10 带有FPU的流水线CPU设计

📌FPU主要是利用add.s、sub.s、mul.s、div.s、sqrt.s[注释1]指令对浮点寄存器文件的数据进行操作，同时也支持lwc1、swc1指令实现寄存器文件和存储器之间的数据交换

英语

1. *bypass*越过、绕过
2.

1. *postpone*延迟
2.

10.1 CPU/FPU流水线模型

10.1.1 FPU指令

带有FPU的流水线CPU在执行之前的20条指令的基础上，增加了单精度浮点指令add.s、sub.s、mul.s、div.s、sqrt.s、lwc1、swc1的执行，新增的7条单精度浮点指令的功能描述如下：

add.s、sub.s、mul.s、div.s fd,fs,ft #fd←fs op ft;

根据fs、ft从浮点寄存器文件中取出单精度浮点数，并根据op表示的操作进行运算，将运算结果写回fd寄存器文件单元

sqrt.s fd,fs #fd←root(fs);

根据fs从浮点寄存器文件中取出单精度浮点数，然后对该数据开方将结果写回fd寄存器文件单元

lwc1 ft,offset(rs) swc1 ft,offset(rs)[注释2]

利用IU单元计算出对应的存储单元地址，lwc1将对应单元的数据写回ft寄存器，而swc1将ft寄存器数据写到该存储器单元

指令格式如下：

Table 10.1 Seven instructions related to the floating-point unit

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Description
lwc1	110001	rs	ft		offset		Load FP word
swc1	111001	rs	ft		offset		Store FP word
add.s	010001	10000	ft	fs	fd	000000	FP add
sub.s	010001	10000	ft	fs	fd	000001	FP subtract
mul.s	010001	10000	ft	fs	fd	000010	FP multiplication
div.s	010001	10000	ft	fs	fd	000011	FP division
sqrt.s	010001	10000	00000	fs	fd	000100	FP square root

10.1.2 基本的CPU/FPU流水线模型

我们分指令来探讨CPU/FPU的流水线模型

lwc1、swc1的流水线模型和lw、sw一致，只不过操作数是来自于浮点寄存器文件

浮点操作（加、减、乘、除、开方）的流水线模型如下图所示，add、sub的执行阶段包含对阶、求和、规格化三步；乘法是求Wallace的sum和c、求和、规格化；而除和开方则因为求xn迭代的存在需要阻塞流水线，至得到xn后即和乘法一样的处理：求Wallace的sum和c、求和、规格化

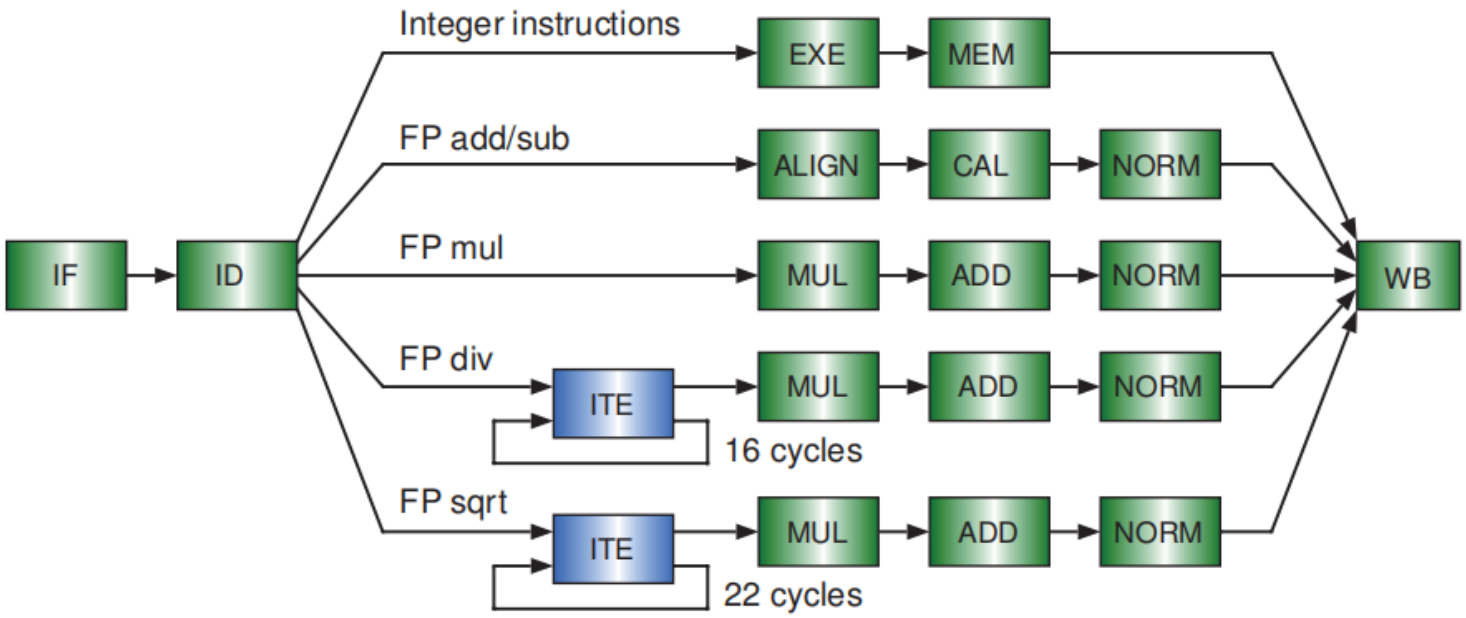

Figure 10.1 Pipeline models of the IU and the floating-point unit

浮点除法和开方的迭代我们采用Newton-Raphson算法去实现，除法迭代公式、开方迭代公式这里不再赘述。这里长时间的迭代有两种实现方式：完全流水方式和无流水方式，其形式见下图。完全流水方式使用一个专门的单元用于这个迭代过程，它可以

实现每一个时钟周期接收一个浮点除法/开方指令[注释3]，但是花销太高；不流水的方式则必须阻塞流水线，不断使用同一个单元去进行迭代，必须等待一定的周期后才能流入下一条浮点除法/开方指令

📌Note that the instruction latencies of the two implementations are the same[注释4]

两种实现方式对于下一条指令何时能够使用结果的延迟是一样的

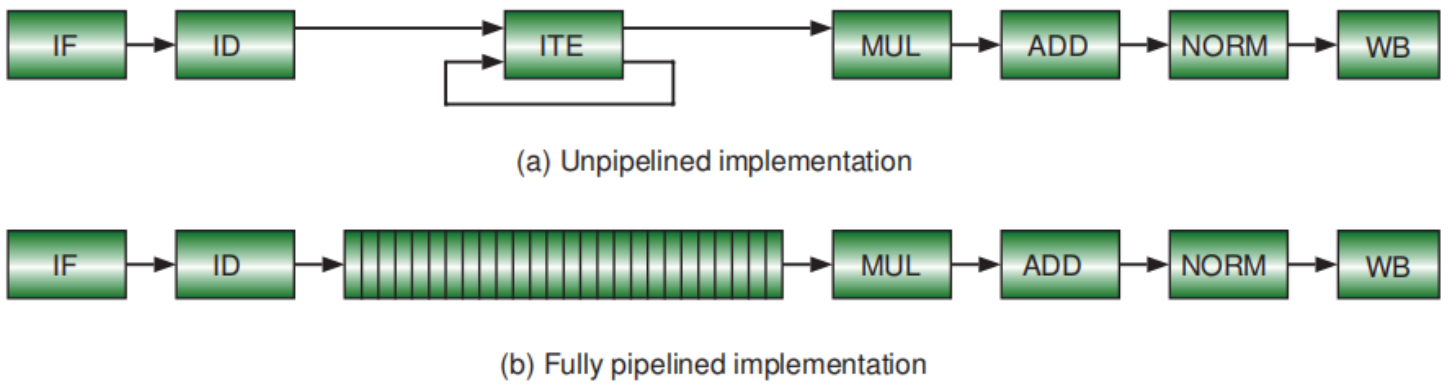

Figure 10.2 Unpipelined and fully pipelined implementations

我们采用的是**无流水方式**，使用**stall信号**[注释5]**在ID阶段阻塞流水线**，从一个小的ROM模块中读取x0构建reg_x的初值使得计算结果的产生只需要进行**3次迭代**

因为被阻塞时仍在ID阶段，所以如下图可以说**FPU也是五段流水——实际是6段（3个EX）**

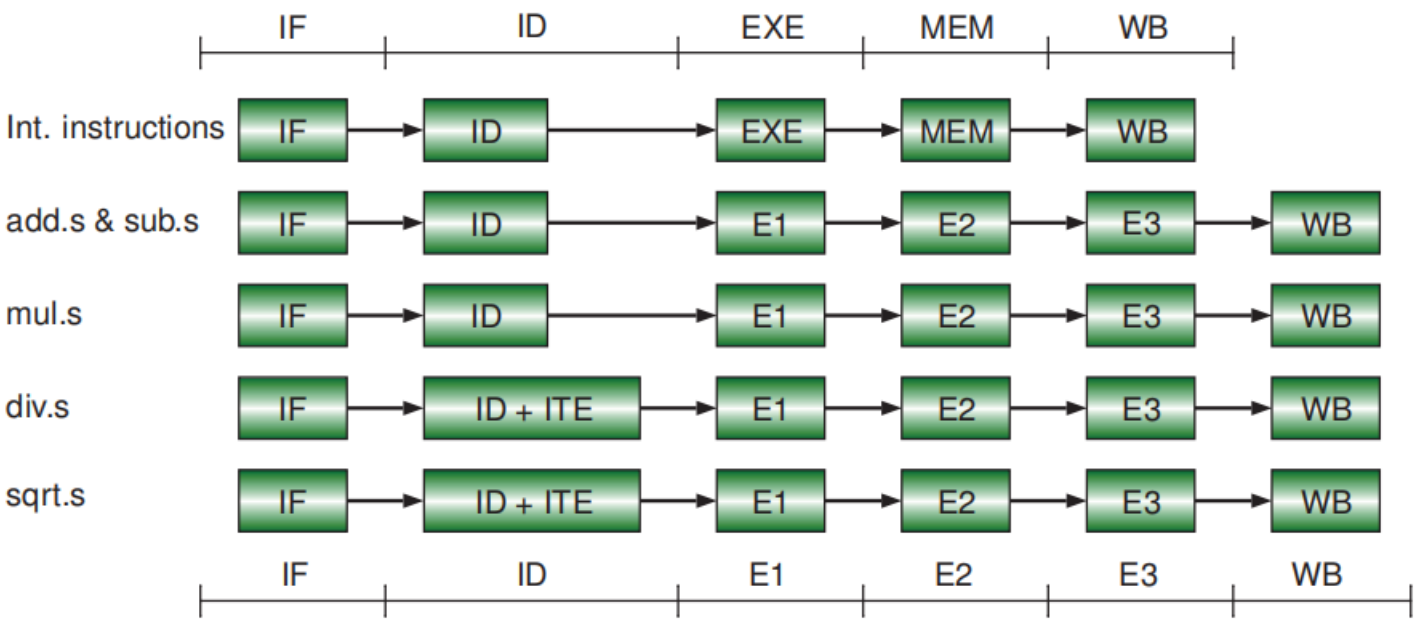

Figure 10.4 Unified floating-point pipeline model

10.2 双写端口的寄存器设计

📌浮点寄存器的0号寄存器并不是常量0，而是一个普通的寄存器

因为lwc1只需要5个时钟周期即可完成写寄存器的操作，但是浮点运算指令加减乘需要6个时钟周期才写寄存器，而开方和除法需要的则更多，因此会出现下图同一时钟周期内写寄存器冲突的情况——且都是lwc1和运算指令的写冲突，lwc1指令在后

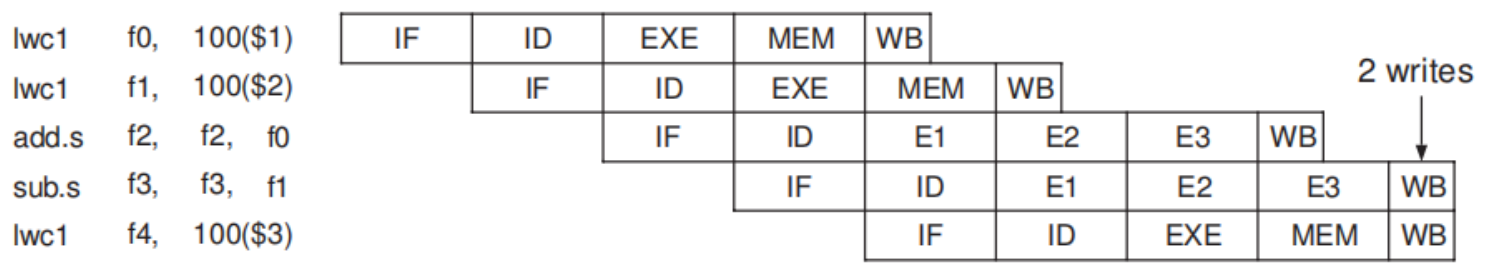

Figure 10.5 Register file requiring two write ports

我们可以采用硬件或者软件调整指令序列的方法延迟第二个指令的写寄存器来解决这个冲突，但是这里我们采用双写端口的寄存器——lwc1写端口的优先级更高来解决冲突

双写端口的寄存器示意图如下所示，有时钟、清零端、两个读端口以及对应的读数据、两组写端口(写使能、写寄存器地址、写数据)，且1组优先级高于0组——**这个优先级体现在当lwc1和运算指令写同一个地址时，最后保存的是lw的写结果。写不同地址时，两端口并行写**

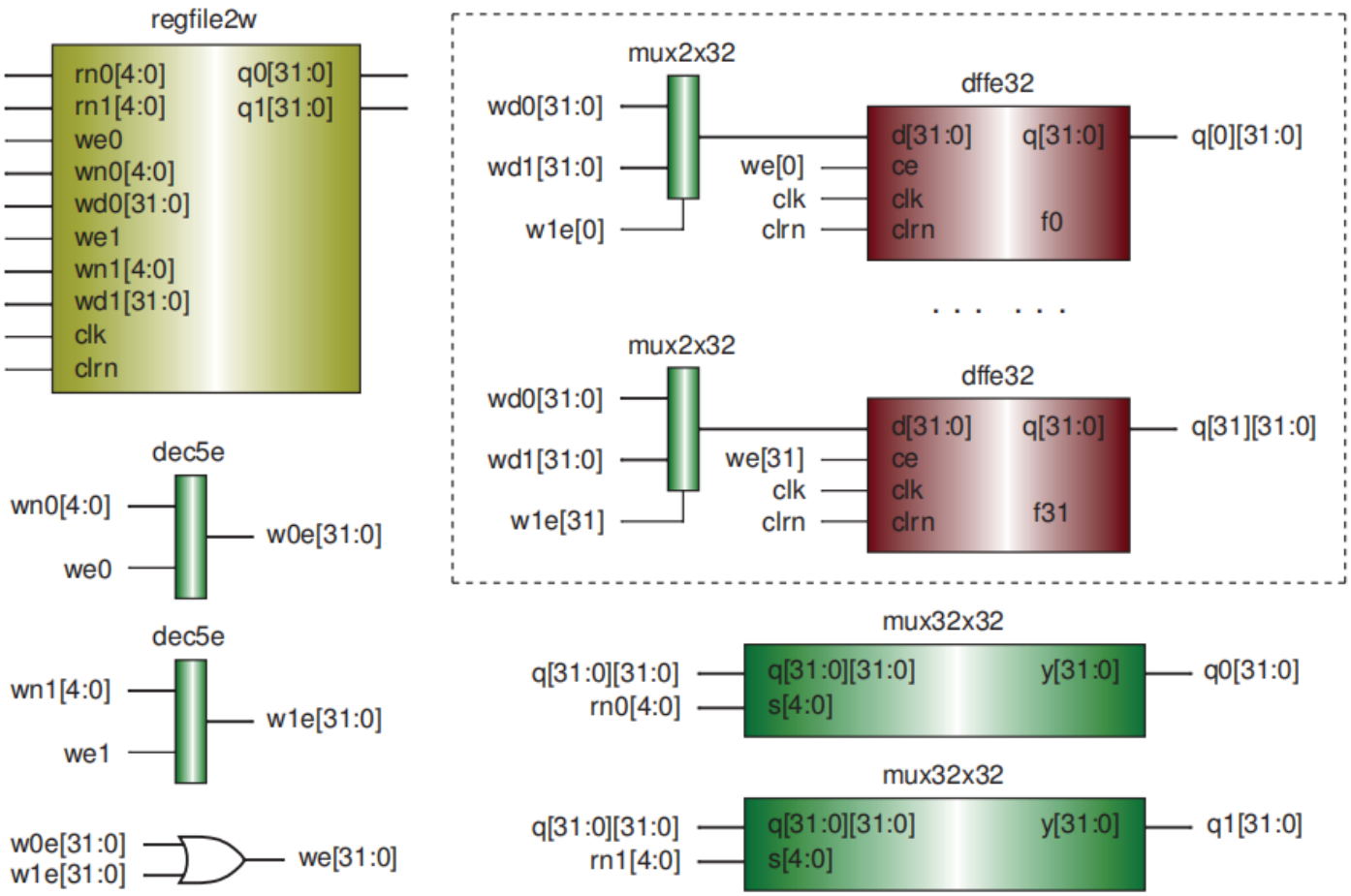

Figure 10.6 Schematic diagram of a register file with two write ports

```
`timescale 1ns / 1ps

module twoWriteReg(
    input clk,clrn,
    input [31:0]wd0,
    input [4:0]wn0,
    input we0,
    input [31:0]wd1,
    input [4:0]wn1,
    input we1,
    input [4:0]rn0,rn1,
    output [31:0]qa,qb
);
```

```
    reg [31:0]regs[31:0];
    assign qa=regs[rn0];
    assign qb=regs[rn1];
    integer i;
    always @(posedge clk or posedge clrn) begin
        if (clrn) begin
            for(i=0;i<32;i=i+1) regs[i]<=32'b0;
        end else if(we1) regs[wn1]<=wd1;
        else if(we0&(~we1|(wn1!=wn0)))regs[wn0]<=wd0;
    end
endmodule
```

Verilog

10.3 数据依赖和流水线阻塞

📌数据依赖包括：运算指令之间的数据依赖；lwc1和运算指令之间的数据依赖；sw和运算指令之间的数据依赖；以及fsqrt、fdiv造成的流水线阻塞

10.3.1 运算指令之间的数据依赖

运算指令的运算结果最早只能是在E3阶段之后取，所以有以下几种情况

1. 相邻指令存在数据RAW

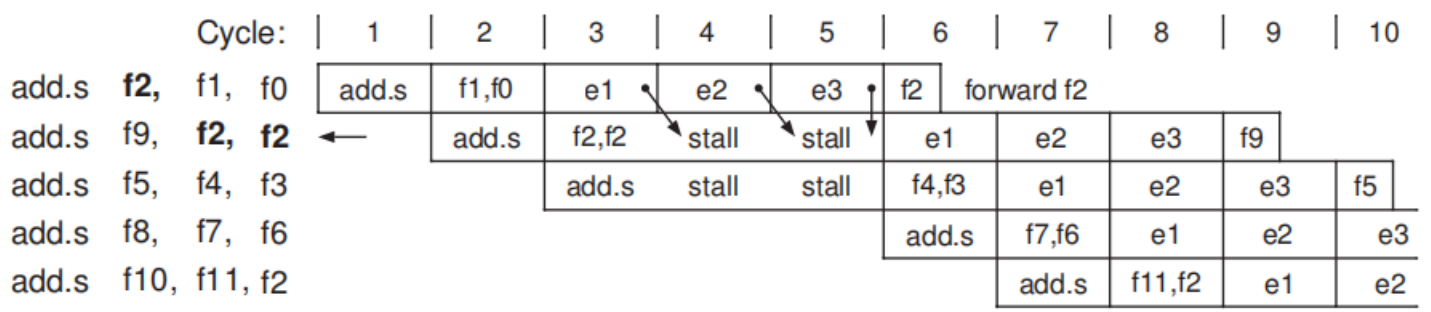

Figure 10.11 Two-cycle stall case for floating-point data hazard

因为当前指令在ID阶段时，写寄存器的指令正位于E1阶段，因此必须进行阻塞，阻塞两个时钟周期，到E3阶段时执行前递

```
i_fs = i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt; // use fs
i_ft = i_fadd | i_fsub | i_fmul | i_fdiv; // use ft
stall_fp=(e1w&(i_fs&(fs==e1n)|i_ft&(ft==e1n)))|
      (e2w&(i_fs&(fs==e2n)|i_ft&(ft==e2n)));
```

Verilog

2. 间隔一条指令存在数据RAW

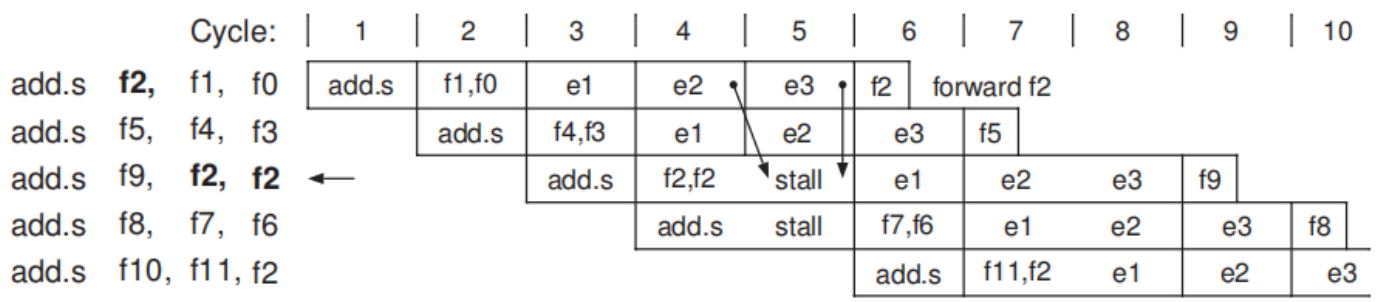

Figure 10.10 One-cycle stall case for floating-point data hazard

因为当前指令在ID阶段时，写寄存器的指令正位于E2阶段，因此必须进行阻塞，阻塞一个时钟周期，到E3阶段时执行前递

```
stall_fp=e2w&(i_fs&(fs==e2n)|i_ft&(ft==e2n));//和紧邻的一块处理
```

Verilog

3. 间隔两条指令存在数据RAW

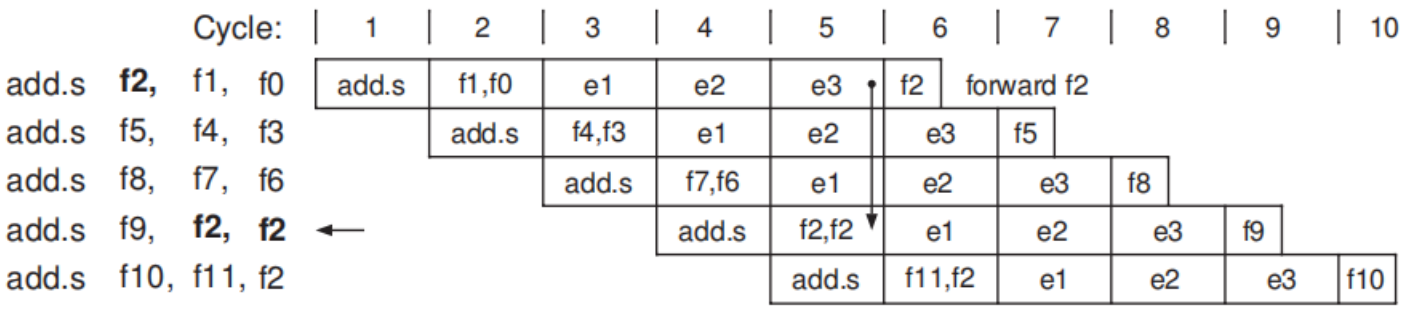

Figure 10.8 No-stall case for floating-point data hazard

直接将e3阶段的结果送至所需指令的ID段

即若e3阶段存在写浮点寄存器，且写的地址有等于ID阶段读的rs或者rt则执行前递

```
fwdfa = e3w & (e3n == fs); // forward fpu e3d to fp a
fwdfb = e3w & (e3n == ft); // forward fpu e3d to fp b
```

Verilog

电路结构如下所示：

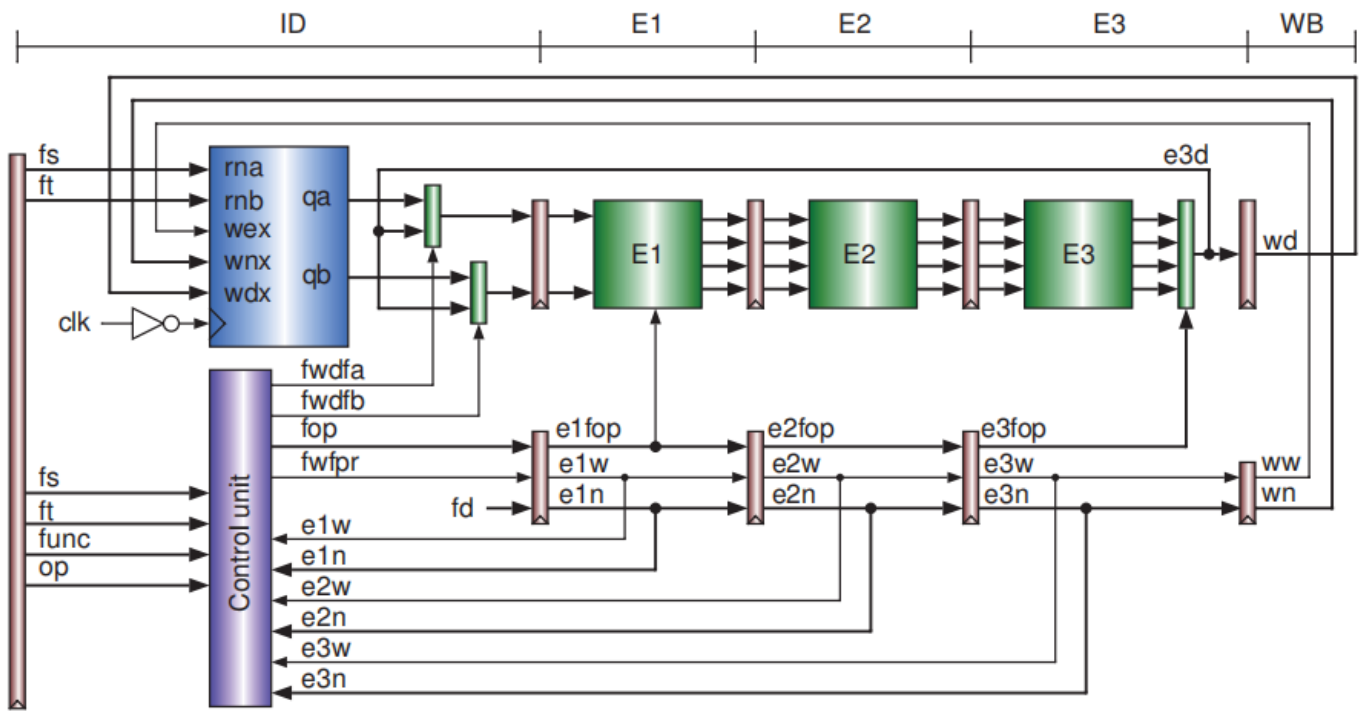

Figure 10.9 Block diagram of forwarding floating-point data

4. 间隔三条的利用写寄存器在前半沿解决数据冒险

10.3.2 lwc1和运算指令之间的数据依赖

lwc1的结果可以提前在MEM阶段取得，所以有以下几种情况

1. lwc1指令与后续指令紧邻

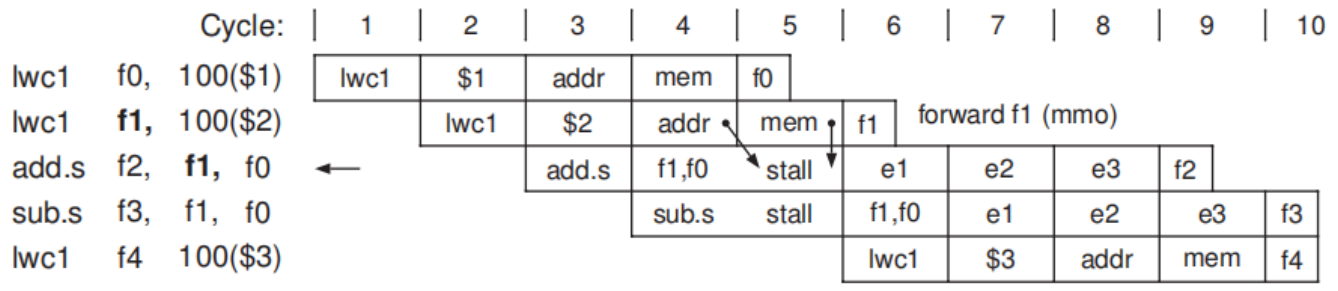

Figure 10.15 One-cycle stall caused by floating-point load

紧邻的指令，当当前指令在ID段时，lwc1指令在EXE段，因此需要stall一个时钟周期使得，lwc1指令位于MEM阶段才可执行前递

```
stall_lwc1 = ewfpr & (i_fs & (ern == fs) | i_ft & (ern == ft));
```

Verilog

2. lwc1指令与后续指令间隔1条指令

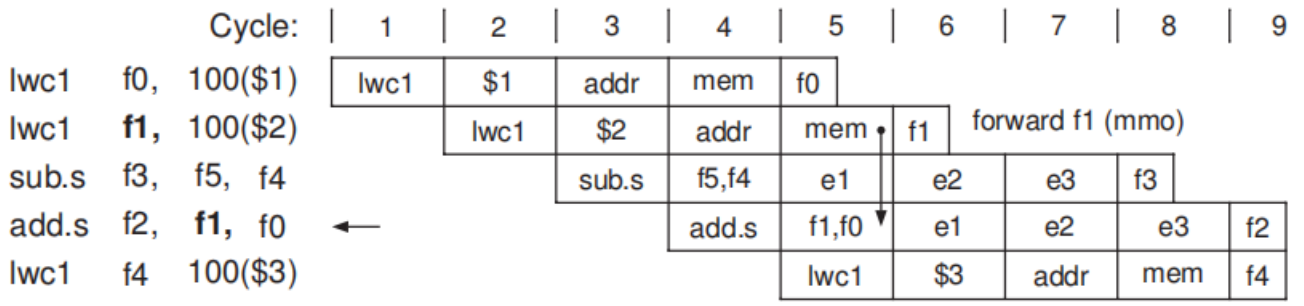

Figure 10.13 Data hazard with floating-point load

可以直接由mem阶段将结果送至当前指令的ID阶段

即如果mem阶段是lw指令且写寄存器地址与当前ID阶段的读寄存器地址rs、rt有相等，则执行数据前递

```
fwdla = mwfpr & (mrn == fs); // forward mmo to fp a
fwdlb = mwfpr & (mrn == ft); // forward mmo to fp b
```

Verilog

3. lwc1指令与后续指令间隔2条指令——利用写寄存器在前半沿解决数据冒险

10.3.3 swc1和运算指令的数据依赖

swc1如果写存储器的结果用到之前运算指令的结果，则涉及到数据依赖

swc1写寄存器位于MEM段，此时会有以下几种情况：

1. 数据依赖发生在紧邻的两条指令之间——E3和MEM

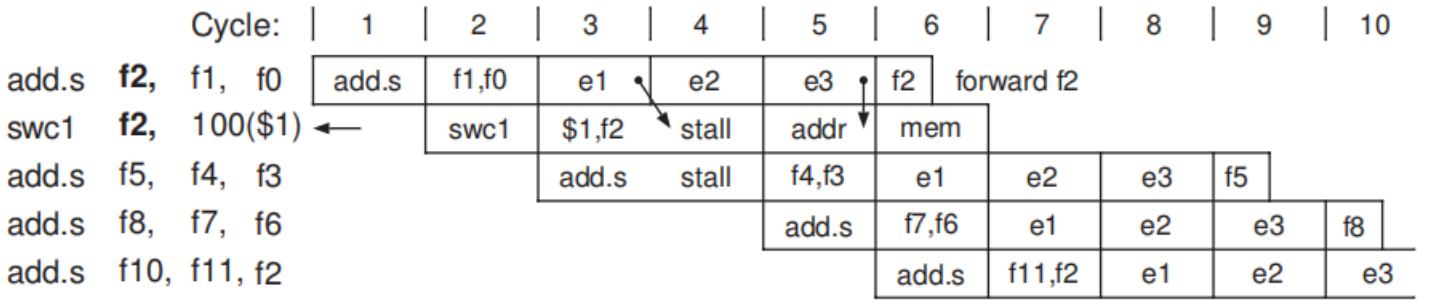

Figure 10.18 One-cycle stall caused by floating-point store

E3规格化完后，mem也执行完毕，因此需要stall一个周期使得E3的结果可以传至EXE阶段方便MEM写存。即如果执行阶段计算得到的结果是当前swc1指令执行阶段要传送到存储器的寄存器数据，则暂停一个周期

```
stall_swc1=swfp&e1w&(ft==e1n);
```

Verilog

2. 中间间隔一条指令——WB和MEM

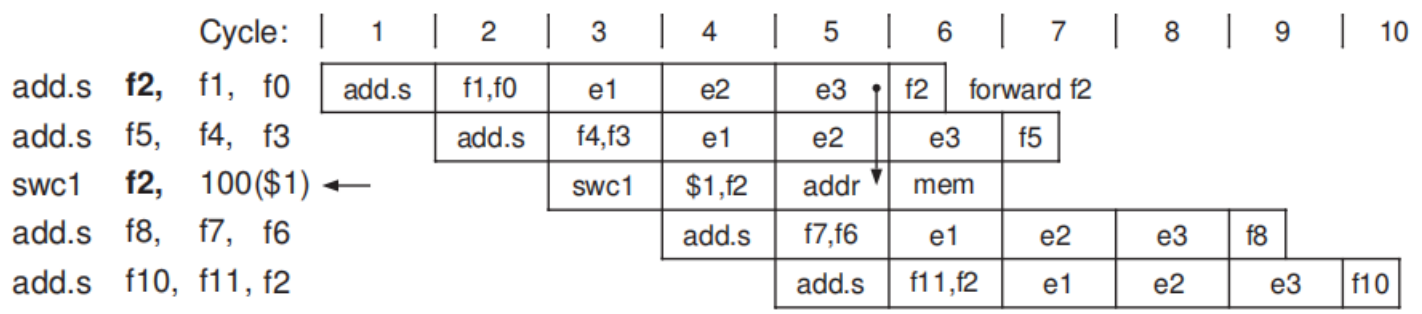

Figure 10.17 Forwarding floating-point data to the execution stage

E3即产生算术结果，可以直接送swc1的EXE阶段，方便下一周期写存

```
fwdfe=swfp&e3w&(ft==e3n) //执行阶段判断
```

Verilog

3. 中间间隔两条指令——E3和ID

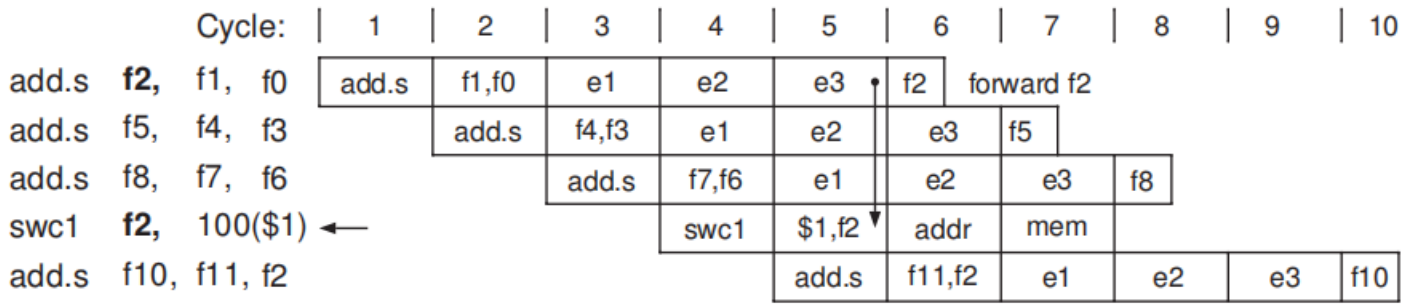

Figure 10.16 Forwarding floating-point data to the instruction decode stage

E3即产生算术结果，可以直接送至swc1指令的取浮点寄存器后

```
fwdf = swfp & e3w & (ft == e3n); //译码阶段判断
```

Verilog

10.3.4 fdiv和fsqrt造成的流水线阻塞

fdiv和fsqrt所造成的阻塞是因为迭代所导致的

```
stall_div_sqrt = stall_div | stall_sqrt;
w_fp_pipe_reg = ~stall_div_sqrt; // 阻塞FP流水线
```

Verilog

除了stall_div_sqrt（会阻塞FP流水线）外，stall_lw、stall_fp、stall_swc1、stall_lwc1都会使得PC、IF/ID寄存器阻塞；阻塞过程中的清空流水线是用复位所有的写信号实现的——wreg、wmem、wf、wfpr

```
stall_others = stall_lw | stall_fp | stall_lwc1 | stall_swc1;

wpcir = ~(stall_div_sqrt | stall_others);
wreg = (i_add | i_sub | i_and | i_or | i_xor | i_sll |
i_srl | i_sra | i_addi | i_andi | i_ori | i_xori |
i_lw | i_lui | i_jal) & wpcir; // int rf
wmem = (i_sw | i_swc1) & wpcir; // memory
wf = (i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt) & wpcir; // fp rf x
wfpr = i_lwc1 & wpcir;
```

Verilog

📌wfpr是IU内部的lwc1写浮点寄存器信号，是在IU内部流水传递的，在各级是ewfpr、mwfpr、wwfpr

swfp是IU内部的控制信号，表示当前是swc1指令

普通的浮点指令的写信号是wf、该信号传递至FPU，在各级流水分别是e1w、e2w、e3w、ww

fs、ft是ID阶段译码浮点指令得到的源寄存器地址信号，传递至浮点寄存器读数据

fd是ID阶段译码得到的目的地址，传送至FPU，在各级流水分别是e1n、e2n、e3n、wn

rn是ID阶段译码得到的lwc1指令的目的寄存器地址信号，在各级流水依次是ern、mrn、wrn

10.4 带有FPU的流水线CPU实现

10.4.1 顶层文件设计

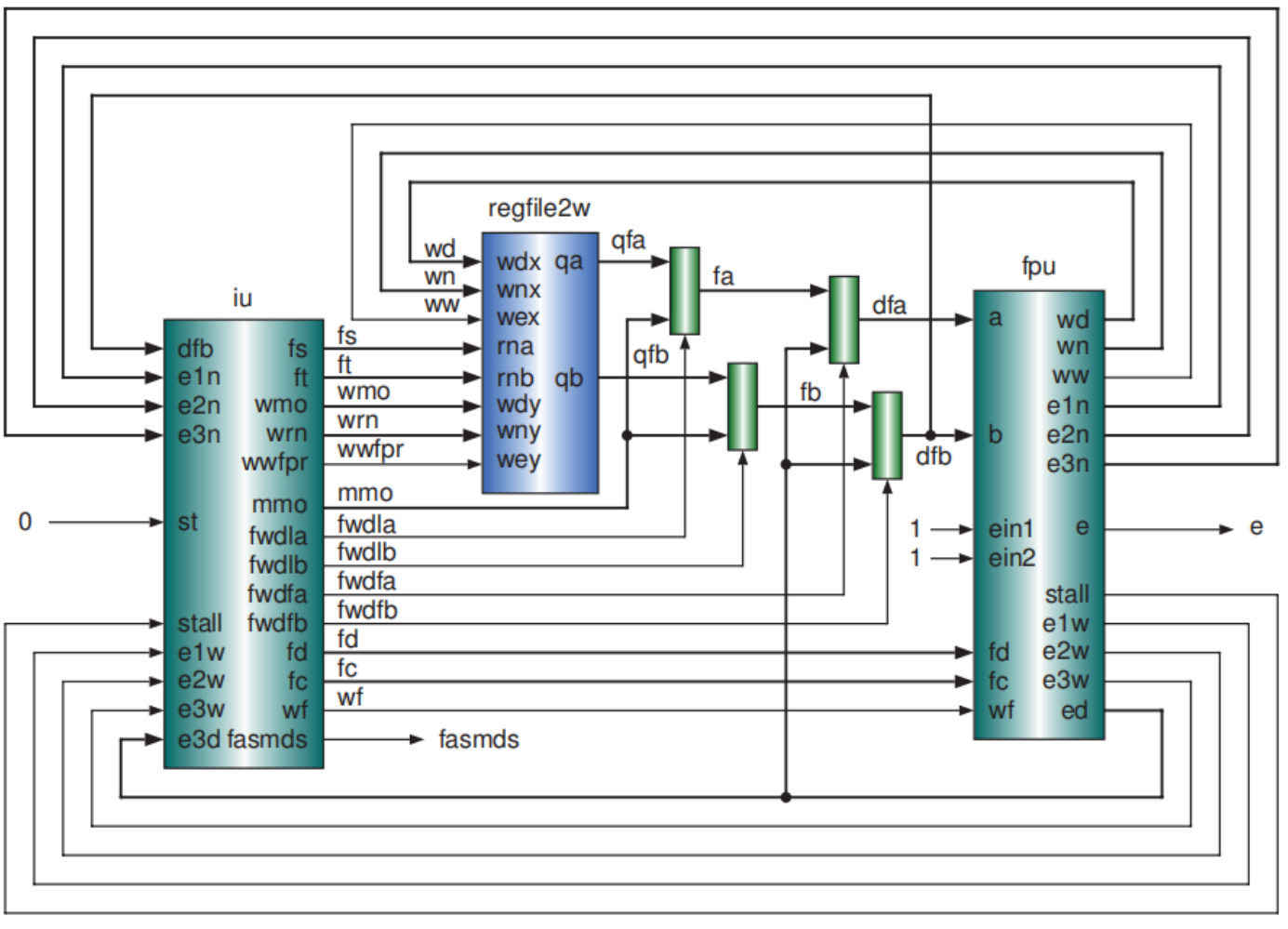

Figure 10.21 Block diagram of CPU with one IU and one FPU

IU即之前的整数运算的CPU，为了实现FPU，在原基础上增加了对增加的7条浮点指令的译码，同时也增加了lwc1、swc1的指令的五级实现

lwc1：wfpr、ewfpr、mwfpr、wwfpr，rn、ern、mrn、wrn，memRead、wmemRead

swc1：ft传送至双写端口浮点寄存器读

采用FPU和IU共用取指、译码阶段的思想，那么IU还需要发送给FPU“wf、wn、fd、fc”信号，对FPU进行相对应的操作，也需要接收FPU的“e1n、e2n、e3n、e1w、e2w、e3w”同IU内部的“指令译码，fs、ft、wfpr、ewfpr和swfp、rn、ern、mrn”来解决之前的3种类型的数据冲突，也需要接收FPU的“stall_div_sqrt”阻塞信号，与

“stall_lw,stall_swfp,stall_fp,stall_lwfp”阻塞流水

10.4.2 FPU设计

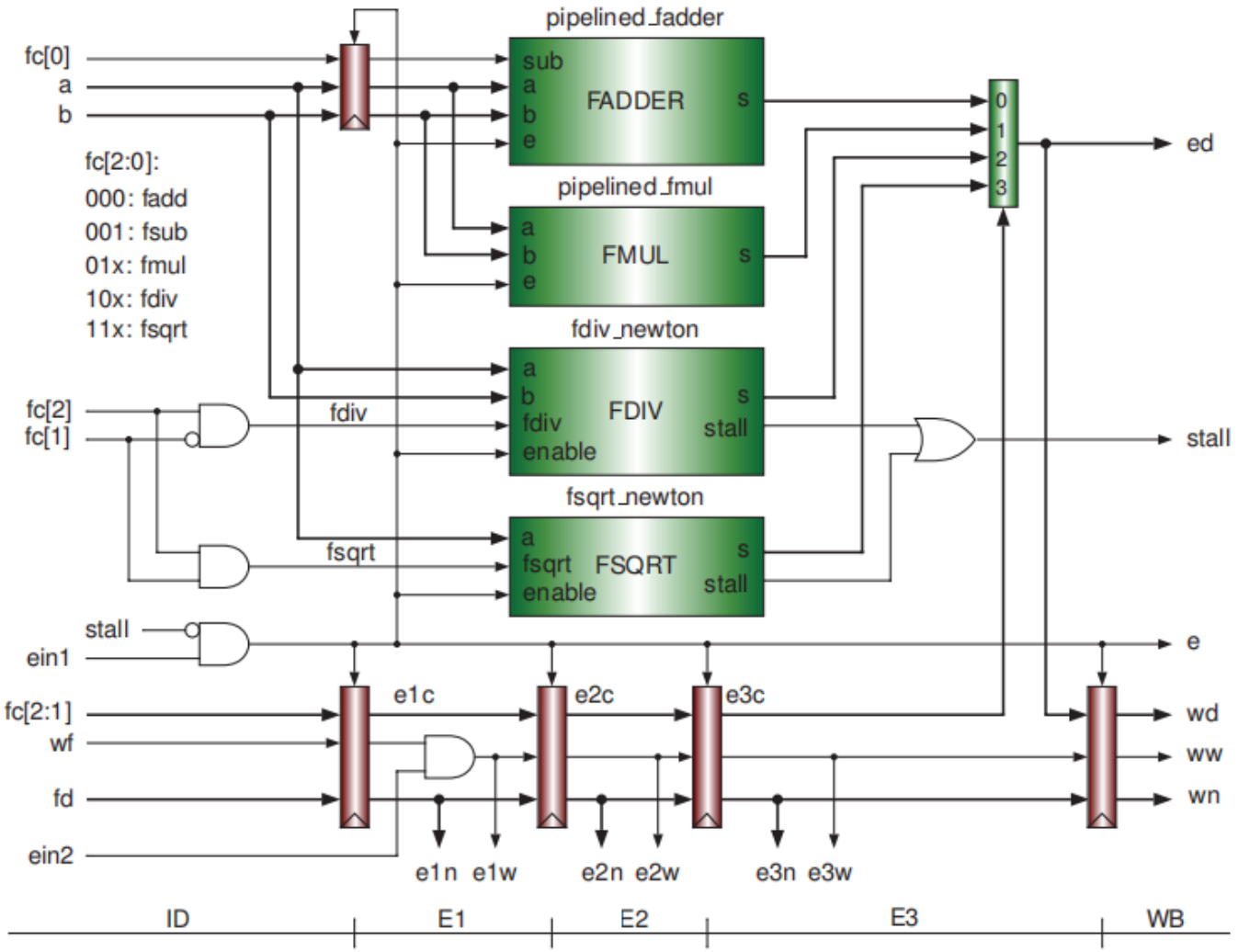

Figure 10.22 Block diagram of the floating-point unit

10.4.3 IU数据通路设计

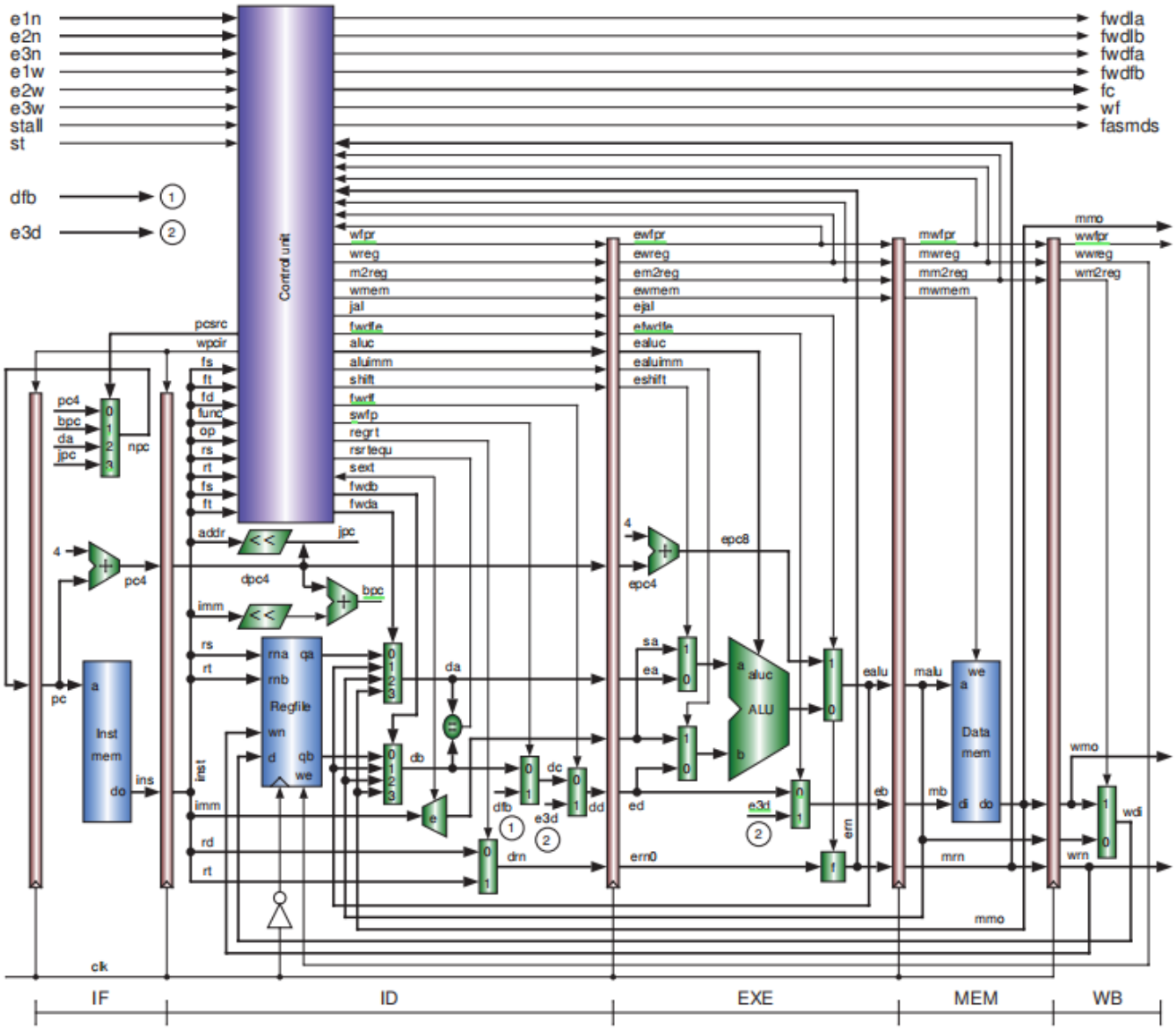

Figure 10.23 Block diagram of the integer unit

[注释1] 尾缀s表示是操作数是单精度浮点数，尾缀d表示是双精度浮点数

[注释2] 为什么指令助记符尾缀是c1，是因为MIPS将FPU称为协处理器1

[注释3] 也是上一节习题，怎么克服FDIV、FSQRT的答案

[注释4] The latency here is defined as the clock cycles the next instruction can use the result.

[注释5] stall=ifdiv|ifsqrt&(busy|count==0)