COL 215

SOFTWARE ASSIGNMENT 2

Objectives:

1. To calculate the combinatorial path in a given sequential circuit with D-flip-flops which has the longest timing delay.
2. Given three implementations for each gate(slow, moderate and fast) to provide an assignment to the gates in the circuit which minimizes the area of the circuit while keeping the maximum combinatorial delay less than a given constraint.

Modelling The Circuit

The circuit can be modelled using directed graph data structures. We define the classes “Signal”, “Gate” and “Circuit” to provide an abstraction to the problem. Since there may be loops present in the original circuit due to the presence of DFFs, we need to modify the graph so that it still remains a DAG (Directed Acyclic Graph).

The important thing to note is that the input of a DFF denotes the end of a combinatorial path (so that it can added in the circuit as a signal with no children) while the output denotes the start of a new combinatorial path (so that it can be added in the circuit as a signal with no parent signal). This helps in generalization so that we do not have to consider the paths containing DFFs separately.

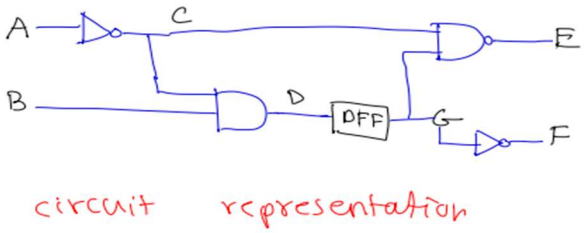
Each signal has attributes *delay*, *children*, *ingate*, *parent1* and *parent2,* where parent1 and parent2 pass through the gate ingate to produce the signal, and children consists of all the signals who have the current signal as an output.

PART – 1

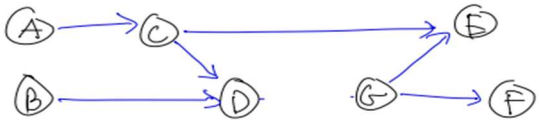
The approach for this part uses a well known graph algorithm called ***topological sort*** **.** The approach uses dynamic programming paradigm approach to calculate the delay of a signal using pre-computed delays of its parents. The topological order ensures that the delay of a signal is calculated only after that of its parents has already been calculated using the recurrence :

$$delay(u) = G.delay + \max\left(delay(u.parent1), delay(u.parent2)\right)$$

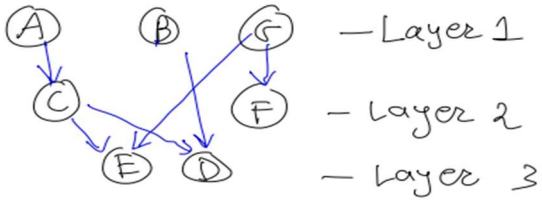
The following figure shows our modelling :



circuit representation



DAG representation



Layer wise topological sort.

The following pseudo-code explains the ***layer-wise-topological-sorting*** algorithm which sorts the circuit graph in the order of execution.

```
LayerWiseTopoSort
Input : the circuit graph G
Dictionary INDEGREE  //stores signal : no. of parents pairs
Queue Q                    // queue data structure for processing signals
For all v in G do
    If INDEGREE(v) == 0 then do
        Q.push(v)
While(Q is not empty)
    New Layer L
    i = Q.size()
    While i > 0 do
        s = Q.front()
        Q.pop()
        Add s to L
        For all children u of s do
            INDEGREE(u) = INDEGREE(u) – 1
            If (INDEGREE(u) == 0) then do
                Q.push(u)
        i = i – 1
   Output layer L
```

Now that we have sorted the circuit graph layer wise, we can apply dynamic programming to calculate the delays of each signal, and finally the maximum delay.

```
MaximumDelay
Input  : the sorted layers (L1,L2,L3..) of the circuit graph

For each signal v in the first layer L(i) do
   v.delay = 0
For each layer L(i) do
    For each signal v in L(i) do
        v.delay = c.ingate.delay + max(v.parent1.delay,v.parent2.delay)
Let MaxDelay be the maximum delay for the last layer
Output MaxDelay
```

Here is the C++ implementation:

```cpp
float Circuit::max_delay()
{
    map<Signal *, int> indegree;
    vector<vector<Signal *>> layers;
    for (int i = 0; i < size; i++)
    {
        int in_deg = 0;
        if (signals[i]->parent1)
            in_deg++;
        if (signals[i]->parent2)
            in_deg++;
        indegree[signals[i]] = in_deg;
    }

    queue<Signal *> Q;
    for (int i = 0; i < size; i++)
    {
        if (indegree[signals[i]] == 0)
        {
            Q.push(signals[i]);
        }
    }
    while(!Q.empty())
    {
        int curr_size = Q.size();
        vector<Signal *> L;
        for(int i = 0; i < curr_size; i++)
        {
            Signal* s = Q.front();
            Q.pop();
            L.push_back(s);
            for(auto u : s->children)
            {
                indegree[u]--;
                if(indegree[u] == 0) Q.push(u);
            }

        }
        layers.push_back(L);
    }

    int num_layers = layers.size();
    for(int i =0; i < layers[0].size(); i++)
    {
        layers[0][i]->delay = 0;
    }
    for(int i = 1; i < num_layers; i++)
    {
        for(auto &u : layers[i])
        {
            if(u->parent1 && u->parent2)
            {

                u->delay = u->ingate.cur_delay + max(u->parent1->delay,u->parent2->delay);
            }
            else if(u->parent1&&!u->parent2)
            {
                u->delay = u->ingate.cur_delay + u->parent1->delay;
            }
        }
    }

    float mx_delay = 0;
    for(auto &u : layers[num_layers-1])
    {
        mx_delay = max(mx_delay,u->delay);
    }



    return mx_delay;

}
```

Analysis Of The Algorithm

Let the total number of signal in the circuit graph G be V and for each $v \in V$ let n_v be the number of children . Let the total edges be E. By a standard result from graph theory, we have $\Sigma_{v \in V} n_v = E$.

The layerwise toplo-sort runs in $O(E + V)$ time, since each vertex is popped out of the queue only once, and for each vertex, we process all of its children in $O(1)$ time. So, the total time complexity is $O(V + \Sigma_{v \in V} n_v) = O(E + V)$ as claimed.

In the MaximumDelay procedure, since we calculate the delay of each signal using pre-computed value in $O(1)$ time, it also runs in $O(V)$. So the overall complexity is still $O(E + V)$, that is , linear in the size of the input.

PART – 2

The given problem is proven to be NP-HARD and currently does not have any deterministic polynomial time algorithms. We first present an exact solution which can be used for small circuits then two approximate algorithms for larger input circuits. Since each gate has 3 possible implementations

```
BruteForce
Input : circuit graph G. total number of gates = k, delay constraint
Set current area = total area of the circuit
Let C be the collection of all k-length ternary sequences of {0,1,2}
For each sequence in C
    Assignt the types to all consequent gates
    Calculate MaximumDelay for this circuit
     If MaxDelay is less than delay constraint, update current area tp be the
minimum
Output current area
```

It can be easily seen that since calculating the maximum delay for a circuit takes linear time in the size of the circuit and the total number of sequences are 3^k, the complexity of the algorithm is given by $O(3^k(V+E))$, which is quite inefficient. Here is the C++ implementation:

```
float brute_force(Circuit &circ, float delay_constraint)
{
    int sz = 0;
    for (auto &s : circ.signals)
    {
        if (s->parent1 || s->parent2)
        {
            sz++;
        }
    }
    vector<vector<int>> combinations = permutations(sz);
    float curr_area = circ.total_area();
    for (auto comb : combinations)
    {
        int count = 0;
        int i = 0;
        vector<pair<float,float>> lol;
        while (count < comb.size() && i < circ.size)
        {
            if (circ.signals[i]->parent1 || circ.signals[i]->parent2)
            {
                circ.signals[i]->ingate.perturb(comb[count]);
                i++;
                count++;
            }
            else
                i++;
        }
        float curr_delay = circ.max_delay();
        if (curr_delay <= delay_constraint)
        {
            curr_area = min(curr_area, circ.total_area());
        }
    }
    return curr_area;
}
```

APPROACH 1

To get a polynomial time algorithm, we have to limit our optimization to certain heurestics. This algorithm first scans all the signals in the circuit and determines which signals have the largest number of children (Which can be done efficiently using mergesort). This determines the priority of the signals.

Then we initially set all the gates to the fast mode, and then keep on perturbing these in the priority order to minimize the area until the delay constraint is satisfied. The following shows this :

```
Algorithm-1
Input : circuit graph G. total number of gates = k, delay constraint
For each signal v ∈ V, assign its priority as v.children.size()
Sort the vertices based on their priority
Assign each gate its fastest implementation
Set current area = total area of the circuit
For each signal v ∈ V,
   Current_delay = MaximumDelay
   If (Current_delay > delay constraint)
       Output the previous current area
   Else
      Make the current implementation of v.ingate faster and update
      Current area
```

Analysis Of The Algorithm

The presented algorithm runs in $O(VE + ElogV)$ time. This can be seen since we iterate over each vertex once and updating the thee maximum delay takes $O(E + V)$, time.

APPROACH - 2

Approach Method 2:

This strategy aims to approach the true minimum by leveraging specific patterns. Initially, we construct the circuit using the slowest gates and assign the appropriate time delays to all the signals to indicate their readiness. Here's a breakdown of the steps:

Identify all the final outputs with time allowances exceeding the delay constraint.

Backtrack along these output signals and mark the signals that belong to their critical path during the backtrack.

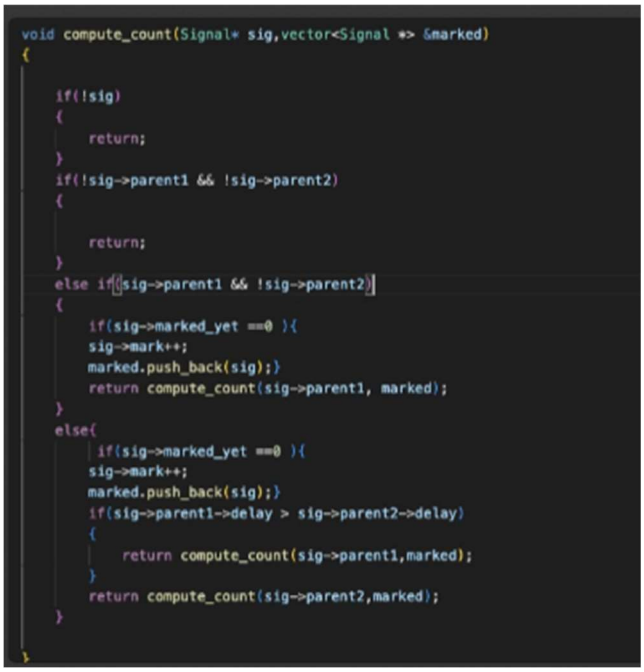
Locate the signal that receives the most markings; we can refer to it as the "bottleneck" signal. Any perturbation to this signal has the greatest impact on the largest number of flagged (undesirable delay time) outputs.

Introduce a perturbation to this "master" signal, recalibrate the delay times, and reevaluate which outputs remain flagged.

Repeat the procedure starting from step 1 until no flagged outputs are left. This method is O(n^3)

Here is the C++ implementation :

Backtracking :

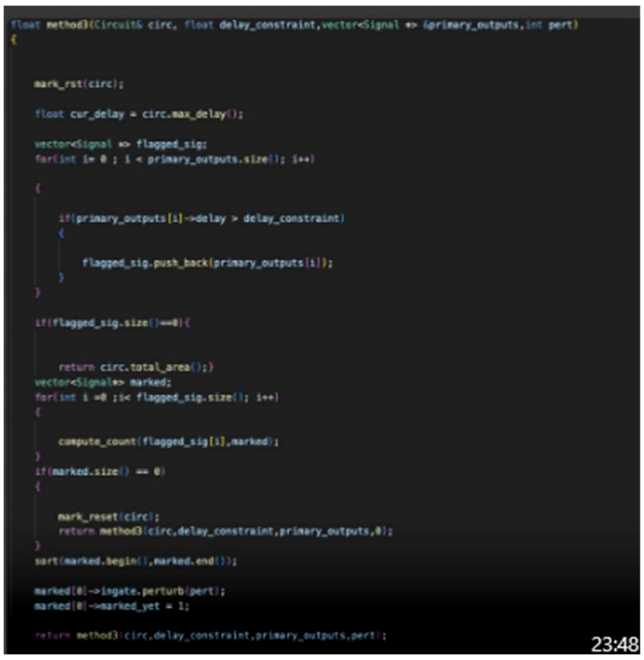


```
void compute_count(Signal* sig,vector<Signal *> &marked)
{

    if(!sig)
    {
        return;
    }
    if(!sig->parent1 && !sig->parent2)
    {

        return;
    }
    else if(sig->parent1 && !sig->parent2)
    {
        if(sig->marked_yet ==0 ){
        sig->mark++;
        marked.push_back(sig);}
        return compute_count(sig->parent1, marked);
    }
    else{
        if(sig->marked_yet ==0 ){
        sig->mark++;
        marked.push_back(sig);}
        if(sig->parent1->delay > sig->parent2->delay)
        {
            return compute_count(sig->parent1,marked);
        }
        return compute_count(sig->parent2,marked);
    }

}
```

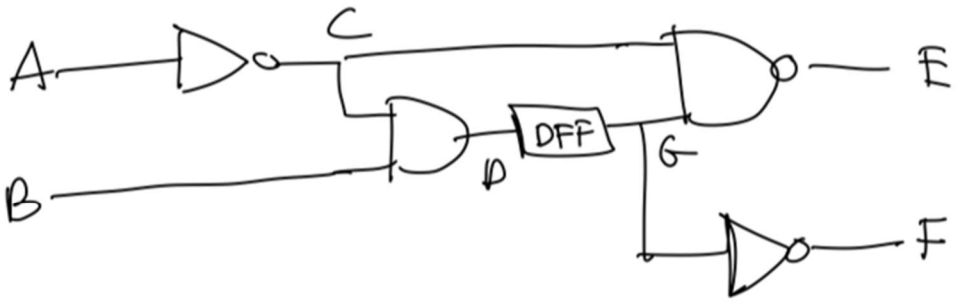
Mark resteers:

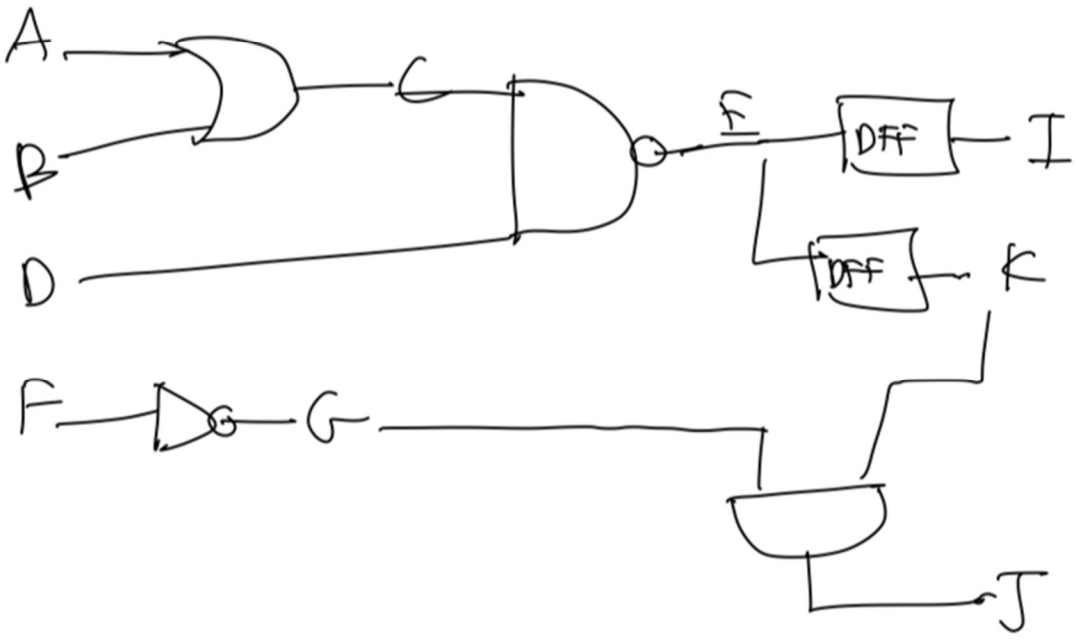
```
void mark_rst(Circuit &circ)

{
    for(int i = 0 ; i< circ.signals.size(); i ++ )
    {
        circ.signals[i]->mark = 0 ;

    }

}
void mark_reset(Circuit &circ)

{
    for(int i = 0 ; i< circ.signals.size(); i ++ )
    {
        circ.signals[i]->marked_yet = 0 ;

    }

}
```

The main method :



Testing :





(4)