ICS-PJ 报告

姓名： 邱深凌

学号： 24300240123

Evaluation

所有代码已经编译完毕。

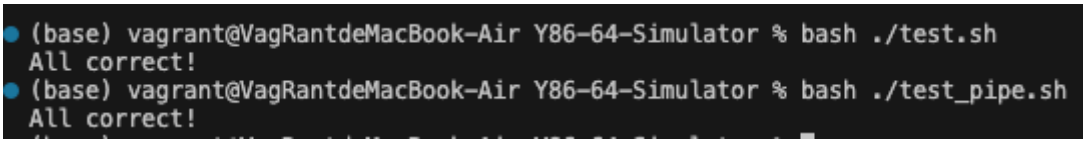
1. 请 cd 到当前目录，建议使用 linux/macOS 环境（项目开发环境为macOS）
2. 对于 SEQ CPU，请运行

```
bash ./test.sh
```

3. 对于 Pipeline CPU，请运行

```
bash ./test_pipe.sh
```

预期效果：



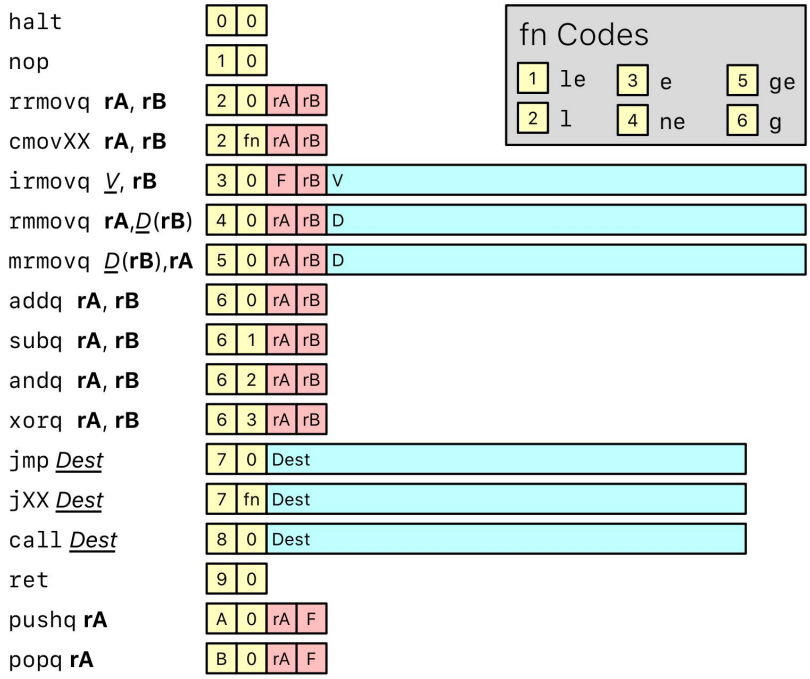
对于流水线cpu，实际上修改test.py为pipe_test.py，略过对中间过程的检查，只保留对于最后状态的检查

一、Y86-64简介

CSCI 370: Computer Architecture

Y86-64 Reference

Instruction Format



Registers

ID	Enc	Usage	
%rdi	7	arg1	caller-saved
%rsi	6	arg2	
%rdx	2	arg3	
%rcx	1	arg4	
%r8	8	arg5	
%r9	9	arg6	
%rax	0	return	
%r10	A	general	
%r11	B	general	
%rbx	3	general	callee-saved
%r12	C	general	
%r13	D	general	
%r14	E	general	
%rsp	4	stack ptr	
%rbp	5	base ptr	
	F	no reg	

Status Conditions

AOK	1	Normal
HLT	2	Halt Encountered
ADR	3	Bad Address
INS	4	Invalid Instruction

HCL Y86-64 Hardware Registers

stage	register(s)	description
Fetch	icode,ifun	Read instruction byte
	rA,rB	Read register byte
	valC	Read constant word
	valP	Compute next PC
Decode	valA,srcA	Read operand A
	valB,srcB	Read operand B
Execute	valE	Perform ALU operation
	cnd	Set/Use Condition Code
Memory	valM	Memory Read/Write
Writeback	dstE	Write back ALU result
	dstM	Write back Mem result
PC Update	PC	Update PC

Y86-64 Data Example

```
    .align 8
Array:
    .quad 0x0000000000000001
    .quad 0x0000000000000002
    .quad 0x0000000000000003
    .quad 0x0000000000000004
```

Assembly Translation Example

```
/* find number of elements in null-terminated list */
long len(long* a) {
  long len;
  for (len = 0; a[len]; ++len)
    ;
  return len;
}
```

```
len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    irmovq $0, %rax         # len = 0
    mrmovq (%rdi), %rdx     # val = *a
    and %rdx, %rdx          # Test val
    je Done                 # If zero, goto Done
Loop:
    addq %r8, %rax          # len++
    addq %r9, %rdi          # a++
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx         # Test val
    jne Loop                # If !0, goto Loop
Done:
    ret
```

Y86-6包括： **15个寄存器** %rax, %rcx, %rdx, %rbx %rsp, %rbp, %rsi, %rdi %r8, %r9, %r10, %r11, %r12, %r13, %r14 每个寄存器有4-bit ID

程序计数器 PC（Program Counter） 下一条将要执行的指令的地址

Condition Codes ZF zero flag SF sign flag OF overflow flag

状态码 Stat

`AOK`：正常运行 `HLT`：执行到halt `ADR`：地址错误 `INS`：非法指令

指令 icode:ifun rA:rB 常数 valC（偏移量等）

调用栈 栈指针`%rsp`维护，支持push pop

二、PIPE CPU简介

流水线寄存器 Pipeline Registers

定义结构体来保存每个阶段之间的状态 **F_reg** 保存程序计数器预测值 `predPC` **D_reg** 保存F取出的指令 **E_reg** 保存D读出的值 **M_reg** 保存E alu结果 **W_reg** 保存M访存结果

Hazards

引入 **Stall** 和 **Bubble** 当 E 需要用到 M 加载的数据时，暂停F和D并向E插入气泡； 采用总是跳转策略。如果E发现条件不满足，取消D和E的指令（注入气泡），并修正PC

三、具体实现

SEQ CPU

本质上就是一个巨大的switch

Pipeline CPU

首先，将程序命令读入，解读程序 `parse_yo_program`

在此基础上，初始化寄存器、内存、flag、PC，同时进入`Y86CPU::run`，run中根据stat为AOK就继续的循环，每次调用`Y86CPU::step()`同时把状态打印出来，并更新stat，后面对于stringstream的处理借助了genAI调试（毕竟parse JSON繁琐且不是重点）

核心处理：Y86CPU::step( )

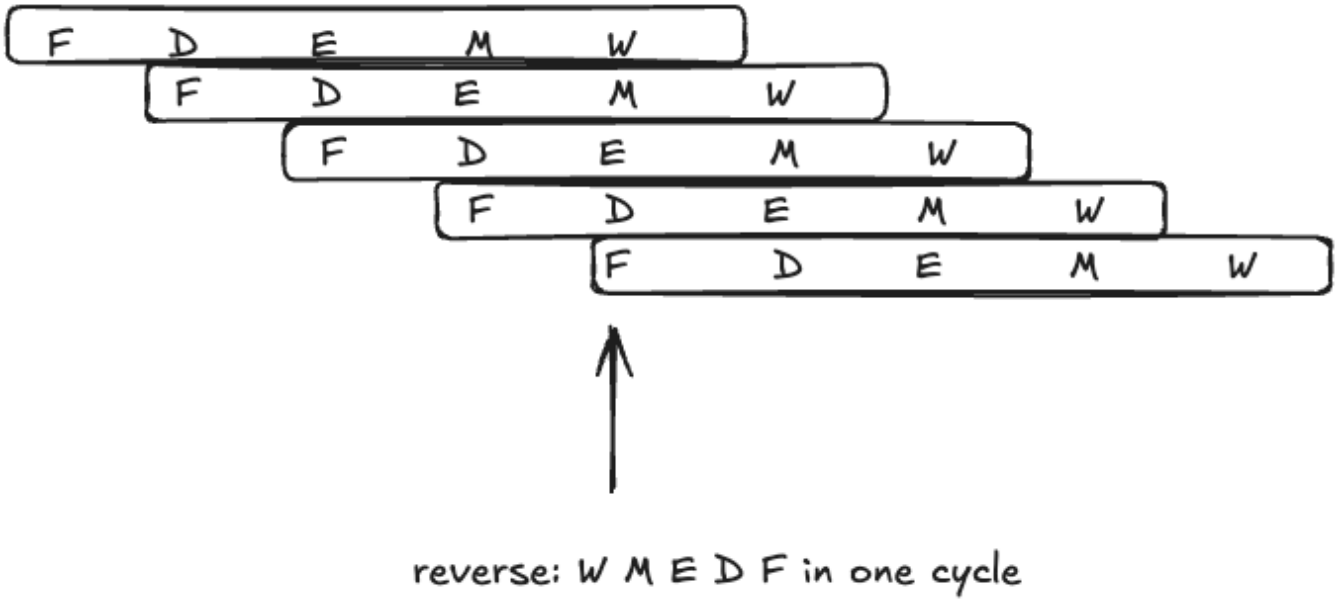
```
void Y86CPU::step() {
    writeback();
    memory1();
    execute();
    decode();
    fetch();

    update_pipereg();

    if (W.stat != 1) {
        if (W.stat == 2) stat = "HLT";
        else if (W.stat == 3) stat = "ADR";
        else if (W.stat == 4) stat = "INS";
```

```
        }
    }
```

依次实现pipeline的各个阶段。这里的顺序也有讲究，实际上作为一个模拟器我们没有办法真正并行 ~~多线程不考虑~~ 因此在实际的串行中，我们必须所有阶段只读旧的流水线寄存器和memory，只写next的流水线寄存器，那么我们实际上从后往前，即把 F D E M W改为W M E D F，就可以实现每次处理都是只依赖后面某些已经算出来的next信息，算是手动实现mask（不然如果F D，那么fetch结果就会影响本次的D，以此类推）



最后，根据writeback的寄存器W检查状态并更新全局状态，进入step时F D E M W保存的是这一时钟周期开始时的流水线状态（reg memory CC flags），之后5个函数分别模拟各个部分这一周期的运行结果，并更新下一周期需要使用的对应参数（W_in, M_in, E_in, D_in, f_predPC），调用`update_pipereg`作为一个周期的subbmittion并检查预测情况（根据情况修改stall bubble PC）并更新W_in为W...

下面对于几个部分的实现进行具体分析（辅助函数，例如更新内存 etc同SEQ）

Y86CPU::writeback( )

`struct W_Register`中维护状态码（实际上step结尾就是按照W的状态决定整个CPU的状态），同时维护icode valE（ALU计算结果），valM（内存中读取的值），valP（预测的下一条指令，snapshot中打印），以及当前PC

writeback还维护了dstE dstM（要写回的寄存器或memory）

根据当前的`W.stat`，如果AOK则把valE写到dstE，把valM写到dstM

特别的是，如果ADR，也要把valE写到dstE，才能保证execute得到的结果正常传递下去，不然prog10.json会出现问题（面向答案特判）

Y86CPU::memory1( ) `struct M_Register`依然维护stat（由E阶段传入）、icode Cnd（E给出的条件计算结果，对于jXX cmovXX使用）。

此外有valE（E结果） valA（第二个op） dstE（目标寄存器编号） dstM（popq mrmovq使用，从这里读取数据写入valM）

在memory处理中，首先把M寄存器中的状态复制到W（下一状态的W），之后根据不同的icode选取内存读/写的地址和内容修改。如果读取，则把内容保存进`W_in.valM`（当然需要判断ADR）；如果写入则调用`write_qword`

Y86CPU::execute( )

ALU逻辑实现，首先`struct E_Register`中除之前出现的，还需要ifun valP（顺序执行时，下一条PC）以及srcA drcB（D得到的指令）

执行层面，先把E reg的参数传给M_in，之后根据icode ifun选择输入和计算，计算得到valE之后更新flags（对于OPq），同时根据flags以及ifun更新Cnd（jXX cmovXX需要的条件）并将Cnd valE写入M_in

对于rmmovq/mrmovq，因为有base和shift，综合使用valA valC

当然，如果W传入的状态有问题，`bool exception = (W.stat != 1 || W_in.stat != 1);`则不更新条件。之所以用W是因为W为这一个时钟周期开始最早的指令（当前流水线最早的指令），而W_in为第二老的指令（memory结束进入wb），E执行的是更新的指令，因而如果之前的指令出现问题，则后续指令不应当修改CC/reg/memory，而是在当前step的最后更改CPU状态

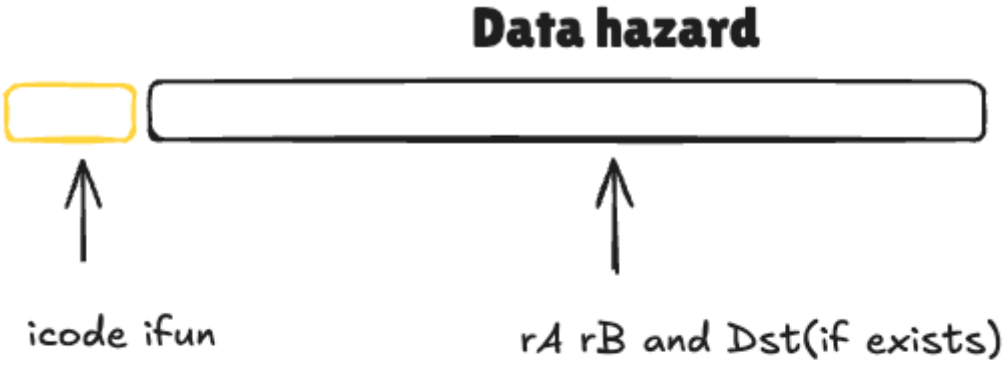
最后把计算结果存入M_in

Y86CPU::decode( )

decode和fetch是hazards的主要地方

首先把搬运的信息传入E_in，之后是一个巨大的if else，根据icode来决定 之后初始化所有寄存器编号为15（即none），并根据icode取出对应操作数

rrmovq/rmmovq/cmovXX/popq srcA=rA ret srcA=4(rsp) OPq/rmmovq/mrmovq srcB=rB pushq/popq/call/ret srcB=4 以此类推，再选择dstE（接收valE） 对于mrmovq/popq，dstM=rA，讲内存读取的值写入rA寄存器




```
uint64_t valA = get_reg_val(srcA);
if (srcA != 15) { // forwarding
    if (srcA == M_in.dstE) valA = M_in.valE;
    else if (srcA == W_in.dstM) valA = W_in.valM;
    else if (srcA == W_in.dstE) valA = W_in.valE;
}

// call and jXX, valA is valP (return addr / next PC)

if (icode == 8 || icode == 7) valA = D.valP;

E_in.valA = valA;
```

```
if ((E.icode == 5 || E.icode == 11) && (E.dstM == srcA || E.dstM == srcB)) {
    F.stall = true;
    D.stall = true;
    E_in.bubble = true;
} else {
    F.stall = false;
    D.stall = false;
    E_in.bubble = false;
}
```

forwarding 更新完之后，把src/dst写入E_in，并执行ALU计算，并进行forwarding，不用等指令真正写回寄存器，直接从M/W阶段的结果里读取，具体而言：

如果PC为call/jXX则valA置为return addr/nextPC； 不然如果刚刚在本周期的execute( )中，有指令要在M写回这个寄存器`M_in.valE`则直接从M_in中读取； 否则，如果刚刚在memory1( )中，有mrmovq指令要写它则`valA = W_in.valM`； 否则，如果刚刚有非load指令要写它则W_in.valE； 再否则，才需要去寄存器文件里读旧的值`valA = get_reg_val(srcA);`

最后，`E_in.valA = valA;`将结果存入下一轮E使用的上下文

stall & bubble 在hazard之后，最后一部分必须加上检查，如果： `E.icode == 5`上一条（现在在E的）是mrmovq（从内存读）； 或者`E.icode == 11`上一条是 popq（也是内存读）； 且： `(E.dstM == srcA || E.dstM == srcB)` 当前decode这条指令的srcA或srcB正好等于E中目的寄存器dstM，

也就是：当前要用的寄存器 = 上一条 load 要填的寄存器，则出现问题，必须出发`stall & bubble`，通过维护两个bool，来维护是否需要停住不更新（或者插入nop）来应对

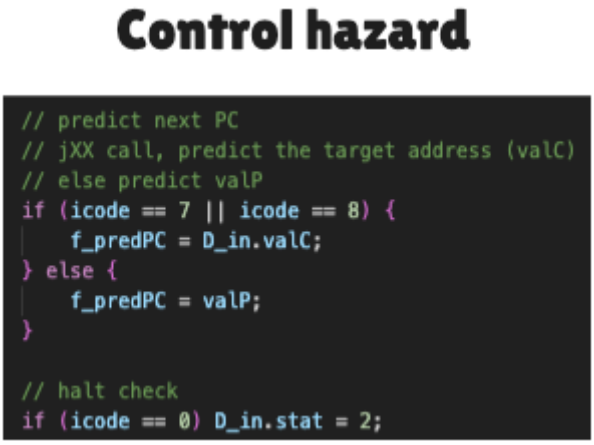
Y86CPU::fetch( )

F的上下文很简单，保存下一条指令的地址（即fetch的结果）即可

具体而言，现在（刚刚实行的是F.predPC），之后先取出F中的预测PC（这是`update_pipereg`实现的）

之后取出取指令的第一个字节的icode ifun（如果这个地址在内存中不存在，标记 stat=ADR），并根据icode读取reg memory保存进D_in

在这里还需要一个下一周期D_in需要的PC（valP），先预测为PC+1（f_pc+1）并读取出对应的rA rB 写到D_in，即跳过icode_ifun的地址。 根据当前icode，判断当前指令长度，读取对应rA rB写入D_in，并维护valP指向已经读过部分读最后（valP += 8）； 之后D_in.PC = f_pc维护当前PC，进入分支预测部分



分支预测：固定跳转 static predict-taken

如果当前指令为jXX / call则预测下一条PC从valC指向的地址开始，否则则顺序执行

最后如果`halt`，则维护D_in的stat为HLT

Y86CPU::update_pipereg( )

```cpp
void Y86CPU::update_pipereg() {
    // Load-Use Hazard do_decode, sets E_in.bubble.
    bool load_use = E_in.bubble;


    bool mispredicted = (E.icode == 7 && !M_in.Cnd);

    // if ret (9) is in D, E, or M stages, stall Fetch
    // else if we mispredicted,  D is wrong ,don't stall
    bool ret_hazard = ((D.icode == 9 && !mispredicted) || E.icode == 9 || M.icode == 9);
    if (mispredicted) {
        D.bubble = true;
        E_in.bubble = true;
        F.stall = false;
    }

    if (ret_hazard) {
        F.stall = true; // stall Fetch
        // bubble D to insert NOP
        bool ret_stall_d = (D.icode == 9 && load_use);
        if (!ret_stall_d) D.bubble = true;
    }

    // change f_predPC
    if (W.icode == 9) {
        f_predPC = W.valM;
        D.bubble = true;
    } else if (mispredicted) {
        f_predPC = E.valP;
    }
    if (!F.stall) F.predPC = f_predPC;

    if (D.bubble) D = D_Register();
    else if (!D.stall) D = D_in;

    if (E_in.bubble) E = E_Register();
    else E = E_in;
    M = M_in;
    W = W_in;
}
```

这是每个时钟周期结束的最后，更新stall bubble，修正分支预测并更新F D E M W的最核心的部分

mispredicted 如果`(E.icode == 7 && !M_in.Cnd)`即当前E指令为jXX且E已经计算好的条件为假，则出现了错误估计（我们是固定跳转）mispredicted的情况，把D / E_in进入bubble，因为decode的是错误PC；同时execute的jXX不应该执行。不过对于分支预测错误，不需要停止fetch，`F.stall = false`

ret hazard 如果D / M / E目前的命令是ret，则出现了ret hazard，必须等待ret运行完成把返回地址从内存读取再顺序执行，因此fetch需要stall（后续不会更新F.predPC），并且对于D也需要bubble，不过如果D load use，它其实已经被stall（根据E_in是否需要bubble来判断，这是decode设置的，如果上一条mrmovq popq要用这一条的结果则bubble）

修改predPC 如果ret已经完成（writeback阶段为ret），则下一条PC为W.valM。此外还需要重新初始化D的上下文，防止ret hazard中错误PC往下传

如果分支预测错误，则`f_predPC = E.valP`，即用正确的fall-through的PC覆盖之前fetch中的预测，之后重新开始取址

最后，一次更新`D = D_in` stc

附录

一、参考

Y86-64 Bryant, R. E., & O'Hallaron, D. R. (2016). Computer systems: A programmer's perspective(3rd ed.). 机械工业出版社 https://www.cs.williams.edu/~jeannie/cs237/slides/lecture14.pdf https://www.bilibili.com/video/BV12p4y1b7Zo/?spm_id_from=333.337.search-card.all.click&vd_source=885276c19c3f499c0ed2a1e7dde296b8 https://ultrafish.io/post/Y86-64-learning-2

流水线 https://www.youtube.com/watch?v=tvrS5L-Y2LY https://bobcn.github.io/2018/03/30/make_y86_cpu_5/ https://github.com/ThayerAckerman/y86-simulator https://www.cs.williams.edu/~jeannie/cs237/slides/lecture19.pdf

代码实现 https://github.com/nightrain-vampire/ICS-Y86 https://github.com/nlohmann/json

分支预测 https://zhuanlan.zhihu.com/p/22469702 https://zhuanlan.zhihu.com/p/602635898

二、Declearation of genAI use

本项目实现中：

基础代码框架由作者完成，`parse JSON`部分使用genAI辅助；

调试过程中使用AI辅助解读diff和排查问题；

完成之后为了整理混乱的代码并解决bug，使用AI进行了一轮代码风格检查，优化封装。

`pipe_test.py`使用AI基于test.py修改而来；

其余内容，包括但不限于代码、最终PJ报告以及汇报ppt都由作者完成，且保证AI生成代码经过作者逐行检查。