Core Wars

TSEA43

Projektrapport

Jonas Hietala
Jesper Tingvall
Jizhi Li

7 Maj, 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

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 där 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].

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], skicka in innehåll till M.A.R.C från en kontrolldator och sätta ut två spelares position. Vi vill även kunna dump ut minnesinnehåll och spelstatus till en VGA skärm. Vår uDator skall kunna utföra alla 10 instruktioner och 4 adresseringsmoder 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.

[1] Se bilaga TODO

[2] http://corewars.nihilists.de/redcode-icws-88.pdf

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 UART.

2.2 Exempel Warriors

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

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.

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.

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]; "Kopiera och sprid" i form av Redcode. Denna Warrior använder replicator strategin.

[3] http://kopimistsamfundet.se

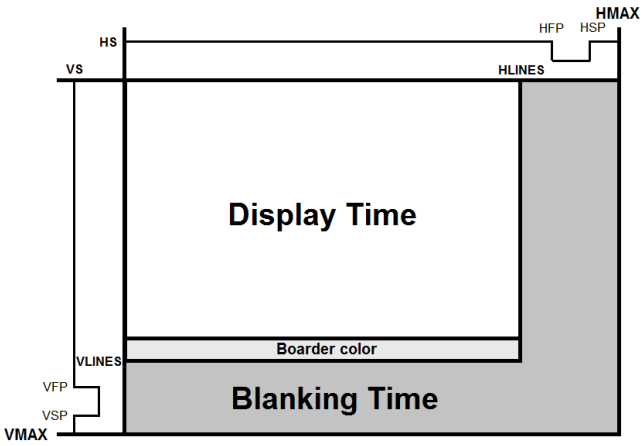
Inseminator
Ännu en Warrior som skapades på skoj men som visade sig vara rätt så effektiv. Den letar upp motståndarens kod och injicerar en massa processer i den i hopp om att motståndaren ej ska förstöra sig egen kod. Detta brukar dock förstöra funktionaliteten i motståndarens kod då den förutsätter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

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.

Dwarf Scout
En dwarf scout är en enkel 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

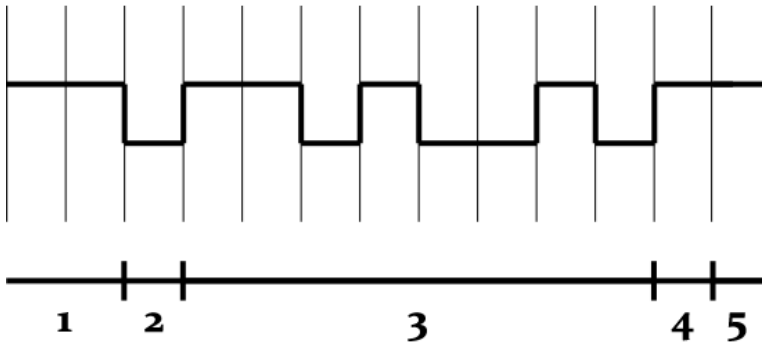
- VSP: 484
- Clk: 25MHz

Vi har blanking time för 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. 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.

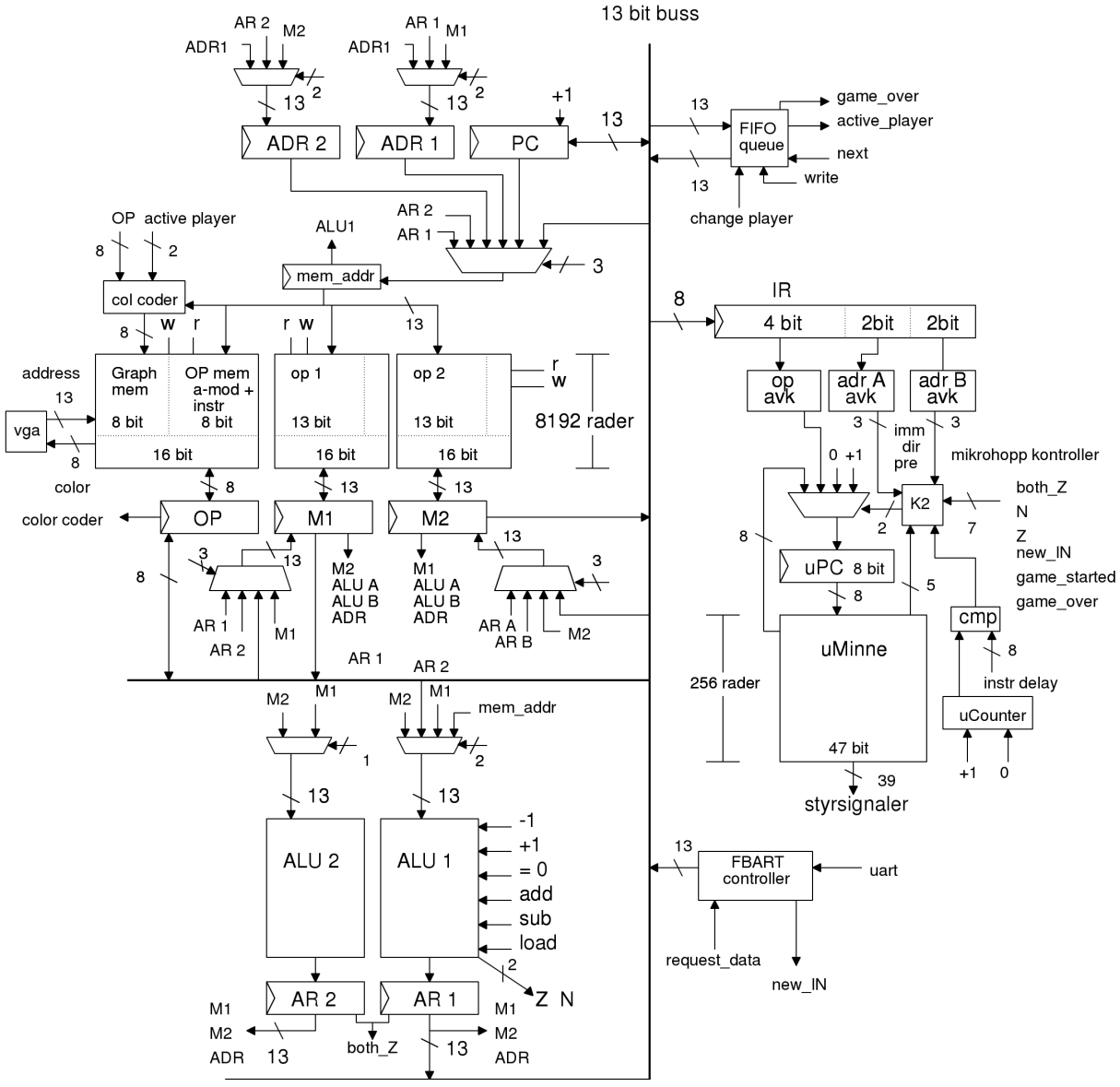


Figur 2: En RS232 överförning. Man har ingen gemensam klocka för sändare och mottagare utan överför endast data, sändaren och mottagaren känner dock till vilken baud rate man överför med. I vårt fall använder vi 115 200 baud.

4 Beskrivning av hårdvara (M.A.R.C)

4.1 Mikrodatorn

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.

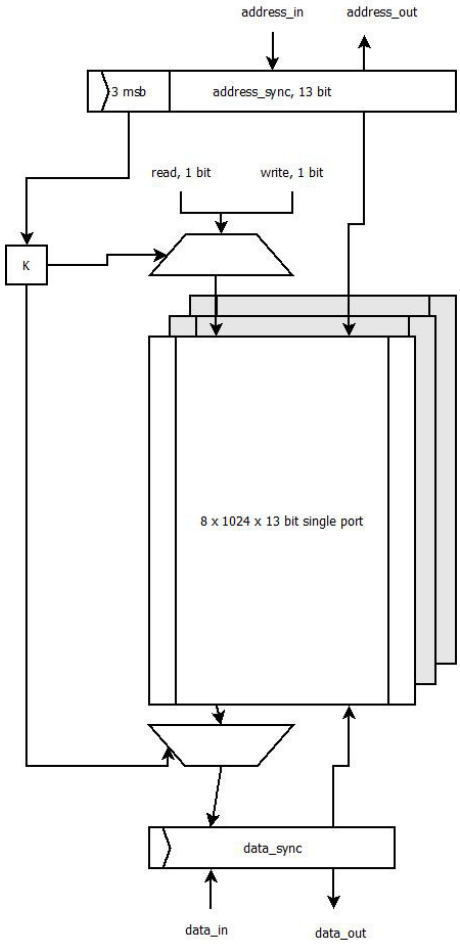
VGA-port kommer endast ta emot färg data när h_counter ¡ HLINES(640 enligt upplösning vi valde) och v_counter ¡ VLINES(480 enligt upplösning vi valde) med hjälp av en enable signal från HLINE och VLINE. Pixelsender använder samma timing och klocka som vga_controller och skickar en 13 bitars adress till 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.

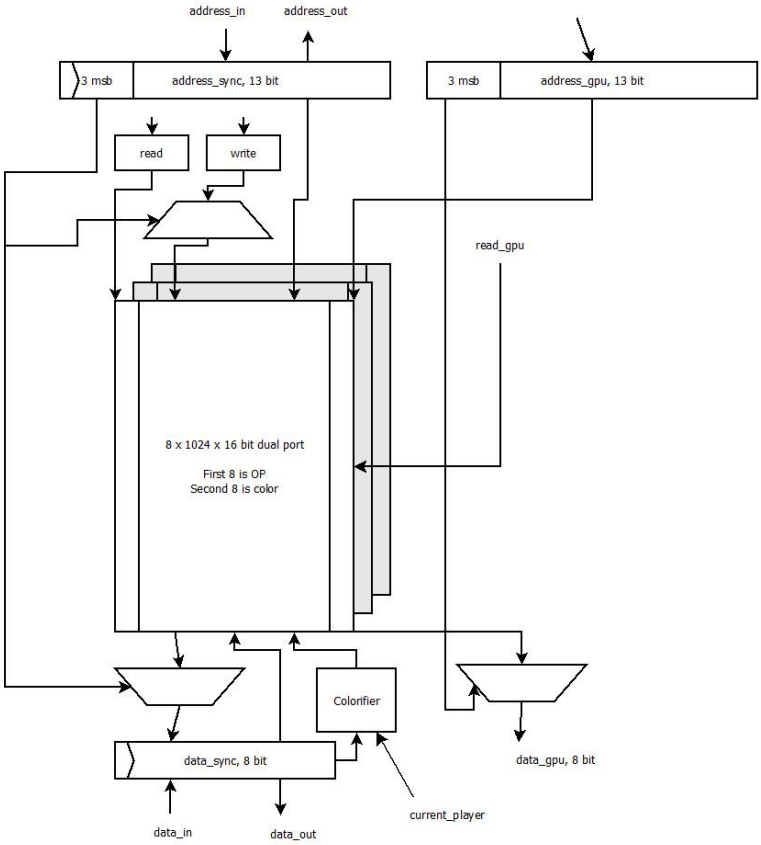
Varje rad Redcode delades upp i 4 delar; instruktion och adresseringsmoder på 8 bitar, operand A på 13 bitar, operand B på 13 bitar och 8 bitar RGB färgning. Det som bäst stämde överens med vår uppdelning var att använda minnesblock utav storleken 1024 x 16 bitar (de 3 sista bitarna används ej dock i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



Figur 4: 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.

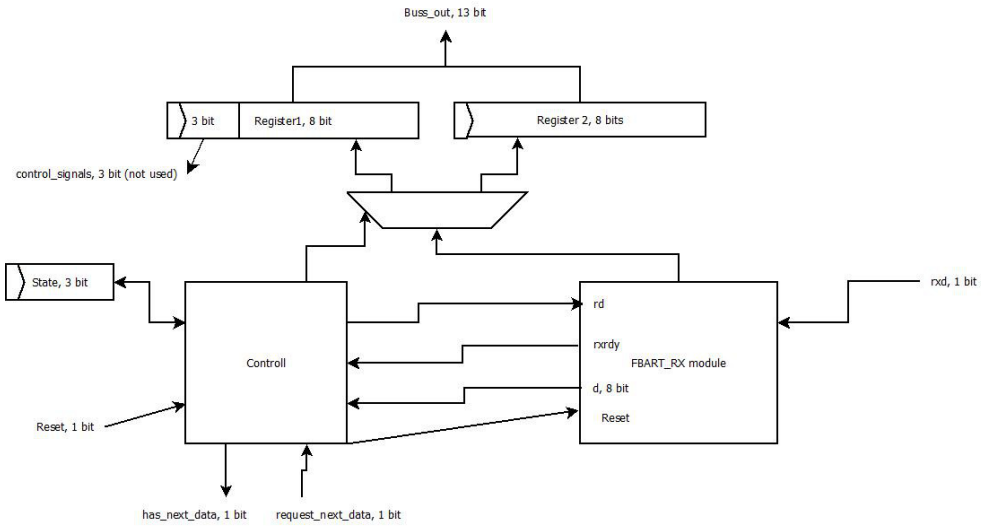
Då vi har olika färg beroende på vilken instruktion vi har i instruktionsminnet var det naturligt att slå samman instruktionsminnet och färgminnet då båda var på 8 bitar. Den resulterande maskinen ses i figur 6. Skillnaden mellan den och operandminnena är att den använder ett dualportminne med den andra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



Figur 5: 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.

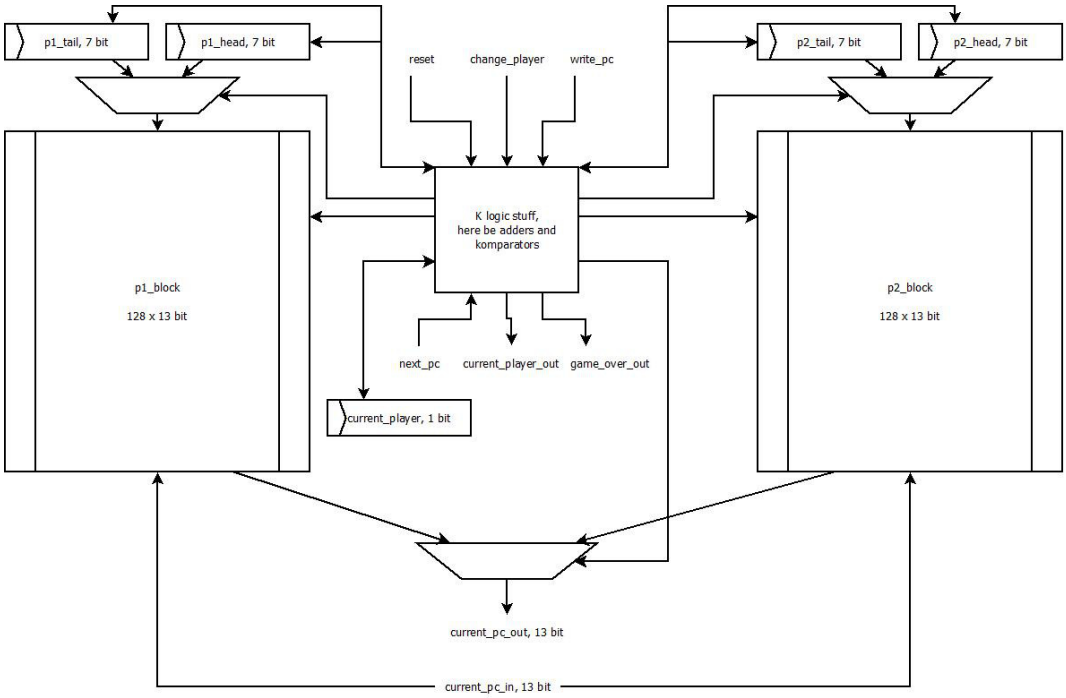


Figur 6: UART kontrollerare. Vi fick även ändra i FBARTen då den gick på en 25 MHz klocka och vårt bygge kör på en 100 MHz klocka. Vi behövde endast öka antalet bitar i en räknare och ändra på en konstant.

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.

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.



Figur 7: Player FIFOs.

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.

A Warriors

A.1 Factory bomber

```
;name Factory bomber
;author Jesper Tingvall
;description Spawns a lot of 'little bombers' all over the memory.
;assert 1
START ADD STEP,  PTR
      MOV LOOPE, <PTR
      MOV LOOP,  <PTR
      SPL @PTR,  0
      JMP START, 0
LOOP  MOV -1,    <-1
LOOPE JMP -1,  0
STEP  DAT -501+8192,   -501+8192
PTR   DAT -151,   -151
```

A.2 Imp spawner

```
;name Imp spawner
;author Jesper Tingvall
;description Spawns a lot of imps all over the memory!

START ADD STEP,  PTR
      MOV IMP,   @PTR
      SPL @PTR,  0
      JMP START, 0
IMP   MOV 0,     1
PTR   DAT 50,    50
STEP  DAT 553,    553
```

B VHDL

B.1 ALU.vhd

```
----------------------------------------------------------------------------------
-- Course:      TSEA43
-- Student:     Jesper Tingvall
-- Design Name: MARC
-- Module Name: ALU
-- Description: Aritmetic logic unit for MARC, consists of two Single Instruction
    Multiple data ALUs.
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity ALU is
    Port (  clk : in std_logic;

            -- ALU Control
            alu_operation : in STD_LOGIC_VECTOR(1 downto 0);
            -- 00 hold
            -- 01 load main buss
            -- 10 +
            -- 11 -

            -- ALU1 answer is zero
            alu1_zeroFlag_out : out STD_LOGIC;

            -- ALU1 answer is negative
            alu1_negFlag : out STD_LOGIC;

            -- Both ALU answers are zero
            alu_zeroFlag : out STD_LOGIC;

            -- Operands to add, subtract or load into the ALU
            alu1_operand : in STD_LOGIC_VECTOR(12 downto 0);
            alu2_operand : in STD_LOGIC_VECTOR(12 downto 0);

            alu1_out : out STD_LOGIC_VECTOR (12 downto 0);
            alu2_out : out STD_LOGIC_VECTOR (12 downto 0)
        );
end ALU;

architecture Behavioral of ALU is

    signal alu1_register : STD_LOGIC_VECTOR (12 downto 0);
    signal alu2_register : STD_LOGIC_VECTOR (12 downto 0);

    signal alu2_zeroFlag : STD_LOGIC;

    -- Temporary signals for calculating negative flag for alu1
```

```
    -- This is a temporary 'fulhax' solution and will result in extra ALU units being
       created.
    signal add_res : STD_LOGIC_VECTOR (12 downto 0);
    signal sub_res : STD_LOGIC_VECTOR (12 downto 0);

   signal alu1_zeroFlag : STD_LOGIC;

begin

  alu1_zeroFlag_out <= alu1_zeroFlag;

    add_res <= alu1_register + alu1_operand;
    sub_res <= alu1_register - alu1_operand;

    alu_zeroFlag <= alu1_zeroFlag and alu2_zeroFlag;

    alu1_out <= alu1_register;
    alu2_out <= alu2_register;

    process(clk)
    begin
        if rising_edge(clk) then

            if alu_operation = "01" then
                alu1_register <= alu1_operand; -- ALU1 = OP1
                alu2_register <= alu2_operand; -- ALU1 = OP1

    if alu1_operand = "0000000000000" then
                    alu1_zeroFlag <= '1';
                else
                    alu1_zeroFlag <= '0';
                end if;

    if alu2_operand = "0000000000000" then
                    alu2_zeroFlag <= '1';
                else
                    alu2_zeroFlag <= '0';
                end if;

            elsif alu_operation = "10" then
                alu1_register <= alu1_register + alu1_operand; -- ALU1 += OP1
                alu2_register <= alu2_register + alu2_operand; -- ALU2 += OP2

                if alu1_register + alu1_operand = "0000000000000" then
                    alu1_zeroFlag <= '1';
                else
                    alu1_zeroFlag <= '0';
                end if;

                if alu2_register + alu2_operand = "0000000000000" then
                    alu2_zeroFlag <= '1';
                else
                    alu2_zeroFlag <= '0';
                end if;

                alu1_negFlag <= add_res(12);
```

```
            elsif alu_operation = "11" then
                alu1_register <= alu1_register - alu1_operand; -- ALU1 -= OP1
                alu2_register <= alu2_register - alu2_operand; -- ALU2 -= OP2

                if alu1_register - alu1_operand = "0000000000000" then
                    alu1_zeroFlag <= '1';
                else
                    alu1_zeroFlag <= '0';
                end if;

                if alu2_register - alu2_operand = "0000000000000" then
                    alu2_zeroFlag <= '1';
                else
                    alu2_zeroFlag <= '0';
                end if;

                alu1_negFlag <= sub_res(12);
            end if;

        end if;
    end process;

end Behavioral;
```

B.2 colorpixSender.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity colorpixSender is
    Port (
      rst : in std_logic;                                  -- reset signal
      clk : in std_logic;                                  -- Universal clock: 100MHz
      indata : in  STD_LOGIC_VECTOR (7 downto 0);          -- data read from
          memory_dual_port
      colorpix : out  STD_LOGIC_VECTOR (7 downto 0);       -- color data to vga-port
      border_color : in std_logic_vector (7 downto 0);     -- color for non-data area
      address : out  STD_LOGIC_VECTOR (12 downto 0));      -- memory address to
          memory_dual_port
end colorpixSender;

architecture Behavioral of colorpixSender is

signal row_cnt : std_logic_vector (3 downto 0) := (others => '0');      -- counter for
     repeating same row 7 times, to have 7 pixels height for each data.
signal column_cnt : std_logic_vector (10 downto 0) := (others => '0');  -- counter for
     row timing
signal unit_cnt : std_logic_vector (2 downto 0) := (others => '0');     -- counter for
     repeating same data 5 times, to have 5 pixels width
signal height : std_logic_vector (11 downto 0) := (others => '0');      -- counter for
     column timing
signal address_mem : STD_LOGIC_VECTOR (12 downto 0) := (others => '0'); --
    temp_address signal
signal pixel_cnt : std_logic_vector(1 downto 0) := "00";                -- counter for
     colorpixSender to use 25MHz clk

begin
  process(clk)
  begin
    if (rising_edge(clk)) then
      if rst = '1' then                                                 -- reset all the
          counters, temp signals and output signals
        colorpix <= (others => '0');
        row_cnt <= (others => '0');
        column_cnt <= (others => '0');
        unit_cnt <= (others => '0');
        pixel_cnt <= "00";
        address_mem <= "0000000000000";
        height <= "000000000000";
      end if;

      if pixel_cnt = "11" then                                          -- execute the
          following code each 4 times 100MHz, which is 25MHz
        pixel_cnt <= "00";

        ---------------
        -- Main start--
        ---------------
```

```
address <= address_mem;                                    -- output the tmp
    address

if height = 525 then                                       -- reset counters
    and address when it reaches the max area
  if column_cnt = 800 then
    height <= (others => '0');
    column_cnt <= (others => '0');
    row_cnt <= (others => '0');
    unit_cnt <= (others => '0');
    address_mem <= (others => '0');
  else
    column_cnt <= column_cnt + 1;
  end if;

elsif height >= 441 and height < 448 then                  -- within the
    display area (notice that we only display for 64*7=448 lines)
  if row_cnt = 6  then
    -------------------
    -- Column6 Start --
    -------------------
    if column_cnt < 801 then
      if column_cnt < 320 then                             -- only display
          half length, because address will reach max.
        if unit_cnt < 5 then                               -- send the same
            data for 5 times
          if unit_cnt = 4 then
            unit_cnt <= (others => '0');
            column_cnt <= column_cnt + 1;
            address_mem <= address_mem + 1;                -- move to next
                address after each 5 cp
          elsif unit_cnt = 1 then
            colorpix(7 downto 0) <= indata(7 downto 0);
            unit_cnt <= unit_cnt + 1;
            column_cnt <= column_cnt + 1;
          else
            column_cnt <= column_cnt + 1;                  -- else time, just
                 increase the column_cnt
            unit_cnt <= unit_cnt + 1;
          end if;
        end if;
      elsif column_cnt = 800 then                          -- reset
          everything on the last cp of column_cnt
        row_cnt <= (others => '0');
        height <= height + 1;
        address_mem <= (others => '0');                    -- this is the end
             of output data. reset the address counter
        column_cnt <= (others => '0');                     -- reset
            column_cnt
        unit_cnt <= (others => '0');                       -- reset the
            unit_cnt(not nessessary, but just in case)
      elsif column_cnt >= 320 and column_cnt < 640 then
        column_cnt <= column_cnt + 1;
        colorpix(7 downto 0) <= "00111111";
      else
        column_cnt <= column_cnt + 1;                      -- this is the
```

```
            blanking area, just incrase the column_cnt here, output nothing
          colorpix(7 downto 0) <= "00000000";
        end if;
      end if;
      -----------------
      -- Column6 End --
      -----------------
    else
      ------------------
      -- Column Start --
      ------------------
      if column_cnt < 801 then
        if column_cnt < 320 then
          if unit_cnt < 5 then
            if unit_cnt = 4 then
              unit_cnt <= (others => '0');
              column_cnt <= column_cnt + 1;
              address_mem <= address_mem + 1;
            elsif unit_cnt = 1 then
              colorpix(7 downto 0) <= indata(7 downto 0);
              unit_cnt <= unit_cnt + 1;
              column_cnt <= column_cnt + 1;
            else
              column_cnt <= column_cnt + 1;
              unit_cnt <= unit_cnt + 1;
            end if;
          end if;
        elsif column_cnt = 800 then
          row_cnt <= row_cnt + 1;
          height <= height + 1;
          address_mem <= address_mem - 64;
          column_cnt <= (others => '0');
          unit_cnt <= (others => '0');
        elsif column_cnt >= 320 and column_cnt < 640 then
          column_cnt <= column_cnt + 1;
          colorpix(7 downto 0) <= "00111111";
        else
          column_cnt <= column_cnt + 1;
          colorpix(7 downto 0) <= (others => '0');
        end if;
      end if;
      ----------------
      -- Column End --
      ----------------
    end if;

  elsif height < 441 then                                    -- this is "normal
      " displaying area
    if row_cnt = 6 then                                      -- 7 pixels height
        for each "gpu_data"
      -------------------
      -- Column6 Start --
      -------------------
      if column_cnt < 801 then                               -- as same as the
          timing in vga_controller (HMAX)
        if column_cnt < 640 then                             -- keep sending
```

```
      128*5 = 640 pixelswithin the display area
    if unit_cnt < 5 then                                    -- send the same
        data for 5 times
      if unit_cnt = 4 then
        unit_cnt <= (others => '0');
        column_cnt <= column_cnt + 1;
        address_mem <= address_mem + 1;                     -- move to next
            address
      elsif unit_cnt = 1 then
        colorpix(7 downto 0) <= indata(7 downto 0);         -- only change the
             input at the first cp
        unit_cnt <= unit_cnt + 1;
        column_cnt <= column_cnt + 1;
      else
        column_cnt <= column_cnt + 1;                       -- else time, just
             increase the column_cnt
        unit_cnt <= unit_cnt + 1;
      end if;
    end if;
  elsif column_cnt = 800 then                               -- reset all the
      counters at HMAX
    row_cnt <= (others => '0');                             -- this is the
        last row of "7 pixels height", reset row_cnt after this row
    height <= height + 1;
    address_mem <= address_mem + 1;                         -- move to next
        address
    column_cnt <= (others => '0');                          -- reset
        column_cnt
    unit_cnt <= (others => '0');                            -- reset the
        unit_cnt(not nessessary, but just in case)

  else
    column_cnt <= column_cnt + 1;                           -- this is the
        blanking are, just incrase the column_cnt here, output nothing
    colorpix(7 downto 0) <= "00000000";
  end if;
end if;
-----------------
-- Column6 End --
-----------------
else
-----------------
-- Column Start --
-----------------
if column_cnt < 801 then
  if column_cnt < 640 then
    if unit_cnt < 5 then
      if unit_cnt = 4 then
        unit_cnt <= (others => '0');
        column_cnt <= column_cnt + 1;
        address_mem <= address_mem + 1;
      elsif unit_cnt = 1 then
      colorpix(7 downto 0) <= indata(7 downto 0);
      unit_cnt <= unit_cnt + 1;
      column_cnt <= column_cnt + 1;
    else
```

```
            column_cnt <= column_cnt + 1;
            unit_cnt <= unit_cnt + 1;
          end if;
        end if;

      elsif column_cnt = 800 then                              -- reset
          everything on the last cp of column_cnt
        row_cnt <= row_cnt + 1;                                -- next row, and
          height increase by 1 too
        height <= height + 1;
        address_mem <= address_mem - 128;                      -- decrease the
          address by 128, back to the beginning
        column_cnt <= (others => '0');                         -- reset
          column_cnt
        unit_cnt <= (others => '0');                           -- reset the
          unit_cnt(not nessessary, but just in case)
      else
        column_cnt <= column_cnt + 1;                          -- this is the
          blanking are, just incrase the column_cnt here, output nothing
        colorpix(7 downto 0) <= ("00000000");
      end if;
      ----------------
      -- Column End --
      ----------------
    end if;
  end if;

else
  if column_cnt = 800 then                                     -- during the
      blanking zone, just increase the counter.
    height <= height + 1;
    column_cnt <= (others => '0');

  elsif column_cnt < 640 then                                  -- within the
      border area
    if unit_cnt < 5 then
      if unit_cnt = 4 then
        unit_cnt <= (others => '0');
        column_cnt <= column_cnt + 1;
      elsif unit_cnt = 1 then
        colorpix(7 downto 0) <= border_color;                  -- output the
          border color only
        unit_cnt <= unit_cnt + 1;
        column_cnt <= column_cnt + 1;
      else
        column_cnt <= column_cnt + 1;
        unit_cnt <= unit_cnt + 1;
      end if;
    end if;

  else
    column_cnt <= column_cnt + 1;
    colorpix(7 downto 0) <= "00000000";
  end if;
end if;
---------------
```

```
      -- Main ends--
      ---------------
    else
      pixel_cnt <= pixel_cnt + 1;
    end if;
  end if;
end process;
end Behavioral;
```

C Script

C.1 Assembler

```
#!/usr/bin/perl -w

use utf8;

# Modern Perl
use strict;
use warnings;

sub say { print "$_\n" for @_; }

use Getopt::Long;

# Command line options
my $help;
my $verbose;
my $debug;
my $obj_out = "";
my $raw;
my $pc;

GetOptions(
    'help|h' => \$help,
    'obj|o=s' => \$obj_out,
    'verbose|v' => \$verbose,
    'debug' => \$debug,
    'raw|r' => \$raw,
    'pc|p=s' => \$pc,
);

if ($obj_out && scalar @ARGV != 2) {
    say "Object output requires two files\!";
}

if ($help || !scalar @ARGV || (!$debug && !$verbose && !$raw && !$obj_out)) {
    my ($name) = $0 =~ /([^\/]+$)/;

    say "Simple assembler";
    say "  usage:";
    say "   $name [option]... [file]...";
    say "  options:";
    say "   -h --help       Show this screen.";
    say "   -o --obj=FILE   Generate binary file output.";
    say "       Must specify two files.";
    say "   -v --verbose    Verbose output signals in a human readable format.";
    say "   -r --raw        Outputs raw binary data.";
    say "   -p --pc         Specify PC for 2nd program, in hex.";
    exit;
}

our %labels;
our %constants;
```

```
# Evaluate an operator statement, like val*2+1
sub evaluate {
    my ($def, $linenum) = @_;

    my @pieces = split /[-+*\/]/, $def;

    say "Evaluating: $def" if $debug;

    for my $piece (@pieces) {

        # Remove whitespace
        $piece = trim ($piece);

        # Remove parenthesis
        $piece =~ /^\(*(.*?)\)*$/;
        $piece = $1;

        # Ignore empty and all numbers
        next if $piece =~ /^\d*$/;

        # Read a label
        if ($piece =~ /^([A-Z0-9]{0,8})[A-Z0-9]*$/i) {
            my $label = $1;

            if (exists $labels{$label}) {

                # Addresses to labels are relative to the line of code
                my $relative = $labels{$label} - $linenum;

                # Surround negative values with parenthesis
                $relative = "($relative)" if $relative < 0;
                $def =~ s/$piece/$relative/;
            }
            elsif (exists $constants{$label}) {

                # Substitute the label with it's value
                $def =~ s/$piece/$constants{$label}/;
            }
            else {
                die "Compile error, no label '$label' in scope.";
            }
        }
        else {
            die "Syntax error, label expected ner '$piece'";
        }
    }

    say "eval '$def'" if $debug;

    my $value = eval $def;
    die "Error: Malformed operator '$def' $@" if $@;

    return $value;
}
```

```
my $out;
if ($obj_out) {
    open $out, '>', $obj_out or die "Couldn't open file $obj_out $!";
    binmode $out;

    my ($p1, $p2) = @ARGV;

    parse ($p1);

    if ($pc) {
        $pc = hex $pc;
    }
    else {
        # We have 8192 lines, PC1 will be at 0 so generate a random number
        # between 100 and 8091
        $pc = int(rand(7991)) + 100;
    }

    # Convert to bin and pad up to 16 bits
    my $pc_bin = "000" . dec2bin ($pc, 13);

    print $out pack ("B16", $pc_bin);

    if ($raw) {
        say $pc_bin;
    }
    elsif ($verbose) {
        my $hex = bin2hex ($pc_bin);

        say "\n$pc_bin  " . join (" ", $hex =~ /../g) . "  ; Player 2 PC\n";
    }

    parse ($p2);
}
else {
    parse ($_) for @ARGV;
}

sub parse {
    my ($src) = @_;

    open my $in, '<', $src or die "Couldn't open file $src $!";

    # Store line of code here when processing
    my @code;

    my $codeline = 0;

    while (my $line = <$in>) {
        chomp $line;

        # Ignore empty lines
        next if $line =~ /^\s*$/;

        # Remove comments, will always match
        my ($code, $comment) = $line =~ /^([^;]*);?(.*)/;
```

```
    # Don't parse a full comment line
    next if !$code;

    # Match a constant
    if ($code =~ /^
                    ([A-Z0-9]{0,8})        # Label necessary
                    [A-Z0-9]*              # Only catch first 8 chars
                    \s+
                    equ                    # Constant instr mnemonic
                    \s+
                    ([-+*\/()A-Z0-9\s+]+)    # Definition
                    /xi)
    {
        push (@code, $code);
    }
    # Match up a line of redcode
    elsif ($code =~ /^(?:
                    ([A-Z0-9]{0,8})?      # Label, not necessary
                    [A-Z0-9]*             # Only catch first 8 chars
                    \s+
                )?
                ([A-Z0-9]+)               # Mnemonic
                (?:\.[A-Z0-9]+)?          # Throw away postfix mod if there is any
                \s+
                ([^,]+)                   # A operand
                (?:                       # B op not mandatory
                    \s*,\s*               # , delimited
                    ([^,]+)               # B operand
                )?
                /xi)
    {
        my ($label, $instr, $a_op, $b_op) = ($1, $2, $3, $4);

        $b_op = "0" if !$b_op;

        # Log label
        if ($label) {
            $labels{$label} = $codeline;
            say "L: $label = $codeline" if $debug;
        }
        $codeline++;

        push (@code, $code);
    }
    # Match end
    elsif ($code =~ /^\s*end/i) {
        last;
    }
    else {
        die "Syntax error.";
    }
}

my $bin_output = "";
```

```
# Print line numbers
my $codeline_bin = dec2bin ($codeline, 16);

if ($raw) {
    say $codeline_bin;
}
elsif ($verbose) {
    my $hex = bin2hex ($codeline_bin);
    say "$codeline_bin  " . join (" ", $hex =~ /../g) . "  ; Number of rows (
        $codeline) $src\n";
    say "   pad    OP  A  B  pad     A op      pad      B op";
}

# Start from -1 as we incr in the beginning
$codeline = -1;

for my $code (@code) {

    # Match a constant
    if ($code =~ /^
                    ([A-Z0-9]{0,8})         # Label necessary
                    [A-Z0-9]*               # Only catch first 8 chars
                    \s+
                    equ                     # Constant instr mnemonic
                    \s+
                    ([-+*\/()A-Z0-9\s+]+)    # Definition
                    /xi)
    {
        my ($label, $def) = ($1, $2);

        # Evaluate
        my $value = evaluate $def, $codeline;

        # And insert
        $constants{$label} = $value;

        say "C: $label = $def ($value)" if $debug;
    }
    # Match up a line of redcode
    elsif ($code =~ /^(?:
                    ([A-Z0-9]{0,8})?      # Label, not necessary
                    [A-Z0-9]*             # Only catch first 8 chars
                    \s+
                )?
                ([A-Z0-9]+)              # Mnemonic
                (?:\.[A-Z0-9]+)?         # Throw away postfix mod if there is any
                \s+
                ([^,]+)                  # A operand
                (?:                      # B op not mandatory
                    \s*,\s*              # , delimited
                    ([^,]+)              # B operand
                )?
                /xi)
    {
        my ($label, $instr, $a_op, $b_op) = ($1, $2, $3, $4);
```

```
$b_op = "0" if !$b_op;

$codeline++;

my %types = (
    '#' => "01",    # Immediate
    '@' => "10",    # Indirect
    '<' => "11",    # Pre-decrement indirect
);

# Default
my $a_mod = "00";   # Direct
my $b_mod = "00";

my $a_type = "";
my $b_type = "";

# Fetch adress modes
if ($a_op =~ /^([#@<])(.*)/) {
    my ($op, $rest) = ($1, $2);

    $a_mod = $types{$op};
    $a_type = $op;
    $a_op = $rest;
}
if ($b_op =~ /^([#@<])(.*)/) {
    my ($op, $rest) = ($1, $2);

    $b_mod = $types{$op};
    $b_type = $op;
    $b_op = $rest;
}

my $a_val = evaluate $a_op, $codeline;
my $b_val = evaluate $b_op, $codeline;

my $a_bin = dec2bin ($a_val, 13);
my $b_bin = dec2bin ($b_val, 13);

my %instr_codes = (
    DAT => '0000',
    MOV => '0001',
    ADD => '0010',
    SUB => '0011',
    JMP => '0100',
    JMZ=> '0101',
    JMN => '0110',
    CMP => '0111',
    SLT => '1000',
    DJN => '1001',
    SPL => '1010',
);

if (!exists $instr_codes{uc($instr)}) {
    say "Code: $code";
    die "Instr '$instr' does not exist!";
```

```
            }

            my $instr_code = $instr_codes{uc($instr)};

            my $op_bin = "00000000$instr_code$a_mod$b_mod";
            my $a_op_bin = "000$a_bin";
            my $b_op_bin = "000$b_bin";

            # Output
            if ($debug) {
                say "I: $instr $a_mod $b_mod $a_val $b_val";
            }

            if ($raw) {
                say "$op_bin$a_op_bin$b_op_bin";
            }
            elsif ($verbose) {
                my $line = "$op_bin$a_op_bin$b_op_bin";
                my $hex = bin2hex ($line);

                # Pretty output, split binary into sections
                # split up hex values into pairs and add code as comment
                say "00000000 $instr_code $a_mod $b_mod 000 $a_bin 000 $b_bin   " .
                    join (" ", ($hex =~ /../g)) .
                    "  ; $instr $a_type $a_op($a_val) $b_type $b_op($b_val)";
            }

            $bin_output .= "$op_bin$a_op_bin$b_op_bin";
        }
        # Match end
        elsif ($code =~ /^\s*end/) {
            last;
        }
        else {
            say "Couldn't match: $code";
            die "Syntax error.";
        }
    }

    if ($obj_out) {
        # Output object values
        print $out pack ("B16", $codeline_bin);

        my $val = pack ("B" . 48 * ($codeline + 1), $bin_output);
        print $out $val;
    }
}

if ($obj_out) {
    close $out;
    my $size = (stat $obj_out)[7];
    say "Wrote $size bytes to $obj_out";
}

sub trim {
    my $string = shift;
```

```
    $string =~ s/^\s+//;
    $string =~ s/\s+$//;
    return $string;
}
# With a specified length
sub dec2bin {
    my ($dec, $l) = @_;
    my $bin = unpack("B32", pack("N", $dec));

    # Force to length
    if (length($bin) < $l) {
        $bin = '0' x ($l - length($bin)) . $bin;
    }
    elsif (length($bin) > $l) {
        # Truncate from the back so 00 1111 1111 -> 1111 1111
        $bin = substr $bin, -$l;
    }

    return $bin;
}
sub dec2hex {
    my $d = shift;
    my $h = sprintf ("%x", $d);
    return $h;
}
sub hex2bin {
    my $h = shift;
    my $hlen = length($h);
    my $blen = $hlen * 4;
    return unpack("B$blen", pack("H$hlen", $h));
}
sub bin2hex {
    my $b = shift;
    return unpack("H*", pack("B*", $b));
}
```

C.2 Mikrokod

```
; Startup, check if we're in game
        jmpS $GAME                              ; Execute game code only if we're
            running
        jmpO +0                                 ; Infinite loop if we've recieved game
             over, reset to break it

; Clear memory contents
        ALU = 0                                 ; Load 0
        ALU1 -> buss, buss -> OP, buss -> M1, buss -> M2, buss -> PC
:CLRMEM PC -> mem_addr                          ; Look at PC
        OP -> mem, M1 -> mem, M2 -> mem         ; Clear it
        ALU++                                   ; Incr
        ALU1 -> PC, jmpZ $LOADP                 ; If 0 we're done looping
        jmp $CLRMEM                             ; Else continue

:LOADP  shall_load, jmp $POLL                   ; If we should start polling (start is
     pressed)
        jmp -1

; Load in program to memory
:POLL   IN -> buss                              ; Temporary start polling fbart

; Load program 1
        ALU = 0, fifo_next                      ; PC1 = 0
        ALU1 -> buss, buss -> FIFO, buss -> PC  ; Insert it to FIFO

        jmpIN $F1NUM                            ; Fetch number of rows from in
        jmp -1                                  ; If not ready, stall
:F1NUM  IN -> ALU1                              ; Store number of rows in ALU1
:F1ROW  jmpIN $F1OP                             ; Fetch OP
        jmp -1
:F1OP   IN -> OP                                ; Store OP
        jmpIN $F1M1                             ; Fetch M1
        jmp -1
:F1M1   IN -> M1                                ; Store M1
        jmpIN $F1M2                             ; Fetch M2
        jmp -1
:F1M2   IN -> M2, PC -> mem_addr                ; Store M2
        OP -> mem, M1 -> mem, M2 -> mem         ; Write line to mem
        ALU--, PC++                             ; Decr row counter, incr PC
        jmpZ $LOAD2                             ; If 0 we're done
        jmp $F1ROW                              ; Else load next row

; Load program 2
:LOAD2  change_player                           ; Change player in FIFO
        jmpIN $F2PC                             ; Fetch PC for player 2
        jmp -1
:F2PC   IN -> buss, buss -> FIFO, buss -> PC    ; Insert to FIFO
        jmpIN $F2NUM                            ; Fetch number of rows
        jmp -1
:F2NUM  IN -> ALU1                              ; Store numbers of rows in ALU1
:F2ROW  jmpIN $F2OP                             ; Fetch OP
        jmp -1
:F2OP   IN -> OP                                ; Store OP
```

```
        jmpIN $F2M1                             ; Fetch M1
        jmp -1
:F2M1   IN -> M1                                ; Store M1
        jmpIN $F2M2                             ; Fetch M2
        jmp -1
:F2M2   IN -> M2, PC -> mem_addr                ; Store M2
        OP -> mem, M1 -> mem, M2 -> mem         ; Write line to mem
        ALU--, PC++                             ; Decr row counter, incr PC
        jmpZ $LEND                              ; If 0 we're done
        jmp $F2ROW                              ; Else load next row
:LEND   game_started, jmpZ 0


; Game sequence
:GAME   change_player                       ; Change players turn
        fifo_next
        FIFO -> PC                          ; Fetch next PC
        PC -> mem_addr
        mem -> OP, mem -> M1, mem -> M2     ; Fetch data
        OP -> IR, jmp $AMOD                 ; Go to adress decoding

; Calculate adress mode for A operand
:AMOD   jmpAimm $BMOD                       ; If immediate we're done
        M1 -> ALU1                          ; Address is a relative offset
        ALU1 += PC                          ; so add PC
        ALU1 -> M1
        jmpAdir $BMOD                       ; If direct, we're done
        M1 -> mem_addr
  mem -> M2           ; Check B address
        M2 -> M1, jmpApre $APRE             ; Move it to A's place. If pre-decr decr
            and come back
:AOFF   M1 -> ALU1                          ; Relative offset, add mem_addr
        ALU1 += mem_addr
        ALU1 -> M1
        jmp $BMOD                           ; Do the same for the B operand

:APRE   M1 -> ALU1                          ; Decr
        ALU--
        ALU1 -> M1, ALU1 -> M2
        M2 -> mem                           ; Write it back where it came from
        jmp $AOFF                           ; Continue

; Calculate adress mode for B operand
:BMOD   PC -> mem_addr                      ; Retrieve data
        mem -> M2
        jmpBimm $INSTR                      ; If immediate we're done
        M2 -> ALU1                          ; Relative address, add PC
        ALU1 += PC
        ALU1 -> M2
        jmpBdir $INSTR                      ; If direct, we're done
        M2 -> mem_addr
        mem -> M2                           ; Check B operand of the address
        jmpBpre $BPRE
:BOFF   M2 -> ALU1                          ; Relative offset, add mem_addr
        ALU1 += mem_addr
        ALU1 -> M2
```

```
        jmp $INSTR                          ; We're done

:BPRE   M2 -> ALU1                          ; Decr
        ALU--
        ALU1 -> M2
        M2 -> mem                           ; Write it back where it came from
        jmp $BOFF                           ; Continue


; Load up instruction and proceed to instruction decoding
; A operand is now in ADR1 and B in ADR2
; If immediate ignore these, they're also in M1 and M2

:INSTR  M1 -> ADR1, M2 -> ADR2, op_addr -> uPC


; Execute instruction
;
; ADR1 is now the absolute address for the A operand
; ADR2 is for the B operand
; M1 and M2 holds copies of ADR1 and ADR2 always
;
; If immediate, the data is instead in M1 or M2

; DAT  Executing data will eat up the PC
:DAT    jmp $END

; MOV  Move A to B
:MOV    jmpAimm $IMOV                       ; Handle A immediate special case
        ADR1 -> mem_addr                    ; Peek at memory from A absolute addr
        mem -> OP, mem -> M1, mem -> M2
        ADR2 -> mem_addr                    ; Copy it to B absolute addr
        OP -> mem, M1 -> mem, M2 -> mem
        jmp $ADDPC                          ; Keep using our PC

; If A immediate, move A to B op specified by B mem address
:IMOV   ADR2 -> mem_addr, M1 -> M2          ; Examine B's absolute address
        M2 -> mem                           ; Move A op there
        jmp $ADDPC                          ; We want to keep our PC

; ADD  Add A to B
:ADD    jmpAimm $IADD                       ; A immediate special case
        ADR1 -> mem_addr                    ; Examine A address
        mem -> M1, mem -> M2
        ADR2 -> mem_addr, M1 -> ALU1, M2 -> ALU2
        mem -> M1, mem -> M2                ; Examine B address
        ALU1 += M1, ALU2 += M2              ; Add them
        ALU1 -> M1, ALU2 -> M2              ; And write back
        M1 -> mem, M2 -> mem
        jmp $ADDPC                          ; Continue

:IADD   ADR2 -> mem_addr                    ; Alter in B's absolute address
        M1 -> ALU1, mem -> M2               ; Add A to B op
        ALU1 += M2
        ALU1 -> M2
        M2 -> mem                           ; Write it back
```

```
        jmp $ADDPC                          ; Continue

; SUB  Sub A from B
:SUB    jmpAimm $ISUB
        ADR2 -> mem_addr                    ; Examine B address
        mem -> M1, mem -> M2
        ADR1 -> mem_addr, M1 -> ALU1, M2 -> ALU2
        mem -> M1, mem -> M2                ; Examine A address
        ALU1 -= M1, ALU2 -= M2              ; Sub them
        ALU1 -> M1, ALU2 -> M2, ADR2 -> mem_addr
        M1 -> mem, M2 -> mem                ; And write back
        jmp $ADDPC                          ; Continue

:ISUB   ADR2 -> mem_addr                    ; Alter in B's absolute address
        mem -> M2                           ; Load B op
        M2 -> ALU1                          ; Sub A op
        ALU1 -= M1
        ALU1 -> M2
        M2 -> mem                           ; Write it back
        jmp $ADDPC                          ; Continue

; JMP  Jump to A
:JMP    M1 -> FIFO, jmp $END                ; Jump to adress of A op

; JMPZ Jump to A if B zero
:JMPZ   jmpBimm $IJMPZ
        ADR2 -> mem_addr                    ; Fetch B op
        mem -> M2
:IJMPZ  M2 -> ALU1
        jmpZ $DOJMPZ                        ; If not zero
        jmp $ADDPC                          ; Continue
:DOJMPZ jmp $JMP                            ; Else do a jump

; JMPN Jump to A if B non-zero
:JMPN   jmpBimm $IJMPN
        ADR2 -> mem_addr                    ; Fetch B op
        mem -> M2
:IJMPN  M2 -> ALU1
        jmpZ $ADDPC                         ; If zero no jump
        jmp $JMP                            ; Else do a jump

; CMP If A eq B skip next instr
:CMP    jmpAimm $ICMP
        ADR2 -> mem_addr                    ; Fetch mem operands +OP spec by B op
        mem -> OP, mem -> M1, mem -> M2
        ADR1 -> mem_addr, M1 -> ALU1, M2 -> ALU2
        mem -> M1, mem -> M2                ; Fetch mem operands spec by A op
        ALU1 -= M1, ALU2 -= M2              ; Compare
        jmpE $CMPOP                         ; If eq compare OP as well
        jmp $ADDPC                          ; Else continue with next instr

:CMPOP  OP -> ALU1
        mem -> OP
        ALU1 -= OP
        jmpZ $SKIP                          ; If eq, skip next instr
        jmp $ADDPC                          ; Else continue as normal
```

```
:ICMP   ADR2 -> mem_addr                    ; Fetch B op
        mem -> M2, M1 -> ALU1
        ALU1 -= M2                          ; Compare
        jmpZ $SKIP                          ; If eq, skip next instr
        jmp $ADDPC                          ; Else continue as normal

; SLT if A is less than B skip next instr
:SLT    jmpAimm $ISLT
        ADR1 -> mem_addr                    ; Fetch B field spec by A
        mem -> M2
        M2 -> ALU1
:SLTCMP ADR2 -> mem_addr                    ; Fetch B field spec by B
        mem -> M2
        ALU1 -= M2                          ; Compare A < B
        jmpN $SKIP                          ; Skip next instr if A < B
        jmp $ADDPC                          ; Else continue

:ISLT   M1 -> ALU1, jmp $SLTCMP             ; Place A in ALU for comparison, rest is
    the same

; DJN Decr B, if not zero jmp to A
:DJN    jmpBimm $IDJN
        ADR2 -> mem_addr                    ; Fetch B field spec by B
        mem -> M2
:DODJN  M2 -> ALU1                          ; Decr
        ALU--
        ALU1 -> M2
        M2 -> mem                           ; Write back
        jmpZ $ADDPC                         ; If zero, continue
        M1 -> FIFO, jmp $END                ; Else jump to A

:IDJN   M2 -> ALU1, PC -> mem_addr, jmp $DODJN          ; B is immediate data, set PC
    as mem_addr

; SPL Place A in process queue
:SPL    PC++
        PC -> FIFO                          ; First add PC++ to queue
        M1 -> FIFO, jmp $END                ; Then add the address of A to queue


:SKIP   PC++, jmp $ADDPC

; Keep the PC for next round
:ADDPC  PC++
        PC -> FIFO

:END    check_gameover
:DELAY  jmpC 0                              ; Start over when we've spent enough time
        jmp $DELAY


;   Direct (default)
; The value is an offset to the memory location.
;
; # Immediate
```

```
; The value is the data
;
; @ Indirect
; Offset to a memory location. B operand of that is an offset to another memory
    location.
;
; < Pre-decrement indirect
; Offset to a memory location. B operand there, B--, inserted again. That is then used
     as an offset to another memory location.
```

C.3 Mikrokodningshjälp

```
#!/usr/bin/perl -w

use utf8;

# Modern::Perl
use strict;
use warnings;

sub say { print "$_\n" for @_; }

use Getopt::Long;

# Command line options
my $help;
my $dest;
my $lines_until_header = 20;
my $verbose;
my $debug;
my $vhdl;

GetOptions(
    'help|h' => \$help,
    'destination|d=s' => \$dest,
    'header|l=i' => \$lines_until_header,
    'verbose|v' => \$verbose,
    'debug' => \$debug,
    'vhdl' => \$vhdl,
);

$dest = "code_output" if !$dest;
my $src = $ARGV[0];

if ($help || !$src) {
    say "Convert control code comments to actual control code.";
    say "   options:";
    say "   -h --help       Show this screen.";
    say "   -v --verbose    Verbose output signals in a human readable format.";
    say "   -vhdl           Makes my life easier.";
    exit;
}

# Ouput format
my $h = "game FIFO IR ADR1 ADR2 OP M1 M2 mem1 mem2 mem3 mem_addr ALU1 ALU2 ALU buss PC
      uPC  uPC_addr";
my $c = " 00   00  0   00  00   0  00 00  00   00   00    000     00   0   000 000  00
     00000 00000000";

my %ALU = (
    load => "001",
    '+' => "010",
    '-' =>"011",
    '++' =>"100",
    '--' => "101",
    '0' => "110",
```

```
);

my %buss = (
    PC => "000",
    OP => "001",
    M1 => "010",
    M2 => "011",
    ALU1 => "100",
    FIFO => "101",
    IN => "110",
);

# For mem1 mem2 mem3
my %mem = (
    read => "01",
    write => "10",
);

my %mem_map = (
    OP => "mem1",
    M1 => "mem2",
    M2 => "mem3",
);

# Single value shorthands will be a shorthand for
# position: $position{<val>} value: $registers{<val>}->{<key>}
my %singles = (
    jmp => "uPC",
    jmpZ => "uPC",
    jmpIN => "uPC",
    jmpC => "uPC",
    jmpS => "uPC",
    jmpAimm => "uPC",
    jmpAdir => "uPC",
    jmpApre => "uPC",
    jmpBimm => "uPC",
    jmpBdir => "uPC",
    jmpBpre => "uPC",
    change_player => "FIFO",
    fifo_next => "FIFO",
    game_started => "game",
    check_gameover => "game",
    shall_load => "game",
);

my %registers = (
    uPC => {
        '++' => "00000",
        op_addr => "00001",

        jmp =>  "00010",
        jmpZ => "00011",
        jmpIN =>"00100",
        jmpC => "00101",
        jmpS => "00110",
        jmpN => "00111",
```

```
    jmpE => "01000",
    jmpL => "01001", # Deprecated!
    jmpO => "01010",

    jmpAimm => "10000",
    jmpAdir => "10001",
    jmpApre => "10010",
    jmpBimm => "10011",
    jmpBdir => "10100",
    jmpBpre => "10101",

    '0' => "11111",
},
PC => {
    buss => "01",
    '++' => "10",
    '0' => "11",
},
IR => {
    buss => "1",
},
ADR1 => {
    buss => "01",
    M1 => "10",
    ALU1 => "11",
},
ADR2 => {
    buss => "01",
    M2 => "10",
    ALU2 => "11",
},
ALU1 => {
    M1 => "00",
    buss => "01",
    M2 => "10",
    mem_addr => "11",
},
ALU2 => {
    M1 => "0",
    M2 => "1",
},
OP => {
    buss => "1",
},
M1 => {
    buss => "01",
    ALU1 => "10",
    ALU2 => "11",
},
M2 => {
    buss => "01",
    ALU1 => "10",
    ALU2 => "11",
},
mem_addr => {
    buss => "001",
```

```
        ALU1 => "010",
        ALU2 => "011",
        ADR1 => "100",
        ADR2 => "101",
        PC => "110",
    },
    FIFO => {
        buss => "01",
        change_player => "10",
        fifo_next => "11",
    },
    game => {
        game_started => "01",
        check_gameover => "10",
        shall_load => "11",
    },
);

# Positions for our subsignals in the grand control scheme
my %positions;

# Calculate positions from output format
my @namechunks = split (/\s+/, trim($h));
my @codechunks = split (/\s+/, trim($c));

my $signallength = length (join "", @codechunks);
my $l = $signallength - 1;

my @vhdl_output;

for my $chunk (@codechunks) {
    my $r = $l - length ($chunk) + 1;

    my $name = shift @namechunks;

    # Output vhdl shortcut for signals
    if ($vhdl) {
        if ($l == $r) {
            push (@vhdl_output, "${name}_code <= signals($l);");
        }
        else {
            push (@vhdl_output, "${name}_code <= signals($l downto $r);");
        }
    }

    say "$name $l .. $r" if $debug;


    $positions{$name} = [$r .. $l];

    $l = $r - 1;
}

if ($vhdl) {
    say $_ for (reverse @vhdl_output);
    say "";
```

```
}

say "" if $debug;

# Reverse the positions (we're using strings 0 indexed to the left but vhdl uses bits
    reversed)
for my $key (keys %positions) {
    my @mod;

    for my $val (@{$positions{$key}}) {
        push (@mod, $signallength - $val - 1);
    }

    # Need to reverse here as otherwise we'll assign eg (43, 42) to something which
        will reverse our code
    $positions{$key} = [reverse @mod];
}

# Convenience function, take reference to list and a string
sub update {
    my ($signal, $pos, $what) = @_;

    @$signal[ @{$pos} ] = split (//, $what);
}

open my $in, '<', $src or die "Couldn't open file $src $!";

my $codeline = 0;
my $rows_since_help = 0;
my $last_was_code = 0;
my $header_shown = 0;

# Collect output here for postprocessing  etc
my @output;

# Push lines here after first read through
my @lines;

# Labels with their line of code
my %labels;

# Search for labels and log their address
while (my $line = <$in>) {
    chomp $line;

    # Ignore comments
    if ($line =~ /^;/) {
        push (@lines, $line);
    }
    # And empty lines
    elsif ($line =~ /^\s*$/) {
        push (@lines, $line);
    }
    # It's a line of code!
    else {
        # Remove comments
```

```
        my ($code) = $line =~ /^([^;]*)/;

        # If we have a label
        if ($code =~ /^:([A-Z0-9]+)\s+(.*)/) {
            # Store it's address (start with 0)
            $labels{$1} = $codeline;

            # Remove label and push it
            push (@lines, $2);
        }
        else {
            push (@lines, $code);
        }

        $codeline++;
    }
}

# List labels
if ($debug) {
    say "Labels:";
    for my $k (keys %labels) {
        say "labels{$k} = " . dec2hex ($labels{$k});
    }
}

if ($vhdl) {
    my %instructions = (
        DAT => "0000",
        MOV => "0001",
        ADD => "0010",
        SUB => "0011",
        JMP => "0100",
        JMPZ => "0101",
        JMPN => "0110",
        CMP => "0111",
        SLT => "1000",
        DJN => "1001",
        SPL => "1010",
    );

    # Create a sorted pair instr -> code
    my @pairs = sort { $a->[1] cmp $b->[1] }
                map { [ $_ => $instructions{$_} ] } keys %instructions;

    for (@pairs) {
        my ($instr, $code) = (@$_);

        die "No $instr label!" if !exists $labels{$instr};

        my $pos = dec2bin ($labels{$instr});
        $pos = '0' x (8 - length($pos)) . $pos;

        my $hpos = dec2hex ($labels{$instr});

        say "\"$pos\" when \"$code\", -- $instr $hpos";
```

```
    }
    say '"11111111" when others;';

    exit;
}

$codeline = 0;

# Process and write
for my $line (@lines) {

    # Comments
    if ($line =~ /^;(.*)/) {
        # If we're in verbose, simply output comments
        if ($verbose) {
            #say $line;
            push (@output, $line);
        }
        # Otherwise transform into vhdl comments
        else {
            push (@output, "-- " . trim ($1));
        }
        $last_was_code = 0;
        next;
    }
    # Simply output empty lines
    elsif ($line =~ /^\s*$/) {
        #say $line;
        push (@output, $line);
        $last_was_code = 0;
        next;
    }

    $codeline++;

    my @signal = ((0) x $signallength);
    my @comments;

    my $buss_used = 0;

    my $curr_mem_addr = "";
    my $mem_data_used = 0;
    my $mem_err = 0;

    my $alu_op = "";
    my $alu_err = 0;

    for my $cmd (split /\s*,\s*/, $line)
    {
        $cmd = trim ($cmd);

        # Don't process label definitions
        if ($cmd =~ /^:/) {
            push (@comments, $cmd);
        }
        # Grab single word affixes eg PC++, ALU--
```

```
elsif ($cmd =~ /^(\S+?)([-+<]+)$/) {
    my ($reg, $op) = ($1, $2);

    if (exists $registers{$reg}->{$op}) {
        update (\@signal, $positions{$reg}, $registers{$reg}->{$op});

        push (@comments, $cmd);
    }
    elsif ($reg =~ /ALU[12]?/) {
        $alu_err = 1 if $alu_op && $alu_op ne $op;
        $alu_op = $op;

        update (\@signal, $positions{$reg}, $ALU{$op});

        push (@comments, $cmd);
    }
    else {
        die "Unknown command: $cmd";
    }
}
# We have a single shorthand notation
elsif (exists $singles{$cmd}) {
    my $reg = $singles{$cmd};

    update (\@signal, $positions{$reg}, $registers{$reg}->{$cmd});

    push (@comments, $cmd);
}
# src -> dest
elsif ($cmd =~ /(\S+)\s*[=-]>\s*(\S+)/) {
    my ($src, $dest) = ($1, $2);

    # src -> buss
    if ($dest eq "buss" && exists $buss{$src}) {

        $buss_used++;

        update (\@signal, $positions{$dest}, $buss{$src});
        push (@comments, $cmd);
    }
    # src -> mem
    elsif ($dest eq "mem") {
        $mem_data_used++;

        if ($src =~ /OP|M1|M2/) {
            # Set mem to write
            update (\@signal, $positions{$mem_map{$src}}, $mem{write});

            push (@comments, $cmd);
        }
        else {
            die "Unknown command: $cmd";
        }
    }
    # mem -> src
    elsif ($src eq "mem") {
```

```
    $mem_data_used++;

    if ($dest =~ /OP|M1|M2/) {
        # Set mem to read
        update (\@signal, $positions{$mem_map{$dest}}, $mem{read});

        push (@comments, $cmd);
    }
    else {
        die "Unknown command: $cmd";
    }
}
# Handle direct
elsif (exists $registers{$dest}->{$src}) {
    update (\@signal, $positions{$dest}, $registers{$dest}->{$src});

    # load if ALU
    if ($dest =~ /^ALU/) {
        my $op = "load";
        update (\@signal, $positions{ALU}, $ALU{$op});
        $alu_err = 1 if $alu_op && $alu_op ne $op;
    }

    $buss_used++ if $dest eq "buss";

    if ($dest eq "mem_addr") {
        $mem_err = 1 if $curr_mem_addr && $curr_mem_addr ne $src;
        $curr_mem_addr = $src;
    }

    push (@comments, $cmd);
}
# Try to route through buss
elsif (exists $registers{$dest}->{buss} && exists $buss{$src}) {

    $buss_used++;

    # Update src -> buss
    update (\@signal, $positions{buss}, $buss{$src});

    # Update buss -> dest
    update (\@signal, $positions{$dest}, $registers{$dest}->{buss});

    # load if ALU
    if ($dest =~ /^ALU/) {
        my $op = "load";
        update (\@signal, $positions{ALU}, $ALU{$op});
        $alu_err = 1 if $alu_op && $alu_op ne $op;
    }

    # Check if mem_addr will get set
    if ($dest eq "mem_addr") {
        $mem_err = 1 if $curr_mem_addr && $curr_mem_addr ne $src;
        $curr_mem_addr = $src;
    }
```

```
        # Comment as src -> buss, buss -> dest
        push (@comments, "$src -> buss");
        push (@comments, "buss -> $dest");
    }
    else {
        die "Unknown command: $cmd";
    }
}
# ALUx += src or ALUx -= src
elsif ($cmd =~ /^(ALU[12])\s*(\+|-)=\s*(\S+)$/) {
    my ($alu, $op, $src) = ($1, $2, $3);

    # Check direct connection
    if (exists $registers{$alu}->{$src}) {

        $alu_err = 1 if $alu_op && $alu_op ne $op;
        $alu_op = $op;

        # Update data
        update (\@signal, $positions{$alu}, $registers{$alu}->{$src});

        # Update alu action
        update (\@signal, $positions{ALU}, $ALU{$op});

        push (@comments, $cmd);
    }
    # Try to route through buss
    elsif (exists $registers{$alu}->{buss} && exists $buss{$src}) {

        $buss_used++;
        $alu_err = 1 if $alu_op && $alu_op ne $op;
        $alu_op = $op;

        # Update src -> buss
        update (\@signal, $positions{buss}, $buss{$src});

        # Update buss -> alu
        update (\@signal, $positions{$alu}, $registers{$alu}->{buss});

        # load ALU
        update (\@signal, $positions{ALU}, $ALU{$op});

        push (@comments, $cmd);
    }
    else {
        die "Unknown command: $cmd";
    }
}
# TODO all jumps does not have addresses?
# Handle jumps eg jmp, jmp 0, jmp +1, jmpS -1, jmpIN
elsif ($cmd =~ /^(jmp\S*)\s+(\S+)/) {
    my ($jmp, $where) = ($1, $2);

    # Check that uPC has support for this jump
    if (exists $registers{uPC}->{$jmp}) {
```

```
# Check to see if we have a relative absolute address
$where =~ /^([+-])?(.*)/;
my $op = $1;
$op = "" if ! $op;
my $val = $2;

my $label = "";

# Convert label def
if ($val =~ /^\$([A-Z0-9]+)/) {
    $label = $1;

    if (exists $labels{$label}) {
        $val = dec2hex ($labels{$label});
    }
    else {
        die "Unknown label: $label";
    }
}

my $bin;
if ($val =~ /^[0123456789ABCDEF]{0,2}$/i) {
    $bin = hex2bin ($val);
}
elsif ($val =~ /^[01]+$/) {
    $bin = $val;
}
else {
    die "Unknown command: $cmd";
}

if ($op =~ /^[+-]$/) {
    my $curr_row = $codeline - 1;
    my $off = bin2dec ($bin);
    my $abs = $op eq "+" ? $curr_row + $off : $curr_row - $off;
    my $new_bin = dec2bin ($abs);
    my $new_hex = dec2hex ($abs);

    my $currhex = dec2hex ($curr_row);

    push (@output, "$curr_row($currhex) $op $off = $abs($new_hex) ->
        $new_bin") if $debug;

    $bin = $new_bin;
}

# Force to length 8
if (length($bin) < 8) {
    $bin = '0' x (8 - length($bin)) . $bin;
}
elsif (length($bin) > 8) {
    # Truncate from the back so 00 1111 1111 -> 1111 1111
    $bin = substr $bin, -8;
}

# Set jump address
```

```
            update (\@signal, $positions{uPC_addr}, $bin);
            # Set jump
            update (\@signal, $positions{uPC}, $registers{uPC}->{$jmp});

            # Include label name in comment
            if ($label) {
                my $addr = dec2hex ($labels{$label});
                push (@comments, "$jmp $label($addr)");
            }
            else {
                # Ugly I know ^^
                my $dec = bin2dec ($bin);
                my $hex = dec2hex ($dec);
                push (@comments, "$cmd($hex)");
            }

        }
        else {
            die "Unknown command: $cmd";
        }
    }
    # var = stuff eg uPC = 0
    elsif ($cmd =~ /(\S+)\s*=\s*(\S+)/) {
        my ($var, $res) = ($1, $2);

        if ($var eq "uPC_addr") {
            die "Unknown command: $cmd";
        }

        # Check special eg uPC = 0
        if (exists $registers{$var} && exists $registers{$var}->{$res}) {
            update (\@signal, $positions{$var}, $registers{$var}->{$res});

            push (@comments, $cmd);
        }
        # ALU = x
        elsif (exists $ALU{$res}) {
            update (\@signal, $positions{ALU}, $ALU{$res});

            push (@comments, $cmd);
        }
        else {
            die "Unknown command: $cmd";
        }
    }
    else {
        die "Unknown command: $cmd";
    }
}

# Can only address all memory with one address at a time
if ($mem_err) {
    push (@comments, "! 2x -> mem_addr !");
}
# Check that we're only using our buss once
if ($buss_used > 1) {
```

```
    push (@comments, "! 2x -> buss !");
}
# Check only one operation for the alu
if ($alu_err) {
    push (@comments, "! 2x alu op !");
}

# Output verbose mode, for humans
if ($verbose) {
    # Output verbose output, format lines like this with the occassional help
        header

    if (!$header_shown || $rows_since_help > $lines_until_header && !
        $last_was_code) {
        #say "    $h";
        push (@output, "    $h");

        $header_shown = 1;
        $rows_since_help = 0;
    }

    $last_was_code = 1;
    $rows_since_help++;

    my $result = "";
    my $last = 0;
    my $signal = join ("", @signal);

    my @codechunks = split (/\s+/, $c);
    my @spacechunks = split (/\S+/, $c);

    # Remove if there's an opening space
    if ($c =~ /^\s/) {
        $result .= shift @spacechunks;
        shift @codechunks;
    }
    # Will split out an empty string space otherwise
    else {
        shift @spacechunks;
    }

    # Bundle a code string by alternating code/space
    for my $code (@codechunks) {
        my $l = length($code);
        my $sig = substr ($signal, $last, $l);

        $result .= $sig;

        my $space = shift @spacechunks;
        $result .= $space if $space;

        $last += $l;
    }

    my $hexline = dec2hex ($codeline - 1);
```

```
        #say "$hexline  $result ; " . join (", ", @comments);
        push (@output, "$hexline  $result ; " . trim (join (", ", @comments)));
    }
    # Output for vhdl copy paste
    else {
        my $res = '"' . join ("", @signal) . '", -- ' . trim (join (", ", @comments));
        #say $res;
        push (@output, $res);
    }
}

# Print output
my $txt = join ("\n", @output);
say $txt;

sub dec2hex {   # Force at least length 2
    my $d = shift;
    my $h = sprintf ("%x", $d);
    $h = "0$h" if length ($h) < 2;
    return $h;
}
sub hex2bin {
    my $h = shift;
    my $hlen = length($h);
    my $blen = $hlen * 4;
    return unpack("B$blen", pack("H$hlen", $h));
}
sub dec2bin {
    my $str = unpack("B32", pack("N", shift));
    $str =~ s/^0+(?=\d)//;   # otherwise you'll get leading zeros
    return $str;
}
sub bin2dec {
    return unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
sub trim {
    my $string = shift;
    $string =~ s/^\s+//;
    $string =~ s/\s+$//;
    return $string;
}
```