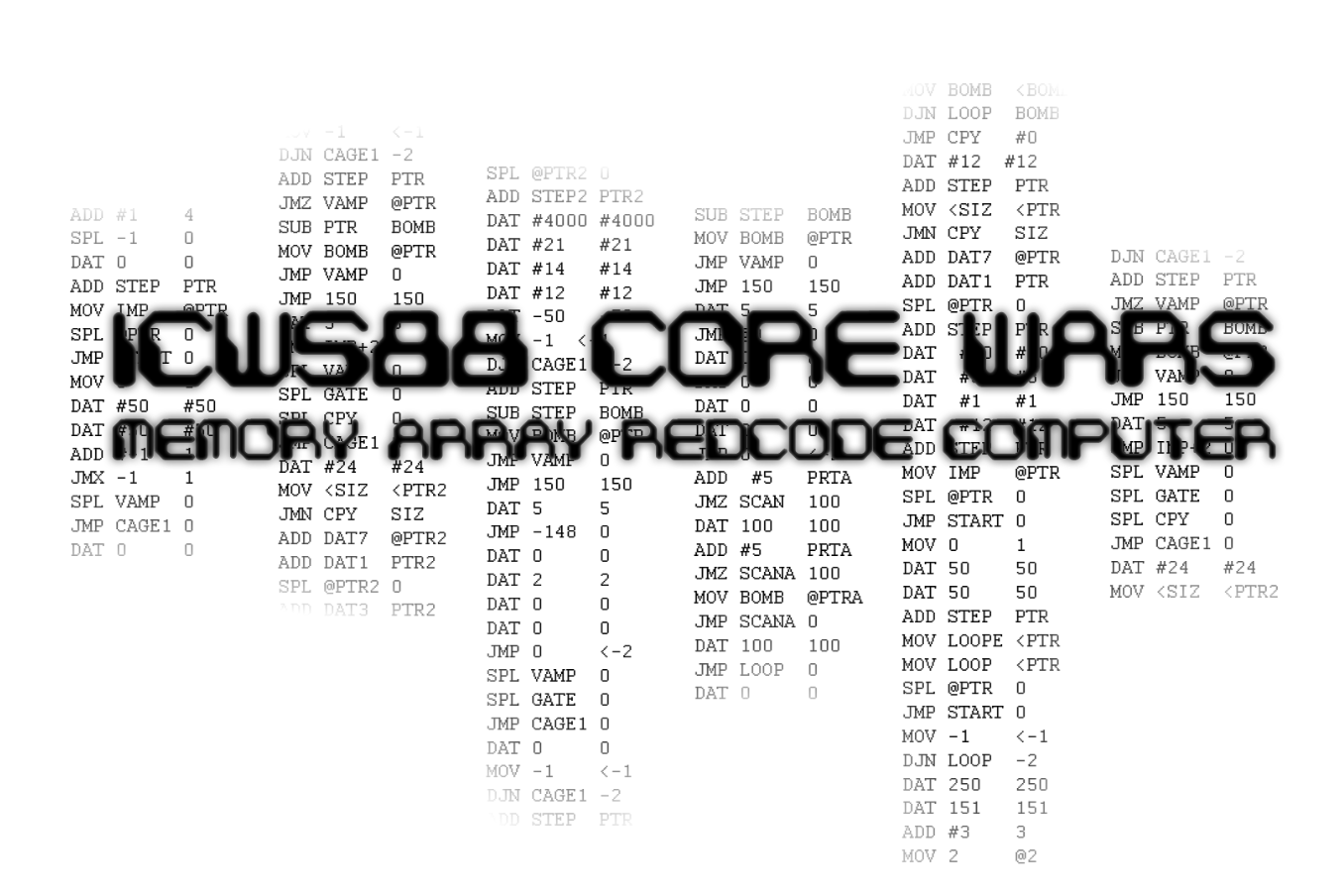
Linköpings universitet

Projekt rapport till digital konstruktion(TSEA43)



Jonas Hietala, jonhi121@student.liu.se

Jesper Tingvall, jesti965@student.liu.se

Jizhi Li, liji050@student.liu.se



15/5 2012

# Sammanfattning

I detta projekt har vi byggt en mikrodator som använder Redcode som assembler. Detta för att kunna spela spelet Core Wars. Vi använder en UART för att skriva in koden till datorns minne och vi kan dumpa ut minnesinnehållet och spelets status på en skärm genom VGA-porten. I rapporten går vi igenom lite Redcode, beskrivning av hårdvara till vår mikrodator och hur vi använder RS232 och VGA standarden. Till sist har vi en del exempel Warriors som visar de vanligaste Core Wars strategierna.

Innehållsförteckning

[Sammanfattning](#_Toc324712586)

[1 Inledning 1](#_Toc324712587)

[2 ICWS88 Redcode 2](#_Toc324712588)

[2.1 Introduktion 2](#_Toc324712589)

[2.2 Exempel Warriors 2](#_Toc324712590)

[3 Teori 4](#_Toc324712591)

[3.1 VGA 4](#_Toc324712592)

[3.2 RS232 5](#_Toc324712593)

[4 Beskrivning av hårdvara (M.A.R.C) 6](#_Toc324712594)

[4.1 µDator (ALU) 6](#_Toc324712595)

[4.2 VGA 7](#_Toc324712596)

[4.3 Minnen 9](#_Toc324712597)

[4.4 UART 11](#_Toc324712598)

[4.5 FIFO 12](#_Toc324712599)

[5 Slutsatser 13](#_Toc324712600)

[6 Referenser 14](#_Toc324712601)

[6.1 Redcode 14](#_Toc324712602)

[6.2 VHDL-kod 15](#_Toc324712603)

[6.3 Mjukvaralistor 15](#_Toc324712604)

[6.4 Script 15](#_Toc324712605)

[6.5 Övriga filer 15](#_Toc324712606)

# 1. Inledning

Vårt mål med projektet i denna TSEA43 kurs var att bygga en dator som kunde köra det eminenta spelet Core Wars. Core Wars är ett ointeraktivt spel i vilken spelarna skriver sina program i Redcode assembler. Målet var att bygga en maskin som använde Redcode som sin assembler och som kunde måla ut spelområdet, d.v.s. minnet, till en VGA skärm och ta emot ny kod via en UART. För mer utförlig information om våra designmål rekommenderas en läsning i vår designskiss[[1]](#footnote-1).

Vi namnger vår dator till M.A.R.C, Memory Array Redcode Computer då simulatorn heter M.A.R.S, Memory Array Redcode Simulator. Värt att nämna är att Core Wars ej refererat till processor kärnan utan till ett gammalt kärnminne.

Vårt mål är att kunna spela Core Wars enligt 1988 standarden[[2]](#footnote-2), skicka in innehåll till M.A.R.C minnet från en kontrolldator genom UART porten och sätta ut två spelares position. Vi vill även kunna dumpar ut minnesinnehåll och spelstatus till en VGA skärm. Vår uDator skall kunna utföra alla 11 instruktionerna, 4 adresseringsmoderna Redcode har samt kunna växla mellan, skapa och ta bort processer. Vi rekommenderar en läsning utav 1988 standarden då vi ej kommer att gå igenom instruktionerna eller adresseringsmoderna i denna rapport.

# 2. ICWS88 Redcode

## 2.1. Introduktion

Vi har programmerat en assembler som kan generera en binärfil ifrån två Warriors skrivna i Redcode. Vi randomiserar också deras startläge. Vi kan sedan skicka den assemblerade koden och startpositionerna till MARC genom en UART.

## 2.2. Exempel Warriors

Det är en hel vetenskap att skriva effektiva redcode Warriors och vi har endast skrapat på ytan. Vi har dock lyckats programmera ett antal exempel Warriors som demonstrerar de vanligaste strategierna och blandningar mellan dem.

**Factory Bomber**

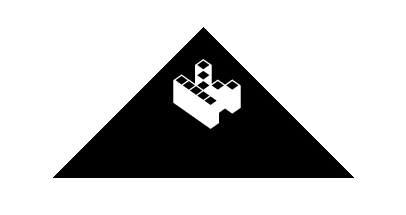
Factory bomber (eller bomber factory då den bygger bombers) formaterar hela minnet via att masskopiera en massa ’little bombers’ till minnet. Dessa databombar minnet och kommer efter ett tag bomba isär orginalkoden. Denna Warrior är därmed en blandning mellan en bombare och en replicator.

**Imp Spawner**

Denna Warrior är ej offensiv och har som stategi att skapar en massa imps. Imp spawner fungerar ungefär som Factory Bomber fast har en annan payload.

**Vampire Bomber Gate Replicator**

Denna otympliga Warrior startade som ett skämt då vi ville se vad som hände om man inkluderade så många strategier som möjligt i en Warrior. Dock blev den inte så dålig som vi först trodde. Först så skapar Warriorn en kopia av sig själv, denna kopia kan dock ej kopiera sig själv, något som borde kunna lösas med hjälp av lite hjärnverksamhet och en texteditor. Efter kopiatorn så har Warriorn en ”bomber cage”, dessa två rader databombar minnet bakåt. Efter cagen kommer vampyrkoden. En vampyr JMP bombar minnet i hopp om att fienden skall hoppa in i dess cage. Den kan därmed sno klockcykler ifrån motståndarens kod. Sist finns en gate ifall resten av koden skulle bli överkörd av en Imp.

**Kopimi**

Denna Warrior scannar minnet efter information, kopierar den och börjar sen exekvera den. Den kan därmed härma en fientlig Warrior om den skulle hitta den. Fungerar skapligt trots att den utvecklades mest för att se vad som hände om man skulle tolka Det Missionerande Kopimistsamfundet missionsbudskap[[3]](#footnote-3); ”Kopiera och sprid” i form av Redcode. Denna Warrior använder replicator strategin.

**Inseminator**

Ännu en Warrior som skapades på skoj men som visade sig vara rätt så effektiv. Den letar upp motståndarens kod och injicerar en massa processer i den i hopp om att motståndaren ej ska förstöra sig egen kod. Detta brukar dock förstöra funktionaliteten i motståndarens kod då den förutsätter oftast att koden exekveras sekventiellt.

**Carpet Bomber**

Carpet bombers är en blandning mellan en bomber och en scanner. De traverserar minnet och lägger in bombers där minnet har ändrats. Denna Warrior är smartare än en vanlig bomber då den inte kommer att bomba ute i tomma minnet. Den kommer också vara lite snabbare än en traditionell bomber som behöver kopiera ut data.

**Core Cleaner**

En core cleaner är ett program som databombar hela minnet. Ofta går man igenom minnet två gånger, den första fyller man minnet med split instruktioner för att slöa ner motståndaren och sedan med DAT-instruktioner för att göra slut på honom.

**Replicator**

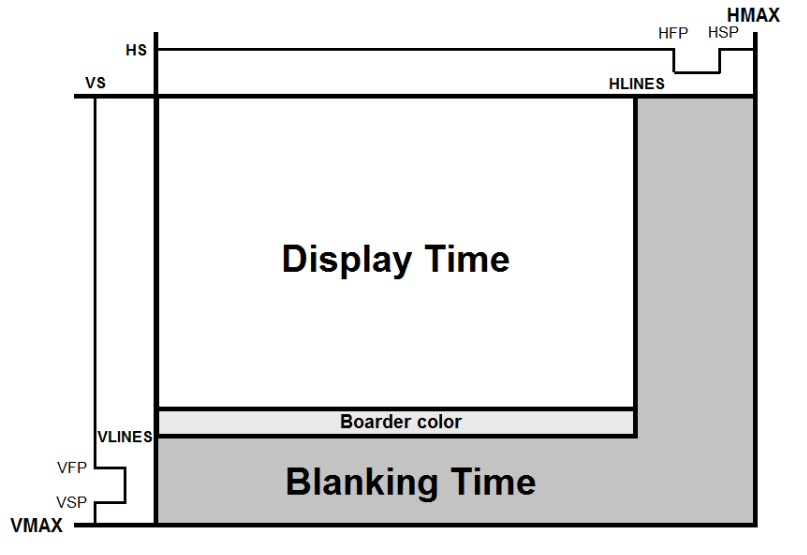
Replicators skapar kopior av sig själva och förökar sig i minnet. De motverkar bombers då bombers inte kan förstöra replicatorn tillräckligt snabbt.

**Dwarf Scout**

En dwarf scout är en vanlig bomber som skyddar sig mot andra bombers genom att se om någon ändrar i minnet i dess närhet. Om så är fallet så kommer den att hoppa till en ny plats i minnet och ta med sig sina processer.

## 3 Teori

### 3.1 VGA



*Figur 1. VGA teori*

När VGA skickar pixeldata till VGA porten kommer skärmen inte ta emot och visa pixel data under hela tiden. Dessutom finns det en speciell timing till olika upplösningar med olika frekvenser. Upplösning 640x480 med frekvens 60Hz har vi följande timing enligt Digilent®.

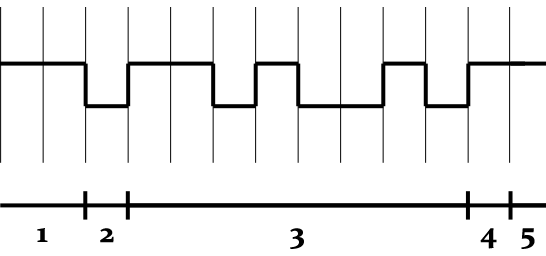
* HMAX: 800
* VMAX: 525
* HLINES: 640
* VLINES: 480
* HFP: 648
* HSP: 744
* VFP: 482
* VSP: 484
* Clk: 25MHz

Orsaken till varför behöver vi har blanking time är att en skärm använder en stråle för att visa varje pixel och strålen flyttar sig från vänster till höger och sedan ner på nästa rad och upprepar denna process. Blanking time är nödvändigt eftersom under blanking time kommer strålen flytta sig från höger till vänster och under denna tid ska skärmen inte visar någon pixel. Mellan front porch och back porch går sync signal ner och upp igen på grund av att det är sync signal som uppdaterar och bestämmer frekvens till skärmen.

På display ytan, kommer varje pixel uppdateras enligt den 8 bitars färg som skärmen har fått genom VGA porten och på blank ytan ska vga porten får ingen färg data alls, annars kommer skärmen visa denna färg när de flyttar sig tillbaka över skärmen.

### 3.2 RS232

Vårt FPGA kort har en USB till RS232 port. Vi använde denna för att föra över den assemblerade spelarkoden till kortet. En överförning inleds av en startbit, därefter följer 8 databitar och en stoppbit. Hastigheten mäts i baud, tecken (på 8 bitar) per sekund. I vårt fall var ledningen hög när ingen överförning var igång (1). Överförningen inleds med att ledningen jordas (2), därefter följer 8 databitar i vald hastighet (3). I slutet av överförningen kommer en stoppbit som är hög (4). Se figur 2.



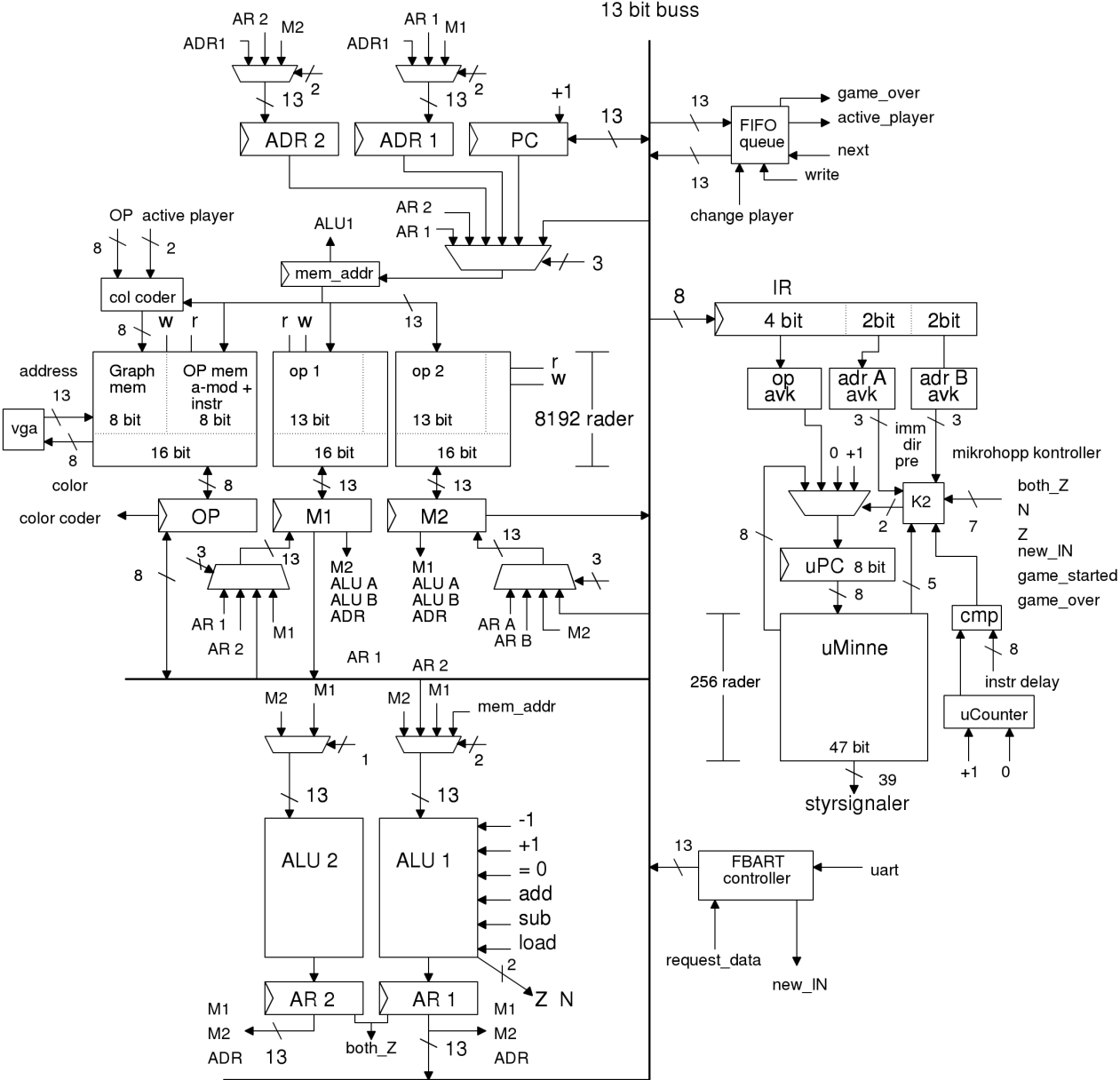
*Figur 2. En RS232 överförning.*

Man har ingen gemensam klocka för sändare och mottagare utan överför endast data, sändaren och mottagaren känner dock till vilken baud rate man överför med. I vårt fall använder vi 115 200 baud.

## 4 Beskrivning av hårdvara (M.A.R.C)

### 4.1 µDator (ALU)

Datorn är en mikroprogrammerad dator med 39 styrsignaler + 8 signaler för hoppaddresser. Mikrominnet är 256 rader långt och mer än 200 rader är använt. Dess huduvuppgifter är att nollställa minnet vid en reset, slussa in program i minnet vid inladdning via fbart och hämtning och exekverande av instruktioner.



*Figur 3. Huvudblockschema*

Blockschemat beskriver vilka register (alla osynliga för programmeraren) som finns och hur de är kopplade med omgivningen. Det finns två ALU:s för att korta ner på antalet klockcykler det krävs för att göra parallella operationer på A och B operanderna. På samma sätt har de flesta registren multiplexade ingångar för att spara tid och för att öka förmågan för parallellism.

Mikrominnet har en mängd olika hopp som den kan göra, den kan bland annat hoppa på både A och B's olika adresseringsmoder eller ALU:ns olika flaggor. För att sakta ner exekveringen fördröjs exekveringen av varje instruktion genom att jämföra en räknare med en fördröjningssignal "instr delay". Detta för man ska kunna följa spelet gång på skärmen.

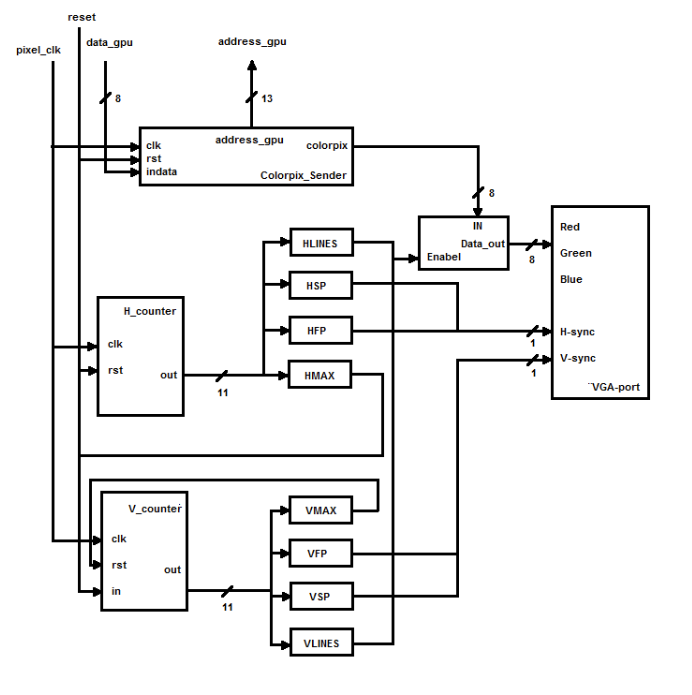
Vid exekvering av en instruktion laddas instruktionen först in till IR, sedan beräknas adresseringsmoderna för A och B och därefter utförs instruktionen. Adressmodsberäkningen är besvärlig då både A och B operanderna kan vara en av de fyra olika moderna. Detta kompliceras ytterligare då vissa instruktioner gör olika saker beroende på vilka adresseringsmoder som används. Efter beräkningen lagras operanderna i M1 och M2, om immediate, och annars i adressregistren ADR1 och ADR2. Schemat visar även var vga, FIFO och fbart controller ansluts.

### 4.2 VGA

VGA är uppdelad i två delar: vga\_controller och pixelsender. Vga\_controller tar hand om timing av signaler till VGA-port och pixelsender använder samma timing som vga\_controller samt hämta färg data urifrån huvudminne. Se figur 2 för detaljer.

I vga\_controller finns det två räknare: h\_counter som räknar antalet horisontella pixlar och v\_counter som räknar antalet vertikala pixlar. Varje gång när h\_counter räknar upp till HMAX, dvs. maximalt antal pixlar på en rad, så kommer h\_counter nollställas och skicka en +1 insignal till v\_counter; v\_counter kommer att nollställas när den uppnå VMAX. (antalet pixel för varje kolumn)

HFP(slutpunkt till horisontal front porch), HSP(slutpunkt till horisontal synkpuls), VFP(slutpunkt till vertikal front porch), VSP(slutpunkt till vertikal synkpuls) kommer vi att använda i vga\_controller. HFP kommer att aktiveras när h\_counter > HFP och skicka jordsignal till H-sync och HSP kommer att aktiveras när h\_counter > HSP eller h\_counter < HFP och skicka högsignal till H-sync. VFP och VSP kommer att skicka sync signal till V-sync med på samma sätt.



*Figur 4. VGAblockschema*

VGA-port kommer endast ta emot färg data när h\_counter < HLINES(640 enligt upplösning vi valde) och v\_counter < VLINES(480 enligt upplösning vi valde) med hjälp av en enable signal från HLINE och VLINE.

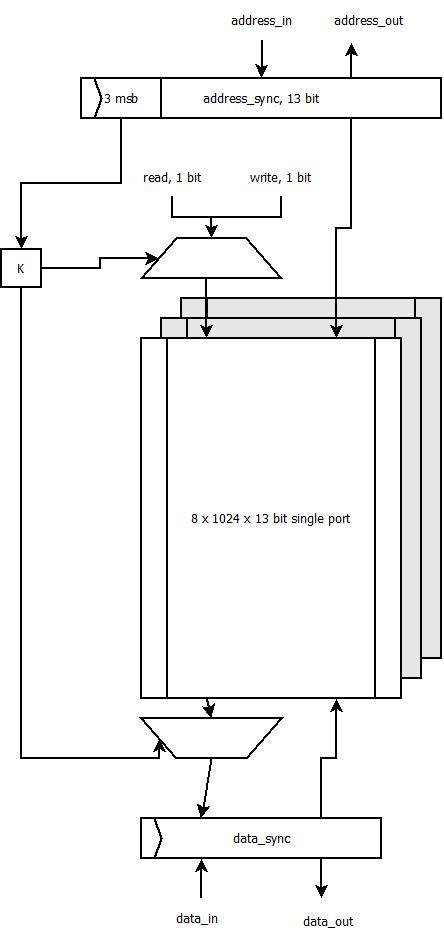
Pixelsender använder samma timing och klocka som vga\_controller och skickar en 13 bitars adress till vårt färgminne, hämtar 8 bitars data på detta adress och då skickar denna data till vga-porten endast när räknare in vga\_controller ligger inom display-ytan.

PixelSender tar hand om address hämtning och färg kod sändning. För att alla data i minnet ska se bra ut på skärmen bestämde vi att visa varje instruktion ska vara 5 pixlar bred och 7 pixlar hög. I så fall kommer vi att visa 128 data per rad och vi behöver 7\*64 = 448 rader för att visa 213 = 8192 adresser. PixelSender skickar data till skärmen var 5:e klockpuls och upprepar detta för varje 128 data 7 gånger, i så fall kan vi ha varje instruktion med 5\*7 pixel storlek. På ”border area” visar vi vilken spelare vinner CoreWar.

### 4.3 Minnen

Vi valde att använda en core size (storlek på spelplan) på 8192 rader, detta brukar vara standard i duell spel men ibland avrundar man till 8000 rader. Om man kör fler än 2 spelare brukar minnet vara betyderlig större, vi ska dock endast ha 2 spelare stöd. Vi behöver enligt (1) 13 bitar för att kunna adressera hela detta område. Då minnet i FPGAN är indelade i block mindre än detta fick vi dela upp minnet på flera block.

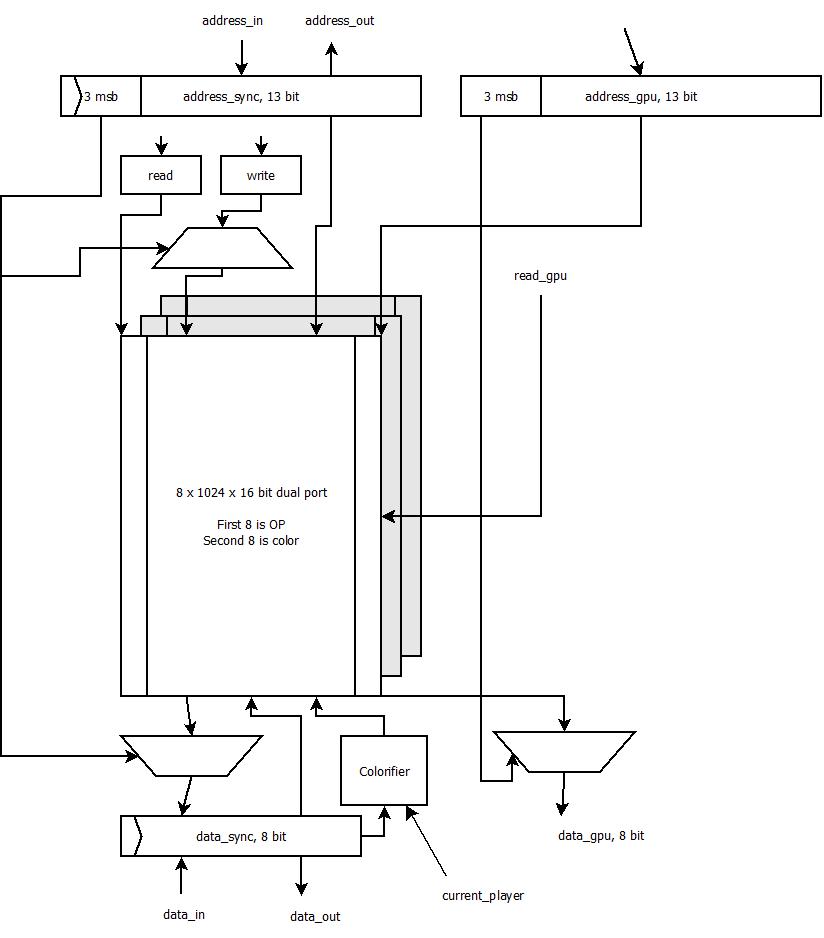
Varje rad Redcode delades upp i 4 delar; instruktion och adresseringsmoder på 8 bitar, operand A på 13 bitar, operand B på 13 bitar och 8 bitar RGB färgning. Det som bäst stämde överens med vår uppdelning var att använda minnesblock utav storleken 1024 x 16 bitar (de 3 sista bitarna används ej dock i operandminnena), se figur 5.



*Figur 5. Operandminnen*

De tre mest signifikanta bitarna styr multiplexern och ser till att rätt minne skriver och läses ifrån. Våra minnen var lite bättre än vad vi först förväntade oss, därför har vi en adress\_sync och data\_sync register, vi skulle kunna ta bort dessa och därmed snabba upp datorns minnesaccess.

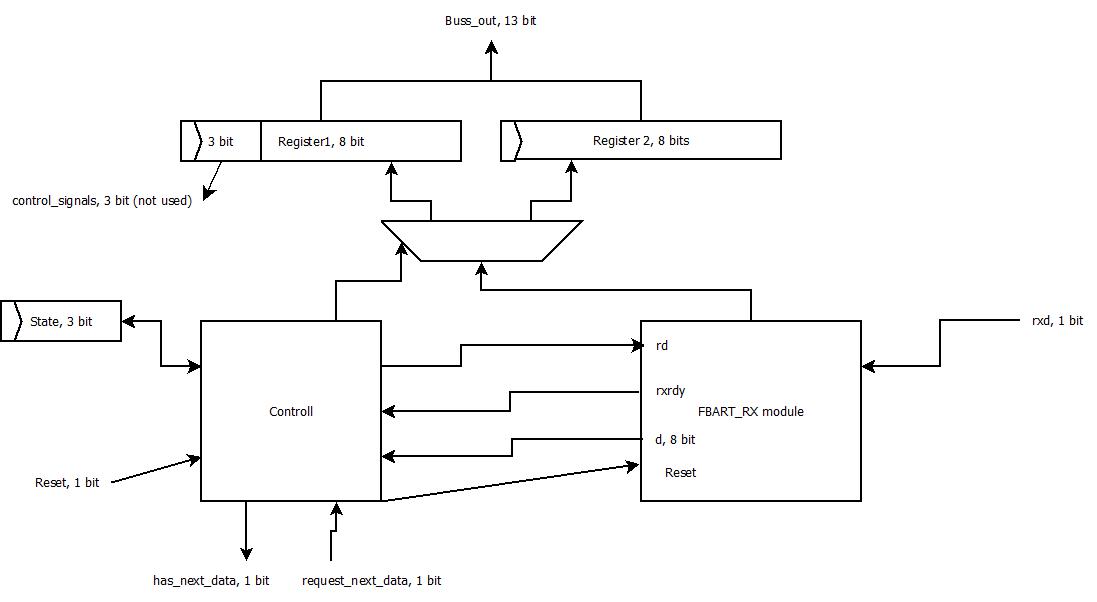
Då vi har olika färg beroende på vilken instruktion vi har i instruktionsminnet var det naturligt att slå samman instruktionsminnet och färgminnet då båda var på 8 bitar. Den resulterande maskinen ses i figur 6. Skillnaden mellan den och operandminnena är att den använder ett dualportminne med den andra adressingången kopplad till GPUn. Färgen skrivs automatiskt till minnet när man skriver in en instruktion i minnet.



*Fig. 6. Instruktions och färgminne*

### 4.4 UART

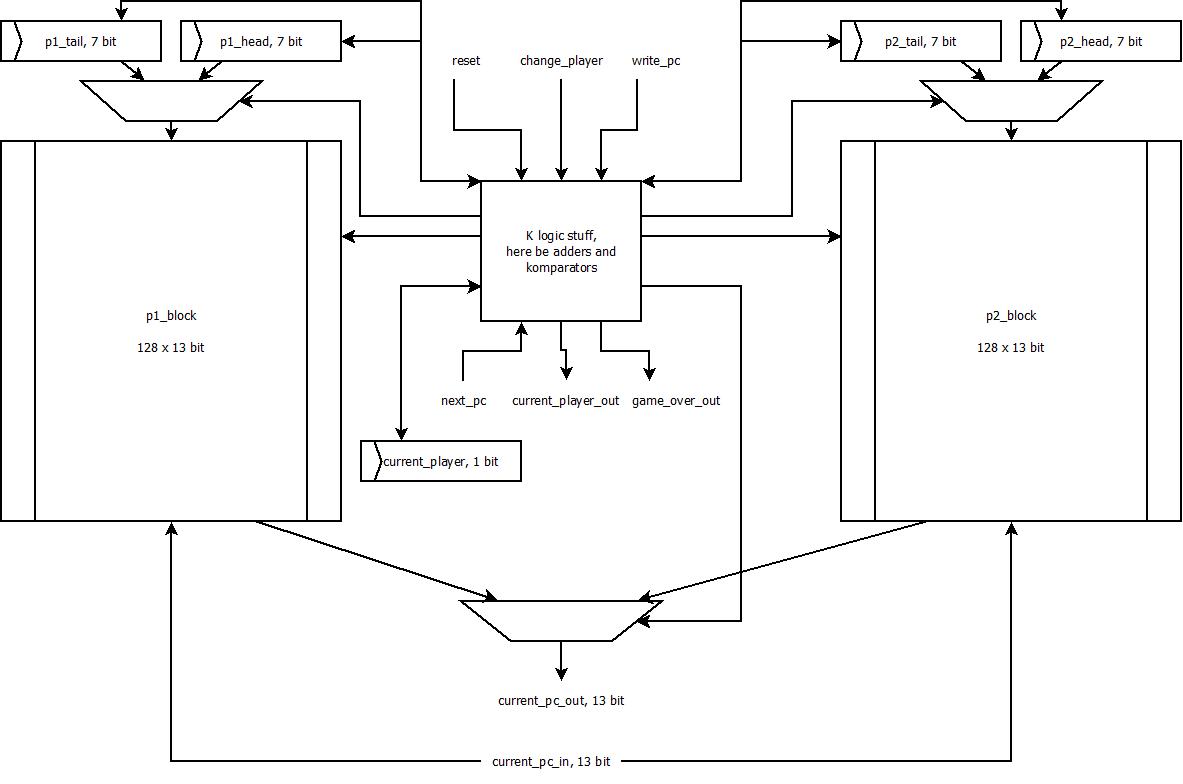
Vår dator använder en 13 bitars buss, det skulle därmed vara trevligt om indatat ifrån vår värddator skulle vara 13 bitar detta med. Då vi använder Anders Nilssons FBART vilken arbetar i 8 bitar skulle det vara trevligt att slå samman två sändningar till en. Det gör vi med modulen i figur 7. Modulen väntar på en databegäran, tar emot två 8 bitars överförningar, slår samman dem till 13 bitar (den kastar iväg 3 bitar) och signalerar att data finns.

*Fig. 7. UART kontrollerare.*

Vi fick även ändra i FBARTen då den gick på en 25 MHz klocka och vårt bygge kör på en 100 MHz klocka. Vi behövde endast öka antalet bitar i en räknare och ändra på en konstant.

### 4.5 FIFO

Då en spelare kan ha flera olika processer igång behöver vi ett sätt att lagra alla programräknare. Vi har implementerat två stycken ”first in first out” köer i vår hårdvara, se figur 8.

*Figur 8. Player FIFOs.*

Headregistret pekar på den översta programräknaren och tailregistret pekar på en sista. När man begär nästa programräknare ökas den nuvarande spelarens head och den översta PCn skrivs till current\_pc\_out. När man skriver in en PC kollas först att den nuvarande spelars kö ej är full, om den ej är full skrivs PC in och tailregistret ökas. Om kön är full görs ingenting. Om någon spelares kö är tom, dvs. headregistret är lika med tailsregistret så signaleras game\_over. Man kan även byta aktiv spelare.

## 5. Slutsatser

Arbetet med projektet gick bra, VHDL var lite motsträvigt men vi lyckades implementera hela CoreWars 1988 standarden och få våra Redcode Warriors att fungera. Core Wars var väldigt kul, både att implementera och att skapa Warriors till.

Implementationen skulle kunna förbättras. Mikrokodningen är inte alls optimerad då det kändes lite onödigt då vi hade en 27 bitars delayräknare efter varje exekverad instruktion. Minnesaccessen skulle kunna förbättras och VHDL koden är onödigt komplex på flera ställen.

Vid fortsatt arbete kan mikrokoden göras snabbare genom mikrokodsoptimering. Till exempel skulle man kunna ha parallell adressavkodning då vi har dubbla ALUs. En nyare standard skulle kunna implementeras då den ger möjligheter till nya variationer av Warriors. Det finns regler om timeouts som vi inte tar hänsyn till. Om kommunikationen till datorn skulle kunna utökas skulle MARC kunna användas som en King of the Hill server för att ställa Warriors mot varandra och ranka dem. Man skulle kunna utöka stödet till mer än två spelare och köra en Free For All. En utökad core size och stöd för fler samtidiga processer skulle kunna läggas till.

# 6. Bilagor

## 6.1 Redcode

j3x-factory-bomber.red

j3x-imp-spawner.red

j3x-inseminator.red

j3x-jumper.red

j3x-jumper-gate.red

j3x-kopimi.red

j3x-little\_bomber.red

j3x-scanner1.red

j3x-vampire.red

j3x-vampire-bomber.red

j3x-vampire-bomber-gate.red

j3x-vampire-bomber-gate-replicator.red

jonas-carpet-bomber.red

jonas-cleaner.red

jonas-dwarfscout.red

jonas-impworm.red

jonas-replicator.red

jonas-stone-of-ages.red

jonas-suck-on-this.red

jonas-survivor.red

jonas-the-big-maker.red

## 6.2 VHDL-kod

*ALU.vhd*

*colorpixSender.vhd*

*FBARTController.vhd*

*fbartrx.vhd*

*MARC.ucf*

*MARC.vhd*

*MARCled.vhd*

*MemoryCell.vhd*

*MemoryCellDualPort.vhd*

*microcontroller.vhd*

*PlayerFIFO.vhd*

*vga.vhd.vhd*

*vgaController.vhd*

## 6.3 Mjukvaralistor

Xilinx ISE Design Suite 13.4 Webpack

ModelSim Student Edition 10.1b

Digilent ADEPT

## 6.4 Script

assembler

control\_code

1. Se bilaga. [↑](#footnote-ref-1)
2. [corewars.nihilists.de/redcode-icws-88.pdf](http://corewars.nihilists.de/redcode-icws-88.pdf) [↑](#footnote-ref-2)
3. kopimistsamfundet.se [↑](#footnote-ref-3)