S-38.220
Postgraduate Course on Signal Processing in Communications,
FALL - 99

Pierre COULON

HUT/Communications Lab.
Otakaari 8 Fin-02015 HUT

Pcoulon@cc.hut.fi

Date: 01.11.1999

ABSTRACT

Nowadays, the users demand always smaller electronic devices such as mobile phone. This paper deals with this problem and presents the folding techniques, which are a practical way to reduce the number of functional units on the silicon area. As the folding transformation produce numerous registers, this paper deals also with a manner to reduce this number of registers. Some examples such as biquad filter program are introduced in order to clarify the different steps of these techniques.

TABLE OF CONTENTS

1. INTRODUCTION

An important factor in designing DSP Architectures is the space occupied by the integrated circuit which is directly link to the space used by the functional units on the silicon area. A manner to reduce the area occupied by these units is simply to reduce the number of units on the silicon area, which is done in applying a folding transformation. By executing multiple algorithm operations on a single functional unit (such as addition operations) or in other words in time multiplexing, the number of functional units in the implementation is reduced, resulting in an integrated circuit with low silicon area. This paper deals with the simpler case of a single clock but these DSP architectures can be operated using multiple clocks.

Sometimes the DSP architecture is simple enough to use ad hoc techniques to reduce the number of adders and multipliers, for instance, but in the general case we need the systematic techniques described in this paper to design the time-multiplexed architectures.

2. FOLDING TRANSFORMATION

2.1 Example of a simple folding transformation.

This simple example is giving in order to clarify the concept of folding. The figure 2.1 shows an example of how 2 addition operations can be time-multiplexed on a single pipelined hardware adder. This DSP circuit computes y(n)=a(n)+b(n)+c(n).

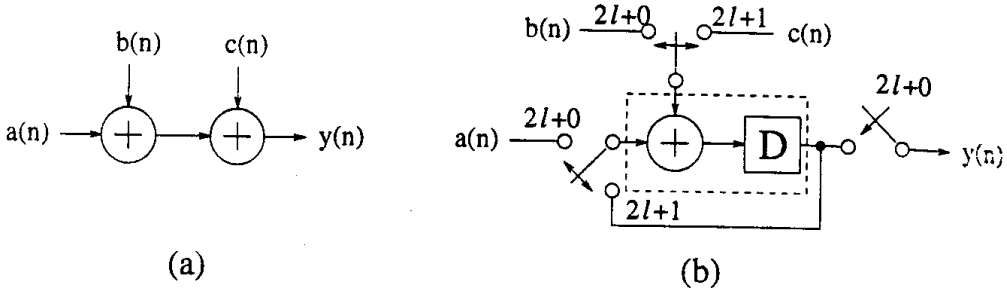


Figure 2-1: (a) A simple DSP program with 2 addition operations. (b) A folded architecture where the 2 addition operations are folded to a single hardware adder with 1 stage of pipelining.

Table 2-1: Operation of the first 6 Cycles of the folded hardware in fig 2-1(b)

Cycle	Adder Input (Left)	Adder Input (top)	System output
0	**a(0)**	**b(0)**	**-**
1	**a(0)+b(0)**	**c(0)**	**-**
2	**a(1)**	**b(1)**	**a(0)+b(0)+c(0)**
3	**a(1)+b(1)**	**c(1)**	**-**
4	**a(2)**	**b(2)**	**a(1)+b(1)+c(1)**
5	**a(2)+b(2)**	**c(2)**	**-**

The table 2.1 gives the process of the folded program shown in figure 2-1(b). Notice that one output sample is produced every 2 clock cycles, and one sample of each input signal is consumed every 2 clock cycles (a(k) is used in cycle 2k) and therefore each input has to remain valid for 2 clock cycles before changing. Moreover it is important to notice that the first implementation requires 2 adders but computes one iteration of the program in the time required to perform an addition: T_{add}. On the other hand, the folded implementation in fig 6.1 (b) uses only 1 adder but computes one iteration of the program in $2*T_{add}$ time. In general there is a compromise between

the number of functional units and the time spend to compute 1 iteration: if the folding factor is N we might increase the computation time by a factor of N.

Moreover, while the folding transformation reduces the number of functional units, it may also lead to an architecture, which uses a large number of registers. This paper will present some technique to reduce the number of registers required to implement a folded DSP architecture and to allocate data to these registers. These techniques are important to keep the area consumed by memory to a minimum.

2.2 Basis of the folding transformation.

This section deals with the mathematical considerations about folding.

Consider the edge e connecting the nodes U and V with *w(e)* delays, as shown in *Fig. 2-2-1*.

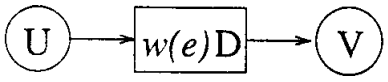


Figure 2-2-1: An edge e, U→V with w(e) delays.

Let the executions of the l-th iteration of the nodes *U* and *V* be scheduled at the time units *Nl+u* and *Nl+v*, respectively, where u and v are the folding orders of the Nodes U and V that satisfy *0 u , v N-1*. Note that the folding order of a node is the time partition to which the node is scheduled to execute in hardware. The functional units that execute the nodes U and V are denoted as H_U and H_V, respectively. Note that N is the number of operations folded to a single functional unit and is also referred to as the folding factor. If H_U is pipelined by P_U stages, then the result of the l^{-th} iteration of the node U is available at the time unit *Nl+u+Pu*. Since the edge *e* has *w(e)* delays, the result of the l^{-th} iteration of the node *U* is used by the $(l+w(e))^{-th}$ iteration of the node *V*, which is executed at *N(l+w(e))+v*. Therefore, the result must be stored for:

$$D_F(U \xrightarrow{e} V) = [N(l + w(e)) + v] - [Nl + P_U + u] = Nw(e) - P_U + v - u$$

time units, which is independent of the iteration number *l*. The edge e is implemented as a path from H_U to H_V in the architecture with $D_F(U \rightarrow V) = D_F(e)$ delays, and data on this path are input to H_V at *Nl+v*, as it is shown in *Fig 2-2-2*.

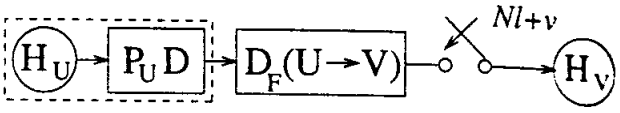


Figure 2-2-2: The folded path corresponding to fig2-2-1. The data begin at the functional unit H_U which has P_U pipelining stages, pass through $D_F(U \rightarrow V)$ delays, and are switched into the functional unit H_V at the time instances $Nl+v$.

Definition of a folding set:

A folding set contains N element (N is the folding factor)

In folding set, the elements are

- executed by the same functional units
- ordered: the j-th element is executed during the time partition j.

Ex: S1={A1,Ø,A2}

A1 can be denoted as (S1|0) and A2, (S1|2). The null operation (S1|1) implies that the functional unit will not be utilized at time instances *3l+1*.

2.3 Folding technique applied on the biquad filter

As an example is always very useful to understand a process, the aim of this section is to analyze the folding technique on the retimed biquad filter

In this example, assume that addition and multiplication require 1 and 2 u.t., and 1 stage pipelined adders and 2 stage pipelined multipliers are available: P_A=1 and P_M=2.

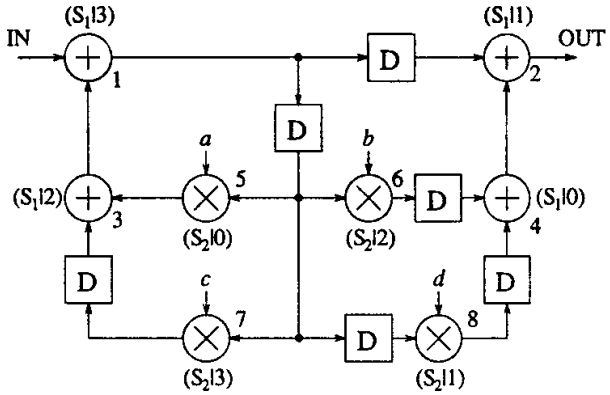


Figure 2-3-1: The retimed biquad filter with valid folding sets assigned

The folding factor is 4. That means that the iteration period is 4 u.t and each node of the biquad filter is exactly executed on every 4u.t in the folded architecture. In the new architecture, each functional unit execute 4 operations of the DSP program.

Folding sets:

S1={4,2,3,1}, only addition operations
S2={5,8,6,7}, only multiplication operations

The folded architecture is obtained from the Data-Flow Graph Fig 2-3-1 by writing the folding equation for each of the 11 edges in the DFG.

$D_F(U \rightarrow V)=N(w_{(U \rightarrow V)})-P_U+v-u$

$D_F(1 \rightarrow 2)=4(1)-1+1-3=1$
$D_F(1 \rightarrow 5)=4(1)-1+0-3=0$
$D_F(1 \rightarrow 6)=4(1)-1+2-3=2$
$D_F(1 \rightarrow 7)=4(1)-1+3-3=3$
$D_F(1 \rightarrow 8)=4(2)-1+1-3=5$
$D_F(3 \rightarrow 1)=4(0)-1+3-2=0$
$D_F(4 \rightarrow 2)=4(0)-1+1-0=0$
$D_F(5 \rightarrow 3)=4(0)-2+2-0=0$
$D_F(6 \rightarrow 4)=4(1)-2+0-2=0$
$D_F(7 \rightarrow 3)=4(1)-2+2-3=1$
$D_F(8 \rightarrow 4)=4(1)-2+0-1=1$

$D_F(1 \rightarrow 8)=5$ means that in the folded architecture, there is an edge from the adder to the multiplier with 5 delays. As the node 8 corresponds to (S2|1) the folded edge 1→8 is switched at the input of the multiplier at $4l+1$.

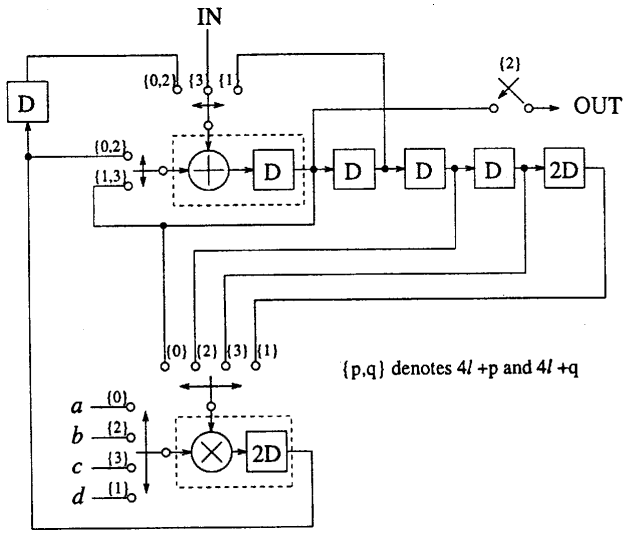


Figure 2-3-2: The Folded biquad filter using the folding sets given in2-3-1

We need $D_F(U \rightarrow V)$ f 0 for every D_F.

Retiming can used to either satisfy this property or determine that the folding sets are not feasible.

Using retiming, the number of delays on the edge U→V is changed from $w(e)$ to $w_r(e)=w(e)+r(V)-r(U)$

$w_r(e)$ is the number of delays in the retimed DFG, and r(X) is the retiming value of the node X.
After retiming, the constraint can be written:
$D'_F(U \rightarrow V)=N(w(e)+r(v)-r(u))-P_U+v-u \geq 0$

as r(U) and r(v) are integers:

$$r(U)-r(V) \leq \left\lfloor \frac{D_F(U \xrightarrow{e} V)}{N} \right\rfloor$$

Taking the previous example about the biquad filter, we have the following table:

Table 2-3-1: Folding Equations and Retiming for Folding Constraints

Edge	Folding Equation	Retiming for Folding Constraints
1→2	$\mathbf{D_F}$(1→2)=-3	**r(1)-r(2) ≤-1**
1→5	$\mathbf{D_F}$(1→5)=0	**r(1)-r(5) ≤0**
1→6	$\mathbf{D_F}$(1→6)=2	**r(1)-r(6) ≤0**
1→7	$\mathbf{D_F}$(1→7)=7	**r(1)-r(7) ≤1**
1→8	$\mathbf{D_F}$(1→8)=5	**r(1)-r(8) ≤1**
3→1	$\mathbf{D_F}$(3→1)=0	**r(3)-r(1) ≤0**
4→2	$\mathbf{D_F}$(4→2)=0	**r(4)-r(2) ≤0**
5→3	$\mathbf{D_F}$(5→3)=0	**r(5)-r(3) ≤0**
6→4	$\mathbf{D_F}$(6→4)=-4	**r(6)-r(4) ≤-1**
7→3	$\mathbf{D_F}$(7→3)=-3	**r(7)-r(3) ≤-1**
8→4	$\mathbf{D_F}$(8→4)=-3	**r(8)-r(4) ≤-1**

The retiming technique tell us that if the constraint graph contains a negative cycle then there is no solution else one solution is r(i) which is the shortest path from the node 9 to the node i as shown in Fig 2-3-4

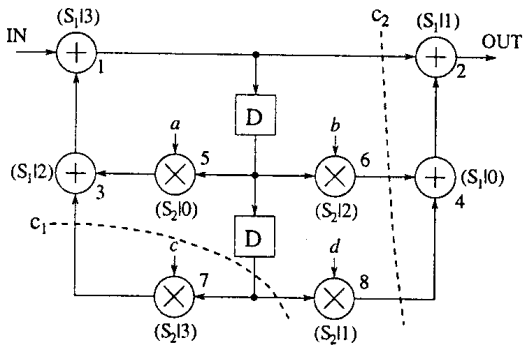


Figure 2-3-3: The original DFG resulting in some negative folded edge delays. The retimed DFG resulting in all nonnegative folded edge delays is shown in Fig 2-3-1

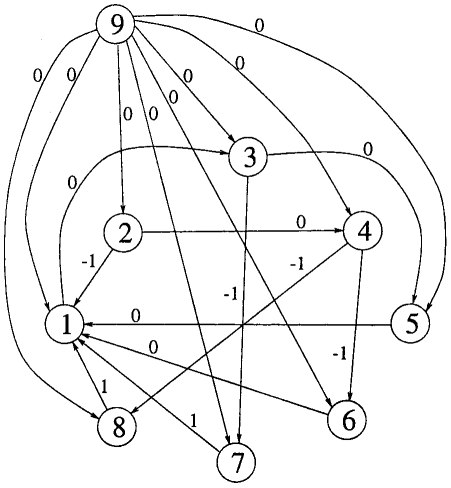


Figure 2-3-4: The constraint graph for the set of inequalities in the right-hand column of Table 2-3-1. Node 9 is the host node.

One solution is here: r(1)=-1, r(2)=0, r(3)=-1, r(4)=0, r(5)=-1, r(6)=-1, r(7)=-2 and r(8)=-1. Another practical way to reach the result is using cutsets. Note that the edge with negative D_F are 1→2, 6→4, 7→3, 8→4. If we add w delays in the edge U→V it increase the D_F by Nw. Thus we just have to increase the number of delays on each edge and that can be done by using cutsets retiming for the cutsets marked c1 and c2 in the previous figure. Note that the resulting DFG is exactly the DFG on figure 2-3-1.

3. REGISTER MINIMIZATION TECHNIQUES

The aim of this section is to reduce the number of registers used in the DSP architecture to keep a small area used by memory on the silicon.

3.1 Lifetime Analysis

First of all, some words about the data life. A data sample or *variable* is *live* from the time it is produced until the time it is consumed then it's *dead.* Each variable need 1 register during each time unit it is live. In lifetime analysis, the number of live variables at each time unit is computed, and the max number of live variables at any time is equal to the minimum number of registers required to implement the DSP program.

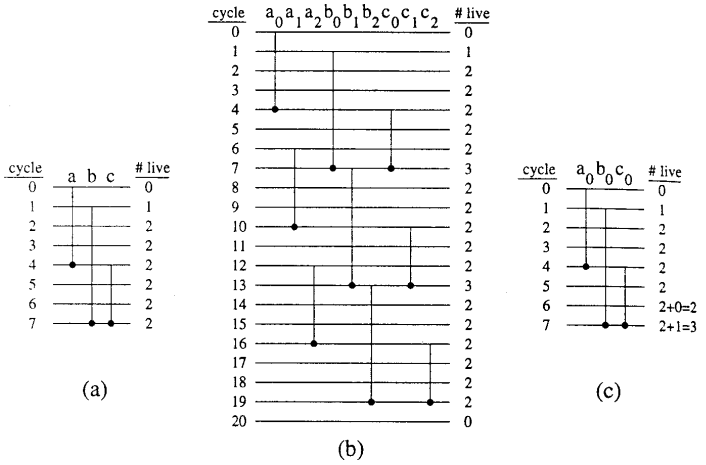


Figure 3-1-1: (a) A linear lifetime chart. (b) The linear lifetime chart explicitly showing 3 iterations of the DSP program assuming the period is N=6. (c) The linear lifetime chart implicitly taking into account the periodicity of the DSP program assuming the period is N=6

The variable is not live during the time it is produce but is live during the clock cycle in which it is consumed.

In the following example, we take 3 variables a, b, c. For instance, a is live during {1,2,3,4}. We have to pay attention to the periodic nature of the DSP program. The effect is shown in the Figure 3-1-1-c. Here, the number of registers require is 2 in the case (a) and 3 in taking into account the periodic nature which is always necessary.

Generally, a lifetime analysis begins with the construction of a lifetime table. In the example of the transpose operation of the 3x3 matrix

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \xrightarrow{transpose} \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

Thus we have the table:

Table 3-1-1: Lifetimes for 3x3 Matrix transpose Operation without latency.

Sample	Tinput	Touput
a	0	0
b	1	3
c	2	6
d	3	1
e	4	4
f	5	7
g	6	2
h	7	5
i	8	8

But in this table we have sometimes Toutput-Tinput negative is unrealistic. Thsu we have to add a latency in order to have Tdiff nonnegative. Thus we have with Tl=4:

Table 3-1-2: Lifetimes for 3x3 Matrix Transpose Operation

Sample	Tinput	Touput without latency	Tdif	Touput	Life Period
a	0	0	0	4	0→4
b	1	3	2	7	1→7
c	2	6	4	10	2→10
d	3	1	-2	5	3→5
e	4	4	0	8	4→8
f	5	7	2	11	5→11
g	6	2	-4	6	6→6
h	7	5	-2	9	7→9
I	8	8	0	12	8→12

With this Table and knowing that the period N is 9 we can draw the lifetime chart. Another way to represent a lifetime chart is to use a circular lifetime chart which help to see the periodic effect.

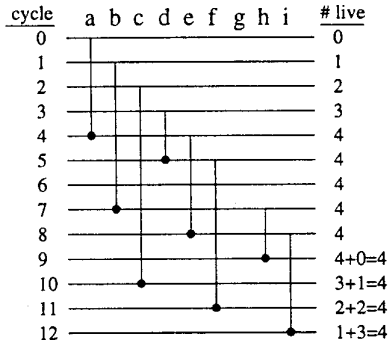


Figure 3-1-2: The linear lifetime chart for the 3X3 matrix transposer with period N=9.

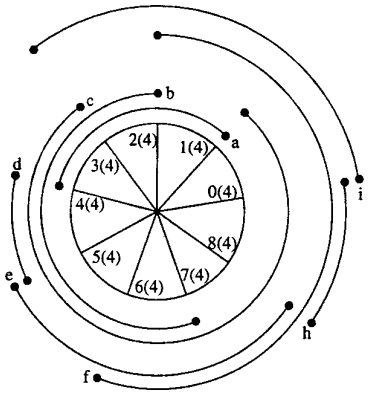


Figure 3-1-3: The circular lifetime chart 3x3 matrix transposer. The corresponding linear lifetime chart is in Fig 3-1-2.

3.2 Data Allocation Using Forward-Backward Register Allocation

Thanks to the lifetime chart, we have obtained the minimum number of registers required. It is now necessary to allocate the data to these registers and the following technique, called Forward-Backward Register Allocation will help us to do so.

The steps to perform the Data Allocation are:

Step 1: Determine the minimum number of registers using the lifetime analysis.

Step 2: Input each at the time step corresponding to the beginning of its lifetime. (for instance: a is the first input variable). If multiple variables are input in a given cycle, these are allocated to initial register and the other variables are allocated to consecutive registers in decreasing order of lifetime.

Step 3: Each variable is allocated in a forward manner until it is dead or it reaches the last register. In forward allocation, if R_i (register i) holds the variable in the current cycle, then R_{i+1} hold the same variable in the next cycle. If R_{i+1} is not available, then the variable is allocated to the first available forward register.

Step 4: Since the allocation is periodic, the allocation. The allocation of the current iteration also repeats itself in subsequent iterations. Thus, if *Ri* is occupied in cycle *l*, then *Ri* would occupy the same variable in cycle $N+l$ => hash the position for *Ri* at time unit $l+N$ for each *j* and *l*.

Step 5: For variables that reach the last R_l and are not yet dead, the remaining period of life is calculated and these variables are allocated in backward manner on a first-come first-served level basis. If several *Ri* are available for the backward allocation:

- try to choose a *Ri* such that backward allocation $R_l \rightarrow Ri$ has already been performed.

- if there are multiple R after the first sort, choose the R with the minimum number of forward registers among all candidates that have a sufficient number of frwd R to complete the allocation of the variable.
After a variable has been allocated backward, allocate it forward until it is dead or it reaches the last register.

Step 6: Repeat steps 4 and 5.

This technique is applied on the previous example: 3x3 matrix transposer and the result table is given below:

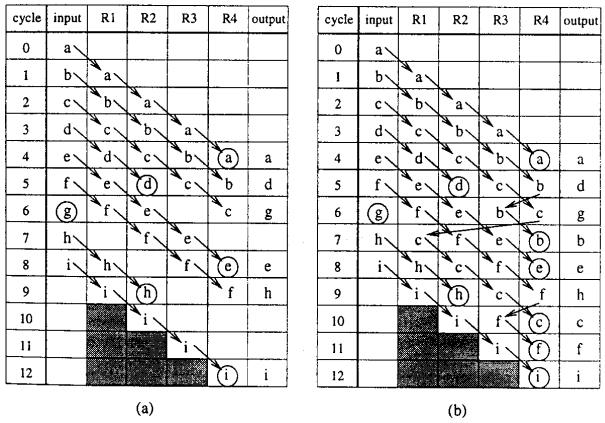


Figure 3-2-1: The allocation table for the 3x3 matrix transposer after steps 1 through 4 of forward-bacward register allocation have been performed. (b) The allocation table after the allocation has been completed.

Analyse of this example:

Step 1: 4 registers are needed according to the lifetime chart.

Step 2: each variable is input at the cycle corresponding to the beginning of its lifetime. NB: No cycles have more than 1 input in this example.

Step 3: allocation in forward manner

Step 4: hashing(cells in grey) to avoid conflicts.

Step 5: figure 3-2-1-b backward allocation is performed.

4. REGISTER MINIMIZATION IN FOLDED ARCHITECTURES

The aim of this section is to apply the previous technique of register minimization described in 3 to a DSP circuit in folded architecture.
The basic procedure is as follows:
1. Perform retiming for folding

2. Write the folding equations

3. Use the folding equation to construct a lifetime table

4. Draw the lifetime chart and determine the required number of registers

5. Perform forward-backward register allocation

6. Draw the folded architecture that uses the minimum number of registers.

4.1 Biquad filter example.

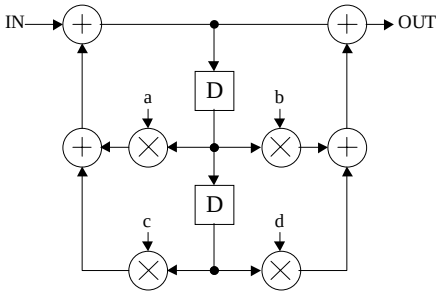


Figure 4-1-1: The allocation table for the 3x3 matrix transposer after steps 1 through

The DFG after retiming for folding can be seen in figure 2-3-1. The folded architecture without any register minimization is shown in figure 2-3-2. This architecture uses 6 registers (the 3 pipelining registers that are internal to the adder and mutliplier are not counted).

As step 1 and 2 have already been performed (folding equations can be seen in section 2-3), we now have to construct the lifetable:

Table 4-1-1: Lifetimes for the Retimed Biquad Filter Shown in Fig 2-3-1

node	$T_{input} \rightarrow T_{output}$
1	4→9
2	-
3	3→3
4	1→1
5	2→2
6	4→4
7	5→6
8	3→4

Note:
In the table, there is one entry for each node in the DFG

T_{input} for the node *U* is equal to : $T_{input} = u+Pu$
(*u* folding order of *U*, *Pu* number of pipelining stages in the functional unit H_U)
T_{input} is the time unit in which the node produces data for the 0^{th} iteration of the DSP program. Ex for the node 1: $T_{input}=3+1=4$

T_{output} for the node U is equal to: $T_{ouput}=u+P_U+max_V\{D_F(U\rightarrow V)\}$
$max_V\{D_F(U\rightarrow V)\}$ represents the longest folded path delay among all edges that begin at the node U. This value of Touput is the latest time that the result of the 0^{th} iteration of the node is used. Ex for node 1: $T_{ouput}=3+1+max\{1,0,2,3,5\}=9$

No latency is required in this DSP program.

The lifetime chart is drawn:

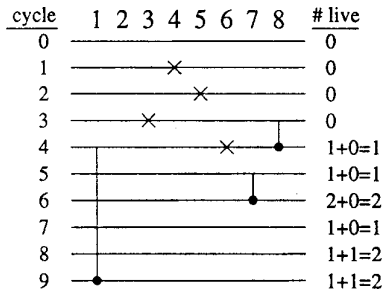


Figure 4-1-2: The lifetime chart corresponding to the lifetime table in Table

Note:
Period N=4.
2 registers are required.
The next step is to perform Forward-Backard register allocation:
Note:
n_i is the output of node i.
Only the variables with nonzero duration are shown. (n_1,n_7 & n_8)

cycle	input	R1	R2	output
0				
1				
2				
3	n_8			
4	n_1	n_8		n_8
5	n_7	n_1		
6		n_7	n_1	n_7
7			n_1	
8			n_1	
9			n_1	n_1

Figure 4-1-3: The allocation table for the folded biquad filter.

The folded biquad filter architecture can now be synthesized.

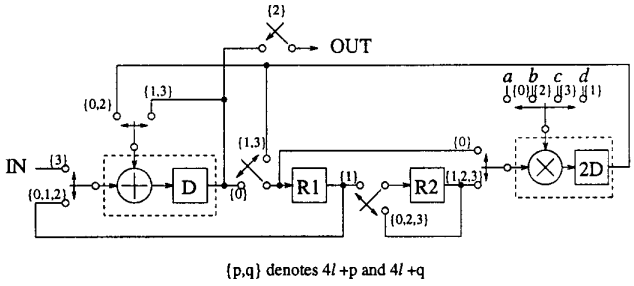


Figure 4-1-4: A folded biquad filter architecture using the minimum number of registers, which is 2.

Note:

For example: *(1→2)* has $D_F(1\rightarrow 2)=1$ delay. This edge starts at the node 1, and after 1 delay the variable n1 is located in he register R_1 in Fig 4-1-3, so there exists an edge from R_1 to the adder at the to time instances *4l+1* because the node has folding order 1. Another example is (1→7) has DF(1→7)=3 delays, and the variable n_1 is in R_2 after 2 delays, so there is an edge from R_2 to the multiplier at the time instances *4l+3* because node 7 has folding order 3.

In this folded architecture the number of registers has been reduced from 6 to 2.

5. FOLDING OF MULTIRATE SYSTEMS

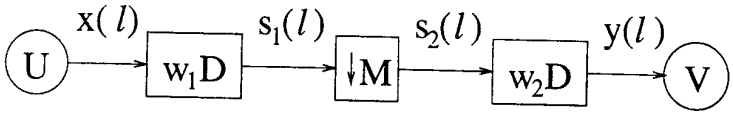
This section deals with folding of multirate systems. Multirate system contains decimators and expanders:

decimator: $y_d(n)=x(Mn)$
expander $y_e(n)=x(n/M)$ if n=kM, 0 otherwise

This functional unit change the data rate.

In the case of a decimator:

Analysis of the DFG below:

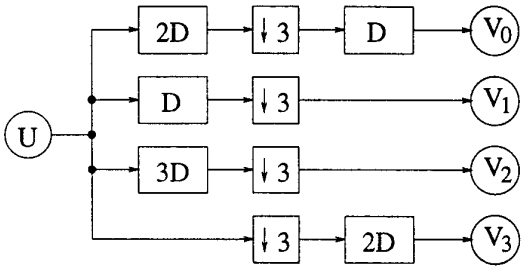


The l-th iteration of the node U is executed at the time unit $Nl+u$ and the l-th iteration of the V is executed at the time unit $N_v l+v=(NM)l+v$.

In the figure above we have the relations between the variables:
$s_1(l)=x(l-w_1)$
$s_2(l)=s_1(M_l)=x(Ml-w_1)$
$y(l)=s_2(l-w_2)=x(M(l-w_2)-w_1)$

and we deduce from these equations
$D_F(U\rightarrow V)= N(Mw_2+w_1)-P_U+v-u.$

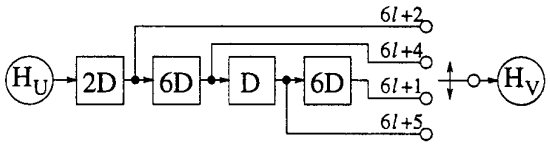
Example:



(a)

N=2
$N_{v0}=N_{v1}=N_{v2}=N_{v3}=6$
The folding orders are u=1, $v_0=1$,$v_2=4$ and $v_3=5$.
And $P_U=1$

The Folding equations are thus:
$D_F(U \rightarrow V_0)=2(3(1)+2)-1+1-1=9$
$D_F(U \rightarrow V_1)=2(3(0)+1)-1+2-1=2$
$D_F(U \rightarrow V_2)=2(3(0)+3)-1+4-1=8$
$D_F(U \rightarrow V_3)=2(3(2)+0)-1+5-1=15$



(b)

In this figure, the number of registers can be reduced using lifetime analysis. Moreover, the equations above to be useful, DF(UV)>=0 must hold given a feasible schedule. Retiming for folding can be used for multirate DFG in a manner similar to that used in single rate DFG.

6. CONCLUSIONS

Folding is a systematic transformation technique for design of time-multipexed architectures. Although folding circuits requires less silicon area, these can be operated at higher speed by exploiting the fine-grain pipelining of the functional units. This result in no net loss in the sampling rate of the system for small folding factors. Folding sets can be designed by any scheduling and allocation techniques. Lifetime analysis can be used to reduce the number of storage units in folded circuit.

REFERENCES

[1] K.K. Parhi, VLSI Digital Signal Processing: FOLDING, Chap 6, J. Wiley & Sons, 1999

[2] K.K. Parhi, "Calculation of minimum number of registers in arbitrary life time chart", IEEE Trans, on circuits and Systems-II, vol. 41,no 6,pp 434-436, June 1994

[3] Denk, T.C.; Parhi, K.K, "Synthesis of folded pipelined architectures for multirate DSP algorithms" VLSI Systems, IEEE Transactions on Volume: 6 4 , Dec. 1998 , Page(s): 595 -607

7. PROBLEM

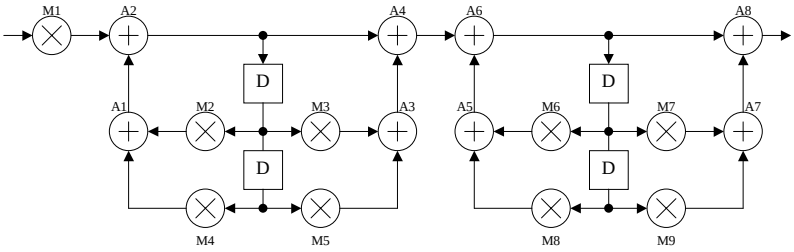


Figure 7-1: The DFG to be folded

a) Perform retiming for folding on the DFG in fig 7-1 so that the folding sets shown below result in nonnegative edge delays in the folded architecture. Assume that the folding factor is N=5, and assume that each multiplier is pipelined by 2 stages and each adder is pipelined by 1 stage. Each operator is clocked with clock period of one u.t. Note that ∅ represents a null operation.

$S_{M1}=\{M_2, M_1, M_3, M_6, M_7\}$

$S_{M2}=\{M_4, \varnothing, M_5, M_8, M_9\}$

$S_{A1}=\{A_4, \varnothing, A_1, A_2, A_3\}$

$S_{A2}=\{A_5, A_6, A_7, A_8, \varnothing\}$

b) Fold the retimed DFG obtained in question a) using the folding sets given in question a).