Faculty of Engineering CEE216/CEE210

Electrical Engineering Dep year Communication

Instructor// Dr: Gihan Naguib Spring 2020

TA: Muhammad Dakheel

**Simulation and Designing 32-bit Mips processor**

Group members: -

1.عبدالله محمد أمين صادق (2820)

2.علاء محمد صديق محمد (2821)

3.جاسر جمال أحمد أحمد (2810)

Group number **(9)**

**Introduction**

It’s so clear to any engineer studies computer architecture that there is some mane steps in designing a processor specially single cycle processor and we’ll conclude them roughly as following below.

**The steps of design a single cycle processor: -**

1) There are 5 main components -functions- in single cycle processor

1. PC 2. Register file 3. ALU 4. Memory which is helpful to -be separated into Instruction\_Mem and Data\_Mem 5. Control unit 6. Pipeline Registers and Forwarding (علاء محمد + عبدالله محمد + جاسر جمال)

2) ALU implementation which is made to execute all the required operations given in the project (عبدالله محمد + جاسر جمال).

3) Register file implementation (جاسر جمال)

4) ALU Data Path. (عبدالله محمد ).

5)Data path j-type instructions (علاء محمد).

6)Connections of the circuit (علاء محمد + عبدالله محمد + جاسر جمال ).

7) Control signals table (علاء محمد + جاسر جمال ).

8) Control unit (عبدالله محمد + علاء محمد )

9) converting test code to machine language then entering instructions to memory (عبدالله محمد+ جاسر جمال ).

10) Repairing problems and errors in the design (عبدالله محمد + علاء محمد) .

11) Following the Execution of the program and evaluating the final values . (عبد الله محمد + علاء محمد + جاسر جمال).

(12Making some modifications to build pipeline stages

(عبد الله محمد + علاء محمد+ جاسر جمال)

13) Making the documentation and the report (جاسر جمال)

Phase 2 Pipelining

**Introduction: -**

In the previous part we built a single cycle processor, we were able to get it to do the required instructions, really, everything that we needed it to do and it works then we verify it by using a test code, so we could write any program that we like using this processor. Here we want to make our processor run faster than it already does, so we use an implementation technique called-five stage-pipelining to improve the performance of this processor. At the end of this part we’ll have built this modified processor.

**The basic idea behind pipelining: -**

We’re going to break our processor up into a series of smaller stages, now each instruction will only need to occupy one stage of our processor at a time, which will free up the rest of our processor to run other instructions ,this means we’ll actually be able to have five instructions running concurrently in our five stages pipeline-modified-processor.

**The effect of using pipelining technique: -**

We get a higher clock and greater performance (much speed up), but as we’ll see through the topic, we’ll have some conditions such as data dependency that reduce the speed of our processor as well as our processor is more complicated, there are a lot of moving pieces-muxes specially-here, new 4 pipeline registers will be add and new “Hazards “logical unit. In addition, things are going to work a little differently than they did before.

**Clock-cycle time and latency: -**

In single cycle it basically is the time of the longest instruction (lw) and we must wait until it finished, otherwise in 5-stage pipelining it basically is the time of the longest stage (Mem) and we can have five instructions running concurrently in our five stages.

Every instruction in pipelining can be implemented in at most 5 clock cycles.

**Building a data-path: -**

(Execution) the 5 clocks cycles are as follows:

1. Instruction fetch cycle (IF):

Send the program counter (PC) to memory and fetch the current instruction from memory then Update the PC to the next sequential PC (next instruction) by adding 2 (since each instruction is 2 bytes) to the PC.

1. Instruction decode/register fetch cycle (ID):

Decode the instruction and read the registers corresponding to register source (rs, rt) from the register file. then we can use a comparator or Xor gate to test the equality on the registers as they are read, for a possible branch then we add a Sign-extender to extend the offset field of the[I-type, B-type and J-type] instructions as we need it in these instructions to (for ex\_ accessing the memory and changing the value of program counter (PC) such as in branch instruction, we Compute the target address by adding the sign-extended offset to PC+2)

1. Execution/effective address cycle (EX):

The ALU operates on the operands prepared in the previse stage, performing one of three functions depending on the instruction type.

* + - 1. Memory reference: The ALU adds the base register with the sign-extended offset to form the effective address.
      2. Register-Register ALU instruction: The ALU performs the operation specified by the ALU opcode on the data that is read from the register file.
      3. Register-Immediate ALU instruction: The ALU performs the operation specified by the ALU opcode on the first value (data) read from the register file with the sign-extended immediate.

1. Memory access (MEM):

If the instruction is (lw), memory reads using the effective address calculated in execution stage. If it is (sw), memory writes the data from the second register (rt) that is read from the register file.

1. Write-back cycle (WB):

Register-Register ALU instruction [R-type] and some of [I-type] instructions such as (lw) instruction: need to Write the result-whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction)-into the register file.

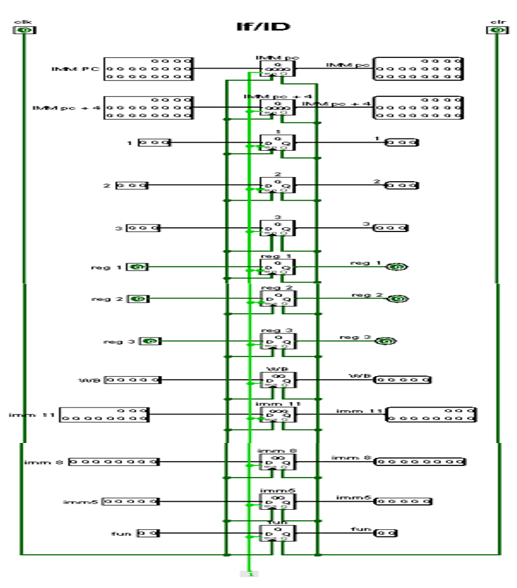
After we Know the (execution)of the 5 stages pipelining, we find that we must ensure that the overlapping among instructions-in the pipeline-can’t cause such a conflict, there are some observations depend on this fact as follow:

* + - 1. The register file is used in the two stages: In (ID) stage for reading and in (WB) stage for writing, we perform the register write in the first half of the clock cycle and the read in the second half.
      2. To start a new instruction every clock, we must increment and store the PC every clock, and its control signals to handle it in related stages.

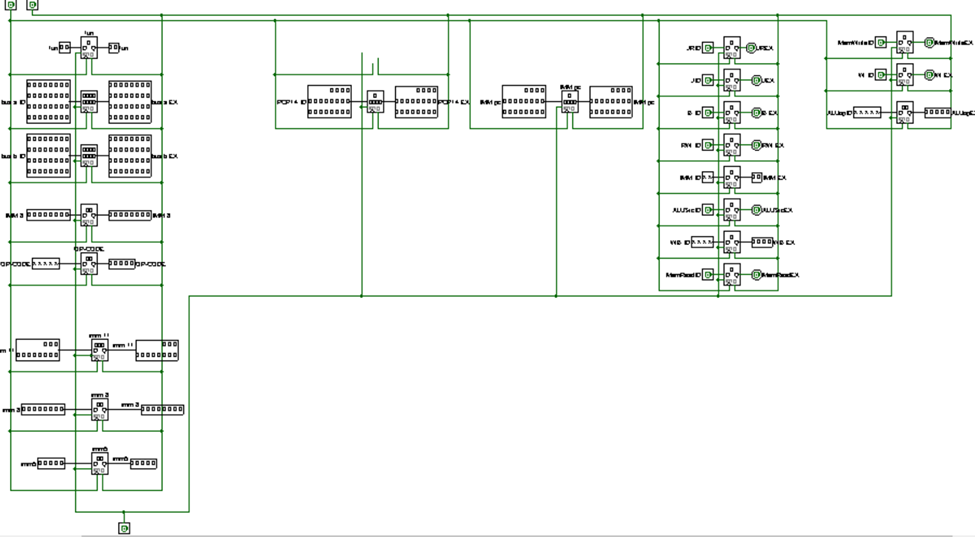
we need to solve these problems, so we built 4 pipeline registers which separate between successive stages of the pipeline so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle.

**Registers between stages:**

1. (IF/ID): This Register is after (IF) stage.

After the instruction (being read from memory using the address in the PC)-fetched-we place it into the IF/ID pipeline register. The next PC address that is written back into the PC is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as branch instructions then we pass needed information to the (ID) stage through the next pipeline register.

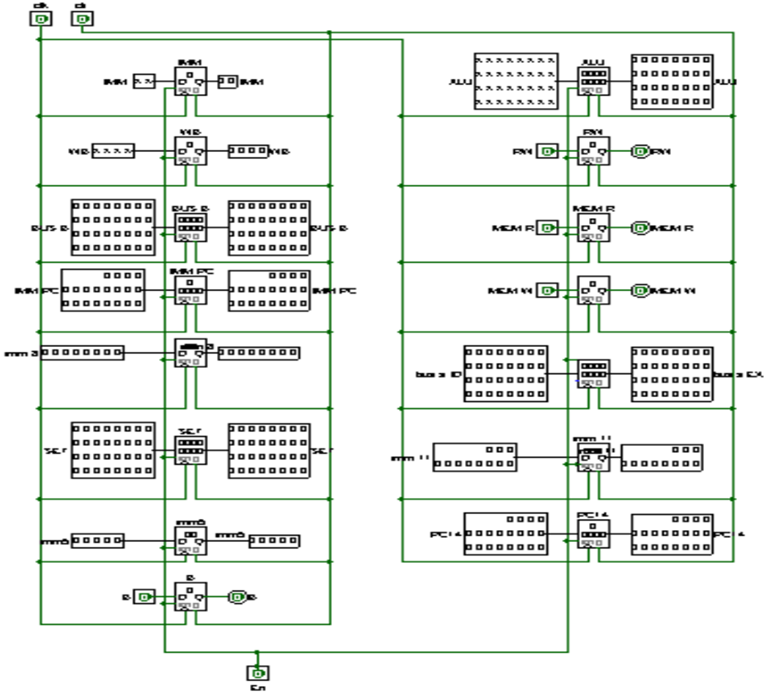
1. (ID/EX): This Register is after (ID) stage:

The instruction portion of the IF/ID pipeline register supplying the 5-bit immediate for I-type instructions and 8-bit immediate for B-type instructions, which are sign-extended to 32-bits as well as the register numbers to read the two registers (rs, rt). All three values are stored in the ID/EX pipeline register, along with the PC address and control signals, we again transfer any information that needed to use in the (EX) stage through the next pipeline register.

1. (EX/MEM): This Register is after EX stage:

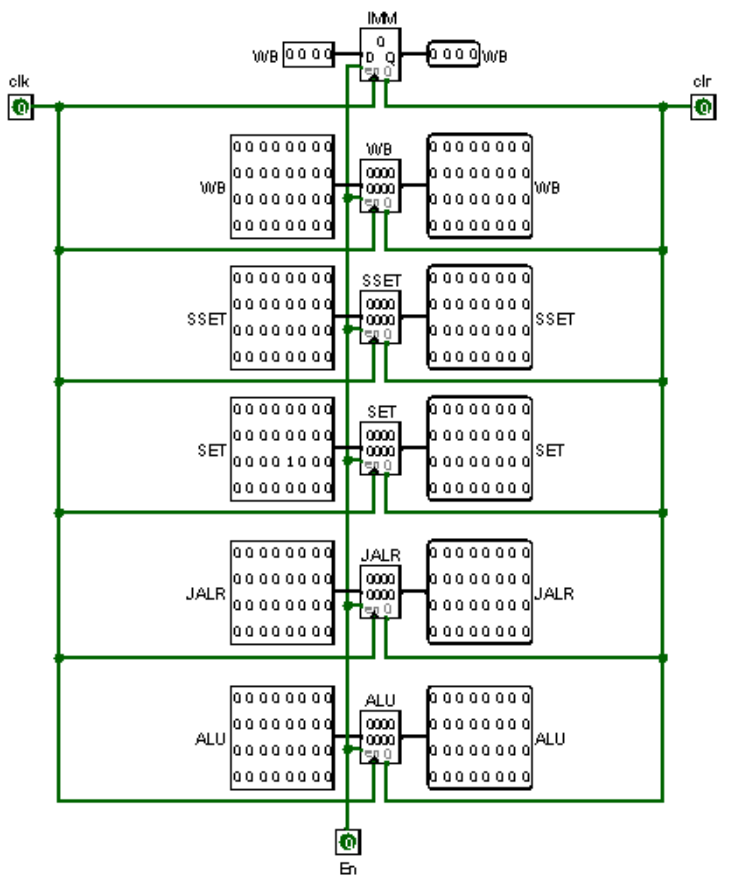
Instruction execute: the instruction reads the contents of (rs,rt)-Reg1 and Reg2- and the sign-extended immediate from the (ID/EX) pipeline register to perform the required operation using the ALU which is placed in the (EX/MEM) pipeline register with the other data needed to be saved such as:

1. PC.
2. Remaining control signal that are needed in the later stages.
3. ALU Result.
4. Rd.
5. Data to be stored in case if the instruction is (SW).
6. Immediate jump target.
7. Address register target.



1. (MEM/WB): This Register is after Mem stage.

Memory access: the instruction reads/stores the data memory using the address from the EX/MEM pipeline register and loads the data into the MEM/WB pipeline register, So the data needed to be saved:

1. PC.
2. The remaining control signals for the later stages -for memory and mem to Reg mux-.
3. ALU output.
4. Memory output.
5. Rd.

**Hazards: -**

When we run 5 instructions through pipeline concurrently, however is not without its risk, so we’ll look at some of the hazards that can arise in our pipeline processor and how we can detect them.

Types of hazards: -

1. Structural (which doesn’t occur in Mips, as) it happens when we use the same memory for storing instructions and data.
2. Data.
3. Control.

Data hazard: -

It occurs when two instructions try to access the same piece of data either in a register or block of memory and there are 3 different type of data hazard as follows:

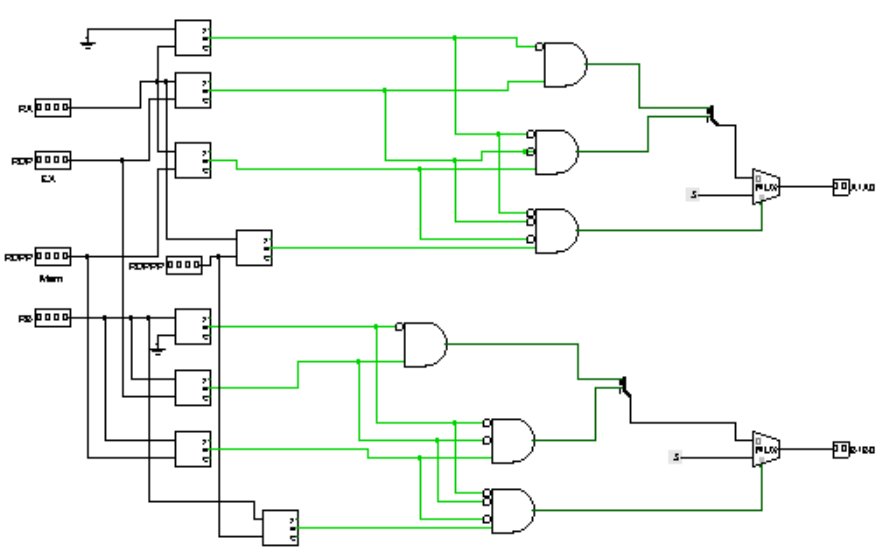
1. Read after write (RAW).
2. Write after read (WAR).
3. Write after Write (WAW).

But it is our good fortune that only (RAW) occurs in Mips, so when we have that :add $r0,$r1,$r2 (wants to write in $r0) then :add $r4,$r0,$r3(wants to read from $r0) # we have (RAW) data hazard.

To solve this issue there are two way: -

1. We can simply don’t do anything (NOP) until the instruction is completed-the value of $r0 updates-, but this way isn’t efficient as it ‘ll reduce the speed and it’ll cost us a long time waiting for the next instruction to be fetched -reading the updated value of $r0-.
2. Appling Forwarding.

Forwarding: -

When we deal with arithmetic instructions, we know that they compute their results in (EX) stage -the 3rdone- then spend the next two stages waiting to write back to register file, so instead of waiting two extra cycles to write those results back, we can just take them from ALU and pass them to it again-something else that wants them-.

Detecting the need to forward: -

• If (EX\_MEM.RegisterRd = ID\_EX.RegisterRs))

- ForwardA =01.

- (Forward from EX\_MEM pipe stage).

• If (MEM\_WB.RegisterRd = ID\_EX.RegisterRs))

- ForwardA = 10.

- (Forward from MEM\_WB pipe stage)

• If (EX\_MEM.RegisterRd = ID\_EX.RegisterRt))

- ForwardB = 01.

- (Forward from EX\_MEM pipe stage)

If (MEM\_WB.RegisterRd = ID\_EX.RegisterRt))

- ForwardB = 10.

- (Forward from MEM\_WB pipe stage).

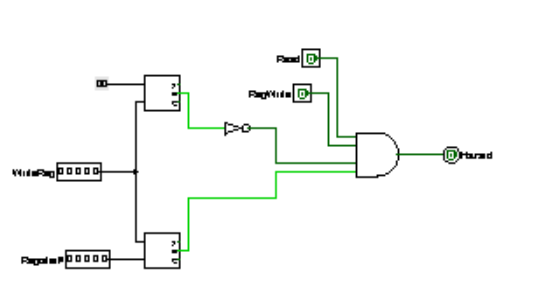
EX/MEM.RegisterRd ≠ 0

MEM/WB.RegisterRd ≠ 0

Control hazards: -

Actually, it isn’t different from data hazard. In this case, we’ve branch instruction that may decide to change the expected value of our (PC), but it does it relatively late in the pipeline 3 or 4 stage which seems to us as (RAW), so we’ll do (NOP) and forwarding to solve it.

Hazard detected: -



Now, we can see our modified-pipeline- data-path:

