An Efficient Algorithm for Exploiting Multiple Arithmetic Units

Abstract:

本文描述了在System/360模型91的浮点区域中利用多个执行单元存在的方法。

技术基础为通用数据总线和寄存器标记方案，它允许同时执行独立的指令，并保留指令流中固有的基本面。公共数据总线能有效地利用执行单元而不需要特别优化代码从而来提高性能。相反，硬件通过"提前查看"大约8条指令，可以自动在本地优化程序执行。

这些技术的应用并不局限于浮点运算或System/360架构，它们可以用在几乎任何计算机有多个执行单元和一个或多个累加器。两个执行单元，以及相关的存储缓冲区、多个累加器和输入/输出总线，都被广泛核对。

1 Introduction:

本文的主题是在IBM System/360型号91中实现浮点指令并行执行的方法。

实现两个单元的并行操作不再是简单地将每条指令分类为定点或浮点的问题，这种分类是独立于以前的指令的。相反，这是一个确定每条指令与之前所有未完成指令的关系的问题。简单地说，目标必须是保持必要的先例，同时允许独立行动尽可能地重叠。

这个目标在模型91中通过一个称为公共数据总线(CDB)的方案实现。CDB所需的硬件体积小，逻辑简单。可以与任意数量的累加器和任意数量的执行单元一起工作。简而言之，它为自动、高效地开发多个执行单元提供了一种硬件算法。

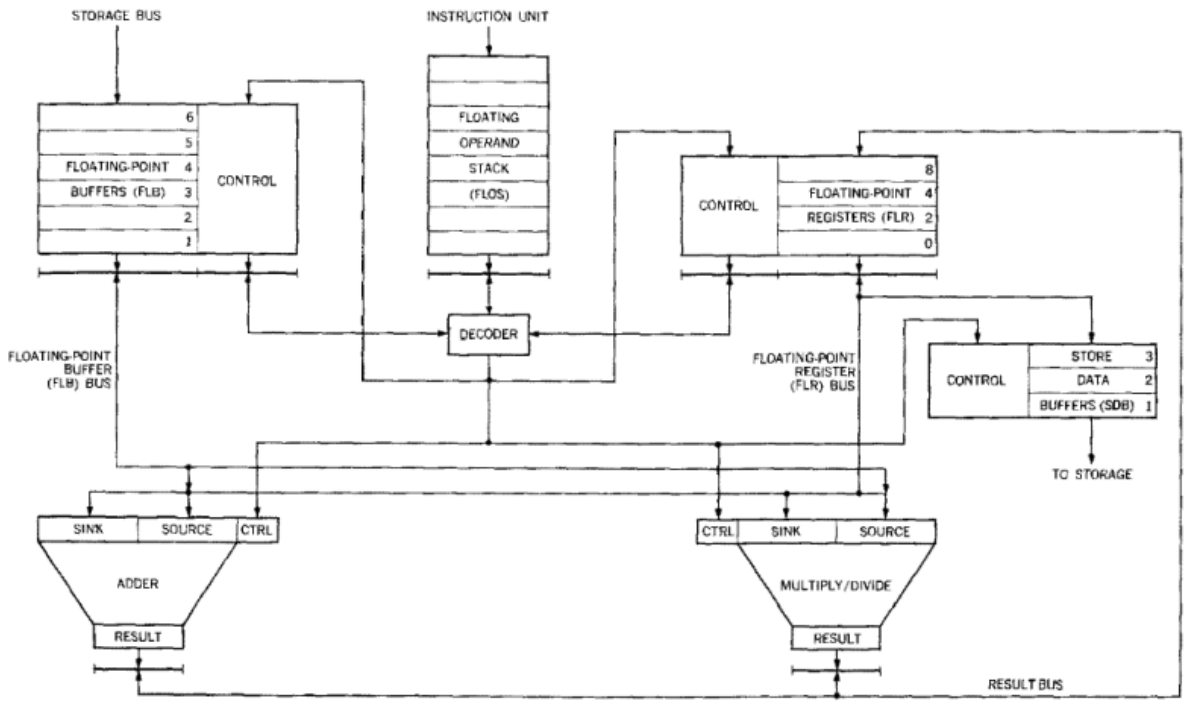


Figure 1 Data registers and transfer paths without CDB.

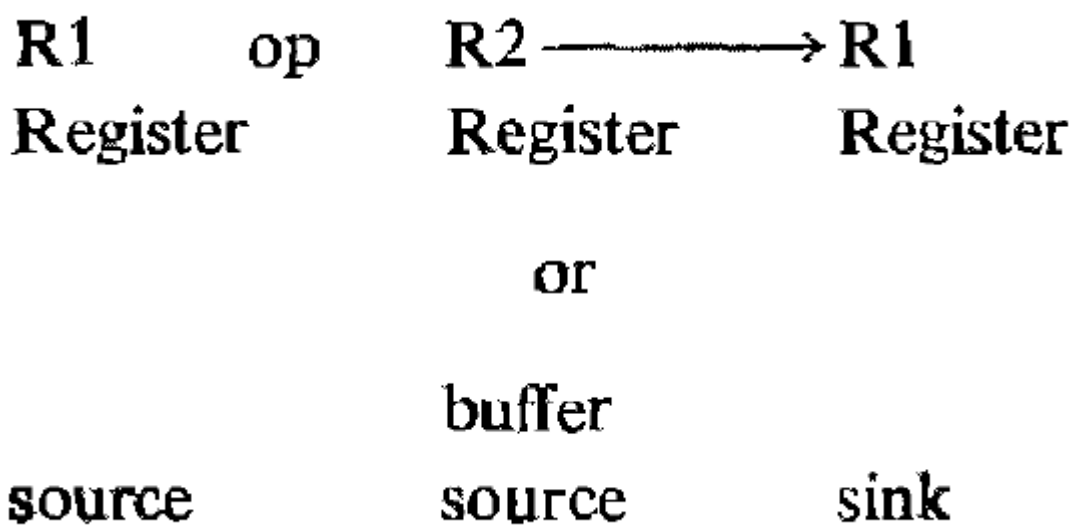
本文的下一节将讨论寄存器、数据路径和执行电路的物理框架，这是由前一篇论文中提出的架构和总体CPU结构所隐含的。在这个框架内，人们可以随后讨论优先级问题、一些可能的解决方案以及选定的解决方案——国开行。最后将对所得到的结果进行总结。

2 Definitions and data paths:

在为浮点运算栈(FLOS)准备指令时，指令单元将“存储-寄存器”和“寄存器-寄存器”的指令映射成“伪寄存器-寄存器”的格式。在这种格式中，R1总是由体系结构定义的四个浮点寄存器(FLR)之一。它通常是指令的接收器，也就是说，它是FLR，它的内容被设置为等于运输的结果。存储运算是唯一的例外，其中R1指定了要放在存储中的运算对象的来源。存储中的字实际上是存储的汇集。(R1和R2指的是System/360架构定义的字段。)

在FLOS“看到”的“伪寄存器-寄存器”的格式中，R2字段可以有三种不同的含义。它可以像在普通的“寄存器-寄存器”指令中一样处理FLR。如果程序包含一个“存储-寄存器”的指令，R2字段会指定指令单元分配的浮点缓冲区(FLB)来接收存储运算对象。最后，R2可以指定指令单元分配的存储数据缓冲区(SDB)来存储指令。在前两种情况下，R2是一个运算对象的来源;在最后一种情况下，它是一个接收器。因此，指令单元将所有的存储映射到6个浮点缓冲区(FLB)和3个存储数据缓冲区(SDB)中，这样FLOS只能看到“伪寄存器-寄存器”的运算。

在讨论优先级时，来源和汇聚之间的区别将变得非常重要，应该牢牢记住。所有的说明(除了存储和比较)都有以下形式:



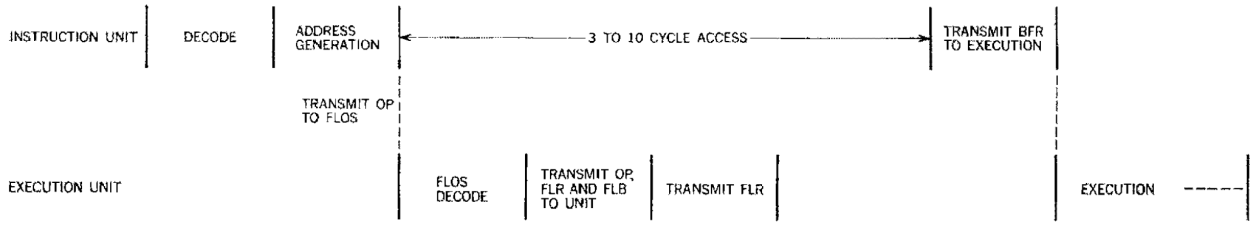


Figure 2 Timing relationship between instruction unit and FLOS decode for the processing of one instruction.

例如，指令AD0, 2的意思是“将寄存器0和寄存器2的双精度的加总放在寄存器0中”，即R0+R2→R0。注意，R1实际上既是来源又是接收，然而，在以后的讨论中，它将被称为汇集，R2为来源。

运算的定义和机器组织结合在一起意味着一组数据寄存器，这些寄存器之间有传输路径。如图1所示。主要的寄存器(FLR's, FLB's, FLOS and SDB's)已经在上面和前面的一篇论文中讨论过了。另外两个寄存器，一个接收器和一个来源，显示了每个执行电路。最初，这些寄存器被认为是执行电路所需要的内部工作寄存器，并以如下所述的方式进行多重使用。后来，它们的功能被开发到预约站的概念下，与它们的"工作"功能分离。

在实际设计一台机器时，数据路径会随着设计的进展而推进。不过，这里将展示一个完整的first-pass数据路径，以方便讨论。为了说明这个运算，让我们依次考虑四种指令——从"从存储器加载寄存器"、"存储到寄存器的运算"、"寄存器到寄存器的运算"和"存储"。让我们先看看在真空中如何实现这些情况；然后，当每一个都嵌入到一个项目的上下文中时，困难就出现了。为了简单起见，将在整个过程中使用双精度(64位操作数)。

图2显示了指令单元对指令的处理和FLOS解码对指令的处理之间的时序关系。当FLOS解码加载时，将接收运算对象的缓冲区还没有从存储器加载。而不是保持解码直到操作数到达，当它"满了"时，FLOS设置与缓冲区相关联的控制位，使其内容被传输到加法器。加法器接收控制信息，使它发送数据到浮点寄存器R1，当它的来源寄存器被缓冲区设置满。

如果指令是一个"存储-寄存器"的算术函数，存储运算对象被处理为in load(控制位使它被转发到适当的单元)，但是浮点寄存器，连同操作，被译码器发送到适当的单元。接收到缓冲区后，单元将执行操作并将结果发送到R1寄存器

在"寄存器-寄存器"的算术指令中，两个浮点寄存器按连续的周期被传输到适当的执行单元。

存储的处理方式类似于"存储-寄存器"的算术函数，不同的是浮点寄存器的内容被发送到存储数据缓冲区而不是执行单元。

到目前为止，一次只处理一条指令已经被证明是相当简单的。现在考虑以下"项目":

Example 1

```
LD    F0   FLB1   LOAD register F0 from buffer 1

MD    F0   FLB2   MULTIPLY register F0 by buffer 2
```

负载可以像以前一样处理，但是乘法呢?当然，F0和FLB2不能像孤立乘法那样发送到乘数，因为FLB1还没有被设置为F0。这个序列说明了基本的优先原则:如果浮点寄存器是另一条未完成指令的接收器，那么它不能参与一个操作。也就是说，在寄存器的内容反映使用该寄存器作为接收器的最近操作的结果之前，不能使用该寄存器。

到目前为止提出的设计还没有纳入任何处理这种情况的机制。任何这类机制必须具备三项功能:

（1）它必须认识到依赖关系的存在。

（2）必须使相关指令的顺序正确。

（3）它必须区分给定的序列和这样的序列：

```
LD    F0,   FLB1
MD    F2,   FLB2
```

在这里，无论如何处置LD，它都必须允许独立MD继续执行。

前两个要求是保持程序逻辑完整性所必需的；第三个是满足性能目标所必需的。下一节将介绍实现这些目标的几种备选方案

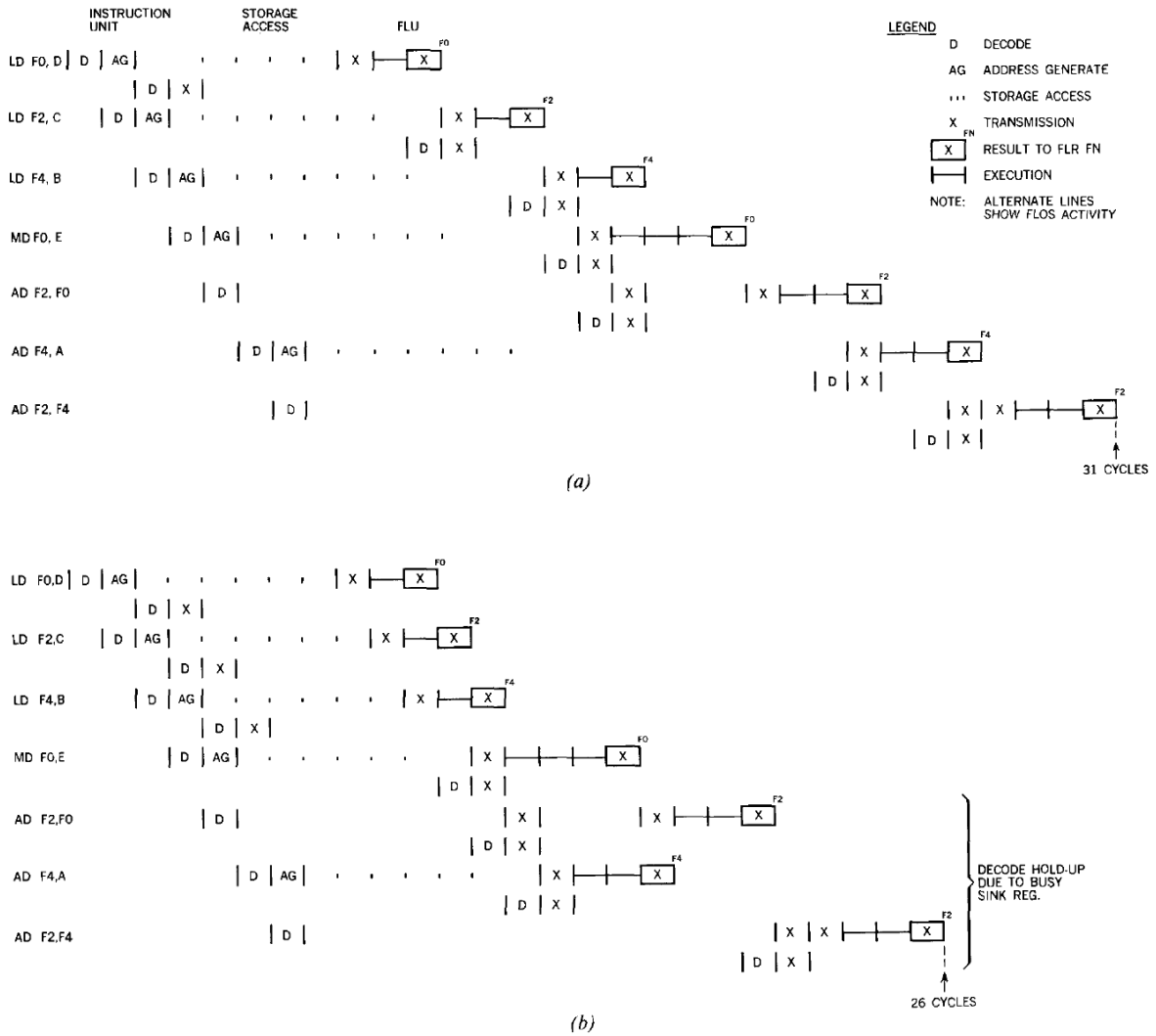


Figure 3 Timing for the instruction sequence required to perform the function A + B + C + D * E : (a) without reservation stations, (b) with reservation stations included in the register set.

3 Preservation of precedence:

也许预留优先级的最简单方案如下。一个“忙碌”位与四个浮点寄存器中的每个寄存器相关联。当FLOS解码发出指令将寄存器指定为接收器时，该位被设置;当执行单元返回结果到寄存器时，它被重置。如果FLOS的接收器的“忙碌”位处于开启状态，则FLOS不能发出任何指令。如果用于注册指令的寄存器的来源有它的忙碌位，FLOS设置与来源寄存器相关联的控制位。当一个结果被输入寄存器时，这些控制位使寄存器通过FLR总线被发送到等待它作为源的单元。

该方案很容易满足前两个要求。第三种是遇到程序员的帮助；他必须使用不同的寄存器来实现重叠。例如，表达式A+ B+C+D * E可编程如下：

Example 2

```
LD   F0,  D      F0 = D
LD   F2,  C      F2 = C
LD   F4,  B      F4 = B
MD   F0,  E      F0 = D * E
AD   F2,  F0     F2 = C + D * E
AD   F4,  A      F4 = A + B
AD   F2,  F4     F2 = A + B + C + D * E
```

忙碌位方案应该允许第二次加法和乘法同时执行(实际上，以任何顺序)，因为它们使用不同的接收器。不幸的是，图3a的时序图显示，不仅预期的重叠没有发生，而且许多周期丢失了传输时间。重叠不能实现，因为第一次加法使用了乘法的结果，加法器必须等待结果。循环失去控制，因为太多的指令使用加法器。除非有一个单元可用来执行指令，否则FLOS不能解码指令。当一个被分配的单元完成执行，它需要一个周期来传输事实到FLOS，以便它可以解码一个等待指令。类似地，当FLOS由于接收寄存器忙碌而被占用时，它不能开始解码，直到结果被输入到寄存器中。

Example 3a

```
LD   F0,  E
MD   F0,  D
AD   F0,  C
AD   F0,  B
AD   F0,  A
```

可以考虑的一种解决方案是添加一个或多个加法器。如果这样做了，一些程序定时，然而，它将变得很明显，执行电路将只使用小部分时间。大部分丢失的时间发生在加法器等待前指令产生的运算对象时。所需要的是一种收集运算对象(和控制信息)的设备，然后在所有条件都满足时参与执行电路。但这正是图1中接收寄存器和源寄存器的功能。因此，更好的解决方案是将不止一组寄存器(控制、接收、源)与每个执行单元相关联。每个这样的集合称为一个预留点。*现在发出的指示将由是否有合适的预留点而定。模型91中有3个加分预留点和2个乘除预留点。为了简单起见，它们被当作实际的单位来对待。因此，在将来，我们将谈到加法器1(A1)、加法器2 (A2)等，以及M/D1和M/D 2。

图3b显示了添加预留站对问题运行时间的影响：消除了五个周期。注意，第二个AD现在与MD重叠，实际上在第一个AD之前执行。虽然速度的提高是令人满意的，忙碌位方法容易实现，但仍然依赖于程序员。注意，表达式可以这样编码：

Example 3b

```
LOOP 1    LD   F0,   Ei
          MD   F0,   Di
          AD   F0,   Ci
          AD   F0,   Bi
          AD   F0,   Ai
          STD  F0,   Fi
          BXH i, −1, 0, LOOP 1 (decrease i by 1,
          branch if i > 0)
LOOP 2    LD   F0,   Ei
          LD   F2,   Ei + 1
          MD   F0,   Di
          MD   F2,   Di + 1
          AD   F0,   Ci
          AD   F2,   Ci + 1
          AD   F0,   Bi
          AD   F2,   Bi + 1
          AD   F0,   Ai
          AD   F2,   Ai + 1
          STD  F0,   Fi
          STD  F2,   Fi + 1
          BXH i, −2, 0, LOOP 2
```

在FLOS看来，循环1的迭代n + 1依赖于迭代n，因为两个迭代中的指令具有相同的汇集。但很明显，这两个迭代实际上是独立的。这个例子说明了两个指令序列可以相互独立的第二种方法。当然，第一种方法是让两个字符串拥有不同的接收寄存器。第二种方法是第二个字符串以加载开始。根据它的定义，加载启动一个新的、独立的字符串，因为它指示计算机销毁指定寄存器的以前的内容。不幸的是，忙碌位方案没有认识到这种可能性。如果要用这个方案实现重叠，程序员必须写循环2。(这种技术被称为翻倍或解开。它需要两倍的存储空间，但通过允许同时执行两次迭代，运行速度更快。)

针对这种情况，对忙碌位方案进行了改进。最吸引人的方法是将比特扩展为计数器。这似乎允许发出具有给定接收器的多个指令。每次发出时，FLOS增加计数器；每次执行时，计数器递减。然而，主要的困难是由于存储运算对象不按顺序返回。这可能导致指令n+1的结果被放在n的结果之前的寄存器中。当n完成时，它会错误地破坏寄存器的内容。

所审议的其他一些建议如果得到执行，其逻辑上的复杂性将危及实现一个快速周期。

4 The Common Data Bus:

前面的部分旨在描述在浮点指令之间实现并发的困难，并展示在设计的发展过程中克服这些困难的一些步骤。回顾过去，很明显，以前的算法之所以失败，是因为没有办法唯一地标识每条指令，并使用这些信息来排序执行，并将结果设置到浮点寄存器中。就FLOS的作用而言，对一个特定指令唯一的是执行它的单元。因此，这必须形成公共数据总线(CDB)的基础。

图4显示了CDB操作所需的数据路径。将图4与图1进行比较，除了预留点外，有明显的变化：缓冲区增加了另一个输出端口。该端口已与来自加法器和乘法器/分法器的结果相结合；这个组合就是国开行。CDB现在不仅可以到达寄存器，还可以到达所有预留点的接收和源寄存器，包括存储数据缓冲区，但不包括浮点缓冲区。这个数据路径将使负载可以在没有加法器的情况下执行，并且将使任何操作的结果在没有首先经过浮点寄存器的情况下对所有单元可用。

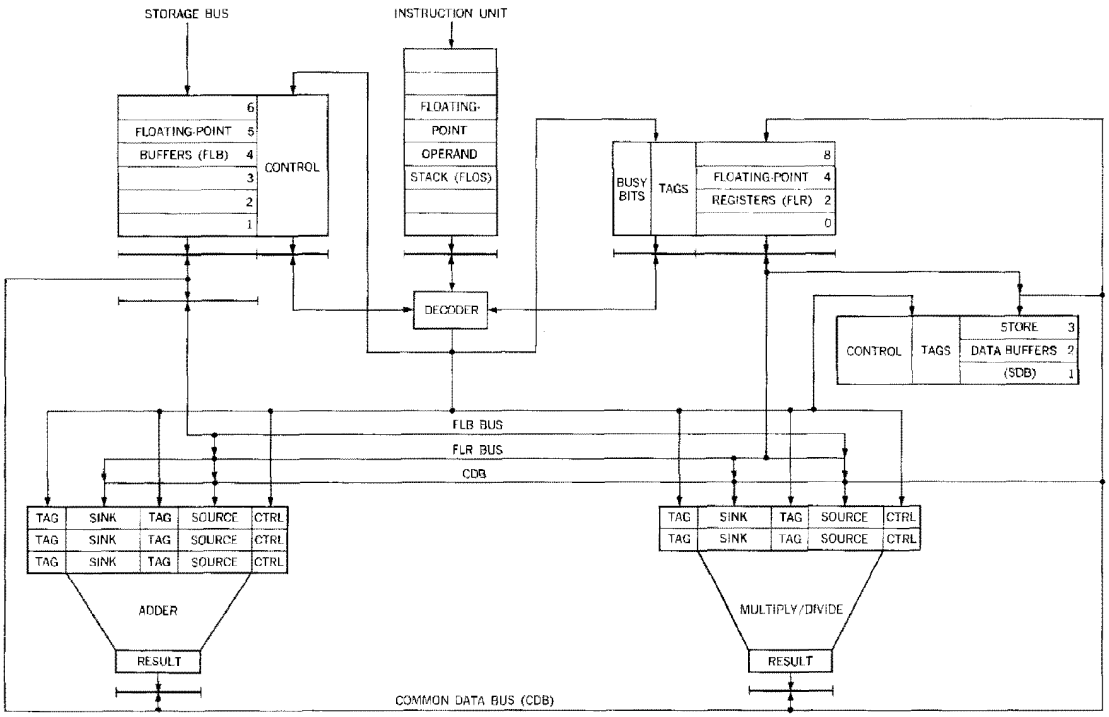


Figure 4 Data registers and transfer paths, including CDB and reservation stations.

请注意，CDB是由所有可以修改寄存器的单元提供的，并且它为所有可以将寄存器作为运算对象的单元提供数据。为国开行提供食物的单位。因此浮点缓冲区1到6被分配为数字1到6；这三个加法器(实际上是预留点)的编号是10到12；这两个乘/除法器是8和9。由于CDB有11个贡献者，一个4位二进制数就足以枚举它们。这个数字称为标签。标记与四个浮点寄存器中的每一个(除了忙碌位之外)、五个预留点中的每个源寄存器和接收寄存器以及三个存储数据缓冲区中的每个相关联。这样总共增加了17个四位标签寄存器，如图4所示。

标签也出现在另一个上下文中。一个标签是由CDB优先级控件生成的，用来识别下一个结果将出现在CDB上的单元。它的用途很快就会清楚。

该综合体的运作如下。在对每条指令进行解码时，FLOS检查每个指定的浮点寄存器的忙碌位。如果那个位是零，寄存器的内容可以通过FLR总线发送到所选单元，就像以前一样。在发出指令(只要求一个单元可用来执行它)时，FLOS不仅设置接收寄存器的忙碌位，而且还将其标记设置为所选单元的指定。源寄存器控制位保持不变。以AD F0, FLBI指令为例。在向加法器1发出此指令后，F0的控制位为：

BB	TAG
1	1010 (A1)

到目前为止，与前面的方法唯一不同的是标记的设置。当FLOS在译码时发现有忙碌位时，就会出现显著的差异。之前，这导致解码暂停，直到比特失效。现在FLOS将发布指令并更新标签。这样做时，它将不传输寄存器内容到所选单元，但它将传输"旧"标签。例如，假设前一个AD后面跟着第二个AD。在这第二个AD的解码结束，F0的控制位将是：

BB	TAG
1	1011 (A2)

一个周期后，A2预留点的汇集标签为1010，即与A1相同，A1是A2需要其结果的单元。

先来看看第一个AD的执行情况。在执行开始一段时间后，但在执行结束之前，"阿尔会向国开行提出请求。由于CDB是由许多源提供的，它的分时由一个中央优先电路控制。如果CDB是空闲的，优先级控制信号请求加法器A1输出它的结果，它广播请求者的标签(在这种情况下是1010)到所有预留点。每个活动的预留点(被选中但等待一个寄存器运算对象)将其接收和源标记与CDB标记进行比较。如果它们匹配，预留点将从CDB摄取数据。以类似的方式，将CDB标记与每个忙碌的浮点寄存器的标记进行比较。所有带有匹配标签的忙碌位寄存器从CDB中取出并重置它们的忙碌位。

前面提到的已经完成了实现保护优先地位目标的两步。首先，第二个AD在第一个AD结束之前不能开始，因为在第一个AD的结果出现在CDB上之前，它不能同时接收它的运算对象。其次，第一个AD的结果不能在第二次AD发出后更改寄存器 F0，因为F0中的标签与At不匹配。这些正是预期的效果。

在进行更详细的考虑之前，让我们概括一下这个方法的本质。浮点寄存器标记标识最后一个单元，它的结果要送到寄存器。当指令发出时，需要一个忙碌的寄存器，标签被发送到选择的单位，而不是寄存器的内容。单元不断地将此标记与CDB优先级控制生成的标记进行比较。当检测到匹配时，单元从CDB中摄取。单元一旦有了两个运算对象就开始执行。它可以从CDB或FLR总线接收一个或两个运算对象；存储到注册指令的源运算对象通过FLB总线传输。

当发出每条指令时，现有的标签被传送到所选单元，然后sink标签被更新。通过以这种方式传递标记，具有相同接收器的所有操作都被正确地排序，而允许其他操作独立进行。最后，浮点寄存器标签控制寄存器本身的改变，从而确保只有最近的指令才能改变寄存器。这有一个有趣的结果，循环如下：

Example 5

```
LOOP  LD    F0,  Ai
      AD    F0,  Bi
      STD   F0,  Ci   STORE
      BXH    i,  −1,  0,  LOOP
```

可以无限期地执行，而不改变F0的内容。通常情况下，只有最后一次迭代将其结果放置在F0中。

如前所述，有两种方法可以启动独立的指令字符串。第一种方法是指定不同的接收寄存器，第二种方法是加载寄存器。CDB处理前者的方式与忙碌位方案基本相同。负载问题，以前是一个难题，现在很简单。不管寄存器标记或忙碌位是什么，加载都会打开忙碌位，并将标记设置为指令单元分配给加载的浮点缓冲区。这将导致后续指令在缓冲区上排序，而不是在加载之前将寄存器标识为其接收器的任何单元上排序。缓冲区控件被设置为在存储运算对象到达时请求CDB。下面的例子和图5清楚地说明了这一点。

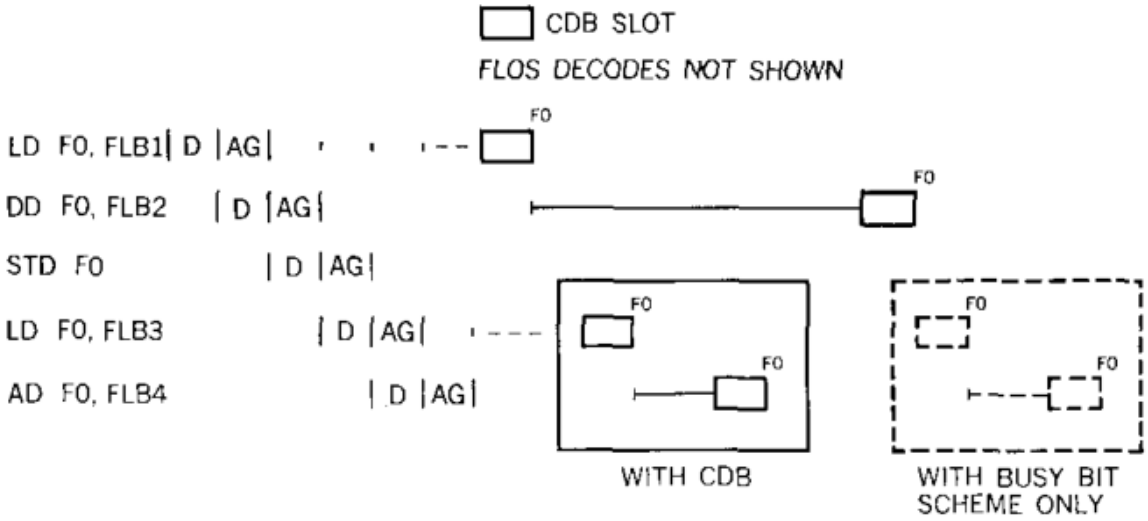
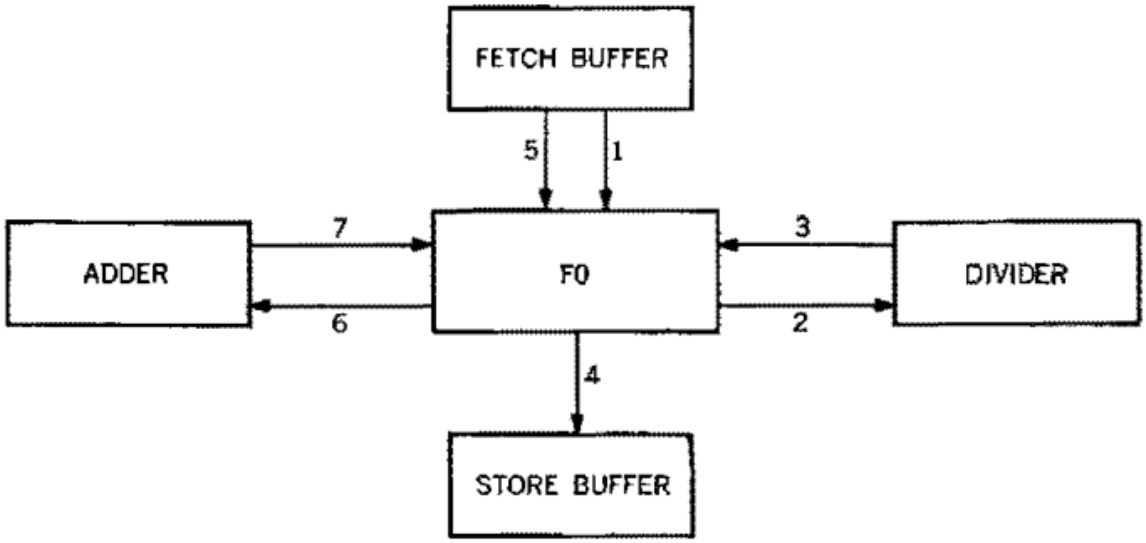


Figure 5 Timing sequence for Example 6, showing effect of CDB.

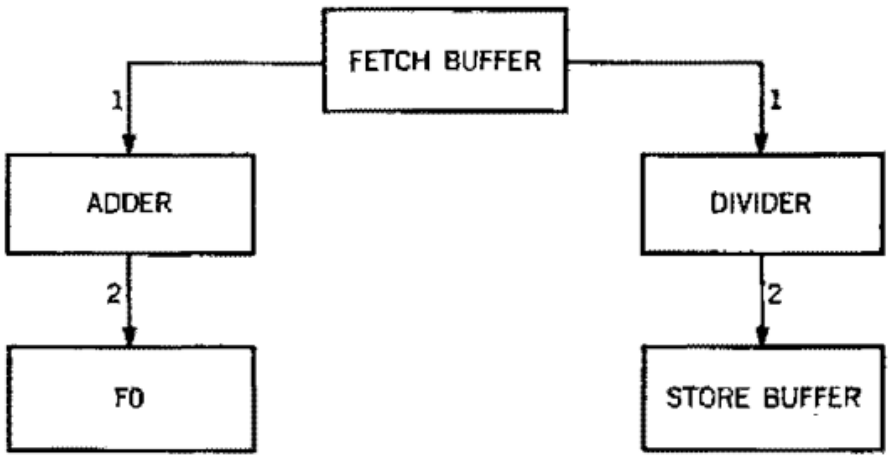
Example 6

```
LD      F0,  FLB1
DD      F0,  FLB2      DIVIDE
STD     F0,  A
LD      F0,  FLB3
AD      F0,  FLB4
```

注意，添加在划分之前完成。图5的虚线部分显示了单独使用忙碌位方案会发生什么。图6显示了这两种模式下的序列。该图图形化地说明了在忙碌位方案中使用单个接收器寄存器造成的瓶颈。因为所有的数据都必须通过这个寄存器，程序被简化为严格的顺序执行，即步骤1到步骤7。另一方面，对于CDB，接收寄存器几乎不会出现，程序被分成两个独立的并发序列。CDB的这种功能避免了循环加倍的需要。



(a)



(b)

Figure 6 Functional sequence for Example 6 (a) with busy bit controls only, (b) with CDB.

CDB使得我们可以在不需要任何时间的情况下有效地执行一些指令。在上面的示例中，存储发生在划分之后的CDB周期中。以类似的方式，通过将源浮点寄存器的标记移动到接收浮点寄存器的标记来完成忙碌寄存器的寄存器到寄存器的负载。例如，在序列中：

```
AD      F0,  FLB1
LDR     F2,  F0      move F0 to F2
```

当LDR被解码时，F0的标签将是1010 (A1)。解码器简单地将F2的标签设置为1010。现在，当AD的结果出现在CDB上时，F0和F2都将会接收，因为CDB标签的1010将会匹配每个寄存器的标签。因此，执行LDR不需要任何单元或额外的时间。

为了澄清这一概念，讨论中省略了一些细节，但实际上只有两个具有操作意义。首先，每个单元在完成执行之前必须向CDB申请两个周期。(这两个周期用于向CDB控件传播请求、在竞争单元之间建立优先级以及向被选择单元传播"选择"信号。)这将任何指令的执行时间限制在两个周期内。(当然，执行速度越快，对并发的需求或从中获得的好处就越少。)它还为加载的访问时间增加了一个周期。由于存在缓冲和重叠，这通常不会导致问题运行时间的增加。第二点与混合精度有关。由于架构定义导致在单精度操作时保留了FLR的低阶部分，因此在以下类型的程序中可能会出现错误：

```
LD      F0,     FLB1
AD      F0,     FLB2
AE      F0,     FLB3
```

因为只有最后一条单精度指令会改变F0，所以双精度AD的低阶结果会丢失。这是通过将一个位与每个寄存器相关联来处理的，以指示一个特定寄存器是未完成的单精度或双精度指令的接收。如果这个位与被解码指令的"长度"不匹配，译码就会暂停，直到忙碌的位失效。虽然这个策略解决了逻辑问题，但它是以牺牲性能为代价的。不幸的是，没有办法可以避免这种情况。但是请注意，所有单精度或所有双精度的程序都以可能的最大速度运行。只有在同一接收寄存器的单精度和双精度之间的接口才会受到延迟。

5 Conclusions:

提出了对高性能计算机设计有一定意义的两个概念。

第一个，预留点，是在一个单位之间传输时间是重要的环境中简单的一种快速的缓冲方法。由于存储访问和电路速度之间的差异以及连续操作之间的依赖关系，可以观察到(给定多个执行单元)每个单元花费大量时间等待运算对象。实际上，当执行电路可以被最先填充的预定工作站所占用时，预定工作站就在等待运算对象。

第二个，也是更重要的创新，CDB，利用预留点和简单的标记方案来保持优先级，同时鼓励并发。与CPU中的各种缓冲一起，CDB有助于降低Model91对编程的敏感性。但是，很明显，程序员仍然在执行对并发性的控制。执行A + B+ C + D * E的两个不同程序清楚地说明了这一点。

看起来CDB给每个操作的执行时间增加了一个周期，但实际上并不是这样。实际上，60-nsec CDB间隔中的30 nsec就可以执行CDB的所有功能。剩下的时间可以。在这种情况下，被执行单元用于实现更短的有效周期。例如，如果一个添加需要120 nsec，那么添加加上CDB所需的时间是150 nsec。因此，就add而言。机器周期可以是50 nsec，此外，即使没有CDB，将结果传输到浮点寄存器和返回到生成结果的单元所需要的时间也差不多。

下面的程序。一个典型的偏微分方程内环，说明了可能的性能提高。

```
LOOP    MD      F0,   Ai
        AD      F0,   Bi
        LD      F2,   Ci
        SDR     F2,   F0
        MDR     F2,   F6
        AD2     F2,   Ci
        STD     F2,   Ci
        BXH     i,  −1,  0,  LOOP
```

如果没有CDB，循环的一次迭代将使用17个循环，允许4个MD, 3个AD，不允许LD或STD。对于这类代码，CDB可以提高大约三分之一的性能。

6 Acknowledgments

7 References