MIPSfpga by Imagination

Lab 16

Basic Instruction Flow – LW Instruction

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

Lab 16
Basic Instruction Flow – LW Instruction

1. Introduction

In this lab we analyze the flow of the load word (`lw`) instruction along the microAptiv pipeline. This I-type instruction (`lw rt, rs(immediate)`) reads a data word from memory into a register (*rt*). It specifies the effective address in memory as the sum of a base address (a register, *rs*) and an offset (a constant provided with the instruction, *immediate*). Section 6.2 of *DDCA* [1] explains the `lw` instruction in detail whereas Figure 1 illustrates its format. The address is computed by adding the contents of the base address (rs) to the sign-extended immediate. The instruction functionality can be expressed as follows:

Reg[rt] = Mem[Reg[rs] + SignExtension(immediate)]

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

Figure 1. `lw` machine instruction format

2. Pipeline Stages

In this section we explain the execution of the `lw` instruction through each stage of the pipeline. The Pre-I-Stage and the I-Stage are not included, as they are completely analogous to the `add` instruction. 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 lw instruction

E-Stage	
Module/Signal Name	**Description**
m14k_rf_reg	**Register file**
rf_adt_e[31:0]	Data read from the register file (base address)
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder

mpc_ir_e[15:0]	Immediate value provided from the instruction register
mpc_aselres_e	Value from forwarding logic (*mpc_aselres_e*=1) or from the register file (*mpc_aselres_e*=0)
m14k_edp	**Execution datapath**
preabus_e[31:0]	Value forwarded from the M-Stage or from the A-Stage
dva_offset_e[31:0]	Sign-extended immediate value
new_dva_e_int[31:0]	Effective address used to read the Data Memory
M-Stage	
Module/Signal Name	**Description**
m14k_dcc	**Data Cache Controller**
dcc_ddata_m[31:0]	Read data from the data cache (hit) or from main memory (miss)
m14k_edp	**Execution datapath**
ld_or_cp_m[31:0]	Read data from the data cache (hit) or from main memory (miss)
A-Stage	
Module/Signal Name	**Description**
m14k_edp	**Execution datapath**
edp_ldcpdata_w[31:0]	Read data from the data cache (hit) or from main memory (miss)
ld_algn_w[31:0]	Aligned data, passed to the register file (*edp_wrdata_w[31:0]*) to write at the W-Stage
m14k_mpc_ctl	**Main pipeline control** – Control
mpc_dcba_w	Value from the Data Memory (*mpc_dcba_w*=1) or from the ALU (*mpc_dcba_w*=0)

a. E-Stage

The three main functions of the E-Stage for a `lw` instruction are to: (1) fetch a register from the register file (RF), (2) add the register and a constant (immediate) provided within the instruction, 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 new structures needed at the E-Stage for executing a `lw` instruction (available within the **m14k_edp** module). As can be observed, a new adder is introduced for computing the effective address (*new_dva_e_int[31:0]*). Similarly to the `add` instruction, the SrcA of the new adder (signal *aop_e[31:0]*) is provided from the register file (*rf_adt_e[31:0]*) or from the forwarding logic (*preabus_e[31:0]*). The other source of the new adder (*dva_offset_e[31:0]*) is computed by sign-extending the constant or immediate value provided with the instruction (*mpc_ir_e[15:0]*). The effective address is used in the next stage for reading the Data Memory.

The following code segment, extracted from line 1125 of **m14k_edp**, defines all this logic:

```
assign dva_offset_e[31:0] = !mpc_selimm_e ? bbusis_e
       : (mpc_atomic_e || mpc_atomic_m) ? {{20{mpc_imsgn_e}}, mpc_ir_e[11:0]}
       : {{16{mpc_imsgn_e}}, mpc_ir_e[15:0]};
```

```
assign new_dva_e_int[31:0] = aop_e[31:0] + dva_offset_e;
```

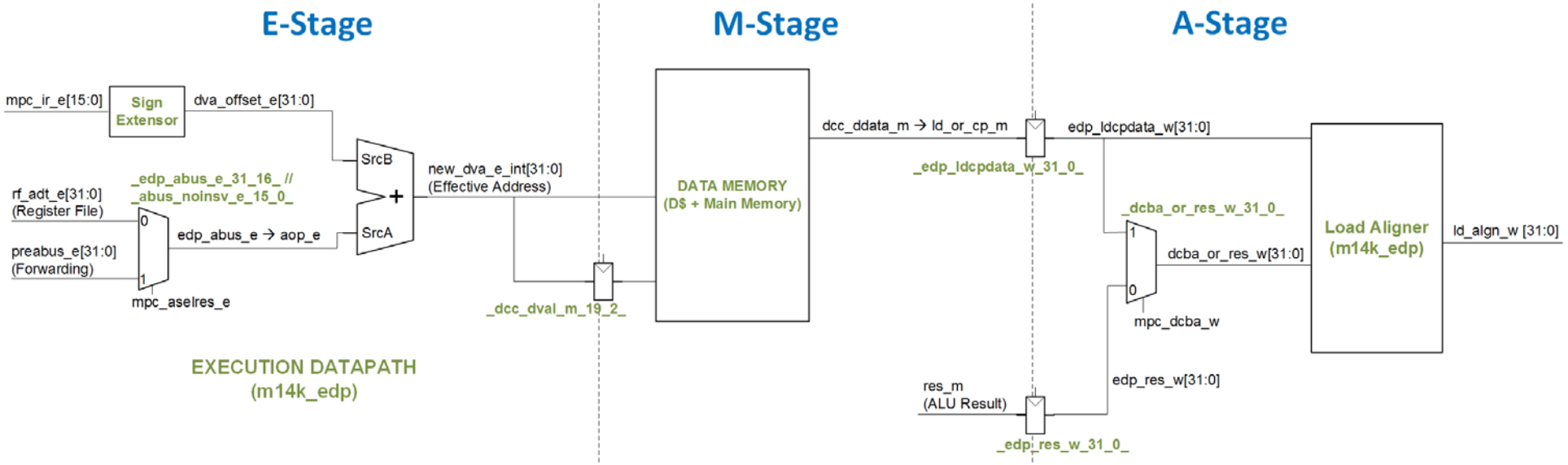


Figure 2. Main structures and signals involved in the E-Stage of a `lw` instruction. For the sake of simplicity, some elements, such as the register file, are not shown here (you can see them in Figure 3 of Lab 14).

b. M-Stage

The M-Stage fetches the data from data memory based on the effective address computed in the previous cycle (*new_dva_e_int[31:0]*). Similar to the Instruction Memory, the Data Memory of microAptiv is made up by a data cache (implemented in module **m14k_dc**) and a main memory, accessed in the case of a data cache (D$) miss. The fetched data is found in signal *dcc_ddata_m[31:0]* within module **m14k_dcc**, and this signal is passed to the **m14k_edp** module, where it is assigned as: *ld_or_cp_m[31:0] = dcc_ddata_m[31:0]*. Similar to the I-Stage, the data is available with a latency of 1 cycle after the effective address (*new_dva_e_int[31:0]*) changes upon a D$ hit, or *n* cycles upon a D$ miss.

c. A-Stage

In this stage, the data read from the D$ (*edp_ldcpdata_w[31:0]*) is aligned within the **m14k_edp** module (Figure 2), and then it is passed to the register file (*edp_wrdata_w*[31:0] = *ld_algn_w [31:0]*). We won´t go into details about the alignment process as it is not required for completing this lab. The interested readers can analyze on their own the functionality included between lines 1035 and 1071 of the **m14k_edp** module.

Note that Figure 2 includes a new multiplexer that selects between the ALU result (*edp_res_w*) or the data read by the *lw* (*edp_ldcpdata_w*), using *mpc_dcba_w* as the control signal. When an arithmetic/logic instruction is executed, *mpc_dcba_w*=0 so the ALU result is selected, and the Load Aligner is skipped. If a `lw` instruction is executed, *mpc_dcba_w*=1, and the value read from the Data Memory is selected and aligned.

d. 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 explained for previous instructions (see Labs 14 and 15).

e. 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 in regards to the `lw` instruction.

- MicroAptiv includes two separate adders, one for computing the operation of arithmetic instructions (such as `add` or `sub`) and another for computing the effective address (EA) of `load/store` instructions. Instead, the processor from [1] carries out all previous operations in the same adder (the one included in the ALU from Figure 7.58).
- The source operands for computing the EA are equal in both designs. Note that a sign extension block is necessary in both processors. Obviously, the one included in microAptiv (shown in the code above) is more complex, as this processor supports a

more extensive ISA, and some instructions require operand extensions other than the one performed for the `lw` instruction. Note however that, for the `lw` instruction, an identical extension is carried out:

- Figure 2:

```
dva_offset_e[31:0] = … {{16{mpc_imsgn_e}}, mpc_ir_e[15:0]}
```

- Figure 7.58 of [1]:

$\texttt{SignImmD}_{31:16} = \texttt{InstrD}_{15}$ and $\texttt{SignImmD}_{15:0} = \texttt{InstrD}_{15:0}$

- In the processor from *DDCA* [1], the analogous stage to the M-Stage is called the Memory Stage. In the Memory Stage, the processor retrieves the data from the data memory, a black box capable of always providing the requested data in one cycle (i.e. an ideal memory).
- The multiplexer shown in Figure 2 for the A-Stage is also included in the processor from [1] (in this case at the Writeback Stage, as the A-Stage does not exist in that design), and both the two inputs and the control signal of the multiplexer are analogous in both systems.

3. Example Simulation

In this section we illustrate the simulation of the `lw` instruction as it flows through the stages of the microAptiv pipeline. We first guide you on the simulation process and then analyze the results in detail. Viewing the behavior of the signals related to a `lw` 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 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 *Lab16_LW\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 register t6 to the address of an array called *test_data[10]*, and then it reads the third element of that array, *test_data[2]*, twice (note that we will examine the second `lw`, which hits in the D$). 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:   3c0e8000    lui   t6,0x8000
80000208:   25ce0250    addiu t6,t6,592
8000020c:   240d0000    li    t5,0
80000210:   00000000    nop
80000214:   8dcd0008    lw    t5,8(t6)
80000218:   00000000    nop
8000021c:   240d0000    li    t5,0
80000220:   8dcd0008    lw    t5,8(t6)
80000224:   1000ffff    b     80000224 <main+0x20>
80000228:   00000000    nop
…
```

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

In file **main.c** you can also view how we define and initialize array *test_data[10]* before accessing it:

```
int __attribute__((aligned(16))) test_data[10]={0xf,0xe,0xd,0xc,0xb,0xa,0x9,0x8,0x7,0x6};
```

Note that, by using the `__attribute__((aligned(16)))`, we are aligning the array to a 16B (one word) boundary (you can easily check this if you look for "*test_data*" and "*.data*" in file **program.dis**).

Figure 1 shows the machine format of an `lw` instruction, which is explained extensively in Section 6.3.2 of *DDCA* [1]. Specifically, for the `lw` instruction used in our program (`lw t5,8(t6)`- 0x8dcd0008 - 1000 1101 1100 1101 0000 0000 0000 1000), each field specifies the following:

- **Opcode**: 100011, indicates an `lw` instruction
- **Base**: 01110, the base register, t6, which is register 14 (as defined by the MIPS ISA and in file "regdef.h")
- **Rt**: 01101, the destination register, t5, which is register 13 (again, see the MIPS ISA or "regdef.h")
- **Offset**: 0x0008, a constant to add to the base register

Now, configure the simulation: The second `lw` instruction is fetched around time 93750ns, thus configure the simulation to run for that time, as explained in Figure 8 of Lab 14. Moreover, use the waveform configuration provided for this exercise (*Lab16_LW**Simulations**testbench_boot_behav.wcfg*) as explained in Figure 7 of Lab 14, and use the text files for initializing the memories (*Lab16_LW**Simulations**SimulationSources*), as explained in Figure 6 of Lab 14.

b. Analysis of the simulation of the *lw* instruction

Figure 4 illustrates a timing diagram of the stages of the execution of the `lw` instruction.

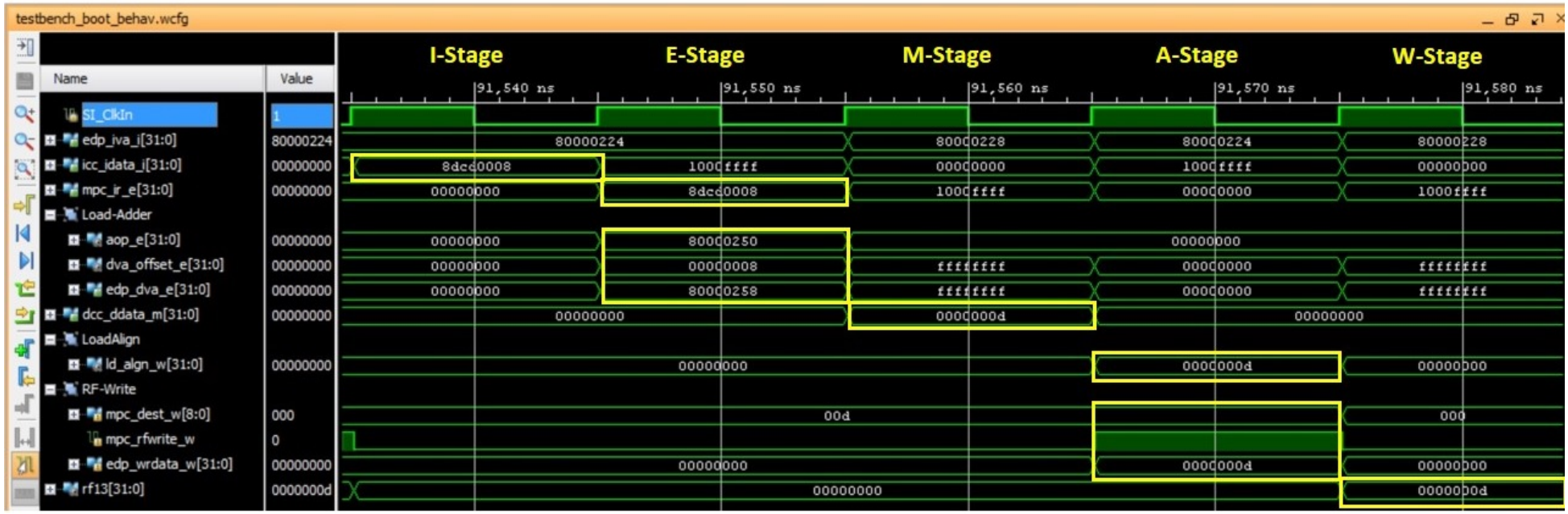


Figure 4. Timing diagram of the execution of a `lw` instruction.

Next we describe the timing diagram of the `lw` instruction in Figure 3.

- 1st cycle, I-Stage:
  - The Instruction read from the I$ is: *icc_idata_i[31:0]*=0x8dcd0008.
- 2nd cycle, E-Stage.
  - The instruction is decoded and the register file is read.
  - The addition is performed at the new adder for computing the effective address:
    - Inputs:
      1. *aop_e*=0x80000250.
      2. *dva_offset_e*=0x00000008.
    - Output:
      1. *edp_dva_e*=0x80000258.
- 3rd cycle, M-Stage:
  - The Data read from the D$ (*dcc_ddata_m[31:0]*=0x0000000d) is registered to the next stage (*edp_ldcpdata_w[31:0] <= ld_or_cp_m[31:0] = dcc_ddata_m[31:0]*).
- 4th cycle, A-Stage:
  - The data read from the D$ is aligned: *ld_align_w*[31:0]=0x0000000d.
  - The inputs for writing the RF in the next cycle are generated:
    - *mpc_rfwrite_w*=1. Write enabled.
    - *mpc_dest_w*[4:0]=01101.
    - *edp_wrdata_w*=0x0000000d.
- 5th cycle, W-Stage.
  - The RF is actually written. Register t5, which corresponds to register $13 according to file *regdef.h*, is updated: *rf13*=0x0000000d.

4. Exercises

Exercise 1. Analyze control signals

Sketch the hardware that feeds the following control signals. Table 2 provides the main modules and signals related with each control signal.

- ***mpc_imsgn_e***. This signal gives the sign of the Immediate, and it is used to sign-extending the 16-bit immediate to 32 bits.
- ***mpc_dcba_w***. This signal selects, at the A-Stage (Figure 2), the ALU result or the data read from the Data Memory.

Table 2. Exercise 1: main modules and signals

mpc_imsgn_e **control signal**

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
mpc_ir_e[31:0]	Instruction register
mpc_dcba_w **control signal**	
Module/Signal Name	**Description**
m14k_mpc_ctl	**Main pipeline control** – Control
loadin_e	Load operation
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
pbus_type_e	Type of memory operation: 0-nil, 1-load, 2-store, 3-pref, 4-sync, 5-ICacheOp, 6-DCacheOp

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

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: `lwpc`. The `lwpc` instruction computes the effective address as the addition between an immediate and the PC, reads memory using that effective address, and stores the read value in a register. You must use an *opcode* not used by any existing MIPS R3 instructions, such as 100111 (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 `lwpc` is shown below.

- Format:

- Description: *rs ← Memory[PC + SignExtend(immediate<<2)]*

The assembler does not support the new mnemonic. Thus, we include the `lwpc` instruction in our assembly code by providing its machine encoding. The following lines show how to include the `lwpc` instruction. You can find this program in the **main.c** file included in folder *Lab16_LW\Simulations\SimulationSources_LWPC*, where everything is provided (the *.elf* file, the text files for initializing memory, etc.). Observe that, similarly to the program from Figure 3, we include two `lwpc` instructions to the same address and analyze the second one, which thus hits in the D$.

```
"   li $t5, 0;"
"   nop;"
"   .word 0x9da00004;" // lwpc $t5, 0x00004
"   nop;"
"   li $t5, 0;"
"   nop;"
```

```
"    .word 0x9da00000;" // lwpc $t5, 0x00000
"    b .;"              // Stay here
```

Table 3. Exercise 3: main signals related to the lwpc instruction

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
lwpc_instr	New signal, which is 1 when an `lwpc` instruction is decoded
maj_ri_e	Trigger *reserved instruction* trap for illegal instruction decode from major opcode
dest_e	Destination register
m14k_edp	**Execution datapath**
aop_e[31:0]	SrcA for the adder from Figure 2. Assign from the PC at the E-Stage (*iva_e*)
dva_offset_e[31:0]	SrcB for the adder from Figure 2. Assign from the sign-extended immediate provided with the instruction

Table 3 includes the signals that we must modify for including an `lwpc` instruction. Moreover, below are some hints to help you implement this instruction:

- **Create a new signal *lwpc_instr***: Define a new signal in module **m14k_mpc_dec** that is 1 when an `lwpc` instruction is detected at decoding, and 0 otherwise. Provide this signal to all the modules that use it.
- **Disable exceptions:** For encoding the `lwpc` 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 = *100111*). 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 `lwpc` instruction.
- **Compute the effective address at the E-Stage:** For computing the effective address we use the adder shown at Figure 2:
    - SrcA comes from the PC, which is available in signal *iva_e* at the E-Stage
    - SrcB comes from the sign-extended immediate (constant provided with the instruction)
- **Select the result of the `lwpc` instruction to write to the register file (*edp_wrdata_w*):** We do not need any change in this case, as the opcode we are using already selects the result from the data memory.

- **Set register file write strobe (*mpc_rfwrite_w*) for `lwpc` instruction**: We do not need any change in this case, as the opcode we are using already sets the write strobe of the register file.
- **Compute destination register (*mpc_dest_w*) for `lwpc` instruction**: Signal *mpc_dest_w* depends on *dest_e*, which we must change for incorporating the destination register of `lwpc`.

To complete this exercise, do the following:

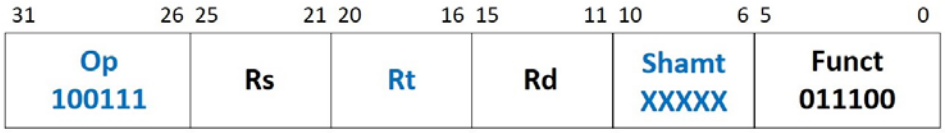
1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_lwpc**).
2. In the new folder, expand the capability of the MIPSfpga system so that it can write to the 7-segment displays on the Nexys4 DDR board, as explained in Lab 5.
3. In the new folder, expand MIPSfpga to implement `lwpc`, modifying 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_lwpc**) following the instructions provided in Step 1 - Lab 1, using the files from the new folder (**rtl_up_lwpc**).
5. Using the program shown above, provided in folder *Lab16_LW**Simulations**SimulationSources_LWPC*, 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 `lwpc` instruction is done around time 91500ns, thus configure the simulation runtime as explained in Lab 14. As for the waveform configuration file, you can use the file used for the `lw` 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 *Lab16_LW**Simulations**SimulationSources_LWPC*, 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 *Lab16_LW**Simulations**SimulationSources_LWPC*).
      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, which contains the result of the `lwpc` 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 $t5 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 *0x8000021c` (set breakpoint before `lwpc` 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: `lwi`

In this exercise you will expand the MIPSfpga (microAptiv) core to perform the load word immediate (`lwi`) instruction (proposed in Exercise 4.2. of [2]). This instruction computes the effective address as the addition between two registers, reads memory using that effective address, and stores the read value in a third register. The machine code format and functionality of `lwi` is shown below.

- Format:



- Description: *Reg[Rd] = Mem[Reg[Rs] + Reg[Rt]]*

Given the instruction format, we encode it as a *special* instruction. We use an *opcode* of 000000 and a *function field* not used by any existing MIPS R3 instructions, such as 011100. Because the assembler does not support the new mnemonic, you must write the `lwi` instruction in machine code in the assembly program. The following lines show how to include two `lwi` instructions. You can find this program in the **main.c** file included in folder *Lab16_LW\Simulations\SimulationSources_LWI*, where everything is ready (the *.elf* file, the text files for initializing memory, etc.). Observe that, similarly to the program from Figure 3, we include two `lwi` instructions to the same address and analyze the second one, which thus hits in the D$.

```
"   lui $t6, 0x8000;"
"   addiu $t6, $t6, test_data;"
"   addiu $t3, $zero, 4;"
```

```
"   nop;"
"   .word 0x01cb681c;"  // lwi $t5, $t6, $t3
"   li $t5, 0;"
"   nop;"
"   .word 0x01cb681c;"  // lwi $t5, $t6, $t3
"   b .;"               // Stay here
```

Table 4. Exercise 4: main signals related to the lwi instruction

Module/Signal Name	Description
m14k_mpc_dec	**Main pipeline control** – MIPS32 instruction decoder
lwi_instr_e	New signal, which is 1 when a `lwi` instruction is decoded and 0 otherwise
spec_ri_e	Trigger *reserved instruction* trap for *special* instruction
mpc_ldst_e	This signal must be 1 for load/store operations
load_e	This signal must be 1 for load operations
m14k_edp	**Execution datapath**
dva_offset_e[31:0]	SrcB of the adder must be provided from *edp_bbus_e[31:0]*

Table 4 includes the signals that we must modify for including a `lwi` instruction. Moreover, below are some hints to help you implement this instruction:

- **Create a new signal *lwi_instr_e***: Define a new signal in module **m14k_mpc_dec** that is 1 when a `lwi` instruction is detected at decoding, and 0 otherwise. Provide this signal to all the modules that use it.
- **Disable exceptions:** For encoding the `lwi` 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* = *011100*). 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 so that it inhibits the *Reserved Instruction* exception for a `lwi` instruction.
- **Decode load instruction:** By default, *special* (i.e., R-type) instructions do not access memory. Thus, we have to change some control signals related to memory access in order to support a `lwi` instruction.
    - Signal *mpc_ldst_e* is set for load/store operations. Thus, we have to modify the computation of this signal.
    - Signal *load_e* is set for load operations. Thus, we have to modify the computation of this signal.

    - Signal *pbus_type_e [2:0]* encodes the memory type. For load instructions, it equals 010. We do not need to change this signal, as it already gets the right value for a `lwi` instruction.
- **Compute the effective address at the E-Stage:** For computing the effective address we use the adder shown in Figure 2:
    - SrcA (signal *aop_e[31:0]*) is computed as for all other *special* (i.e., R-type) instructions
    - SrcB (signal *dva_offset_e[31:0]*) comes from the register file or from the forwarding logic, through signal *edp_bbus_e[31:0]*
- **Select the value read from the data memory to write to the register file (*edp_wrdata_w*):** The values established above for the control signals concerning the load (*mpc_ldst_e* and *load_e*) select the data from memory. Thus, no changes are required here.
- **Set register file write strobe (*mpc_rfwrite_w*) and compute destination register (*mpc_dest_w*) for `seq` instruction**: These signals are correctly computed for *special* instructions. Thus, again no changes are required.

To implement the `lwi` instruction, do the following:

1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_lwi**).
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 `lwi`, modifying the Verilog files of the soft-core by using the instructions provided above. You will have to change the two Verilog files included in Table 4 (**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_lwi**) following the instructions provided in Step 1 - Lab 1, using the files from the new folder (**rtl_up_lwi**).
5. Using the program shown above, and also provided in folder *Lab16_LW**Simulations**SimulationSources_LWI*, debug your implementation with Vivado's XSIM simulator. Follow the steps explained in Section 4 of Lab 14 for configuring the simulator. The fetch of the first `lwi` instruction is done around 91540ns, thus configure the simulation runtime as explained in Lab 14. As for the waveform configuration file, you can use the file used for the `lw` 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 *Lab16_LW**Simulations**SimulationSources_LWI*, 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 *Lab16_LW**Simulations**SimulationSources_LWI*).
  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 `lwi` 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 *0x80000220` (set breakpoint before second `lwi` 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)

5. References

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

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