****

**Lab 18**

**MicroAptiv’s Hazard Logic**



**Lab 18**

**MicroAptiv’s Hazard Logic**

# Introduction

In this lab we analyze microAptiv’s Hazard Unit. Data hazards are handled analogously in microAptiv and in the processor implemented in *DDCA* [1], thus, before starting with this lab, it is recommended that you study Section 7.5.3 of [1], where the Hazard Unit is explained in detail. Note that, as opposed to data hazards, control hazards are handled differently in microAptiv, where a delay slot is used (Lab 17), and in the processor from *DDCA* [1].

This lab includes an initial section (Section 2) with a theoretical explanation of the Hazard Unit used in microAptiv. Then, Section 3 explains how to display Core signals related with the Hazard Unit on the LEDs, so that you can analyze them using the slow clock option explained in Lab 10. Finally, the lab proposes some exercises where you study, by means of simulation and execution on the board, several instruction sequences that include different hazards.

# MicroAptiv’s Hazard Unit

In this section we explain the logic provided by microAptiv for handling two different hazard situations: RAW hazard between arithmetic-logic instructions, and RAW hazard between an lw instruction and a subsequent arithmetic-logic instruction.

## RAW hazard between arithmetic-logic instructions

Figure 3 in Lab 14 and Figure 2 in Lab 15 illustrate, respectively, the Arithmetic and Logic Units available in microAptiv. As shown in those figures, each operand of an arithmetic-logic instruction comes from the register file, if no hazard exists, or it is forwarded from a subsequent stage, if a hazard is detected. Figure 1 adds some more details (highlighted in blue) to the figures referred above. Specifically, two forwarding paths are included, one from the M-Stage to the E-Stage and another one from the A-Stage to the E-Stage. Note that a forwarding path from the W-Stage to the E-Stage is not necessary, as the register file is capable of sending the data being written to the *Rd* at the W-Stage directly to the *Rs* and *Rt* read ports. Table 1 describes the main signals related with microAptiv’s Hazard Unit.

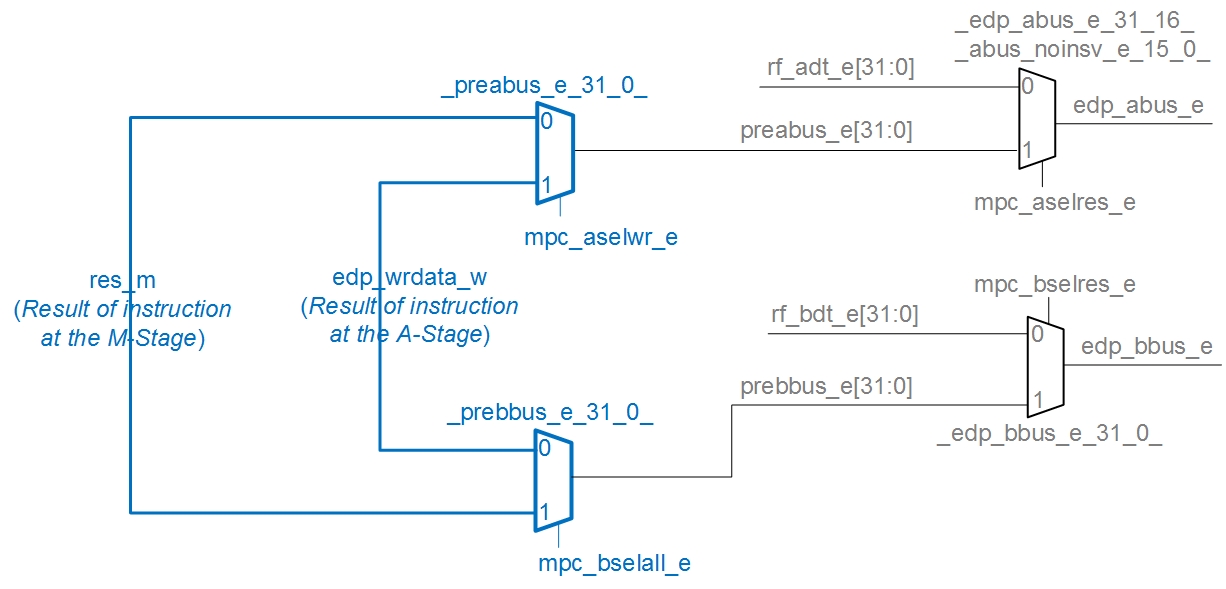


Figure 1. MicroAptiv forwarding logic.

Table 1. Main signals related with the Hazard Unit

|  |  |
| --- | --- |
| **Module/Signal Name** | **Description** |
| **m14k\_mpc\_ctl** | **Main pipeline control** – Control |
| *mpc\_aselres\_e* | Adder SrcA provided by the register file (*mpc\_aselres\_e*=0) or through forwarding (*mpc\_aselres\_e*=1) |
| *mpc\_bselres\_e* | Adder SrcB provided by the register file (*mpc\_bselres\_e*=0) or through forwarding (*mpc\_bselres\_e*=1) |
| *mpc\_aselwr\_e* | Forwarding value provided by the instruction at the M-Stage (*mpc\_aselwr\_e*=0) or by the instruction at the A-Stage (*mpc\_aselwr\_e*=1) |
| *mpc\_bselall\_e* | Forwarding value provided by the instruction at the M-Stage (*mpc\_bselall\_e*=1) or by the instruction at the A-Stage (*mpc\_bselall\_e*=0) |
| **m14k\_edp** | **Execution datapath** |
| *rf\_adt\_e* | First value read from the register file |
| *rf\_bdt\_e* | Second value read from the register file |
| *preabus\_e* | First value from forwarding logic |
| *prebbus\_e* | Second value from forwarding logic |
| *res\_m* | Result of instruction at the M-Stage |
| *edp\_wrdata\_w* | Result of instruction at the A-Stage |

As shown in Figure 1, signals *mpc\_aselres\_e* and *mpc\_aselwr\_e* determine the source of the first operand provided to the Arithmetic/Logic Unit, whereas signals *mpc\_bselres\_e* and *mpc\_bselall\_e* determine the source of the second operand. The next two examples explain the value that the Hazard Unit assigns to these signals under different situations.

**Example 1**. Forwarding from the M-Stage to the E-Stage.

add **$8**, $9, $10

sub $16, **$8**, $15

When the sub instruction is at the E-Stage, the Hazard Unit detects that its first operand depends on the result of the preceding instruction (add instruction), and sets the multiplexer control signals of Figure 1 as follows:

*mpc\_aselres\_e* = 1

*mpc\_aselwr\_e* = 0

*mpc\_bselres\_e* = 0

*mpc\_bselall\_e* = the value of this signal does not affect in this case

**Example 2**. Forwarding from the A-Stage to the E-Stage.

slt **$14**, $17, $18

or $4, $5, $6

and $12, $16, **$14**

When the and instruction is at the E-Stage, the Hazard Unit detects that its second operand depends on the result of the slt instruction, and sets the control signals of the multiplexers of Figure 1 as follows:

*mpc\_aselres\_e* = 0

*mpc\_aselwr\_e* = the value of this signal does not affect in this case

*mpc\_bselres\_e* = 1

*mpc\_bselall\_e* = 0

## RAW hazard between a lw and a subsequent arithmetic-logic instruction

This kind of RAW hazard also uses the forwarding logic from Figure 1. However, in addition to that logic, when an arithmetic-logic instruction is dependent on an immediately preceding load instruction, a bubble must be inserted between the two instructions, as the lw instruction only has the requested data after traversing the Alignment Logic at the A-Stage (Lab 16). The bubble makes the add and subsequent instructions to be stopped for one cycle, while earlier in-flight instructions (lw and previous instructions) are allowed to progress. The main signals related with the bubble insertion are shown in Table 2.

Table 2. Main signals related with the bubble insertion

|  |  |
| --- | --- |
| **Module/Signal Name** | **Description** |
| **m14k\_mpc\_ctl** | Main pipeline control – Control |
| *prod\_cons\_stall\_req* | RAW dependence between instruction at M-Stage and instruction at E-Stage |
| *ldst\_m* | Instruction at the M-Stage is a load or a store |
| *mpc\_run\_ie* | Advance (*mpc\_run\_ie*=1) or not (*mpc\_run\_ie*=0) the I-Stage and the E-Stage |

A pipeline stage is stalled by inhibiting the updating of the Pipeline Registers (PRs). Specifically, the add instruction is stalled at the E-Stage by inhibiting the updating of the following PRs: *\_ie\_pipe\_out\_50\_0\_*, *\_int\_ir\_e\_31\_0\_*, and *\_edp\_iva\_i\_31\_0\_*. You can observe in the soft-core that all these registers are controlled by signal *mpc\_run\_ie*, which is 1 when the I-Stage and E-Stage can progress, and 0 when they must stall. The next example explains the value that the Hazard Unit assigns to these signals.

**Example**. Forwarding from a lw instruction at the A-Stage to a sub instruction at the E-Stage:

lw **$8**, 0($9)

sub $16, **$8**, $15

When the sub instruction is at the E-Stage and the lw instruction is at the M-Stage, the Hazard Unit inserts a bubble, by making *mpc\_run\_ie* = 0. As a result, the sub and subsequent instructions stall, whereas the lw and previous instructions advance to the next stage (*mpc\_run\_m*, which controls stalling of the instruction at the M-Stage, and *mpc\_run\_w*, which controls stalling of the instruction at the A-Stage, are both equal to 1).

In the following cycle, when the sub instruction is again at the E-Stage but the lw instruction is at the A-Stage, the Hazard Unit detects a dependency among both instructions and sets the control signals of the multiplexers of Figure 1 as follows:

*mpc\_aselres\_e* = 1

*mpc\_aselwr\_e* = 1

*mpc\_bselres\_e* = 0

*mpc\_bselall\_e* = the value of this signal does not affect in this case

# Display signals through the board peripherals

In this section we illustrate how to propagate Core signals to the board peripherals. We first reconfigure MIPSfpga for supporting the slow clock option (explained in detail in Lab 10) and then we reconfigure it to output some signals related with the Hazard Logic. Specifically, we output through the LEDs the following signals:

* LED5: *clock signal*
* LED4: *mpc\_aselres\_e*
* LED3: *mpc\_bselres\_e*
* LED2: *mpc\_aselwr\_e*
* LED1: *mpc\_bselall\_e*
* LED0: *mpc\_run\_ie*

Note that you could use the same procedure for propagating any other signal that you wish to analyze on the LEDs.

In folder *Part3\_Core\Lab18\_HazardLogic\Verilog* we provide both a *bitfile* and a set of Verilog files that you can use for creating the *bitfile* yourself. To do the latter, use the files provided in the *Part3\_Core\Lab18\_HazardLogic\Verilog* folder along with the original MIPSfpga rtl\_up files to create a Vivado project and a bitfile (refer to Lab 1 for a refresher of how to do this). Notice that some of the files replace the original files. These Verilog files were created according to the following steps:

**Step 1**. Modify MIPSfpga RTL files to support the slow clock option, as explained in detail in Lab 10

**Step 2**. Modify MIPSfpga Core RTL files to propagate signals to the top of the module hierarchy

**Step 3**. Modify MIPSfpga system RTL files to propagate signals to the LEDs

**Step 1. Modify MIPSfpga RTL files to support the slow clock option, as explained in detail in Lab 10**

Starting with the default MIPSfpga system provided in *MIPSfpga\_GSG\rtl\_up*, include the necessary hardware to support the slow clock option, as explained in Lab 10.

**Step 2. Modify MIPSfpga Core RTL files to propagate signals to the top of the module hierarchy**

Starting with the MIPSfpga system implemented in Step 1, modify the core RTL files in order to propagate some signals to the top of the module hierarchy. We want to be able to analyze the signals controlling the multiplexers shown in Figure 1 (*mpc\_aselres\_e*, *mpc\_aselwr\_e*, *mpc\_bselres\_e*, *mpc\_bselall\_e*), and signal *mpc\_run\_ie*, which determines if the I-Stage and the E-Stage must stall. Thus, we need to propagate those signals through three levels of hierarchy: module **m14k\_core**, module **m14k\_cpu** and module **m14k\_top**.

Modify file **core/m14k\_core.v** as follows:

1. Add the output ports for the group of signals stated above by substituting line 371 (“antitamper\_present);”) for the following lines:

antitamper\_present

`ifdef MFP\_DEMO\_PIPE\_BYPASS

,

mpc\_aselres\_e,

mpc\_aselwr\_e,

mpc\_bselall\_e,

mpc\_bselres\_e,

mpc\_run\_ie

`endif

);

`ifdef MFP\_DEMO\_PIPE\_BYPASS

output mpc\_aselres\_e;

output mpc\_aselwr\_e;

output mpc\_bselall\_e;

output mpc\_bselres\_e;

output mpc\_run\_ie;

`endif

2. Include the **mfp\_config.vh** header file by adding the following line at the beginning:

`include "mfp\_config.vh"

Modify file **core/m14k\_cpu.v** as follows:

1. Add the connections for module **m14k\_core** instantiation by substituting line 1002 (“.tcb\_bistfrom(tcb\_bistfrom));”) for the following lines:

.tcb\_bistfrom(tcb\_bistfrom)

`ifdef MFP\_DEMO\_PIPE\_BYPASS

,

.mpc\_aselres\_e(mpc\_aselres\_e),

.mpc\_aselwr\_e(mpc\_aselwr\_e),

.mpc\_bselall\_e(mpc\_bselall\_e),

.mpc\_bselres\_e(mpc\_bselres\_e),

.mpc\_run\_ie(mpc\_run\_ie)

`endif

);

2. Add the output ports for the group of signals stated above by substituting line 253 (“SI\_PCInt);”) for the following lines:

SI\_PCInt

`ifdef MFP\_DEMO\_PIPE\_BYPASS

,

mpc\_aselres\_e,

mpc\_aselwr\_e,

mpc\_bselall\_e,

mpc\_bselres\_e,

mpc\_run\_ie

`endif

);

`ifdef MFP\_DEMO\_PIPE\_BYPASS

output mpc\_aselres\_e;

output mpc\_aselwr\_e;

output mpc\_bselall\_e;

output mpc\_bselres\_e;

output mpc\_run\_ie;

`endif

3. Include the **mfp\_config.vh** header file by adding the following line at the beginning:

`include "mfp\_config.vh"

Finally, modify file **core/m14k\_top.v** in similar fashion as **core/m14k\_cpu.v**.

**Step 3. Modify MIPSfpga system RTL files to propagate signals to the LEDs**

As explained before, we want to output through the LEDs the clock signal (LD5) and signals *mpc\_aselres\_e* (LD4), *mpc\_bselres\_e* (LD3), *mpc\_aselwr\_e* (LD2), *mpc\_bselall\_e* (LD1), and *mpc\_run\_ie* (LD0).

For that purpose, modify file **system/mpf\_sys.v** as follows:

1. Output the set of signals stated above through the LEDs by adding, right before the end of the module (line 367), the following lines:

`ifdef MFP\_DEMO\_PIPE\_BYPASS

assign IO\_LED = { { (`MFP\_N\_LED-6) { 1'b0 } },

HCLK,

mpc\_aselres\_e, // Bypass res\_m as src A

mpc\_bselres\_e, // Bypass res\_m as src B

mpc\_aselwr\_e, // Bypass res\_w as src A

mpc\_bselall\_e, // Bypass res\_w as src B

mpc\_run\_ie

};

`endif

2. Modify the connections for module **mfp\_ahb\_withloader** instantiation by substituting line 303 (“.IO\_LED (IO\_LED),”) for the following lines:

`ifdef MFP\_DEMO\_PIPE\_BYPASS

.IO\_LED ( ),

`else

.IO\_LED ( IO\_LED ),

`endif

3. Include the connections for module **m14k\_top** instantiation by substituting line 281 (“SI\_PCInt);”), the following lines:

SI\_PCInt

`ifdef MFP\_DEMO\_PIPE\_BYPASS

,

mpc\_aselres\_e,

mpc\_aselwr\_e,

mpc\_bselall\_e,

mpc\_bselres\_e,

mpc\_run\_ie

`endif

);

4. Add, after line 177 (“wire MFP\_Reset\_serialload; // …”), the following lines:

`ifdef MFP\_DEMO\_PIPE\_BYPASS

wire mpc\_aselres\_e;

wire mpc\_aselwr\_e;

wire mpc\_bselall\_e;

wire mpc\_bselres\_e;

wire mpc\_run\_ie;

`endif

Finally, define MFP\_DEMO\_PIPE\_BYPASS in file **system/mfp\_config.vh**, by adding the following line:

`define MFP\_DEMO\_PIPE\_BYPASS

# Exercises

## Exercise 1. Analyze a RAW hazard between arithmetic-logic instructions

The following program contains 3 consecutive Arithmetic-Logic (AL) instructions with 2 RAW hazards; specifically, the add instruction depends on the results of the two previous addiu instructions. Note that, as we did in Labs 13 and 17, we are using the assembler directive “*.set noreorder*”, which tells the assembler that the programmer is in control and thus it must not move instructions about.

loop:

add $t1, $zero, $zero;

add $t2, $zero, $zero;

nop;

**addiu $t1, $zero, 1;**

**addiu $t2, $zero, 2;**

**add $t3, $t2, $t1;**

b loop;

nop;

You must perform the next steps:

* Sketch the hardware for the forwarding logic explained in Section 2.a.
* Explain the results of the simulation of the program shown above, provided in folder *Lab18\_HazardLogic\Exercise1\SimulationSources*. For performing this simulation you should create a new Vivado project (as explained in Lab 14). Use the new soft-core, modified as explained in Section 3. Moreover, use the waveform configuration file provided in *Lab18\_HazardLogic\Exercise1\testbench\_boot\_behav\_Forwarding.wcfg*.
* Explain the results of the execution on the board of the previous program from the point of view of the Forwarding Logic. Follow the next steps:
  1. Synthesize the new processor, as explained in Step 3 – Lab 1.
  2. Program the FPGA board, as explained in Step 4 – Lab 1.
  3. Download the program to the board, as explained in Step 3 – Section 2 of Lab 2. Make sure that switches 0 and 1 are low.
  4. Once the program is running, turn on switch 0, which activates the slow clock. Observe and explain the results.
* Create other programs presenting other RAW hazards between arithmetic-logic instructions, simulate them on Vivado’s simulator and execute them on the board, and explain the results.

## Exercise 2. Analyze a RAW hazard between a lw and a subsequent sub

The following program contains a RAW hazards between a lw instruction and a subsequent add instruction.

lui $t6, 0x8000;

addiu $t6, $t6, test\_data;

loop:

add $t1, $zero, $zero;

nop;

**lw $t1, 0($t6);**

**add $t2, $zero, $t1;**

b loop;

nop;

You must perform the next steps:

* Sketch the hardware for the forwarding logic explained in Section 2-b.
* Explain the results of the simulation of the program shown above, provided in folder *Lab18\_HazardLogic\Exercise2\SimulationSources*. For performing this simulation you should create a new Vivado project (as explained in Lab 14). Use the new soft-core, modified as explained in Section 3. Moreover, use the waveform configuration file provided in *Lab18\_HazardLogic\Exercise2\testbench\_boot\_behav\_Forwarding.wcfg*.
* Explain the results of the execution on the board of the previous program from the point of view of the Forwarding Logic. Follow the next steps:
  1. Synthesize the new processor, as explained in Step 3 – Lab 1.
  2. Program the FPGA board, as explained in Step 4 – Lab 1.
  3. Download the program to the board, as explained in Step 3 – Section 2 of Lab 2. Make sure that switches 0 and 1 are low.
  4. Once the program is running, turn on switch 0, which activates the slow clock. Observe and explain the results.

# References

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