

FINAL REPORT

on the completion of the contract 170895W0281
**Integrated Collaborative Model in Research and Education with
Emphasis on Small Satellite Technology**

Supported by **the European Office of Aerospace Research and
Development (EOARD), United States Air Force**
223/231 Old Marylebone Road, London NW1 5TH UK

Type of Proposal: *Response to US Government Broad Agency Announcement (BAA)*

Principal Investigator:
Prof. Péter ARATÓ, Dr. Sc.
Head of Department
Department of Process Control
Technical University of Budapest
Műegyetem rkp. 9
H-1521 Budapest
Phone: (361) 463 2699
Fax: (361) 463 2204
e-mail: arato@fsz.bme.hu

Contents

	Part
Detailed description and text book of curriculum on high-level logic synthesis	A
Description and user's manual of the multiuser educational design tool PIPE	B
Standard benchmark set solved in the frame of the curriculum	C
Experiences and statistics of the curriculum	D
Developing a VLSI module generator as a part of a collaborative engineering curriculum	E

Budapest, January, 1996

19990204 017

DTIC QUALITY INSPECTED 4

AG F99-05-0846

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE January 1996	3. REPORT TYPE AND DATES COVERED Final Report	
4. TITLE AND SUBTITLE Integrated Collaborative Model in Research and Education with Emphasis on Small Satellite Technology			5. FUNDING NUMBERS F6170895W0281	
6. AUTHOR(S) Prof. Peter Arato				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Technical University of Budapest Building R, Muegyetem rkp.9 Budapest H-1521 Hungary			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) EOARD PSC 802 BOX 14 FPO 09499-0200			10. SPONSORING/MONITORING AGENCY REPORT NUMBER SPC 95-4025	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (Maximum 200 words) This report results from a contract tasking Technical University of Budapest as follows: Develop an integrated, collaborative model in research and education with emphasis on small satellite technology.				
14. SUBJECT TERMS EOARD			15. NUMBER OF PAGES Too many to count	
			16. PRICE CODE N/A	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

HIGH-LEVEL LOGIC SYNTHESIS

Detailed description and text book of the curriculum
at the Department of Process Control, Faculty of Electrical Engineering and
Informatics, Technical University of Budapest

Edited by
Péter Arató

Authors:
Péter Arató, István Jankovits, Tamás Visegrády

Budapest, 1996

Contents

1. Introduction	1
2. Elementary Operation Graph (EOG)	4
3. Reducing the restarting period	6
3.1. Inserting buffers	6
3.2. Applying multiple copies of operations	8
3.3. Combining the methods	10
3.4. Symbolic representation of recursive loops	11
3.5. Handling of conditional branches	13
4. Synchronisation	15
5. Examples for applying the algorithms RESTART and SYNC	20
5.1. Example 1	20
5.2. Example 2	22
6. Scheduling as arrangement of synchronizing delay effects	26
7. Allocation	30
7.1. Covering of non concurrent operations	30
7.2. Topological cover of operations	35
8. Multiple-process recursive loops	38
8.1. Overlapped (pipelined) utilisation of recursive loops	38
8.2. Loop scheduling	41
8.3. Classification of recursive problems	43
8.4. External synchronisation	45
9. Control principles	48
9.1. Centralized control path	48
9.2. Distributed control path	51
10. Scheduling methods	54
10.1. Stages of scheduling and allocation	55
10.2. Initial allocation	56
10.3. Initial approximation of the optimal solution	59
10.4. Scheduling	61
10.4.1. Scheduling using Integer Linear Programming (ILP)	62
10.4.2. List scheduling	64
10.4.3. Practical applications of list scheduling with hardware constraints	67
10.4.4. Force-directed scheduling	71
10.5. Conditional execution	73
10.5.1. Worst-case model	74
10.5.2. Probability-based model	74
10.5.3. Realisation	75
10.6. Examples	77
References	98
For further reading	99

1. Introduction

The high-speed digital signal processing and most of the real time applications require **special-purpose hardware units** executing a special task or being able to solve a limited problem set. Due to the technological development, the size and the price of such units are being reduced but the speed and the complexity of the executable tasks are increasing. The functional parts of the units are the **data path** consisted of **processors** and the **data connections** between them and the **control part** co-ordinating the data path. One of the ways of increasing the processing speed is the **pipeline mode** enabling to introduce new input data before obtaining the results for the previous ones. The frequency of the data introduction expresses how often the unit can be restarted with new input data. This **restarting period** determines the **throughput** of the unit. The longest time collapsing from the introduction of an input data until receiving the result calculated on it is called the **latency** of the unit. Reducing the restarting period may cause a longer latency. Based on the problem to be solved by the special-purpose unit, defining the processors, the data connections and the control part under several constraints (speed, cost, size, technology, etc.) is the **structure design**. There are many commercial computer-aided design (CAD) tools (VIEWLOGIC, LOG/IC, ABEL, XILINX, CADENCE, etc.) starting with the structure as input and yielding a complete documentation for fabricating the unit (layout for VLSI ASIC, FPGA programs, etc.). In most cases, the input of these tools is a register transfer level (RTL) description of the structure. Among the RTL structural descriptive languages, VHDL is the most wide-spread and standardized. The procedure from the structural description to the realization in silicon is called **silicon compilation** and can be executed by commercial CAD tools called **silicon compilers**.

Obviously many different structures can be designed for a given task or problem set to be solved. Designing an advantageous structure for silicon compilation is called **high-level synthesis (HLS)** which starts with a specification of the problem to be solved by the unit and provides the RTL description of the structure. Based on the initial specification, a **behavioral prescription** is generated firstly which refers to fictive **elementary operations** of the problem to be solved. There are many variations how to split a problem into elementary operations, therefore the effectiveness of the HLS procedure is strongly influenced by this step. Unfortunately, the optimal behavioral prescription as a decomposition cannot be generated without trials and heuristics. The most advantageous formal specification of the behavioral prescription is a **dataflow-like representation** which is easy to be described also by VHDL on its behavioral level. The next step of the HLS is to **schedule** and **synchronize** the elementary operations by a proper control in order to fulfill the throughput requirements without violating the other constraints (available building blocks as hardware resources, technology, etc.). Based on a schedule, the processors can be specified by constructing proper subsets of elementary operations executable by the same processor. This step of the HLS is called **allocation** which covers the elementary operations by a

set of real processors already representing the structural design. The allocation constraints may require identical elementary operations in the same processors, or a limited complexity of the processors, or a regular structure (systolic array), etc. Each of these constraints may need different schedules for a beneficial structure. Therefore, the scheduling and the allocation steps are not independent of each other and trials and heuristics could not be avoided for finding an optimal solution. Each step of the HLS involves NP-complete problems. In this sense, no systematic method can be formulated for a global optimum in the HLS. However, there exist a lot of approaching methods for finding locally optimal or simply beneficial solutions in the steps of the HLS separately. These HLS methods are very important and efficient, since they usually yield a more advantageous structure for the silicon compilation than a structure defined by intuition. After having defined the structure, almost all the freedom of the behavioral prescription is lost. The HLS methods are dedicated to control this freedom-losing step by step in order to provide a beneficial structure for the further design phasis.

Based on the above considerations, Figure I illustrates the main steps of HLS.

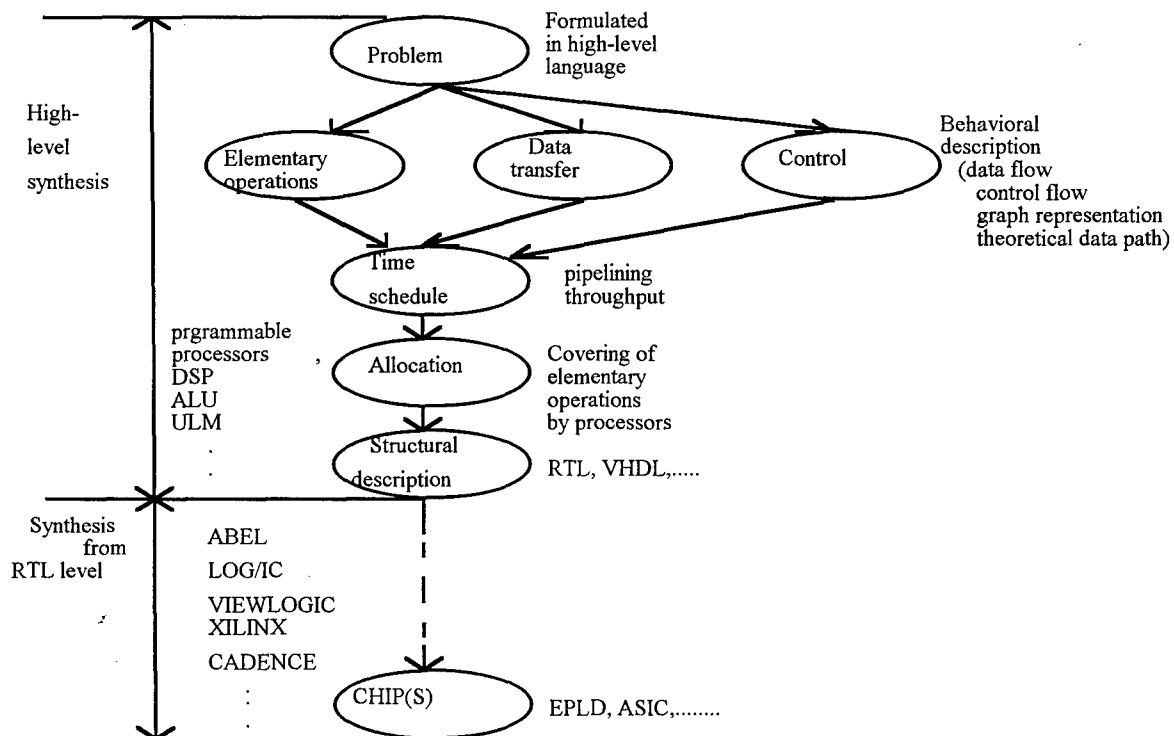


Figure I
The main steps of high-level synthesis

In this book, models and methods are presented for the high-level logic synthesis of pipeline structures. The problem to be solved by the structure is initially split into elementary theoretical operations with arbitrary duration times. The behavioral prescription of this system is based on a dataflow-like representation which provides an easy way to formulate the scheduling and

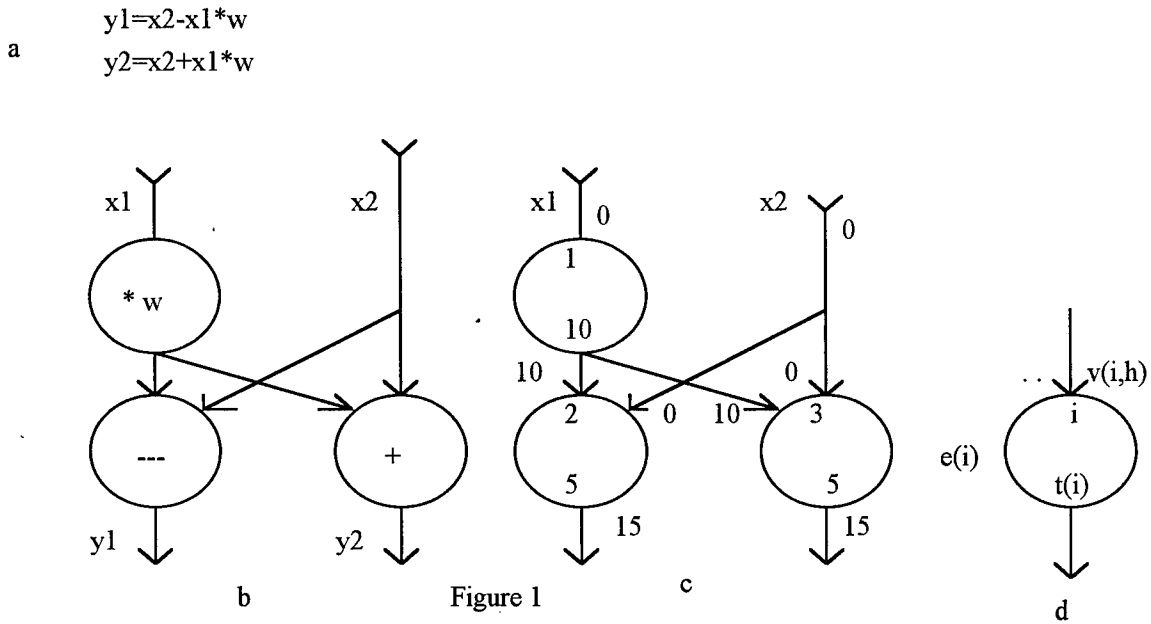
allocation steps of the methods. In the basic method presented firstly, the pipeline mode needs no extra efforts and the method ensures a restarting period which can be given in advance. The mobility and scheduling of the elementary theoretical operations are represented by inserting extra buffer registers into the dataflow-like data path and a structural pipelining can be established by applying extra copies of some operations. The minimal number of buffers to be inserted, the optimal selection of the operational units to be applied in multiple copies and the minimal number of the required copies are the main goals in the first design phase. Based on these results, the second part of the method provides a solution to the resource allocation problem. It is proven that the concurrence of two elementary operations can be considered as a compatibility relation between them. Thus, a proper cover of the non-concurrent operations can represent the hardware resources i. e. the real processors. Calculating the cover, several constraints for the types of processors and the data path structure can be taken into consideration.

Besides the basic model and method, the well-known HLS methods (list scheduling, force-directed scheduling, ILP method) are also presented based on the most relevant references on this field.

In the appendix, the usage of a HLS CAD software tool is illustrated on standard benchmark problems.

2. The Elementary Operation Graph (EOG)

The problem to be solved can be considered as a sequence of elementary operations between the input ($x_1, \dots, x_n$) and the output data ($y_1, \dots, y_m$). The data connections and the elementary operations represent a principal data path for the problem to be solved. The control of this principal data path can be imagined as a centralized counter or simple distributed handshake units (shown later). The pipeline scheduling and allocation are accomplished on the principal data path and the resulting structure also involves the control specification. The principal data path is assumed to be synchronized by a clock signal, which also influences of the elementary operations, i. e. the **duration** or **execution time** is specified by the number of the clock periods between the beginning and the end of the operation.



a: The problem to be solved, b: A data-flow representation, c: The Elementary Operation Graph (EOG), d: Notations for EOG

A simple graph representation of the principal data path is illustrated in Figure 1 for the basic cell of a fast Fourier-transformation algorithm as the problem to be solved. Applying the notation of Figure 1.d, the numbers at the inputs of the elementary operations $e(i)$ of the EOG refer to the points of time, at which the first data arrive on these inputs. For example, $v(2,1)=10$ on the left input of $e(2)$, because the first data arrives from $e(1)$ in the 10-th clock cycle. In this case, $e(1)$ is the predecessor of $e(2)$ (in notation: $e(1) \rightarrow e(2)$) and $t(1)=10$ involves that $e(1)$ provides its first output in the 10-th clock cycle. The elementary operations of the EOG are assumed to have a dataflow-like character [7]:

- a./ $e(i)$ is started only after having finished every $e(j)$, for which $e(j) \rightarrow e(i)$ holds.
- b./ $e(i)$ requires all its input data during the whole duration time $t(i)$.
- c./ $e(i)$ may change its output during the whole duration time $t(i)$.
- d./ $e(i)$ holds its actual output stable until its next start.

For the sake of simplicity, each $e(i)$ in the EOG (each node) may have only one output data (leaving edge) and at most two input data (arriving edge) except the conditional branches (shown later). If the output supplies several inputs; then several edges may represent the same single output.

The latest first output of the whole EOG determines the latency (L) of the principal data path. In Figure 1.c, $L=15$. In a pipeline mode, the second input data of the EOG is introduced earlier than L . In this way, the cyclic restarting with new input data occurs more frequently than the period determined by L . The pipeline mode means that the **restarting period** (R) is shorter than L . Thus, the throughput for input data sequences can be increased depending on the value of R . It is obvious, that there are some limitations for decreasing the value of R , because the duration times of the operations in the EOG and the data connections strongly influence the earliest acceptance of the new data. In the next chapters, a method will be outlined for achieving a desired restarting period by some modifications of the EOG.

3. Reducing the restarting period

3.1. Inserting buffers

Let it be assumed that the EOG does not contain loops. In this case, the EOG can be considered as a simple assembly of independent sequences of operations starting with an operation at the input of the EOG and ending with an operation at the output of the EOG. Let these sequences be called **transfer sequences (TS)**.

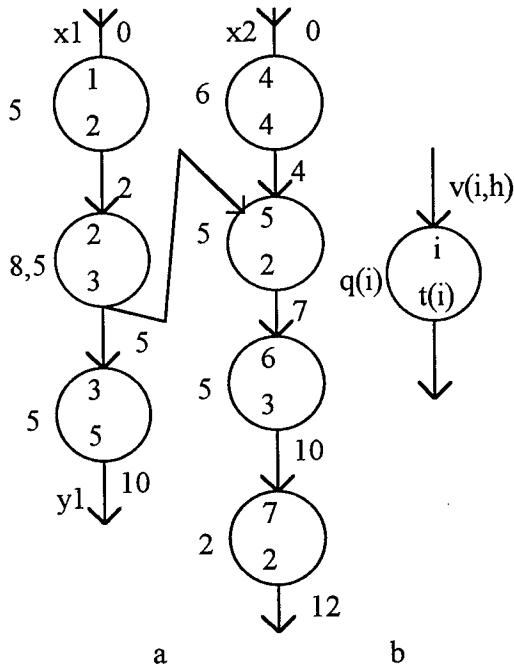


Figure 2

Elementary operation graph for illustrating the transfer and busy time sequences

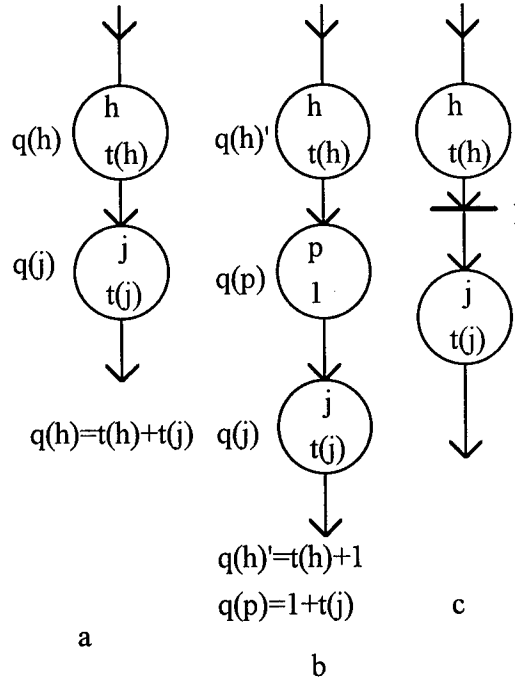


Figure 3

- a: A fragment of a transfer sequence
- b: The new busy times after inserting the extra buffer register
- c: A symbolic notation for the buffer register inserted additionally for reducing the restarting period

The EOG in Figure 2 involves the TS-s, as follows:

$$S(1,1) = e(1), e(2), e(3)$$

$$S(1,2) = e(1), e(2), e(5), e(6), e(7)$$

$$S(4,1) = e(4,1) = e(4), e(5), e(6), e(7) \quad \text{or with a simplified notation:}$$

$$S(1,1) = 1, 2, 3$$

$$S(1,2) = 1, 2, 5, 6, 7$$

$$S(4,1) = 4, 5, 6, 7$$

The notation $S(i,k)$ means the k -th TS beginning with $e(i)$.

Each $S(i,k)$ involves a sequence of duration times $D(i,k)$. For example, $D(4,1)$ belongs to $S(4,1)$:

$$D(4,1)=t(4),t(5),t(6),t(7) \text{ or simply } D(4,1)=4,2,3,2$$

If $e(i) \rightarrow e(j)$ then according to conditions b and c in the previous chapter, $e(i)$ must not be restarted with new input data more frequently than the time period $t(i)+t(j)$ allows it. Otherwise, $e(j)$ could not receive stable input data during its whole duration time. Thus, $e(i)$ can be considered in a busy state during the time domain $q(i)=t(i)+t(j)$. In Figure 2, the values of $q(i)$ are given on the left side of the nodes. Note, that $q(2)$ has two different values depending on the successor operations $e(3)$ or $e(5)$, and for the $e(i)$ -s driving directly the outputs of the EOG, $t(j)=0$ is considered. Thus, to each $S(i,k)$, a busy time sequence $Q(i,k)$ can be ordered. For example:

$$Q(1,1)=q(1),q(2),q(3) \text{ i.e. } 5,8,5$$

$$Q(1,2)=q(1),q(2),q(5),q(6),q(7) \text{ i.e. } 5,5,5,5,2, \text{ where } q(2) \text{ occurs with different values depending on the TS.}$$

If each $S(i,k)$ is considered separately, then the above constraints does not allow to restart it in shorter time periods than the maximal value in $Q(i,k)$. In this sense, the shortest restarting period $\min R(i,k)$ of $S(i,k)$ can be expressed:

$$R(i,k) \geq \max Q(i,k) + 1$$

$\min R(i,k) = \max Q(i,k) + 1$, where +1 stands for an extra clock cycle to properly separate the restarting periods.

According to Figure 2: $\min R(1,1)=9$

$$\min R(1,2)=6$$

$$\min R(4,1)=7$$

It is trivial that the minimal restarting period for the whole EOG is:

$$\min R = \max(\min R(i,k)) = \max(\max Q(i,k) + 1).$$

In Figure 2: $\min R=9$

To reduce the value of $\min R$, additional buffer registers may be inserted into the EOG. The principle of the method is illustrated on a fragment of a TS in Figure 3. By inserting a buffer register as an additional special operation $e(p)$ with $t(p)=1$, the busy time of $e(h)$ can be reduced if $t(j)>1$. Let it be assumed that $\max(Q(i,k))=q(h)$ and $t(h)>1$ and $t(j)>1$ hold before inserting the buffer. In this case, $q(h)'<q(h)$ holds after the insertion of the buffer and so the modified value of $\min R(i,k)$ can be smaller than it was originally. This way of reducing the restarting period by inserting a buffer after $e(h)$ cannot be effective any more if $\max(Q(i,k))=t(h)+1$ has been achieved. If $e(h)$ has no successor i.e. it produces one of the outputs of the whole EOG, then $q(h)=t(h)$ is interpreted [7], and so the buffer insertion after $e(h)$ has no sense. Thus, the minimal value of the restarting period obtainable by buffer insertion is:

$$\min R = \max(\max D(i,k) + 2)$$

To achieve this limit, a buffer register is required after each $e(h)$ having a busy time $q(h)$ greater than $\min R - 1$ and having no successor operation with a duration time 1 excepted the $e(h)$ -s at the

output of the EOG [7],[8]. In Figure 4, the only necessary buffer insertion for the EOG in Figure 2 is illustrated providing $\min R = 5 + 2 = 7$. It is trivial that the latency of the EOG may increase in consequence of inserting buffers, but this is not the case in Figure 4, since the latency is determined by the longest transfer sequence i. e. ending with y_2 which is not affected by the buffer insertion.

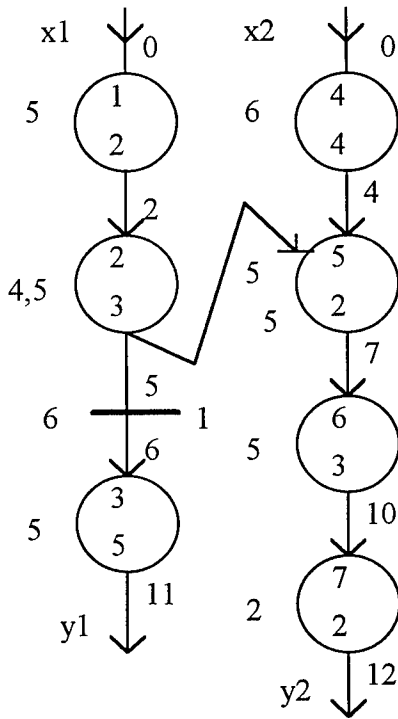


Figure 4

The modified EOG of Figure 2 by inserting a buffer for achieving $\min R = 7$

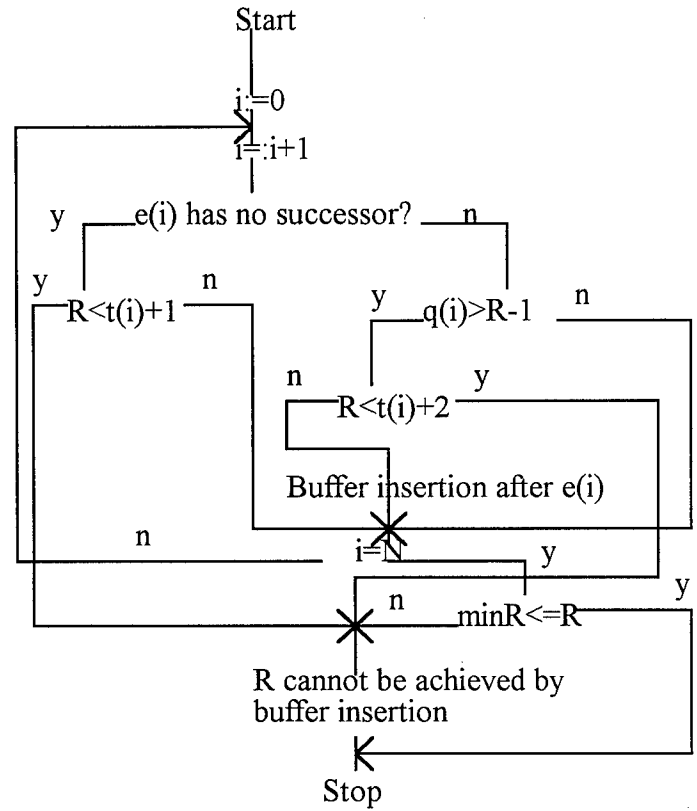


Figure 5

The flow diagram of the algorithm SEPTUN (R denotes the desired restarting period, N is the number of the operations in EOG)

Based on the principle outlined above, a simple algorithm can be formulated for achieving a desired value of the restarting period R by inserting the fewest pieces of buffers. This algorithm is illustrated by the flow diagram in Figure 5 and called SEPTUN, since it is derived from the separate tuning of the transfer sequences as shown above.

3.2. Applying multiple copies of operations

To achieve a shorter restarting period than the value $\min R$ obtainable by buffer insertion, **multiple copies** of some operations must be applied. Let it be assumed that the desired restarting period for the transfer sequence in Figure 6.a is $R = 8$. Applying the algorithm SEPTUN, the buffer insertion

can provide only $\min R=22$ as it is shown in Figure 6.b. It can be seen that the limitation of the further reduction is represented by $e(3)$.

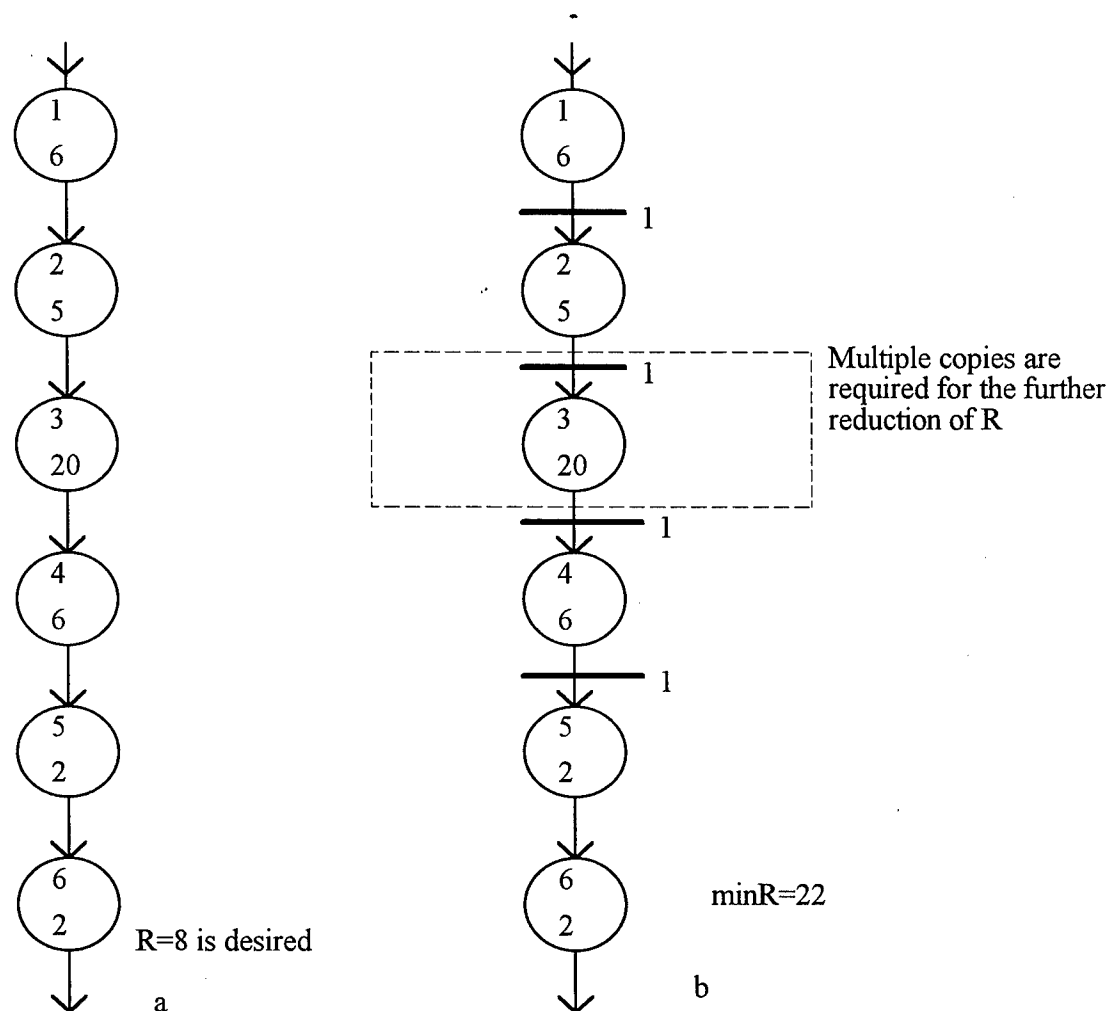


Figure 6
a: A transfer sequence with a desired value of $R=8$
b: The buffer insertion after applying SEPTUN

Applying multiple copies of $e(3)$ and the buffer on its input may allow the further reduction of the restarting period if a proper control is assumed (shown later). Let this situation be examined in Figure 7. If the control of the structure ensures that the input buffers of the copies are restarted with new data periodically after each other, then the arriving times of the first data can be expressed as follows:

$$v(i(2), h) = v(i(1), h) + R$$

$$v(i(3), h) = v(i(1), h) + 2 \cdot R$$

.

.

.

$$v(i(c(i), h) = v(i(i), h) + (c(i) - 1) \cdot R$$

So, the first copy receives the first data at $v(i(1),h)$ and its next data arrives at $v(i(1),h)+c(i)*R$. Obviously, each copy has a time interval of $c(i)*R$ between its two subsequent data. According to the busy time condition

$$c(i)*R \geq t(i)+t(j)+1$$

must hold. Thus, for a desired R , the minimal required value of $c(i)$ can be expressed as:

$$c(i) = \lceil (t(i)+t(j)+1)/R \rceil,$$

where the symbol $\lceil \dots \rceil$ denotes the smallest integer which is greater or equal to the value of the expression within these brackets.

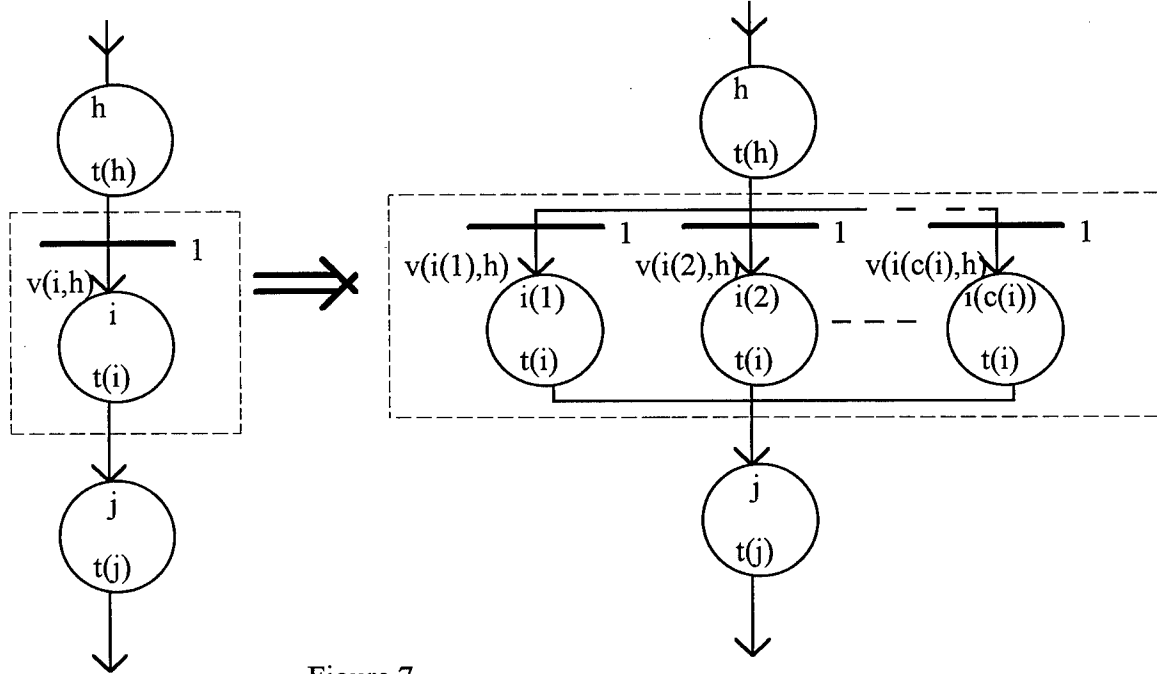


Figure 7

Replacing $e(i)$ by $c(i)$ copies

Note that each copy of $e(i)$ needs an extra buffer at its input, since otherwise the reduced restarting period allowable only by applying multiple copies would hurt the busy time condition assumed for the elementary operations. Without these input buffers, a new data would change the input of a previous copy too early. A proper control (shown later) is to be applied for enabling only actual input buffer each restarting period.

3.3 Combining the methods

If the algorithm SEPTUN inserts a buffer between $e(i)$ and $e(j)$, then for further reduction of the restarting period, at least

$$c(i) = \lceil (t(i)+2)/R \rceil.$$

copies of $e(i)$ are required to achieve the desired value of R . It is trivial that this buffer can be left out without increasing the minimal number of copies if

$$\lceil (t(i)+2)/R \rceil = \lceil (t(i)+t(j)+1)/R \rceil \text{ holds.}$$

Considering again the transfer sequence in Figure 6, the desired value $R=8$ requires at least

$$[(20+1+1)/8]=3$$

copies of $e(3)$. In this case $[(20+4+1/8)] > [(20+2)/8]$, therefore the buffer register between $e(3)$ and $e(4)$ cannot be neglected without increasing the required number of copies for $e(3)$.

It may occur that a buffer inserted by SEPTUN is connected directly to multiplied operations only. Obviously, such buffers can always be neglected, since the unavoidable input buffers of the multiple copies can take over their tasks.

Generally, it can be assumed that inserting buffers is not as expensive as applying multiple copies. Therefore, the reduction of the pipeline restarting period should be started by the SEPTUN algorithm and applying multiple copies is only the next step, if the desired value of R cannot be achieved by SEPTUN alone. In this case, the optimality in realizing a desired value of the restarting period can be formulated, as to insert the minimal number of buffer registers and applying as few copies as possible. For this aim, SEPTUN can easily be modified and completed, as shown in Figure 8 by the flow diagram of the algorithm called RESTART. It can be seen that preserving the buffers inserted by the SEPTUN algorithm, the minimal value of the busy time cannot be smaller than 2 in an EOG without a recursive loop (shown later), since an inserted buffer cannot represent a shorter busy time than 2. Thus, the shortest restarting period obtainable by the algorithm RESTART would be 3 without the last step before the lower stop label. This step, however, neglects all of the buffers inserted by SEPTUN, if each operation becomes to be multiplied. This is always the case if $R=1$ is desired in a loopless EOG, i. e. each operation is replaced by $c(i)=[(t(i)+t(j)+1)/1]=q(i)+1$ copies.

3.4. Symbolic representation of recursive loops

Recursive loops in an EOG require special handling, because the nature of a loop represents the lower limit in reducing the pipeline restarting period. In Figure 9, a symbolic loop-representation is illustrated. It is trivial that the first operation of the loop can start to process new data only after the result obtained with previous data has already arrived from the last operation of the loop. Therefore, the duration of the loop is the sum of all durations inside the loop. In Figure 9:

$$T(i)=t(k)+t(l)+t(m)+t(n).$$

Prescribing an extra clock cycle for the proper separation, the loop limits the restarting period to $T(i)+1$. In case of more than one loops in an EOG

$$\min R \geq \max T(i)+1$$

holds. In the symbolic representation, the loop $e(r(i))$ is considered as a single operation during the calculation of $\min R$, but it is assumed to be divided into four parts $e(r(i))/1$, $e(r(i))/2$, $e(r(i))/3$, $e(r(i))/4$. Following from the recursive character, the pipeline restarting period of these parts cannot be made shorter by inserting buffers between them or by multiplying them.

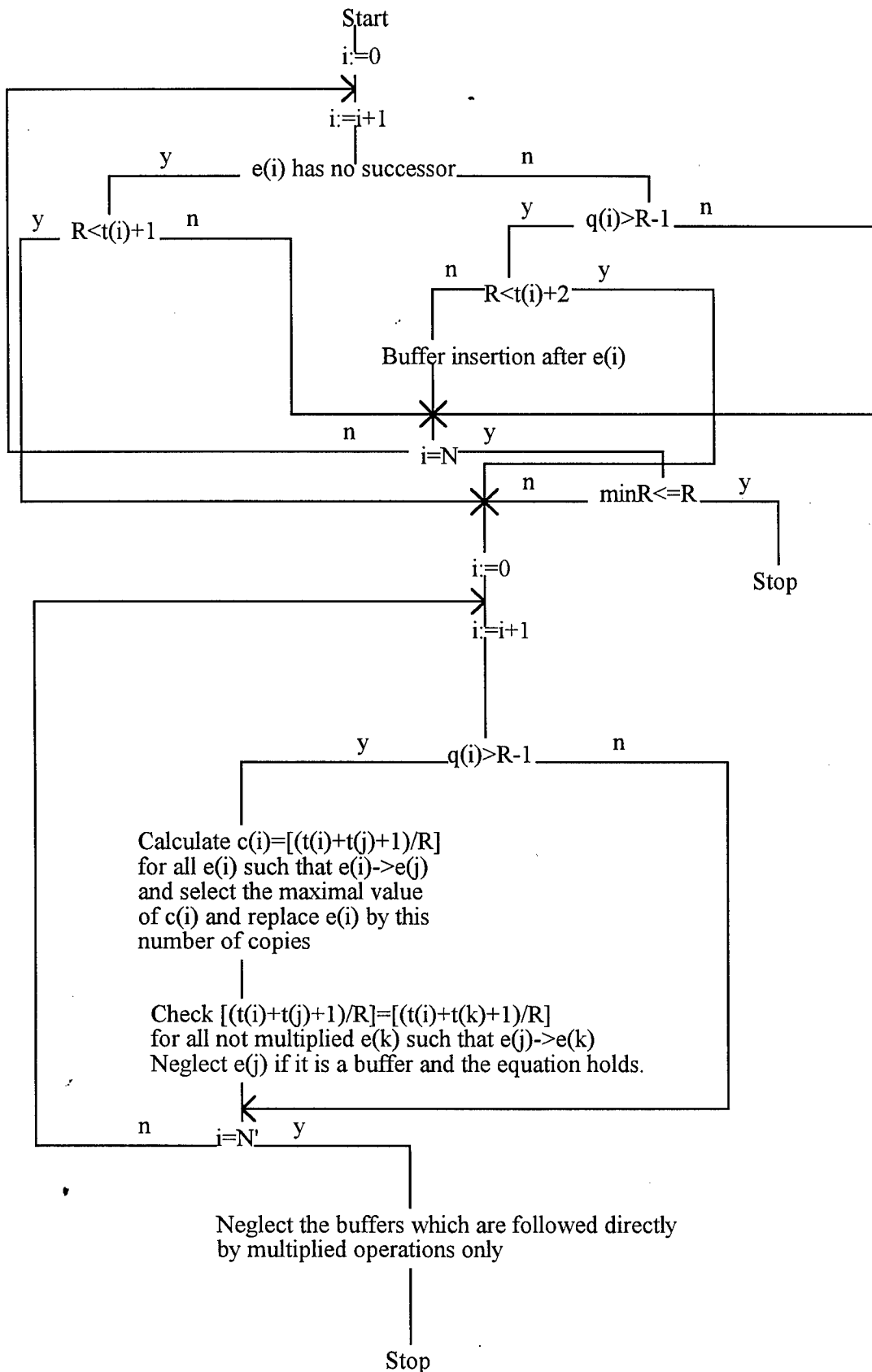


Figure 8
The flow diagram of the algorithm RESTART. (R denotes the desired restarting period, N is the number of operations in EOG before the buffer insertion and N' is it after that).

Emphasizing this fact, the loop parts are symbolized by squares instead of circles as shown in Figure 9. Obviously, a recursive loop cannot consist of a single operation only, since a direct feed-back from the output to the input of the same operation would hurt the constraint requiring stable input data during the whole duration time. This conflict can be eliminated in EOG by establishing the feed-back of the single operation through a buffer register inserted at its output.

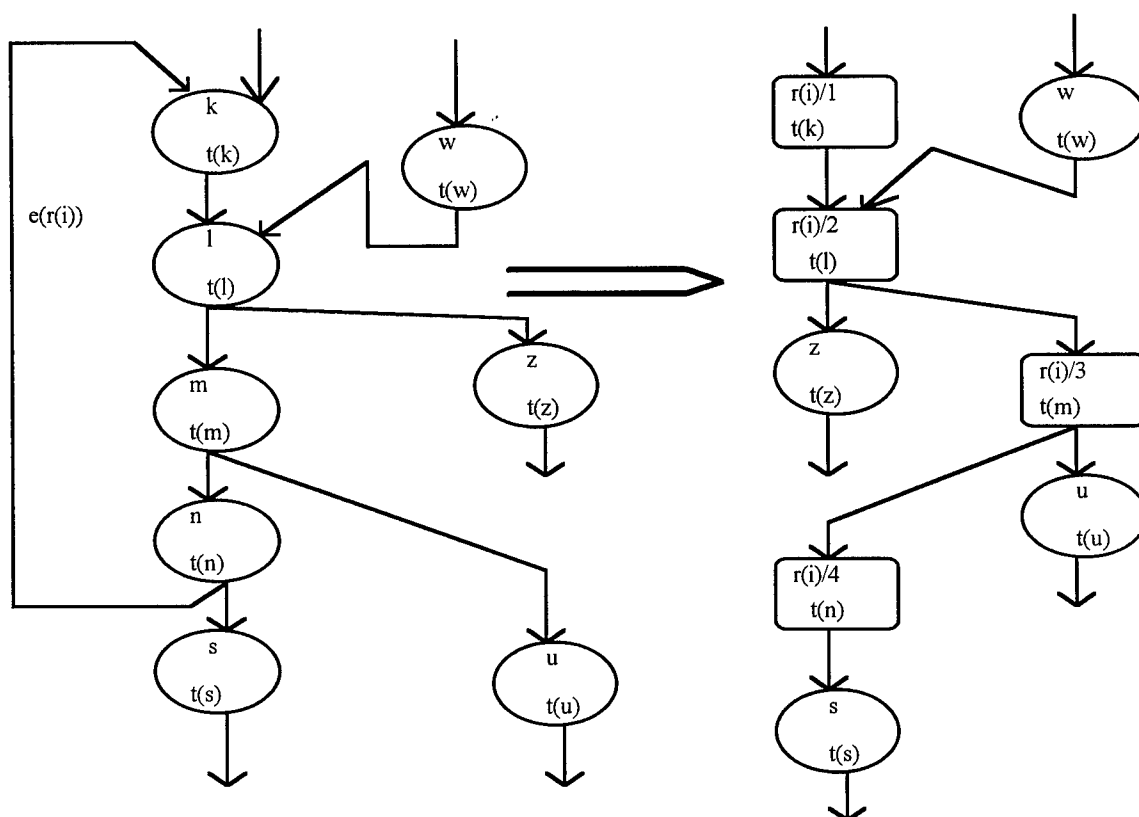


Figure 9

Symbolic representation of a recursive loop

3.5. Handling of conditional branches

Conditional branches in an EOG do not need special handling, since the condition checking can be considered as a special operation, the result of which enables the next operations receiving the conditions formally as normal additional data inputs. In this way, alternative sections are generated in the transfer sequences without any changes in the formal handling of the EOG except during the allocation procedure (shown later). In Figure 10, the symbolic representation of a simple conditional branch is illustrated. The condition checking is a comparator in this case and the alternative sections are closed by a multiplexer (MUX) operation which is also controlled by the output of the condition checking operation. The same problem could be solved by other EOG structures as well. For example, if the condition checking output were connected only to the MUX operation, then the calculation would be also correct, but the two sections of operations would not be alternative any more, which is not advantageous during the allocation procedure

(shown later). Obviously, it is always unavoidable to insert special operations with three inputs into the EOG for the above simple formal handling of conditional branches.

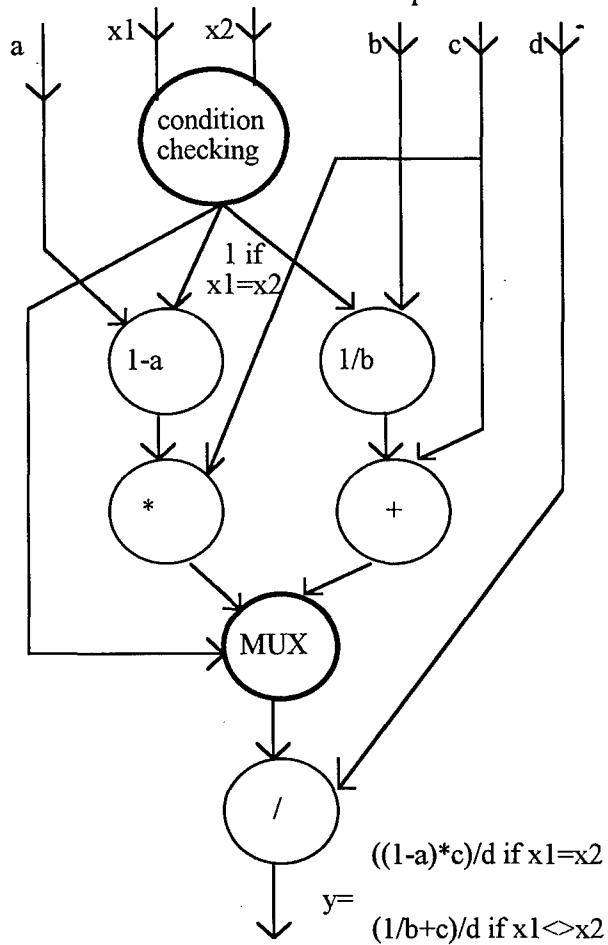


Figure 10
Symbolic representation of a conditional branch

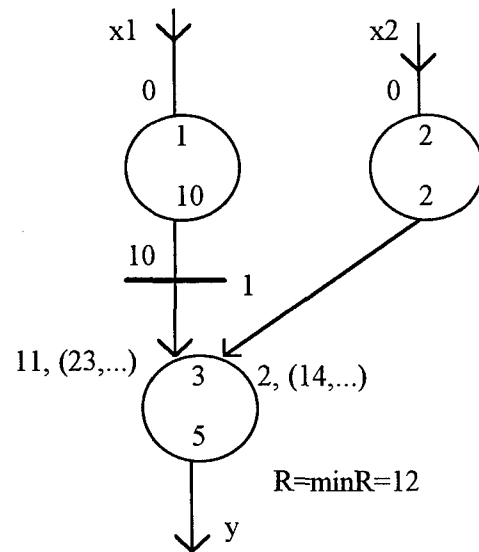


Figure 11
Illustration of the synchronization problem

4. Synchronisation

The dataflow-like character of the EOG involves that an operation can be started by the arrival of all of its input data. An other assumption for the operations is that they need all of their input data to be unchanged during the whole duration. This condition is not met automatically even for an initial EOG, and the modifying effects of the algorithm RESTART may also cause conflicts in this sense. The problem is illustrated in Figure 11. For the inputs of $e(3)$, the data arrival times are: $v(3,2)=2$ and $v(3,4)=11$, where the number 4 refers to the buffer as $e(4)$ inserted between $e(1)$ and $e(3)$. If the pipeline restarting period $R=\min R=12$, then the second data arrives at 23 and 14 respectively. It means that $e(3)$ senses the next input change after $v(3,4)=11$ at 14. Therefore, the input data of $e(3)$ are unchanged between 11 and 14 only in the clock cycles 12, 13, i.e. for 2 clock cycles instead of the required $t(3)=5$. This synchronization problem may always arise, if an operation has two or more inputs.

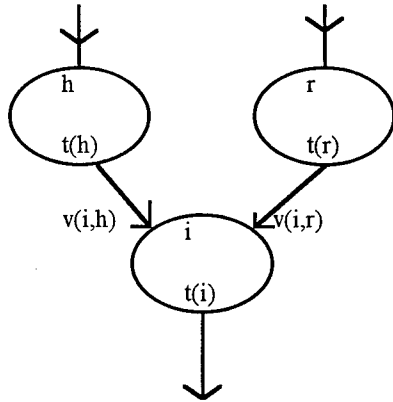


Figure 12
The general situation for the synchronization

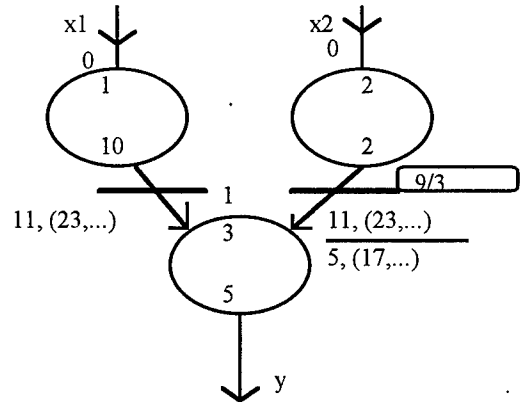


Figure 13
A symbolic representation of inserting buffers into the EOG in Figure 11

In Figure 12, let it be assumed that

$$b(i) = \max(v(i,h), v(i,r), \dots) \text{ and } b(i) = v(i,r)$$

In this case, $b(i)$ or $v(i,r)$ can be called the earliest possible starting time of $e(i)$.

For each $e(h) \rightarrow e(i)$, a time difference $z(i,h) = b(i) - v(i,h)$ can be introduced. The second change of the output of $e(h)$ occurs at $v(i,h) + R - t(h)$. Therefore, the time interval, in which both input data of $e(i)$ are unchanged:

$$v(i,h) + R - t(h) - b(i) \text{ or substituting } b(i) = z(i,h) + v(i,h):$$

$$R - z(i,h) - t(h)$$

Since the input data of $e(i)$ must be stable simultaneously for at least $t(i)$, the inequality

$$R - z(i,h) - t(h) \geq t(i)$$

must hold for the proper operation. If it is not the case, then an extra delay effect $p(i,h)$ is required between $e(h)$ and $e(i)$. The minimal value of this delay is

$$\min p(i,h) = z(i,h) + t(h) + t(i) - R.$$

The maximal allowable value of the delay effect is $\max p(i,h) = z(i,h)$, since a longer delay would increase the latency. Thus, the allowable interval for the delay effect is:

$$z(i,h) + t(h) + t(i) - R \leq p(i,h) \leq z(i,h).$$

If $e(i)$ is a copy of a multiplied operation then

$$\min p(i,h) = z(i,h) + 1 + t(i) - c(i) * R,$$

since $e(h)$ is always a buffer register with $t(h)=1$ and only each $c(i)$ -th restarting period can cause changes at the input of $e(i)$.

Obviously, a negative or zero result for $\min p(i,h)$ means that no synchronization problem arises on this input even without extra extra delay effects.

If the data path between $e(h)$ and $e(i)$ is inside a recursive loop and the other input of $e(i)$ from $e(r)$ is outside the loop and $b(i)=v(i,r)$, then let $e(i)$ be called a **loop-border operation**. In this case, the possible extra delay effect between $e(h)$ and $e(i)$ would increase the total duration of the loop. To avoid this drawback, the delay effect must be transferred to the input of the loop. The first operation of a recursive loop, i. e. the input of the loop is always free from synchronizing problem, since the first feed-back input data of this operation is an undefined initial value and so its arriving time can be considered as to be the same as the arriving time of the other input. Thus, the second and the further data cannot cause any conflicts because the restarting period is always longer than the duration of the loop.

If the extra synchronizing delay effects were realized as delay operations with durations corresponding to the required values of the delay effects, then the restarting period calculated by RESTART may be changed, since the new operations would produce new busy time sequences. A possible way of avoiding this effect is to realize the extra delays by connecting after each other as many buffer registers as the required value of the delay effect. Thus, the required number of buffers is $p(i,h)$. In this way, the new busy time sequences cannot influence the restarting period calculated by the algorithm RESTART, since the duration of each new operation is 1. In this case, $t(h)=1$ must be replaced in $\min p(i,h)$, because the immediate predecessor of $e(i)$ is always at least one buffer register, if the extra delay effect is required. Calculating $\min p(i,h)$ with this assumption, a negative or zero result does not always mean now that the synchronization problem would not occur without extra delay effect, since at least a single buffer register has to be inserted as the immediate predecessor of $e(i)$. Otherwise, $t(h)=1$ would not be allowed during the calculation of $\min p(i,h)$ and the result with the original $t(h)$ value may be positive indicating the synchronisation problem.

In Figure 11, the interval for $p(3,2)$ is as follows:

$$\max p(3,2) = z(3,2) = 9$$

$$\min p(3,2) = z(3,2) + 1 + t(3) - 12 = 9 + 1 + 5 - 12 = 3$$

$$9 \geq p(3,2) \geq 3$$

A symbolic representation of the upper and lower bounds of the delay effect required for the synchronisation is shown in Figure 13.

If $e(i)$ is a loop-border operation, then a single buffer register inserted as the immediate predecessor of $e(i)$ would increase the loop duration by 1, but it would allow $t(h)=1$ for the calculation of $\min p(i,h)$. Thus, the minimal required number of buffer register at the loop input is $\min p(i,h)-1$, the negative or zero value of which indicates that no buffer register is needed at the loop input only the single one inside the loop. Generally, allowing a longer loop duration, a considerable reduction may be obtained in $\min p(i,h)$ and so in the number of buffer registers at the loop input. Obviously, this solution can be applied only then, when the longer loop duration does not prevent achieving the desired restarting period for the whole EOG.

It is trivial that the inputs arriving from different conditional branches do not need any synchronisation between each other, since they never can be active in the same restarting period. (For example, the upper inputs of the MUX operation in Figure 10).

The algorithm SYNC based on the above considerations is summarized by the flow diagram in Figure 14. Besides the calculation of the intervals for $p(i,h)$, the buffer representation of the required delay effects is also illustrated. The meaning of the ASAP and ALAP constraints will be discussed later in the chapter outlining the scheduling procedures. The only situation which is not illustrated by the flow diagram is shown in Figure 15, assuming that more than one operations inside a common loop require synchronization. For example, it may happen that calculating separately the delay effects $p(i,h)$ and $p(m,j)$, even the inequality $\min p(m,j) > z(i,h)$ would hold, if the representing buffer registers were placed before the loop. Thus, a new synchronization situation occurs for $e(i)$, if its input from $e(h)$ is delayed to such an extent that its input from $e(r)$ requires synchronizing delay effects. Therefore, a repeated execution of the algorithm SYNC cannot be avoided to overcome this difficulty. After the first run, the new arriving times are to be calculated assuming the delay effect $\max(p(i,h), p(m,j))$ obtained to the loop input by this first run. Starting with these new arriving times, the second run of SYNC always yields a correct synchronization.

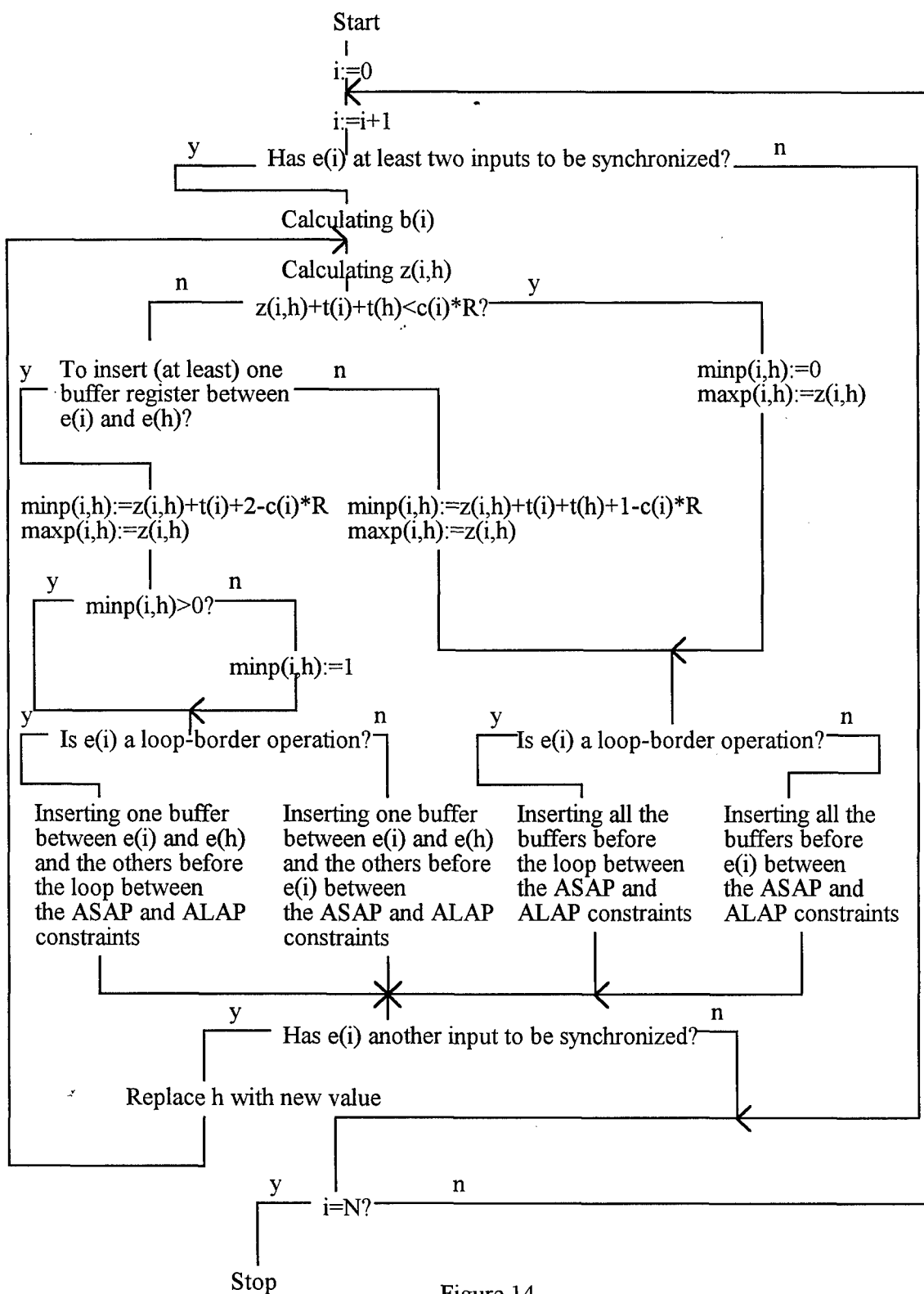


Figure 14
The flow diagram of the algorithm SYNC
(N is the number of the operations in EOG).

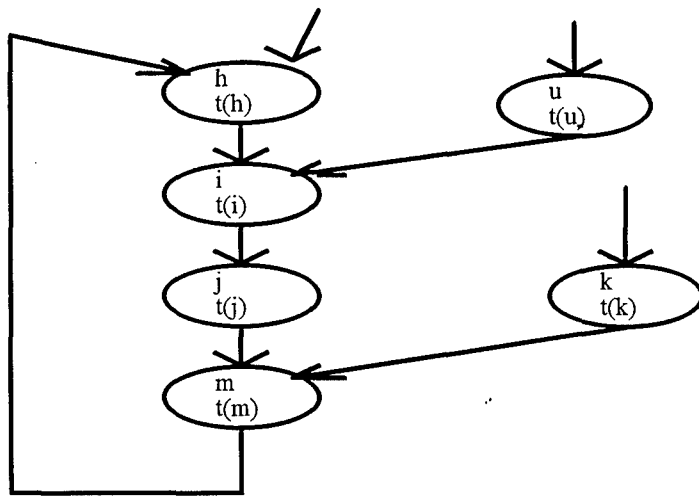


Figure 15
Illustration for the synchronisation problem
if more operation in a common loop need delay effects

5. Examples for applying the algorithms RESTART and SYNC

Example 1

The digital convolution algorithm is to be realized for 3 stages:

$$y(i) = w_1 * x(i-2) + w_2 * x(i-1) + w_3 * x(i),$$

where $y(i)$ denotes the actual output (result),

w_1, w_2, w_3 are the constant weights,

$x(i)$ denotes the actual input data,

$x(i-1)$ and $x(i-2)$ stand for the input data received one and two restart earlier, respectively.

The first step is to specify the elementary operations to be applied. Since inputs from the earlier restarts are to be preserved, a shift register of 2 bits are convenient to use for each data bit. The serial inputs receive the new data bits and the old data bits are obtainable from the output of the second register bit stage providing that a single shift step are executed by each restart. Applying multipliers and adders as further elementary operations, the EOG is shown in Figure 16. The duration times are assumed as follows:

$e(1), e(2), e(3)$ are multipliers with $t(1)=t(2)=t(3)=20$,

$e(4), e(5)$ are adders with $t(4)=t(5)=10$,

$e(6), e(7)$ are shift registers with $t(6)=t(7)=1$.

Since w_1, w_2 and w_3 are constants, each multiplier can be considered as having only one data input.

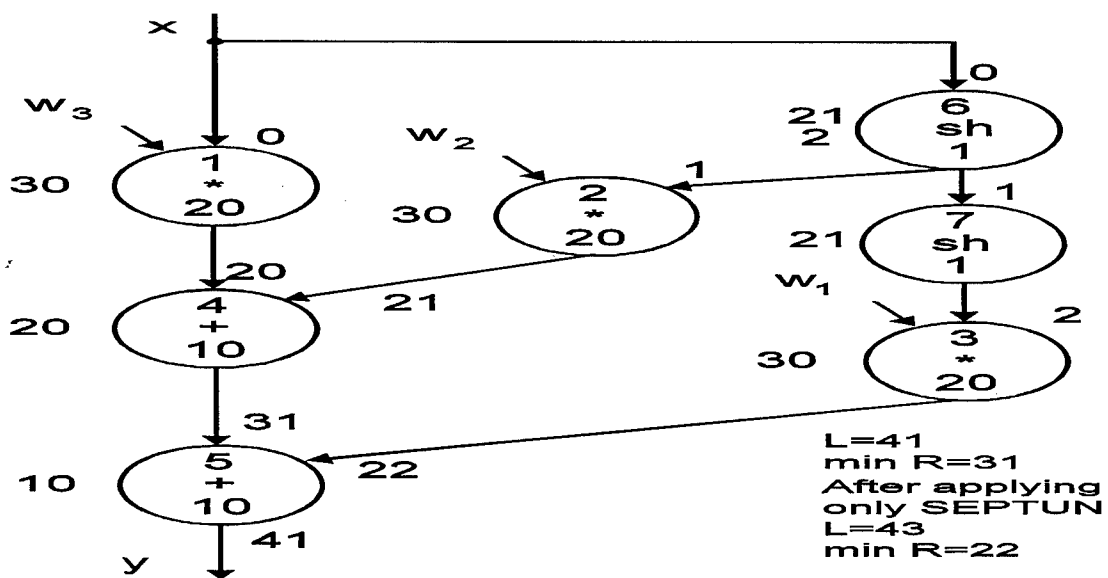


Figure 16

The EOG for the digital convolution algorithm

Following from the algorithm, the EOG can provide the first valid output only at the third start (Restart2):

Start (Restart0): $w3*x(1)+w2*?+w1*?=?$

Restart1: $w3*x(2)+w2*x(1)+w1*?=?$

Restart2: $w3*x(3)+w2*x(2)+w1*x(1)=y(3)$

Restart3: $w3*x(4)+w2*x(3)+w1*x(2)=y(4)$

etc.

Without any modifications, the EOG allows $\min R=31$, after applying the algorithm SEPTUN, $\min R=22$ could be achieved. For the further reduction of the restarting period, the algorithm RESTART must be applied.

Let the desired restarting period be $R=5$. The calculations according to RESTART are as follows: Single buffer registers are to be inserted after each elementary operation except $e(5)$ and multiple copies are required for $e(1)$, $e(2)$, $e(3)$, $e(4)$, $e(5)$. Each inserted buffer would be followed directly only by a multiplied operation with unavoidable input buffers, therefore each inserted buffer can be neglected in this case according to the algorithm RESTART. The output of each multiplied operation is connected to the input buffers of an other multiplied operation, therefore $t(h)=1$ is to be applied in each expression for the minimal number of copies:

$$c(1)=c(2)=c(3)=\lceil(20+1+1)/5\rceil=5$$

$$c(4)=\lceil(10+1+1)/5\rceil=3$$

$$c(5)=\lceil(10+1+0)/5\rceil=3$$

The modified EOG is illustrated in Figure 17. The synchronizing delay effects are to be calculated only for $e(4)$ and $e(5)$:

$$z(4,1)=1$$

$$1 \geq p(4,1) \geq 1+1+10-3*5=-3$$

$$z(5,3)=10$$

$$10 \geq p(5,3) \geq 10+1+10-3*5=6$$

It is trivial that the negative result for $\min p(4,1)$ means that no synchronization problem can arise even without any delay effects on this input. Therefore, a negative value for $\min p(i,h)$ is to be considered as zero in this case, since $e(4)$ is a multiple operation and so the necessary input buffers represent $t(h)=1$ for each copy. Thus, the delay effects calculated for $e(4)$ and $e(5)$ can be symbolized in Figure 17 preceding the input buffers of the copies.

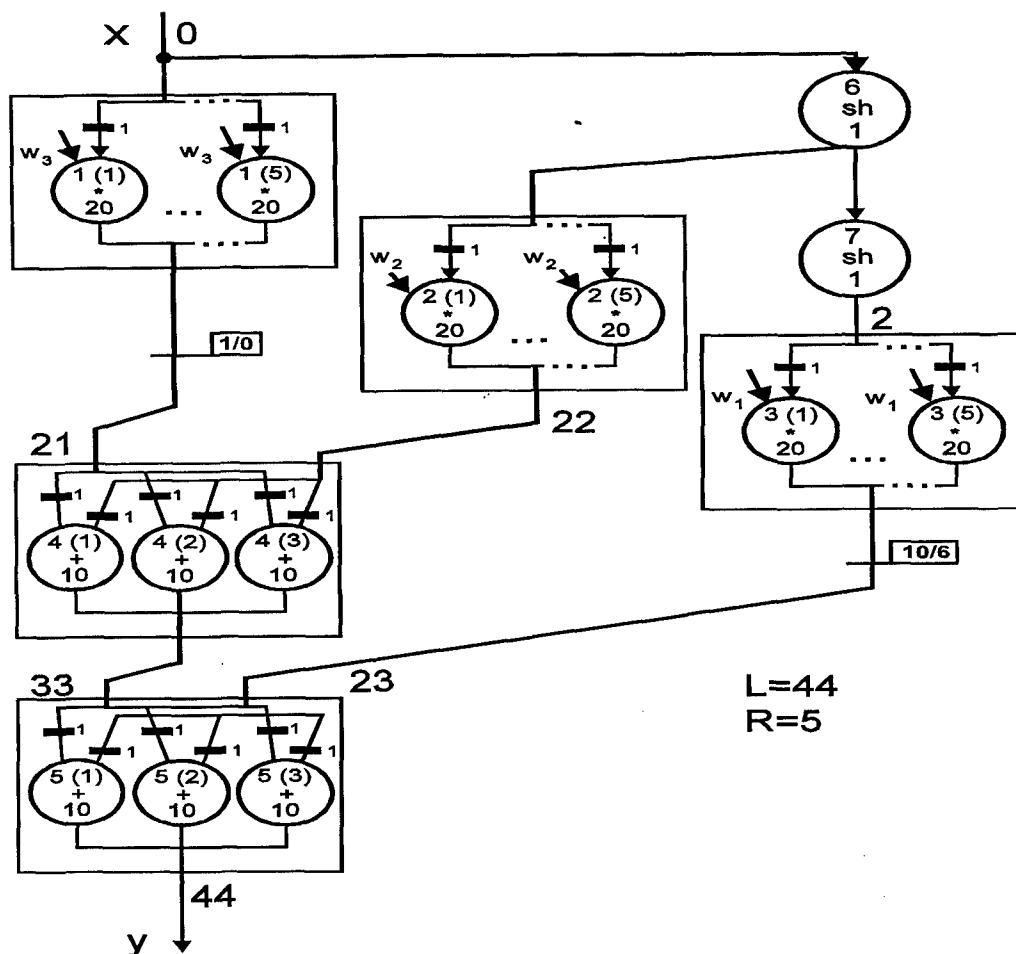


Figure 17

The modified EOG of Figure 16 after applying the algorithm RESTART for $R=5$

Example 2

The problem to be solved is to calculate the expression:

$$y(i+1) = y(i) + x1(i) * x2(i) + \text{SQRT}((x3(i) * x4(i))),$$

where $y(i+1)$ is the actual output (result),

$y(i)$ is the output obtained at the previous restart

$x1(i)$, $x2(i)$, $x3(i)$, $x4(i)$ are the actual input data.

Applying adders, multipliers and SQRT operations as elementary operations, a possible EOG is shown in Figure 18. Let the duration times be assumed as follows:

$e(1)$, $e(2)$ are multipliers with $t(1)=t(2)=20$,

$e(3)$ is the SQRT operation with $t(3)=20$,

$e(4)$, $e(5)$ are adders with $t(4)=t(5)=10$.

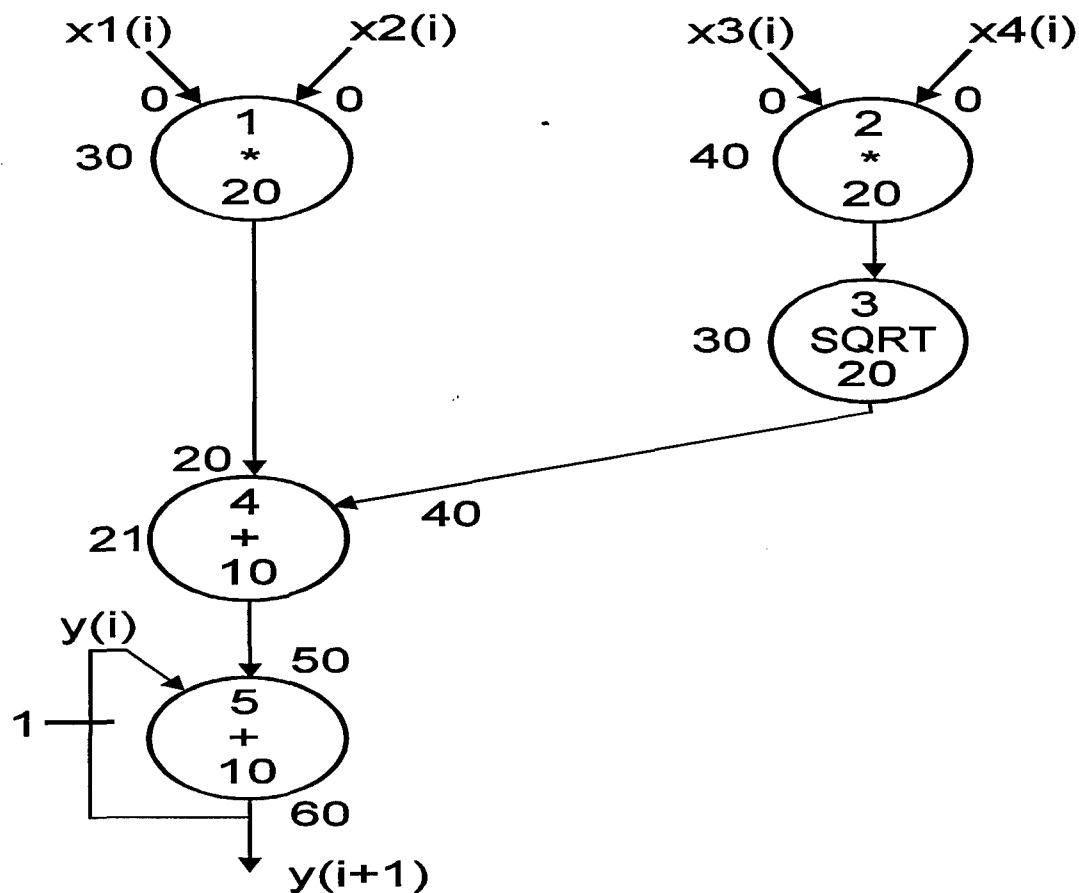


Figure 18

The EOG for calculating the expression $y(i+1)=y(i)+x1(i)*x2(i)+SQRT((x3(i)*x4(i)))$

The buffer register at the output of $e(5)$ is unavoidable, since otherwise $e(5)$ would form the loop alone, which is not allowed in EOG.

Without any modifications, the EOG allows $\min R=41$. Applying the algorithm RESTART only $\min R=12$ can be achieved because of the recursive loop. The symbolic representation is shown in Figure 19. The calculations are as follows:

Single buffer registers are to be inserted after $e(1)$, $e(2)$, $e(3)$, $e(4)$ and multiple copies are needed for $e(1)$, $e(2)$, $e(3)$ to achieve $\min R=12$.

$$c(1)=c(2)=c(3)=\lceil (20+1+1)/12 \rceil = 2$$

No inserted buffers can be neglected at the output of the multiplied operations. The modified EOG is shown in Figure 20. The synchronizing delay effects are to be calculated only for $e(4)$:

$$21 \geq p(1,4) \geq 21+10+1-12=20.$$

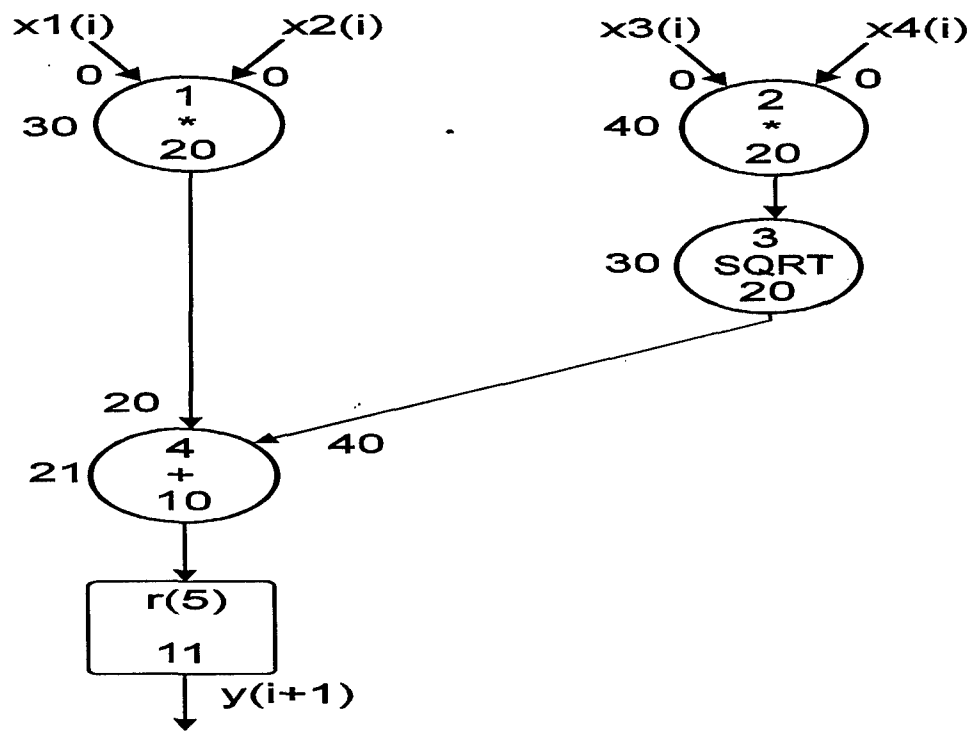


Figure 19

Symbolic representation of the recursive loop in Figure 18

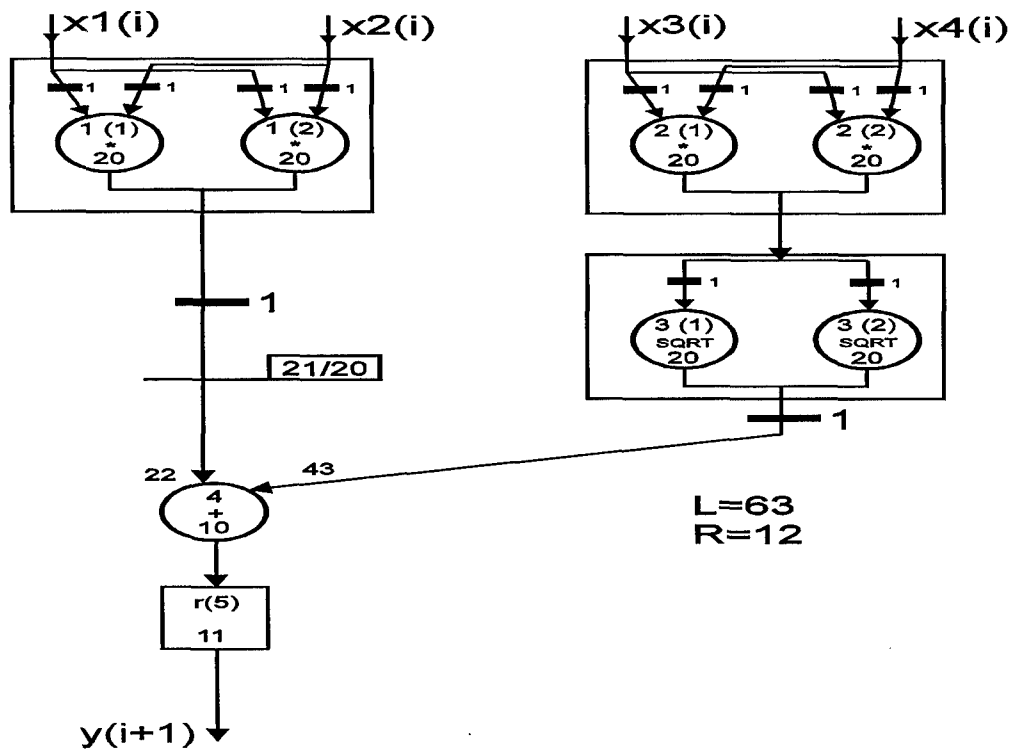


Figure 20

The modified EOG of Figure 19 after applying the algorithm RESTART for $R=12$

An alternative initial EOG for the the problem to be solved is illustrated in Figure 21. In this case, the duration of the recursive loop is 20, therefore no shorter restarting period would be possible than $\min R=21$. Obviously, this solution for the EOG is not advantageous. The general rule is that the duration of recursive loops should be kept as short as possible.

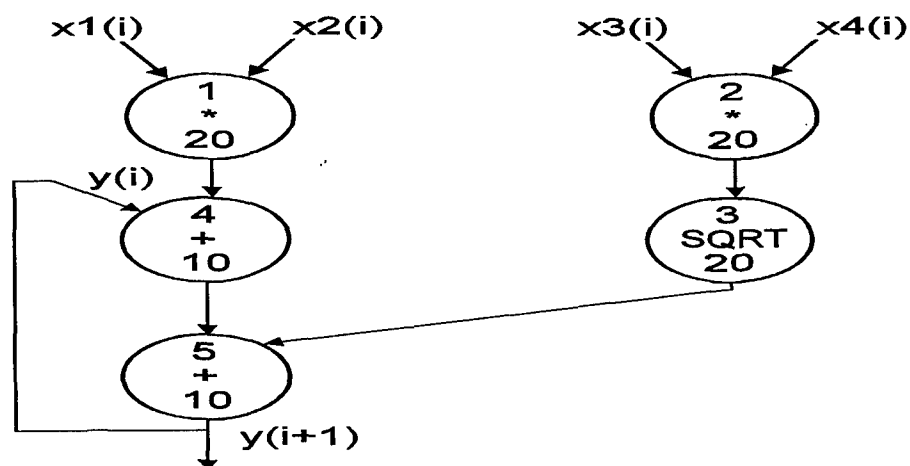


Figure 21

An alternative EOG for the problem in Figure 18

6. Scheduling as arrangement of synchronizing delay effects

The above buffer representation of the synchronizing delay effects can be used advantageously during the scheduling procedures, because the buffers can be considered as delay units movable separately along certain parts of the data path without affecting the pipeline restarting period, latency or synchronization. In this way, all possible situations can be simulated for the starting times of the operations, since the arrangement of the synchronizing buffers determines the starting time of an operation inside its allowed **mobility** domain. The data dependency determined by the EOG, the restarting period and the latency are the constraints for the allowed mobility domain. Two extreme cases can be defined as constraints for the mobility domain: each operation is started as soon as possible (ASAP schedule) or each operation is started as late as possible (ALAP schedule). For example, if all of the 9 synchronizing buffers are assumed between **e(3)** and **e(2)** as shown in Figure 13, then this situation corresponds to the ASAP scheduling. Obviously, the synchronizing effect would not be changed, if all of the 9 synchronizing buffer registers were placed at the input of **e(2)**. This arrangement would represent the ALAP schedule for the EOG in Figure 11. Thus, the upper bound of **p(3,2)** can be used for calculating the maximal mobility of **e(2)**. Considering the minimal required value of the synchronizing delay effect calculated as **minp(3,2)=3**, the corresponding 3 buffer registers would not represent the same delay effect at the input of **e(2)** as they do between **e(2)** and **e(3)**. The reason of this is that during the calculation of **minp(3,2)**, **t(h)=1** has been assumed, which is not true any more if there is no buffer register between **e(2)** and **e(3)**. In this case, **t(h)=2** has to be taken into consideration because of **t(2)=2**. Thus, the new value for **minp(i,h)** transferred to the input of **e(2)** would be: $9+2+5-12=4$. An other possible arrangement could be obtained, if a single buffer register was left between **e(2)** and **e(3)**. In this case 2 buffer registers, i. e. altogether 3, would be enough at the input of **e(2)** for representing **minp(i,h)**, because **t(h)=1** would be guaranteed by the single buffer register. The algorithm SYNC provides always the ASAP schedule, if all the synchronizing delay effects **maxp(i,h)** are placed between **e(i)** and **e(h)** or -in the case of recursive loops- at the loop input. Starting from this schedule, the ALAP schedule can be obtained systematically by moving the delay effects step by step from the output of each elementary operation to its inputs. This procedure is to be started for the operations which produce the output data of the EOG and continued successively upwards until the inputs of the EOG. During the relocation of the delay effects, the latency of the EOG must not increase and the synchronization obtained as the ASAP schedule must not be hurt. Two possible conflicts are shown in Figure 22.

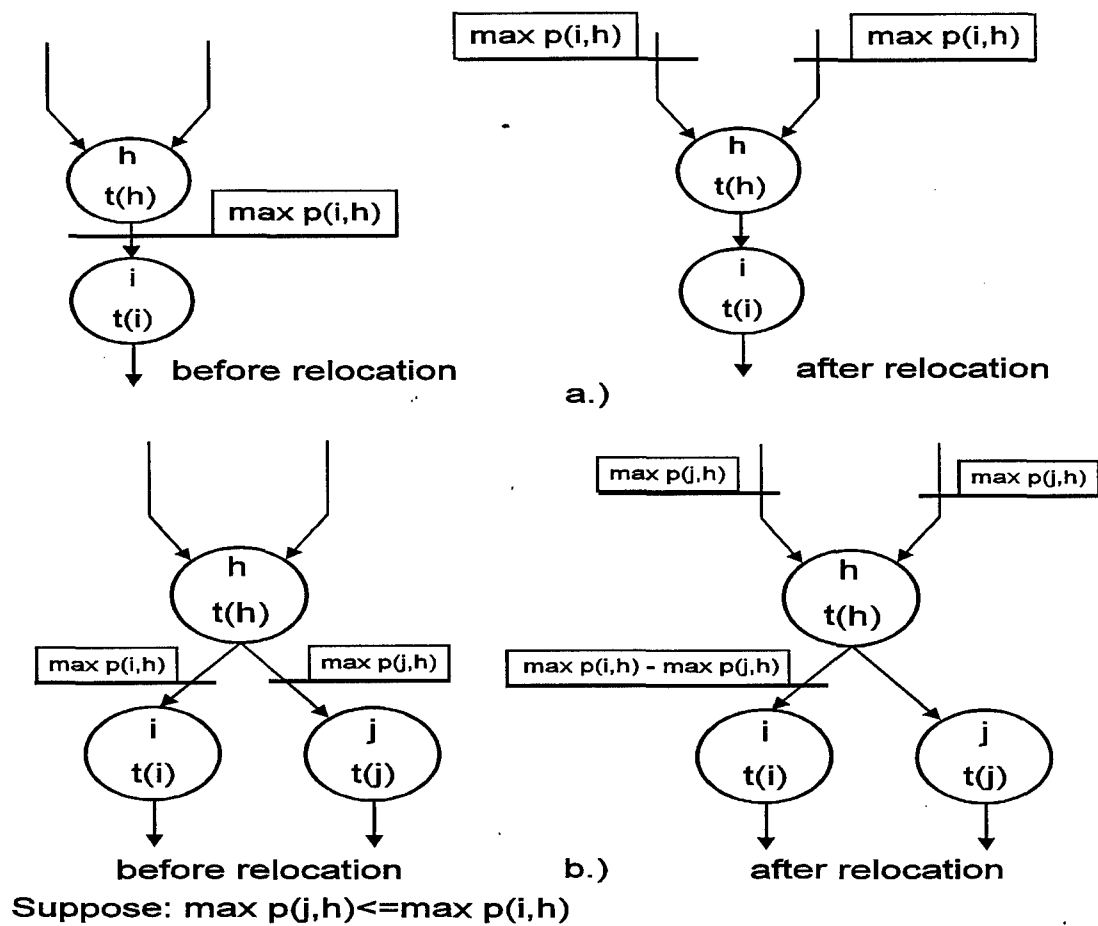


Figure 22

Relocation of synchronizing delay effects in the case of operations
a.) with collector property, b.) with distributor property

If an elementary operation has more than one input (**collector property**), then the delay effects originating from its output must be repeated at each input. Otherwise, the synchronization would be hurt. If the output of an elementary operation is connected to the inputs of more than one other operation (**distributor property**), then different synchronizing delay effects may occur along each connection. In this case, only the smallest delay effect is allowed to be transferred to the inputs of the operation with the distributor property and this smallest value must be subtracted from each delay effect occurring along the other connections at the output. Based on these considerations, the ALAP schedule of the EOG in Figure 17 is shown in Figure 23, where only the values of $\max p(i,h)$ are symbolized, since the delay effects according to $\min p(i,h)$ would not mean the ALAP schedule.

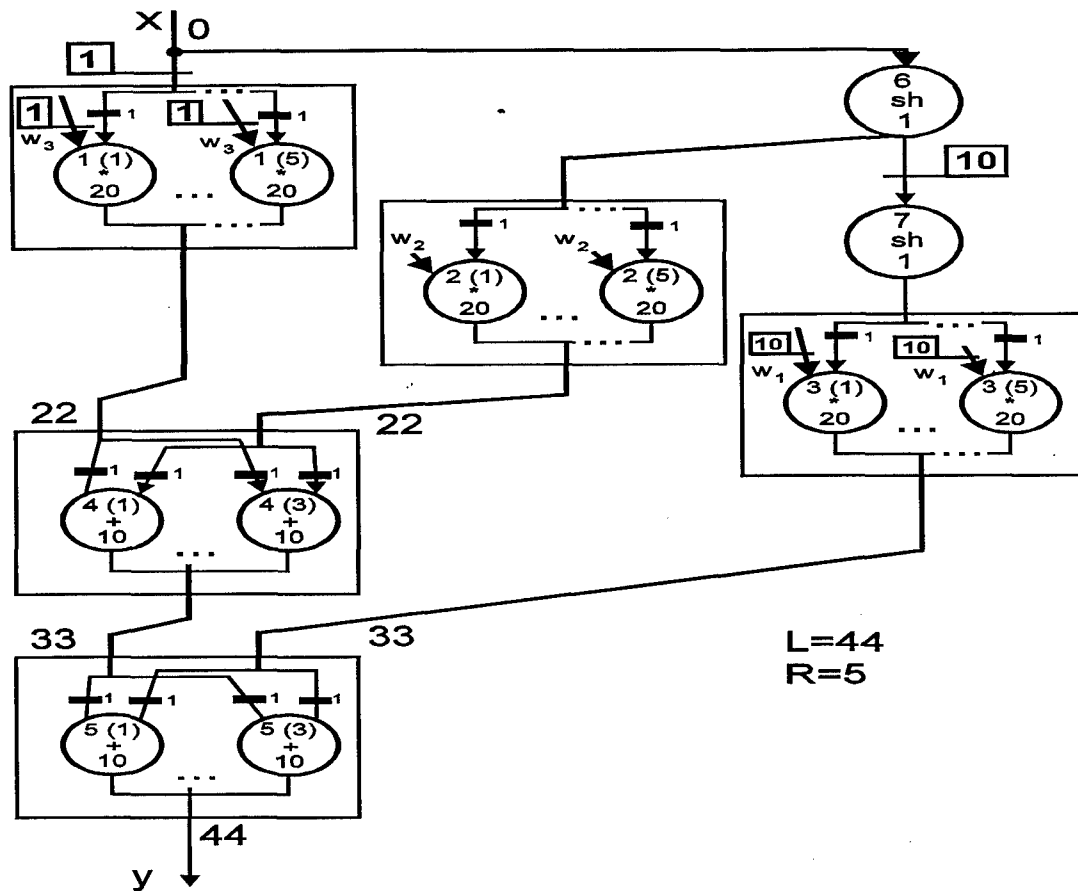


Figure 23
The ALAP schedule for the EOG in Figure 17

Note that the delay effect 1 at the input of $e(1)$ must not be pushed further to the input x of the EOG, because influencing $e(6)$ is not allowed during the relocation of $\max p(4,1)=1$. In other words, the branching point at an input of the EOG is always to be handled as an elementary operation during the relocation procedure. Obviously, the delay effects at inputs w_i are only formal, since these inputs are supplied with constant values which can be assumed to be available permanently.

Having the ASAP and ALAP schedules, the mobility time domain $\text{mob}(i)$ for an elementary operation $e(i)$ can be expressed as follows:

$$\text{mob}(i) = (\text{al})b(i) - (\text{as})b(i),$$

where $(\text{al})b(i)$ and $(\text{as})b(i)$ stand for the earliest possible starting time $b(i)$ of $e(i)$ according to the ALAP and ASAP schedule respectively.

To schedule an EOG means to assign a starting time for each $e(i)$ inside its mobility time domain $\text{mob}(i)$, i. e. between its ASAP and ALAP constraints.

With the buffer register representation of the synchronizing delay effects, the scheduling procedure can be formulated as to arrange the buffer registers of number $\max p(i,h)$ inside the

ASAP and ALAP constraints for each $e(i)$ as it has already been indicated in Figure 14. Each possible schedule could be obtained if starting with the ASAP schedule, the relocation procedure towards the ALAP schedule were stopped at every possible intermediate step for each individual buffer register of $\maxp(i,h)$ provided by the algorithm SYNC. Even by relocating a single buffer register from the output to the input of an $e(i)$, a new schedule can be generated. Therefore, every possible arrangement of the buffer registers representing the synchronizing delay effects $\maxp(i,h)$ establishes a different schedule for the EOG.

The aim of the scheduling in high-level synthesis is to ensure the best conditions possible for covering the elementary operations by real resources called processing units or processors. This is the **allocation** step of the synthesis and it is detailed in the next chapter. It is trivial, however, already at this stage that the schedule of the EOG has a strong influence on the efficiency of the allocation. For example, if the number and types of the processors are given in advance as constraints, then the allocation means to cover the elementary operations by their proper disjoint subsets, each of which is to be realized by a single processor. In this case, only such elementary operations can be drawn together in common subsets which are never busy at the same time, since otherwise timing conflicts would arise among the elementary operations sharing the same processor. Obviously, the quality of the solution and even the solvability of this allocation problem strongly depend on the schedule, since the concurrence of the elementary operations can be modified choosing an other schedule. How to determine the most advantageous schedule for the given allocation constraints, this is the problem to be solved by the scheduling methods. No algorithms are existing for the optimal solution, since the problem is NP complete. However, many practical approaches have been proposed in the literature for solving the scheduling problem. The quality and performance of these methods can be judged only by comparing their results obtained for characteristic benchmark examples. The basic concepts of these practical approaches to the scheduling problem are detailed in a separate chapter after having been introduced the most important constraints for the allocation in the next chapter.

7. Allocation

The aim of the resource allocation algorithm is to decide which elementary operations are to be realized by common real processors[5],[6]. It means that proper subsets of the elementary operations are to be found under several constraints (cost, area, data path complexity, processor type, etc.).

7.1. Covering of non concurrent operations

One of the possible strategies is based on the possibility that non concurrent operations may share a common processor. It is trivial that the concurrence of the elementary operations is strongly influenced by the length of the pipeline restarting period and the scheduling. The four possible **time overlapping** (concurrence) situations between two operations are shown in Figure 24, where $s(i)$, $s(j)$ and $f(i)$, $f(j)$ denote the starting and finishing points of time of $e(i)$ and $e(j)$, respectively and $b(j) > b(i)$ is assumed.

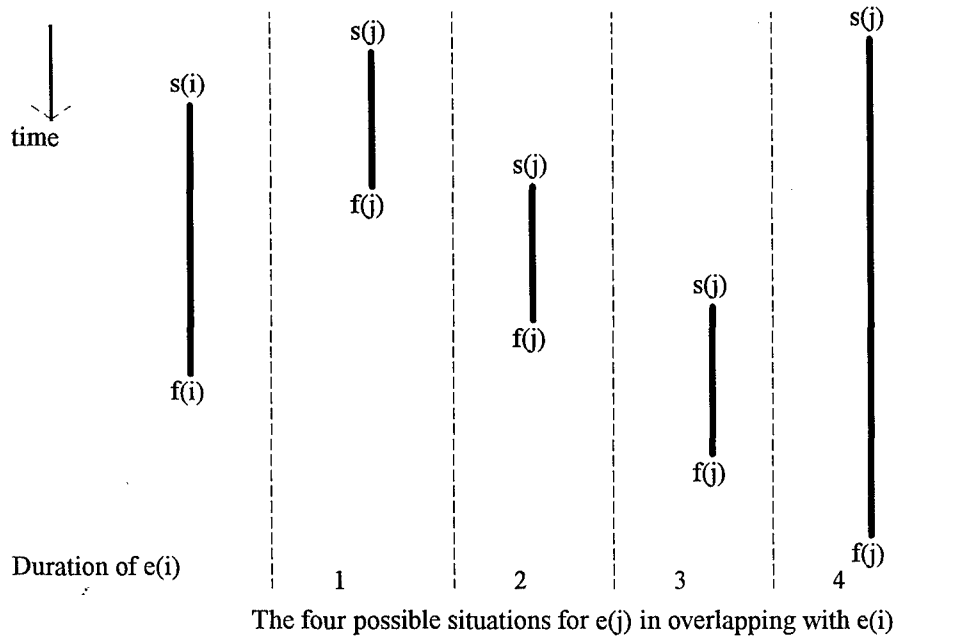


Figure 24
Concurrence situations of operations

The finishing points of time can be expressed as: $f(i) = s(i) + q(i)$ and $f(j) = s(j) + q(j)$. In a pipeline mode, the starting times of the operations $e(i)$ and $e(j)$ can be expressed as follows:

$$s(i) = b(i) + k(i) * R \quad \text{and} \quad s(j) = b(j) + k(j) * R,$$

where $k(i)$ and $k(j)$ are arbitrary non-negative integers representing the serial numbers of the starts of $e(i)$ and $e(j)$, respectively. If $e(i)$ and /or $e(j)$ are multiplied and their numbers of copies are $c(i)$ and $c(j)$, then the expressions for the starting times of their $n(i)$ -th and $n(j)$ -th copies can

be considered as generic forms including also the above expressions for the case $c(i)=c(j)=n(i)=n(j)=1$:

$$s(i, n(i)) = b(i) + ((n(i) - 1 + k(i) * c(i)) * R \quad \text{and} \quad s(j, n(j)) = b(j) + ((n(j) - 1 + k(j) * c(i)) * R,$$

where $1 \leq n(i) \leq c(i)$ and $1 \leq n(j) \leq c(j)$.

Note that $k(i)$ and $k(j)$ represent also the serial number of the restarting period, but only if $c(i)=c(j)=1$.

The four overlapping situations can be characterized by the following inequalities:

$$s(i) \leq s(j) \leq f(i)$$

$$s(j) \leq s(i) \leq f(j)$$

Introducing the above generic expressions for the starting times,

$$(b(i) - b(j)) / R \leq K \leq (b(i) - b(j) + q(i)) / R$$

$$(b(i) - b(j)) / R \geq K \geq (b(i) - b(j) - q(j)) / R$$

can be obtained, where

$$K = n(j) - n(i) + k(j) * c(j) - k(i) * c(i). \quad (1)$$

The left sides of both inequalities are identical, therefore they can be substituted by a single inequality as the necessary and sufficient condition for the concurrence of $e(i)$ and $e(j)$:

$$(b(i) - b(j) - q(j)) / R \leq K \leq (b(i) - b(j) + q(i)) / R \quad (2)$$

The operations $e(i)$ and $e(j)$ are overlapping in time i.e. concurrent if and only if inequality (2) holds at least for one integer K satisfying equation (1). Based on this result, the necessary and sufficient condition also for the non-concurrence of $e(i)$ and $e(j)$ can be expressed. Firstly, let $c(i)=c(j)=n(i)=n(j)=1$ be assumed, i. e. neither $e(i)$, nor $e(j)$ are multiplied. In this case, any integer K satisfying inequality (2) excludes the non-concurrence. The non-existence of such integer K requires that the integer part of the left and the right side of inequality (2) must be equal:

$$(b(i) - b(j) - q(j)) / R = INT + FR1$$

$$(b(i) - b(j) + q(i)) / R = INT + FR2,$$

where INT and $FR1$, $FR2$ denote the integer and the fraction parts respectively.

The assumption $b(j) > b(i)$ involves that the left side of inequality (2) and so, INT , $FR1$, $FR2$ are all negative. Expressing INT from the above first equality:

$$INT = (b(i) - b(j) - q(j)) / R - FR1$$

and substituting it in the second one:

$$FR2 - FR1 = (q(i) + q(j)) / R$$

can be obtained. Since $FR2 > FR1$ and both are negative, $FR2 - FR1 < 1$ holds. Thus,

$$(q(i) + q(j)) / R < 1$$

$$\text{or} \quad q(i) + q(j) < R \quad (3)$$

can be written, as a trivial necessary condition for the non-concurrence. It expresses that the sum of the busy times of $e(i)$ and $e(j)$ must be shorter than the restarting period, otherwise, the overlapping could not be avoided. An other necessary condition can be concluded from the requirement that the right side of inequality (2) must be negative:

$$(b(i)-b(j)+q(i))/R < 0$$

or simplified:

$$b(j)-b(i) > q(i), \quad (4)$$

which is also trivial, since the time difference between the starting times of $e(i)$ and $e(j)$ must be longer than the busy time of $e(i)$, if $b(j) > b(i)$. Otherwise, the non-concurrence would be hurt. However, inequality (4) alone is only a necessary condition of the non-concurrence, since it holds also for the case $b(j) = b(i) + R$, which involves the concurrence of $e(i)$ and $e(j)$ in the second restarting period. Such situations can be excluded by considering that $FR2$ must not be zero for the non-concurrence, i. e. the right side of inequality (2) must not be equal to INT , otherwise INT would be the integer solution for K . Obviously, $FR1 = 0$ cannot occur, because $FR2 > FR1$ and both are negative.

Thus

$$(b(i)-b(j)+q(i))/R < INT$$

and introducing the expression for the common integer part of both sides of inequality (2):

$$(b(i)-b(j)+q(i))/R < Int((b(i)-b(j)-q(j))/R) \quad (5)$$

If all of the inequalities (3), (4) and (5) hold, then they represent together the necessary and sufficient condition for the non-concurrence in the case of $c(i) = c(j) = 1$, because these conditions are just sufficient for excluding the existence of an integer K solution in inequality (2).

If $c(i)$ or $c(j)$ or both of them differ from 1, i. e. $e(i)$ or $e(j)$ are multiplied, then inequality (2) always has integer solutions for K , because the length IL of the interval for K is always greater or equal to 1 in this case:

$$IL = (q(i) + q(j))/R \geq 1,$$

since $q(i)/R > 1$ or $q(j)/R > 1$ holds, if $e(i)$ or $e(j)$ is multiplied. Therefore, the non-concurrence would require that no integer solution of K satisfies equation (1) for any non-negative integer values of $k(i)$ and $k(j)$ as variables. It can be proven that such values of $k(i)$ and $k(j)$ can be found for each integer solution of K . It means, that $e(i)$ and $e(j)$ are always concurrent if at least one of them is multiplied. For the proof, let inequality (2) be rearranged as follows:

$$(b(i)-b(j)-q(j))/R - n(j) + n(i) + k(i) * c(i) \leq k(j) * c(j) \leq (b(i)-b(j)+q(i))/R - n(j) + n(i) + k(i) * c(i)$$

Choosing a positive integer $k(i)$, which makes both sides positive and introducing the notation:

$$A = (b(i)-b(j))/R - n(j) + n(i) + k(i) * c(i),$$

inequality (2) can be expressed as

$$A - q(j)/R \leq k(j) * c(j) \leq A + q(i)/R$$

Obviously, the length of the interval for $k(j) * c(j)$ is the same as it has been for K before the rearranging. If $IL \geq c(j)$ holds, then there exists at least one non-negative $k(j)$ which satisfies the above inequality, i. e. inequality (2). This condition can be rewritten as follows:

$$R * c(j) \leq q(i) + q(j)$$

Let $R * c(j)$ be substituted by its upper bound derived from the calculation rule of $c(j)$ according to the algorithm RESTART:

$$(q(j)+1)/R+1 > c(j) \text{ that is } q(j)+1+R > R \cdot c(j).$$

Since both sides are always integers,

$$q(j)+R \geq R \cdot c(j)$$

can be written

Thus:

$$q(j)+R \leq q(i)+q(j) \text{ that is } R \leq q(i),$$

which always holds, if $c(i) > 1$. At this stage, the proof is completed, since no constraints have been assumed for $c(j)$ and non-negative integer values for $k(i)$ and $k(j)$ can always be found in the above way.

Note that it was assumed for the proof that the minimal numbers of copies have been applied for the restarting period. Therefore, the copies of multiplied operations are always concurrent, if the minimal numbers of copies have been calculated by the algorithm RESTART. In this case, it is obvious that the multiplied copies are always overlapping each other in time and their new restarts occur just after the end of their busy state at most one clock period later. This free time is not enough for the non-concurrence with any kind of elementary operations, since even the shortest busy time is 2 clock cycles.

Conditional branches in EOG need special considerations during the concurrence checking. If $e(i)$ and $e(j)$ are in separate branches of the same conditional checking, then they are never executed in the same restarting period, i. e. $k(i)=k(j)$ never holds for the case $c(i)=c(j)=n(i)=n(j)=1$. Therefore, the solution $K=0$ does not exclude the non-concurrence, since equation (1) can be rewritten, as $0=k(j)-k(i)$, which would hold only for $k(i)=k(j)$ and it is impossible now. Obviously, any nonzero integer solutions for K exclude the non-concurrence. It means that the non-concurrence can hold only if the left side of the inequality (2) is greater than -1 and the right side of it is smaller than +1:

$$(b(i)-b(j)-q(j))/R > -1 \text{ and } (b(i)-b(j)+q(i))/R < +1, \text{ or rewritten:}$$

$$b(j)+q(j) < b(i)+R \tag{6}$$

$$b(i)+q(i) < b(j)+R \tag{7}$$

Both above inequalities express the necessary non-concurrence conditions that the busy time of one of the operations must be finished earlier than the execution of the other one begins in the next restarting period. These conditions involve the most pessimistic assumption that the conditional branches may alternate with the restarting periods. If $c(i)=c(j)=n(i)=n(j)=1$ and both of the inequalities (6) and (7) hold, then they represent the necessary and sufficient condition for non-concurrence in the case of operations in separate branches of the same conditional checking, because these conditions are just sufficient for excluding any nonzero integer K from the solutions of inequality (2). If either $e(i)$ or $e(j)$ is multiplied, then the non-concurrence is not excluded only if $K=0$ is the only integer solution of inequality (2) also in this case. By adding both sides of inequalities (6) and (7),

$$q(i)/R + q(j)/R < 2$$

can be obtained, which can hold if only one of $q(i)/R$ and $q(j)/R$ is greater than 1. It means that at most one of $c(i)$ and $c(j)$ can be 2 and the other must be 1, i. e. $c(i)+c(j) \leq 3$, provided that $c(i)$ and $c(j)$ are determined by the algorithm RESTART. In consequence, the non-concurrence of multiple operations being in separate branches of the same conditional checking is possible only in the cases $c(i)=2, c(j)=1$ or $c(i)=1, c(j)=2$. This result can easily be explained as follows. If a multiple operation has only two copies, then the busy times of each copies last not longer than the end of every second restarting period. Otherwise, the algorithm RESTART would not calculate 2 copies. Thus, the time left after end of the busy time until the beginning of the next restart may be enough for matching the busy time of a non-multiple operation from an alternative conditional branch, provided that inequalities (6) and (7) hold. Since the most frequent turn of a conditional branch is every second restarting period, this situation is sufficient for the non-concurrence. Obviously, more copies would not allow the non-concurrence, because their busy times would cover at least two restarting period and the pessimistic alternating turn of the conditional branches would exclude the non-concurrence.

Based on the above considerations, the algorithm of the concurrence checking (CONCHECK) is summarized by the flow chart in Figure 25.

Obviously, multiple copies of the same elementary operation cannot be covered by a common processor based on the non-concurrence property. If, however, the processor library to be used contains so called **structurally pipelined** processors, then this type of processors can cover several copies of a multiplied elementary operation. Namely, **structural pipelining** means that the processor is able to accept new data before completing the previous one. This is just the case, if an elementary operation is multiplied and considered as a processor. Thus, multiplying the elementary operations is equivalent to forming structurally pipelined processors. Therefore, using this type of processors, the result obtained by the algorithm CONCHECK may be improved, since the copies of the same multiplied elementary operations can be additionally covered by common structurally pipelined processors.

It can be proven [8] that the concurrence is a compatibility relation between two operations. Based on the above conditions, the compatibility checking can be executed for each pair of operations. The maximal compatibility or incompatibility classes can be obtained by the well-known algorithms [9]. For finding the proper subsets of operations to be realized by common processors, a proper cover must be constructed with respect to the actual constraints for the processors.

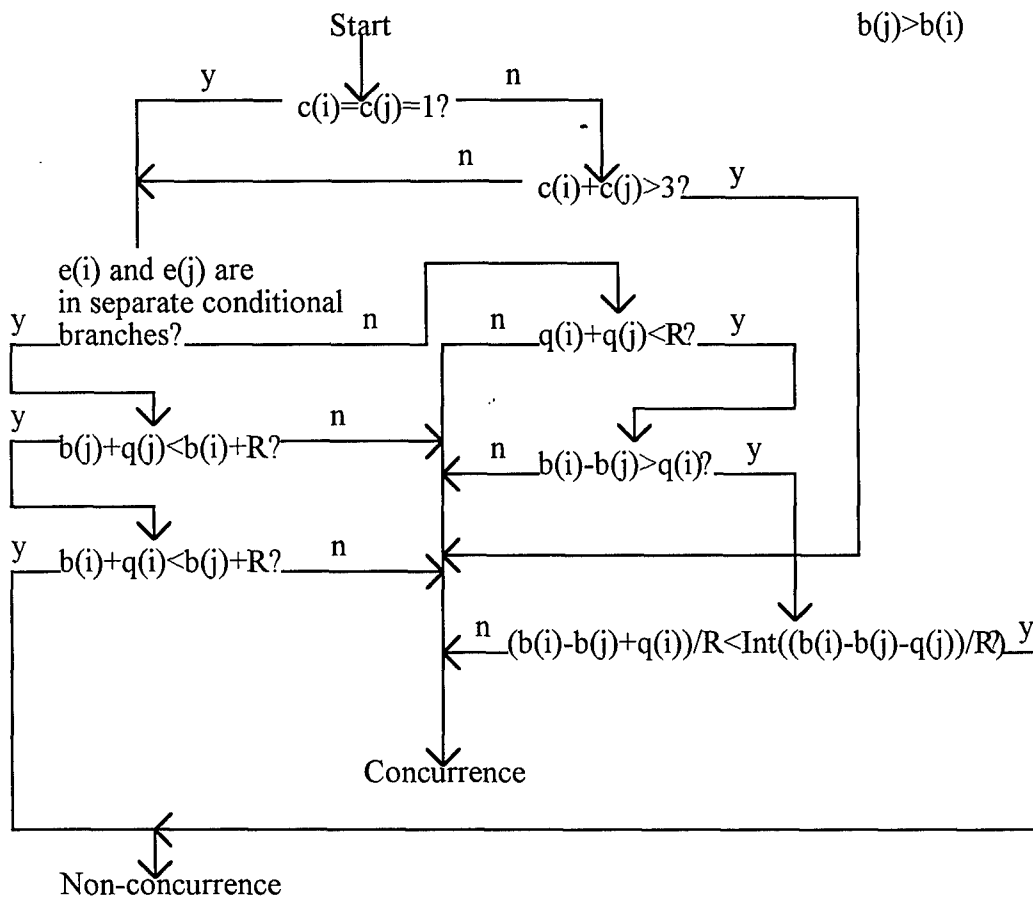


Figure 25
The flow diagram of the algorithm CONCHECK
if $b(j) > b(i)$ is assumed

7.2. Topological cover of operations

Another strategy of allocation could be the splitting of the EOG into parts containing data-connected operations. These parts specify the processors to be realized. The constraints for forming the processors may be the number and complexity of the operations to be combined in common processors or the simplicity of the data connections between the processors and in many cases the regularity of the structure. As an example for the regularity, let the EOG of the digital convolution algorithm in Figure 16 be considered. In Figure 26, this EOG is completed with the dotted adder and shift register. The synchronizing delay effects are to be calculated for $e(4)$ and $e(5)$ only. If the $\text{minp}(i, h)$ values are applied, then no extra buffers are required, since

$$\text{minp}(4, 2) = \text{minp}(5, 3) = 9 + 10 + 10 - 31 = -2.$$

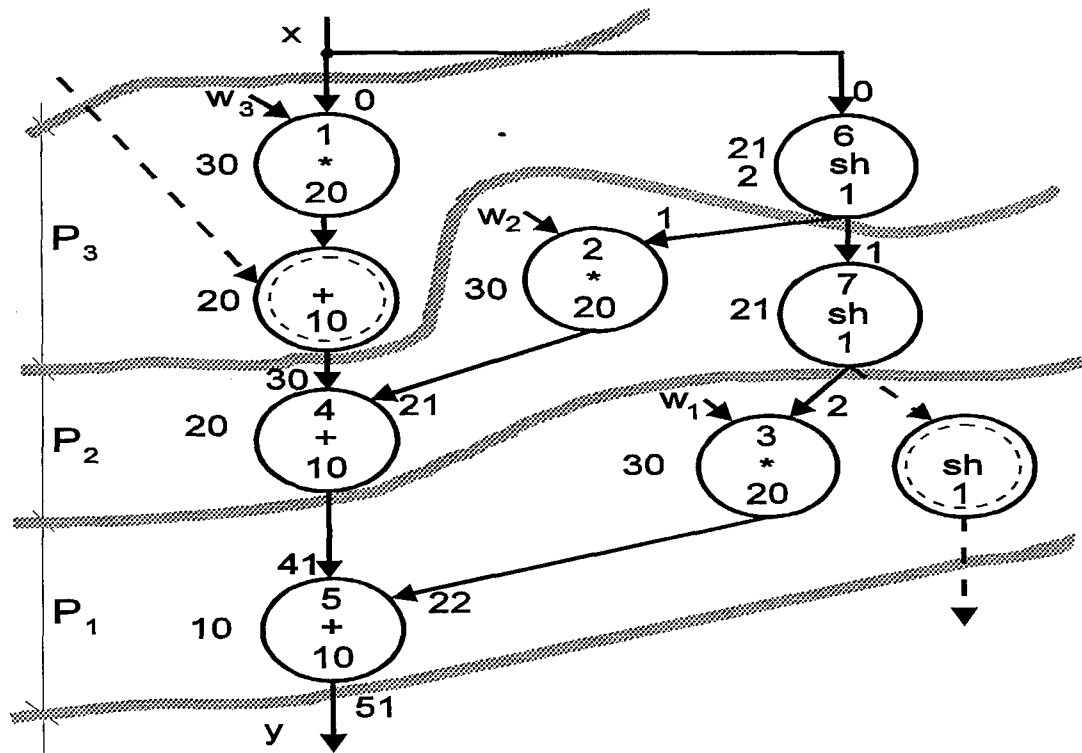


Figure 26
Completion and splitting of the EOG in Figure 16

It can be seen that the completions do not affect the restarting period, but makes the EOG look symmetrical. Therefore, the separation of identical EOG parts becomes possible as indicated by the lines crossing the EOG. Each part may form an identical processor as shown in Figure 27 with the input-output specification as follows:

$$y(\text{out}) = y(\text{in}) + w_i * x(\text{in})$$

$$x(\text{out}) = x(\text{in}-1),$$

where $x(\text{in}-1)$ denotes the input data occurred one restart earlier, than the actual one.

The regular structure consisting of such processors is illustrated in Figure 28 which is similar to one of the systolic realisations of the digital convolution algorithm.[10]

Finding or establishing the symmetry of an EOG cannot be executed systematically without intuition and trials. This task may become more difficult after the algorithms RESTART and scheduling, since the arrangements of extra delay effects and multiple copies of operations may eliminate even the inherent symmetry of the initial EOG. For example, the EOG in Figure 17 would be more difficult for establishing identical parts by completion outlined above and a completion usually needs a reschedule.

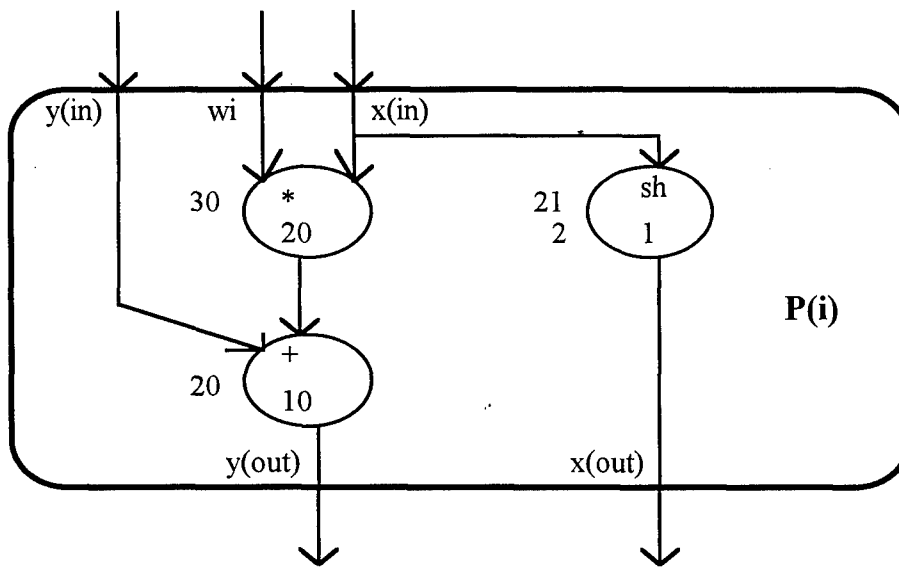


Figure 27
The internal structure of a processor obtained in Figure 26

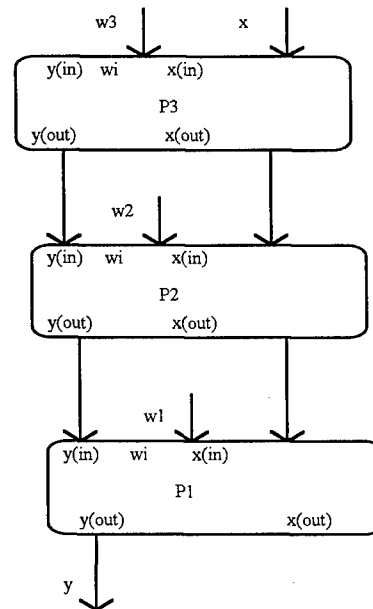


Figure 28
Systolic realisation of the digital convolution algorithm

8. Multiple-process recursive loops

Recursive loops are usually considered to be unavailable to overlapped execution during the scheduling phase of high-level synthesis. This is caused by the special nature of recursive execution: an iterative algorithm may not be fed the next data before the final result of the previous iteration is ready. As calculation of the next value requires a minimum of $T(i)$ cycles ($T(i)$ is **loop duration** of loop i (defined as the sum of execution times for the operations in the loop), a restart period under

min $R = \max(T(i)+1, i: \text{every loop})$

time cycles is impossible in a data path containing recursive loops as subsequent data would enter slower loops before their **iteration** would end. (Outside recursive loops it is useful to measure time in *iterations*, with one iteration being equal to the amount of time cycles it takes for the data to finish one complete turnaround in the recursion.)

There are some notable exceptions, however, to the general case. In a special type of problems, recursive solutions are needed to calculate values of *identical* functions for **different processes**. In this case, as the operations inside the loop (core of the loop, from now on referred to as **the loop**) may hold more than one data simultaneously as long as there is no collision between them. In this way, the loop may process data with a higher throughput than the loop iteration would permit.

Such an example is the centralised control of robots using the *computed torque technique*. Realisations of the computed torque technique require periodic calculations of a dynamic model for the robot joints to deal with changes in the environment. For the scheduling phase, this calculation may be run simultaneously with torque calculations, in a *conditional execution branch* with a probability of $1/N$ if a calculation is required every N th iteration.

As conditional execution would be a special problem in scheduling, the conditional branches are treated as ordinary (non-conditional) branches in the graph.

Most of the published methods handle the recursive loop latencies ($T+1$) as the minimal value of the restarting time (R). This constraint causes design methods to minimise the total execution time in loops. The block approach treats recursive sections as parts of the data graph that are elementary operations and so unavailable for optimisation.

8.1. Overlapped (pipelined) utilisation of recursive loops

Decomposing a loop sequence to real elementary operations makes it possible to tune the behaviour of the loop itself.

In the execution of a recursive loop, some parts of the loop are *busy* while the others are *idle*. As execution of such a loop is strictly sequential, the busy state "propagates" along the data path in time (Fig. 29.).

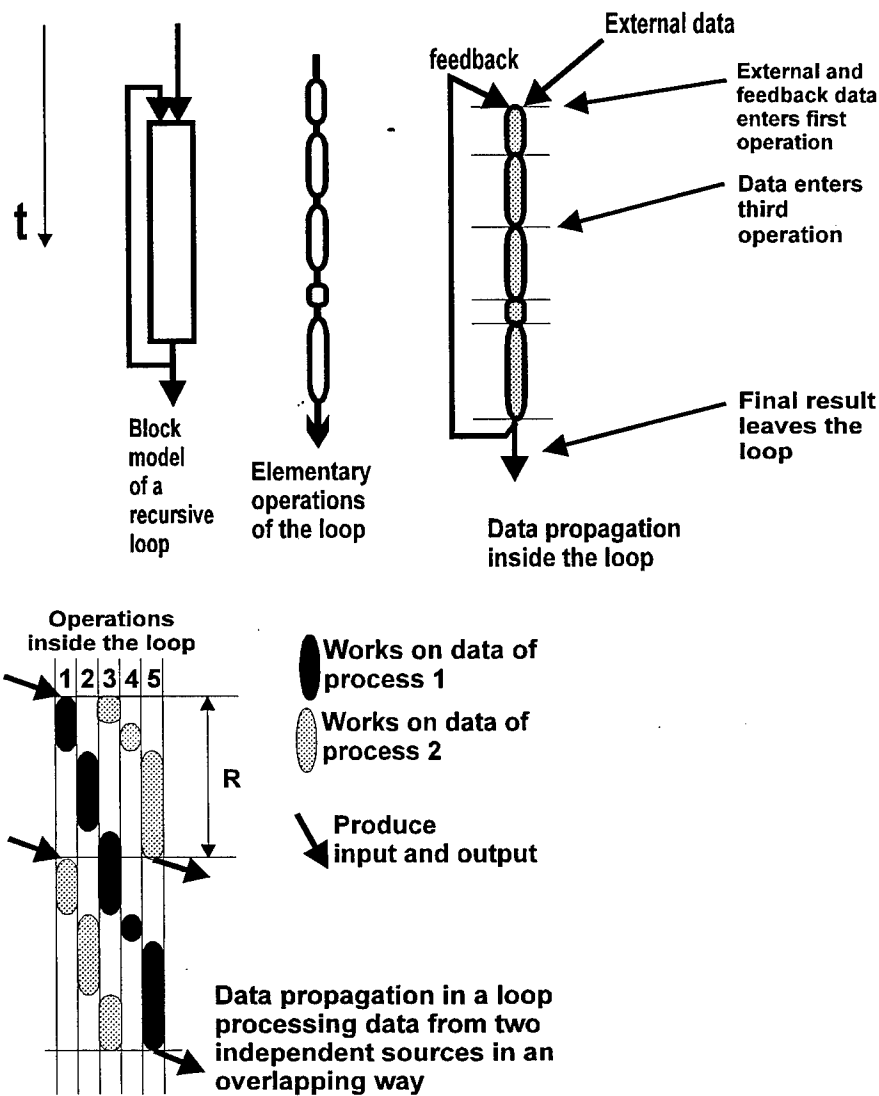


Figure 29.

Idle and utilized times in an overlapped (pipelined) recursive data path

As time limits permit, it is possible to introduce new data (belonging to another process) to the start of the loop that runs through the loop without data conflict with the previous data. This overlapped execution exploits the inherent idle states of the loop. With such a structure, more than one process may use the loop, if a strict schedule of data introduction is maintained. Such a loop is referred to as a **multiple-process (recursive) loop**. Processes forward data to the loop once in every $T(i)+1$ cycles (and they fetch data at the same rate) while the loop itself serves more processes and so takes and outputs data at a higher frequency.

As in recursive overlapping (pipeline) mode data is put in the system in a mixed order to the loop, a few definitions are needed. A **complete iteration** is equal to the time a process uses the loop, i.e. between the cycle of sending data to the loop for the first time and receiving data from the loop for the last time. An **iteration** is the time of one turnaround of the loop, i.e. time of all loop

elementary operations and the delay in the feedback; the number of iterations in a complete iteration is referred to as **loop depth** or **iteration depth**, $g(i)$.

A **data packet** or **packet** is data belonging to a process; a packet consists of a single set of input values that starts the iteration (**initial data**) and the intermediate values that are present in the system before the iteration is finished (**iteration data**). During iterative execution the process needs to supply the initial value of the iteration and the recursion is finished when exit criteria are met. During iteration the process need not supply more data since the next initial value arrives from the previous iteration through the feedback branch.

As an example in the case of iterative solutions of non-linear equations, the $f(x)$ value is checked if it is in the neighbourhood of zero or not. In a well-known solution method (**tangent method**) a non-zero $f(x)$ moves the x (initial) value to another x' where $f(x')$ is hopefully smaller in absolute value than $f(x)$ so the iteration improves the quality of the x value. The new x' is found as the intersection of the x axis and the tangent of $f(x)$ in $(x, f(x))$. This algorithm may result in an oscillation deadlock between two suitable points (see Fig. 30.). It is clear, however, that the process needs to supply only one value (the initial value x), as all the following x values are the feedback of the previous x' value.

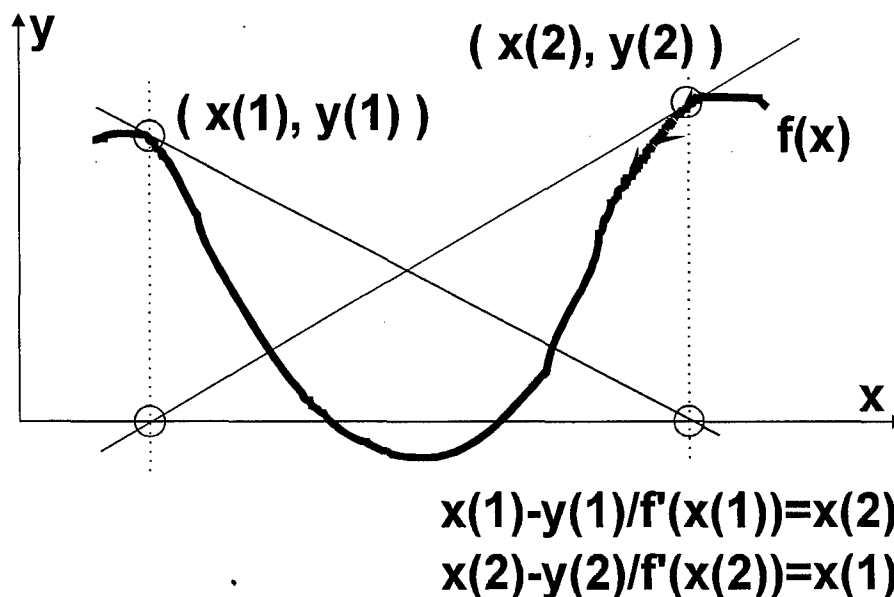


Figure 30.

Oscillating states in an iterative solution of an equation

Another type of recursion is where both the feedback and the process supply data. Such a case is a circuit that realises exponential averaging on a given input sequence. This may be described with an equation

$$y(i) = y(i-1) + a * (x(i) - y(i-1)).$$

It is obvious that this structure depends on both the previous output ($y(i-1)$) and the current input ($x(i)$).

A multiple-process loop must be properly synchronized so that

- 1) data exiting the loop and returning to the loop input must meet the next **corresponding** input data and
- 2) no data overrun occurs inside the loop i.e. the structure of the loop must be available for the desired restart period.

By using multiple-process recursive loops, we make possible for separate processes to share the same resources in such a way that the recursive section of processes is realised only once. Every loop features a receptor element as its first operation, which handles feedback from the previous iteration and receives new data from the process.

To tune the loop and to guarantee proper execution the previous and the new data of the same process (the first fed back from the loop output, the second arriving to the loop receptor from the outside of the loop) must arrive in the same time to the receptor element. This means a synchronisation problem in the transfer sequences connected to the loop (external synchronisation) and an additional synchronisation inside the loop (loop scheduling). Since the external synchronisation is loop dependent, this problem will be discussed after the classification of the recursive problems. As the basic synchronisation tool is the delay (buffer), a recursive structure optimised for pipelined execution is likely to execute slower than the non-pipelined loop. This speed loss may be small, especially if loop execution time much greater than 1, as the buffers cause a latency increase of 1 each.

8.2. Loop scheduling

It is obvious to discuss the scheduling problem for the easiest case, when the whole elementary operation graph consists of one loop, therefore external synchronisation is eliminated. In this structure $T=T(1)$. It will be assumed that the graph contains only one receptor element at the beginning of the recursion. The decision whether the data must stay in the loop for another iteration or the recursion has finished will also be made by this receptor element (so the loop shall behave like a "test-before" construction, like a WHILE instruction in high-level languages).

To be able to schedule other types of loops, they must be transformed to "test-before" constructions. A REPEAT-UNTIL loop in Pascal may be transformed to a WHILE in general like from

...

Repeat

Loop Instructions;

Until (Condition);

we get

```

...
Loop Instructions;
While (Condition) Do Begin
    Loop Instructions;
End;

```

Another possible solution is the manipulation of the loop condition:

```

...
var LoopCondition;
LoopCondition:=TRUE;

While (LoopCondition OR Condition) Do Begin
    Loop Instructions;
    LoopCondition:=FALSE;
End;

```

which solution guarantees that the loop always runs for its first iteration, yet it is a test-before loop. The second solution is preferred as it does not duplicate operations in the graph.

The same is usually possible with FOR type loops of C and Pascal. The loop execution condition should be connected to operations in the loop, otherwise the loop is infinite or never executes, which are not practical recursive cases.

The loop is considered to be able to handle the data of N processes simultaneously. This means that the $N+1$. data arriving to the first receptor will be the next fresh data of the first process. To avoid a data synchronisation problem between the feedback of the loop output (initial value of the next iteration) and the new data, the loop duration (T) should be an integer multiple of the restart time (R):

$$T = (N + 1) * R$$

As T is a function of R and N values that are given (set by the problem and/or hardware limits), the equation will hold just for special cases; otherwise we have to solve the synchronisation problem inside the loop. To solve this problem the difference between T and R must be realised as additional delay inside the loop. In this way the expression will be modified:

$$T' = T + p = (N + 1) * R$$

where p is the number of the inserted buffers (delay elements). The number of inserted buffers is determined by 2 factors:

$$p = p(s) + p(f)$$

- *Scheduling the loop for the aimed restarting time* (calculation of $p(s)$): for this problem any suitable scheduling method may be used to tune the opened loop. $p(s)$ is the sum of the number of the buffers inserted to the places where the transfer scores make it necessary and the buffers are inserted before the multiplied elements.

- *Synchronisation between input and feedback of the loop* (calculation of $p(f)$): $p(f)$ may be expressed as

$$p(f) = (N + 1) * R - T - p(s)$$

$$\text{where } T + p(s) \leq (N + 1) * R.$$

In a system where

$$T + p(s) > (N+1) * R$$

the feedback of the system will be slower than the system input (i.e. the external data source) and this must be taken into consideration when designing the external synchronisation. In this case the feedback must not be further delayed so $p(f)=0$ is used.

8.3. Classification of recursive problems

Data propagation properties may be classified as one of three main classes based on the number of iterations. The number of iterations a packet spends in a recursive loop (**loop depth, $g(i)$**) is either a finite (constant or variable) or an infinite value.

1) Finite loop depth, variable number of iterations

Some problems iterate for a variable number of times before their data exits the loop. These applications should finish within a finite number of iterations, otherwise they are unavailable for real-time usage. To prevent this infinite iteration, the system should be equipped with a **watchdog** mechanism that finishes the iteration after a pre-set turn-around time, regardless of system state. (Infinite number of iterations is usually a fatal system error, for example a non-linear system approximation stuck in an oscillating set of states.)

Iterative solutions of differential equations are remaining in an iterative state for a finite number of loop transitions, while the exact loop depth is data dependent; this kind of problems is a typical example of variable-depth calculations. For these calculations no exact time requirement rule may be given as the time needed to finish processing a packet is unknown at design time. The time needed for the first packet to leave and a new initial data to be introduced to the iteration is

$$\min(R * (g(k) * (N+1) + k-1), k:\text{every process})$$

as packet k enters the loop in time cycle $R*(k-1)$ and it must run through $g(k)$ iterations, each using $R*(N+1)$ cycles. A similar expression tells the termination of the last iteration:

$$\max(R * (g(k) * (N+1) + k-1), k:\text{every process})$$

for similar reasons.

2) Finite loop depth, constant number of iterations

The method to treat special recursive loops may be applied to some applications that are not strictly recursive. Such loops include the FOR-NEXT (ENDFOR) loops of high-level programming languages. For our considerations these loop types are treated as special recursive tasks, like a feedback system where the feedback branch is not really important in the structure.

The set of problems with a *fixed, finite number* of iterations is typical for the equivalent of FOR loops (which generally run for a constant number of times). In the case of a fixed-order FIR filter the number of iterations is related to the order of the filter, which is a constant based on the nature of the problem.

To model this set of problems, we assume that the hardware has to put x initial data through the recursive calculations. Every packet of these data stays in the loop for $g(0)$ iterations. Non-overlapping execution would finish with these data in $g \cdot T \cdot x$ cycles as x packets must terminate all their iterations (g), each of those lasts $g \cdot T$ time cycles. The loop is tuned for N -times overlapping, and $T = (N+1) \cdot R$.

As x is a large number, we may suppose that it is an integer multiple of N . (Otherwise the data sequences may be padded with extra data to the nearest integer multiple of N).

The total time of overlapping (pipelined) calculations can be expressed as

$$(x/N) \cdot g \cdot (N+1) \cdot R + N \cdot R.$$

The result may be explained as follows: N data packets can be processed using a recursive loop shared by N processes. Such a loop requires

$$g \cdot (N+1) \cdot R$$

time cycles for the first result to leave the loop; this is the first time cycle in which the loop is available for new data. The next $N-1$ results are put out of the loop after that time cycle with a period time of R , so subsequent packets may be fed to the loop with the same rate.

The system must process $x/N-1$ complete data N -tuples before the last packet. The last packet features $N-1$ packets before itself so the total execution time is the sum of the execution times for the N -tuples plus the time difference between the first and last initial data in the last data N -tuple (equal to $N \cdot R$ time cycles),

$$(x/N) \cdot g \cdot (N+1) \cdot R + N \cdot R.$$

This total execution time is less than

$$g \cdot T \cdot x$$

(time required without loop overlapping). From the expressions we may decide whether it is worth to apply this method or not. If the ratio of the execution times is less than 1, then the new design solves the problem faster. To check this, the following feasibility inequality must be

$$(x/N) \cdot g \cdot (N+1) \cdot R + N \cdot R << g \cdot T \cdot x$$

As x is usually greater than 1 (the system is designed to process a large number of packets), for the left side of the above inequality

$$(x/N) \cdot g \cdot (N+1) \cdot R >> N \cdot R$$

holds. As this is true, neglecting $N \cdot R$ from the left side of the feasibility inequality yields

$$R/T << N/(N+1).$$

This inequality shows that the more the restarting period is decreased against the loop duration, the more efficient structure will be achieved and from the left side it can be seen that as more and more data is introduced to the structure, the result can be more and more efficient.

3) Infinite loop depth

Infinite loop depth is presented in the case of continuous calculations. Robots, for example, need to calibrate the dynamic model of their environment periodically, as long as the robot is moving. In this case the overlapped loop execution is usually slower than the non-pipelined version as modifying the loop for overlapping increases T (as it inserts buffers to the data path). Note that the loop itself becomes slower during scheduling, but this leads to an increase in speed as it may process more than one packet of data simultaneously.

One of the advantages is the reduced cost of hardware components, which is generally possible as the nature of calculations is identical for every process. Absolute time gain is therefore expressed in the terms of realisation costs, with a typical 3-process robot subsystem presenting only one fully utilised loop instead of three, partially idle systems.

8.4. External synchronisation

External synchronisation will be illustrated on an example (Fig. 31.) which is used for data filtering in voice-transfer processes. The graph checks for the predictability (and the compressibility) of voice transfer, which may be used in monitoring digital compressed audio transmissions and for regulating sampling time.

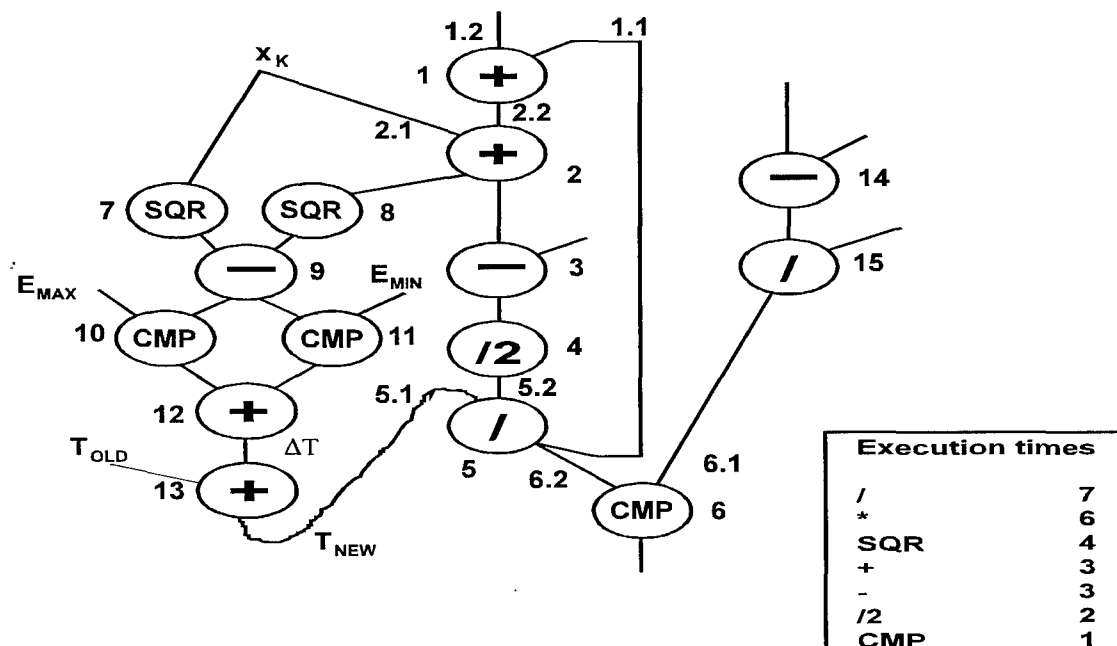


Figure 31.

Data path of a system that monitors compressed voice transmission

The system is a model of the following:

1) Branch 7-8-9-10-11-12-13 is responsible for the adaptive tuning of the sampling time. This choice is based on the difference in signal energy:

Operation 7 (SQR) produces the square of the signal value $x(k)$, while operation 8 takes another signal value and squares it. The difference of these signals is compared to two values: to a maximum in operation 10 and a minimum in operation 11. Should the difference exceed a given value, operation 10 outputs a negative value as ΔT ; operation 11 outputs a positive ΔT value if the difference is close to zero. These two factors are added to find the final change in sampling time in operation 12 (the method increases sampling time for a signal that does not change in a long time and conversely increases sampling frequency if a signal contains high-frequency components). The output of operation 12 is added to the previous sampling time in operation 13 to update the value.

2) Branch 1-2-3-4-5 is the approximation of the signal based on its previous behaviour. This is a model which approximates the current value as if it would follow the average of previous samples. (It follows signal behaviour based on a digital low-pass filter, comparing the result to the actual observations (branch 14-15)).

3) Branch 14-15 measures the real signal. Its output is the slope of the signal, which is compared to the slope of the approximation in operation 6.

This recursive algorithm contains all the possible data conflicts can arise. The synchronisation problems are negotiated here in the time order which eliminates any iterative steps during the scheduling process. In this example two different, but not independent loop exist (the first follows the 1, 2, 3, 4, 5, and the second, the critical path follows the 1, 2, 8, 9, 10, 12, 13, 5 sequence). In the first step the shorter loop must be tuned to the critical recursive path, inserting extra buffers to the shorter path. In this example it means that 7 buffers must be placed before operation number 5, to the 5.2 part.

The synchronisation problem can be simply solved by several algorithms before elements 2, 3, 9, 10, 11, 12 and 15, handling the whole graph without feed-backs. The value of the T must be calculated after this step as the synchronisation may introduce delay elements to the critical recursive path, which increases T .

Another class of the synchronisation problems is when the receptor element uses the result of the loop (receptor 6 in Fig. 31.). In this case the algorithm is loop-dependent, because the result of the loop is available in different time cycles depending on the type of the loop (see in section "Classification of recursive problems").

This problem can be eliminated by inserting a shift register which can produce any of the stored data in any time step depending on the control of the system. The length of the shift register is determined by the worst case (the longest possible loop execution time). For the problems of the

first class (see in section "Classification of recursive problems") it modifies the structure of the control logic. In this case the control logic picks up the suitable data from the shift register when the recursive loop finished an iteration.

9. Control principles

The scheduled and allocated EOG represents the structural design of the data path only. A proper control is required additionally to co-ordinate the elementary operations according to the EOG. This control has to ensure the correct data dependence and timing not only between the processors, but also inside the processors between the elementary operations covered by common processor. Data multiplexing and demultiplexing are also to be controlled for establishing the required data connections between the processors covering more than one elementary operations. It is also a control task to synchronize the starting points of time and to distribute the input data for the copies of multiplied operations. The control models are classified basically as **centralized** and **distributed control path**.

9.1. Centralized control path

Let the EOG in Figure 32 be considered as an example. Without detailing the scheduling and allocation steps (the duration times are not shown), let the processors **P1...P6** be assumed as a result for the cover of the elementary operations as follows:

P1:e(1),e(5),e(10) **P2:**e(2),e(7) **P3:**e(9),e(3) **P4:**e(4) **P5:**e(6) **P6:**e(8)

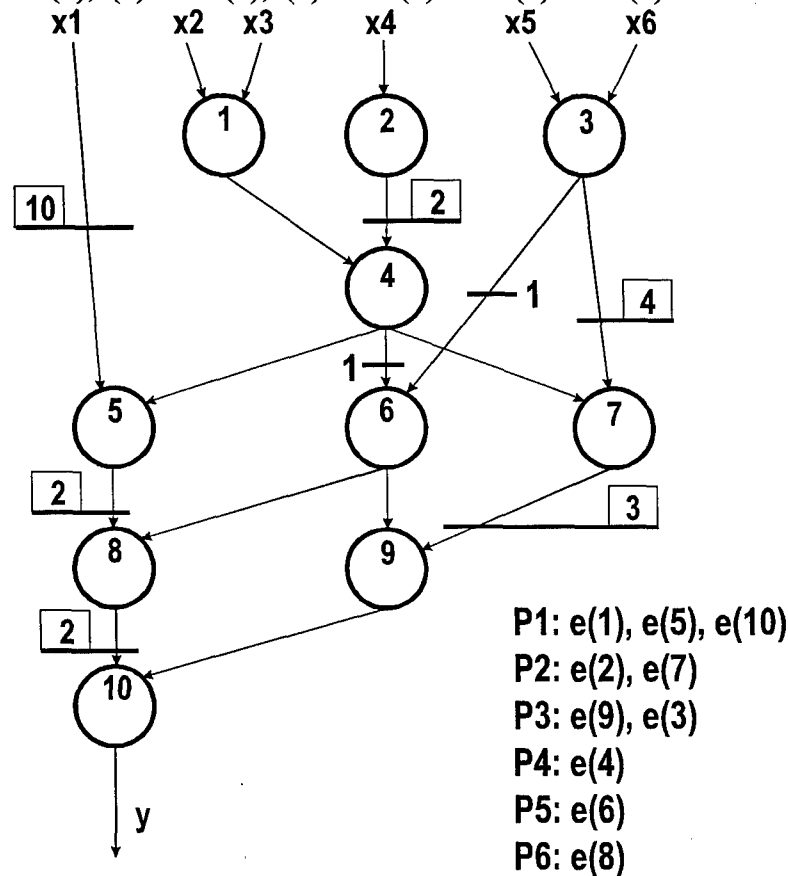


Figure 32

Example for covering the EOG by processors

In Figure 33, the internal structures of processors are illustrated only symbolically. The elementary operations covered by a common processor are dotted, because they generally cannot be separated any more in practical realisations of the processor shared by them. Only due to this symbolic separation is it allowable to neglect the data multiplexing and demultiplexing inside the processors. Obviously, the buffer registers and the delay effects generated by the algorithm RESTART and SYNC need also proper control signals in order to satisfy their timing constraints. The **centralized control path** is basically a **counter** driven by the system clock. The actual content of the counter determines the elementary operations and buffers to be started by producing the proper start signals (**st1...st18**). These output signals of the control path must be generated in **1-from-n (one-hot) code** by a modulo **L** counter. In non pipelined mode, each EOG component has a single start signal, since it is started at most once during the time corresponding to **L**, since each **e(i)** receives a starting pulse (**st..**) in the **b(i)**-th clock cycle. In a pipeline mode, however, an elementary operation is started more times during **L** depending on the value of **L/R**. In this case, each **e(i)** is affected by more start signals which can be considered as to be connected in logic OR. (No such case is illustrated in Figure 33). The pipeline starting of each **e(i)** occurs in every **b(i)+k*R**-th clock cycle, where **k** is an arbitrary integer.

Note that the serial numbering of the start outputs in Figure 33 does not always reflect the order of magnitude of the counter content. For example:

$$(st11) < (st10),$$

where the brackets stand for counter content generating the given start signal. It is trivial that **e(9)** must be started by **st10** later than **e(3)** by **st11** according to the data dependence prescribed by the EOG.

As it has already been shown in Chapter 3.2, multiplied elementary operations need special control considerations. In Figure 34, the **internal control path** is based on a modulo **2*c(i)** counter and the **st1...st2*c(i)** outputs must be generated in **1-from n (one-hot)** code also in this case. The input **sti** receives a pulse from the main control path each time, when the start of the multiplied **e(i)** is required. Therefore, each **sti** pulse must generate exactly two subsequent start pulses at the outputs **st1...st2*c(i)**, the first one for the input buffer and the second one for the copy being just in turn. This is the task of the **count control unit** in Figure 34.

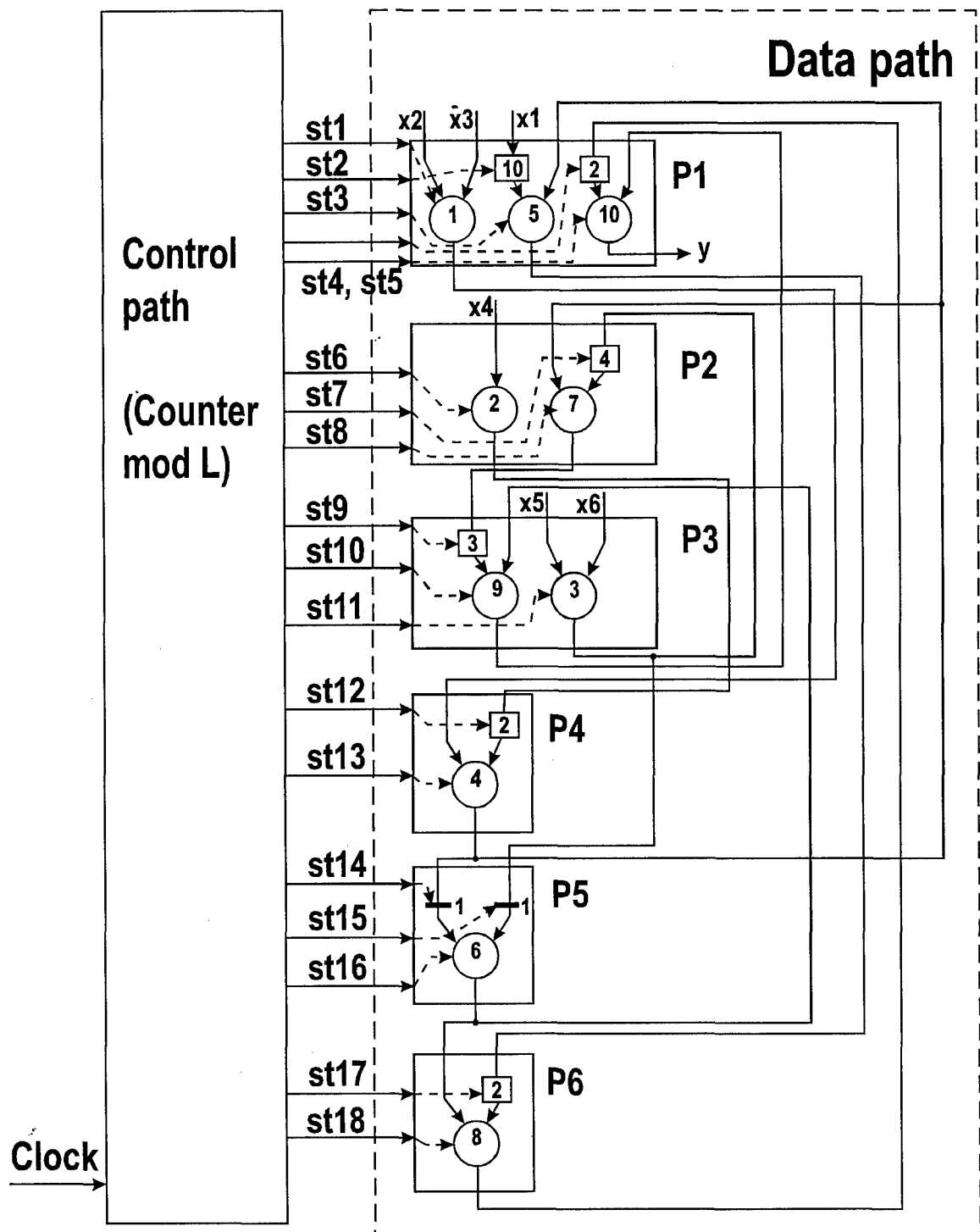


Figure 33

Centralized Control Structure for the EOG in Figure 31

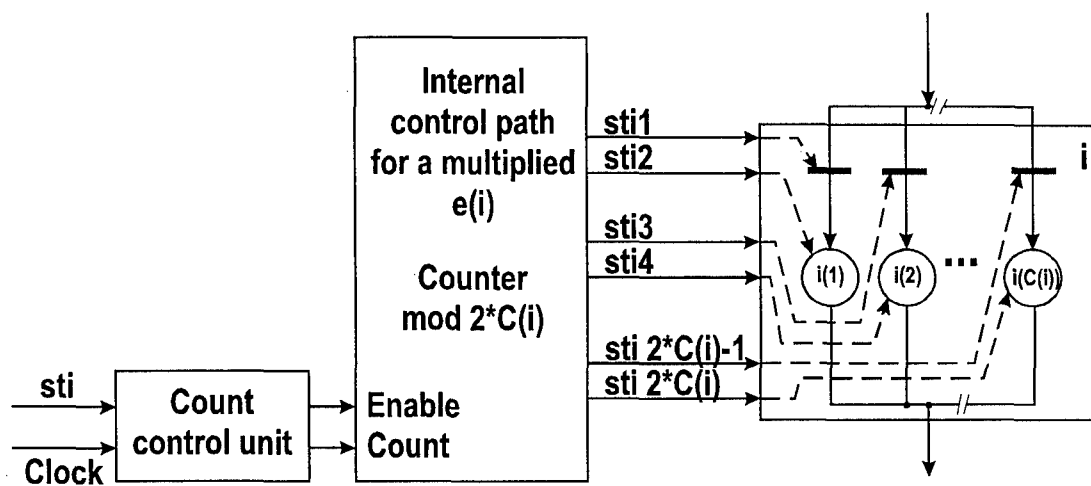


Figure 34

Control structure for multiplied operations

The main advantage of the centralized control structure is the simplicity of the implementation. In most cases the realisation is a **Johnson counter**. Outputs of this counter may be directly connected to the corresponding start signals. Since this kind of control path is relatively simple and the model matches with the internal structure of most of the processors, the extra silicon area occupation is small compared to data path operations. The centralized control has a disadvantage. If the EOG contains a large number of operations, then the generated VLSI die will be huge and the interconnection delays of the controlling signals cannot be neglected. This delay decreases the highest operation frequency. It is especially true if the target technology is Multi Chip Module and the design does not fit onto one die, since the I/O pads cause additional delay between the functional elements. To avoid this problem, the **distributed control structure** should be used.

9.2. Distributed control path

The centralized control path can be eliminated by distributing the control task among the elementary operations. The dataflow-like character of the EOG and the constraints for the operations (see Chapter 2) require that the start signal (**sti**) of an **e(i)** must be generated only then, when every **e(j)**, for which **e(j) → e(i)**, has already finished the operation. Thus, each elementary operation must take part in generating the start signal of its next neighbours determined by the data connections (edges) in the EOG. For this aim, the busy time intervals of the operations (see Chapter 7) can be represented by signals on extra 1-bit connection lines chained through the EOG and accompanying each data connection. Such an extra edge as **busy line** can specify the busy state of **e(i)** by the Boolean signal value **B(i)**, as follows:

B(i)=1 if and only if **e(i)** is busy, i. e. $b(i) \leq t \leq f(i)$, where **t** is the time parameter, else **B(i)=0**.

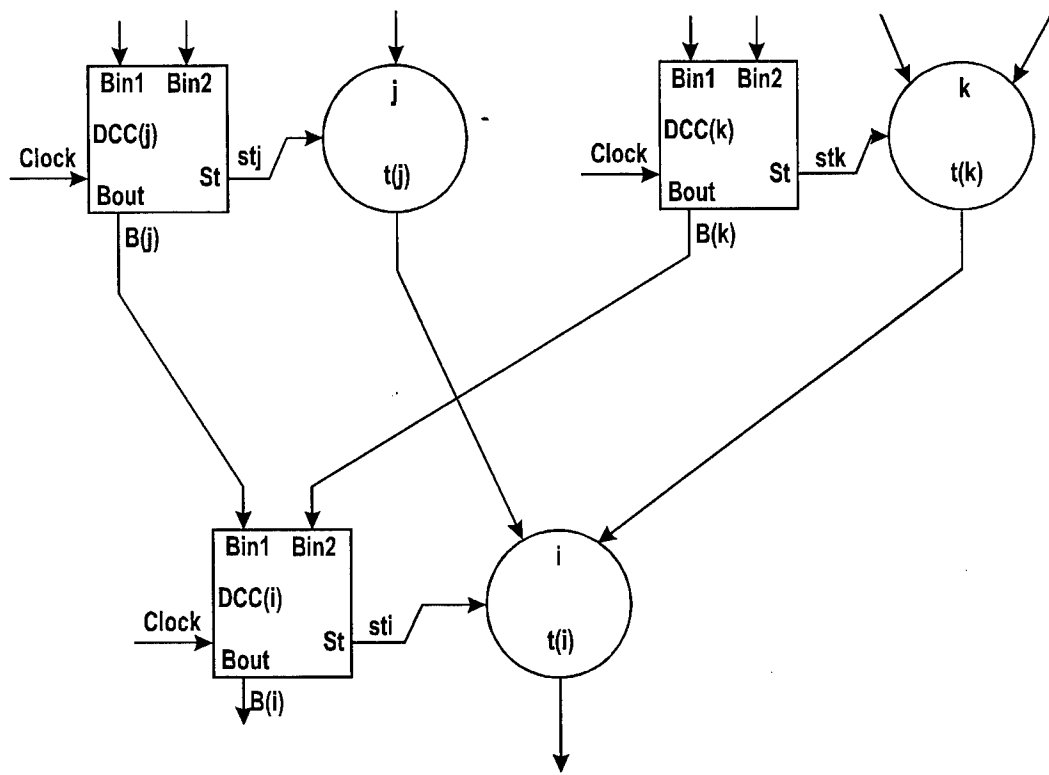


Figure 35

Completion of the EOG with distributed control cells

To evaluate the busy signals from the direct predecessors and to generate its own start and busy signal, these are extra tasks for **e(i)** in the case of distributed control. Therefore, each **e(i)** supposed to be provided with an extra **distributed control cell DCC** as shown in Figure 35. The required function of a **DCC(i)** is to generate the start pulse **sti** in the same timing as the centralized control path does it. It means that the system clock must be taken into consideration as time base, consequently, **DCC(i)** can be specified as a synchronous sequential machine. The specifications of all **DCC**-s are identical except the duration of **Bout=1** which is determined by the duration (execution) time of the operation. With the input-output notations in Figure 36, the task of a **DCC** can be formulated as follows:

The cell waits for the falling edges on **Bin1** and **Bin2**. After having received both of them, the later one enables the next single clock pulse to the **st** output as **sti**. The falling edge of this single clock pulse activates **Bout=1** as **B(i)** which lasts until **t(i)** clock cycles. Only these output changes can occur, but only then, when new falling edges arrive on **Bin1** and **Bin2**.

Such a simple sequential machine is to be realized to each elementary operation for performing the distributed control yielding a **mixed data and control path**.

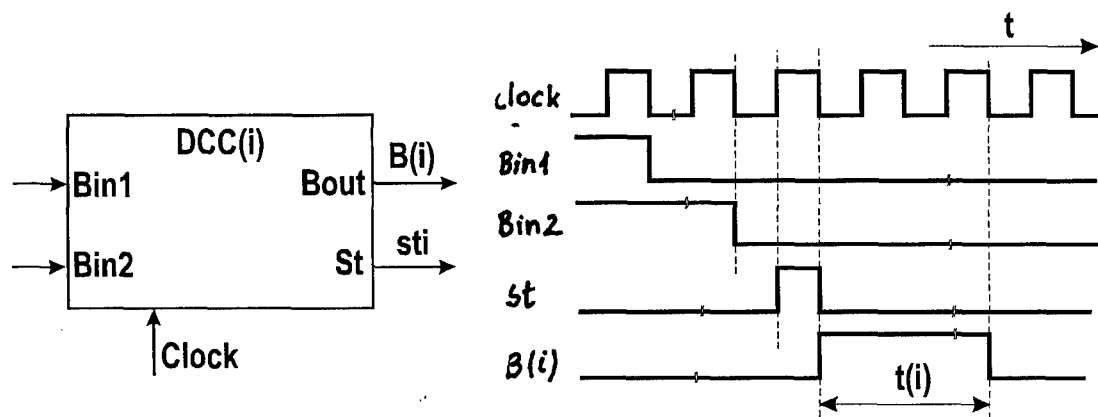


Figure 36

Specification of the distributed control cell DCC(i) as a synchronous sequential machine

After the allocation step, the processors may represent many elementary operations also with the corresponding DCC-s. Obviously, the DCC-s inside the same processor can be realized only by a single sequential machine, but the multiplexing and demultiplexing of its inputs and outputs, and the different **Bout=1** durations must be ensured.

If **e(i)** receives the input data of the system, then the inputs **Bin1** and **Bin2** of DCC(i) are to be connected to the output **Bout** of the DCC belonging to one the elementary operations which produces the result data at the end of the longest transfer sequence. In this way., **e(i)** can receive new input data only after a time corresponding to the latency, i. e. it is a non pipeline mode. In a pipeline mode, the inputs **Bin1** and **Bin2** of DCC(i) must be generated from outside as often as **e(i)** is to be restarted with new system input data.

The above DCC principle could be applied for realizing the distributed internal control of multiplied operations, but in practical cases, no simpler solutions can be obtained, then the counter-based control path in Figure 34, since the practical values of **c(i)** need a relatively short counter.

10. Scheduling methods

During the course of high-level synthesis the step directly following precise task description is the phase of scheduling and allocation. This phase requires an exact description of the problem using the data path, a graph representation of data dependencies and propagation in the system. In this graph nodes are elementary operations, with the directed branches between them indicating immediate data dependencies.

Some of the elementary operations are reading system data entry ports while others output data directly to system outputs. The delay between reading the first system input and writing the last system output port is system latency (L). Latency is equal to the execution time of the longest sequence of branches present in the data path. It is usually expressed as an integer value, the time divided by the time unit (a cycle). The minimum of latency in a given system may be found by examining the graph. Note, however, that actual latency may be extended with no limit by inserting delay to the data path.

The hardware unit is considered to operating by introducing data periodically so that new data values are written to system input ports every R cycles, where R is restart time, expressed in the same units (cycles) as latency. A restart time exceeding system latency means a low utilisation for the system as there are idle cycles when the elements are not working at all. In a non-overlapping mode of execution restart time is equal to system latency, data is fed to the system when the previous set of output values has been calculated. With the exception of simple systems a non-overlapping execution mode works with data propagating sequentially through the data path, leaving elementary operations in an idle state until the arrival of the next data packet. The ratio of these inherent idle cycles may be decreased using the pipelined execution method.

The overlapped (pipeline) method of feeding data to a system is the case where restart time is less than latency. It is possible to use pipelined execution in systems where the idle state of processors propagates so that data belonging to the next data packet does not disrupt previous output values before they are read from the output of the operation. The lowest "safe" restart time may be found using the execution time of the neighbouring operations. As the pipelined execution mode results in an increased utilisation, data throughput is increased in such systems.

Structural and functional pipeline are two expressions often encountered in scheduling, even if their meaning is not always made clear. Functional pipeline refers to the system operating in the previous way i.e. the data graph itself is used in a pipelined way, with the data operations themselves executing without internal overlapping. In this method the graph itself serves as source of the overlapping.

Structural pipeline presents another method to speed up execution. In this description elementary operations may feature internal overlapping, so that they are composed of multiple

stages. An operation becomes available to receive data as it empties its first stage, so the apparent execution time is less than the actual time needed to complete the calculations. As this method uses operations that are complicated (i.e. they may not be called elementary), structural pipeline is outside the scope of scheduling.

Note that operation multiplication works like structural pipeline, but it no longer refers to the complicated processors as elementary. **Superscalar design** is frequently used to denote a processor that is internally multiplied, usually like "n-way superscalar" for a processor that has n parallel branches.

The Intel Pentium processor may be a typical operation (processor) in a complex data path as its internal two-way superscalar structure is almost equivalent to a multiplied operation with $c=2$. (This is only an approximation as the two execution paths in the Pentium chip (the U and V pipes) are not completely identical. The difference is based on some of the complex instructions that have to be performed by the U pipe.) From the user's point of view this internal structure is visible as a CPU that has approximately twice the data throughput of a similar 486 system, as the U and V pipes are similar to 486 systems in processing power.

10.1. Stages of scheduling and allocation

A system graph before scheduling is a functional description of the system where no properties of the elementary operations are prescribed. The scheduling problem may be described as finding the timing of elementary operations in such a way that the graph is suitable for a feasible hardware realisation. The stage following scheduling is allocation, where the scheduled graph is tested for feasibility. A graph is feasible if it may be built under without violating one or more constraints set by the operating environment or manufacturing process. As scheduling is using information based on some the knowledge of the allocation method, some allocation steps may be performed before scheduling (initial allocation). Outer constraints of the system are to be taken into account during this stage.

The usual types of constraints are latency, restart time and hardware. Latency must be kept under a feasible limit in systems where delay between writing system inputs and reading system outputs must be limited. Digital controllers used in process control are typical applications where latency is a lower limit for controller dead-time, an unpleasant property of digital controllers. As an increased value of latency presents additional problems to the design of process control, it must not be increased over a feasible value. It is possible, however, to design for a given latency value and find that the optimal solution does not use all the available latency. (Fig. 37.)

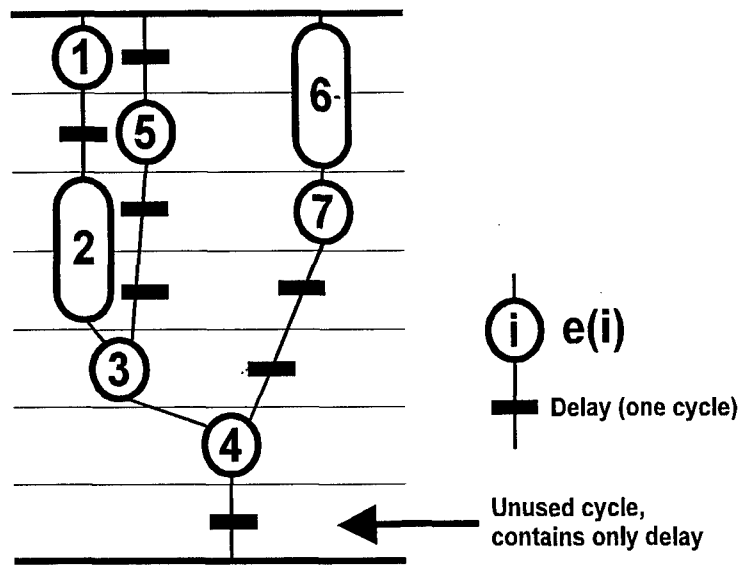


Figure 37.

Visual representation of a scheduling plan

Restart time is critical in applications where data must be fed to the system with a given frequency. Data acquisition units are classified using frequency, not latency. In these systems reduction of restart time to a lower value is a gain worth the increase in latency, staying under a feasible value. Decreasing restart time usually increases latency, as it involves inserting delay to the data path (unless the system was capable of running with the desired restart time).

Hardware limits are encountered in the design of systems where the realised hardware is bounded in some way, be it power dissipation, silicon area or cost. Applications used in extreme environments belong to this category. A data acquisition unit mounted on an information-gathering satellite is a typical member of this class, as satellites are usually designed to stay under power and mass limits.

The best solution is to compromise between the conflicting system properties with the dominant bound given an edge over the others.

10.2. Initial allocation

After finding the graph representation of a problem the properties of the elementary operations should be found. During the creation of the system graph only the functions performed by the operations are fixed. It is the responsibility of the designer to find the physical properties of the processors. As the actual representation of hardware capable of performing the desired function depends on system constraints, selection of the module library must precede scheduling. A bit-serial multiplier (slow, cheap but simple) may be a better solution in a flow meter than a parallel model (faster, more expensive, requires more silicon) while a parallel has the advantage of speed to be exploited in a PLC.

Choosing the module library fixes execution time for the elementary operations as the capabilities of a given construction are usually known. Knowing the modules also enables the designer to set up classes for the elementary operations, which contain elementary operations that may be realised in the same kind of processor. Addition and subtraction may be performed in the same kind of processor with an additional control bit carrying the sign of one of the inputs. It is the responsibility of the control hardware to supply the additional control signals required to implement multiple operation types.

After finding the module library, the data graph may be described using the time frame of its elementary operations. As the execution times of the operations are fixed by now, a unique limit to the earliest and latest execution times of a given operation may be found. These times are called ASAP (As Soon As Possible) and ALAP (As Late As Possible) values. The ASAP value is the maximum of the length of data dependencies between the elementary operation and system inputs (considered to appear at cycle 0), while the ALAP value is L minus the longest operation sequence between the operation and the system output (Fig. 38.). Operations may be started at any of the time cycles between their ASAP and ALAP times. An operation with its ASAP value (set by its predecessors) exceeding its ALAP time (bound by its successors) is violating the latency constraint of the system, and requires repeated calculation of the latency.

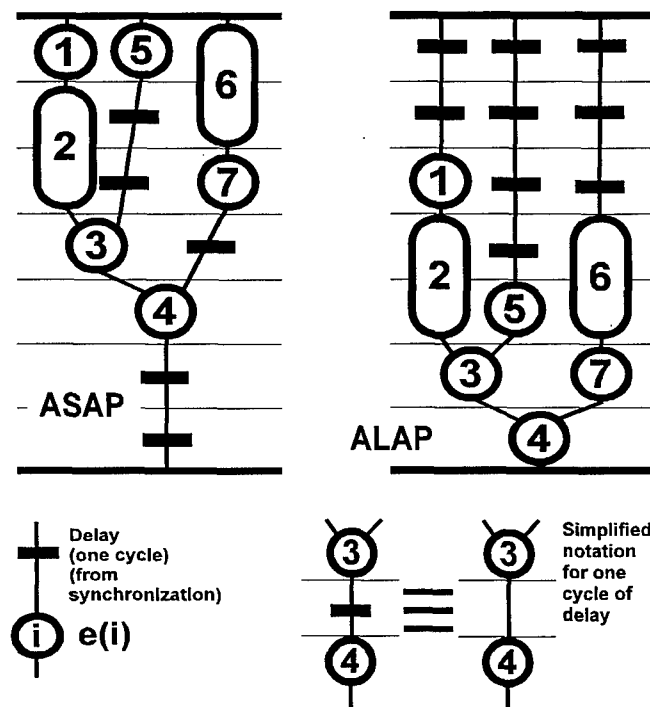


Figure 38.
ASAP and ALAP values if $L=7$

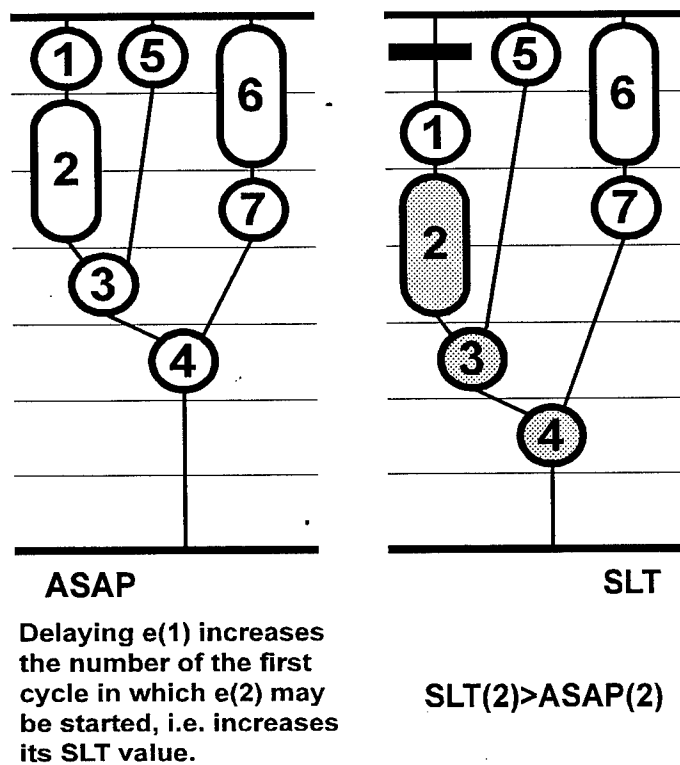


Figure 39.
ALAP and SLT

As ASAP and ALAP values are properties of the graph, they may be found using the properties of the data dependencies only. The first and last possible starting cycle may change for an operation, depending on its predecessors and successors (i.e. inserting delay to the graph). This change of time frame may be expressed as change in the ASAP and/or ALAP values of the operation. For this reason, it is feasible to introduce a set of **current (actual) limit times**, which are the actual boundaries of the time frame. During scheduling, the ASAP and ALAP cycles may not always describe the current time frame of an operation (since inserting delay after an element decreases its ALAP cycle), so the terms **Soonest Limit Time (SLT)** and **Latest Limit Time (LLT)** are used to describe the actual properties. (Fig. 39.)

An operation with an equal value as ASAP and ALAP is blocked in its place, with no freedom. As this operation may not be moved without violating global timing constraints is not subject to scheduling. From now on we do not deal with the scheduling of these so-called **fixed graphs** (graphs where all the elementary operations are fixed), as they are completely defined prematurely and so impossible to improve by scheduling.

The operations with an equal ASAP and ALAP value form the critical transfer sequences of the graph. Inserting additional delay to these paths is impossible if the system is to be designed under a latency constraint.

10.3. Initial approximation of the optimal solution

Even if the composition of an optimal scheduling plan is unknown, it is possible to find the result of an optimal set of SLT and LLT values, i.e. the minimum of hardware cost in the following sense:

Initial approximation is possible as the total number of utilised processors does not change if an operation is delayed or brought forth (Fig. 40.). A load presented by an operation and the processing power of a processor may be measured in units: one unit is **defined** to be **one** processor's worth of processing power in **one time cycle**. This way an operation (which always executes in one processor, as it is elementary and so may not be split) executing for $t(i)$ time cycles occupies

$$t(i) * 1$$

processing units as it uses the resources of one processor for $t(i)$ cycles. N of these operations, executed simultaneously, would use

$$t(i) * N$$

units. It is obvious that the scheduling of an operation does not affect the total units needed to process the data as it depends on only the execution time, which does not change during scheduling. Unit requirements of a given type of operations may be added to find the total number of units required to complete all the elementary operations of that particular type.

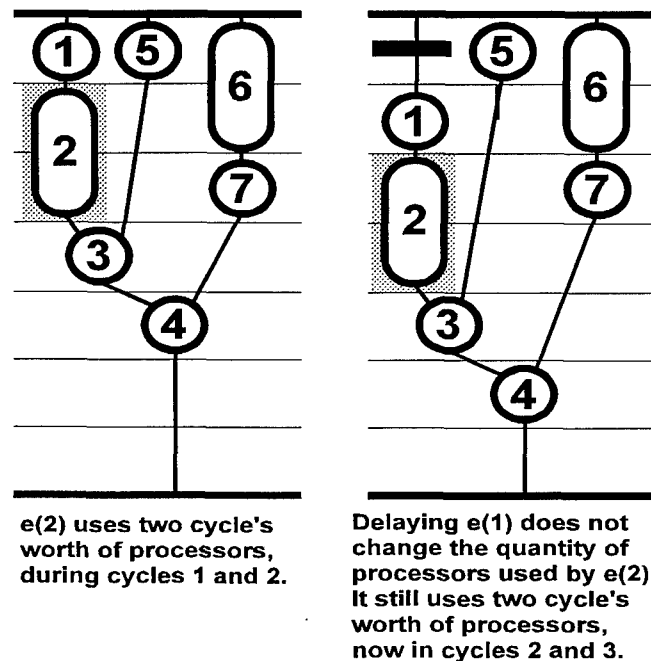


Figure 40.

Processor utilization

The total load of a data dependency graph must be compared to the processing capacity

offered by the processors in the system. A processor, when built, is available to suitable operations in the graph (those with an appropriate type) in all time cycles, i.e. between time cycles 0 and L-1. N processors of a given type offer $N \cdot L$ units of processing power to be used by operations.

During pipelined execution, time cycles are mapped to the 0..R-1 domain. As the pipelined system accepts data every R time cycles, the units offered by the built-in processor capacity are grouped so that a processor is considered to offer its capacity for R time cycles. N processors of a given type offer a sum of $N \cdot R$ units during overlapping execution.

This way in the implementation stage, the built-in processor capability is equal to the number of processors multiplied by latency (restart time for pipelined structures), as every processor adds 1 cycle's worth of processor capacity to total system capability for every cycle in which it is available. This must be equal to or greater than the sum of utilised units. Subtracting the number of units utilised by operations from the total units in the implementation equals the unused (idle, never utilised) units. This number may not be changed by altering the timing values of operations (as we have already proved that the load of an operation (expressed in units) may not change during scheduling). The only factor that affects the number of these unused units is the number of processors, which may be a non-negative integer only. Changing the number of processors changes the number of available units with L or R.

Any solution wasting more than L (or, in case of pipelined execution, R) total units means it is not optimal, possibly subject to iterative tuning of initial allocation and scheduling. This is easily proven as if the number of unused units exceeds L (or R), decreasing the number of processors by one decreases available processing power by L (or R) units, which still exceeds the number of units needed.

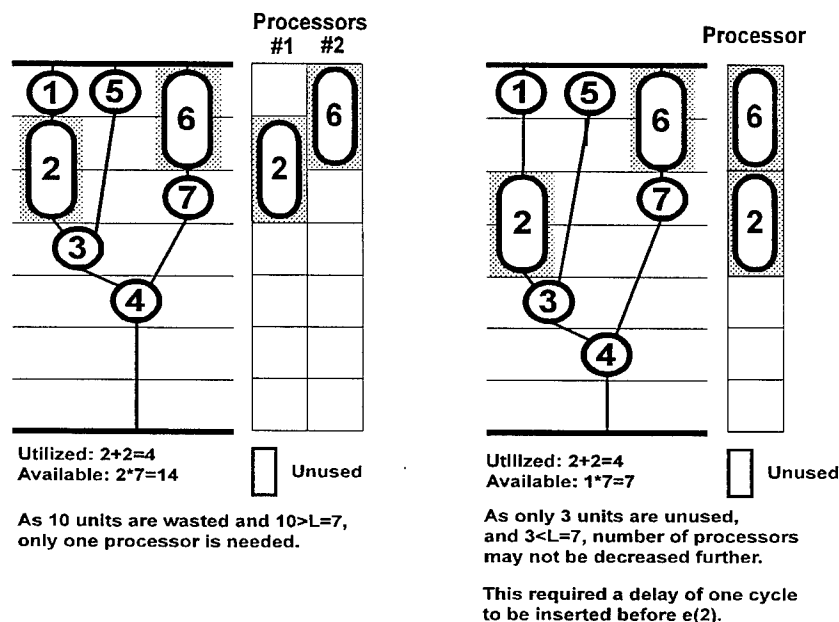


Figure 41.

Some scheduling algorithms (notably the hardware-constrained list scheduling) generate a solution that exactly matches the prescribed hardware utilisation properties. The disadvantage is the increase of latency, which may be checked only after the maximum is reached.

For these algorithms, an iterative process may be used that starts off with the optimal number of processors for all of the processor types, relaxing the bound until the latency requirement is met.

10.4. Scheduling

Following the phase of initial allocation the graph to be scheduled may be described using a set of ASAP and ALAP values, a description of processor properties and classes. Solution of the scheduling problem removes any degree of freedom in time domain from the graph by prescribing a fixed starting time for the operations. This transformation strips any mobility from the elementary operations while pre-processing the graph to be available for a successful allocation, resulting in an increased utilisation of the hardware.

Finding the optimal scheduling plan is unfortunately an NP-complete problem. As the optimal solution of NP-complete problems may be found in a time that is increasing as an exponential function of system complexity, a suitable polynomial-time replacement is needed. Solution methods based on heuristics are used to achieve a scheduling plan near the optimum in less time.

Integer Linear Programming (ILP) based methods transform data dependencies and processor properties into linear systems of equalities and inequalities. Solution of these systems may be performed by an external solver package or internal resources. As ILP-based solutions are using external resources during the calculations, the time required to complete such a task may not be easily forecast. ILP-based scheduling may be extended to handle pipelined execution.

List scheduling is a collection of fast but not optimal methods. Elementary operations are scheduled based on a priority function, with the operation with the highest priority being scheduled before the others. The priority function is generally based on operation mobilities, i.e. related to the urgency of the given elementary operation. List scheduling usually ignores any special properties of the graph (i.e. pipelined execution) and produces a fast, but not always optimal solution. Execution time is a low (first or second) order polynomial function of graph complexity.

Force-directed scheduling is a scheduling method with a polynomial (third order) function of graph complexity. This scheduling algorithm schedules operations based on concurrence as a function based on the utilisation of individual processors. As a constant utilisation is optimal,

force-directed scheduling tries to balance the number of concurrent operations, with a cost function based on the deviation from the ideal concurrence. This cost function resembles the force exerted by springs based on Hooke's Law (hence the name force-directed).

Force-directed scheduling is a modular method, easily modified to suit special needs. Possible extensions are optimisation for bus width, transfer (memory), conditional branches and more. All the modifications are introduced as additional components of the cost function.

10.4.1. Scheduling using Integer Linear Programming (ILP)

A possible way to solve the scheduling problem is to transform data dependencies to equalities, which in turn may be solved using any software or hardware designed for the task. As a solver may be fine-tuned for the ILP problem, such a solution needs to concentrate only on the scheduling process itself.

An ILP-based scheduler uses a set of linear equations and inequalities to describe system behaviour. The variables are related to the starting times for the elementary operations, with other constants and variables representing external constraints. Additional equalities are used to make up data dependencies, inhibit violations of time frames and finally to define a cost function needed to qualify results.

10.4.1.1. Equations for an ILP-based scheduler

In a given graph, after calculating the $s(i)$ ASAP and $l(i)$ ALAP times for elementary operations, all operations (from 1 to N) are assigned a number of binary variables, $x(i,t)$ in such a way that $x(i,t)$ is 1 if the i . operation is started in the t . time cycle, 0 otherwise. The system is considered to have a latency of L , operating with a restart time of R . A maximum cost of D serves as an upper limit for realisation. J is the number of processor types, with a weight factor of $w(j)$ belonging to the j . processor type. The weight values are relative values, presenting a way to describe the quality of a given solution. The number of processors of type j to be built into the system is $M(j)$, operation $e(i)$ uses a processor of type $j(i)$.

As we permit a different execution time for each of elementary operations, the i . elementary operation started at the t . cycle ($x(i,t)=1$) requires $t(i)$ time cycles to complete its task. For this reason, the processor in which $e(i)$ is executing remains inaccessible for new data during the t $(t+t(i)-1)$. time cycles. Any data arriving to processor input ports ruins the calculations. To describe the occupied state of the processor, an operation started in cycle T is considered to be occupying one processor for every $t>T$ time cycles s. t. $x(i,T)=1$ and $t-T<t(i)$; this is the definition of $x'(i,t)=(1:t-T<t(i))$, otherwise the value is 0.

Our sets of equations are constructed in the following way:

- a) All elementary operations must start in one of their respective time frames, so

$$\text{sum}(x(i,t):s(i) \leq t \leq l(i))=1.$$

This property removes a number of trivial zero variables ($x(i,t)$ outside the time frame), reducing system complexity.

- b) All operations are started in such a way that the number of simultaneously used processors of type j does not exceed $M(j)$, for every possible j : $1 \leq j \leq J$. This is described using a set of inequalities: $\text{sum}(x'(i,t):j(i)=j)-M(j) \leq 0$, where $x'(i,t)$ refers to the utilisation of the processor executing operation i during the t . cycle, corrected based on $t(i)$. t scans time cycles 0 to $L-1$.

- c) Data dependencies are not violated, so all operations are started at least $t(i)$ cycles before their direct successors. In other words, for every $e(a)$ and $e(b)$ pair such that $e(a)$ is a direct predecessor of $e(b)$:

$$\text{sum}(t*x(a,t):s(a) \leq t \leq l(a)) - \text{sum}(t*x(b,t):s(b) \leq t \leq l(b)) < 0.$$

As data dependencies are transitive, it is not necessary to set up inequalities for indirect connections.

- d) Total cost of the system does not exceed its allowable maximum:

$$C = \text{sum}(M(j)*w(j):1 \leq j \leq J) \leq D.$$

Point a) yields a set of N equations, b) leads to a set of $J*L$ inequalities, c) results in an inequality for every direct connection, d) presents an additional inequality. As the number of variables is equal to the total cycle number of the time frames, number of inequalities is rapidly increasing as a function of N .

ILP-based solutions may be extended to handle pipelined execution mode by modifying point b) and d). Instead of $M(j)$ we introduce the overlapping utilisation $M'(j)$, which is equal to the simultaneously used processors in the case of pipelined execution. These numbers may be generated by folding the $x'(i,t)$ utilisations so that they represent the periodicity based on R , i.e. an operation started in $t=R$ adds one to the utilisation in time cycle 0. (All cycles are transformed to the range $0 \leq t \leq R-1$.) This modified set is composed of inequalities of the following type: $\text{sum}(x'(i,t):j(i)=j)-M'(j) \leq 0$, one for every pair of j and t values such that $1 \leq j \leq J$, $0 \leq t \leq R-1$.

The virtual folding of the time domain may be performed by grouping inequalities belonging to time cycles in such a way that the inequalities of cycles i and j are added if and only if i and j

are folded to the same cycle i.e. $i \bmod R = j \bmod R$. This reduces the number of inequalities to $j \cdot R$, which is a significant decrease of system complexity, especially for slow systems ($R \ll L$). To calculate cost we simply modify d) so that

$$C = \sum (M'(j) \cdot w(j) : 1 \leq j \leq K) \leq D.$$

After composing the inequalities, the optimal solution must be found. This solution consists of binary numbers for $x(i,t)$, which in turn prescribes the optimal starting time for operations.

10.4.1.2. Disadvantages

ILP-based scheduling depends on an external system to solve the sets of inequalities. It is not usual to arrange a solution in the scheduler itself. While an external package may be highly optimised for the task, the time needed to find the optimal solution may not be predicted. It is feasible, however, to reduce the number of variables based on heuristics, which is a generally accepted method to speed up calculations. No universal rule exists, however to perform such a speedup. Hardware realisations are better defined as neural networks (Hopfield), as these are capable of solving linear programming problems.

10.4.2. List scheduling

List scheduling is a collective name for simple methods using relatively small calculating power based on primitive priority functions. Total number of steps to perform list scheduling is proportional to the first (at most second) power of system complexity. Elementary operations are put into a list based on their priority value, with the scheduling process scanning the time domain. Conflicts (contradictions caused by identical starting time for operations) are dealt with based on the priority function, with the operations having lower priority being delayed. This delay is equivalent to a single cycle performed by a buffer.

As the main execution order of list scheduling is scanning in time domain, list scheduling is not sensitive to the internal order of elementary operations. List scheduling requires a suitable priority function, which is based on the mobility of the operations. The mapping of mobility to priority should be (strictly) monotonously decreasing so operations with a lower mobility are at an advantage during scheduling. The easiest way to transform mobility to priority is to subtract it from a suitable positive integer value.

A useful extension of list scheduling is the case of resource-bound list scheduling. This algorithm penalises the violation of hardware cost constraint. A suitable priority function is based on a first-order decreasing function of mobility, with an upper limit set by operation concurrence. This composite function does not deal with a distribution not violating system constraints while it penalises the usage of additional hardware. An unfortunate disadvantage of

resource-bound list scheduling is the iterative way to find optimal hardware constraints (starting from the optimum, relaxing bounds until fulfilling latency condition).

List scheduling in itself does not involve analysis of pipelined structures. Multi-cycle operations may be scheduled using these algorithms, using a structure similar to look-ahead prediction. A possible solution of this problem is the introduction of long delay, lasting long enough so the placement of the delayed operation is not tried before the processor occupied by the winning operation is freed.

List scheduling depends on the internal resources of the scheduler to solve its problems. Digital hardware implementations are possible.

10.4.2.2. Disadvantages

List scheduling, being the offspring of a simple function is a very fast algorithm, terminates in a short time with a (usually) non-optimal solution. Compared to ILP-based scheduling, the total execution time may be approximated by an upper bound. The results received after list scheduling must be checked for activation and idle time. (If optimal hardware requirements are not found before.)

As list scheduling presents a priority function based on local relations (i.e. mobilities), it produces a local optimum for all of the time cycles, which in turn may yield non-optimal pipelined utilisation.

10.4.2.3. Execution of general list scheduling

A typical list scheduling algorithm scans data propagation in the system in time domain. It requires the latency and complete description of the graph, calculates ASAP and ALAP times and constructs the priority function. The SLT and LLT values are started as ASAP and ALAP cycle numbers. List scheduling simulates a straightforward method of delaying operations, which is performed by fixing high-priority operations and delaying others (i.e. increasing their SLT values).

The scheduler itself runs a loop for every time cycle, finding operations that could be started in the current time cycle. In the case of competition, operations are given advantage in decreasing order of priority. As this method starts with a fixed value of system latency, it is suitable for scheduling systems operating under an execution time (latency) constraint. Most of the practical applications fall into this category.

All the operations eligible for immediate start (winners) are started (i.e. their SLT and LLT values are set to the current time value), while other competitors (the losers) are delayed. This delay is usually performed by increasing their SLT. A delayed operation has got its priority

function value increased as its LLT time does not change (i.e. remains equal to the ALAP value) while delaying it increases its SLT. As the priority function is highest for the operations with the lowest current mobility, a general list scheduling method preserves the latency of the system. (It is impossible to win over an operation if it is currently in its ALAP cycle, as it would require a negative mobility.) Elementary operations that are affected by delayed operations through a data dependency are also subject to delay so data dependency relations are not violated.

Checking pipelined utilisation must be performed for systems operated in overlapping mode. The local nature of the list scheduling does not enable the scheduler to directly check for violations of hardware constraints in an easy way. A possible solution is to employ back-tracking, but this makes the list scheduler slower.

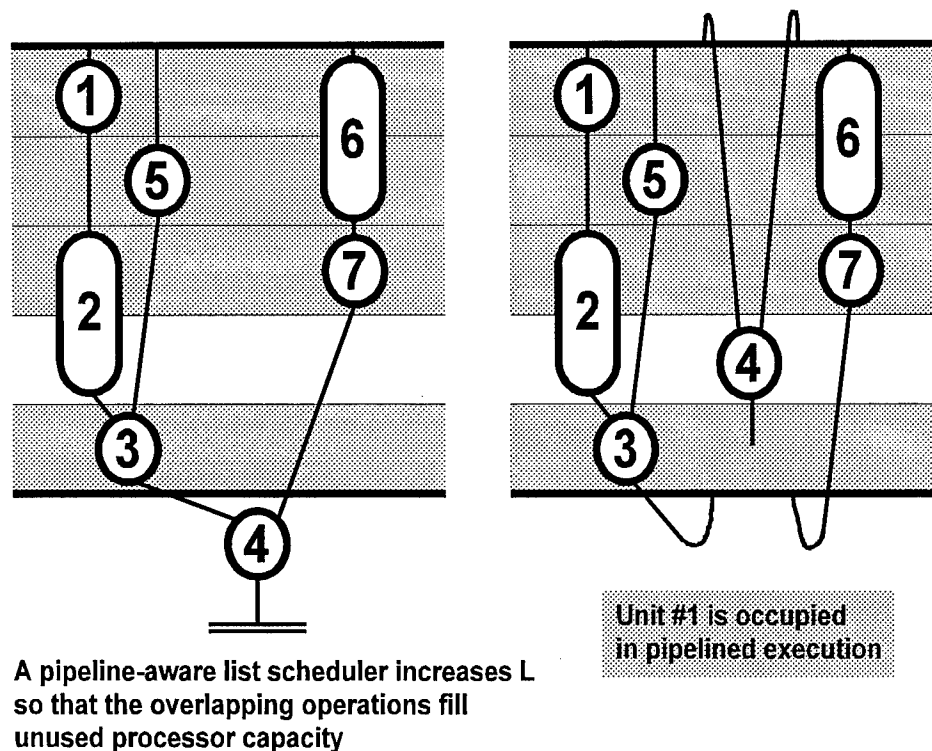


Figure 42.

List scheduling a pipelined structure

10.4.2.4. List scheduling under a hardware constraint

Systems that are to be realised under a fixed cost of hardware components may be scheduled using a modified type of list scheduling. This algorithm does not pre-set the ALAP (and thus the LLT) values of elementary operations, so total latency must be checked after fixing the last operation in its place. This hardware-bound scheduling method scans time cycles in increasing order, with the elementary operations checked for competition based on their SLT times. As long as the total number of processors required to start all operations with their SLT values

equal to the current time cycle stays lower than processor number constraint, they are all started in the current time cycle. In case of violation, the excess must be delayed. There are no generally useful rules to select the operations to be delayed, but a comparison of total execution time following a given operation may help. After scanning the data dependency graph the final value of latency may only be checked, but not modified.

List scheduling is capable of finding an exact match for the prescribed hardware constraints if possible. A feasible balance between hardware costs and latency must be found. Due to the fast execution of list scheduling, an iterative solution is not a problem.

As with the general list scheduling algorithm, list scheduling under a hardware constraint is incapable of dealing with pipelined execution in detail.

10.4.3. Practical applications of list scheduling with hardware constraint

A typical application of hardware-constrained list scheduling is code generation for a RISC-based processor. (Up-to-date CISC processors employ the same speedup technics, so an Intel Pentium or P6 (Pentium Pro) is considered to be RISC in this context.)

Most RISC processors use an internal multiprocessor structure, i.e. instructions are fed to *execution units* (we use this name to avoid confusion with processor, which now refers to the RISC CPU itself). Units are the resources that must be properly utilised as they may work simultaneously. A PowerPC 604, for example, contains three integer units, a floating-point and a load-store unit. Two integer units are single-cycle (i.e. basic operations: compare, add, rotate etc.) while the last is a multiple cycle unit performing integer multiply and divide instructions. The floating point unit houses instructions that all have a latency of three cycles. Load-store is used in address generation.

The 5 execution units are fed by an instruction fetch stage which scans program memory for instructions. It is capable of supplying maximum 4 instructions during one cycle, so in an optimal case a program sequence is executed in 1 cycle for every 4 operations it contains. Trying to load data to a unit that is occupied is called a *data overrun*; such an operation is prohibited and results in a delay which ends when the unit becomes available.

In examples S refers to a single-cycle, M# to a multi-cycle instruction of # cycles, F# to a floating-point instruction (length is # cycles) and L to a load-store operation. (!F) refers to the situation when the full fetch capability (4 instructions/cycle) may not be fully used due to data overrun in the F unit (in this case, the next instruction must wait). Units of type F, L and M are singular, while S units are built twice.

A sample program consists of four branches, which feature operations in memory that are in a prescribed order:

Branch 1 is L-S-S (operations 1, 2 and 3)

Branch 2 is M3-F2-M3-L-S (operations 4, 5, 6, 7 and 8)

Branch 3 is S-L-S (operations 9, 10 and 11)

Branch 4 is L-F2-S-S (operations 12, 13, 14 and 15)

The operations of a branch must follow each other in strict order in memory, but there is no prescribed order of the branches. They may even be stored mixed.

A program of the following composition would be executed in 6 cycles (indices refer to operation numbers):

M₃₄F₂₅L₁S₂S₃L₁₂S₉F₂₁₃L₁₀S₁₄S₁₅M₃₆S₁₁L₇S₈

which sequence would be fed to the processor as

(cycle 0) M₃₄-F₂₅-L₁-S₂ ends as the 4-instruction fetch frame is exhausted.

(M₃ keeps on executing in cycles 1 and 2, F₂ in cycle 1.)

(cycle 1) S₃-L₁₂-S₉ (!F) ends as the following instruction is of type F, which causes a data overrun in the F execution unit. (It may not be loaded as the F unit did not yet finish its previous calculations)

(cycle 2) F₂₁₃-L₁₀-S₁₄-S₁₅ ends as the 4-instruction fetch frame is exhausted.

(cycle 3) M₃₆-S₁₁-L₇-S₈ ends as the 4-instruction fetch frame is exhausted.

As the last M₃ instruction keeps on executing in cycles 4 and 5, the total execution time (reading the code to the processor and executing the program) takes 6 cycles.

A similar program, which fulfils the branch conditions is executed in 8 cycles:

L₁S₂S₃S₉M₄F₂₅M₃₆L₇L₁₀L₁₂S₁₁F₂₁₃S₈S₁₄S₁₅

as the fetch happens like

(cycle 0) L₁-S₂-S₃ (!S) ends as the next instruction is an S, which may not be

fetches as it causes a data overrun

(cycle 1) S₉-M₃₄-F₂₅ (!F) ends as the following instruction is of type M, which causes a data overrun in the M execution unit. (The next M operation may be fetched in cycle 4 as the M₃ started in this cycle finishes in cycle 4.)

(cycle 4) M₃₆-L₇ (!L) data overrun in unit L finishes cycle

(cycle 5) L₁₀ (!L) no more instructions may be fetched, data overrun occurs in unit L

(cycle 6) L₁₂-S₁₁-F₂₁₃-S₈

(!S) data overrun in unit S

(cycle 7) S₁₄-S₁₅ input sequence is finished.

In addition to scheduling issues (i.e. prescribing the execution order of data-independent instructions) common programming technics include *operation type swapping*. A common usage of this (often used during creation of computer demos) is to transform some of the floating-point operations to integers or vica versa. This method enables the programmer to feed the processor with a homogenous mixture of different operation types, which increases data throughput as it decimates data overruns (a common disadvantage of similar operations). This algorithm is outside the domain of scheduling, as it may only be modelled using a transformation of the data graph. (It swaps the operation type and execution time of operations.)

A PowerPC 604 microprocessor could be described as a system with a maximum of 2 type S and a maximum of 1 each of type L, M and F processors. Execution times are 1, 1-3, 3 and 2 respectively. After finding the optimal latency for a given instruction sequence, the actual latency must be calculated (based on the fetch limit of 4 instructions every time cycle).

Operations that are not optimised for the given processor structure are likely to cause data overruns (referred to as **pipeline stalls**) often. This is the reason why most programs for Intel-based PCs feature special code for Pentium processors.

As some of the speed improvement of CISC processors is based on the increase in execution unit numbers (the Intel Pentium features a two-way internal pipeline (*two-way superscalar structure*), the Pentium Pro a similar *three-way superscalar structure*)), care must be taken for programs to produce code that is subject to optimisation. The Pentium Pro, for example, was optimised for 32-bit-based input streams; the Windows 95 operating system of Microsoft contains enough 16-bit code to disable its internal pipeline. As the 100 % speed increase

between a Pentium and a Pentium Pro is based mainly on the improvement of pipeline properties, Windows 95 runs *slower* on Pentium Pro-based systems than on Pentiums.

As the execution pipes of the Pentium processor are not identical, a regular sandwich-like program structure is needed to keep the processor occupied without stalls. This structure must be crafted carefully so that the program code contains a simple instruction following a complicated one. Simple instructions fit to the V pipe of the processor while the U pipe works with the complicated one, thus making an extra cycle unnecessary. A suitable model for this is the U pipe being a general processor (capable of holding any kind of operations) while the V pipe is a processor that may be used only for executing single-cycle operations.

10.4.3.1. Loop unrolling

Another typical usage of scheduling is **loop unrolling**, a trick in common usage. This method is useful in the core of time-critical applications, which are usually executed millions of times. The increase in throughput is the result of a homogenous, and therefore easily optimized code. The following code fragment (using Intel assembly for a numeric coprocessor)

```
loop:
fld    [esp+8]
fmul   [ebx+eax*4]
fadd   [ecx+eax*4]
fstp   [ecx+eax*4]
inc    eax
cmp    eax,ebp
jle    loop
```

is the trivial solution to a section of Gaussian elimination. This code offers nothing to optimize as the small number of operations makes it difficult to tune anything. Expanding the loop 3 times yields

```
loop:
fld    [esp+8]
fmul   [ebx+eax*4]
fadd   [ecx+eax*4]
fstp   [ecx+eax*4]
fld    [esp+8]
fmul   [ebx+eax*4+4]
fadd   [ecx+eax*4+4]
fstp   [ecx+eax*4+4]
fld    [esp+8]
fmul   [ebx+eax*4+8]
fadd   [ecx+eax*4+8]
fstp   [ecx+eax*4+8]
add    eax,3
cmp    eax,ebp
jle    loop
```

which increases the number of operations (and the bytes occupied) to 300 % of the original. As there are more operations now, the operations in the branch may be reordered and the numeric coprocessor may be stuffed with some of the simple instructions; this increases

pipeline utilization.

10.4.4. Force-directed scheduling

Force-directed scheduling is a modular scheduling algorithm, based on probabilistic approximations of utilisation. It is based on the idea of balancing operation load in such a way that the difference of minimum and maximum of concurrently used processors is a small value. In this case the number of processors required is greater than or equal to the maximum value, which in turn results in a high average utilisation.

The deviation from the average utilisation is weighted with the concurrent load in a way similar to Hooke's law ($F = -W \cdot \Delta W$). Scheduling seeks the minimum value of a force-like quantity. Additional components may be also introduced to the algorithm, usually as an additional component of the force. This flexibility, in addition to the good results achieved with force-directed scheduling make this algorithm one of the feasible, useful methods. Sometimes the algorithm serves as an example or as a special form of a benchmark.

Force-directed scheduling is capable of dealing with optimisation of both elementary operations and delays (i.e. buffers). Introducing buffers to the scheduling stage requires changes in the data models, so it is not usually used. This modification also increases the number of elementary operations in the graph, which increases scheduling time.

The load of an elementary operation is equal to the number of processors used during the execution of the operation, thus it is 1 for every time cycle in which the operation is **active**, for every fixed operation. The load of moving operations is calculated in a different way:

As the starting cycle of a moving operation is unknown, it may be approximated only. A suitable approximation is to use a uniform probability function for every cycle of the time frame. The **start probability function** is defined so that

$V(i,t)$ is equal to the probability of starting elementary operation $e(i)$ in time cycle t .

$$V(i,t) = 1/(l(i)-s(i)+1) \text{ for every } t \text{ such that } s(i) \leq t \leq l(i); \text{ otherwise } V(i,t)=0.$$

The $V(i,t)$ function is trivial for fixed operations, as it is equal to 1 for the starting cycle $t=s(i)=l(i)$.

As multicycle operations are using a processor for every cycle in which they are working, a function must be used to describe the actual load of elementary operation $e(i)$:

$G(i,t|k)$ is equal to the load of elementary operation $e(i)$ in time cycle t , with the assumption that $e(i)$ is started in time cycle k .

$G(i,t|k) = 1$ for every t such that $k \leq t \leq k+t(i)-1$; otherwise $G(i,t|k) = 0$.

The total load of $e(i)$ is a function based on $V(i,t)$ and $G(i,t|k)$:

$$U(i,t) = \sum(G(i,t|k) * V(i,k) : s(i) \leq k \leq l(i)) \text{ for } 0 \leq t \leq L-1.$$

After finding the total load for all of the operations, the load of operations using the same type of operations must be totalled to find the utilisation of processors as a function of time:

$W(j,t)$ is equal to the number of processors of type j used in time cycle t ; the **load function for operations of type j** .

$$W(j,t) = \sum(U(i,t) : j(i) = j) \text{ for } 0 \leq t \leq L-1.$$

During pipelined execution time cycles which are congruent modulo R are happening simultaneously, so the load functions must be folded to reflect this. The folding process transforms the load in cycle t to cycle $(t \bmod R)$:

$C(j,t)$ is equal to the number of processors of type j used in time cycle t during pipelined execution, the **overlapped load function of operations of type j** .

$$C(j,t) = \sum(W(j,t) : n \bmod R = t \text{ for } (0 \leq n \leq L-1)) \text{ for } 0 \leq t \leq R-1.$$

(For non-pipelined graphs $C(j,t) = W(j,t)$)

The number of processors to be built to the hardware unit must be found as

$$\max(\text{whole}(C(j,t)) : 0 \leq t \leq L-1),$$

where the $\text{whole}(x)$ function returns the smallest integer value greater than or equal to x . A uniform load is desirable as it results in an overall high utilisation and low idle percentage.

The force function describes the relative quantity of two scheduling plans. It is defined as sum in the form

$$F(\Delta) = \sum(\sum(w(j) * C(j,t) * \Delta C(j,t) : 0 \leq t \leq R-1) : 1 \leq j \leq J),$$

where $w(j)$ is the relative cost of type j processors.

Calculation of the F function requires the value of $C(j,t)$ in both of the scheduling plans. A negative value of F means a transition to a better scheduling plan. The optimal starting cycle for an operation may be found by comparing the transition results from its initial, moving state to the states found after fixing the operation to every cycle in its time frame. The time cycle resulting in the smallest F value is chosen as the optimal solution, after which the operations loses its mobility and is fixed to the optimal t cycle.

Force-directed scheduling scans all of the moving elements, finding the minimum of F for all of their possible starting cycles. As the optimum is found, the element is fixed there, and the next element becomes available for scheduling.

10.4.4.2. Disadvantages

As force-directed scheduling scans the elementary operations sequentially, it is not immune to the effects of ordering the operations (transitions to local optima does not always end in a global optimum). A typical example of that is the so-called **bottleneck**, an operation with multiple inputs and outputs separating the graph (Fig. 43.). Premature scheduling of a bottleneck operation may result in a disaster for other operations as it reduces the time frames for a lot of operations.

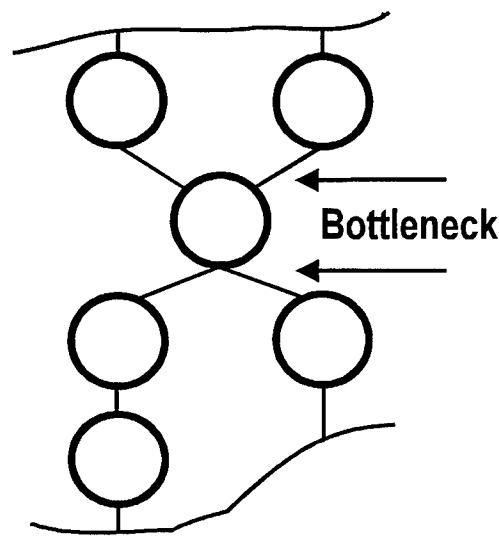


Figure 43.

Section of a graph containing a bottleneck

As force-directed scheduling has an execution time of order 3 as a function of system complexity, it executes much slower than list scheduling.

Due to the floating-point operations in a force-directed algorithm, hardware implementations are much more difficult to design than list-scheduling or ILP-based scheduling.

10.5. Conditional execution

Elementary operation graphs may contain branches that are executed if and only if an expression is valid. Presence of such a branch is a serious problem during scheduling as conditional execution may be treated as a static (worst-case) problem or using a heuristic model (which requires previous information on the composition of the data processed by the

hardware). Choosing the model of the conditional execution is not easy as both methods are useful only under conditions.

10.5.1. Worst-case model

Worst-case models of conditional execution prescribe the system to be able to handle any combination of inputs without decreasing performance. This condition during scheduling requires the scheduler to find the worst-case input and find a hardware plan that is capable of dealing with the problem. Worst-case design is usually an overestimation of system load and increases realisation costs due to the additional resources that must be available (but not used in a system that never approaches the worst-case load).

The worst-case model of the conditional branches treats the branches as if they were executing always fully parallel (requiring the maximum of resources the branches need in a given time cycle). A system designed to deal with such a load would not be ran over during regular usage as the actual usage of processors may not exceed the worst-case value. This property, however, means that some of the built-in processors would not be used most of the time.

10.5.2. Probability-based model

The opposite of worst-case design is the probability-based method of conditional execution. This approximation presents a model of conditional execution that needs the distribution of input values. The conditional branches are given weight values depending on the probability of execution in the mutually exclusive branches. The condition at the beginning of the branches is said to take the branches with a similar distribution. After this step, data propagation depends on the inputs following this distribution; data that deviates may slow down the system.

A typical usage of probability-based conditional execution is the internal structure of RISC processors. Such a processor features different types of execution units, each capable of dealing with a given set of instructions. Instructions executed inside the processor are entering one of the units depending on the type of operation they perform. In the case of a PowerPC 604, for example (which CPU contains two multi-cycle integer, one multi-cycle integer, a floating-point and a load-store unit), conditional execution is present as the processor activates one of its execution units depending on the type of the next instruction. As the PowerPC 604 may fetch 4 instructions in a single clock cycle, worst-case design would suggest a minimum of 4 pieces of all execution units (more for multi-cycle types). The designers of the PowerPC chose to reduce the number of units to one in all types save single-cycle integer operations (which are built twice). This realisation means that the PowerPC 604 keeps on running at

maximum speed (starting the execution of four instructions every clock cycle) if the instructions may be dispatched without *stalls*. As the number of units is lower than the worst-case value, the processor may encounter instructions that may not be dispatched to the respective units as they are still processing the previous instruction. This situation (a data overrun or stall) prevents execution of instructions and results in idle clock cycles for the processor. From the outside the processor may be modelled as an execution unit that executes instructions with a speed that is not independent of the input sequence. This property is a disadvantage if the method is compared to worst-case design (as it changes the characteristics of the elementary operations).

RISC (and internal pipeline) processors designed like the PowerPC 604 are to be programmed in such a way that program instruction sequences are designed using knowledge of the internal structure. For general-purpose hardware, such an assumption may not be made, so worst-case design may be more useful for systems that are latency-bound.

10.5.3. Realisation

Force-directed scheduling is an algorithm that may be easily extended to process graphs containing conditional execution. This modification does not modify the cost function, rather the generation of the load functions (W, C and U). The modified functions are used to find the force values and the algorithm proceeds normally after this step.

Scheduling a conditional execution block is done in two steps: the inside and the outside of the block are to be treated in a different way; the order is not prescribed. A conditional execution block (or conditional block) is the set of branches between the 'fork' operation (the one with the distributor property) and the 'join' operation (the receptor).

From the outside, the worst-case processor load in a given time cycle is equal to the maximum that is possible inside the block (i.e. the maximum of processor requirements for all branches). This may be expressed as the

$$U'(i,t) = \max (U(i,t,k) : 1 \leq k \leq b)$$

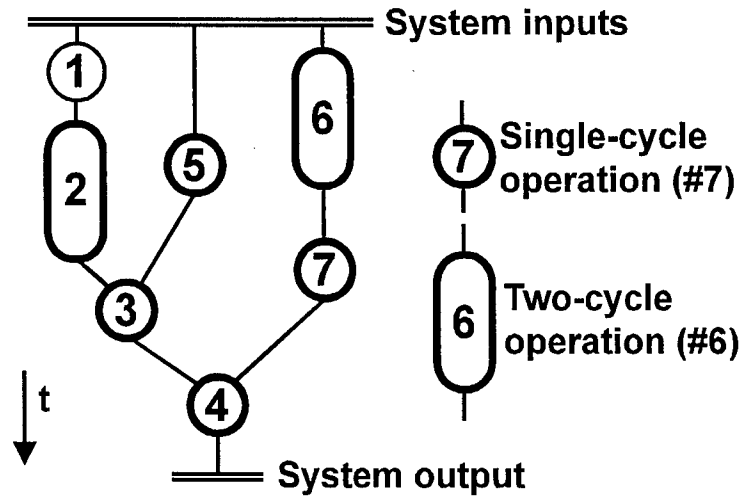
worst-case load function of the conditional block, where b is the number of conditional branches inside the block and $U(i,t,k)$ is the $U(i,t)$ load function for branch k ; this function is the sum of load functions for the elementary operations in a given branch. The worst-case load function means the extreme value of the load of operations inside the block. Scheduling operations outside the block may use the $U'(i,t)$ function as a load model of the block.

Scheduling the operations inside the block may be done using the load function(s) of the operations outside that block. Multiple conditional blocks use the worst-case load functions as

they are independent (a block is independent of the other).

10.6. Examples

For comparison of different scheduling algorithms, we use the same problem graph. During scheduling the pipelined execution mode is also used, with a restart time of 5 cycles.



Our system is operating in an environment such that latency must be equal to or less than 7. This is equivalent to $ALAP(4) \leq 6$. (This constraint prescribes L to be less than 8, and is set by the external units connected to our device. Any solution resulting in L greater than 7 must be discarded as it ruins the timing for the environment. A latency value below 8 is suitable, probably extended to 7 using inserted delay elements.)

We distinguish between two different processors, one used for execution single-cycle operations, the other one for multiple-cycle execution. Single-cycle (type 1) processors' cost is one-half of the cost of multi-cycle (type 2) processors. ($w(1)=1$, $w(2)=2$)

10.6.1. ILP-based scheduling

The respective cost for processors is proportional to the execution time, i.e. $w(1)=1$, $w(2)=2$. Total system cost must be kept under $D=12$. Number of processors is $M(1)$ and $M(2)$, total implementation cost is $C=w(1)*M(1)+w(2)*M(2)=M(1)+2*M(2)$.

Operation number i	ASAP cycle $s(i)$	ALAP cycle $l(i)$	Unit type $j(i)$	Execution time $t(i)$
1	0	2	1	1
2	1	3	2	2
3	3	5	1	1
4	4	6	1	1
5	0	4	1	1
6	0	3	2	2
7	2	5	1	1

The non-trivial starting time variables ($x(i,t)$) are found based on the ASAP and ALAP values:

t	$x(1,t)$	$x(2,t)$	$x(3,t)$	$x(4,t)$	$x(5,t)$	$x(6,t)$	$x(7,t)$
0	$x(1,0)$	0	0	0	$x(5,0)$	$x(6,0)$	0
1	$x(1,1)$	$x(2,1)$	0	0	$x(5,1)$	$x(6,1)$	0
2	$x(1,2)$	$x(2,2)$	0	0	$x(5,2)$	$x(6,2)$	$x(7,2)$
3	0	$x(2,3)$	$x(3,3)$	0	$x(5,3)$	$x(6,3)$	$x(7,3)$
4	0	0	$x(3,4)$	$x(4,4)$	$x(5,4)$	0	$x(7,4)$
5	0	0	$x(3,5)$	$x(4,5)$	0	0	$x(7,5)$
6	0	0	0	$x(4,6)$	0	0	0

As there are multiple-cycle operations present, they use their processors both during their first and second cycle. For this reason, the number of processors used in the implementation of such an operation is expressed by $x'(i,t)$:

t	$x'(i,t)$ for every single-cycle operation	$x'(2,t)$	$x'(6,t)$
0	$x(i,0)$	0	$x(6,0)$
1	$x(i,1)$	$x(2,1)$	$x(6,1)+x(6,0)$
2	$x(i,2)$	$x(2,2)+x(2,1)$	$x(6,2)+x(6,1)$
3	$x(i,3)$	$x(2,3)+x(2,2)$	$x(6,3)+x(6,2)$
4	$x(i,4)$	$x(2,3)$	$x(6,3)$
5	$x(i,5)$	0	0
6	$x(i,6)$	0	0

For a pipelined execution mode with $R=5$, the sixth cycle is executing simultaneously with cycle 0 of the next data packet. This is also true for the 7th and 1st cycles, so this pipelined utilisation is found by *folding* the table:

t	$x''(i,t)$ for every single-cycle operation	$x''(2,t)$	$x''(6,t)$
0	$x(i,0)+x(i,5)$	0	$x(6,0)$
1	$x(i,1)+x(i,6)$	$x(2,1)$	$x(6,1)+x(6,0)$
2	$x(i,2)$	$x(2,2)+x(2,1)$	$x(6,2)+x(6,1)$

$$\begin{aligned}
& x(6,0) - M(2) \leq 0 \\
& x(2,1) + x(6,1) + x(6,0) - M(2) \leq 0 \\
& (\text{as } x'(2,1) = x(2,1) \text{ and } x'(6,1) = x(6,0) + x(6,1)) \\
& x(2,2) + x(2,1) + x(6,2) + x(6,1) - M(2) \leq 0 \\
& x(2,3) + x(2,2) + x(6,3) + x(6,2) - M(2) \leq 0 \\
& x(2,3) + x(6,3) - M(2) \leq 0
\end{aligned}$$

The trivial lines (i.e. for $t=5$: $0 - M(2) \leq 0$) are omitted.

b.2) Pipelined execution, time domain is folded:

In addition to multicycle operations, system time must be folded to reflect the overlapping execution, so we need the overlapping $x''(i,t)$ values. The total of simultaneously utilised processors must be found for every t such that $0 \leq t \leq R-1$:

$$\begin{aligned}
& x(1,0) + x(5,0) + x(3,5) + x(4,5) + x(7,5) - M(1) \leq 0 \\
& x(1,1) + x(5,1) + x(4,6) - M(1) \leq 0 \\
& x(1,2) + x(5,2) + x(7,2) - M(1) \leq 0 \\
& x(3,3) + x(5,3) + x(7,3) - M(1) \leq 0 \\
& x(3,4) + x(5,4) + x(7,4) - M(1) \leq 0
\end{aligned}$$

$$\begin{aligned}
& (\text{as } x''(4,0) = x(4,0) + x(4,5) = x(4,5), \\
& x''(3,0) = x(3,0) + x(3,5) = x(3,5), \\
& x''(7,0) = x(7,0) + x(7,5) = x(7,5), \\
& x''(4,1) = x(4,1) + x(4,6) = x(4,6), \\
& x''(3,1) = x(3,1) + x(3,6) = 0 \text{ and} \\
& x''(7,1) = x(7,1) + x(7,6) = 0.)
\end{aligned}$$

Other operations (e.g. $e(1)$) are unaffected because of their ALAP times: $l(1) \leq R-1$

$$x(6,0)-M(2)\leq 0x(2,1)+x(6,1)+x(6,0)-M(2)\leq 0$$

$$x(2,2)+x(2,1)+x(6,2)+x(6,1)-M(2)\leq 0$$

$$x(2,3)+x(2,2)+x(6,3)+x(6,2)-M(2)\leq 0$$

$$x(2,3)+x(6,3)-M(2)\leq 0$$

(Note that because of the ASAP and ALAP times of operations #2 and #6, the folding of inequalities did not affect the system for operations of type 2.)

c) Data dependencies must not be violated. This prescribes the following relations:

$$e(1)\rightarrow e(2), e(2)\rightarrow e(3), e(3)\rightarrow e(4), e(5)\rightarrow e(3), e(6)\rightarrow e(7), e(7)\rightarrow e(4).$$

Note that the following indirect dependencies are not used:

$$e(1)\rightarrow e(3), e(1)\rightarrow e(4), e(5)\rightarrow e(4), e(6)\rightarrow e(4).$$

(They result in a number of redundant inequalities.)

$$0*x(1,0)+1*x(1,1)+2*x(1,2)-1*x(2,1)-2*x(2,2)-3*x(2,3)\leq -1$$

$$1*x(2,1)+2*x(2,2)+3*x(2,3)-3*x(3,3)-4*x(3,4)-5*x(3,5)\leq -1$$

$$3*x(3,3)+4*x(3,4)+5*x(3,5)-4*x(4,4)-5*x(4,5)-6*x(4,6)\leq -1$$

$$0*x(5,0)+1*x(5,1)+2*x(5,2)+4*x(5,4)-3*x(3,3)-4*x(3,4)-5*x(3,5)\leq -1$$

$$0*x(6,0)+1*x(6,1)+2*x(6,2)+3*x(6,3)-$$

$$-2*x(7,2)-3*x(7,3)-4*x(7,4)-5*x(7,5)\leq -1$$

$$2*x(7,2)+3*x(7,3)+4*x(7,4)+5*x(7,5)-4*x(4,4)-5*x(4,5)-6*x(4,6)\leq -1$$

The redundancy of the $e(1)\rightarrow e(3)$ dependency is clear as it is equivalent to the sum of the $e(1)\rightarrow e(2)$ and $e(2)\rightarrow e(3)$ inequalities. (Trivial proof for the transitive property of ' $\rightarrow$ '.)

d) Total implementation cost must be kept below cost limit:

$$C=M(1)+2*M(2)\leq D,$$

$$D=12.$$

Solutions minimise C. A possible solution is the following:

a) Non-pipelined execution:

$$x(1,0)=x(2,1)=x(3,3)=x(4,6)=x(5,1)=x(6,3)=x(7,4)=1,$$

$$M(1)=M(2)=1,$$

$$C=3 \leq D.$$

b) Pipelined execution:

$$x(1,0)=x(2,2)=x(3,4)=x(4,6)=x(5,3)=x(6,0)=x(7,3)=1,$$

$$M(1)=M(2)=1,$$

$$C=3 \leq D.$$

10.6.2. List scheduling, no hardware constraint

Our system is operating in an environment such that latency must be equal to or less than 7. This is equivalent to $ALAP(4) \leq 6$. We distinguish between two different processors, one used for execution single-cycle operations, the other one for multiple-cycle execution.

Number of processors ($M(1)$ and $M(2)$) must be minimised. Priority ($p(i)$) is equal to $4-m(i)$, operations with a lower priority value are at a disadvantage.

Operation number i	SLT $s(i)$	LLT $l(i)$	Unit type $j(i)$	Mobility $(l(i)-s(i))$ $m(i)$	Execution time $t(i)$
1	0	2	1	2	1
2	1	3	2	2	2
3	3	5	1	2	1
4	4	6	1	2	1
5	0	4	1	4	1
6	0	3	2	3	2
7	2	5	1	3	1

Time domain is scanned in increasing order:

0) Competition between $e(1)$ and $e(5)$. As $p(1)=2 > p(5)=0$, $e(5)$ is delayed, this affects no other operation as $e(5)$ was not in its ALAP cycle. The winner, $e(1)$ is fixed so $s(1)=l(1)=0$. $e(6)$ is fixed without competition as there is no other operation of type 2 that may be started in $t=0$.

i	$s(i)$	$l(i)$	$m(i)$
1	0	0	0
2	1	3	2
3	3	5	2
4	4	6	2
5	1	4	3
6	0	0	0
7	2	5	3

Cycle 1) No competition, $e(2)$ and $e(5)$ are started.

i	s(i)	l(i)	m(i)
1	0	0	0
2	1	1	0
3	3	5	2
4	4	6	2
5	1	1	0
6	0	0	0
7	2	5	3

Cycles 2), 3) and 4): e(7), e(3) and e(4) are fixed without competition.

Cycles 5) and 6): no operations remain to be scheduled for these cycles.

Final result:

i	s(i)	l(i)	m(i)
1	0	0	0
2	1	1	0
3	3	3	0
4	4	4	0
5	1	1	0
6	0	0	0
7	2	2	0

This scheduling plan requires **one** unit of type 1 and **two** units of type 2 as the second cycle of e(6) is operating simultaneously with the first cycle of e(2), which requires more processors than the optimal solution $M(2)=1$. Total latency is 5. (This may be extended to 7 by inserting a delay of two cycles somewhere, if latency should be exactly 7. This is a decision that must be made judging the properties of the environment.)

Should the system be operated with a restart time of 5, there would be no change in the number of processors required as $L=5$ is equal to $R=5$.

10.6.3. List scheduling under hardware constraint

Our system is operating in an environment such that only one processor is available for type 1 and one for type 2 ($M(1)=M(2)=1$). System latency must be found.

Priority ($p(i)$) is equal to the total execution time for operations following $e(i)$, Operations with a lower priority value are at a disadvantage.

Operation number i	Initial ASAP cycle $s(i)$	Unit type $j(i)$	Priority $m(i)$	Execution time $t(i)$
1	0	1	4	1
2	1	2	2	2
3	3	1	1	1
4	4	1	0	1
5	0	1	2	1
6	0	2	2	2
7	2	1	1	1

Time domain is scanned in increasing order:

0) Competition between $e(1)$ and $e(5)$. As $p(1)=4 > p(5)=2$, $e(5)$ is delayed. The winner, $e(1)$ is fixed so $s(1)=l(1)=0$. $e(6)$ is fixed without competition as there is no other operation of type 2 that may be started in $t=0$.

i	$s(i)$
1	0
2	1
3	3
4	4
5	1
6	0
7	2

1) $e(5)$ is started. $e(2)$ may not be started as the only available type 2 processor is occupied; it contains the data of $e(6)$ in its second cycle. Delaying $e(2)$ increases $s(3)$ and $s(4)$ due to data dependencies.

i	$s(i)$
1	0
2	2
3	4
4	5
5	1
6	0
7	2

2) $e(2)$ and $e(7)$ are started. There is no competition.

3) No operation is started as there is no $s(i)=3$ value in the table.

4) and 5): $e(3)$ and $e(4)$ are started without competition.

Final result:

i	s(i)
1	0
2	2
3	4
4	5
5	1
6	0
7	2

This scheduling plan requires one processor of type 1 and one of type 2, as prescribed. Latency, however, is increased to 6.

Pipelined execution with a restart time of 5 violates system constraints, as $e(4)$ is started in $t=5$, $e(1)$ in $t=0$, which cycles are happening simultaneously. This results in a collision of data if a single processor is used for $e(1)$ and $e(4)$. A pipeline-aware list scheduler would delay $e(4)$ so that it fits into a 'hole' in pipelined mode (Fig 42.).

The first unoccupied cycle would be $t=8$, so $e(4)$ would be fixed to $s(4)=8$. This increases latency to $L=9$.

10.6.4. Force-directed scheduling, non-pipelined execution

To calculate the functions of the force-directed scheduling, we use a common denominator of 60, so most of the calculations are transformed to integers.

i	s(i)	l(i)	j(i)	t(i)	V(i,0)	V(i,1)	V(i,2)	V(i,3)	V(i,4)	V(i,5)	V(i,6)
1	0	2	1	1	20/60	20/60	20/60				
2	1	3	2	2		20/60	20/60	20/60			
3	3	5	1	1				20/60	20/60	20/60	
4	4	6	1	1					20/60	20/60	20/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	15/60	15/60	15/60			
7	2	5	1	1			15/60	15/60	15/60	15/60	

As there are multi-cycle operations in the graph, $U(i,t)$ must be adjusted to reflect this. For other operations, $U(i,t)=V(i,t)$. $C(j,t)=W(j,t)$ as there is no overlapping in time domain.

(Type 2 operations are in bold, changes are highlighted with italic.)

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	0	2	1	1	20/60	20/60	20/60				
2	1	3	2	2		20/60	40/60	40/60	20/60		
3	3	5	1	1				20/60	20/60	20/60	
4	4	6	1	1					20/60	20/60	20/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
C(1,t)					32/60	32/60	47/60	47/60	67/60	55/60	20/60
C(2,t)					15/60	50/60	70/60	70/60	35/60	0/60	0/60

We find the optimal starting cycle for e(1) first. Time cycles are scanned in increasing direction, from s(1)=0 to l(1)=2.

Fixing e(1) to cycle 0 yields

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	0	0	1	1	60/60						
2	1	3	2	2		20/60	40/60	40/60	20/60		
3	3	5	1	1				20/60	20/60	20/60	
4	4	6	1	1					20/60	20/60	20/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
C0(1,t)					32/60	32/60	47/60	47/60	67/60	55/60	20/60
C0(2,t)					15/60	50/60	70/60	70/60	35/60	0/60	0/60
C(1,t)					72/60	12/60	27/60	47/60	67/60	55/60	20/60
C(2,t)					15/60	50/60	70/60	70/60	35/60	0/60	0/60
$\Delta C(1,t)$					40/60	-20/60	-20/60	0	0	0	0
$\Delta C(2,t)$					0	0	0	0	0	0	0

(C0(i,t) denotes the initial, C(i,t) the adjusted C() values. $\Delta C(i,t)=C(i,t)-C0(i,t)$)

$F(\Delta)=(32*40-32*20-47*20)/3600=-300/3600$, the negative sign means an improvement of the initial schedule.

Fixing e(1) to cycle 1 results in

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1		60/60					
2	2	3	2	2			30/60	60/60	30/60		
3	4	5	1	1					30/60	30/60	
4	5	6	1	1						30/60	30/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
C0(1,t)					32/60	32/60	47/60	47/60	67/60	55/60	20/60
C0(2,t)					15/60	50/60	70/60	70/60	35/60	0/60	0/60
C(1,t)					12/60	72/60	27/60	47/60	67/60	75/60	30/60
C(2,t)					15/60	30/60	60/60	90/60	45/60	0	0
$\Delta C(1,t)$					-20/60	40/60	-20/60	-20/60	-10/60	20/60	10/60
$\Delta C(2,t)$					0	-20/60	-10/60	20/60	10/60	0	0

$$F(\Delta)=2*(-50*20-70*10+70*20+35*10)/3600+ \\ +(-32*20+32*40-47*20-47*20-67*10+55*20+20*10)/3600$$

$$F(\Delta) = -410/3600$$

Fixing $e(1)$ to cycle 2 causes

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
C0(1,t)					32/60	32/60	47/60	47/60	67/60	55/60	20/60
C0(2,t)					15/60	50/60	70/60	70/60	35/60	0/60	0/60
C(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C(2,t)					15/60	30/60	30/60	90/60	75/60	0	0
$\Delta C(1,t)$					-20/60	-20/60	40/60	-20/60	-40/60	20/60	40/60
$\Delta C(2,t)$					0	-20/60	-40/60	20/60	40/60	0	0

$$F(\Delta) = 2 * (-50 * 20 - 40 * 70 + 20 * 70 + 40 * 35) / 3600 +$$

$$+ (-20 * 32 - 20 * 32 + 40 * 47 - 20 * 47 - 40 * 67 + 20 * 55 + 40 * 20) / 3600$$

$$F(\Delta) = -5120/3600$$

As $F(\Delta)$ had its minimum when $e(1)$ was fixed to cycle 2, this position is the initial scheduling plan for the next operation. Note that the successors of $e(1)$ are also fixed as $e(1)$ is started in its ALAP cycle:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
C(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C(2,t)					15/60	30/60	30/60	90/60	75/60	0	0

The next operation to schedule is $e(6)$. Time domain is scanned in increasing order, from cycle 0 to cycle 3. As we fix $e(6)$ to cycle 0:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	0	2	2	60/60	60/60					
7	2	5	1	1			15/60	15/60	15/60	15/60	
C0(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C0(2,t)					15/60	30/60	30/60	90/60	75/60	0	0
C(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C(2,t)					60/60	60/60	0	60/60	60/60	0	0
$\Delta C(1,t)$					0	0	0	0	0	0	0
$\Delta C(2,t)$					45/60	30/60	-30/60	-30/60	-15/60	0	0

$$F(\Delta) = 2 * (45 * 15 + 30 * 30 - 30 * 30 - 30 * 90 - 15 * 75) / 3600 + (0) / 3600 = -6300 / 3600.$$

Setting cycle 1 as starting time for $e(6)$ yields

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	1	1	2	2		60/60	60/60				
7	3	5	1	1				20/60	20/60	20/60	
C0(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C0(2,t)					15/60	30/60	30/60	90/60	75/60	0	0
C(1,t)					12/60	12/60	72/60	32/60	32/60	85/60	60/60
C(2,t)					0	60/60	60/60	60/60	60/60	0	0
$\Delta C(1,t)$					0	0	-15/60	5/60	5/60	5/60	0
$\Delta C(2,t)$					-15/60	30/60	30/60	-30/60	-15/60	0	0

$$F(\Delta) = 2 * (-15 * 15 + 30 * 30 + 30 * 30 - 30 * 90 - 15 * 75) / 3600 +$$

$$+(-15*87+5*27+5*27+5*75)/3600=-5160/3600.$$

Setting $s(6)=l(6)=1$ results in

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	2	2	2	2			60/60	60/60			
7	4	5	1	1					30/60	30/60	
C0(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C0(2,t)					15/60	30/60	30/60	90/60	75/60	0	0
C(1,t)					12/60	12/60	72/60	12/60	42/60	95/60	60/60
C(2,t)					0	0	60/60	120/60	60/60	0	0
$\Delta C(1,t)$					0	0	-15/60	-15/60	15/60	15/60	0
$\Delta C(2,t)$					-15/60	-30/60	30/60	30/60	-15/60	0	0

$$F(\Delta)=2*(-15*15-30*30+30*30+30*90-15*75)/3600+$$

$$+(-15*87-15*27+15*27+15*75)/3600=2520/3600.$$

Note that this large positive value means a definite change in quality, as the new schedule requires two type 2 processors due to $C(2,3)=2$.

Starting e(6) in cycle 3 produces a similar collision in cycles 3 and 4:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	3	3	2	2				60/60	60/60		
7	5	5	1	1						60/60	
C0(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C0(2,t)					15/60	30/60	30/60	90/60	75/60	0	0
C(1,t)					12/60	12/60	72/60	12/60	12/60	120/60	60/60
C(2,t)					0	0	0	120/60	120/60	0	0
$\Delta C(1,t)$					0	0	-15/60	-15/60	-15/60	45/60	0
$\Delta C(2,t)$					-15/60	-30/60	-30/60	30/60	45/60	0	0

$$F(\Delta)=2*(-15*15-30*30-30*30+30*90+45*75)/3600+$$

$$+(-15*87-15*27-15*27+45*75)/3600=9360/3600.$$

The optimum was found at cycle 0, so the scheduling of e(7) starts off from the following initial scheduling plan:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	0	2	2	60/60	60/60					
7	5	5	1	1			15/60	15/60	15/60	15/60	
C(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
C(2,t)					60/60	60/60	0	60/60	60/60	0	0

As e(2) and e(6) are fixed by now, C(2,t) may not change any more. The steps for e(7) and e(5) are the following:

C0(1,t)	12/60	12/60	87/60	27/60.	27/60	75/60	60/60
---------	-------	-------	-------	--------	-------	-------	-------

Fixing e(7) to cycle 2:

C(1,t)	12/60	12/60	132/60	12/60	12/60	60/60	60/60
$\Delta C(1,t)$	0	0	45/60	-15/60	-15/60	-15/60	0

$$F(\Delta) = (45 \cdot 87 - 15 \cdot 27 - 15 \cdot 27 - 15 \cdot 75) / 3600 = 1980 / 3600.$$

Fixing e(7) to cycle 3:

C(1,t)	12/60	12/60	72/60	72/60	12/60	60/60	60/60
$\Delta C(1,t)$	0	0	-15/60	45/60	-15/60	-15/60	0

$$F(\Delta) = (-15 \cdot 87 + 45 \cdot 27 - 15 \cdot 27 - 15 \cdot 75) / 3600 = -1620 / 3600.$$

Fixing e(7) to cycle 4:

C(1,t)	12/60	12/60	72/60	12/60	72/60	60/60	60/60
$\Delta C(1,t)$	0	0	-15/60	-15/60	45/60	-15/60	0

$$F(\Delta) = (-15 \cdot 87 - 15 \cdot 27 + 45 \cdot 27 - 15 \cdot 75) / 3600 = -1620 / 3600.$$

Note that this plan is equivalent to the previous one as e(7) competes only with e(5) without data dependency, so e(5) introduces a uniform load on the processors in these cycles. In other cycles e(7) increases the number of processors needed above 1, which is a waste of hardware resources.

Fixing e(7) to cycle 5:

C(1,t)	12/60	12/60	72/60	12/60	12/60	120/60	60/60
$\Delta C(1,t)$	0	0	-15/60	-15/60	-15/60	45/60	0

$$F(\Delta) = (-15 \cdot 87 - 15 \cdot 27 - 15 \cdot 27 + 45 \cdot 75) / 3600 = 1260 / 3600.$$

As a minimum was found for F in cycles 3 and 4, we are free to choose one of them. In this case, due to lack of data dependencies, the choice is irrelevant, so 3 is chosen.

After fixing e(7) to cycle 3, the initial conditions for e(5) are the following:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	0	2	2	60/60	60/60					
7	5	5	1	1				60/60			
C(1,t)					12/60	12/60	72/60	72/60	12/60	60/60	60/60
C(2,t)					60/60	60/60	0	60/60	60/60	0	0

As we scan time domain from cycle 0 to 4, e(5) is either increasing C(1,t) to 60/60 (in cycles 0, 1 and 4) or to 120/60 (in cycles 2 and 3). The F values are: 2160/3600 for an operation collision (cycles 2 and 3), where the maximum of C(1,t) is increased to 120/60; -1440/3600 for any other cycle (which sets the maximum of C(1,t) to 60/60, resulting in a need of one processor for type 1 elements.) Operation e(5) may be fixed to any of these cycles, so cycle 0 (global ASAP cycle) is chosen:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	60/60						
6	0	0	2	2	60/60	60/60					
7	5	5	1	1				60/60			
C(1,t)					60/60	0	60/60	60/60	0	60/60	60/60
C(2,t)					60/60	60/60	0	60/60	60/60	0	0

This scheduling plan requires one processors for both types, which is an optimal solution as the system needs 4 units of type 2 processors ($t(2)+t(6)=4$) and 5 units of type 1 processors ($t(1)+t(3)+t(4)+t(5)+t(7)=5$).

An implementation with one type 1 processor builds the system with $1*L=7$ units of type 1 processors, which means unused 2 units. As only 3 units of type 2 processors are idle, the solution is optimal.

10.6.4.2, Force-directed scheduling, pipelined execution

During pipelined (with $R=5$) $C(j,0)=W(j,0)+W(j,1)$ and $C(j,1)=W(j,1)+W(j,6)$.

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	0	2	1	1	20/60	20/60	20/60				
2	1	3	2	2		20/60	40/60	40/60	20/60		
3	3	5	1	1				20/60	20/60	20/60	
4	4	6	1	1					20/60	20/60	20/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
W(1,t)					32/60	32/60	47/60	47/60	67/60	55/60	20/60
W(2,t)					15/60	50/60	70/60	70/60	35/60	0/60	0/60
C(1,t)					87/60	52/60	47/60	47/60	67/60		
C(2,t)					15/60	50/60	70/60	70/60	35/60		

First operation to be scheduled is e(1):

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	0	0	1	1	60/60						
2	1	3	2	2		20/60	40/60	40/60	20/60		
3	3	5	1	1				20/60	20/60	20/60	
4	4	6	1	1					20/60	20/60	20/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
W(1,t)					72/60	12/60	27/60	47/60	67/60	55/60	20/60
W(2,t)					15/60	50/60	70/60	70/60	35/60	0/60	0/60
C0(1,t)					87/60	52/60	47/60	47/60	67/60		
C0(2,t)					15/60	50/60	70/60	70/60	35/60		
C(1,t)					127/60	32/60	27/60	47/60	67/60		
C(2,t)					15/60	50/60	70/60	70/60	35/60		
$\Delta C(1,t)$					40/60	-20/60	-20/60	0	0		
$\Delta C(2,t)$					0	0	0	0	0		

$$F(\Delta) = (40 \cdot 87 - 20 \cdot 52 - 20 \cdot 47) / 3600 = 1500 / 3600.$$

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1		60/60					
2	2	3	2	2			30/60	60/60	30/60		
3	4	5	1	1					30/60	30/60	
4	5	6	1	1						30/60	30/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
W(1,t)					12/60	72/60	27/60	27/60	57/60	75/60	30/60
W(2,t)					15/60	30/60	60/60	90/60	45/60	0/60	0/60
C0(1,t)					87/60	52/60	47/60	47/60	67/60		
C0(2,t)					15/60	50/60	70/60	70/60	35/60		
C(1,t)					87/60	102/60	27/60	27/60	57/60		
C(2,t)					15/60	30/60	60/60	90/60	45/60		
$\Delta C(1,t)$					0	50/60	-20/60	-20/60	-10/67		
$\Delta C(2,t)$					0	-20/60	-10/60	20/60	10/60		

$$F(\Delta) = ((50 \cdot 52 - 20 \cdot 47 - 20 \cdot 47 - 10 \cdot 67) + 2 \cdot (-20 \cdot 50 - 10 \cdot 70 + 20 \cdot 70 + 10 \cdot 35)) / 3600 = 150 / 3600.$$

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	2	2	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
W(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
W(2,t)					15/60	30/60	30/60	90/60	75/60	0/60	0/60
C0(1,t)					87/60	52/60	47/60	47/60	67/60		
C0(2,t)					15/60	50/60	70/60	70/60	35/60		
C(1,t)					87/60	72/60	87/60	27/60	27/60		
C(2,t)					15/60	30/60	30/60	90/60	75/60		
$\Delta C(1,t)$					0	20/60	40/60	-20/60	-40/60		
$\Delta C(2,t)$					0	-20/60	-40/60	20/60	40/60		

$$F(\Delta) = ((20*52 + 40*47 - 20*47 - 40*67) + 2*(-20*50 - 40*70 + 20*70 + 40*35)) / 3600 = -2700 / 3600.$$

The best result is to fix e(1) to cycle 2. This results in the following initial plan:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	2	3	2	2				60/60	60/60		
3	4	5	1	1						60/60	
4	5	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	3	2	2	15/60	30/60	30/60	30/60	15/60		
7	2	5	1	1			15/60	15/60	15/60	15/60	
W(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
W(2,t)					15/60	30/60	30/60	90/60	75/60	0/60	0/60
C(1,t)					87/60	72/60	87/60	27/60	27/60		
C(2,t)					15/60	30/60	30/60	90/60	75/60		

This plan fixes e(2), e(3) and e(4) to their ALAP positions, so the next operation to schedule is e(6). Time cycles are scanned from 0 to 3:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	0	2	2	60/60	60/60					
7	2	5	1	1			15/60	15/60	15/60	15/60	
W(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
W(2,t)					60/60	60/60	0	60/60	60/60	0	0
C0(1,t)					87/60	72/60	87/60	27/60	27/60		
C0(2,t)					15/60	30/60	30/60	90/60	75/60		
C(1,t)					87/60	72/60	87/60	27/60	27/60		
C(2,t)					60/60	60/60	0	60/60	60/60		
$\Delta C(1,t)$					0	0	0	0	0		
$\Delta C(2,t)$					45/60	30/60	-30/60	-30/60	-15/60		

$$F(\Delta) = 2 * (45 * 15 + 30 * 30 - 30 * 30 - 30 * 90 - 15 * 75) / 3600 = -6300 / 3600$$

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	1	1	2	2		60/60	60/60				
7	3	5	1	1				20/60	20/60	20/60	
W(1,t)					12/60	12/60	72/60	32/60	32/60	80/60	60/60
W(2,t)					0	60/60	60/60	60/60	60/60	0	0
C0(1,t)					87/60	72/60	87/60	27/60	27/60		
C0(2,t)					15/60	30/60	30/60	90/60	75/60		
C(1,t)					92/60	72/60	72/60	32/60	32/60		
C(2,t)					0	60/60	60/60	60/60	60/60		
$\Delta C(1,t)$					5/60	0	-15/60	5/60	5/60		
$\Delta C(2,t)$					-15/60	30/60	30/60	-30/60	-15/60		

$$F(\Delta) = -5100/3600$$

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	2	2	2	2			60/60	60/60			
7	4	5	1	1					30/60	30/60	
W(1,t)					12/60	12/60	72/60	12/60	42/60	90/60	60/60
W(2,t)					0	0	60/60	120/60	60/60	0	0
C0(1,t)					87/60	72/60	87/60	27/60	27/60		
C0(2,t)					15/60	30/60	30/60	90/60	75/60		
C(1,t)					102/60	72/60	72/60	12/60	42/60		
C(2,t)					0	0	60/60	120/60	60/60		
$\Delta C(1,t)$					15/60	0	-15/60	-15/60	15/60		
$\Delta C(2,t)$					-15/60	-30/60	30/60	30/60	-15/60		

$$F(\Delta) = 2700/3600$$

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	3	3	2	2				60/60	60/60		
7	5	5	1	1						60/60	
W(1,t)					12/60	72/60	72/60	12/60	12/60	120/60	60/60
W(2,t)					0	0	0	120/60	120/60	0	0
C0(1,t)					87/60	72/60	87/60	27/60	27/60		
C0(2,t)					15/60	30/60	30/60	90/60	75/60		
C(1,t)					132/60	72/60	72/60	12/60	12/60		
C(2,t)					0	0	0	120/60	120/60		
$\Delta C(1,t)$					45/60	0	-15/60	-15/60	-15/60		
$\Delta C(2,t)$					-15/60	-30/60	-30/60	30/60	45/60		

$$F(\Delta) = 7290/3600$$

As e(6) is fixed to cycle 0, the initial plan for the scheduling of e(7) is

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	0	2	2	60/60	60/60					
7	2	5	1	1			15/60	15/60	15/60	15/60	
W(1,t)					12/60	12/60	87/60	27/60	27/60	75/60	60/60
W(2,t)					60/60	60/60	0	60/60	60/60	0	0
C(1,t)					87/60	72/60	87/60	27/60	27/60		
C(2,t)					60/60	60/60	0	60/60	60/60		

As we are after the scheduling of both type 2 operations, the $F(\Delta)$ values are depending only on the $C(1,t)$ function. The steps for e(7) are:

C0(1,t)	87/60	72/60	87/60	27/60	27/60
---------	-------	-------	-------	-------	-------

Fixing e(7) to cycle 2:

C(1,t)	72/60	72/60	132/60	12/60	12/60
$\Delta C(1,t)$	-15/60	0	45/60	-15/60	-15/60

$$F(\Delta) = (-15 \cdot 87 + 45 \cdot 87 - 15 \cdot 27 - 15 \cdot 27) / 3600 = 1800 / 3600.$$

Fixing e(7) to cycle 3:

C(1,t)	72/60	72/60	75/60	72/60	12/60
$\Delta C(1,t)$	-15/60	0	-15/60	45/60	-15/60

$$F(\Delta) = (-15 \cdot 87 - 15 \cdot 87 + 45 \cdot 27 - 15 \cdot 27) / 3600 = -1800 / 3600.$$

Fixing e(7) to cycle 4:

C(1,t)	72/60	72/60	75/60	12/60	72/60
$\Delta C(1,t)$	-15/60	0	-15/60	-15/60	45/60

$$F(\Delta) = (-15 \cdot 87 - 15 \cdot 87 - 15 \cdot 27 + 45 \cdot 27) / 3600 = -1800 / 3600$$

Note that this plan is equivalent to the previous one as e(7) competes only with e(5) without data dependency, so e(5) introduces a uniform load on the processors in these cycles. In other cycles e(7) increases the number of processors needed above 1, which is a waste of hardware resources.

Fixing e(7) to cycle 5:

C(1,t)	132/60	72/60	75/60	12/60	12/60
$\Delta C(1,t)$	45/60	0	-15/60	-15/60	-15/60

$$F(\Delta) = (45 \cdot 87 - 15 \cdot 87 - 15 \cdot 27 - 15 \cdot 27) / 3600 = 1800 / 3600$$

As a minimum was found for F in cycles 3 and 4, we are free to choose one of them. In this case, due to lack of data dependencies, the choice is irrelevant, so 3 is chosen.

After fixing e(7) to cycle 3, the initial conditions for e(5) are the following:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	0	4	1	1	12/60	12/60	12/60	12/60	12/60		
6	0	0	2	2	60/60	60/60					
7	3	3	1	1				60/60			
W(1,t)					12/60	12/60	72/60	72/60	12/60	60/60	60/60
W(2,t)					60/60	60/60	0	60/60	60/60	0	0
C(1,t)					72/60	72/60	72/60	72/60	12/60		
C(2,t)					60/60	60/60	0	60/60	60/60		

As e(5) is the last of the operations, it does not cause a problem with data dependencies. As we scan time domain in increasing order from 0 to 4, all cycles between 0 and R-1 are scanned. The scheduling results:

For cycles 0, 1, 2 and 3: $F(\Delta) = (2 * (-12 * 72) + 48 * 72 - 12 * 12) / 3600 = 1584 / 3600$

For cycle 4: $F(\Delta) = (3 * (-12 * 72) + 48 * 12) / 3600 = -2016 / 3600$

Fixing e(5) to cycle 4 sets the final scheduling plan:

i	s(i)	l(i)	j(i)	t(i)	U(i,0)	U(i,1)	U(i,2)	U(i,3)	U(i,4)	U(i,5)	U(i,6)
1	1	1	1	1			60/60				
2	3	3	2	2				60/60	60/60		
3	5	5	1	1						60/60	
4	6	6	1	1							60/60
5	4	4	1	1					60/60		
6	0	0	2	2	60/60	60/60					
7	3	3	1	1				60/60			
W(1,t)					0	0	60/60	60/60	60/60	60/60	60/60
W(2,t)					60/60	60/60	0	60/60	60/60	0	0
C(1,t)					60/60	60/60	60/60	60/60	60/60		
C(2,t)					60/60	60/60	0	60/60	60/60		

Which, with a latency of 7 and restart time of 5 requires one processor of both types (for comparison: pipeline-aware hardware-bound list scheduling produced L=9).

References

- [1] R. Camposano
From behaviour to structure: High-level synthesis, IEEE Design & Test of Comput., 10 (1990) 8-19.
- [2] R. Camposano and W. Rosenstiel
Synthesizing circuits from behavioural descriptions, IEEE Trans. Comput. Aided Des., 2 (1989) 171-180.
- [3] C.-T. Hwang, J.-H. Lee and Y.-C. Hsu
A formal approach to the scheduling problem in high-level synthesis, IEEE Trans. Comput. Aided Des., 10(4), (April 1991) 464-475.
- [4] P.G. Paulin and J.P. Knight
Force-directed scheduling for the behavioural synthesis of ASICs, IEEE Trans. Comput. Aided Des., 6 (1989) 661-679.
- [5] N. Park and A. Parker
SHEWA: A program for synthesis of pipelines, Proceedings of the 23rd Des. Automation Conference, (1986) 454-460
- [6] P. Marwedel and W. Rosenstiel
Synthese von RT-Strukturen aus Verhaltensbeschreibungen, Informatik Spektrum, 15 (1992) 454-460.
- [7] P. Arató
Logic synthesis of VLSI Structures based on a pipelined data flow model, Department of Process Control, Technical University of Budapest, Report 1990.
- [8] P. Arató and H. Fahner
A scheduling and allocation method for the high-level synthesis of pipelined data path, Department of Process Control, Technical University of Budapest, Report, 1992.
- [9] H. M. Lipp
Entwurf digitaler Schaltungen- Formale Hilfsmittel, Institut für Technik der Informationsverarbeitung, Technical University of Karlsruhe, Germany
- [10] H. T. Kung
Why Systolic Architecture
COMPUTER vol. 15, no. 1. p.37-46, Jan. 1982.

For further reading

High-Level VLSI Synthesis

R. Camposano, W. Wolf

Kluwer Academic Publisher, 1991.

High-level Synthesis

D. Gajski

Kluwer Academic Publisher, 1992.

Software performance optimization

Programming RISC engines

- Neal Margulis

- Doctor Dobb's Journal, February 1990

Programming the Pentium Processor - Ramesh Subramaniam, Kiran Kundargi

- Doctor Dobb's Journal, June 1993

Optimizing Pentium Code

- Mike Schmit

- Doctor Dobb's Journal, January 1994

Pentium Optimizations and Num. Perf.

- Stephen S. Fried.

- Doctor Dobb's Journal, January 1994

Processor realizations

PowerPC 604 RISC Microprocessor Technical Summary

- Order number MPC604/D

- Motorola Inc., 1994.

Scheduling methods

Force-directed scheduling for the behavioral synthesis of ASIC's

- Paulin P.G., Knight J.P.

- IEEE Trans. on CAD, 1989., p. 661.

A Formal Approach to the Scheduling Problem in HLS

- Hwag Ch-T., Lee J-H., Hsu Y-Ch.

- IEEE Trans. on CAD, 1991., p. 464.

PIPE Users Manual

version 1.0

Edited by Szabolcs Szigeti

1995

Department of Process Control

Technical University of Budapest

1. Introduction

Pipe was developed at the *Department of Process Control, Technical University of Budapest*, as an educational software tool for designing pipeline data flow devices.

Pipe uses an elementary operations graph (*EOG*) where the nodes of the graph denote elementary operations and the edges their data-interconnections.

Given a predefined restarting period pipe - if necessary - inserts buffers to meet this period. Synchronisation buffers are also inserted.

Pipes generates different variations of the graph by moving the synchronisation buffers. For every variation allocation is performed: every elementary operation, that are not working concurrently may be combined into one unit. Pipe tries to find these units.

The software itself is written in C++ and runs under several variations of the UNIX multiuser operating system.

2. Usage

The general format of the invocation of the *pipe* program is¹:

```
pipe [-s] [-v] [-b] [-Xd] [-p graph] restart [input_file]
```

The name of the program is *pipe*. This should be in a directory which is accessible by the users (it is in their search path). Some run time parameters may be changed by optional command line switches following the program name. Their order is not important. **Table 1** summarises them.

Switch	Explanation
-s	The scheduling is not tight. Care should be taken when using this switch as it may increase the number of variation by several magnitudes.
-v	Verbose mode. During the processing additional information is displayed. This includes the number of variations, number, place and types of buffers inserted and the current best graph.
-b	Buffers are normally excluded from allocation. This switch forces buffers to be allocated. This may lead to exponentially increased processing times.
-Xd	Activate <i>d</i> debug option. Only valid if pipe is compiled with debugging enabled. More than one debug option may be given, to list currently available options use -X- .
-p <i>graph</i>	Dump the input graph to a file named <i>graph</i> after inserting buffers. Useful for debugging.

Table 1.: Command line switches

The only mandatory command line parameter is the restart time which should follow the switches if any. This should be given as an integer greater than 2.

The last parameter is the name of the input file. If non is given, *pipe* reads its standard input.

The format of the input is described in detail in section 3.

¹ Text in **bold** should be typed exactly as shown, text in *italics* should be replaced by appropriate names and anything between brackets is optional.

3. Input

Pipe uses as simple hardware description language as input. this declares functional elements and gives the interconnection between them.

The following BNF (Backus-Naur Form) description illustrates pipe's input language:

```

graph      := graphid iodesc fedesc graphdesc outcn
graphid    := GRAPH : name
iodesc     := ioitem | iodesc ioitem;
ioitem     := INPUT namelist | OUTPUT namelist;
namelist   := NAME | namelist , name;
fedesc     := feitem | fedesc feitem;
feitem     := PROCESSOR NAME INPUT: NUMBER DELAY: NUMBER |
              PROCESSOR NAME DELAY: NUMBER INPUT: NUMBER |
              PROCESSOR NAME NUMBER NUMBER;
signal     := name Fname (signallist);
signallist := siglistelem | siglist , sigelem;
sigelem    := INPUTname | SIGNALname;
outcn      := OUTname SIGNALname;

```

3.1 Keywords and Identifiers

Inputs, outputs, processors (graph nodes) and interconnections (graph edges) are identified by identifiers of the maximum length of 32 characters². They may contain alphanumerical (a-z, 0-9) characters and underscore. the first character can not be numeral. Case is insensitive³. Forward declarations are not allowed.

The following keywords are reserved, and may not be used as an identifier: **graph**, **input**, **output**, **processor**, **delay**, **out**. Words are separated by blanks and/or tabs.

3.2 Graph declaration

The graph's name is declared by the graph keyword followed by a colon and the name of the graph. The following line declares a graph named *my_graph*:

```
graph:    my_graph
```

² This is a compile time option, and may be changed.

³ Also a compile time option.

3.3 I/O declarations

Inputs and outputs are declared by the input and output keywords, with the I/O identifiers separated by commas. This example declares *in_a*, *in_b* as an input and *out_x* as an output:

```
input:    in_a, in_b
output:   out_x
```

3.4 Processor declarations

The processor keyword is used to declare processing elements. Two properties have to be given here: the number of inputs and the delay (time from valid input to valid output). The following three lines are all valid declarations of a processor named *sum* with 4 inputs and a delay of 10:

```
processor sum 10 4
processor sum delay: 10 input: 4
processor sum input: 4 delay: 10
```

3.5 Processor instantiations

Processors are instantiated in a form similar to a function call: the arguments are the inputs, the value of the function is the output. Inputs may be named, i.e. using the output of a previously instantiated processor, or unnamed, when the input is an other processor. In this example a processor (*divide*) takes *m1* and *m2* as an input and its output is named as *result*:

```
result    divide(m1, m2)
```

Of course, the processor *divide* has to be defined in a processor statement, and must have exactly two inputs.

In a similar fashion, *divide* takes *m1* as one input and the output of *decrement* as the other input:

```
result    divide(m1, decrement(m2))
```

3.6 Output connections

The outputs declared with the output keyword have to be connected to processor outputs. This line connects *result* to *out_x*:

```
out_x     result
```

4. Output

Pipe's output contains the result of allocation: which functional elements are combined into one.

The result is a table showing which elementary operations have been allocated into one processor. The following listing is a sample output from the FIR example (section 3.).

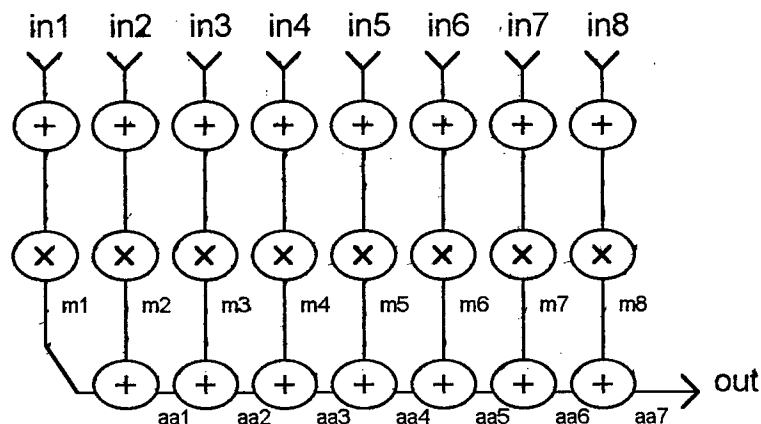
Processor number 10 and 13 contain two operations (aa2, aa7 and aa5, aa6), all the other processors contain only one. Note that allocation does not attempt to deal with operations that will not fit into one processor: in this case multipliers are not allocated, because their delay is more than half restart period. However their number is still given in the results listing.

```
----- Results of the allocation -----
Proc 1 => (a1,adder2)
Proc 2 => (a2,adder2)
Proc 3 => (a3,adder2)
Proc 4 => (a4,adder2)
Proc 5 => (a5,adder2)
Proc 6 => (a6,adder2)
Proc 7 => (a7,adder2)
Proc 8 => (a8,adder2)
Proc 9 => (aa1,adder2)
Proc 10 => (aa2,adder2) (aa7,adder2)
Proc 11 => (aa3,adder2)
Proc 12 => (aa4,adder2)
Proc 13 => (aa5,adder2) (aa6,adder2)
Proc 14 => (aa7,adder2)
Processor: adder2 -- 14
Processor: mult -- 8
```

Number of buffers: 56

5. Example

The FIR filter (see the graph below) is a simple device containing adders and multipliers.



The following listing describes the FIR filter for pipe.

Graph: FIR_FILTER

Input: in1, in2, in3, in4, in5, in6, in7, in8

Output: out

Processor	adder1	delay:	2	input:	1
Processor	adder2	delay:	2	input:	2
Processor	mult	delay:	5	input:	1

```

m1  mult (adder1 (in1))
m2  mult (adder1 (in2))
m3  mult (adder1 (in3))
m4  mult (adder1 (in4))
m5  mult (adder1 (in5))
m6  mult (adder1 (in6))
m7  mult (adder1 (in7))
m8  mult (adder1 (in8))

```

```

aa1  adder2  (m1, m2)
aa2  adder2  (aa1, m3)
aa3  adder2  (aa2, m4)
aa4  adder2  (aa3, m5)
aa5  adder2  (aa4, m6)
aa6  adder2  (aa5, m7)
aa7  adder2  (aa6, m8)

```

```

out  aa7

```

6. Installing and porting

Pipe is distributed in C++ source. To compile, you will need the followings:

- A **UNIX** or UNIX like operating system⁴. Pipe is verified to work under SunOS 4.1, HP-UX 8, HP-UX 9 and NetBSD-1.0.
- A C++ compiler. During development the Free Software Foundation's **G++** compiler⁵ (versions 2.5.4 and 2.7.0) was used.
- **Yacc** or equivalent compiler-compiler. The precompiled grammar is provided in the file `gram.cc`. If you do not make changes in the grammar file, it is possible to install pipe without yacc.

First unpack the compressed tar archive using the following command:

```
zcat pipe.tar.Z | tar xvf -
```

This should create a directory named `pipe`. Go to this directory. There is a configuration script, run it:

```
./configure
```

If necessary, edit the file `conf.h`, it contains some values that you might wish to change.

Start compilation:

```
make
```

After a while an executable named `pipe` should appear. Move this file where other users can access it.

Pipe was written by having portability a goal. However due to some incompatibility between the different UNIX systems, you may have to change the source. These changes should not be difficult.

⁴ Pipe was compiled under MS-DOS, but is not guaranteed to work because the awkward memory management scheme of this system. It would probably mean little trouble to compile it under OS/2 or Windows NT.

⁵ Available on the Internet from `prep.ai.mit.edu` via anonymous ftp.

Standard Benchmark Set Solved in the Frame of the Curriculum

edited by
István Jankovits

**Technical University of Budapest
Faculty of Electrical Engineering and Informatics
Department of Process Control**

Benchmarks

This report summarises the works has been done at the Technical University of Budapest, Department of Process Control on the field of high-level synthesis. The benchmarks that can be found on the next pages has been worked out for the students of the course which name is High-Level Synthesis of digital designs. The students had 1 month to solve the problems as it is defined on each cover page. The students worked in pairs. Every group had to solve 8 problems:

- 1. The Data-Flow Graph**

(Generating a DFG -or Elementary Operation Graphf EOG- from the textual description of the task)

- 2. The VHDL Behavioral Description**

- 3. The Input File of the WinSam**

- 4. WinSam Results**

(The WinSam is a test version of a force-directed based HLS tool has been developed by a Ph.D. student in the Process Control Department)

- 5. The Input File of the PIPE**

- 6. PIPE Results**

(see Users Manual)

(One student has to solve points 3 and 4, while the other has to complete points 5 and 6.)

- 7. The Scheduled Data-Flow Graph**

(The students has to choose one solution and has to generate the DFG of the scheduled and allocated structure.)

- 8. Structural VHDL Description**

Contents

- I. Elliptic Filter
- II. FIR Filter
- III. Aproximation of an Angles Cosinus
- IV. Aproximation of an Angles Sinus
- V. Fast Fourier Transformation
- VI. Expansion of a 3×3 Determinant
- VII. Differential Equation Solver
- VIII. Conclusion

Elliptic Filter

Design a unit which can solve the 5th order elliptic filtering. The unit should work with pipelined restarting period.

To solve the problems only the following elementary operations can be used:

*Buffer, INC, DEC, +, -, *, DIV, MOD*

The execution timeratios are (in the same order) :

1 / 2 / 2 / 4 / 4 / 8 / 12 / 12 [clock cycle]

Tasks

The list of the students is located in the SEEGER server in the \UAGUEST\LOGIC\NAGYHF SUBDIRECTORY. In order to solve the following problems the first students in the list should use the *WinSam*, and the second ones should use the PIPE design tool. The subdirectories, where the design tools can be found, have the same name as the design tool has.

Solve the following problems

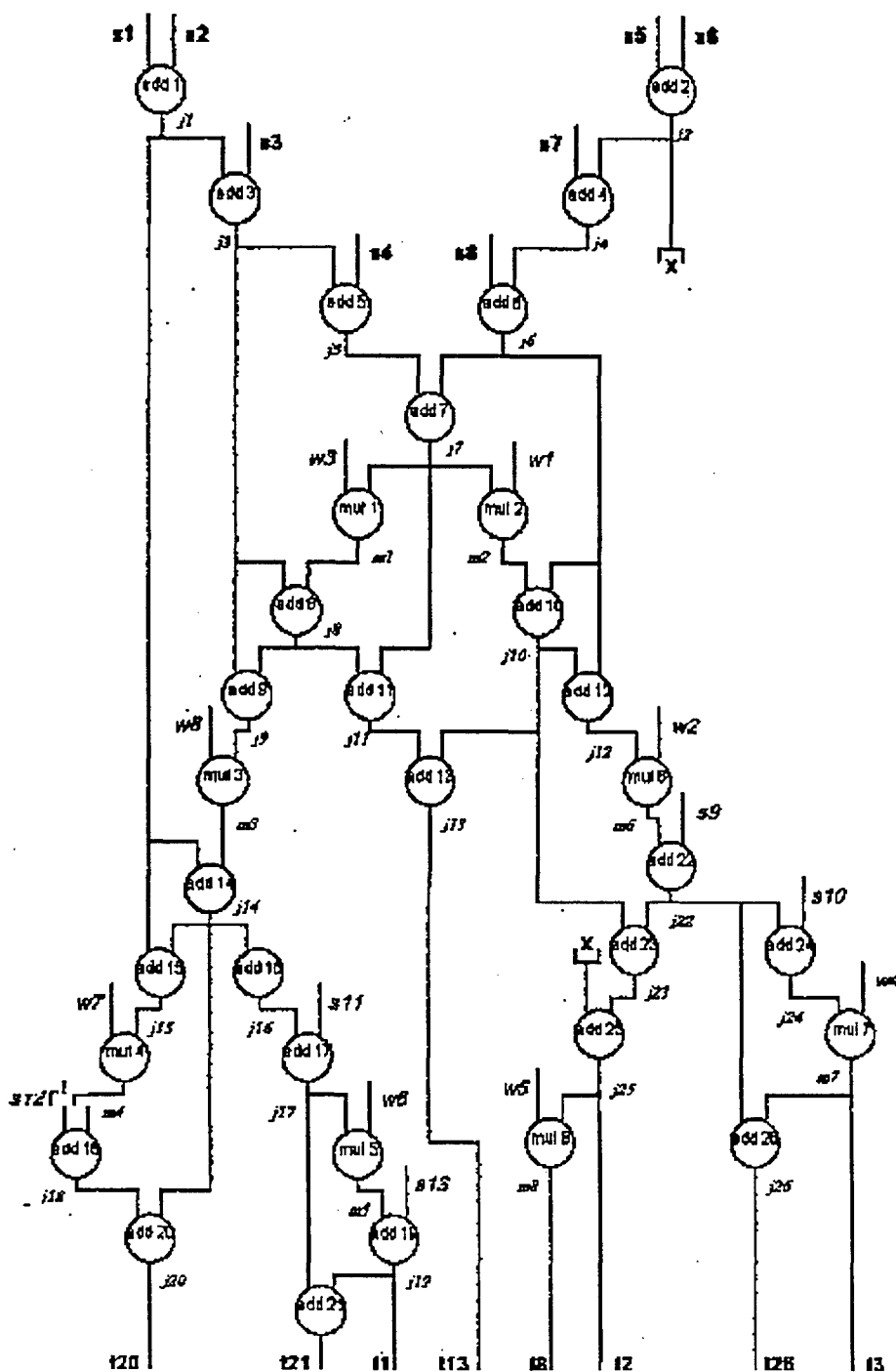
- generate the Data-Flow Graph (DFG) from the problem
- produce the behavioral description of the DFG in VHDL
- produce the description of the DFG in the hardware description language (HDL) of the given design tool (PIPE or WinSam)
- Run the design tool with the input file that was generated in point c) with different parameters such as:
 - WinSam: (*restart_s/restart_f*) 14/16, 14/18, 14/20, 16/16, 16/20;
 - PIPE: 14, 16, 18, 20, 22;
- compare the results, analyse the reasons of the output results and propose a moution of the best design.
- produce the structural description of the resulted structure and its controle in VHDL.
- simulate the structural VHDL description was produced in point f)

The format of the ideal solution

- The format of the report must be Win Word 2.0
- The coverpage should contain:
 - title of the task
 - name and group of the student
- The format of the pictures must be PCX
- The final file (report) should be copied into the directory:
/U/GUEST/LOGIC/BEAD
(Once a file was created it cannot be deleted, but a second copy with different name might be accepted.)
- The name of the file must be: second_name.DOC
- The date of the handing in is the date when the file is created.

The deadline of the handing in is the last day of the semester !

1. The Data-Flow Graph



2. The VHDL Behavioral Description

ENTITY IIR IS

PORT (S1, S2, S3, S4, S5, S6, S7, S8 : IN INTEGER; J20, J21, J19, J13,
M08, J26, M07 : OUT INTEGER);
END IIR;

SIGNAL

J01, J02, J03, J04, J05, J0, J07, J08, J09, J10, J11, J12, J13, J14, J15, J16, J17,
J18, J19, J20, J21, J22, J23, J24, M01, M02, M03, M04, M05, M06, M07,
M08:INTEGER;

CONSTANT

W1, W2, W3, W4, W5, W6, W7, W8: INTEGER;

ARCHITECTURE FUNC OF IIR IS

J01 <= S1 + S2 AFTER 4NS;
J02 <= S5 + S6 AFTER 4NS;
J03 <= J01 + S3 AFTER 4NS;
J04 <= J02 + S7 AFTER 4NS;
J05 <= J03 + S4 AFTER 4NS;

J06 <= J04 + S8 AFTER 4NS;
J07 <= J05 + J06 AFTER 4NS;
J08 <= J03 + M01 AFTER 4NS;
J09 <= J03 + J08 AFTER 4NS;
J10 <= M02 + J06 AFTER 4NS;

J11 <= J08 + J07 AFTER 4NS;
J12 <= J10 + J06 AFTER 4NS;
J13 <= J11 + J10 AFTER 4NS;
J14 <= J01 + M03 AFTER 4NS;
J15 <= J01 + J14 AFTER 4NS;

J16 <= J14 + M03 AFTER 4NS;
J17 <= J16 + S11 AFTER 4NS;
J18 <= S12 + M04 AFTER 4NS;
J19 <= M05 + J06 AFTER 4NS;
J20 <= J18 + J14 AFTER 4NS;

J21 <= J17 + J19 AFTER 4NS;
J22 <= M06 + S9 AFTER 4NS;
J23 <= J10 + J22 AFTER 4NS;
J24 <= J22 + S10 AFTER 4NS;

J25 <= J02 + J23 AFTER 4NS;
J26 <= J22 + M07 AFTER 4NS;

M01 <= J07 * W3 AFTER 8NS;
M02 <= J07 * W1 AFTER 8NS;
M03 <= W8 * J09 AFTER 8NS;
M04 <= W7 * J15 AFTER 8NS;
M05 <= J17 * W6 AFTER 8NS;
M06 <= J12 * W2 AFTER 8NS;
M07 <= J24 * W4 AFTER 8NS;
M08 <= W5 * J25 AFTER 8NS;

END FUNC;

3. The Input File of the WinSam

```
restart_s : 20
restart_f : 28
net_num: 1
cover_num: 1
sync_weight: 99
max_net: 1
step : 1
not_same : 1
add_time : 0
in,0,0,0,0,8:add1,add1,add3,add5,add2,add2,add4,add6
add1 4,1,0,2,3:add3,add14,add15
add2 4,1,0,2,2:add4,add25
add3 4,1,0,2,3:add5,add8,add9
add4 4,1,0,2,1:add6
add5 4,1,0,2,1:add7
add6 4,1,0,2,3:add7,add10,add12
add7 4,1,0,2,5:mul1,mul1,add11,mul2,mul2
add8 4,1,0,2,2:add9,add11
add9 4,1,0,2,2:mul3,mul3
add10 4,1,0,2,3:add12,add13,add23
add11 4,1,0,2,1:add13
add12 4,1,0,2,2:mul6,mul6
add13 4,1,0,2,1:out
add14 4,1,0,2,3:add15,add16,add20
add15 4,1,0,2,2:mul4,mul4
add16 4,1,0,2,2:add17,add17
add17 4,1,0,2,3:add21,mul5,mul5
add18 4,1,0,2,1:add20
add19 4,1,0,2,2:add21,out
add20 4,1,0,2,1:out
add21 4,1,0,2,1:out
add22 4,1,0,2,4:add23,add26,add24,ad24
add23 4,1,0,2,1:add25
add24 4,1,0,2,2:mul7,mul7
add25 4,1,0,2,3:out,mul8,mul8
add26 4,1,0,2,1:out
mul1,12,1,0,2,1:add8
mul2,12,1,0,2,1:add10
mul3,12,1,0,2,1:add14,add16
mul4,12,1,0,2,1:add18,add18
mul5,12,1,0,2,1:add19
mul6,12,1,0,2,1:add22,add22
mul7,12,1,0,2,1:add26,out
mul8,12,1,0,2,1:out
```

out,0,0,0,8,0:

4. WinSam Results

restart_s	restart_f	elements	buffers
14	18	27	39
16	16	30	38
14	20	27	41
16	20	27	41
20	20	31	17
14	24	26	34
16	24	26	34
20	24	32	15
24	24	32	15
14	28	22	28
16	28	22	28
20	28	25	9
24	28	25	9
28	28	25	9
16	30	21	25
20	30	26	5
24	30	26	5
28	30	26	5
30	30	26	5

5. The Input File of the PIPE

graph: ellip

input: s1,s2,s3,s4,s5,s6,s7,s8

output: t20,t21,t1,t13,t8,t2,t26,t3

processor add 4 2

processor mul 8 2

j1 add(s1,s2)

j2 add(s5,s6)

j3 add(j1,s3)

j4 add(j2,s7)

j5 add(j3,s4)

j6 add(j4,s8)

j7 add(j5,j6)

m1 mul(j7,j7)

m2 mul(j7,j7)

j8 add(j3,m1)

j10 add(j6,m2)

j9 add(j3,j8)

j11 add(j8,j7)

j12 add(j10,j6)

m3 mul(j9,j9)

j13 add(j11,j10)

m6 mul(j12,j12)

j14 add(j1,m3)

j22 add(m6,m6)

j15 add(j1,j14)

j16 add(j14,m3)

j23 add(j10,j22)

j24 add(j22,j22)

m4 mul(j15,j15)

j17 add(j16,j16)

j25 add(j2,j23)

m7 mul(j24,j24)

j18 add(m4,m4)

m5 mul(j17,j17)

m8 mul(j25,j25)

j26 add(j22,m7)

j20 add(j18,j14)

j19 add(m5,m5)

j21 add(j17,j19)

t1 j19

t2 j25

t3 m7

t20 j20

t21 j21

t13 j13

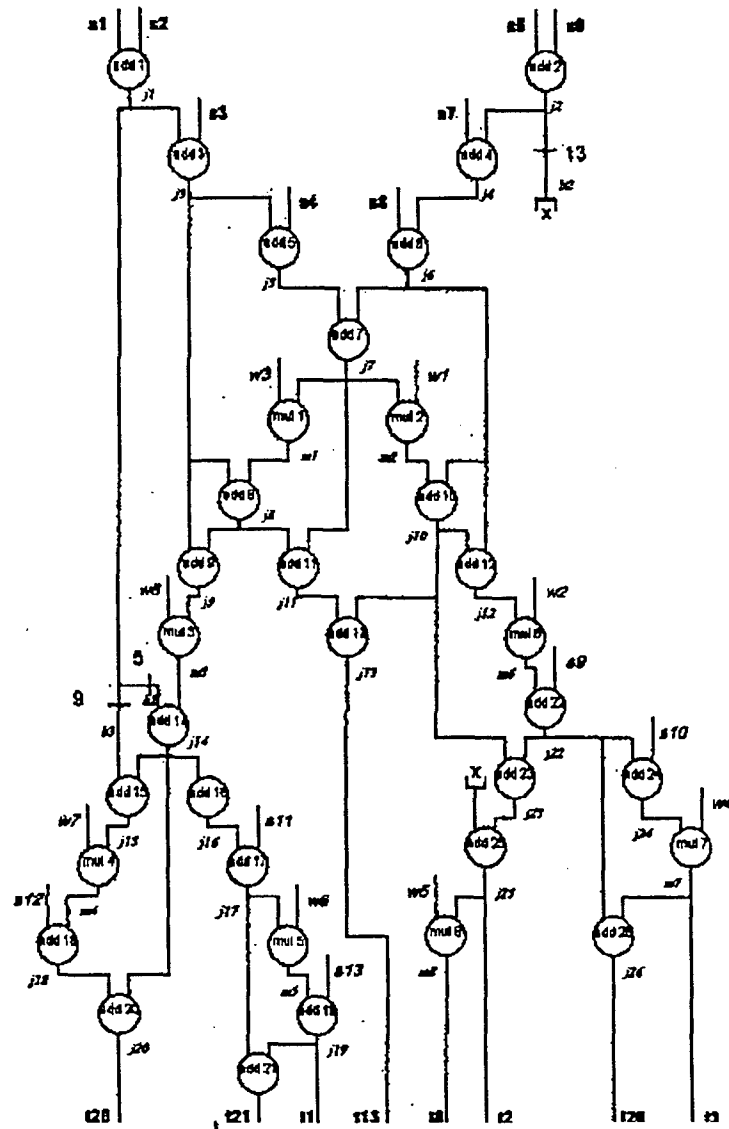
t8 m8

t26 j26

6 PIPE Results

R	Processors	Buffers
5	34	540
10	32	466
20	23	244
30	16	145
40	16	115
50	13	71
60	10	56

7. The Scheduled Data-Flow Graph



8. Structural VHDL Description:

PACKAGE Global IS

SIGNAL CLK:Integer:=0;

SIGNAL buf0_en:Integer:=0;

SIGNAL buf1_en:Integer:=0;

SIGNAL buf2_en:Integer:=0;

SIGNAL buf3_en:Integer:=0;

SIGNAL buf4_en:Integer:=0;

SIGNAL buf5_en:Integer:=0;

SIGNAL buf6_en:Integer:=0;

SIGNAL buf7_en:Integer:=0;

SIGNAL buf8_en:Integer:=0;

SIGNAL add01_en:Integer:=0;

SIGNAL add02_en:Integer:=0;

SIGNAL add03_en:Integer:=0;

SIGNAL add04_en:Integer:=0;

SIGNAL add05_en:Integer:=0;

SIGNAL add06_en:Integer:=0;

SIGNAL add07_en:Integer:=0;

SIGNAL add08_en:Integer:=0;

SIGNAL add09_en:Integer:=0;

SIGNAL add10_en:Integer:=0;

SIGNAL add11_en:Integer:=0;

SIGNAL add12_en:Integer:=0;

SIGNAL add13_en:Integer:=0;

SIGNAL add14_en:Integer:=0;

SIGNAL add15_en:Integer:=0;

SIGNAL add16_en:Integer:=0;

SIGNAL add17_en:Integer:=0;

SIGNAL add18_en:Integer:=0;

SIGNAL add19_en:Integer:=0;

SIGNAL add20_en:Integer:=0;

SIGNAL add21_en:Integer:=0;

SIGNAL add22_en:Integer:=0;

SIGNAL add23_en:Integer:=0;

SIGNAL add24_en:Integer:=0;

SIGNAL add25_en:Integer:=0;

SIGNAL add26_en:Integer:=0;

SIGNAL mul1_en:Integer:=0;

SIGNAL mul2_en:Integer:=0;

SIGNAL mul3_en:Integer:=0;

SIGNAL mul4_en:Integer:=0;
SIGNAL mul5_en:Integer:=0;
SIGNAL mul6_en:Integer:=0;
SIGNAL mul7_en:Integer:=0;
SIGNAL mul8_en:Integer:=0;

CONSTANT period:Integer:=28;

CONSTANT buf0_ofs:Integer:=25;
CONSTANT buf1_ofs:Integer:=29;
CONSTANT buf2_ofs:Integer:=33;
CONSTANT buf3_ofs:Integer:=5;
CONSTANT buf4_ofs:Integer:=9;
CONSTANT buf5_ofs:Integer:=9;
CONSTANT buf6_ofs:Integer:=29;
CONSTANT buf7_ofs:Integer:=53;
CONSTANT buf8_ofs:Integer:=49;

CONSTANT add01_ofs:Integer:=0;
CONSTANT add02_ofs:Integer:=0;
CONSTANT add03_ofs:Integer:=4;
CONSTANT add04_ofs:Integer:=4;
CONSTANT add05_ofs:Integer:=8;
CONSTANT add06_ofs:Integer:=8;
CONSTANT add07_ofs:Integer:=12;
CONSTANT add08_ofs:Integer:=28;
CONSTANT add09_ofs:Integer:=32;
CONSTANT add10_ofs:Integer:=28;
CONSTANT add11_ofs:Integer:=32;
CONSTANT add12_ofs:Integer:=32;
CONSTANT add13_ofs:Integer:=36;
CONSTANT add14_ofs:Integer:=48;
CONSTANT add15_ofs:Integer:=52;
CONSTANT add16_ofs:Integer:=52;
CONSTANT add17_ofs:Integer:=56;
CONSTANT add18_ofs:Integer:=68;
CONSTANT add19_ofs:Integer:=72;
CONSTANT add20_ofs:Integer:=72;
CONSTANT add21_ofs:Integer:=76;
CONSTANT add22_ofs:Integer:=48;
CONSTANT add23_ofs:Integer:=52;
CONSTANT add24_ofs:Integer:=52;
CONSTANT add25_ofs:Integer:=56;
CONSTANT add26_ofs:Integer:=68;

CONSTANT mul1_ofs:Integer:=16;
CONSTANT mul2_ofs:Integer:=16;
CONSTANT mul3_ofs:Integer:=36;

```
CONSTANT mul4_ofs:Integer:=56;
CONSTANT mul5_ofs:Integer:=60;
CONSTANT mul6_ofs:Integer:=36;
CONSTANT mul7_ofs:Integer:=56;
CONSTANT mul8_ofs:Integer:=60;
```

```
FUNCTION start(offset, act:Integer) RETURN Boolean;
END Global;
```

```
PACKAGE BODY Global IS
```

```
FUNCTION start(offset, act: Integer)RETURN Boolean IS
BEGIN
  IF act<offset THEN
    RETURN FALSE;
  ELSIF (act-offset) MOD period=0 THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END start;
```

```
END Global;
```

```
USE Global.ALL;
```

```
ENTITY scheduler IS
END;
```

```
ARCHITECTURE sched OF scheduler IS
BEGIN
  sched proc:
  PROCESS
  VARIABLE act : Integer:=0;
  BEGIN
```

```
    IF start(buf0_ofs,act) THEN buf0_en<=1;ELSE buf0_en<=0; END IF;
    IF start(buf1_ofs,act) THEN buf1_en<=1;ELSE buf1_en<=0; END IF;
    IF start(buf2_ofs,act) THEN buf2_en<=1;ELSE buf2_en<=0; END IF;
    IF start(buf3_ofs,act) THEN buf3_en<=1;ELSE buf3_en<=0; END IF;
    IF start(buf4_ofs,act) THEN buf4_en<=1;ELSE buf4_en<=0; END IF;
    IF start(buf5_ofs,act) THEN buf5_en<=1;ELSE buf5_en<=0; END IF;
    IF start(buf6_ofs,act) THEN buf6_en<=1;ELSE buf6_en<=0; END IF;
    IF start(buf7_ofs,act) THEN buf8_en<=1;ELSE buf7_en<=0; END IF;
    IF start(buf8_ofs,act) THEN buf8_en<=1;ELSE buf8_en<=0; END IF;
```

```

IF start(add01_ofs,act) THEN add01_en<=1;ELSE add01_en<=0; END IF;
IF start(add02_ofs,act) THEN add02_en<=1;ELSE add02_en<=0; END IF;
IF start(add03_ofs,act) THEN add03_en<=1;ELSE add03_en<=0; END IF;
IF start(add04_ofs,act) THEN add04_en<=1;ELSE add04_en<=0; END IF;
IF start(add05_ofs,act) THEN add05_en<=1;ELSE add05_en<=0; END IF;
IF start(add06_ofs,act) THEN add06_en<=1;ELSE add06_en<=0; END IF;
IF start(add07_ofs,act) THEN add07_en<=1;ELSE add07_en<=0; END IF;
IF start(add08_ofs,act) THEN add08_en<=1;ELSE add08_en<=0; END IF;
IF start(add09_ofs,act) THEN add09_en<=1;ELSE add09_en<=0; END IF;
IF start(add10_ofs,act) THEN add10_en<=1;ELSE add10_en<=0; END IF;
IF start(add11_ofs,act) THEN add11_en<=1;ELSE add11_en<=0; END IF;
IF start(add12_ofs,act) THEN add12_en<=1;ELSE add12_en<=0; END IF;
IF start(add13_ofs,act) THEN add13_en<=1;ELSE add13_en<=0; END IF;
IF start(add14_ofs,act) THEN add14_en<=1;ELSE add14_en<=0; END IF;
IF start(add15_ofs,act) THEN add15_en<=1;ELSE add15_en<=0; END IF;
IF start(add16_ofs,act) THEN add16_en<=1;ELSE add16_en<=0; END IF;
IF start(add17_ofs,act) THEN add17_en<=1;ELSE add17_en<=0; END IF;
IF start(add18_ofs,act) THEN add18_en<=1;ELSE add18_en<=0; END IF;
IF start(add19_ofs,act) THEN add19_en<=1;ELSE add19_en<=0; END IF;
IF start(add20_ofs,act) THEN add20_en<=1;ELSE add20_en<=0; END IF;
IF start(add21_ofs,act) THEN add21_en<=1;ELSE add21_en<=0; END IF;
IF start(add22_ofs,act) THEN add22_en<=1;ELSE add22_en<=0; END IF;
IF start(add23_ofs,act) THEN add23_en<=1;ELSE add23_en<=0; END IF;
IF start(add24_ofs,act) THEN add24_en<=1;ELSE add24_en<=0; END IF;
IF start(add25_ofs,act) THEN add25_en<=1;ELSE add25_en<=0; END IF;
IF start(add26_ofs,act) THEN add26_en<=1;ELSE add26_en<=0; END IF;

```

```

IF start(mul1_ofs,act) THEN mul1_en<=1;ELSE mul1_en<=0; END IF;
IF start(mul2_ofs,act) THEN mul2_en<=1;ELSE mul2_en<=0; END IF;
IF start(mul3_ofs,act) THEN mul3_en<=1;ELSE mul3_en<=0; END IF;
IF start(mul4_ofs,act) THEN mul4_en<=1;ELSE mul4_en<=0; END IF;
IF start(mul5_ofs,act) THEN mul5_en<=1;ELSE mul5_en<=0; END IF;
IF start(mul6_ofs,act) THEN mul6_en<=1;ELSE mul6_en<=0; END IF;
IF start(mul7_ofs,act) THEN mul7_en<=1;ELSE mul7_en<=0; END IF;
IF start(mul8_ofs,act) THEN mul8_en<=1;ELSE mul8_en<=0; END IF;

```

```

act :=act+1;
CLK<=act;
IF act=100 THEN WAIT;-- 100 periodus
END PROCESS;

```

```

END sched;

```

FIR Filter

Design a unit which can solve the 8th order FIR filtering. The unit should work with pipelined restarting period.

To solve the problems only the following elementary operations can be used:

*Buffer, INC, DEC, +, -, *, DIV, MOD*

The execution timeratios are (in the same order) :

1 / 2 / 2 / 4 / 4 / 8 / 12 / 12 [clock cycle]

Tasks

The list of the students is located in the SEEGER server in the \U\GUEST\LOGIC\NAGYHF SUBDIRECTORY. In order to solve the following problems the first students in the list should use the *WinSam*, and the second ones should use the PIPE design tool. The subdirectories, where the design tools can be found, have the same name as the design tool has.

Solve the following problems

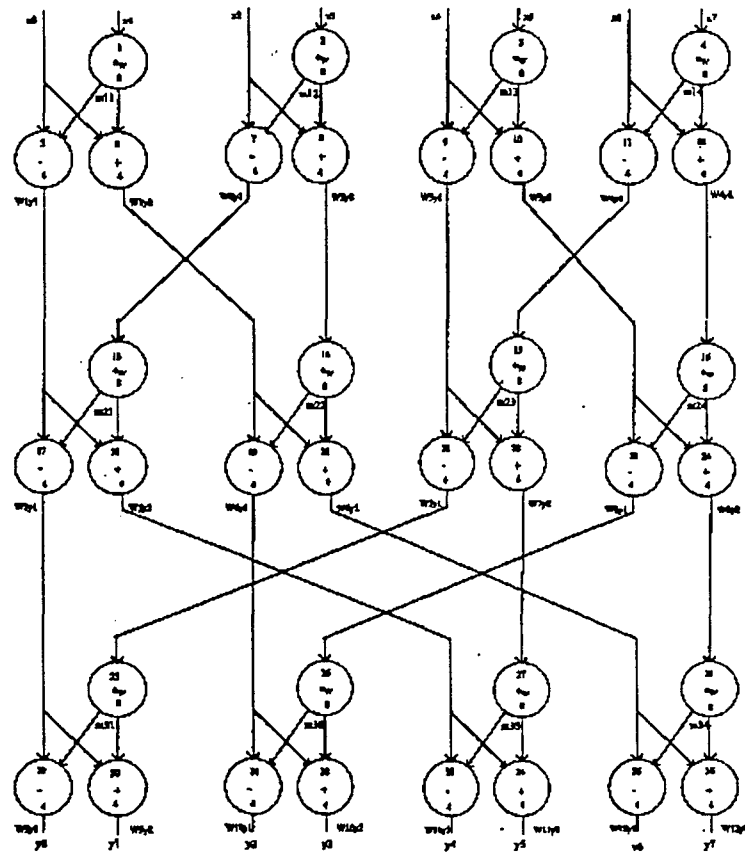
- a) generate the Data-Flow Graph (DFG) from the problem
- b) produce the behavioral description of the DFG in VHDL
- c) produce the description of the DFG in the hardware description language (HDL) of the given design tool (PIPE or WinSam)
- d) Run the design tool with the input file that was generated in point c) with different parameters such as:
 - WinSam: (*restart_s/restart_f*) 14/16, 14/18, 14/20, 16/16, 16/20;
 - PIPE: 14, 16, 18, 20, 22;
- e) compare the results, analyse the reasons of the output results and propose a moution of the best design.
- f) produce the structural description of the resulted structure and its controle in VHDL.
- g) simulate the structural VHDL description was produced in point f)

The format of the ideal solution

- a) The format of the report must be Win Word 2.0
- b) The coverpage should contain:
 - title of the task
 - name and group of the student
- c) The format of the pictures must be PCX
- d) The final file (report) should be copied into the directory:
/U/GUEST/LOGIC/BEAD
(Once a file was created it cannot be deleted, but a second copy with different name might be accepted.)
- e) The name of the file must be: second_name.DOC
- f) The date of the handing in is the date when the file is created.

The deadline of the handing in is the last day of the semester !

1. The Data-Flow Graph



2. The VHDL Behavioral Description

```
ENTITY mul IS
    PORT (x: IN Adat; y: OUT Adat); END mul ;
ARCHITECTURE vis OF mul IS
BEGIN
    y <= x*w AFTER 8ns
END vis
```

```
ENTITY add IS
    PORT (x,y: IN Adat; z: OUT Adat); END add ;
ARCHITECTURE vis OF add IS
BEGIN
    z <= x+y AFTER 4ns
END vis ;
```

```
ENTITY sub IS
    PORT ( x, y : IN Adat ; z : OUT Adat ) ; END sub ;
ARCHITECTURE vis OF sub IS
BEGIN
    z <= x-y AFTER 4ns
END vis
```

```
ENTITY fft IS
    PORT (x0,x1,x2,x3,x4,x5,x6,x7: IN Adat; y0, y1, y2, y3, y4, y5, y6, y7 : OUT
        Adat ) ;
END fft;
```

```
ARCHITECTURE struc OF fft IS
    COMPONENT mul PORT (x: IN Adat; y: OUT Adat); END COMPONENT
    COMPONENT add PORT (x,y: IN Adat; z: OUT Adat); END COMPONENT
    COMPONENT sub PORT (x,y: IN Adat; z: OUT Adat); END COMPONENT
```

```
SIGNAL ll,m12,m13,m14,wly1,wly2,w2y1,w2y2,w3y1,w3y2,w4y1,w4y2,
m21,m22,m23,m24,w5y1,w5y2,w6y1,w6y2,w7y1,w7y2,w8y1,w8y2,m31,m32,m33,m34;
BEGIN
    m11: mul PORT MAP(x4);
    m12: mul PORT MAP(x6);
    m13: mul PORT MAP(x5);
    m14: mul PORT MAP(x7);
```

wlyl: sub PORT MAP (x0,m11);
w2yl: sub PORT MAP (x2,m12);
w3yl: sub PORT MAP (x1,m13);
w4yl: sub PORT MAP (x3,m14);
wly2: add PORT MAP (x0,m11);
w2y2: add PORT MAP (x2,m12);
w3y2: add PORT MAP (x1,m13);
w4y2: add PORT MAP (x3,m14);

m21: mul PORT MAP(w2yl);
m22: mul PORT MAP(w2y2);
m23: mul PORT MAP(w4yl);
m24: mul PORT MAP(w4y2);

w5yl: sub PORT MAP (wlyl,m21);
wóyl: sub PORT MAP (wly2,m22);
w7yl: sub PORT MAP (w3yl,m23);
w8yl: sub PORT MAP (w3y2,m24);
w5y2: add PORT MAP (wlyl,m21);
w6y2: add PORT MAP (wly2,m22);
w7y2: add PORT MAP (w3yl,m23);
w8y2: add PORT MAP (w3y2,m24);

m31: mul PORT MAP(w7yl);
m32: mul PORT MAP(w8yl);
m33: mul PORT MAP(w7y2);
m34: mul PORT MAP(w8y2);

w9yl: sub PORT MAP (w5yl,m31);
wl0yl: sub PORT MAP (wóyl,m32);
wllyl: sub PORT MAP (w5y2,m33);
wl2yl: sub PORT MAP (wóy2,m34);
w9y2: add PORT MAP (w5yl,m31);
wlOy2: add PORT MAP (wóyl,m32);
wlly2: add PORT MAP (w5y2,m33);
wl2y2: add PORT MAP (wóy2,m34);

END struct

3. The Input File of the WinSam

```
restart_s:6 restart_f:6
net_num:1
cover_num:1
sync_weight:4
max_net:1
step:1
not_same:1
add_time:0
in,0,0,0,0,2:ina,inb
ina,0,0,0,1,6:m1,m2,m3,m4,s1,s2
inb,0,0,0,1,6:s3,s4,a1,a2,a3,a4
m1,8,1,0,1,2:s1,a1
m2,8,1,0,1,2:s2,a2
m3,8,1,0,1,2:s3,a3
m4,8,1,0,1,2:s4,a4
s1,4,1,0,2,2:s5,a5
s2,4,1,0,2,1:m5
s3,4,1,0,2,2:s7,a7
s4,4,1,0,2,1:m7
a1,4,1,0,2,2:s6,a6
a2,4,1,0,2,1:m6
a3,4,1,0,2,2:s8,a8
a4,4,1,0,2,1:m8
m5,8,1,0,1,2:s5,a5
m6,8,1,0,1,2:s6,a6
m7,8,1,0,1,2:s7,a7
m8,8,1,0,1,2:s8,a8
s5,4,1,0,2,2:s9,a9
s6,4,1,0,2,2:s10,a10
s7,4,1,0,2,1:m9
s8,4,1,0,2,1:m10
a5,4,1,0,2,2:s11,a11
a6,4,1,0,2,2:s12,a12
a7,4,1,0,2,1:m11
a8,4,1,0,2,1:m12
m9,8,1,0,1,2:s9,a9
m10,8,1,0,1,2:s10,a10
m11,8,1,0,1,2:s11,a11
m12,8,1,0,1,2:s12,a12
s9,4,1,0,2,1:out
s10,4,1,0,2,1:out
s11,4,1,0,2,1:out
s12,4,1,0,2,1:out
a9,4,1,0,2,1:out
```

a10,4,1,0,2,1:out
all,4,1,0,2,1:out
a12,4,1,0,2,1:out
out,0,0,0,8,0:

4. WinSam Results

r_s	r_f	processors	buffers
6	10	36	60
7	10	36	60
9	12	36	56
10	10	36	56
10	13	36	56
12	12	36	56
12	15	36	48
12	18	24	40
13	13	36	16
13	15	32	16
14	14	36	16
15	15	32	16

5. The Input File of the PIPE

input: x0,x1,x2,x3,x4,x5,x6,x7

output: y0,y1,y2,y3,y4,y5,y6,y7

processor add 4 2

processor sub 4 2

processor mul 8 1

m11 mul (x4)

m12 mul (x6)

m13 mul (x5)

m14 mul (x7)

w1y1 sub (m11, x0)

w2y1 sub (m12, x2)

w3y1 sub (m13, x1)

w4y1 sub (m14, x3)

w1y2 add (m11, x0)

w2y2 add (m12, x2)

w3y2 add (m13, x1)

w4y2 add (m14, x3)

m21 mul (w2y1)

m22 mul (w2y2)

m23 mul (w3y1)

m24 mul (w3y2)

w5y1 sub (m21, w1y1)

w6y1 sub (m22, w1y2)

w7y1 sub (m23, w3y1)

w8y1 sub (m24, w3y2)

w5y2 add (m21, w1y1)

w6y2 add (m22, w1y2)

w7y2 add (m23, w3y1)

w8y2 add (m24, w3y2)

m31 mul (w5y1)

m32 mul (w6y1)

m33 mul (w5y2)

m34 mul (w6y2)

w9y1 sub (m31, w7y1)

w10y1 sub (m32, w8y1)

w11y1 sub (m33, w7y2)

w12y1 sub (m34,w8y2)

w9y2 add (m31, w7yl)

wl0y2 add (m32,w8yl)

wlly2 add (m33,w7y2)

wl2y2 add (m34,w8y2)

y0 w9y1

y1 w9y2

y2 w10y1

y3 w10y2

y4 w11y1

y5 w11y2

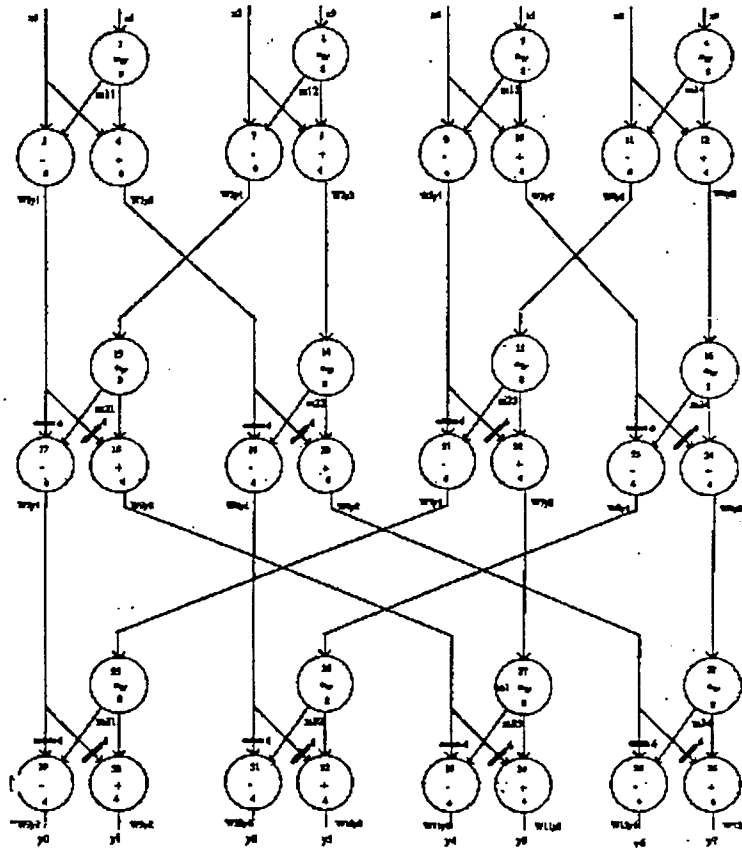
y6 w12y1

y7 w12y2

6 PIPE Results

R	L	n(add)	n(sub)	n(mul)	buff	cost
6	39	12	12	12x2	436	724
8	48	12	12	12x2	340	628
10	41	10	10	12	208	360
11	41	8	8	12	184	344
12	41	10	10	12	160	336
13	36	12	12	12	64	224
14	36	10	10	12	32	256
15	36	8	8	12	16	192
16	36	8	8	12	0	176
17	36	8	8	12	0	160
18	36	8	8	12	0	160
19	36	8	8	12	0	160
20	36	6	6	12	0	144
26	36	8	8	8	0	128
30	36	4	4	8	0	96

7. The Scheduled Data-Flow Graph



8. Structural VHDL Description:

ENTITY mul IS

PORT (x: IN Adat; y: OUT Adat); END mul ;

ARCHITECTURE vis OF mul IS

BEGIN

y <= x*w AFTER 8ns END vis

ENTITY add IS

PORT (x,y: IN Adat; z: OUT Adat); END add ;

ARCHITECTURE vis OF add IS

BEGIN

z <= x+y AFTER 4ns END vis ;

ENTITY sub IS

PORT (x,y: IN Adat; z: OUT Adat); END sub ;

ARCHITECTURE vis OF sub IS

BEGIN

z <= x-y AFTER 4ns

END vi 5

ENTITY buf IS

PORT (x: IN Adat; y: OUT Adat); END buf;

ARCHITECTURE vis OF buf IS

BEGIN

y <= x AFTER 4ns END vis

ENTITY fft IS

PORT (x0,x1,x2,x3,x4,x5,x6,x7 : IN Adat; y0, y1, y2, y3, y4, y5, y6, y7 : OUT Adat) ;

END fft;

ARCHITECTURE struc OF fft IS

COMPONENT mul PORT (x: IN Adat; y: OUT Adat);

END COMPONENT

COMPONENT buf PORT (x: IN Adat; y: OUT Adat);

END COMPONENT

COMPONENT add PORT (x,y: IN Adat; z: OUT Adat);

END COMPONENT

```
COMPONENT sub PORT (x,y: IN Adat; z: OUT Adat);  
END COMPONENT  
SIGNAL ll,m12,m13,m14,wly1,wly2,w2yl,w2y2,w3yl,w3y2,w4yl,w4y2,  
m21,m22,m23,m24,w5yl,w5y2,w6yl,w6y2,w7yl,w7y2,w8yl,w8y2,m31,m32,  
m33,m34,bl,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15, bló;
```

```
BEGIN
```

```
m11: mul PORT MAP(x4);  
m12: mul PORT MAP(x6);  
m13: mul PORT MAP(x5);  
m14: mul PORT MAP(x7);
```

```
wly1: sub PORT MAP (x0,m11);  
w2yl: sub PORT MAP (x2,m12);  
w3yl: sub PORT MAP (x1,m13);  
w4yl: sub PORT MAP (x3,m14);  
wly2: add PORT MAP (x0,m11);  
w2y2: add PORT MAP (x2,m12);  
w3y2: add PORT MAP (x1,m13);  
w4y2: add PORT MAP (x3,m14);
```

```
bl: buf PORT MAP(wly1);  
b2: buf PORT MAP(wly1);  
b3: buf PORT MAP(wly2);  
b4: buf PORT MAP(wly2);  
b5: buf PORT MAP(w3yl);  
b6: buf PORT MAP(w3yl);  
b7: buf PORT MAP(w3y2);  
b8: buf PORT MAP(w3y2);
```

```
m21: mul PORT MAP(w2yl);  
m22: mul PORT MAP(w2y2);  
m23: mul PORT MAP(w4yl);  
m24: mul PORT MAP(w4y2);
```

```
w5yl: sub PORT MAP (bl,m21);  
w6yl: sub PORT MAP (b3,m22);  
w7yl: sub PORT MAP (b5,m23);  
w8yl: sub PORT MAP (b7,m24);  
w5y2: add PORT MAP (b2,m21);  
w6y2: add PORT MAP (b4,m22);  
w7y2: add PORT MAP (b6,m23);  
w8y2: add PORT MAP (b8,m24);
```

```
m31: mul PORT MAP(w7yl);  
m32 : mul PORT MAP (w8yl ) ;  
m33: mul PORT MAP(w7y2);  
m34: mul PORT MAP(w8y2);
```

b9: buf PORT MAP(w5yl)
b10: buf PORT MAP(w5yl)
b11: buf PORT MAP(wóyl)
b12: buf PORT MAP(wóyl)
b13: buf PORT MAP(w5y2)
b14 : buf PORT MAP (w5y2)
b15: buf PORT MAP(wóy2)
b16: buf PORT MAP(wóy2)

w9yl: sub PORT MAP (b9,m31);
wl0yl: sub PORT MAP (b11,m32);
wllyl: sub PORT MAP (b13,m33);
wl2yl: sub PORT MAP (b15,m34);
w9y2: add PORT MAP (b10,m31);
wl0y2: add PORT MAP (b12,m32);
wlly2: add PORT MAP (b14,m33);
wl2y2: add PORT MAP (b16,m34);

END stuc

Aproximation of an angles cosinus

Design a unit which can calculate the cosinus of an angle which is between 0 and Π . The unit should work with pipelined restarting period.

To solve the problems only the following elementary operations can be used:

*Buffer, INC, DEC, +, -, *, DIV, MOD*

The execution timeratios are (in the same order) :

1 / 2 / 2 / 4 / 4 / 8 / 12 / 12 [clock cycle]

Tasks

The list of the students is located in the SEEGER server in the \U\GUEST\LOGIC\NAGYHF SUBDIRECTORY. In order to solve the following problems the first students in the list should use the *WinSam*, and the second ones should use the PIPE design tool. The subdirectories, where the design tools can be found, have the same name as the design tool has.

Solve the following problems

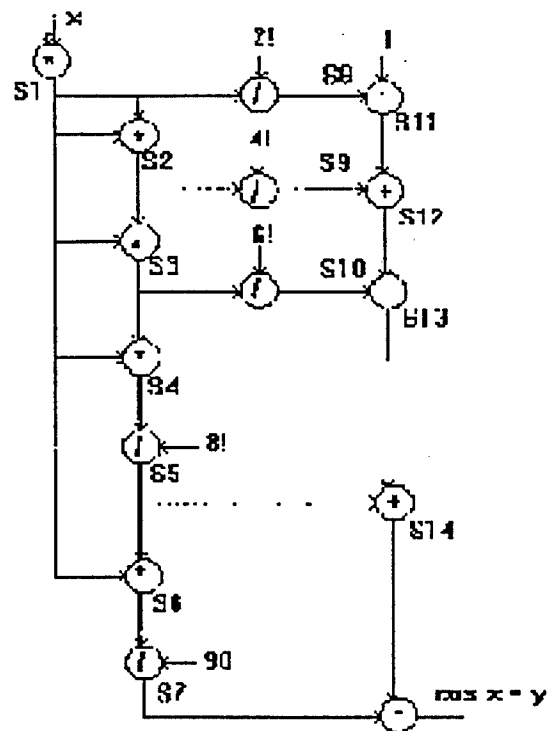
- a) generate the Data-Flow Graph (DFG) from the problem
- b) produce the behavioral description of the DFG in VHDL
- c) produce the description of the DFG in the hardware description language (HDL) of the given design tool (PIPE or WinSam)
- d) Run the design tool with the input file that was generated in point c) with different parameters such as:
 - WinSam: (*restart_s/restart_f*) 14/16, 14/18, 14/20, 16/16, 16/20;
 - PIPE: 14, 16, 18, 20, 22;
- e) compare the results, analyse the reasons of the output results and propose a moution of the best design.
- f) produce the structural description of the resulted structure and its controle in VHDL.
- g) simulate the structural VHDL description was produced in point f)

The format of the ideal solution

- a) The format of the report must be Win Word 2.0
- b) The coverage should contain:
 - title of the task
 - name and group of the student
- c) The format of the pictures must be PCX
- d) The final file (report) should be copied into the directory:
/U/GUEST/LOGIC/BEAD
(Once a file was created it cannot be deleted, but a second copy with different name might be accepted.)
- e) The name of the file must be: second_name.DOC
- f) The date of the handing in is the date when the file is created.

The deadline of the handing in is the last day of the semester !

1. The Data-Flow Graph



1.2.2.2.

2. The VHDL Behavioral Description

ENTITY cosszam IS

PORT (x: IN Integer; z : OUT Integer);

END cosszam ;

ARCHITECTURE func OF cosszam IS

SIGNAL S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13,

S14 : Integer ; BEGIN

S1 <= (x*x)/256	AFTER 8 ns; -- mult1
S2 <= {S1*S1)/256	AFTER 8 ns; -- mult2
S3 <= (S1*S2)/256	AFTER 8 ns; -- mult3
S4 <= (S1*S3)/256	AFTER 8 ns; -- mult4
S5 <= S4/40320	AFTER 12 ns; -- div1
S6 <= (S1*S5)/256	AFTER 8 ns; -- mult5
S7 <= S6/90	AFTER 12 ns; -- div2
S8 <= S1/2	AFTER 12 ns; -- div3
S9 <= S2/24	AFTER 12 ns; -- div4
S10 <= S3/720	AFTER 12 ns; -- div5
S11 <= 256-S8	AFTER 4 ns; -- sub1
S12 <= S11+S9	AFTER 4 ns; -- add1
S13 <= S12-S10	AFTER 4 ns; -- sub2
S14 <= S13+S5	AFTER 4 ns; -- add2
z <= S14-S7	AFTER 4 ns; -- sub3

END func;

3. The Input File of the WinSam

```
restart_s:15
restart_f:15
net_num:1
cover_num:1
sync_weight:4
max_net:1
step:1
not_same:1
add_time:0
in,0,0,0,0,1:mul1
mul1,8,1,0,1,6:div1,mul2,mul2,mul3,mul4,mul5
div1,12,1,0,1,1:sub1
sub1,4,1,0,1,1:add1
mul2,8,1,0,2,2:div2,mul3
div2,12,1,0,1,1:add1
add1,4,1,0,2,1:sub2
mul3, 8,1,0,2,2:div3,mul4
div3,12,1,0,1,1:sub2
sub2,4,1,0,2,1:add2
mul4,8,1,0,2,1:div4
div4,12,1,0,1,2:add2,mul5
add2,4,1,0,2,1:sub3
mul5,8,1,0,2,1:div5
div5,12,1,0,1,1:sub3
sub3,4,1,0,2,1:out
out,0,0,0,1,0:
```

4. WinSam Results

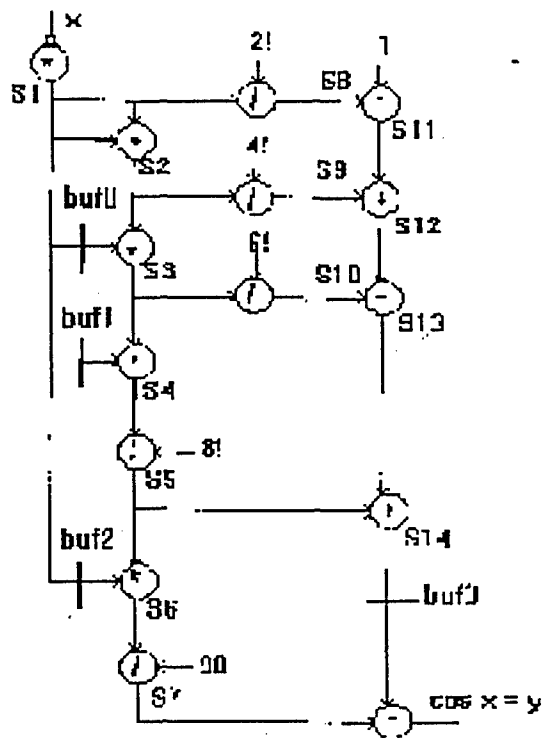
Restart_s	Restart_f	Processzoro k	Bufferek
14	14	15	18
14	20	12	15
14	26	11	32
14	32	8	11
22	22	14	4
26	26	14	2
26	32	13	2
26	38	12	1
36	36	12	1
36	48	9	1

5. The Input File of the PIPE

```
#cos_számitás
graph:cos
input:x
output:result
processor mult 8 2
processor div 12 2
processor add 4 2
processor subb 4 2
s1 mult(x,x)
s2 mult(s 1,s 1 )
s3 mult(s1,s2)
s4 mult(s1,s3)
s5 div(s4,s4)
s6 mult(s1,s5)
s7 div{s6,s6)
s8 div(s1,s1 )
s9 div(s2,s2)
s10 div(s3,s;)
s11 sub{s8,s8)
s12 add(s9,s11)
s13 sub(s12,s10)
s14 add(s5,s13)
re sub(s14,s7)
result re
```

Restart period	Processors				Cost	Buffer nu.	Latency
(R)	div(5)	mult(5)	add(2)	sub(3)	(C)	(B)	(L)
68	2	3	1	1	56	0	68
51	3	3	1	1	68	2/88	68
34	4	5	1	1	96	19/88	68
26	5	5	1	2	112	34/88	68
23	5	5	1	3/2	116/112	44/88	68
20	5	5	2	2	116	66/98	71
16	5	5	2	2	120	79/119	75
14	5	5	2	3/2	120/116	87/119	75
13	10	5	1	2	172	150/17	87
						0	
11	10	5	2	2	176	142/13	79
						8	

7. The Scheduled Data-Flow Graph



8. Structural VHDL Description:

ENTITY cosszam IS

PORT {x : IN Integer; z : OUT Integer}; END cosszam;

ARCHITECTURE func OF cosszam IS

SIGNAL

S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18 :

Integer; SIGNAL

S19,S20,S21,S22,S23,S24,S25,S26,S27,S28,S29,S30,S31,S32:

Integer; SIGNAL S33,S34,S35,S36 : Integer;

BEGIN

```

S1 <=(x*x)/256      AFTER 8 ns; --mull
S2 <=S1             AFTER 1 ns; --buf1
S3 <=S1             AFTER 1 ns; --buf2
S8 <=S7             AFTER 1 ns; --buf5
S9 <=S1             AFTER 19ns;   --buf6
S10 <=((58*S9)/256) AFTER 8 ns; --mul4
S11 <=S10           AFTER 1ns;  --buf7
S12 <=S11/40320     AFTER 12ns;  --div1
S13 <=S12           AFTER 1ns;  --buf8
S14 <=S1            AFTER 1ns;  --bufp
S15 <=((S13*S14)/256) AFTER 8ns;  --mul5
S16 <=S15           AFTER 1ns;  --buf10
S17 <=S16/90        AFTER 12ns;  --div2
S18 <=S1            AFTER 1ns;  --buf11
S19 <=S18/2         AFTER 12ns;  --div3
S20 <=S4            AFTER 1ns;  --buf12
S21 <=520124        AFTER 12ns;  --div4
S22 <=S7            AFTER 1ns;  --buf13
S23 <=S22/1720      AFTER 12 ns;  --div5
S24 <=S19           AFTER 1ns;  --buf14
S25 <=256-S24       AFTER 4ns;  --sub1&3
S26 <=S25           AFTER 5ns;  --buf15
S27 <=S21           AFTER 1ns;  --buf16
S28 =S26+S27        AFTER 4ns;  --add1
S29 =528            AFTER 5ns;  --buf17
S30 =S23            AFTER 1ns;  --buf18
S31 <=S29-S30       AFTER 4ns;  --sub2
S32 <=S31           AFTER 5ns;  --buf19
S33 <=S12           AFTER 1ns;  --buf20
S34 <=S32+S33       AFTER 4ns;  --add2
S35 <=S34           AFTER 18ns;  --buf21

```

S36 <=S17 AFTER 1ms, --bu122
z <=S35-S36 AFTER 4n.s; --sub18c3
END func;

Aproximation of an angles sinus

Design a unit which can calculate the sinus of an angle which is between 0 and Π . The unit should work with pipelined restarting period.

To solve the problems only the following elementary operations can be used:

*Buffer, INC, DEC, +, -, *, DIV, MOD*

The execution timeratios are (in the same order) :

1 / 2 / 2 / 4 / 4 / 8 / 12 / 12 [clock cycle]

Tasks

The list of the students is located in the SEEGER server in the \UGUEST\LOGIC\NAGYHF SUBDIRECTORY. In order to solve the following problems the first students in the list should use the *WinSam*, and the second ones should use the PIPE design tool. The subdirectories, where the design tools can be found, have the same name as the design tool has.

Solve the following problems

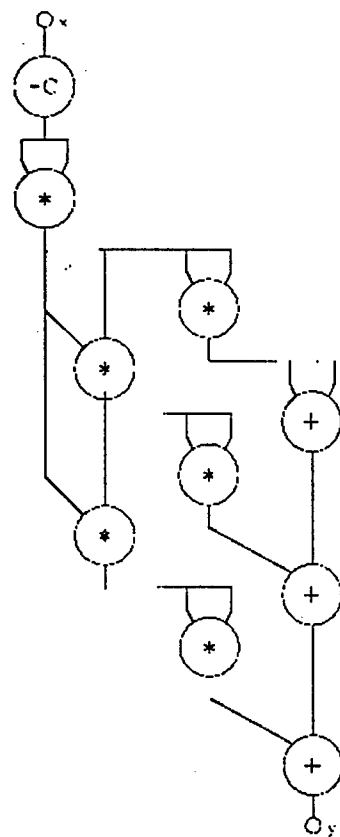
- a) generate the Data-Flow Graph (DFG) from the problem
- b) produce the behavioral description of the DFG in VHDL
- c) produce the description of the DFG in the hardware description language (HDL) of the given design tool (PIPE or WinSam)
- d) Run the design tool with the input file that was generated in point c) with different parameters such as:
 - WinSam: (*restart_s/restart_f*) 14/16, 14/18, 14/20, 16/16, 16/20;
 - PIPE: 14, 16, 18, 20, 22;
- e) compare the results, analyse the reasons of the output results and propose a moution of the best design.
- f) produce the structural description of the resulted structure and its controle in VHDL.
- g) simulate the structural VHDL description was produced in point f)

The format of the ideal solution

- a) The format of the report must be Win Word 2.0
- b) The coverpage should contain:
 - title of the task
 - name and group of the student
- c) The format of the pictures must be PCX
- d) The final file (report) should be copied into the directory:
/U/GUEST/LOGIC/BEAD
(Once a file was created it cannot be deleted, but a second copy with different name might be accepted.)
- e) The name of the file must be: second_name.DOC
- f) The date of the handing in is the date when the file is created.

The deadline of the handing in is the last day of the semester !

1. The Data-Flow Graph



2. The VHDL Behavioral Description

```
PACKAGE Global IS
TYPE Vektor IS ARRAY 0 TO 8 OF INTEGER;
END Global;
USE Global.ALL;
ENTITY Fir_szuro IS
    PORT ( x: IN Vektor; y: OUT INTEGER );
END Fir_szuro;
ARCHITECTURE vis OF Fir_szuro IS
    SIGNAL    a,b : Vektor(1 TO 8);
    SIGNAL    c : Vektor(1 TO 4);
    SIGNAL    d : Vektor(1 TO 2);
    CONSTANT w: Vektor(1 TO 8);
BEGIN
PROCESS
    VARIABLE j,k : INTEGER;
    BEGIN
        FOR i IN 1 TO 8 LOOP
            a(i)<= x(i-1) + x(i) AFTER 4ns;
            b(i)<= a(i) * w(i) AFTER 8ns;
            IF (i MOD 2)=0 THEN j=i/2; c(j)<= b(i-1) + b(i) AFTER 4ns; ENDIF;
            IF (i MOD 4)=0 THEN k=i/4; d(k)<= c(i/2-1) + c(i/2) AFTER 4ns; ENDIF;
        END LOOP;
    END PROCESS ;
    y<= d(1) + d(2) AFTER 4ns;
END vis; .
```

3. The Input File of the WinSam

restart:9
latency:10 net num:1
cover num:1
sync weight:4
max_net:1
step:1
not_sanze:1
add_time:0
in,0,0,0,8:add1,add2,add3,add4,add5,add6,add7,add8
add1,4,1,0,2,1:mul1
add2,4,1,0,2,1:mul2
add3,4,1,0,2,1:mul3
add4,4,1,0,2,1:mul4
add5,4,1,0,2,1:mul5
add6,4,1,0,2,1:mul6
add7,4,1,0,2,1:mul7
add8,4,1,0,2,1:mul8
mul1,8,1,0,1,1:add9
mul2,8,1,0,1,1:add9
mul3,8,1,0,1,1:add10
mul4,8,1,0,1,1:add10
mul5,8,1,0,1,1:add11
mul6,8,1,0,1,1:add11
mul7,8,1,0,1,1:add12
mul8,8,1,0,1,1:add12
add9,4,1,0,2,1:add13
add10,4,1,0,2,1:add13
add11,4,1,0,2,1:add14
add12,4,1,0,2,1:add14
add13,4,1,0,2,1:add15
add14,4,1,0,2,1:add15
add15,4,1,0,2,1:out
out,0,0,0,1,0:

4. WinSam Results

5. The Input File of the PIPE

graph:szuro
input:x0,x1,x2,x3,x4,x5,x6,x7,x8
output:result

processor add 4 2
processor mult 8 2

a1 add (x0,x1)
a2 add (x1,x2)
a3 add (x2,x3)
a4 add (x3,x4)
a5 add (x4,x5)
a6 add (x5,x6)
a7 add (x6,x7) .
a8 add (x7,x8)

m1 mult (a1,a1)
m2 mult (a2,a2)
m3 mult (a3,a3)
m4 mult (a4,a4)
m5 mult (a5,a5)
m6 mult (a6,a6)
m7 mult (a7,a7)
m8 mult (a8,a8)

a11 add (m1,m2)
a12 add (m3,m4)
a13 add (m5,m6)
a14 add (m7,m8)

a21 add (a11,a12)
a22 add (a13,a14)

re add (a21,a22)
result re

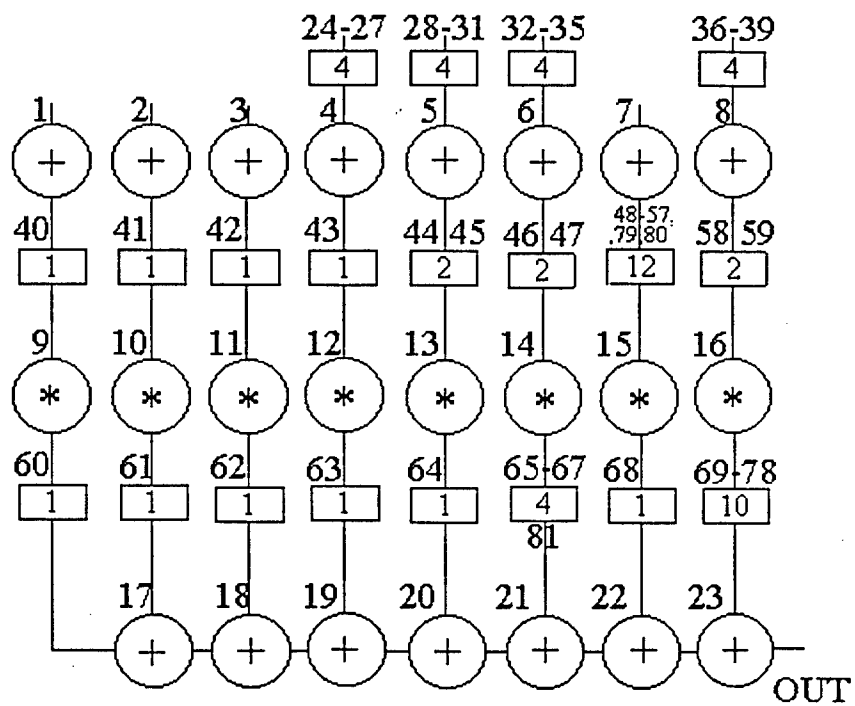
6 PIPE Results

R	6	9	11	12	13	14	18	21	24
L	27	30	26	26	24	24	24	24	24
P(add)	15	15	14	14	14	14	14	11	8
p(mul)	16	16	8	8	8	8	8	8	8
B	110	44	16	16	0	0	0	0	0

7. The Scheduled Data-Flow Graph

FIR filter

INPUT



8. Structural VHDL Description:

PACKAGE Global IS

TYPE Vektor IS ARRAY 0 TO 38 OF INTEGER;

SIGNAL v en IS Vektor:=(0 TO 38 => 0);

CONSTANT v offs IS Vektor:=(0 TO 7 =>0; 8 TO 15 =>4; 16 TO 23 =>5; 24 TO

31=>13; 32 TO 35 =>14;

36 TO 37 =>18; 38 =>22);

CONSTANT period: INTEGER:=10;

SIGNAL Clk: INTEGER:=0;

FUNCTION Start(offset,act: INTEGER) RETURN BOOLEAN;

END Global ;

PACKAGE BODY Global IS

FUNCTION Start(offset,act : INTEGER) RETURN BOOLEAN IS

BEGIN

IF act<offset THEN RETURN FALSE;

ELSIF (act-offset) MOD period=0 RETURN TRUE;

ELSE RETURN FALSE;

END IF;

END Start;

END Global ;

ENTITY Scheduler IS

END;

ARCHITCTURE sched OF Scheduler IS

BEGIN.

PROCESS

VARIABLE act: INTEGER:=0;

BEGIN

FOR i IN 0 TO 38 LOOP

IF Start(v offs(i),act) THEN v en(i)<=1; ELSE v en(i)<=0; END IF;

END LOOP;

act:= act + 1;

Global. Clk<=act;

END PROCES S ;

END sched;

ENTITY buff IS

GENERIC (t: INTEGER :=1);

PORT (Clk,en,x: IN INTEGER; y: OUT INTEGER);
END buff;

ARCHITECTURE func OF buff IS
BEGIN

PROCESS(Clk,en)

VARIABLE started: BOOLEAN:= FALSE;

VARIABLE cnt: INTEGER:= 0;

BEGIN ,

IF en EVENT AND en=1 THEN started:= TRUE; cnt:= t; END IF;

IF started THEN

IF cnt=0 THEN

y<=x ;

started:=FALSE;

END IF;

IF Clk'EVENT THEN cnt:= cnt -1; END IF;

END IF;

END PROCESS;

END func;

ENTITY add IS

GENERIC (t: INTEGER :=4);

PORT (Clk,en,x1,x2: IN INTEGER; y: OUT INTEGER);

END buff;

ARCHITECTURE func OF buff IS
BEGIN

PROCESS(Clk,en)

VARIABLE started: BOOLEAN:= FALSE;

VARIABLE cnt: INTEGER:= 0;

BEGIN

IF en'EVENT AND en=1 THEN started:= TRUE; cnt:= t; END IF;

IF started THEN

IF cnt=0 THEN

y<=x 1 + x2;

started:=FALSE;

END IF;

IF Clk'EVENT THEN cnt:= cnt -1; END IF;

END IF;

END PROCESS ;

END func;

```

ENTITY mul IS
  GENERIC (t: INTEGER :=8);
  PORT (Clk,en,x: IN INTEGER; y: OUT INTEGER);
END buff;
ARCHITECTURE func OF buff IS
BEGIN
  CONSTANT w: INTEGER;
  PROCESS(Clk,en)
    VARIABLE started: BOOLEAN:= FALSE;
    VARIABLE cnt: INTEGER:= 0;
    BEGIN
      IF en EVENT AND en=1 THEN started:= TRUE; cnt:= t; END IF;
      IF started THEN
        IF cnt=0 THEN
          y<=x * w;
          started:=FALSE;
        END IF;
        IF Clk'EVENT THEN cnt:= cnt -1; END IF;
      END IF;
    END PROCESS ;
  END func;

```

```

ENTITY Fir_szuro IS
  PORT ( x: IN Vektor(0 TO 8); y: OUT INTEGER );
  END Fir_szuro ;
ARCHITECTURE vis OF Fir szuro IS
  SIGNAL a,b,bl,b2 : Vektor(1 TO 8);
  SIGNAL      c : Vektor(1 TO 4);
  SIGNAL      d : Vektor(1 TO 2);
  CONSTANT   w: Vektor(1 TO 8);
  BEGIN
  PROCESS
    VARIABLE j,k : INTEGER;
    BEGIN
      FOR i IN 1 TO 8 LOOP
        a(i)<= x(i-1 ) + x(i) AFTER 4ns;
        b 1 (i)<= a(i) AFTER 1 ns ;
        b(i)<= bl(i) * w(i) AFTER 8ns;
        b2(i)<= b(i) AFTER 1ns;
        IF (i MOD 2)=0 THEN j=i/2; c(j)<= b2(i-1) + b2(i) AFTER 4ns;
      ENDIF;
    END PROCESS
  ENDIF;

```

```

        IF (i MOD 4)=0 THEN k=i/4; d(k)<= c(i/2-1 ) + c(i/2) AFTER 4ns ;
    ENDIF;

    END LOOP;

    END PROCESS;

    y<= d(1) + d(2) AFTER 4ns;

    END vis;

```

ARCHITECTURE func OF Fir_szuro IS

COMPONENT buffer

PORT(c,e,x: IN INTEGER; y: OUT INTEGER);

END COMPONENT;

COMPONENT adder

PORT(c,e,x1,x2: IN INTEGER; y: OUT INTEGER)

END COMPONENT;

COMPONENT mult

PORT(c,e,x: IN INTEGER; y: OUT INTEGER);

END COMPONENT;

FOR add 1,add2,add3,add4,add5,add6,add7,add8,add9,add 10,add 11,add 12,

add13,add14,add15 : adder USE ENTITY WORK.add(func);

FOR mull,mul2,mul3,mul4,mul5,mul6,mul7,mul8 :

mult USE ENTITY WORK.mul(func); FOR

OTHERS : buffer USE ENTITY WORK.buff(func);

SIGNAL a,b,b 1,b2 : Vektor(1 TO 8);

SIGNAL c : Vektor(1 TO 4);

SIGNAL d : Vektor(1 TO 2) ;

BEGIN

add1 : adder PORTMAP(Clk,v en(0),x(0),x(1),a(1));

add2 : adder PORTMAP(Clk,v en(1),x(1),x(2),a(2));

add3 : adder PORTMAP(Clk,v en(2),x(2),x(3),a(3));

add4 : adder PORTMAP(Clk,v en(3),x(3),x(4),a(4));

add5 : adder PORTMAP(Clk,v en(4),x(4),x(5),a(5));

add6 : adder PORTMAP(Clk,v en(5),x(5),x(6),a(6));

add7 : adder PORTMAP(Clk,v en(6),x(6),x(7),a(7));

add8 : adder PORTMAP(Clk,v en(7),x(7),x(8),a(8));

buf0 : buffer PORTMAP(Clk,v en(8),a(1),b1(1)) ;

buf1 : buffer PORTMAP(Clk,v en(9),a(2),b1(2));

buf2 : buffer PORTMAP(Clk,v en(10),a(3),b1(3));

buf3 : buffer PORTMAP(Clk,v en(11),a(4),b1(4));

buf4 : buffer PORTMAP(Clk,v en(12),a(5),b1(5));

buf5 : buffer PORTMAP(Clk,v en(13),a(6),b1(6));

```

buf6 : buffer PORTMAP(Clk,v en(14),a(7),bl(7));
buf7 : buffer PORTMAP(Clk,v en(15),a(8),bl(8));
mul1 : mult PORTMAP(Clk,v en(16),b1(1),b(1));
mul2 : mult PORTMAP(Clk,v en(17),bl(2),b(2));
mul3 : mult PORTMAP(Clk,v en(18),b1(3),b(3));
mul4 : mult PORTMAP(Clk,v en(19),bl(4),b(4));
mul5 : mult PORTMAP(Clk,v en(20),b 1 (5),b(5));
mul6 : mult PORTMAP(Clk,v en(21),bl(6),b(6));
mul7 : mult PORTMAP(Clk,v en(22),b 1 (7),b(7));
mul8 : mult PORTMAP(Clk,v en(23),bl(8),b(8));
buf0 : buffer PORTMAP(Clk,v en(24),b(1),b2(1));
buf1 : buffer PORTMAP(Clk,v en(25),b(2),b2(2));
buf2 : buffer PORTMAP(Clk,v en(26),b(3),b2(3));
buf3 : buffer PORTMAP(Clk,v en(27),b(4),b2(4));
buf4 : buffer PORTMAP(Clk,v en(28),b(5),b2(5));
buf5 : buffer PORTMAP(Clk,v en(29),b(6),b2(6));
buf6 : buffer PORTMAP(Clk,v en(30),b(7),b2(7));
buf7 : buffer PORTMAP(Clk,v en(31 ),b(8),b2(8));
add9 : adder PORTMAP(Clk,v en(32),b2(1),b2(2),c(1));
add10 : adder PORTMAP(Clk,v en(33),b2(3),b2(4),c(2));
add11 : adder PORTMAP(Clk,v en(34),b2(5),b2(6),c(3));
add12 : adder PORTMAP(Clk,v en(35),b2(7),b2(8),c(4));
add13 : adder PORTMAP(Clk,v en(36),c( 1 ),c(2),d(1 ));
add14 : adder PORTMAP(Clk,v en(37),c(3),c(4),d(2));
add15 : adder PORTMAP(Clk,v en(38),d(1),d(2),y);

```

END struct;

USE Glogbal.ALL;

Fast Fourier Transformation

Design a unit which can solve the algorithm of the Fast Fourier Transformation (FFT).
The unit should work with pipelined restarting period.

To solve the problems only the following elementary operations can be used:

*Buffer, INC, DEC, +, -, *, DIV, MOD*

The execution timeratios are (in the same order) :

1 / 2 / 2 / 4 / 4 / 8 / 12 / 12 [clock cycle]

Tasks

The list of the students is located in the SEEGER server in the \U\GUEST\LOGIC\NAGYHF SUBDIRECTORY. In order to solve the following problems the first students in the list should use the *WinSam*, and the second ones should use the PIPE design tool. The subdirectories, where the design tools can be found, have the same name as the design tool has.

Solve the following problems

- generate the Data-Flow Graph (DFG) from the problem
- produce the behavioral description of the DFG in VHDL
- produce the description of the DFG in the hardware description language (HDL) of the given design tool (PIPE or WinSam)
- Run the design tool with the input file that was generated in point c) with different parameters such as:
 - WinSam: (*restart_s/restart_f*) 14/16, 14/18, 14/20, 16/16, 16/20;
 - PIPE: 14, 16, 18, 20, 22;
- compare the results, analyse the reasons of the output results and propose a moution of the best design.
- produce the structural description of the resulted structure and its controle in VHDL.
- simulate the structural VHDL description was produced in point f)

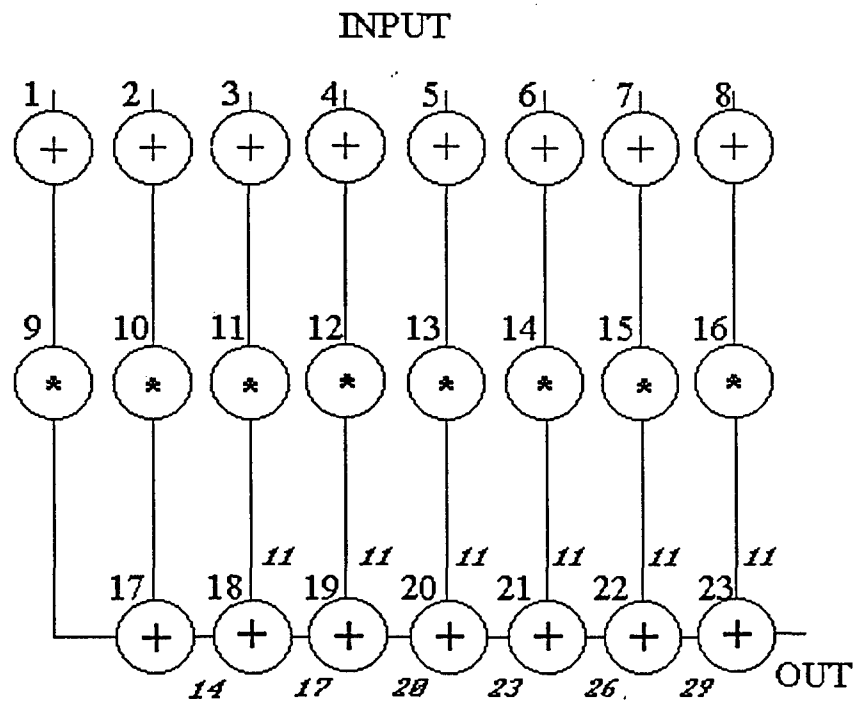
The format of the ideal solution

- The format of the report must be Win Word 2.0
- The coveragepage should contain:
 - title of the task
 - name and group of the student
- The format of the pictures must be PCX
- The final file (report) should be copied into the directory:
/U/GUEST/LOGIC/BEAD
(Once a file was created it cannot be deleted, but a second copy with different name might be accepted.)
- The name of the file must be: *second_name.DOC*
- The date of the handing in is the date when the file is created.

The deadline of the handing in is the last day of the semester !

1. The Data-Flow Graph

FIR filter



2. The VHDL Behavioral Description

```
entity add is
  port (x1,x2: in real; y: out real);
end add;
```

```
architecture behv of add is
  y<=x1+x2 after 4 ns;
end behv;
```

```
entity sub is
  port (x1,x2: in real; y: out real);
end add;
```

```
architecture behv of add is
  y<=x1-x2 after 4 ns;
end behv;
```

```
entity mul is
  port (x1,x2: in real; y: out real);
end add;
```

```
architecture behv of add is
  y<=x1*x2 after 8 ns;
end behv;
```

```
entity cons is
  port (c1,c2,c3,c4,c5:out real);
end cons;
```

```
architecture behv of cons
  c1<=1.570796 after 0 ns;
  c2<=1.0 after 0 ns;
  c3<=0.5 after 0 ns;
  c4<=4.16667e-2 after 0 ns;
  c5<=-1.3889e-3 after 0 ns;
end behv;
```

```
entity sinus is
  port (x: in real; y: out real);
end sinus;
```

architecture struct of sinus is

component add port (x1,x2: in real; y: out real); end component;

component sub port (x1,x2: in real; y: out real); end component;

component mul port (x1,x2: in real; y: out real); end component;

component cons port (c1,c2,c3,c4,c5: out real); end component;

signal c1,c2,c3,c4,c5,s1,s2,s3,s4,s5,s6,s7,s8,s9: real;

begin

cons:cons port map(c1,c2,c3,c4,c5);

sub1:sub port map(x,c1,s1);

mul1:mul port map(s1,s1,s2);

mul2:mul port map(s2,c3,s3);

mul3:mul port map(s2,s2,s4);

add1:add port map(s3,c2,s5);

mul4:mul port map(s4,c4,s6);

mul5:mul port map(s2,s4,s7);

add2:add port map(s5,s6,s8);

mul6:mul port map(s7,c5,s9);

add3:add port map(s8,s9,y);

end struct;

3. The Input File of the WinSam

```
restart_s:16
restart_f:17
net_num:1
cover_num:1
sync_weight:4
max_net:1
step:1
not_same:1
add_time:0
in,0,0,0,0,1:kiv1
kiv1,4,1,0,2,1:szor1
szor1,8,1,0,2,3:szor4,szor2,szor3
szor4,8,1,0,2,1:kiv2
kiv2,4,1,0,2,1:osszl
szor2,8,1,0,2,2:szor5,szor3
szor5,8,1,0,2,1:osszl
szor3,8,1,0,2,1:szor6
szor6,8,1,0,2,1:kiv3
osszl,4,1,0,2,1:kiv3
kiv3,4,1,0,2,1:ki
ki,0,0,0,1,0:
```

4. WinSam Results

Restart for scheduling is 16.

Restart for working is 17.

Additional time to latency is 17.

(Force Directed) Processor elements:9

(szor2,kiv3) (kiv1) (szor1) (szor4) (kiv2) (szor5) (szor3)
(szor6) (osszl)

(Force Directed) Buffer elements:6

(Buf2) (Buf1) (Buf0) (Buf4) (Buf3) (Buf5)

kiv1	n=1	A=0	t=4	:szor1,
szor1	n=1	A=4	t=8	:Buf 0, Buf1, Buf2,
Buf2	n=1	A=12	t=1	:szor3,
Buf1	n=1	A=12	t=1	:szor2,
Buf0	n=1	A=12	t=1	:szor4,
szor4	n=1	A=13	t=8	:kiv2,
kiv2	n=1	A=21	t=4	:osszl,
szor2	n=1	A=13	t=8	:Buf3, Buf4,
Buf4	n=1	A=21	t=1	:szor3,
Buf3	n=1	A=21	t=1	:szor5,
szor5	n=1	A=22	t=8	:osszl,
szor3	n=1	A=22	t=8	:Buf5,
Buf5	n=1	A=30	t=1	:szor6,
szor6	n=1	A=31	t=8	:kiv3,
osszl	n=1	A=30	t=4	:kiv3,
kiv3	n=1	A=39	t=4	:ki,

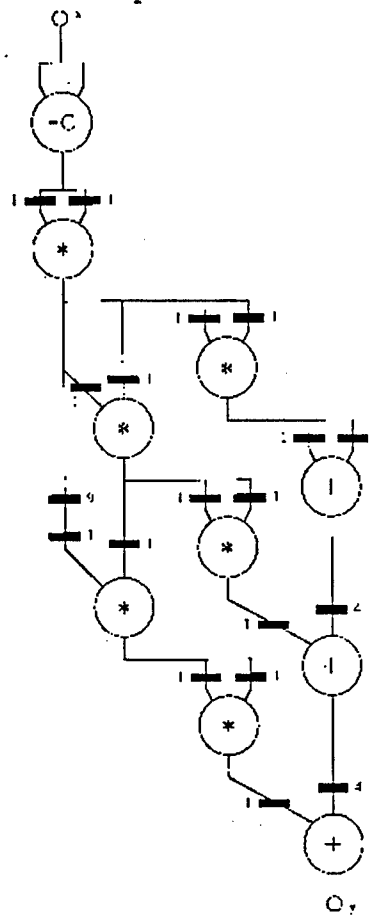
5. The Input File of the PIPE

```
graph:sinus
input:x
output:ki
processor add 4 2
processor sub 4 2
processor mul 8 2
s1 sub(x,x)
s2 mul(s1,s1)
s3 mul(s2,s2)
s4 mul(s2,s2)
s5 add(s3,s3)
s6 mul(s4,s4)
s7 mul(s2,s4)
s8 add(s5,s6)
s9 mul(s7,s7)
y add(s8,s9)
ki y
```

6 PIPE Results

R	L	Cost
4	46	301
6	44	203
8	47	159
10	45	97
12	45	87
14	43	75
16	43	73
18	40	67
20	40	65
22	40	59
24	40	57
26	40	56
28	40	40
30	40	40

7. The Scheduled Data-Flow Graph



8. Structural VHDL Description:

```
ENTITY szinuszképzo IS
PORT (x: IN REAL;clk: IN BIT; y: OUT REAL);
END szinuszképzo;
```

```
ENTITY osszeg IS
PORT (a,b: IN REAL;ck: IN BIT;p: IN INTEGER; ki: OUT REAL);
END osszeg;
```

```
ARCHITECTURE viselk OF osszeg IS
VARIABLE sa: INTEGER;
BEGIN
  PROCESS (a,b,ck,p);
    sa:=0;
    FOR sa+4<p LOOP;
      IF (ck'EVENT AND CK='1') THEN sa:=sa+1;
    END LOOP ;
    ki <= a+b;
  END PROCESS;
END viselk;
```

```
ENTITY kulonb IS
PORT (a,b: IN REAL;ck: IN BIT;p: IN INTEGER; ki: OUT REAL);
END kulonb;
```

```
ARCHITECTURE viselk OF kulonb IS
VARIABLE sa: INTEGER;
BEGIN
  PROCESS (a,b,ck,p);
    sa:=0;
    FOR sa+4<p LOOP;
      IF (ck'EVENT AND CK='1') THEN sa:=sa+1;
    END LOOP;
    ki <= a-b;
  END PROCESS;

END viselk;
```

```
ENTITY szorzo IS
PORT (a,b: IN REAL;ck: IN BIT;p: IN INTEGER; ki: OUT REAL);
END szorzo;
```

```
ARCHITECTURE viselk OF szorzo IS
VARIABLE sa: INTEGER;
BEGIN
  PROCESS (a,b,ck,p);
    sa:=0;
    FOR sa+8<p LOOP;
      IF (ck'EVENT AND CK='1') THEN sa:=sa+1;
    END LOOP;
    ki <= a*b;
```

END PROCESS;

END viselk;

ENTITY puffer IS

PORT (a: IN REAL; ck: IN BIT; p: IN INTEGER; ki: OUT REAL);

END puffer;

ARCHITECTURE viselk OF puffer IS

VARIABLE sa: INTEGER;

BEGIN

PROCESS (a, ck, p);

sa:=0;

FOR sa+1<p LOOP;

IF (ck'EVENT AND CK='1') THEN sa:=sa+1;

END LOOP;

ki <= a;

END PROCESS;

ARCHITECTURE szerk OF szinuszkepzo IS

CONSTANT pif: REAL:=1.57;

CONSTANT ek: REAL:=0.5;

CONSTANT egy: REAL:=1.0;

CONSTANT eh: REAL:=0.0417;

CONSTANT hh: REAL:=0.00139;

CONSTANT s1: INTEGER:=0;

CONSTANT s2: INTEGER:=4;

CONSTANT s3: INTEGER:=13;

CONSTANT s4: INTEGER:=21;

CONSTANT s5: INTEGER:=30;

CONSTANT s6: INTEGER:=13;

CONSTANT s7: INTEGER:=22;

CONSTANT s8: INTEGER:=22;

CONSTANT s9: INTEGER:=31;

CONSTANT s10: INTEGER:=39;

CONSTANT sp0: INTEGER:=12;

CONSTANT sp1: INTEGER:=12;

CONSTANT sp2: INTEGER:=12;

CONSTANT sp3: INTEGER:=21;

CONSTANT sp4: INTEGER:=21;

CONSTANT sp5: INTEGER:=30;

SIGNAL k1, sz1, sz2, sz3, sz4, sz5, sz6, k2, k3, os: REAL;

COMPONENT osszeg PORT (a, b: IN REAL; ck: IN BIT; p: IN INTEGER; ki: OUT REAL);

COMPONENT kulonb PORT (a, b: IN REAL; ck: IN BIT; p: IN INTEGER; ki: OUT REAL);

COMPONENT szorzo PORT (a, b: IN REAL; ck: IN BIT; p: IN INTEGER; ki: OUT REAL);

BEGIN

e1: kulonb PORT MAP (pif, x, clk, s1, k1);

e2: szorzo PORT MAP (k1, k1, clk, s2, sz1);

p0: puffer PORT MAP (sz1, clk, sp0, pu0);

e3: szorzo PORT MAP (ek, p0, clk, s3, sz4);

```
e1: kulonb PORT MAP (e9, sz1, clk, s1, k2);
e5: osszeg PORT MAP (k2, sz5, clk, s5, os);
p1: puffer PORT MAP (sz1, clk, sp1, pu1);
e6: szorzo PORT MAP (pu1, pu1, clk, s6, sz2);
p3: puffer PORT MAP (sz2, clk, sp3, pu3);
e7: szorzo PORT MAP (pu3, eh, clk, s7, sz5);
p2: puffer PORT MAP (sz1, clk, sp2, pu2);
p4: puffer PORT MAP (sz2, clk, sp4, pu4);
e8: szorzo PORT MAP (pu4, pu2, clk, s8, sz3);
p5: puffer PORT MAP (sz3, clk, sp5, pu5);
e9: szorzo PORT MAP (pu5, hh, clk, s9, sz6);
e10: kulonb PORT MAP (os, sz6, clk, s10, y);
END szerk;
```

Expansion of 3*3 Determinant

Design a unit which can solve the expansion of a 3*3 determinant. The unit should work with pipelined restarting period.

To solve the problems only the following elementary operations can be used:

*Buffer, INC, DEC, +, -, *, DIV, MOD*

The execution timeratios are (in the same order) :

1 / 2 / 2 / 4 / 4 / 8 / 12 / 12 [clock cycle]

Tasks

The list of the students is located in the SEEGER server in the \UAGUEST\LOGIC\NAGYHF SUBDIRECTORY. In order to solve the following problems the first students in the list should use the *WinSam*, and the second ones should use the PIPE design tool. The subdirectories, where the design tools can be found, have the same name as the design tool has.

Solve the following problems

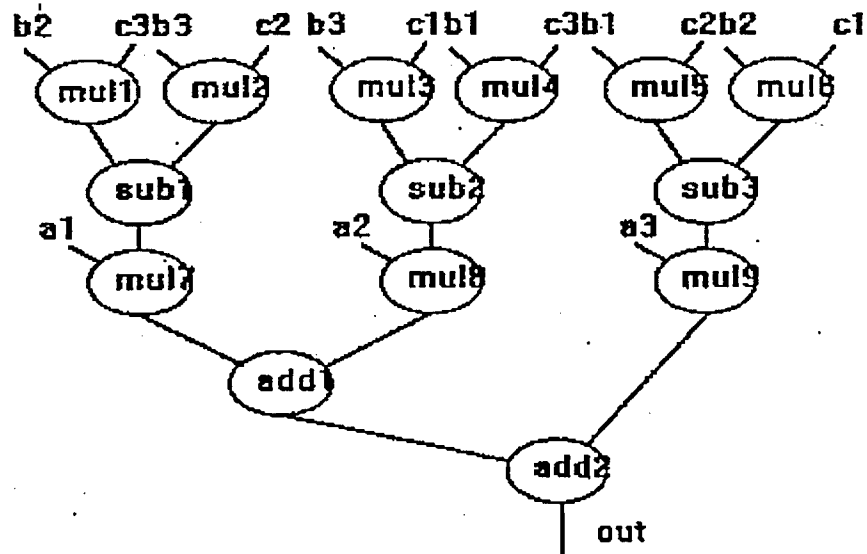
- a) generate the Data-Flow Graph (DFG) from the problem
- b) produce the behavioral description of the DFG in VHDL
- c) produce the description of the DFG in the hardware description language (HDL) of the given design tool (PIPE or WinSam)
- d) Run the design tool with the input file that was generated in point c) with different parameters such as:
 - WinSam: (*restart_s/restart_f*) 14/16, 14/18, 14/20, 16/16, 16/20;
 - PIPE: 14, 16, 18, 20, 22;
- e) compare the results, analyse the reasons of the output results and propose a moution of the best design.
- f) produce the structural description of the resulted structure and its controle in VHDL.
- g) simulate the structural VHDL description was produced in point f)

The format of the ideal solution

- a) The format of the report must be Win Word 2.0
- b) The coverage should contain:
 - title of the task
 - name and group of the student
- c) The format of the pictures must be PCX
- d) The final file (report) should be copied into the directory:
/U/GUEST/LOGIC/BEAD
(Once a file was created it cannot be deleted, but a second copy with different name might be accepted.)
- e) The name of the file must be: second_name.DOC
- f) The date of the handing in is the date when the file is created.

The deadline of the handing in is the last day of the semester !

1. The Data-Flow Graph



2. The VHDL Behavioral Description

ENTITY det IS

PORT (a1, a2, a3, b1, b2, b3, c1, c2, c3 : IN INTEGER; y : OUT INTEGER);
END det;

ARCHITECTURE func OF det IS;

SIGNAL S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13 : INTEGER;

BEGIN

S1 <= b2 * c3 AFTER 8 ns;

S2 <= b3 * c2 AFTER 8 ns;

S3 <= b3 * c1 AFTER 8 ns;

S4 <= b1 * c3 AFTER 8 ns;

S5 <= b1 * c2 AFTER 8 ns;

S6 <= b2 * c1 AFTER 8 ns;

S7 <= S1 - S2 AFTER 4 ns;

S8 <= S3 - S4 AFTER 4 ns;

S9 <= S5 - S6 AFTER 4 ns;

S10 <= a1 * S7 AFTER 8 ns;

S11 <= a2 * S8 AFTER 8 ns;

S12 <= a3 * S9 AFTER 8 ns;

S10 <= S10 + S11 AFTER 4ns;

y <= S13 + S12 AFTER 4ns;

END func;

3. The Input File of the WinSam

```
restart_s : 8
restart_f : 10
net_num: 1
cover_num: 1
sync_weight: 4
max_net: 1
step : 1
not_same : 1
add_time : 0
in ,0,0,0,0,9:mul1,mul2,mul3,mul4,mul5,mul6,mul7,mul8,mul9
mul1,8,1,0,2,1:sub1
mul2,8,1,0,2,1:sub1
mul3,8,1,0,2,1:sub2
mul4,8,1,0,2,1:sub2
mul5,8,1,0,2,1:sub3
mul6,8,1,0,2,1:sub3
sub1,4,1,0,2,1:mul7
sub2,4,1,0,2,1:mul8
sub3,4,1,0,2,1:mul9
mul7,8,1,0,2,1:add1
mul8,8,1,0,2,1:add1
mul9,8,1,0,2,1:add2
add1,4,1,0,2,1:add2
add2,4,1,0,2,1:out
out,0,,0,0,1,0
```

4. WinSam Results

restart_s	restart_f	buffers	proc.
8	10	22	14
8	16	22	14
8	20	17	10
10	24	6	4
10	12	15	12
12	14	15	14
14	16	4	14
14	20	3	14
16	20	3	14
16	18	3	14
10	16	15	12
6	10	22	14
6	14	20	12
6	12	22	12
12	20	11	12

5. The Input File of the PIPE

graph: det2

input: a, b, c, d, f, g, h, i

output: out

processor aos delay:4 input:2

processor mul delay:8 input:2

u1 mul (a, e)

u2 mul (b, f)

u3 mul (c, d)

u4 mul (c, e)

u5 mul (a, f)

u6 mul (b, d)

u7 mul (u1, i)

u8 mul (u2, g)

u9 mul (u3, h)

u10 mul (u4, g)

u11 mul (u5, h)

u12 mul (u6, i)

u13 mul (u7, u8)

u14 mul (u10, u11)

u15 mul (u13, u9)

u16 mul (u14, u12)

u17 mul (u15, u16)

out u17

6 PIPE Results

$$\text{cost} = \sum_i t(i) * \text{nu_of_e}(i)$$

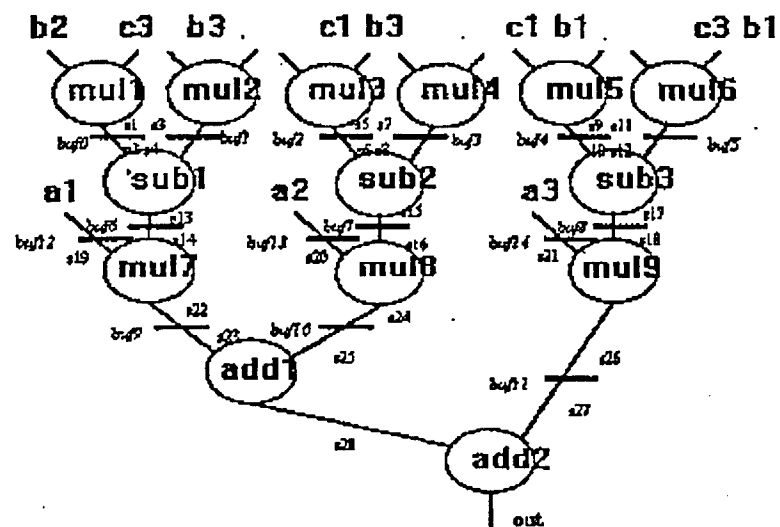
R	cost
4	331
5	259
6	227
7	239
8	263
9	285
10	150
11	146
12	146
13	132
14	132
15	132
16	132
17	132

7. The Scheduled Data-Flow Graph

R=12

sub1, add2 can be allocated

(WINSAM)



8. Structural VHDL Description:

```
ENTITY det IS
PORT (a1,a2,a3,abl,b2,b3, Cl, G2,G3 : f N INTEGER; y : OUT IHTEGER);
END dett;
ARCHITECTURE func OF det IS
SIGNAL S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12: INTEGER;
SIGNAL S13,S14,S15,S16,S17,S18,S19,S20,S21,S22: INTEGER;
SIGNAL S23,S24,S25,S26, s27, s28 : INTEGER;
BEGIN
S1<= b2* c3 AFTER 8ns;
S3<= b3* c2 AFTER 8ns;
S5<= b3* cl AFTER 8ns;
S7<= b1* c3 AFTER 8ns;
S9<= b1 * c2 AFTER 8ns;
S11<= b2* cl AFTER 8 ns;

S2<= S1 AFTER 1ns;
S4<= S3AFTER 1ns;
S6<= S5AFTER 1ns;
S8<= S7 AFTER 1ns;
S10<= S9AFTER 1ns;
S12<= S11AFTER 1ns;

S13<= S2- S4 AFTER 4ns;
S15<= S6- S8 AFTER 4ns;
S17<= S10- S12AFTER 4ns;

S14<= S13AFTER 1ns;
S16<= S15AFTER 1ns;
S18<= S17 AFTER 1ns;

S19<= a1 AFTER 1 ns;
S20<= a2 AETER 1 ns;
S21<= a3 AFTER 1 ns;

S22<= S19 * S14 AFTER 8 ns;
S24<= S20 * S16 AETER 8 ns;
S26<= S21 * S18 AFTER 8 ns;

S23 <= S22 AFTER 1 ns;
S25 <= S24 AFTER 1 ns;
S27 <= S26 AFTER 1 ns;

S28 <= S23 + S25 AFTER 4 ns;
y <= S28 + S27 AFTER 4 ns;
END func;

PACKAGE Global IS
SIGNAL clk : INTEGER := 0;

SIGNAL BUFO_en : INTEGER := 0;
SIGNAL BUF1_en : INTEGER := 0;
SIGNAL BUF2_en : INTEGER := 0;
SIGNAL BUF3_en : INTEGER := 0;
SIGNAL BUF4_en : INTEGER := 0;
```

SIGNAL BUF5_en : INTEGER := 0;
SIGNAL BUF6_en : INTEGER := 0;
SIGNAL BUF7_en : INTEGER := 0;
SIGNAL BUF8_en : INTEGER := 0;
SIGNAL BUF9_en : INTEGER := 0;
SIGNAL BUF10_en : INTEGER := 0;
SIGNAL BUF11_en : INTEGER := 0;
SIGNAL BUF12_en : INTEGER := 0;
SIGNAL BUF13_en : INTEGER := 0;
SIGNAL BUF14_en : INTEGER := 0;

SIGNAL MUL1_en : INTEGER := 0;
SIGNAL MUL2_en : INTEGER := 0;
SIGNAL MUL3_en : INTEGER := 0;
SIGNAL MUL4_en : INTEGER := 0;
SIGNAL MUL5_en : INTEGER := 0;
SIGNAL MUL6_en : INTEGER := 0;

SIGNAL SUB1_en : INTEGER := 0;
SIGNAL SUB2_en : INTEGER := 0;
SIGNAL SUB3_en : INTEGER := 0;

SIGNAL MUL7_en : INTEGER := 0;
SIGNAL MUL8_en : INTEGER := 0;
SIGNAL MUL9_en : INTEGER := 0;

SIGNAL ADD1_en : INTEGER := 0;
SIGNAL ADD2_en : INTEGER := 0;

CONSTANT period : INTEGER := 12;

SIGNAL BUFO_offs : INTEGER := 8;
SIGNAL BUF1_offs : INTEGER := 8;
SIGNAL BUF2_offs : INTEGER := 8;
SIGNAL BUF3_offs : INTEGER := 8;
SIGNAL BUF4_offs : INTEGER := 8;
SIGNAL BUF5_offs : INTEGER := 8;
SIGNAL BUF6_offs : INTEGER := 13;
SIGNAL BUF7_offs : INTEGER := 13;
SIGNAL BUF8_offs : INTEGER := 13;
SIGNAL BUF9_offs : INTEGER := 22;
SIGNAL BUF10_offs : INTEGER := 22;
SIGNAL BUF11_offs : INTEGER := 22;
SIGNAL BUF12_offs : INTEGER := 11;
SIGNAL BUF13_offs : INTEGER := 11;
SIGNAL BUF14_offs : INTEGER := 11;

SIGNAL MUL1_offs : INTEGER := 0;
SIGNAL MUL2_offs : INTEGER := 0;
SIGNAL MUL3_offs : INTEGER := 0;
SIGNAL MUL4_offs : INTEGER := 0;
SIGNAL MUL5_offs : INTEGER := 0;
SIGNAL MUL6_offs : INTEGER := 0;

SIGNAL SUB1_offs : INTEGER := 9;
SIGNAL SUB2_offs : INTEGER := 9;
SIGNAL SUB3_offs : INTEGER := 9;

```

SIGNAL MUL7_offs : INTEGER :=14;
SIGNAL MUL8_offs : INTEGER :=14;
SIGNAL MUL9_offs : INTEGER :=14;

```

```

SIGNAL ADD1_offs : INTEGER :=23;
SIGNAL ADD2_offs : INTEGER :=27;

```

```

FUNCTION start (offset, act : INTEGER) RETURN BOOLEAN;
END Global;

```

```

PACKAGE BODY Global IS

```

```

    FUNCTION start (offset, act : INTEGER) RETURN BOOLEAN IS
    BEGIN
        IF act< offset THEN RETURN FALSE;
        ELSEIF (act-offset) MOD period = 0 THEN RETURN TRUE;
        ELSE RETURN FALSE

```

```

        END IF;
    END start;
END Global;

```

```

USE Global.ALL;

```

```

ENTITY scheduler IS
END;

```

```

ARCHITECTURE sched OF scheduler IS

```

```

BEGIN

```

```

    PROCESS

```

```

        VARIABLE act : INTEGER :=0;

```

```

    BEGIN

```

```

        IF start(BUF0_offs,act) THEN BUFO_en <=1; ELSE BUFO_en <= 0; END IF;
        IF start(BUF1_offs,act) THEN BUFO_en <=1; ELSE BUF1_en <= 0; END IF;
        IF start(BUF2_offs,act) THEN BUFO_en <=1; ELSE BUF2_en <= 0; END IF;
        IF start(BUF3_offs,act) THEN BUFO_en <=1; ELSE BUF3_en <= 0; END IF;
        IF start(BUF4_offs,act) THEN BUFO_en <=1; ELSE BUF4_en <= 0; END IF;
        IF start(BUF5_offs,act) THEN BUFO_en <=1; ELSE BUF5_en <= 0; END IF;
        IF start(BUF6_offs,act) THEN BUFO_en <=1; ELSE BUF6_en <= 0; END IF;
        IF start(BUF7_offs,act) THEN BUFO_en <=1; ELSE BUF7_en <= 0; END IF;
        IF start(BUF8_offs,act) THEN BUFO_en <=1; ELSE BUF8_en <= 0; END IF;
        IF start(BUF9_offs,act) THEN BUFO_en <=1; ELSE BUF9_en <= 0; END IF;
        IF start(BUF10_offs,act) THEN BUFO_en <=1; ELSE BUF10_en <= 0; END IF;
        IF start(BUF11_offs,act) THEN BUFO_en <=1; ELSE BUF11_en <= 0; END IF;
        IF start(BUF12_offs,act) THEN BUFO_en <=1; ELSE BUF12_en <= 0; END IF;
        IF start(BUF13_offs,act) THEN BUFO_en <=1; ELSE BUF13_en <= 0; END IF;

```

```

        IF start (MUL1_offs, act) THEN MUL1_en <= 1; ELSE MUL1_en <= 0 ; END IF;
        IF start (MUL2_offs, act) THEN MUL1_en <= 1; ELSE MUL2_en <= 0 ; END IF;
        IF start (MUL3_offs, act) THEN MUL1_en <= 1; ELSE MUL3_en <= 0 ; END IF;
        IF start (MUL4_offs, act) THEN MUL1_en <= 1; ELSE MUL4_en <= 0 ; END IF;
        IF start (MUL5_offs, act) THEN MUL1_en <= 1; ELSE MUL5_en <= 0 ; END IF;
        IF start (MUL6_offs, act) THEN MUL1_en <= 1; ELSE MUL6_en <= 0 ; END IF;

```

```

        IF start(SUB1_offs,act) THEN SUB1_en <= 1; ELSE SUB1_en <= 0; END IF;
        IF start(SUB2_offs,act) THEN SUB2_en <= 1; ELSE SUB2_en C= 0; END IF;
        IF start (SUB3_offs,act) THEN SUB3_en <= 1; ELSE SUB3_en <= 0; END IF;

```

```
IF start (ADD1_ofs, act) THEN ADD1_en <= 1; ELSE ADD1_en <= 0; END IF;  
IF start(ADD2_ofs,act) THEN ADD2_en <= 1; ELSE ADD2_en <= 0; END IF;
```

```
act :=act + 1;  
clk <= act;  
IF act = 100 THEN WAIT;  
END PROCESS;  
END sched;
```

ENTITY BUFF IS

```
PORT (clk,en : IN INTEGER; a : IN INTEGER; z : OUT INTEGER);  
END BUFF;
```

ARCHITECTURE func OF BUFF IS

```
BEGIN  
PROCESS (clk, en);  
VARIABLE started : BOOLEAN := FALSE;  
VARIABLE cnt : INTEGER : 0;  
BEGIN  
IF en'EVENT AND en = 1 THEN  
started := TRUE;  
cnt :=1;  
END IF;  
IF started THEN  
IF cnt = 0 THEN  
z <= a ;  
started := FALSE;  
END IF ;  
IF clk'EVENT THEN  
cnt := cnt - 1;  
END IF;  
END IF;  
END PROCESS;  
END func;
```

ENTITY ADD IS

```
PORT (clk,en : IN INTEGER; a,b : IN INTEGER ; z : OUT INTEGER);  
END ADD;
```

ARCHITECTURE func OF ADD IS

```
BEGIN  
PROCESS (clk, en);  
VARIABLE started : BOOLEAN := FALSE;  
VARIABLE cnt : INTEGER : 0;  
BEGIN  
IF en'EVENT AND en = 1 THEN  
started := TRUE;  
cnt :=4;  
END IF;  
IF started THEN  
IF cnt = 0 THEN  
z <= a + b;  
started := FALSE;  
END IF ;  
IF clk'EVENT THEN  
cnt := cnt - 1;  
END IF;  
END IF;  
END IF;
```

```
END PROCESS;  
END func;
```

```
ENTITY SUB IS  
  PORT (clk,en : IN INTEGER; a,b : IN INTEGER : z : OUT INTEGER);  
END SUB;
```

```
ARCHITECTURE func OF SUB IS  
BEGIN  
  PROCESS (clk, en);  
    VARIABLE started : BOOLEAN := FALSE;  
    VARIABLE cnt : INTEGER : 0;  
  BEGIN  
    IF en'EVENT AND en = 1 THEN  
      started := TRUE;  
      cnt :=4;  
    END IF;  
    IF started THEN  
      IF cnt = 0 THEN  
        z <= a - b;  
        started := FALSE;  
      END IF ;  
      IF clk'EVENT THEN  
        cnt := cnt - 1;  
      END IF;  
    END IF;  
  END PROCESS;  
END func;
```

```
ENTITY MUL IS  
  PORT (clk,en : IN INTEGER; a,b : IN INTEGER; z : OUT INTEGER);  
END MUL;
```

```
ARCHITECTURE func OF MUL IS  
BEGIN  
  PROCESS (clk, en);  
    VARIABLE started : BOOLEAN := FALSE;  
    VARIABLE cnt : INTEGER : 0;  
  BEGIN  
    IF en'EVENT AND en = 1 THEN  
      started := TRUE;  
      cnt :=8;  
    END IF;  
    IF started THEN  
      IF cnt = 0 THEN  
        z <= a * b;  
        started := FALSE;  
      END IF ;  
      IF clk'EVENT THEN  
        cnt := cnt - 1;  
      END IF;  
    END IF;  
  END PROCESS;  
END func;
```

ARCHITECTURE struct OF det Is

```
COMPONENT BUF PORT (c, e, a : IN INTEGER; z : OUT INTEGER );
END COMPONENT;
COMPONENT ADD. PORT (c,e,a,b : IN INTEGER; z : OUT INTEGER );
END COMPONENT;
COMPONENT SUB PORT (c, e, a, b : IN INTEGER; z : OUT INTEGER );
END COMPONENT;
COMPONENT MUL PORT (c,e, a, b : IN INTEGER; z : OUT INTEGER );
END COMPONENT;
```

```
FOR MUL1 :      MUL  USE ENTITY Work.MUL (func);
FOR MUL2 :      MUL  USE ENTITY Work.MUL (func);
FOR MUL3 :      MUL  USE ENTITY Work.MUL (func);
FOR MUL4 :      MUL  USE ENTITY Work.MUL (func);
FOR MUL5 :      MUL  USE ENTITY Work.MUL (func);
FOR MUL6 :      MUL  USE ENTITY Work.MUL (func);
```

```
FOR SUB1 :      SUB  USE ENTITY Work.SUB(func);
FOR SUB2 :      SUB  USE ENTITY Work.SUB(func);
FOR SUB3 :      SUB  USE ENTITY Work.SUB(func);
```

```
FOR MUL7 :      MUL  USE ENTITY Work.MUL(func);
FOR MUL8 :      MUL  USE ENTITY Work.MUL(func);
FOR MUL9 :      MUL  USE ENTITY Work.MUL (func);
```

```
FOR ADD1 :      ADD  USE ENTITY Work. ADD(func);
FOR ADD2 :      ADD  USE ENTITY Work. ADD(func);
```

```
FOR OTHERS :    BUF  USE ENTITY Work.BUF(func);
```

```
SIGNAL S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12: INTEGER;
SIGNAL S13, S14, S15, S16, S17, S18, S19, S20, S21, S22 :INTEGER;
SIGNAL S23, S24, S25, S26, S27, S28 : INTEGER;
```

BEGIN

```
BUF0 :      BUF  PORT MAP (clk,BUF0_en,S1,S2);
BUF1 :      BUF  PORT MAP (clk,BUF0_en,S3,S4);
BUF2 :      BUF  PORT MAP (clk,BUF0_en,S5,S6);
BUF3 :      BUF  PORT MAP (clk,BUF0_en,S7,S8);
BUF4 :      BUF  PORT MAP (clk,BUF0_en,S9,S10);
BUF5 :      BUF  PORT MAP (clk,BUF0_en,S11,S12);
BUF6 :      BUF  PORT MAP (clk,BUF0_en,S13,S14);
BUF7 :      BUF  PORT MAP (clk,BUF0_en,S15,S16);
BUF8 :      BUF  PORT MAP (clk,BUF0_en,S17,S18);
BUF9 :      BUF  PORT MAP (clk,BUF0_en,S22,S23);
BUF10 :     BUF  PORT MAP (clk,BUF0_en,S24,S25);
BUF11 :     BUF  PORT MAP (clk,BUF0_en,S26,S27);
BUF12 :     BUF  PORT MAP (clk,BUF0_en,a1,S19);
BUF13 :     BUF  PORT MAP (clk,BUF0_en,a2,S20);
BUF14 :     BUF  PORT MAP (clk,BUF0_en,a3,S21);
```

```
MUL1 :      MUL  PORT MAP (clk,MUL1_en,b2,c3,S 1);
MUL2 :      MUL  PORT MAP (clk,MUL2_en,b3,c2,S3);
MUL3 :      MUL  PORT MAP (clk,MUL3_en,b3,c1,S5);
MUL4 :      MUL  PORT MAP (clk,MUL4_en,b1,c3,S7);
MUL5 :      MUL  PORT MAP (clk,MUL5_en,b1,c2,S9);
```

```
MUL6      :      MUL  PORT MAP (clk,MUL6_en,b2,cl,S11);

SUB1      :      SUB  PORT MAP (clk,SUB1_en,S2,S4,S13);
SUB2      :      SUB  PORT MAP (clk,SUB2_en,S6,S8,S15);
SUB3      :      SUB  PORT MAP (Clk,SUB3_en,S10,S12,S17);

MUL7      :      MUL  PORT MAP(clk,MUL7_en,S19,S14,S22);
MUL8      :      MUL  PORT MAP (clk,MUL8_en,S20,S16,S24);
MUL9      :      MUL  PORT MAP (clk,MUL9_en,S21,S18,S26);

ADD1      :      ADD  PORT MAP(Clk, ADD1_en, S23,S25,S28);
ADD2      :      ADD  PORT MAP (clk, ADD2_en, S2 8, S2 7, y);
END struct;
```

Differential Equation Solver

Design a unit which can count the solution of a $y''+3xy'+3y=0$ kind differential equation in the following way:

$$x1=x+dx \quad y1=y+dy \quad u1=u-3xudx-3ydx$$

The unit should work with pipelined restarting period.

To solve the problems only the following elementary operations can be used:

*Buffer, INC, DEC, +, -, *, DIV, MOD*

The execution timeratios are (in the same order) :

1 / 2 / 2 / 4 / 4 / 8 / 12 / 12 [clock cycle]

Tasks

The list of the students is located in the SEEGER server in the \UAGUEST\LOGIC\NAGYHF SUBDIRECTORY. In order to solve the following problems the first students in the list should use the *WinSam*, and the second ones should use the PIPE design tool. The subdirectories, where the design tools can be found, have the same name as the design tool has.

Solve the following problems

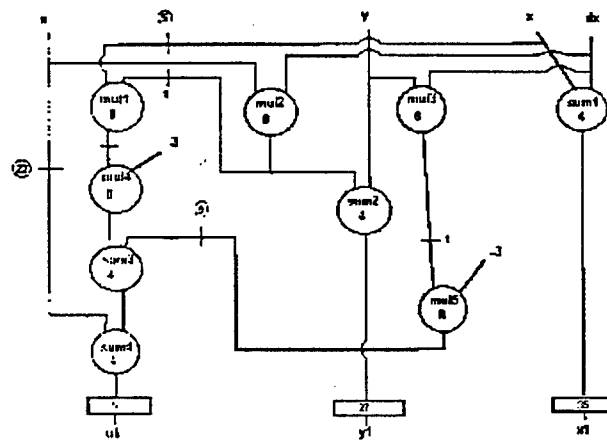
- generate the Data-Flow Graph (DFG) from the problem
- produce the behavioral description of the DFG in VHDL
- produce the description of the DFG in the hardware description language (HDL) of the given design tool (PIPE or WinSam)
- Run the design tool with the input file that was generated in point c) with different parameters such as:
 - WinSam: (*restart_s/restart_f*) 14/16, 14/18, 14/20, 16/16, 16/20;
 - PIPE: 14, 16, 18, 20, 22;
- compare the results, analyse the reasons of the output results and propose a moution of the best design.
- produce the structural description of the resulted structure and its controle in VHDL.
- simulate the structural VHDL description was produced in point f)

The format of the ideal solution

- The format of the report must be Win Word 2.0
- The coverage should contain:
 - title of the task
 - name and group of the student
- The format of the pictures must be PCX
- The final file (report) should be copied into the directory:
/U/GUEST/LOGIC/BEAD
(Once a file was created it cannot be deleted, but a second copy with different name might be accepted.)
- The name of the file must be: second_name.DOC
- The date of the handing in is the date when the file is created.

The deadline of the handing in is the last day of the semester !

1. The Data-Flow Graph



2. The VHDL Behavioral Description

ENTITY diff IS

PORT (x,y,u,dx : IN integer ; x1,y1,ul : OUT integer);

END diff;

ARCHITECTURE vis OF diff IS

SIGNAL m1,m2,m3,m4,m5,s : integer;

BEGIN

m2<=dx*u AFTER 8ns;

m1<=x*m2 AFTER 8ns;

m3<=y*dx AFTER 8ns;

x1<=x+dx AFTER 4ns;

y1 <=m2+y1 AFTER 4ns;

m4<=m1*(-3) AFTER 8ns;

m5<=m3*(-3) AFTER 8ns;

s<=m4+m5 AFTER 4ns;

ul<=s+u AFTER 4ns;

END vis;

3. The Input File of the WinSam

restart: 10
latency: 30
net_num: 1
cover_num: 1
sync_weight: 4
max_net: 1
step: 1
not_same: 1
add_time: 0
in, 0, 0, 0, 0, 6: mul1, mul2, mul3, sum1, sum2, sum4
mul1, 8, 8, 0, 2, 1: mul4
mul2, 8, 8, 0, 2, 2: sum2, mul1
sum1, 4, 4, 0, 2, 1: out
sum2, 4, 4, 0, 2, 1: out
mul3, 8, 8, 0, 2, 1: mul5
mul4, 8, 8, 0, 2, 1: sum3
mul5, 8, 8, 0, 2, 1: sum3
sum3, 4, 4, 0, 2, 1: sum4
sum4, 4, 4, 0, 2, 1: out
out, 0, 0, 0, 3, 0:

4. WinSam Results

restart_s	restart_f	processor sz.	pufferszam
10	10	9	11
10	20	6	7
10	30	6	6
10	40	5	4
10	50	5	4
15	40	2	7

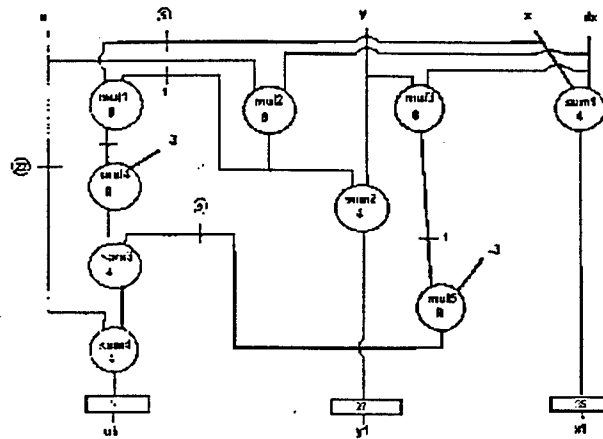
5. The Input File of the PIPE

```
#diff egyenlet
graph:diff
input:u,y,x,dx
output: u1,y1,x1
processor mul 8 2
processor sum 4 2
m2 mul (u,dx)
m1 mul (x,m2)
m3 mul (y,dx)
m4 mul (m1,m1)
m5 mul (m3,m3)
s sum (m4,m5)
temp1 sum (Li,s)
temp2 sum (m2,y)
temp3 sum (x,dx)
u1 temp1
y1 temp2
x1 temp3
```

6 **PIPE Results**

R	processors	+ buffers	Cost
5	18	135	247
6	14	129	225
9	13	92	184
10	9	51	107
12	8	43	95
13	7	41	89
15	8	35	87
20	8	14	66
25	7	8	56
30	6	3	47
35	6	0	40

7. The Scheduled Data-Flow Graph



8. Structural VHDL Description:

ENTITY diff IS

PORT (x,y,u,dx: IN integer ; xl,yl,ul : OUT Integer);

END diff;

ARCHITECTURE func OF diff IS

SIGNAL m1,m2,m3,m4,m5,s,p1,p2,p3,sh1,sh2,sh3 : Integer;

PACKAGE Global IS

SIGNAL CLK : Integer:=0;

SIGNAL buf1_en :Integer:=0;

SIGNAL buf2_en :Integer:=0;

SIGNAL buf3_en :Integer:=0;

SIGNAL sh1_en :Integer:=0;

SIGNAL sh2_en :Integer:=0;

SIGNAL sh3_en :Integer:=0;

SIGNAL mul1_en :Integer:=0;

SIGNAL mul2_en :Integer:=0;

SIGNAL mul3_en :Integer:=0;

SIGNAL mul4_en :Integer:=0;

SIGNAL mul5_en :Integer:=0;

SIGNAL sum1_en :Integer:=0;

SIGNAL sum2_en :Integer:=0;

SIGNAL sum3_en :Integer:=0;

SIGNAL sum4_en :Integer:=0;

CONSTANT q :Integer:= -3;

CONSTANT period : Integer :=13;

CONSTANT buf1_offs : Integer :=8;

CONSTANT buf2_offs : Integer :=17;

CONSTANT buf3_offs : Integer :=8;

CONSTANT sh1_offs : Integer :=0;

CONSTANT sh2_offs : Integer :=0;

CONSTANT sh3_offs : Integer :=17;

CONSTANT mul1_offs : Integer :=9;

CONSTANT mul2_offs : Integer :=0;

CONSTANT mul3_offs : Integer :=0;

CONSTANT mul4_offs : Integer :=18;

CONSTANT mul5_offs : Integer :=9;

CONSTANT sum1_offs : Integer :=0;

CONSTANT sum2_offs : Integer :=8;

CONSTANT sum3_offs : Integer :=26;

CONSTANT sum4_offs : Integer :=30;

FUNCTION start(offset, act : Integer) RETURN Boolean;
END Global;

PACKAGE BODY Global IS

```
FUNCTION start( offset, act : Integer) RETURN Boolean IS
BEGIN
  IF act< offset THEN RETURN FALSE;
  ELSIF (act-offset) MOD period =0 THEN RETURN TRUE;
  ELSE RETURN FALSE;
  END IF;
END start;
END Global;
```

USE Global.ALL;

ENTITY scheduler IS
END;

ARCHITECTURE sched OF scheduler IS

BEGIN

sched_proc:

PROCESS

VARIABLE act : Integer :=0;

BEGIN

```
  IF start(buf1_ofs,act) THEN buf1_en<=1;ELSE buf1_en<=0; END IF;
  IF start(buf2_ofs,act) THEN buf2_en<=1;ELSE buf2_en<=0; END IF;
  IF start(buf3_ofs,act) THEN buf3_en<=1;ELSE buf3_en<=0; END IF;
  IF start(sh1_ofs,act) THEN sh1_en<=1;ELSE sh1_en<=0; END IF;
  IF start(sh2_ofs,act) THEN sh2_en<=1;ELSE sh2_en<=0; END IF;
  IF start(sh3_ofs,act) THEN sh3_en<=1;ELSE sh3_en<=0; END IF;
  IF start(mul1_ofs,act) THEN mul1_en<=1;ELSE mul1_en<=0; END IF;
  IF start(mul2_ofs,act) THEN mul2_en<=1;ELSE mul2_en<=0; END IF;
  IF start(mul3_ofs,act) THEN mul3_en<=1;ELSE mul3_en<=0; END IF;
  IF start(mul4_ofs,act) THEN mul4_en<=1;ELSE mul4_en<=0; END IF;
  IF start(mul5_ofs,act) THEN mul5_en<=1;ELSE mul5_en<=0; END IF;
  IF start(sum1_ofs,act) THEN sum1_en<=1;ELSE sum1_en<=0; END IF;
  IF start(sum2_ofs,act) THEN sum2_en<=1;ELSE sum2_en<=0; END IF;
  IF start(sum3_ofs,act) THEN sum3_en<=1;ELSE sum3_en<=0; END IF;
  IF start(sum4_ofs,act) THEN sum4_en<=1;ELSE sum4_en<=0; END IF;
  act :=act+1;
```

CLK <= act;

IF act=200 THEN WAIT;

END PROCESS;

END sched;

ENTITY buffIS

GENERIC (t : Integer :=1);

```

ENTITY buff IS
  GENERIC ( t: Integer :=1);
  PORT (clk, en : IN INTEGER ; z : OUT INTEGER);
END buff;

```

```

ARCHITECTURE func OF buff IS
BEGIN
  PROCESS (clk,en)
    VARIABLE started : Boolean :=FALSE;
    VARIABLE cnt : Integer :=0;
  BEGIN
    IF en'EVENT AND en=1 THEN- _enable
      started :=TRUE;
      cnt :=t;
    END IF;
    IF started THEN
      IF cnt=0 THEN
        z<=a;
        started:=FALSE;
      END IF ;
    END IF;
  END PROCESS;
END func;

```

```

ENTITY mul IS
  GENERIC (t:Integer :=12);
  PORT (clk, en : IN Integer ; z : OUT Integer);
END buff;

```

```

ARCHITECTURE func OF mul IS
BEGIN
  PROCESS (clk, en)
    VARIABLE started : Boolean :=FALSE;
    VARIABLE cnt : Integer :=0;
  BEGIN
    IF en'EVENT AND en=1 THEN
      started :=TRUE;
      cnt :=t;
    END IF;
    IF started THEN
      IF cnt :=0 THEN
        z<=a*b;
        started:=FALSE;
      END IF;
    IF clk'EVENT THEN

```

```

        cnt := cnt-1;
    END IF;
END IF;
END PROCESS;
END func;

```

```

ENTITY add IS
    GENERIC(t:Integer:=4);
    PORT (clk, en: IN Integer; a : IN Integer; z : OUT Integer);
End buff;

```

```

ARCHITECTURE func OF add IS
BEGIN
    PROCESS (clk, en)
        VARIABLE started : Boolean :=FALSE;
        VARIABLE cnt : Integer:=0;
    BEGIN
        IF en'EVENT AND en=1 THEN
            started := TRUE;
            cnt :=t;
        END IF;
        IF started THEN
            IF cnt :=0 THEN
                z<=a+a;
                started:=FALSE;
            END IF;
            IF clk'EVENT THEN
                cnt := cnt-1;
            END IF;
        END IF;
    END PROCESS;
END func;

```

```

ARCHITECTURE struct OF sample IS
    COMPONENT buf PORT ( c, e, a: IN Integer, z: OUT Integer ); END
    COMPONENT;
    COMPONENT mul PORT ( c, e, a: IN Integer, z: OUT Integer ); END
    COMPONENT;
    COMPONENT add PORT ( c, e, a: IN Integer, z: OUT Integer ); END
    COMPONENT;

```

```

FOR mul12 : mul USE ENTITY Work mul(func);

```

```
FOR add4 : add USE ENTITY Work add(func);
FOR OTHERS : buf USE ENTITY Work buf(func);
```

```
SIGNAL J01,J02,J03,J04,J05,J06,J07,J08,J09,J10,J11,J12,J13,J14,J15,
      J16,J17,J18,J19,J20,J21,J22,J23,J24
      M01,M02,M03,M04,M05,M06,M07,M08 :Integer;
```

```
CONSTANT S9,S10,S11,S12,
      W1,W2,W3,W4,W5,W6,W7,W8 : Integer;
```

```
BEGIN
```

```
add01 :add4 PORT MAP (CLK,add01_en, s1,s2,J01);
add02 :add4 PORT MAP (CLK,add02_en, s5,s6,J02);
add03 :add4 PORT MAP (CLK,add03_en, J01,s3,J03);
add04 :add4 PORT MAP (CLK,add04_en, J02,s7,J04);
add05 :add4 PORT MAP (CLK,add05_en, J03,s4,J05);
add06 :add4 PORT MAP (CLK,add06_en, J04,s8,J06);
add07 :add4 PORT MAP (CLK,add07_en, J05,J06,J07);
add08 :add4 PORT MAP (CLK,add08_en, B3,M01,J08);
add09 :add4 PORT MAP (CLK,add09_en, B4,J08,J09);
add10 :add4 PORT MAP (CLK,add10_en, M02,J06,J10);
add11 :add4 PORT MAP (CLK,add11_en, J08,J07,J11);
add12 :add4 PORT MAP (CLK,add12_en, J10,B5,J12);
add13 :add4 PORT MAP (CLK,add13_en, J11,J10,J13);
add14 :add4 PORT MAP (CLK,add14_en, B0,M03,J14);
add15 :add4 PORT MAP (CLK,add15_en, B1,J14,J15);
add16 :add4 PORT MAP (CLK,add16_en, J14,M03,J16);
add17 :add4 PORT MAP (CLK,add17_en, J16,S11,J17);
add18 :add4 PORT MAP (CLK,add18_en, S12,M04,J18);
add19 :add4 PORT MAP (CLK,add19_en, M05,s13,J19);
add20 :add4 PORT MAP (CLK,add20_en, J18,B8,J20);
add21 :add4 PORT MAP (CLK,add21_en, J17,J19,J21);
add22 :add4 PORT MAP (CLK,add22_en, M06,s9,J22);
add23 :add4 PORT MAP (CLK,add23_en, B6,J22,J23);
add24 :add4 PORT MAP (CLK,add24_en, J22,s10,J24);
add25 :add4 PORT MAP (CLK,add25_en, B2,J23,J25);
add26 :add4 PORT MAP (CLK,add26_en, J22,M07,J26);
```

```
mul1 :mul12 PORT MAP (CLK,mul1_en, J07,W3,M01);  
mul2 :mul12 PORT MAP (CLK,mul2_en, J07,W1,M02);  
mul3 :mul12 PORT MAP (CLK,mul3_en, J09,W8,M03);  
mul4 :mul12 PORT MAP (CLK,mul4_en, J15,W7,M04);  
mul5 :mul12 PORT MAP (CLK,mul5_en, J17,W6,M05);  
mul6 :mul12 PORT MAP (CLK,mul6_en, J12,W2,M06);  
mul7 :mul12 PORT MAP (CLK,mul7_en, J24,W4,M07);  
mul8 :mul12 PORT MAP (CLK,mul8_en, J25,W5,M08);
```

```
buf0 :buf1 PORT MAP (CLK,buf0_en, J01,B0);  
buf1 :buf1 PORT MAP (CLK,buf1_en, J01,B1);  
buf2 :buf1 PORT MAP (CLK,buf2_en, J02,B2);  
buf3 :buf1 PORT MAP (CLK,buf3_en, J03,B3);  
buf4 :buf1 PORT MAP (CLK,buf4_en, J03,B4);  
buf5 :buf1 PORT MAP (CLK,buf5_en, J06,B5);  
buf6 :buf1 PORT MAP (CLK,buf6_en, J10,B6);  
buf7 :buf1 PORT MAP (CLK,buf7_en, J13,B7);  
buf8 :buf1 PORT MAP (CLK,buf8_en, J14,B8);
```

```
END stuct;
```

```
END IF;  
END PROCESS;  
END func;
```

```
ENTITY sum IS  
  GENERIC (t:Integer :=4);  
  PORT (clk,en :IN Integer;a,b : IN Integer; z : OUT Integer);  
END sum;
```

```
ARCHITECTURE func OF sum IS  
BEGIN  
  PROCESS (clk,en)  
    VARIABLE started :Boolean := FALSE;  
    VARIABLE cnt : Integer :=0;  
  BEGIN  
    IF en'EVENT AND en=1 THEN started := TRUE;  
      cnt :=t;  
    END IF;  
    IF started THEN  
      IF cnt :=0 THEN  
        z<=a+b;  
        started := FALSE;  
      END IF;  
      IF clk'EVENT THEN  
        cnt := cnt-1;  
      END IF;  
    END IF;  
  END PROCESS;  
END func;
```

```
ENTITY shift1 IS  
  GENERIC (t:Integer :=5);  
  PORT (clk,en : IN Integer; a : IN Integer ; z : OUT Integer);  
END shift1;
```

```
ARCHITECTURE func OF shift1 IS  
BEGIN  
  PROCESS ( clk,en)  
    VARIABLE started : Boolean := FALSE;  
    VARIABLE cnt : Integer :=0;  
  BEGIN  
    IF en'EVENT AND en=1 THEN  
      started := TRUE;  
      cnt := t;  
    END IF;
```

```

    IF started THEN
        IF cnt :=0 THEN
            z<=a;
            started := FALSE;
        END IF;
        IF clk'EVENT THEN
            cnt := cnt-1;
        END IF;
    END IF;
END PROCESS;
END func;

```

```

ENTITY shift2 IS
    GENERIC ( t: Integer :=22);
    PORT ( clk,en : IN Integer; a : IN Integer; z : OUT Integer);
END shift2;

```

```

ARCHITECTURE func OF shift2 IS
BEGIN
    PROCESS ( clk,en)
        VARIABLE started : Boolean := FALSE;
        VARIABLE cnt : Integer:=0;
    BEGIN
        IF en'EVENT AND en=1 THEN
            started := TRUE;
            cnt :=t;
        END IF;
        IF started then
            IF cnt :=0 THEN
                z<=a;
                started := FALSE;
            END IF;
            IF clk'EVENT THEN
                cnt := cnt-1;
            END IF;
        END IF;
    END PROCESS;
END func;

```

```

ENTITY shift3 IS
    GENERIC ( t: Integer :=9);
    PORT (clk,en : IN Integer; a : IN Integer; z : OUT Integer);
END shift3;

```

```

ARCHITECTURE func OF shift3 IS
BEGIN

```

```
    PORT (clk, en: IN Integer; a : IN Integer; z :OUT Integer);  
END buff;
```

```
ARCHITECTURE func OF buff IS
```

```
BEGIN
```

```
    PROCESS(clk,en)
```

```
        VARIABLE started : BOOLEAN :=FALSE;
```

```
        VARIABLE cnt :Integer:=0;
```

```
    BEGIN
```

```
        IF en'EVENT AND en=1 THEN started :=TRUE;
```

```
            cnt :=t;
```

```
        END IF;
```

```
        IF started THEN
```

```
            IF cnt =0 THEN
```

```
                z <=a;
```

```
            started :=FALSE;
```

```
            END IF;
```

```
            IF clk'EVENT THEN
```

```
                cnt :=cnt-1;
```

```
            END IF;
```

```
        END IF;
```

```
    END PROCESS;
```

```
END func;
```

```
ENTITY mul IS
```

```
    GENERIC (t:Integer :=8);
```

```
    PORT (clk,en : IN Integer;a,b : IN Integer ; z : OUT Integer);
```

```
END mul;
```

```
ARCHITECTURE func OF mul IS
```

```
BEGIN
```

```
    PROCESS (clk,en)
```

```
        VARIABLE started : Boolean := FALSE;
```

```
        VARIABLE cnt : Integer :=0;
```

```
    BEGIN
```

```
        IF en'EVENT AND en=1 THEN
```

```
            started := TRUE;
```

```
            cnt :=t;
```

```
        END IF;
```

```
        IF started THEN IF cnt :=0 THEN
```

```
            z<=a*b;
```

```
            started :=FALSE;
```

```
        END IF;
```

```
        IF clk'EVENT THEN
```

```
            cnt := cnt-1;
```

```
        END IF;
```

```

PROCESS ( clk,en)
  VARIABLE started : Boolean := FALSE;
  VARIABLE cnt : Integer:=0;
BEGIN
  IF en'EVENT AND en=1 THEN
    started := TRUE;
    cnt :=t;
  END IF;
  IF started then
    IF cnt :=0 THEN
      z<=a;
      started := FALSE;
    END IF;
    IF clk'EVENT THEN
      cnt := cnt-1;
    END IF;
  END IF;
END PROCESS;
END func;

```

ARCHITECTURE struktura OF diff IS

```

  COMPONENT buff PORT (c,e,a: IN Integer; z: out Integer ); END COMPONENT;
  COMPONENT mul PORT (c,e,a: IN Integer; z: out Integer ); END COMPONENT;
  COMPONENT sum PORT (c,e,a: IN Integer; z: out Integer ); END COMPONENT;
  COMPONENT shift1 PORT (c,e,a: IN Integer; z: out Integer ); END
COMPONENT;
  COMPONENT shift2 PORT (c,e,a: IN Integer; z: out Integer ); END
COMPONENT;
  COMPONENT shift3 PORT (c,e,a: IN Integer; z: out Integer ); END
COMPONENT;

```

```

FOR mul1 : mul USE ENTITY Work.mul(func);
FOR mul2 : mul USE ENTITY Work.mul(func);
FOR mul3 : mul USE ENTITY Work.mul(func);
FOR mul4 : mul USE ENTITY Work.mul(func);
FOR mul5 : mul USE ENTITY Work.mul(func);
FOR sum1 : sum USE ENTITY Work.sum(func);
FOR sum2 : sum USE ENTITY Work.sum(func);
FOR sum3 : sum USE ENTITY Work.sum(func);
FOR sum4 : sum USE ENTITY Work.sum(func);
FOR sh1 : shift1 USE ENTITY Work.shift1(func);
FOR sh2 : shift2 USE ENTITY Work.shift2(func);
FOR sh3 : shift3 USE ENTITY Work.shift3(func);
FOR OTHERS : buff USE ENTITY Work.buff(func);

```

```

SIGNAL m1,m2,m3,m4,m5,s,p1,p2,p3,sh1,sh2,sh3 : Integer;

```

BEGIN

```
buf1 : buf PORT MAP(CLK,buf1_en,m2);  
buf2 : buf PORT MAP(CLK,buf2_en,m1);  
buf3 : buf PORT MAP(CLK,buf3_en,m3);  
mul1 : mul PORT MAP(CLK,mul1_en,x,p1);  
mul2 : mul PORT MAP(CLK,mul2_en,u,dx);  
mul3 : mul PORT MAP(CLK,mul3_en,y,dx);  
mul4 : mul PORT MAP(CLK,mul4_en,p2,q);  
mul5 : mul PORT MAP(CLK,mul5_en,p3,q);  
sum1 : sum PORT MAP(CLK,sum1_en,x,dx);  
sum2 : sum PORT MAP(CLK,sum2_en,m2,y);  
sum3 : sum PORT MAP(CLK,sum3_en,m4,sh3);  
sum4 : sum PORT MAP(CLK,sum4_en,u,s);  
sh1 : shift1 PORT MAP(CLK,sh1_en,s5,x);  
sh2 : shift2 PORT MAP(CLK,sh2_en,s5,u);  
sh3 : shift3 PORT MAP(CLK,sh3_en,s5,m5);
```

END struktura;

Conclusions

Comparing the results of this two high-level synthesis tool (PIPE, WinSam) we found that WinSam can reduce the number of the needed synchronisation buffers in a very efficient way. In those cases where the input structure (represented by an EOG) is symmetrical, the PIPE and the WinSam found nearly the same solution. In the structures where the EOG is less symmetrical, that means more mobility, the PIPE could find the better solution. The reason of this that WinSam follows a definite philosophy to fix the mobilities, while pipe generates all the possible situations and the decision about the best structure is made after all design steps has been executed. The 'price' of the more beneficial result is the sometimes 2...5 times slower processing time. We are still working to apply the method which can reduce the needed synchronisation buffers into the PIPE.

The benchmarks and the solutions were published in this report are the results of a continues work. Each group had three consultation times where they had to show their supervisor the temporary results. Sometimes we found that the students in one group couldn't work together which resulted different EOGs (It is impossible to compare the methods when the starting points, the input structure is different.). It proved the fact that for the solution of a HLS tool is a function of the structure was build up from the original task.

References

FIR filter:

Unified System Construction (USC)
Alice C. Parker, Kayhan Kucukcakar, Shiv Prakash, Jen-Pin Weng
PISYN-High-Level Synthesis of Application Specific Pipelined Hardware
Albert E. Casavant, Ki Soo Hwang, Kristen N. McNall

Diferential Equation Solver:

Force-Directed Scheduling for the Behavioral Synthesis of ASIC's
Pierre G. Paulin, John P. Knight
Global Scheduling and Allocation Algorithms in the HAL System
Pierre G. Paulin
Scheduling and Assgnment in High-Level Synthesis
Wolfgang Rosenstiel, Heinrich Kramer

Elliptic Filter:

High-Level Synthesis in the THEDA System
Yu-Chin Hsu, Youn-Long Lin

Experiences and Statistics of the Curriculum

edited by
Tamás Visegrády

Technical University of Budapest
Faculty of Electrical Engineering and Informatics
Department of Process Control

Experiences with the Curriculum

After successfully completing two semesters of High-level synthesis, the first results were statistically analyzed. The results are used as feedback to improve the properties of the education.

Student background

Integrated into the studies of Computer Systems and Microelectronics branches, the majority of the students on the course (more than 80 %) had a strong practical background in digital computing systems. The material presented during the lessons was based partly on this knowledge base. All the students were seniors (3rd grade or over). Should the course be presented to junior students, a more detailed description of the fundamentals would be required to guarantee proper understanding. This should be avoided as the increase in the necessary number of hours would be significant.

As most of the students started the course with a strictly practical background, the organization of the classes was a mixture of theoretical and practical sections. It would be possible to synchronize the progress in theory with parallel practical subjects, which would result in an improved communication between different fields and also remove some of the practical lessons to enable deeper studies in some directions.

Student effort

As well as attending the lectures, the students were required to complete two design problems. The first task was an introduction to the design systems used in high-level synthesis and concentrated on the technical problems. The task was taken from a section of high-level synthesis (scheduling), where the students had to compare various algorithms and write a report on the results and the conditions. The estimated net work time was between 5 and 10 hours (with some previous calculations), after which the students were able to independently prepare input for a high-level synthesis design system, monitor the progress of the design process and evaluate the results.

The communication channels used during the evaluation of the results were strictly non-verbal (i.e. e-mail). The theoretical background was presented during the lectures. The technical skills used were introduced in a session of practical lessons.

All the problems were given to the students on an individual basis. The problems were based on real design processes, with some generalizations to ensure diversity. As the problems were generated using a randomizer, students could cooperate in teams in the general solution, with additional effort to substitute their personal problems to the general solution individually.

The total time span of the the design process was four weeks. All the students were all able to finish the process by the deadline, with the first results returned in 2 (two) days. The results were officially accepted after testing, with personal notes posted to the students.

By carefully choosing the problems and prescribing the result format, the correct answers could be generated using the design systems. As a result, unflawed answers were recognized without human intervention, while incorrect results had to be examined using human resources. This enabled the reviewers to concentrate on the typical problems and to find critical points in the courses. This feedback was used during the composition of the coursebooks.

The theoretical and practical lectures to be used during the first design process were presented by T. Visegrady.

The second design process was performed in the last third of the semester. This task was a complete design, enveloping the initial, intermediate and (part of) final sections of high-level synthesis (graph generation, scheduling and allocation). The total time required was between 30 and 50 net hours of work for every student. The students were given matched problems in groups, so they could work in teams. Every group had subtle differences in the individual problems, so they could cooperate in general and refine the results on an individual basis.

Necessary theoretical material had been presented at the distribution of the problems. As the work involved significant individual research, the first results required more than two weeks to turn up. The last results were submitted just before the final deadline, so the estimated time frame was suitable for the task. The second design task was based on the

successful completion of the first design process, so it required no separate introduction to the technics of design, except for the subtle differences between the design systems.

The theoretical background was presented and the consulting lectures were led by I. Jankovics and I. Beres.

Any student successfully submitting both design problems has an overall knowledge of automatic tools employed during high-level synthesis, had working experience with a high-level scheduler (with significant knowledge on the internals of it) and is able to use the PIPE or WINSAM design systems for a complete design.

**Developing a VLSI Module-generator
Interfaced to the PIPE as a Part of a
Collaborative Engineering Curriculum**

Edited by Zoltán Sugár

**Department of Process Control
Technical University of Budapest**

Developing a VLSI Module-generator Interfaced to the PIPE as a Part of a Collaborative Engineering Curriculum

Zoltán Sugár

Abstract

*This report presents a model for designing pipelined data-flow structures with primary focus on module and control path generation part of the design flow. Control-data path interface has been developed for different requirements. A CAD program associated with the model generates a synthesizable description about the data path elements and the control part from a high-level data-flow description providing a useful tool to the developer with hiding quite a large amount of implementation difficulties. The developed CAD tool has been integrated into a framework environment with a standard and easy-to-use interface that may improve the **collaboration** among research and engineering teams.*

Introduction

Since the complexity of digital systems has been increased to a very high level the electronics designers are less and less concerned for circuit layouts or circuit diagrams. It is especially true for the high speed data processing elements, in which case there is significant importance of the parallel and regular structures, like systolic and wavefront arrays and pipelined circuits. New synthesis algorithms are raising the level of abstraction to reduce the engineering costs, and development time and to improve the reliability of the design process. High-level logic synthesis establishes a connection between this abstract level and the register transfer or gate level representation of the circuit. Thus the high level synthesis is built on the widely used RTL synthesis. Since no commercial CAD software is available for this kind of design methods, there is of no significance in the industry.

The design method presented in this report introduces a way for tuning the data-flow graph to achieve the best cost per performance value. Scheduling and allocation algorithms can handle the typical pipelined structures. The module generator has been developed can realise the basic operation and the control path automatically. A CAD software has been also developed which is

integrated to a commercial framework environment. It provides an easy-to-use, user friendly tool that can extend the existing ASIC design flow to a higher level of abstraction.

Nowadays there is an importance of the **collaborating engineering**. It means that the given design is accomplished by several and mostly independent engineering teams. On the other side, the **collaboration** may be between among different research groups, often at the far points of the world with dividing the complex research activities into smaller parts. As an example, a strong **cooperation** has been built between the University of New Hampshire and the Technical University of Budapest. The backbone of this **cooperation** is a student exchange program. For the sake of successful **cooperation**, a powerful communication method is required based on the well-known international standards. The developed CAD tool may satisfy this requirement. The input and the output of this tool are conform with the ISO VHDL standard. The input parser may accept a well-defined subset of the VHDL.

The report is organised into four sections. The first section introduces the basic scheduling and allocation algorithms. The second section discusses the module generation procedures. The third section presents two different ways for controlling the data path. The last section discusses the CAD tool associated with the algorithms.

1. Scheduling and Allocation Procedures

High-level synthesis starts with the behavioral description of the circuit. Synthesis procedures can usually accept a data flow graph as input. This graph can be derived from a HDL description, for example from a VHDL or a Verilog behavioral specification. The data flow graph is a directed graph in which the nodes represent the functional elements and the edges establish the logical connections between them. It is assumed that functional elements can have several inputs but only one output. In reality this restriction does not cause any limitation, because the physical outputs can be collected into one logical connection. Two basic properties may be assigned to the functional elements, the number of inputs and the delay time. This delay time determines the number of control steps required for finishing the operation. As described later, these properties may be extended by additional information to improve the scheduling and allocation algorithms.

The system is considered to be pipelined if the scheduled data flow graph is able to receive new data before finishing the operation of the last functional element. Obviously, throughput is maximised if the frequency of the data changes is maximised. The shortest permissible interval of the data changes on the inputs is called minimal restarting period. Especially, in the case of hard real-time applications, it is required to reduce the latency time, the time between applying the data for the inputs of the circuit and arriving the corresponding result on the outputs. Unfortunately, it is not possible to increase the throughput of a system by decreasing the restarting period and simultaneously reducing the latency time, because these parameters correlate with each other.

The most important scheduling and allocation algorithms - list scheduling, modified ASAP/ALAP, ILP, force-directed, etc.) - are published. These methods have been examined and are applicable for pipelined structures.

2. Module Generation

Main purpose of the module generation is to map the scheduled, allocated and controlled data-flow graph to RT or gate-level representation based on the given constraints. In most cases this description is represented by hardware description languages and it may be direct input of widely used RTL synthesisers and logic optimisers. The advantage of the RT level representation is the cell library independence. The module generator has also significance at the beginning of the high-level design methodology, in the pre-allocation step.

Since only the module generator has knowledge about the internal structure and the behaviour of the functional elements, thus only it can determine the cost functions and the delays. It should be noted that the module generator does not decide the circuit implementation in the pre-allocation phase, just offers a choice list with the corresponding properties and just later, during the optimisation, scheduling and allocation stage makes the optimal and less expensive decision.

Consider for example the following simple problem. Somewhere in the data-flow graph a multiplication operation must be implemented. The module generator has two types of modules for multiplication, a traditional shift-and-add and a Radix-4 multiplier, with 9 and 3 control steps of delay and 90 and 170 relative cost respectively. Assuming that the required restarting period is equal to 4 and the traditional multiplier does not support functional pipelining,

obviously it must be multiplied by three times. Hence the relative cost is over 270, the scheduler chooses the Radix-4 multiplier. This example demonstrates a simple case, in reality the mechanism of making decision is much more complicated.

In the pre-allocation phase the module generator provides additional properties for the functional elements to improve the efficiency of the scheduling and allocation procedures. A functional element cannot be restarted under its and the next immediate successor's operations and the functional element must hold its output under operation of the next successor. This means that the result of functional element must be latched into a register in the last clock cycle of its operation. Consequently, if the functional element is implemented by combinatorial logic or sequential logic but the last register does not play role in forming the result, the operation may be restarted immediately again without violating communication rules between the functional elements. Obviously, these registers may be placed at the inputs of the functional element to latch the input data in the first cycle of the operation. In this case the previous functional element in the transfer sequence may be restarted earlier. From the point of view of scheduling these registers can be treated as pipeline buffers. The scheduler has a freedom to place these to the input or to the output of the functional element to get an optimal result. In most cases less optimisation may be applied if the operation is implemented by sequential logic. For example, generally the input and output registers cannot be exchanged, but if the functional element stores the input data, or the output registers do not play act in forming the result, the considerations discussed above may be taken into account.

If the delay time of a functional element exceeds the required restarting period, the operation must be multiplied. This means that the physical copies of the functional elements must be connected in parallel and multiplexers must be inserted into the data-path with special control elements to switch the data to the appropriate functional element. This method increases the total cost more than

$$(c - 1) \cdot C_{FE} \text{ times, where} \quad (8)$$

c denotes the required number of copies,

C_{FE} denotes the relative cost of the given functional element.

thus it should be avoided, if possible. If the operation may be divided into smaller suboperations, that do not form a loop, if the delay of suboperations are

smaller than $T_{required}-1$ and pipeline registers are inserted between these suboperations, the functional element may be restarted in every

$$\max(t_{di1}, t_{di2}, \dots, t_{din}) + 1 \text{ control step.} \quad (9)$$

t_{din} denotes the delay of n . suboperation of i . functional element.

This structure is called functional pipelined operation and in some cases it can substitute the direct multiplication.

A typical example, the internal structure of a floating point adder is shown in Fig. 1. Since the cost of this operation is extremely high, the multiplication

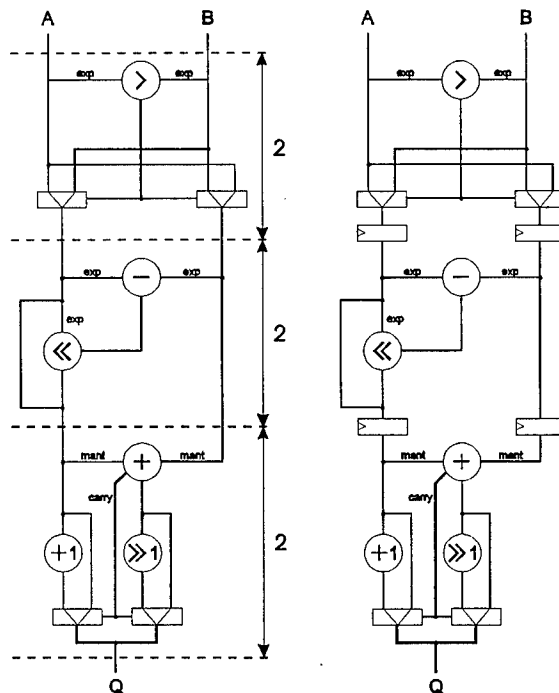


Fig. 1. Functional pipelining

as a scheduling method is not applicable. The adder may be split into smaller operation. The numbers of pipelined stages depend on the required restarting period. In this example three set of pipeline register must be inserted.

This feature of functional elements provides more freedom for the allocation procedures. Obviously, if an operation is functional pipelined and it may be restarted in every T_{rf} interval, where T_{rf} is less or equal to half of the restarting period of data-flow graph,

In the pre-allocation phase it must be decided which operation may be combined with each other later by the allocator. For example, it is easy to implement a floating point adder and a subtractor in one processor, or two multipliers with different widths.

3. Control Path

The scheduled and allocated data path cannot function properly without external components. It must be extended with the control path which can provide control signals for starting the operations of the functional elements, for writing the data into the buffers and for the multiplexers to connect the appropriate data path before and after the combined and multiplied operations.

The control path ensures that functional elements start their operations in proper time and then all data are stable on the inputs.

Two models were developed for controlling the data path that require two different interfaces from the functional elements. These two methods are the centralised and distributed control path. The main benefit of these models is that they may be generated automatically based on the timing information provided by the scheduled and allocated data-flow graph.

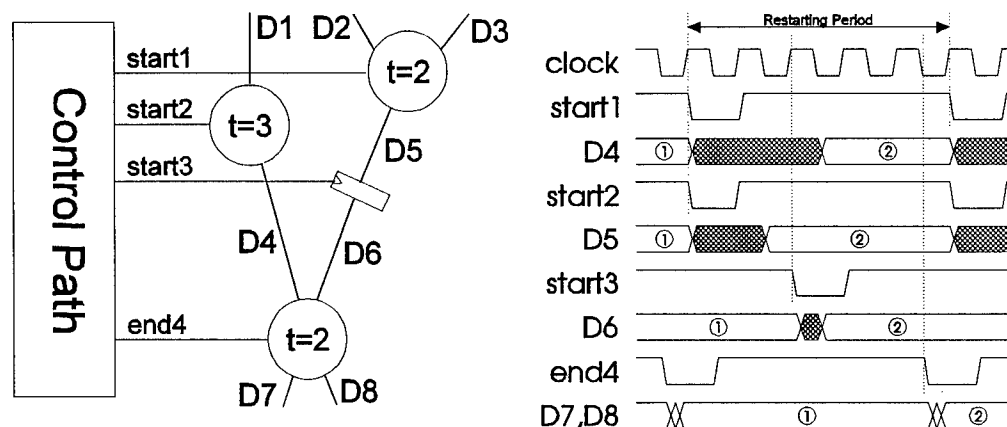


Fig. 2. Centralised controlling scheme

The centralised control path may be completely separated from the data path, as shown in Fig. 2. with the corresponding timing diagram. In this model every functional element has a start signal. If this signal is activated the functional element starts its operation. This signal is synchronised with the raising edge of the system clock signal. Instead of the start signal, the functional elements can have an end signal that indicates the end of the operation. If the functional element is implemented by a combinatorial logic these signals may be used for writing the input data or the result into the buffer. According to the input-output specification of the functional elements, the end signal must be delayed with a half clock cycle. The main advantage of the centralised controlling scheme is the simplicity of the implementation. In most cases the realisation is a Johnson counter. Outputs of this counter may be directly connected to the corresponding start signals of the functional elements. Since this kind of control path is relatively simple and the model matches with the internal structure of most of the functional elements, the extra silicon area occupation is small compared to data path operations. The centralised controlling scheme has a disadvantage. If the data-flow graph contains a large number of functional elements then the generated VLSI die will be huge and the interconnection delays of the controlling signals cannot be neglected. This

delay decreases the highest operation frequency. It is especially true if the target technology is Multi Chip Module and the design does not fit onto one die, since the I/O pads cause additional delay between the functional elements.

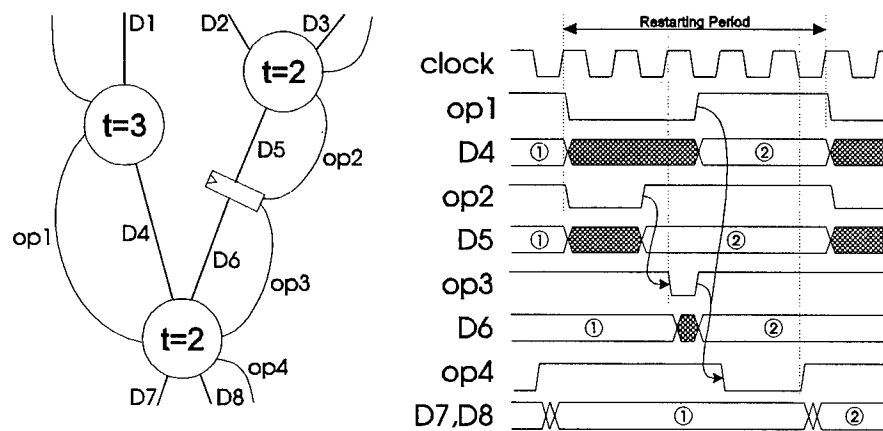


Fig. 3. Distributed controlling scheme

To avoid this problem, the distributed controlling scheme should be used. In the literature for this model, control elements are defined that are in handshake relationship with the functional elements. This communication is time consuming and makes slower the operations. In this model every functional element contains this control element to eliminate this unnecessary communication. The functional element activates a signal under its operation, as shown in Fig. 3. This signal is deactivated in the last clock cycle of the operation and this rising edge and the next rising edge of the system clock signal starts the next functional element in the transfer sequence. This controlling scheme can handle the problem mentioned above, since the interconnection delays of the control signals may be compared to the delays of the data path. The other benefit of distributed control path is the system does not require a strict restarting interval. It means that if the data-flow graph does not contain recursive loops it can be restarted with equal or longer interval than the actual restarting period. The only disadvantage associated with this model is the large number of extra logic required to implement that.

As a conclusion, if the data-flow graph is small and the operation frequency is not high the centralised control path gives a small and efficient solution. In the case of a high performance and complicated design, however, only the distributed controlling scheme can guarantee a reliable operation.

4. Synthesis Software Products

State-of-the-art CAD systems should provide more and more aid for the electronic designers. This means that a good design system should support consistent descriptions in all design description domains (system specification, behavioral, structural and physical levels) and integrated tools for the simulations and for the verifications. It is also required to automate the design steps as much as possible to reduce the needed development time and hence the cost of engineering.

Modern CAD systems can accomplish these tasks with providing consistent database management and access, powerful communication protocol for the different kind of tools and user friendly, uniform graphics environment for the user. This complex system is called framework. In most cases it may be extended by programming in a vendor dependent language. Nowadays the simulation procedure is one of the most critical points of the design process. Of course all description domains have to be simulated with the same simulator, if possible. In this case the electronic designer can apply the same test vectors (or with minor modifications) which increases the reliability of simulation and simultaneously reduces the required development time. The framework should provide the back annotation feature to improve the accuracy of the descriptions at the higher abstraction levels. Finally, CAD systems should support the available ASIC technologies (FPGA, VLSI and MCM).

When a new design methodology has been developed, it is recommended to connect to an existing framework environment, unless we wish to create a completely new CAD system. Since in most cases the new tool covers just a few design steps thus probably the extension is the better choice and it requires less effort.

A special type of the extensions is the interfacing. In this case the new CAD tool is not a part of the framework environment, just a standalone application that has a well-defined interface. This interface is usually a standard hardware description language (VHDL or Verilog), a netlist format (EDIF) or physical description file (for VLSI layouts CIF, GSDII, etc.) depending on the abstraction level. Interfacing has several disadvantages. First of all, it does not provide a uniform database management and access. Because of the required data conversion some information may have been lost. Implementation of the simulation method was discussed earlier with the forward and back annotations is also not easy. Because of these reasons, interfacing as an extension method is

recommended in the case of small applications that realise a small number of design steps and where these constraints do not cause flexibility loss.

The real integration of a CAD tool does not have these disadvantages but requires much more programming effort from the CAD engineers because of the complexity of the commercial framework products.

4.1 High-Level Synthesis Tool Integration to the Design Framework II

One of the most popular frameworks is the Design Framework II[®] of the Cadence OPUS that is extensible with C or SKILL programming languages. A new high-level synthesis tool has been developed and integrated into this framework. This program provides a user-friendly environment for editing, simulating and synthesising the data-flow graph, as illustrated in Fig. 4. The synthesis procedures cover the algorithms discussed in this report. The created data-flow graph may be a part of a larger design, its inputs and outputs may be connected to other schematics and HDL descriptions.

It means that the complex design flow may be separated into smaller parts and all engineers work only on a specific stage of the design (system specification, testing, placing and routing, final verification, etc.). This framework environment provides a huge aid for this kind of **cooperation** and **collaboration** and ensures the uniform and safe database exchange among the engineers. By using FrameMaker[®] product, the documentation may be created parallel with the design.

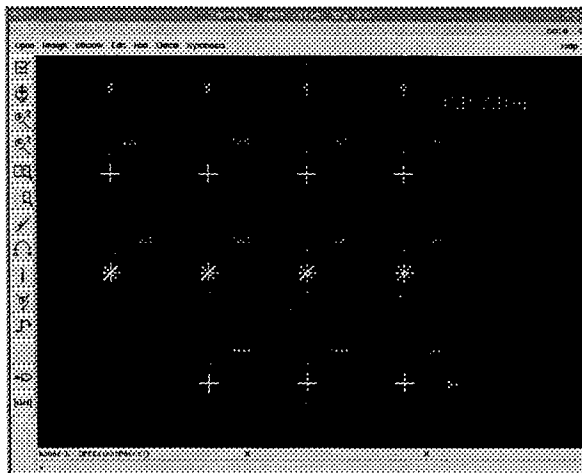


Fig. 4. Data-flow graph editor

The design flow of high-level synthesis is shown in Fig. 5. The synthesis procedure starts with the data-flow graph that may be entered by using the graphics editor or may be derived from a VHDL description via a VHDL to DFG compiler. This VHDL interface may be useful because the well-known benchmarks are written in VHDL. The graph is stored on the disks in the standard CDBA

format (Cadence Database) and a completely new view type is associated with it (*dfg*). The view of the VHDL description is derived from the *vhdl* view type. This description may be simulated by the LeapFrog[®] simulator. The nodes of

the data-flow graph are described in three different views. The *node* view contains the symbol and I/O information and is derived from *symbol* view type. The *functional* view describes the behaviour of the operation and it is written in Verilog language. Finally, the *nodeprop* view defines the properties associated with the processor. This information is stored as a set of SKILL statements. The predefined operators are collected in the library *DfgLib*. The user may extend this library by creating the listed views.

After defining the HLS constraints and selecting the type of the scheduling by the user, the synthesis procedures may be started in the given order. If required, the optimiser creates the optimised data-flow graph with the view *dfg.opt*. The scheduler and the allocator can create a cost per performance curve to aid the designer in selecting the run configuration with the most optimal trade-off. Immediately after the scheduling the data-flow graph may be simulated standalone or with the other part of the design by the Verilog-XL[®] simulator. As a final step of synthesis, the module generator creates the controlled data-flow graph (with the view of *cdfg*) and the Verilog RT level description of processors and control logic. The module generator provides additional constraints for RTL synthesiser, based on the original user parameters and the internal structure of the data-flow graph. The next design step is the technology mapping for the available technologies (gate arrays, VLSI, MCM). Using the final verification tools the exact delay times may be extracted from the layout for simulation and back annotation purpose.

A FIR filter design has been accomplished based on the design methodology discussed in this paper. The layout, shown in Fig. 6., has been

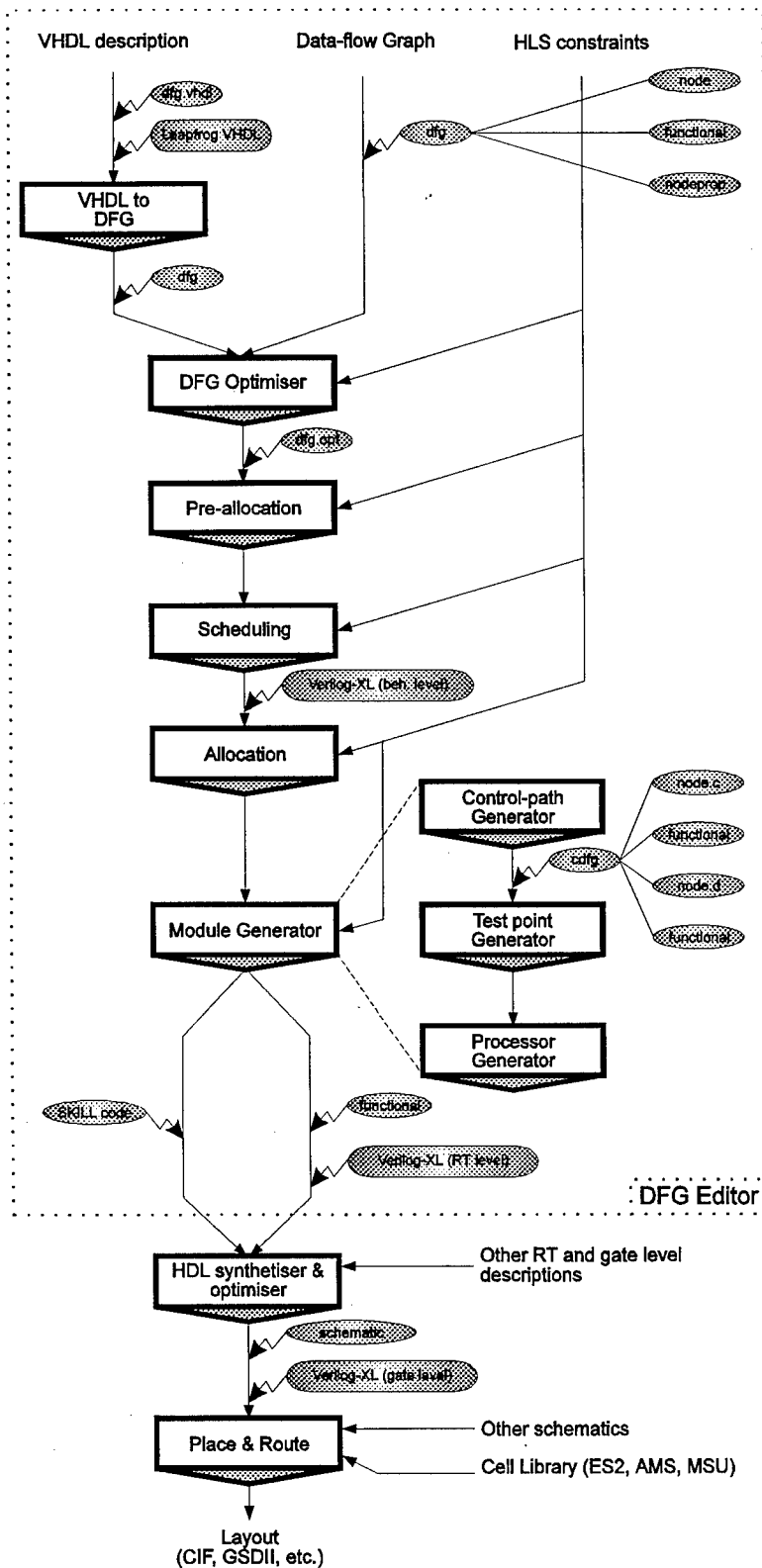


Fig. 5. Design flow of HLS tool in the Design Framework II

created by the Cell Ensemble[®] product using ES2 technology.

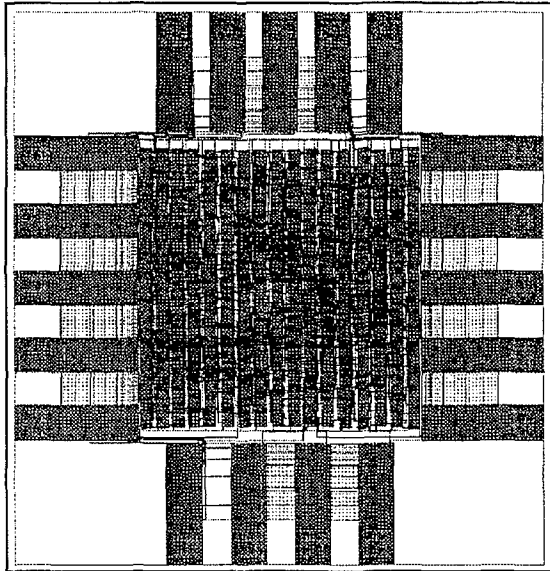


Fig. 5. Layout of the FIR filter