Linköpings Universitet
Campus Norrköping
ITN/TNE094. Digitalteknik och konstruktion
Projektarbete
HT2023

Game of Life

Ett projekt i digitalteknik och konstruktion

Ett projekt av:
Mohammed Al-Obaidi
Esaias Ernfridsson
Maximilian Hallqvist
Jotham Kidane

Sammanfattning

Conway's Game of Life är en simulering av celler på ett rutnät, där varje cell kan vara i ett av två tillstånd: levande eller död. Denna simulering följer grundläggande regler som styr cellernas beteende baserat på antalet grannar de har i det omgivande rutnätet. De enkla reglerna ger upphov till komplexa och ibland oväntade mönster och beteenden, vilket har gjort spelet till en populär plattform för att utforska egenskaper i cellulära automater.

Målet med detta projekt är att implementera Game of Life på en FPGA (Field-Programmable Gate Array) och göra simuleringen synlig på en bildskärm via dess VGA-port. För att kunna implementera Game of Life på en FPGA så är det viktigt att kunna förstå och översätta de grundläggande reglerna för Game of Life till en konfiguration som är optimerad för FPGA-miljön.

Innehållsförteckning

1 Inledning

John Horton Conway hade som mål att utveckla en oförutsägbar cellulär automat, en cellulär automat som simulerar “liv”, med oändlig tillväxt och utveckling. John lyckades med det han föresatt sig för att göra genom skapandet av Game of Life, en Turing komplett cellulär automat. Om en enhet eller ett program klassificeras som Turing komplett betyder det att det kan användas för att simulera vilket annat datorprogram eller system som helst och har de nödvändiga funktioner för att hantera komplexa beräkningar.

1.1 Syfte

Syftet med projektet är att undersöka och implementera *Conway's Game of Life* på en *Field programmable gate array*, FPGA, av modell *DE10 Lite* från *Intel*.

1.2 Metod

Metoden för att uppfylla syftet är en faktainsamling kring ämnets olika delar från internet, föreläsningsanteckningar från kursen digitalteknik och konstruktion samt Naveen Kumar Dasanadoddi Venkategowda, universitetslektor vid Linköpings Universitet, Campus Norrköping.

Projektets kritiska delar involverar implementering av reglerna *Conway's game of life* är uppbyggt av, generering av startposition och en intuitiv visning av resultatet. Dessa delar skapas med kod i språket *SystemVerilog* som sedan implementeras på FPGA'n.

2 Teoretisk Ram

2.1 Game of Life

"Conway's Game of Life" är en cellulär automat utvecklad av den brittiska matematikern John Horton Conway under 1970-talet. En cellulär automat är i grund och botten en simulering av celler på ett rutnät, i Conway's Game of Life, så har varje cell två möjliga tillstånd, död eller levande.

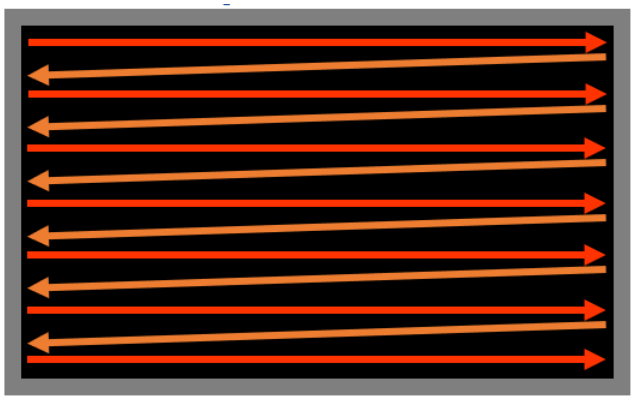
Beteendet hos cellerna som befinner sig i rutnätet bestäms av fyra enkla regler: En levande cell med färre än två levande grannar dör, en levande cell med två eller tre levande grannar lever vidare till nästa generation, en levande cell med mer än tre levande grannar dör och en tom cell med exakt tre levande grannar blir en levande cell.

Det finns några återkommande mönster som ofta förekommer i Game of Life. Dessa mönster faller under tre olika kategorier som kallas "Still Lifes", "oscillators" och "Space Ships". Still Lifes är mönster som ej ändrar form, Oscillators däremot är "oscillerande" mönster som efter ett antal generationer återgår till deras ursprungliga mönster/form. Tredje kategorin, "Space Ships", avser alla former som förflyttar sig över rutnätet utan att förlora sina former.

2.2 Visning av resultat

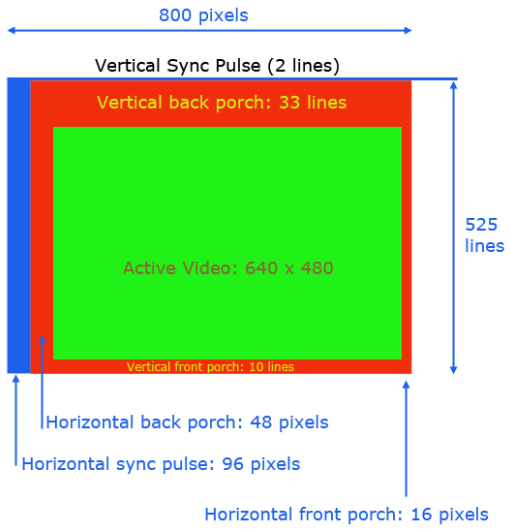
Inbyggt på DE10an finns en port för *Video Graphics Array,* VGA, tillsammans med en *digital to analog converter*, DAC, som tillsammans används för att kommunicera med en skärm [1]. I manualen förklaras även kortfattat hur VGA-protokollet fungerar samt vad portarna kallas i SystemVerilog [1].

Bilden genereras på skärmen från vänster till höger en pixel i taget. När en rad av pixlar är klar påbörjas nästa rad, se Figur 2.2.1. Signalen som skickas för att åstadkomma detta består av fem delar, röd, grön och blå i analoga värden från 0 – 0.7 V samt *horizontal sync* och *vertical sync*. Färgsignalerna ger tillsammans färgen åt en pixel medan horizontal sync och vertical sync signalerar att en ny rad har påbörjats respektive en ny *frame*, en ny bild [2].

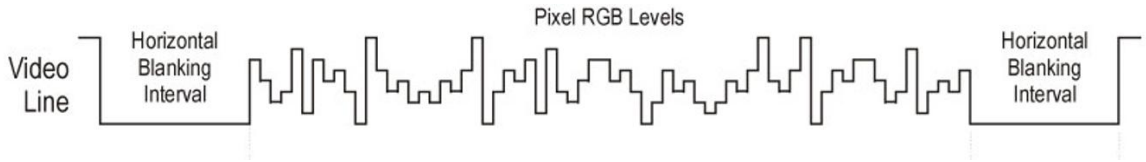


Figur 2.2.1 Visualisering av hur en bild genereras

Enligt [3] kan protokollet visualiseras i ett tidsdiagram, se Figur 2.2.2. Detta innebär att signalen som ska skickas bara har data i form av röd, grön och blå under vissa tillfällen. Resultatet blir en signal som i Figur 2.2.3. De exakta tiderna för protokollet syns i bilaga 6.1.



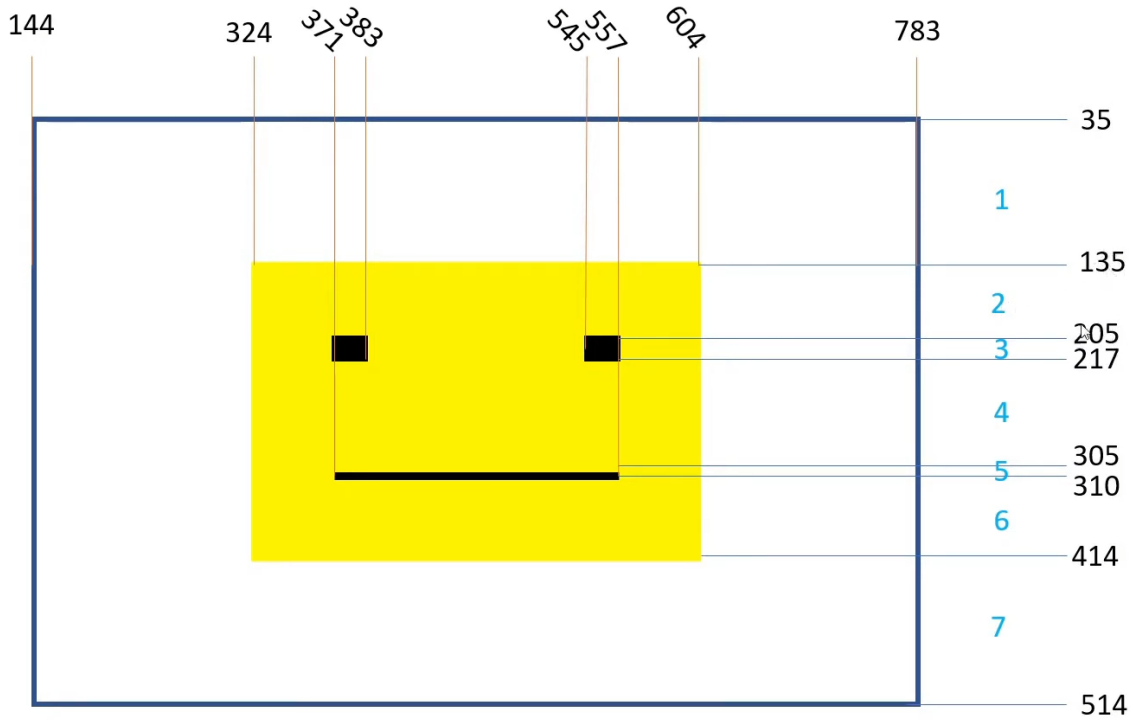
Figur 2.2.2 timing diagram för VGA



Figur 2.2.3 VGA videosignal

Koden som krävs för att generera signalerna innefattar en *clock divider*, räknare, funktioner för horizontal och vertical sync samt en funktion för att beräkna när data ska skickas till skärmen. En *clock divider* använder den interna klockan för att generera en ny frekvens [3]. Den interna klockan på FPGA'n från 50Mhz [1] till 25,175 MHz enligt tabellen i bilaga 6.1, men 25Mhz anses vara tillräckligt nära [3]. Detta innebär att frekvensen endast behöver halveras. Räknarna innefattar pixel- och linjeräknare som används i funktioner för horizontal- och vertical sync. Dessa används sedan för att räkna ut när färgvärden ska skickas till skärmen [4].

För att skapa bilden som ska visas kan en serie av villkors-satser användas för att dela in skärmens pixlar i områden. För att åstadkomma detta används två räknare, en för pixlar respektive en för rader [5].



Figur 2.2.4 Smiley genererad med en serie villkorssatser [5]

3 Genomförande

3.1 Simulationens logik

Displayens pixlar delas in i större rektanglar kallat celler som utformar ett rutnät. Vardera cell behandlas av en designerad modul som implementerar och uppdaterar varje cells status kontinuerligt. Denna modul behandlar varje individuell cells åtta omringande celler (grannar). Modulen tilldelas utöver detta en egen *klocka* samt ett slumpmässigt värde att ge cellen vid återställning. Instanserna av modulen skapas i en *generate sats* vars syfte är att skapa kodblock iterativt d.v.s. skapa flera instanser av modulen Cell. Grannarna till varje cell väljs ut med koordinater relativa till cellens egen position enligt tabell 1. Dessa valideras noga för att inte hamna utanför dataflödet och därmed undvika oförutsägbara fel. I cellmodulen summeras antalet levande grannar som därefter används i implementeringen av spelets regler.

Tabell 3.1.1 Tilldelning av grannar till cell

X (x,y)	X (x,y)	X (x,y)	X (x,y)	X (x,y)
X (x,y)	Granne (-1,-1)	Granne (0,-1)	Granne (1,-1)	X (x,y)
X (x,y)	Granne (-1,0)	Cell (0,0)	Granne (1,0)	X (x,y)
X (x,y)	Granne (-1,1)	Granne (0,1)	Granne (1,1)	X (x,y)
X (x,y)	X (x,y)	X (x,y)	X (x,y)	X (x,y)

3.2 Slumpad start

I början av varje ny simulering krävs att celler placeras i rutnätet. Detta är eftersom en cell endast kan födas då den har precis tre levande grannar runt om sig, d.v.s. utan initiala celler kan inga nya uppstå. Lösningen, en slumpmässig-tal-generator skapar ett binärt nummer med lika många bitar som varje rad har celler. Varje cell tilldelas korresponderande bit från talet och proceduren upprepas radvis i rutnätet. På så sätt bildas en slumpmässig uppsättning celler vid start av simulering och återställning.

3.3 Video Graphics Array

På grund av begränsad tid användes en exempelkod från hemsidan GitHub skriven av alias Icbeams [4]. Projektet innehåller flera filer, däribland en VGA-kontroller samt en exempelkod för att generera ett mönster med färg på skärmen. Koden till VGA-kontrollen innefattar delarna uppskrivna i del 2.2 och skapar protokollets tidsdiagram.

Modulen för att generera klockfrekvensen har input i_clk och i_rst_n, *input clock* respektive *input reset*. Reset är definierad som en invers, systemet ska starta om då reset är falskt. Logiken inuti kan beskrivas som: om i_rst_n är sann, får o_clk, den nya halverade *output clock,* inversen av o_clk's nuvarande värde. Eftersom detta ska generera en reguljär klockfrekvens används en *D-flip-flop*, en simpel minneskrets, [6] som aktiveras på i_clk's positiva flank eller resets negativa flank. Se modulen *pixel_clock_generator* i bilaga 7.2. Klockfrekvensen som skapas används till andra modulen, *sync_pulse_generator*, där protokollet skapas.

För att räkna antalet pixlar används en *D-flip-flop* som aktiveras på klockans positiva flank eller reset´s negativa flank. Denna räknar upp från 0 till 800, se Figur 2.2.2, och sparar värdet i variabeln *pixel_*count. När maxvärdet 800 är nått återgår den till 0 nästa flank. Räknaren för linjer fungerar enligt samma princip, men med det extra villkoret att *pixel_count* har nått sitt maxvärde på 800. Den aktuella linjens nummer sparas i variabeln *line_count*.

Signalerna *horizontal-* och *vertical sync* skapas genom att även här använda två *D-flip-flops*. Signalen *horizontal sync* har villkoret att om *pixel_count* är lika med 96, antar den värdet true tillsdess att *pixel_count* når sitt maxvärde och återgår till false. *Vertical sync* signalen skapas på samma vis, men tar i stället hänsyn till *line_count* då den är lika med 2, se bilaga 7.1, tabell 7.1.2.

Med samma metod som tidigare beräknas tiden där data ska skickas till skärmen med villkor baserat på tabell 7.1.2, i bilaga 7.1 samt i figur 2.2.2.

Alla *D-flip-flops* nämnt i stycket tar även hänsyn till en gemensam reset, *i_rst_n,* för att kunna starta om hela systemet. Modulen *sync_pulse_generator* skickar ut signalerna för *vertical-* och *horizontal sync* samt signalerna för när färgdata ska skickas till skärmen.

3.4 Sammansättning av spellogik och VGA-kod

Med spellogiken och VGA-koden färdigställda saknades endast kod för att göra simuleringen synligt på bildskärmen. Denna kod består av två huvuddelar: uppdelning av skärmen och visning av färg.

Pixlar på bildskärmen itereras igenom för varje pixel där positionen sparas i variablerna *pixel_count* och *lines*. En *always_comb* sats används för att implementera kombinatorisk logik som konstant behandlar dessa värden. Därefter hittas korresponderande cell i rutnätet där dess levnadsstatus kontrolleras. Om en cell är vid liv designeras den färgen svart och om död vit.

Logikens syfte är att uppskatta hur många cellbredder in pixeln befinner sig i rutnätet, både vertikalt och horisontellt. Efter att den sista pixeln har räknats så börjar loopen om igen.

Här används en for loop för att för att skapa kombinatorisk logik för att översätta positionen på displayen till en cell (index) i rutnätet.

För färg så bestämdes det att fyra bitar skulle användas, detta främst då många färger ej behövs, det lämnar även dörren öppen för fler färgvariationer om det skulle behövas. Färgerna determineras utifrån värdena på tre rader; raden för rött, raden för grönt och raden för blått. Beroende på hur höga värden man sätter på varje rad så kan man få fram upp till 16 olika nyanser per färg. Den vita bakgrunden fås genom att sätta de tre färgraderna till deras maxvärden. Den svarta cellfärgen uppnås genom att göra motsatsen, alltså alla tre raderna till minimivärdena. Slutligen då cellerna inte försvinner vid bortgång så har vi att dem antar samma färg som bakgrunden.

4 Resultat

Projektets syfte, skapa *Game of Life* på en FPGA, uppnåddes. Cellerna följer spelreglerna och resultatet visas på en skärm. Cellerna får sina färger beroende på deras status, svart om cellen är levande och vit om den är död. Några generationer efter den slumpade starten uppleves detta som en vit bakgrund med svarta, levande celler. Efter några minuter med den nuvarande storleken på rutnätet nås ett stabilt stadie.

I nuläget simuleras 768 celler där varje cell är 20 gånger 20 pixlar. Rutnätet har höjden 24 celler och bredden 32 celler. Detta valdes på grund av att skärmen är rektangulär, men cellerna är kvadratiska. Detta är inte nödvändigt men ansågs se bättre ut. Antalet celler i rutnätet och hur många pixlar på skärmen varje cell har kan enkelt ändras på för att simulera fler. Dock har gränsen för antalet celler som kan simuleras inte testats på grund av kodens kompileringstid på ca. 7 minuter.

5 Diskussion

5.1 Vad kan bli bättre

Det finns en del som går att förbättra eller bygga vidare på. Som tidigare nämnt är större rutnät något som skulle kunna göras. Ett sätt att göra detta är genom att öka bildskärmsupplösningen så fler celler får plats på skärmen, ett annat sätt är att minska storleken på cellerna. Fler celler ger en längre simulering samt mer utrymme för tillägg.

Seeds är något som är användbart i fall där det velas ha scenarion som går att replikera eller strukturer som tar lång tid att simulera fram. *Seeds* är i detta fall en startposition för levande celler vid simuleringsstart. Då cellerna alltid börjar på samma ställe så kommer simuleringen alltid vara densamma.

En större förbättring som kan göra simuleringen mer intressant är distinkta celltyper. Olika celltyper kan ha unika drag såsom olika färger och spelregler. En potentiell regel kan vara att en celltyp behöver färre grannar för att födas, en annan att cellen kan klara sig en period utan grannar. Vissa grupper av celler kan programmeras att, när lagda kriterier uppfyllts, bilda större komplexa strukturer, så kallade moderskepp.

Ett mörkt läge där bakgrunden är i svart kan vara något användare föredrar så att ha det som alternativ må vara till fördel.

5.2 Metod

Då syftet med detta projekt var att bland annat utöka sina kunskaper i SystemVerilog, valdes i första hand att översätta kod och logik från tidigare lärda programmeringsspråk. Därför var metoden som i början användes att försöka bygga upp spelet utan assistans då konceptet i grunden var förstådd. Delar av koden såsom synkroniseringen för VGA var för stora och komplexa för projektets tidsram och kopierades därav mestadels från en GitHub-källa för att användas som bas. Därefter i mål att bättre adaptera koden till vårt projekt så omskrevs delar av koden stegvis. Bit för bit togs rader bort för att se om de var nödvändiga. Likaså skrevs nya rader in som bättre uppfyller projektets behov.

6 Referenser

[1] Terasic, ”DE10-Lite User Manual,” 2020. [Online]. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiM1pHTqZ6DAxVcHRAIHftfCuYQFnoECBAQAQ&url=https%3A%2F%2Fwww.terasic.com.tw%2Fcgi-bin%2Fpage%2Farchive_download.pl%3FLanguage%3DChina%26No%3D1021%26FID%3Da13a2782811152b477e60203d34b1. [Använd 23 11 2023].

[2] E. Sanchez, ”http://lslwww.epfl.ch,” - - -. [Online]. Available: http://lslwww.epfl.ch/pages/teaching/cours_lsl/ca_es/VGA.pdf. [Använd 23 11 2023].

[3] *Simple Tutorials For Embedded Systems, How to Create VGA Controller in Verilog on FPGA? | Xilinx FPGA Programming Tutorials,* 2018.

[4] Icbeams, *DE10-Lite_VGA,* 2021.

[5] *Dom, VGA Image Driver (Make A Face) On Intel FPGA,* 2020.

[6] S. L. Harris och D. M. Harris, Digital Design and Computer Architecture, ARM Edition, San Francisco: Morgan Kaufmann, 2015.

[7] ”Tinyvga,” SECONS Ltd, 2008. [Online]. Available: http://tinyvga.com/vga-timing/640x480@60Hz. [Använd 20 11 2023].

7 Bilagor

7.1 Tidtabeller för VGA [7]

General timing	
Screen refresh rate	60 Hz
Vertical refresh	31.46875 kHz
Pixel freq.	25.175 MHz

Tabell 7.1.1 VGA Tidtabbel 1

Horizontal timing (line)			Vertical timing (frame)		
Scanline Part	Pixels	Time [µs]	Frame part	Lines	Time [µs]
Visible area	640	25.422045680238	Visible area	480	15.253227408143
Front porch	16	0.63555114200596	Front porch	10	0.31777557100298
Sync pulse	96	3.8133068520357	Sync pulse	2	0.063555114200596
Back porch	48	1.9066534260179	Back porch	33	1.0486593843098
Whole line	800	31.777557100298	Whole frame	525	16.683217477656

Tabell 7.1.2 VGA Tidtabbel 2

7.2 Källkoden

Källkoden hittas på github via länken:

https://github.com/MaximilianHq/Canway-s-Game-Of-Life/blob/main/GameOfLife.sv