Elevator Controller

Aviral Kumar, 140070031
Saurabh Garg, 140070003
Siddhant Garg, 14D070027
Ritwick Chaudhry, 14D070063

April 6, 2016
## Contents

0.1 High Level System Diagram : ................................. 2
0.2 Extra Things which we implemented : ........................ 4
    0.2.1 Energy Minimising Elevator Scheduling Algorithm: ........ 4
    0.2.2 Dynamic Test Bench Import from file ...................... 6
    0.2.3 Override Switch ........................................... 7
0.3 Description of Functionalities of Blocks : ..................... 7
0.4 Algorithm : ..................................................... 14
0.5 Some Tricky Cases & Testing : ................................ 14
0.6 Assumptions : .................................................. 15
0.7 Plan for Division of Work among team members ................. 15
0.1 High Level System Diagram:

Our system is composed of 4 types of controllers, the Main Elevator Controller, the Door Controllers, the Motor Controller and the Floor_To_Go module. The Door Controller is used to keep doors open and closed when instructed so by the Main Controller. The Motor Controller is responsible for the motion of the Lift Motor (Up, Down or Idle). The Floor_To_Go (FTG) module calculates the next Floor and the next Direction the lift should go to. Then there’s a memory element which stores the path traversed by the elevator for audit purposes.

The high level system diagram follows.
0.2 Extra Things which we implemented:

After we were completed with our basic version of the project, we decided to implement some more functionalities, which are described below.

0.2.1 Energy Minimising Elevator Scheduling Algorithm:

This algorithm comes into play when the elevator is idle at a floor, and multiple requests from different floors come up simultaneously. This is an **NP-Hard** problem, so algorithms to solve this in polynomial time can only be Approximation Algorithms. Ours is one such approximation algorithm. The algorithm is given as follows:

As the future requests, are not known to us beforehand, so, we can only frame an algorithm depending upon the current Requests and by assuming that all the people, who made the requests want to move exactly one floor up/down (depending upon the request, whether it is up or down, and we then find out the optimal direction in which the lift should move, provided the above stated assumption. This is an approximation and would give results better than a naive solution.

Moreover, once the lift is into motion, it caters to the requests with priority along its current direction of motion, as it does in the normal case, when it is in motion.

The choice of the direction of motion given by the algorithm is based on an inequality found out by comparing the energies spent when the lift moves up and when it moves down.

For the purpose of derivation let us declare some variables:

$H_d$ : The floor with the highest down request from the floors above the `currFloor` $H_u$ : The floor with the highest up request from the floors above the `currFloor`

$h_d$ : The floor with the lowest down request from the floors below the `currFloor`

$L_u$ : The floor with the lowest up request from the floors below the `currFloor`

$L_d$ : The floor with the lowest down request from the floors below the `currFloor`

$l_u$ : The floor with the highest up request from the floors below the `currFloor`

The total energy consumed is a function of the net distance travelled, and hence depends upon the highest and lowest floors which have a request high (either up or down request). So, the following are defined:

\[
H = \max(H_u, H_d) \\
L = \min(L_u, L_d)
\]

Now, four cases can arise, depending upon whether $H$ is $H_u$ or $H_d$, and likewise for $L$.

For each case, we get a separate linear inequality. The derivation is below.

1. **Case 1** : $H = H_u, L = L_u$ : In this case, we’ll write down the energy needed to move up first and that needed to move down first, and compare the two values.

   \[
   \text{Energy(Up)} = 5(H_u + 1 - \text{currFloor}) + 2(\text{currFloor} - l_u) + 3(l_u + 1 - L_u) \\
   \text{Energy(Down)} = 5(\text{currFloor} - L_u) + 2(H_u - h_d + 2) + 3(H_u + 1 - \text{currFloor})
   \]

   From these equations, we get that we move upwards first if,

   \[
   \text{Energy(Up)} < \text{Energy(Down)} \\
   \Rightarrow 5(\text{currFloor}) - 1 > 3l_u + 2h_d
   \]

   If the opposite is True, then we move Down first.

   Similarly, doing the same thing for the other three cases, we get the following:
2. Case 2: \( H = H_u, L = L_d \):

\[
\text{Energy(Up)} > \text{Energy(Down)} \\
\Rightarrow 5(\text{currFloor}) < 3l_u + 2h_d + 11
\]

3. Case 3: \( H = H_d, L = L_u \):

\[
\text{Energy(Up)} > \text{Energy(Down)} \\
\Rightarrow 5(\text{currFloor}) - 1 < 3l_u + 2h_d
\]

4. Case 4: \( H = H_d, L = L_d \):

\[
\text{Energy(Up)} > \text{Energy(Down)} \\
\Rightarrow 5(\text{currFloor}) + 6 < 3l_u + 2h_d
\]

The derivation of these inequalities just involves writing the energies and then simplifying the expressions.

We have implemented these by finding out the values of the variables, finding out which case the situation falls into and then applying the inequality, to find the value of newDir.

**Optimality Proof:**

Consider a case which can serve as a "Worst Case" for this algorithm. By analysing the performance of the algorithm in the worst case, we can comment upon the optimality of the algorithm. Consider, a case when \( H = H_d, L = L_u \), in this case, suppose, the algorithm tells you to go up. (i.e. \( E(\text{Down}) > E(\text{Up}) \)). Now suppose the person who gets in at floor \( L_u \) wants to go to the topmost floor. Mathematically, we can state this as:

\[
E(\text{Up}) < E(\text{Down})(1) \text{ and } (1) \Rightarrow 5(\text{currFloor}) > 1 + 3l_u + 2h_d(3)
\]

Here, the highest request is \( H_d \) and the Lowest request is \( L_u \). The worst case for this scenario would be when the \( L_u \) is for going to the topmost floor i.e. \( H \). We derived the above formulae assuming that the request were made to go exactly 1 floor in the specified direction. However here, since the request by \( L_u \) is not for going just one floor, our algorithm may not give the correct answer. For this specific case, let \( E_1 \), and \( E_2 \) denote the energies consumed, on Going up and down respectively.

\[
E_1 = \text{Energy(GoingUpFirst)} : \\
= 5 \times (H_d - \text{CurrFloor}) + 5 \times (\text{CurrFloor} - L_u) + 3 \times (H - \text{CurrFloor}) \\
= 5 \times (H_d - L_u) + 3 \times (H - \text{CurrFloor}) \\
= 3 \times H_d + 3 \times H - 5 \times L_u - 3 \times \text{CurrFloor}
\]

\[
E_2 = \text{Energy(GoingDownFirst)} : \\
= 5 \times (\text{CurrFloor} - L_u) + 3 \times (H_d - \text{CurrFloor}) + 2 \times (H_d - h_d) + 2 \\
= 2 \times \text{CurrFloor} + 5 \times H_d - 5 \times L_u - 2 \times h_d + 2
\]
By the algorithm we have implemented, it tells us to go up first if,
5(currFloor) > 1 + 3lu + 2hd holds, which means that the value of CurrFloor is
greater than \( \frac{3 \times lu + 2 \times hd + 1}{5} \)

For finding the maximum inefficiency, let the energy while going up first be more
than the energy while going down

\[
(E_1 \geq E_2) \Rightarrow -2 \times H_d + 3 \times H - \text{CurrFloor} + 2 \times h_d - 2 \geq 0
\]

\[
\Rightarrow 3 \times H + 2 \times h_d \geq 2 \times H_d + \text{CurrFloor} + 2
\]

The percentage Energy difference between \( E_1 \) and \( E_2 \) is \( \Delta E \):

\[
\Delta E = \frac{3 \times H + 2 \times h_d - 2 \times H_d - \text{CurrFloor} - 2}{2 \times \text{CurrFloor} - 5 \times lu + 5 \times H_d - 2 \times h_d + 2} \times 100
\]

\[
\leq \frac{15 \times H + 8 \times h_d - 3 \times lu - 10 \times H_d - 11}{6 \times lu + 25 \times H_d - 25 \times lu - 6 \times h_d + 12} \times 100
\]

Putting \( H_d = h_d \) and \( L_u = l_u \) so as to find the maximum numerator and minimum
denominator simultaneously for finding the maximum bound on the \( \Delta E \)

\[
\Delta E \leq \frac{15 \times H - 2 \times h_d - 3 \times lu - 11}{19 \times h_d - 19 \times lu + 12} \times 100
\]

Since the given case requires \( H_d \) to be the highest request and \( L_u \) to be the lowest request

\[
\Rightarrow h_d \text{ and } l_u \text{ both exist } \Rightarrow h_d \geq 2 + l_u
\]

\[
\Delta E \leq \frac{15 \times H - 2 \times (l_u + 2) - 3 \times lu - 11}{50} \times 100
\]

Putting the value of \( H \) as 5 (in our case)

\[
\Delta E \leq \frac{12 - l_u}{10} \times 100
\]

This is the worst case upper bound, even in which it performs inefficiently within
some bound. Also this is a loose bound and a tighter bound can be obtained using
more advanced analysis of the same

The algorithm we found works quite efficiently for most cases we tested (works
exactly for some cases as well).

### 0.2.2 Dynamic Test Bench Import from file

We have added the functionality of being able to import the test bench from a text
file rather than directly writing the entire sequence directly in the test bench. The
information for the test bench has to be encoded in a particular way as follows:

(a) For a request, the format is

\[
1 \text{ <Current Floor> <Requested Floor> <Number of People> <Weight of people>}
\]

(b) For a delay, the format is

\[
0 \text{ <Delay in Time_Periods>}
\]

(c) For fire, the format is 2 which depicts fire alarm is high
A python script is then run on this input text file and the algorithm for the elevator’s floor scheduling is implemented. Finally the text file to be directly read from the test bench is created. This file is encoded in the following fashion:

(a) For a delay, the format is
   0 <Delay in Time_Periods>

(b) For the Floor Sensor Input, the format is
   1 <10 bit array>
   Two bits for each floor’s dual floor sensors

(c) For the Requests, the output format is
   2 <A 13 bit binary number>
   Bit 0-4 for the in-lift requests and the remaining 8 bits for the button on the floors outside to go up or down

(d) For the Weight Input, the output format is
   3 <Weight in a 9 bit binary number>

(e) For the Fire Alarm, the output format is where 1 denotes Fire alarm is high
    4 1

This text file is then directly read in the VHDL test bench using file input/output operations and the test bench is thus created.

0.2.3 Override Switch

There is an override button, which when pressed ignores all intermediate requests and the lift is sent to the floor which is pressed along with the override button. This is to handle emergency cases where a floor’s priority is raised and the elevator’s normal floor scheduling algorithm is overriden.

This is particularly useful for applications in places like hospitals, where the ICU or the emergency unit of the hospital is usually present on the certain floor. In case of an emergency the override is used and the floor where the emergency unit is present are pressed simultaneously. This overrides the normal elevator’s scheduling algorithm and then the lift opens on that particular floor without opening on any other floor in the interim time.

0.3 Description of Functionalities of Blocks:

Floor-To-Go (FTG Module)

Inputs:

(a) Current Floor, currFloor
(b) Current Direction of Motion of the elevator, currDir
(c) Requests Array of 13 requests, req[13]
(d) Fire Alarm, fireAlarm
Outputs:
(a) Next Floor the elevator needs to go on, nextFloor
(b) New Direction the elevator must take, newDir

Working:

This module find out the next floor, and the next direction to which the elevator controller must move to by processing the requests and the current state of the elevator. This module is purely combinational i.e. it justs applies some "ite" (if-then-else) operations to figure out the next floor.

The functioning of this module is basically similar to the arbiter. The priority of a floor is determined by the direction of motion currDir of the elevator. Any floor reachable in the direction of currDir is at a higher priority than a floor opposite to this directiopn relative to the currFloor.

Case 0: fireAlarm = 1 : If currDir = down, then set newFloor := currFloor -1 else if currDir = up /idle assign currFloor to newFloor, with the newDir being the same as currDir.

Case 1: currDir = up and not fireAlarm : Check whether a up request is high among the floors above the currFloor. As soon as the first such floor is found, assign this value to the newFloor, newDir = currDir. If no such floor is found above the elevator, then we need to go to the highest floor above the currFloor with a down request set to high. If such a floor is found, assign it to newFloor without changing the direction of motion. Else, check for requests for the floors below. If any request is found ( req_up, req_down or in_lift ) then just set the currDir = down.

Case 2 : currDir = idle and not fireAlarm : In this case, the elevator is stationary (denoted by idle initially, then check the requests from various floors above and below it. If a request for a floor above it is found to be high, assign newFloor this value and assign newDir = up, and similarly if a request at a floor below is found to be high. If multiple requests are found to be high simultaneously, then assign one of them randomly.

The case for currDir = down follows analogously from the first Case.

Pseudocode:

floorsAbove := [currFloor+1.....5];
floorsBelow := [1....currFloor-1];
max_flr := currFloor; //max_flr is used to find the maximum floor above the currFloor

if fireAlarm = 1, then:
    if currDir = down, then:
        newDir := down;
        newFloor := currFloor - 1;
        return;
    end if;
    newFloor := currFloor;
    if currDir = idle, then:
        newDir := idle;
return;
else
    // Case for currDir = down follows analogusly
    return;

if currDir is up, then:
    for i in floorsAbove:
        if req_up[i] or in_req[i] is high, then:
            newFloor := i;
            newDir := currDir;
            return;
        else if req_dn[i] is high, then:
            max_flr := max(max_flr, i);
        end if;
    done

if max_flr is currFloor, then: //No requests found above the current floor
    //Check for any request at floor below
    newFloor := currFloor;
    for i in floorsBelow:
        if req_up[i] or in_req[i] or req_dn[i] is high, then:
            newDir := down;
            break;
        end if;
    newDir := idle; //Set newDir to idle if there is no request to go to any floor
        // above or below the currFloor
    end if;

if newFloor is not set, then: // not set means no value assigned in this iteration
    newFloor := max_flr;
    newDir := currDir;
end if;

if currDir is idle, then:
    //Apply arbiter logic here
    for i in floorsAbove+floorsBelow+currFloor:
    if req_up[i] or in_list[i] or req_down[i] is high:
        newFloor := i;
        newDir := (i < currFloor)? (down): ((i>currFloor)? (up) : idle);
        break;
    end if;
end if;

Motor Controller

Inputs:
Motion bit and Direction bit (from the Main Controller): idle, dirBit

Outputs:
(a) Motor up signal, motorUp
(b) Motor down signal, motorDown
Working:

This controller controls the functioning of the motors needed to move the elevator up and down by setting the values of `motorUp` and `motorDown` from the received values of `idle` and `dirBit`. The Main Controller continuously keeps sending some values of `idle` and `dirBit` continuously and the Motor Controller thereby calculates the values of `motorUp` and `motorDown` and sends them to the motor. The state diagram shown below explains the working of this controller in detail.

```
TRANSITION FORMAT:
INPUT 'idle'
INPUT 'dirBit' /
OUTPUT 'motorUp'
OUTPUT 'motorDown'
```

Door Controller

Inputs:

(a) Weight, coming from the weight sensor, `weight`
(b) Open or Close Flag, `State`
(c) Forever, becomes high when fire alarm is high `forever`
Outputs:

(a) Done Flag, to the Main Controller \texttt{done}

Working:

The door controller controls the opening and closing of doors by controlling the motors at each floor for this purpose. It maintains internal state variables, \texttt{open} and \texttt{op\_counter}. The value of \texttt{op\_counter} is incremented by some fixed quantity \(x\), determined by the weight value, \texttt{weight}. When \texttt{op\_counter} reaches its maximum value, \(\text{max(op\_counter)}\), the lift then transitions from the open to close state. All this is explained in much more detail through the State Diagram.

\begin{center}
\begin{tikzpicture}[node distance=1.5cm,thick,main node/.style={fill=blue!20,draw,font={\sffamily\Large\bfseries}}]
\node[main node] (Open) {DoorOpen};
\node[main node] (Close) [right of=Open] {DoorClose};
\draw[->] (Open) to [loop above] node {1,0,0/0} (Open);
\draw[->] (Open) to node {1,1,x/0} (Close);
\draw[->] (Close) to node {0,-,-/1} (Open);
\draw[->] (Close) to [loop below] node {1,0,max (op\_counter)/1} (Close);
\end{tikzpicture}
\end{center}

\(x < \text{max(op\_counter)}\)

\(x = x + a\)

\(a\) is inversely proportional to \texttt{weight}

Memory Element: First In First Out Queue

Inputs:

- The current Floor the elevator is on, \texttt{currFloor}
- The weight of people in the elevator currently, \texttt{weight}
- To display output, \texttt{read\_log} becomes high

Outputs:

- Display the \texttt{log}, \texttt{logcat}
- The total energy needed in the process, \texttt{energy}

Working:

This Memory element stores a sequence of \texttt{currFloor} values in order in which they were visited. Whenever the elevator closes at the \texttt{currFloor}, this value is stored in the FIFO, along with the weight of people in the elevator after the elevator door closes on that floor. Doing, this will capture all movements of the lift.
along with the weights carried by the lift in between floors where it stopped. The encoding we will use to store data in FIFO will be \textit{currFloor} (3-bit vector) concatenated with \textit{weight} (9-bit vector).

So, the energy needed to store is given by the formula,

$$Energy = \sum_{i} \begin{cases} 2 & \text{floor}[i+1] < \text{floor}[i] \\ 0 & \text{floor}[i+1] = \text{floor}[i] \\ 3 & \text{floor}[i+1] > \text{floor}[i] \end{cases}$$

This will be computed by another signal which outputs the value of the sum every time the floor changes.

**Main Elevator Controller :**

**Inputs :**

1. Current Weight, \textit{weight}
2. Fire Alarm Status, \textit{fireAlarm}
3. Request Array, \textit{req[13]}
   (a) 6 \textit{req\_up} and \textit{req\_down} requests from floors 2, 3 and 4
   (b) 1 \textit{req\_up} request at floor 1
   (c) 1 \textit{req\_down} request at the floor 5.
   (d) 5 \textit{in\_lift} requests for floors 1,2,3,4 and 5
   (e) \textit{Reset} input to reset the elevator
   (f) \textit{Clock}
   (g) \textit{Override} for the emergency override functionality
4. Floor sensors, 2 from each floor \textit{floorSens[10]}
5. Next Floor to go to, returned by the FTG Module \textit{nextFloor}
6. 5 Done Flags \textit{done}, determines that the door was open and closed properly and now the elevator can start moving
7. \textit{ReadLog} On enabling starts outputting the log stored in the memory element representing elevator movement

**Outputs :**

1. 5 \textit{Open/Close} flags, one to each Door Controller, along with weight of people, \textit{weight}
2. Motor Controller Flags, \textit{dirBit} and \textit{idle}, to control the up and down movements of the motor.
3. Current Direction of Motion of the elevator, \textit{currDir}
4. Requests Array of 13 requests, \textit{req[13]}
5. Fire Alarm Status, \textit{fireAlarm}
6. Stored sequence of values in the memory element, \textit{logValues}

**Internal Variables :**

1. The floor the lift is \textit{currFloor}
2. The direction the lift is moving in currently, \textit{currDir}
Working:

The working of the main controller is divided into 2 main parts:

1. **When Fire Alarm is High:** The fire Alarm is provided as an input \texttt{fireAlarm} to the Main Elevator Controller, which in turn passes it to the \textbf{FTG Module} which reacts to this stimulus. There are two cases possible here, which are listed below.

   - **Case 1:** \texttt{currDir = down or currDir = up} : The FTG module immediately sets the \texttt{currDir := down} and the \texttt{newFloor := currFloor} if \texttt{currDir = up} and \texttt{newFloor := currFloor-1} if \texttt{currDir = down}. Thus the lift goes down to the just lower floor (thus making both the floor sensors, \texttt{floorSens[currFloor]} = 1 or \texttt{floorSens[currFloor-1]} = 1, depending upon the case). Therefore the lift opens when it reaches the floor just below, i.e. \texttt{currFloor}. (or \texttt{currFloor-1}, depending upon the case).

   - **Case 2:** \texttt{currDir = idle} : The Main Controller in this case just opens the door of \texttt{currFloor} by setting the \texttt{Open/Close} signal to high, and the doors remain open henceforth.

2. The other case is explained with the algorithm below.
0.4 Algorithm:

The algorithm goes as follows:

1. The elevator is assumed to be at floor 1, in idle state. Untill no request comes up, the FTG module keeps on giving the value newFloor = 1 & newDir = idle. Therefore, the Open/Close value for each of the Door Controllers is 0 (signifying closed door) and the done value received by the Elevator Controller is also 1. The dirBit and idle also remain at -(don't care) and 1 respectively.

2. For the general case, when requests req[13] keeps on changing dynamically. Assume that the elevator is on a floor say x. On one side, the FTG module keeps on taking the currFloor and the currDir and the requests, req[13] and computing the values of newFloor and newDir, and sending these out to the Elevator Controller. Meanwhile, parallel to this, the Door Controller is in action. The Open/Close signal for the door of the corresponding currFloor is 1 (Signifying the door is open), and the Door Controller runs through its FSM, opening the door first, waiting appropriate amount of time and then closing it. (See the State Diagram of the Door Controller). It is guaranteed that the Door processing time is greater than the time needed for FTG module for its computation. As the Door processing is over, the Open/Close values are set back to 0 and the done values are set to 1. At this instant, the currDir := newDir and then the currDir is taken and this acts as a signal for starting the Motor Controller by changing the dirBit and idle signals to currDir and 0 respectively.

3. Now, the lift is into motion, where all Door Controllers are in closed Door state, and the Motor Controller is moving the elevator in the previously set direction.

4. Even when the elevator is in motion, the FTG module is active parallely, continuously processing the inputs to give the newFloor and newDir values.

5. Stopping the elevator at a floor: The main controller compares the values of its currFloor to the newFloor value. If at any instant of time, currFloor = newFloor and floorSens[currFloor]1 and floorSens[currFloor]2 (i.e. both the floor sensors for that floor are high), the main controller stops the elevator at this floor, by setting idle to 1 and then it makes the request for this floor low, back again. Also, the value of the currFloor is updated to the present floor. But, the currDir variable can be updated only by the FTG module. (For an example, if a lift comes at a floor, say 3, by coming up, the main controller makes it come to idle state (although this doesn’t update its currDir variable). While the process of picking up and(or) dropping people is on, the FTG module, still assigns the newFloor and newDir values based on the previous currDir i.e. up. (i.e. it scans the floors above 3 for a request).

6. Updating elevator’s currFloor variable: The value of the currFloor is updated by the inputs of the floorSens[5] to the floor for which floorSens[i] is high.

0.5 Some Tricky Cases & Testing:

We plan to test our system by incorporating some tricky test cases into our test bench. Two of these are mentioned below. Moreover the system will also be tested for its efficiency in terms of the energy consumed. We will also try to implement some other algorithms for handling the Case 2 mentioned under the FTG Module, and figure out how we can decrease the energy consumed by testing it against some test cases and choose the one which decreases the energy consumed more frequently.

Some of the tricky test cases are:

1. Case 1: currFloor = 2, req_down[3] = 1, req_up[4] = 1 all other requests are 0: Then the lift first goes to the 4th floor due to its currDir as being up which is intuitive according to the real life elevator functioning.
2. **Case 2**: The lift was moving up and stopped at a floor, say 3. Before halting at floor 3 there was no request for floor 4 or floor 5. But while the door was open at floor 3 a request for floor 4 or 5 came: This is handled by our design as the **FTG Module** works in parallel even while the door is open. The next direction of motion is decided only after the door gets closed. Therefore the lift will handle the requests that arrive after the door opened on the current floor.

3. **Case 3**: The lift was idle at the 3rd floor. At the same instant, two requests, one at floor 4 and the other at floor 2 came up. The elevator will then respond to this situation by entertaining the request at 2 first, and then 4, because of arbiter like behaviour. This is correct because at this moment, nothing can be predicted about the future requests, and the requests currently have the same priority (both are 1 floors each from 3).

0.6 **Assumptions**:

1. We have two floor sensors at each floor, and both give a high signal **iff** lift is at that floor, and only when this happens, the door can be opened.
2. All signals are high for time sufficient to meet the requirements of other modules/controllers.
3. Sensors function properly.
4. **(REMOVED)** If a lift is idle at a floor, and both an up and a down request come simultaneously, then it is assumed that we can cater to any one of these requests (without any priority). **Now we are catering to requests by using an approximate algorithm so as to minimize energy**.
5. Maximum weight inside the elevator is 400 kgs, else the doors don’t close.

0.7 **Plan for Division of Work among team members**

We did most of the project all sitting together, but if major contributions are to be seen, then

1. Aviral did the Floor-To-Go Module, Scheduling Algorithm and Main Controller(minor)
2. Saurabh did Door Controller, Motor Controller and Automated Test Bench generation
3. Siddhant did the Main Controller(major), Test Benches and Scheduling Algorithm
4. Ritwick did the Memory module, Main Controller(major) and test benches.

All of us were involved in debugging the code when it wasn’t working.