EECE 421-Computer Architecture
Course Project
Extending Single Cycle RV32I Processor Core

Group Members:

Hadil Abdel Samad (hma142@mail.aub.edu)

Andrew Bejjani (aab68@mail.aub.edu)

Presented to:

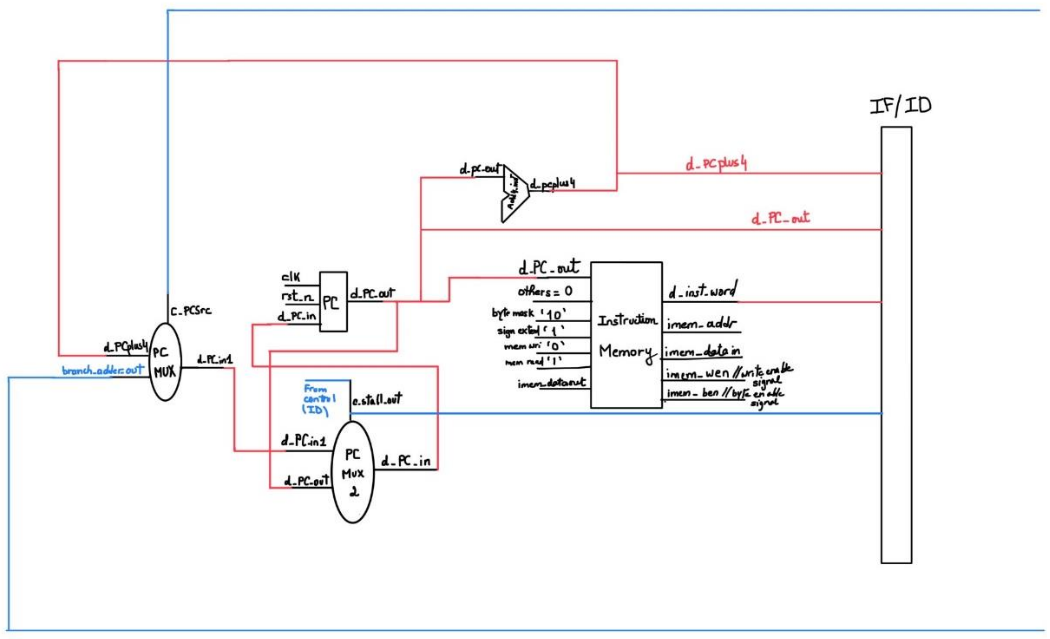
Dr. Mazen Saghir

Table of Contents

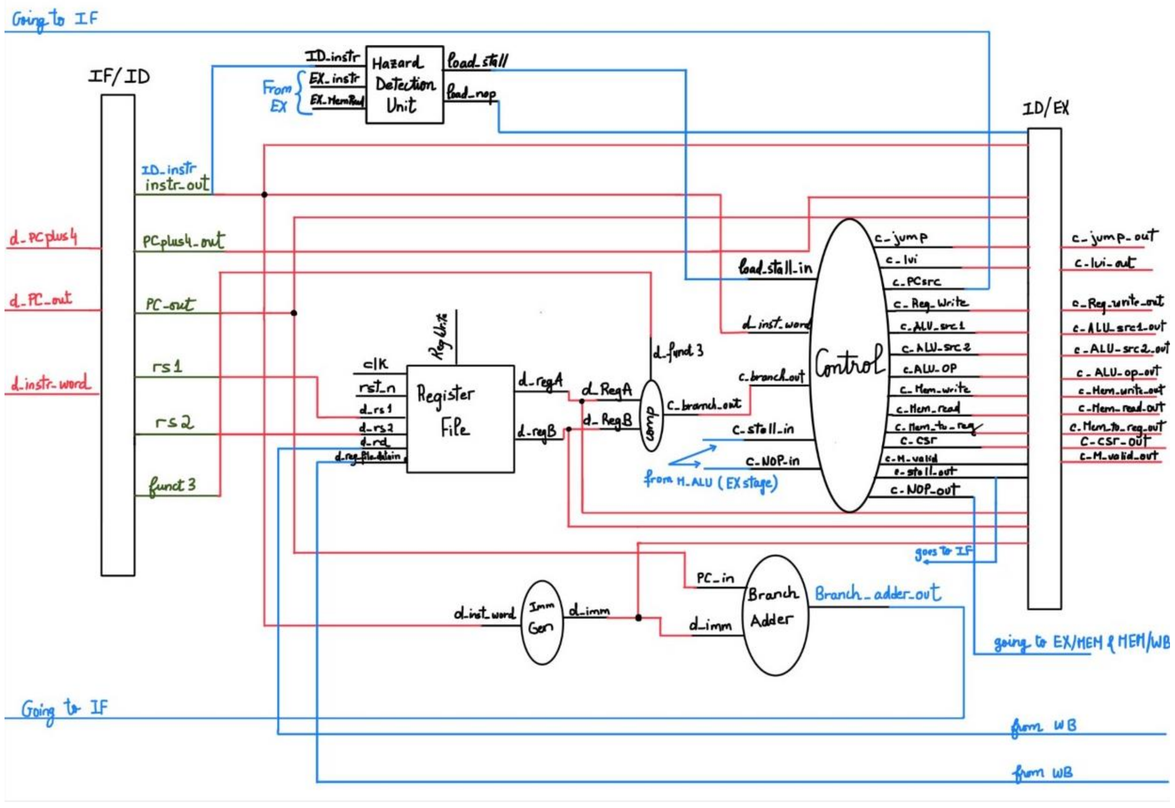
Pipelining the Datapath

I. Datapath Diagram

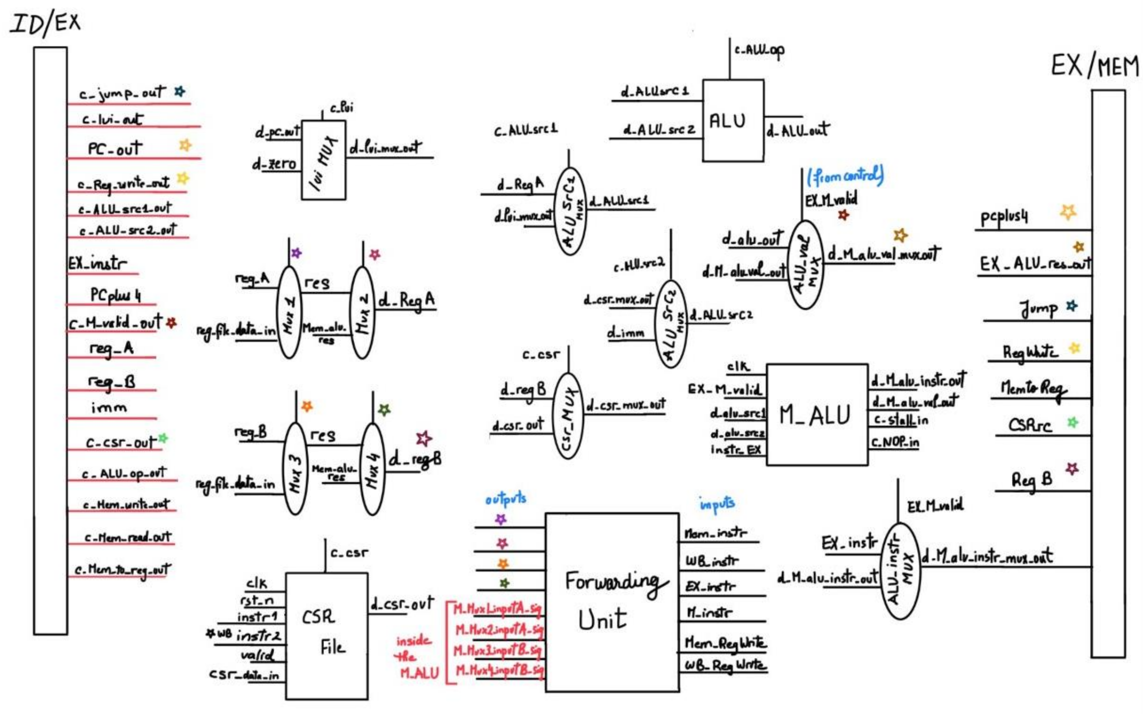
- IF-Stage



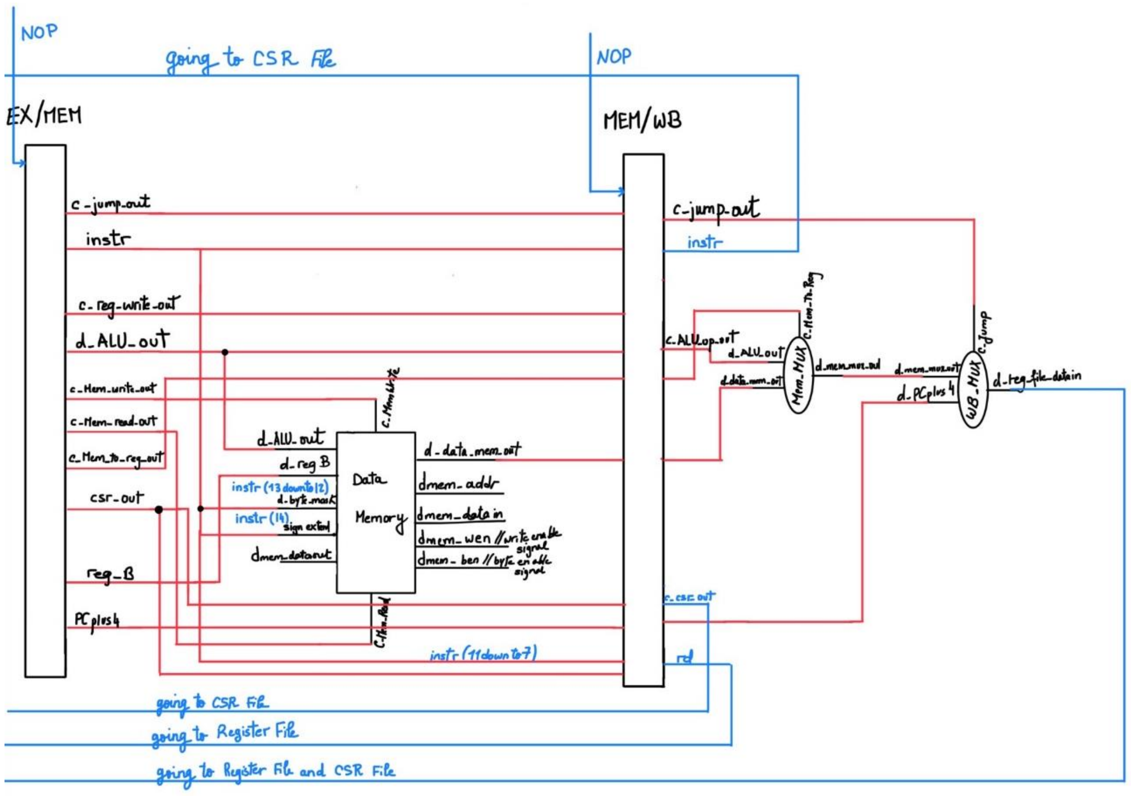
- ID-Stage



- EX-Stage



- MEM-Stage & WB-Stage



II. Components Added

- Branch_Adder: Added in the ID stage to calculate the address of the branch instruction, it has 2 inputs: the PC value and the output of the immediate generator.

- IF/ID Bus: It takes as inputs the PC + 4 value, the output of the PC, and the instruction word (output of instruction memory). It has as output, these three values, in addition to rs1, rs2 and function 3 needed in the ID stage.

- ID/EX Bus: It takes as inputs all the outputs of the control block (except for PCSrc signal which goes to an MUX in the IF stage), the instruction word, the PC value along with the PC + 4 value, the two outputs of the register file (values of registers A and B) and the output of the immediate generator. It takes as outputs the same values.

- EX/MEM Bus: It takes as inputs all the outputs of ID/EX bus except the value of registers A, the immediate value, the lui control signal and the 2 MUXs ALU SRC1 and SRC2's control signal, they will be used in the execution, no need for their propagation. It also takes as inputs the output of the ALU. It takes as outputs the same values.

- MEM/WB Bus: It takes inputs the control signals related to the jump, writing to the register the content of memory (c_mem_to_reg), the csr signal, c_reg_write_out and the output of the ALU. Also, it takes PC + 4 needed for the write back MUX and the propagated instruction which will be needed for the register file and the csr file.

- M_Alu: A separate block that performs multiplication operations. It takes as inputs the control signal EX_m_valid which tells if the instruction is a valid M_extension instruction and can differentiate between multiply and divide, d_alu_sr1 and d_alu_src2 as the typical ALU inputs, instr_EX which is the output of the ID/EX bus, and the clock. It takes as outputs d_M_alu_instr_out which is the instruction, d_M_alu_val_out which is the result of the multiplication, c_stall_in and c_nop_in used as inputs to the control unit in the ID stage to stall the ID and IF stages when we have a multiply instruction and to insert NOPs to later stages.

- MAlu_bus: A bus used inside the multiplication ALU needed to emulate a latency of 5 clock cycles for M-extension instructions.

- ALU_instr_mux: A newly added MUX used to select between the EX_instr and d_M_alu_instr_out based on the EX_M_valid control signal which tells if an instruction is a valid multiply instruction.

- ALU_val_mux: A newly added MUX to select between the output of the original ALU and the output of the M_ALU also based on the EX_M_valid control signal.

- CSR file: Unit for counting the number of instructions and number of clock cycles. CSR_cycles holds the least significant 32 bits of the number of clock cycles which are stored in CSR_mem[0]. CSR_cycles_h holds the upper 32 bits which are stored in CSR_mem[1]. CSR_instret holds the least significant 32 bits of the number of instructions executed in the datapath and are stored in CSR_mem[2]. CSR_instret_h holds the upper 32 bits which are stored in CSR_mem[3]. This unit also supports instructions CSRRS and CSRRC.

- CSR_MUX: A newly added MUX to select between the output of the csr file and inputB of the regfile. If we have a csr instruction, the output of the csr file will be chosen since it will need to enter the

ALU. If we don't have a csr instruction, inputB will be chosen and the instruction will proceed normally.

III. Modifications to the Control Unit

<u>We added as output:</u>

- c_m_valid: Signal needed as input to the m_alu and as a select input to the 2 added MUXs: alu_instr_mux and alu_val_mux used in the EX-stage, it tells whether the instruction is a valid multiply instruction or not.
- c_stall_out: Signal needed to stall the pipeline when we have a multiply instruction. It will be used to stall the IF/ID bus and the ID/EX bus and to prevent the PC from incrementing when stalling. For this reason, it will also serve as an input to the newly added MUX which checks if we are stalling and feeds the PC back its un-incremented PC.
- c_nop_out: Signal needed for the EX/MEM bus, to insert NOPs when we have M-extension instructions since they have a latency of 5 clock cycles.

<u>We added as input:</u>

- c_stall_in: This comes from the output of the m_alu unit from the EX-stage.
- c_nop_in: This is also coming from one output of the m_alu unit from the EX-stage.

Both of these instructions should cause the c_stall_out and c_nop_out to be set to 1 by checking needed conditions, for example by checking if the instructions in the ID stage and EX stage are both multiplication instructions then we should not stall since we should pipeline both of them. However, if we have a multiply instruction in the EX-stage and a divide instruction in the ID stage, then we should stall since we can't pipeline a multiply with a divide and vice versa.

IV. CSRfile

- We initialized an entity called "csr" of 7 inputs (clk, rst_n, csr_sig, csr_sig_wb, instr_first, instr_second, csr_data_in) and 1 output (csr_out). Csr_sig and csr_sig_wb become '1' when we have a csr instruction, instr_first is our 32-bit instruction in the EX stage and instr_second is our instruction in the WB stage, and csr_data_in is the value that will be written back to the CSR file after going through the ALU. Finally, csr_out is the value of our csr that is mapped to the 12 bits address, and that will pass through the ALU, where operations happen such as clear or set, and will be written to rd.

- We need to keep track of the instruction coming back from the WB stage to be able to update the value of the csr accordingly. Therefore, instr_second and csr_sig_wb will be needed to know which csr value we should change if we had to.

- Reading should be in the EX-stage and writing should be in the WB-stage. Instr_second will be used for writing and instr_first will be used for reading.

- We added a signal called csr_sig_wb which indicates that the instruction from the WB stage is a CSR instruction so that we can know whether we should update the value stored in any CSR instruction.

- We also modified the way we are keeping track of how many instructions have passed by checking whether the instruction coming from the WB stage is a valid instruction or not. SO basically, at the end of an instruction we are updating the count, and this is done by validating that the instruction is neither a NOP (X"00000013" or X"00000000") nor undefined (for this we checked if it has at least 1 bit with a value of 0 or 1).

- We also did some changes to the ALU file, package file, and control file to support the CSR instructions by adding the necessary signals.

V. Multiply Unit Pipelined

We created another ALU unit specifically for M-extension instructions. Inside of this unit, we had every instruction pass through a multiply bus to emulate a latency of 5. Since in vhdl, a multiply can actually execute instantly, we pass the instruction in the bus 4 times and do nothing with it, and then do the actual multiply in the last stage to simulate it actually executing in 5 clock cycles.

We also had to check the possibility of pipeline a multiply with another multiply. We rely on the signal M_valid, which is 2 bits, to differentiate between M_extension instructions and normal instructions, and between multiply and divide instructions. If the MSB of M_valid is '1', then it is an M_extension instruction, else it is a normal instruction. If the MSB is '1' and the LSB is '0', then it is a multiply instruction. If the MSB is '1' and the LSB is '1', then it is a divide or rem instruction. We are able to differentiate between the multiply and divide instructions based on the MSB of the funct3. So to check for the possibility of pipelining, we set the nop_signal and the stall_signal according to the following conditions:

For the NOP signal, it is 1 bit, and it is set to:

- '0' if a divide or rem instruction has fully propagated through all the buses.
- '0' if a multiply instruction has fully propagated through all the buses.
- '1' if we currently have a multiply or divide instruction in the 1st bus or has passed the first bus and hasn't satisfied the previous conditions.
- '0' if any of the previous conditions weren't met (not an M_extension instruction)

For the Stall signal, it is 2 bits (MSB bit is to differentiate between M_extension instructions and normal instructions, LSB bit is to differentiate between a multiply instruction and a divide instruction). We know when to stall if the MSB bit is 1. (For example: if the Stall signal is "10", we are stalling and we have a multiply, if it is "01" we are not stalling and we have a divide). So, we set the signal to:

- "01 when a divide instruction has fully propagated through all of the buses and when the last instruction in the last bus is the same as the first instruction in the first bus to know that no new divide instructions have entered, since if they entered, we would need to stall.
- "00" when a multiply has fully propagated all of the buses and when the instruction in the 5th stage is the same as the instruction in 4th stage, since if we have 2 pipelined multiply instructions, we need to keep stalling even if the first multiply instruction ended, since the second multiply would still be in the bus, so we need to wait for the second multiply to finish also.
- "10" when a multiply instruction hasn't finished executing yet, so we check if there is another multiply in the ID stage, we pipeline, if not we stall.
- "11" when a divide instruction hasn't finished executing yet, so we stall regardless of what we have in the ID stage since we can't pipeline anything with the divide instruction.

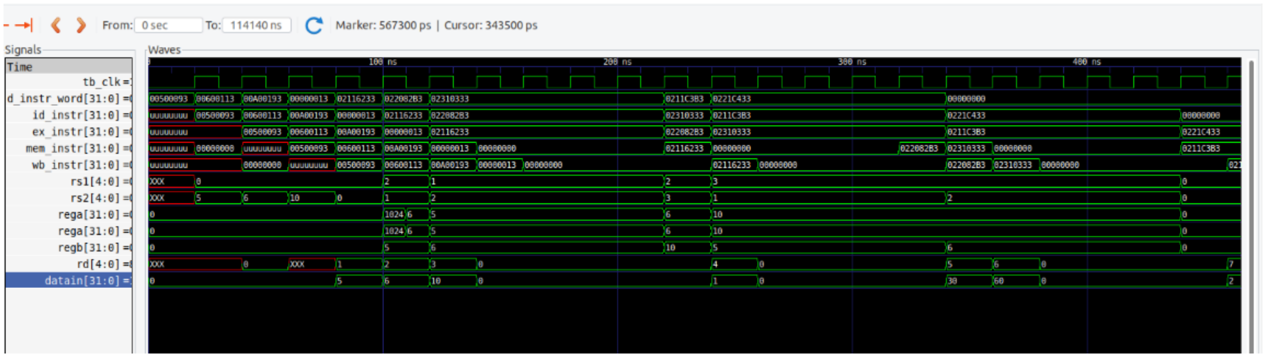
VI. Testing the Multiply Pipelined Datapath

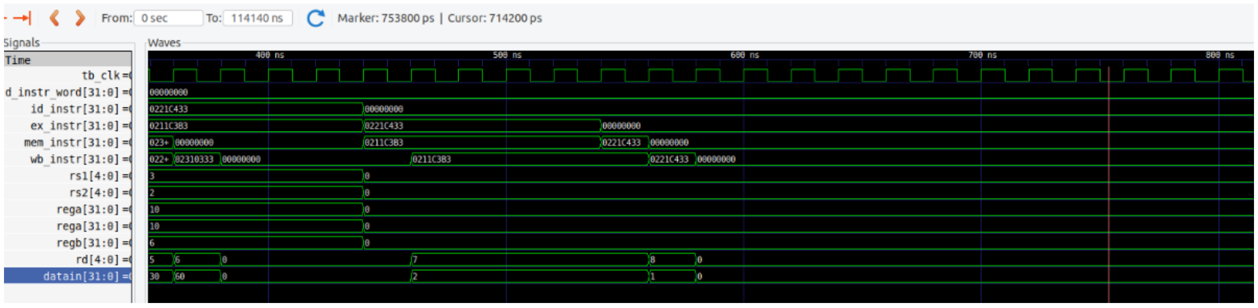
Assembly Code:

```
addi x1,x0,5
addi x2, x0, 6
addi x3,x0,10
nop
```

rem x4, x2, x1
mul x5, x1, x2
mul x6, x2, x3
div x7 ,x3, x1
div x8, x3, x2

<u>Graph:</u>





- Our first instruction is addi x1,x0,5 we can see how it propagates from the IF-stage to the ID-stage to the EX-stage to the MEM-stage and finally to the WB-stage where the value of x1 will become 5 that is demonstrated by datain.

- The second instruction x2,x0,6 was fetched when the first instruction is in the ID-stage, it writes back 1 clock cycle after the writing back of the first instruction and the value of datain became 6, which is the value of x2.

- The third instruction becomes in the WB-stage 1 clock cycle after the second instruction finishes writing back, same as the previous 2 instructions before.

- The fourth instruction is a NOP instruction.

- The fifth instruction is rem x4,x2,x1. It enters the EX-stage and we can verify that it takes 5 clock cycles to execute before it enters the mem stage. We can also verify its value which is 1 since the remainder of the division of 6 by 5.

- The sixth instruction is mul x5, x1, x2 is stalled in the ID-stage since it can't enter the EX-stage before the rem instruction finishes since we can't pipeline both. After the rem instruction finishes execution, this instruction enters the EX stage and we can see that it takes 5 clock cycles as well, passes through memory and then WB where the value of datain becomes 30 which is the multiplication of 5 and 6.

- The seventh instruction is mul x6, x2, x3, it is pipelined with the previous instruction since we can pipeline 2 multiply instructions. It also has a latency of 5 clock cycles in the EX-stage. Finally in the WB stage it writes a value of 60, which is shown as datain of the RegFile.

- The eighth instruction is div x7 ,x3, x1, it can't enter the EX-stage unless the 2 multiply instructions finish execution, so it is stalled in the ID stage. Once both multiply instructions finish execution, it enters the EX-stage for 5 clock cycles as shown in the figure before writing back the value 2 which is shown in datain.

- The nineth instruction is div x8, x3, x2. The divide unit is not pipelined; hence it must wait for the previous divide instruction to finish executing before it enters the EX-stage. Then it writes back and the value of datain becomes 1.

Implementing Data Forwarding

I. Added Components

- Forwarding Unit: Added in the EX-stage, it takes as inputs the memory instruction, the write back instruction, the EX-instruction, the M_instruction, the Mem_RegWrite and the WB_RegWrite. It checks if one of the source registers of an instruction that is available in the ID/EX bus is equal to the destination register of another instruction that is available in the EX/MEM, if yes, we forward. It checks also if one of the source registers of an instruction that is available in the ID/EX bus is equal to the destination register of another instruction that is available in the MEM/WB if yes, we forward also. It also supports forwarding for M-Extension instructions. It takes as outputs the select signals needed for the 4 MUXs that we've added. There are also 4 additional signals that are needed for the 4 MUXs implemented internally inside the M_ALU. These 4 MUXs inside of the M_ALU are needed to forward data from 2 pipelined multiply instructions, since the regular forwarding doesn't support this kind of forwarding.

- MUX1/MUX2/MUX3/MUX4: We added 4 MUXs needed for the data forwarding, their select signal comes from the Forwarding Unit. We attached MUX1 and MUX2 so that the result between what comes from the register file and register A will be the output of MUX1 and the input of MUX2 along with the result from the ALU at the memory stage. Same for MUXs 3 and 4 but for register B instead of A.

II. Modifications in the M_ALU

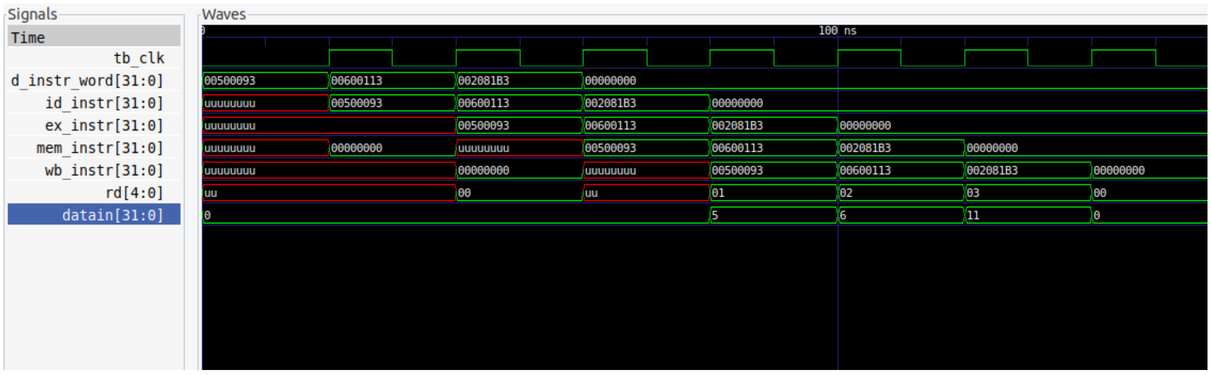
To support forwarding for two pipelined multiply instructions, we had to forward data from the MEM stage or WB stage directly to the M_ALU unit, under the same conditions as before (by checking source and destination registers). This data will be forwarded to the 4th stage of the multiply unit, and if any dependencies were present, the ALU will use the forwarded values to compute the multiply and divide in the last stage. This is done by also adding 4 MUXs inside of the M_ALU. The MUXs will use select signals generated from the forwarding unit.

III. Testing the Forwarding Unit

Assembly Code 1 (Testing for non-forwarded multiply instructions):

```
addi x1,x0,5
addi x2,x0,6
add x3,x1,x2
```

Graph:

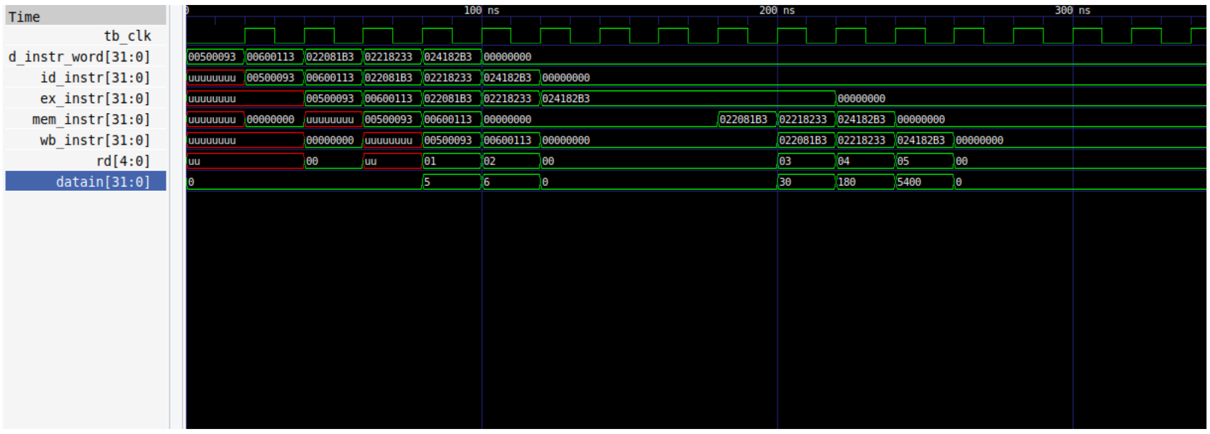


We can see how the first two instructions wrote back their results 5 and 6. The third instruction depends on the previous 2 instructions, yet its value appeared in datain which verified our implementation of forwarding for non-multiply instructions. This was not achievable before since we were using 2 NOPs for it to work. However, now it is supported.

Assembly Code 2 (Testing for forwarded multiply instructions):

addi x1,x0,5
addi x2,x0,6
mul x3,x1,x2
mul x4,x3,x2
mul x5,x3,x4

Graph:



Here we want to test forwarding for multiplication instructions. The first two addi instructions are executed normally. The first mul instruction depends on the previous 2 addi instructions hence we need forwarding, we need to forward from the WB stage of the first addi instruction, and from the memory stage of the second addi instruction. We can see that the first multiply instruction writes back 5 cycles after the second addi instruction writes back which verifies our implementation. For the third mul instruction, it enters the EX-stage pipelined after the first mul but it depends on the result of the previous mul instruction and the second addi instruction. For this we should forward from the MEM-stage of the first multiply once the second multiply reaches the last stage of the multiply, hence it should write back 1 clock cycle after the first mul instruction writes back which what happened in the graph. Same for the last mul instruction that wrote back after 1 clock cycle. The first mul uses the normal forwarding unit, the second mul uses the normal and the special forwarding unit. The third multiply uses the special forwarding unit which is dedicated to forward results of multiply instructions to pipelined multiply instructions.

Implementing Hazard Detection

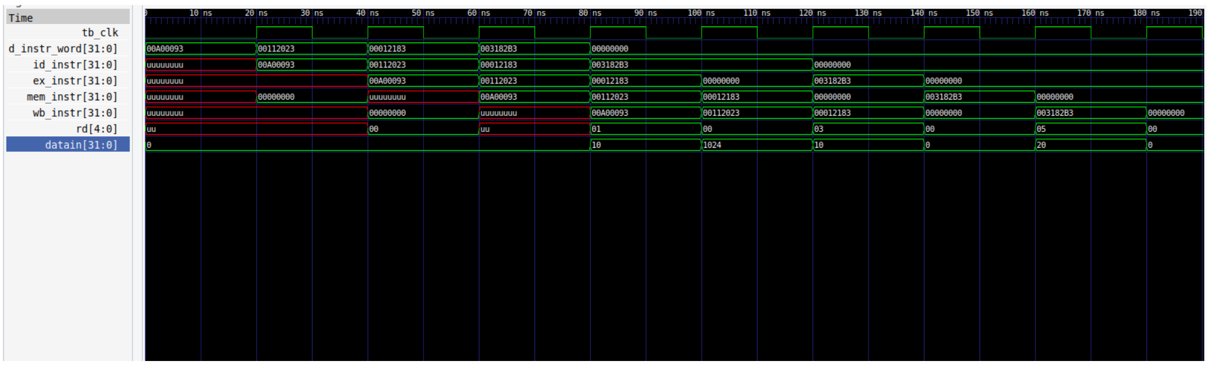
I. Added Components

- Hazard Detection Unit: It takes as inputs the ID-stage instruction, the EX-stage instruction, and the EX_MemRead signal. We need the hazard detection to check if the destination register of a load instruction that will be available in the EX stage is equal to one of the source registers of the instruction in the ID stage. Its outputs will be 2 signals: load_stall needed to stall the IF/ID bus and the PC and load_nop needed to inject a NOP into the ID/EX bus, which enable the load to reach the memory stage and its dependent instruction to still be in the ID stage. The load_stall output of the hazard detection unit will be an added input to the control signal, where it will set the stall signal already present in the control unit (used for the multiply) to '1'. Accordingly, this signal will stall all previous stages along with the PC. The load_nop will directly go to the ID/EX bus where it will insert 1 NOP.

II. Testing the Hazard Detection Unit

<u>Assembly Code:</u>

```
addi x1, x0, 10
sw x1, 0(x2)
lw x3, 0(x2)
add x5, x3, x3
```


Time 0 10 ns 20 ns 30 ns 40 ns 50 ns 60 ns 70 ns 80 ns 90 ns 100 ns 110 ns 120 ns 130 ns 140 ns 150 ns 160 ns 170 ns 180 ns 190
tb_clk
d_instr_word[31:0] 00A00093 00112023 00012183 003182B3 00000000
id_instr[31:0] uuuuuuuu 00A00093 00112023 00012183 003182B3 00000000
ex_instr[31:0] uuuuuuuu 00A00093 00112023 00012183 00000000 003182B3 00000000
mem_instr[31:0] uuuuuuuu 00000000 uuuuuuuu 00A00093 00112023 00012183 00000000 003182B3 00000000
wb_instr[31:0] uuuuuuuu 00000000 uuuuuuuu 00A00093 00112023 00012183 00000000 003182B3 00000000
rd[4:0] uu 00 uu 01 00 03 00 05 00
datain[31:0] 0 10 1024 10 0 20 0

For the dependency between the sw and the addi, the forwarding unit will be used to deal with this dependency. However, for the dependency between the add and the lw, the hazard detection unit will be used here since the two source registers of the add instruction depend on the result of the load instruction at the MEM-stage. Looking at the graph we can see how the add instruction stalled for a one clock cycle in the ID-stage waiting for the result of the load in the MEM-stage and then forwarding took place and was completed and we can see that the add instruction has successfully written its result which is 20 presented in the datain. We can also see the injection of the NOPs represented by all 0s in the EX-stage and that propagates to the MEM and WB stages. Finally, we can verify our code by checking the value written into register x5, which is 20.

Handling Control Hazard

I. Modifications

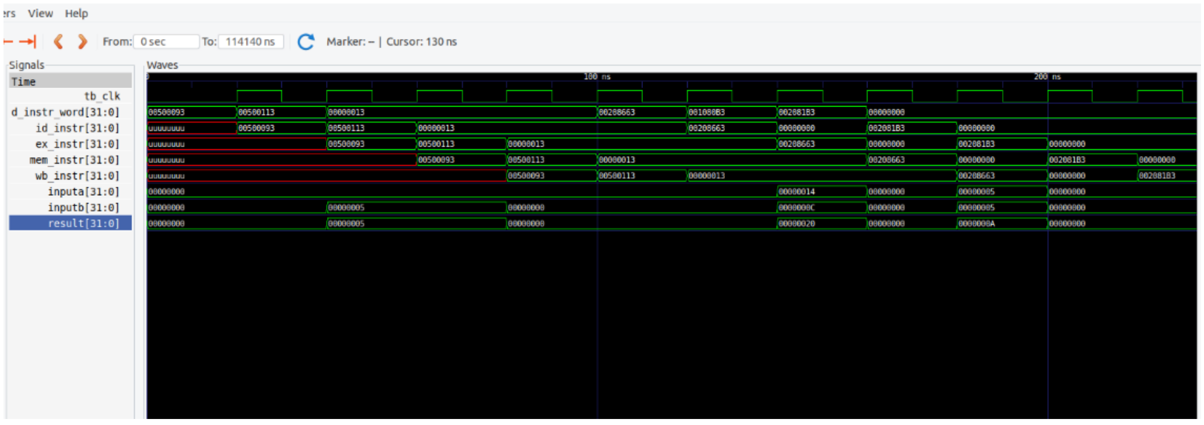
- We usually need to flush instructions if a branch is taken or if there is a jump instruction.
- We did a condition based on PCSrc. For flushing we need PCSrc to be 1 meaning we want the address that is the output of the branch_adder in our case, we don't want the PC + 4.
- To do the flushing we set all outputs of the IF/ID bus to 0 by checking when PCSrc is '1', which means that we have a branch or jump.

II. Testing Flushing

<u>Assembly Code:</u>

```
addi x1, x0, 5
addi x2, x0, 5
nop
nop
nop
beq x1, x2, Test
add x1,x1,x1
add x2,x2,x2
Test:
add x3, x1, x2
```

<u>Graph:</u>



```
d_instr_word[31:0]  00500093 00500113 00000013 00208663 001080B3 002081B3 00000000
id_instr[31:0]      uuuuuuuu 00500093 00500113 00000013 00208663 00000000 002081B3 00000000
ex_instr[31:0]      uuuuuuuu 00500093 00500113 00000013 00208663 00000000 002081B3 00000000
mem_instr[31:0]     uuuuuuuu 00500093 00500113 00000013 00208663 00000000 002081B3 00000000
wb_instr[31:0]      uuuuuuuu 00500093 00500113 00000013 00208663 00000000 002081B3
inputa[31:0]        00000000 00000014 00000000 00000005 00000000
inputb[31:0]        00000000 00000005 00000000 0000000C 00000000 00000005 00000000
result[31:0]        00000000 00000005 00000000 00000020 00000000 0000000A 00000000
```

Here, we're testing how a branch instruction will flush the following add instruction before branching out. The first 2 addi instructions execute normally. Then 3 NOPs were inserted since forwarding is not supported for branch instructions since they execute in the ID stage. We can notice in the graph how the instruction in the ID stage directly after the branch instruction is flushed out since the branch is taken. We can also see how the newly fetched instruction is the add x3, x1, x2 instruction whose result from the ALU is visible in the execution stage which is "A" in hex meaning 10 in decimal.