Linköpings universitet

Projekt rapport till digital konstruktion(TSEA43)

# Core War



Jonas Hietala

Jesper Tingvall

Jizhi Li



Linköpings universitet

Linköping 2012

## Sammanfattning

## Innehållsförteckning

## 1 Inledning

## 2 ICWS88 Redcode

### 2.1 Introduktion

### 2.2 Exempel warriors

### 3 Teori

### 3.1 VGA

### 3.2 RS232

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

### 4.1 µDator (ALU)

### 4.2 VGA

### 4.3 Minne

### 4.4 UART

### 4.4 FIFO

## 5 Slutsatser

## 6 Referenser

### 6.1 Redcode

### 6.2 VHDL-kod

### 6.3 Komponentförteckningar

### 6.4 Programlistor

### 6.5 Exempel warriors

## 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. Huvudmålet var att bygga en maskin som använde Redcode som sin assembler och som kunde måla ut spelomådet, dvs. 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.

Vårt mål är att kunna spela CoreWars enligt '88 standarden och skicka in innehåll till M.A.R.C minnet från PC genom UART porten och sätta två spelare, och då dumpar ut minnesinnehåll och spelstatus. Vår uDator skall kunna utföra alla 11 instruktionerna, 4 adresseringsmoderna samt kunna växla mellan, skapa och ta bort processer. Vi vill också ha möjligt att visa minnesinnehåll och spelstatus tydligt via en extern skärm genom vga porten som ett bonus krav.

## 2 ICWS88 Redcode

### 2.1 Introduktion

CoreWars '88 standarden specifierar utseendet av Redcode, CoreWars programmeringsspråk, som vi har gjort en assembler till. Assemblern kan generera en binärfil från två warriors där vi randomiserar deras startläge som vi sedan kan skicka till MARC genom uart. (referera till bilagan för att se source code!)

### 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 och en bombare.

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 trodde. Först så skapar warriors 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

[KOPIMI]

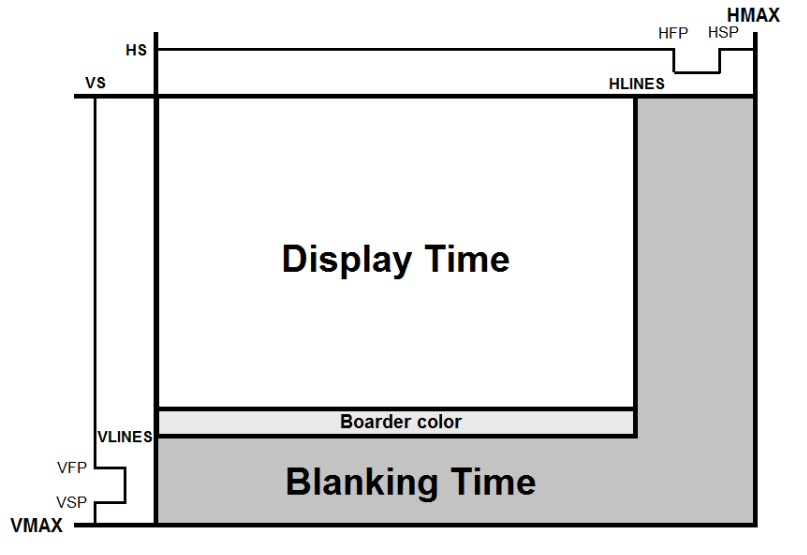
Denna warrior scannar minnet efter information, kopierar den och börjar sen exekvera den. Den kan därmed härma en fientlig warrior om den skulle hitta den. Fungerar skapligt trots att den utvecklades mest för att se vad som hände om man skulle tolka Det Missionerande Kopimistsamfundet missionsbudskap; ”Kopiera och sprid”. Denna warrior använder replicator stategin.

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 egna kod. Detta brukar dock förstöra funktionaliteten i motståndarens kod då den förutsätter oftast att koden exekveras sekventiellt.

### 3 Teori

### 3.1 VGA



*Fig. 7. VGA teori*

När vga skickar pixel data till vga porten, kommer skärm inte ta emot och visa pixel data under hela tiden. Dessutom finns det ett speciallt timing till olika upplösningar med olika frekvenser, till 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 skärm använder strålen för att visa varje pixel och strålen flytter från vänster till höger och då flytter till nästa rad och upprepar denna process liksom hur h\_counter och v\_counter beter sig. Blanking time är nödvändigt eftersom mellan blanking time kommer strålen flytta sig från höger till vänster och under denna tid kan den 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 8 bitar färg som skärm har fått genom vga porten och på blank ytan, ska vga porten får ingen färg data alls, annars kommer skärm(strålen) visar detta färg när de flytter sig 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 3.

[rs232.psd]

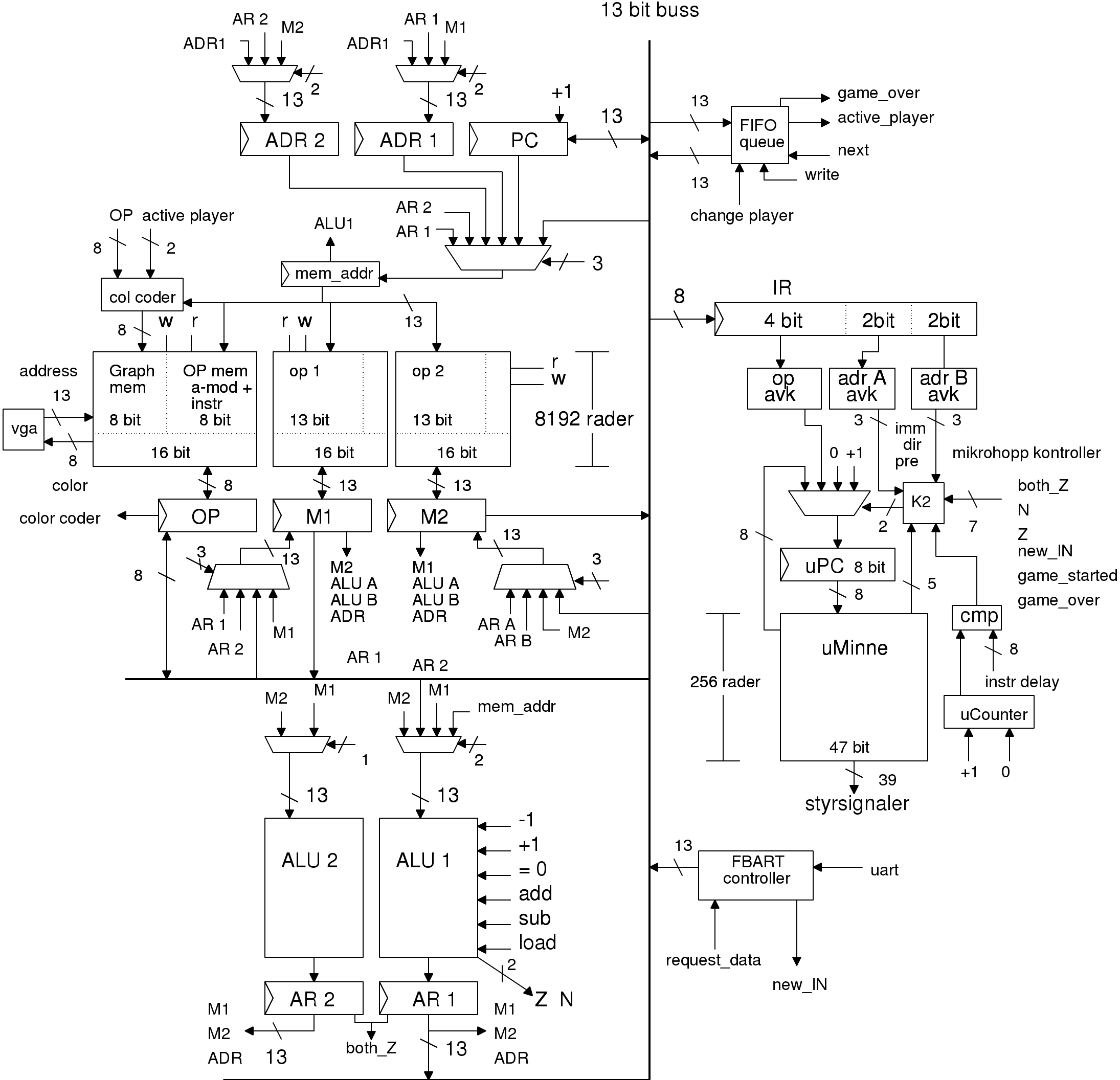
Figur 3. En RS232 överförning.

Man har ingen gemensam klocka 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.



*Fig. 1. 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 parallelism.

Mikrominnet har en mängd olika hopp 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".

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 adressregistrena ADR1 och ADR2.

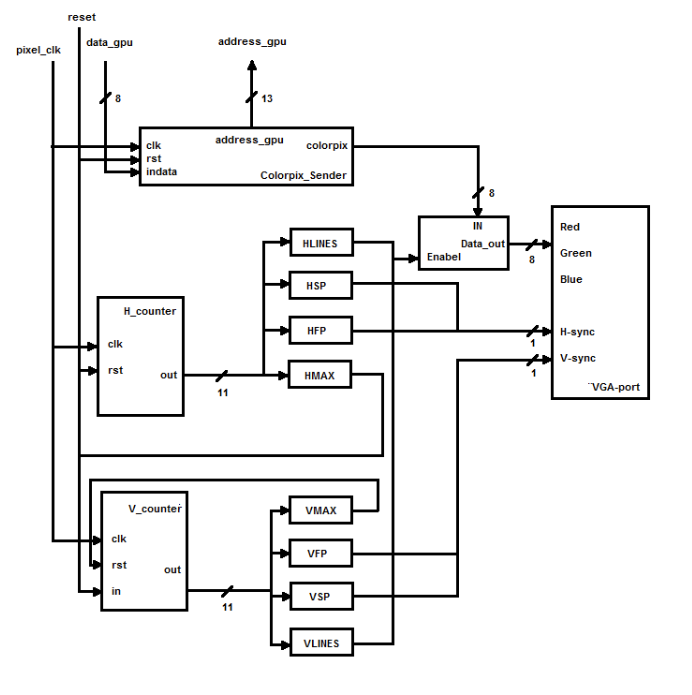
Schemat visar även var vga, FIFO och fbart controller ansluts.

### 4.2 VGA

VGA är delade upp till två delar: vga\_controller och pixelsender. Vga\_controller tar hand om att kontrollerar 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 horisontal pixel och v\_counter som räknar antalet vertikal pixel. Varje gång när h\_counter räknar upp till HMAX, dvs maximalt pixel vi har till skärm för varje rad, då kommer h\_counter själv 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 synch pulse), VFP(slutpunkt till vertikal front porch), VSP(slutpunkt till vertikal synch pulse) kommer vi använda i vga\_controller. HFP kommer att aktiveras när h\_counter > HFP och skicka ’0’ signal till H-sync och HSP kommer att aktiveras när h\_counter > HSP eller h\_counter < HFP och skicka ’1’ signal till H-sync. VFP och VSP kommer att skicka sync signal till V-sync med på samma sätt.



*Fig. 2. 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.

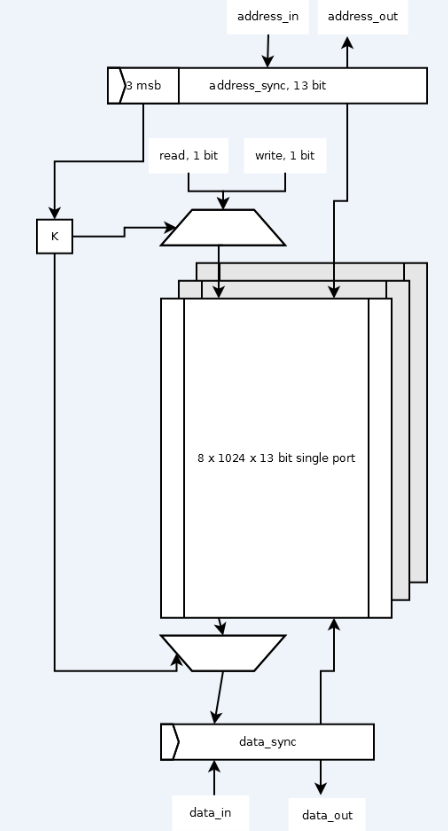
Pixelsender använder samma timing och klock som vga\_controller, skcika ett 13 bitar address till minne, hämta 8 bitar data på detta address, upprepar och då skicka detta data till vga-porten endast när räknare in vga\_controller ligger inom display-ytan.

PixelSender tar hand om address hämtning och färg kod sändning. För att alla data på skärm ska ses bra och tydligt ut, bestämde vi att visa varje data ska ha 5 pixel bredd och 7 pixel höjd. I så fall, kommer vi ha möjligt att visa 128 data per rad och vi behöver 7\*64 = 448 rader för att visa 213 = 8192 adresser. PixelSender skickar data till skärmen varje 5 klockor och upprepar detta för varje 128 data för 7 gånger, i så fall kan vi ha varje data med 5\*7 pixel storlek. På ”border area” visar vi vilken spelare vinner CoreWar.

### 4.3 Minnen

Vi valde att använda en core size(1) 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 8 bitar, operand A 13 bitar, operand B 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).



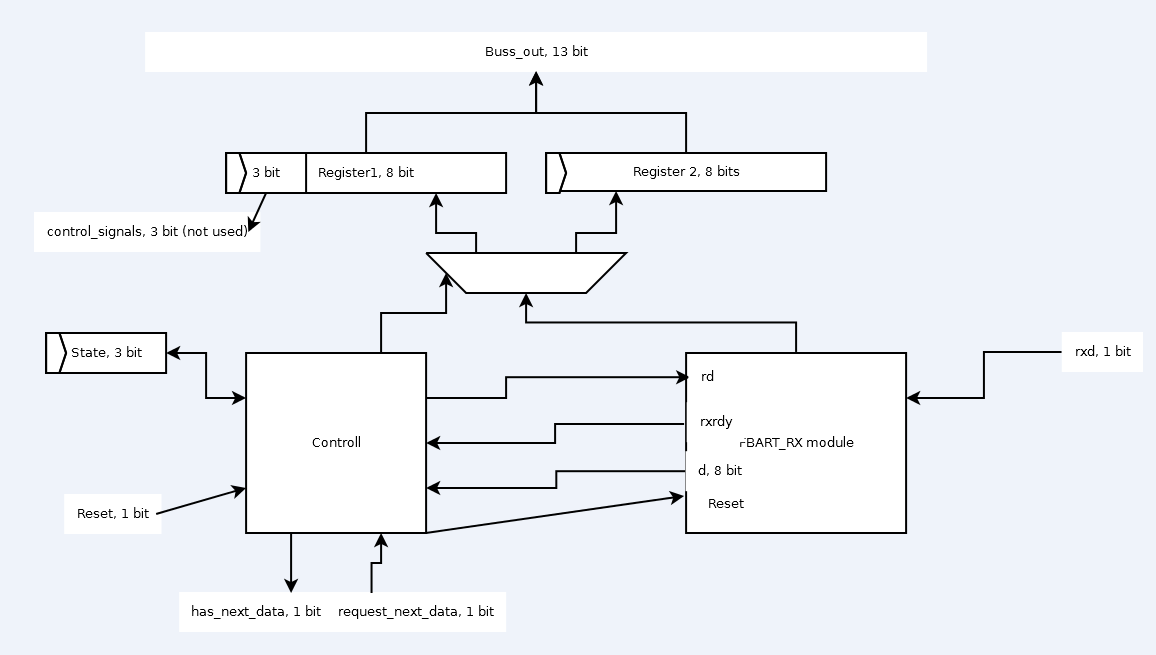
*Fig. 3. Operandminnen*

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 datorn vid minnesaccess.

Då vi har olika färg beroende på vilken instruktion vi har i minnet var det naturligt att slå samman instruktionsminnet och färgminnet då båda var på 8 bitar. Den resulterande maskinen ses i figur 4. 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.

### 4.4 UART

Vår dator använder en 13 bitars buss, det skulle därmed vara trevligt om indatat ifrån vår värddator skulle vara i 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 43434. Modulen väntar på en data begäran, tar emot två 8 bitars överförningar, slår samman dem till 13 bitar (den kastar iväg 3 bitar) och signalerar att data finns.

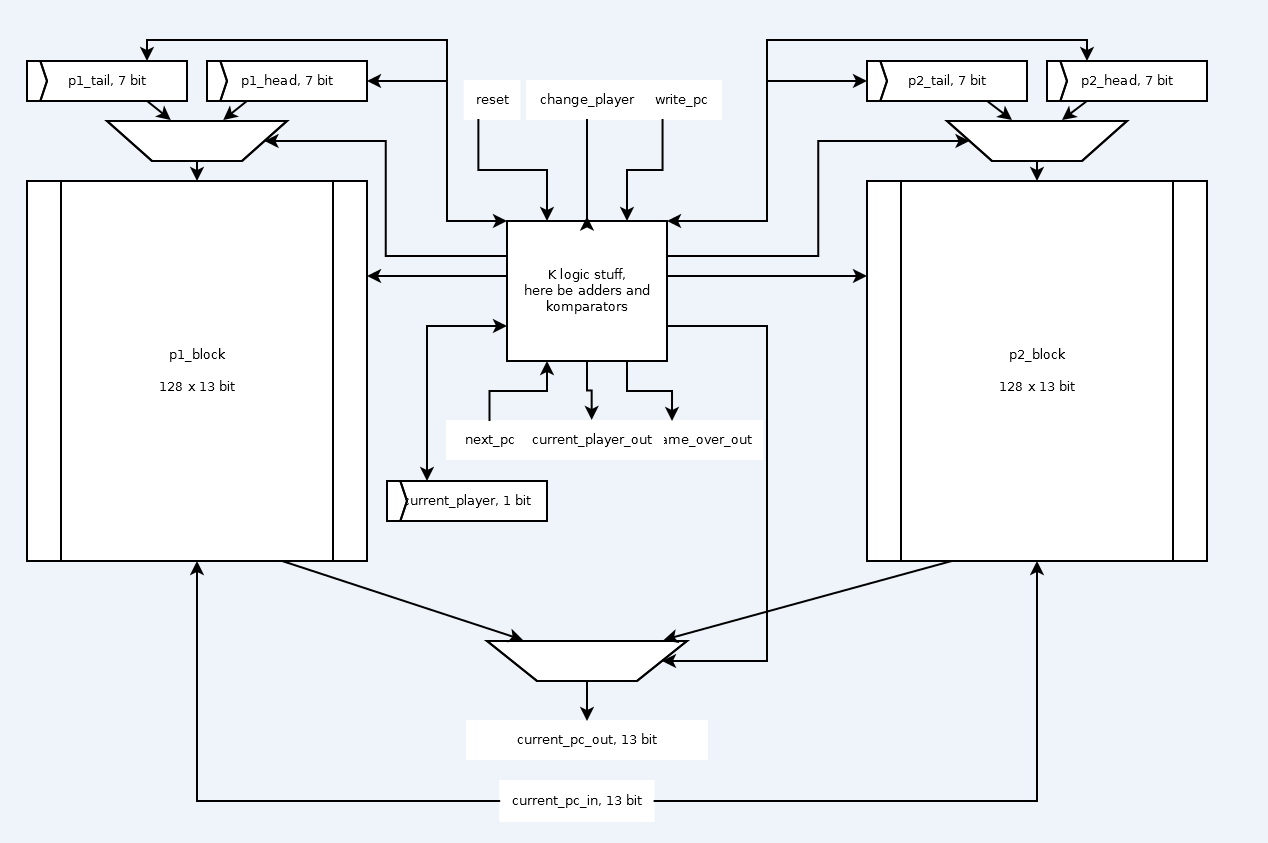
*Fig. 5. UART controll circuit.*

Slår samman två FBART överförningar till en 16 bits överförning

VI fick även ändra i FBARTen då den gick på en 25 Mhz klocka och vårt bygga kör på 100 Mhz. Vi behövde endast öka antalet bitar i en räknade 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 999.

*Fig. 6. 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 annars 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, CoreWars var väldigt roligt.

Tillbakablick på arbetet:

* Kul! CoreWars är awesome
* vhdl är jobbigt
* Kompilatorn är dum
* Ej säker på modellbeskrivning (beskriver vhdl verkligen?)

Förbättringar:

* Snabba upp minnesaccess
* Rensa upp i vhdl
* Möjligt att komprimera och förbättra mikrod

Förslag till fortsatt arbete:

* Fixa allt som är crap i koden
* Implementera en mera modern standard
* Större coresize
* Fler spelare
* Expandera kommunikation till datorn (koth server?!)
* Finns regler om timeout och liknande

## 6 Referenser

### 6.1 Redcode

### 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

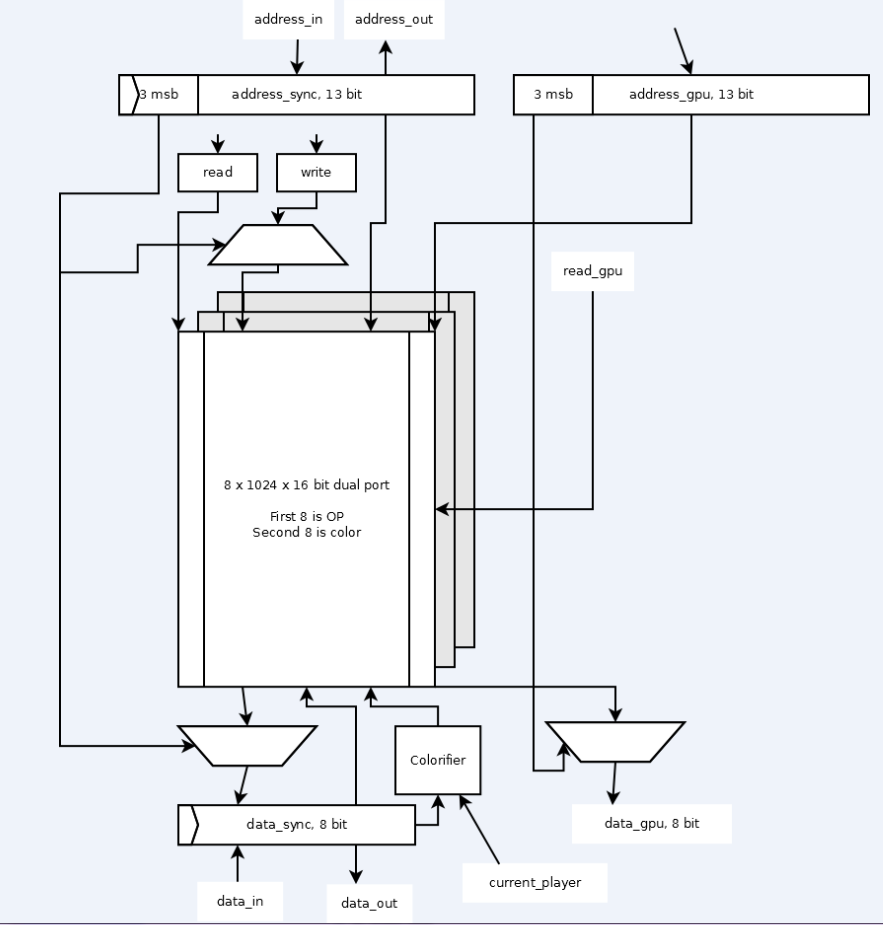
vgaController.vhd

### 6.3 Komponentförteckningar

### 6.4 Programlistor

### 6.5 Exempel warriors

### 6.6 Designskiss



*Fig. 4. Instruktions och färgminne*