A register allocation problem solved by constraint logic programming and interval arithmetic

Corresponding author: El Mostapha Aboulhamid
email: aboulham@iro.umontreal.ca
Telephone: (514) 343-6822
Fax: (514) 343-5834

Address: Département IRO, Universite de Montreal,
C.P. 6128 Succ "Centre ville", Montreal, PQ, H3C-3J7, CANADA

A register allocation problem solved by constraint logic programming and interval arithmetic

I. E. Bennour and E. M. Aboulhamid

Département d'Informatique et Recherche Opérationnelle
Université de Montréal
CP 6128, Centre Ville, H3C-3J7, PQ, CANADA

Abstract: This paper addresses the memory allocation problem in data path synthesis. It shows that constraint logic programming (CLP) extended by interval arithmetic is an efficient paradigm to solve hard practical problems in the area of digital circuit design. Case studies, much more complex than the existing high level synthesis benchmarks, have been solved in less than 3 minutes.

I. INTRODUCTION

A common approach to high level synthesis involves data-flow graph scheduling, functional unit allocation, interconnection and memory allocation. Memory allocation maps constants and variables of a data-flow graph to storage elements (e.g., ROM, registers, register files). Both storage elements and their control part occupy a significant portion of the chip area. Therefore, it is important to minimize the number of storage elements and to organize them in such a way that control, address generation and decoding hardware are reduced.

Conventional memory allocation approaches [1-3] can be classified into two categories. In the first category, variables are mapped to registers based on a lifetime analysis of variables. The lifetime interval of a variable is the time interval between its first value assignment and its last use. Multiple variables can share a same register if their lifetime intervals do not overlap with one another. Lifetime analysis approaches aim to minimize

the number of registers but not the control hardware. These approaches are effective only when the number of registers is small, otherwise the number of interconnections and multiplexers become very large and the hardware architecture becomes very irregular. In the second category, variables are mapped to registers which are then grouped into register files (or multiport memories) based on disjoint access time. Registers can be grouped in the same register file if they are not accessed simultaneously. Register file organization is profitable only if the number and the size of register files are small: a large size of register files not only adds more address generation and decoding hardware, but also leads to longer access delay due to the decoding circuits and long data driving lines. In this paper, we propose a new register organization, called circular FIFO, as an alternative to separated register organization and register file organization. *A circular FIFO* is a row of shift registers where the output of a register is connected to the input of the following one, and the output of the rightmost register is optionally connected to the input of the leftmost register. A circular FIFO is shown in Figure 1. The leftmost register (R_0) and the rightmost register (R_{M-1}) are called *the header* and *the tail of the FIFO*, respectively. Data enter the FIFO from the header and are visible only at the tail. When a data item is inserted in the FIFO header all registers are shifted. The circularity in the FIFO adds more flexibility to data access capability: it allows multi-access to a same data, and it removes the first-in first-out constraint between data, since a data item can be reinjected in the header once it reaches the tail. Separated register organization is a special case of the FIFO organization where each FIFO contains only one register. By grouping registers into FIFOs, we reduce the number of control signals. Moreover, the use of FIFO structures may reduce the number of storage registers, since variables are not constrained to be kept in the same registers during all their lifetime. Comparing with file registers, circular FIFOs do not need memory address decoding hardware, and hence they do not suffer from decoding delays, which grow proportionally with the size of a register file. Figure 2 shows the register file based architecture and the FIFO based architecture. In the FIFO based architecture register files may still be used, but in a reduced number.

To resolve the FIFO allocation problem, we used CLP-BNR [4], a constraint logic programming (CLP) language, based on prolog augmented with relational interval

arithmetic. CLP-BNR provides a unified framework to express and solve dynamically a set of constraints over reals, integers and booleans using interval narrowing techniques.

The organization of the paper is as follows. Section II defines the FIFO allocation problem. Section III presents an overview of the interval constraint paradigm, and a formulation of the FIFO allocation problem using interval constraints. Finally, experimental results are presented in section IV.

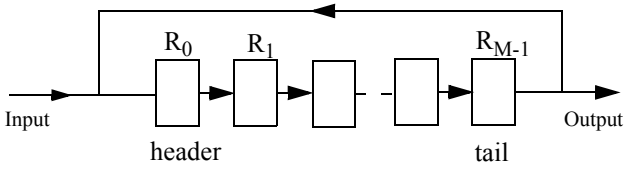


Figure 1. Circular FIFO

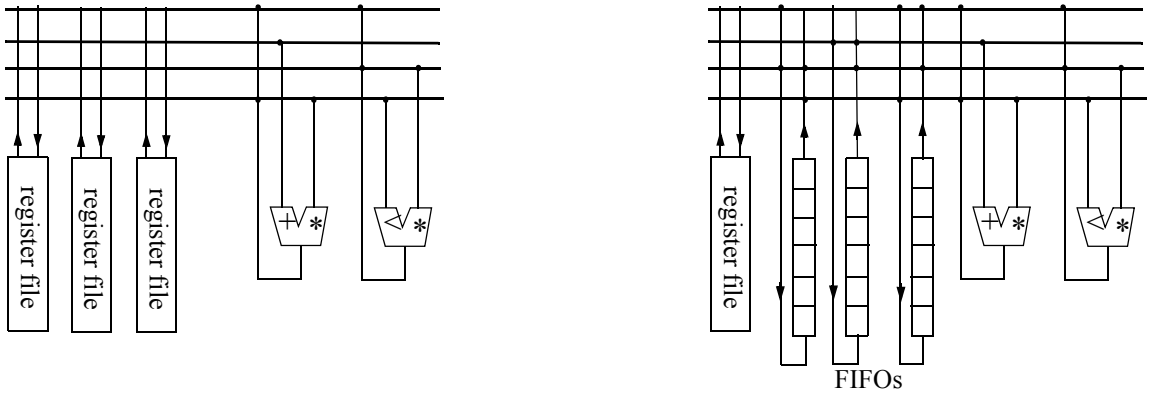


Figure 2. (a) Register file based architecture (b) FIFO based architecture

II. PROBLEM DEFINITION

A circular FIFO is defined by a set of shift registers $(R_0, R_1, \ldots, R_{M-1})$, where R_i is connected to R_{i+1}, for $i = 0, \ldots, M-2$, and R_{M-1} is connected R_0. Circular FIFOs are controlled by three operations:

- *Insert* (v_i): insertion of a variable labelled v_i in the FIFO header. After an insertion all registers of the FIFO are right shifted and the data item in the FIFO tail is lost.
- *Shift*: right shift all registers.
- *Rotate*: it is equivalent to *Insert (content of the tail)*.

During a control-step, at most one control-operation can be performed on an FIFO.

A variable v_i is defined by its write-time and its read-time(s), fixed by the scheduling task.

$$v_i: (Write\text{-}time_i, Read\text{-}time_i^{1}, Read\text{-}time_i^{2}, \ldots, Read\text{-}time_i^{n_i}).$$

v_i should be inserted in a FIFO at $Write\text{-}time_i$, and it should be available in the FIFO tail at each $Read\text{-}time_i^{j}$, for $j = 1, \ldots, n_i$. The lifetime interval of v_i is equal to $[Write\text{-}time_i, Read\text{-}time_i^{n_i}]$.

The example in Figure 3 illustrates the functioning of the circular FIFO. We observe that the circularity in the FIFO allows multi-access to a same variable (e.g., v_2), and it removes the first-in first-out constraint (e.g., v_3 enters the FIFO before v_4 and leaves after v_4). Notice that the FIFO size should always be greater than the maximal number of variables alive at the same time, and smaller than $\underset{v_i}{Min}\ (Read\text{-}time_i^{1} - Write\text{-}time_i + 1)$, otherwise some variables will not have enough time to traverse the FIFO from the header to the tail.

A *control sequence* of a circular FIFO is a sequence of control operations (insert, rotate, shift, no operation). A control sequence is *valid* for a set of variables, if it guarantees that each variable in the set is inserted in the FIFO at its writing time, and it is available in the tail at its reading time(s). It may happen that, no valid control sequence exists for a set of variables, i.e., variables in this set cannot be mapped into a same FIFO.

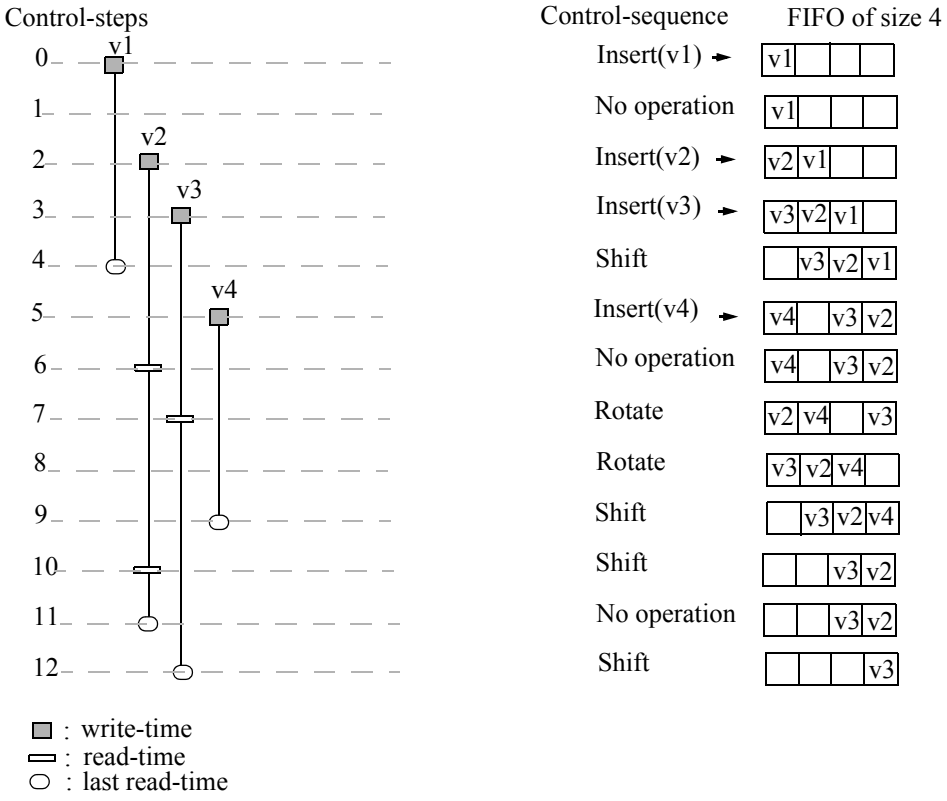


Figure 3. Illustration of the functioning of the circular FIFO

Now we can define the two problems related to allocating variables to circular FIFOs.

- **The single-FIFO allocation problem:** Given a set of variables $V = \{v_i\}$, defined by their write-time and read-times, can these variables be mapped to a same circular FIFO? If they can, what is the minimal size of such a FIFO?

- **The multi-FIFO allocation problem:** Given a set of variables $V = \{v_i\}$, defined by their write-time and read-time(s), find a mapping from V to circular FIFOs that optimizes the number of FIFOs and the total number of registers.

We suspect that the single-FIFO allocation problem is NP-complete. We solve it exactly using CLP and interval constraint paradigms. The multi-FIFO allocation problem is NP-hard. A heuristic approach is taken to resolve it. It is based on iterative resolutions of the single-FIFO allocation problem. The following steps summarize this heuristic (developed also using the same CLP environment):

– The set of variables V is divided into disjoint *clusters* C_l (subsets*)* such that: (1) variables in the same cluster do not have neither the same writing-times nor the same reading-times, (2) if two variables v_i and v_j belong to a same cluster and the writing-time of v_j is greater than the writing-time of v_i then the reading-time of v_j is be greater than the reading-time of v_i, (3) in each cluster C_l, the maximal number of variables in C_l alive at the same time is smaller than or equal to the minimal live-time among all variables in C_l. These criteria increase the likelihood of mapping successfully all variables in a cluster to a same circular FIFO.

– For each cluster, check if it can be mapped to a same circular FIFO using the exact resolution of the single-FIFO problem. If not, the variables causing the failure are removed from the current cluster and are redistributed on other clusters if possible. New clusters are added if necessary. This process is repeated until all the clusters are mapped to some feasible circular FIFOs.

– The last step is to reduce the number of clusters, i.e., the number of circular FIFOs. Repetitively, we pick the cluster containing the minimal number of variables, then we try to redistribute all its variables on the other clusters. We check if a variable can be added into a cluster by solving the single-FIFO problem. This process is repeated until all the clusters are considered.

III. FORMULATION OF THE SINGLE-FIFO ALLOCATION PROBLEM USING INTERVAL CONSTRAINTS

A. A brief overview of the interval constraint paradigm [4] [5]

CLP-BNR is a constraint logic programming system based on relation interval arithmetic. The use of interval arithmetic allows reasoning about domains of variables rather than fixed values. An interval is a closed bounded set of numbers, it defines either a continuous range of real numbers laying between a lower and an upper bound or a discrete range of integer values laying between integer bounds. An interval $[a, b]$ is:

$$[a, b] = \{x \ / \ a \le x \le b\}$$

The two endpoints of an interval X are denoted by $\overline{X}$ and $\underline{X}$. Thus, $X = [\underline{X}, \overline{X}]$. The interval $[x, x]$ is a degenerate interval which is not distinguish from x. Two intervals are

equal if their corresponding endpoints are *equal*. If x is in the interval X, we write $x \in X$. Operations on intervals can be either the basic arithmetic operations defined on the reals (+, -, *, /, min, max, sin, cos, etc.), or arithmetic relations (equality, inequality, inclusion, etc.). In the following, we give a semantic of some of these operations.

- Arithmetic operations

  Interval addition is defined as

  $$[\underline{X}, X] + [\underline{Y}, Y] = [\underline{X} + \underline{Y}, X + Y]$$

  The negation of an interval, from which the rules of interval subtraction can be deduced, is defined as

  $$-X = -[\underline{X}, X] = [-X, -\underline{X}] = \{-x \mid x \in X\}$$

  The max of two intervals is defined as

  $$max(X, Y) = [max(\underline{X}, \underline{Y}), max(X, Y)]$$

- Arithmetic relations

  The *equality constraint* between two intervals X and interval Y, denoted $X == Y$, is *true* if the intervals X and Y can be constrained to be equal by *narrowing* (reducing) X and/ or Y.

  Example.

  If $X = [2, 5]$ and $Y = [4, 8]$, then $X == Y$ is true because both X and Y can be reduced to the same interval $[4, 5]$.

  If $X = [2, 5]$ and $Y = [6, 8]$, then $X == Y$ is false.

  The *less than or equal constraint* between an interval X and an interval Y, denoted $X \leq Y$, is true if the interval X can be constrained to an interval Z where each element of Z is less than or equal to an element of Y.

  Example.

  If $X = [2, 15]$ and $Y = [6, 8]$, then $X \leq Y$ is true because X can be reduced to the interval $[2, 8]$ which is less than or equal the interval Y.

  If $X = [9, 15]$ and $Y = [6, 8]$, then $X \leq Y$ is false.

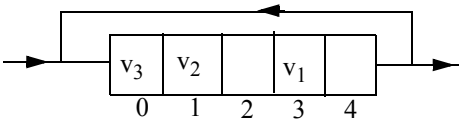
B. Formulation of the single-FIFO allocation problem

The formulation of the single-FIFO allocation problem is based on the successive states of the FIFO during the execution of a control sequence (sequence of shift, insert and rotate operation). If we label the shift registers composing a circular FIFO from 0 to *M*-1, then a FIFO state can be defined by the set of variables inside the FIFO and their position. The *distance* from a variable v_i to a variable v_j, denote by $Dist(v_i, v_j)$, is defined as following:

$$Dist(v_i, v_j) = (pos(v_j) - pos(v_i)) \bmod M$$

where $pos(v_i)$ is the v_i position inside the FIFO.

Example:



$$Dist(v_2, v_1) = (3-1) \bmod 5 = 2 \ , \ Dist(v_2, v_3) = (0-1) \bmod 5 = 4$$

Based on the distance definition, we have:

$Dist(v_i, v_j) \in [0, M-1], \forall v_i, v_j$, where *M* is the FIFO size

$Dist(v_i, v_i) = 0, \forall v_i$

$Dist(v_i, v_j) = M - Dist(v_j, v_i), \quad \forall v_i \neq v_j$

$Dist(v_i, v_j) = (Dist(v_i, v_k) + Dist(v_k, v_j)) modulo\ M, \quad \forall v_i \neq v_j \neq v_k$

Notice that, due to the last equality, the distance is still defined even between variables which are not into the FIFO at the same time (i.e., variables which are not alive at the same time). Thus, the distance values between pairs of variables are sufficient to capture all the successive states taken by FIFO during the execution of a control sequence. It can be proved that finding a valid control sequence for a set of variables is equivalent to finding feasible distance values among variables.

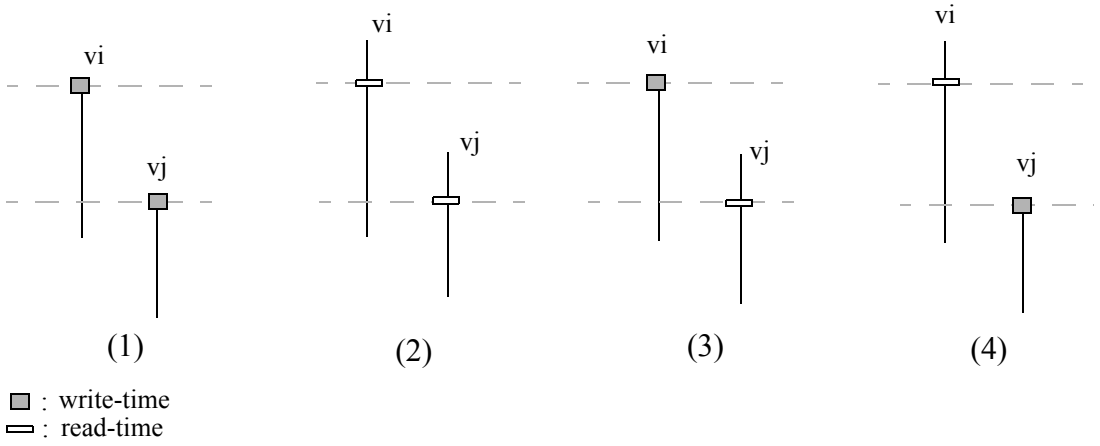


Figure 4. The four precedence orders between write and read operations

To state the constraints which a distance between two variables should satisfy, we distinguish four cases depending on the precedence order between their write and read operations. These cases are shown in Figure 4.

Case 1: *two successive write operations*

$Dist(v_j, v_i)$ should satisfy:

$$Dist(v_j, v_i) \leq Write\text{-}time_j - Write\text{-}time_i$$

because between the insertion-time of v_i and the insertion-time of v_j into the FIFO the maximal number of control operations (shift, insert, rotate) that can be done is equal to $(Write\text{-}time_i - Write\text{-}time_j - 1)$.

Case 2: *two successive read operations*

$Dist(v_j, v_i)$ should satisfy:

$$Dist(v_j, v_i) \leq Read\text{-}time_j^{\,k} - Read\text{-}time_i^{\,l}$$

This constraint guarantees that, after the reading-time of v_i there are still enough control-steps to propagate v_j until the FIFO tail.

Case 3: *write operation followed by read operation*

$Dist(v_j, v_i)$ should satisfy:

$$Dist(v_j, v_i) \leq Read\text{-}time_j^{\,k} - Write\text{-}time_i + 1$$

This constraint guarantees that, after the writing-time of v_i there are still enough control-steps to propagate v_j until the FIFO tail.

Case 4: *read operation followed by write operation*

$Dist(v_j, v_i)$ should satisfy:

$$Dist(v_j, v_i) = M - 1 \text{ , if } Write\text{-}time_j = Read\text{-}time_i^{\,l}$$

$$Dist(v_j, v_i) \le Write\text{-}time_j - Read\text{-}time_i^{\,l} - 1 \text{ , otherwise}$$

If $Write\text{-}time_j = Read\text{-}time_i^{\,l}$, then the constraint states that, when v_j is inserted in the FIFO header, v_i should be in the tail; otherwise it states that between the reading-time of v_i and the writing-time of v_j in the FIFO the maximal number of shift and rotate operations that can be done is equal to $(Write\text{-}time_i - Read\text{-}time_i^{\,l} - 1)$.

Figure 5 shows the set of distance constraints for the example used in Figure 3.

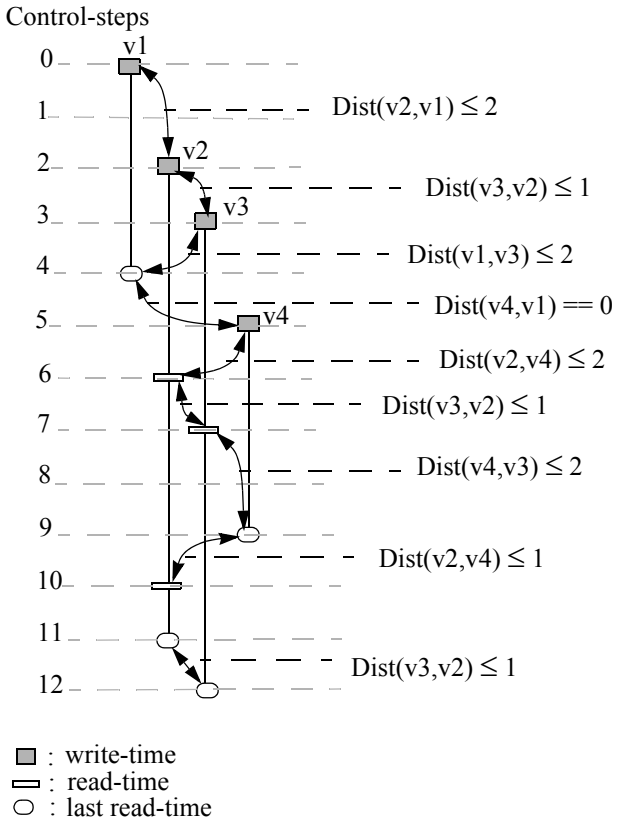


Figure 5. The set of distance constraints for the example used in Figure 3

Now we give the complete formulation of the single-FIFO allocation problem using relational interval arithmetic:

Minimize the FIFO size M under:

(a) $M \in [M\_min, M\_max]$

where *M_min* is the maximal number of variables alive at the same time, and

$$M\_max = \min_{v_i} (Read\text{-}time_i^{1} - Write\text{-}time_i + 1)$$

(b) $Dist(v_i, v_j) \in [1, M-1]$, $\forall v_i \neq v_j$ and v_i, v_j are alive at the same time

(c) $Dist(v_i, v_j) \in [0, M-1]$, $\forall v_i \neq v_j$ and v_i, v_j are not alive at the same time

(d) $Dist(v_i, v_j) == (M - Dist(v_j, v_i))$, $\forall v_i \neq v_j$

(e) $Dist(v_i, v_j) == (Dist(v_i, v_k) + Dist(v_k, v_j)) modulo\ M, \quad \forall v_i \neq v_j \neq v_k$

(f) $Dist(v_i, v_j) \leq c_f$

Constraint (a) defines the possible values of the FIFO size. Constraint (b) states that variables which are alive at the same time cannot share the same registers into the FIFO: their distances should be greater than zero. Constraints (c), (d) and (e) are the distance's properties. (f) is the set of distance constraints between variables that should be satisfied, as discussed previously in cases (1) to (2). It is important to mention that all these constraints are directly expressible in CLP-BNR.

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We present in Tables 1 and 2 experimental results obtained using the proposed circular FIFO allocation approach. Table 1 summarizes the results of four benchmarks frequently used by the high level synthesis community. All benchmarks are scheduled using one adder and one multiplier. The second column of Table 1 indicates the minimal number of registers necessary to hold variables and constants. This number corresponds to the maximal number of variables and constants alive at the same time. The third column corresponds to the number of circular FIFOs obtained using our allocation approach. The total number of registers used in the final design is indicated in the fourth column. We observe that the number of registers used is always optimal, and that these registers are

grouped in a small number of FIFOs, which reduces considerably the control signals. Figure 6 shows the obtained design of the polynomial divider benchmark. To evaluate the efficiency of our approach in general case, we have used random examples, which are more complex than usual high level synthesis benchmarks. The results are given in Table 2. For each example, we generate a set of variables with random lifetime intervals, then we perform the FIFO allocation algorithm. The results are quite interesting. First, the number of circular FIFO (third column) is very small comparatively to the minimal number of registers (second column), in average each FIFO contains five to six registers. Second, the total number of registers used (fourth column) exceeds the optimum by 3 registers at most.

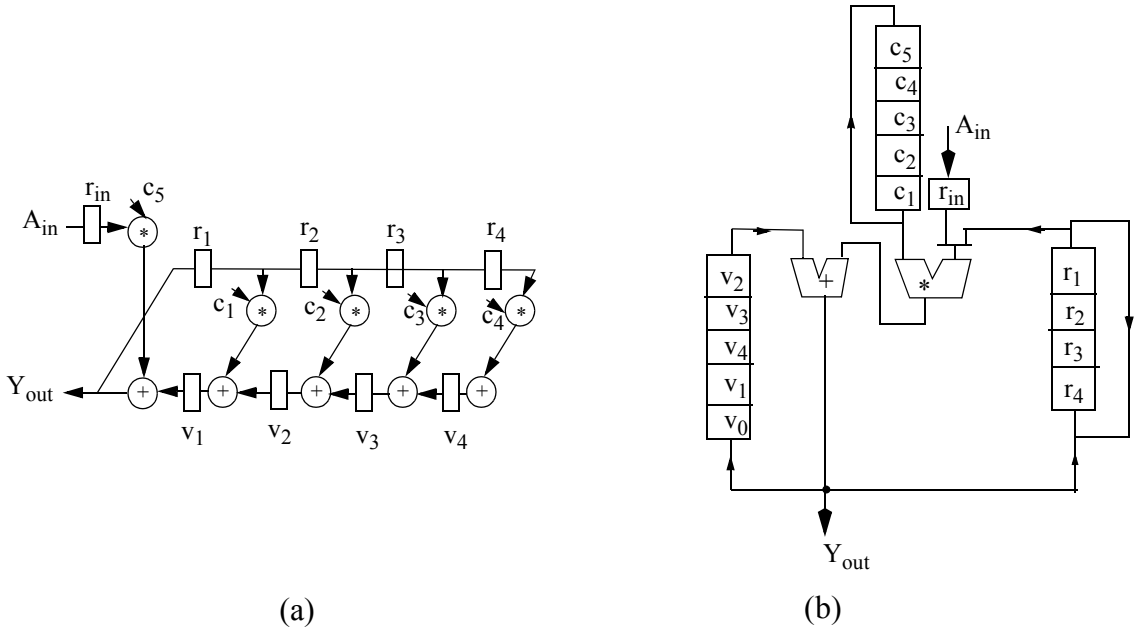
The low CPU times taken to solve the relatively large random examples (up to 100 variables) reflect the solving power of the interval constraint paradigm. By expressing the constraints of the problem over interval-valued variables rather then over integer variables, we have noted that the interval narrowing resolution technique reduces considerably the resolution time. The dichotomous fashion used by CLP-BNR to trim intervals (partial enumerations using the *solve* prolog function) is very efficient in practice to identify and to remove invalid solution spaces.

Table1 : Benchmark results

Benchmark	Minimal # of registers	# of FIFOs	# of registers used	CPU time (sec.)
IIR filter	20	5	20	10
FIR filter	15	3	15	7
Polynomial divider	15	3	15	7
Three order filter	7	2	7	2

Table2 : Experimental results using random examples

# of variables	Minimal # of registers	# of FIFOs	# of registers used	CPU time (sec.)
20	14	3	15	10
20	15	4	17	16
20	12	4	12	7
20	11	3	12	18
20	11	3	13	9
30	18	3	19	171
30	16	3	18	44
30	18	4	18	32
30	17	3	17	494
30	20	4	21	63
50	26	4	27	85
50	23	5	26	86
50	22	4	25	76
50	28	4	30	66
50	22	3	22	177
70	37	5	39	155
70	30	5	32	82
70	31	6	32	32
70	33	6	36	109
70	34	6	36	52
100	50	7	53	158
100	42	6	45	221
100	43	6	45	239
100	39	5	42	198



(a) (b)

Figure 6. (a) DFG of the polynomial divider (b) its implementation

V. CONCLUSIONS

We have resolved efficiently a memory allocation problem using the interval constraint paradigm. We showed that, for some problems, like the single-FIFO, expressing constraints using interval arithmetic is straightforward. The experimental results obtained on relatively large examples reflect the solving power of this paradigm. Constraint logic programming extended by interval arithmetic is a very promising paradigm for solving other CAD problems.

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.

[2] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis*, Kluwer Academic Publishers, Boston, 1992.

[3] R. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, Boston, 1991.

[4] *BNR PROLOG, User Guide,* Version 4, Ottawa, Canada, 1993.

[5] R.E. Moore, *Methods and Applications of Interval Analysis*, SIAM, 1979.