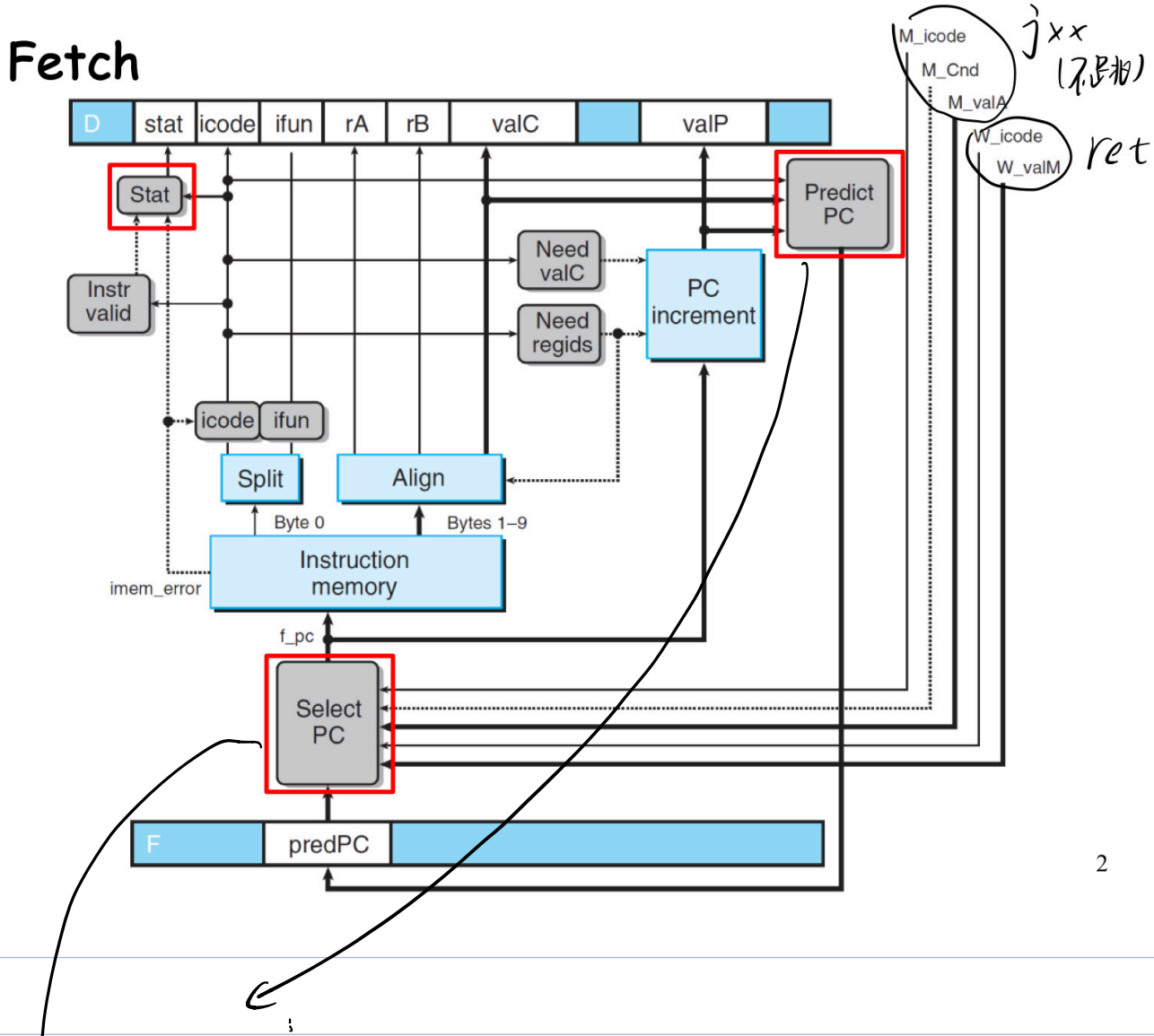
fetch:



f_icode in { IJXX, ICALL }: f_valC;

1: f_valP; 预测会跳

M_icode == IJXX && !M_cnd : M_valA

W_icode == IRET : W_valM

下滑线前的大写字母代表这个数据是从哪个阶段传回来的。

为什么不 Memory 读了以后直接传回来，时间周期太长。

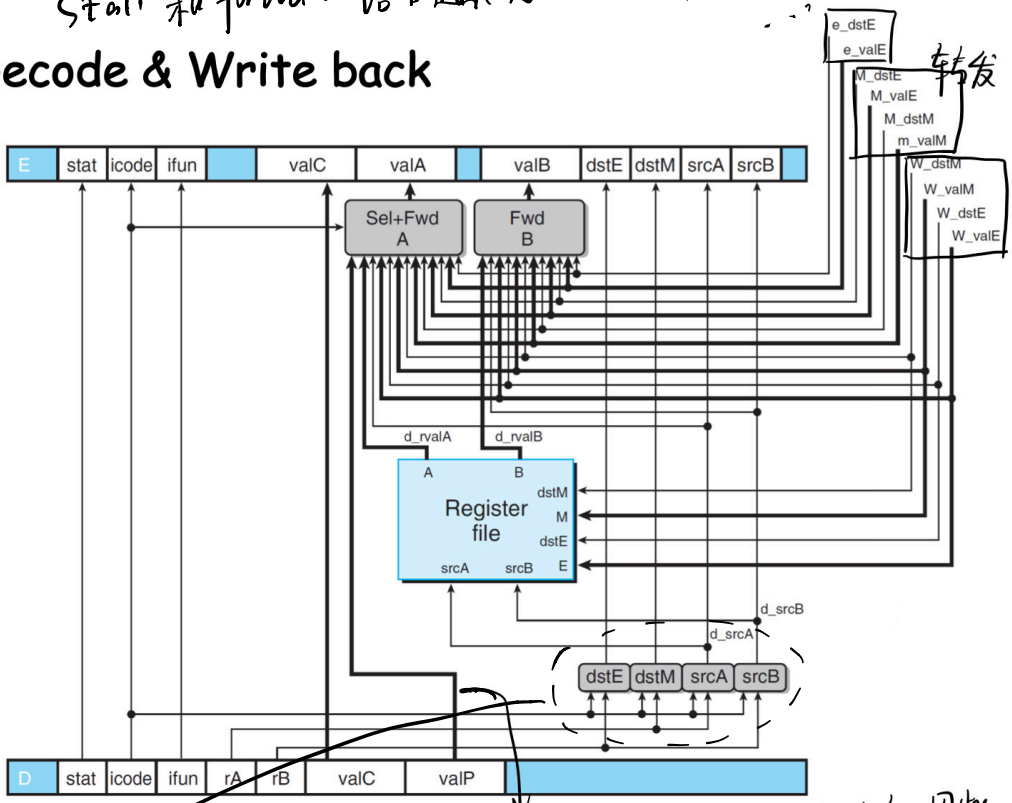
M阶段快结束了 还要重新取指

处理加载/使用

↑

Stall 和 forward 结合起来是以处理所有可能类型的数据冒险

Decode & Write back



依 Jxx Call 会将 valP 上传 因此不会与A寄存器冲突

- srcA: read port address for valA {rA, %rsp}
- srcB: read port address for valB {rB, %rsp}
- dstE: write port address for valE {rB, %rsp}
- dstM: write port address for valM {rA}

rrmovq irmovq cmovXX ← OP %rA, %rB | popq

mrmovq %rsp ← pushq

```
int dstE = [
    icode in { IRRMOVQ, IIRMOVQ,
               IOPQ} : rB;
    icode in { IPUSHQ, IPOPQ, ICALL,
               IRET }: RRSP;
    1 : RNONE;  # Don't need register
];
// How to consider cmovXX ?
    icode==IRRMOVQ&&ifun&&Cnd:rB;
```

```
## What should be the A value?
int d_valA = [
  # Use incremented PC
    D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
    d_srcA == E_dstE : e_valE;
  # Forward valM from memory
    d_srcA == M_dstM : m_valM;
  # Forward valE from memory
    d_srcA == M_dstE : M_valE;
  # Forward valM from write back
    d_srcA == W_dstM : W_valM;
  # Forward valE from write back
    d_srcA == W_dstE : W_valE;
  # Use value read from register file
    1 : d_rvalA;
];
```

先写内存 再写E

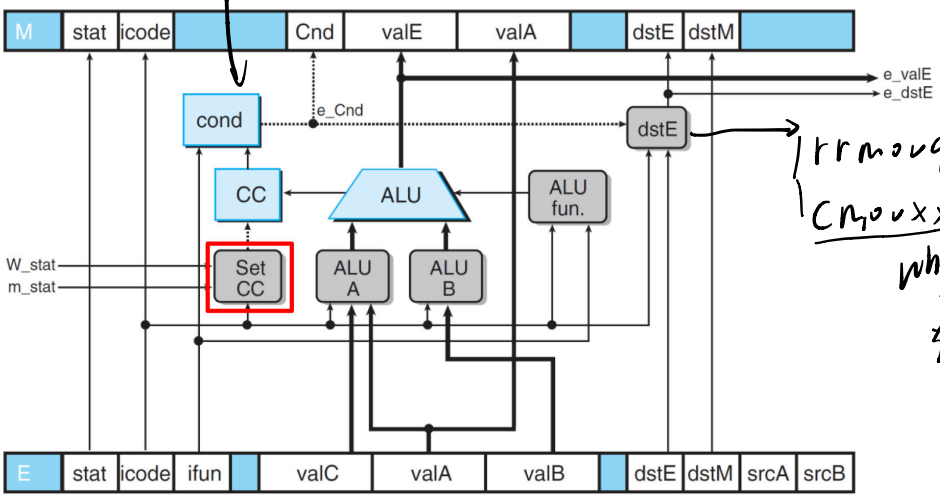
保证pop正确执行

The order matters!!

由近及远 execute → memory → write back

Execute

jxx 在 execute 阶段 判条件码



需要 cond 和 icode

need to check 是否移动

rrmovq is valE

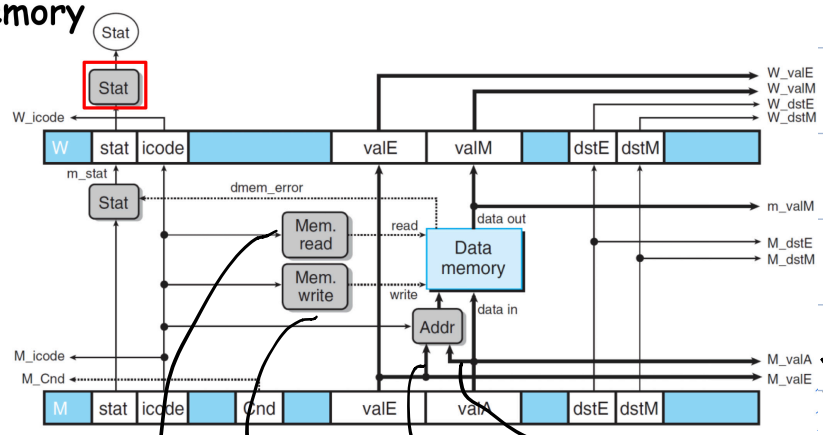
cmovxx

why

在 rrmovq 执行阶段会

将 valE = 0 + valA

Memory



jxx PC

register file

DL%rB)

- Mem. read: should word be read?
- Mem. write: should word be written? → R[%rsp],
- Mem. addr.: Select address {valA, valE}
- Mem. data.: Select data {valA, valP} → 在PIPEIP由 SELECTA 决定
- Stat: Compute the status code

写内存 {rrmovq R[%rA]
{push R[%rA]