

Laboratory 10

10. Pipeline MIPS CPU Design (2): 16-bits version

10.1. Objectives

Study, design, implement and test

- **MIPS 16 CPU, pipeline version with the modified program without hazards**

Familiarize the students with

- Pipeline CPU design
- Xilinx® ISE WebPack
- Digilent Development Boards (**DDB**)
 - [Digilent Basys Board – Reference Manual](#)
 - [Digilent Basys 2 Board – Reference Manual](#)
 - [Digilent Basys 3 Board – Reference Manual](#)

10.2. Transforming the MIPS 16 Single-Cycle CPU to a Pipeline CPU

! You must attend/read lecture 8 in order to fully understand the Pipeline CPU

Remember that an instruction execution cycle (lecture 4) has the following phases:

- | | | |
|---------|---|------------------------------------|
| • IF | – | Instruction Fetch |
| • ID/OF | – | Instruction Decode / Operand Fetch |
| • EX | – | Execute |
| • MEM | – | Memory |
| • WB | – | Write Back |

The data-path of the single-cycle processor (32-bit version), including the control unit and the necessary control signals, is presented in the next figure. In order to reduce the complexity of the data-path the control signals were not explicitly connected, but rather they can be easily identified by their names.

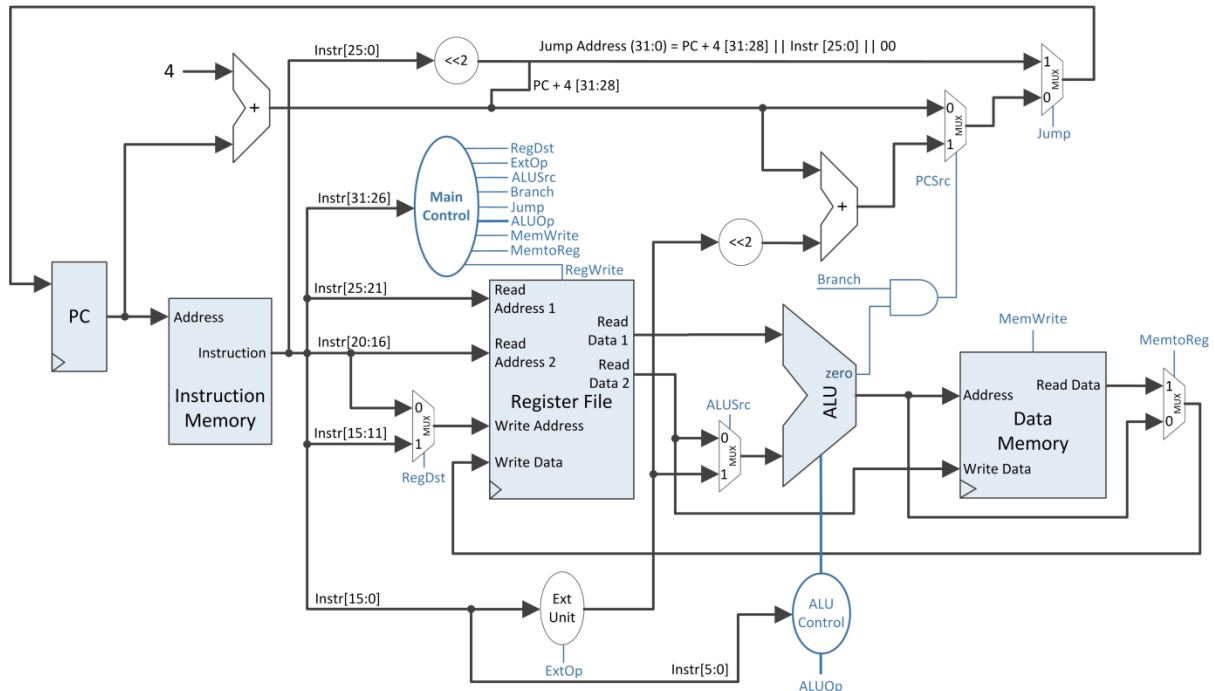


Figure 10-1: MIPS 32 Single-Cycle Data-Path + Control

The main issue with the single-cycle MIPS CPU is the length of the critical path, for the load word instruction (see lecture 04). The necessary time for transmitting the data along the critical path must be covered by the clock cycle time. This results in a long cycle time (slow clock).

In order to reduce the clock cycle time, the solution is to partition the data-path along the critical path with rising edge triggered registers (D flip-flops). These registers are inserted between the MIPS 32 functional units that coincide with the instruction execution phases: IF, ID, EX, MEM, WB. In this manner, one can simultaneously execute at most 5 instructions, each of them executing one of the five execution phases. The pipeline execution units are also referred to as **stages**.

The data-path together with the control unit for the pipelined MIPS 32 CPU is presented in Figure 10-2.

Each intermediate register will be referred depending on its position between the pipeline stages. The register between the IF stage and the ID stage is IF/ID, the one between ID and EX is ID/EX, etc.

The role of these intermediate registers is to hold the intermediate results of the instruction execution in order to provide these results to the next stage, in the next clock cycle.

Furthermore, the execution on the data-path depends on the control signals values, which are specific for each instruction. So, through the intermediate registers (starting with the ID/EX register) the control signals will also be provided for the next stages. The control signals are symbolically grouped after the stage name where they belong.

The control signals are transmitted together with the intermediate results until the stages where they are needed.

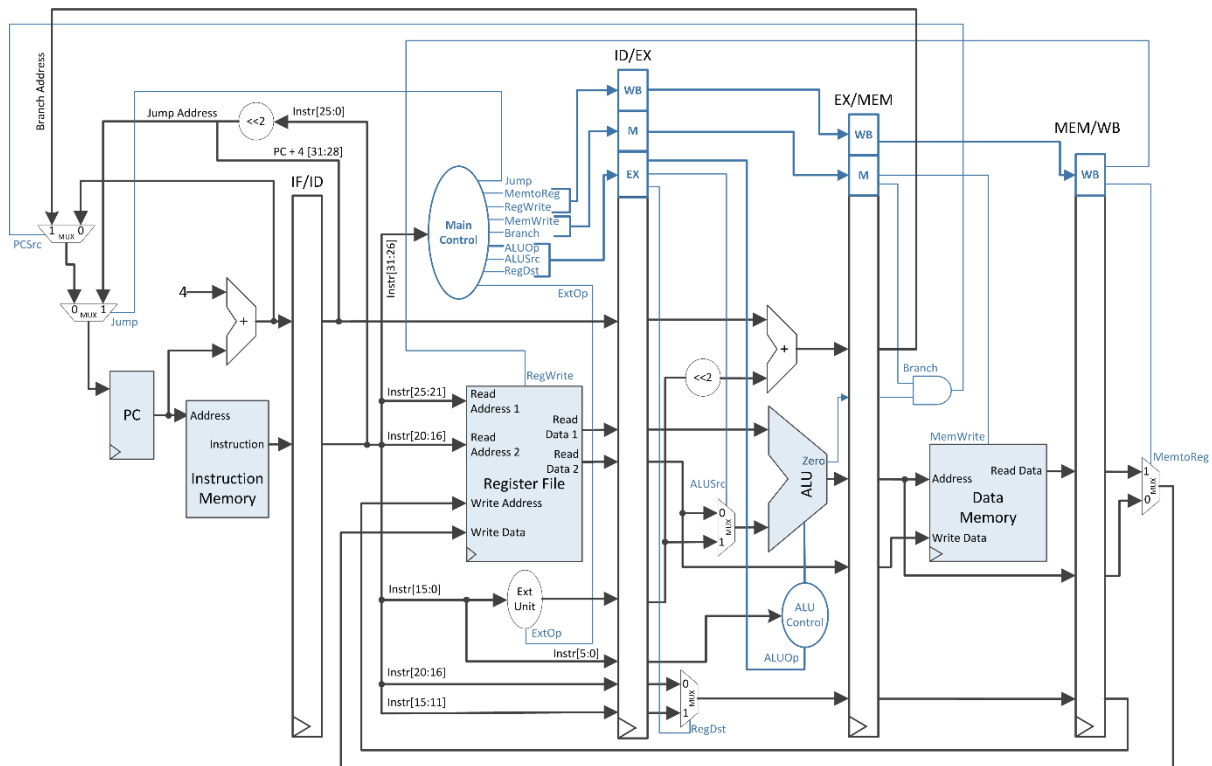


Figure 10-2: MIPS 32 Pipeline Data-Path + Control, obtained from the partitioning of the Single-Cycle Data-Path

Lecture 8 explains in more detail the design of the pipeline CPU; the details presented so far represent the necessary knowledge for transforming your own MIPS 16 single-cycle CPU into a pipeline one.

One notable difference between the two data-paths is that the multiplexer used for selecting the write address for the Register File is placed in EX, not in ID as in the single-cycle CPU case. There are 2 possible solutions:

- c) Leave it in the ID stage. In this case, the **RegDst** signal will not be transmitted through ID/EX and will be connected directly from the control unit.
- d) Move it to the EX stage, modifying the input / output ports of the ID and EX units, and transmit the **RegDst** signal according to the presented pipeline data-path.

Observation: The **MemRead** signal will be ignored, as in the single-cycle case.

10.3. Hazards in MIPS

Hazards are situations in which an instruction cannot be executed in the next clock period. The hazard can be classified as:

1. **Structural Hazards (resource dependency)**
 - 2 instructions try to use the same resource simultaneously for different purposes → resource constraints
2. **Data Hazards (data dependency)**
 - Attempt to use data before it is ready (available)
 - For an instruction in the ID phase, the operands might still be processed in other pipeline stages
3. **Control Hazards (condition and control dependency)**
 - The branch decision and branch target address are not known until the MEM stage. The jump address is computed in the ID stage.
 - Pipelining of jumps, branches and other instructions that modify the sequential flow of the program

These hazards have been thoroughly presented during lecture 8 (you are encouraged to read them!). Optimal solutions (in hardware) are relying on forwarding and stalling the pipeline (see the lecture notes for reference). For your MIPS 16 pipeline implementation, you should implement the software solution, modifying your program such that the data and control hazards are avoided.

The basic change in your program should be the following: introduce NoOp (No Operation) instructions between the instructions where the hazard exists.


NoOp instruction should not change anything in your processor (ex. sll \$0, \$0, 0; add \$0, \$0, \$0, etc.)

10.3.1. Structural Hazards

Structural hazards occur when instructions from two different pipeline stages are trying to use the same resource in the same cycle.

Special attention should be given to the structural hazard that can occur when two instructions at distance of 3 are using the same register (from the RF). In the following example, we presume that instr1 and instr2 do not have hazard with other instructions.

Structural hazard at RF	
add \$1, \$1, \$2	
instr1	
instr2	
add \$3, \$1, \$4	



Pipeline diagram (in each clock cycle we present the pipeline stage for each instruction):

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	...
add \$1, \$1, \$2	IF	ID	EX	MEM	WB				
instr1		IF	ID	EX	MEM	WB			
instr2			IF	ID	EX	MEM	WB		
add \$3, \$1, \$4				IF	ID	EX	MEM	WB	

During clock cycle CC5, the new value of \$1, generated by the first instruction, is in WB stage, being unwritten yet in RF. Therefore, in ID stage, the 4th instruction will read the old value of \$1, in cycle CC6 EX receiving the incorrect value.

There are 2 possible solutions:

1. Recommended: Modify RF block such that the writing is done in the middle of the clock cycle (test the falling edge – $\text{clk} = 0$ & $\text{clk}'\text{event}$). In this case, the RF read (being asynchronous), in the second part of CC5 the correct value of \$1 occurs and it is propagated forward to EX at CC5-CC6 transition.
2. Introduce a NoOp instruction

Without Hazard
add \$1, \$1, \$2
instr1
instr2
NoOp
add \$3, \$1, \$4

Attention! In the following, it is assumed that you have chosen the 1st option. Otherwise, introduce an extra NoOp where necessary.

10.3.2. Data Hazards

Data hazards (Read After Write or Load Data Hazard) occur when the current instruction use as source(s) the register that will be written by other instructions that are still executing in the pipeline.

(!) In order to establish where these hazards occur you need to draw the pipeline diagram and to understand how pipelining is done (when operands are read).

The following example contains most of the data hazards that might occur in your pipelined MIPS.

Instr. Nb.	Program
1	add \$1, \$2, \$3
2	add \$3, \$1, \$2
3	add \$4, \$1, \$2
4	add \$5, \$3, \$2
5	lw \$3, 5(\$5)
6	add \$4, \$5, \$3
7	sw \$3, 6(\$5)
8	beq \$3, \$4, -6

Hazard identification process is solved with NoOp insertions, starting from first instruction to the last one. Example:

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
add \$1, \$2, \$3	IF	ID	EX	MEM	WB(\$1)				
add \$3, \$1, \$2		IF	ID(\$1)	EX	MEM	WB(\$3)			
add \$4, \$1, \$2			IF	ID(\$1)	EX	MEM	WB		
add \$5, \$3, \$2				IF	ID(\$3)	EX	MEM	WB(\$5)	
lw \$3, 5(\$5)					IF	ID(\$5)	EX	MEM	WB(\$3)

Instr\Clk	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	...
add \$5, \$3, \$2	IF	ID(\$3)	EX	MEM	WB(\$5)				
lw \$3, 5(\$5)		IF	ID(\$5)	EX	MEM	WB(\$3)			
add \$4, \$5, \$3			IF	ID(\$3, \$5)	EX	MEM	WB(\$4)		
sw \$3, 6(\$5)				IF	ID(\$3)	EX	MEM	WB	
beq \$3, \$4, -6					IF	ID(\$4)	EX	MEM	WB

Hazards are solved iteratively, starting from the first occurrence. Solving a hazard between 2 successive instructions implicitly solves the hazards between the first instruction and the instruction at distance +2. Example: between instruction 1 and 2, and 1 and 3, there is a RAW hazard (after \$1). Hazard between 1 and 2 is solved first, delaying instruction 2 with 2 cycles (it should have ID on cycle CC5) => 2 NoOp. Therefore, all following instructions will be delayed with 2 cycles, so the hazard between 1 and 3 is also resolved.

There is a RAW hazard between the 2nd and 4th instructions, after \$3, which can be solved by delaying with 1 cycle, inserting a NoOp after 2 or before 4.

All other hazards are being solved, resulting the following program:

Instr. Nb.	Program
1	add \$1, \$2, \$3
2	NoOp
3	NoOp
4	add \$3, \$1, \$2
5	NoOp
6	add \$4, \$1, \$2
7	add \$5, \$3, \$2
8	NoOp
9	NoOp
10	lw \$3, 5(\$5)
11	NoOp
12	NoOp
13	add \$4, \$5, \$3
14	NoOp
15	sw \$3, 6(\$5)
16	beq \$3, \$4, -6

10.3.3. Control Hazards

Control hazards occur at instructions that alter the sequential flow of the program, when the next sequential instructions that follow (3 for BEQ and 1 for J) are implicitly executed.

For conditional jump instructions (beq, bne, etc.), the next 3 instructions will implicitly be executed, being already in the pipeline. Therefore, a simple (but not efficient) solution is to insert 3 NoOp's.

For unconditional jumps (j, jal, etc.), based on the data-path from Figure 5-5, these instructions are computing the jump address (and writing it in the PC register) in the ID stage. It means that only next instruction starts the execution, so one NoOp needs to be inserted after the j instruction. A better solution would be to insert the instruction that is after j before it, with condition that this instruction is not also a jump instruction (j, beq, etc.)

10.4. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- Xilinx project with "test_env" including the complete and correct implementation of the pipeline MIPS 16 CPU.

10.4.1. Verify the MIPS 16 Pipeline CPU design

You can evaluate the critical path by checking the clock frequency: Go to **Processes** → **Design Summary/Reports**. Open **Detailed Reports** → **Synthesis Report** and watch the section related to clock signal.

You should notice an increase in frequency (of 30-50%) caused by the pipelining of your MIPS (compared to the one observed in laboratory 9). One can observe that the increase in speed is not proportional to the number of pipeline stages. There is a multitude of reasons for that: stages are not balanced, the resulting circuit depends on the board's technology, the memories are implemented as distributed RAMs, etc.

10.4.2. Program analysis and hazard removal (paper and pencil)

Based on the example in section 3, identify the hazards in your program. Insert NoOp instructions where such an instruction is needed. Draw the pipeline diagram for at least 5 successive instructions in your program (for all, if there are not any hazards).

Note: By introducing the NoOp's you will need to adjust the addresses for the jump instructions in your program.

Modify the (assembly) program in the instruction memory

10.4.3. Test and evaluate the MIPS 16 pipeline

Test your design on the FPGA board. You have 2 options:

- a. If your pipeline implementation was correct, without any mapping mistakes etc., then watching your final results is enough (results should be identical with your single cycle implementation).
- b. If the results are different, then you should trace your program step-by-step.

Use the same display procedure as the one used for your single-cycle MIPS (with the multiplexor on switches for selecting different data to be displayed on the SSD). It is important to understand that now your outputs (for your switches configuration) will not be the same as in the single-cycle implementation. You have 5 instructions in the pipeline; some of them will be NoOp.

You can display the control signals on the LEDs. Use the delayed control signals, i.e. the control signals delayed to the stage where they are used.

If necessary, display other signals/change the displayed signals, from different stages, on the SSD.

10.4.4. Hardware optimizations for the MIPS Pipeline CPU (optional).

If you have finished and tested your MIPS pipeline CPU, you can modify your solution in order to implement the following components of the complete pipeline processor:

- a. Hazard detection unit
- b. Forwarding unit
- c. Move the branch in the ID stage
- d. Hardware stalls for the LW (RAW hazard), BEQ and J instructions.

In the end, you should have a complete pipeline implementation, as it is presented in the lecture material.

10.5. References

- [1] Computer Architecture Lectures 3, 4 & 8 slides.
- [2] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 5th edition, ed. Morgan–Kaufmann, 2013.
- [3] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: A Quantitative Approach", 5th edition, ed. Morgan-Kaufmann, 2011.
- [4] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
- [5] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
- [6] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
 - Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.