Malaysian Journal of Computer Science, Vol. 15 No. 1, June 2002, pp. 15-27

REGISTER OPTIMISATION BY EQUIVALENCE ANALYSIS

Mohamed Fettach, Lahcen Elarroum and Abdellatif Hamdoun
Faculty of Sciences Ben M'sik, University Hassan II, Casablanca, Morocco
LTI, Faculté des Sciences Ben M'sik
BP 7955, Sidi Othmane, Casablanca, Maroc
Tel. : 212 22 70 46 71
Fax : 212 22 70 46 75
email: fettachm@yahoo.fr
elarroum@yahoo.fr
alhamdoun@yahoo.fr

ABSTRACT

Traditionally, the register allocation is based on the lifetime analysis of variables. A register can be shared by multiple variables if they have mutually disjointed lifetime intervals. In this paper we attempt to extend the register sharing by another type of analysis called equivalence analysis. After the register allocation by a conventional register allocation algorithm such as left edge algorithm, some incompatible registers can possibly have the same content or their contents can be included in the contents of some other registers in any state of a design. Such registers are totally or partially equivalent and they can be merged into a single register. Our approach offers then a supplement potential for the register optimisation. Hence, it is allowed to go beyond minimisation by lifetime analysis. However, it does not only optimise the number of registers but also reduces the interconnection cost and the number of functional units previously allocated. Therefore, it reduces the implementation cost and improves the design performance.

Keywords: ***High-level synthesis, Register optimisation, Equivalence analysis, Interconnection cost***

1.0 INTRODUCTION

High-Level Synthesis (HLS) is the design process which transforms a behavioural description of a digital design into its description of Register Transfer Level (RTL) structure [1]. Two major tasks are usually distinguished in HLS: *Scheduling* and *Hardware allocation*. Scheduling is the process of partitioning arithmetic and logic operations into states (or control steps) such that operations scheduled in the same state can be executed concurrently. Hardware allocation is the process of selecting hardware units, that is, functional units (fu) to perform the arithmetic and logic operations, registers to store value of variables, and connections between the functional units and registers for data value transfers. The goal of the hardware allocation is to minimise the total amount of hardware elements. Hardware allocation is usually subdivided into three interdependent subtasks: (1) functional unit assignment, (2) register allocation and (3) data transfer allocation. The results of one subtask will affect the performance of the others significantly.

This paper is concerned with the register allocation. Value of variables which are generated in one state and used in a later state must be stored in registers. Although we can trivially allocate a distinct register to each variable, a register can be shared by multiple variables if their lifetimes do not overlap. The lifetime of a variable is the time of a period in which the value of the variable must be saved in a register. Register allocation is the problem of mapping variables onto a minimum set of registers according to their lifetime analysis. In order to minimise the number of registers, the possibility of register sharing is used. However, having less registers does not necessarily guarantee that the final design will be optimal. The register merging can have a direct impact on interconnection cost. Indeed, after register merging, more traffic is needed between functional units and registers that results in an increase of interconnection cost.

Many techniques [2-9] have been developed for allocating as few registers as possible taking advantage of the register sharing possibility between different variables. However, after the register allocation by a conventional register allocation algorithm such as the clique partitioning algorithm [3] and [4], the left edge algorithm [5] or the bipartite weighted matching algorithm [6], some incompatible registers can possibly have the same content (totally equivalent registers) or their contents can be included in the contents of some other registers (partially equivalent registers). Our approach allows us to identify and to merge such registers. It is based on the equivalence analysis that is a novel method of register optimisation. It consists of partitioning registers whose utility phases overlap, in classes such as any class regroups the totally or partially equivalent registers. Registers of a same class can be replaced by a single register. The utility phase of a register is a subset of states during which the register is useful. If we assume that each variable is allocated to a same register, then the utility phase of a register represents the amount of the lifetimes of variables allocated to this register. The utility phase of a register R_i can be represented by an interval <$SS(R_i)$, $ES(R_i)$>, where the Starting State of the register R_i ($SS(R_i)$) is the state at which the register R_i is defined and the Ending State of the register R_i ($ES(R_i)$) is the state at which the register R_i is used for the last time. The equivalence analysis is allowed to go beyond minimisation by compatibility analysis in merging some incompatible registers. Two registers are said to be in*compatible* if they are useful simultaneously, e.g. if their utility phases overlap. Three cases are possible for utility phases of incompatible registers (Fig. 1). In the first case (Fig. 1(a)), the two registers R_i and R_j can be merged if they are equivalent in any state of their utility phase. In the second case (Fig. 1(b)), the register R_j can be replaced by the register R_i if they are equivalent in any state of the utility phase in common. In the last case, we can decompose the utility phases of the registers R_i and R_j into three segments (Fig. 1(c)). Since the registers R_i and R_j are compatible in segments 1 and 3, they can be merged into a single register if they are equivalent in any state of the segment 2. However, the register merging based on the equivalence analysis does not require additional interconnect elements, but on the contrary, it allows to save registers, buses and functional units as will be proved subsequently. Therefore, after the register allocation has been carried out by a conventional register allocation algorithm, our approach performs a postprocessing step to complete the register optimisation. The equivalence analysis has been used in the theory of automaton to minimise the number of states [10] and [11].

This paper is structured as follows: Sections 2 and 3 describe the totally and partially equivalence analysis respectively. Section 4 discusses the impact of register merging on the interconnection cost. Section 5 concludes the paper. Finally, some definitions of terms used in the paper are included in the Appendix.

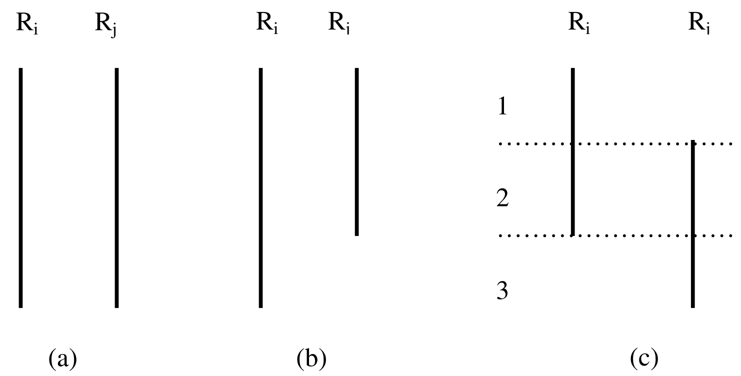


Fig. 1: Possible cases of incompatible registers

2.0 TOTAL EQUIVALENCE

Some registers can possibly have the same content in any state of digital design. Since the state graph can be cyclic, the content of a register in a state can be different at every passage by this state (except the initialisation of the corresponding variable). Therefore, a register cannot have one same content in any state of a design. However, it is not necessary to know the explicit content of any register in the different states. We only need to know if contents of two registers are identical or not in any state of the design. To solve this problem, we will state the following theorems:

Theorem 1:
Two registers R_i and R_j, defined by the two following operations: $R_i = R_{i1}$ $\mathbf{op_i}$ R_{i2} and $R_j = R_{j1}$ $\mathbf{op_j}$ R_{j2} which are scheduled in a same state S_k, are equivalent in this state if:

- $\mathbf{op_i}$ and $\mathbf{op_j}$ are of the same type,
- registers R_{i1} and R_{j1} are equivalent , and
- registers R_{i2} and R_{j2} are equivalent.

Proof:
Since the registers R_{i1} and R_{j1} on the one hand, the registers R_{i2} and R_{j2} on the other hand are equivalent, they have the same content. Since the two operations $\mathbf{op_i}$ and $\mathbf{op_j}$ are of the same type and operands being the same, it is obvious that results can be the same.

If the operations $\mathbf{op_i}$ and $\mathbf{op_j}$ are commutative, and the registers R_{i1} and R_{j1} and/or R_{i2} and R_{j2} are not equivalent, we can exchange the operands of one operation (for example the operation $\mathbf{op_i}$) and then verify if the pairs of registers (R_{i1}, R_{j2}) and (R_{i2} and R_{j1}) are equivalent.

Theorem 2:
Two incompatible registers R_i and R_j are totally equivalent if they are equivalent in any state where at least one of them is defined.

Proof:
So that one can speak of the equivalence between two registers in a state, it is necessary that one of them is defined in this state. If the two registers R_{i1} and R_{j1} are equivalent in any state S_k where at least one of these registers is defined, they are totally equivalent.

2.1 Implication Graphs

According to theorem 1, in order for the registers R_i and R_j to be equivalent, it is necessary that R_{i1} and R_{j1} on the one hand, R_{i2} and R_{j2} on the other hand are also equivalent. Since, the registers R_{i1} and R_{j1} (item for the registers R_{i2} and R_{j2}) can be defined by others operations:

$$R_{i1} = R_{i11}\ \mathbf{op_{i1}}\ R_{i12};$$
$$R_{j1} = R_{j11}\ \mathbf{op_{j1}}\ R_{j12};$$

The registers R_{i1} and R_{j1} are equivalent if:

- $\mathbf{op_{i1}}$ and $\mathbf{op_{j1}}$ are of the same type,
- registers R_{i11} and R_{j11} are equivalent, and
- registers R_{i12} and R_{j12} are equivalent.

This procedure will be repeated for all corresponding used pairs of registers (R_{i11}, R_{j11}), (R_{i12}, R_{j12}), …etc. However, given that the number of registers in a design is limited, this procedure converges rapidly. We can modelise this procedure by a directed graph (Fig. 2). The nodes represent the pairs of registers and any edge directed from a node X_m to a node X_n means that the registers corresponding to the node X_n are equivalent if the registers corresponding to the node X_m are equivalent. One can say that X_m implies X_n and the graph is known as an *Implication Graph*. Similar graphs are used in the reduction of states in a sequential machine [11].

2.2 Equivalence Table

To determine the equivalent registers, we construct a table called an *equivalence table*. The lines and columns of the equivalence table are the registers of the design. This table is triangular because the equivalence relation between registers is reflexive and symmetrical. Every cell (i, j) of the equivalence table contains: 0 if the registers R_i and R_j are not directly equivalent, 1 if the registers R_i and R_j are directly equivalent, or the pairs of registers used by the source operations of the registers R_i and R_j. However, the equivalence table can be completely specified while applying the following rule:

> *Two registers R_i and R_j are equivalent if and only if they are not implied by any pair of registers as no equivalent.*
>
> Indeed, the no equivalence of a pair of registers can imply the no equivalence of all pairs where these registers are reused. The non-existence of the no equivalence for a pair of registers permits us to suppose that these registers are equivalent.

2.3 Algorithm

The totally equivalence analysis is described by the **algorithm 1**. It is done separately for every type of operation. For each state of the state graph, we establish the equivalence relations between the registers defined by the operations scheduled in this state. Then, we construct the equivalence table. In order to completely specify the equivalence table, we construct the implication graphs. The equivalence table completely specified is then treated from the left to the right in an iterative way. For each iteration, we treat a column of the table. For each column, we determine an equivalence class of the corresponding register. The equivalence classes represent the maximal sets of equivalent registers. The complexity of this algorithm is $O(n\_o.n\_s.r^2)$, where n_o is the number of operations, n_s is the number of states and r is the number of registers.

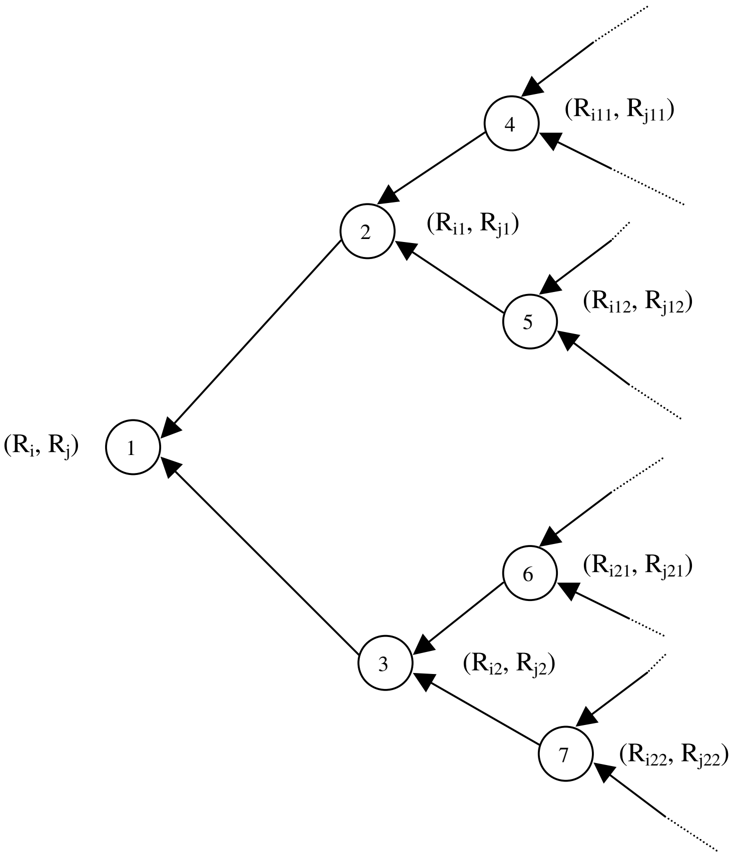


Fig. 2: An example of an Implication Graph

Algorithm 1
For every type of operations, **do**

1. **For** every state of the state graph, establish the equivalence relations between registers defined by operations scheduled in this state,
2. Construct the equivalence table,
3. Construct the implication graphs,
4. Specify completely the equivalence table,
5. The equivalence table will be treated from the left to the right,
    - a- Set i = 1,
    - b- Determine equivalence classes, as:

      $C_i = \{R_i\} \cup \{R_j \,/\, T_{ij} = 1; j = i+1, i+2, ..., r\}$

      where T_{ij} is the value in the cell (i, j) of the equivalence table and r is the number of registers,
    - c- Point all elements of C_i,
    - d- Increment i, **if** i = r **then** stop, **else** continue,
    - e- **If** R_i is pointed **then** return to step (d), **else** return to step (b).

2.4 Example

Fig. 3 shows an example of a state graph. In this graph we have noted only operations of the addition type. Note that states without operations are states where operations of others types are scheduled. For any state, we establish the equivalence relations between the registers defined by the operations scheduled in this state. From these equivalence relations, we construct the equivalence table (Table 1).

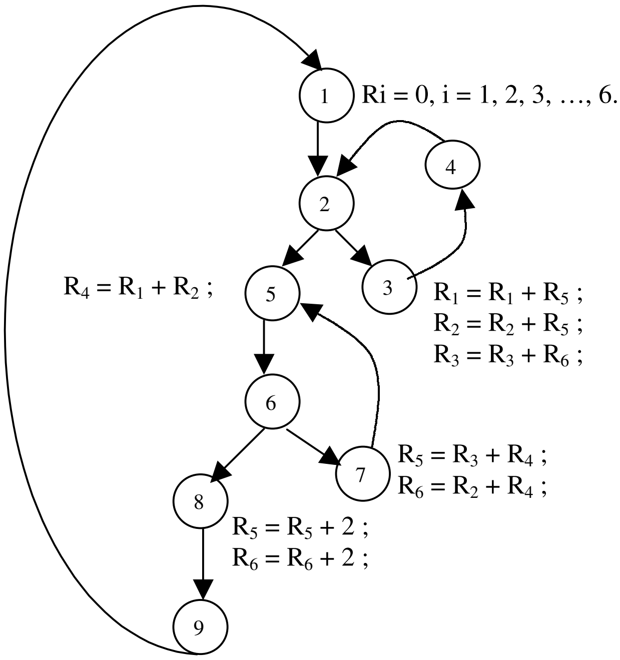
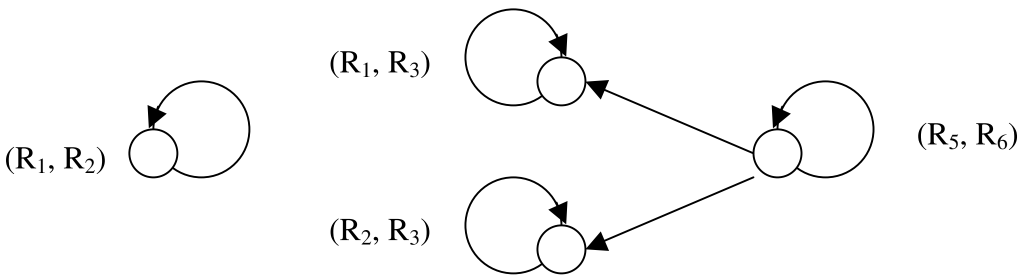


Fig. 3: An example of a State Graph

Table 1: The equivalence table incompletely specified

R_2	(R_1, R_2)				
R_3	(R_1, R_3) & (R_5, R_6)	(R_2, R_3) & (R_5, R_6)			
R_4	0	0	0		
R_5	0	0	0	0	
R_6	0	0	0	0	(R_2, R_3) & (R_5, R_6)
	R_1	R_2	R_3	R_4	R_5

The equivalence table (Table 1) is incompletely specified. In particular, the registers R_1 and R_3 are equivalent if the pairs of registers (R_1, R_3) and (R_5, R_6) are equivalent. Similarly, the registers R_2 and R_3 are equivalent if the pairs of registers (R_2, R_3) and (R_5, R_6) are equivalent. These pairs of registers are not explicitly no equivalent. Then, we try to completely specify the equivalence table. We obtain the following implication graphs:



We remark that one node can implicate itself. Since there is not explicit no equivalence, all pairs of registers are assumed equivalent. The equivalence table is now completely specified (Table 2).

Table 2: The equivalence table completely specified

R_2	1				
R_3	1	1			
R_4	0	0	0		
R_5	0	0	0	0	
R_6	0	0	0	0	1
	R_1	R_2	R_3	R_4	R_5

Finally, we obtain the following equivalence classes: $C_1 = \{R_1, R_2, R_3\}$, $C_2 = \{R_4\}$, $C_3 = \{R_5, R_6\}$. Therefore, we need only three registers instead of five: $r_1 = \{R_1, R_2, R_3\}$, $r_2 = \{R_4\}$, $r_3 = \{R_5, R_6\}$.

3.0 PARTIAL EQUIVALENCE

In the total equivalence analysis, we have assumed that all registers have the same bit width. However, the registers in a digital design do not necessarily have the same bit width. Hence, the content of a register can be included in the content of another register, such registers are *partially equivalent*. The partial equivalence analysis allows us to improve the register optimisation while increasing the number of equivalent registers. There are three possible cases to have the content of a register R_j included in the content of a register R_i (Fig. 4).

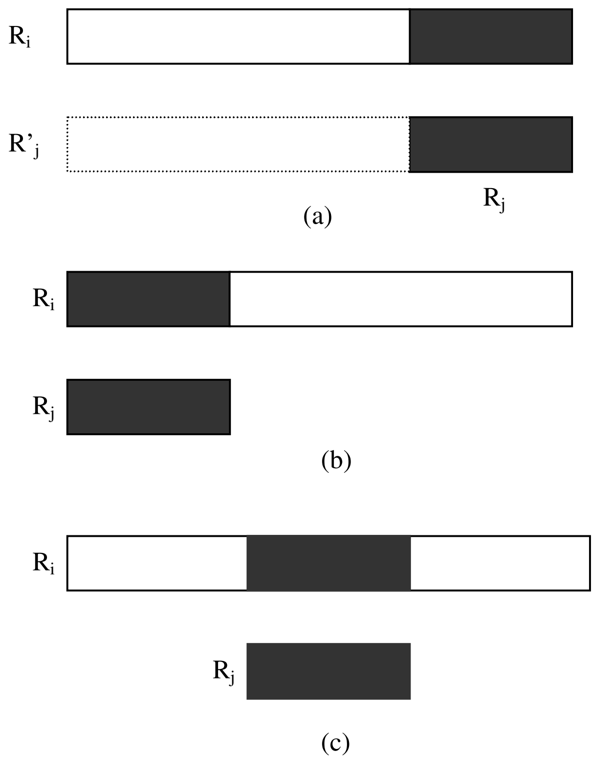


Fig. 4: Possible cases in partial equivalence analysis

3.1 First Case

The part of register R_i whose content is identical to the register R_j is completely on the right of R_i (Fig. 4(a)).

Since the arithmetic and logic operations transmit bits from the right side towards the left one, we can complete the length of the register R_j by arbitrary bits to have two registers with the same bit width. The part added to register R_j plays no role, we can choose it identical to the corresponding part of register R_i. Thus, this case can amount to the one of the total equivalence studied previously. The registers R_i and R'_j in Fig. 4(a) are totally equivalent and they can be replaced by one register.

3.2 Second Case

The part of register R_i identical to register R_j is completely on the left side of R_i (Fig. 4(b)).

The problem consists in finding a rule permitting to know if two registers R_i and R_j are partially equivalent. We will limit it to operations of the following form: R_i:= R_i **op** c, where c is a constant.

Definition:

Let R_i and R_j be two registers with different bit widths. Let w_i and w_j denote the bit widths of registers R_i and R_j respectively, with $w_i > w_j$. We suppose that registers R_i and R_j are defined in a state S_k by:

R_i := R_i **op** x;
R_j := R_j **op'** y;

The registers R_i and R_j are partially equivalent in the state S_k, if the operation **op** modifies the part of register R_i identical to register R_j in the same way as the operation **op'** modifies register R_j.

However, it is impossible to establish a general rule for two different operations **op** and **op'**. Therefore, we will search for a relative rule for every type of operation.

3.2.1 Addition

Example: Let us suppose that registers R_i and R_j are defined in the state S_k by the following operations:

R_i := R_i + 16;
R_j := R_j + 2;

We also suppose that registers R_i and R_j possess 8 and 5 bits respectively, and they are partially equivalent in the previous states. We can represent registers R_i and R_j as well as operations as shown in Fig. 5.

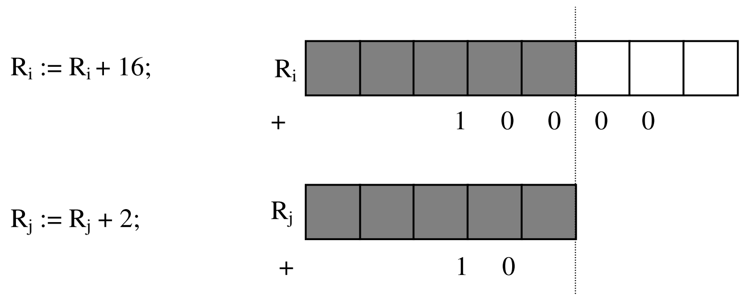


Fig. 5: An example of the partial equivalence with addition operation

We remark that everything that is on the left of the dotted vertical line is identical for the two registers. Although expressions of operations that define registers R_i and R_j are different, the registers are partially equivalent in the state S_k. Adding 16 to register R_i corresponds to adding 2 to register R_j.

Rule 1:
If the following operations are scheduled in a same state S_k:

R_i := R_i + x;
R_j := R_j + y;

then *registers R_i and R_j are partially equivalent in the state S_k if $x = y.2^m$, where m is the difference of bit widths of the two registers ($m = w_i - w_j$).*

If we add a constant that is a multiple of 2^m to the content of register R_i, then there is no carry that passes from the m^{th} bit of register R_i to the part of register R_i identical to the content of register R_j.

Examples:
Let us assume that registers R_i and R_j have the following bit widths $w_i = 8$ and $w_j = 6$, ($m = w_i - w_j = 2$). If registers R_i and R_j are defined by the following operations:

1. R_i := R_i + 8;
   R_j := R_j + 2;
   Then, registers R_i and R_j are partially equivalent, since $8 = 2.2^2 = 2.2^m$.

2. $R_i := R_i + 5$;
   $R_j := R_j + 1$;
   Then, registers R_i and R_j are not partially equivalent, since $5 \neq 1.2^2$.

Remark:
Unlike the operation of addition, it is impossible to obtain a general rule for the operations of subtraction ($R_i := R_i$ - x; $R_j := R_j$ - y). It is necessary to consider two possible cases for each operation, (R_i >x and R_i <x) for the first operation and (R_j >y and R_j <y) for the second operation. In general, we can have the two cases in a same state S_k, since the state graph can be cyclic and the content of the register R_i (or R_j) can be changed at every passage by the state S_k. For this reason, we will not study the partial equivalence for the operation of subtraction.

3.2.2 Multiplication

Let us suppose that the following operations, that define registers R_i and R_j, are scheduled in a same state S_k:

$R_i := R_i * x$;
$R_j := R_j * y$;

Rule 2:
The two registers R_i and R_j are partially equivalent in the state S_k if $R_i = R_j \,.\, 2^m$ and $x = y$, where m is an integer.

Indeed, so that the operation (R_j * x) modifies the part of the register R_i identical to the content of the register R_j in the same way as the operation (R_j * y) modifies the register R_j, it is necessary that the sum of the partial products does not give a carry to add to the part of register R_i identical to register R_j.

Example:
Let < R_k > denotes the content of a register R_k.
If < R_i > = 16, < R_j > = 2 and the two registers are defined by the following operations in a state S_n:

$R_i := R_i * 3$;
$R_j := R_j * 3$;

Then, the registers R_i and R_j are partially equivalent in the state S_n. The content of register R_j is included in the content of register R_i as shown in Fig. 6.

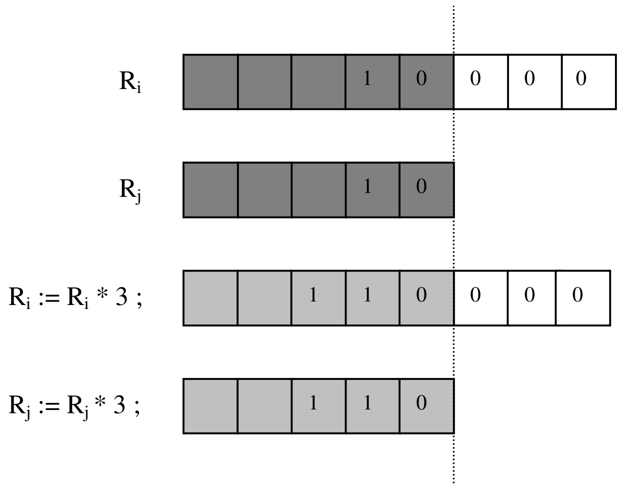


Fig. 6: An example of the partial equivalence with multiplication operation

3.2.3 Right Shift

Rule 3:
Since, the content of register R_j is included in the one for register R_i, *the registers R_i and R_j are partially equivalent in the state S_k for n operations of shift, such as $n < w_j$, where w_j is the bit width of the register R_j.*

Example: division by 2.

If $< R_i > = 16$, $< R_j > = 2$ and the two registers R_i and R_j are defined by the following operations in a state S_n:

$R_i := R_i / 2;$

$R_j := R_j / 2;$

Then, the registers R_i and R_j are partially equivalent in the state S_n, since the content of register R_i is included in the content of register R_i.

3.3 Third Case

The part of the register R_i identical to the register R_j is somewhere between the corresponding positions to the first and the last cases (Fig. 4(c)).

Since, we can add some arbitrary bits on the left of register R_j, we recover the second case while adding on the left of register R_j the corresponding part of register R_i. So the third case amounts to the second case.

3.4 Algorithm

The partial equivalence analysis is described by the **Algorithm 2**. It is done in the same way as the total equivalence analysis. They differ by the type of equivalence relations to establish between registers. The complexity of this algorithm is $O(n\_o.n\_s.r^2)$, where n_o is the number of operations, n_s is the number of states and r is the number of registers.

Algorithm 2

For every type of operations, **do**

1. **For** every state of the state graph establish the equivalence relations between registers defined by operations scheduled in the current state,
2. Construct the equivalence table,
3. The equivalence table will be treated from the left to the right,
    a. Set i = 1,
    b. Determine equivalence classes, as:
       $C_i = \{R_i\} \cup \{R_j / T_{ij} = 1; j = i+1, i+2,..., r\}$
       where T_{ij} is the value in the cell (i, j) of the equivalence table and r is the number of registers,
    c. Point all elements of C_i,
    d. Increment i, **if** i = r **then** stop, **else** continue,
    e. **If** R_i is pointed **then** return to step (d), **else** return to step (b).

4.0 IMPACT ON INTERCONNECTION COST

We will focus our discussion on the interconnection between functional units and registers. As stated earlier, the register merging can have a direct impact on interconnection cost. Indeed, it can cause additional data transfers which require additional interconnection elements. It especially occurs if the register merging is based on the utility phase analysis where the contents of the registers to be merged are not necessarily the same, as shown in Fig. 7. Since, the two operations of addition are scheduled in different states S_i and S_j, they can be bound to a same functional unit fu (see Fig. 7(b)). If the registers R_{11} and R_{21} (the registers R_{12} and R_{22} respectively) have disjoint utility phases, they can be merged into a single register. Fig. 7(b) and Fig. 7(c) show the Register-Transfer Logic (RTL) structure before and after the register merging respectively. We remark that the RTL structure after merging has less registers but at the expense of two added connections. However, if the registers to be merged have the same content, e.g. if they are totally equivalent, then the register merging in this case does not require additional interconnection elements but unlike, it allows immediate saving of registers, buses and functional units, as illustrated by the example in the Fig. 8. Since the two operations of addition are scheduled in a same state S_k, they are bound to two functional units fu_1 and fu_2. The registers R_1 and R_2 are equivalent in the state S_k if the registers R_{11} and R_{21} on the one hand and the registers R_{12} and R_{22} on the other hand are equivalent (Theorem 1). If all these pairs of registers are equivalent in any state of the design, then these registers can be merged. If we compare the RTL structure before merging (Fig. 8(b)) with the RTL structure after merging (Fig. 8(c)), we remark that the latter one has less registers, functional units and interconnections than the former one. Thus, the register merging based on totally equivalence analysis does not only reduce the number of registers but also reduces the interconnection cost and the number of functional units previously allocated. This can result in a lower cost implementation. Similarly, the register merging based on the partial equivalence analysis can also reduce the implementation cost of a design as indicated in Fig. 9. We assume that the registers R_i and R_j have 8 and 5 bits respectively and they are partially equivalent in any state. After the register merging, the two operations can be implemented by the sub-circuit required for the execution of the first one. The result of the operation $R_j = R_j + 2$, e.g. the content of register R_j can be extracted from the content of register R_i. Fig. 9(c) shows the necessary interconnect at the output port of register R_i in the third case (Fig. 4(c)). Consequently, the register optimisation by equivalence analysis leads to a lower cost implementation of a design. In addition, it improves the speed of the digital systems since this parameter depends on the number of the interconnection elements.

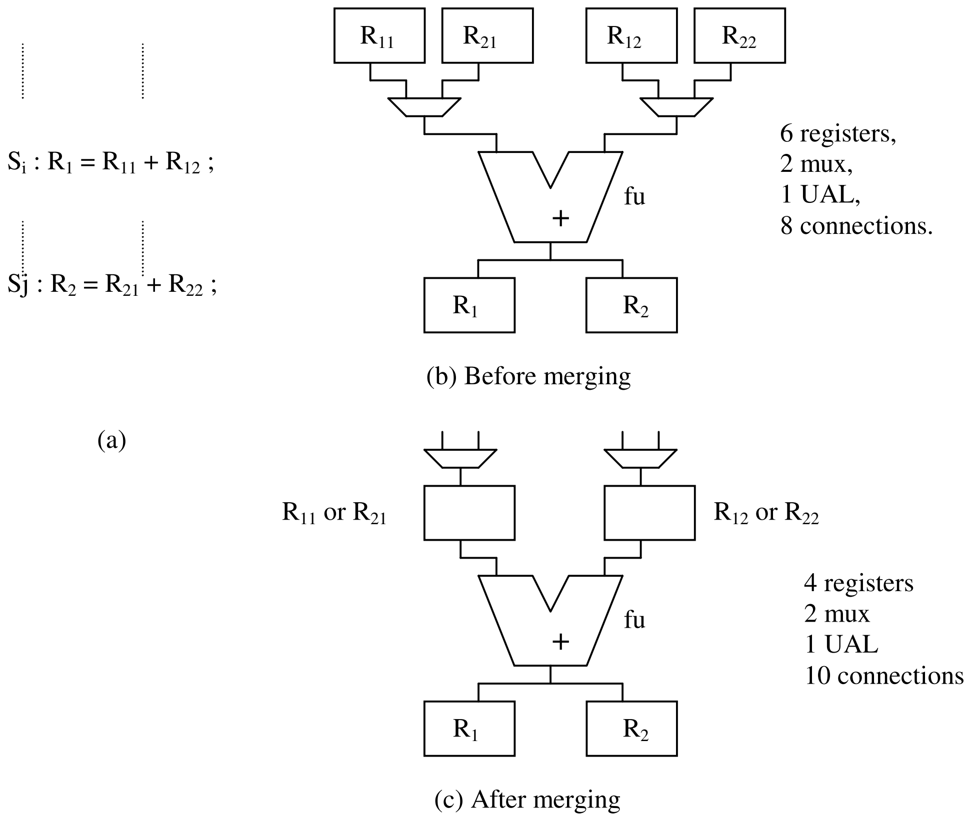


Fig. 7: Register merging based on utility phase analysis

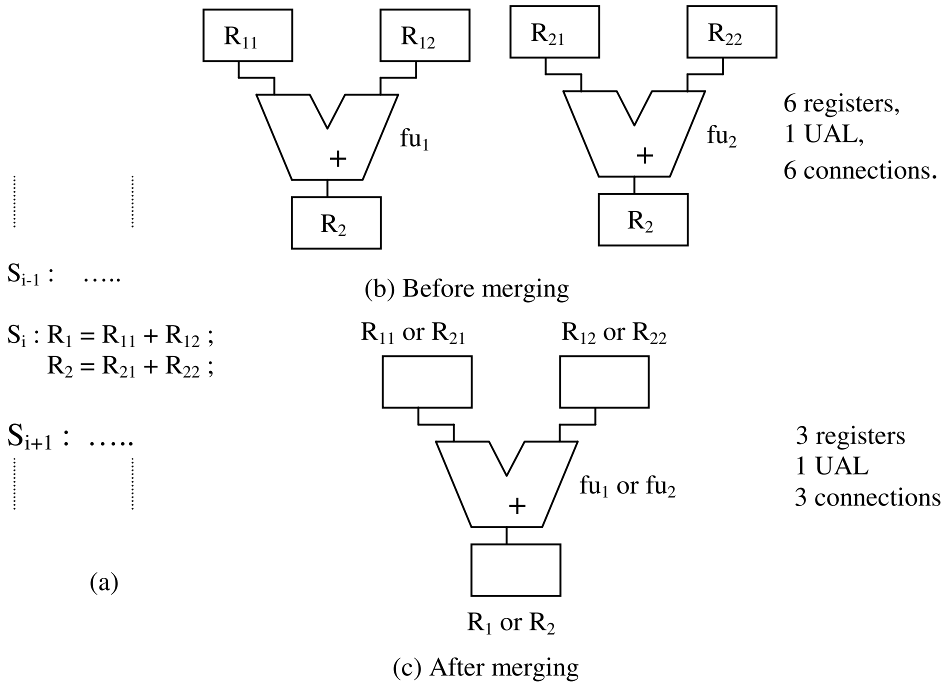


Fig. 8: Register merging based on total equivalence analysis

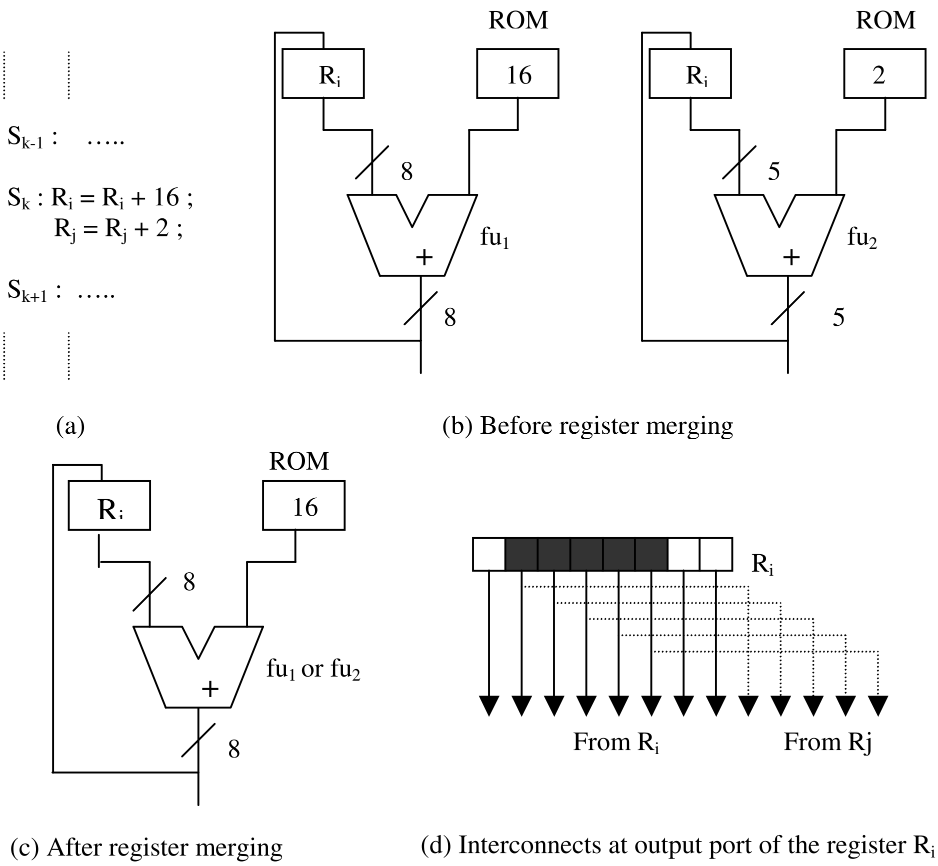


Fig. 9: Register merging based on partial equivalence analysis

5.0 CONCLUSION

In this paper, we have proposed a novel register optimisation method. The method is based on the equivalence analysis between registers before hand allocated by a conventional register allocation algorithm. Our approach is a post-processing step that allows to go beyond minimisation by existing approaches based on the lifetime analysis. It reduces the implementation cost of a design at several levels. It allows to optimise the number of registers and functional units previously allocated as well as the interconnections.

We will extend the partial equivalence analysis between registers defined by operations having general forms.

APPENDIX

Our approach is applicable to scheduled behavioral descriptions with functional unit assignment information, that we represent by state graphs.

A *State Graph* SG = (S, E_S) is a directed graph possibly cyclic. Any node $S_i \in S$ represents a state and any unidirectional edge $e_{ij} = (S_i, S_j) \in E_S$ represents a state transition from the state S_i to the state S_j.

The state graph includes information on both control and data flows, and on the schedule. Each state of the SG is annoted by operations scheduled in this state.

Since we assume that the register allocation is done previously, then the operations manipulate registers.

A register is said to be *defined* in a state if there exists an operation scheduled in this state that can possibly modify its content.

A register is said to be *used* in a state if it appears as operand in the expression of a arithmetic or logic operation scheduled in this state.

A register is said to be *useful* in a state, if it contains the value of a variable that might be used later. A register is useful from the time when it is first written until the time that its content is last read.

Two registers are said to be *compatible* if they are not useful simultaneously, e.g. if their utility phases do not overlap.

Two registers are said to be *totally equivalent* if they have the same content in any state of a design.

Two registers are said to be *partially equivalent* if the content of one register is included in the content of the other register in any state of a design.

A *source operation* of a register is the operation whose output operand should be bound to this register.

A *destination operation* of a register is an operation whose one of its input operands has been bound to this register.

ACKNOWLEDGMENT

Comments and suggestions by Professor Forrest Brewer from the Department of Electrical and Computer Engineering of the University of California at Santa Barbara, are gratefully acknowledged.

REFERENCES

[1] M. C. McFarland, A. C. Parker, R. Camposano, "Tutorial on High-Level Synthesis", in *Proc. of the 25th Design Automation Conference*, July 1988, pp. 330-336.

[2] S. Devadas and A. R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis". *IEEE Transaction on CAD*, Vol. 8, No. 7, July 1980, pp. 768-781.

[3] C.-J. Tseng and D. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems". *IEEE Transaction on CAD*, Vol. 5, No. 3, 1986, pp. 379-395.

[4] P. G. Paulin and J. P. Knight, "Force Directed Scheduling for Behavioral Synthesis of ASICs". *IEEE Transaction on CAD*, Vol. 8, No. 6, 1989, pp. 661-679.

[5] K. Kurdahi and A. Parker, "REAL: A Program for Register Allocation", in *Proc. 24th Design Automation Conf.*, 1987, pp. 210-215.

[6] C. -Y. Huang, Y. -S. Chen, Y. -L. Lin, and Y. -C. Hsu, "Data Path Allocation Based on Bipartite Matching", in *Proc. 27th Design Automation Conference*, 1990, pp. 499-503.

[7] L. Stok, "Transfer Free Register Allocation in Cyclic Data Flow Graphs", in *Proc. European Conference on Design Automation*, Brussels, March 1992, pp. 181-186.

[8] L. Stok and R. Van den Born, "Easy: Multiprocessor Architecture Optimization", in *Proc. International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, Grenoble, 1988, pp. 313-328.

[9] C. Park, T. Kim, and C. L. Liu, "Register Allocation – A Hierarchical Reduction Approach". *Journal of VLSI Signal Processing 9*, 1998, pp. 269-285.

[10] J. E. Hopcroft and J. D. Ullman, *Automata Theory, Languages, and Computation*. Reading, MY, Addison-Wesley, 1977.

[11] Z. Kohavi, *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.

BIOGRAPHY

Mohamed Fettach received his DES (equivalent to M.S) in electronic engineering from University Hassan II, Morocco. He is currently an Assistant Professor at the Faculty of Sciences Ben M'sik, Casablanca, Morocco. He is a PhD candidate in electronic engineering. His research interests include computer-aided design of electronic systems and high-level synthesis.

Lahcen Elarroum received his DES (equivalent to M.S) in electronic engineering from University Hassan II, Morocco. He is currently an Assistant Professor at the Faculty of Sciences Ben M'sik, Casablanca, Morocco. His research interests include logic synthesis, optimisation and high-level synthesis.

Abdellatif Hamdoun received his Diploma Engineering degree in electrical engineering in 1979 and his PhD degree in digital electronic engineering from the Technical University "Fridiriciana", Karlsruhe, Germany. He is currently a Professor at the Faculty of Sciences Ben M'sik, Casablanca, Morocco. His research interests include high-level synthesis and logic design. He is a member of IEEE.