

Computer Organization and Design (HY219) Winter Semester 2021-2022

Laboratory 6

1. Implementation of the MIPS processor (10 credits)

a. Extending the MIPS micro-architectural pipeline (bypass, stall) (8 units)

The use of the micro-architectural pipeline (pipeline) is necessary in every modern processor, but it creates a number of problems in the proper operation of the system. In this workshop you will be given an initial implementation of the MIPS pipeline microarchitecture in Verilog and you will be expected to implement some additions that are intended to correct some of these problems. The pipeline micro-architecture can execute

commands that follow the R format as well as the load/store commands we've encountered from previous labs. This lab asks you to implement bypass and stall techniques in order for these commands to be executed correctly in every case. The theory for this workshop is covered up to Chapter 4.7 of the book.

Figure 1 shows the diagram of the architecture you should implement, by enriching the data part (*data path unit in cpu.v*) and the control part (*control unit in control.v*) of the micro-architecture you are given. We suggest the gradual implementation of the micro-architecture so that you facilitate the process of verifying the correct operation. At each step it would be good to check in detail correct circuit operation before proceeding to the next step. I suggest to

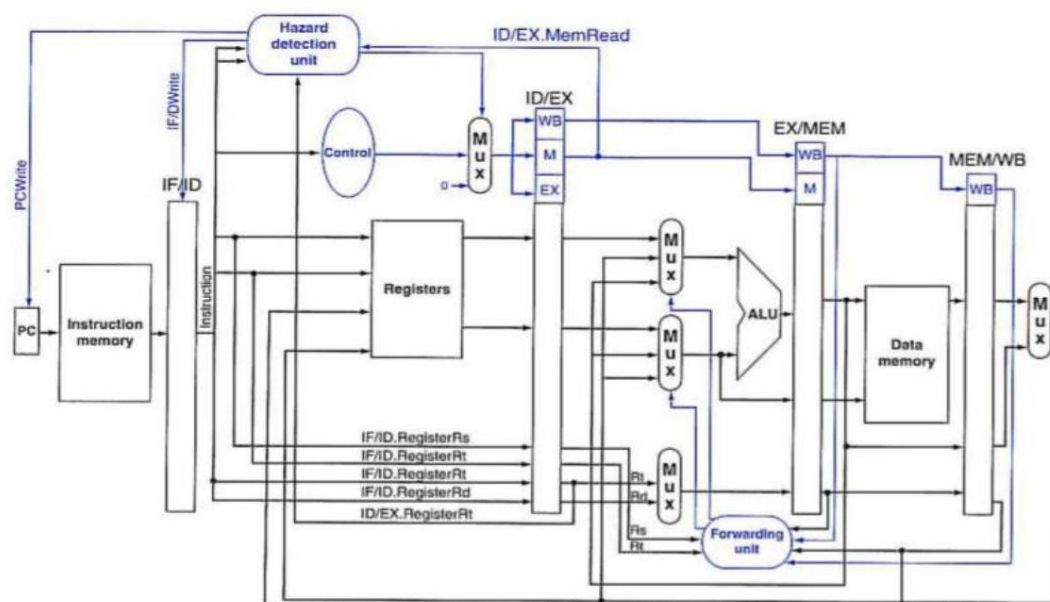



Figure 1. MIPS architecture with pipeline, risk detection unit and bypass unit. This architecture can execute format-R commands, as well as load/store commands. The picture is similar to picture 4.60 in the book.

follow the steps below.

- a. Study the code given to you to understand both its function and where you should place the *stall* and *bypass*. For a better understanding you should simulate the code we give you using Icarus. Use a sequence of commands that exercises as many parts of the pipeline as possible.
- b. Implement the forwarding unit (bypass). We remind you that this unit forwards results produced by the ALU (from a format R instruction) or read from memory (from an **lw instruction**) to immediately following instructions that need to read these results. You should be aware of edge cases such as those shown in the following command sequence:

```
or $a0, $a0, $t2
add $a0, $a0, $v0
slt $sp, $a0, $t1
```



The \$a0 register of the last command comes from the **add** command which at the time of promotion is in the **MEM** stage of the pipeline and not from the **or** command which is in the **WB stage**.

- c. There is a final problem that arises when a command accepts data from an immediately preceding load command. The commands after the load should be delayed for one machine cycle, while a bubble should also be created in the pipeline between the Load command and the immediately following command. So you should implement a risk detection module that creates this bubble and does everything else needed to run the program correctly.

For questions b. and c. you should write code in the *control.v* file to implement the bypass and stall detection logic. You should also write code in the *cpu.v* file to place the extra circuits in the data section.

b. Simulation to Verify Correct Operation

The suggested program for controlling the pipeline is given in the program.hex files while testbench.v contains the assembly code in mnemonic form. We assume that each register is initialized with the value **reg[i] = i**, and comment the results after each command is executed (in testbench.v). Your implementation should give the correct results at every step of the execution and PASS every command to get an A in the lab. Observe the output.txt file that the testbench generates to see if your code is working correctly. You should not change testbench.v in this lab.

c. Add Extra Command (3 credits)

The query asks you to extend the architecture of the first query with the **addi** command and the left-shift **sll** and **sllv** commands. Note that both the **sll** and **sllv** shift commands follow the R-format. The **sll** command requires the use of the *shamt* field (bits [10:6] of the command) to specify the shift size. Consult the opcodes of these instructions before you start decoding them in Verilog.

Place the following commands at the end of the previous series of commands to test the correct operation of your final circuit. Use the same testbench.v file as in the first query.

```
lw $v0, 8($t2) sll $s4,      # $v0 = $2 = 28 = 0x1C
$v0, 12 sllv $s6, $s4,      # $s4 = $20 = 0x0001c000
$sp addi $s6, $s6, -100     # $s6 = $22 = 0x00038000
                             # $s6 = $22 = 0x00037f9c
```