# HW2-Event-driven gate-level circuit simulation using a timing wheel data structure

Student Name: Amirhosein Yavarikhoo

Student ID: 810199514

***Abstract*— In this homework we design a simplified kernel and understand the way an event-driven simulator kernel works. Circuit used for testing is c432.**

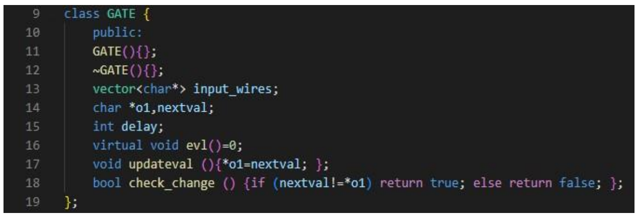
***Keywords*— event-driven, gate-level circuit, timing wheel, simulation in c++, object-oriented programming**

## I. Introduction

In the previous assignment, we used several ways to get ourselves closer to concurrency in c++. In this programming language we don't have timing so we have to devise a clock in order to base our events on it. Timing wheel also uses this clock to implement each event in the circuit.

## II. Data Structures

First we have to design an environment to implement the circuit with it. For this assignment, a mother class called GATE is used to define all gates that are going to be used.



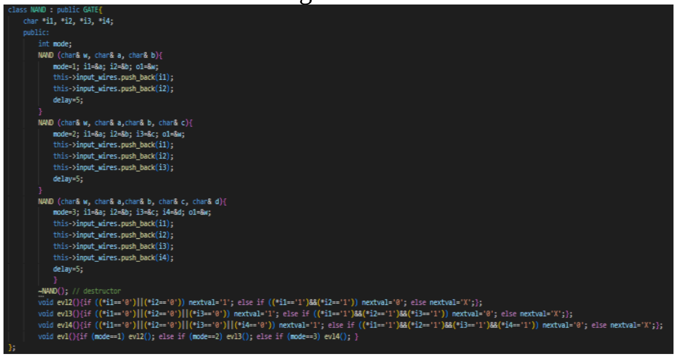
```
class GATE {
    public:
    GATE(){};
    ~GATE(){};
    vector<char*> input_wires;
    char *o1,nextval;
    int delay;
    virtual void evl()=0;
    void updateval (){*o1=nextval; };
    bool check_change () {if (nextval!=*o1) return true; else return false; };
};
```

Fig. 1 GATE class

Basically everything that all classes have in common are defined in this class. Char* o1 is the output of the gate. Input_wires is a vector that stores pointers to wires that are used as input. Nextval will be discussed later. Delay is implemented as an integer and evl function is declared purely virtual so that we can describe it in every other gate according to their functionality. Updateval updates the output when called and check_change is a function that gets called upon an event in inputs of the gate. If this event changes the output of the gate, this function return true value. When simulating with delay, the output shouldn't change instantly so we need a variable to save that output. This is where nextval comes in. in evl function of all gates, changes are applied to nextval. This helps us to use less resource as we can use o1 as previous value indicator.

In this assignment, 6 types of gates are used: AND,OR,NAND,XOR,NOT and BUFFER. Buffer will be explained later in details of implementing timing wheel. NAND gate will be explained as a complete example. Other gates are similar to this gate and only differ in functionality and constructors.

Let's take a look at NAND gate:

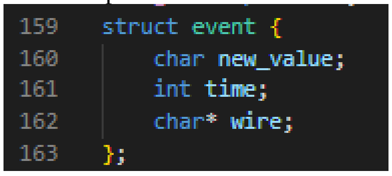


```
class NAND : public GATE{
    char *i1, *i2, *i3, *i4;
    public:
        int mode;
        NAND (char& w, char& a, char& b){
            mode=1; i1=&a; i2=&b; o1=&w;
            this->input_wires.push_back(i1);
            this->input_wires.push_back(i2);
            delay=5;
        }
        NAND (char& w, char& a,char& b, char& c){
            mode=2; i1=&a; i2=&b; i3=&c; o1=&w;
            this->input_wires.push_back(i1);
            this->input_wires.push_back(i2);
            this->input_wires.push_back(i3);
            delay=5;
        }
        NAND (char& w, char& a,char& b, char& c, char& d){
            mode=3; i1=&a; i2=&b; i3=&c; i4=&d; o1=&w;
            this->input_wires.push_back(i1);
            this->input_wires.push_back(i2);
            this->input_wires.push_back(i3);
            this->input_wires.push_back(i4);
            delay=5;
            }
        ~NAND(); // destructor
        void evl2(){if ((*i1=='0')||(*i2=='0')) nextval='1'; else if ((*i1=='1')&&(*i2=='1')) nextval='0'; else nextval='X';};
        void evl3(){if ((*i1=='0')||(*i2=='0')||(*i3=='0')) nextval='1'; else if ((*i1=='1')&&(*i2=='1')&&(*i3=='1')) nextval='0'; else nextval='X';};
        void evl4(){if ((*i1=='0')||(*i2=='0')||(*i3=='0')||(*i4=='0')) nextval='1'; else if ((*i1=='1')&&(*i2=='1')&&(*i3=='1')&&(*i4=='1')) nextval='0'; else nextval='X';};
        void evl(){if (mode==1) evl2(); else if (mode==2) evl3(); else if (mode==3) evl4(); }
};
```

Fig. 2 AND gate

Here we have 3 different constructors for NAND gate. This is due to having multiple input NAND gates in c432 circuit. Each input wire is pushed into input_wire vector when constructor is called. In constructor, we identify evl mode and delay of the gate. Due to having multiple inputs, different evl functions are declared and a main evl function is used with if-statements to decide which one of these functions should be called.

Here is a picture of event struct:



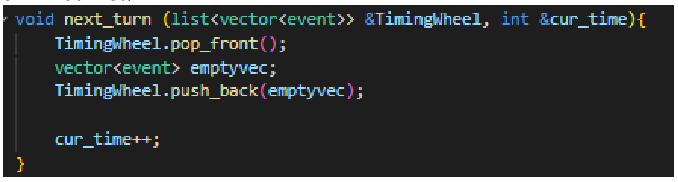
```
struct event {
    char new_value;
    int time;
    char* wire;
};
```

Fig. 3 event structure

Each event has a value, a time when the event should be implemented and a pointer to a wire.

## III. Timing Wheel

Timing wheel is essentially a linked list of a vector of events. To implement the linked list, list library is used. Utility functions are needed to implement different actions of this linked list.



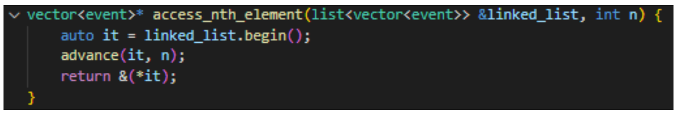
```
void next_turn (list<vector<event>> &TimingWheel, int &cur_time){
    TimingWheel.pop_front();
    vector<event> emptyvec;
    TimingWheel.push_back(emptyvec);

    cur_time++;
}
```

Fig. 4 turning the wheel

Turning the wheel means that the current time should be popped of the list, a new empty time should be added at the end of the list and timer should be incremented. Next_turn function does this for us.

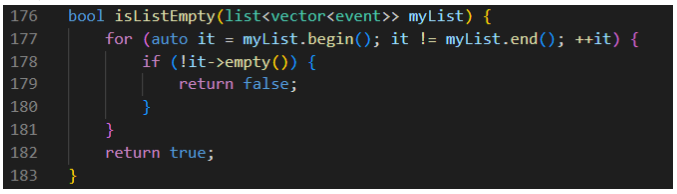
In implementing the simulation, after each event is done, we may need to issue a new event (if the output of the gate is changed due to the previous event.). To do so, we need a function that points to an element of list that has that time in order to push the event in its vector. The following function does this for us and returns a pointer to the vector of the needed list element.



```
vector<event>* access_nth_element(list<vector<event>> &linked_list, int n) {
    auto it = linked_list.begin();
    advance(it, n);
    return &(*it);
}
```

Fig. 5 access function

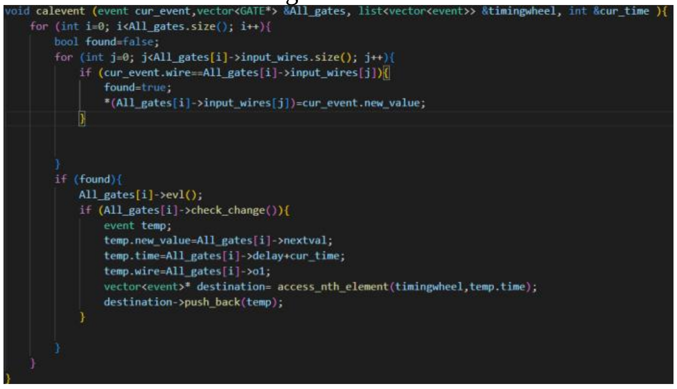
The simulation ends when no events remained. In other words, each vector of each element of the linked list should be empty in order to finish the simulation. The following function does this for us and returns a value to indicate empty status of the list.



```
bool isListEmpty(list<vector<event>> myList) {
    for (auto it = myList.begin(); it != myList.end(); ++it) {
        if (!it->empty()) {
            return false;
        }
    }
    return true;
}
```

Fig. 6 empty status

Now we get to event calculation part. This function has access to the current event, all gates, timing wheel and time. Using a for loop, we identify gates that use the event's wire as input. Buffer gates are used for this purpose. Since outputs of the whole c432 circuit isn't used as inputs for any gates, we use dummy gates such as buffer with 0 delay. If the event's wire matches the gate's input, a found flag is issued and the new value will be implemented. Now that the input value is implemented, we call evl function of that gate to see if output would be changed or not. If output change is triggered (this is the part we use check change from class) a new event is issued and scheduled in the timing wheel.



```
void calevent (event cur_event,vector<GATE*> &All_gates, list<vector<event>> &timingwheel, int &cur_time ){
    for (int i=0; i<All_gates.size(); i++){
        bool found=false;
        for (int j=0; j<All_gates[i]->input_wires.size(); j++){
            if (cur_event.wire==All_gates[i]->input_wires[j]){
                found=true;
                *(All_gates[i]->input_wires[j])=cur_event.new_value;
            }


        }
        if (found){
            All_gates[i]->evl();
            if (All_gates[i]->check_change()){
                event temp;
                temp.new_value=All_gates[i]->nextval;
                temp.time=All_gates[i]->delay+cur_time;
                temp.wire=All_gates[i]->o1;
                vector<event>* destination= access_nth_element(timingwheel,temp.time);
                destination->push_back(temp);
            }

        }
    }
}
```

Fig. 7 calevent function

At last, the main timing wheel function is called. Before any events happen, we push the events taken from user into the timing wheel (taking inputs from user will be discussed later.). We use a while loop with empty status condition. If the whole list is empty, while loop will end and the simulation will be finished. An iterator is used to access each element of the timing wheel. If the element is empty, the output at that time will be printed and timing wheel will turn. If it's not empty, events will be implemented. To access each event in the vector of events, another iterator is used and for each event, calevent function is called. After all the events of the vector are implemented, outputs will be printed and the wheel will turn.

## IV. Getting Events from User

According to the format explained in the assignment, each line consists of #time and 36 inputs. Each of the inputs indicate input wires of the circuit. These wires are sorted according to their numbers. After taking input from user is finished, user has to type done in order to start the simulation.

## V. Testbench Results

Five different scenarios are tested. Inputs and outputs will be stored in files in the archive.