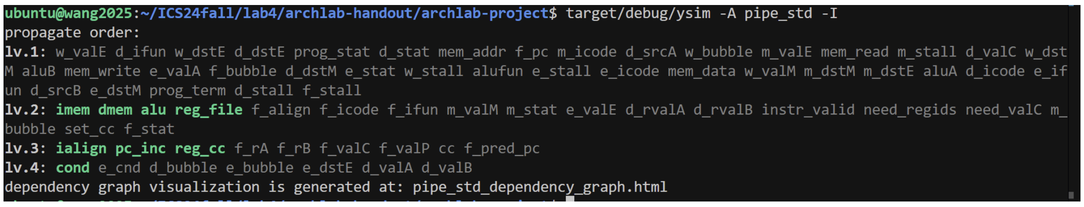
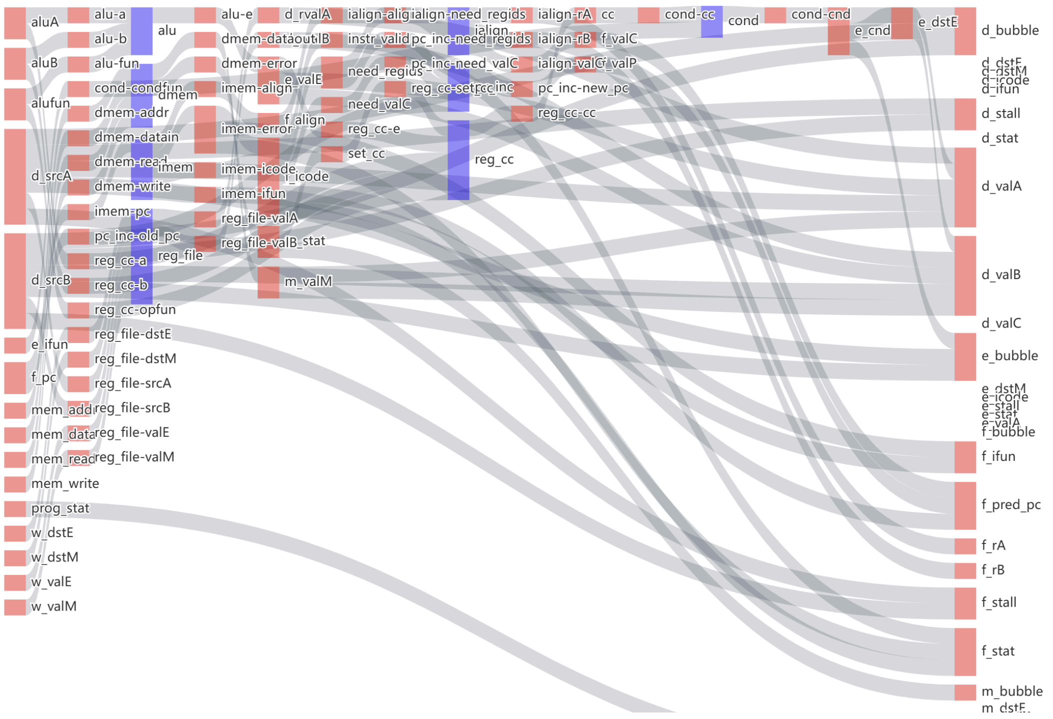
ArchLab-PartC

1.将 `pipe_std.rs` 复制进来，并按照Part-B方法添加 `IOPQ` 指令

2.观察可视化图，发现lv4只有 `cond` ,尝试对其进行改动



```
ubuntu@wang2025:~/ICS24fall/lab4/archlab-handout/archlab-project$ target/debug/ysim -A pipe_std -I
propagate order:
lv.1: w_valE d_ifun w_dstE d_dstE prog_stat d_stat mem_addr f_pc m_icode d_srcA w_bubble m_valE mem_read m_stall d_valC w_dst
M aluB mem_write e_valA f_bubble d_dstM e_stat w_stall alufun e_stall e_icode mem_data w_valM m_dstM m_dstE aluA d_icode e_if
un d_srcB e_dstM prog_term d_stall f_stall
lv.2: imem dmem alu reg_file f_align f_icode f_ifun m_valM m_stat e_valE d_rvalA d_rvalB instr_valid need_regids need_valC m_
bubble set_cc f_stat
lv.3: ialign pc_inc reg_cc f_rA f_rB f_valC f_valP cc f_pred_pc
lv.4: cond e_cnd d_bubble e_bubble e_dstE d_valA d_valB
dependency graph visualization is generated at: pipe_std_dependency_graph.html
```



3.分析四级的原因：需要先 `set cc` ,然后再根据 `reg_cc` 中的内容决定 `cond` ，计算 `e_cnd` 。但是只有 `OPQ` 和 `IOPQ` 指令需要进行 `set cc` ，而这两个指令是不需要 `Cond（CC，ifun）` 的。因此可以修改逻辑：

a.将 `cc` 存入M寄存器中

b.对于 `OPQ` 和 `IOPQ` 指令，进行 `set cc` 把 `e_cc` 置为新设置的值，然后存入M寄存器中

c.对于其他指令，利用M寄存器读出来的 `cc` 值计算 `e_cnd` 。把 `e_cc` 设置为从M寄存器读出来的值，根据需要计算再存回M寄存器中（保持不变)。

修改后对任何需要 `e_cnd` 情况值均与之前相同，因此其他部分无需任何修改。

完整实现如下

```
//! This architecture is used cooperatively with the `ncopy.ys` to gra
//! ncopy task.
//!
//! You can rewrite everything in the macro block of `crate::define_st
//! `sim_macro::hcl`. You should not change other outside code.

crate::define_stages! {
    FetchStage f {
        pred_pc: u64 = 0
    }
    DecodeStage d {
        stat: Stat = Bub, icode: u8 = NOP, ifun: u8 = 0,
        rA: u8 = RNONE, rB: u8 = RNONE,
        valC: u64 = 0, valP: u64 = 0
    }
    ExecuteStage e {
        stat: Stat = Bub, icode: u8 = NOP, ifun: u8 = 0,
        valC: u64 = 0,
        valA: u64 = 0, valB: u64 = 0,
        dstE: u8 = RNONE, dstM: u8 = RNONE,
        srcA: u8 = RNONE, srcB: u8 = RNONE
    }
    /// Memory Access Stage
    MemoryStage m {
        stat: Stat = Bub, icode: u8 = NOP, cnd: bool = false,
        valE: u64 = 0, valA: u64 = 0,
        dstE: u8 = RNONE, dstM: u8 = RNONE, cc: ConditionCode = Condit
            sf:false,
            of:false,
            zf:false
        }
    }
    WritebackStage w {
        stat: Stat = Bub, icode: u8 = NOP, valE: u64 = 0,
        valM: u64 = 0, dstE: u8 = RNONE, dstM: u8 = RNONE
    }
}

sim_macro::hcl! {

// Specify the CPU hardware devices set.
// This will imports all items from the hardware module.
#![hardware = crate::architectures::hardware_pipe]
```

```
// Specify the program counter by an intermediate signal. This value i
// debugger. Conventionally, when we create a breakpoint at the line o
// debugger seems to stop before executing the line of code. But in th
// The breakpoint take effects when the current cycle is executed (so
// is calculated) and before the next cycle enters.
//
// Changing this value to other signals makes no difference to the sim
// But it affects the behavior of the debugger.
#![program_counter = f_pc]

// Specify a boolean intermediate signal to indicate whether the progr
// be terminated.
#![termination = prog_term]

// This attribute defines the identifiers for pipeline registers. For
// identifier `f` is the short name in [`crate::define_stages`], and `
// arbitrarily chosen.
//
// e.g. M.valA is the value at the start of the cycle (you should trea
// read-only), m.valA is the value at the end of the cycle (you should
#![stage_alias(F => f, D => d, E => e, M => m, W => w)]

use Stat::*;

// You can use `:====: title :====:` to declare a section. This helps
// your code and the information displayed by debugger. It makes no di
// the simulation. That means it does not alter the evaluation order o
:==============================: Fetch Stage :========================

// What address should instruction be fetched at
u64 f_pc = [
    // Mispredicted branch. Fetch at incremented PC
    M.icode == JX && !M.cnd : M.valA;
    // Completion of RET instruction
    W.icode == RET : W.valM;
    // Default: Use predicted value of PC (default to 0)
     1 : F.pred_pc;
];

@set_input(imem, {
    pc: f_pc
});

// Determine icode of fetched instruction
u8 f_icode = [
    imem.error : NOP;
    1 : imem.icode;
```

```
];

// Determine ifun
u8 f_ifun = [
    imem.error : 0xf; // FNONE;
    1 : imem.ifun;
];


// Is instruction valid?
bool instr_valid = f_icode in { NOP, HALT, CMOVX, IRMOVQ, RMMOVQ,
    MRMOVQ, OPQ, JX, CALL, RET, PUSHQ, POPQ,IOPQ};

// Determine status code for fetched instruction
Stat f_stat = [
    imem.error : Adr;
    !instr_valid : Ins;
    f_icode == HALT : Hlt;
    1 : Aok;
];

// Does fetched instruction require a regid byte?
bool need_regids
    = f_icode in { CMOVX, OPQ, PUSHQ, POPQ, IRMOVQ, RMMOVQ, MRMOVQ,IOP

// Does fetched instruction require a constant word?
bool need_valC = f_icode in { IRMOVQ, RMMOVQ, MRMOVQ, JX, CALL,IOPQ };

@set_input(pc_inc, {
    need_valC: need_valC,
    need_regids: need_regids,
    old_pc: f_pc,
});

u64 f_valP =  pc_inc.new_pc;

[u8; 9] f_align = imem.align;

@set_input(ialign, {
    align: f_align,
    need_regids: need_regids,
});

u64 f_valC =  ialign.valC;
u8 f_rA = ialign.rA;
u8 f_rB = ialign.rB;
```

```
// Predict next value of PC
u64 f_pred_pc = [
     f_icode in { JX, CALL } : f_valC;
     1 : f_valP;
];

@set_stage(f, {
    pred_pc: f_pred_pc,
});

@set_stage(d, {
    icode: f_icode,
    ifun: f_ifun,
    stat: f_stat,
    valC: f_valC,
    valP: f_valP,
    rA: f_rA,
    rB: f_rB,
});

:=======================: Decode and Write Back Stage :===============

// What register should be used as the A source?
u8 d_srcA = [
    D.icode in { CMOVX, RMMOVQ, OPQ, PUSHQ } : D.rA;
    D.icode in { POPQ, RET } : RSP;
    1 : RNONE; // Don't need register
];

// What register should be used as the B source?
u8 d_srcB = [
    D.icode in { OPQ, RMMOVQ, MRMOVQ ,IOPQ} : D.rB;
    D.icode in { PUSHQ, POPQ, CALL, RET } : RSP;
    1 : RNONE; // Don't need register
];

// What register should be used as the E destination?
u8 d_dstE = [
    D.icode in { CMOVX, IRMOVQ, OPQ,IOPQ} : D.rB;
    D.icode in { PUSHQ, POPQ, CALL, RET } : RSP;
    1 : RNONE; // Don't write any register
];

// What register should be used as the M destination?
u8 d_dstM = [
    D.icode in { MRMOVQ, POPQ } : D.rA;
    1 : RNONE; // Don't write any register
];

u64 d_rvalA = reg_file.valA;
u64 d_rvalB = reg_file.valB;

// What should be the A value?
// Forward into decode stage for valA
u64 d_valA = [
    D.icode in { CALL, JX } : D.valP; // Use incremented PC
    d_srcA == e_dstE : e_valE; // Forward valE from execute
    d_srcA == M.dstM : m_valM; // Forward valM from memory
    d_srcA == M.dstE : M.valE; // Forward valE from memory
    d_srcA == W.dstM : W.valM; // Forward valM from write back
    d_srcA == W.dstE : W.valE; // Forward valE from write back
    1 : d_rvalA; // Use value read from register file
];

u64 d_valB = [
    d_srcB == e_dstE : e_valE; // Forward valE from execute
    d_srcB == M.dstM : m_valM; // Forward valM from memory
    d_srcB == M.dstE : M.valE; // Forward valE from memory
    d_srcB == W.dstM : W.valM; // Forward valM from write back
    d_srcB == W.dstE : W.valE; // Forward valE from write back
    1 : d_rvalB; // Use value read from register file
];

u64 d_valC = D.valC;
u8 d_icode = D.icode;
u8 d_ifun = D.ifun;
Stat d_stat = D.stat;

@set_stage(e, {
    icode: d_icode,
    ifun: d_ifun,
    stat: d_stat,
    valC: d_valC,
    srcA: d_srcA,
    srcB: d_srcB,
    valA: d_valA,
    valB: d_valB,
    dstE: d_dstE,
    dstM: d_dstM,
});

:==============================: Execute Stage :======================

// Select input A to ALU
```

```
u64 aluA = [
    E.icode in { CMOVX, OPQ } : E.valA;
    E.icode in { IRMOVQ, RMMOVQ, MRMOVQ,IOPQ } : E.valC;
    E.icode in { CALL, PUSHQ } : NEG_8;
    E.icode in { RET, POPQ } : 8;
    1 : 0; // Other instructions don't need ALU
];

// Select input B to ALU
u64 aluB = [
    E.icode in { RMMOVQ, MRMOVQ, OPQ, CALL, PUSHQ, RET, POPQ,IOPQ } :
    E.icode in { CMOVX, IRMOVQ } : 0;
    1 : 0; // Other instructions don't need ALU
];

// Set the ALU function
u8 alufun = [
    E.icode in { OPQ, IOPQ} : E.ifun;
    1 : ADD;
];

@set_input(alu, {
    a: aluA,
    b: aluB,
    fun: alufun,
});

// Should the condition codes be updated?
bool set_cc = E.icode in {OPQ , IOPQ} &&
    // State changes only during normal operation
    !(m_stat in { Adr, Ins, Hlt }) && !(W.stat in { Adr, Ins, Hlt });

u64 e_valE = alu.e;

@set_input(reg_cc, {
    a: aluA,
    b: aluB,
    e: e_valE,
    opfun: alufun,
    set_cc: set_cc,
});
ConditionCode ccm=M.cc;

ConditionCode e_cc = [
    set_cc:reg_cc.cc;
    1:ccm
];
```

```
u8 e_ifun = E.ifun;


@set_input(cond, {
    cc: ccm,
    condfun: e_ifun,
});

bool e_cnd = cond.cnd;

// Generate valA in execute stage
u64 e_valA = E.valA;    // Pass valA through stage

// Set dstE to RNONE in event of not-taken conditional move
u8 e_dstE = [
    E.icode == CMOVX && !e_cnd : RNONE;
    1 : E.dstE;
];

u8 e_dstM = E.dstM;
u8 e_icode = E.icode;
Stat e_stat = E.stat;

@set_stage(m, {
    stat: e_stat,
    dstM: e_dstM,
    icode: e_icode,
    dstE: e_dstE,
    cnd: e_cnd,
    valE: e_valE,
    valA: e_valA,
    cc:e_cc,
});

:===============================: Memory Stage :======================

// Select memory address
u64 mem_addr = [
    M.icode in { RMMOVQ, PUSHQ, CALL, MRMOVQ } : M.valE;
    M.icode in { POPQ, RET } : M.valA;
    // Other instructions don't need address
];

// Set read control signal
bool mem_read = M.icode in { MRMOVQ, POPQ, RET };

// Set write control signal
```

```
bool mem_write = M.icode in { RMMOVQ, PUSHQ, CALL };

u64 mem_data = M.valA;

@set_input(dmem, {
    read: mem_read,
    write: mem_write,
    addr: mem_addr,
    datain: mem_data,
});

// Update the status
Stat m_stat = [
    dmem.error : Adr;
    1 : M.stat;
];

u8 m_icode = M.icode;

u64 m_valM = dmem.dataout;
u64 m_valE = M.valE;
u8 m_dstE = M.dstE;
u8 m_dstM = M.dstM;

@set_stage(w, {
    stat: m_stat,
    icode: m_icode,
    valE: m_valE,
    valM: m_valM,
    dstE: m_dstE,
    dstM: m_dstM,
});

:=============================: Write Back Stage :====================

// Set E port register ID
u8 w_dstE = W.dstE;

// Set E port value
u64 w_valE = W.valE;

// Set M port register ID
u8 w_dstM = W.dstM;

// Set M port value
u64 w_valM = W.valM;
```

```
@set_input(reg_file, {
    srcA: d_srcA,
    srcB: d_srcB,
    dstE: w_dstE,
    dstM: w_dstM,
    valM: w_valM,
    valE: w_valE,
});

// Update processor status (used for outside monitoring)
Stat prog_stat = [
    W.stat == Bub : Aok;
    1 : W.stat;
];

bool prog_term = [
    prog_stat in { Aok, Bub } : false;
    1 : true
];

:========================: Pipeline Register Control :================

// Should I stall or inject a bubble into Pipeline Register F?
// At most one of these can be true.
bool f_bubble = false;
bool f_stall =
    // Conditions for a load/use hazard
    E.icode in { MRMOVQ, POPQ } && E.dstM in { d_srcA, d_srcB } ||
    // Stalling at fetch while ret passes through pipeline
    RET in {D.icode, E.icode, M.icode};

@set_stage(f, {
    bubble: f_bubble,
    stall: f_stall,
});

// Should I stall or inject a bubble into Pipeline Register D?
// At most one of these can be true.
bool d_stall =
    // Conditions for a load/use hazard
    E.icode in { MRMOVQ, POPQ } && E.dstM in { d_srcA, d_srcB };

bool d_bubble =
    // Mispredicted branch
    (E.icode == JX && !e_cnd) ||
    // Stalling at fetch while ret passes through pipeline
    // but not condition for a load/use hazard
    !(E.icode in { MRMOVQ, POPQ } && E.dstM in { d_srcA, d_srcB }) &&
      RET in {D.icode, E.icode, M.icode};

@set_stage(d, {
    stall: d_stall,
    bubble: d_bubble,
});

// Should I stall or inject a bubble into Pipeline Register E?
// At most one of these can be true.
bool e_stall = false;
bool e_bubble =
    // Mispredicted branch
    (E.icode == JX && !e_cnd) ||
    // Conditions for a load/use hazard
    E.icode in { MRMOVQ, POPQ } && E.dstM in { d_srcA, d_srcB };

@set_stage(e, {
    stall: e_stall,
    bubble: e_bubble,
});

// Should I stall or inject a bubble into Pipeline Register M?
// At most one of these can be true.
bool m_stall = false;
// Start injecting bubbles as soon as exception passes through memory
bool m_bubble =
    m_stat in { Adr, Ins, Hlt } || W.stat in { Adr, Ins, Hlt };

@set_stage(m, {
    stall: m_stall,
    bubble: m_bubble,
});

// Should I stall or inject a bubble into Pipeline Register W?
bool w_stall = W.stat in { Adr, Ins, Hlt };
bool w_bubble = false;

@set_stage(w, {
    stall: w_stall,
    bubble: w_bubble,
});
}

mod nofmt {
    use super::*;
    use crate::{
```

```
    framework::PipeSim,
    utils::{format_ctrl, format_icode},
};
impl PipeSim<Arch> {
    // print state at the beginning of a cycle
    pub fn print_state(&self) {
        // For stage registers, outputs contains information for t

        #[allow(non_snake_case)]
        let PipeRegs {
            f: _,
            d: D,
            e: E,
            m: M,
            w: W,
        } = &self.cur_state;
        let PipeRegs { f, d, e, m, w } = &self.nex_state;

        println!(
            r#"Stat    F {fstat}    D {dstat}    E {estat}    M {m
icode   f {ficode} D {dicode} E {eicode} M {micode} W {wicode}
Control F {fctrl:6} D {dctrl:6} E {ectrl:6} M {mctrl:6} W {wctrl:6}"#,
            fstat = Aok,
            dstat = D.stat,
            estat = E.stat,
            mstat = M.stat,
            wstat = W.stat,
            // stage control at the end of last cycle
            // e.g. dctrl is computed in fetch stage. if dctrl is
            // then in the next cycle, D.icode will be NOP.
            // e. Controls are applied between cycles.
            fctrl = format_ctrl(f.bubble, f.stall),
            dctrl = format_ctrl(d.bubble, d.stall),
            ectrl = format_ctrl(e.bubble, e.stall),
            mctrl = format_ctrl(m.bubble, m.stall),
            wctrl = format_ctrl(w.bubble, w.stall),
            // ficode is actually computed value
            ficode = format_icode(d.icode),
            dicode = format_icode(D.icode),
            eicode = format_icode(E.icode),
            micode = format_icode(M.icode),
            wicode = format_icode(W.icode),
        );
    }
}
```

```
}
```