DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Computer Architecture, ECEC-355

Pipelining RISC-V Simulation With Data Forwarding Report

By: Jack Frye, Kelvin Lu, Chris Uzokwe
TA: Shihao Song

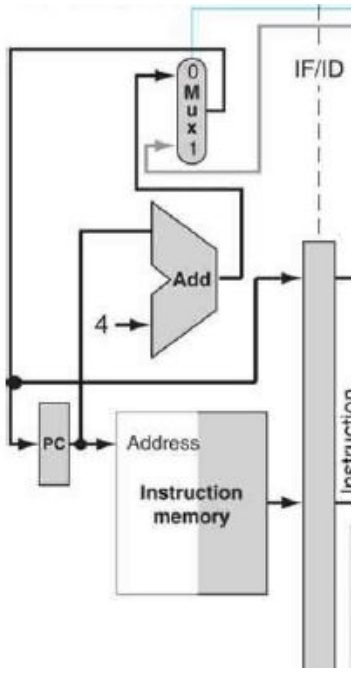
3-17-2019

Completion of the Five Stages

The completion of the five stages was similar to the completion of the core class from project 1. We know that generally, a pipeline and a single cycle architecture consist of the same elements, the biggest difference being the execution timing for all of the different operations. That being said, we are able to get a headstart on the completion of the stages by porting the functions associated with the single cycle implementation to their respective stage in the pipeline.

Fetch (IF) Stage

In the instruction fetch stage, the instruction in read from memory using the address in the PC. The PC address is incremented by 4 and written back to the PC in preparation for the next cycle. The used PC address is passed down in the pipeline's IF/ID register, in case it is needed for other calculations (i.e., beq). The simplicity of this stage can be designed using the instruction memory module from the single cycle.



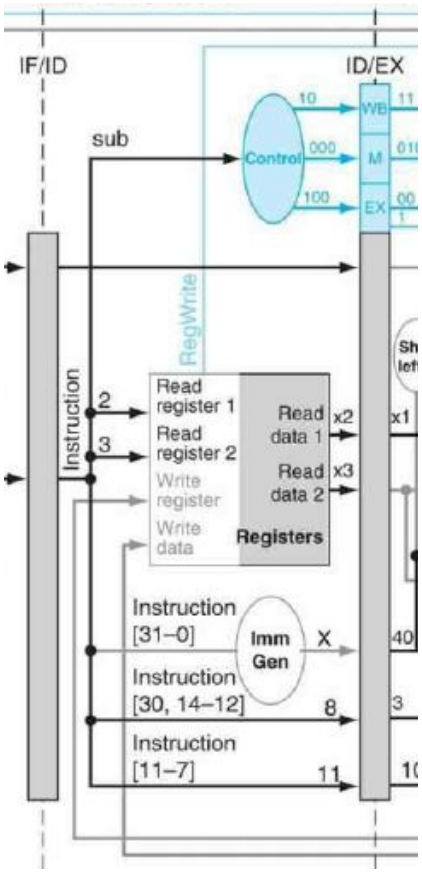

```
if(pc_src == 0)
{
        PC += 4;
}
else
{
        PC = add_sum;
}

/*
 * TODO, fix me. Simulate IF Stage here.
 * Example on how to extract fields have been given.
 * */
if_id_reg.valid = 1;

// For demonstration, I assume all instructions are R-type.
if_id_reg.WB = 1;

if_id_reg.rd_index = (instruction.instruction >> 7) & 31;
if_id_reg.rs_1_index = (instruction.instruction >> (7 + 5 + 3)) & 31;
if_id_reg.rs_2_index = (instruction.instruction >> (7 + 5 + 3 + 5)) & 31;
```

The code snippet above along with the pipeline visualization provides more insight on how this stage was completed. The PC branch decision uses a simple if statement for checking, and the

instruction breakdown in instruction memory simply takes the bits of the retrieved instruction and loads them into the register to be fetched.

Instruction Decode (ID) Stage

The instruction decode stage is where the information received from the fetch is essentially parsed and distributed among the rest of the pipeline. It is in this stage where we find our operands from the register locations, and our immediate sign extended field that may or may not be used. These pieces of information are passed down the ID/EX pipeline register, which like for all of the other registers, we'll be using a C struct to simulate that held information.

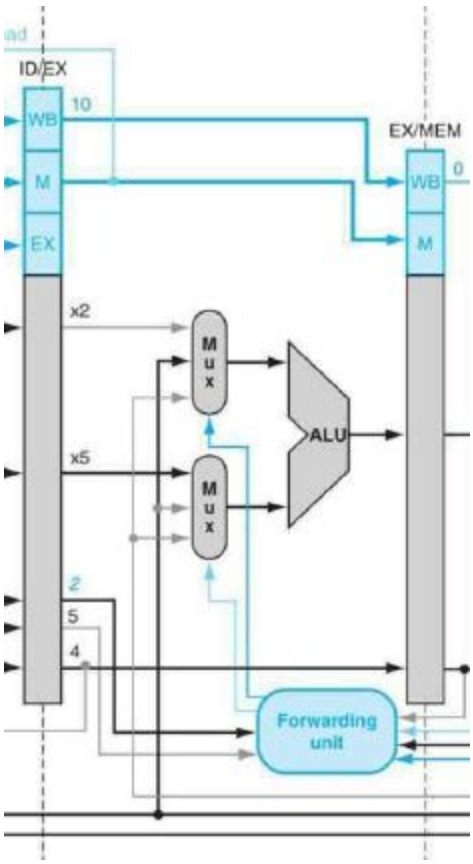



```
    id_ex_reg.branch = control.get_branch();
    id_ex_reg.mem_read = control.get_mem_read();
    id_ex_reg.mem_to_reg = control.get_mem_to_reg();
    id_ex_reg.alu_op1 = control.get_alu_op_0();
    id_ex_reg.alu_op2 = control.get_alu_op_1();
    id_ex_reg.mem_write = control.get_mem_write();
    id_ex_reg.alu_src = control.get_alu_src();
    id_ex_reg.reg_write = control.get_reg_write();

    id_ex_reg.PC = if_stage->if_id_reg.PC;

    id_ex_reg.read_data1 = registers.read_reg(if_stage->if_id_reg.rs_1_index);
    id_ex_reg.read_data2 = registers.read_reg(if_stage->if_id_reg.rs_2_index);

    imm_gen.set_imm_gen(op_code, if_stage->if_id_reg.instruction);
id_ex_reg.imm_gen_result = imm_gen.get_imm_gen_result();

    id_ex_reg.funct7 = (if_stage->if_id_reg.instruction >> 25) & 0x7F;
id_ex_reg.funct3 = (if_stage->if_id_reg.instruction >> 12) & 0x7;
```

As you can see in the code and pipeline portion above, the decode stage is similar to the instruction memory of the fetch stage, in that it assigns bits of information to various points in the ID/EX register, except it is more extensive. It's the control, and and alu function bits that will decide most of our pipelines operation.

Execution (EX) Stage

In the execution stage, the register contents are read and manipulated as desired. So, this could include a bit shift, add, subtract, AND, or OR. The specifications will come from The control unit and funct bits, which will be passed over from the decode stage.

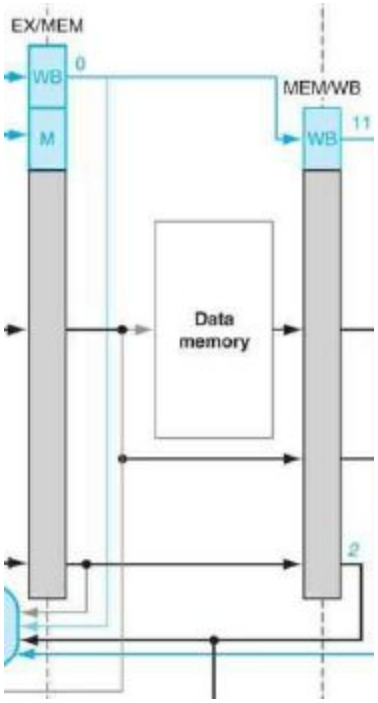



```
int64_t mux_read_data2;
if (id_stage->id_ex_reg.jump) {
    //registers->assign_reg(write_register, PC+4);
    //PC = imm_gen_result;
}
if (id_stage->id_ex_reg.jalr) {
    //PC = registers->read_reg(read_register1);
    //PC += registers->read_reg(write_register);
    //PC += imm_gen_result;
}
else {
    // ALU Mux
    if (id_stage->id_ex_reg.alu_src) {
        mux_read_data2 = id_stage->id_ex_reg.imm_gen_result;
    }
    else
    {
        mux_read_data2 = id_stage->id_ex_reg.read_data2;
    }

    alu.set_alu_ops( id_stage->id_ex_reg.read_data1,
                     mux_read_data2,
                     id_stage->id_ex_reg.alu_op1,
                     id_stage->id_ex_reg.alu_op2,
                     id_stage->id_ex_reg.funct7,
                     id_stage->id_ex_reg.funct3);

    ex_mem_reg.alu_out = alu.get_alu_result();

    ex_mem_reg.alu_zero = alu.get_alu_is_zero();
}
```

The execution stage depicts the processing of chosen data from the other sections of the register. Although this section of code depicts the alu operands being muxed between the immediate data and register, it is worth noting that the forwarding unit information can also be considered in the case of avoiding hazards. This concept will be elaborated on later.

Memory (MEM) Stage

In the memory stage, memory is accessed to be written back to a register. The memory address can be calculated in the execution stage, and passed over through the EX/MEM pipeline register. This may include a load, store, or jal instruction.

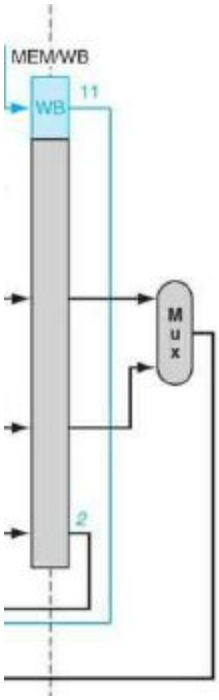



```
if (ex_stage->ex_mem_reg.mem_read)
{
    mem_wb_reg.data_mem_read = data_memory.read_data(ex_stage->ex_mem_reg.alu_out);
}

    mem_wb_reg.alu_zero = ex_stage->ex_mem_reg.alu_zero;
    mem_wb_reg.alu_out = ex_stage->ex_mem_reg.alu_out;

    /* Branching for IF stage */
    if(ex_stage->ex_mem_reg.branch && ex_stage->ex_mem_reg.alu_zero)
    {
            if_stage->pc_src = 1;
            if_stage->add_sum = ex_stage->ex_mem_reg.add_sum;
    }
```

The memory stage simply depends on writing or reading data from the data memory. This means we can check whether such action is necessary, and assign data as specified -- the code snippet shows a basic action of this.

Write Back (WB) Stage

The write back stage represents the final stage in the pipeline. It reads data from the MEM/WB pipeline register back into the register file. The register destination is specified in the instruction and will be passed down to this stage.




```
int64_t tmp;
tmp = (mem_stage->mem_wb_reg.mem_to_reg) ? mem_stage->mem_wb_reg.data_mem_read : mem_stage->mem_wb_reg.alu_out;

if (mem_stage->mem_wb_reg.reg_write)
{
        id_stage->registers.assign_reg(mem_stage->mem_wb_reg.rd_index, tmp);
}
```

For the write back, we can simply reassign the registers designated in the pipeline.

Hazard Detection Unit

The hazard detection unit of the pipelined processor architecture is the logical component responsible for handling errors in the pipeline when different stages attempt to access register data at inappropriate times and also when branching and other control flow mechanisms change the way instructions are fetched.

The two types of hazard specified above that can occur in the pipeline are data and control hazards. Data hazards are the former described hazard and occur when a subsequent instruction requires the output of the previous task but accesses the data at a stage in the pipeline before the register's contents are appropriately updated.

As an example, when a load instruction is immediately followed by an operational instruction, the operational instruction (assuming it was the immediate next instruction) will attempt to access the register two clock cycles before the data will actually be updated. A visualization of this is provided in Figure 1.

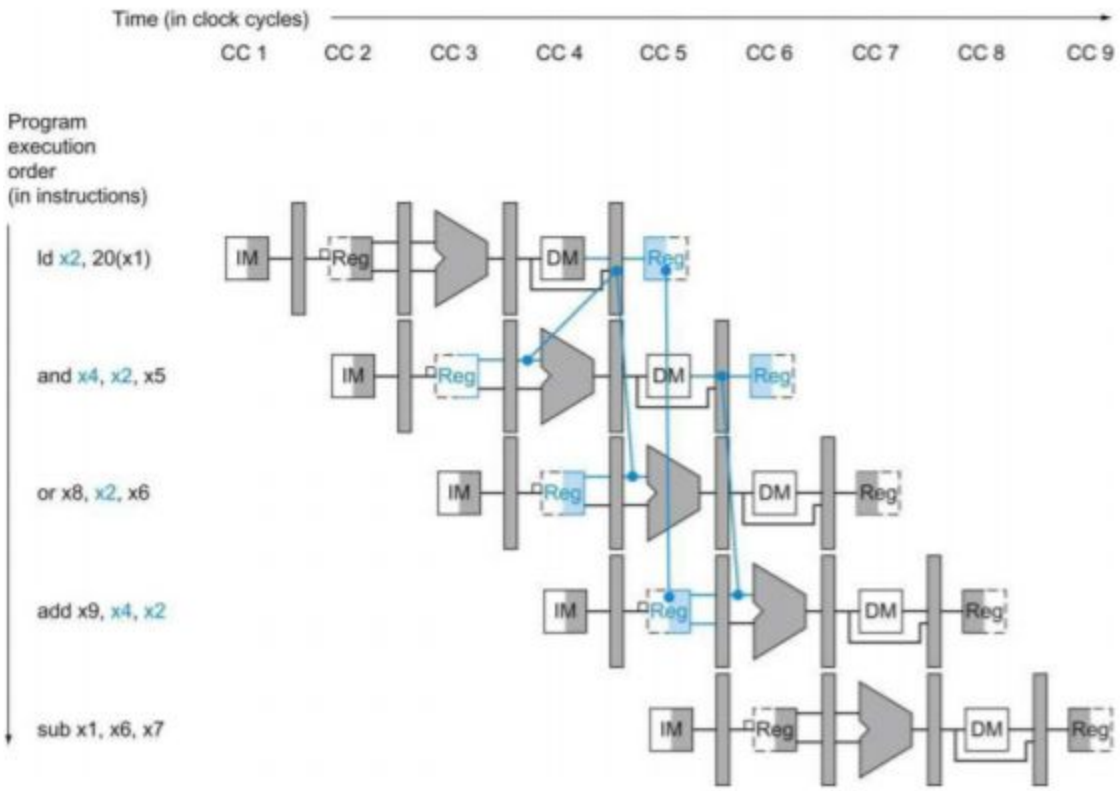


Figure 1. A data hazard scenario in which subsequent instructions attempt to access x2 before the first ld instruction completes its write-back (WB) stage.

In order to detect such a hazard, conditional logic was inserted into the Instruction Decode (ID) stage to check the following: **if the ID/EX has the MemRead signal high and if the ID/EX destination register is equal to either IF/ID source registers**. If the following logic is true at any point, the pipeline must introduce a stall nops instruction so that the hazard is avoided. Since the dependence between the load and following instruction goes backward in time, this hazard cannot be solved by data forwarding. The nops instruction was achieved by setting all seven control signals to low (zero). By setting all controls to zero, the instruction preserves the state of the pipeline, memories, registers, and delays new instruction fetching.

Control or branch hazards in contrast to data hazards occur due to the fact the the pipeline must fetch an instruction at every clock cycle but does not actually know what instruction to fetch from a branch instruction until the MEM stage.

The method chosen to handle this hazard in this project was to always assume the branch is not taken. This allows our implementation of RISC-V pipeline simulator to less frequently have to stall. In the case that the branch is taken, the simulator discards or flushes the wrong instructions by deasserting all control signals similar to the stall done mentioned in the load-use data hazard above. However, this was done at the IF, ID, and EX stage since the control would not be detected until the MEM stage.

Forwarding Unit

The method of hazard control we will be implementing is the data forwarding unit. Since we are able to understand the dependencies during each clock cycle, we can understand when information from a certain register or calculation must be used, and essentially eliminate the necessity for a write back stage to be used (although obviously, the information will still have to be written back in the write back stage), and the data from the executed command can just be piped directly to where it needs to be used.

The entire forwarding unit logic implemented in the simulator was programmed by checking for two types of data hazards which can be resolved via forwarding. The following types and their conditions are below:

EX Hazard

1.) If EX/MEM.RegWrite
2.) and EX/MEM.RegisterRd does not equal x0
3.) and EX/MEM.RegisterRd equals ID/EX.RegisterRs1 or ID/EX.RegisterRs2
    a.) For case of RegisterRs1 → ForwardA = 0b10
    b.) For case of RegisterRs2 → ForwardB = 0b10

MEM Hazard

1.) If MEM/WB.RegWrite
2.) And MEM/WB.RegisterRd does not equal x0
3.) And not conditions 4 through 6
4.) And EX/MEM.RegWrite
5.) And EX/MEM.RegisterRd does not equal x0)
6.) And EX/MEM.RegisterRd equals ID/EX.RegisterRs1 or ID/EX.RegisterRs2
    a.) For case of RegisterRs1 → ForwardA = 0b01
    b.) For case of RegisterRs2 → ForwardB = 0b01

Table 1. A complete table for the forwarding logical unit which explains where it comes from and what happens in the processor

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.