CA Lab Project Spring 2021

Final Report

Habib University

Dhanani School of Science and Engineering

Group Members:

Alisha Momin – am05757

Anmol Jumani – aj06198

Sana Fatima – sf06199

Syeda Areesha Najam – sn05985

Abstract:

For the course project of Computer Architecture we are required to build a 5-stage pipelined processor that will be able to execute bubble sort program. We have been assigned with three tasks in total. In the first task of this project we will be working on single cycle processor (already done in lab 11) and try to execute a bubble sort program on it along with modification in few modules. For the second task the processor will be modified and a 5-stage pipelined processor will be implemented where forwarding will be done to tackle data hazards.In the third task flushing and stalling circuitry will be introduced to handle structural and control hazards. The project will also include the concepts of forwarding, installing and flushing the pipeline.

In this report we will discuss the implementation of all these three tasks in detail.

Pre requisite required for this project:

For task 1 implementation we used a few modules that were already implemented in previous labs. The modules included,

- Program_Counter
- Instruction_Memory
- Instruction_Parser
- ALU_64_BIT
- Register_file
- Control_Unit
- ALU_Control
- adder
- mux_2x1
- Data_Memory
- immediate_data_gen (refer this as extractor in circuit)

Below is the description of each module,

- **PROGRAM COUNTER:**
  The program counter also known as PC register has the address of current instruction and in order to read the next instruction program counter needs to be incremented by 4 as each instruction is 4 byte long. Program counter is updated at the end of every clock cycle. Program counter takes three inputs clk, reset, a 64-bit input, PC_In; and a 64-bit output, PC_Out. Basically we initialize PC_Out to 0 if reset signal is high, else reflect the value of PC_In to PC_Out, at the positive edge of clock. PC is used to determine where to fetch the next instruction from the instruction memory.
- **INSTRUCTION MEMORY:**
  The instruction memory is used to store instructions. During the processor execution, it is required to only read the instruction from the instruction memory and not to write any data or instruction in it. Hence, we could say that the instruction memory is the read-only memory. Instruction_Memory, having a 64-bit input, Inst_Address, and a 32-bit output, Instruction. We declare a 1x16bytes instruction memory internally in the module and initialized its contents. The 64-bits width of Inst_Addr input can access 2^64

memory locations. The behavior of this module is designed as such that whenever an Inst_Address field is changed, the 32-bit instruction corresponding to the Inst_Address should appear at the 32-bit output port, Instruction.

- **ALU_64_BIT:**

  The arithmetic logic unit (ALU) is the brain of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. It performs the operation that is shown by the instruction. It has 3 inputs A and B of 64 bits and 4-bit wide operation selector value called ALUop and 2 outputs that is the result of the operation is the output value i.e. Result which is 64 bit wide and additional output named ZERO whose output should be set to 1 if the Result is 0, else we set it to 0.

- **REGISTER FILE:**

  Register files are necessary for computer memory. For a computer to be able to function, it needs to have some form of memory. There are two different forms of memory devices: external memory devices and local memory devices. External memory devices are items such as RAM, ROM, disk drives, etc. Registers provide local memory, which allows for temporary storage of data that is about to be processed. In our module register is used to save and store numbers. This module have the input address ports for reading register 1, register 2, and for writing data in a destination register. It have a 64-bits data input port, and two 64-bits data output ports to read the values from two different registers and have a clk, reset, and a RegWrite signal which controls when the data should be written to a register. It is used to put all the data (instructions) into data memory. Therefore we can view the register file as an array of registers.

- **CONTROL UNIT:**

  The control unit takes the instruction and generates appropriate signals for data path elements. It also helps in executing operations in a proper sequence that is given by the program that needs to be executed. Control_Unit takes a 7-bit wide input, named Opcode, and generate 7 output signals. Out of these seven outputs, one is ALUOp which is 2-bits wide, and the remaining six are 1-bit wide, which are Branch, MemRead, MemtoReg, MemWrite, ALUSrc, and RegWrite. The control unit fetches data and instructions from memory and uses operations of the ALU to carry out those instructions using that data. [3]

- **ALU CONTROL:**

  It is used to perform all operations for R-type instructions, it also performs addition for loads and stores, and is also used for performing subtraction for branches. The ALU control unit decides which type of result will be output from the ALU. ALU_Control takes a 2-bit input, named ALUOp and a 4-bit input, named Funct, and produces a 4-bit output, called Operation. While the number of bits used for the encoding of the ALUOp and ALU Control output signals is determined by the number of operations it controls, the encoding of these signals is really arbitrary. [1]

- **ADDER:**

  The adder takes in two input values, performs addition of those two inputs and outputs the result. It is basically used to increment the program counter in order to hold the address of the next instruction. Adder takes two 64-bits inputs, A and B; add them, and reflect the results at the 64-bit output port, OUT.

- **MUX_2x1:**

  A 2-to-1 multiplexer consists of two inputs A and B and each of them are 64-bits wide. It has one select input SEL and one output data_out. Depending on the select signal, the output is connected to either of the inputs. Since there are two input signals, only two ways are possible to connect the inputs to the outputs, so one select is needed to do these operations. [2]

- **DATE MEMORY:**

  The data memory function implements the functionality for reading and writing data from and to memory. It takes two inputs among which one is for the data to be written in memory and other is for address for accessing memory location and the output is the read data that was accessed from the memory location. In this module, the data at the input port, Write_Data, that is written at the positive edge of clk signal and when MemWrite signal is asserted (i.e. high). The data can be read from memory at any instant whenever the Mem_Addr value changes and when MemRead signal is asserted.

- **IMMEDIATE DATA EXTRACTOR:**

  Immediate data extractor takes the 32-bit input instruction and extracts the 12-bit immediate data field depending on the type of instruction. Then sign-extend these 12-bits to 64-bits output imm_data.

- **INSTRUCTION PARSER:**

  Instruction parser takes a 32-bit input, named instruction, and generates different outputs. The outputs are 6 bit wide Opcode and funct7 and a 2 bit funct3 and 4 bit wide rd,rs1,rs2.

TASK 1:

https://edaplayground.com/x/9QCD

Objective:

Modify the single-cycle processor to be able to run the bubble sort code on it. Test and verify that it is doing the sorting correctly.

Implementation of Single-cycle Processor:

All of these modules and registers with their functionalities are described above. These modules are mentioned in separate .v files and are being used in main design.sv where the actual implementation of single cycle processor has been done in such a way that it is able to run bubble sort code. All these files are first included in design.sv and then clock and reset signals are provided as inputs in the single cycle Processor's main module. With the help of provided circuit implementation below we instantiated all the modules.

At first, the Program counter module is instantiated i.e. the PC_In input is given through a wire and similarly the output is stored in another wire made of same bits as of PC_Out. The output of Program Counter (PC_Out) was then transferred to Instruction Memory, Adder1 and Adder2. We can understand how modules are instantiated one after another and how inputs and outputs are transferred via wires declare initially in the main module from the figure attached above.

PC_Out of Program counter were sent to the Adder1 where the input A is the PC_Out of program counter and B is taken as 64'h4 which is 4 in decimal(in our code we set a variable b which is equal to 64'h4). Also the PC_Out of program counter were sent to the Adder2 where the input A is the PC_Out of program counter and b is the imm_data from data extractor module which is shifted left by 1 (in our code it is represented as b1). PC_Out of program counter is also given as inputs to Inst_Addr of Instruction Memory module and it was instantiated. Then the output Instruction from Instruction Memory is given as input of Instruction parser, immediate data extractor and ALU control module.

In Immediate data extractor the input is taken from the output Instruction of instruction memory module and the output is imm_data which is used in adder2 by shifting that left by 1. The instruction from instruction memory is also used in ALU control input Funct in which we have passed {Instruction [30], Instruction [14:12]} and the input ALUOp is taken from the control unit and in result we get the output Operations (in our code Funct = Functx). In Instruction parser, the instruction from instruction memory is taken as input which got parsed and gives an output as Opcode, rd, func3, rs1, rs2 and func7.

Next comes the Register file. So the input of register file is clk, reset, RS1, RS2, RD, RegWrite, WriteData. Over here RS1, RS2, RD is transferred from the output of instruction parser rs1, rs2, and rd. Whereas RegWrite is taken from the output of control unit module and the input of WriteData of register file is taken from the output of mux3 which all execute and give us the output ReadData1 and ReadData2. Next comes the control unit. The input of control unit is the output Opcode from the instruction parser module and once it is executed it give us the output of Branch, MemRead, MemtoReg, ALUop, MemWrite, ALUSrc and RegWrite.

Next comes the Mux. In our single cycle processor we have 3 mux with different input and outputs. Mux1 is the mux after register file in which we take A, B and sel as an input and out as an output. Over here A is the ReadData2 which is the output of register file module. B is the

imm_data from immediate data extractor. Sel is the ALUSrc from the control unit and the output is data_out1 which is transferred to ALU64 input. Mux2 is the mux after adder2 in which A is the output of adder1, B is the output of adder2 and sel is the AND gate of branch and Zero (in our code Zero = equal_) and the output of the mux2 is then transferred to the PC_In which is basically the input of program counter. Mux 3 is the mux after Data memory in which A is the output from the ALU64 result. B is the output from Data Memory Read_Data and sel is basically the output of Control unit ALUOp and the output of mux3 is transferred to the register file WriteData.

Next comes ALU64 in which we have a, b, ALUOp and Funct as an input and Result and equal as an output. The input A is transferred from Register file module output ReadData1 and B is transferred from mux1 output.

Next comes the Data memory has the input clk and other input are mem_addr is the input taken from the output of ALU64 module and Write_data is the output of register file module and MemWrite is the output from the control unit module and MemRead is taken from the control unit output these all execute and give an output of Read_Data.

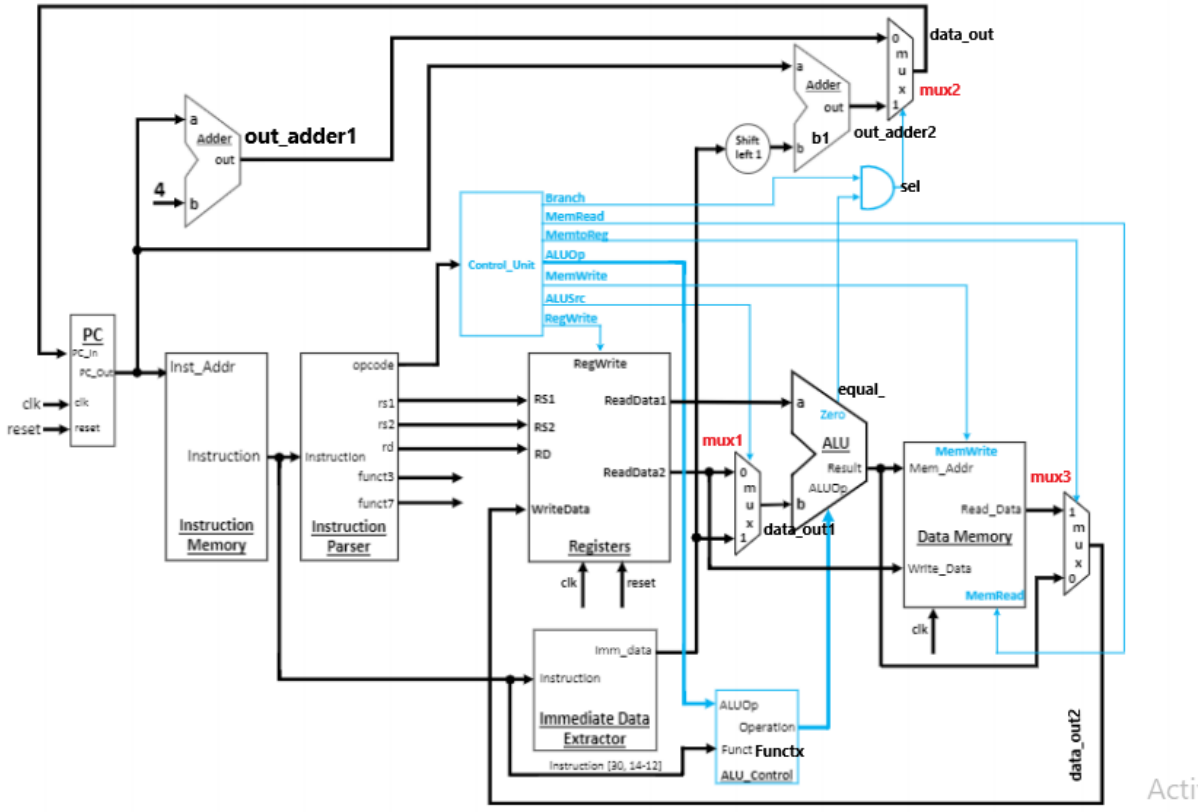


Fig. 11.1. Single cycle processor.

Bubble Sorting:

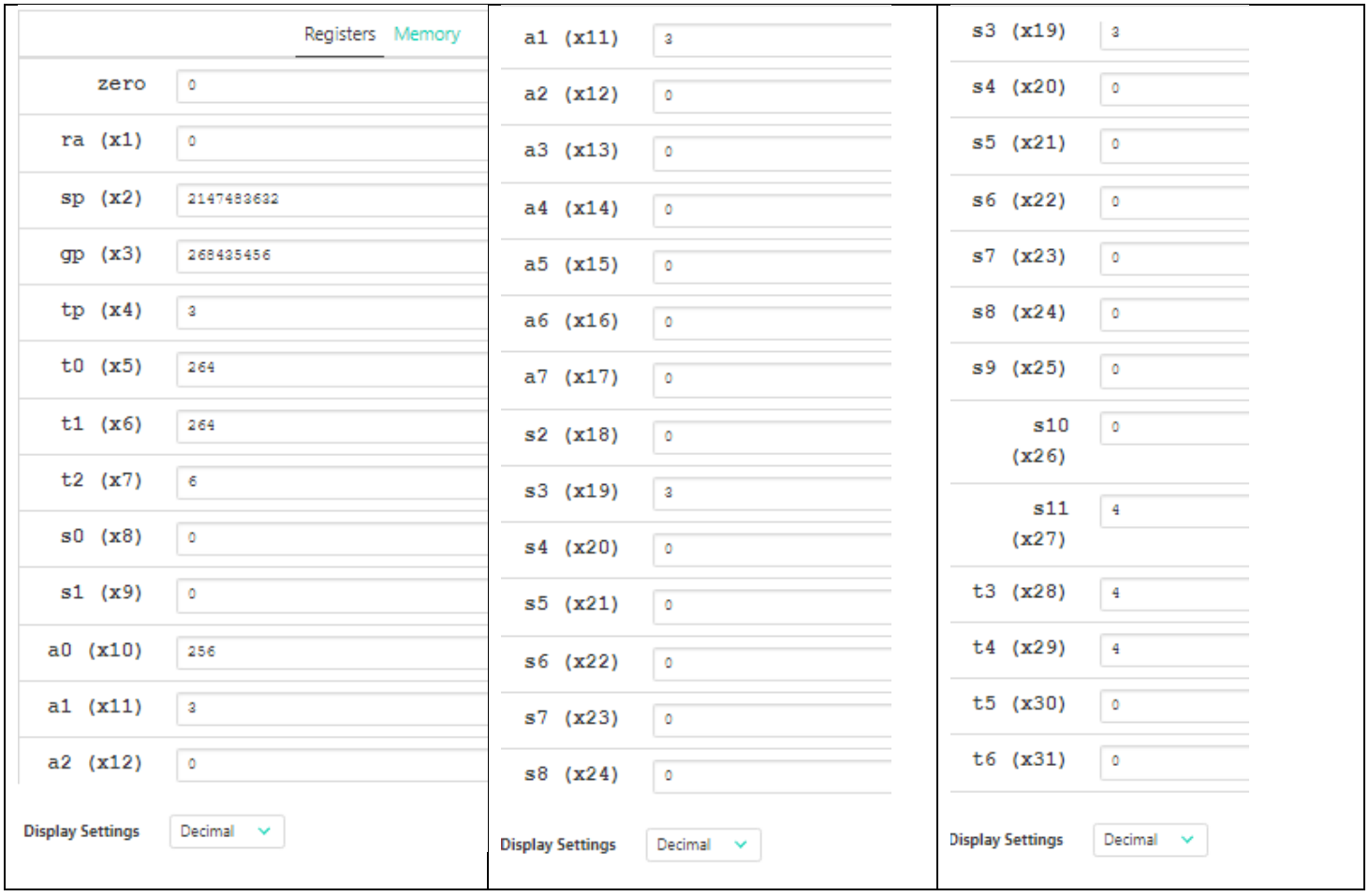
Bubble sort code was implemented in lab 4 which has now been used for the project to see how it works on RISC V processor (single cycle and pipelined). Below is the explanation for the code and the attached screenshots show its simulation.

- Using addi and store word instructions we set up the array of length 3. Then using addi instructions we are initializing the base address of our array a in register x10, and the length of array a has been initialized in register x11. Then we have started implementing the function of bubble sort. So first we have checked whether the length of our array is that is initialized in x11 is zero or not. If the input length of array is zero we will final exit from the program and bubble sort will not run while if the length is non zero we will move to next instruction where a integer variable i = 0 is being initialized to register x4. Then in loop 1 the first instruction checks if length of array equals i if true we move to exit1 which ultimately takes us to final exit because if we see, exit1 checks if x0 = 0 which is always true so if i that has been initialized in x4 equals the length that has been initialized in x11 the code exits. Otherwise it goes to next instruction where by using add instruction a correspondence is made between variable j and register x19 that is associated to variable i in x4. Loop 2 keeps running until variable j in x19 is not equal to length in x11 and when they become equal we jump to exit2 which performs incrementing of i by 1 (i+=1) (line 47 of snippet) and then it goes to exit1 and from that it goes to final exit. While j is not equal to length it goes in the loop, we have taken two temporary registers x5 and x6 slli instruction does shifting. So variable i in x4 is now in x5 by shifting it to 2 bits and similarly variable j in x19 is now in x6 b shifting it to 2 bits. Then we are adding x10 to x5 which becomes address of a[i] and x10 to x6 which becomes address of a[j]. Then a[i] and a[j] are being copied from memory to temporary registers that are x28 and x29 respectively. Now it checks if i in x28 is greater than j in x29 we go to iterate which increments variable j by 1 (j+=1) (line 45 of snippet) and if it is not greater swapping is performed (line 38-40) and then storing is performed (things in register are put in memory). This way the code of bubble sort works.

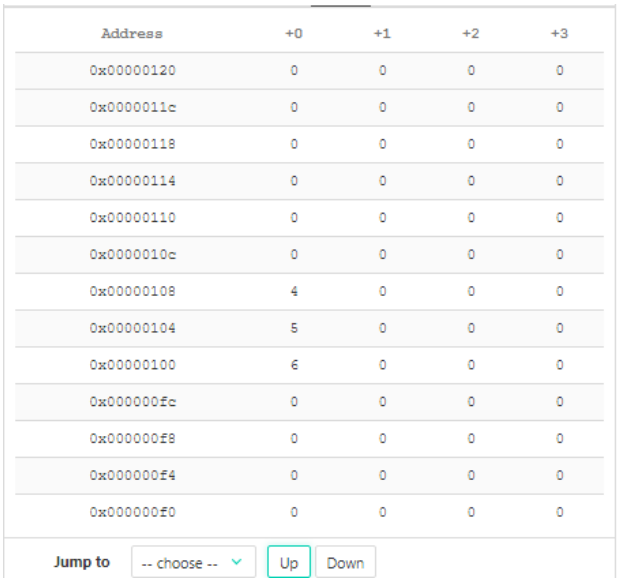
Code:

```
#setting up array of length 3
addi x7, x0 ,4
sw x7, 0x100(x0)
addi x7, x0 ,5
sw x7, 0x104(x0)
addi x7, x0 ,6
sw x7, 0x108(x0)

addi x10, x0, 0x100 #address of array a
addi x11, x0, 3     #length of array
#jal x1, bubble
#beq x0, x0, FinalExit


#Function call
bubble:
#base condition: if (len==0)
beq x11, x0, FinalExit
#jalr x0, 0(x1)

addi x4, x0, 0   #initialize i
Loop1:
    beq x4, x11, Exit1    #i==len then exit loop
    add x19, x0, x4      #j = i
    Loop2:
        beq x19, x11, Exit2     #j==len then exit loop
        slli x5, x4, 2          #temp reg x5
        slli x6, x19, 2         #temp reg x6
        add x5, x5, x10         #address of a[i]
        add x6, x6, x10         #address of a[j]

        #copy a[i] and a[j] from memory in to temp registers

        lw x28, 0(x5)
        lw x29, 0(x6)
        bge x28, x29, iterate    #if not a[i] >= [j]

        add x27, x0, x28    #temp = a[i]
        add x28, x0, x29    #a[i] = a[j]
        add x29, x0, x27    #a[j] = temp

        sw x28, 0(x5)
        sw x29, 0(x6)

        iterate: addi x19, x19, 1  #j+=1
        beq x0, x0, Loop2
    Exit2: addi x4, x4, 1   #i+=1
    beq x0, x0, Loop1

Exit1: beq x0, x0, FinalExit
FinalExit: nop
```

Simulation:



Memory:



Address	+0	+1	+2	+3
0x00000120	0	0	0	0
0x0000011c	0	0	0	0
0x00000118	0	0	0	0
0x00000114	0	0	0	0
0x00000110	0	0	0	0
0x0000010c	0	0	0	0
0x00000108	4	0	0	0
0x00000104	5	0	0	0
0x00000100	6	0	0	0
0x000000fc	0	0	0	0
0x000000f8	0	0	0	0
0x000000f4	0	0	0	0
0x000000f0	0	0	0	0

Jump to -- choose -- Up Down

Instruction wise Waveform Description/Explanation:

This is the set of instructions which we are taking as an example to explain our implemented pipelined processor:

- beq x11, x0, FinalExit (in binary 100|00000101|1000110001100011)
- addi x4, x0, 0 (in binary 000000000000|0000|0|000|00100|0010011)
- beq x4, x11, Exit1 (in binary 10010110010|00000110|01100011)

Instruction Memory:

Instruction Memory has a 64-bit input Inst_Address, and a 32-bit output i.e. Instruction. Here the instruction is generated using:

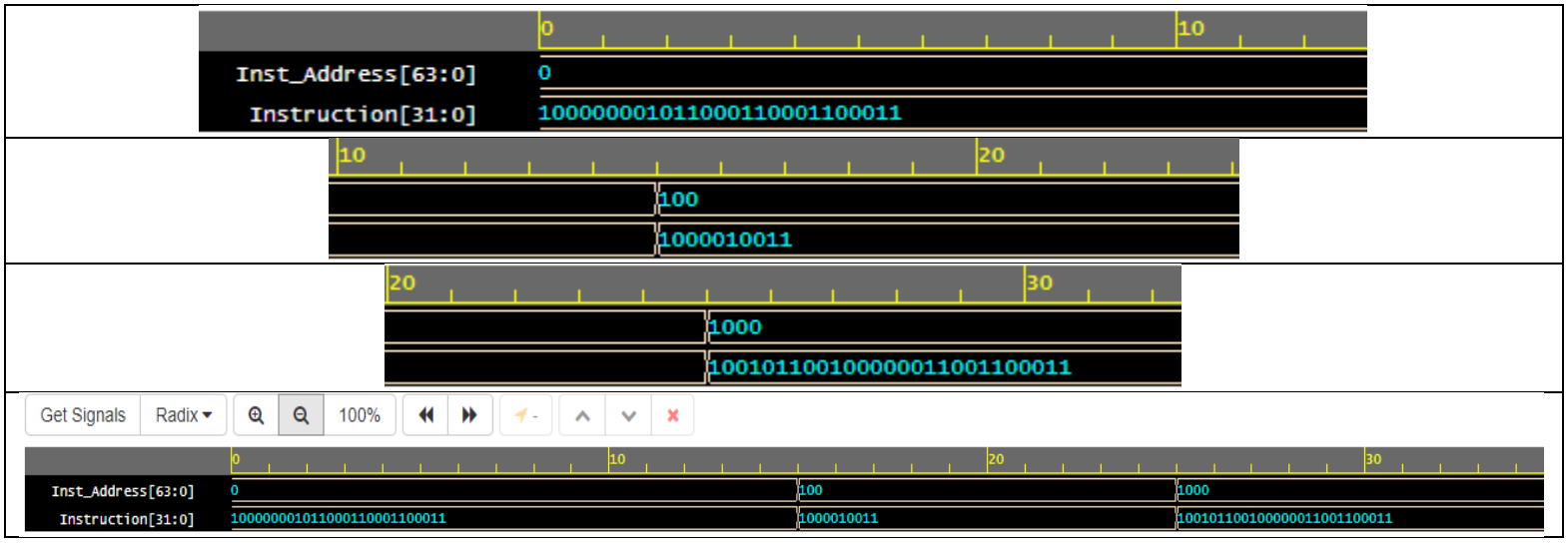
```
174    assign Instruction = {memory[Inst_Address+3], memory[Inst_Address+2],
175          memory[Inst_Address+1], memory[Inst_Address]};
176  endmodule
```

Which means that when the Inst_Address is 0 then memory [3], memory [2], memory [1], memory [0] will form the first instruction that is beq x11, x0, FinalExit.

Similarly when Inst_Address is 4 (100 in binary), then memory [7], memory [6], memory [5], memory [4] will form the second instruction that is addi x4, x0, 0 and for the Inst_Address is 8 (1000 in binary), then memory [11], memory [10], memory [9], memory [8] will form the third instruction that is beq x4, x11, Exit1.

Note that for loading the instruction memory for all three tasks we first simulated the bubble sort code and from there we generated a dump file. This file is also attached with our submission. The dump file contains hex code for all instructions. We converted those hex codes in machine language and then this machine code is allotted at four appropriate indexes of memory. You may refer to the .v file attached in our submission to get a better idea of one by one execution of instructions.

This exact result is shown in the waveform below:

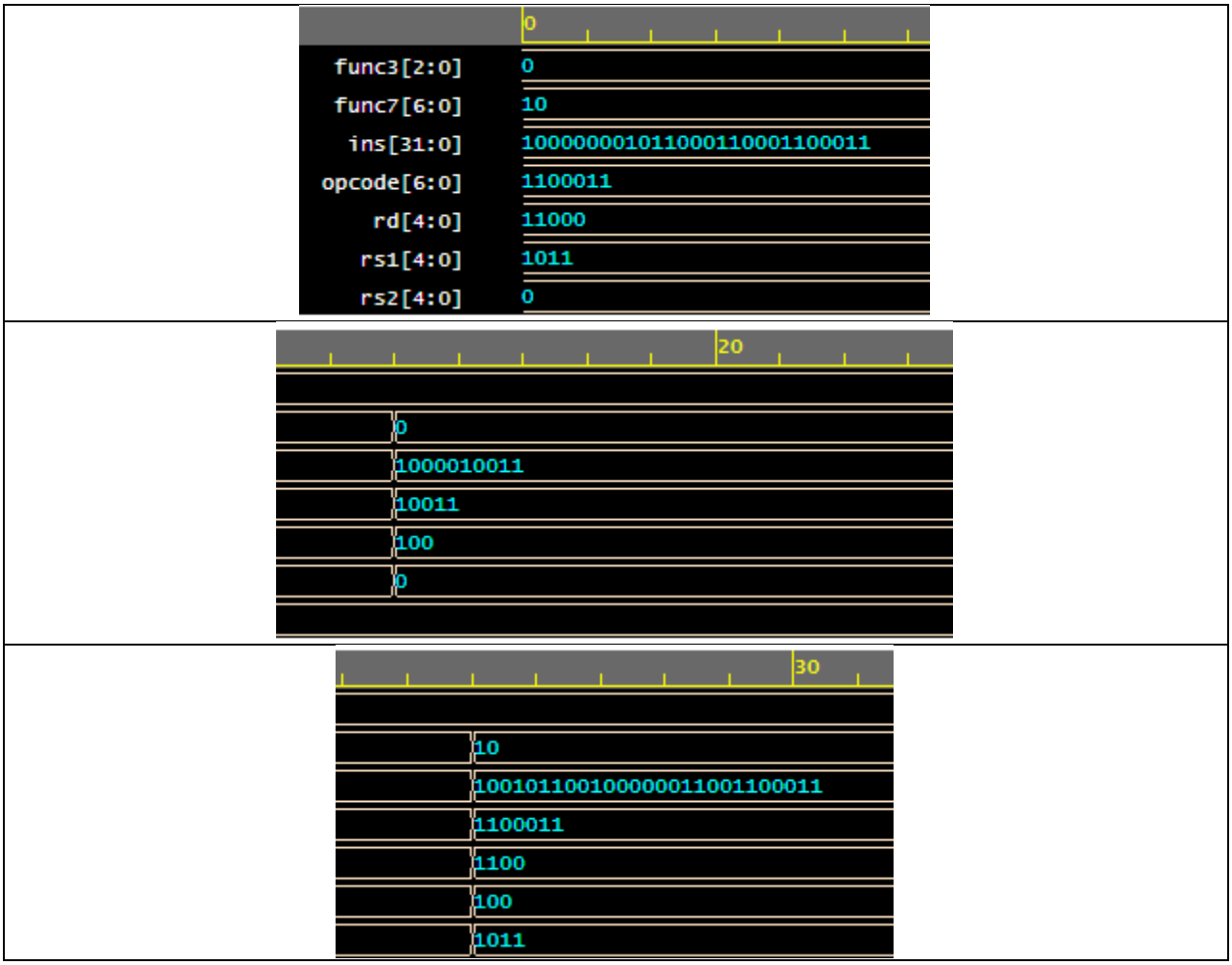


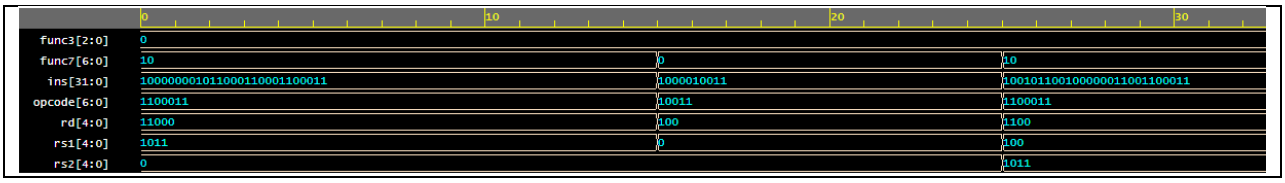
Instruction Parser:

For this module, the input is the instruction which comes from the Instruction Memory. This module breaks the instruction into Opcode, rs1, rs2, func3 and func7 in the following way:

```
module instruction_parser(
  input [31:0] ins,
  output [6:0] opcode, [4:0] rd,[2:0] func3,[4:0] rs1,[4:0] rs2,[6:0] func7
);
  assign opcode = ins[6:0];
  assign rd = ins[11:7];
  assign func3 = ins[14:12];
  assign rs1 = ins[19:15];
  assign rs2 = ins[24:20];
  assign func7 = ins[31:25];

endmodule

```

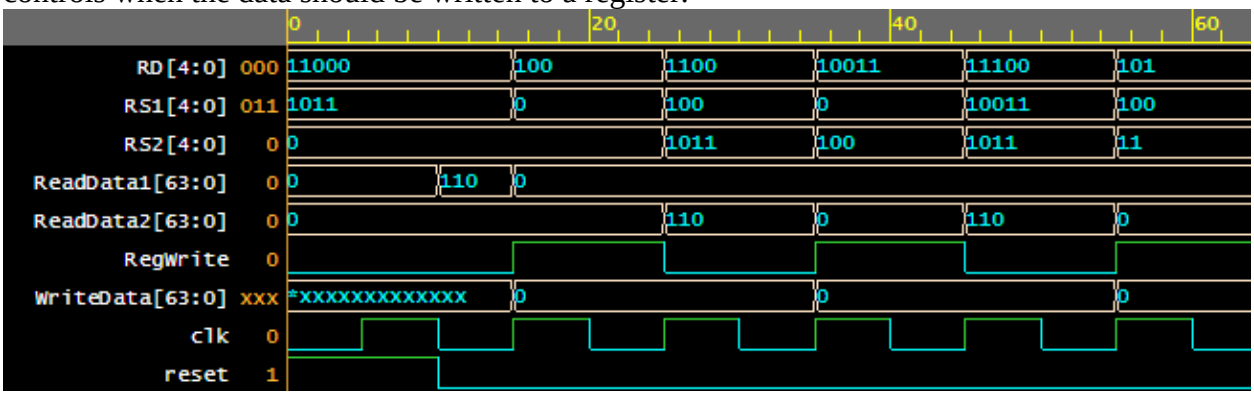
The waveform generated for the module looks like:





Register File:

This module have the input address ports for reading register 1, register 2, and for writing data in a destination register. It have a 64-bits data input port, and two 64-bits data output ports to read the values from two different registers and have a clk, reset, and a RegWrite signal which controls when the data should be written to a register.



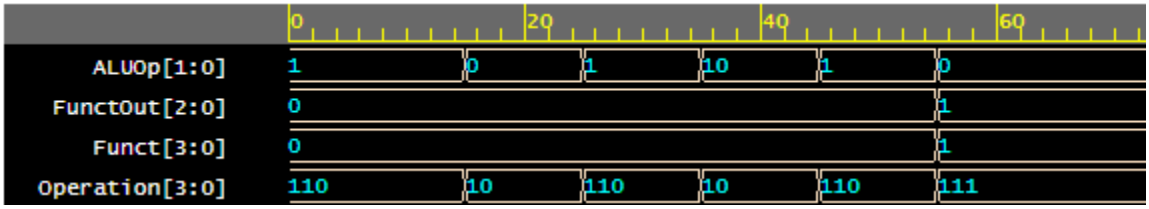
ALU Control:

For this module, the inputs are ALUOp and Funct and the output is Operation. The given pictures is taken from the ALU control shows the operation to be performed according to the inputs of ALUOp and Funct.

```
8   always @(*)
9      begin
10       //performs operation according
11       //to ALUop and funct
12       case(ALUOp )
13         2'b00: // I/S-type
14           case ({Funct[2:0]})
15             3'b001: Operation = 4'b0111; //slli
16             default: Operation = 4'b0010; //add (r-type)etc.
17           endcase
18         2'b01: // sb-type
19           Operation = 4'b0110;    //beq, bne
20
21         2'b10 : // r-type
22           case(Funct)
23             4'b0000:
24              Operation = 4'b0010; //add
25             4'b1000:
26              Operation = 4'b0110; //sub
27             4'b0111:
28              Operation = 4'b0000;
29             4'b0110:
30              Operation = 4'b0001;
31           endcase
32       endcase
```

The example we are using above has 2 beq and 1 addi and we know that beq is SB type and addi is I type instructions. The ALUop code for beq is 1 (01 in binary) where the Funct bits is don't care for both the beq. While the ALUop code for addi is 0 (00 in binary) where the Funct bits is don't care for addi. We can better observe the waveform by looking at our instructions i.e.

- beq x11, x0, FinalExit: (ALUOp: 01, Funct: XXXX hence Operation: beq (0110)
- addi x4, x0, 0: (ALUOp: 00, Funct: XXXX hence Operation: addi (0010)
- beq x4, x11, Exit1: (ALUOp: 01, Funct: XXXX hence Operation: beq (0110)

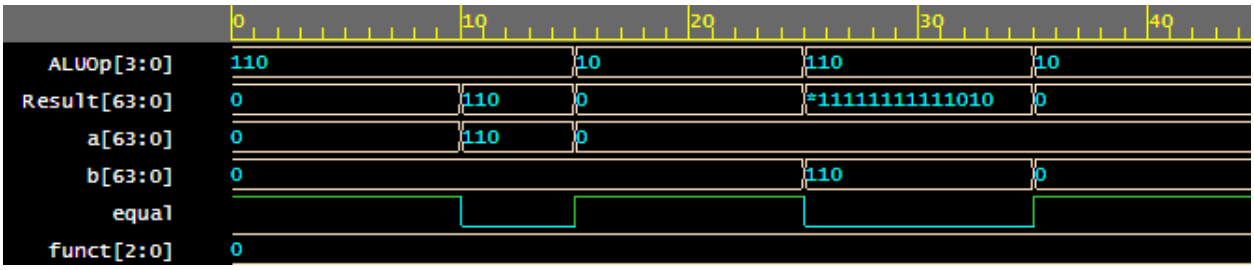


ALU64:

The module has 4 inputs a, b, ALUOp and funct and two outputs Result and equal. Below is the code of ALU64 module.

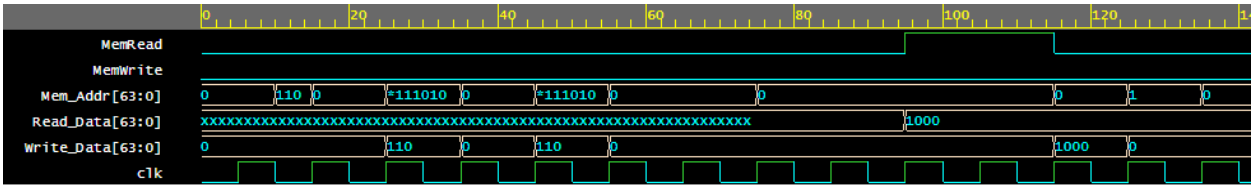
This module generates the result according to the ALUOp input. If ALUOp is 0 (in decimal) then the result will have value of AND operation between A and B, if ALUOp is 1 result will have the value of OR operation between A and B and so on. It further has cases to assign the values to equal. In last it checks that if Result and Funct both are 0 then equal will be set to 1. And if a is greater than equal to b and Funct is 5 in decimal then equal is set to 1 and if both these above cases doesn't imply then the equal is set to zero.

- beq x11, x0, FinalExit: (ALUOp: 110, a=0, b=0, Result = a-b = 0, Funct=0 when result and Funct is zero than equal= 1) (ALUOp: 110, a=110, b=0, Result = a-b = 110, Funct=0 and equal= 0)
- addi x4, x0, 0: (ALUOp: 10, a=0, b=0, Result= a+b=0, Funct=0 when result and Funct is zero then equal = 1)
- beq x4, x11, Exit1: (ALUOp: 110, a=0, b=110, Result= a-b=(-6)(1111111111111010 in binary), Funct=0 therefore equal = 0)

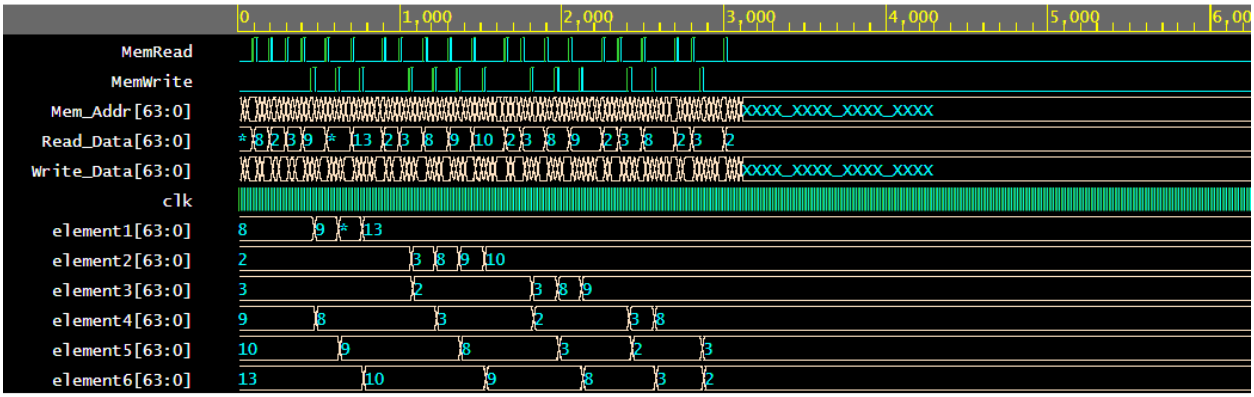


Data Memory:

In this module, the data at the input port, Write_Data, that is written at the positive edge of clk signal and when MemWrite signal is asserted (i.e. high).



Final EP wave of Task 1:



Task 2:

https://edaplayground.com/x/UazY

Objective:

You then proceed to modify the said processor to make it a pipelined one (5 stages). You test and run each instruction separately to verify that the pipelined version can at least execute one instruction correctly in isolation.

Description of modules (other than those discussed above):

- **Forwarding Unit:**

  It is a hardware solution to deal with the data hazards. The idea is to pass proper values early from the pipeline registers to ALU rather than waiting for the WB stage to write in the register file. Basically, it is an optimization in pipelined processors to avoid deficits that occur due to pipeline stalls.

  As per coding it takes following as inputs,

  1. rs1 and rs2 (source registers, 5 bits) from register ID/EX
  2. RegWrite (enables writing to a register) from EX/MEM and MEME/WB
  3. rd (destination register, 5 bits) from EX/MEM AND MEM/WB register

  And following outputs,

  1. reg [1:0] forward_A and forward_B.

  Now, considering these inputs, the module sets forward_A to 2'b00 if there doesn't occur any data dependency i.e. no same register is used in two consecutive instructions.

  Whereas it sets forward_A to 2'b01 if rd from MEM/WB is equal to rs1 from ID/EX (not consecutive registers) and EX/MEM_rd is not equal to ID/EX rs1 (consecutive registers) then we set forward_A to2'b 01. Meanwhile we also see that rd and RegWrite from MEM/WB should also not be 0. Setting forward_A to 2'b01 allows forwarding Mux to transfer MEM_WB_MUX_out to ALU_64_bit.

  Moreover, we assign forward_A with value 2'b10 if rd from EX_MEM is equal to rs1 from ID/EX. Meanwhile we also see that rd and RegWrite from EX/MEM should also not be 0. Here we are checking on consecutive registers (i.e. consecutive instructions). By setting forward_A to 2'b10, we allow forwarding Mux to transfer EX_MEM_ALU_Main (result of previous instruction) to ALU_64_bit.

  Same goes for forward_B, but this time we consider rs2 from ID/EX register and compare it to the rd from EX/MEM register and MEM/WB register.

- **Mux 4 x 1:**

  This multiplexer outputs *data_out* of 64 bits based on selection line. The selection line is of 2 bits that tells which input data should get transferred on the other end.

  Let's discuss the *registers* in the processor that are *IF/ID, ID/EX, EX/MEM and MEM/WB.* The naming convention suggests that each of these register is attached or linked to their previous register or the other way that is upcoming register, like EX/MEM

is linked to the ID/EX. Generally, registers (often called as processor registers) are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU.

- **IF/ID:**

  It takes following inputs from Instruction Memory,

  1. clk, reset,
  2. [31:0] instruction,
  3. [63:0] PC_Out,

  And gives following outputs,

  1. reg [31:0] IF_ID_instruction,
  2. reg [63:0] IF_ID_PC_Out

  This register is designed in a way that it takes inputs, one of 32 bits and other of 64 bits from Instruction memory and save it in registers of 32 bits and 64 bits which are then declared as output to further functional units.

- **ID/EX:**

  The inputs for this register come from IF/ID register after going through different functional units. The inputs are following,

  1. input clk, reset,

  Directly from IF/ID,

  1. input [3:0] IF_ID_instruction,

  From Instruction parser i.e.

  1. input [4:0] IF_ID_rd, IF_ID_rs1, IF_ID_rs2

  From Register File,

  1. input [63:0] IF_ID_ReadData1, IF_ID_ReadData2

  From Immediate Data Generation,

  1. input [63:0] IF_ID_imm_data

  From Program Counter,

  1. input [63:0] IF_ID_PC_Out

  From Control Unit,

  1. input [1:0] IF_ID_ALUOp
  2. input IF_ID_ALUSrc
  3. input IF_ID_BranchEq,
  4. input IF_ID_BranchGt,
  5. input IF_ID_MemRead,
  6. input IF_ID_MemWrite,

7. input IF_ID_RegWrite,
8. input IF_ID_MemtoReg,

And gives following outputs,

1. output reg [3:0] ID_EX_instruction,
2. output reg [4:0] ID_EX_rd, ID_EX_rs2, ID_EX_rs1,
3. output reg [63:0] ID_EX_imm_data, ID_EX_ReadData2,
4. output reg [63:0] ID_EX_ReadData1, ID_EX_PC_Out,
5. output reg ID_EX_ALUSrc,
6. output reg [1:0] ID_EX_ALUOp,
7. output reg ID_EX_BranchEq,
8. output reg ID_EX_BranchGt,
9. output reg ID_EX_MemRead,
10. output reg ID_EX_MemWrite,
11. output reg ID_EX_RegWrite,
12. output reg ID_EX_MemtoReg);

This register is designed in way that it declares all the output registers to 0 id the reset signal is high, else on positive edge of clocks it save all the respective inputs to their respective output registers.

- **EX/MEM:**

  The inputs for this register come from previous register that is ID/EX while going through different functional units installed in between these two registers. Following are the inputs,

  1. input clk, reset

  Directly from ID/EX register,

  1. input [4:0] ID_EX_rd
  2. input ID_EX_zero
  3. input ID_EX_Great
  4. input ID_EX_BranchEq
  5. input ID_EX_BranchG,
  6. input ID_EX_MemRead
  7. input ID_EX_MemWrite
  8. input ID_EX_RegWrite
  9. input ID_EX_MemtoReg

  From 4 x 1 mux,

  1. input [63:0] ID_EX_MUX_FB

  From ALU_64_bit,

  1. ID_EX_ALU

  From adder,

1. input [63:0] ID_EX_Adder

And following are the outputs,

1. output reg [4:0] EX_MEM_Rd,
2. output reg [63:0] EX_MEM_MUX_FB, EX_MEM_ALU,
3. output reg [63:0] EX_MEM_Adder,
4. output reg EX_MEM_zero,
5. output reg EX_MEM_Great,
6. output reg EX_MEM_BranchEq,
7. output reg EX_MEM_BranchGt,
8. output reg EX_MEM_MemRead,
9. output reg EX_MEM_MemWrite,
10. output reg EX_MEM_RegWrite,
11. output reg EX_MEM_MemtoReg);

This register is designed in way that it declares all the output registers to 0 id the reset signal is high, else on positive edge of clocks it save all the respective inputs to their respective output registers.

- **MEM/WB:**

Most of the inputs for this register comes directly from the previous register and some of those reach to MEM/WB while going through Data Memory. Following are the inputs,

1. input clk, reset,

Directly form EX/MEM,

1. input [4:0] EX_MEM_rd
2. input [63:0] EX_MEM_ALU
3. input EX_MEM_RegWrite
4. input EX_MEM_MemtoReg

From Data Memory,

1. input [63:0] EX_MEM_ReadData,

This register is designed in way that it declares all the output registers to 0 id the reset signal is high, else on positive edge of clocks it save all the respective inputs to their respective output registers.

Implementation of RISC V Pipelined processor:

In this we will generally discuss how we have implemented the pipelined version of RISC V Processor. Initially by looking at the circuit implementation provided to us, we figured out the modules we can use from previous labs. Those were following,

- Program_Counter
- Instruction_Memory

- ALU_64_BIT
- register_file
- Control_Unit
- ALU_Control
- adder
- mux_2x1
- immediate_data_gen (refer this as extractor in circuit)

Then comes two more modules and four registers,

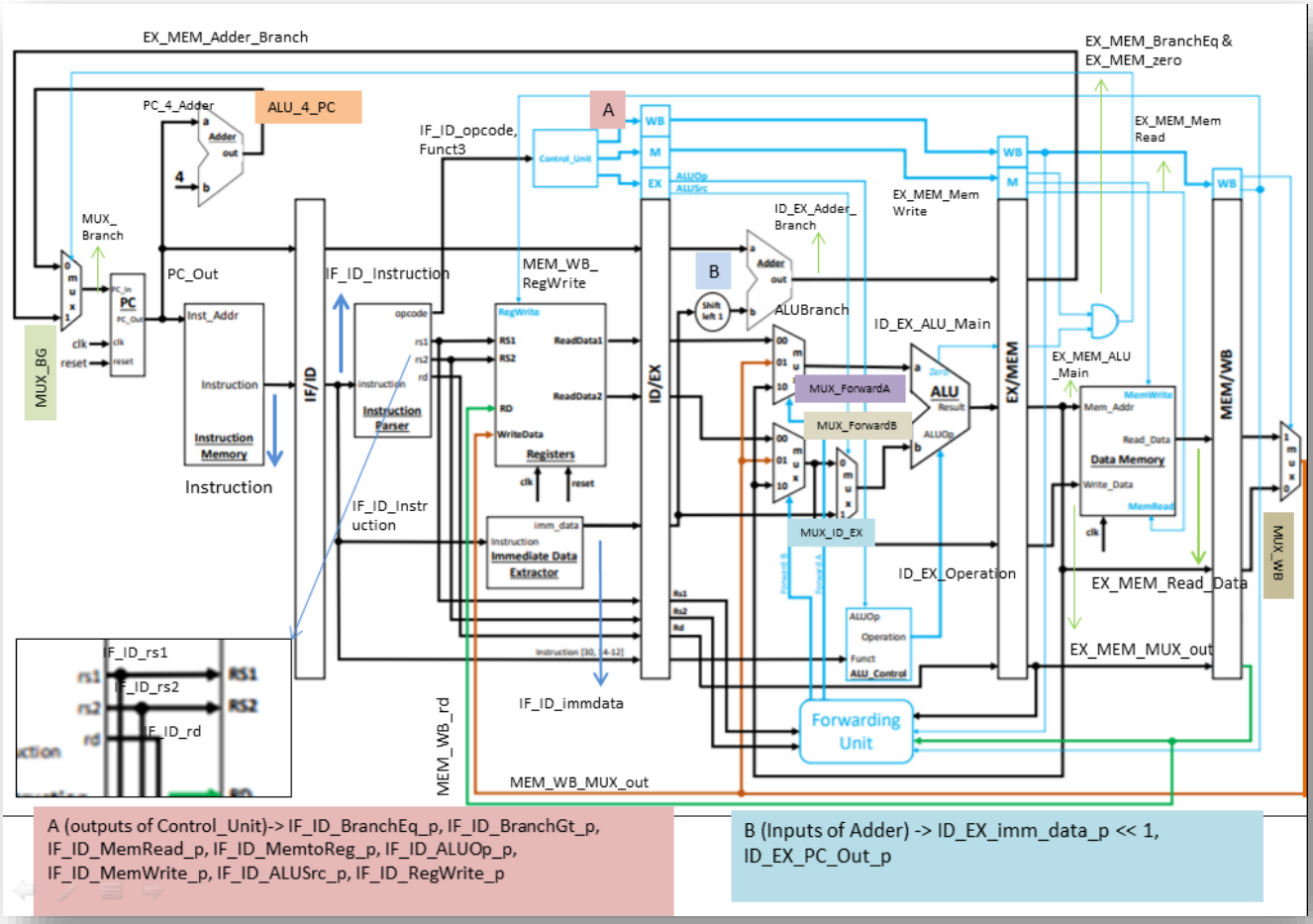
- Forwarding_Unit
- Mux_4x1
- IF/ID
- ID/EX
- EX/MEM
- MEM/WB

All of these modules and registers with their functionalities are described above. These modules are mentioned in separate .v files and are being used in main design.sv where the actual implementation of pipeline processor has been done. At first all of these files are included in design.sv. The clock and reset signal are then provided as inputs for pipeline RISC V Processor main module. As per the provided circuit implementation, we instantiated the modules one by one. At first, the Program counter module is instantiated i.e. the PC_In input is given through a wire and similarly the output is stored in another wire made of same bits as of PC_Out. Considering the figure attached below we can understand how modules are instantiated one after another and how inputs and outputs are transferred via wires declare initially in the main module. The output of Program Counter was then transferred to Instruction_Memory and it was instantiated. Next to it, the instruction from Instruction_Memory was then stored in IF/ID register and then onwards it was given to Instruction_parser module where it got parsed and came out as opcode, rd, func3, rs1, rs2 and func7. At the same time the instruction from IF/ID register was also transferred to Immidiate_data_gen, which outputs imm_data. The transfer of instruction was done via wires that can be seen in the figure attached below.

Now the said outputs from Instruction_parser were then given as inputs to register_file, Control_Unit and ID/EX. Then the outputs from Control_Unit, Register_file and Immidiate_data_gen were then transferred to ID/EX through specific wires initialized in the beginning of code. Moving on to the ouputs of ID/EX mentioned above, these outputs were then transferred to ALU_Control, Forwarding_Unit, Adder and EX/MEM register through wires of specific bits (shown in figure below). Meanwhile some of the outputs were given to respective MUXs and then got transferred to ALU_64_bit. The outputs from Control_Unit that were stored in ID/EX were then transferred to EX/MEM register. The zero signal and result from ALU_64_bit, the output of adder ALU, the destination register from ID/EX and the instruction coming through 4 x 1 MUX were stored in EX/MEM register. Other than this, the outputs of Control_Unit coming through ID/EX were stored in EX/MEM register also. All the inputs required for Data_Memory were given by EX/MEM and its output of ReadData was then stored in MEM/WB register. The stored Control_Unit instructions in EX/MEM were transferred to MEM/WB register. You can see from figure attached below, the output of Adder ALU that was

stored in register EX/MEM was then given as input to MUX_A. The Control_Unit instruction those were stored in MEM/WB resister got transferred to register_file and Forwarding_Unit. Other ReadData instruction, MemtoReg and ALU instruction were given to MUX_D whose output was then given to register_file as its input for WriteData. The output of MUX_A was then given to Program Counter whose output was then transferred to Adder_p. The other input for Adder_p is 64'd4. The ouput of Adder_p was then again given back to MUX_A as its input.

In order to understand out implementation more specifically, refer to the figure attached below.



Instruction wise Waveform Description/Explanation:

This is the set of instructions which we are taking as an example to explain our implemented pipelined processor:

- add x19, x0, x1 (in binary: 0000000|00001|00000|000|10011|0110011)
- sub x12. x19, x3 (in binary: 0100000|00011|10011|000|01100|0110011)
- add x13, x5, x12 (in binary: 0000000|01100|00101|000|01101|0110011)

Instruction Memory:

For this module, the input is just the instruction address (which is coming as an output of program counter) and the output is the instruction.

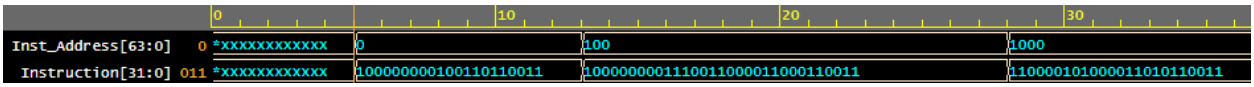
Here the instruction is generated using:

```
always @(Inst_Address)
  begin
    assign Instruction = {registers[Inst_Address+3], registers[Inst_Address+2],
     registers[Inst_Address+1], registers[Inst_Address]};
  end
```

Which means that when the instruction address is 0 then registers [3], registers [2], registers [1] and registers [0] will form the first instruction. (add x19, x0, x1)

Similarly when instruction address is 4 (100 in binary), then registers [7], registers [6], registers [5] and registers [4] will form the second instruction (sub x12. x19, x3) and for instruction address 8 (1000 in binary) then registers [11], registers [10], registers [9] and registers [8] will form the third instruction (add x13, x5, x12)

This exact result is shown in the waveform below:

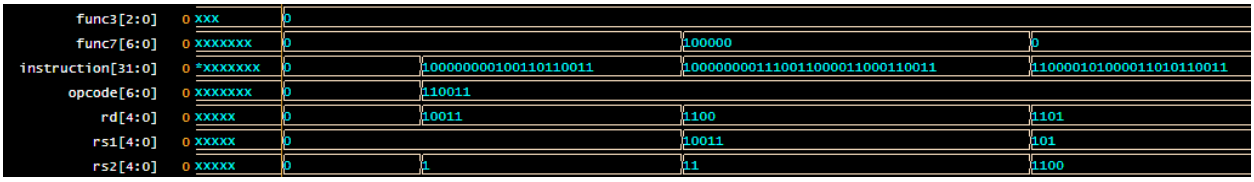


Instruction Parser:

For this module, the input is the instruction which comes from the Instruction Memory. This module breaks the instruction into opcode, rs1, rs2, func3 and func7 in the following way:

```
opcode = instruction[6:0];
rd = instruction[11:7];
func3 = instruction[14:12];
rs1 = instruction[19:15];
rs2 = instruction[24:20];
func7 = instruction[31:25];
```

The waveform generated for the module looks like:

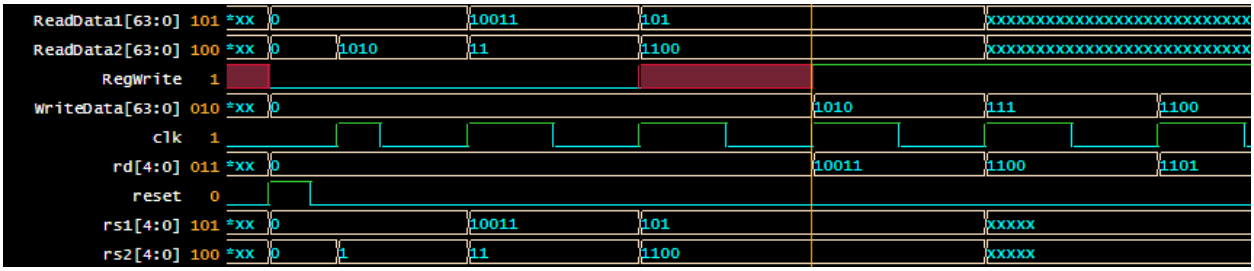


Register File:

For this module, the inputs are clk, reset, WriteData, rs1, rs2, rd, RegWrite, and the outputs are ReadData1, ReadData2. Here the WriteData is basically the values that we have to write in the registers as the solution of the instructions. Whereas ReadData1 and ReadData2 will take on the values of rs1 and rs2 if and only if RegWrite is 1.

The given waveform shows that since RegWrite, for R type instructions is 1, hence ReadData1 = rs1 and ReadData2 = rs2. For WriteData, the values as shown in waveform below are 10, 7 and 12 because:

- add x19, x0, x1: Here x0 = 0 and x1 = 10 i.e. 0+10 = 10 which is stored in x19
- sub x12. x19, x3: Here x19 = 10 and x3 = 3 i.e. 10-3 = 7 which is stored in x12
- add x13, x5, x12: Here x5 = 5 and x12 = 7 i.e. 6+7 = 12 which is stored in x13



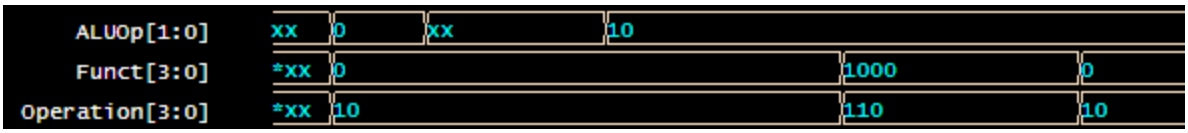
ALU Control:

For this module, the inputs are ALUOp and Funct and the output is Operation. The given pictures is taken from the pdf of lab 10 which shows the operation to be performed according to the inputs of ALUOp and Funct.

Instruction Type	ALUOp [1:0]	Funct	Operation
I/S-Type (ld, sd)	00	xxxx	0010
SB-Type (Beq)	01	xxxx	0110
R-Type	10	0000	0010 add
		1000	0110 sub
		0111	0000
		0110	0001

The table above shows that the ALUOp code for both, subtraction and addition (as we have taken only these two R type instructions in our example) is 2 (10 in binary). Whereas the Funct bits is 0 for addition and 8 (1000) for subtraction. We can better observe the waveform by looking at our instructions i.e.

- add x19, x0, x1: ALUOp: 10, Funct: 0000 hence Operation: add (0010)
- sub x12. x19, x3: ALUOp: 10, Funct: 1000 hence Operation: sub (0110)
- add x13, x5, x12: ALUOp: 10, Funct: 0000 hence Operation: add (0010)



ALU 64-bit:

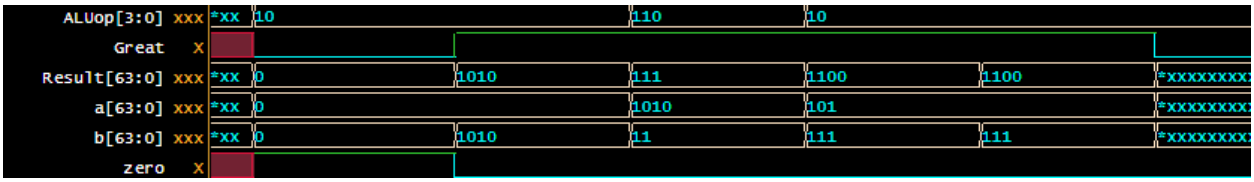
For this module, the inputs are a, b (coming from the forwarding multiplexers) and ALUOp (coming from ALU Control module). The outputs of this module are Result, zero and Great. Based on ALUOp, the arithmetic operation is performed on a and b, which is given the name of Result. Considering the Result, we set Great and zero to either 1 or 0. In our case it can be seen that zero has always been set to 0 whereas Great is always 1.

As given in the table below, we can see that the ALUOp for add instruction is 0010 and for subtract instruction it is 0110, it is observant that the first operation performed is add, then subtract and then add again.

[3:0] ALUOp				Function
Ainvert	Binvert	Operation [1]	Operation[0]	
0	0	0	0	AND
0	0	0	1	OR
0	0	1	0	Add
0	1	1	0	Subtract
1	1	0	0	NOR

The instructions that we took as an example will generate result as written below:

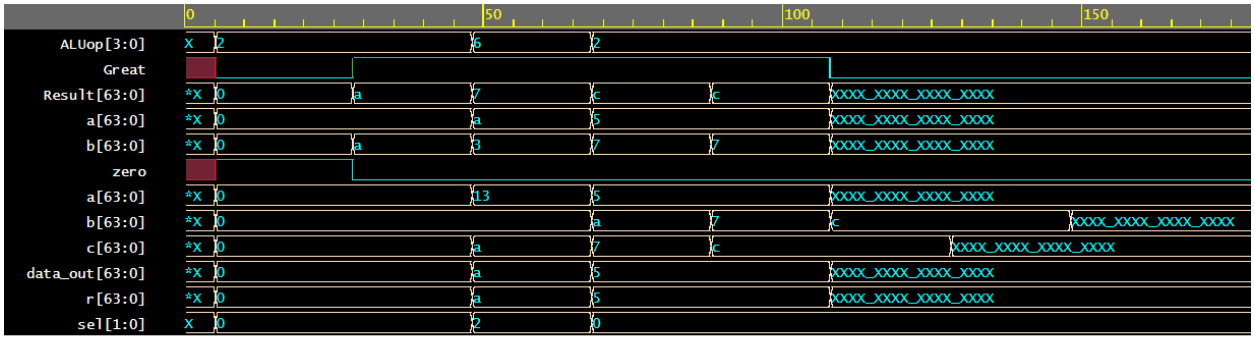
- add x19, x0, x1: ALUOp: 10, Result: a + b = 0 + 1010 = 1010 (10 in decimal)
- sub x12. x19, x3: ALUOp: 10, Result: a – b = 1010 - 0011 = 0111 (7 in decimal)
- add x13, x5, x12: ALUOp: 10, Result: a + b = 0111 + 0101 = 1100 (12 in decimal)



Data Memory:

For this module, the inputs are MemWrite, MemRead, MemAdd, WriteData and the only output is ReadData1. From the circuit implementation we can see that we are having correct inputs for WriteData i.e. values stored in rs2 registers which are coming via the forwarding mux's. Since we know that data memory handles the functionalities for writing and reading from memory, then working on R-type instructions and having the said signals as 0 is obvious (MemWrite, MemRead). Whereas MemAdd is equal to the Result coming in from ALU 64-bit (stored in EX/MEM). ReadData is 'don't care' because nothing has been written in memory.

Final EP wave of Task 2:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

TASK 3:

https://edaplayground.com/x/VEvp

Objective:

You introduce circuitry to detect hazards (data, control, and structural if needed) and try to handle them in hardware i.e. by stalling, and flushing the pipeline.

Modules other than those discussed above:

- Hazard Detection Unit:

  This module has following inputs,

  - rs1 and rs2 from IF/ID
  - rd from ID/ED
  - ID_EX_MemRead

  And following outputs,

  - hazard_MUX_SEL
  - IF_ID_write,
  - PC_write

  We are performing stalling in this module. For this, we set all above mentioned outputs to 0 if rd from ID/EX is same as rs1 from IF/ID rs1 or rs2 otherwise all outputs are set high. But declaring them 0 we basically stall them that are holding upcoming instructions. Further we will see how these outputs are significant in implementation.

Implementation of RISC V Pipelined processor (with stalling and flushing):

This task is basically the extension of task 02 (for other details you may look in to description given above). In task 02 we handled the data hazard via forwarding. Whereas, in task 03 we are required to handle control and structural hazard using flushing and stalling.

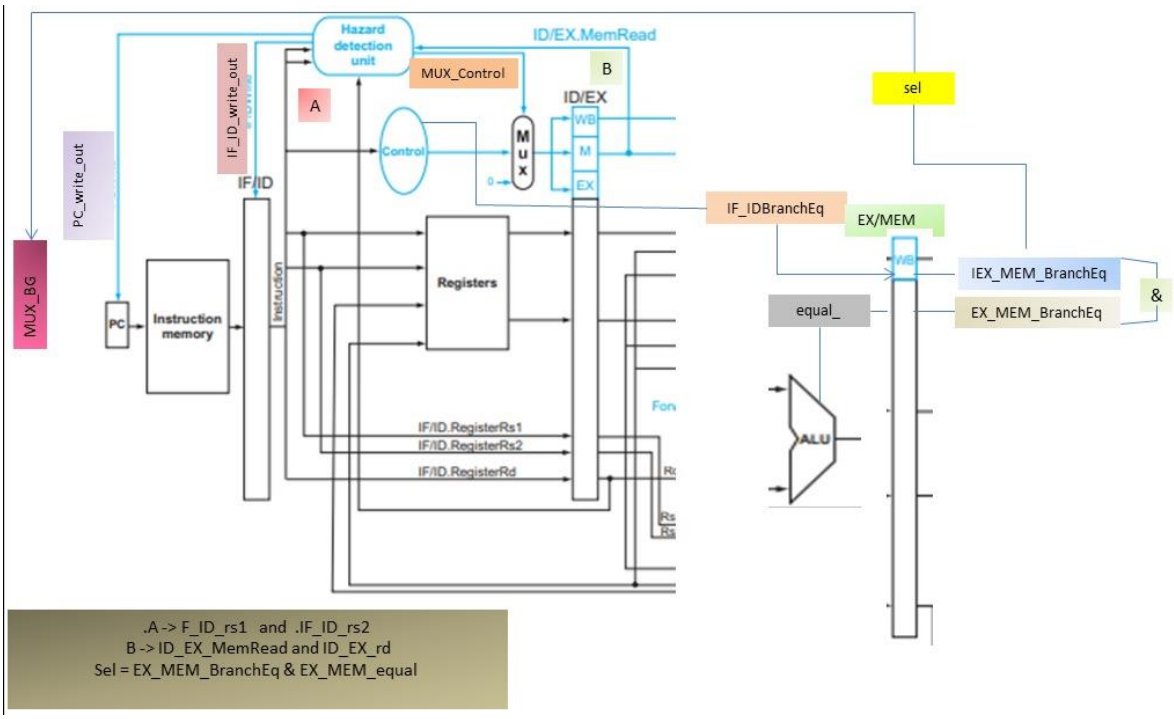
We have used the same circuit as that for task 02 to implement task 03, but this time we have added an extra module named as Hazard Detection Unit to implement stalling. Moreover, for every pipelined register we have used input signal of flush t unlike in task02. Initially the same logic of implementation that is defined in task 02 description in implemented. Next to it we created a new .v file of name Hazard_Detection_Unit. The description of module with in this file is explained above, this file is then imported in design.sv and we instantiated it with appropriate inputs and outputs. Refer to the picture attached below to understand how

connections are made. As per inputs and if conditions provided it gives 3 outputs. All of them either come out to be 1'b1 or 1'b0. We stall the pipeline by setting outputs of the said module to 1'b0 for which we say that we have stalled the pipeline. Basically, it stalls the instruction in the fetch stage, to prevent the instruction in that stage from being overwritten by the next instruction in the program.

We can see in the figure attached below that the hazard_MUX_sel is provided as selection line to MUX_Control which holds all the control unit's outputs whenever hazard is detected and hazard detection signal raises. Otherwise the regulation of signals goes smoothly. Then comes output PC_write from hazard detection unit. This output is sent to Program Counter as its input. By setting it to 1 we allow the program counter to jump on consecutive next instruction but by setting it to 1'b0 we basically place a hold on PC_Out. Now comes the last output of hazard detection unit that is IF_ID write. Whenever IF_ID write comes out to be 1'b1, the incoming instruction passes out as its ouput otherwise output instruction is set as 0.

When it comes to branching, for branching we know that pipeline already calls in consective instructions to branch instruction until it detected that whether branching is need to be done or not. For this purpose we have introduced a signal named as flushing in this task. We set flush to 1'b1 whenever the sel signal comes out to be high. Note that wire sel in design.sv basically performs AND operation on branch signal coming from Control Unit and equal signal coming from ALU_64 bit. Now after detecting signal for sel, we give this signal as input to registers except MEM/WB in place of signal flushing. Now in the modules defined for these registers, we set all the outputs to 1'b0 if flushing is equal to 1'b1 otherwise all the inputs are transferred as outputs smoothly. This is due to our desire of flushing the unwanted instruction that has already entered in pipeline.

This is how we are performing branching and hazard detection. Other than this, the same implementation as in task 02 has been done here.


Hazard detection unit
ID/EX.MemRead
MUX_Control
B
A
ID/EX
WB
M
EX
Control
M u x
0
IF_ID_write_out
PC_write_out
IF/ID
sel
IF_IDBranchEq
EX/MEM
WB
IEX_MEM_BranchEq
EX_MEM_BranchEq
&
equal_
MUX_BG
PC
Instruction memory
Instruction
Registers
ALU
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd
.A -> F_ID_rs1 and .IF_ID_rs2
B -> ID_EX_MemRead and ID_EX_rd
Sel = EX_MEM_BranchEq & EX_MEM_equal

References

[1] https://www.cs.fsu.edu/~hawkes/cda3101lects/chap5/script.html

[2] https://www.electronicshub.org/multiplexerandmultiplexing/

[3] https://www.britannica.com/science/computer-science/Architecture-and-organization#ref671770