IMPLEMENTATION OF CLOCK GATING LOGIC BY MATCHING FACTORED FORMS

A.kirankumari[1], P.P. Nagaraja Rao[2]
[1]*M. Tech scholar in VLSI System Design, Department of ECE, Sri Venkatesa Perumal College of Engineering & Technology, Puttur, Chittoor (dt), A.P.*
[2]*Associate Professor & HOD, Department of ECE, Sri Venkatesa Perumal College of Engineering & Technology, Puttur, Chittoor (dt), A.P.*

***Abstract* —** Clock gating is one among the most widespread circuit technique to scale back power consumption. Clock gating is sometimes done at the register transfer level (RTL). Automatic synthesis of clock gating in gate level has been less explored, however it's certainly additional convenient to designers.Clock gating consists of 2 steps: extraction of gating conditions by merging gating conditions of individual flip-flops, implementation of the gating conditions with minimum quantity of further gates.In this paper,We show a way to do factored form matching, within which gating operates in factored kinds ar matched, as way as possible, with factored kinds of the mathematician functions of existing combinable nodes within the circuit; further gates are then introduced, however just for the portion of gating functions that don't seem to be matched. sturdy matching identifies matches that ar explicitly gift within the factored forms, and weak matching seeks matches that ar inexplicit the logic and so are tougher to get.

***Keywords*: Clock gating, gating function, factored form.**

1.INTRODUCTION

Clock gating is a well understood power optimization technique employed in both ASIC and FPGA designs to eliminate unnecessary switching activity. This methodnusually requires the designers to add a small amount of logic to their RTL code to disable or deselect unnecessarily active sequential elements, e.g. registers.

The condition in which clock is gated, called a gating function , is typically specified by designers during RTL design stage; the logic for this condition is added by the designers. A prime example is a load-enable register, illustrated in Figure 1(a). When EN is 0, the register keeps the current value of Q; otherwise D is loaded at the next clock edge. A gating function generates EN in this case.

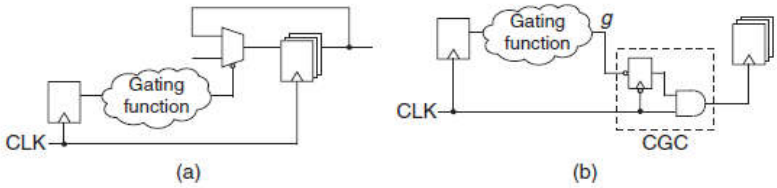


Fig. 1. Load-enable registers

(a) before and (b) after clock gating

Clock gating can be applied by using a circuit shown in Figure 1(b), which is functionally equivalent to Figure 1(a). The multiplexer is removed and the clock signal itself is gated with the EN signal. A circuitry within the dotted box is called a clock gating cell (CGC); a latch is needed to remove any glitches in EN. Relative to the case in Figure 1, clock gating strategies can be considerably more sophisticated in real circuits and generally involve analysis of when registers may be kept idle and subsequent synthesis of clock gating (EN) signals which is determined by the designers.

RTL clock gating has two significant limitations: 1) the designer has to provide a gating function and 2) registers whose gating functions are not specified are left ungated. One way to resolve these problems is to take each ungated register, connect its input and output to an exclusive-NOR gate, and use the output of that gate as a gating function of register However, this technique can only be applied when the register input arrives sufficiently early so that a delay through the exclusive-NOR gate and the CGC can be tolerated.

In this paper, we propose a new technique to simplify gating functions. The idea is to use the existing logic as far as possible while the gating functions are synthesized. This is achieved by matching factored forms of the gating functions with those of existing logic nodes, thus we call this technique as factored form matching. We will present two matching methods: 1) strong matching (SM), which looks for matches that are explicitly apparent in an expression of the circuit logic and 2) weak matching (WM), which seeks matches that are logically present but cannot be found by inspection. Experiments on test circuits show reductions in the area of gating functions between 11% and 39%, with an average of 24%. We contrast these results with those from Boolean division in which, on average, 8% reduction is observed. The rest of this paper is organized as follows. Factored form matching is introduced in Section 2, and the matching algorithm is presented and analyzed. In Section 3, we present our experimental results, and, finally, a conclusion is summarized in Section 4.

2. FACTORED FORM MATCHING

If we have a set of gating functions {F1, F2, . . .}, such that each Fi enables (Fi = 0) and disables (Fi = 1) the clock to a set of flip-flops. Each gating function can be represented in a factored form, or equivalently as a factoring tree, which we will assume to be binary. Our task is to find some existing logic of a combinational circuit, which, together with a few extra gates, implements a gating function. Simultaneously, we have to maximize the proportion of the gating function that is provided by existing logic, so as to minimize the number of extra gates. Because the existing logic can also be represented as a factoring tree, the problem can be recast as that of finding the parts of a factoring tree of a gating function that can be replaced by factoring trees taken from the existing logic; we call this process as factored form matching, which consists of SM and WM.

An overview of factored form matching is shown in Fig. 2, in which it is performed for each individual gating function *Fi* or *F* separately, so as to simplify the notation. A factoring tree of *F*, denoted by *TF*, is first obtained (L1) by factoring , which is performed by recursive algebraic division using a kernel of *F* as a divisor.1 Matching is then performed in two steps: 1) SM (L3–L9) and 2) WM (L10–L15).

Algorithm *Factored_Form_Matching* (F)
L1 $T_F \leftarrow$ *Factoring* (F)
L2 $\mathbb{L}_{sm} \leftarrow \emptyset$, $\mathbb{L}_{wm} \leftarrow \emptyset$
L3 **foreach** internal node n **do**
L4 **if** n contains literals not in F **then** skip n
L5 $T_n \leftarrow$ *Factoring* (n)
L6 $\mathcal{M}_n \leftarrow$ *Strong_Match* (T_F, T_n)
L7 **if** $\mathcal{M}_n \neq \emptyset$ **then**
L8 **if** *Containment* $(\mathbb{L}_{sm}, \mathcal{M}_n) \neq$ yes **then**
L9 $\mathbb{L}_{sm} \leftarrow \mathbb{L}_{sm} + (n, \mathcal{M}_n)$
L10 **foreach** unskipped internal node n **do**
L11 $\mathcal{M}_n \leftarrow$ *Weak_Match* (T_F, T_n)
L12 **if** $\mathcal{M}_n \neq \emptyset$ **then**
L13 **if** *Containment* $(\mathbb{L}_{sm}, \mathcal{M}_n) =$ yes **then** skip n
L14 **else if** *Containment* $(\mathbb{L}_{wm}, \mathcal{M}_n) =$ yes **then** skip n
L15 **else** $\mathbb{L}_{wm} \leftarrow \mathbb{L}_{wm} + (n, \mathcal{M}_n)$

Function *Containment* $(\mathbb{L}, \mathcal{M})$
L16 **foreach** $(n, \mathcal{M}_n) \in \mathbb{L}$ **do**
L17 **foreach** pair (g, g'), where $g \in \mathcal{M}_n$ and $g' \in \mathcal{M}$ **do**
L18 **if** $g \subseteq g'$ **then** $\mathcal{M}_n \leftarrow \mathcal{M}_n - g$
L19 **else if** $g' \subset g$ **then** $\mathcal{M} \leftarrow \mathcal{M} - g'$
L20 **if** $\mathcal{M}_n = \emptyset$ **then** $\mathbb{L} \leftarrow \mathbb{L} - (n, \mathcal{M}_n)$
L21 **if** $\mathcal{M} = \emptyset$ **then** return yes

Fig. 2.An algorithm for factored form matching:

Where T is the factoring tree,Mn is the set of subtrees (in SM) or sets of vertices (in WM) of TF that are to be matched with Tn, and L is the list of the resulting pairs (n,Mn).

A.Strong Matching

Consider the factoring tree of a gating function F shown in Fig. 3(a), and the factoring trees of three of the existing internal nodes shown in (b)–(d). The subtree of F marked N1 and the tree n1 are structured in different ways, but they actually represent the same Boolean expression abc+acd+bd; we say that N1 and n1 are equivalent . N2 and n2 are also equivalent; in fact, these trees have the same structure, except for the ordering of children; we say that such trees are syntactically equivalent .

Finally, N3 and n3 are syntactically equivalent, and children are ordered in the same way; rather obviously, we call these identical factored forms. Any of these three types of equivalence are said to provide a strong match in this paper.

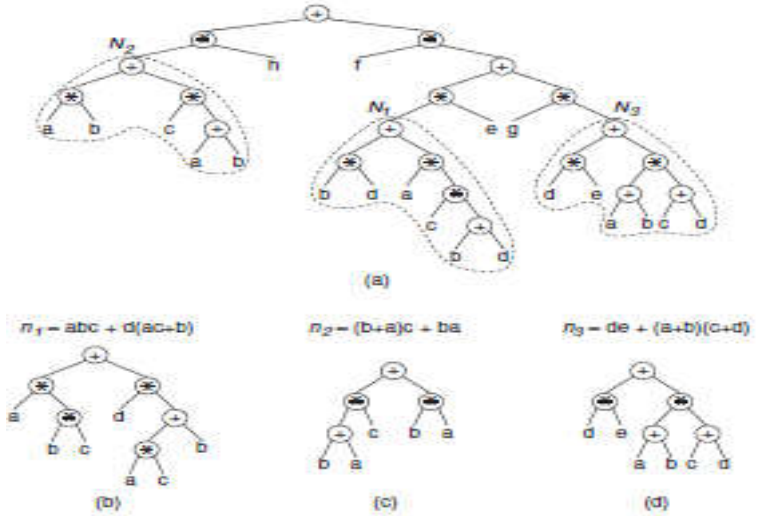


Fig. 3. Factoring trees of (a) a gating function F: of internal nodes (b) n1, which is equivalent to subtree N1, (c) n2, which is syntactically equivalent to subtree N2, and (d) n3, which is identical to subtree N3.

Finding the most general strong match involves the detection of equivalent factored forms. This, however, is a problem of combinational equivalence checking, which is proven to be co-NP-complete.2 Thus, we focus on finding syntactically equivalent and identical factored forms in this paper.

I)Detection of Syntactically Equivalent and Identical Factored Forms

Pseudocode for the algorithm that we use to detect syntactically equivalent factored forms is shown in Fig. 4. T is a large factoring tree (similar to F in Fig. 3), which may contain one or more subtrees that are syntactically equivalent to a smaller tree t (such as n1 in Fig. 3). For each vertex v in T , we check whether the size of subtree rooted at v, denoted by Tv , is the same as that of t (L4); note that, we only consider v that is the same as the root of t (L2) by the definition of syntactical equivalence. Then, we label both Tv and t, level by level, starting at the deepest level.

```
Algorithm Syntactically_Equivalent (T, t)
L1    M ← ∅
L2    foreach vertex v ∈ T, where v = root of t do
L3        T_v ← subtree rooted at v
L4        if |T_v| = |t| then
L5            foreach level i of t and T_v from deepest do
L6                V_v ← vertices at level i of T_v
L7                V_t ← vertices at level i of t
L8                if Labeling (V_v, V_t) = fail then skip v
L9            M ← M + T_v
L10   Return M
```

Fig4. Algorithm to detect syntactically equivalent factored forms.

II)SM Algorithm

The procedure Strong_Match called by the algorithm in Fig. 2 itself calls either Syntactically Equivalent or Identical, depending on how the factoring trees of F and the existing internal nodes ni are obtained. We might apply quick factoring to the ni terms to reduce runtime, and good factoring to F to obtain a smaller tree.3 Because different factoring methods may use different divisors, they can yield syntactically equivalent as well as identical factored forms. On the other hand, if quick factoring is applied to both F and the ni terms, we only need to look for identical factored forms, which takes less time. It can readily be shown that the complexity of Syntactically_Equivalent is O(V log V) and that of Identical is O(V), where V is the number of vertices.

B.Weak Matching

In searching for a strong match between a subtree of a gating function and the factoring tree corresponding to an existing logic node, we only look for syntactically equivalent and identical factored forms, and so the two trees must have the same structure. This is a rigid requirement, especially for large trees. Consider Fig.5, in which no subtree of F strongly matches n. The vertices of F within the dotted circles collectively, however, constitute the same expression as n, because

$$F = aeg(d + f(b + c)) + abh + defg$$
$$= n(aeg) + abh + defg.$$

All the nodes n that are not skipped are checked for weak match (L10 of Fig. 2).

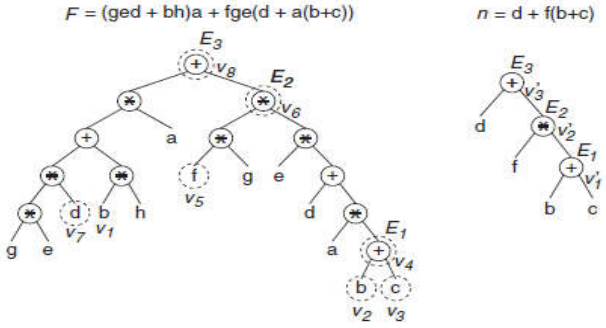


Fig5. Weak match of a gating function F with an internal node n.

I)WM Algorithm: WM algorithm (denoted by Weak_Match) pseudocode is shown in Fig6. T is a large tree (like F in Fig.5), which may contain one or more sets of vertices that are weakly matched to a smaller tree t (similar to n in Fig. 5). Initially, for each vertex v in T and t, we assign its subexpression Ev and factor F(v) to v and 1, respectively (L2), subexpression of each leaf becomes itself (such as Ev5 = f) and factors for all vertices are initialized to one.

Now, we traverse the tree *t* in postorder. For each vertex *v* visited during the traverse, the expression using subexpressions of its children and operation of itself is assigned to *Ev_* (like *Ev* _2= *E*2 = *f E*1); note that, we only consider *v_* that is not a leaf (L3–L6). *Tv_* is the set of three tuples *(vl, vr ,* and *vA)*, all vertices of which are in *T* ; in each tuple, *vA* is the LCA of *vl* and *vr* and is equal to *v_*, and *vl* and *vr* have the same subexpression as left and right child of *v_*, respectively (L23–L26).

```
Algorithm Weak_Match (T, t)
L1   M ← ∅, Ψ(t) ← Postorder(t)
L2   foreach vertex v ∈ T ∪ t do E_v ← v, F(v) ← 1
L3   foreach vertex v' ∈ Ψ(t), where v' is not a leaf do
L4       (v'_l, v'_r) ← children of v'
L5       if v' = × then E_v' ← E_v'_l × E_v'_r
L6       else E_v' ← E_v'_l + E_v'_r
L7       M_new ← ∅, M' ← ∅, T_v' ← Candidate(T, v', v'_l, v'_r)
L8       foreach tuple (v_l, v_r, v_A) ∈ T_v' do
L9           F(v_l) ← Factor(v_l, v_A) × F(v_l)
L10          F(v_r) ← Factor(v_r, v_A) × F(v_r)
L11          if v' = × or (v' = + and F(v_l) = F(v_r)) then
L12              E_v_A ← E_v', M' ← M' + {v_l, v_r}
L13              if v' = × then F(v_A) ← F(v_l) × F(v_r)
L14              else F(v_A) ← F(v_l)
L15      if M' = ∅ then Return ∅
L16      foreach pair (V, V'), where V ∈ M and V' ∈ M' do
L17          foreach v ∈ V' do
L18              if v is a leaf and v ∈ V then skip (V, V')
L19              if v is not a leaf and v ∉ V then skip (V, V')
L20          V' ← V' + LCA(V'), M_new ← M_new + {V ∪ V'}
L21      M ← M_new
L22  Return M

Function Candidate (T, v', v'_l, v'_r)
L23  T ← ∅
L24  foreach pair (v_l, v_r) ∈ T, where E_v_l = E_v'_l and E_v_r = E_v'_r do
L25      v_A ← LCA(v_l, v_r), if v_A = v' then T ← T + (v_l, v_r, v_A)
L26  Return T

Function Factor (v_l, v_A)
L27  if v_l = v_A then Return 1
L28  if parent of v_l = × then
L29      Return Factor(parent of v_l, v_A) × expression of sibling of v_l
L30  else Return Factor(parent of v_l, v_A)
```

Fig6.Algorithm to detect weak matches.

For each three-tuple (vl, vr , and vA) in Tv', we find two factors F(vl) and F(vr) by ANDing their initial factors and the expression obtained from Factor (L27–L30). If the check passes, Ev A is assigned to Ev' and F(vA) is initialized either to F(vl) × F(vr) when vA is a × operation or to F(vl) when vA is a + operation, and a set of vl and vr is added to M', which is a set of sets of vertices that are matched to the children of v' (L8–L14). If M' is empty, because no vertex in T matched to v' exists, we return empty set (L15).

Otherwise, we pick a pair (V, V'), which, respectively, are extracted from M and M', one by one;Mis a set of sets of vertices in T matched to the vertices that are already traversed in t. We then, for each vertex v in V',check if v is a leaf and not contained in V or if v is not a leaf and contained in V; if all v satisfy the check condition, we unify V, V', and LCA of V'power 4 and add it to M new by which M will be replaced (L16–L21).

C.Factored Form Matching

L in Fig. 2 is the list of pairs (n,Mn); in each pair n is an internal node5 of a combinational circuit and Mn is a set of subtrees (or sets of vertices) of TF that matches with Tn, which is a factoring tree of n (L5). Two lists Lsm and Lwm are initialized and then updated to track strong and WM,respectively (L2).

We restrict our Boolean manipulation to algebraic to limit computation time; therefore, if a node n contains literals that are not in F, then n is dropped from further matching attempts (L4).

I)Containment in SM: Strong_Match returns a set of subtrees (L6). We can use the function Containment to check whether some of these subtrees are redundant because they are contained in subtrees that are already determined to be strong matches.

Example : Consider Fig. 7, in which the tree n1 is to be matched with subtrees N1 and N2. We set Mn1 = {N1, N2} and Lsm = {(n1,Mn1)} (L9). We continue to determine Mn2 = {N3}. Then, we perform the containment check because N2 ⊂ N3, N2 can be discarded and we update Mn1= {N1} (L18) and Lsm = {(n1,Mn1), (n2,Mn2)}.Similarly, Mn3 = {N4}, and N1 ⊂ N4. Discarding N1 makes Mn1 empty. Thus, (n1,Mn1) can also be discarded from Lsm (L20), and finally Lsm = {(n2,Mn2), (n3,Mn3)} ={(n2, {N3}), (n3, {N4})}.

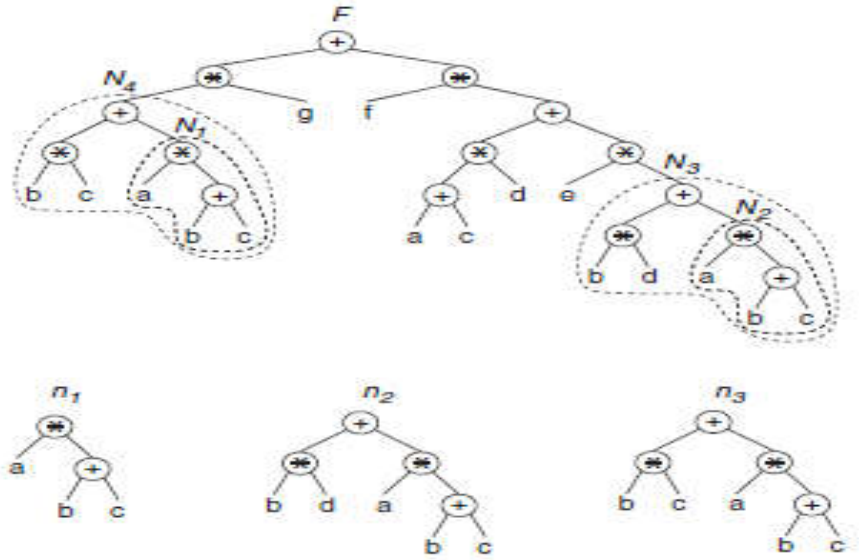


Fig7. Checking for subtree containment in SM.

We might think that the subtrees that are determined as strong matches might have a partial mutual intersection without one being wholly contained in the other; but this never happens, as we show by proving the following proposition.

Proposition 1: Let, ni and n j be strong matches with subtrees Ni and Nj of F, respectively. If Ni ∩ Nj = , then either Ni ⊆ Nj or Nj ⊆ Ni.

II) Containment in WM: Now, we turn our attention to the possibility of nested matches: each weak match is checked to see whether it is contained in or contains other strong or weak matches (L12–L15).

Example: Consider Fig8: n1 and n2 are strong matches with subtrees N1 and N2, respectively. The set of vertices of F that weakly matches with n3 includes N1; thus n1 is dropped from the list of strong matches. The vertices that weakly match with n4 make n3 redundant. The three vertices of F that weakly match with n5 constitute a subset of N2; thus n5 is also redundant. Matches n2 and n4 remain after the containment check.

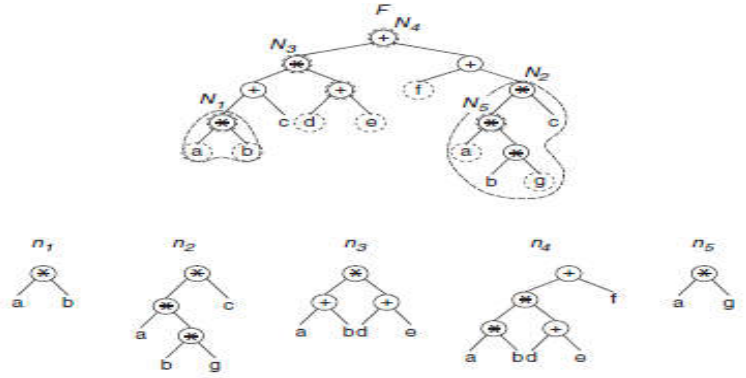


Fig. 8. Finding nested matches: n1 and n2 are strong matches and the remainders are weak matches.

3.EXPERIMENTAL RESULTS

To assess factored form matching (denoted by Matching in Table 1), we implemented two additional methods of simplification: using Boolean division , in which approximation ,in which product terms (in sum-of-products form of a gating function) whose probability is smaller than 0.001 are declared as don't-care set, a new gating function is submitted to two level minimization followed by implementation in multi-level logic. The number of gates to implement gating functions without any simplification, given in the second column of Table 1,serves as a reference to compute the difference of the number of gates (denoted as Diff) of each simplification method.

Division achieves about 10% reduction in the number of gates, but there is wide variation (from 0% to 34%). This is expected since the existence of larger divisor relies on a chance, in particular when algebraic division is applied. Approximation achieves more reduction, but this comes at the cost of reduced gating probability. Average gating probability of gated flip-flops is 0.67 in 'No simplification', while that of Approximation is 0.62; note that Division and Matching do not sacrifice gating probability.

Matching achieves substantial reduction in gate count compared to both division and ap- proximation. The matching takes about 10% more runtime than division method in our current mplementation, ranging from 1 to 340 seconds (the last column of Table 1). The majority of runtime is due to the detection of weak matches as can be expected from its complexity. Weak matches account for only 10% of total number of matches; its contribution in gate count reduc- tion (the second last column of Table1) however is 25%, which demonstrates its importance in the factored form matching.

Table 1: Number of extra gates to implement gating functions. 'No simplification' serves as a reference to compute difference

Circuit	No simplification	Division		Approximation		Matching		
	# Extra gates	# Extra gates	Diff. (%)	# Extra gates	Diff. (%)	# Extra gates	Diff. (%)	Run-time (s)
s1423	78	52	-33.3	54	-30.8	32	-59.0	4.7
s5378	305	282	-7.5	279	-8.5	264	-13.4	12.7
s9234	148	136	-8.1	127	-14.2	121	-18.2	11.3
s13207	284	264	-7.0	260	-8.5	232	-18.3	10.8
s15850	431	414	-3.9	366	-15.1	367	-14.8	63.25
s35932	1401	1401	–	1334	-4.8	1255	-10.4	233.0
s38584	1317	1302	-1.1	1277	-3.0	1215	-7.7	998.7
b04	66	61	-7.6	55	-16.7	47	-28.8	26.8
b07	30	30	–	30	–	21	-30.0	19.5
b12	300	280	-6.7	271	-9.7	236	-21.3	44.3
b17	289	272	-5.9	279	-3.5	246	-14.9	339.9
ac97	67	44	-34.3	37	-44.8	24	-64.2	2.1
i2c	184	168	-8.7	166	-9.8	153	-16.8	14.7
fir	223	211	-5.4	201	-9.9	189	-15.2	12.8
mem_ctrl	611	600	-1.8	512	-16.2	486	-20.5	267.4
ps2	57	41	-28.1	41	-28.1	37	-35.1	22.5
sasc	119	101	-15.1	97	-18.5	91	-23.5	5.5
systemcaes	287	253	-11.8	233	-18.8	226	-21.6	57.7
usbf_top	158	134	-15.2	86	-45.6	81	-48.7	31.8
wb_dma	606	538	-11.2	575	-5.1	516	-14.9	300.0
Average			-10.6		-15.6		-24.9	

6.1. PROPOSED METHOD RESULTS

In this chapter all the simulation results which are done using Xilinx ise 9.1 are shown in below results.

A. SIMULATION RESULTS

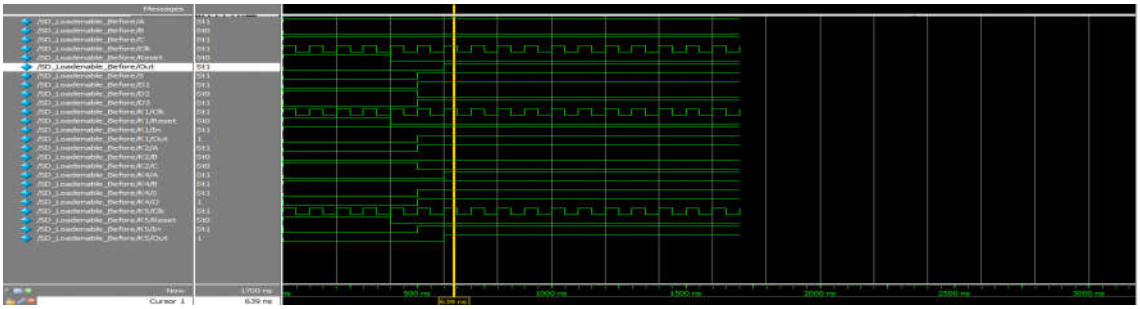


Fig 1: BEFORE CLOCK GATING

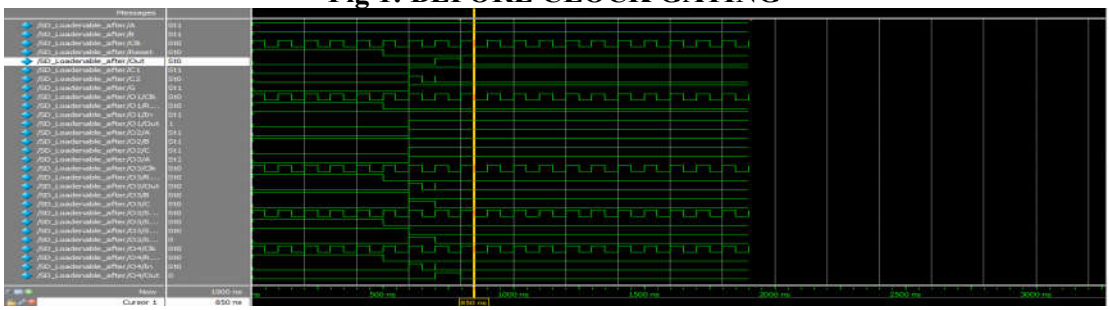


Fig 2: AFTER CLOCK GATING

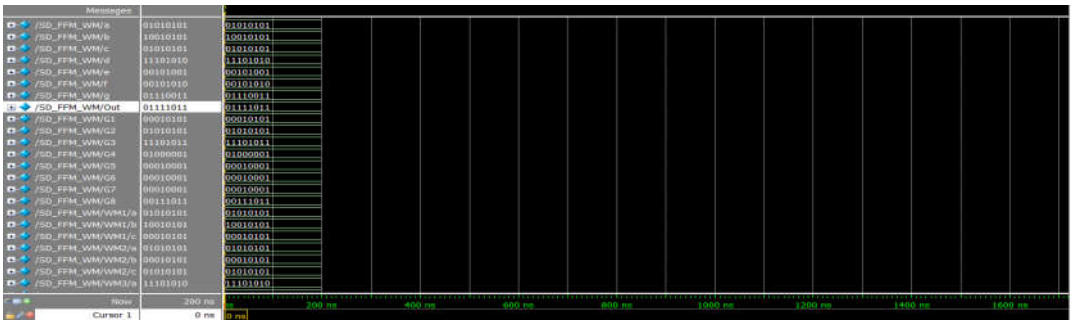


Fig 3: MATCHED FACTORED FORMS: WEAK MATCHING

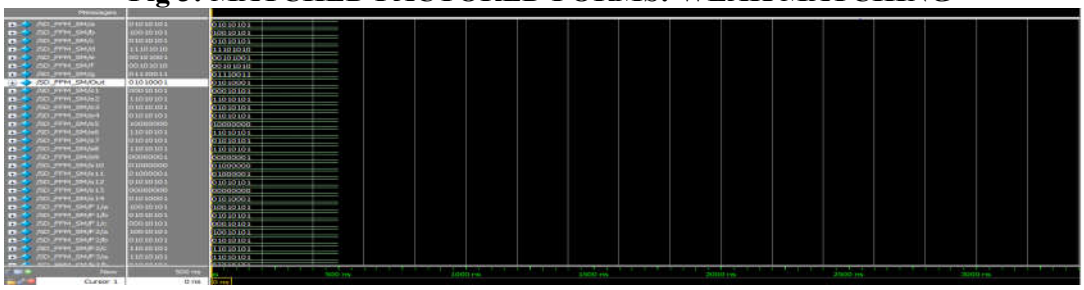


Fig 4: MATCHED FACTORED FORMS: STRONG MATCHING

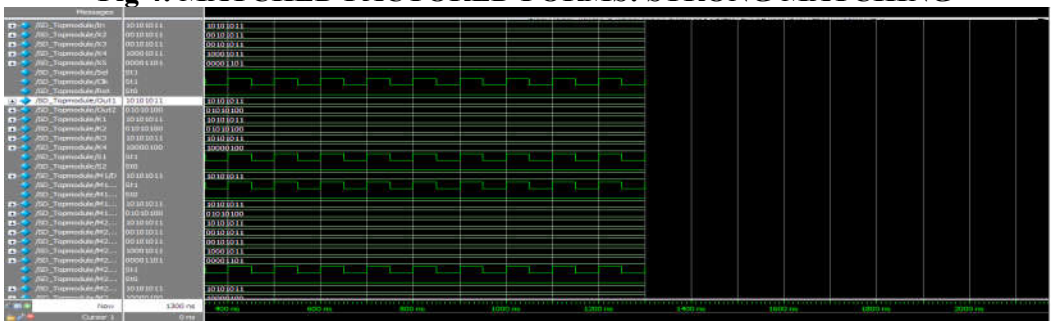
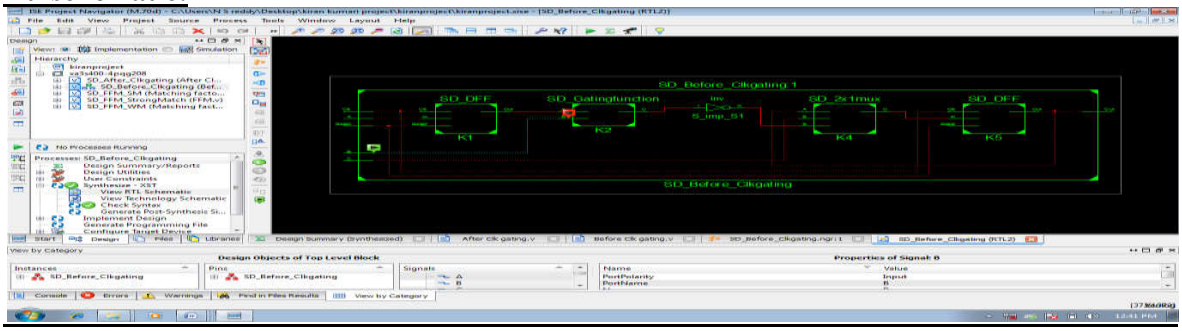


Fig 5: C17 CIRCUIT IMPLEMENTATION

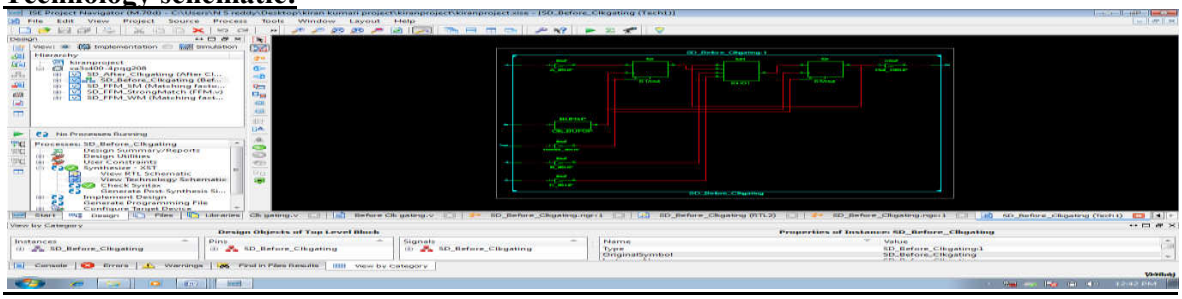
B. SCEMATIC DESIGNS:

1. Before clock gating:

Rtl schematic:

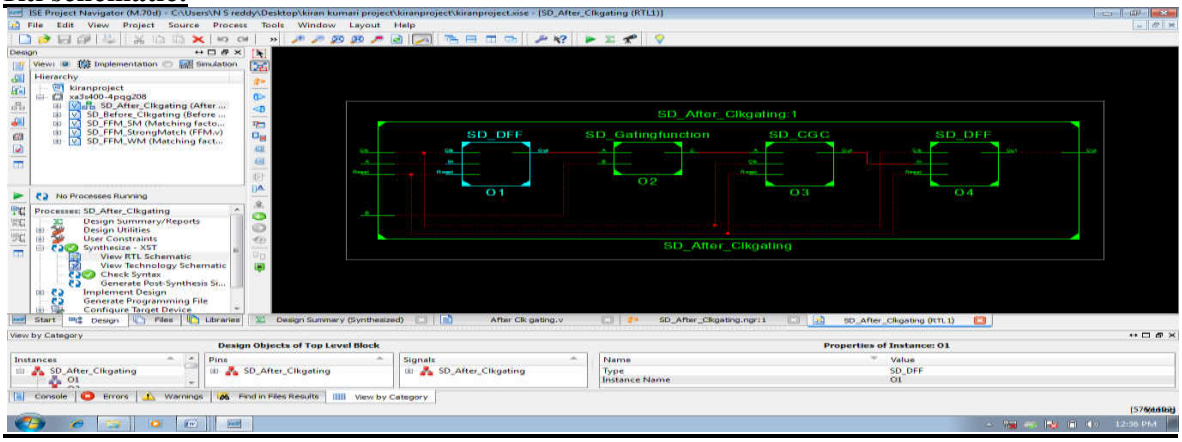


Technology schematic:

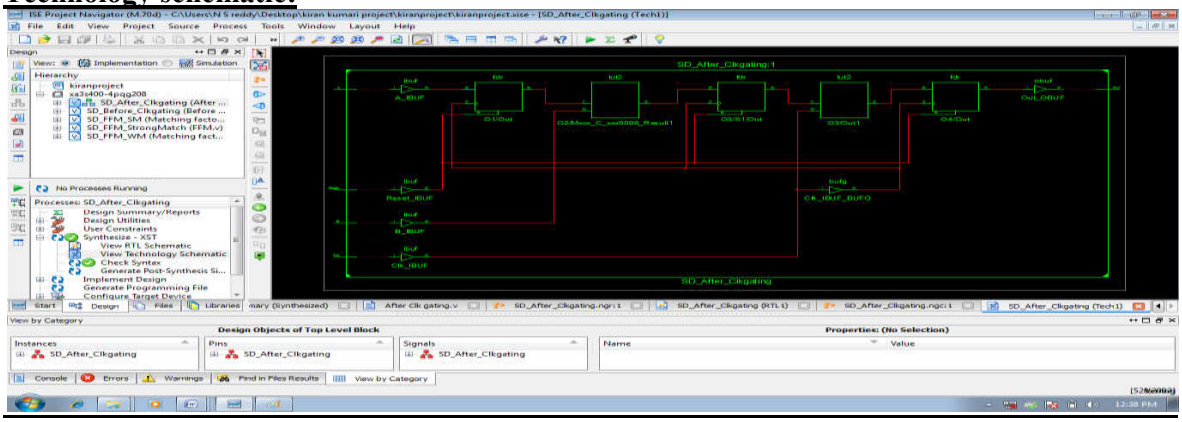


2. After clock gating:

Rtl schematic:

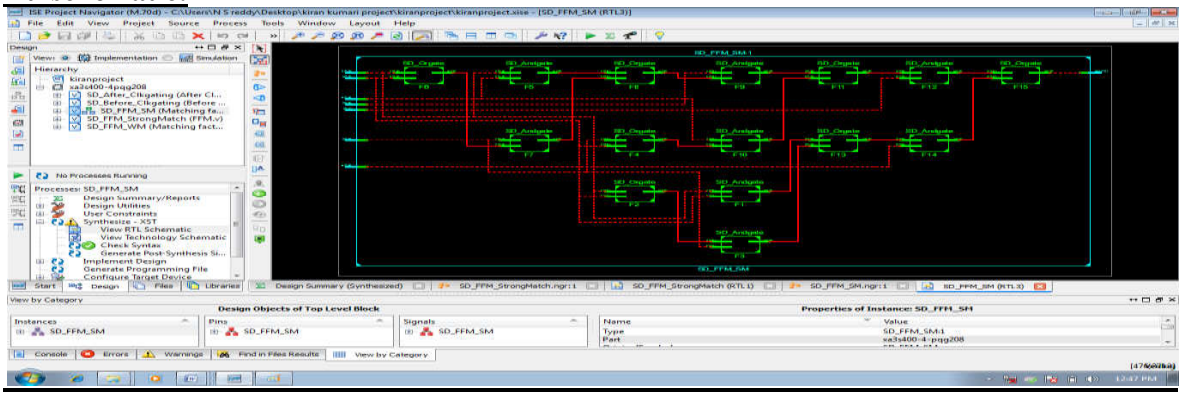


Technology schematic:

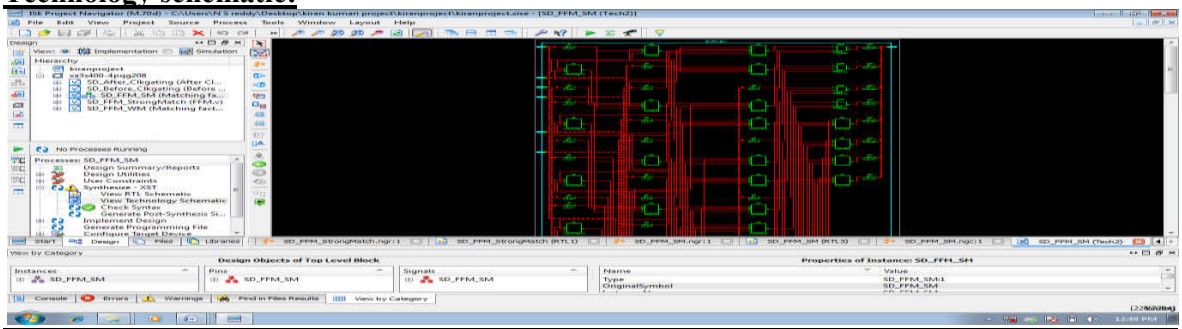


3. Strong matching:

Rtl schematic:

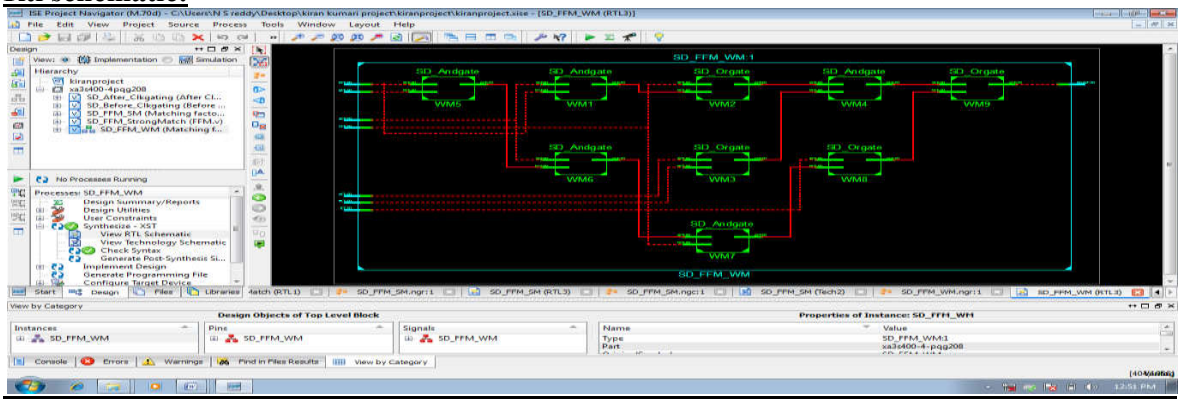


Technology schematic:

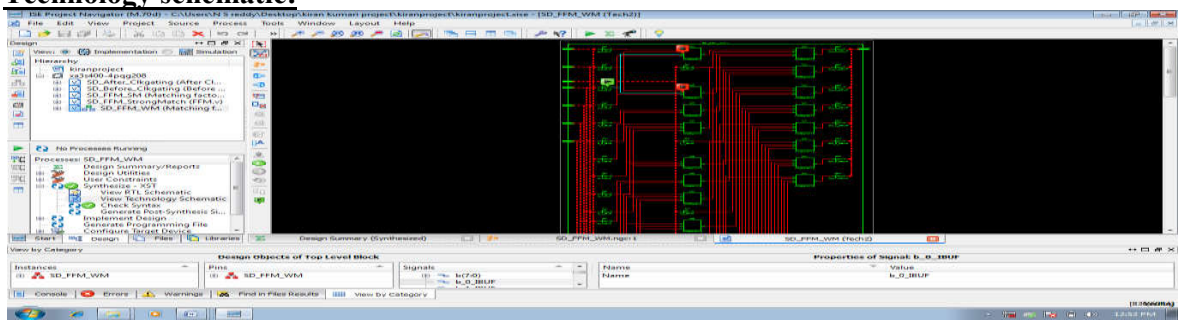


4. Weak matching

Rtl schematic:

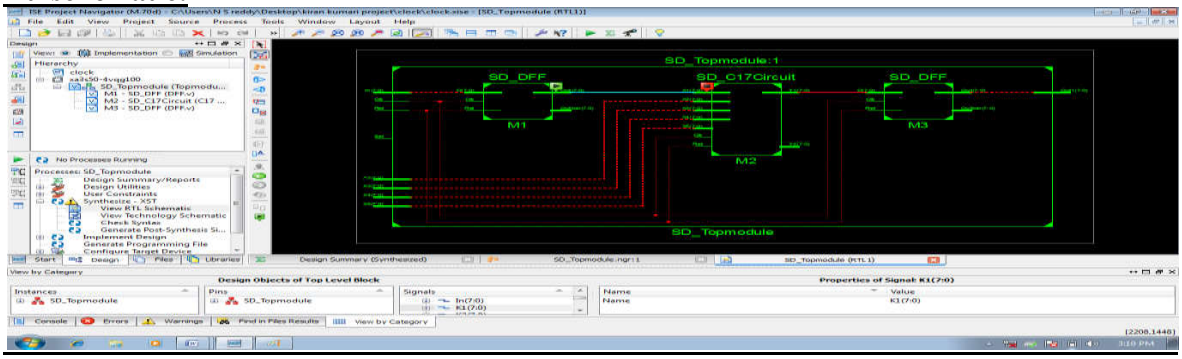


Technology schematic:

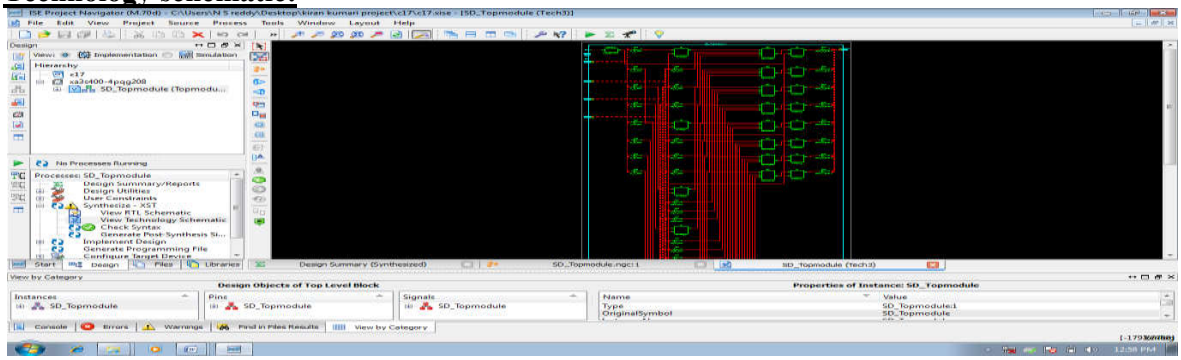


5 c17 circuit:

Rtl schematic:



Technology schematic:



CONCLUSION

Gate-level clock gating synthesis is not yet in main stream use, but is a promising methodology due to its convenience offered to designers. New solutions for two key problems of the synthesis, merge and simplification, have been proposed. The iteration of minimum weight perfect matching formulation yields balanced grouping of flip-flops without too much decrease in average gating probability; this is in an effort to overcome the limitation of greedy approach to merge. Factored form matching has been proposed to utilize existing combinational logic as much as possible, this is an extension of a prior method using Boolean division.

REFERENCES

[1] I. Han and Y. Shin, "Synthesis of clock gating logic through factored form matching," in Proc. Int. Conf. IC Design Tech., Jun. 2012,pp. 1–4.
[2] D. Chinnery and K. Keutzer, Closing the Power Gap Between ASIC & Custom, Norwell, MA, USA: Kluwer, 2007.
[3] S. Unger, "Double-edge-triggered flip-flops," IEEE Trans. Comput.,vol. 30, no. 6, pp. 447–451, Jun. 1981.
[4] R. Pokala, R. Feretich, and R. McGuffin, "Physical synthesis for performance optimization," in Proc. Int. ASIC Conf. Exhibit., Sep. 1992, pp. 34–37.
[5] Power Compiler User Guide, Synopsys, Inc., Mountain View, CA, USA, Dec. 2010.
[6] F. Theeuwen and E. Seelen, "Power reduction through clock gating by symbolic manipulation," in Proc. Symp. Logic Archit. Design, Dec. 1996,pp. 184–191