MIPS Open Developer Day Package. Lab MO6 - The first glance into pipelining

0. Note. This documentation is in the process of being updated

This documentation is in the process of being updated to match the latest version of the source code. The planned updates include:

1. Informing the user how to use makefiles instead of short Linux scripts, Windows batch files and GUI. To see all available make actions just run *make* in any board or program directory example:

   make in subdirectories of *boards* directory:

   ```
   make help   - show this message
   make all    - clean, create the board project and run the synthesis
   make clean  - delete synth folder
   make create - create the board project
   make open   - open the board project
   make build  - build the board project
   make load   - program the FPGA board
   ```

   make in subdirectories of *programs* directory:

   ```
   make help       - show this message
   make all        - alternative for: compile program size disasm readmemh srecord
   make program    - build program.elf from sources
   make compile    - compile all C sources to ASM
   make size       - show program size information
   make disasm     - disassemble program.elf
   make readmemh   - create verilog memory init file for simulation
   make srecord    - create Motorola S-record file to use it with UART loader
   make clean      - delete all created files
   make load       - load program into the device memory, run it and detach gdb
   make debug      - load program into the device memory, wait for gdb commands
   make attach     - attach to the device, wait for gdb commands
   make uart       - load program into the device memory using UART loader
   make modelsim   - simulate program and device using Modelsim
   make icarus     - simulate program and device using Icarus Verilog
   make gtkwave    - show the result of Icarus Verilog simulation in GTKWave
   ```

2. Adding support for *make uart UART=N* where *N* is USB-to-UART device number (*/dev/ttyUSB0, 1, 2, ... N*).

3. Light Sensor in peripheral integration lab is now integrated not as an addional GPIO (General Purpose I/O) device inside GPIO AHB-Lite slave, but as a separate AHB-Lite slave.

4. Source code for pipeline bypass lab (*Lab MO6 - The first glance into pipelining*) does not match the documentation. Please look inside the code to see what it does.

5. The new CorExtend / UDI - User Defined Instructions example does not have an instruction. Please refer to MIPS Open Day slides to figure out what it does.

1. Introduction

In this lab a student directly observes pipeline forwarding in action. The pipeline forwarding, or bypassing, is one of the most important microarchitectural tricks in CPU design. During the lab, a student connects bypass control signals of MIPSfpga CPU core to LED signals on FPGA board, and watches with his eyes the pattern of flashing LEDs, when running different assembly programs on the synthesized MIPSfpga system configured inside the FPGA. Obviously, in order to see anything meaningful with this way of observation, the design uses a really slow clock, 0.75 Hz, less than a beat per second.

The method used in this lab is not a substitute for traditional debug, trace and performance measuring methods used to evaluate microarchitectural features by an experienced CPU designer. This lab is intended for a beginning student who wants to visualize the work of the pipeline in a most straightforward fashion, without using too much imagination. The subsequent labs in MIPSfpga package use more complicated and indirect, but more generally applicable methods to explore the pipelining.

2. The theory of operation

Pipelining is a technique to improve the throughput of a digital design by splitting the data processing into stages and moving the items to be processed through those stages, starting new items before finishing old items. The process works just like an assembly line in a car factory: while each individual car takes a long time to assemble, the assembly line as a whole produces a new car every few minutes.

Figure 1 show the application of the pipelining principle to CPU design, using as an example the classic five-stage pipeline utilized in early implementations of MIPS architecture back in 1980s. The items to be processed are the CPU instructions, and the stages include: fetching the instruction from the instruction memory; decoding the instruction; reading the values of the applicable registers (visible to the software, they are called *the architecture registers*); performing arithmetic and data memory operations; and finally, writing the results back to the architecture register file.

Figure 1. An illustration of the idea of pipelining from the slides that accompany book *Digital Design and Computer Architecture by David Harris and Sarah Harris, 2012* (later referred as *DDCA*).

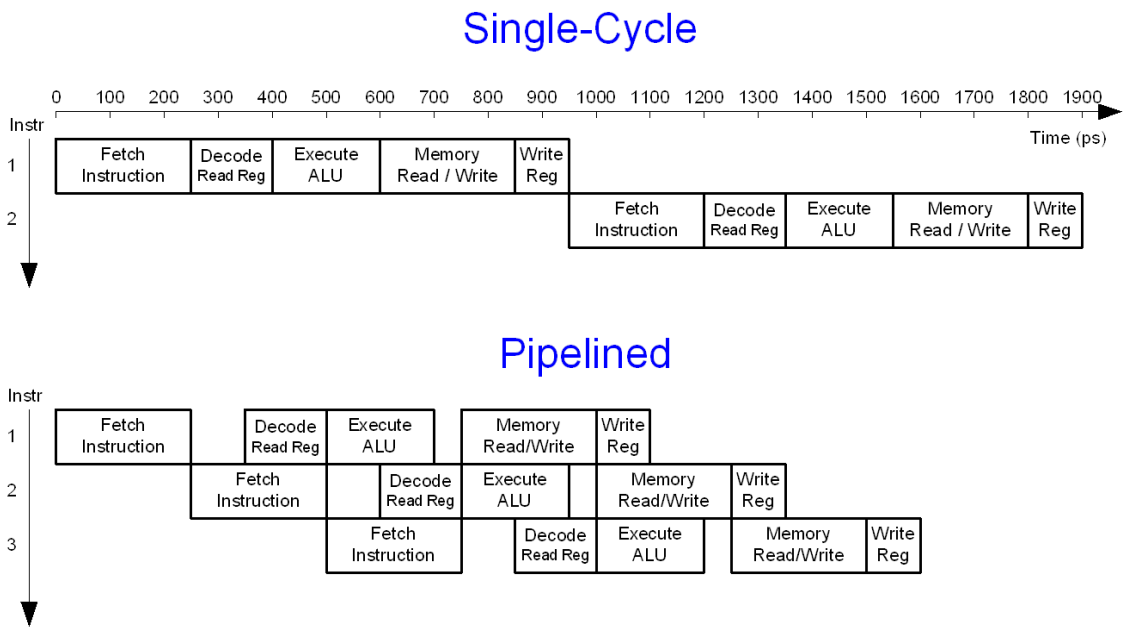


Figure 2 illustrates how a sequence of MIPS instructions progresses through the classic MIPS pipeline.

Figure 2. An illustration that shows how a sequence of MIPS instructions progresses through the classic MIPS pipeline. From *DDCA*.

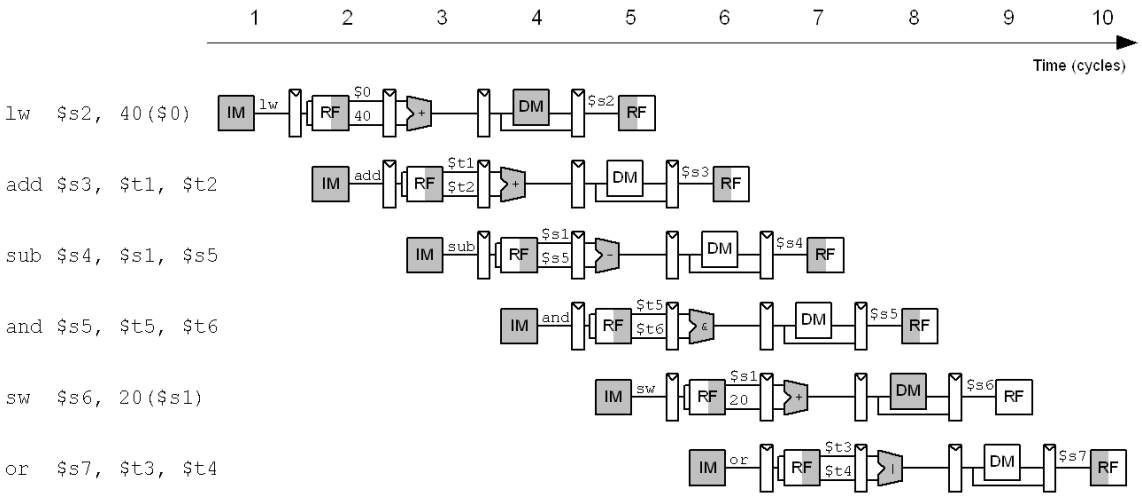
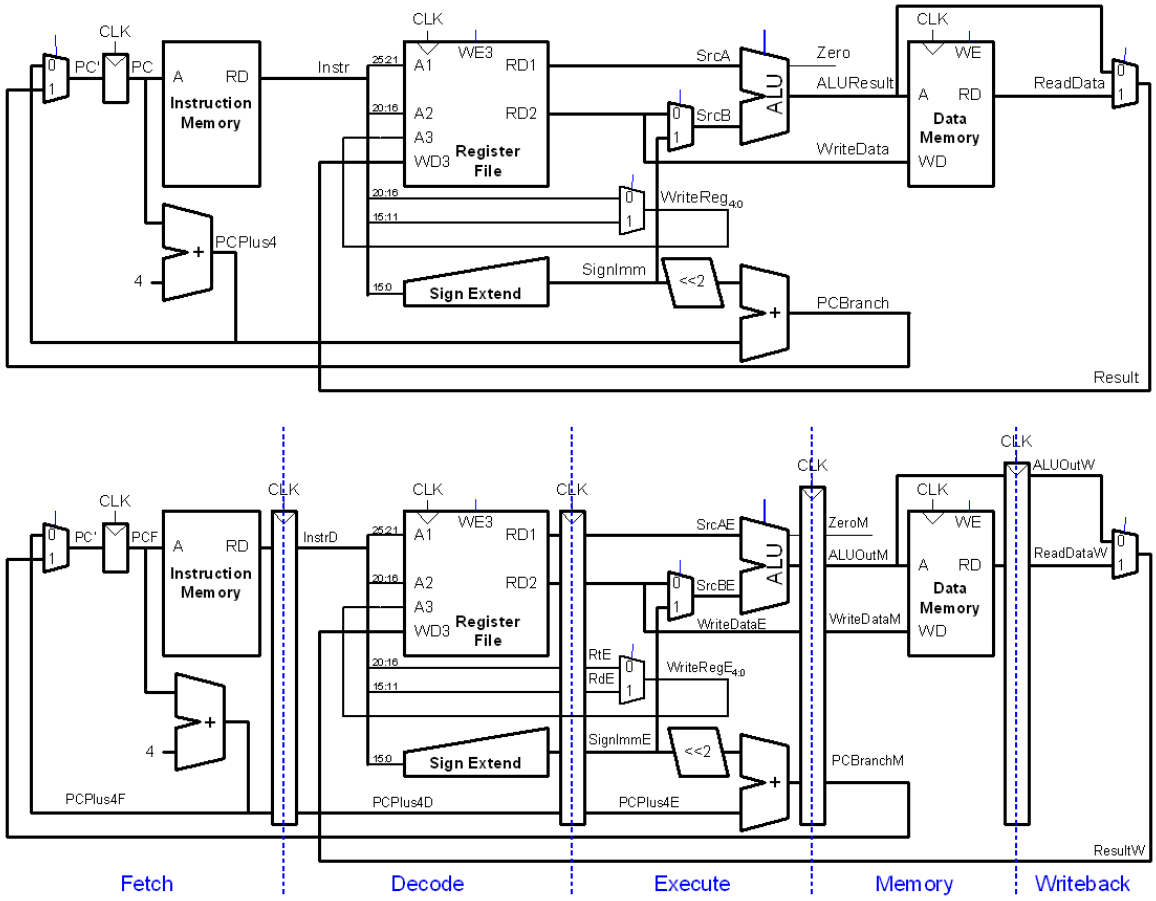


Figure 3 shows the difference between a single-cycle, non-pipelined microarchitectural implementation of MIPS architecture and its classic pipelined implementation. Notice the added flip-flops, non-architectural registers, invisible for the software engineer.

Figure 3. The difference between a single-cycle, non-pipelined microarchitectural implementation of MIPS architecture and its classic pipelined implementation.



A fundamental problem with the straightforward pipelined implementation is the presence of so-called data hazards. Such hazards occur when an instruction uses data that are supposed to be generated by one of the previous instructions, and this data is not yet written into the architectural register (architectural register is a register visible to software).

There are several methods used to resolve data pipeline hazards. One method is to simply add empty (*nop*) instructions to the software. Another method is to add so-called stalls to the hardware, which forces an instruction to wait for the previous instruction to complete before using its result. Both methods result in slowing down the

program execution. However there is another method, called pipeline forwarding, or, more specifically, bypassing that involves forwarding the calculated data from an earlier instruction to the instruction which uses the earlier instruction's result even before the data gets written into the applicable architectural register. This method is illustrated in **Figure 4** and the applicable schematic is shown on **Figure 5**.

Figure 4. A sequence of MIPS instructions progressing through the classic MIPS pipeline when the forwarding is present. From *DDCA*.

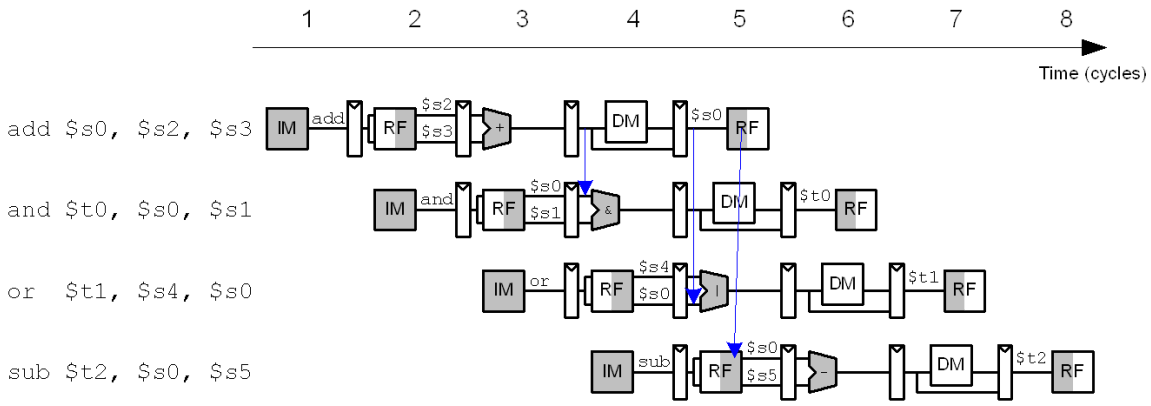
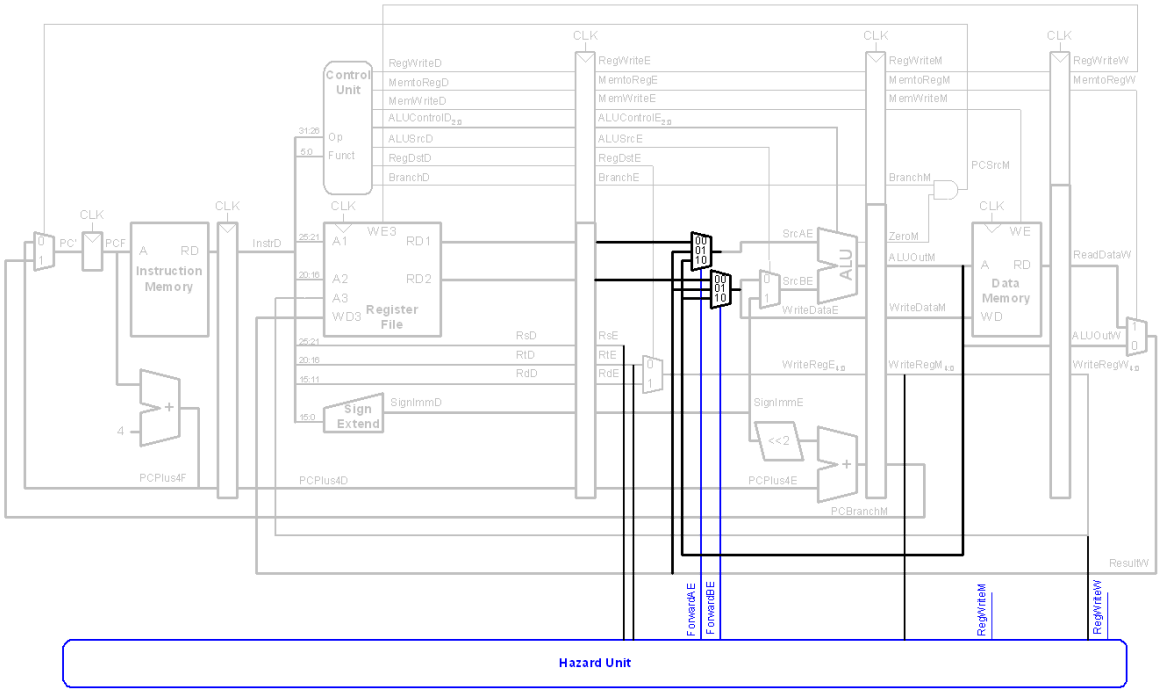
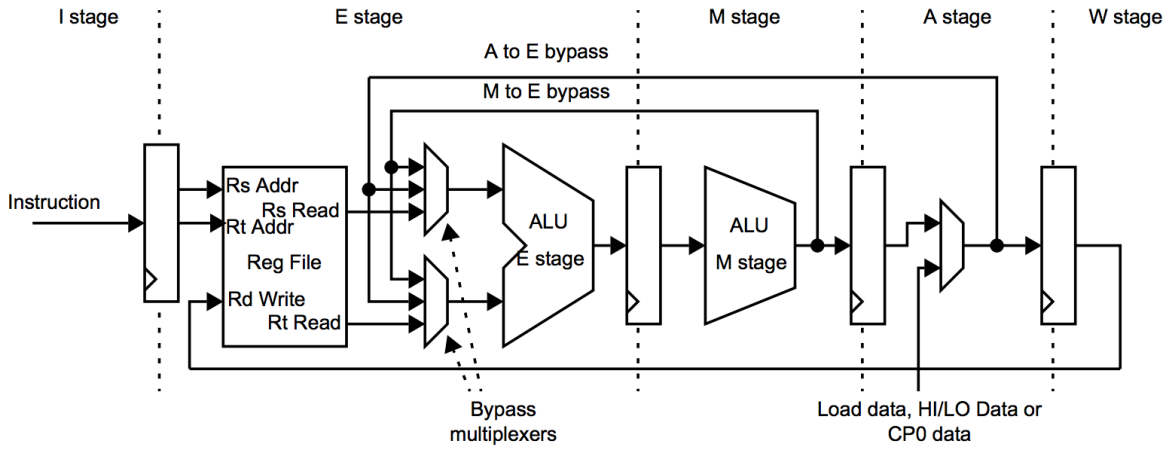


Figure 5. The additional logic needed for the classic MIPS pipeline to support pipeline forwarding / bypasses. From *DDCA*.



The pipeline in the *MIPS microAptiv UP* core, which is used in MIPSfpga, is similar to, but not the same as, the classic MIPS pipeline. The difference is due to the fact that the pipeline of MIPSfpga core is somewhat better balanced to achieve higher clock frequency. As a result, bypass signals in MIPSfpga are different, as shown on **Figure 6**.

Figure 6. Pipeline bypass signals in *MIPS microAptiv UP* core used in MIPSfpga.



3. Lab steps

This section outlines the sequence of steps, necessary to complete the lab. Almost all generic steps in this lab are the same as in *MIPSfpga 2.0 Lab MO1. Using MIPSfpga with Serial Loader Flow that does not require BusBlaster board and OpenOCD software*. Such generic steps are not described in this section. Only the steps different from *Lab MO1* are explained in details.

3.1. Backup core RTL code

This lab requires you to modify core RTL code to connect the internal signals to the external LEDs. Before modifying code for the core, we recommend you back it up.

Under Windows:

```
cd MIPSfpga_top_directory
cp -r core_rtl/* core_rtl_golden
```

Under Linux:

```
cd MIPSfpga_top_directory
xcopy /i core_rtl core_rtl_golden
```

3.2. Examine core RTL to find the bypass signals

We need to figure out which signals are involved in the pipeline bypass. One way to do this is to simply search for *"bypass"* string in core RTL files. The search should be case-insensitive:

Under Windows:

```
cd core_rtl
findstr /i bypass *
```

Under Linux:

```
cd core_rtl
grep -i bypass *
```

You will notice the following four signals:

```
m14k_mpc.v:output mpc_aselres_e;  // Bypass res_m as src A
m14k_mpc.v:output mpc_aselwr_e;   // Bypass res_w as src A
m14k_mpc.v:output mpc_bselall_e;  // Bypass res_w as src B
m14k_mpc.v:output mpc_bselres_e;  // Bypass res_m as src B
```

Those signals are involved in bypass control. The name (*m14k_mpc*) of the module which produces those signals means "master pipeline controller".

3.3. Modify core RTL files to propagate the bypass signals to the top of the core's module hierarchy

In order to connect the bypass signals to the external LEDs, we need to propagate those signals to the top through three levels of hierarchy: *m14k_core*, *m14k_cpu* and *m14k_top*.

Open *core_rtl/m14k_core.v* in any editor, add *`include "mfp_ahb_lite_matrix_config.vh"* before the module and the following code to the module port declarations:

File *core_rtl/m14k_core.v*

```
//      antitamper_present);

        antitamper_present

`ifdef MFP_DEMO_PIPE_BYPASS
        ,
        mpc_aselres_e,
        mpc_aselwr_e,
        mpc_bselall_e,
        mpc_bselres_e
`endif

);

`ifdef MFP_DEMO_PIPE_BYPASS
output mpc_aselres_e;
output mpc_aselwr_e;
output mpc_bselall_e;
output mpc_bselres_e;
`endif
```

Modify *core_rtl/m14k_cpu.v* in three places: add *`include "mfp_ahb_lite_matrix_config.vh"* before the module, add the output ports for the bypass signals, add the connections for *m14k_core* instantiation:

File *core_rtl/m14k_cpu.v*

```
. . . . . .

`include "mfp_ahb_lite_matrix_config.vh"

module m14k_cpu(

        . . . . . .

//        SI_PCInt);

        SI_PCInt

`ifdef MFP_DEMO_PIPE_BYPASS
        ,
        mpc_aselres_e,
        mpc_aselwr_e,
        mpc_bselall_e,
        mpc_bselres_e
`endif

);

`ifdef MFP_DEMO_PIPE_BYPASS
```

```
output mpc_aselres_e;
output mpc_aselwr_e;
output mpc_bselall_e;
output mpc_bselres_e;
`endif

. . . . . .

m14k_core core (

        . . . . . .

        .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)

`endif
        );
```

Modify *core_rtl/m14k_top.v* in similar fashion as *core_rtl/m14k_cpu.v*.

3.4. Review lab code modifications in *system_rtl* and *testbench* directories

Search for *MFP_DEMO_PIPE_BYPASS* symbol in *system_rtl* and *testbench* directories. Review the code fragments where that symbol occurs.

Modify the configuration parameters in the file *system_rtl/mfp_ahb_lite_matrix_config.vh* as follows:

```
//
//  Configuration parameters
//

// `define MFP_USE_WORD_MEMORY
// `define MFP_INITIALIZE_MEMORY_FROM_TXT_FILE
   `define MFP_USE_SLOW_CLOCK_AND_CLOCK_MUX
   `define MFP_USE_UART_PROGRAM_LOADER
// `define MFP_DEMO_LIGHT_SENSOR
// `define MFP_DEMO_CACHE_MISSES
   `define MFP_DEMO_PIPE_BYPASS
```

Review the following fragments of *system_rtl/mfp_system.v*:

```
. . . . . .

`ifdef MFP_DEMO_PIPE_BYPASS
    wire         mpc_aselres_e;
    wire         mpc_aselwr_e;
    wire         mpc_bselall_e;
    wire         mpc_bselres_e;
`endif

    m14k_top m14k_top
    (
```

```
        . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

        .UDI_toudi              ( UDI_toudi             )

`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         )
`endif

    );

    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    `ifdef MFP_DEMO_CACHE_MISSES

    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    `elsif MFP_DEMO_PIPE_BYPASS

    assign IO_GreenLEDs = { { `MFP_N_GREEN_LEDS - 5 { 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
    };

    `endif
```

3.5. Connect the board to the computer

For *Digilent* boards, such as *Nexys4*, *Nexys4 DDR* or *Basys3*, this step is obvious. For *Altera/Terasic* boards some additional steps required:

1. Connect USB-to-UART connector to FPGA board. Either *FT232RL* or *PL2303TA* that you can by from AliExpress or Radio Shack will do the job. *TX* output from the connector (green wire on *PL2303TA*) should go to pin 3 from right bottom on Terasic DE0, DE0-CV, DE1, DE2-115 (right top on DE0-Nano) and *GND* output (black wire on *PL2303TA*) should be connected to pin 6 from right bottom on Terasic DE0, DE0-CV, DE1, DE2-115 (right top on DE0-Nano). Please consult photo picture in *Lab MO1* to avoid short-circuit or other connection problems.
2. For *FT232RL* connector: make sure to set 3.3V/5V jumper on *FT232RL* part to 3.3V.
3. For the boards that require external power in addition to the power that comes from USB, connect the power supply. The boards that require the extra power supply include *Terasic DE2-115*.
4. Connect FPGA board to the computer using main connection cable provided by the board manufacturers. Make sure to put USB cable to the right jack when ambiguity exists (such as in *Terasic DE2-115* board).
5. Make sure to power the FPGA board (turn on the power switch) before connecting the UART cable from USB-to-UART connector to the computer. Failing to do so may result in electric damage to the board.
6. Connect USB-to-UART connector to FPGA board.

3.6. Run the synthesis and configure the FPGA with the synthesized MIPSfpga system

This step is identical to the synthesis step in *Lab MO1*

3.7. Go to the lab directory and clean it up

Under Windows:

```
cd programs\06_pipeline_bypasses
00_clean_all.bat
```

Under Linux:

```
cd programs/06_pipeline_bypasses
./00_clean_all.sh
```

3.8. Review the lab program code

The main program is located in file *programs/06_pipeline_bypasses/main.c*. After reset and running the boot sequence the *main()* function calls one of several routines written in assembly language, depending on the position of three switches in the board: *sw[4]*, *sw[5]*, *sw[6]*:

File *programs/06_pipeline_bypasses/main.c*

```
#include "mfp_memory_mapped_registers.h"

int main ()
{
    MFP_7_SEGMENT_HEX = MFP_SWITCHES >> 4;

    switch (MFP_SWITCHES >> 4)
    {
        case 0: demo_bypass_a_from_alu_instruction        (); break;
        case 1: demo_bypass_b_from_alu_instruction        (); break;
        case 2: demo_bypass_a_and_b_from_alu_instruction  (); break;
        case 3: demo_bypass_a_from_load_instruction       (); break;
        case 4: demo_bypass_b_from_load_instruction       (); break;
        case 5: demo_bypass_a_and_b_from_load_instruction (); break;
    }

    return 0;
}
```

The assembly subroutines are located in *programs/06_pipeline_bypasses/demo_bypasses.S*. They written in a way to cause different types of pipeline forwarding in the simplest fashion, with one forwarded register and without cycle timing effects due to cache misses. For example, subroutine *demo_bypass_a_from_alu_instruction*:

File *programs/06_pipeline_bypasses/demo_bypasses.S*

```
        . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

        .text

        .global demo_bypass_a_from_alu_instruction

demo_bypass_a_from_alu_instruction:

        li      t0, 0x001
        li      t1, 0x010
1:
        addiu   t1, t0, 1
        addu    t2, t1, t0
        b       1b
```

```
nop
```

3.9. Prepare the first software run

Following the procedure described in *Lab MO1*, compile and link the program, generate Motorola S-Record file and upload this file into the memory of the synthesized MIPSfpga-based system on the board.

Under Windows:

1. cd programs\06_pipeline_bypasses
2. run 02_compile_and_link.bat
3. run 08_generate_motorola_s_record_file.bat
4. run 11_check_which_com_port_is_used.bat
5. edit 12_upload_to_the_board_using_uart.bat based on the result from the previous step - set the working port in "set a=" assignment.
6. Make sure the switches 0, 1, 4, 5, 6 on FPGA board are turned off. Switches 0 and 1 control the speed of the clock, while switches 4, 5, and 6 determines which pipeline bypass demo function is called after reset and boot sequence - see *3.8. Review the lab program code*. If the switches 0 and 1 are not off, the loading through UART is not going to work. For more details about the switchable clock see *MIPSfpga Lab MO2. Using switchable clock to observe the CPU cycle-by-cycle*.
7. run 12_upload_to_the_board_using_uart.bat

Under Linux:

If uploading program to the board first time during the current Linux session, add the current user to *dialout* Linux group. Enter the *root* password when prompted:

```
sudo adduser $USER dialout
su - $USER
```

After that:

1. cd programs/06_pipeline_bypasses
2. run ./02_compile_and_link.sh
3. run ./08_generate_motorola_s_record_file.sh
4. run ./11_check_which_com_port_is_used.sh
5. edit ./12_upload_to_the_board_using_uart.sh based on the result from the previous step - set the working port in "set a=" assignment
6. Make sure the switches 0, 1, 4, 5, 6 on FPGA board are turned off. Switches 0 and 1 control the speed of the clock, while switches 4, 5, and 6 determines which pipeline bypass demo function is called after reset and boot sequence - see *3.8. Review the lab program code*. If the switches 0 and 1 are not off, the loading through UART is not going to work. For more details about the switchable clock see *MIPSfpga Lab MO2. Using switchable clock to observe the CPU cycle-by-cycle*.
7. ./run 12_upload_to_the_board_using_uart.sh

3.10. The first run

1. Set the switches 4, 5 and 6 on FPGA board to off position. Such setting means that after the reset the program will run assembly demo number 0. Make sure the switches 0 and 1 are off, otherwise the boot sequence (a sequence of processor instructions before *main* function) will take too long, since these switches control the clock frequency.

2. Reset the processor. The reset buttons for each board are listed in the table below:

Board	Reset button
Digilent Basys3	Up

Digilent Nexys4	Dedicated CPU Reset
Digilent Nexys4 DDR	Dedicated CPU Reset
Terasic DE0	Button/Key 0
Terasic DE0-CV	Dedicated reset button
Terasic DE0-Nano	Button/Key 0
Terasic DE1	Button/Key 0
Terasic DE2-115	Button/Key 0
Terasic DE10-Lite	Button/Key 0

3. Turn the switch 0 on. This will switch the system clock from 25 MHz to 0.75 Hz, less than one beat per second. You should see LED 7 start blinking, it is connected straight to the system clock.
4. Check that the seven-segment display shows "0" which is consistent with switches 4, 5, 6 and the code of the program.

5. Now turn your attention to LEDs 0, 1, 2, 3. They are connected to the following bypass control signals:

LEDs (left to right)	**Bypass control signals**	
	Name	**Function (according to the comment in Verilog code)**
3	mpc_aselres_e	Bypasses result from M stage (res_m) as source A (rs) for the next instruction
2	mpc_bselres_e	Bypasses result from M stage (res_m) as source B (rt) for the next instruction
1	mpc_aselwr_e	Bypasses result from A stage (res_w) as source A (rs) for the next instruction
0	mpc_bselall_e	Bypasses result from A stage (res_w) as source B (rt) for the next instruction

6. You can see LED 3 blinking from time to time, indicating bypass of ALU result to the source A of the next instruction. This is consistent with demo 0 code:

   File *programs/06_pipeline_bypasses/demo_bypasses.S*

```
        . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
1:
        addiu   t1, t0, 1
        addu    t2, t1, t0
        b       1b
        nop
```

3.11. The subsequent runs

Set the switches 4, 5 and 6 on FPGA board to another position (001), set the switch 0 to off position, then reset the processor, turn switch 1 on and observe the patterns of blinking again.Repeat for all 6 bypass demos, with switches 4, 5 and 6 set to 000, 001, 010, 011, 100, 101. Does the expected behavior, based on reviewing the assembly code, always matches the pattern? If not, look to the logic that uses bypass control signals and try to explain what happens.