游弋的灵魂之重定时

前面的三章我们介绍了 VLSI-DSP 硬件架构一些最重要的基本概念，比如迭代边界，之后简单讨论了一下最常见的两项技术：流水线和并行处理。从这一章开始，将隆重推出四件神兵利器，这里要讲第一件，游弋的灵魂之重定时（retiming）。谁是游弋的灵魂？这个疑问暂且记在心里，看完这章就得到答案了。接下来的内容分两节：

1. 延时（也就是寄存器）是如何在系统中游弋的？
2. 重定时的两个典型用法：流水（pipeline）重定时和割集（cut set）重定时。

讲解：第一节、重定时的来由、做法及性质

很多时候，我们想改变原始系统中延时的数量和分布，以改善系统的某些性能（如面积、速度和功耗）。具体的，流水线就是改变系统延时数目的一个特例。加入流水线后，系统中的延时数目增加了，所付出的主要代价就是面积变大，当然这带来了更快的运行速度。反过来，有时不需要那么快的运行速度，而是想减小面积，可能需要“撤去某些流水线”，以减少延时的数目。注意：延时的多少等同于寄存器的多少。为了能在各个性能指标之间进行灵活的折衷，就希望能制定一套如何来增加或减少系统延时数目以及改变系统延时分布的方法，重定时技术就应运而生了。

所谓的重定时就是一种，在保持系统的功能不变的前提下，改变系统延时数目和分布的方法。重定时在同步电路设计中有很多应用，包括缩短系统的时钟周期、减少系统中寄存器的数目、降低系统的功耗和逻辑综合的规模。以上具体的四种应用我们暂时不拿出来讲，大家可以参照书上的相关文献进行深入的学习，这里要讲解的是重定时最基本的做法和性质，有了这些基本知识，要深入去学习重定时的某项应用就轻松多了。

重定时基于那么一个简单的条件——系统的时不变性（time-invariant system），也就是说时不变系统才可使用重定时（更严格的说是，时不变计算节点才可进行重定时）。首先看看时不变系统的定义：

> 如果系统的输入输出关系不随时间而改变，那么这个系统就称为时不变系统。这样就意味着输入信号的延时会导致输出信号的延时，如若不然，就是时变系统。用数学公式表示为
>
> $$\begin{array}{ccc} y(n) & = & T[x(n)] \\ y(n-k) & = & T[x(n-k)] \end{array} \quad ,k \in \mathrm{Z}$$
>
> 其中 T 表示一个时不变系统，公式的意义是，输入 x(n)延时 k 个周期将导致输出 y(n)也延时相同的 k 个周期。——参考 胡广书《数字信号处理》第一册，1.5 节 离散时间系统的基本概念。

练习：给定系统

$$\begin{array}{cccc} 1) & y(n) & = & nx(n) \\ 2) & y(n)-ay(n-1) & = & bx(n) \quad ,y(-1)=0 \wedge b \neq 0 \end{array}$$

其中 n>=0，分别判断系统的时不变性。

答案：

1） 因为

$$T[x(n)] = nx(n) \Rightarrow T[x(n-k)] = nx(n-k)$$

但是

$$y(n-k)=(n-k)x(n-k)$$

显然，

$$(n-k)x(n-k)\neq nx(n-k)\Leftrightarrow y(n-k)\neq T\left[x(n-k)\right]$$

所以，公式 1）所示的系统不满足时不变性，是一个时变系统。

2） 令 $w(n)=y(n)-ay(n-1)$，则有

$$w(n)=bx(n)$$

因为

$$T\left[x(n-k)\right]=bx(n-k)$$

而且

$$w(n-k)=bx(n-k)$$

也就是说

$$w(n-k)=y(n-k)-ay(n-k-1)=T\left[x(n-k)\right]$$

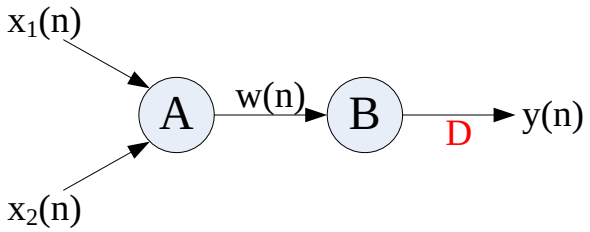
简化为

$$y(n-k)-ay(n-k-1)=T\left[x(n-k)\right]$$

所以，公式 2）所示系统满足时不变性。

注：对信号系统不是很熟的同志可以看看相关的书籍，直接跳过这里的讲解也是没有太大问题的。之所以从时不变系统开始讲，是为了能明白重定时的本质而已。

例子：将延时 D 看成是一个算子，则下图所示的系统



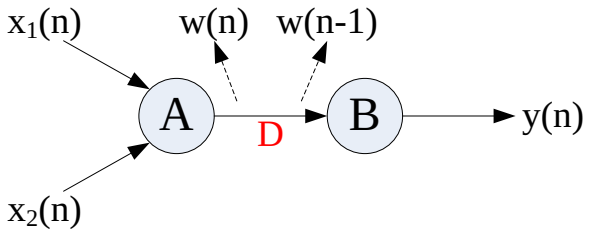
可用公式表示为

$$y(n)=D\left[B\left[w(n)\right]\right]=D\left[B\left[A\left[x_1(n),x_2(n)\right]\right]\right]$$

对于时不变系统而言，输出延时 k 个周期，也相当于输入延时 k 个周期，所以有

$$y(n)=B\left[D\left[w(n)\right]\right]=B\left[w(n-1)\right]$$

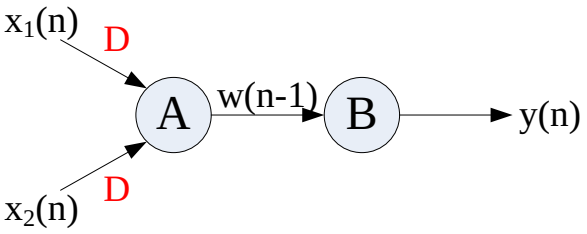
对应系统结构为



更进一步，有

$$\begin{aligned} y(n) &= B\left[D\left[w(n)\right]\right] = B\left[D\left[A\left[x_1(n),x_2(n)\right]\right]\right] \\ &= B\left[A\left[D\left[x_1(n),x_2(n)\right]\right]\right] = B\left[A\left[x_1(n-1),x_2(n-1)\right]\right] \end{aligned}$$

对应的系统结构为



End

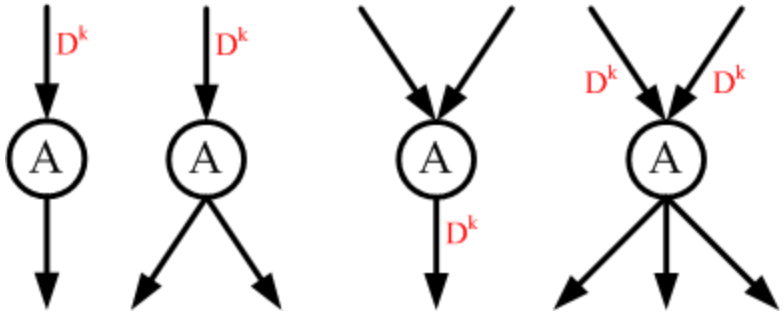
在时不变系统中，延时算子用 D 表示，D^k 表示延时 k 个周期，k 为非负整数，D^0（k=0）表示没有延时。假设系统为 y(n)=T[x(n)]，对 y(n)延时一个周期表示如下

$$\begin{aligned} y(n-1) = D[y(n)] &= D[T[x(n)]] \\ &= T[D[x(n)]] \\ &= T[x(n-1)] \end{aligned}$$

推而广之，对 y(n)延时 k 个周期，有

$$\begin{aligned} y(n-1) = D^k[y(n)] &= D^k[T[x(n)]] \\ &= T\left[D^k[x(n)]\right] \\ &= T[x(n-k)] \end{aligned}$$

下图所示四种重定时的情况动画图：1）单输入单输出节点，2）单输入多输出节点，3）多输入单输出节点，4）多输入多输出节点。



对于 1）单输入单输出节点，D^k 可以从一条输入边移动到一条输出边，反之亦然；2）单输入多输出节点，D^k 从一条输入边同时移动到多条输出边，且每条输出边都增加相同的 k 个延时，反之，每一条输出边同时提供一个 D^k，才能将其“合并”移到输出边；3）多输入单输出节点 类似 2）的情况；4）多输入多输出节点，每一条输入边同时提供一个 D^k，才能“合并”移动到每一条输出边，反之，每条输出边要同时提供一个 D^k，才能“合并”移动到每一条输出边。

大家仔细看动画，总而言之，任一节点的每一条输入边都减少一个 D^k，则其每一条输出边必增加一个 D^k；反之，每一条输出边都减少一个 D^k，则其每一条输入边必增加一个 D^k。

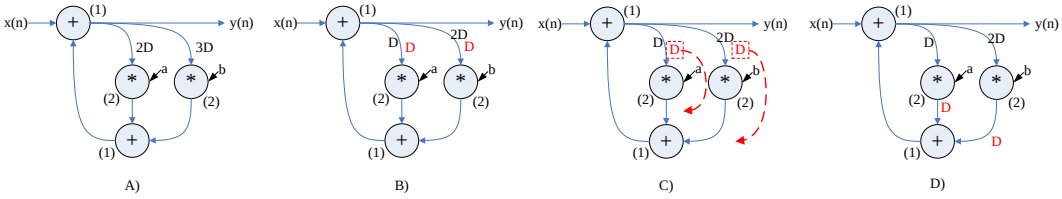
下面以课本的 IIR 滤波器为例，来说明如何使用重定时来缩短时钟周期和减少寄存器数目两个问题。IIR 的迭代公式如下

$$y(n) = ay(n-2) + by(n-3) + x(n)$$

直接根据迭代公式画出 DFG 如图一 A)，假设加法节点计算时间为 1u.t.，乘法节点计算时间

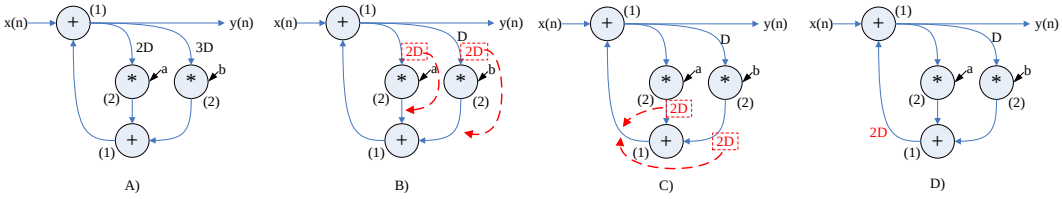
为 2u.t.。仔细观察图一 A)的 DFG，存在两条等长的关键路径，都是*++，长度为 1+1+2=4u.t.。

1）要得到更快的运行速度，必须想办法斩断这两条关键路径。利用重定时可以改变系统中延时的数目和分布，显然只要能在关键路径*++上放上若干延时将其斩断，同时又能保证系统的其他部分不产生比*++更长的关键路径就大功告成了。具体做法是，将两个乘法节点输入边的延时个分出一个来，如图一 B)红色 D 所示；然后将这两个红色的延时重定时（也就是移动）到乘法节点之后的输出边上，如图一 C)虚线所指的移动路径；最后得到如图一 D)所示的新结构。新结构存在三条等长的关键路径，两条为*，另一条为++，长度为 2u.t.，显然新 DFG 运行频率比旧 DFG 运行频率快一倍。



图一、y(n)=a*y(n-2)+b*y(n-3)+x(n) 原始DFG及其重定时版本

2） 另一方面，如果系统不需要那么高的运行频率，而是想尽可能减少实现的面积，那么就属于寄存器最小化问题。具体做法是，先将两个乘法节点输入边的延时分出两个来，如图二 B)所示；然后移动到乘法器之后，得到图二 C）的结构；继续将这些延时往前移，注意此处涉及到的加法节点只有一条边，故而两个输入边的延时“合并”，在加法节点的输出边上出现红色 2D 的延时。比较图二 A)和 D）两个结构，新结构相当于把原始结构的 4 个延时减少到 2 个延时，当然了新结构的关键路径为+*+，长度为 4u.t.，速度上没有改善。



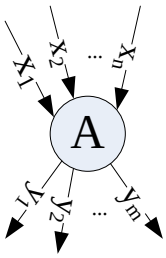
图二、y(n)=a*y(n-2)+b*y(n-3)+x(n) 原始DFG及其重定时版本

从上面这个 IIR 滤波器的例子，可以看出重定时是缩短系统时钟周期和减少系统寄存器数目的利刃。值得注意的是，重定时作用远不如此，比如重定时可以用来减少开关动作降低系统功耗：在具有大电容的节点输入端插入寄存器能够减少这些节点的开关动作率，从而导出低功耗解决方案。——重定时的低功耗设计，作为思考题，大家可以发贴说说自己的见解。

至此，大家对重定时应该都有一个直观的理解了，知道如何“手动”地对系统进行简单的重定时以改善某些性能指标。但是，实际系统往往有成百上千，甚至上万个节点，要想“手动”重定时，给出 1）时钟周期最小，2）寄存器最少，3）两者混合的某个折衷，三个不同的系统重定时方案，估计比登天还难。其实，重定时是一个改变系统延时的数目和分布的一个规则，利用重定时能导出满足特定单个目标或多个目标的 系统实现方案，换句话说重定时设计其实是一个优化求解的问题，而且往往是大规模的。要解决那么一个大规模优化问题，应该用计算机和优化算法来处理，需要我们去做的是，把重定时数学建模为一个可优化的问题，以及给出特定的目标函数。。。（这里说多了，但不知道大家有没有优化计算、优化搜索这些概念，其实也不是很难，推荐一个工程领域的优化工具 modefrontier，大家感兴趣可以看看，

此外近年来兴起的智能计算也可参考参考）

重点内容来了：**重定时的数学模型及其性质**

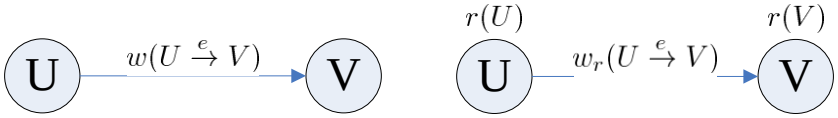


如图的一个节点 ，共有 n 条输入边和 m 条输出边。直观的重定时做法是，如果所有的输入边每一条都减少 k 个延时，那么所有的输出边每一条必增加 k 个延时；反之，所有的输出边每一条都减少 k 个延时，则所有输入边每一条必增加 k 个延时。为了进行自动重定时（让计算机来处理），必须告诉计算机每一个节点所需进行的重定时情况，总而言之，无非是那么三种情况（k 为非负整数）：

1） 对每一条输入边增加 k 个延时，同时在每一条输出边上减去 k 个延时；
2） 对每一条输出边增加 k 个延时，同时在每一条输入边上减去 k 个延时；
3） 不进行任何输入输出边的延时个数调整。

这么一来，只需为每个节点赋予一个整数值 k（可正可负，也可为 0）。如果 k>0，表示情况 1），每条输入边增加 k 个延时，同时每条输出边减去 k 个延时；如果 k<0，表示情况 2），每条输出边增加-k（也就是 k 的绝对值）个延时，同时每条输入边减去-k 个延时；k=0，就是情况 3），什么也不干。总结起来就是，输入边加上 k 个延时，输出边减去 k 个延时，k 为可正可负可为零的整数。

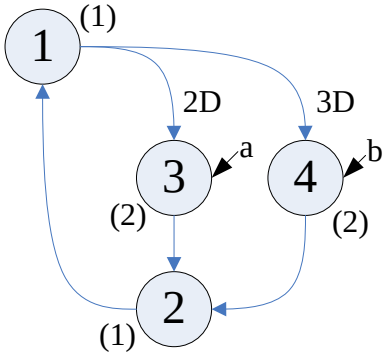
在任一个多节点系统中，可以为每个节点赋予一个整数值，用于指定每个节点需要进行何种情况的重定时操作。



如上图示，U 和 V 两个相连节点的情况，分别为每个节点赋予一个整数值 r(U)和 r(V)，边的权值就是边上延时的数目，左图原始的 DFG 中的一条边，右图为重定时之后的系统的相应边。根据上面所规定的做法，边 e 为节点 U 的输出边，所以边 e 的权值需要减去 r(U)个延时，同时边 e 又是节点 V 的输入边，所以需要再加上 r(V)个延时，所以重定时之后新边的延时为

$$w_r(e) = w(e) + r(V) - r(U)$$

这就是重定时方程的由来了。不过话说回来，因为一开始我们并不知道为每个节点赋予什么样的值才是合理的或者说是最佳的。假设随便设定一些整数值，如果发现 DFG 中某条边的重定时延时 w_r(e)是个负值，那么说明前面所设定的节点值是不合理的，自然界就没有延时为负数的 情况，而且没法解释其物理意义。所以规定，只有当 DFG 中所有边的 w_r(e)都是大于等于零时，所设定的节点值才是合法的，所进行的重定时才是可以实现的。

再以课本上的 IIR 滤波器为例，重新画出 DFG 如右 

，

为了方便计算节点直接用序号表示。总共四个节点，假设为每个节点所赋之值为 r(i)，i=1，2，3，4。共有 5 条边，进行重定时之后的边的权值计算如下:

$$\begin{array}{rcccccl} w_r(1\to 3) & = & w(1\to 3)+r(3)-r(1) & = & 2+r(3)-r(1) & \geqslant & 0 \\ w_r(1\to 4) & = & w(1\to 4)+r(4)-r(1) & = & 3+r(4)-r(1) & \geqslant & 0 \\ w_r(2\to 1) & = & w(2\to 1)+r(1)-r(2) & = & r(1)-r(2) & \geqslant & 0 \\ w_r(3\to 2) & = & w(3\to 2)+r(2)-r(3) & = & r(2)-r(3) & \geqslant & 0 \\ w_r(4\to 2) & = & w(4\to 2)+r(2)-r(4) & = & r(2)-r(4) & \geqslant & 0 \end{array}$$

这是一个多元一次不等式方程组，求解这个方程组所得的解，就是合法的重定时实现。另外，可以设定多个目标，比如系统时钟周期和寄存器个数等，在此不等式方程组的约束条件下，搜索出使得规定目标最小化的解。

题外话：关于优化求解这个话题我们以后再详细讨论，因为涉及到一些优化方面的内容，我需要弄懂并找到好的方法之后才能和大家讨论怎么来做，当然了也欢迎各位踊跃提出自己处理这个问题的好方法。我个人拟用 modefrontier 来处理，貌似 matlab 的 lmi 也能求解一些简单的情况，课本上也给出了一些求解的方法，感兴趣的同志可以去看，然后发贴讨论。

上面详细的讨论了重定时来由来和做法，下面就是重定时的性质了。每一种方法都有其自身的一些特殊性质，明白这些性质将进一步加深我们对重定时的理解。

性质一　重定时的路径 $p=V_0 \xrightarrow{e_0} V_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{k-2}} V_{k-1} \xrightarrow{e_{k-1}} V_k$ 的权重是 $w_r(p)=w(p)+r(V_k)-r(V_0)$。

证明：直接根据计算式子化简就可以了

$$\begin{aligned} w_r(p) &= \sum_{i=0}^{k-1} w_r(e_i) \\ &= \sum_{i=0}^{k-1} (w(e_i)+r(V_{i+1})-r(V_i)) \\ &= \sum_{i=0}^{k-1} w(e_i) + \left(\sum_{i=0}^{k-1} r(V_{i+1}) - \sum_{i=0}^{k-1} r(V_i)\right) \\ &= w(p)+r(V_k)-r(V_0) \end{aligned}$$

End

性质二　重定时不改变环路中的延时数。

证明：对于环路，路径的起点和终点是一样的，都是同一个点。假设选第 i 个节点为起始点，同时也是终点，则有

$$w_r(p) = w(p) + r(V_i) - r(V_i) = w(p)$$

End

题外话：在第二章迭代边界中，曾说到为什么迭代边界是环路所特有。Tangfei 网友说只要能增加环路中的延时，不就打破迭代边界的限制了吗？当时我的回答很生硬，直接说环路中做不到增加新的延时（非环路可以做到，但环路做不到）。这个问题在这里将得到一个解释，按照重定时规则，环路中是不会增加新的延时的（性质二），因为所有改变寄存器数目和分布的调整都必须遵循重定时的规定。——献给好学的 Tangfei，希望他对迭代边界的疑惑能在此完全清除。

性质三　重定时不改变 DFG 的迭代边界。

证明：这个只能用显然成立来解释了。既然性质二已经说明，环路延时不会变化，而迭代边界又只与环路相关，所以重定时不能改变 DFG 的迭代边界。

End

性质四　每一个节点的重定时值增加同一个常数 j，不会改变从 G 到 G_r 的映射。

证明：你会认为 这两个式子不是得到相同的结果吗？

$$w_r(e) = w(e) + (r(U) + j) - (r(V) + j)$$
$$w_r(e) = w(e) + r(U) - r(V)$$

End

终于结束这一节的内容了。重定时是一项非常伟大的技术，流水线就是一个重定时的特例。本章的第二节，将轻松的来小试一下牛刀，练习两个典型的重定时技术：流水重定时（也就是流水线）和割集重定时。关于高级重定时技术的问题，先让某些牛人发表自己的看法，在寒假之后我在跟大家一起说说我自己的做法。我的做法（基于 modefrontier）是一个傻瓜式方法，虽然不是最强大，但应该是最容易操作的方法之一。

题外话：接近期末了，事情渐渐多了起来，所以发贴的时间也变得漫长和不确定，请大家原谅。不过再怎么忙，我都会把问题弄清楚想明白再与大家讨论。不论我多用心都不能保证自己的理解以及表达完全的正确，所以希望大家批评指正（到现在才体会到为什么每一本书的序言都会说到这些话^_^）。

上一节我们基于线性系统的时不变性引出了一种改变系统延时数目和分布的强大技术：重定时。如果将延时看成是士兵，那么重定时就是一种排兵布阵的战略，通过将合适数量的士兵安置到合适的位置，以逸待劳，就能给狠狠对侵略的敌人予以重击。

对系统的 DFG 应用重定时是非常直观的，从前一节的各个例子中很容易看明白并掌握它；但是对于大型逻辑电路系统，其 DFG 往往包含成千上万个节点，边也是“不计其数”，此时人工进行 DFG 的重定时是不太可能的。由此我们给出了一套严格的数学方法，将重定时问题（包括时钟周期最小化重定时和寄存器最小化重定时）转化为以线性不等式方程组为约束的最小化求解问题。这类最小化问题的求解可以使用 matlab 自带的 lmi 工具箱或者是优化算法（如进化计算）进行求解。

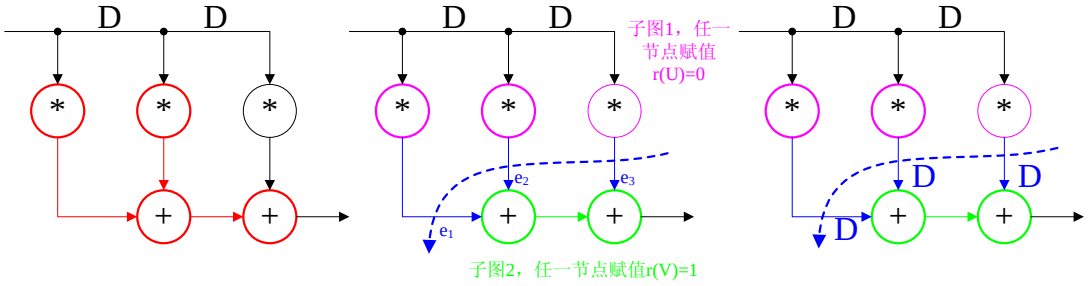
这里暂不去讨论重定时的优化求解问题，而是作为高级话题以后再研究。作为重定时的实践，以下将讨论两个典型的重定时用法：流水线重定时和割集重定时。

讲解：第二节、重定时的两个典型用法：流水（pipeline）重定时和割集（cut set）重定时

一、流水重定时

流水重定时，实际就是平时所说的流水线技术。在第三章——打怪抢宝流水与并行中，给出了流水线技术的做法：在单向割集的每一条边上插入同等数目的流水线寄存器，即可得到流水结构。

何以见得流水线技术就是重定时的一种特例呢？如下图一所示为 3 阶 FIR 滤波器的流水重定时与流水线的等价示例，假设加法节点计算时间为 1u.t.，乘法节点计算时间为 2u.t.。



A) 3阶FIR直接形式DFG　　B) 单向割集与流水重定时　　C) 流水重定时后的结构

图一、流水重定时与流水线的等价示例

图一 A)为 3 阶 FIR 滤波器直接形式 DFG，关键路径为*++，如红色路径所示，长度为 2+1+1=4u.t.；如图一 B)，确定一个单向割集，将原始 DFG 分为两个不相连的子图，并对子图 1 的任一个节点 U 赋予重定时值 0，子图 2 的任一节点 V 赋予重定时值 1；则根据重定时方程可以算出单向割集中的三条“绿色”边重定时后的权值分别为

$$\begin{aligned} w_r(e_1) &= w(e_1)+r(\mathrm{V})-r(\mathrm{U}) = 0+1-0 = 1 \\ w_r(e_2) &= w(e_2)+r(\mathrm{V})-r(\mathrm{U}) = 0+1-0 = 1 \\ w_r(e_3) &= w(e_3)+r(\mathrm{V})-r(\mathrm{U}) = 0+1-0 = 1 \end{aligned}$$

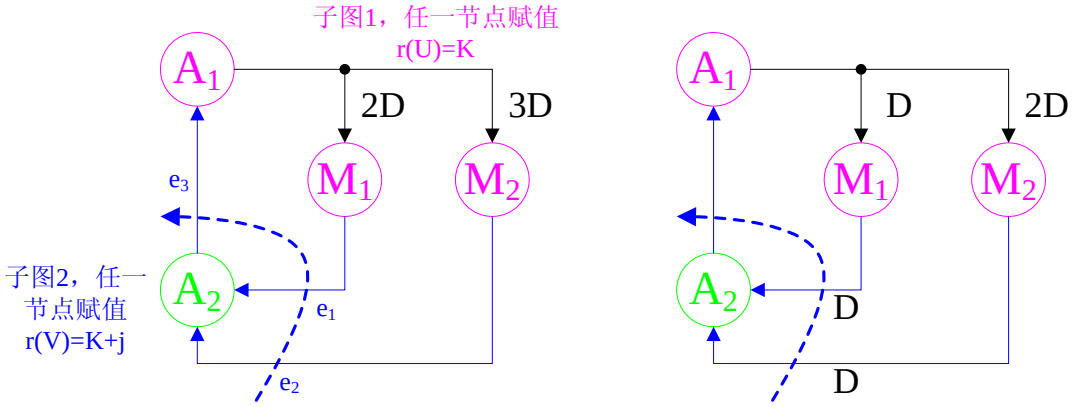
重定时后的结果，就是在单向割集的每一条上插入一个延时。值得注意的是，对于子图 1 或者子图 2 内的所有边，重定时后的边的权值不变，原因就是子图内的所有节点都具有相同的重定时值 j (j=0 or 1)，所以有

$$w_r(e) = w(e)+r(\mathrm{V})-r(\mathrm{U}) = w(e)+j-j = w(e)$$

总结：流水重定时，首先是确定一个单向割集，该割集将原始 DFG 分离为 2 个不相连的连通子图（子图内部节点是连通的，而子图之间不连通）；然后对割集边所指向的子图中每一个节点赋予重定时值 K+j，对另一个子图赋予重定时值 K；如此一来，对于割集中的每一条边，重定时之后的延时增加 K+j-K=j 个延时，当 j=1 时，就是一级流水线的单向割集插入。

二、割集重定时

更一般的，不限定割集为单向割集，对普通的割集也能应用重定时技术，统称为割集重定时。如下图二 A)所示的 iir 系统，A1、A2 为加法节点，计算时间为 1u.t.，M1、M2 为乘法节点，计算时间为 2u.t.。



A) iir 原始DFG　　B) 割集重定时后的DFG

图二、割集重定时示例

对子图 1 的任一节点赋予重定时值 K，子图 2 的任一节点赋予重定时值 K+j，根据重定时方程可以算出割集中三条“绿色”边重定时后的权值

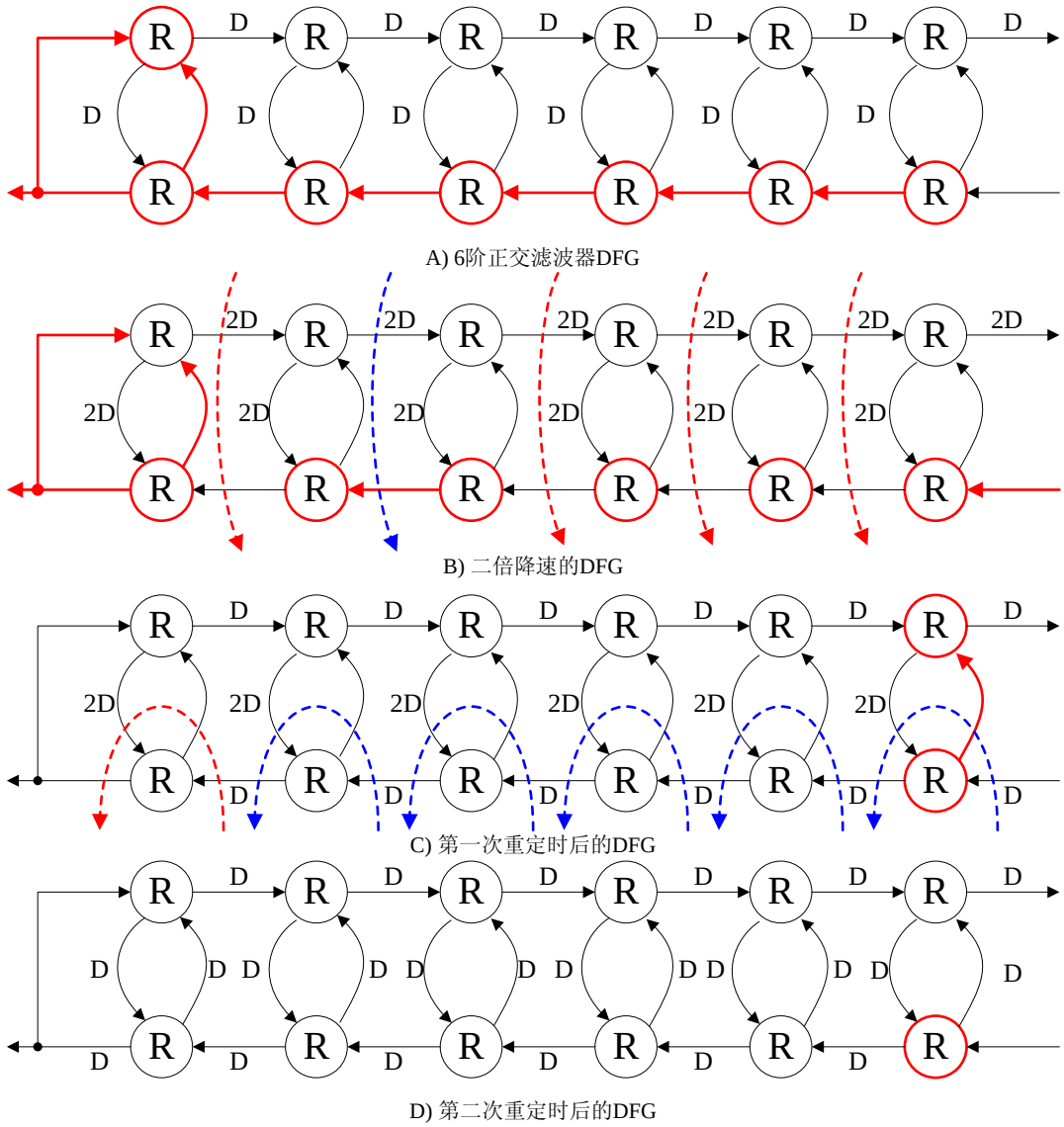
$$\begin{aligned} w_r(e_1) &= w(e_1)+r(\mathrm{V})-r(\mathrm{U}) &= w(e_1)+K+j-K &= j \\ w_r(e_2) &= w(e_2)+r(\mathrm{V})-r(\mathrm{U}) &= w(e_2)+K+j-K &= j \\ w_r(e_3) &= w(e_3)+r(\mathrm{V})-r(\mathrm{U}) &= w(e_3)+K-(K+j) &= 1-j \end{aligned}$$

令 j=1，得到一个重定时结构如图二 B)所示。分析图二 A)和 B)的关键路径可知，重定时将原始 DFG 中的关键路径*++，斩断为*或者++，关键路径长度从 2+1+1=4u.t.缩短为 2u.t.。

动手试试：对下图所示的 6 阶正交滤波器结构进行重定时，以获得最小的迭代周期，假设坐标旋转数字计算机 R 计算时间为 T u.t.，且节点不可拆分。

分析图三 A)的结构，关键路径如红色路径所示，长度为 7T u.t.，要想切断关键路径，最为直接想到的就是看能不能将 DFG 分为若干级，每一级只包含上下两个 R 节点。如图三 B)所示的割集线，如果能在下面的边（自右向左）插入一个寄存器，那么就有可能斩断关键路径，但是，如果直接进行上面的边减去一个延时，下面的边加上一个延时，那么上面的边就会变为零延时，从而产生新的关键路径，且长度也是 7T u.t.。为了做到这一点，首先我们需要对 DFG 进行 2 倍降速，2 倍降速就是使得原始 DFG 的每条边的延时都变为原来的 2 倍，输入序列相应的隔一位插入一个 0，输出也是隔一位取一个点，这样就能保证 2 倍降速后的电路功能不变（关于 n 倍降速的电路问题，留作思考题大家仔细去体会体会）。2 倍降速之后再进行割集重定时就得到图三 C)的 DFG，此时的关键路径长度变为 2T u.t.，仔细观察该 DFG，其实还可以进一步进行割集重定时，如图三 C）的割边所示，再次进行割集重定时，

就会得到最终版本——图三 D）的 DFG，最终版本中的关键路径长度已经缩短到一个 R 节点，长度为 T u.t.，这就是该电路所能达到的最小迭代周期了。



A) 6阶正交滤波器DFG

B) 二倍降速的DFG

C) 第一次重定时后的DFG

D) 第二次重定时后的DFG

图三、6阶正交滤波器最小化时钟周期重定时

题外话：做完这个例题，真的很让人觉得兴奋，感觉自己就像是个小小诸葛亮，羽毛扇一挥，调动千军万马，巧妙布阵痛击敌人。例题虽然简单，但的确让人领略了重定时的强大之处，想想经过巧妙重定时后的电路，将原始 DFG 迭代周期从 7T u.t.缩短到 T u.t.，加速 7 倍，令人叹为观止。

小结，这一节我们讨论了重定时技术的两种特例：流水线重定时和割集重定时，说起来流水线重定时又是割集重定时的特例，也就是说流水线重定时“相当于”将割集重定时的割集限制为单向割集的情况。对于高级的重定时应用我们暂时不去讨论，因为我正以此为课程论文进行“研究”，试图提供一种更为灵活的求解方法，等做好了再与大家分享，敬请关注^_^。。。