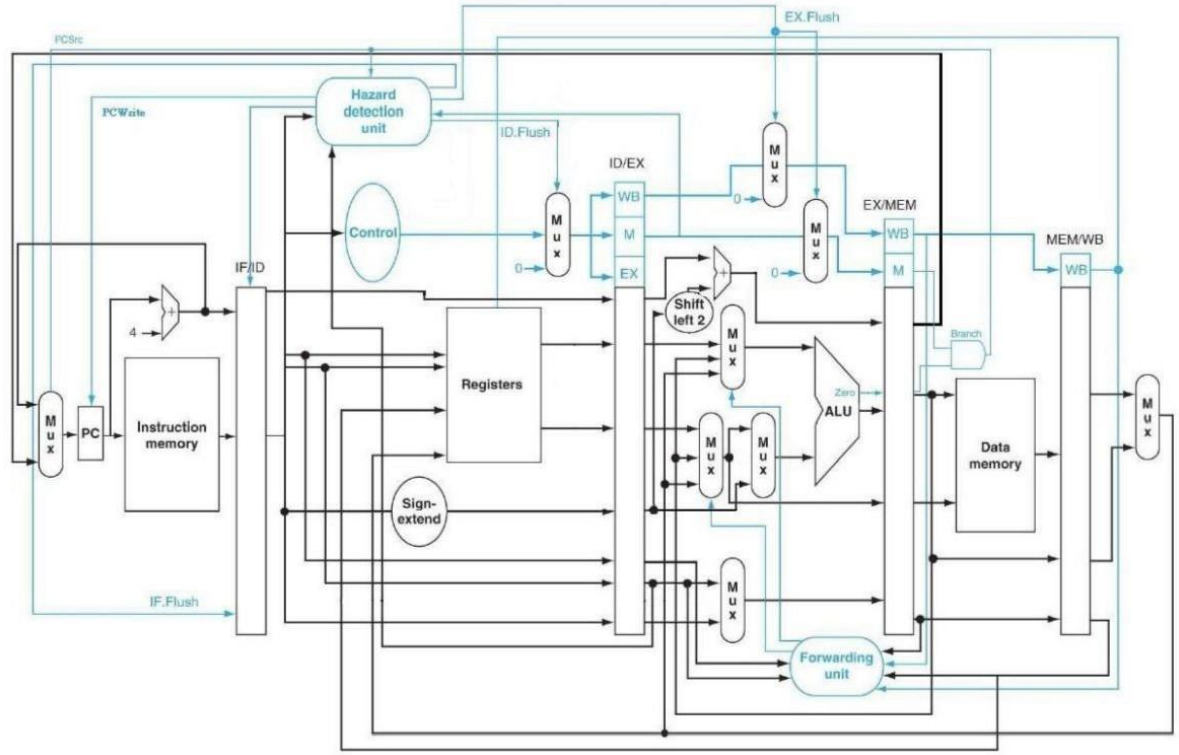
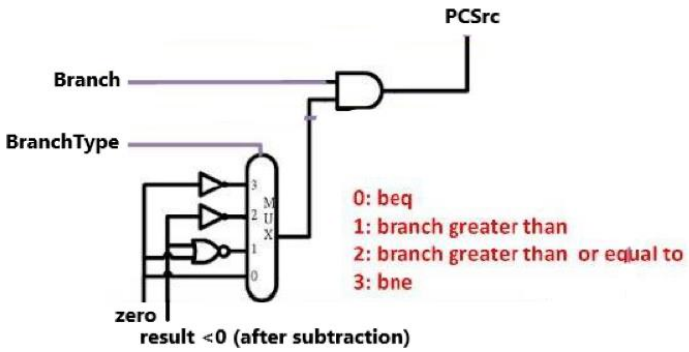
Computer Organization CO Lab 4

Architecture diagrams:



Path of Branch (similar architecture but in different order):

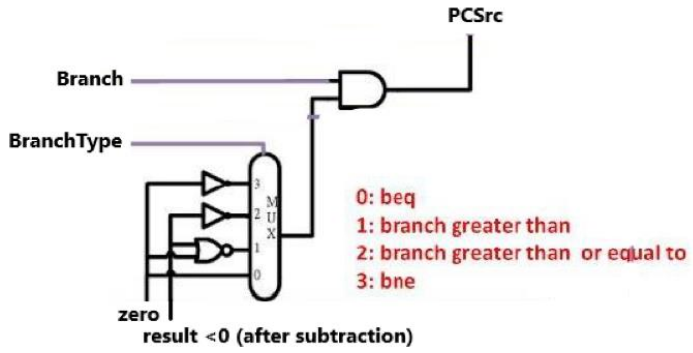
- **0: beq, 1: bne, 2: bge, 3: bgt**



Hardware module analysis:

- **Top module: Advanced_Pipelined_CPU**
- (previous description in Colab 4) There is mainly 5 stages of pipelining:
    - Instruction Fetch (IF)

    - Program Counter: “count” for the next instruction that needs to do in the following clock cycle.
    - Adder: PC+4
    - Instruction memory: fetch the instructions from the text file according to the Program Counter.
- Instruction Decode (ID)
    - Decoder: decode the opcode of the instruction and output to other modules for use.
    - Register File: performs the functions of getting the value from the register according to the instructions and storing the result of ALU into the register specified by the current instruction.
- Execution (EX)
    - ALU Control: further decode the function code of the instruction and output to ALU for use.
    - Adder: add the value of Program Counter for the next instruction with the address stated by instruction[15:0], when is the instruction specified by branch operation.
    - Sign Extend: extend the 16-bit address stored in instruction into 32-bit.
    - multiplexor: for selecting the value of a register or the value extending the last 15 address of instruction according to the ALUSrc
    - ALU: perform some operations such as *and*, *or*, *addition*, *subtraction*, *set on less than*, *nor* and *multiplication(simplified version)*
- Data Access (MEM)
    - Data Memory: Read or Write the data from memory according to MemRead/ MemWrite.
    - **Branch combinational circuit: a combinational circuit that decides that branch taken or not when a branch instruction including beq, *bne, bge,* and *bgt* instructions is given.**
    - **similar architecture diagram but in different order:**



- **0: beq, 1: bne, 2: bge, 3: bgt**

```
MUX_4to1 #(.size(1)) MuxBranch(
    .data0_i(MEM_zero),                          //beq
    .data1_i(~MEM_zero),                         //bne
    .data2_i(MEM_zero | (MEM_ALU_result < 0)),   //bge
    .data3_i(MEM_ALU_result < 0),                //bgt
    .select_i(MEM_BranchType),
    .data_o(MEM_Branch_final)
);
```

```
assign IF_PCSrc = MEM_Branch & MEM_Branch_final;
```

- Write Back (WB)
    - multiplexor: for selecting the input from Data Memory or the result of ALU according to MemtoReg (simple modification: select the value from Data Memory when MemtoReg is 1, otherwise select the result of ALU).

- For each pipelined stage(exclude WB stage), there is a pipeline register storing the output including the control signals that need to pass to the next stage.
    - IF/ID
    - ID/EX
    - EX/MEM
    - MEM/WB
    - **Modification of Pipeline register: added a parameter *ctrl_size* to state the size of all of the control signals, *valid_i* to decide whether the current signal will be prevented to be updated, and *flush_i* to flush all the control signals.**
- **For Forwarding, there is a specialized Forward Unit to deal with the data hazard.**
    - **EX hazard: ForwardA/B = 2, The data required from the EX/MEM register is forward to the current EX stage**

- **MEM hazard: forwardA/B = 1, The data required from the MEM/WB register is forward to the current EX stage**

```
//EX hazard
if ((EX_MEM_RegWrite_i & (EX_MEM_RegisterRd_i !=0)
   & (EX_MEM_RegisterRd_i == ID_EX_RegisterRs_i)))
    ForwardA_o <= 2'b10;
//MEM hazard
else if(MEM_WB_RegWrite_i & (MEM_WB_RegisterRd_i !=0)
    & !(EX_MEM_RegWrite_i & (EX_MEM_RegisterRd_i !=0)
       & (EX_MEM_RegisterRd_i == ID_EX_RegisterRs_i))
    & (MEM_WB_RegisterRd_i == ID_EX_RegisterRs_i))
    ForwardA_o <=2'b01;
else
    ForwardA_o <= 2'b00;

//EX hazard
if( EX_MEM_RegWrite_i & (EX_MEM_RegisterRd_i !=0)
   & (EX_MEM_RegisterRd_i == ID_EX_RegisterRt_i))
    ForwardB_o <= 2'b10;
//MEM hazard
else if(MEM_WB_RegWrite_i & (MEM_WB_RegisterRd_i !=0)
   & !(EX_MEM_RegWrite_i & (EX_MEM_RegisterRd_i !=0)
       & (EX_MEM_RegisterRd_i == ID_EX_RegisterRt_i))
   & (MEM_WB_RegisterRd_i == ID_EX_RegisterRt_i))
    ForwardB_o <= 2'b01;
else
    ForwardB_o <= 2'b00;
```

- **To deal with load hazard and control hazard (branch hazard), a Hazard Unit is designed to deal with them:**
  - **load hazard: to stall the pipeline for one cycle, PCWrite_o is used to prevent PC update, IF_IDWrite is used to prevent IF_ID update, and Control is to flush the control signal of ID_EX pipeline register.**

```
always @(*)begin
    if(ID_EX_MemRead_i &((ID_EX_RegisterRt_i == IF_ID_RegisterRs_i)|
       (ID_EX_RegisterRt_i==IF_ID_RegisterRt_i)))
    begin
        PCWrite_o <= 0;
        IF_IDWrite_o <= 0;
        Control_o <= 0;
    end
    else begin
        PCWrite_o <= 1;
        IF_IDWrite_o <= 1;
        Control_o <= 1;
    end
end
```

  - **control hazard: to flush the control signal of the pipeline register, a signal Flush_o is used**

```
always @(*)begin
    if(Branch_i)
        Flush_o <= 1;
    else
        Flush_o <= 0;
end
```

Problems You Met and Solutions

1. **After finish the load hazard detection unit, there is some unusual error that the Forward Unit forward the data from the wrong place, but finally it was just my careless mistake that I take ID/EX.RegisterRd as the first input, and so on.**
    a. **Solution: correct it to EX/MEM.RegisterRd**

Result:

test data 1:

```
Register=======================================================

r0=    0, r1=   16, r2=  256, r3=    8, r4=   16, r5=    8, r6=   24, r7=   26
r8=    8, r9=    1, r10=   0, r11=   0, r12=   0, r13=   0, r14=   0, r15=    0
r16=   0, r17=   0, r18=   0, r19=   0, r20=   0, r21=   0, r22=   0, r23=    0
r24=   0, r25=   0, r26=   0, r27=   0, r28=   0, r29=   0, r30=   0, r31=    0

Memory=====================================================

m0=    0, m1=   16, m2=   0, m3=   0, m4=   0, m5=   0, m6=   0, m7=    0
m8=    0, m9=    0, m10=   0, m11=   0, m12=   0, m13=   0, m14=   0, m15=    0
r16=   0, m17=   0, m18=   0, m19=   0, m20=   0, m21=   0, m22=   0, m23=    0
m24=   0, m25=   0, m26=   0, m27=   0, m28=   0, m29=   0, m30=   0, m31=    0
```

Summary:

The Advanced Pipelined CPU is a Pipelined CPU that can do basic MIPS instructions (without j type) including pseudo-instructions such as *bne, bge* and *bgt* that can solve the data and control hazards well.