  

# Micro-rollback and Self-recovery Synthesis

Byung Wook Jeon and Chidchanok Lursinsap
The Center for Advanced Computer Studies of
The University of Southwestern Louisiana
Lafayette, LA 70504
Email: bwj@cacs.usl.edu and lur@cacs.usl.ed

*Abstract* - **A new approach to high level synthesis with *micro-rollback* and *self-recovery* method is presented whose objective is to generate a *reliable* system from the behavioral descriptions. The approach is formulated as a *rollback point insertion* problem based on the *probability* function of being inserted rollback points after the given *CDFG* is scheduled. This function allows us not only to search the problem space efficiently but also to avoid the exhaustive search.**

## 1 Error Detection Scheme

When one of the components of system fails, it causes an *error,* i.e., it results in an erroneous internal system state that can lead to system failure unless some special action is taken by the system to recover or to reconstruct a valid internal state. In order to prevent system failure, such errors must be detected soon after they occur. Thus, error detection is a vital step in all recovery schemes and the system must include a mechanism for detecting the failure of its components. The key to developing an effective error detection scheme is to minimize the distance between error occurrence and detection in both space and time. Ideally, these distance can be reduced to zero so that as soon as an error occurs, i.e., a component produces incorrect results, the error is detected by all the other system components that are receiving this erroneous information. This *ideal* can be achieved if all the components in the system are *self-checking* so that in addition to their normal outputs they also indicate to the rest of the system whether these outputs are correct.

No component can be self-checking with respect to all possible combinations of hardware faults. Instead, for all likely faults, the component must either produce the correct output or produce the output that contains an error indication. A component that satisfies this requirement is said to be *fault secure* [1]. No component is fault secure with respect to all possible combinations. Since the component is not guaranteed to produce a noncode output[1] immediately following the occurrence of the first fault, several additional different faults may exist in a component without any indication to the rest of the system and may cause the destruction of the fault secure property of the component. In order to prevent this situation, a component must be *self-testing* [1] such that it is guaranteed to produce a noncode output, due to the occurrence of one or more faults, before additional faults can occur and lead to the failure of the self-checking mechanism. Thus, *self-checking* must be both *fault secure* and *self-testing* so that the reliable error detection can be provided. It is therefore assumed that the error detection scheme must consist of such self-checking components. We further assume that the data flowing in the data path are coded. The error detection scheme in our

[1]Output that contain an error indication

system can be achieved by applying such self-checking design and coding techniques in the literature [17].

# 2 Micro-rollback Scheme

In this section we present the rollback technique for multiple retries in the high level synthesis which is at the heart of our system. Some constraints such as the number of available registers, the maximum allowable recovery time, and the number of retries are given by the user. The retries of a sequence of microinstructions are performed on the basis of control steps. Although a large number of retries may not be necessary from the point of view of recovery time, the need for multiple retries arises due to the fact that a single retry may not guarantee the disappearance of the error. If an error is detected while a sequence of instructions is being processed, our rollback algorithm brings the processing back a few control steps to the state that had existed in the past before the error first occurred, thus returns the system to an *error free* state where the offending microinstructions can be retried. We mean the state of the system is the contents of all storage information between two consecutive control steps, such as program status word, register contents, program counter, and the instruction register.

In order to be able to perform such an operation, the system must store all such information necessary to undo the state changes that have occurred in the last few control steps. Thus, setting up the *rollback points* is necessitated at the control step boundaries. This implies that the scheduled *CDFG* is partitioned into several segments and each rollback point is introduced at the beginning of every segment. A rollback point consists of a set of registers which store the data flowing into that particular segment. This set of registers stores the data that is only external to that segment, and the contents of these registers must be stored until the next rollback point is located. The registers having the contents of the previous rollback point will be overwritten with those of the current one's after a new rollback point is located. While the execution of the microinstructions is continued, the remaining registers are used to contain information for them. When backing up the processing to the previous *error free state* is necessitated in response to an error signal, all the required information which is the contents of registers at the most recently located rollback point is readily available. Note that no memory access is required and in turn the time required to save and load the status across the control step boundaries can be saved. To illustrate our rollback scheme, we define the following terminologies:

- $N$ : The minimum number of control steps required to perform a given *CDFG*.
- $N_i$ : The total number of $i-$ type operations in a given *CDFG*.
- $N_r$ : The number of retries which is specified by the user.
- $N_{rp}$ : The number of rollback points inserted at the control step boundaries.
- $r_{min}$ : The minimum number of registers required to execute a given *CDFG*.
- $r_{avail}$ : The maximum number of available registers specified by the user.

- $T_{max}$ : The maximum allowable recovery time for $N_r$ retries given by the user.
- $D_{max}$ : The maximum allowable recovery distance which is defined by $\lfloor T_{max}/N_r \rfloor$ .
- $T_i$ :The time needed to run the set of microinstructions from the last rollback point to $C_i$.
- $T_{r_i}$ : The recovery time for $N_r$ retries at any control step $C_i$ (i.e., $N_r \cdot T_i$ ).

The value of $T_{max}$ is also assumed to be greater than the maximum execution time among those of different type functional units. Furthermore, we assume the values of $T_{max}$ and $T_{r_i}$ are specified in terms of the number of control steps.

## 2.1 Insertion Function

In selecting a good candidate location where the rollback point can be set up, the most important factor to be considered is the probability of being inserted a rollback point at each control step boundary. Let $n$ denotes the number of control steps required to perform the remaining part of a scheduled *CDFG*. Since $T_{max}$ and $N_r$ are given by the user, the maximum allowable recovery distance can be computed by $\lfloor T_{max}/N_r \rfloor$. Let the insertion function $P_1^{D_{max}}(n,i)$ be the probability that a rollback point is inserted between two successive control steps $C_i$ and $C_{i+1}$ , $1 \leq i < n$ , in the *CDFG*. The subscript 1 indicates the fact that the rollback point can be inserted. In order to be able to compute this probability, we must find all possible patterns of rollback points being set up at every control step boundary. These patterns can be illustrated by the binary number sequences of length $n$. For example, a sequence 101... indicates the following facts:

- The rollback point was set up at the point before $C_1$ , i.e. , the initial state of the system is saved, and
- The rollback point was not inserted between $C_1$ and $C_2$ , and so on.

Note that the probability of being inserted a rollback point depends on the maximum allowable recovery time $T_{max}$ , number of retries $N_r$ , recovery time $T_{r_i}$,, and the location of the previous rollback point. The initial state of the system must be stored before the processing begins to start. Thus, inserting an element into $i$ -th position in the sequence must be satisfied the following conditions:

- 1 must be inserted if either $i = 0$ or there is 1 in $(i - D_{max})$-th position and there is no 1 in the range $[i - D_{max}, i - 1]$ , where $D_{max} < i \leq n$ .
- 0 can not be inserted if either $i = 0$ or there is 1 in $(i - D_{max})$-th position and there is no 1 in the range $[i - D_{max}, i - 1]$ , where $D_{max} < i \leq n$ .
- Otherwise, either an 1 or a 0 can be inserted

All possible patterns can be generated on the basis of these features. For example, we may generate all possible sequences for $D_{max} = 2$ and $n = 4$ as follows:

| $sequence_i$ | $rp_0$ | $rp_1$ | $rp_2$ | $rp_3$ |
|---|---|---|---|---|
| $sequence_1$ | 1 | 0 | 1 | 0 |
| $sequence_2$ | 1 | 0 | 1 | 1 |
| $sequence_3$ | 1 | 1 | 0 | 1 |
| $sequence_4$ | 1 | 1 | 1 | 0 |
| $sequence_5$ | 1 | 1 | 1 | 1 |

During the generation of such sequences, it can be easily seen that these sequences have the same characteristics as the *Fibonacci* sequence. Figure 5 illustrates how such sequences can be generated. In the following context, the first position (i.e., 0-th position) in the sequence is not considered due to the fact that it is always set to one. Thus, we can derive the following formula to compute $P_1^{D_{max}}(n,i)$. Given $D_{max}$ and $n$, let $f^{D_{max}}(n)$ denotes the number of all possible sequences. Then, $f^{D_{max}}(n)$ are the $D_{max}$*-th order* Fibonacci numbers and defined by the rules

$$f^{D_{max}}(n) = \sum_{k=1}^{D_{max}} f^{D_{max}}(n-k), \text{ if } n > D_{max};$$

$$f^{D_{max}}(n) = 2^n - 1, \text{ if } n = D_{max};$$

$$f^{D_{max}}(n) = 2^n, \text{ if } 1 \le n \le D_{max} - 1.$$

That is, they are started with $2^1, 2^2, \cdots, 2^{D_{max}-1}$'s, then $2^{D_{max}} - 1$, and then each number is the sum of the previous $D_{max}$ values. When $D_{max} = 2$, this is the usual Fibonacci sequence; for larger values of $D_{max}$ the equation can be solved by using the generating function[7].

Let the functions $f_0^{D_{max}}(n,k)$ and $f_1^{D_{max}}(n,k)$ denotes the number of 0's and 1's on the $k$-th position respectively for some given integers $n$ and $D_{max}$, where $1 \le k \le n$. Then, the function $f^{D_{max}}(n)$ can be redefined in terms of two functions $f_0^{D_{max}}(n,k)$ and $f_1^{D_{max}}(n,k)$ by

$$f^{D_{max}}(n) = f_0^{D_{max}}(n,k) + f_1^{D_{max}}(n,k).$$

These numbers also have the characteristics of the Fibonacci number. Thus, these functions can be defined by the similar way described previously. Due to the limited space, we will not describe the detailed derivations for these functions.

Note that the initial conditions of these recurrence functions are depending on the function $f^{D_{max}}(n)$. This provides an important fact that the number of 1's occurrences on $i$-th position in all possible patterns can be represented in term of the number of all possible sequences. Consequently, the probability $P_1^{D_{max}}(n,i)$ can be computed by the following rule:

$$P_1^{D_{max}}(n,i) = \frac{f_1^{D_{max}}(n-1,i-1)}{f^{D_{max}}(n-1)}$$

This is the definition of the insertion function used in our rollback scheme. The function $P_0^{D_{max}}(n,i)$ denotes the probability that a rollback point is not inserted at the location between two successive control steps $C_i$ and $C_{i+1}$, $1 \le i < n$, in the $CDFG$. It can be defined in a similar fashion. However, since we only have interests in finding the location that the rollback point can be inserted among control steps, it is not further considered.

## 2.2 Rollback Point Insertion Algorithm

It is necessary to find the appropriate points in between every two consecutive control steps and set up the rollback points on those locations in order to support the rollback scheme. The following procedure processes a such task and is iterated until all necessary rollback points are inserted. A rollback point consists of a set of registers, i.e., *rollback registers,* that store the values received from previous segment and these registers can not be shared with others. The remaining registers can be used for storing the intermediate results of computations.

**STEP 1** Construct the *live- variable* graph from the scheduled *CDFG.* The initial state of the system is stored before the first control step. That is, the external input values of *CDFG* are stored into *rollback registers.* Compute the maximum allowable recovery distance $D_{max}$ for $T_{max}$ and $N_r$. Set *current_check_point* to zero. This variable specifies the current possible rollback point between two consecutive control steps.

**STEP 2** If the length of critical path minus *current_check_point* is less than or equal to one, the process of inserting the rollback points is terminated. Otherwise, the following *STEP 3 - 6* are repeated. That is, these steps are iterated until all the necessary rollback points are set up in a given scheduled *CDFG.*

**STEP 3** Compute the *insertion function* for the remaining control steps in order to decide the possible rollback point. Select the earlier one of either the control step number whose insertion function has the largest value among those or the value of *current_check_point* plus maximum allowable recovery distance $D_{max}$. If there is a tie among the computed values of insertion functions for the corresponding control steps, the earliest point will be chosen.

**STEP 4** Increase the value of *current_check_point* by the value selected in *STEP 3.* This value of *current_check_point* specifies the *most probable* point between two consecutive control steps where the rollback point can be inserted after the most recently located rollback point. Compute the number of registers required for each control step from the previous rollback point to *current_check_point* .

**STEP 5** If the largest number of registers computed in *STEP 3* is greater than the given number of available registers $r_{avail}$, the value of *current_check_point* is subtracted by the control step number which has the largest number of registers and then repeat this step. Otherwise, go to *STEP 6.*

**STEP 6** If the recovery time $T_{r_i}$ is greater than $T_{max}$ (i.e., $T_i > D_{max}$), decrement *current_check_point* by one and repeat this process. Otherwise, the rollback point is set up at *current_check_point.* Then, the *live-variable* graph is modified such that the life regions of variables received values from the previous segment are extended to the rollback point set up in this step and go to *STEP 2.*

The algorithm is illustrated for the scheduled *CDFG* shown in Figure 4 where $T_{max} = 2$, $N_r = 1$, $r_{min} = 4$, and $r_{avail} = 6$. Let us consider the initial live-variable graph for this

*CDFG* which is given in Figure 6. Let $rp_i$ denotes the *current_check_point* variable in the algorithm. Since $N_r = 1$, $D_{max} = T_{max} = 2$ and $T_{r_i} = T_i$. The contents of variables $y$, $u$, $x$, which are the initial status of *CDFG* are saved on the point $rp_0$. We then compute the insertion function for each control step in the range [ $rp_1$ ,$rp_0 + T_{max}$ ] as follows:

$$\text{For control step 1, } P_1^2(4,1) = \frac{f_1^2(3,1)}{f^2(3)} = \frac{3}{5}$$

$$\text{For control step 2, } P_1^2(4,2) = \frac{f_1^2(3,2)}{f^2(3)} = \frac{4}{5}$$

Thus, $rp_2$ is chosen to be the most probable point. From the live-variable graph shown in Figure 5, the number of registers required to execute the operations in the range [ $C_1$, $C_2$ ] is 6 and $T_{r_2} = T_2 = 2 \leq D_{max} = 2$. Therefore, a rollback point is inserted in $rp_2$ without examining another possible location $rp_1$. The live-variable graph is then modified to reflect the fact that the saved variables can not be shared with the others in the range [$C_1$, $C_2$]. Similarly, next rollback point is inserted on $rp_3$ and the final live-variable graph is given in Figure 7.

# 3 Further Considerations

In this section, the basic rollback algorithm is extended to handle real world constraints such as mutually exclusive operations, chained operations, and multicycle operations.

## 3.1 Mutually Exclusive Operations

When two operations can never be both executed at the same time, they are said to be *mutually exclusive* and therefore can share hardware. The existence of the conditional statements such as *if-then-else* and *case* statements causes some operations to be mutually exclusive, since only one branch in a conditional block occurs at a time. Such operations can be scheduled into the same control step without increasing the number of functional units if they are performed on the same unit. In other words, they can be simply treated by taking them as a single operation. Similarly, two variables are *mutually exclusive* if they can never be both alive at the same time. They can also share a register for storage, even though their life times are overlapping. In order to handle such variables for the mutually exclusive operations, we must color the edges in the initial live-variable graph. Thus, the basic algorithm described in the last section can be used to handle mutually exclusive operations with coloring algorithm. The algorithm used for the coloring is a modified version of the node coloring algorithm in [10].

## 3.2 Chained Operations

As the duration of one control step is increased, more operations can be performed during one control step if there are enough components available to perform the operations. When more than one operation is performed sequentially within one state, the operations are said

to be *chained* together. Chaining allows higher utilization of functional units, but not the hardware sharing, by assigning them within the same control step. It can be also eliminated the need for registers to save data between two successive chained operations, because the data produced by the first one is consumed in the same state by the second. Thus, for the case of the chained operations, it is not necessary to modify the proposed rollback schemes.

## 3.3 Multicycle Operations

The possibility of scheduling operations that require more than one control step to execute has been also incorporated into the system. This feature can be implemented by taking into account the fact that multicycle operation can not be preempted to the completion of its execution. That is, once the multicycle operation is assigned, the functional unit cannot be shared by other operations until the operation is completed. In order to reflect this fact, we simply use the following strategy in the course of examining the possible location that a rollback point can be inserted;

- After the most probable location is decided, check the location if there is a multicycle operation.
- If so, *current_check_point* is decreased to the control step at which that multicycle operation begin to start.
- Otherwise, proceed the basic algorithm as usual. In other words, there is no effect on this scheme due to the above fact.

# 4 Experiment

We have implemented the proposed algorithm in *C* on a SUN SPARK station 2 system. The proposed system has been tested on several examples which were used in some systems described in the literature [9, 11, 12]. However, direct comparisons are not possible to make due to the following facts:

- There has been no published report for rollback and recovery system from the behavioral synthesis point of view which considers the real environment such as chained and multicycle operations.
- Since the requirement of our design is reliability of the system, it may involve either temporal or physical redundancy.

Thus, instead of giving the experimental results for several examples, we will classify the results for a popular example into several cases, and then intensively analyze the effect on the system. This will be followed by a brief discussion of another example borrowed from [9].

The first example is the differential equation which were described in [12]. Two types of overhead should be introduced due to our design requirement. One is *register* overhead and the other is *recovery time* overhead [13, 16]. They are defined by

- $register\_overhead = (r_{avail} - r_{min}/r_{avail}) \times 100$
- $recovery\_time\_overhead = (N_{rp}/N) \times 100$

We consider two cases in each table. In the first case we hold the maximum allowable recovery time $D_{max}$ constant and compute the number of rollback points inserted for the various number of available registers $r_{avail}$. since $r_{min} \leq r_{avail}$, $r_{avail}$ begins with $r_{min}$. The columns show the result of this case. In second case we fix $r_{avail}$ and compute the number of rollback points required for the various $D_{max}$. Each row illustrates the results for this. The entries of each table indicate the number of rollback points required in this example.

Table 1 specifies the result for the case of which neither chained nor multicycle operation is allowed. After the scheduling process, $r_{min} = 5$ . The assumptions for this case are as follows.

- one control step is equivalent to $100ns$
- delay time: ALU = $60ns$, multiplier =$80ns$

Table 1 shows that the optimum values for $r_{avail}$ and $D_{max}$ lie in the range of 20% – 60% of register overhead and 20% – 40% of recovery time overhead, respectively. The result of the experiment that has been taken into account the multicycle operations are summarized in the Table 2. In this case, $r_{min} = 6$. We also made the following assumptions:

- one control step is equivalent to $60ns$
- delay time: ALU = $40ns$, multiplier =$80ns$

| $r_{avail}$ | $D_{max}$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 6 | 3 | 2 | 1 | 1 | 1 | 1 | 1 |
| 7 | 3 | 2 | 1 | 0 | 0 | 0 | 0 |
| 8 | 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 9 | 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 10 | 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 11 | 3 | 1 | 1 | 0 | 0 | 0 | 0 |

Table 1: The case for one cycle operations only.

| $r_{avail}$ | $D_{max}$ | | | | | |
|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 |
| 6 | 4 | 3 | 3 | 3 | 1 | 1 |
| 7 | 3 | 3 | 2 | 2 | 0 | 0 |
| 8 | 2 | 3 | 1 | 1 | 0 | 0 |
| 9 | 2 | 3 | 1 | 1 | 0 | 0 |
| 10 | 2 | 2 | 1 | 1 | 0 | 0 |
| 11 | 2 | 2 | 1 | 1 | 0 | 0 |

Table 2: The case for the multicycling operations.

For the case of multicycle operations, the suitable values of $r_{avail}$ and $D_{max}$ lie in 13% – 50% of register overhead and 30% – 50% of recovery time overhead, respectively.

Table 3 shows the result that the chained operations are considered. $r_{min}$ is 6 as the result of scheduling process.

For this case, the following assumptions are made:

| $r_{avail}$ | $D_{max}$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 6 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 8 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 9 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 10 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 11 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |

Table 3: The case for the chained operations.

- one control step is equivalent to $100ns$
- delay time: ALU = $40ns$, multiplier =$80ns$

In general, it can be easily seen that the register overhead and recovery time overhead have conflicting requirements. In other words, as the recovery time overhead becomes greater we should insert more rollback point, and vice versa. Similarly, we can estimate those suitable values for each case by analyzing the corresponding tables. ¿From this analysis of the results, it can be observed that as the duration of one control step is increased, more register overhead can be expected. Conversely, we can expect more recovery time overhead as there are more multicycle operations. Another example shown in Figure 8 is the fifth-order wave digital elliptical filter borrowed from [9]. This example contains multicycle operations and the assumption we made in the previous example is also applied. This figure illustrates the final result of our algorithm when $r_{min} = 11$ for given $r_{avail} = 15 \leq 37\%$ and $T_{max} = 6$.

# 5 Conclusion

This paper presented a new approach to high level synthesis for the micro-rollback and self-recovery system whose objective is to generate a self-recoverable system from a given behavioral description. Ragahvendra and Lursinsap [13] proposed this new approach to high level synthesis. However, they did not consider the real world constraints such as chained and multicycle operations. As we described in the previous sections, such constraints have been incorporated into our system.

The probability function for the rollback point insertion allows us to avoid searching all the locations between two consecutive control step in *CDFG* in order to find the position that the rollback point can be inserted. From the result of the experiment, we have formulated a way of arriving at the reasonable values of the number of additional registers and the allowable recovery time in spite of conflicting requirements.

# References

[1] D. A. Anderson and G. Metz. Design of Totally Self-checking Check Circuits from m-Out-of-n. *IEEE Trans. Computers*, C-22(3):263–269, March 1973.

[2] M. O. Ball and F. Hardie. Effects and Detection of Intermittent Failures in Digital Systems. Technical Report IBM 67-825-2137, IBM, 1967.

[3] M. Brodsky. Hardening RAMs Against Soft Errors. In *Electronics*, volume 53. McGraw-Hill, April 1980.

[4] K. M. Chandy and C. V. Ramamoorthy. Rollback and Recovery Strategies for Computer Programs. *IEEE Trans. Computers*, c-21(6):546–556, June 1972.

[5] J. J. Horning et al. A Program Structure for Error Detection and Recovery. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, volume 16, pages 177–193. Berlin: Springer-Verlag, 1974.

[6] J. P. Roth et al. Phase II of an Architectural Study for a Self-Repairing Computer. Technical Report SAMSO-TR-67-106, U. S. AirForce Space and Missile Division, El Segundo, CA, 1967.

[7] D. E. Knuth. *The Art of Computer Programming. Volume 3/Sorting and Searching.* Addison-Wesley, 1973.

[8] Israel Koren, Zahava Koren, and Stephen Y. H. Su. Analysis of a Class of Recovery Procedures. *IEEE Trans. Computers*, c-35(8):703–712, August 1986.

[9] S. Y. Kung, H. J. Whitehouse, and T. Kailath. *VLSI Modern Signal Processing.* Prentice Hall, 1985.

[10] N. Park and A. C. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-7(3), March 1988.

[11] A. C. Parker, G. T. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proceedings of the 23th IEEE/ACM Design Automaton Conference*, pages 461–66, July 1986.

[12] P. G. Paulin and J. P. Knight. Force-directed Scheduling in Automatic Data Path Synthesis. In *Proceedings of the 24th IEEE/ACM Design Automaton Conference*, pages 195–202, July 1987.

[13] Vijay Raghavendra and Chidchanok Lursinsap. Automated Micro-Roll-Back Self-Recovery Synthesis. In *Proceedings of the 28th IEEE/ACM Design Automaton Conference*, pages 385–390, July 1991.

[14] D. D. Siewiorek, V. Kim, H. Mashburn, S. R. McConnel, and M. M. Tsao. A Case Study of C.mmp and Cm* : Part I-Experiences with Fault Tolance in Multiprocessor Systems. *Proceeding of the IEEE*, 66(10):1178–1199, October 1978.

[15] Yuval Tamir and Marc Tremblay. High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback. *IEEE Trans. Computers*, 39(4):548–554, April 1990.

[16] J. Shambhu Upadhyaya and Kewal K. Saluja. A Watchdog Processor Based General Rollback Technique with Multiple Retries. *IEEE Tran. Software Engineering*, SE-12(1):87–97, January 1986.

[17] John Wakerly. *Error Detecting Codes, Self-checking Circuits and Applications.* Elsevier Science, 1978.

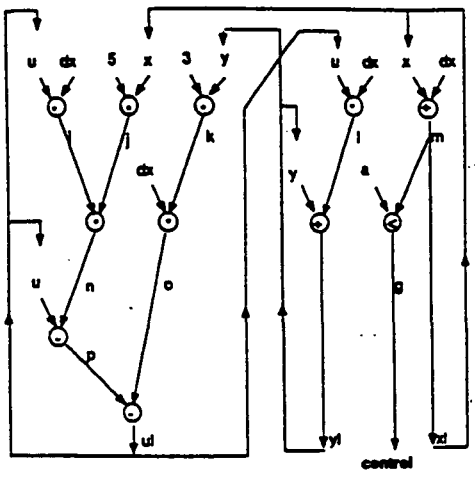


Figure 1: The data flow graph for the example

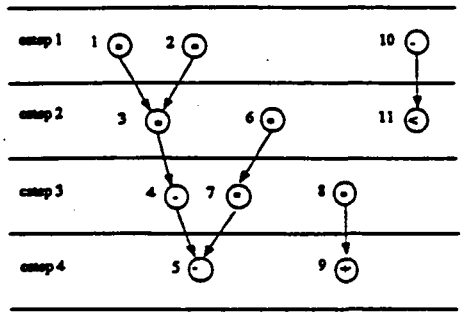


Figure 2: The scheduled data flow graph for the example

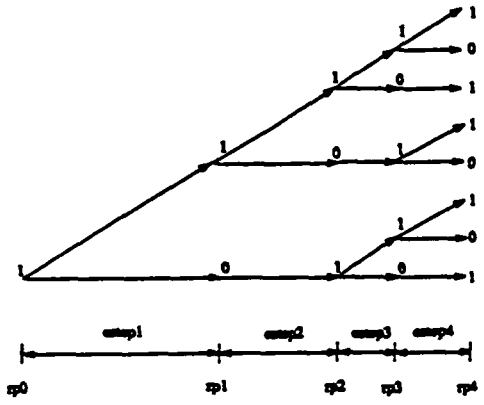

Figure 3: The generation of all possible sequences

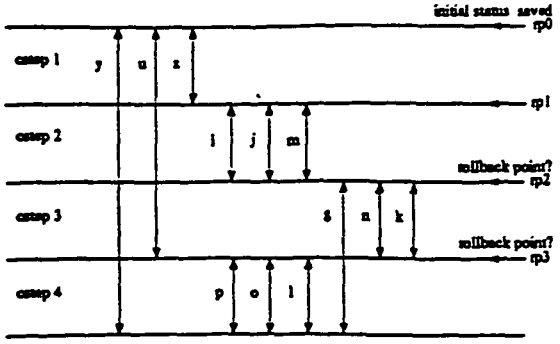

Figure 4: initial live variable graph for the scheduled CDFG

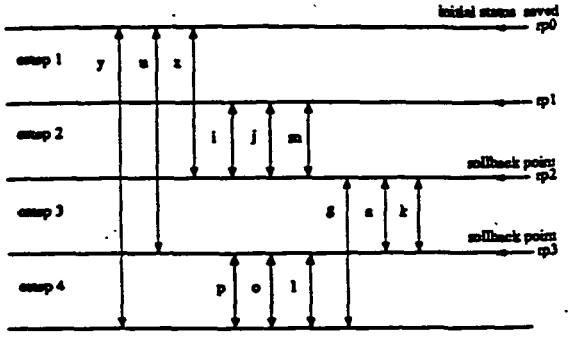

Figure 5: The final live variable graph

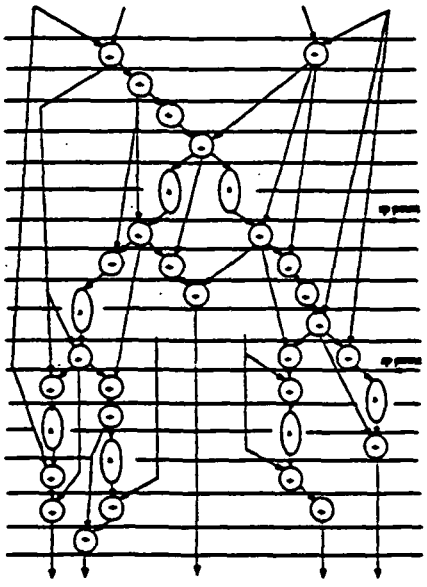

Figure 6: Fifth-order wave digital elliptical filter example