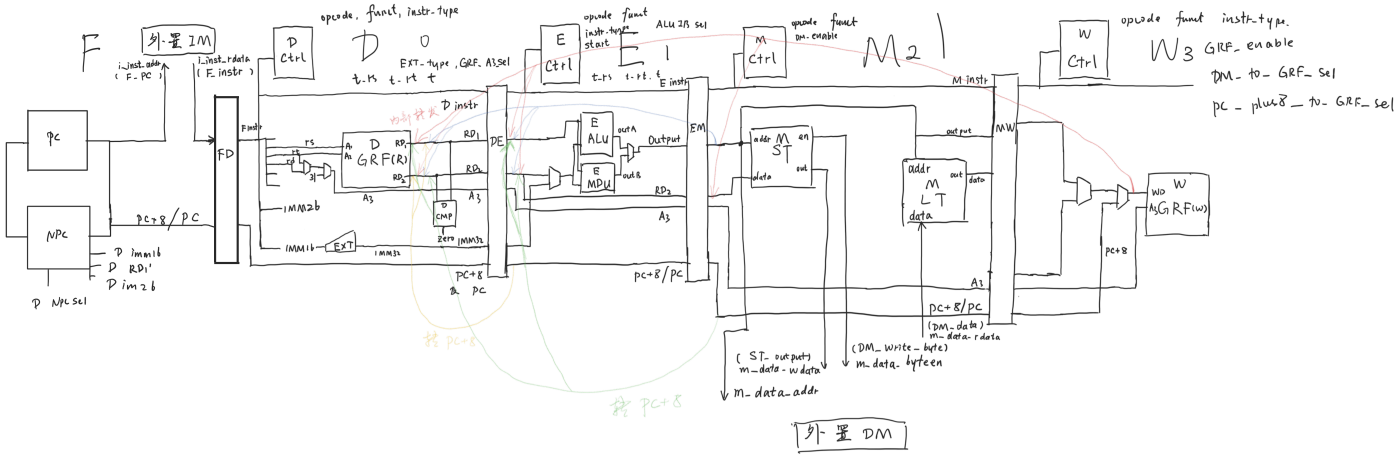
设计草稿

已实现指令:

```
add, sub, and, or, slt, sltu, lui
addi, andi, ori
lb, lh, lw, sb, sh, sw
mult, multu, div, divu, mfhi, mflo, mthi, mtlo
beq, bne, jal, jr
```

设计架构



修改 NPC

由于延迟槽的存在，在执行 `beq` `bne` 等时，其在 D 级才会判断是否发生跳转。此时 F 级的 PC 为 D 级的 PC + 4。故当 D 级确定发生跳转时，NPC 只需要计算 F_PC + offset || 2'b00 即可，因为 F_PC = D_PC + 4。

将 PC + 8 流水至 W 级

由于延迟槽的存在，执行 `jal` `jalr` 等需要把 PC + 8 存入 GRF。这时就需要将 PC + 8 流水到 W 级。

修改 Controller

定义 D E M W 级编号为 0 1 2 3，相邻流水级之间时间差为 1。
某级到 D 级的时间差为其编号。

新增端口如下：

名称	类型	方向	位宽	备注
t_rs	wire	O	3:0	指令在 D 级，再经过多少时间使用 GPR[rs]，缺省为 4'hf
t_rt	wire	O	3:0	指令在 D 级，再经过多少时间使用 GPR[rt]，缺省为 4'hf
t	wire	O	3:0	能产生新数据的指令在 D 级， 再经过多少时间算出新数据并能转发，缺省为 4'hf

转发需要在流水级寄存器后。

为每个流水级实例化一个 Controller，内部计算每种指令的 `t_rs`，`t_rt` 和 `t`。
例如：

- `add`：
  `t_rs` = 4'h1（E 级 ALU 需要使用）
  `t_rt` = 4'h1（E 级 ALU 需要使用）
  `t` = 4'h2（EMReg 后可以转发）
- `beq`：
  `t_rs` = 4'h0（D 级 CMP 需要使用）
  `t_rt` = 4'h0（D 级 CMP 需要使用）
  `t` = 4'hf（不产生新数据）

提取 CMP 至 D 级

从 ALU 中分离 zero 的运算逻辑，独立为一个 CMP 模块，放在 D 级。这是由于延迟槽的存在，beq 等相对跳转指令可以在 D 级计算是否需要跳转，让下一条指令（延迟槽）进入 F 级。故需要把 zero 运算逻辑提前到 D 级。

名称	类型	方向	位宽	备注
inputA	wire	I	31:0	第一个输入
inputB	wire	I	31:0	第二个输入
type	wire	I	5:0	指令种类信号，用以决定 CMP 比较逻辑，与 Controller 中定义一致

名称	类型	方向	位宽	备注
zero	wire	O	1	为 1 时发生相对跳转

设计 MDU

MDU 与 ALU 并列存在于 E 级，负责计算和乘除法相关指令，以及读写 HI 和 LO 两个寄存器。端口定义如下：

名称	种类	位宽	方向	备注
clk	wire	1	I	时钟信号
reset	wire	1	I	同步复位信号
inputA	wire	31:0	I	第一个操作数
inputB	wire	31:0	I	第二个操作数
instr_type	wire	5:0	I	指令种类
start	wire	1	I	start 为 1 时，MDU 开始运算，并将 busy 置为 1 若干周期
outputB	wire	31:0	O	MDU 输出结果，视情况与 ALU 输出的 outputA 进行选择
busy	wire	1	O	MDU 使用中信号。当其为 1 时，阻塞所有需要用到 MDU 的指令

修改 ST

ST 是我原来的为 DM 设计的预处理模块。因为 DM 外置，因此需要进行修改。将写入 DM 的使能信号改为 4 位宽，分别对应 4 个字节的写入使能，这样就能实现 `sb` `st` 等功能。

转发

需求者：D 级 RD1 和 RD2（给 CMP）。E 级 RD1 和 RD2（给 ALU）。

供给者：E 级 PC + 8（jal 等），M 级 PC + 8（jal 等），EM 寄存器 outputA（来自 ALU），WD（包括 PC + 8 等多种可能）。

转发路线：从后往前的所有组合情况。

tuse 和 tnew

- X 级 的 tuse：指令在 X 级，还剩多少时间要使用 rs 或 rt。指令每经过一级，该数字要减一。
- X 级的 tnew：指令在 X 级，还剩多少时间可以转发新算出的数据。指令每经过一级，该数字要减一。

它们均等于某指令的 `t_rs` / `t_rt` / `t` 减掉 X 级的编号。即到 D 级的时间差。

判断转发条件

1. 要发送的是否是 PC + 8
2. 起点流水级携带的目标寄存器不是 $0
3. 起点流水级携带的目标寄存器与终点的寄存器为同一个
4. 终点 tuse，起点 tnew 存在
5. tuse >= tnew，即后一条指令使用时间晚于或等于前一条指令产生时间

阻塞

实现方式：

1. 保持 PC：PC_en 置为 0。
2. 保持 FD 寄存器：FD_en 置为 0。
3. 插入 `nop` ：清空 DE 寄存器，DE_clr 置为 1。

判断阻塞条件

1. 起点流水级携带的目标寄存器不是 $0
2. 起点流水级携带的目标寄存器与终点的寄存器为同一个
3. 终点 tuse，起点 tnew 存在
4. tuse < tnew，即后一条指令使用时间早于前一条指令产生时间

当一条指令进入 D 级时即可判定需不需要阻塞，故只需要看转发到 D 级的可行性即可。

清空延迟槽

这在 RTL 中常表现为一个名为 `NullifyCurrentInstruction()` 的函数。
当满足清空延迟槽的条件时（比如 CMP 来判断是否需要清空延迟槽），需要引出一个信号 `null_signal` 。当**非阻塞**且**清空延迟槽**时，清空 FD 寄存器。
即：

```
assign FD_clr = (!stall && null_signal) ? 1 : 0;
```

GRF 写使能悬空

一条指令，在运算之后才能判断它最终是否要进行写寄存器的操作（比如在 CMP 进行比较或在 ALU 运算），则需要引一个信号一直流水到 W 级。用这个信号结合使能信号来判断最终是否允许写 GRF。对于悬空的 t 值等，同样需要用这个信号来判断存在与否。

或者检测到这个信号之后，将写入地址之间换成 0 号寄存器，流水到 W 级。

GRF 写目标悬空

一条指令，在运算后才能得知最终写入的寄存器是哪一个。一般来说，是在 M 级算出这个地址（最大化减小阻塞的出现），在 M 级定义一个 `M_exact_write_addr`，替换原来的 `M_write_addr`，将其流水到 W 级。

然后修改转发路径中所有用到了 `M_write_addr` 的地方，改为 `M_exact_write_addr`。注意如果要写的内容和 PC + 8 无关，那么关于 PC + 8 的转发路径中的 `M_write_addr` 不要改。

阻塞的方法

然后为阻塞增加条件。由于运算地址是 M 级算出，那么该指令在 E 级时无从得知要写入哪个寄存器。这时后对于 D 级的指令，如果不是 `j` `jal` 这种不用寄存器的，则必须考虑是否要将其阻塞到 D 级。一般来说可以暴力阻塞（如果 E 级指令是xxx，则 `stall` 置为 1），或着考虑 D 级可能用到哪些会冲突的寄存器，按情况阻塞（比如已知 E 级指令要用到的只可能是奇数号寄存器，那么当 D 级指令要用奇数号寄存器时阻塞，用偶数时可以不阻塞）。
该指令在 M 级的时候，阻塞的判断条件中要用 `M_exact_write_addr`。

思考题

1. 为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？
   答：乘除法运算时间长，且计算结果不需要立刻写入 GRF 或 DM 中。如果整合进 ALU，效率低下。HI 和 LO 独立保存乘除法结果，不会立即使用，即在计算中不影响 ALU 的正常流水。
2. 真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明。
   答：首先CPU会初始化三个通用寄存器用来存放被乘数，乘数，部分积的二进制数，部分积寄存器初始化为 0。然后在判断乘数寄存器的低位是低电平还是高电平：
   如果为低（0）则将乘数寄存器右移一位，同时将部分积寄存器也右移一位，乘数寄存器低位溢出的一位丢弃，部分积寄存器低位溢出的一位填充到乘数寄存器的高位，同时部分积寄存器高位补 0。如果为 1 则将部分积寄存器加上被乘数寄存器，在进移位操作。
   当所有乘数位处理完成后部分积寄存器做高位乘数寄存器做低位就是最终乘法结果。
3. 请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的？
   答：首先 `busy` 信号只影响进入 MDU 的指令。故需要在 Controller 输出一个判断当前指令是否

为进入 MDU 指令的信号 `D_is_mult`。MDU 负责根据正在起作用的指令进行倒计时，计时结束前 busy 均为 1，用 `(D_ismult && (E_start || E_busy))` 这个条件来判断是否进行阻塞。

4. 请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）
   答：首先第 n 位使能信号为 1 就表明第 n 字节需要写入，形成一一映射关系，很清晰。其次对于所有关于按字节/半字写入的指令，都有统一的提供使能信号的方式，提高了扩展性。
5. 请思考，我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？
   答：不是。需要高频进行按字节读写时效率更高。
6. 为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？
   答：严格按照流水级进行信号的命名，这样方便在信号多的时候分清楚级次。为易混淆的信号增加特殊的命名，比如写入 DM 的数据和 写入 GRF 的数据都能用 `data` 表示，就分别用 `DM_data GRF_data` 表示。
7. 在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？
   答：
   两条计算指令紧邻时的情况，转发：

```
addi $t0, $t0, 0x00000001
ori $t0, $t0, 0x00000000
```

一条访存类指令紧跟一条计算类指令的情况，阻塞：

```
lw $t0, 0($0)
addi $t0, $t0, 0x00000001
```

8. 如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证**覆盖**了所有需要测试的情况；如果你是**完全随机**生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了**特殊的策略**，比如构造连续数据冒险序列，请你描述一下你使用的策略如何**结合了随机性**达到强测的效果。
   答：使用随机生成的测试数据，可能出现阻塞，转发的触发情况测试不足等问题，需要自行增加一些相关的测试。

测试

```
# ========================================
# 1. 初始化与基础运算
# ========================================
addi $s0, $zero, 0      # $s0 用于累积校验和 (Checksum)
lui  $t0, 0x0000        # $t0 = 0 (Data Base Address)

addi $t1, $zero, 10     # $t1 = 10
addi $t2, $zero, 20     # $t2 = 20

# 测试 add/sub/or/and
add  $t3, $t1, $t2      # $t3 = 30
sub  $t4, $t3, $t1      # [Hazard: EX-EX] $t4 = 30 - 10 = 20
or   $t5, $t1, $t2      # $t5 = 10 | 20 = 30
and  $t6, $t1, $t2      # $t6 = 10 & 20 = 0 (1010 & 10100) -> wait, 10=0xA, 20=0x14. 010

add  $s0, $s0, $t3      # Checksum += 30
add  $s0, $s0, $t4      # Checksum += 20

# ========================================
# 2. 乘除法与 HI/LO (MD Extension)
# ========================================
# 测试 mult (有符号)
addi $t7, $zero, -2     # $t7 = -2
mult $t1, $t7           # 10 * (-2) = -20. Hi=FFFFFFFF, Lo=FFFFFFEC

# 测试 mflo 在 mult 后立即读取 (应该 Stall 或正确读取)
mflo $t8                # $t8 = -20
mfhi $t9                # $t9 = -1

add  $s0, $s0, $t8      # Checksum += -20

# 测试 divu (无符号)
# $t2 = 20, $t1 = 10
addi $k0, $zero, 3
divu $t2, $k0           # 20 / 3 = 6 ... 2

mflo $s1                # $s1 = 6 (Quotient)
mfhi $s2                # $s2 = 2 (Remainder)

add  $s0, $s0, $s1      # Checksum += 6
add  $s0, $s0, $s2      # Checksum += 2

# 测试 mtlo/mthi/mflo/mfhi 数据传递
```

```
    addi $k1, $zero, 100
    mtlo $k1
    mflo $k1                # $k1 应该读回 100
    add  $s0, $s0, $k1      # Checksum += 100

    # ========================================
    # 3. 内存访问与 Load-Use 冒险
    # ========================================
    # 准备数据
    addi $a0, $zero, 0x1234
    sll  $a0, $a0, 16       # $a0 = 0x12340000
    ori  $a0, $a0, 0x5678   # $a0 = 0x12345678
    sw   $a0, 0($t0)        # Mem[0] = 0x12345678

    # 测试 SB (Store Byte)
    addi $a1, $zero, 0xFF
    sb   $a1, 4($t0)        # Mem[4] 低字节 = FF

    # 测试 Load-Use Stall
    lw   $a2, 0($t0)        # $a2 = 0x12345678
    add  $a3, $a2, $a2      # [Load-Use Hazard] 必须 Stall. $a3 = ...F0

    # 测试 LB (Load Byte Signed)
    lb   $a3, 4($t0)        # Mem[4] 是 0xFF, lb 扩展为 0xFFFFFFFF (-1)
    add  $s0, $s0, $a3      # Checksum += -1

    # ========================================
    # 4. 控制流与延迟槽
    # ========================================
    # 测试 BEQ (Not Taken)
    beq  $t1, $t2, fail_jump
    addi $s0, $s0, 1        # [Delay Slot] 必须执行 Checksum += 1

    # 测试 JAL
    jal  my_func
    addi $s0, $s0, 2        # [Delay Slot] 必须执行 Checksum += 2

    # 返回后继续
    j    finish
    nop

fail_jump:
    addi $s0, $zero, 0xBAD  # 错误标记
    j    finish
    nop
```

```
my_func:
    # 测试 JALR
    addi $s0, $s0, 10       # Checksum += 10
    jalr $ra
    addi $s0, $s0, 5        # [Delay Slot] 必须执行 Checksum += 5 (在跳转动作发生前执行)

finish:
    # 最终结果写回 Mem[0x50] (80)
    # 预期 Checksum 计算:
    # Init: 0
    # ALU: +30, +20 = 50
    # MD: -20, +6, +2, +100 = 88 (Accum: 138)
    # MEM: -1 = 137
    # Branch Slot: +1 = 138
    # JAL Slot: +2 = 140
    # Func: +10 = 150
    # JALR Slot: +5 = 155 (0x9B)

    addi $t9, $zero, 0x50
    sw   $s0, 0($t9)        # 将校验和写入地址 0x50

    # 结束，不再进行任何操作，等待 Testbench 捕获
    nop
    nop
    nop
    nop
```