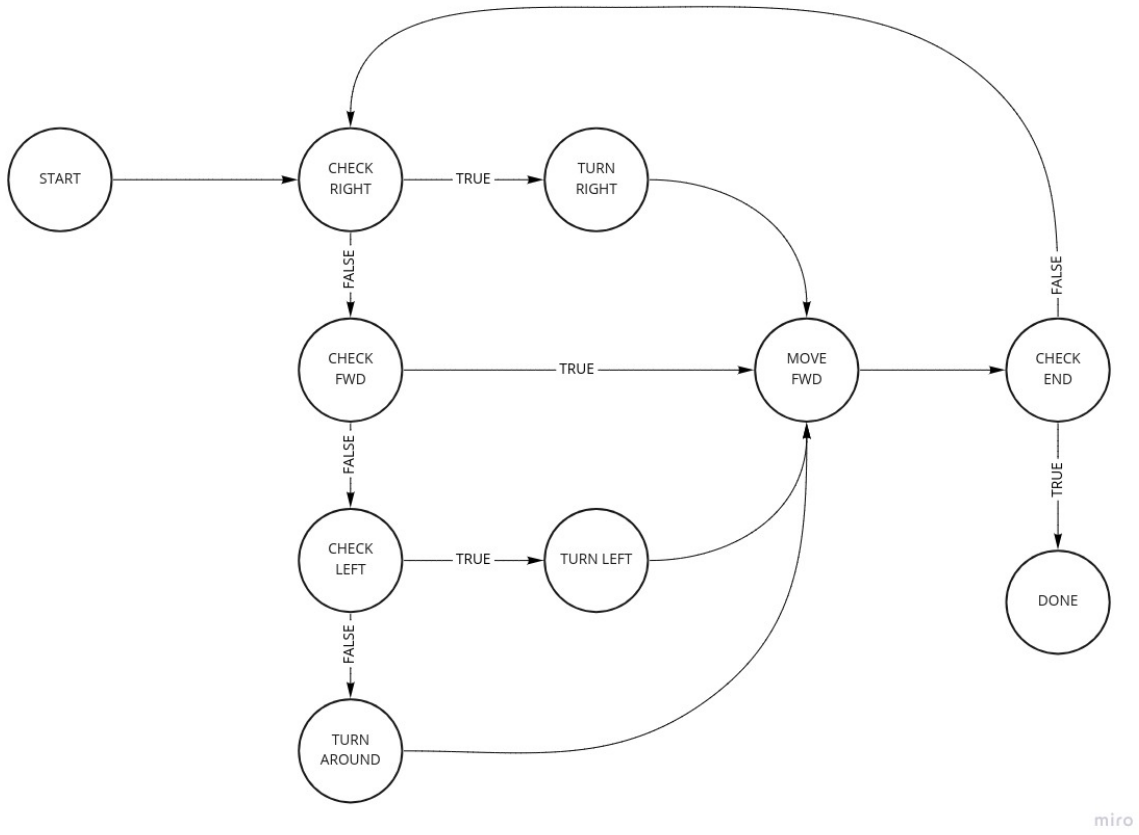
Negoita Mihaela
331AA

Tema 2 - Maze

Inainte de a incepe implementarea propriuzisa a sistemului in Verilog, am incercat sa inteleg algoritmul de rezolvarea a labirintului Wall Follower cu regula mainii drepte. Astfel am implementat algoritmul intr-un alt limbaj (javascript) folosind concepte high-level (while-uri, if-uri, functii) si intreg input-ul stocat in memorie intr-o matrice. Dupa ce am reusit sa rezolv problema folosind aceasta abordare, am trecut la a identifica starile pentru automat. Am ajuns la starile si tranzitiile din diagrama de mai jos (explicate imediat dupa diagrama). Dupa ce am finalizat diagrama, am reimplementat algoritmul in javascript dar folosing un finite state machine cu starile si tranzitiile identificate.



Starile identificate sunt:

- **start**: starea initiala a sistemului folosita pentru initializarea algoritmului (pozitia actuala este setata ca fiind pozitia de incepere oferita de test) si se seteaza starea urmatoarea **check right**
- **check right**: starea care marcheaza inceputul unei iteratii; verifica daca exista sau nu un perete in dreapta, daca nu exista un perete seteaza urmatoarea stare **turn right**, daca exista un perete seteaza urmatoarea stare **check fwd**
- **check fwd**: verifica daca exista sau nu un perete in fata, daca nu exista un perete in fata seteaza urmatoarea stare **move fwd**, daca exista un perete seteaza urmatoarea stare **check left**

- **check left**: verifica daca exista sau nu un perete in stanga, daca nu exista un perete in fata seteaza urmatoarea stare **turn left**, daca exista un perete seteaza urmatoarea stare **turn around**
- **turn right**: schimba directia de deplasare prin labirint spre dreapta si seteaza urmatoarea stare **move fwd**
- **turn left**: schimba directia de deplasare prin labirint spre stanga si seteaza urmatoarea stare **move fwd**
- **turn around**: schimba directia de deplasare prin labirint opusa directiei curente (dreapta -> stanga, o rotire de 180 de grade) si seteaza urmatoarea stare **move fwd**
- **move fwd**: marcheaza locatia curenta ca fiind parcursa, se deplaseaza o celula in directia de parcurgere in labirint si seteaza starea urmatoare **check end**
- **check end**: starea care marcheaza sfarsitul unei iteratii; verifica daca s-a ajuns la ierirea din labirint, in cazul pozitiv seteaza urmatoarea stare **done**, altfel revine la starea **check right** (incepe o noua iteratie)
- **done**: starea finala a automatului; marcheaza pozitia curenta ca fiind parcursa si semnaleaza finalul algoritmului (folosind flag-ul done).

Separarea starilor **check left**, **turn left**, **check right**, **turn right** si **check fwd** este necesara deoarece automatul depinde de un sistem extern (tester-ul) pentru a prelua informatia depsre labirint (daca exista sau nu perete la o pozitie precizata). Acest sistem are neveoie de un posedge al semnalului de clock pentru a pregatii informatia, deci sistemul trebuie sa astepte un clock cycle pentru a primii informatia inainte de a face orice asertie despre labirint.

Parcurgerea starilor automatului este imeplementata astfel:
- **state** stocheaza starea curenta a automatului si **next_state** stocheaza starea urmatoare a automatului
- fiecare stare seteaza **next_state** la starea urmatoare dorita
- in fiecare clock cycle **state** devine **next_state**, realizand astfel tranzitia intre stari
- in sistem exista un **always** block care depinde de **state** in interiorul caruia se iau decizii diferite in funcie de starea in care se afla automatul

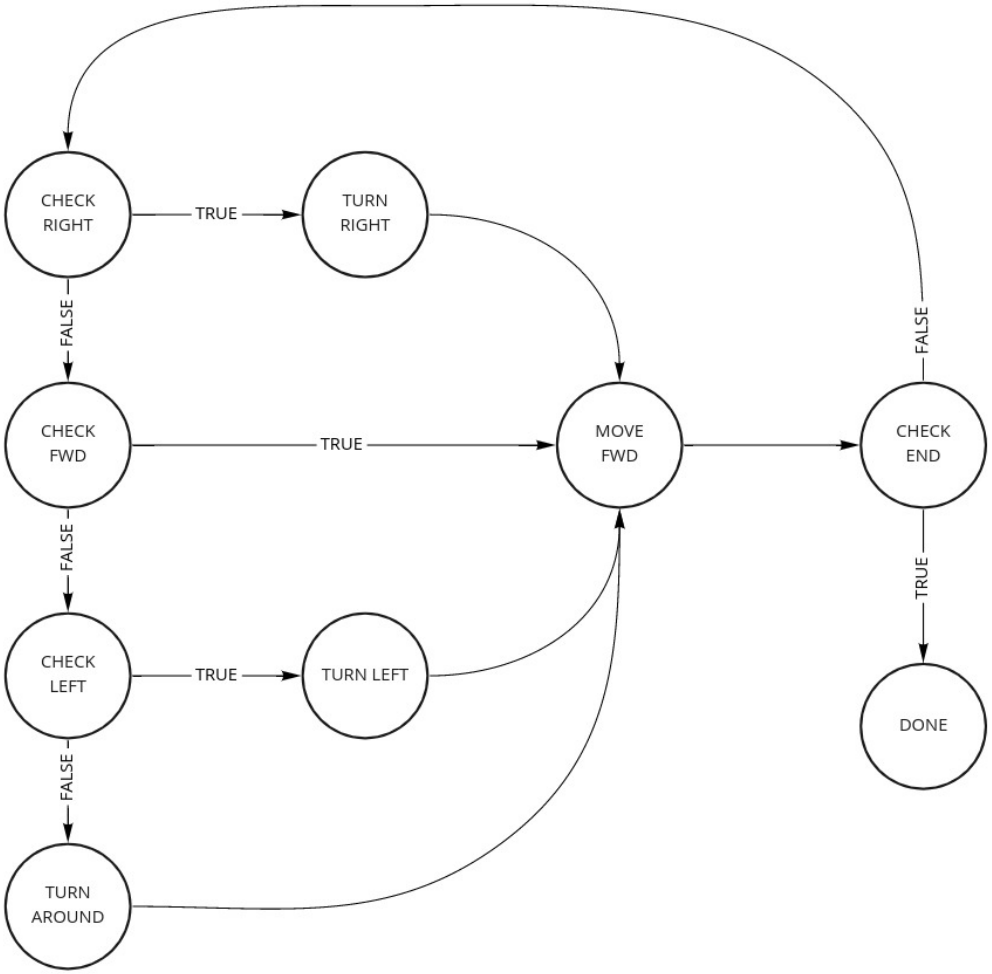
Datorita metodei de tranzitie prin starile automatului, inseamna ca cu cat exista mai multe stari si mai multe tranztitii cu atat sistemul va necesita mai multe clock cycle-uri pentru a rezolva labirintul, practinc setand tinta optimizatilor ca fiind reducerea de stari/tranzitii din automat.

Inainte de a optimiza automatul insa, am decis sa implementez aceasta versiune in Verilog pentru a verifica ca totul va functiona asa cum a functionat in implementarea din javascript si pentru a avea o masuratoare de baza pentru a verifica eficienta algoritmului dupa optimizari.

Am ales doua teste pentru a obtine o masuratoare de baza: cel mai simplu test (base) pentru a putea vedea usor diferentele de clock cycle-uri, si un test moderat de complex (simple4) pentru a vedea imbunatatirea cantitativa a performantei.

Algoritmul de baza (implementarea se poate vedea aici) a reusit sa finalizeze testul base in **35 clock cycles** si testul simple4 in **1432 clock cycles**.

Negoita Mihaela
331AA



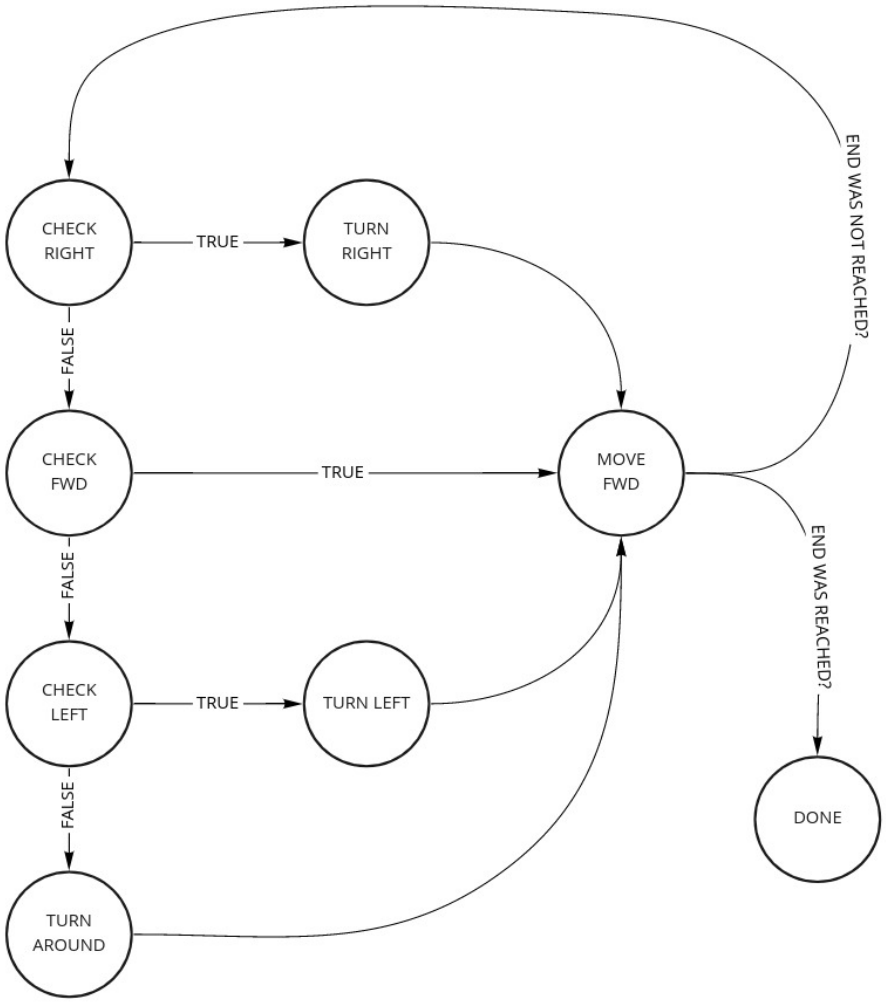
Optimizarea #1

Am inceput prin a elimina starea **start**. Am mutat logica din aceasta stare in starea **check right**. Am adaugat un flag numit **init_flag** initializat cu 0. Starea **check right** verifica acesta flag inainte de a executa restul logicii. Daca flag-ul nu este setat, atunci se executa logica de initializare care era in starea **start** si apoi se seteaza flag-ul cu 1. Acest lucru rezulta in executarea logicii din starea **start** o singura data, indiferent de cate ori se va executa starea **check right**.

Aceasta optimizare nu are legatura cu iteratia efectiva a algoritmului, asa ca nu a facut o diferenta semnificativa, reducant numarul de clock cycles doar cu 1.

miro

Negoita Mihaela
331AA



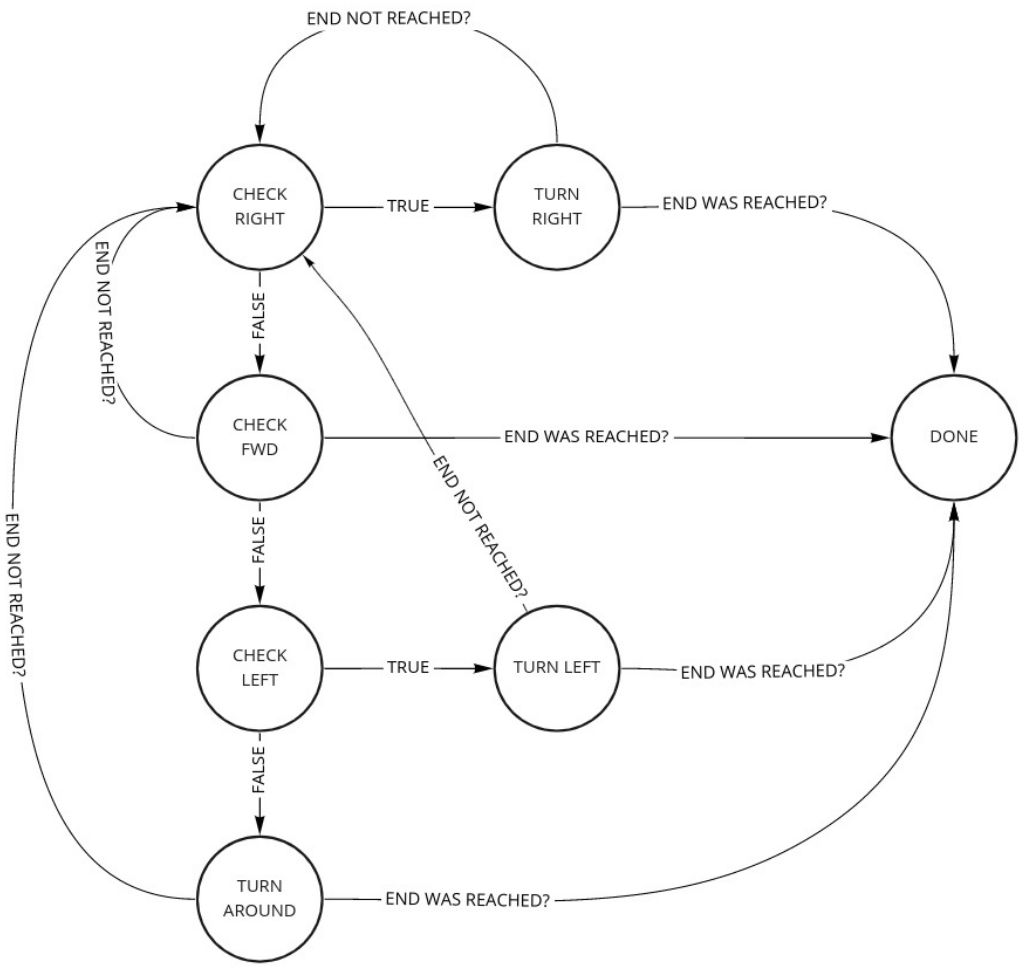
Optimizarea #2

Am eliminat starea **check end**. Am mutat logica din aceasta stare intr-un task numit **check_end**, apoi am apelat acest task la finalul starii **move fwd** prin care ajunge executia indiferent de starile anterioare.

Starea **check end** era setata la fiecare iteratie, eliminand-o, am obtinut o reducere a numarului de clock cycles cu 1 per fiecare iteratie.

miro

Negoita Mihaela
331AA



Optimizarea #3

Am eliminat starea **move fwd**. Am mutat logica din aceasta stare intr-un task numit **move_forward**, apoi am apelat acest task la finalul starilor **turn right**, **chek fwd**, **turn left** si **turn around**. De asemenea am apelat si **check_end** la finalul acestor stari.

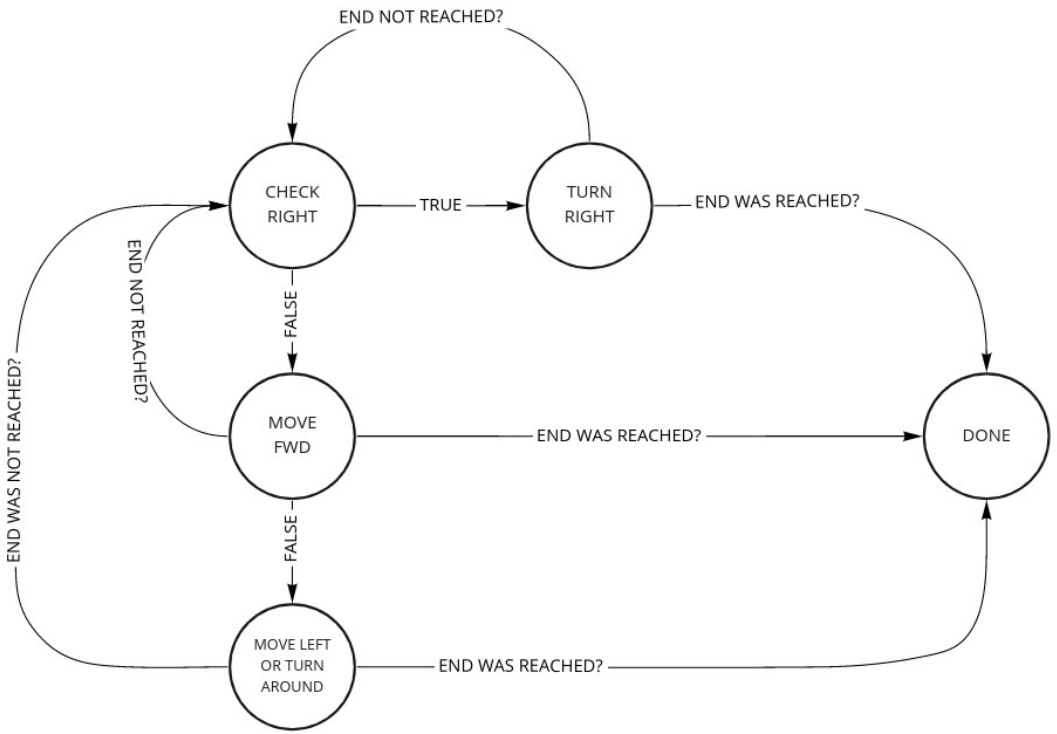
Starea **move fwd** era setata la fiecare iteratie, eliminand-o am obtinut o reducere numarului de clock cycles cu 1 per fiecare iteratie.

Negoita Mihaela
331AA

Optimizarea #4

Am eliminat starile **move left** si **turn arround**. Am mutat logica din aceste stari in interiorul starii **check left**, eliminand astfel 2 tranzitii.

Aceasta optimizare reduce numarul de clock cycles cu 2 per fiecare iteratie.



miro

Negoita Mihaela
331AA

Concluzii

Dupa optimizari am rulat din nou testele base si simple4 pentru a observa diferentelese si am obtinut urmatoarele rezultate

- base: 20 **clock cycles** (redus de la 35, imbunatatire de 42%)
- simple4: 859 **clock cycles** (redus de la 1432, imbunatatie de 40%)

Am rulat si restul testelor si am obtinut urmatoarele rezultate:

Test	#clk inainte de optimizare	#clk dupa optimizare	Diff %
base	35	20	42,86
simple1	448	269	39,96
simple2	1053	630	40,17
simple3	2116	1269	40,03
simple4	1432	859	40,01
complex1	12185	7290	40,17
complex2	14874	8899	40,17
complex3	12271	7344	40,15
complex4	9635	5764	40,18
complex5	14605	8740	40,16

In medie algoritmul optimizat este mai eficient cu 40,39% fata de algoritmul de baza.

In concluzie am reusit sa optimizez automatul initial fara a duplica cod prin eliminarea starilor inutile si refolosind task-uri.