Computer System hw II

1 Pipeline Design Concept

A

1. A 2. A 3. A 4. A 5. B

B

对于流水线CPU，各个阶段所需要花费的时间为：

- IF: 100ps
- ID: 100ps
- EXE: 30ps
- MEM: 500ps
- WB: 100ps

所以流水线的最小时钟周期取决于花费时间最长的阶段:

$$Clock_{pipeline} = clk_{MEM} = 500ps$$

流水线CPU执行一条指令所需的时间为：

$$Time\_Cost_{pipeline} = 5 \times Clock_{pipeline} = 2500ps$$

对于单周期CPU，时钟周期等于各个阶段的时间花费总和：

$$Clock_{single} = \sum_{i=1}^{5} clk_i = 830ps$$

执行一条指令所需的时间:

$$Time\_Cost_{single} = Clock_{single} = 830ps$$

综上所述，流水线CPU相比于单周期CPU实现的指令加速比为：

$$Speed\_Up = \frac{Clock_{single}}{Clock_{pipeline}} = \frac{830ps}{500ps} \times 100\% \approx 166\%$$

C

由B可知，在取值指令走最长路径的情况下，各个阶段的时间花费为：

- IF: 100ps
- ID: 100ps
- EXE: 30ps
- MEM: 500ps
- WB: 100ps

可以看到在时钟周期为500ps的情况下，EXE阶段只有30ps实在浪费，并且ID与EXE在功能上相近，所以可以将ID阶段与EXE阶段合并为ID/EXE。

但在这种情况下我们需要考虑forwarding机制的数据前递过程。

合并之后，Pipeline的时钟周期长度不变，阶段变为4个，执行一条指令所需要的时间为

$$Time\_Cost_{pipeline} = 4 \times 500ps = 2000ps$$

时钟周期长度还是取决于花费时间最长的阶段$Clock_{pipeline} = 500ps$

在单周期CPU中，时钟周期保持不变$Clcok_{single} = 830ps$

指令加速比为：

$$Speed\_Up = \frac{Clock_{single}}{Clock_{pipeline}} = \frac{830ps}{500ps} \times 100\% \approx 166\%$$

2 Data Hazard and Forwarding

A

源代码如下：

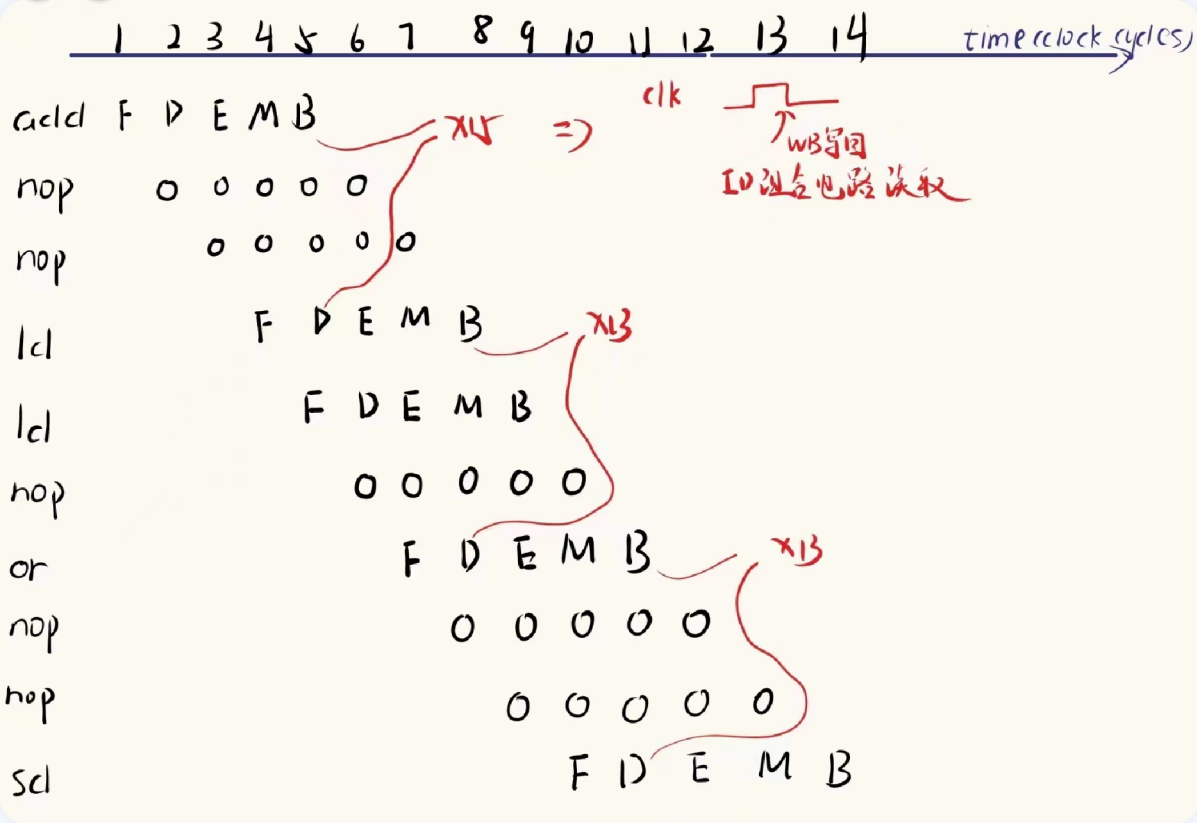
```
add x15, x12, x11 ; 1
ld x13, 4(x15)    ; 2
ld x12, 0(x2)     ; 3
or x13, x15, x13  ; 4
sd x13, 0(x15)    ; 5
```

可以看到，1指令与2指令之间，2指令与4指令，4指令与6指令间同样存在数据冲突。插入nop指令后的代码段如下

```
add x15, x12, x11 ; 1
nop
nop
ld x13, 4(x15)    ; 2
ld x12, 0(x2)     ; 3
nop
or x13, x15, x13  ; 4
nop
nop
sd x13, 0(x15)    ; 5
```

但是对于这段代码而言，我无法通过任何的指令重拍减少nop数，所以nop数为5且不可减少。（Reg下降沿写回）

时序表如下：



B

IF	ID	EX(no FW)	EX(full FW)	EX(FW from EX/MEM)	EX(FW from MEM/WB)	MEM	WB
200ps	100ps	110ps	130ps	120ps	120ps	200ps	100ps

Table 2: Execution time of different stage

EX to 1st only	MEM to 1st only	EX to 2nd only	MEM to 2nd only	EX to 1st and EX to 2nd
5%	20%	5%	10%	10%

Table 1: fowarding instructions partition.

- Q1: 在full forwarding的情况下，我们仍需对ld-use的指令进行stall。在表1中可知，有$20\% + 10\%$的指令是MEM前递，这些指令一定是$ld - use$类型前递，但此时只考虑MEM to 1st的stall，需要对他们进行一拍的stall。

在full forwarding且没有控制跳转的情况下，设指令条数为n，当$n \to \infty$时, 则总cycle数为

$$Cycle_{all} = Cycle_{normal} + Cycle_{stall} = n + n \times (20\%) = 1.2n$$

所以，$Cycle_{stall}$占比为：

$$ans = \frac{Cycle_{stall}}{Cycle_{all}} \times 100\% \approx 17\%$$

- Q2: 在hw所设计的五级流水线中，由于WB阶段Reg下降沿写回，所以ld指令的数据forward只可能在MEM/WB阶段Forwarding。**等到下一个时钟周期MEM中读出的数据已经在时钟下降沿写回寄存器，不需要前递了。**

> 观察下面两段代码：
>
> ```
> ld x1, 8(x4)
> addi x2, x1, 4
> addi x1, x1, 5
> ```
>
> 此时MEM to 1st和MEM to 2nd均由触发， 但由于我们MEM to 2nd不需要stall，那么这种情况等同于MEM to 1st。
>
> stall一拍后，ld指令取出的数据同时传给1st和2nd，此时，我们可以把MEM to 1st and MEM to 2nd的情况合并到MEM to 1st

综上所述，MEM to 1st and MEM to 2nd没有必要存在。

C

对于五级流水线来说他的时钟周期长度是恒定的，取决于最长的阶段-MEM阶段所需要的时间。

故：

$$Clock = 200ps$$

下面我们分别分析这两种情况（forwarding only EXE/MEM， forwarding only MEM/WB）下的CPI：

（设所需运行的指令数为n，且$n \to \infty$）

Forwarding only EXE/MEM

EX to 1st only	MEM to 1st only	EX to 2nd only	MEM to 2nd only	EX to 1st and EX to 2nd
5%	20%	5%	10%	10%

Table 1: fowarding instructions partition.

根据表1，如果仅在EXE/MEM处forwarding，那么只有EX to 1st，EX to 2nd两种指令可能执行。其余指令的stall情况如下：

> - 对于单EXE to的指令，若是在MEM/WB中的则stall 1拍，若是在EXE/MEM中则不用stall
> - 所有的MEM to 1st， 均stall 2拍（一拍取数据，一拍传入写回）MEM to 2nd stall一拍
> - 所有的EX to 1st and EX to 2nd均stall一拍（处于EXE/MEM的可以forwarding，处于MEM/WB的需要Core等待其写入）

当$n \to \infty$时，总cycle数为：

$$\begin{aligned} Cycle_{all} &= Cycle_{MEM} + Cycle_{EXE} + Cycle_{normal} \\ &= 2 \times n \times 20\% + 5\% \times n + 10\% \times n + 10\% \times n + n \\ &= 1.65n \end{aligned}$$

CPI为：

$$CPI_1 = \frac{Cycle}{n} = 1.65$$

即，$CPI_1 = 1.65$

Forwarding only MEM/WB

仅在MEM/WB阶段forwarding时，我们可以确定所有的MEM to指令均可以forwarding，但MEM to 1st指令仍需一个时钟周期将数据读出。

> 此时，对于EXE to 1st and EXE to 2st的指令，我们需要stall 1拍。

> 因前一条指令的result停在MEM/WB寄存器，可以forwarding，但于此同时，停在EXE/MEM寄存器的result不能forwarding，所以此时，我们需要在EXE/MEM处stall一拍，并在EXE/MEM 处插入一个bubble，让MEM/WB的结果写回，然后EXE/MEM的result传入MEM/WB阶段间寄存器，将问题转化为单EXE to，这样就可以forwarding了。
>
> 所以对于双EXE to的指令，我们需要stall 1拍。

综上所述，当$n \to \infty$时，总cycle数为：

$$\begin{aligned} Cycle_{all} &= Cycle_{EXE} + Cycle_{MEM} + Cycle_{normal} \\ &= 5\% \times n + 10\% \times n + 20\% \times n + n \\ &= 1.35n \end{aligned}$$

所以CPI为：

$$CPI_2 = \frac{Cycle_{all}}{n} = 1.35$$

即,$CPI_2 = 1.35$

综上，Forwarding only MEM/WB更好

3 Control Hazard and Branch Prediction

A

```
int A[ 6 ] = {3 , 7 , 4 , 9 , 2 , 1 } ;
    int a = 0 ;
    for ( i =0; i <6; i ++){
    if (A[ i ]<5)
        a = a+1;
    else
        a = a+2;
}
```

将这段C代码手搓成Risc-v汇编

```
.global main
main:
    li t0, 0          # t2 寄存器用于累加 a 的值
    li t1, 0          # t1 寄存器用于索引 i 初始化为 0
    li t2, 0x800000   # 数组起始地址

loop:
    bge t1, 6, end_loop  # 如果 i ≥ size 跳转到循环结束
```

```
    lw t4, 0(t2)            # 加载 A[i] 的值到 t4
    addi t2, t2, 4          # 迭代下一次元素
    blt t4, 5, less_than_five  # 如果 A[i] < 5 跳转到 less_than_five

    # A[i] ⩾ 5 的情况
    addi t0, t0, 2          # a += 2
    j continue_loop         # 跳转到 continue_loop

less_than_five:
    addi t0, t0, 1          # a += 1

continue_loop:
    addi t1, t1, 1          # i++
    j loop                  # 跳回循环的开始

end_loop:

    # 程序结束或返回
```

以下数据均为6次迭代后的结果

Always-Taken

共判断错误45次，判断正确5次

正误比为$\frac{5}{45}$

Always-not-Taken

共判断错误5次，判断正确45次

正误比为$\frac{45}{5}$

综上，Always-Not-Taken策略更加精准。

B

根据题意，我们先将代码段提取出来然后根据从0,1,2,3,4,5,6,7,0,1....的顺序进行编号

```
.global main ;int A[ 6 ] = {3 , 7 , 4 , 9 , 2 , 1 } ;
main:
    li t0, 0         # 0
    li t1, 0         # 1
    li t2, 0x800000  # 2

loop:
```

```
    bge t1, 6, end_loop  # 3
    lw t4, 0(t2)         # 4
    addi t2, t2, 4       # 5
    blt t4, 5, less_than_five  # 6

    # A[i] ⩾ 5 的情况
    addi t0, t0, 2       # 7
    j continue_loop      # 0

less_than_five:
    addi t0, t0, 1       # 1

continue_loop:
    addi t1, t1, 1       # 2
    j loop               # 3

end_loop:
```

我们定义00为强跳，01为若跳，10为弱不跳，11为强不跳

现在根据指令运行情况填写下表：

entries	0	1	2	3	4	5	6
0	start->00->01	01->10	10	10->11	11	11	11
1	00->01->10	10	10->11	11->11	11->11	11->11	11
2	00->01->10	10->11	11->11	11->11	11->11	11->11	11
3	00->01	01->10->11	11->11->11	11->11->11	11->11->11	11->11->11	11->11->10->end
4	00->01	01->10	10->11	11->11	11->11	11->11	11
5	00->01	01->10	10->11	11->11	11->11	11->11	11
6	00->00	00->01	01->00	00->01	01->00	00->00	00
7	00	00->01	01	01->10	10	10	10

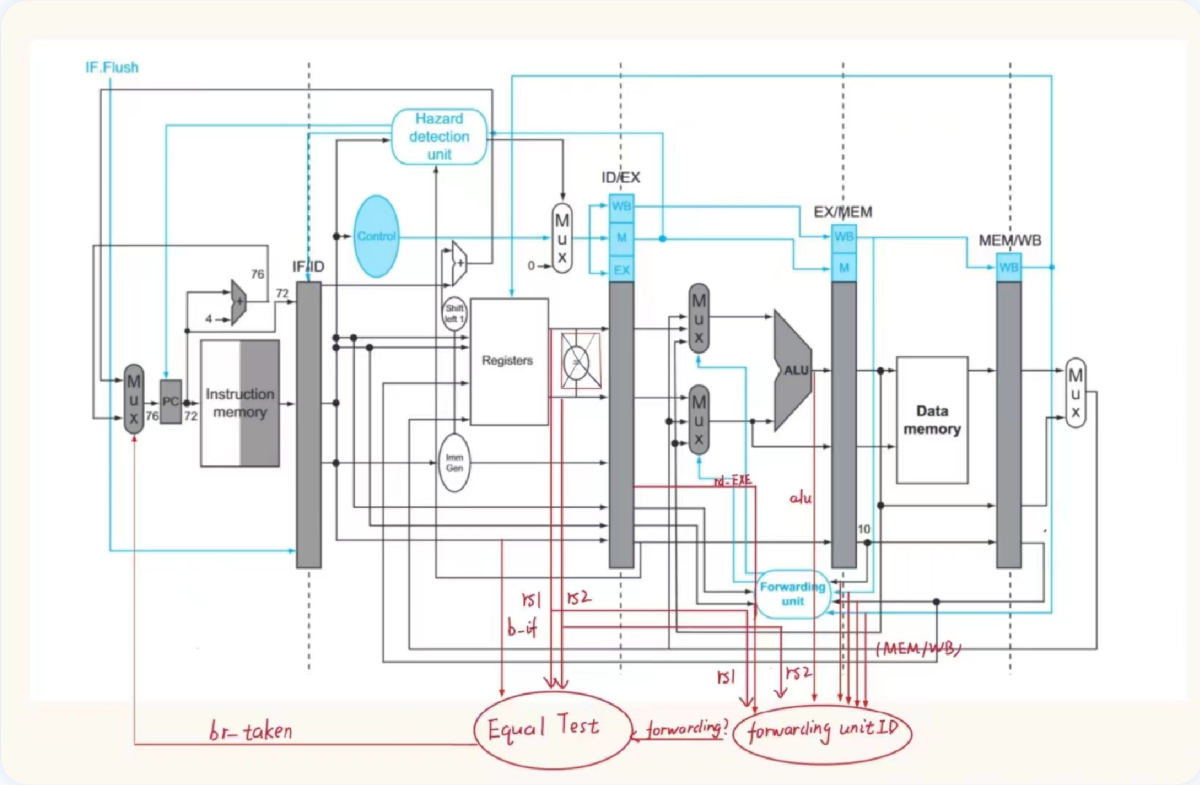
观察表中数据，如果我们定义'->'两端的状态相同为此条指令判断正确，否则为判断错误，经过详细统计，上表中共判断正确24，判断错误26，正误比为$\frac{24}{26}$大于静态判断的$\frac{5}{45}$。

对于动态分支预测为什么比静态分支预测好，我们给出如下的理由：

1. **历史信息**：动态分支预测器会维护分支指令的历史信息。它们存储了每个分支指令的历史行为，包括最近的分支结果。这个历史信息用于预测未来的分支指令。静态分支预测器不考虑历史信息，而动态分支预测器则根据历史信息来调整预测。

2. **适应性**：动态分支预测器是自适应的。它们根据实际的运行时条件来调整预测策略。如果某个分支指令在过去被频繁预测为"Taken"，但最近几次未被采纳，动态分支预测器可以调整预测为"Not Taken"，以适应分支行为的变化。
3. **多级预测**：动态分支预测器通常使用多级预测机制，包括局部历史、全局历史、混合历史和神经分支预测等。这些多级预测机制可以同时考虑多个方面的历史信息，从而提高了预测的准确性。
4. **动态更新**：动态分支预测器在运行时动态更新预测信息。每当分支指令执行并分支结果确定时，预测器会根据实际结果对历史信息和预测状态进行调整。这种实时更新可以更好地适应程序的运行时变化。

C



（注：原图Equal Test画的太小了，连线连不了一点，我把他揪出来放大了看）

为了实现在ID阶段的分支预测，我们需要在原图基础上加一个forwarding unit来判断什么时候需要数据前递。

解释一下图中的信号：

- b_if：判断ID阶段的指令是否为分支指令
- rs1，rs2：ID阶段两个寄存器的数值
- br_taken：输送给pc_MUX的最终跳转信号
- rd_EXE，alu，rd_MEM，rd_WB....：与正常forwarding的数据传送相同。

- 关于b_if的连线问题：

在图中我们引入了一个ID阶段的判断信号b_if，理论上来讲这个信号我们给Equal Test和forwarding unit都是可以的。forwarding unit的最终目的给Equal Test传递前递数据，是否进行pc的跳转选择是最后一步要做的事情。所以将b_if连入Equal Test，在最后一步输出br_taken也是可以的。

Instructions	**stall cycles**	**rs1 forward stage**	**rs2 forward stage**
addi x1, x0, 1.....beq	0	MEM	EXE
addr x1, x0, 1..ld..beq	1		EXE
ld..ld..beq	3		MEM
addi..addi.beq	0		EXE