CS6132 Advanced Logic Synthesis

Final Project Report – Technology Mapping

110062802 呂宸漢

- **Data Structure**

由於這次作業所需要的演算法都與圖論有關，因此我選擇使用 LEDA [1]內的 GRAPH (Parameterized Graphs)作為我的資料結構。這個結構的操作與一般的 graph (Graphs)相同，不過他的每個 node 與 edge 內都可以儲存使用者自訂的 structure，因此在需要儲存多個資訊時較一般的 graph 方便。

每個 node 內我會儲存 name (對應的 gate 的名字)、type (AND、OR、NOT、BUFFER、CONST_0、CONST_1)、level (在 DMIG 中的 level)、label 與 xBar (在 Flow Map 中的 label 與 xBar)。而 edge 上則只記錄 capacity 的數值。

- **Algorithm**

本次作業將 Technology Mapping 問題拆成兩個 stage 完成：第一部份是將原始 multi-input 的圖 decompose 成 two-input 的圖，第二部份才是將 two-input 的圖轉換成 K-input 的 LUT。以下我會分別介紹兩部份的演算法。

1. Decomposition

這個部份我們必須將 multi-input 的圖拆成 two-input 的圖。最簡單的方式就是直接將 multi-input 的 node 替換成由一堆 two-input 的 node 組合而成的樹，這個方法雖然簡單，可是不同的樹狀結構會直接影響到電路的 level，進而影響在第二部分 K-LUT mapping 後的 level。

為了最佳化拆成 two-input 的 level，因此我採用[2]所提出的方法(Decompose Multi-Input Gate, DMIG)，此演算法的 pseudo code 如 **Algorithm 1** 所示。先選擇一顆大於兩個 input 的 node，從這些 input 中選出兩個 level 較小的 input nodes 並新建一顆 node 用來接收這兩個 input node，再將這顆新的 node 接回原本的 node 即可。利用此方法即可在執行數次後將 multi-input 的 node 拆成 two-input 的 node 樹，且可以讓 node 新增的 level 最小。

雖然該演算法並未提及 decompose multi-input node 的順序，不過可以看到在 decompose 時會用到其他 node 的 level，且會用到的 level 都是要拆的 node 的 input，因此我會先用 LEDA 中的 TOPSORT (topological sort)將 node 排成拓樸排序，再一一計算 node 的 level。如果 node 沒有 input node，則將 node

的 level 設為 0；如果 node 只有一個或兩個 input node，則將 node 的 level 設成 input node 中較大的 level + 1；如果 node 有大於兩個 input，則使用此演算法拆成 two-input 再計算 level。

algorithm decompose-multi-input-gate (DMIG)
let $V = input(v) = \{u_1, u_2, \ldots, u_m\}$;
while $|V| > 2$ **do**
let u_i and u_j be the two nodes of V with smallest levels;
introduce a new node x;
$input(x) = \{u_i, u_j\}$;
$level(x) = \max(level(u_i), level(u_j)) + 1$;
$V = (V - \{u_i, u_j\}) \cup \{x\}$;
end-while;
connect the only two nodes left in V to v as its inputs;
return the binary tree $T(v)$ rooted at v;
end-algorithm;

Algorithm 1: Decompose multi-input gate

2. K-LUT Mapping

將 multi-input 的圖拆成 two-input 的圖後，我們必須在 two-input 的圖上找出哪些 node 可以組成 LUT 並讓組合後的 LUT circuit level 越小越好。為了能讓 LUT 可以包含越多的 node 以降低 circuit level，我採用[3]所提出的方法 (Flow Map)，此演算法的 pseudo code 如 **Algorithm 2** 所示。Flow Map 分為 Labeling Phase 與 Mapping Phase 兩大部分。以下我會分別介紹兩部分的目的與實作內容。

■ Labeling Phase

labeling phase 主要是要找每個 node 可以往上包含到哪些 node，而這些 node 合起來的 input 必須小於等於給定的 input 數量。為了可以更有效地尋找哪些 node 可以包在一起，Flow Map 會將 node 做 label 並利用 label 對子圖做一些改動再檢查哪些 node 可以包在一起。

labeling phase 一開始會先將 primary input node 的 label 設為 0，對所有不是 primary input 的 node 做拓樸排序，再以此順序一一對 node 做處理。從拓樸排序選擇一個 target node 後會先建立子圖N_t，這個子圖就是以 target node 往上包含他的 predecessors 直到 source node 的子圖。接著從 target node 的 input node 中找出最大的 label 的數值，將跟最大 label 相同的 node 們與 target node 合併，即可得到子圖N_t'。最後再將N_t'中除了 source 與 target 之外的 node 切成兩個 node，將兩個 node 之間以一個 capacity 等於 1 的 edge 相連，再將其餘的 edge 的 capacity 設為∞，即可得到這個階段最終的子圖N_t''。

```
algorithm Flow Map
    /* phase 1: labeling network */
    for each PI node v do
        l(v) := 0;
    T := list of non-PI nodes in topological order;
    while T is not empty do
        remove the first node t from T;
        construct the network N_t;
        let p = max{l(u): u ∈ input(t)};
        transform N_t into N_t' by collapsing all nodes in N_t with label p into t;
        transform N_t' into N_t'' as follows:
            split every node in {x: x ∈ N_t', x ≠ s, x ≠ t} into two
                and connect them with a bridging edge of capacity 1;
            assign all non-bridging edges capacity ∞;
        compute a cut (X'', X̄'') in N_t'' s.t. e(X'', X̄'') ≤ K
            using the augmenting path algorithm;
        if (X'', X̄'') is not found in N_t'' then
            X̄_t := {t};   l(t) := p + 1;
        else
            induce a cut (X'', X̄'') in N_t from the cut (X'', X̄'') in N_t'';
            X̄_t := X̄;    l(t) := p;
        endif
    endwhile;
    /* phase 2: generate K-LUTs */
    L := list of PO nodes;
    while L contains non-PI nodes do
        take a non-PI node v from L;
        generate a K-LUT v' to implement the function of v
            such that input(v') = input(X̄_v);
        L := (L − {v}) ∪ input(v');
    endwhile;
end-algorithm;
```

Algorithm 2: Flow map

接著就可以利用 MAX_FLOW_T(max flow)尋找子圖N_t''上的 maximum volume min-cut，檢查 cut 是否可以小於等於給定的 K 值。如果 cut 大於 K 值，則將$\overline{X_t}$設成$\{t\}$，並將 target node 的 label 設成最大的 input label + 1；如果 cut 小於等於 K 值，則先取得 flow residual graph，在圖上從 source node 開始往下尋找可以走到的 node，並用子圖N_t的所有 node 扣掉剛剛那些可以走到的 node 就可以得到$\overline{X_t}$，再將 target node 的 label 設成最大的 input label，如此便完成一個 target node 的 labeling。當所有 node 都被 label 完後，就可以結束 labeling phase，繼續往下做 mapping phase。

■ Mapping Phase

mapping phase 主要是利用 labeling phase 所得到的 xBar 轉成 LUT 並計算 LUT 的 logic function，再用 LUT 的 input 往上找其他 LUT，直到所有 node 都有被 LUT 包起來即可結束 mapping phase。

mapping phase 一開始會先建立一個 list *L*儲存所有 primary output node，當*L*中包含除了 primary input node 以外的 node 時，則取出一個 node，將 node 的 xBar 包成 LUT 並計算 LUT 的 function，並將這些 node 的 external input 設成 LUT 的 input，最後將這些 input 加入*L*，重複執行以上動作直到*L*是空的即可得到 mapping 玩的 LUT 與 logic function。

計算 logic function 的方式也很簡單，由於我們知道 LUT input 的數量，因此我們只需要列舉 input 個數的 variable 的 truth table，並走訪 xBar 內的所有 node 即可得到 logic function。為了減少計算複雜度，我會利用 decompose 時的 level 將 xBar 內的 node 排序，此排序相當於取 xBar 的拓樸排序，以此排序走訪 node 即可在只走訪一次的情況下計算出 LUT 的 logic function。

由於在計算 LUT 的 function 時可能會遇到有些 function 的 output 都是 0 或都是 1，此時，我們可以將這 LUT 的 output node 直接設成 CONST_0 或 CONST_1，如此可以確保輸出到 BLIF 的格式正確且有機會可以減少電路的 level。在做完此動作後，電路的 level 可能就不是最大的 label 數值，必須要重新計算 level，以防在轉換時 level 有變導致數值錯誤，這裡要特別小心。

- **Result**

testcases	k = 2		k = 3		k = 4		k = 5		k = 6	
	level	LUTs	level	LUTs	level	LUTs	level	LUTs	level	LUTs
9symml	12	233	7	149	6	109	5	73	4	48
alu4	31	1232	16	922	12	763	9	555	8	483
big2	22	9627	12	6750	10	5557	8	5124	7	3729
C1355	12	214	9	172	4	74	4	74	4	74
C6288	90	2408	45	989	26	734	23	748	20	608
cht	5	339	4	194	3	69	2	40	1	36
cm138a	4	30	2	17	2	21	2	21	1	8
des	14	6798	9	5476	6	3985	6	3945	3	977
i2	11	231	7	192	5	101	5	87	4	68
i3	6	126	4	114	3	50	3	46	3	46
i4	8	246	5	182	4	142	3	82	3	78
k2	14	2876	9	2341	7	1464	6	1373	5	1170
sample01	4	7	2	4	1	2	1	2	1	2
sample02	4	12	2	7	2	5	1	3	1	3
z4ml	8	252	6	199	4	106	4	53	3	28

- **Reference**

[1] LEDA, http://www.algorithmic-solutions.info/leda_guide/Index.html
[2] DAG-Map Graph-Based FPGA Technology Mapping for Delay Optimization
[3] FlowMap an Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs