B, J

Lab 6 – Flow Control Instructions

The branch and jump instructions are used to transfer the flow of control from one part of the instruction memory to another part. The branch instructions transfer control based upon the comparison of the two register operands and if the condition is true then the control will be transferred to another part of the instruction. The jump instructions provide unconditional control transfer to the target. To implement these operations we are required to modify our datapath in such a way that we can use our ALU to perform the calculation of the address of the target to which the control will be transferred. This will be performed by adding the immediate value to the current value of the program counter. The immediate value will be calculated during the decode stage and the calculation of the address along with the comparison for the branch operation will be performed in the execute stage.

Branch/ Jump Operation – Fetch Phase

The value of PC for the next instruction will be changed in case the branch condition becomes true or the jump instruction is executed. Listing 6.1 demonstrates that the PC value is updated with the result of the ALU when the branch or jump is taken.

```
assign pc_next  = exe2if_fb.jump_br_taken ? exe2if_fb.alu_pc
                : imem_update_addr        ? (pc_ff + 32'd4)
                : pc_ff;
```

Listing 6.1. PC value updated for Branch/Jump operation.

Branch/ Jump Operation – Decode Phase

The decode operation for branch instructions uses the **func3** bit-field of the instruction machine code to select the operands, comparison operation to be performed, ALU operation to be performed, request for branch instruction and immediate value which are going to be used inside the execution stage. However, these selections for jump instructions are based on the opcode of the instruction. There are two jump instructions which include the jump and link (jal) instruction and the jump and link register (jalr) instructions. Recall that branch instructions are B-type while jump operations follow J-type encoding format. The target address for branch and jal operations is constructed using pc-offset addressing. However, for the jalr instruction register-offset addressing is used. The code listings implementing the load and store operations at instruction decode phase are given in Listing 6.2 and 6.3, respectively.

```
OPCODE_BRANCH_INST : begin
   id2exe_ctrl.alu_opr1_sel     = ALU_OPR1_PC;
   id2exe_ctrl.alu_opr2_sel     = ALU_OPR2_IMM;
   id2exe_ctrl.alu_cmp_opr2_sel = ALU_CMP_OPR2_REG;
   id2exe_ctrl.alu_ops          = ALU_OPS_ADD;
   id2exe_ctrl.branch_req       = 1'b1;
   id2exe_data.imm              = {{20{instr_codeword[31]}}, instr_codeword[7],
                                  instr_codeword[30:25], instr_codeword[11:8], 1'b0};

   case (funct3_opcode)
       3'b000  : id2exe_ctrl.branch_ops = BR_EQ;      // Branch equal
       3'b001  : id2exe_ctrl.branch_ops = BR_NE;      // Branch not equal
       3'b100  : id2exe_ctrl.branch_ops = BR_LT;      // Branch less than
       3'b101  : id2exe_ctrl.branch_ops = BR_GE;      // Branch greater than or equal signed
```

```
        3'b110  : id2exe_ctrl.branch_ops = BR_LTU;     // Branch less than unsigned
        3'b111  : id2exe_ctrl.branch_ops = BR_GEU;     // Branch greater than or equal unsigned
        default : id2exe_ctrl.branch_ops = BR_NONE;    // No branch
    endcase // funct3_opcode
    end // OPCODE_BRANCH_INST
```

Listing 6.2. Instruction decoding for Branch operation.

```
OPCODE_JALR_INST : begin
    id2exe_ctrl.rd_wb_sel     = RD_WB_INC_PC;
    id2exe_ctrl.alu_opr1_sel = ALU_OPR1_REG;
    id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
    id2exe_ctrl.alu_ops       = ALU_OPS_ADD;
    id2exe_ctrl.rd_wr_req     = 1'b1;
    id2exe_ctrl.jump_req      = 1'b1;
    id2exe_data.imm            =   {{12{instr_codeword[31]}}, instr_codeword[19:12],
                                   instr_codeword[20], instr_codeword[30:21], 1'b0};

end // OPCODE_JALR_INST

// JAL operation
OPCODE_JAL_INST : begin
   id2exe_ctrl.rd_wb_sel     = RD_WB_INC_PC;
   id2exe_ctrl.alu_opr1_sel = ALU_OPR1_PC;
   id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
   id2exe_ctrl.alu_ops       = ALU_OPS_ADD;
   id2exe_ctrl.rd_wr_req     = 1'b1;
   id2exe_ctrl.jump_req      = 1'b1;
   id2exe_data.imm           = {{12{instr_codeword[31]}}, instr_codeword[19:12],
                               instr_codeword[20], instr_codeword[30:21], 1'b0};

end // OPCODE_JAL_INST
```

Listing 6.3. Instruction decoding for Jump operation.

Branch/Jump Operation – Execute Phase

Branch instruction requires comparison between two operands. Listing 6.4 illustrates comparison based on the subtract operation which can be used to utilize different flags based on the result of the subtraction between two operations.

```
// Difference calculation for comparison operations (branch and set-less-then operations)
assign cmp_operand_1 = id2exu_data.rs1_data;
assign cmp_operand_2 = (id2exu_ctrl.alu_cmp_opr2_sel == ALU_CMP_OPR2_IMM)
                      ? id2exu_data.imm
                      : id2exu_data.rs2_data;

assign cmp_output     = {1'b0, cmp_operand_1} - {1'b0, cmp_operand_2};
assign cmp_not_zero  = |cmp_output[`XLEN-1:0];
assign cmp_neg        = cmp_output[`XLEN-1];
assign cmp_overflow  = (cmp_neg & ~cmp_operand_1[`XLEN-1] & cmp_operand_2[`XLEN-1]) |
                       (~cmp_neg & cmp_operand_1[`XLEN-1] & ~cmp_operand_2[`XLEN-1]);
```

Listing 6.4. Comparison performed for branch operation.

There are different branch operations available in the RV 32I and based on the opcode of the type of branch the execution unit will use the comparison flags for checking different branch conditions. Listing 6.5 illustrates the use of comparison flags for different branch instructions.

```
// Evaluate the branch comparison result
always_comb begin
  // set comparison by default
  branch_res = 1'b0;
  case (id2exu_ctrl.branch_ops)
     BR_EQ:   branch_res = ~cmp_not_zero;
     BR_NE:   branch_res = cmp_not_zero;
     BR_LT:   branch_res = (cmp_neg ^ cmp_overflow);
     BR_LTU:  branch_res = cmp_output[`XLEN];         // Check if the carry-flag bit is set
     BR_GE:   branch_res = ~(cmp_neg ^ cmp_overflow);
     BR_GEU:  branch_res = ~cmp_output[`XLEN];        // Carry flag bit is cleared
     default: branch_res = 1'b0;
  endcase
end
```

Listing 6.5. Comparison flags used for branch condition check.

The jump instructions write their return address to the destination register. Listing 6,6 shows that the execute stage will update the value of PC in the fetch stage when the branch instruction is executed and the branch condition is true or the jump instruction is executed.

```
// Feedback signals from EXE to IF stage
assign exe2if_fb.jump_br_taken   = id2exu_ctrl.jump_req || (id2exu_ctrl.branch_req &
                                   branch_res);
assign exe2if_fb.alu_pc          = exe2mem_alu_result;
assign exe2if_fb_o               = exe2if_fb;
```

Listing 6.6. Feedback signals from execution to fetch stage.