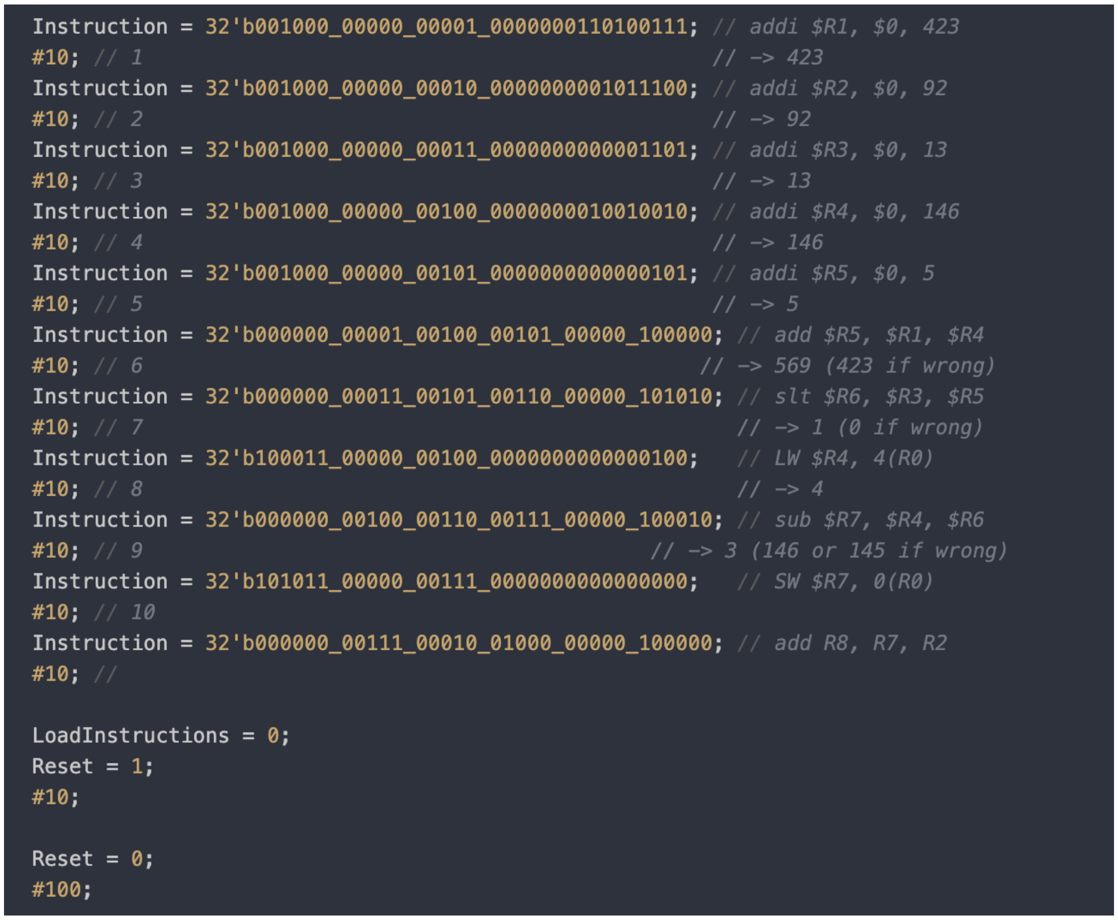
Joshua Arrevillaga
Jason Calalang
EC413 Lab 07: Pipelined CPU

Figure 1: We used the provided tests for our testbench



```
Instruction = 32'b001000_00000_00001_0000000110100111; // addi $R1, $0, 423
#10; // 1                                              // -> 423
Instruction = 32'b001000_00000_00010_0000000001011100; // addi $R2, $0, 92
#10; // 2                                              // -> 92
Instruction = 32'b001000_00000_00011_0000000000001101; // addi $R3, $0, 13
#10; // 3                                              // -> 13
Instruction = 32'b001000_00000_00100_0000000010010010; // addi $R4, $0, 146
#10; // 4                                              // -> 146
Instruction = 32'b001000_00000_00101_0000000000000101; // addi $R5, $0, 5
#10; // 5                                              // -> 5
Instruction = 32'b000000_00001_00100_00101_00000_100000; // add $R5, $R1, $R4
#10; // 6                                            // -> 569 (423 if wrong)
Instruction = 32'b000000_00011_00101_00110_00000_101010; // slt $R6, $R3, $R5
#10; // 7                                                // -> 1 (0 if wrong)
Instruction = 32'b100011_00000_00100_0000000000000100;   // LW $R4, 4(R0)
#10; // 8                                                // -> 4
Instruction = 32'b000000_00100_00110_00111_00000_100010; // sub $R7, $R4, $R6
#10; // 9                                       // -> 3 (146 or 145 if wrong)
Instruction = 32'b101011_00000_00111_0000000000000000;   // SW $R7, 0(R0)
#10; // 10
Instruction = 32'b000000_00111_00010_01000_00000_100000; // add R8, R7, R2
#10; //

LoadInstructions = 0;
Reset = 1;
#10;

Reset = 0;
#100;
```

Arbitration Logic
The arbitration logic in the pipeline is implemented within the Forwarding Unit module, where a 1-ahead is tested, and then a 2-ahed is tested. For example, if an instruction in the EX stage depends on a load instruction in the MEM stage, this creates a 1-ahead hazard. If an instruction in the EX stage requires a register in the WB phase, then it would be considered a 2-ahead. Since the last instruction before the current instruction would have the most up-to-date register, the 1-ahead hazard should have precedence over the two-ahead. This is shown in the arbitration logic by testing for 1-aheads before testing for 2-aheads.

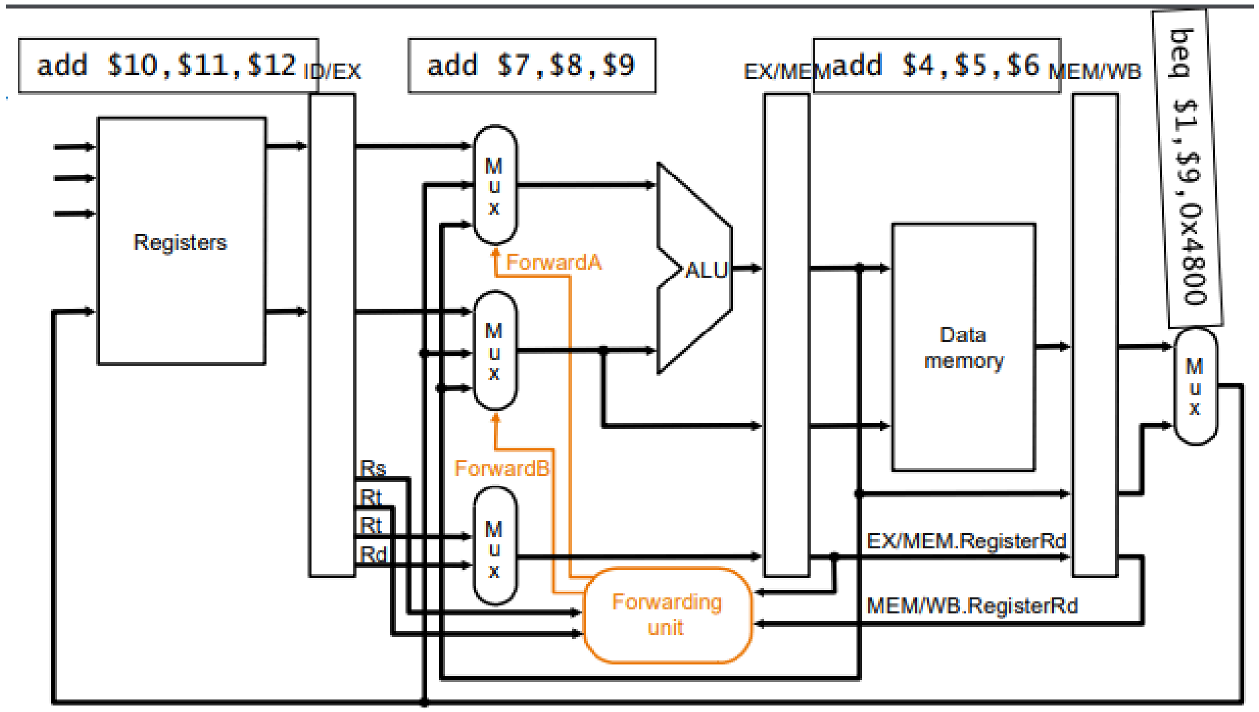
1-ahead Hazard:
In For1Ahead.v, the module detects a 1-ahead hazard by checking if the destination register in the MEM stage (MemDest) matches either of the source registers in the EX stage (IDEX_Rs or IDEX_Rt). If a match is found and the MEM_RegWrite signal is active (indicating that the MEM stage will write to this

register), the module sets ForwardA or ForwardB to 2'b10. This forwards the required data directly from the MEM stage to the ALU in the EX stage, bypassing the register file. This logic ensures that data required by an instruction in the EX stage, which was calculated by the immediately preceding instruction in the MEM stage, is available without introducing a stall, thereby maintaining pipeline efficiency.

2-ahead Hazard:
For 2-ahead hazard handling, For1Ahead.v compares the destination register of the WB stage (WriteBackDest) with the source registers in the EX stage (IDEX_Rs and IDEX_Rt). When a match is detected and RegWriteWB is active (indicating a write operation to this register in the WB stage), ForwardA or ForwardB is set to 2'b01, forwarding the data from the WB stage to the ALU in the EX stage. This allows the EX stage to use data that is two stages behind (currently in the WB stage) without waiting for it to be written back to the register file, thus avoiding stalls in the pipeline.

Writes to $0:
Handling writes to register $0 is essential in this pipeline to maintain the MIPS architecture standard. In the register file implementation within CPU.v, any attempt to write to $0 is ignored or overridden to ensure it always holds zero. This is achieved by adding logic to check if the destination register (WriteBackDest or MemDest) is $0 and bypassing the write if so. This handling maintains $0 as a constant zero. Additionally, this check ensures that the forwarding logic respects this convention, meaning that even if $0 appears as a destination in forwarding paths, it will consistently hold a zero value, thus preventing unexpected behavior in dependent instructions.

Figure 2: Schematic provided by class notes including the forwarding unit


add $10,$11,$12
ID/EX
add $7,$8,$9
EX/MEM
add $4,$5,$6
MEM/WB
beq $1,$9,0x4800
Registers
Mux
ForwardA
ALU
ForwardB
Data memory
Rs
Rt
Rt
Rd
EX/MEM.RegisterRd
MEM/WB.RegisterRd
Forwarding unit