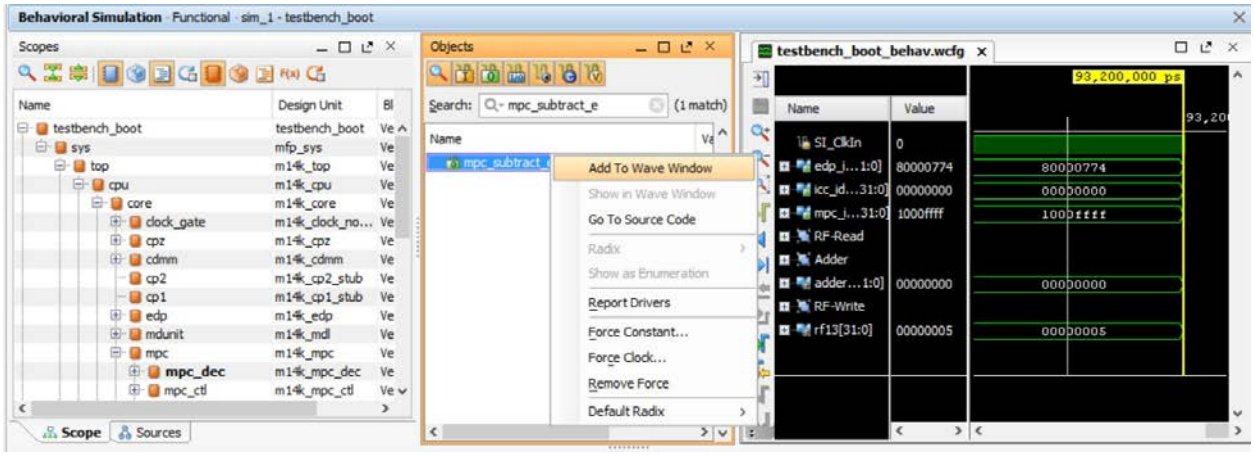


Figure 16. Add new signal to the timing diagram.

Step 4. Re-run the simulation

Re-execute the simulation as explained in Step 2. The new signal (*mpc_subtract_e*) should now be included in the results.

Exercise 2. Analyze control signals

Sketch the hardware that feeds the following control signals, used at the A-Stage for controlling the RF write. Table 5 provides the main modules and signals related to each control signal. Focus on the main signals and do not go into many details, as it could get too complex.

- *mpc_dest_w[8:0]*. This signal determines the destination register, the register within the register file that is written.
- *mpc_rfwrite_w*. This signal determines if the register file is written or not.

Table 5. Exercise 2: main modules and signals

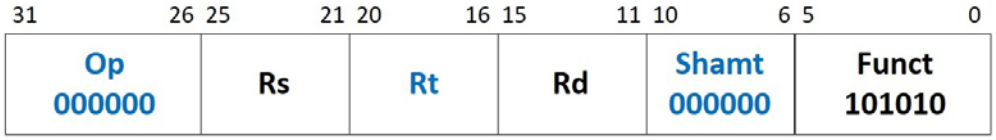
***mpc_dest_w* control signal**	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
dest_m	Destination register at the M-Stage
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
dest_e	Destination register at the E-Stage
***mpc_rfwrite_w* control signal**	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
mpc_exc_w	Exception at the W-Stage
mpc_run_w	It notifies if the W-Stage is executing or not
pvd_m	Instruction at the M-Stage has valid destination
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
vd_e	Instruction at the E-Stage has valid destination

Exercise 3. Adding new instructions to the soft-core: Set if equal (seq)

In this exercise you will expand the MIPSfpga (microAptiv) core to perform the `seq` instruction (proposed in Exercise 4.2. of [4]). This instruction sets the destination register to 1 when the two source operands are equal and resets it to 0 otherwise. It is similar to a `slt` (set if less than) instruction, already included in microAptiv's instruction set and analyzed above in Exercise 1.

The format and description of a `slt` instruction are the following:

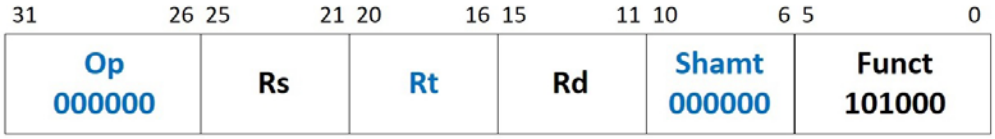
- Format:



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

The format and description of a `seq` instruction are the following:

- Format:



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

Note that we are using the same format for `seq` as for `slt`, except with a Funct field of 101000. Because the assembler does not support the new mnemonic, you must write the `seq` instruction in machine code in the assembly program. The following lines show how to include two `seq` instructions with different result. You can find this program in the **main.c** file of folder *Lab14_ADD\Simulations\SimulationSources_SEQ*, where everything is ready (the *.elf* file, the text files for initializing memory, etc.).

```
"   li $t3, 4;"
"   li $t4, 7;"
"   li $t5, 0;"
"   li $t6, 0;"
"   nop;"
"   .word 0x016c6828;"  // seq $t5, $t3, $t4    ,,    $t5=0
"   .word 0x016b7028;"  // seq $t6, $t3, $t3    ,,    $t6=1
"   b .;"               // Stay here
```

Table 6. Exercise 3: main signals related to `seq` instruction

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
seq_instr / *seq_instr_m*	Two new signals. The first (*seq_instr*) is 1 when a `seq` instruction is decoded. This signal is used both at the E and M-Stages, so it must be registered into a second new signal (*seq_instr_m*)
spec_ri_e	Trigger *reserved instruction* trap for *special* instruction
slt_sel_e	This signal is registered into *slt_sel_m* and used to select the result from `slt/seq` instruction at the M-Stage
m14k_edp	**Execution datapath**
acmp_e	First source of the following comparator: `edp_cndeq_e = acmp_e == edp_bbus_e;`
bit0_m	Result of `slt/seq` instruction
edp_cndeq_m	New signal belonging to the M-Stage, which obtains the value from signal *edp_cndeq_e* (E-Stage). Thus, a new register is required.

Table 6 includes the signals that we must include or modify for including a `seq` instruction. Moreover, below are some hints to help you implement this instruction:

- **Create two new signals (*seq_instr* and *seq_instr_m*)**: Define a new signal in module **m14k_mpc_dec** that is 1 when a `seq` instruction is detected at decoding, and 0 otherwise. Provide this signal to all the modules that use it and register it into signal *seq_instr_m* for using it at the M-Stage.
- **Disable exceptions:** For encoding the `seq` 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* = *101000*). 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 `seq` instruction.
- **Compute the result of the `seq` instruction (rs==rt) at the E-Stage:** For computing the result of the `seq` instruction, we have several options:
    - The obvious solution is to include a new hardware for computing the equality condition (==) among the two operands, but it is the most expensive one from the HW point of view.
    - The `slt` instruction is computed by subtracting *rs-rt* in the "Carry Propagate Adder" included in module **m14k_edp**, called **m14k_edp_add_simple**. We can use this same operation for computing the equality condition, by *ORing* the 32 bits resulting from the subtraction.
    - The cheapest option, which we recommend you to use, would probably be to reuse a comparator already available in the pipeline. As such, at module *m14k_edp*, the following logic is already available at the E-Stage:

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

      Note that for a `seq` instruction, *acmp_e* must be assigned from signal *edp_abus_e*, thus multiplexer *_acmp_e_31_0_* must be modified. As for signal *edp_bbus_e*, it can be used as it is.
      Also, note that we need the result of the comparison at the M-Stage, so we have to register *edp_cndeq_e* into a new signal *edp_cndeq_m*.
- **Select the result of the `seq` instruction to write to the register file (*edp_wrdata_w*):** One possible implementation is to use signal *bit0_m* (as examined in Exercise 1, this signal provides the result of a `slt` instruction to the register file) for the result of the `seq` instruction. For that purpose:

    - Using *seq_instr_m* as a control signal, include the result of the seq instruction (*edp_cndeq_e*) in the computation of *bit0_m* at the **m14k_edp** module.
    - Modify signal *slt_sel_e* for selecting *bit0_m* at the M-Stage as the instruction result. This is done at module **m14k_edp_buf_misc_pro** using signal *mpc_udislt_sel_m*, which depends on *slt_sel_e*.
- **Set register file write strobe (*mpc_rfwrite_w*) and compute destination register (*mpc_dest_w*) for seq instruction**: As examined in Exercise 2, *mpc_rfwrite_w* depends on *vd_e*, whereas *mpc_dest_w* depends on *dest_e*. Both signals (*vd_e* and *dest_e*) are computed at module **m14k_mpc_dec**, and they need no changes for including the seq instruction.

To complete this task, do the following:

1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_seq**).
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 the capability of the microAptiv core to implement seq, modifying the Verilog files conforming the soft-core, following the instructions provided above. You will have to change the two Verilog files included in Table 6 (**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. You can directly edit and modify the files (using any text editor, such as *Sublime Text*) contained in the **rtl_up_seq** folder (note that the project source files will automatically update), or you can edit the files using the text editor included in Vivado.
4. Create a new Vivado project (**project_seq**) following the instructions provided in Step 1 - Lab 1, using the files from the new folder (**rtl_up_seq**).
5. Using the program shown above, provided in folder *Lab14_ADD**Simulations**SimulationSources_SEQ*, debug your implementation with Vivado XSIM simulator. Follow the steps explained in Section 4 for configuring the simulator. Specifically, the fetch of the first seq instruction is done around time 91450ns, thus configure the simulation runtime as explained in Figure 10. As for the waveform configuration file, you can use the *testbench_boot_behav.wcfg* used for the add instruction as a starting point (Figure 9), and then add the necessary signals depending on your implementation as explained in Exercise 1 (Figure 15 and Figure 16).
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 *Lab14_ADD**Simulations**SimulationSources_SEQ*, by commenting line "b . ;" (as shown in Figure 17). 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 *Lab14_ADD\Simulations\SimulationSources_SEQ*).

```
14  int main() {
15
16      int x,y;
17
18      asm volatile
19      (
20          "   li $t3, 4;"
21          "   li $t4, 7;"
22          "   li $t5, 0;"
23          "   li $t6, 0;"
24          "   nop;"
25          "   .word 0x016c6828;"  // seq $t5, $t3, $t4
26          "   .word 0x016b7028;"  // seq $t6, $t3, $t3
27  //          "   b .;"             // Stay here
28      );
```

Figure 17. Comment infinite loop.

Then, analyze on your own the code after the commented branch. This code will output, on the 7-segment displays, the value of register $t5 (when the switches on the board are all low), which contains the result of the first `seq` instruction, and the value of register $t6 (when the right-most switch is high), which contains the result of the second `seq` instruction.

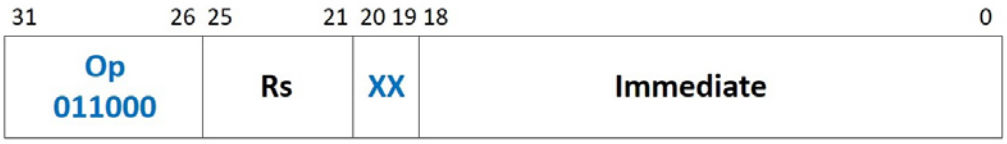
- Step 2 – Synthesize the new processor, as explained in Step 3 – Lab 1: Click on the **Generate Bitstream** button  at the top of the window. Now wait for synthesis, placement, routing, and bitstream generation to complete. This typically takes around 10-20 minutes or more, depending on your computer speed.
- Step 3 – Program the FPGA board, as explained in Step 4 – Lab 1: Click on **Open Hardware Manager** in the **Flow Navigator** window on the left. Make sure that the Nexys4 DDR FPGA board is turned on and connected to your computer, and click on **Open Target → Auto Connect**. Finally, click on **Program Device → xc7a100t_0**, select the bitfile if it is not selected yet, and click on **Program**.
- Step 4 – Download the program to the board using the script **loadMIPSfpga.bat**, 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 or $t6 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 *0x80000218` (set breakpoint before first `seq` 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)
- `stepi` (execute 1 instruction)
- `i r` (list the register file contents)

Exercise 4. Adding new instructions to the soft-core: Add immediate unsigned to PC (addiupc)

In ISA Release-6, MIPS included several new instructions that use the program counter (PC) as a source register. In this exercise you will expand MIPSfpga to implement one such instruction: `addiupc`. The `addiupc` instruction adds an immediate to the PC and places the result in a register. You must use an *opcode* not used by any existing MIPS R3 instructions, such as 011000 (this *opcode* is chosen for the sake of simplicity; MIPS R6 architecture uses a different *opcode* for this instruction). The machine code format and functionality of `addiupc` is shown below.

- Format:



- Description: *Rs ← PC + SignExtend(Immediate<<2)*

Similarly to the `seq` instruction in Exercise 3, the assembler we are using does not support the new mnemonic. Thus, we include an `addiupc` instruction in our assembly code by using its machine encoding. The following lines show how to include an `addiupc` instruction. You can find this program in the **main.c** file of folder *Lab14_ADD\Simulations\SimulationSources_ADDIUPC*, where everything is ready (the *.elf* file, the text files for initializing memory, etc.).

```
"   li $t4, 0;"
"   .word 0x61800004;" // addiupc $t4, 0x00004
"   b .;"              // Stay here
```

Table 7. Exercise 4: main signals related to `addiupc` instruction.

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
addiupc_instr	New signal, which is 1 when an `addiupc` instruction is decoded
maj_ri_e	Trigger *reserved instruction* trap for illegal instruction decode from major opcode
alu_sel_e	Select ALU for output at the M-Stage
maj_vd_e	Valid destination
dest_e	Destination register

m14k_edp	Execution datapath
aop_e	Adder SrcA. Include the PC (*iva_e*)
bop_e	Adder SrcB. Include the *SignExt(Imm<<2)*

Table 7 includes the signals that we must modify for including an `addiupc` instruction. Moreover, below are some hints to help you implement this instruction:

- **Create a new signal *addiupc_instr***: Define a new signal in module **m14k_mpc_dec** that is 1 when an `addiupc` instruction is detected at decoding, and 0 otherwise. Provide this signal to all the modules that use it.
- **Disable exceptions:** For encoding the `addiupc` instruction, we use an *op* (operation code) 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 = *011000*). 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 *maj_ri_e* handles illegal instruction decode from major opcode. Change this signal for inhibiting the *Reserved Instruction* exception for an `addiupc` instruction.
- **Compute the result of the `addiupc` instruction at the E-Stage:** For computing the result of the `addiupc` instruction, we have several options:
  - The obvious solution is to include a new hardware for computing the addition.
  - A cheaper option, which we recommend you to use, would probably be to reuse the adder already available in the processor (module **m14k_edp_add_simple**). For that purpose, we must incorporate the new inputs:
    - The PC is already available at the E-Stage in signal *iva_e*. Pass this signal to the adder through *aop_e*.
    - Incorporate the *SignExt(Imm<<2)* as an input to the adder through signal *bop_e*.
- **Select the result of the `addiupc` instruction to write to the register file (*edp_wrdata_w*):** Signal *alu_sel_e* is used in the `add` instruction (and other arithmetic-logic instructions) for selecting, at the M-Stage (*mpc_alusel_m*), the result of the instruction. Modify that signal to select the result of the `addiupc` instruction.
- **Set register file write strobe (*mpc_rfwrite_w*) for `addiupc` instruction**: As examined in Exercise 2, *mpc_rfwrite_w* depends on *vd_e*, which depends on several signals (such as *maj_vd_e*). In this case, differently to the `seq` instruction from Exercise 4, we must change computation of *maj_vd_e*.
- **Compute destination register (*mpc_dest_w*) for `addiupc` instruction**: As examined in Exercise 2, *mpc_dest_w* depends on *dest_e*, which we must change for incorporating the destination register of `addiupc`.

To complete this task, perform the following:

1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_addiupc**).
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 `addiupc`, modifying the Verilog files conforming the soft-core, following the instructions provided above. You will have to change the two Verilog files included in Table 6 (**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. You can directly edit and modify the files (using any text editor, such as *Sublime Text*) contained in the **rtl_up_addiupc** folder (note that the project source files will automatically update), or you can edit the files using the text editor included in Vivado.
4. Create a new Vivado project (**project_addiupc**) following the instructions provided in Step 1 - Lab 1, using the files from the new folder (**rtl_up_addiupc**).
5. Using the program shown above, provided in folder *Lab14_ADD**Simulations**SimulationSources_ADDIUPC*, debug your implementation with Vivado XSIM simulator. Follow the steps explained in Section 4 for configuring the simulator. Specifically, the fetch of the `addiupc` instruction is done around time 91380ns, thus configure the simulation runtime as explained in Figure 10. As for the waveform configuration file, you can use the *testbench_boot_behav.wcfg* used for the `add` instruction as a starting point (Figure 9), and then add the necessary signals depending on your implementation as explained in Exercise 1 (Figure 15 and Figure 16).
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 *Lab14_ADD**Simulations**SimulationSources_ADDIUPC*, by commenting line "`b .;`" (as explained in Figure 17). 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 *Lab14_ADD**Simulations**SimulationSources_ADDIUPC*).
      Then, analyze on your own the code after the commented branch. This code will output, on the 7-segment displays, the value of register $t6, which contains the result of the `nand` instruction. You can look for the `nand` instruction in file **program.dis**, in order to know its PC.
    - 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 $t6 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 *0x8000020c` (set breakpoint before `addiupc` 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)

6. References

[1] "MIPS32® microAptiv™ UP Processor Core Family Software User's Manual -- MD00942".

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

[3] "MIPS32® microAptiv™ UP Processor Core Family Implementor's Guide -- MD00940".

[4] "Computer Organization and Design", 5th Edition. David A. Patterson and John L. Hennesy. Morgan Kaufmann, 2013.