MIPSfpga by Imagination

Lab 18

MicroAptiv's Hazard Logic

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

Lab 18
MicroAptiv's Hazard Logic

1. 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.

2. 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.

a. 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.

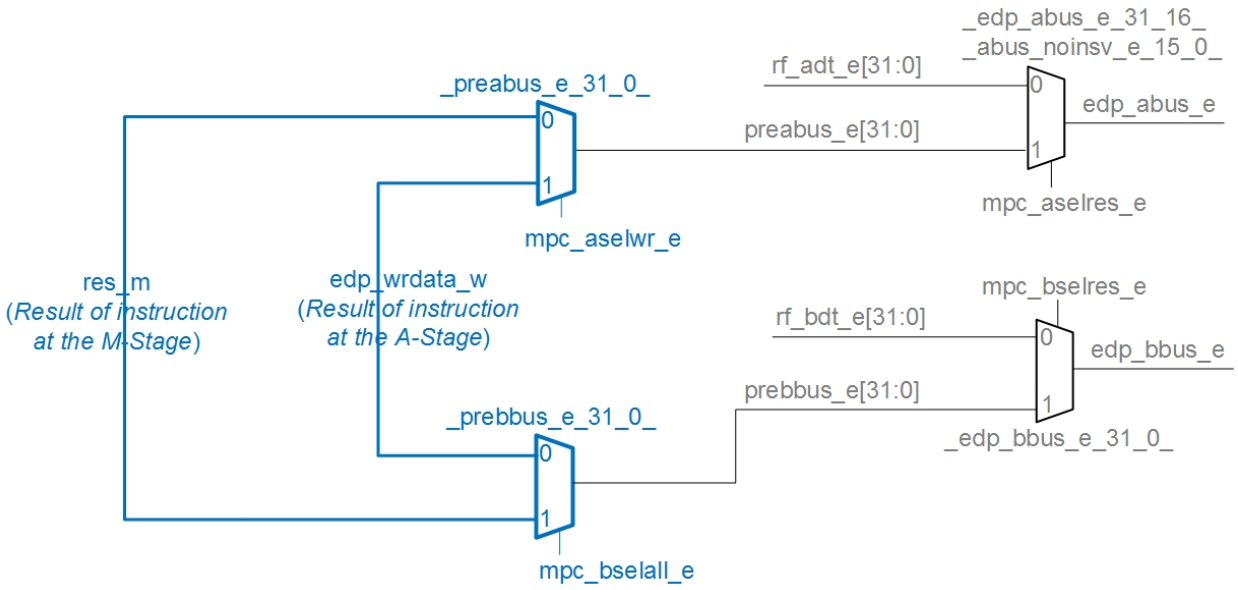

Figure 1. MicroAptiv forwarding logic.

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

b. 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

3. 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
```

4. 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.

5. References

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