# VE370 Project 2 Group Report

## ALU Related Modules

This part describes the ALU related modules in the pipelined processor, including the ALUop signal, the ALU control unit and the ALU unit. The processor first produces a 2-bit ALUop signal by the control unit. Then the ALUop signal is processed by the ALU control unit which will output a 4-bit ALU control signal. Finally the ALU unit executes the computation according to the ALU control signal.

### ALUop Signal

ALUop signal is a 2-bit signal, produced by the control unit according to different instruction. ALUop signal is one of the inputs of the ALU control unit, which determines the type of computation of the ALU unit. How ALUop is designed is listed in the table below, together with the output result of the ALU control unit. R-type instructions share the same ALUop signal 10, to further identify different R-type instructions funct code need to be checked later. lw, sw and addi share ALUop of 00. beq and bne share ALUop of 01. andi produces ALUop of 11.

### ALU Control Unit

ALU control unit takes two inputs, ALUop and funct, and outputs an 4-bit ALU control signal. The design is in the table below.

| Op code | Operation | ALUop | Funct | ALU control | ALU function |
|---|---|---|---|---|---|
| lw (100011) | load word | 00 | xxxxxx | 0010 | add |
| sw (101011) | save word | | | | |
| addi (001000) | add immediate | | | | |
| beq (000100) | branch if equal | 01 | xxxxxx | 0110 | subtract |
| bne (000101) | branch if not equal | | | | |
| andi (001100) | and immediate | 11 | xxxxxx | 0000 | AND |
| R-type (000000) | add | 10 | 100000 | 0010 | add |
| | sub | | 100010 | 0110 | subtract |
| | AND | | 100100 | 0000 | AND |
| | OR | | 100101 | 0001 | OR |
| | slt | | 101010 | 0111 | slt |
| j | jump | J-type instruction never uses any ALU related modules. | | | |

In this project, 12 instructions are to be implemented, including lw, sw, beq, bne, add, sub, AND, OR, slt, addi, andi, j. Among these, j (jump) never uses ALU so its ALU control code doesn't matter, by default set as 000000. The funct code of all the I-type instructions doesn't matter, marked as xxxxxx in the table. For the R-type instructions, different ALU control signals are determined by the funct code.

lw, sw, addi, add require the ALU unit to do add function, so they share an ALU control code of 0010. Likewise, beq, bne, sub require subtract function, output 0110. andi, AND require AND function, output 0000. OR outputs 0001, indicating OR function. slt outputs 0111, indicating slt function.

### ALU Unit

The ALU unit takes the ALU control signal from the previous ALU control unit, to determine the type of computation.

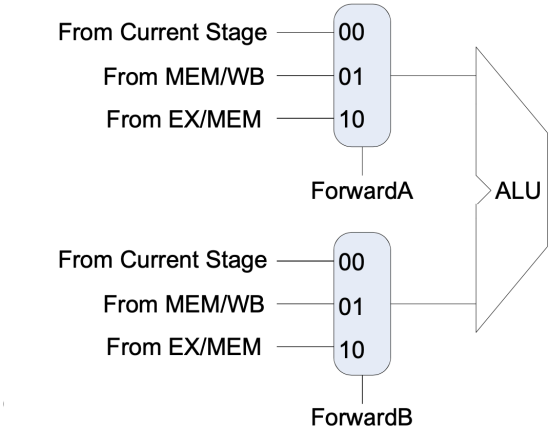
| ALU control | ALU function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | slt |

## Data Hazard

This part describes how data hazards in this pipelined processor are solved. The data hazards that will occur can be categorized into three kinds, R-type data hazard, data hazard on branch, and data hazard on lw.

### Data Hazard on R-type instructions

Data hazard caused by R-type instructions is the simplest and can be solved by a forwarding unit, see the figure below. The unit controls the inputs of the ALU unit to make sure the value of the register being used is updated, if the previous one or two instruction changes the value of Rd (Rd is not $0), and the previous Rd is the Rs or Rt of the current instruction.



The logic of the forwarding unit is as follows:

Initially forwardA = 0, forwardB = 0.

For 1&2 hazard:

If (regWrite.MEM && regRdMEM && (regRdMEM == regRsEX))

forwardA = 2'b10;

If (regWrite.MEM && regRdMEM && (regRdMEM == regRtEX))

forwardB = 2'b10;

For 1&3 hazard (notice that we need to make sure there is no 1&2 hazard):

If (regWrite.WB && regRdWB && (regRdWB == regRsEX)

&& forwardA != 2'b10)

forwardA = 2'b01;

If (regWrite.WB && regRdWB && (regRdWB == regRtEX)

&& forwardB != 2'b10)

forwardB = 2'b01;

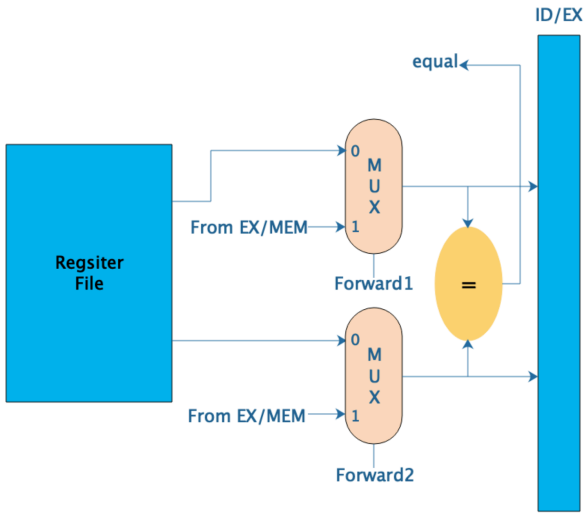
This unit is named as `R_forward` in Appendix.

**Data Hazard on Branch**

Since the branch instruction result will be produced by the comparator early in the ID stage, the forwarding unit is not enough to solve the data hazard caused by beq and bne. Here 5 more signals are introduced, forward1, forward2, PCWrite, IFID_write, and ID_flush.

Initially forward1 = 0, forward2 = 0, PCWrite = 1, IFID_write = 1, and ID_flush = 0.

Amongst, forward1 and forward2 are two select signals of muxes, as shown below. If the instruction in the MEM stage writes back to Rd (Rd is not $0), the instruction in ID stage is beq/bne, and Rd of the MEM stage is Rs/Rt in the ID stage, then forward1/forward2 = 1.



Specifically:

```
if (EXMEM_regWrite && !EXMEM_memRead && ID_branch) begin
    if (EXMEM_regRd == ID_regRs) forward1 = 1;
    if (EXMEM_regRd == ID_regRt) forward2 = 1;
end
```

PCWrite, IFID_write, and ID_flush are used to insert stalls. If the EX stage instruction before beq/bne (currently in the ID stage) writes back to Rd (Rd != $0), and Rd of the EX stage is Rs/Rt in the ID stage, then insert one stall, making PCWrite = 0, IFID_write = 0 and ID_flush = 1.

Else if the MEM stage instruction reads data from the data memory, meaning it will write back to Rd (Rd != $0) in next clock cycle; and the ID stage instruction is beq/bne, and Rd of the MEM stage is Rs/Rt in the ID stage, insert one stall as well.

Specifically:

```
if (IDEX_regWrite && ID_branch) begin
    if (EX_regDst == ID_regRs || EX_regDst == ID_regRt) begin
        PCWrite = 0; IFID_write = 0; ID_flush = 1;
    end
end

if (EXMEM_memRead && ID_branch) begin
    if (EXMEM_regRd == ID_regRs || EXMEM_regRd == ID_regRt) begin
        PCWrite = 0; IFID_write = 0; ID_flush = 1;
    end
end
```

This unit is named as brc_hazard in Appendix.

**Data Hazard on lw**

The third kind of data hazard is caused by the lw instruction. Different from R-type instructions, the earliest time the lw instruction produces its result is the MEM stage. While forwarding back in time is impossible, a stall after the lw instruction is necessary. Again we use three signals to control stalls, PCWrite, IFID˙write, and ID_flush.

Initially PCWrite = 1, IFID˙write = 1, and ID_flush = 0.

If the EX stage instruction is lw, and Rd (Rd != $0) of lw is Rs/Rt of the ID stage instruction, insert a stall.

Specifically:

```
if (IDEX_memRead && (IFID_regRs == IDEX_regRt || IFID_regRt == IDEX_regRt)) begin
    PCWrite     = 0;
    ID_flush    = 1;
    IFID_write  = 0;
end
```

This unit is named as lw_hazard in Appendix.