计算机组成

CHAPTER 1: PRELUDE

> $1\text{byte (B)} = 8\text{bits (b)}$
> $1\text{word} = 32\text{bits}$
> $1\text{double world} = 64\text{bits}$

设计的八大思想

1. **Design for Moore's Law**：集成晶体管数每18～24个月翻一倍
    - Design for where it will be when finishes rather than design for where it starts.
2. **Use Abstraction to Simplify Design**：采用抽象简化设计
    - 层次化、模块化设计
    - 更高级的模块系统基于更基础的系统，层层设计
3. **Make the Common Case Fast**：加速大概率事件
4. **Performance via Parallelism**：并行处理
5. **Performance via Pipelining**：流水线
    - 每个流程同时进行，每一个流程的工作对象是时间上相邻的产品。与先生产完一个产品才开始生产下一产品相对
    - 速度取决于最慢的流程，需要每一个流程的时间是相对均匀的
6. **Performance via Prediction**：预测
    - 先当作if条件成立，执行完内部任务，如果后面发现确实成立，直接就可以执行
    - 如果预测成功就可以加速，预测失败纠错成本也不高
7. **Hierarchy of Memories**：存储器层次

- Disk / Tate ▷ Main Memory (DRAM) ▷ L2-Cache (SRAM) ▷ L1-Cache (On-Chip) ▷ Registers

> Example：Library reserved desk，图书馆中热门的书单独放出，方便读者借还，相当于是放在了Hierarchy更顶层的Memory中

8. **Dependability via Redundancy**：通过冗余提高可靠性
   - 一个模块down了以后不会剧烈影响整个系统

性能

计算机性能

- **Responce Time** / **Execution Time**：响应时间 / 执行时间

  完成任务的时间。计算机的性能用响应时间的倒数表示：

$$\text{Performance} = \frac{1}{\text{Execution Time}} \tag{1}$$

- **Throughput**：吞吐率

  单位时间完成的工作量

CPU性能

- **Elapsed Time**：运行时间

  总响应时间，包括出入输出、操作系统开销等的时间

- **CPU Time**：CPU执行时间

  CPU处理任务的时间，不含输入输出、操作系统开销等时间

- **Clock Rate** / **Clock Cycle Time**：时钟频率 / 时钟周期

$$\begin{aligned}\text{CPU Time} &= \text{Clock Cycle} \times \text{Clock Cycle Time} \\ &= \frac{\text{Clock Cycle}}{\text{Clock Rate}}\end{aligned} \tag{2}$$

- **CPI**：每条指令的平均周期数（Average cycles pre instruction），总周期/总指令数

$$\begin{aligned}\text{CPI} &= \frac{\text{Clock Cycle}}{\text{Instruction Count}} \\ &= \frac{\sum_{i=1}^{n} \text{CPI}_i \times \text{Instruction Count}_i}{\text{Instruction Count}}\end{aligned} \tag{3}$$

总结如下：

$$\text{CPU Time} = \text{IC} \times \text{CPI} \times \text{Clock Cycle Time} \tag{4}$$

- **BIPS**：Billion Instruction Per Second，每秒可执行的指令数（单位为billion，十亿）

> 注意：$1\text{s} = 1 \times 10^9\text{ns} = 1 \times 10^{12}\text{ps}$

Amdahl's Law

边际收益递减

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{Improvement Factor}} + T_{\text{unaffected}} \tag{5}$$

> Example: Multiply accounts for 80s/100s. How much improvement in multiply performance to get 5 times overall?
>
> $$20 = \frac{80}{n} + 20 \Rightarrow \text{Can't be done!} \tag{6}$$

MIPS：Millions of Instructions per Second

每秒百万条指令数

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} \tag{7}$$

- 无法解释：
  - 不同电脑间ISA（指令集）的差异
  - 不同指令间复杂程度的差异

CHAPTER 3: ARITHMETIC FOR COMPUTER

> 反码：1's complement code
>
> 补码：2`s complement code，反码 + 1，用补码表示负数

> 无符号数：unsigned integer
>
> 有符号数：2's complement integer（注意这里指的不是要取该数的补码，而是指该数是有符号数，负数由补码表示）

> 二进制B：Binary，0b开头
>
> 八进制O：Octal，0o开头
>
> 十进制D：Decimal
>
> 十六进制H：Hexadecimal，0X开头

> Example: $101 \overset{\text{反码，逐位取反}}{\rightarrow} 010 \overset{+1}{\rightarrow} 011 \rightarrow -3$

ALU: Arithmetic-logic Unit

加法

- **Half Adder**

$$\begin{aligned} \text{Sum} &= a \oplus b \\ \text{Carry} &= ab \end{aligned} \tag{8}$$

- **Full Adder**

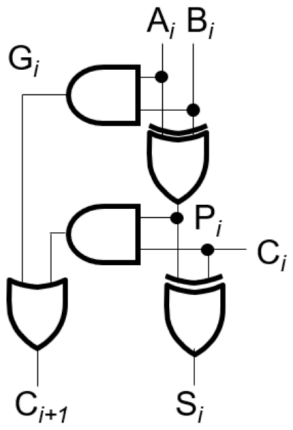
$$\begin{aligned} \text{Sum} &= a \oplus b \oplus \text{Carry}_{\text{In}} \\ \text{Carry}_{\text{Out}} &= b\text{Carry}_{\text{In}} + a\text{Carry}_{\text{In}} + ab \end{aligned} \tag{9}$$

- **Complete ALU**

  行波进位，逐位计算，前一个全加器的CarryOut输入后一个全加器的CarryIn

- **CLA: Carry Lookahead Adder** 行波进位加法器

  产生进位的可能有两种：如果两加数均为1，或者，一个加数为1且CarryIn为1。由此，我们可以构造一个下面这样的全加器：



逐层递推，我们可以实现每一位进位的提前计算：

$$
\begin{aligned}
c_1 &= g_0 + (p_0 \cdot c_0) \\
c_2 &= g_1 + (p_1 \cdot c_1) \\
&= g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0)
\end{aligned} \tag{10}
$$

其中$g = a \cdot b,\ p = a + b$

> Overflow：判断溢出只需要看运算结果的符号是否正确即可，因为不存在其他溢出的可能结果（比如，两正数相加，不可能溢出两轮到正数，如果溢出结果一定为负）

减法

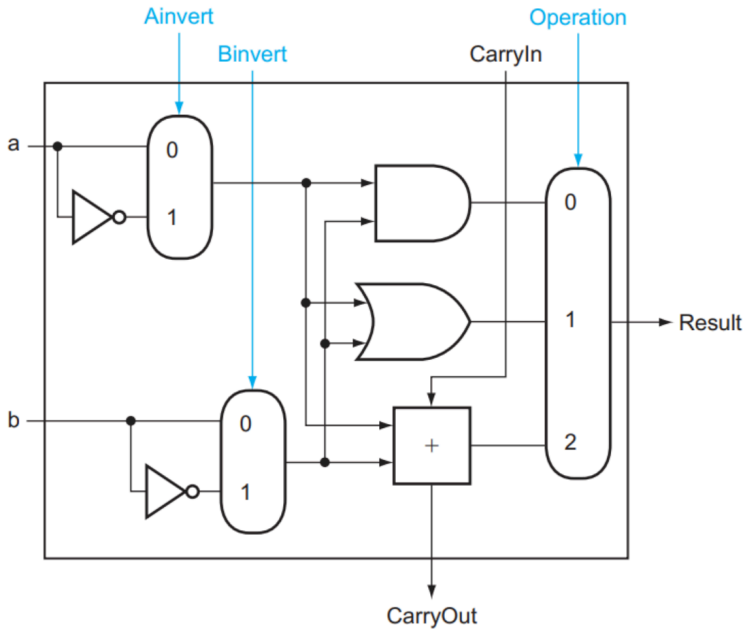
对b取补码（取反加1），再与a相加

1. Invert b
2. $1\text{st } \mathrm{Carry}_{\mathrm{In}} = 1$

1-bit ALU

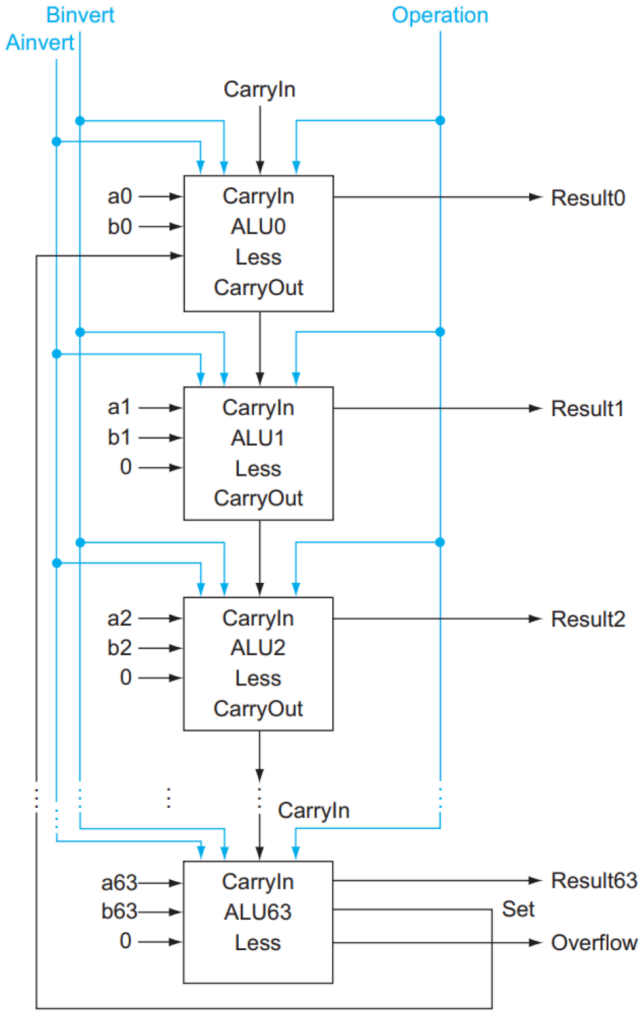
这是一个支持与、或、或非和加减法的一位ALU。其中或非（NOR）的实现方法如下：

$$
a \oplus b = \neg(a \vee b) = \neg a \wedge \neg b \tag{11}
$$



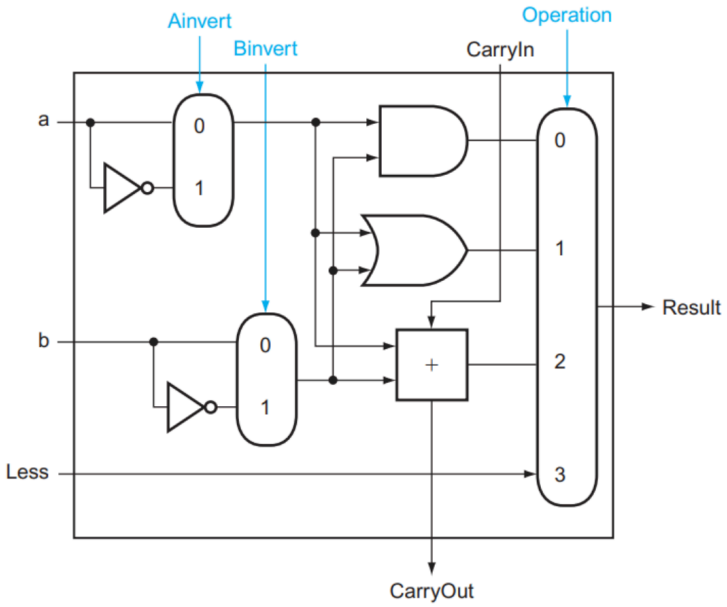
64-bit ALU

通过将一位ALU串联（前一个的CarryOut输入到后一个的CarryIn），可以构造出一个64位的ALU：



- Less和Set：实现SLT（Set Less Than）比较大小的操作。将两数相减，如果为负则输出1（将减法结果的最高位输出即可）。
- Overflow：在ALU63中添加一个额外的或非门实现溢出的判断。

因此，ALU1~ALU6的结构如下，其中ALU0的Less来自ALU63的Set（加法器的最高位），其余Less给0：



而对于ALU63，需要加上一个溢出判断，可以通过CarryIn和CarryOut异或实现。

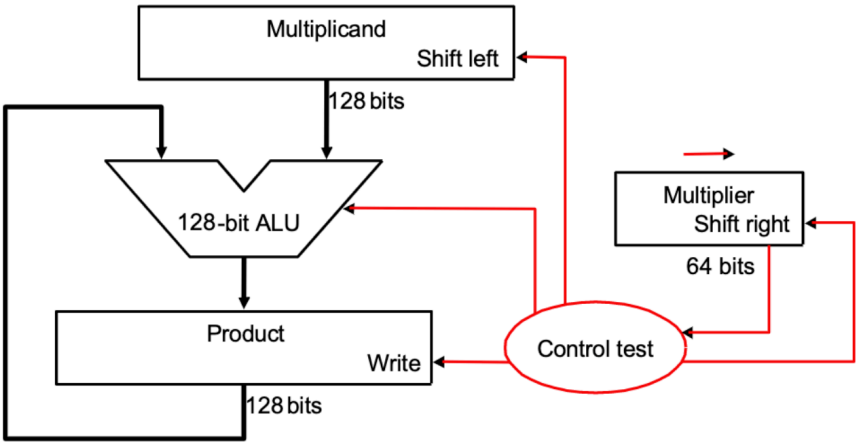
乘法

> 被乘数：Multiplicand，乘数：Multiplier，积：Product

$$\text{Multiplicand} \times \text{Multiplier} = \text{Product} \tag{12}$$

Version1 被乘数左移，乘数右移

因为被乘数和乘数都需要移动，所以Multiplicand、Multiplier和Product都需要128位，比较浪费空间

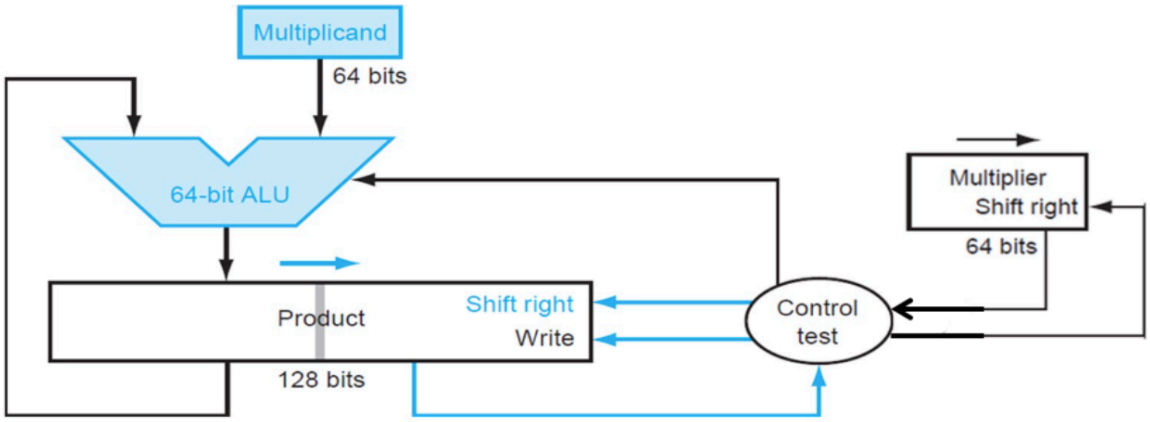


1. 判断Multiplier寄存器的最低位是否是1
2. 如果是，则将Multiplicand寄存器的值加到Product寄存器里
3. 将Multiplier寄存器的值右移一位（这是为了不断拿出每一位，相当于在枚举Multiplier的每一位），将Multiplicand寄存器的值左移一位（相当于将Multiplicand提高一个数量级，模拟竖式）
4. 做满64次终止

Version2 被乘数不左移，右移乘积

因为被乘数不需要移动，所以Multiplicand只需要64位。

其实也是模拟竖式，将当前的运算结果右移是降低了其数量级，也就相当于提升了被乘数的数量级。巧妙之处在于Product右移就相当于提高了被乘数的数量级，但因为这样移动利用的是Product寄存器中无用的空间，避免了被乘数的移动带来无意义的空间损耗

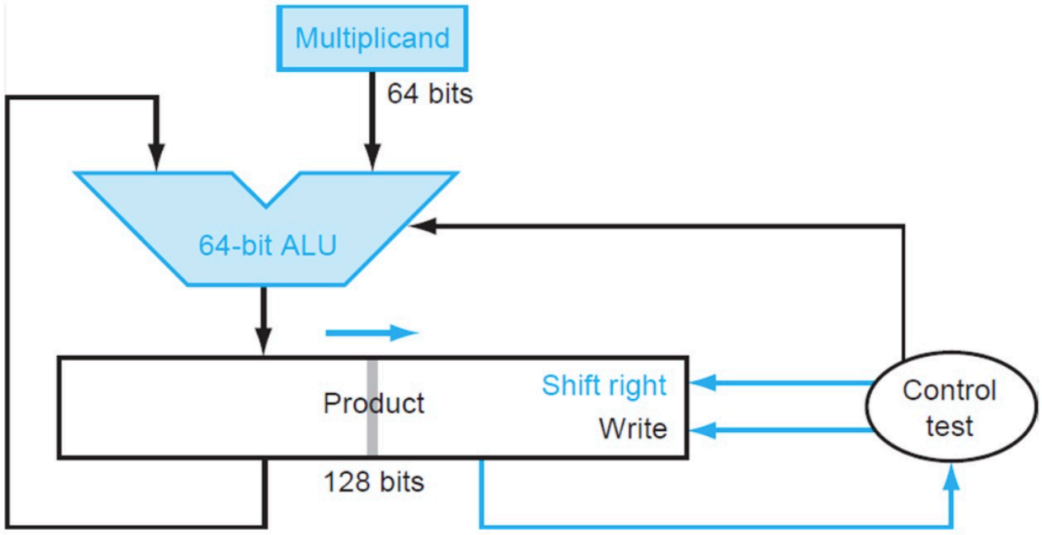


1. 判断Multiplier寄存器的最低位是否是1
2. 如果是，则将Multiplicand寄存器的值加到Product寄存器的左半部分里

3. 将Multiplier寄存器的值右移一位，将Product寄存器的值右移一位
4. 做满64次终止

Version3 乘数和被乘数都不移动

我们发现Product右侧存在大量空间，且每次右移Product都需要右移Multiplier，前者所需空间增加一位，后者所需空间减少一位；因此Product的后64位刚好可以用于存放Multiplier



1. 判断Product寄存器的最低位是否是1
2. 如果是，则将Multiplicand寄存器的值加到Product寄存器的左半部分里
3. 将Product寄存器的值右移一位
4. 做满64次终止

有符号乘法

常规的操作是直接判断正负，并在运算结果前加上符号位。同时，Booth算法原生支持有符号乘法（不要求）。

Booth's Algorithm

> Booth算法只需要了解即可，一般不作考查要求

因为位移运算比加法运算的速度要快，Booth算法的思路就在于尽可能减少加法的次数，而用位移代替。

乘数某位为0时，不需要执行加法操作，只需要位移即可（因为加了也是加0，不对结果产生影响）。所以Booth's Algorithm的思路是尽可能减少乘数中1的个数。

我们发现：

$$0011\ldots1100 = 0100\ldots0000 - 0000\ldots0100 \tag{13}$$

这样我们便将连续的1转化为两个1和连续的0了。如此，遇到连续的1也只需要位移而无需进行加法，只需要在头（10）和尾（01）处进行加法即可。

- Q_{-1}：乘数的上一位，初始为0
- Q_0：乘数的最后一位

Q_0Q_{-1}	**操作**
10	在Product寄存器中减去Multiplicant
11	直接位移
01	将Multiplicant加到Product寄存器中
00	直接位移

除法

> 除数：Divisor，被除数：Dividend，商：Quotient，余数：Remainder
>
> $$\frac{\text{Dividend}}{\text{Divisor}} = \text{Quotient} \tag{14}$$

Version1 除数右移，商左移

除数存在Divisor左半部分，被除数寄存在Remainder中；每次右移一位除数，左移一位Quotient，相当于是降低除数的数量级，模拟竖式

1. Remainder减去Divisor
2. 如果小于0，说明不够除，则将Divisor再加回去，Quotient最低位给0
3. 如果大于0，Quotient最低位给1
4. Divisor右移，Quotient左移

Version2 被除数与余数合一，左移

我们发现被除数根本就不需要一直寄存，竖式除法中真正用到的其实是余数，所以便不保留被除数，而将余数的初始值设为被除数。同时，每次操作Remainder都需要左移，会空出右边的空间，正好将商填入其中

每次操作Remainder左移，相当于增加其数量级（也就是减小除数的数量级，模拟竖式）。注意本方法需要初始化，先将余数（被除数）左移一次，以在最右侧留出一位空间给商；作为补偿，最后要将余数的左半部分右移一位才会获得真正的余数

0. Remainder左移
1. Remainder（左半边）减去Divisor
2. 如果小于0，说明不够除，则将Divisor再加回去，Remainder最右边给0
3. 如果大于0，Remainder最右边给1
4. Remainder左移
5. 上述循环完毕后，Remainder的右半边为计算结果；左半边右移1位，得到余数

Iteration	Step	Divisor	Remainder
Initial	初始值	0010	0000 0111
	余数左移，右边留出一位空间	0010	0000 1110
1	余数减去除数，发生溢出	0010	1110 1110
	余数加回除数，最右一位给0，左移	0010	0001 1100
2	余数减去除数，发生溢出	0010	1111 1100
	余数加回除数，最右一位给0，左移	0010	0011 1000
3	余数减去除数，未溢出	0010	0001 1000
	最右一位给1，左移	0010	0011 0001
4	余数减去除数，未溢出	0010	0001 0001
	最后一位给1，左移	0010	0010 0011
Final	余数的左半部分右移1位，即为余数	0010	0001 0011

浮点数

> Fraction指的是小数部分，Significand指的是完整的数字

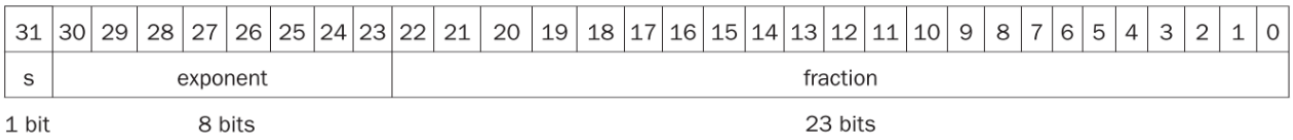
格式

存储格式：Sign符号位 + Exponent数量级 + Fraction小数部分

$$(-1)^{\text{Sign}} \times (1.\text{Fraction}) \times 2^{\text{Exponent}-\text{Bias}} \tag{15}$$

单精度与双精度

- **单精度**：1位Sign + 8位Exponent + 23位Fraction，Bias = 127



通常来说，Exponent的全0和全1是保留编码，因此单精度绝对值得下限是：

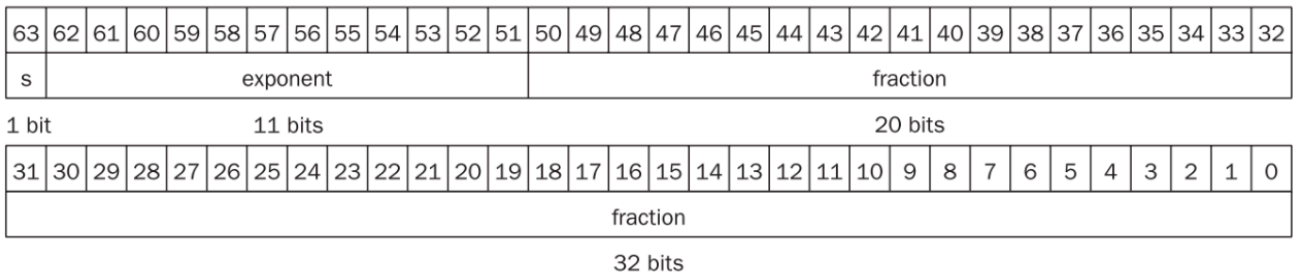
$$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38} \tag{16}$$

绝对值的上限是：

$$\pm 2.0 \times 2^{127} \approx \pm 3.4 \times 10^{38} \tag{17}$$

> 上限是$254 - 127 = 127$，而不是255，因为全1保留（注意Exponent是无符号数，最大数是全1的）
>
> 同样地，最低只能是1，因此下界是-126

- **双精度**：1位Sign + 11位Exponent + 52位Fraction，Bias = 1023



双精度绝对值的下限是：

$$\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308} \tag{18}$$

上限是：

$$\pm 2.0 \times 2^{1023} \approx \pm 1.8 \times 10^{308} \tag{19}$$

> Example: Show the binary representation of -0.75 in IEEE single precision format.
>
> $$\begin{aligned} -0.75 &= -2^{-1} - 2^{-2} \\ &= -0.11_2 \\ &= -1.1 \times 2^{-1}{}_2 \end{aligned} \tag{20}$$
>
> Therefore, the IEEE single precision format is:
>
> $$1 \quad 0111\ 1110 \quad 100\ 0000\ 0000\ 0000\ 0000\ 0000 \tag{21}$$

浮点运算

- **浮点加法**

  以$1.000_2 \times 2^{-1} - 1.110_2 \times 2^{-2}$为例

  1. **Alignment**：小指数对其到大指数，$-1.110_2 \times 2^{-2} = -0.111 \times 2^{-1}$

     > 小指数对其到大指数在之前补0，可能会导致末尾丢失；大指数对其到小指数在后面补0，可能会导致较高位丢失。显然后者损失的精度更大

  2. **Addiction Fraction**：加减，$1.000_2 - 0.111_2 = 0.001_2$
  3. **Normalize**：结果规范化，$0.001_2 \times 2^{-1} = 1.000_2 \times 2^{-4}$，同时检查是否出现overflow和underflow
  4. **Rounding**：将上面结果的位数转化为标准位数，如果舍入结果还需要规范化，返回上一步

- **浮点乘法**

  分别处理Sign、Exponent和Fraction

  1. **Add Exponents**：将两个Exponent相加，并减去一个Bias（因为Bias加了2次）
  2. **Multiply the Significands**：将两个1.Fraction相乘
  3. **Normolize**：规格化，和加法一样，检查是否出现overflow和underflow
  4. **Rounding**：舍入
  5. **Sign**：根据两个操作数的Sign确定结果的符号位

- **浮点除法**

和乘法非常接近，唯二的区别就是两Exponent应当相减，以及两数相乘处改为相除

- **精确算数**

  > Round Down / Round Up：向下/向上取整
  >
  > Round towards 0：向0取整，相当于是保留整数部分
  >
  > Round to Nearest Even：小数部分5开头时向偶数舍入，其余四舍五入

  IEEE Std 745浮点数最后还会多出3位，分别为Guard、Round和Sticky，让计算结果在舍入时可以更加精确

  其中Guard和Round相当于是留出的小数位，在加法中仅需要用到Guard进行舍入；乘除法中会用到Guard和Round两位。只要舍入中被移除的位右边有非零位，则Sticky置1，这样就可以判断右移过程中是否损失了小数位的1，实现round to nearest even

CHAPTER 2: INSTRUCTIONS

> **Instruction in Wide Variety**
>
> 有很多特点会带来指令的多样性，比如：
>
> 1. Types of internal storage in processor. 存储器的种类
> 2. The number of the memory operand in the instruction. 指令中操作数存储的位置（内存编号）
>
> **Stored-program Concept**
>
> 1. Instructions are represented as numbers.
> 2. Programs can be stored in memory to be read or written just like numbers.

Operations of the Computer Hardware

设计原则

1. **Simplicity Favors Regularity** 简明规范
   - Only 1 operation per instruction. 每条指令仅执行一个操作

- Exactly 3 variables.

2. **Smaller is Faster**
   - Arithmetic instructions use register operands 算数指令采用寄存器作操作数
   - RISC-V: 32×64bit register file RSIC-V指令采用32个64bit（即1 double word）的寄存器，用来寄存常用的数据。这个数量是成本和性能的平衡
3. **Good Design Demands Good Compromises** 有好的机器码对指令进行编译
   - All instructions in RISC-V have the same length.

RISC-V Instruction Set

The Memory Alignment of RISC-V

1. **Memory is Byte Addressed**：每个地址都定位到Memory中的一个字节（byte）
2. **Little Endian**：最低位存储在最低地址，也就是说最低位存在最前面，和正常阅读顺序相反

> 因为前面提到每个地址定位到一个字节，假设我们有一个16进制数12345678（因为一位16进制数需要4位2进制表达，所以1byte可以表示2位16进制数）
>
> Big End表示如下：
>
> | 地址 | 0x00 | 0x01 | 0x10 | 0x11 |
> | --- | --- | --- | --- | --- |
> | 内容 | 12 | 34 | 56 | 78 |
>
> Little End表示如下：
>
> | 地址 | 0x00 | 0x01 | 0x10 | 0x11 |
> | --- | --- | --- | --- | --- |
> | 内容 | 78 | 45 | 34 | 12 |

3. **No Alignment Requirement in Memory**：允许数据在任何内存地址存储，而不强制要求特定的数据类型（如整数或浮点数）必须放置在特定的对齐边界上。这样更节省空间，但可能会影响数据读取的速度

> 这里给出一个数据对齐的例子。我们定义如下结构体：

```
struct{
int   a;
char  b;
char  c[2];
char  d[3];
float e;
}
```

给出下面两种对齐方式：

<table>
<tr><td colspan="4">e</td></tr>
<tr><td>d[1]</td><td>d[2]</td><td>No use</td><td>No use</td></tr>
<tr><td>b</td><td>c[0]</td><td>c[1]</td><td>d[0]</td></tr>
<tr><td colspan="4">a</td></tr>
</table>

正确对齐

<table>
<tr><td colspan="2">e</td><td>No use</td><td>No use</td></tr>
<tr><td>d[1]</td><td>d[2]</td><td colspan="2">e</td></tr>
<tr><td>b</td><td>c[0]</td><td>c[1]</td><td>d[0]</td></tr>
<tr><td colspan="4">a</td></tr>
</table>

错误对齐

右边的错误对齐是因为一次只能读取一个地址，即内存中的一行，右边的对齐方式会导致e无法正常读出

RISC-V指令格式

RISC-V一共有7大指令格式，汇总在本章末尾给出。源寄存器用于读取，目标寄存器内存储的地址（或内容）进行修改以指向操作结果（或改变寄存器所存储的内容）

1. **R-type**

   2个源寄存器（操作数），1个目标寄存器（结果），操作数进行运算后存入目标寄存器

funct7	rs2	rs1	funct3	rd	opcode

```
op  rd, rs1, rs2
```

以 `add  x9, x20, x21` 为例，其机器码应为：

funct7	rs2	rs1	funct3	rd	opcode
$0_{10} = 0000000_2$	$21_{10} = 10101_2$	$20_{10} = 10100_2$	$0_{10} = 000_2$	$9_{10} = 01001$	$51_{10} = 0110011_2$

$$\Rightarrow 0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_2 = 015A04B3_{16} \quad (22)$$

2. **I-type**

   1个源寄存器，1个目标寄存器，将操作数与立即数进行运算后存入目标寄存器

i(11:0)	rs1	funct3	rd	opcode

   ```
   op  rd, imm(rs1)
   op  rd, rs1, imm
   ```

   > 除了实现addi这样需要常数的运算操作，imm还可用作地址的偏移量，比如：
   >
   > ```
   > ld  x9, 64(x22)
   > ```

3. **S-type**

   2个源寄存器，因为目标是将数据写入内存（即存储），不会改变寄存器所寄存的内存地址（改变的是其所指向的内存地址存储的值），所以无目标寄存器

funct6	i(5:0)	rs1	funct3	rd	opcode

   ```
   op  rs2, imm(rs1)
   op  rs2, rs1, imm
   ```

   > 以sd为例。将rs2的值写入rs1偏移后的位置
   >
   > ```
   > sd  rs2, 64(rs1)
   > ```

4. **SB-type**

   无目标寄存器，2个源寄存器。注意到SB型指令是不需要i(0)的，因为跳转一条指令PC + 4，所以立即数一定是4的倍数，即最低2位一定为0，由此可以省去一位

i(12, 10:5)	rs2	rs1	funct3	i(4:1, 11)	opcode

   ```
   op  rs1, rs2, imm
   ```

寻址方式

> 考察PC取值范围时，答案一定是4的倍数，所以不会出现± 2的情况，一定是-4。还有注意16进制与2进制的转换，2进制每4位合为1位16进制，老老实实对着看

1. **Immdiate Addressing**

```
addi  x5, x6, 4
```

2. **Register Addressing**

```
add   x5, x6, x7
```

3. **Base Addressing**：基于一个基准进行偏移

```
ld    x5, 100(x6)
```

4. **PC-relative Addressing**：基于Program Count的寻址

```
beq   x5, x6, L1
```

常用指令

书上出现的所有指令均列在本章末尾，此处仅列出较为常见的

1. **ld、sd、add、addi和sub**

   这些是最基本的指令，下面给出一个例子，将C语言于RISC-V对应起来

```
A[12] = h + A[8]; // h -> x21, A -> x22
```

```
ld  x9, 64(x22) # 将后一个操作数的偏移载入到x9中（从内存中载入到寄存器）
add x9, x21, x9 # op1 = op2 + op3
sd  x9, 96(x22) # 将前一个操作数载入到后一个的偏移中（从寄存器中载入到内存）
```

   > x22中寄存的是A(0)在内存中的地址，而非A(0)的具体值
   >
   > 注意到偏移量（offset）是以byte为单位的，因此第一个offset为$8 \times \text{sizeof}(\text{int}) = 64$，第二个为$12 \times \text{sizeof}(\text{int}) = 96$

   addi即add immediate，用于简化直接与常数相加的过程。下面两段代码等效：

```
ld  x9, AddrConstant4(x3)
add x22, x22, x9
# Assume that AddrConstant4 is address pointer of constant 4
```

```
addi  x22, x22, 4
```

2. **slli、srli、and、andi、or、ori、xor和xori**

   位运算，分别为shift left、shift right、与、或、异或

3. **slt、blt、bltu、bge和bgeu**

```
slt x5, x19, x20  # 如果x19 < x20，则x5 = 1
```

```
blt rs1, rs2, L1  # 如果rs1 < rs2，跳到语句L1
bge rs1, rs2, L2  # 如果rs1 >= rs2，跳到语句L2
```

   bltu、bgeu用于无符号数（unsigned）的比较，用法与blt、bge一致

4. **beq和bne**

   - **if-else**

```
if (i == j)
  f = g + h;
f = f - i;
```

```
    beq x22, x23, L1  # 如果x22和x23相等，跳转到L1
    add x19, x20, x21
L1: sub x19, x19, x22
```

     注意本程序段和if-else分支不同，这样写无论如何 `L1` 都会被运行。因此，我们引入 `bne` 实现if-else效果

```
if (i == j)
  f = g + h;
else
  f = g - h;
```

```
      bne x22, x23, Else  # 如果x22和x23不想等，直接跳转到
Else
      add x19, x20, x21
      beq x0, x0, EXIT
Else: sub x19, x20, x21
Exit: ...
```

- **循环**

```
Loop: g = g + A[i];
      i += j;
      if (i != h)
        goto Loop;
// i -> x22, A -> x25,
```

```
Loop: slli  x10, x22, 3     # x10 = 8 * i，即x10 =
sizeof(int) * i
      add   x10, x10, x25   # x10 = x25 + sizeof(int) *
i，定位到A[i]
      ld    x19, 0(x10)
      add   x20, x20, x19   # g += A[i]
      add   x22, x22, x23   # i += j
      bne   x22, x21, Loop  # 如果i和h不想等，返回到Loop
```

while循环同样也是被支持的，其实只是将判断提前了，循环返回处选择一个始终为真的条件

```
while (save[i] == k)
  i += 1;
// k -> x24
```

```
Loop: slli  x10, x22, 3
      add   x10, x10, x25
      ld    x9, 0(x10)    # 到此为止和上面一样，目的是把
save[i]取出来
      bne   x9, x24, Exit # 如果save[i]和k不想等，直接退出
      addi  x22, x22, 1
      beq   x0, x0, Loop  # 这里始终为真，也就是说程序不会在此
退出，唯一的判断在上面的bne
Exit: ...
```

5. **jal和jalr**

   Jump and link (register)，寄存下一条指令的地址，并跳转到指定位置

```
jal   x1, offset  # x1寄存下一条指令的地址，并向后跳转offset执行
 (x1 = PC + 4, PC += offset)
jalr  x1, 0(x0)   # x1寄存下一条指令的地址，并跳转到x0 + 0处执行
 (x2 = PC + 4, PC = x0 + 0)
```

6. **lr.d和sc.d**

   在多条程序同时运转时，如果一个程序需要在某地址写入，另一个程序需要在某地址读出，两者的先后顺序可能会导致意料之外的问题。所以需要引入其他指令来解决这个问题

```
lr.d  rd, (rs1)        # ld，并预留该内存地址（将改地址记入一个保留表中，后续sc.d将会检查该地址是否发生了改变）
sc.d  rd, (rs1), rs2   # 将rs2存到rs1中，如果自lr.d以来rs1指向的地址未发生改变，rd设置为0，否则非0（后续可以增加检验步骤，如果rd != 0，重做lr.d和sc.d）
```

递归的实现

```c
int fact(int n) {
  if (n < 1)
    return 1;
  else
    return (n * fact(n - 1));
}
```

```
# 为了可以递归并计算，栈里要寄存2个元素，返回位置x1、此时n的大小x10
Fact: addi  sp, sp, -16   # sp是栈指针，初始化时往回退2个位置
      sd    x1, 8(sp)     # 栈的第1个位置：返回位置x1
      sd    x10, 0(sp)    # 栈的第0个位置：此时的n值x10
      addi  x5, x10, -1   # x5 = n - 1
      bge   x5, x0, L1    # 如果n >= 1，跳过下面一段跳转到L1
      addi  x10, x0, 1    # 直到n = 0时，进入这一步；令x10为1，此时x10用于记录计算结果
      addi  sp, sp, 16    # sp向后16，出栈
      jalr  x0, 0(x1)     # 跳转到刚刚返回的位置，即L1的第三行
L1:   addi  x10, x10, -1  # n -= 1
      jal   x1, Fact      # 回跳到Fact（此时n减掉了1），x1指向下一句(addi)
      addi  x6, x10, 0    # x6暂存当前计算结果
```

```
        ld      x10, 0(sp)      # 将出栈的n值载入x10
        ld      x1, 8(sp)       # 将出栈的返回位置载入x1
        addi    sp, sp, 16      # sp向后16，继续出栈
        mul     x10, x10, x6    # 计算结果
        jalr    x0, 0(x1)       # 跳转到上一个返回位置（即刚刚出栈的返回位置）
```

章末汇总

指令格式汇总

	6	1	5	5	3	5	7
R	funct7		rs2	rs1	funct3	rd	opcode
I	i[11:0]			rs1	funct3	rd	opcode
I	funct6	i[5:0]		rs1	funct3	rd	opcode
S	i[11:5]		rs2	rs1	funct3	i[4:0]	opcode
SB	i[12, 10:5]		rs2	rs1	funct3	i[4:1, 11]	opcode
UJ	i[20, 10:1, 11, 19:12]					rd	opcode
U	i[31:12]					rd	opcode

> **op** / **opcode**: Basic operation of the instructions.
>
> **rd**: Destination register number.
>
> **funct3**: 3-bit function code (additional opcode).
>
> **rs1**: First register source operand.
>
> **rs2**: second register source operand.
>
> **funct7**: 7-bit function code (additional opcode).
>
> **i**: Immediate, constant operand or offset added to base address.

指令汇总

Category	Inst	Example & Comments			FMT	OpCode	Funct3	Funct6/7
Arithmetic	add	Add	add rd, rs1, rs2	rd = rs1 + rs2	R	0110011	000	0000000
	sub	Subtract	sub rd, rs1, rs2	rd = rs1 - rs2	R	0110011	000	0100000
	addi	Add imm	addi rd, rs1, -20	rd = rs1 + (-20)	I	0010011	000	n.a.
	slt	Set if less than	slt rd, rs1, rs2	rd = rs1 < rs2 ? 1 : 0	R	0110011	010	0000000
	sltu	slt, unsigned	sltu rd, rs1, rs2	rd = rs1 < rs2 ? 1 : 0	R	0110011	011	0000000
	slti	slt, imm	slti rd, rs1, imm	rd = rs1 < imm ? 1 : 0	I	0010011	010	n.a.
	sltiu	slt, imm & unsigned	sltiu rd, rs1, imm	rd = rs1 < imm ? 1 : 0	I	0010011	011	n.a.
	mul	mul, lower 64 of result	mul rd, rs1, rs2	rd = rs1 * rs2 (lower 64)	R	-	-	-
	mulh	mul, upper 64 of result	mulh rd, rs1, rs2	rd = (rs1 * rs2) >> 64	R	-	-	-
	mulhu	mulh, unsgn * unsgn	mulhu rd, rs1, rs2	rd = (rs1 * rs2) >> 64	R	-	-	-
	mulhsu	mulh, sgn * unsgn	mulhsu rd, rs1, rs2	rd = (rs1 * rs2) >> 64	R	-	-	-
	div	divide	div rd, rs1, rs2	rd = rs1 / rs2	R	-	-	-
	divu	divide unsigned	divu rd, rs1, rs2	rd = rs1 / rs2	R	-	-	-
	rem	remainder	rem rd, rs1, rs2	rd = rs1 % rs2	R	-	-	-
	remu	remainder unsigned	remu rd, rs1, rs2	rd = rs1 % rs2	R	-	-	-
Data transfer	ld	Load dword	ld rd, 40(rs1)	rd = [rs1 + 40]	I	0000011	011	n.a.
	sd	Store dword	sd rs2, 40(rs1)	[rs1 + 40] = rs2	S	0100011	011	n.a.
	lw	Load word	-	-	I	0000011	010	n.a.
	lwu	Load unsgn word	-	-	I	0000011	110	n.a.
	sw	Store word	-	-	S	0100011	010	n.a.
	lh	Load half word	-	-	I	0000011	001	n.a.
	lhu	Load unsgn hword	-	-	I	0000011	101	n.a.
	sh	Store hword	-	-	S	0100011	001	n.a.
	lb	Load byte	-	-	I	0000011	000	n.a.
	lbu	Load unsgn byte	-	-	I	0000011	100	n.a.
	sb	Store byte	-	-	S	0100011	000	n.a.
	lui	Load upper imm	lui rd, 0x12345	rd = 0x12345000	U	0110111	n.a.	n.a.
	auipc	Add upper imm to PC	auipc rd, 0x12345	rd = PC + 0x12345000	U	0010111	n.a.	n.a.
Logical	and	And	and rd, rs1, rs2	rd = rs1 & rs2	R	0110011	111	0000000
	or	Or	or rd, rs1, rs2	rd = rs1 | rs2	R	0110011	110	0000000
	xor	Exclusive or	xor rd, rs1, rs2	rd = rs1 ^ rs2	R	0110011	100	0000000
	andi	And imm	andi rd, rs1, imm	rd = rs1 & imm	I	0010011	111	n.a.
	ori	Or imm	ori rd, rs1, imm	rd = rs1 | imm	I	0010011	110	n.a.
	xori	Xor imm	xori, rd, rs1, imm	rd = rs1 ^ imm	I	0010011	100	n.a.
Shift	sll	shift left logical	sll rd, rs1, rs2	rd = rs1 << rs2	R	0110011	001	0000000
	srl	shift right logical	srl rd, rs1, rs2	rd = rs1 >> rs2 (zExt)	R	0110011	101	0000000
	sra	shr arithmetic	sra rd, rs1, rs2	rd = rs1 >> rs2 (sExt)	R	0110011	101	0100000
	slli	shl logical imm	slli rd, rs1, imm	rd = rs1 << imm	I	0010011	001	000000
	srli	shr logical imm	srli rd, rs1, imm	rd = rs1 >> imm (zExt)	I	0010011	101	000000
	srai	shr arith imm	srai rd, rs1, imm	rd = rs1 >> imm (sExt)	I	0010011	101	010000
Conditional Branch	beq	branch if equal	beq rs1, rs2, offset	if (rs1==rs2) PC+=offset	SB	1100011	000	n.a.
	bne	branch if not equal	bne rs1, rs2, offset	if (rs1!=rs2) PC+=offset	SB	1100011	001	n.a.
	blt	br if less than	blt rs1, rs2, offset	if (rs1<rs2) PC+=offset	SB	1100011	100	n.a.
	bge	br if greater or eq	bge rs1, rs2, offset	if (rs1>=rs2) PC+=offset	SB	1100011	101	n.a.
	bltu	blt, unsigned	bltu rs1, rs2, offset	if (rs1<rs2) PC+=offset	SB	1100011	110	n.a.
	bgeu	bge, unsigned	bgeu rs1, rs2, offset	if (rs1>=rs2) PC+=offset	SB	1100011	111	n.a.
Unconditional Branch	jal	jump and link	jal rd, offset	rd=PC+4; PC+=offset	UJ	1101111	n.a.	n.a.
	jalr	jump and link reg	jalr rd, 100(rs1)	rd=PC+4; PC=rs1+100	I	1100111	000	n.a.

地址对照

Register	Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-x7	t0-t2
x8	s0/fp
x9	s1
x10-x11	a0-a1
x12-x17	a2-a7
x18-x27	s2-s11
x28-x31	t3-t6

CHAPTER 4: PROCESSOR

> 作为汇编语言，其在底层硬件上要有相应的实现。本章就介绍RISC-V的一种硬件实现

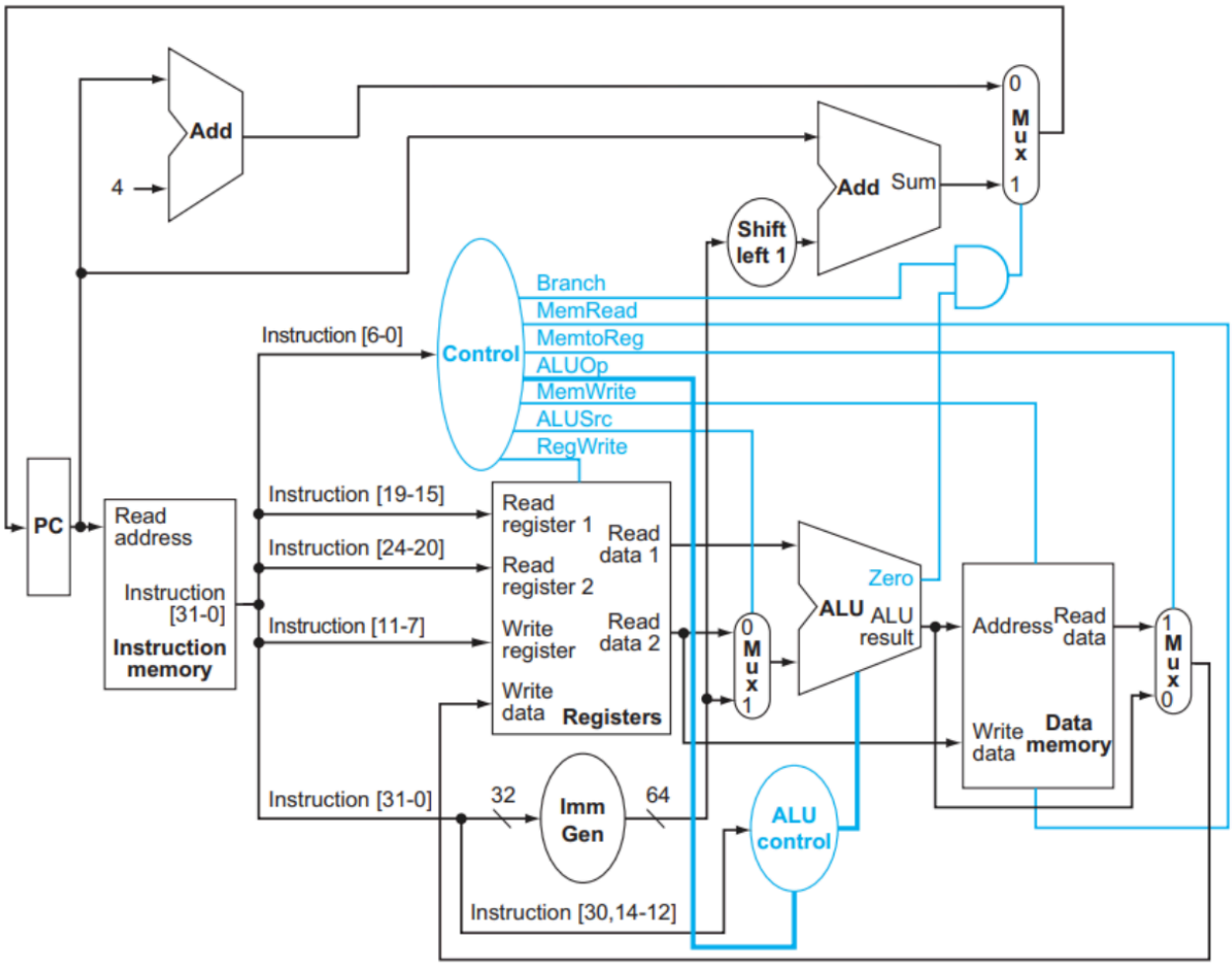
Datapath 数据通路

Instruction Execution

1. **取指**（Fetch the instruction from the memory）：将指令从内存中取出
2. **译码**（Decode and read the registers）：汇编语言 -> 机器语言
3. **操作**：根据不同的指令类型，运用ALU进行相关操作
    - **内存引用**（Memory Reference）：ALU用于计算内存地址（比如计算偏移后的地址）
    - **算数逻辑**（Arithmetic Logical）：ALU用于进行算数或逻辑运算

- **分支指令**（Branch）：ALU用于分支比较

这是一个RISC-V核心指令的硬件实现，蓝色的部分是控制通路（选择ALU进行的操作类型），黑色的部分是数据通路（负责操作数的操作和传输）。组合起来是一个完整的数据通路，实现指令执行



因为每个时钟周期中，一条路径上只能进行一个操作。因此，如果将指令（instruction）和数据（data memory）存在同一地方会导致效率低下，实际上在内存中两者分块存储，互不干扰

同时，采用MUX保证同一条数据通路上不同时存在多个相互干扰的数据

Instruction Execution in RISC-V

> 其实RISC-V的指令执行过程和上面的一般过程是一样的，这里只是更细化一些而已
>
> 因为Memory Access只有ld和sd需要，所以只有I型指令会需要经过所有的前五条步骤

1. **Fetch**：取指，PC + 4

2. **Decode & Read Operands**：指令译码为机器语言，读取两个操作数（不管有没有用到，都会读取rs1和rs2）
3. **Executive Control**：调用ALU进行相应计算
4. **Memory Access**：读写memory。如果指令是rd或ld等I型指令，需要对memory进行读写操作的，则需进行这一步；如果是R型指令或branch指令，因为其操作数和结果直接存在寄存器中，所以不需要进行这一步
5. **Write Results to Register**：对于R型指令，ALU的计算结果存入rd；对于I型指令，Memory data存入rd
6. **Modify PC** for branch instructions

不同指令类型的数据通路

> 本部分的图在平板上有画，个人感觉没有必要搬上来，有需要的话可以参考平板

1. **R-format**

   对于R型指令而言，其需要2个源寄存器（Read Register），1个目标寄存器（Write Register）；将读到的数据给到ALU，进行计算后返回给目标寄存器存储

   ALUop = 10，ALUSrc、Branch、MemRead、MemWrite、MemtoReg均为0，RegWrite = 1

2. **I-format**

   对于I型指令，有一个Immediate Generation Unit，负责将32位指令中的立即数取出并转化为64位的格式，这样就可以与rs1相加（虽然rs1是5位的，但别忘了RISC-V中的寄存器是64位的，5位的rs1只是寄存器的编号而已）。在I型指令中，ALU用于加法运算，执行偏移量（offset）的相加

   在ld中，ALUop = 00，ALUSrc、MemWrite、RegWrite、MemtoReg为1，其余为0

3. **S-format**

   S型指令和I型指令很像。区别在于，前者在Data memory中读出数据写入reg，后者将reg读出的数据写入memory中（建议以ld和sd为例进行记忆）

   在sd中，ALUop = 00，ALUSrc、MemRead为1，MemtoReg无所谓（因为RegWrite = 0，无论如何不会写入）

4. **Branch Instructions**

ALU：执行减法运算（因为分支指令需要进行操作数的比较）

目标地址计算：如果满足条件，PC需要加上偏移量

1. **符号位扩展**：处理负值偏移
2. **左移1位**：因为SB型指令中的立即数是没有 `i[0]` 的，所以在ALU计算时需要左移1位补0
3. PC += 拓展后的imm
4. **Branch = 1**：无论条件是否满足，Branch都为1。因为给到MUX的是Branch和Zero取与，Branch = 1不代表一定跳转到计算地址，如果Zero给0，则依旧跳转到PC+ 4

ALUop = 01，Branch = 1，MemtoReg = X，其余为0

5. **J/jal-format**

   注意此处jump = 1，MemtoReg = 10，RegWrite = 1，ALUSrc = ALUop = X（因为用不到ALU的运算结果，算出什么都可以）

Signals for Datapath

因为一个单元需要负责所有指令的实现，而不同类型的指令需要通过不同的数据通路。所以我们需要一些控制信号来负责不同

7 Control Signals

Signal Name	0 (00)	1 (01)	10
RegWrite	-	数据写入Register	-
ALUSrc	ALU的第二个输入来自rs2	ALU的第二个输入来自拓展后imm	-
Branch	PC + 4	PC来自branch的目标跳转地址（无论是否为PC + 4都置为1）	-
Jump	PC + 4，或来自branch的目标跳转地址	PC由jump的目标地址决定	-
MemRead	-	从内存中读入数据	-
MemWrite	-	向内存中写入数据	-
MemtoReg	写入Register的信号来自ALU	写入Register得信号来自Memory	写入Register的信号为PC + 4

ALU Control

opcode	ALUop	funct7	funct3	ALU function	ALU Control
ld	**00**	XXXXXXX	XXX	add	**0010**
sd	**00**	XXXXXXX	XXX	add	**0010**
beq	**01**	XXXXXXX	XXX	subtract	**0110**
R-type	**10**	0000000 0100000 0000000 0000000 0000000	000 000 111 110 010	add subtract AND OR Slt	**0010** **0110** **0000** **0001** **0111**

ALUop来自opcode，决定指令类型；funct和funct3共同决定ALU进行的计算；ALU执行的计算类型和ALU control一一对应

Pipelining 流水线

单周期处理器存在诸多问题。因为Longest delay determines clock period，时钟周期 = 执行单条指令所需要的最长时间（ld最长，因为其经历的过程最多），而所有指令都共用一个时钟周期；这会导致很多指令本不需要这么长时间也被迫消耗一个时钟周期才能完成

同时，因为单周期处理器要等一条指令完成后才能进行下一条指令，会导致运行速度缓慢

Five Stages 五级流水线

在RISC-V中，一条指令的执行一共有5个步骤

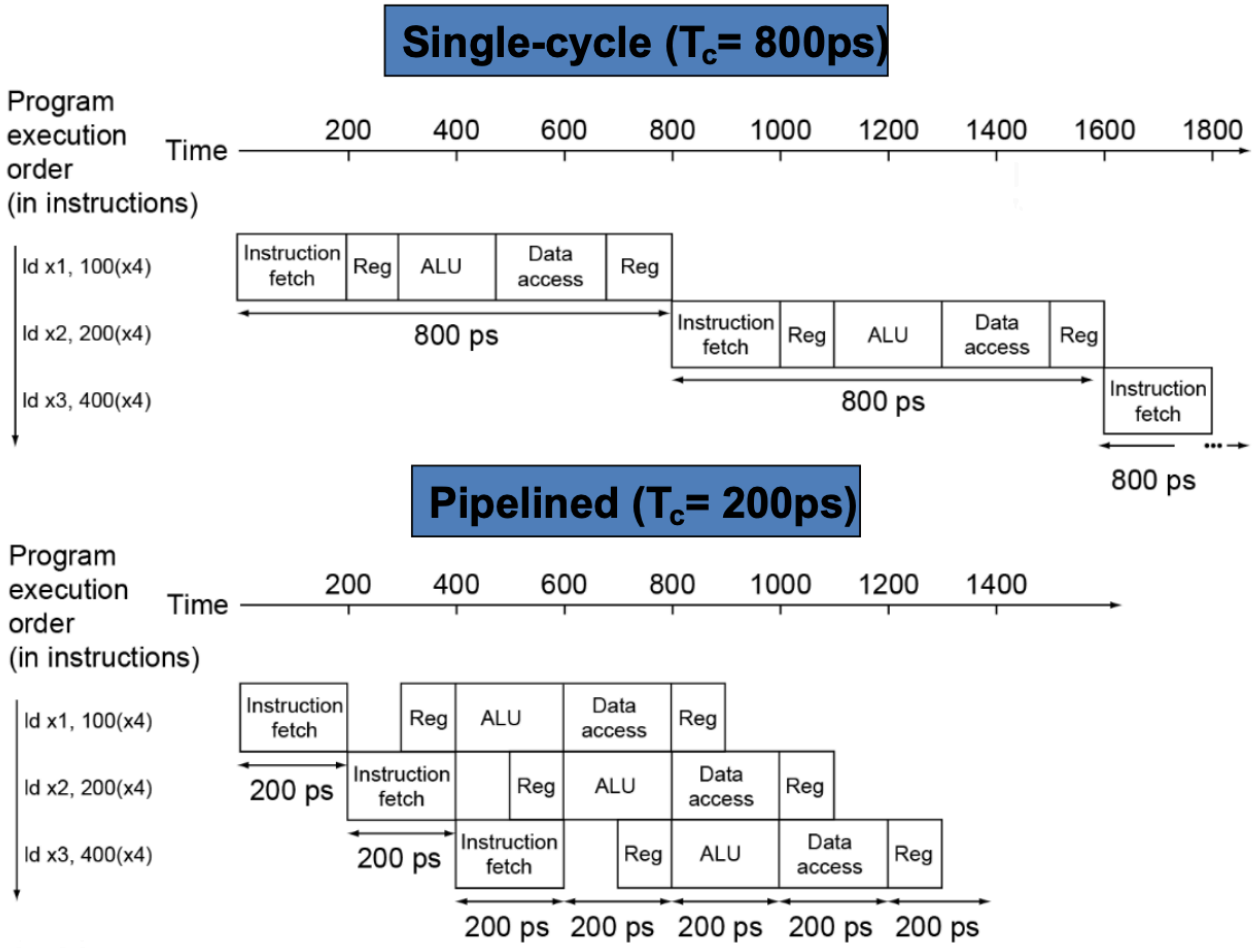
1. **IF**：Instruction Fetch，从memory中取出指令
2. **ID**：Instruction Decode & register read，指令解码、读入寄存器
3. **EX**：Execution，处理计算
4. **MEM**：访问内存
5. **WB**：Write Back，将结果写回寄存器

流水线

1. **Balanced**：即每个stage时间相同。因为流水线的时钟周期由最长stage决定，所以unbalanced的加速效果会更少

   > 实际上，stage间存在一定的传输延时，所以流水线的Latency会比单周期略长。时钟周期也会比上述的理想情况长一些
   >
   > 容易发现，一个M stage的流水线处理器执行N条指令所需要的时间为：
   >
   > $$\text{Clock Cycle} \times (M - 1 + N) \tag{23}$$

2. **Latency**：Time for each instruction，一条指令从开始到结束所需要的时间。Speedup due to increased throughput, latency does not decrease，通过提高吞吐量（单位时间内执行的指令数），无法缩减单条指令完成所需要的时间



Pipelining Hazards

Structure Hazard

1. **定义**：硬件不支持多条指令在同一时钟周期进行，一个需要用到的资源正处于忙碌状态

> 注意和Data Hazaed进行区分，Structure Hazard是因为几个信号同时竞争一个资源带来的，而Data Hazard是因为数据读写顺序混乱导致的逻辑错误
>
> - ld和sd对内存读写的竞争
> - Control信号的传递
> - 本条指令在ID阶段时，需要解码上一步IF读出的指令；但此时下一条指令正在进行IF，读出的新指令会覆盖之前的指令，那解码的到底是本条指令还是下条呢?

2. **解决方法**：在每两个stage间使用一些寄存器存储可能会冲突的内容。具体内容放在RISC-V Pipelined Datapath介绍

Data Hazard

1. **定义**：一条指令需要用到上一条指令完成的数据读写

> 比如说下一条指令的ID需要用到上一条的WB，但此时上一条指令实际上正在进行EX步骤，还未写入，这样就会导致产生逻辑错误

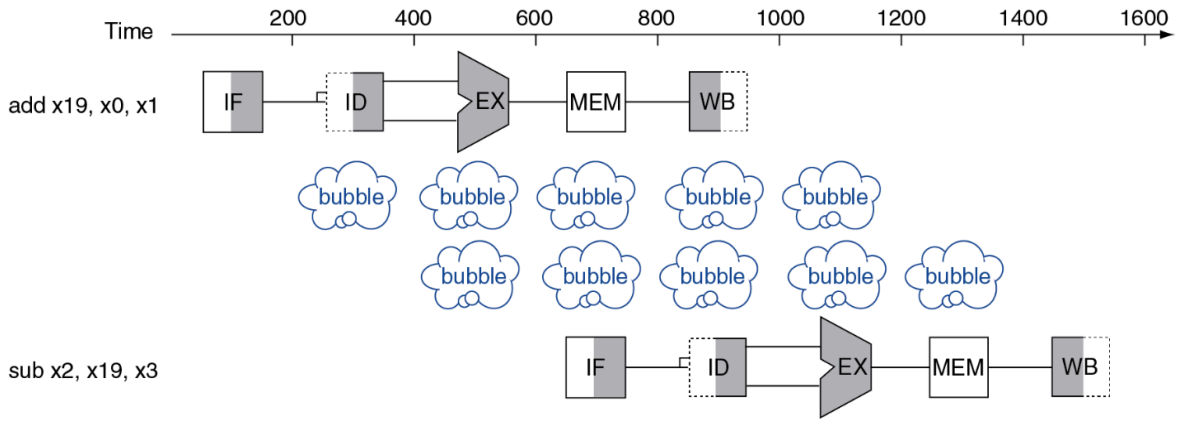
```
addi  x22, x22, 1
add   x23, x22, x21  # 这条指令需要用到上一条x22的结果，但按照流水线会导致上一条还未存入本条就已开始运行
```

2. **解决方法**

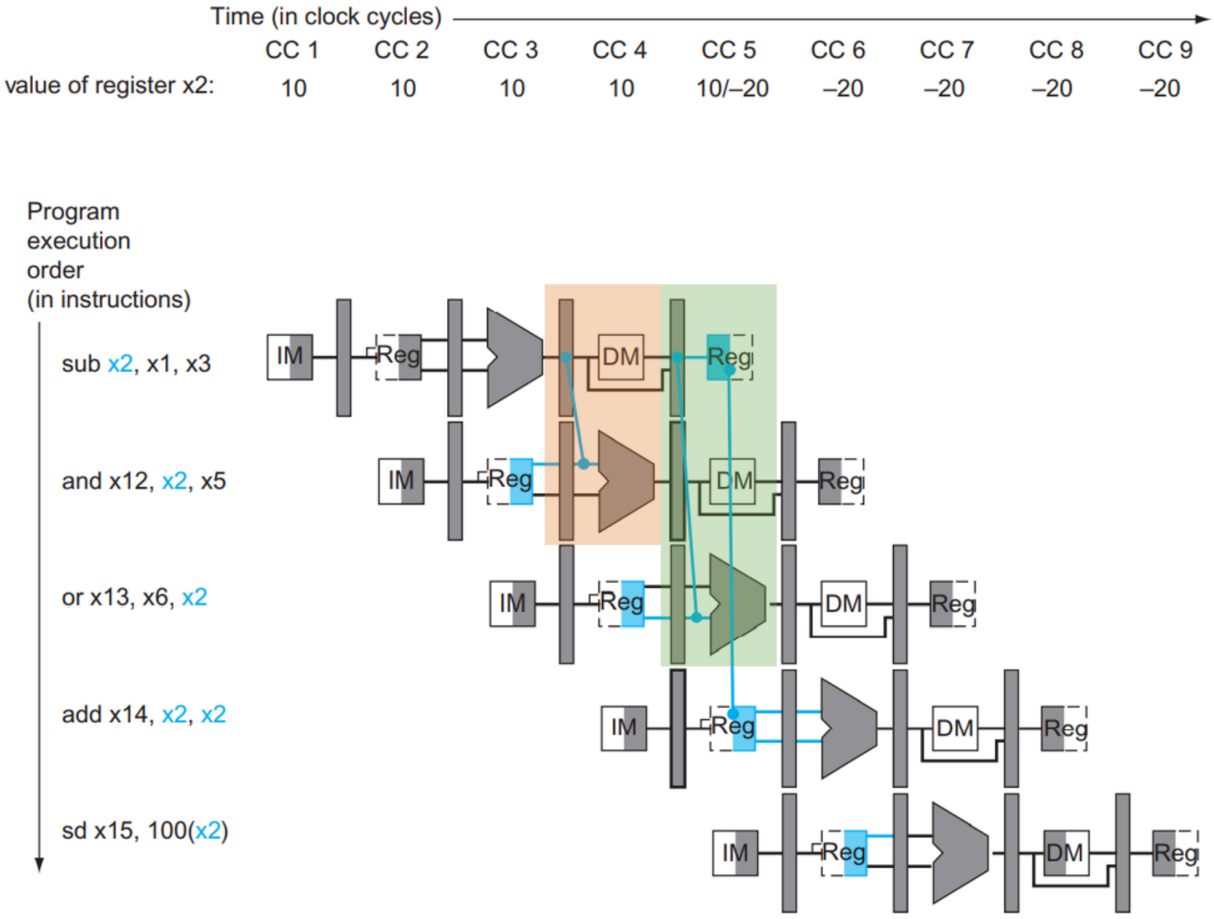
   - **Forwarding / aka Bypassing**

     首先，我们让IF、ID指令（因为它们涉及读出）均占用后半个时钟周期，而WB使用前半个时钟周期。如此可将WB和ID组合到一个时钟周期中而不会互相干扰

     > 利用时钟的上升沿和下降沿（课本中采用上升沿作为一个时钟周期的开始）实现，我们可以将ID的开始设置为时钟的下降沿

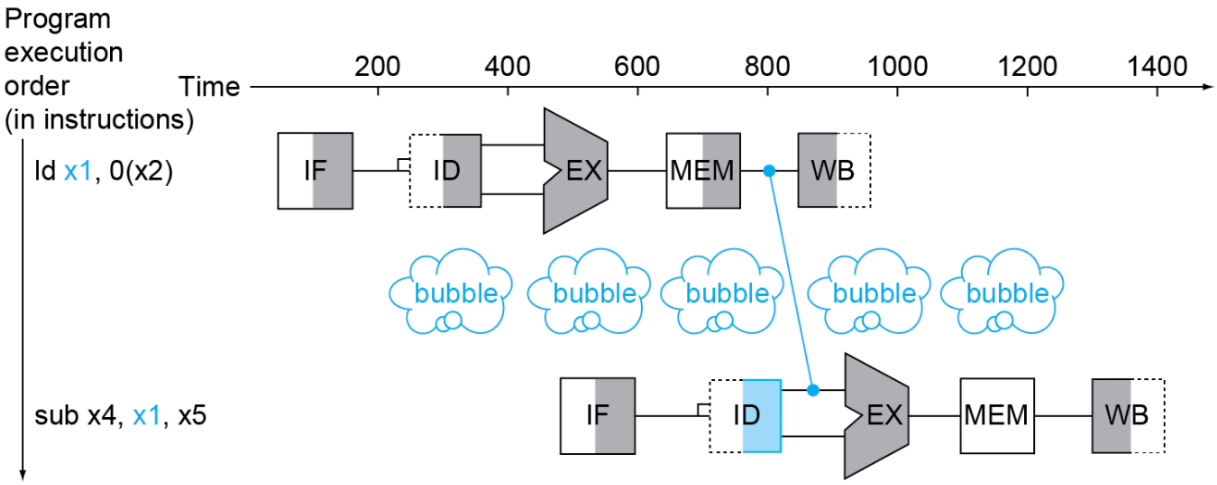


     像上面这样采用几个bubble虽然可以解决问题，但消耗时间略多，还有改进空间。观察到虽然下一条指令ID要用到的操作数在上一条指令的WB阶段才写入，但实际上在EX阶段就已求得，只需将上条指令EX的输出引入本条指令EX的输入即可。实现如下（图中的竖框框是为了解决Structure Hazard而引入的寄存器）



- **Load-use Data Hazard**：Can't always avoid stalls by forwarding

  对于load指令，因为要等MEM步骤才能获得数据，即本条指令的EX步骤只有在上一条的MEM结束后才能执行。因此只能让后一条指令间隔一行执行



3. **Code Scheduling to Advoid Stalls**

   通过改变语句顺序来减少ld所需要的bubble

```
ld  x1, 0(x0)
ld  x2, 8(x0)
# bubble
add x3, x1, x2
sd  x3, 24(x0)
ld  x4, 16(x0)
# bubble
add x5, x1, x4
sd  x5, 32(x0)
```

对于上面这段代码，没有bubble的情况下需要$7+5-1=11$个周期完成，外加2个bubble，一共需要13个时钟周期完成

```
ld  x1, 0(x0)
ld  x2, 8(x0)
ld  x4, 16(x0)
add x3, x1, x2
sd  x3, 24(x0)
add x5, x1, x4
sd  x5, 32(x0)
```

转换语句顺序，避免ld之后立即调用其作用的寄存器。修改后无需bubble，可在11个时钟周期内完成

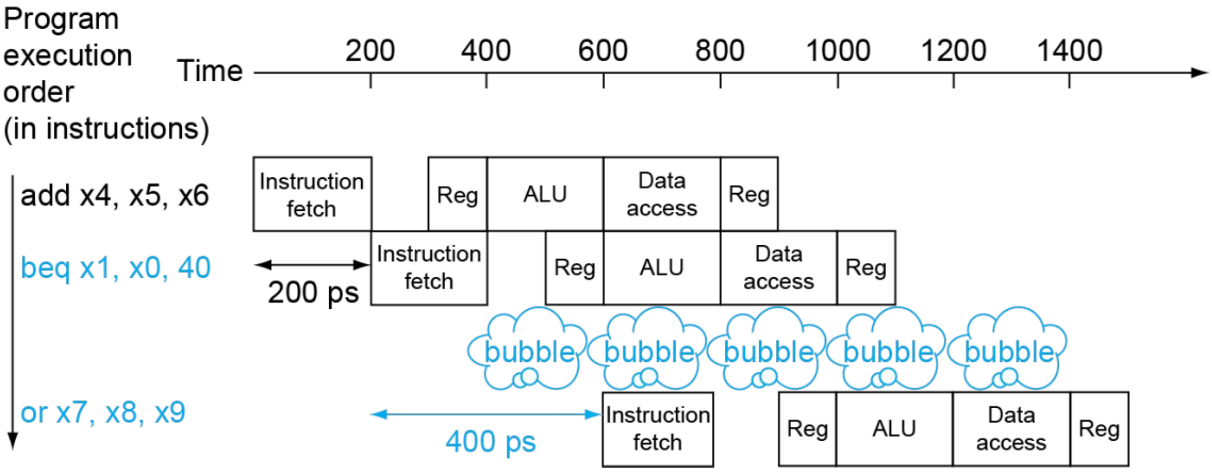
Control Hazard

1. **定义**：下一条指令是否为PC + 4取决于上一条的结果

> Control Hazard即用于branch的数据未准备好，导致PC跳转出现问题。本应跳过的语句可能会因为branch跳转不及时而运行
>
> ```
>     beq   x5, x0, L1
>     addi  x5, x5, 1    # 本条指令是否执行取决于上一条beq的结果，但按照流水线会导致无论beq如何都运行本条
>     ...
> L1: ...
> ```

2. **解决方案**
    - **将比较提前到ID阶段进行**

添加硬件，让比较与ID同时进行。无需等待上一条指令完全完成才进行下一条，但还是需要一行bubble。因此在复杂的流水线中较少采用



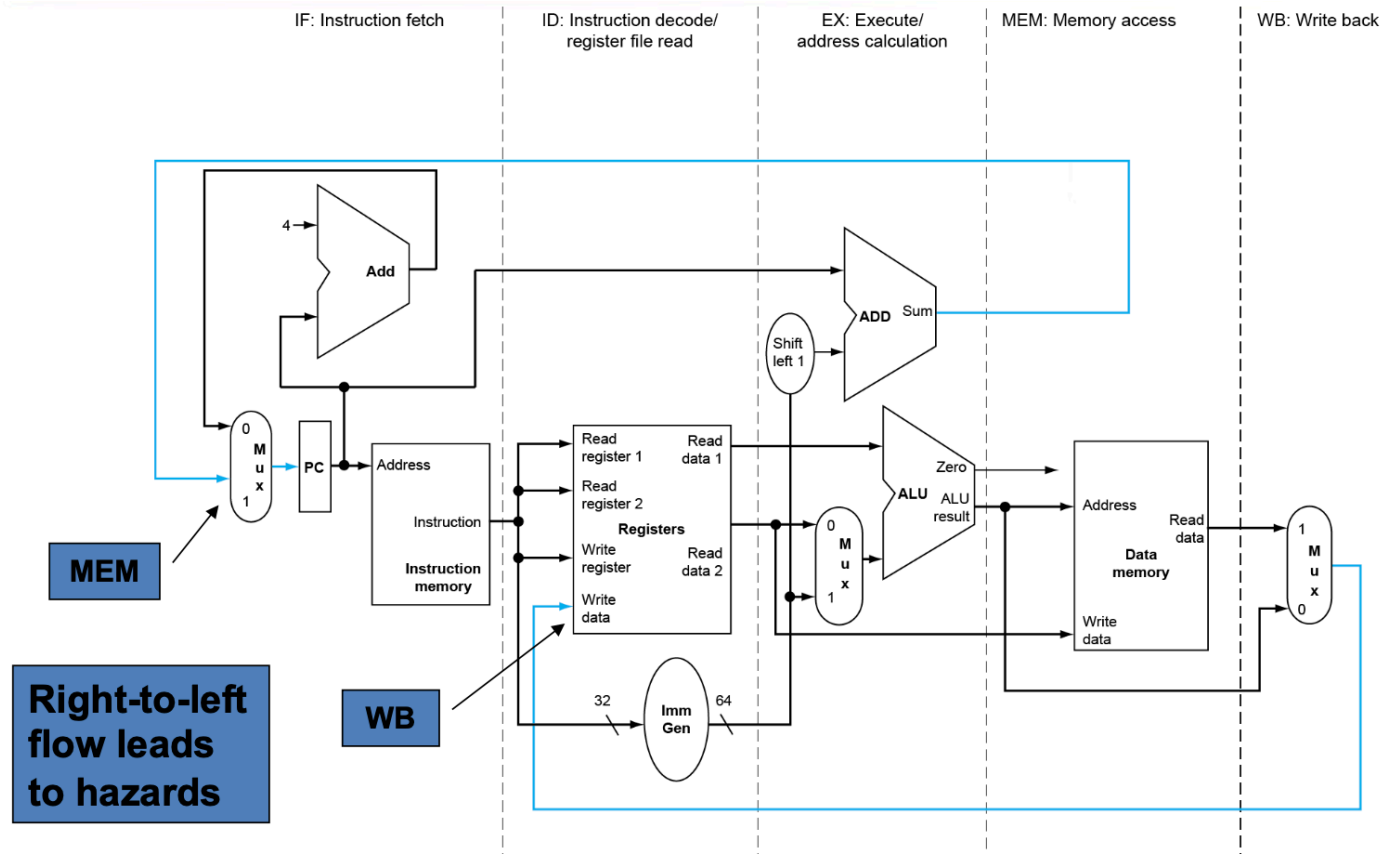
- **Prediction Outcome of Branch**

  Only stall if prediction is wrong，只要预测正确就无需stall；如果错误，将之后所有的运行都flush冲洗掉，re-fetching，跳转到正确的地方运行

  1. **Static Branch Prediction**：假设某种判断总是为真/假
  2. **Dynamic Branch Prediction**：有专门的硬件用于统计之前branch的情况，并假设之后的behavior will continue the trend（比如记录上次本branch的情况，并认为本次判断会和上次结果一致）

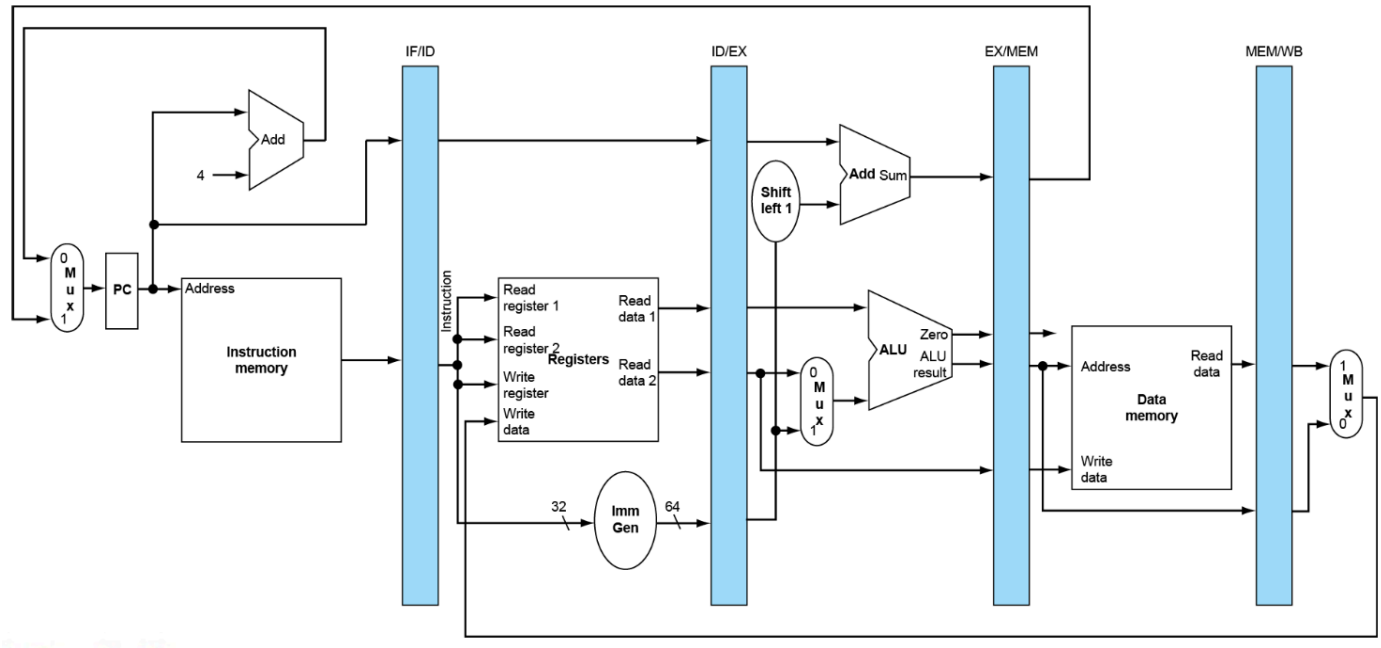
RISC-V Pipelined Datapath

下面是一个单周期处理器的示意，虚线划分出了硬件上的5个阶段。正常来说，在流水线实现中，一个处理器会同时执行5条指令，所有步骤从左向右进行；但一些情况下数据需要从右向左传递，导致数据冒险或控制冒险



假设我们直接用单周期处理器运行流水线，会发现每条指令都需要自己独立的数据通路，才能保证指令间不互相干扰。但事实上，这个问题可以通过在虚线上（即每个阶段间）引入寄存器来实现，寄存器中寄存的相当于是前一条指令在该阶段运行出的结果（to hold information produced in previous cycle）

因此，当后一条指令需要用到前一条指令的结果时，只需要从对应的寄存器中抓取即可（因为寄存器是夹在两个周期之间的，所以它们的名字就是两个周期的名字，很直观）

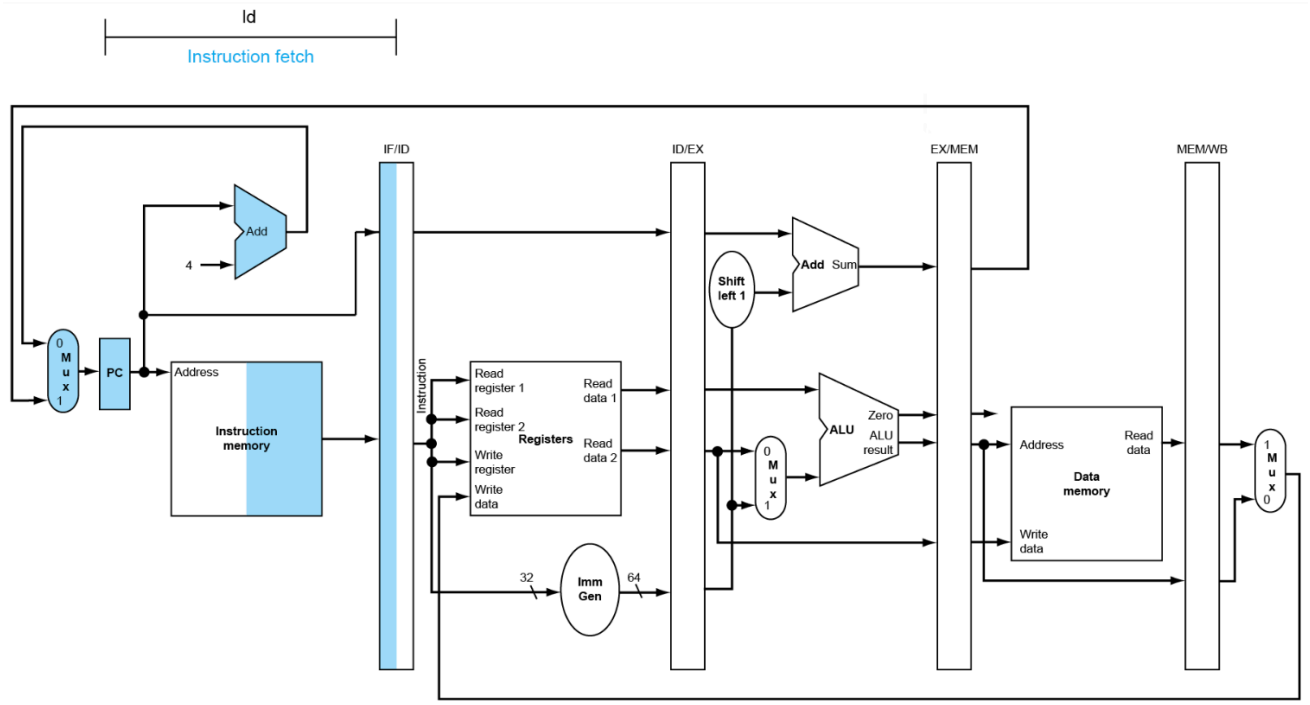


Pipeline Operation

我们以ld指令为例，一步步分析这些寄存器在指令执行中进行的操作。当寄存器左半部分被高亮时，代表其正在写入；同理，右半部分高亮则为读出

> 这一部分非常平白直观，不太清楚有何考点，但以防万一我还是写在这里了。所以图片也不给全了，不然篇幅太长，看上去过于混乱

1. **IF取指**：IF/ID寄存器写入

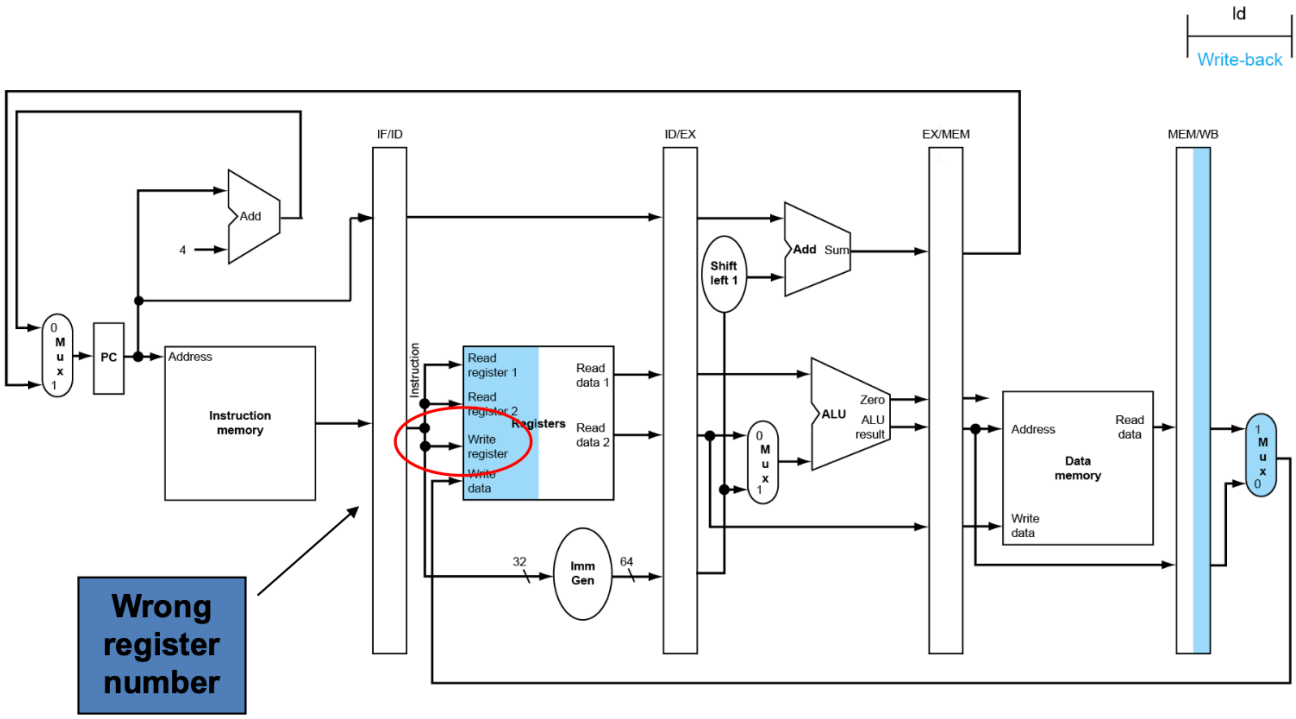


> 注意到，IF安排在后半个时钟周期，这主要是为了让流水线的运行更加顺畅，而并没有在明面上解决任何hazard（换句话说，这应该不是一个考点）

2. **ID解码**：IF/ID读出，ID/EX写入

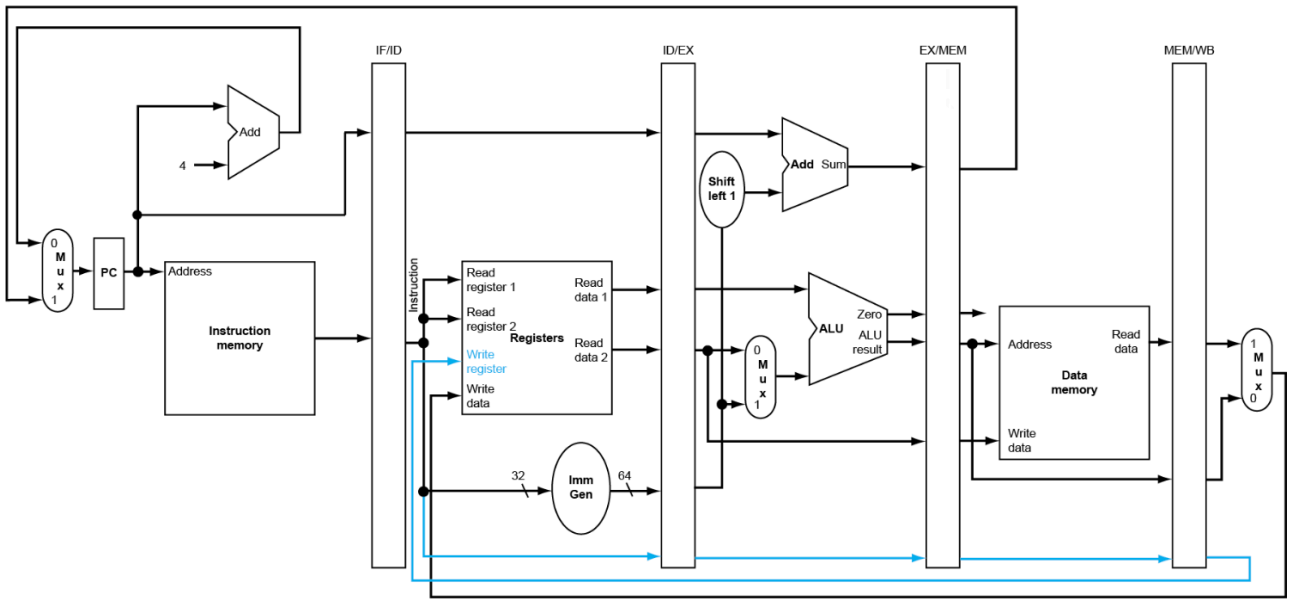
> 同样地，ID被安排在后半个时钟周期，这是为了解决Data Hazard而作的一个改进，防止WB和ID的冲突

3. **EX运算**：ID/EX读出，EX/MEM写入
4. **MEM内存访问**：EX/MEM读出，MEM/WB写入。MEM占用后半个时钟周期
5. **WB寄存器写入**：正常来说，如下图，运行结果将会从MEM/WB读出，给到Registers的Write data进行写入



但是我们仔细思考一个问题，此时的Write register还是原先那个吗？因为流水线是不间断运行的，当本条指令进行WB的时候，其后面的2条指令已经完成ID了，Write register被改变。因此，简单地采用上述数据通路会产生错误

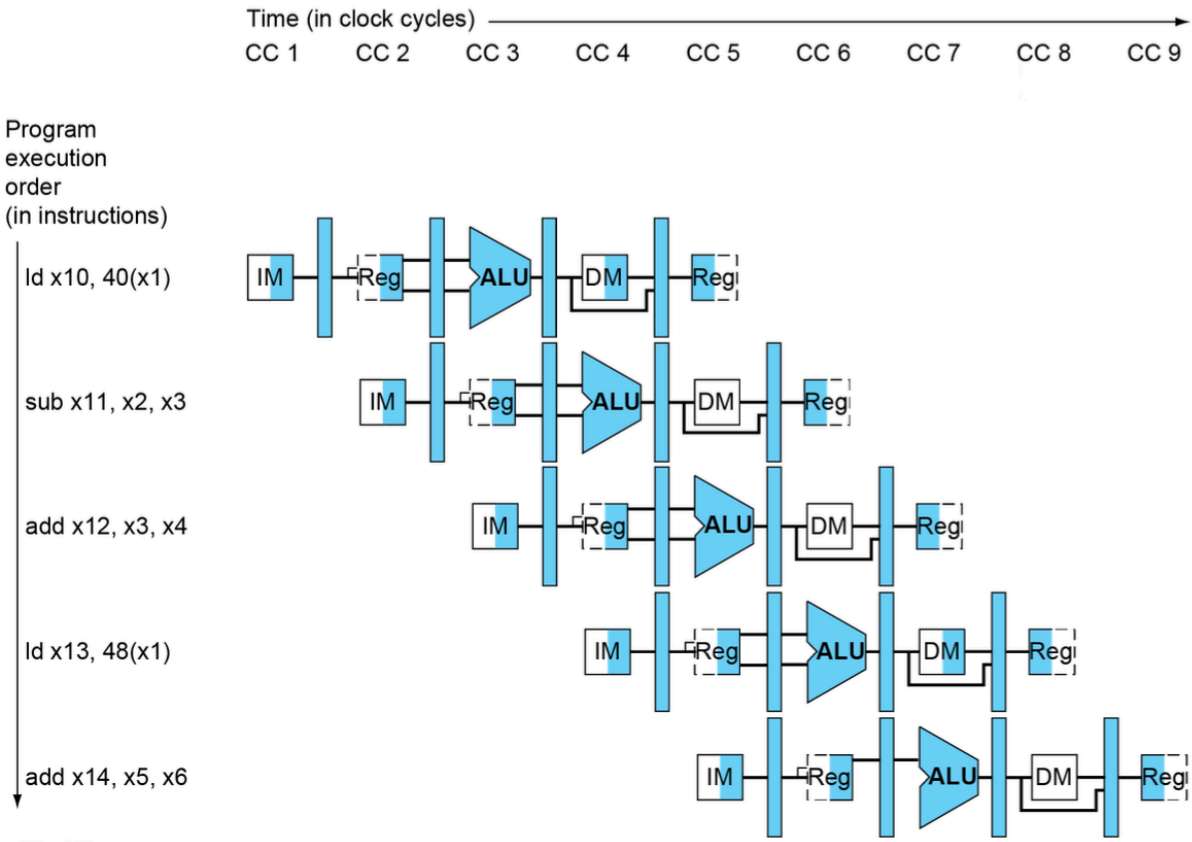
为了解决这个问题，我们引入一条额外的数据通路用于传递rd。这样，rd的传递和指令的运行同步进行，就不会产生上述冲突了



Multi-cycle Pipeline Diagram

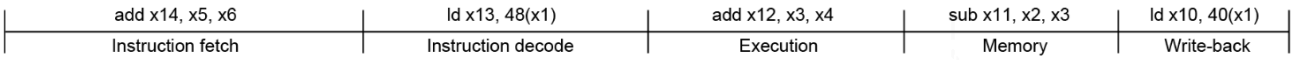
1. **Form Showing Resource Usage**

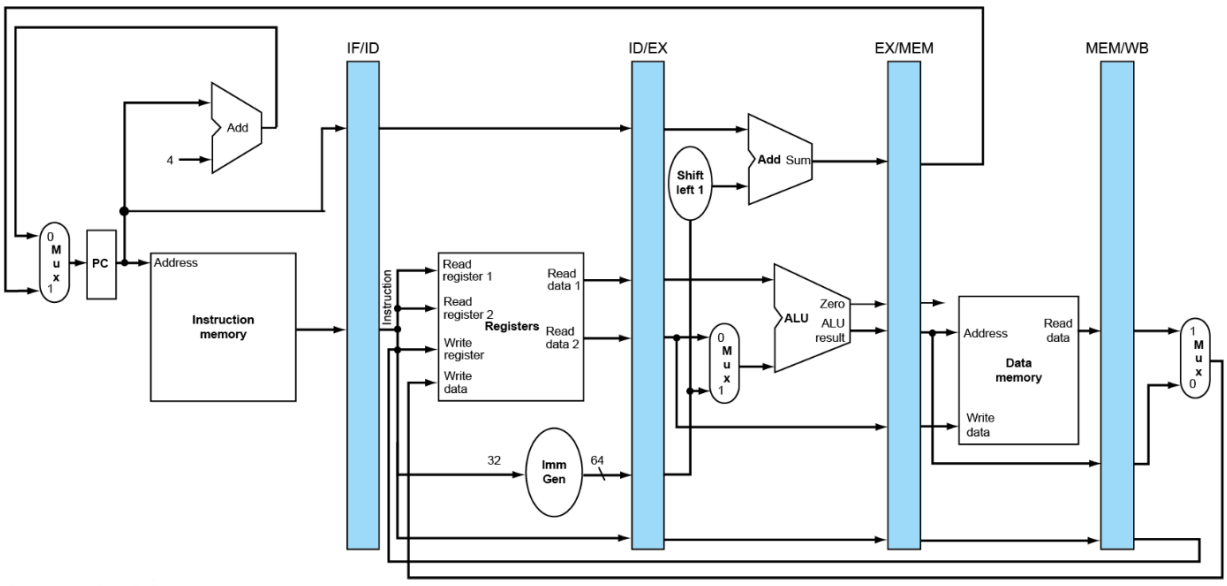
   下面这张流水线图显示了一串指令流水线实现中的资源占用



2. **State of Pipeline in a Given Cycle**

下面这是给定时钟周期的硬件占用图，相当于是上面这张图的在某时刻的截面

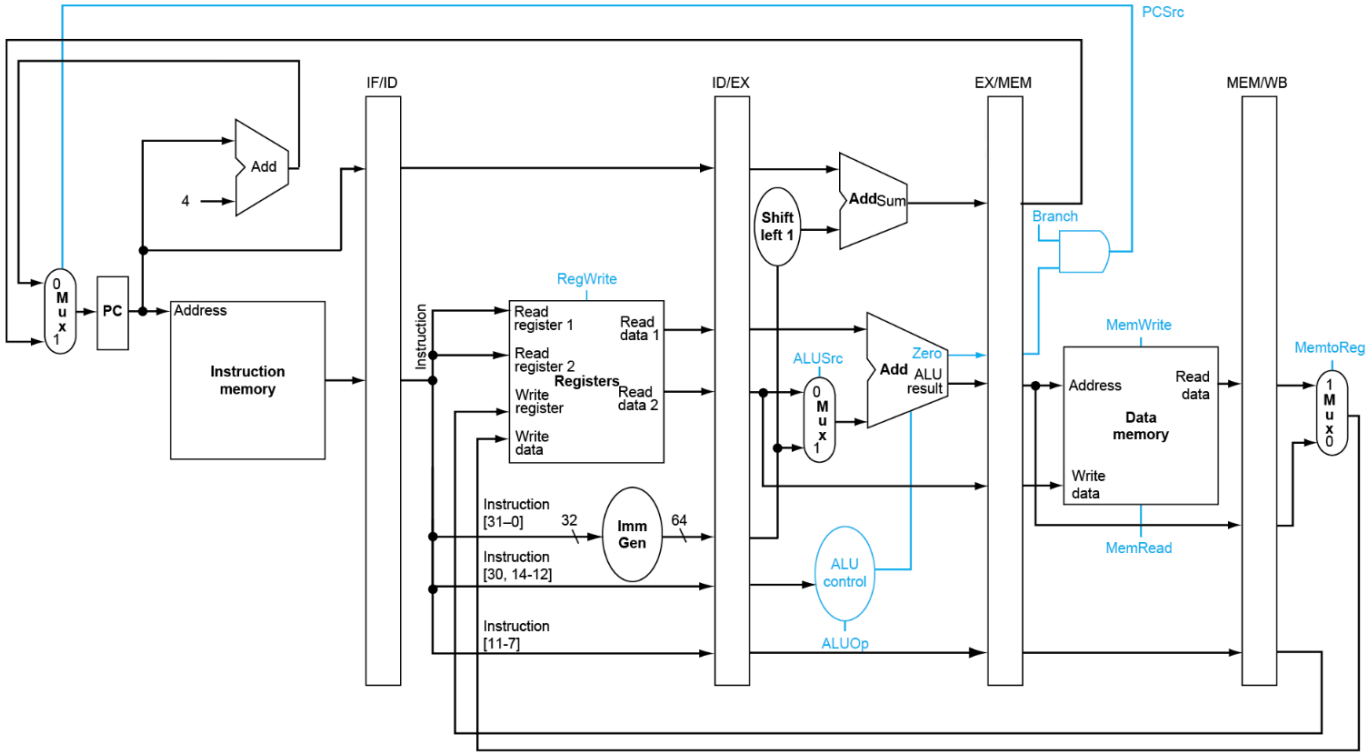




> 好吧我承认我没理解上面两张图有什么特别的，本质上没有新东西，非常直观

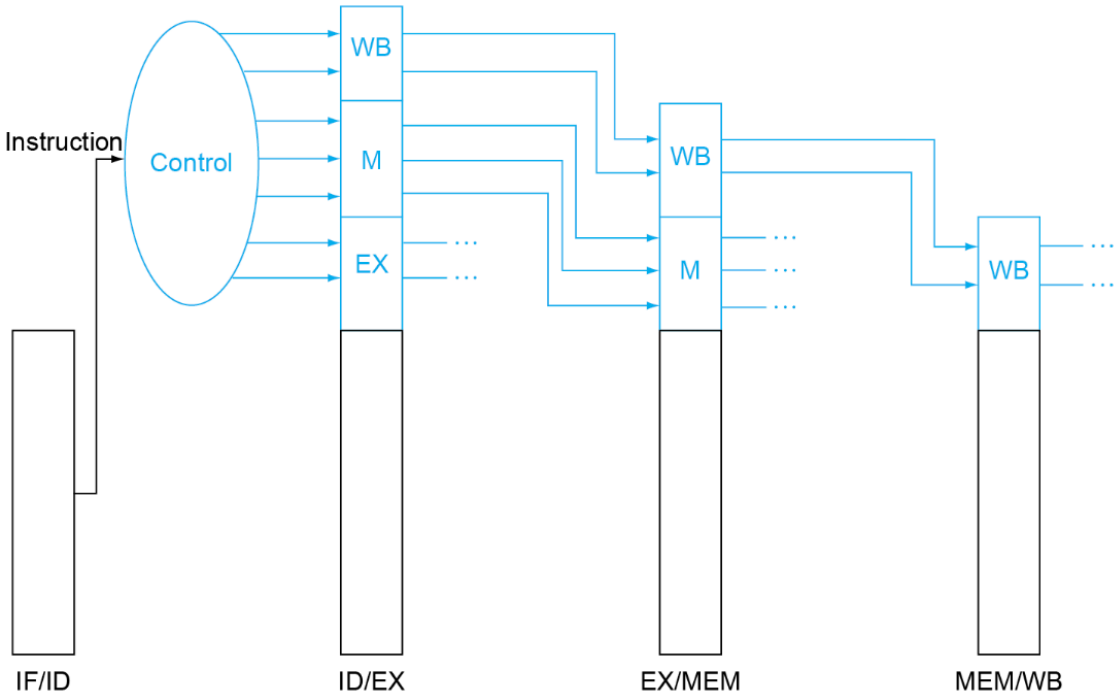
Pipelined Control 流水线控制通路

解决了Structure Hazard和Data Hazard，我们接下来考虑流水线中的控制通路。我们现将单周期的控制通路直接引入流水线：

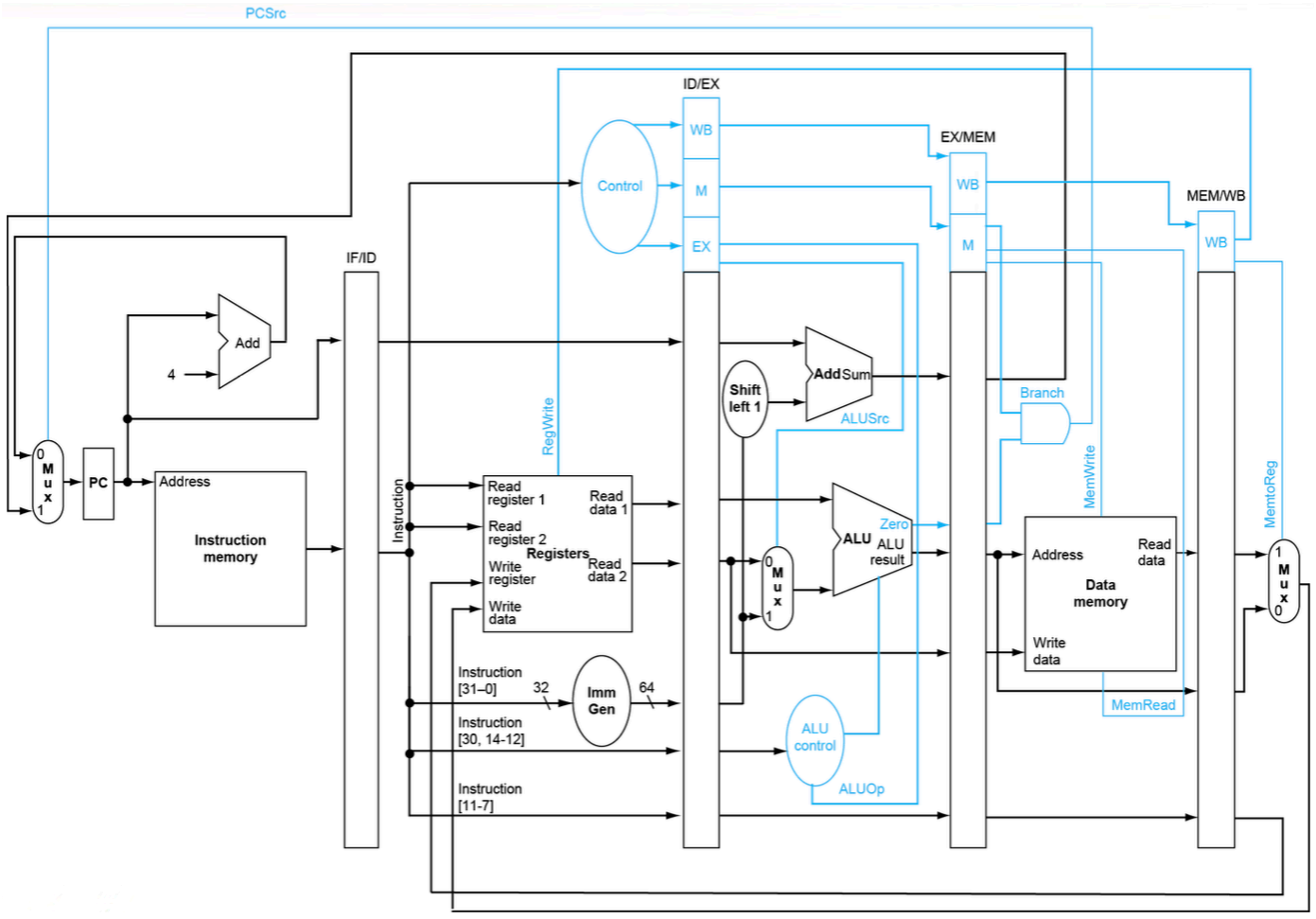


和数据通路一样，因为控制信号的产生和使用不在同一阶段，而当需要使用时当初产生的控制信号已经被覆盖，所以我们同样需要使用寄存器记录控制信号

我们将控制信号按照需要使用的阶段进行分类，并逐个通过寄存器接力到需要使用的位置。其中，EX阶段需要使用的有ALUop、ALUSrc，MEM阶段用到的有MemRead、MemWrite、Branch，WB阶段用到的有MemtoReg、RegWrite、Jump（图中未列出）



由此，我们可以画出完整的流水线处理器



Dependencies & Forwarding

寄存器采取XX/XX.RegisterXX的形式命名，最前面的部分代表寄存器的位置，最后面的部分代表寄存器存储的内容（Rs1、Rs2、Rd）

ID/EX.RegisterRs1: register number for Rs1 sitting in ID/EX pipeline register

数据冒险的判断

冲突发生条件

> 后1条或2条指令的rs用到本条指令的rd时，会发生数据冲突。条件判断均为rs在ID/EX寄存器中，在哪里发生冒险就从哪里forwarding

1. **Forward from EX/MEM Pipeline Reg**：后1条指令的rs用到本rd

   EX/MEM.RegisterRd = ID/EX.RegisterRs1

   EX/MEM.RegisterRd = ID/EX.RegisterRs2
2. **Forward from MEM/WB Pipeline Reg**：后2条指令的rs用到本rd

   MEM/WB.RegisterRd = ID/EX.RegisterRS1

   MEM/WB.RegisterRd = ID/EX.RegisterRs2

Detecting the Need to Forward

1. 如果rd不涉及寄存器写入，则不需要forwarding

   EX/MEM.RegWrite, MEM/WB.RegWrite
2. 如果目标寄存器是x0，则不管是否写入都是0，不需要forwarding

   EX/MEM.RegisterRd $\neq 0$, MEM/WB.RegisterRd $\neq 0$

Double Data Hazard

从上面可以看出，实际上一共有2种hazard，它们完全可能同时发生。在处理此类事件时，我们采用如下原则：

优先使用EX Hazard，即如果与前一条产生风险，不考虑与前面第二条的风险

> 这个原则很好理解的，以下面这段程序为例：

```
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
```

关注到第三条语句，它似乎可以与第一句发生MEM风险，与第二句发生EX风险，产生冲突。但如果我们只要按照第2句与第1句发生EX风险、第3句也与第2句发生EX风险进行forward即可。

所以说优先考虑EX风险，相当于大家都管好自己，就不会出问题

Forwarding Condition

从上面几点总结下来，需要forward的条件如下：

1. **EX Hazard**

```
if   (EX/MEM.RegWrite                 # 需要写入
     and (EX/MEM.RegisterRd != 0)   # 目标寄存器不是x0
     and (EX/MEM.RegisterRd == ID/EX.RegisterRs1)):
   ForwardA = 10

if   (EX/MEM.RegWrite
     and (EX/MEM.RegisterRd != 0)
     and (EX/MEM.RegisterRd == ID/EX.RegisterRs2)):
   ForwardB = 10 # A和B分别代表rs1和rs2
```

> Forward = 00：无需forward
>
> Forward = 10：从EX/MEM进行forward，即将之前ALU的运算结果forward过来，forwarded from the prior ALU result
>
> Forward = 01：从MEM/WB进行forward，即forwarded from data memory or an earlier ALU result

2. **MEM Hazard**

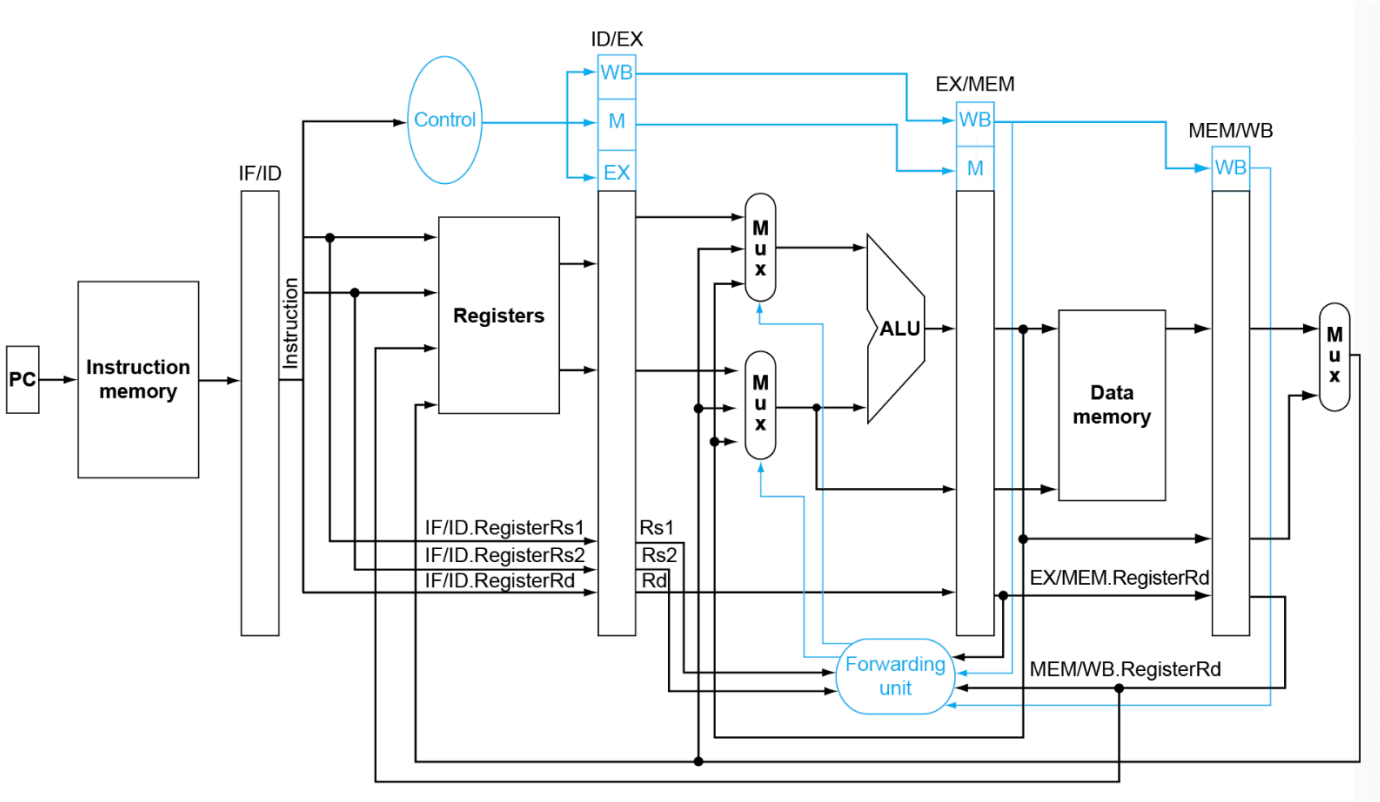
```
if   (MEM/WB.RegWrite
     and (MEM/WB.RegisterRd != 0)
     and not(EX/MEM.RegWrite
             and (EX/MEM.RegisterRd != 0)
             and (EX/MEM.RegisterRd == ID/EX.RegisterRs1)) #
这是判断EX Hazard的条件
     and (MEM/WB.RegisterRd == ID/EX.RegisterRs1)):
```

```
  ForwardA = 01

if  (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != 0)
    and not(EX/MEM.RegWrite
            and (EX/MEM.RegisterRd != 0)
            and (EX/MEM.RegisterRd == ID/EX.RegisterRs2))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs2)):
  ForwardB = 01
```

Datapath with Forwarding

由上面的forwarding思路，我们可以给出如下前递通路。其实就是将EX/MEM和MEM/WB中存储的rd、ID/EX中的rd拿出来，给到Forwarding unit，用WB信号进行控制（蓝色部分）



Load-Use Hazard Detection

因为涉及到load的冲突需要插入一行bubble（stall）才能解决（因为需要等到上一步的MEM阶段结束才能得到操作数，开始下一阶段的EX），因此需要进行特殊的处理

因为越早发现成本越小，因为stall后下条指令需要重做，所以在下一条指令的IF阶段即判断

Load-Use冒险的发生与处理

判断

1. **判断是否存在load**：ID/EX.MemRead（本条指令涉及从Mem中读取）
2. **本条指令的rd为下一条指令的rs**：ID/EX.RegisterRd = IF/ID.RegisterRs1/2

处理：Stall

发现存在load-use hazard后，应当立马进行一个周期的stall，让前一条指令多领先一个周期。
1-cycle stall allows MEM to read data for ld, forward from WB to EX stage

1. **Force control values in ID/EX register to 0**：EX, MEM and WB do nop (no-operation)，控制数据全为0

   > 这样看似会导致所有指令都停止，但实际上本条指令还是正常运行的，下条指令被stall。因为监测到冲突后要后一个周期才能进行改变，而此时本条指令已经运行到EX/MEM阶段了，不会受影响

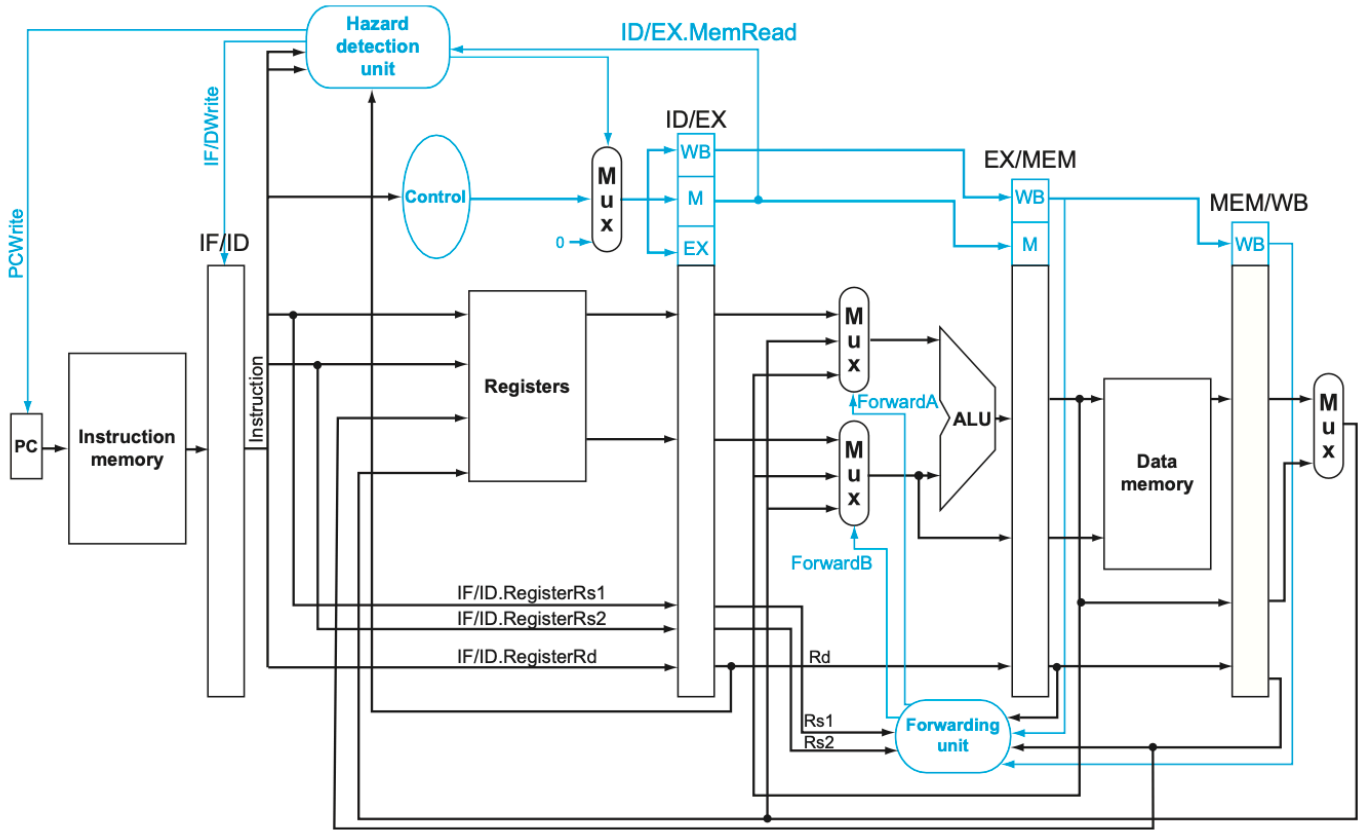
2. **Prevent update of PC and IF/ID register**：本条指令之后的指令都重新进行一遍当前步骤

   Using instruction is decoded again

   Following instruction is fetched again

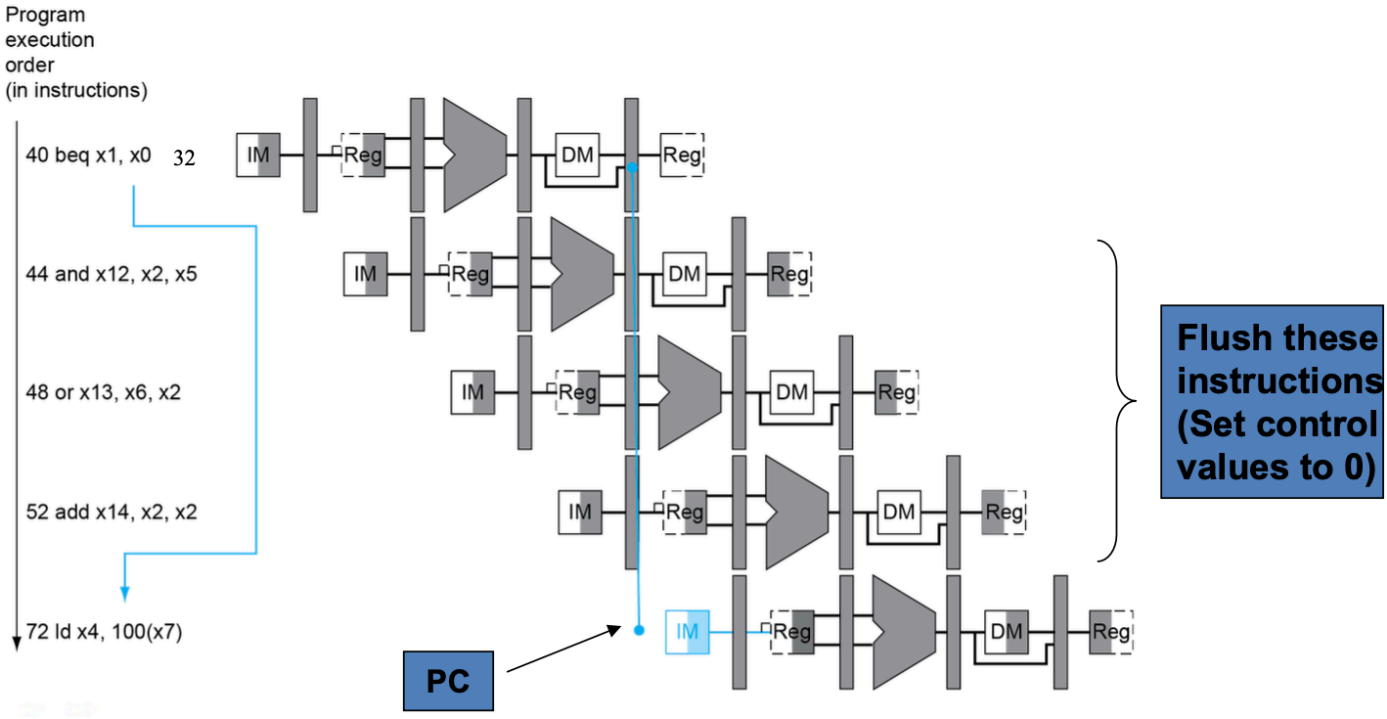
Datapath with Hazard Detection

现在我们可以给出含load检测的数据通路了。将IF/ID、ID/EX和ID/EX.MemRead给到Hazard Detection Unit中，作为控制信号（从上面的判断条件中就能看出），输出给到PCWrite、IF/IDWrite和ID/EX，分别负责Prevent update of PC and IF/ID register和Force control values in ID/EX register to 0



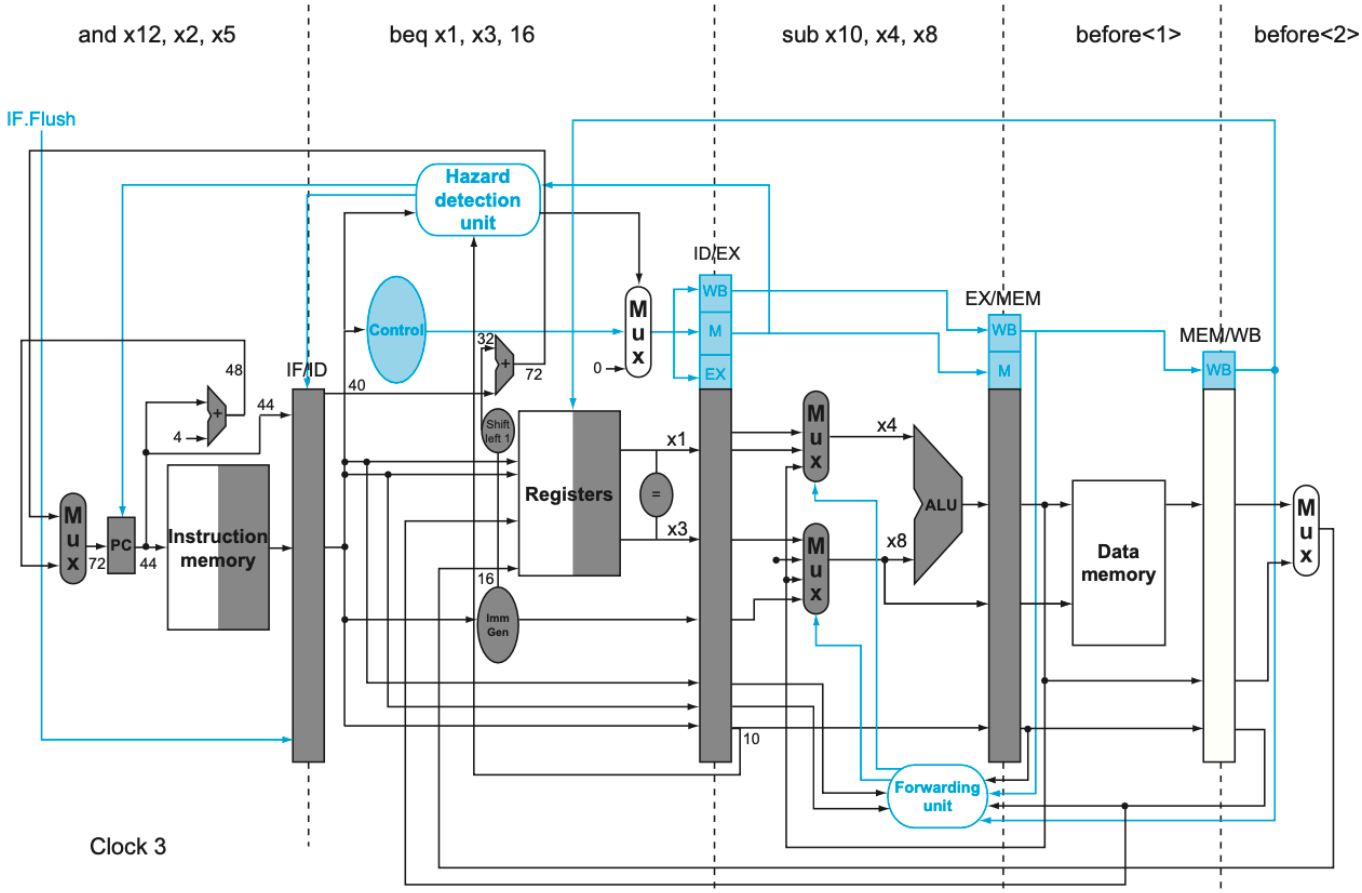
Branch Hazards

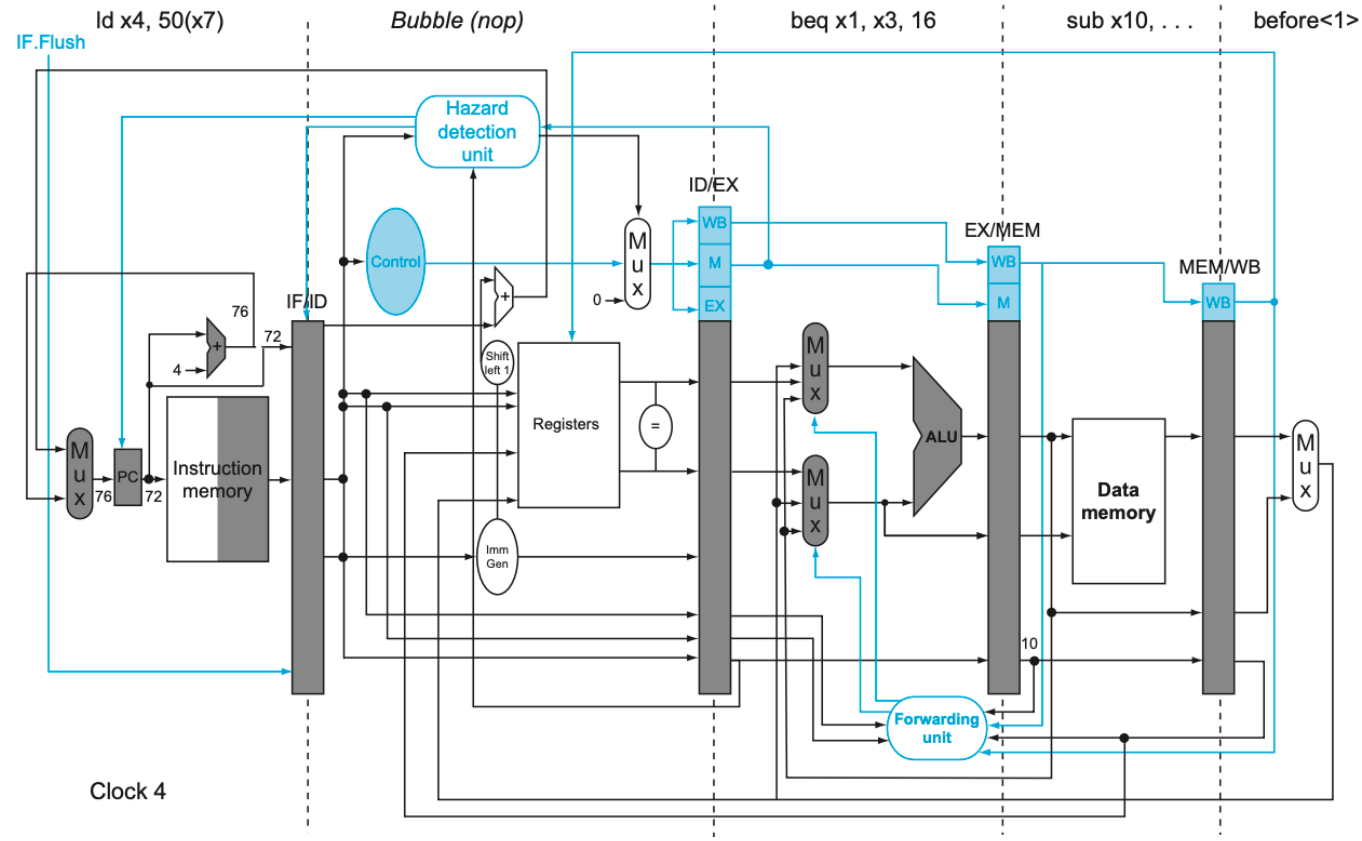
如下图，如果遇到错误预判的branch语句，需要将后面的几条指令全部冲销掉，并跳转到指定的位置



Reducing Branch Delay

但是像上面这样操作，flush一次会损耗3个时钟周期，开销过大。为了减少branch指令的代价，我们采用额外的硬件将判断提前至ID阶段。这样就只需要flush一条指令、stall一个周期了。为了实现这一点，我们需要添加target address adder和register comparator





Dynamic Branch Prediction

一种实现方案是分支预测缓存（branch prediction buffer / branch history table），这是一个由branch指令地址定位（indexed by recent branch instruction address）的存储器，用于存储此分支上次的运行结果。每次执行一条新的分支指令时，都先查询table，根据上次的结果运行

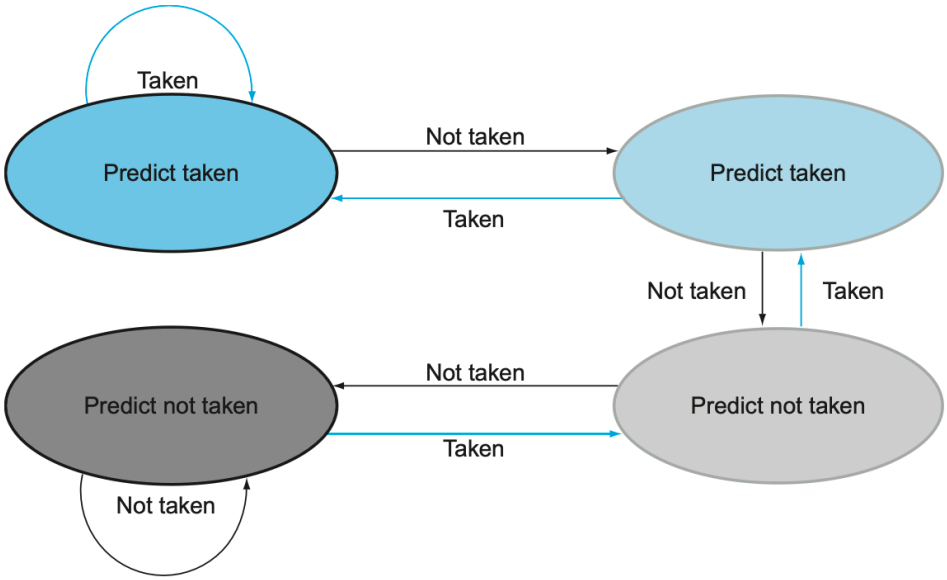
1. **1-Bit Predictor**：如果是1位的buffer，因为只能记录2种状态（taken / not taken），会导致错误2次

   我们考虑下面这个二重循环：

   ```
   for (int i = 1; i <= 9; i++)
     for (int j = 1; j <= 9; j++)
       printf ("%d", i * j);
   ```

   当j到达10时，prediction为taken（因为j = 9时taken），但实际为not taken，错判1次；随后外层循环+1，内层j = 1，应当taken，但因为刚才是not taken，再次错误

2. **2-Bit Predictor**：Only change prediction on 2 successive mispredictions，可以避免上面的问题



Calculating the Branch Target

只有predictor还不够，branch的目标地址也需要计算并暂存。因此我们需要一个branch target buffer（indexed by PC when instruction fetched），保证如果预测为taken，立即从中抓取目标地址

Exceptions and Interrupts

定义

> 意外和中断可以由多种原因导致，可能来自CPU内部，也可能来自外部。在本书中其实并不严格区分例外与中断，只需要记住来自外部I/O的干扰属于interrupt即可

- **Exception**：Arises within CPU, e.g., undefined opcode, syscall, 系统重启...
- **Interrupt**：From an external I/O controller

Handling Exceptions

1. **Save PC of offending (or interrupted) instruction**：在Supervisor Exception Program Counter（SEPC，64bits）中记录发生例外的指令地址，将控制权移交操作系统
2. **Save indication of the problem**：Supervisor Exception Cause Register（SCAUSE，64bits，大多数位不使用）中记录例外发生的原因
3. **Jump to handler**：跳转处理特殊情况

Vectored Interrupt 向量式中断

普通的方法中，如果发生例外，需要跳转到一个统一的地址，由操作系统判断例外发生的原因并跳转处理。

向量式中断采用一个vector table base register记录一个基地址，并将例外原因作为偏移量。发生例外时，根据例外发生的原因，将对应的exception vector address加到vector table base register上，跳转到handler address处理

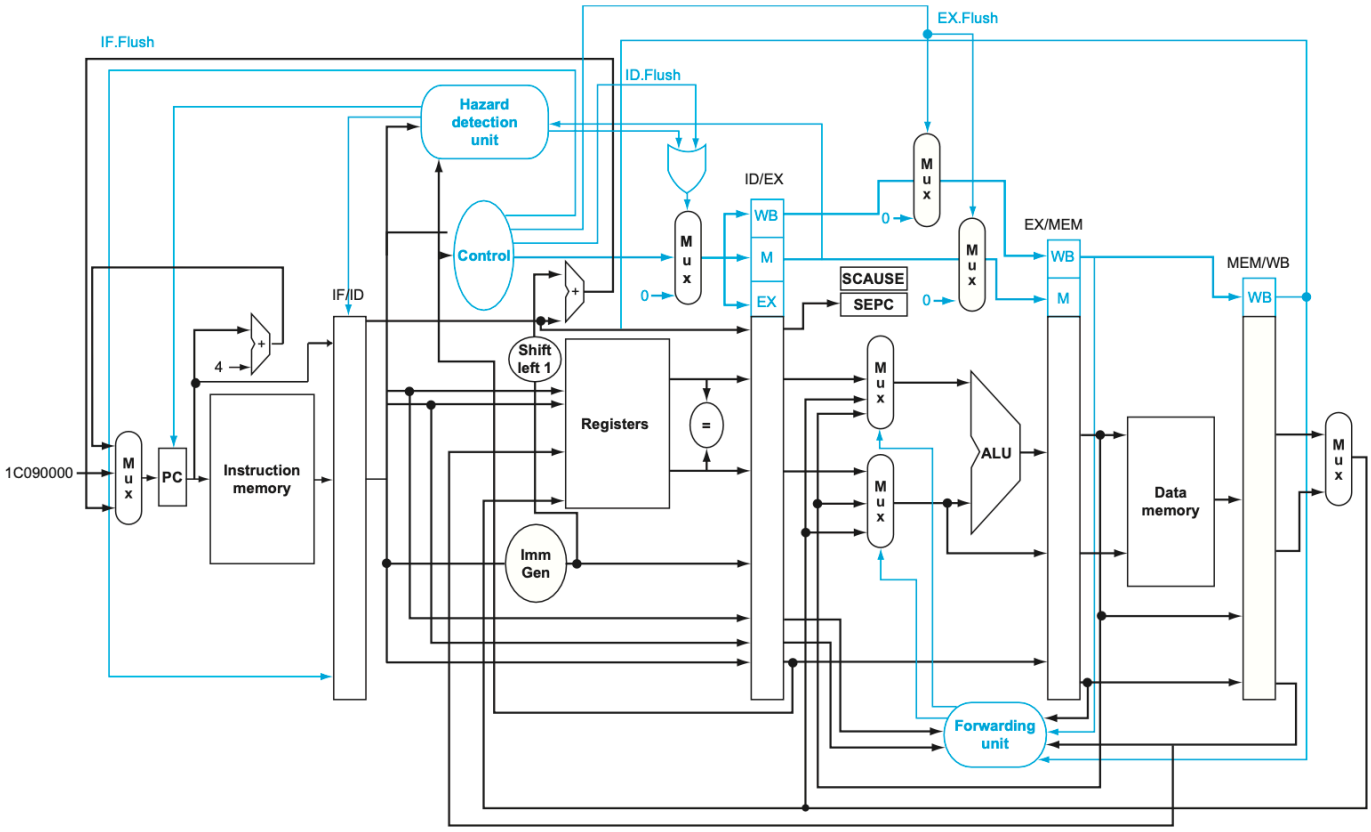
Handler Actions

1. **Read Cause**, transfer to relevant handler 根据中断原因跳转到对应的处理区域
2. **Determine action required**
3. **If restartable**：take corrective action and use SEPC to return to program
4. **Otherwise**：terminate program and report error using SEPC, SCAUSE...

Exception in a Pipeline

实际上这是control hazard的另一种形式。在流水线中发生例外时，我们需要完成之前的指令，并清除掉所有之后的指令。为了实现这一点，需要加入额外的控制信号：

- PC输入的MUX中添加一个$1C090000_{16}$（这是例外的入口地址，发生例外需要跳转执行）
- ID.Flush与Stall取或，给到ID/EX寄存器，用于冲销后面的指令
- EX.Flush给到EX/MEM寄存器，用于冲销本条指令



Example

> 因为这个过程可能稍微有些复杂，所以还是给出一个例子方便理解

假设我们正在执行下面的指令：

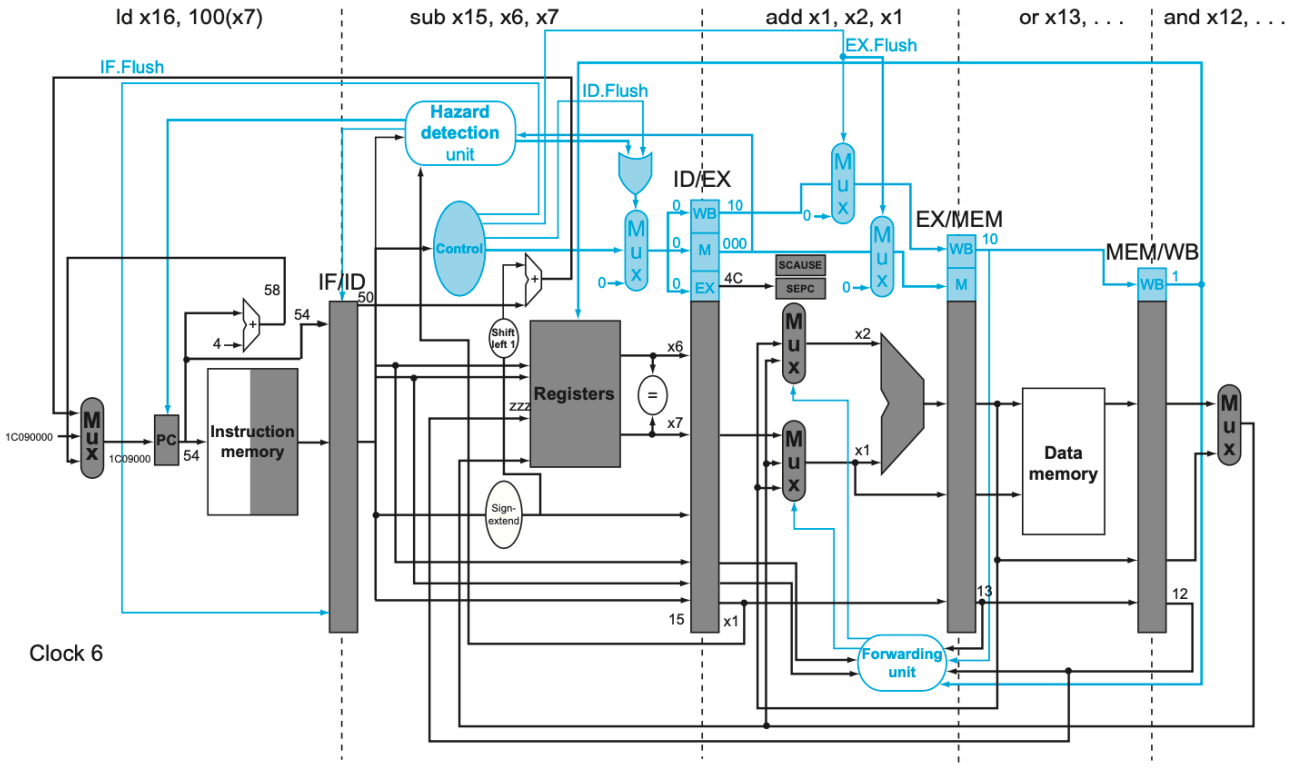
```
40  sub   x11,  x2,   x4
44  and   x12,  x2,   x5
48  or    x1,   x2,   x1
4C  add   x1,   x2,   x1
50  sub   x15,  x2,   x1
54  ld    x16,  100(x7)
```

Assume the instructions to be invoked on an exception begin like this 假设例外程序入口为：

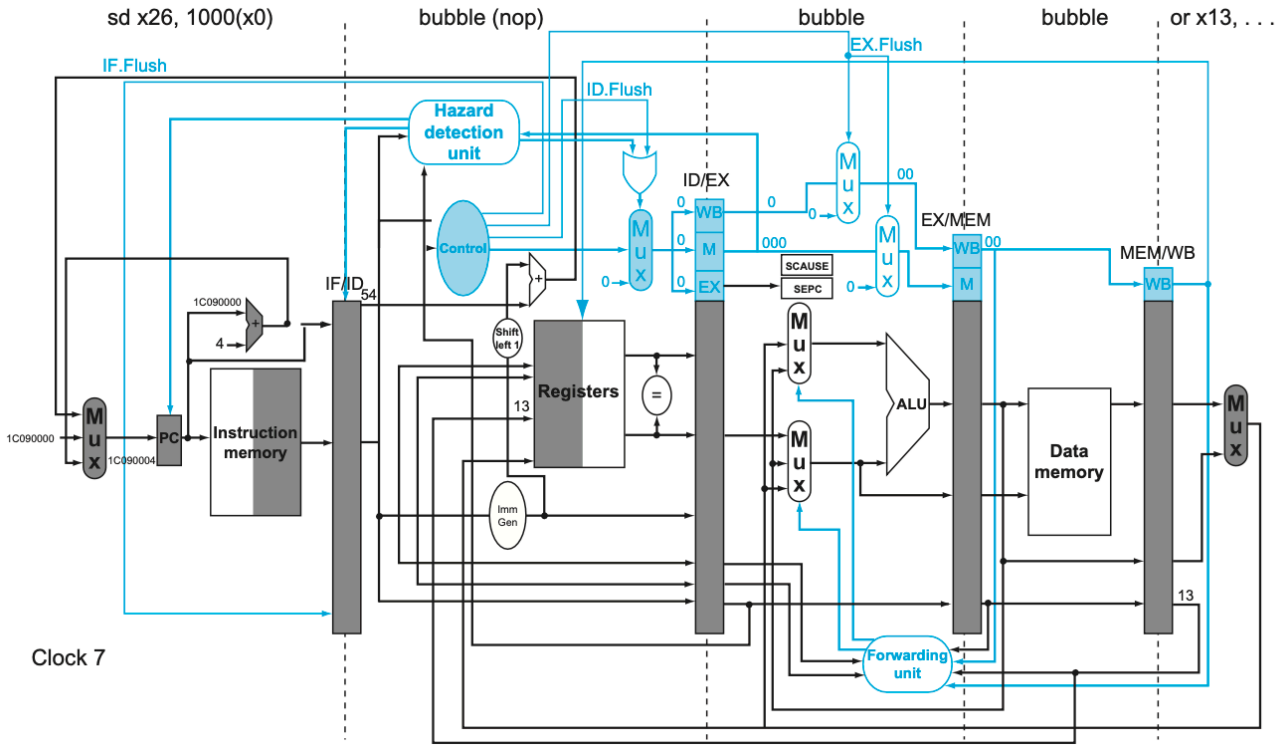
```
1C09 0000    sd     x26,  1000(x10)
1C09 0004    sd     x27,  1008(x10)
```

假设在add阶段发生例外，则我们的硬件情况如下：

1. add运行到EX阶段时，发生例外，需要进行冲销。ID.Flush和EX.Flush将ID/EX和EX/MEM寄存器中的数据冲销；1C09 0000给到PC，实现跳转执行



2. 向后一个周期，add及其之后的2条指令均被flush，产生3个bubble



Multiple Exceptions

当出现多条例外同时发生时，SCAUSE中记录当前优先级最高的例外信息。如果意外指令重叠，可以同时处理多个意外

一种最为简单的实现就是，出现一条例外处理一条，处理完再返回并重新执行原语句。这是基于precise exception特性的，即SEPC中保存的是发生例外指令的地址，而非发生例外时PC地址（因为发生例外时，指令已经执行到EX阶段，此时PC已经+8了，如果是imprecise exception，则SEPC记录的是+8后的地址，无法返回到发生例外的地址）

Instruction-Level Parallelism 指令间并行性（ILP）

提升ILP的方法

流水线就是指令间并行的体现。想要提升ILP，有如下方法：

1. **Deeper Pipeline**：Less work per stage => Shorter clock cycle，更多操作可以重叠执行
2. **Multiple Issue**：增加流水线内部功能部件数量，每周期可以发出多条指令
    - Start multiple instructions per clock cycle

- Replicate pipeline stage => Multiple pipelines，因为每周期同时发出多条指令，相当于是有多条指令重叠执行，达到多条流水线并行处理的效果
- CPI < 1 => Use Instruction Per Cycle (IPC)

  > 比如，有一个4GHz 4-way multiple-issue的流水线处理器（每秒4G个时钟周期，每个时钟周期最多发射4条指令），则有：
  >
  > 16BIPS，peak CPI = 0.25，peak IPC = 4

- 缺点：Dependencies reduces

3. **Speculation 猜测**：Speculation也是一种提升ILP的重要方法，和prediction思路一致，猜测一条指令是否可以执行（Start operation as soon as possible）。如果猜错，返回重做

   The processor usually buffers the speculative results until it knows they are no longer speculative，猜测的结果通常存入缓冲区，确认后再写入

Multiple Issue

操作：想要保证multiple issue的正确执行，有2个必须的操作

1. **Packaging instructions into issue slots**：将同时发射的指令打包到一个指令槽中
2. **Dealing with data and control hazards**：处理冒险

分类：根据指令发射与否的判断方式，多发射可以分为如下2种形式：

1. **Static Multiple Issue**：由编译器完成指令发射的判断
2. **Dynamic Multiple Issue**：执行过程中由硬件动态实现，编译器可以帮助重排指令以减少冒险

Static Multiple Issue

编译器负责去除冒险，将指令重排并打包入issue packets。同一个package中的指令是没有dependency的，可以视为一条事先定义的大语句，这也就是Static Multiple Issue的原名：Very Long Instruction Word（VLIW）

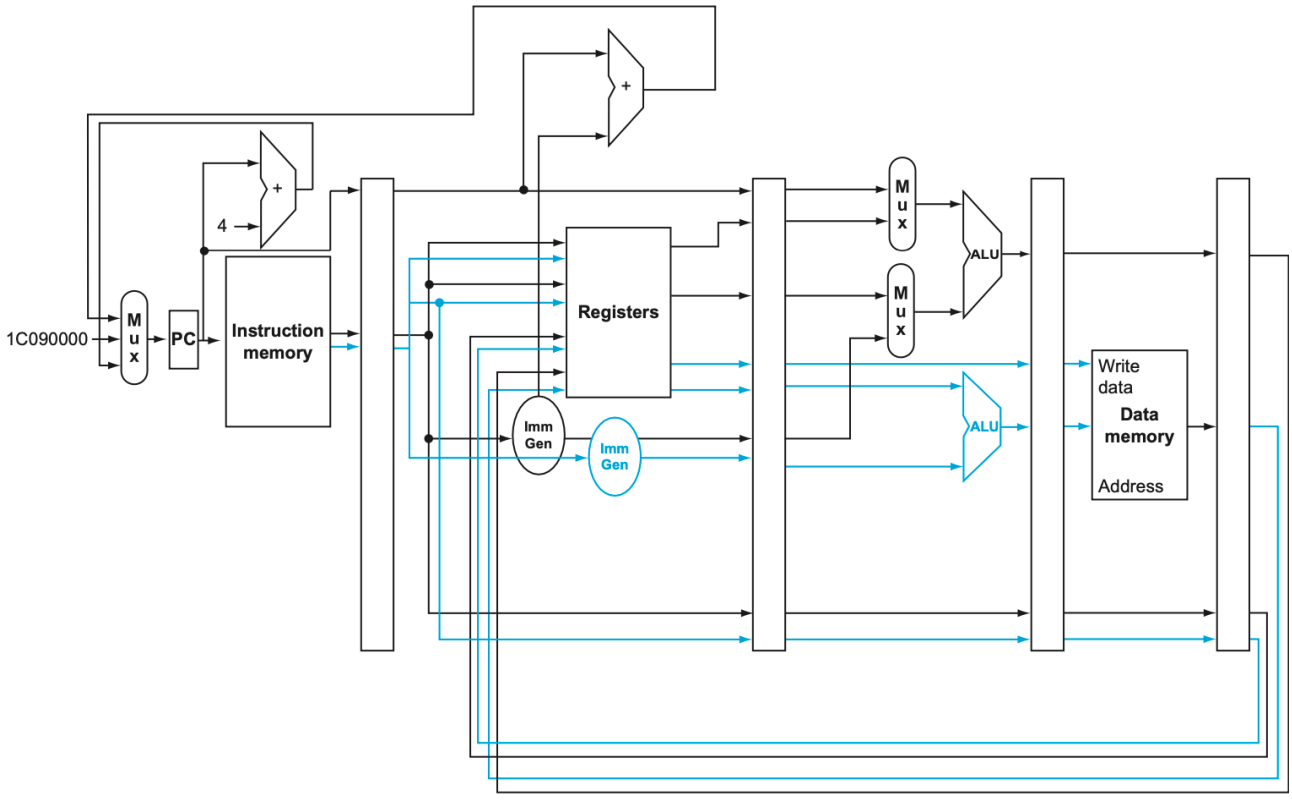
Example：RISC-V with Static Dual Issue

1. **打包**：将2条指令打包，其中一条为ALU opration或branch，另一条为load或store；因此它是64-bit aligned。如果有一条指令无法使用，就用nop填充（pad an unused instruction with nop）

   于是，我们可以给出如下的流水线示意图：

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

2. **Datapath**：因为每一时间每个阶段都有2条指令在执行，因此需要双倍的数据通路和双倍的计算硬件（蓝色部分实际上组成了另一个独立的和黑色部分相同的数据通路）
   - 寄存器翻倍，因为数据通路上的数据量翻倍
   - ALU翻倍，否则无法同时执行2个独立的计算（Imm Gen同理）
   - 如果存在hazard，同样是一个时钟周期的停顿，称为use latency（当然这里一停就是停2条指令了）



3. **Scheduling**：这里给出一个调度的例子

```
Loop: ld    x31, 0(x20)
      add   x31, x31, x21
      sd    x31, 0(x20)
      addi  x20, x20, -8
      blt   x22, x20, Loop
```

	ALU/branch	load/store	cycle
Loop:	nop	ld x31, 0(x20)	1
	addi x20, x20, -8	nop	2
	add x31, x31, x21	nop	3
	blt x22, x20, Loop	sd x31, 0(x20)	4

$$\text{IPC} = 5/4 = 1.25 \tag{24}$$

4. **Loop Unrolling**：将循环展开多次，从中进一步探索优化空间。假设我们将上面的循环展开4次：

	ALU/branch	load/store	cycle
Loop	addi x20, x20, -32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28, x28, x21	ld x30, 16(x20)	3
	add x29, x29, x21	ld x31, 8(x20)	4
	add x30, x30, x21	sd x28, 32(x20)	5
	add x31, x31, x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22, x20, Loop	sd x31, 8(x20)	8

$$\text{IPC} = 14/8 = 1.75 \tag{25}$$

Dynamic Multiple Issue & Scheduling

动态多发射处理器也称为superscalar processor，由CPU决定指令是否发射。虽然理论上全都由CPU负责，但实际情况中还是需要编译器的支持

动态多发射处理器也需要重排以减少stall，此过程需要如下硬件的支持：

- **Instruction Fetch and Issue Unit**：取指、译码、将各指令发送到相应的functional unit上执行
- **Multiple Functional Units**：每个functional unit上都有buffers，称为reservation stations，用于保存指令的操作和所需要的操作数
- **Commit Unit**：提交单元中保存了已完成指令的执行结果，在真正需要提交时更新寄存器或内存。此中的buffer称为reorder buffer

Register Renaming：Reservation stations和reorder buffers提供了register renaming的方式

1. 发射指令时，指令会被拷贝到相应的reservation station中。如果操作数在寄存器堆reorder buffer中，则拷贝入reservation station，等所有操作数都准备好，并且其他功能部件可用，就可以发射。此时寄存器中的副本无需保存，如果出现寄存器的写操作，其可以被覆盖（overwritten）
2. 如果操作数不在寄存器堆或reorder buffer中，说明其正在等待某个functional unit的计算结果，等其计算完毕将会直接被拷贝入reservation station

从上面的过程可以看出，指令是乱序执行的，称为out-of-order execution。取指和译码都需要按序执行，提交时也按照指令顺序写入执行结果，这种保守的模式称为in-order commit

Power Efficiency

- 动态重排和猜测都非常消耗能量 Complexity of dynamic scheduling and speculations requires power
- 多核设计可以有效减小能耗和流水线级数 Note the drop in pipeline stages and power as companies switch to multicore designs

实例

ARM Cortex-A53

- ARMv8 instruction set
- Static in-order pipeline
- 为了实现floating-point和SIMD操作，增加2个stage

Intel Core i7 920

- Dynamic pipeline scheduling, out-of-order execution and speculation
- Misprediction cause a penalty of ≈ 15 cycles

> 剩下的内容比较散，等题目做到再记吧。不然没用的内容太多了