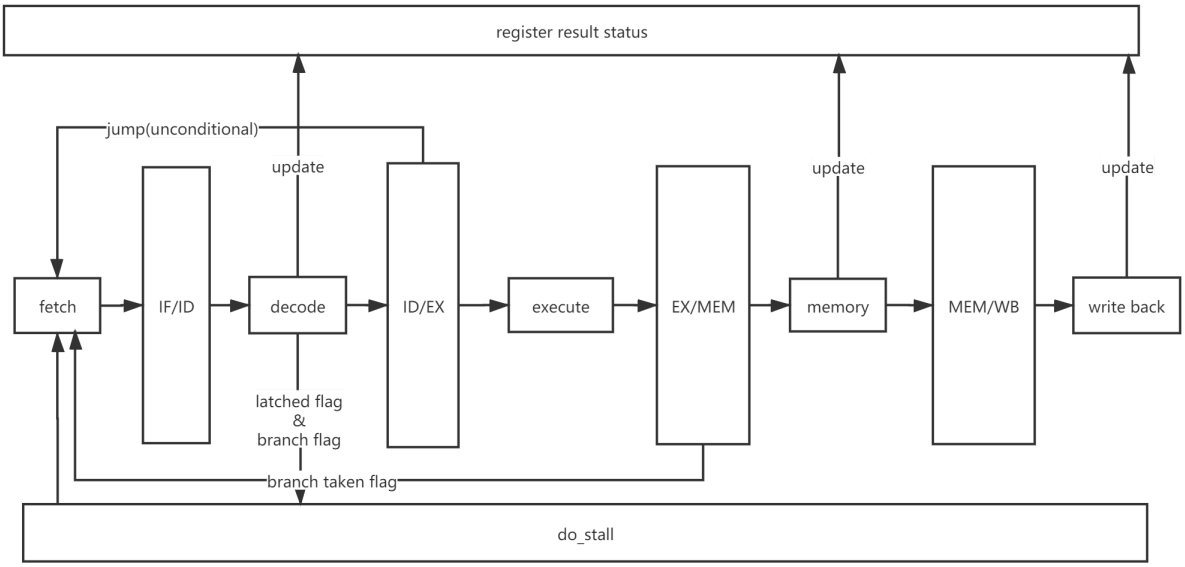
Part2-design.pdf

sim-pipe-withstall

> 结果在 trace-withstall.txt 中，317 个 cycle



Pipeline stages and structures

流水线分为以下5个阶段

1. `IF`：根据是否有跳转以及跳转是否发生来决定下一条指令的地址为跳转地址或者PC+4
2. `ID`：译码阶段，得到下一阶段需要执行的ALU运算以及ALU操作数，判断是否需要stall，是否为跳转指令；
   `LW` 和 `SW` 指令计算地址所需的两个操作数作为ALU操作数被得到，对应的ALU运算为加法
3. `EX`：执行ALU运算，`LW` 和 `SW` 指令的地址计算也会在此阶段发生，同时在此阶段会通过比较得到 `BNE` 是否跳转
4. `MEM`：从内存中 `LW` 或向内存中 `SW`
5. `WB`：写回寄存器，在此阶段根据指令是否为SYSCALL，决定是否结束程序执行

流水线不同阶段之间的信息存在四个buffer中：`ifid_buf`，`idex_buf`，`exmem_buf`，`memwb_buf`， 除此之外，还有一个 `control_buf` 来实时记录将会被写入的寄存器

IF/ID: `ifid_buf`

```
/*define buffer between fetch and decode stage*/
struct ifid_buf {
    md_inst_t inst;     /* instruction that has been fetched */
    md_addr_t PC;       /* pc value of current instruction */
    md_addr_t NPC;      /* the next instruction to fetch */
};
```

`ifid_buf` 定义了 `IF` 和 `ID` 之间的 buffer

ID/EX: `idex_buf`

```
typedef enum {
```

```
        ALU_NOP = 0,            /* NOP: OTHERS */
        ALU_ADD,                /* 加法: LW, SW, ADD, ADDU, ADDU, ADDIU */
        ALU_BNE,                /* 比较: BNE */
        ALU_AND,                /* AND: ANDI */
        ALU_SLT,                /* Set If Less: SLTI */
        ALU_SHTL                /* 移位: SLL, LUI */
    } alu_func_t;

    /* define values related to operands, all possible combinations are i
ncluded */
    typedef struct{
        int in1;            /* input 1 register number */
        int in2;            /* input 2 register number */
        int in3;            /* input 3 register number */
        int out1;           /* output 1 register number */
        int out2;           /* output 2 register number */
    }oprand_t;

    /*define buffer between decode and execute stage*/
    struct idex_buf {
        md_addr_t PC;       /* PC value of the current instruction */
        md_inst_t inst;     /* instruction in ID stage */
        int opcode;         /* can get ALU func code according to opcode
 */
        oprand_t oprand;    /* oprands */
        int latched;        /* whether should be stalled, used in do_stal
l() */
        alu_func_t func;    /* ALU function */
        int instFlags;      /* can get aluA/B, dstE/M and rw flag accordi
ng to this */
        int aluA;           /* ALU A */
        int aluB;           /* ALU B */
        int dstE;           /* dst reg for ALUOut & LW */
        int dstM;           /* mem address of SW */
        int rw;             /* rw=1: SW; rw=2: LW; rw=0: others */
        int branch;         /* if BNE, then 1; else 0 */
        int jump;           /* if JUMP, then 1; else 0 */
        int target;         /* JUMP target */
    };
```

`idex_buf` 定义了 `ID` 和 `EXE` 之间的 buffer：

- `dstE` 是在 `WB` 阶段要写入的寄存器，`LW` 的目标寄存器也存在该值中
- `dstM` 是在 `MEM` 阶段因为 `SW` 指令需要访问的寄存器：将该寄存器中的值写入内存中
- `rw` 与内存操作相关：1表示写入(`SW`)，2表示读取(`LW`)，其他表明不需要对内存操作
- `branch` 表示当前指令是否为有条件的跳转指令
- `jump` 表示当前指令是否为无条件跳转指令
- `target` 只代表无条件跳转的目标地址

EX/MEM: `exmem_buf`

```
/*define buffer between execute and memory stage*/
struct exmem_buf{
    md_addr_t PC;
    md_inst_t inst;     /* instruction in EX stage */
    int ALUOutput;      /* ALU output, while be stored to dstE */
    int dstE;           /* dst reg for ALUOut & LW */
    int dstM;           /* mem address for SW */
    int target;         /* bne */
    int needJump;       /* whether need to jump */
    int rw;             /* rw=1: SW; rw=2: LW; rw=0: others */
};
```

`exmem_buf` 定义了 `EX` 和 `MEM` 之间的 buffer：

- `ALUOutput` 是此阶段执行ALU运算的结果
- `dstE` 是在 `WB` 阶段要写入的寄存器， `LW` 的目标寄存器也存在该值中
- `dstM` 是在 `MEM` 阶段因为 `SW` 指令需要访问的寄存器：将该寄存器中的值写入内存中
- `rw` 与内存操作相关：1表示写入(`SW`)，2表示读取(`LW`)，其他表明不需要对内存操作
- `needJump` 表示有条件跳转指令的跳转是否发生
- `target` 表示的是有条件跳转指令在跳转发生时的目标地址

MEM/WB: `memwb_buf`

```
/*define buffer between memory and writeback stage*/
struct memwb_buf{
    md_addr_t PC;
    md_inst_t inst;     /* instruction in MEM stage */
    int memload;        /* value LW from mem */
    int ALUOutput;      /* ALU output, while be stored to dstE */
    int dstE;           /* dst reg for ALUOut & LW */
    int rw;             /* rw=2: LW; else: others */
};
```

`memwb_buf` 定义了 `MEM` 和 `WB` 之间的 buffer：

- `ALUOutput` 是此阶段执行ALU运算的结果
- `memload` 是从内存中 `LW` 的值
- `dstE` 是在 `WB` 阶段要写入的寄存器， `LW` 的目标寄存器也存在该值中
- `rw` 与内存操作相关，在 `MEM` 阶段 `SW` 指令已经结束，只需要记录是否为 `LW` 指令，使 `WB` 阶段能够判断写入 `dstE` 中的值是从内存中获得的 `memload` 还是 `EX` 阶段计算得出的 `ALUOutput` 即可

`control_buf`

```
struct control_buf{
    int regs;
```

```
};
```

`control_buf` 只有一个成员 `regs` ，以 bitmap 的方式标记了将要写入的寄存器，类似于 scoreboard 中的 `Register Result Status` ，用于在 decode 判断是否存在数据冲突

Function design

sim_main()

下面是程序的主要逻辑部分：

```
    regs.regs_PC = regs.regs_PC - sizeof(md_inst_t);

    while (TRUE) {
        /*start your pipeline simulation here */
        do_stall();
        do_wb();
        do_mem();
        do_ex();
        do_id();
        do_if();
        do_trace();
    }
```

五个阶段的执行顺序与流水线的顺序是相反的，这是为了防止在阶段处理中改变了buffer中的信息影响下一个阶段的执行，其中

- `do_stall()`：处理的是数据冲突和控制冲突
- `do_trace()`：是为了打印流水线每个cycle各个stage处理的指令

do_id()

这个函数对应的是 `decode` 阶段，作用就是对 `ifid_buf` 中的指令进行译码。

- 如果是 `NOP` 指令，函数直接返回

```
    if(de.inst.a == NOP) return;
```

- 如果不是 `NOP` 指令，会首先根据 `machine.def` 中定义的信息获得指令的 `instFlags` 和 输入输出寄存器信息

```
    #define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3)\
    if (OP==de.opcode){\
        de.instFlags = FLAGS;\
```

```
        de.oprand.out1 = O1;\
        de.oprand.out2 = O2;\
        de.oprand.in1 = I1;\
        de.oprand.in2 = I2;\
        de.oprand.in3 = I3;\
        goto READ_OPRAND_VALUE;\
    }
    #define DEFLINK(OP,MSK,NAME,MASK,SHIFT)
    #define CONNECT(OP)
    #include "machine.def"
```

在上面的宏定义中，一旦找到指令的定义，就会跳过之后的 `if` 语句，开始实际的处理过程

- 在译码之前，需要判断与前面正在执行的指令是否存在数据依赖，此处考虑的数据依赖只有 `RAW`
  - 不存在 `WAW`：因为流水线是 `in-order commit` 的，指令的 commit 顺序和 issue 顺序相同
  - 不存在 `WAR`：在实际pipeline中之所以有 `WAR` 是因为同一个cycle中，`ID` 和 `WB` 寄存器可能是同时执行的，倘若先对后issue的指令译码之后，再指令先issue的指令的写回，那么可能就会导致错误的结果
    - 但是因为在我们的仿真器中，虽然 `ID` 和 `WB` 阶段还是会在同一个cycle中执行，但是因为程序的执行是串行的，所以这种冲突在仿真器中并不会出现
  - 不存在 `RAR`：这个是显然的，都是读寄存器的话，不会有冲突
- 如果存在 `RAW` 冲突，那么将 `latched` 设为1之后，直接返回，否则设为0，表示没有冲突，不需要stall

```
    if((de.oprand.in1 != DNA && (ctl.regs & (1 << de.oprand.in1)))  /* 判
断 in1 寄存器上的冲突 */
    ||
    (de.oprand.in2 != DNA && (ctl.regs & (1 << de.oprand.in2)))){   /* 判
断 in2 寄存器上的冲突 */
        de.latched = 1;
        return;
    } else {
        de.latched = 0;
    }
```

- 如果不存在数据冲突，那么我们首先根据指令的 opcode 得到 `EX` 阶段的 ALU 运算的 function code：

```
    switch(de.opcode){
        case LUI:
            de.func = ALU_SHTL;
            break;
        case ADD:
        case ADDU:
        case ADDIU:
```

```
        case LW:        /* SW 和 LW 指令计算内存地址也是加法运算，所以统一用ALU_
ADD */
        case SW:
            de.func = ALU_ADD;
            break;
        case SLL:
            de.func = ALU_SHTL;
            break;
        case SLTI:
            de.func = ALU_SLT;
            break;
        case ANDI:
            de.func = ALU_AND;
            break;
        case BNE:
            /* 表示该指令是有条件的分支跳转指令，只有在EX阶段之后才能确定下一条指令的
地址 */
            de.branch = 1;
            /* BNE指令的 func code 应该是 ADD 或者 SUB，此处自定义了ALU_BNE，对
流水线的执行没有影响 */
            de.func = ALU_BNE;
            de.target = fd.PC + 8 + ((de.inst.b & 0xffff) << 2);
            break;
        case JUMP:
            /* 表示该指令是无条件跳转指令，取指阶段要直接跳转到目标地址 */
            /* 因为不需要等到EX阶段确定下一条指令的地址，所以和BNE做了区分 */
            de.jump = 1;
            de.target = (fd.PC & 0xf0000000) | (TARGI(fd.inst) << 2);
            break;
        default:
            /* 其他指令一定要将 ALU func 清除，否则会影响EX阶段执行 */
            de.func = ALU_NOP;
            break;
    }
```

- 确定 `aluA` / `aluB` 和 `dstE` (写入的寄存器):

```
    /* alu A */
    if(de.instFlags & F_DISP){      /* LW, SW */
        de.aluA = de.oprand.in2 != DNA ? GPR(de.oprand.in2) : 0;
    } else if(de.opcode == LUI){    /* LUI */
        de.aluA = IMMI(de.inst);
    }else {                         /* ADD, ADDU, ADDIU, SLTI, SLL, ANDI
 */
        de.aluA = de.oprand.in1 != DNA ? GPR(de.oprand.in1) : 0;
    }

    /* alu B */
    if(de.instFlags & F_IMM || de.instFlags & F_DISP){      /* ADDIU, AND
I, LW, SW, SLTI, LUI */
        de.aluB = IMMI(de.inst);
```

```
    } else if(de.opcode == LUI){         /* LUI */
        de.aluB = 16;
    }else if(de.func == ALU_SHTL){       /* SLL */
        de.aluB = IMMI(de.inst);
    }else {                              /* ADD, ADDU */
        de.aluB = de.oprand.in2 != DNA ? GPR(de.oprand.in2) : 0;
    }

    /* dstE */
    de.dstE = de.oprand.out1;
    if(de.dstE != DNA) {
        ctl.regs ^= 1 << de.dstE;    /* 在ctl.regs对相应寄存器做标记 */
    }
```

- 因为 `SW` 和 `LW` 指令都是对内存的操作，需要额外分别处理：

```
    /* store */
    if(de.instFlags & F_STORE){
        de.dstM = GPR(de.oprand.in1);    /* 源寄存器 */
        de.rw |= 1;                      /* rw=1表示store */
    } else{
        de.rw &= ~1;
    }

    /* load */
    if(de.instFlags & F_LOAD){
        de.rw |= 2;                      /* rw=2表示load */
    } else{
        de.rw &= ~2;
    }
```

- 至此，`do_id()` 函数完成

do_ex()

- 这个函数的功能很简单，只需要根据 `idex_buf` 中的 ALU func code 和 `aluA/B` 做相应的 ALU 运算，条件跳转是否发生也需要在此判断
- 如果指令为 `NOP`，直接返回
- 如果不是 `NOP`

```
    switch (de.func){
        case ALU_ADD:    /* 显然，LW 和 SW 指令的内存地址也存在了 ALUOutput中 */
            em.ALUOutput = de.aluA + de.aluB;
            break;
        case ALU_AND:
            em.ALUOutput = de.aluA & de.aluB;
            break;
        case ALU_SHTL:
            em.ALUOutput = de.aluA << de.aluB;
```

```
            break;
        case ALU_SLT:
            em.ALUOutput = de.aluA < de.aluB;
            break;
        case ALU_BNE:
            em.needJump = de.aluA != de.aluB;
            break;
        default:
            em.ALUOutput = 0;
            break;
    }

    /* 如果判断得到不需要跳转的话，需要将 branch 清空 */
    /* 因为控制冲突是基于de.branch做判断的，所以如果判断不需要跳转的话，一定要清零
*/
    if(!em.needJump){
        de.branch = 0;
    }
```

do_mem()

- 该阶段从内存中读数据或者向内存中写入数据，取决于 `rw flag` 的值是 1 还是 2

```
    switch(em.rw){
        case 1: /* store：从 dstM 寄存器取数据，存到内存 ALUOutput 的位置上 */
            WRITE_WORD(em.dstM, em.ALUOutput, _fault);
            if(_fault != md_fault_none)
                DECLARE_FAULT(_fault);
        case 2: /* load：将内存地址为 ALUOutput的值读出来放在 memload 中 */
            mw.memload = READ_WORD(em.ALUOutput, _fault);
            if(_fault != md_fault_none)
                DECLARE_FAULT(_fault);
        default:
            break;
    }
```

do_wb()

- 写回寄存器
- **SYSCALL 指令需要在这个阶段处理：是为了保证前面的指令都已经处理完成**

```
    if(mw.dstE != DNA){
        ctl.regs ^= 1 << mw.dstE;    /* !!!: 此处一定要将ctl.regs中对应标记清除
*/
        if(mw.rw & 2){          /* 如果是 LW 指令，那么写入寄存器的值在 memload
中 */
            SET_GPR(mw.dstE, mw.memload);
        } else{
```

```
            SET_GPR(mw.dstE, mw.ALUOutput);
        }
    }
    if(mw.inst.a == SYSCALL){       /* 如果是 SYSCALL 指令，那么程序执行结束
 */
        mw.inst = MD_NOP_INST;
        do_trace();                 /* 结束之前需要将最后这个cycle的状态打印出
来 */
        SYSCALL(mw.inst);
    }
```

do_if()

- fetch 下一条指令，需要考虑跳转的情况

```
    if(em.needJump){        /* 如果 BNE 指令需要跳转 */
        regs.regs_PC = em.target;
        em.needJump = 0;
        de.branch = 0;
    } else if(de.jump) {    /* 如果是 JUMP 指令需要跳转 */
        regs.regs_PC = de.target;
        de.jump = 0;
    } else {                /* PC+4 */
        regs.regs_NPC = regs.regs_PC + sizeof(md_inst_t);       /* regs.h
 */
        regs.regs_PC = regs.regs_NPC;
    }
    fd.PC = regs.regs_PC;
    MD_FETCH_INSTI(fd.inst, mem, fd.PC);
```

do_stall()

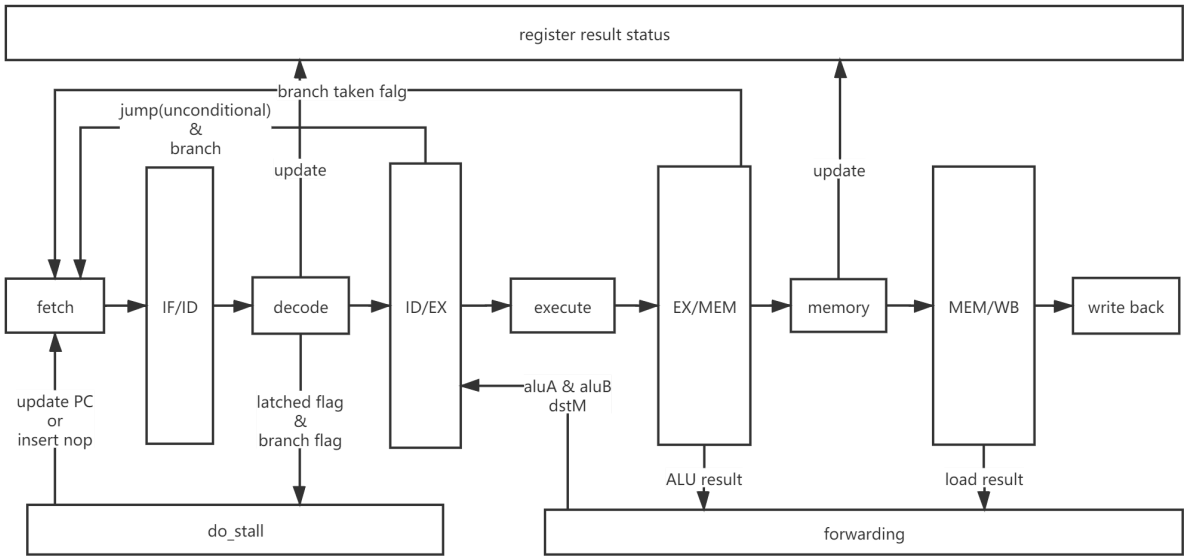
- 处理了控制冲突和数据冲突

```
    if(de.branch){      /* BNE 指令需要 stall */
        regs.regs_PC = de.PC;
        fd.PC = regs.regs_PC;
        fd.inst.a = NOP;
    }

    if(de.latched){     /* decode 阶段判断出数据冲突需要stall */
        regs.regs_PC = de.PC;   /* 退后了 fetch 阶段的PC，使PC+4仍是当前的那条
指令，相当于stall */
        fd.PC = regs.regs_PC;
        fd.inst = de.inst;      /* 在do_id()中，de.inst = fd.inst，使decode
阶段再次对指令译码，相当于stall */
        de.inst.a = NOP;        /* EX 阶段执行 NOP */
    }
```

sim-pipe-withoutstall

> 结果在 trace.txt 中，166 个 cycle



- 将withstall的版本设计好之后，加上转发和分支预测即可，下面只介绍更改的部分

Pipeline stages and structures

流水线仍然分成相同的五个 stage

流水线不同阶段之间的 buffer 与上面的设计相同：`ifid_buf`，`idex_buf`，`exmem_buf`，`memwb_buf`，`control_buf` 同样被用于实时记录将要被更新的寄存器

其中 `idex_buf` 、 `control_buf` 有稍有改动

ID/EX: `idex_buf`

```
/*define buffer between decode and execute stage*/
struct idex_buf {
    int srcA;
    int srcB;
};
```

- 上面是新增的两个成员属性，`srcA` 表示的是 `aluA` 来自哪个寄存器，如果不存在 `aluA` 或者 `aluA` 是指令中解析出来的立即数，那么 `srcA` 为 `DNA`
- 同样的，`srcB` 对应的则是 `aluB`

`control_buf`

- 因为我们加上了转发机制，那么只有 `LW` 指令会导致数据冲突，所以我们只需要在 `regs` 中记录 `LW` 指令更新的寄存器即可

Function design

sim_main()

- 程序的主逻辑不变，在 `do_id()` 之后增加了 `do_forward()` 用于 forwarding

```
    /* set up initial default next PC */
    regs.regs_PC = regs.regs_PC - sizeof(md_inst_t);
    /* maintain $r0 semantics */
    regs.regs_R[MD_REG_ZERO] = 0;

    while (TRUE) {
        /*start your pipeline simulation here */
        do_stall();
        do_wb();
        do_mem();
        do_ex();
        do_id();
        do_forward();
        do_if();
        do_trace();
    }
```

- 放在 `do_id()` 后面执行转发的原因很明显：在 `ID` 译码阶段确定了操作数来源之后，就可以通过转发获得操作数

do_id()

- 同样的，获取指令的定义之后，会首先判断是否存在不可以利用转发解决的数据依赖，如果存在的话，直接返回，保证继续 decode 的指令的操作数一定是可以获得的：要么是立即数，要么通过转发可以获得
- 改变的地方是：

```
    /* src A & alu A */
    if(de.instFlags & F_DISP){      /* LW, SW */
        de.srcA = de.oprand.in2;
    } else if(de.opcode == LUI){
        de.srcA = DNA;
        de.aluA = IMMI(de.inst);
```

```
    }else {                         /* ADD, ADDU, ADDIU, SLTI, SLL, ANDI
*/
        de.srcA = de.oprand.in1;
    }

    /* src B & alu B */
    if(de.instFlags & F_IMM || de.instFlags & F_DISP){    /* ADDIU, ANDI,
LW, SW, SLTI, LUI */
        de.srcB = DNA;
        de.aluB = IMMI(de.inst);
    } else if(de.opcode == LUI){
        de.srcB = DNA;
        de.aluB = 16;
    }else if(de.func == ALU_SHTL){   /* SLL */
        de.srcB = DNA;
        de.aluB = IMMI(de.inst);
    }else {             /* ADD, ADDU */
        de.srcB = de.oprand.in2;
    }
```

- 如上面代码所示，加上转发机制之后，可以先将源操作数的寄存器用 `srcA` 和 `srcB` 保存下来，等待 `do_forward()` 转发得到具体数值

do_stall()

```
    if(de.branch && em.needJump){   /* 因为始终预测 BNE 不会跳转，所以如果发现
预测错误，就需要将当前 ID 阶段的指令设为无效指令 */
        de.inst = MD_NOP_INST;
    }

    if(de.latched){         /* 数据冲突与之前的处理方式相同 */
        regs.regs_PC = de.PC;
        fd.PC = regs.regs_PC;
        fd.inst = de.inst;
        de.inst = MD_NOP_INST;
    }
```

do_if()

```
    if(em.needJump){         /* 与之前的处理相同 */
        regs.regs_PC = em.target;
        em.needJump = 0;
        de.branch = 0;
    } else if(de.jump) {
        regs.regs_PC = de.target;
        de.jump = 0;
    } else {                /* 包含了预测分支不会跳转的情况 */
```

```
            regs.regs_NPC = regs.regs_PC + sizeof(md_inst_t);       /* regs.h
*/
            regs.regs_PC = regs.regs_NPC;
        }
        fd.PC = regs.regs_PC;
        MD_FETCH_INSTI(fd.inst, mem, fd.PC);
```

do_forward()

```
        forward(&de.aluA, &de.srcA);
        forward(&de.aluB, &de.srcB);
        forward(&de.dstM, &de.oprand.in1);
```

- 需要转发得到数值的是 `aluA` 、 `aluB` 和 `SW` 指令内存基地址
- 具体转发函数的实现如下：

```
        void forward(int *val, int *src){
            if(*src == DNA){        /* 如果 src 为 DNA，表示不需要转发 */
                return;
            }

            if(*src == em.dstE){    /* dstE 可能有 ALU 算数运算更新或 LW 指令更新
*/
                if(!(em.rw & 2)){   /* LW 指令在 MEM 阶段之后才可以确定具体的数值
*/
                *val = em.ALUOutput;
                return;
                }
            }

            if(*src == mw.dstE){
                if(mw.rw & 2){      /* LW 指令更新 */
                    *val = mw.memload;
                } else {            /* 非 LW 指令 */
                /*实际不会被执行到，需要对 ALUOutput 转发的话，在前一个周期中已经处理了
*/
                    *val = mw.ALUOutput;
                }
                return;
            }

            *val = GPR(*src);   /* 如果前面都不是的话，那么简单的从寄存器中将数值读出
*/
            /* 这实际不是转发，但是放在这儿可以简化分情况处理 */
        }
```

Else

其他的处理与 withstall 的情况基本相同

改进：ID阶段判定BNE分支的版本

> 结果在 bonus/trace.txt 中，156 个 cycle，因为将分支预测提前至 ID，不会因为分支预测错误浪费一个 cycle

基于上面的过程，只需要对代码做很小的改动即可：

在 `do_id()` 中将对 `BNE` 指令的处理改为：

```
    de.branch = 1;
    de.srcA = de.oprand.in1;
    de.srcB = de.oprand.in2;
    de.target = fd.PC + 8 + ((de.inst.b & 0xffff) << 2);
    do_forward();       /* !!!: 做一次转发，得到需要比较的两个操作数 */
    de.jump = de.aluA != de.aluB;
    /* 同样因为之前判断过是否存在不可以通过转发解决的数据依赖，此处一定是可以获得的 */
```

除此之外，不需要另加额外的处理