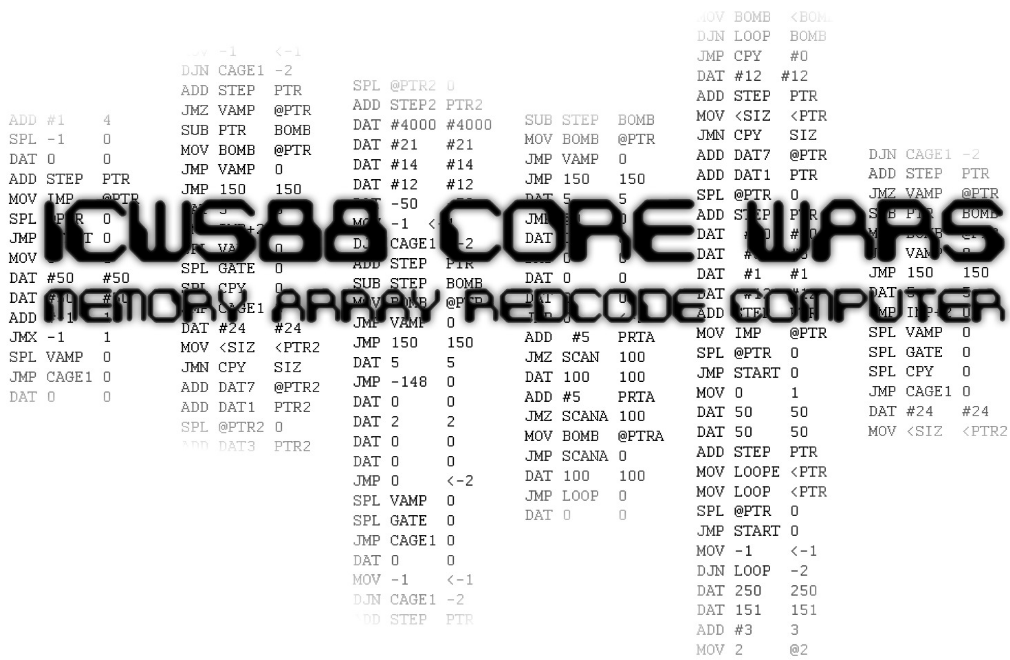
TSEA43

Projektrapport



Jonas Hietala
Jesper Tingvall
Jizhi Li

15 April, 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.

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.

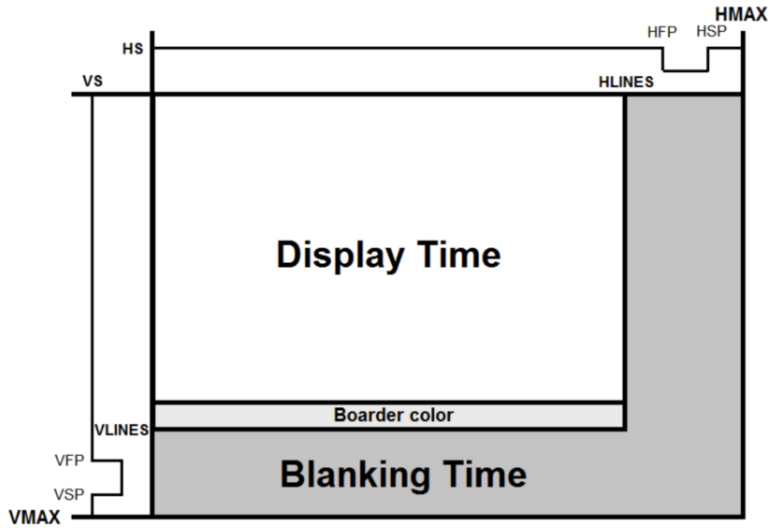
Core Cleaner
En core cleaner är ett program som databombar hela minnet. Ofta går man igenom minnet två gånger, den första fyller man minnet med split instruktioner för att slöa ner motståndaren och sedan med DAT-instruktioner för att göra slut på honom.

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] http://kopimistsamfundet.se

3 Teori

3.1 VGA



Figur 1: Display timing

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®.

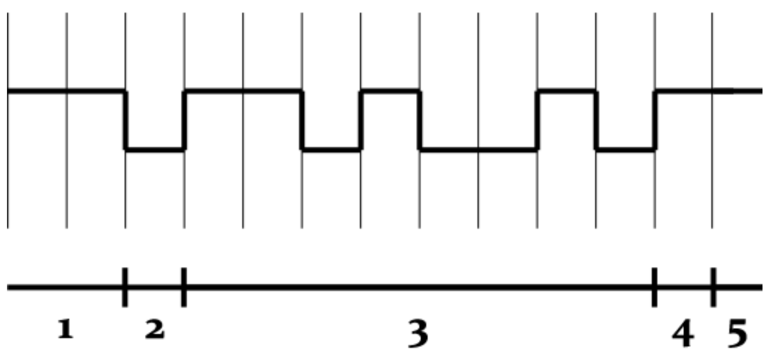
- HMAX: 800
- VMAX: 525
- HLINES: 640
- VLINES: 480
- HFP: 648
- HSP: 744
- VFP: 482
- VSP: 484
- Clk: 25MHz

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.

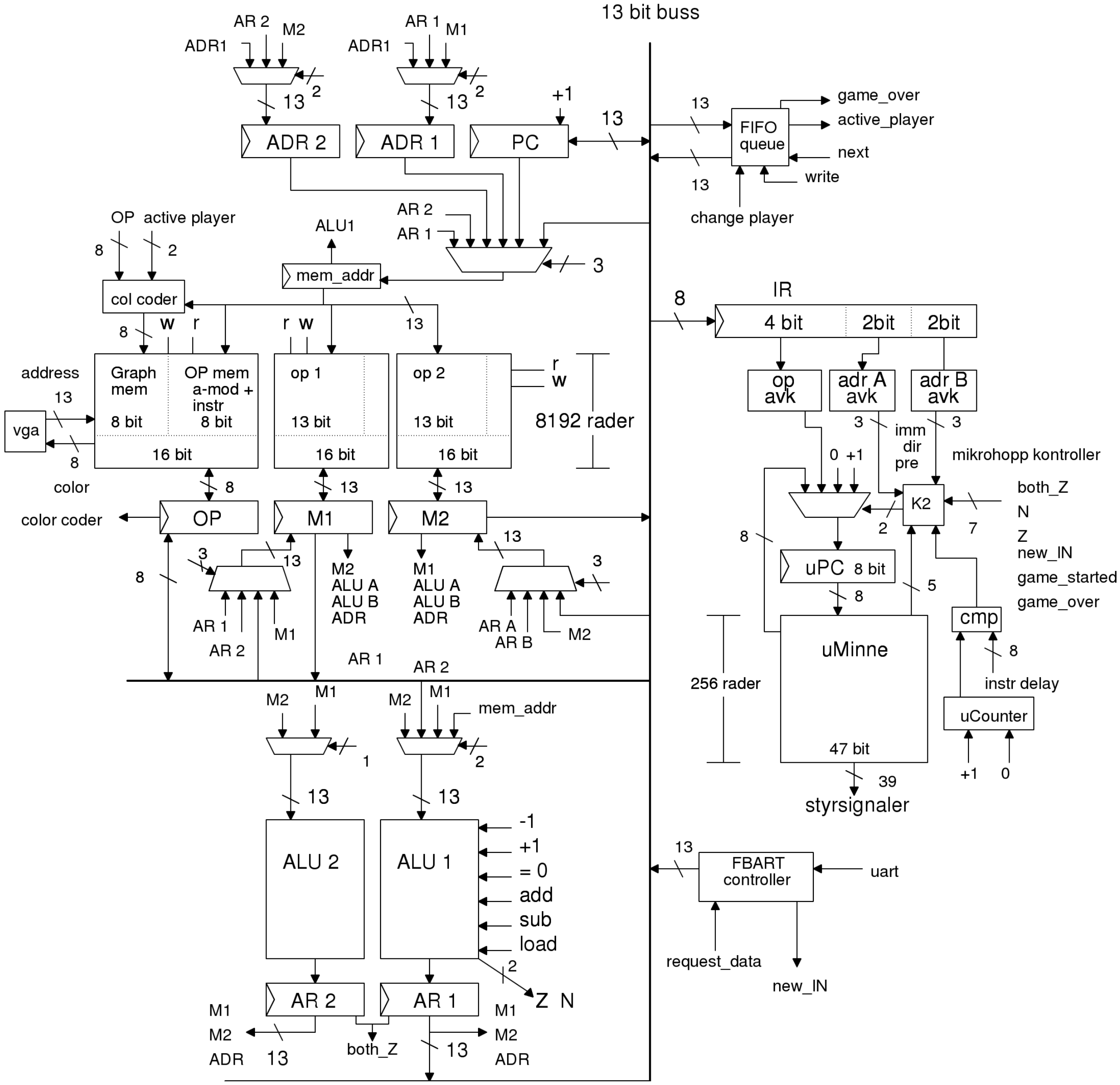


Figur 2: En RS232 överförning. Man har ingen gemensam klocka för sändare och mottagare utan överför endast data, sändaren och mottagaren känner dock till vilken baud rate man överför med. I vårt fall använder vi 115 200 baud.

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.

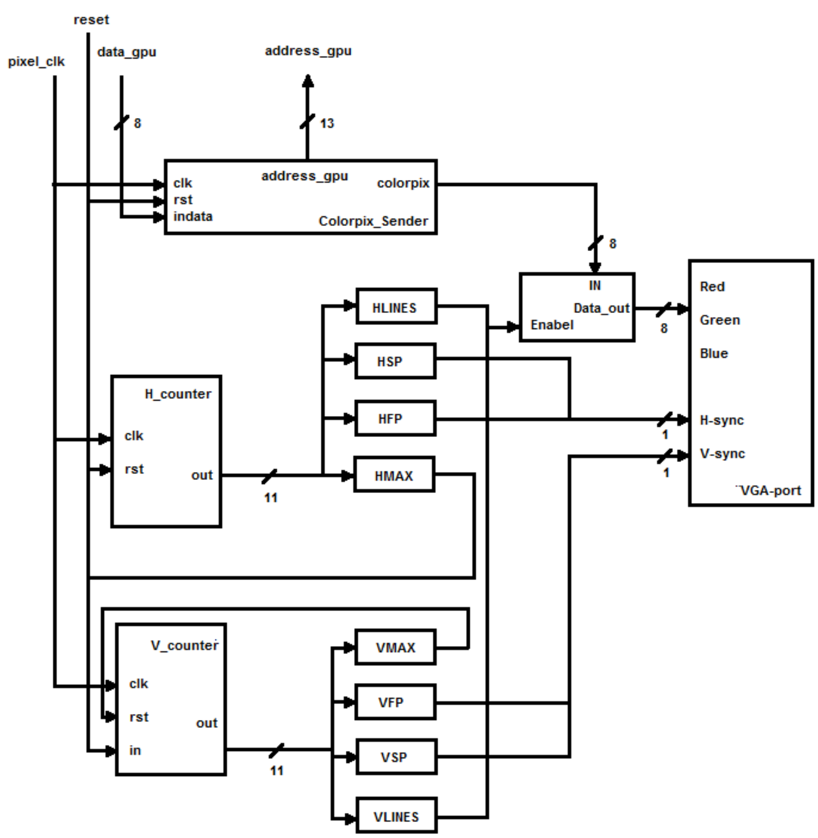
Vid exekvering av en instruktion laddas instruktionen först in till IR, sedan beräknas adresseringsmoderna för A och B och därefter utförs instruktionen. Adressmodsberäkningen är besvärlig då både A och B operanderna kan vara en av de fyra olika moderna. Detta kompliceras ytterligare då vissa instruktioner 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 4 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: VGA blockschema

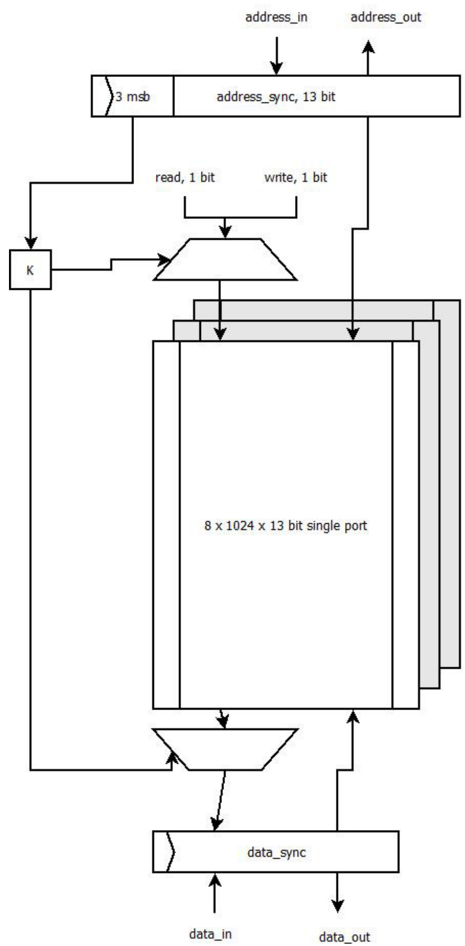
VGA-port kommer endast ta emot färg data när h_counter ¡ HLINES(640 enligt upplösning vi valde) och v_counter ¡ VLINES(480 enligt upplösning vi valde) med hjälp av en enable signal från HLINE och VLINE. Pixelsender använder samma timing och klocka som vga_controller och skickar en 13 bitars adress till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

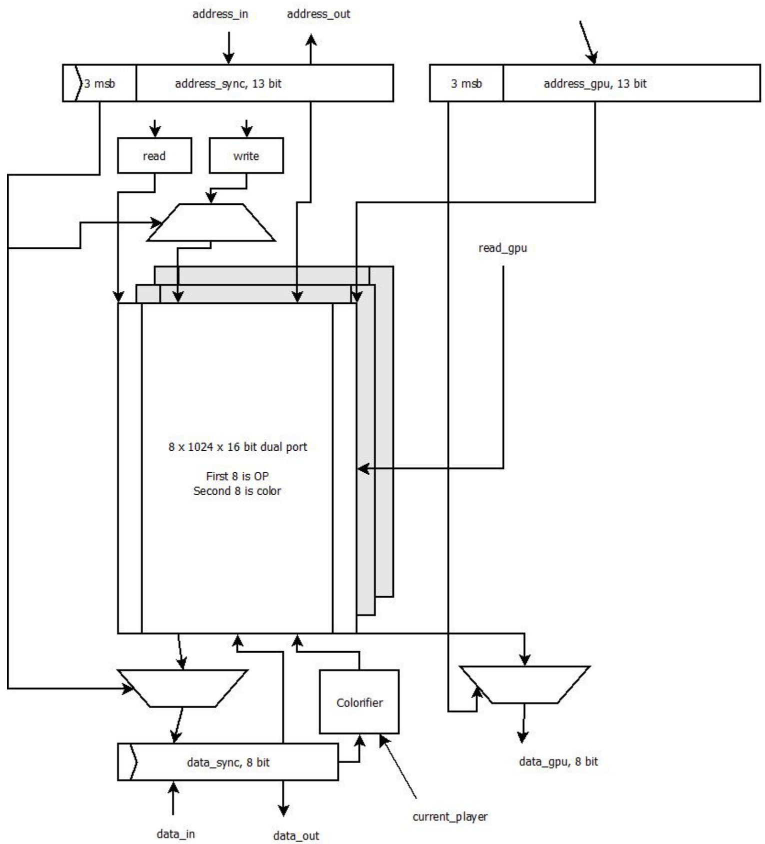
Varje rad Redcode delades upp i 4 delar; instruktion och adresseringsmoder på 8 bitar, operand A på 13 bitar, operand B på 13 bitar och 8 bitar RGB färgning. Det som bäst stämde överens med vår uppdelning var att använda minnesblock utav storleken 1024 x 16 bitar (de 3 sista bitarna används ej dock i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

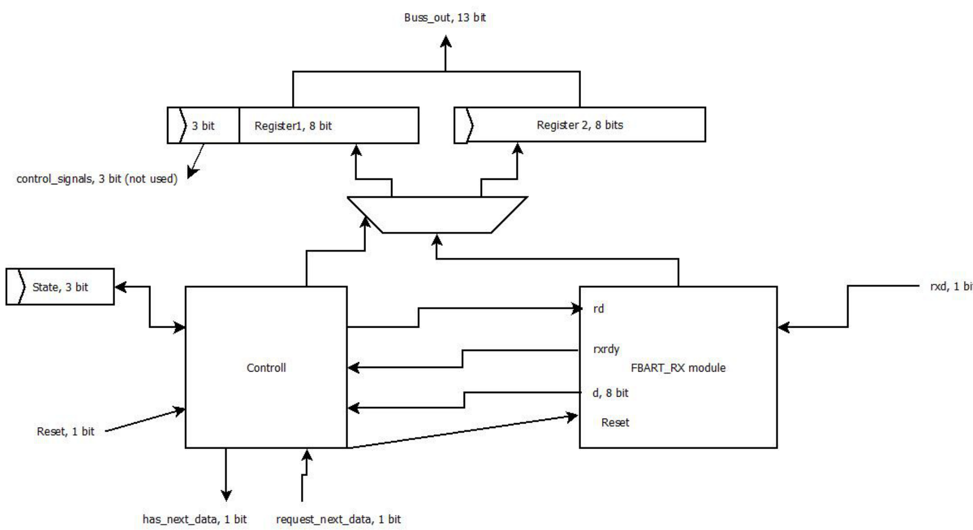
Då vi har olika färg beroende på vilken instruktion vi har i instruktionsminnet var det naturligt att slå samman instruktionsminnet och färgminnet då båda var på 8 bitar. Den resulterande maskinen ses i figur 6. Skillnaden mellan den och operandminnena är att den använder ett dualportminne med den andra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



Figur 6: Instruktions och färgminne

4.4 UART

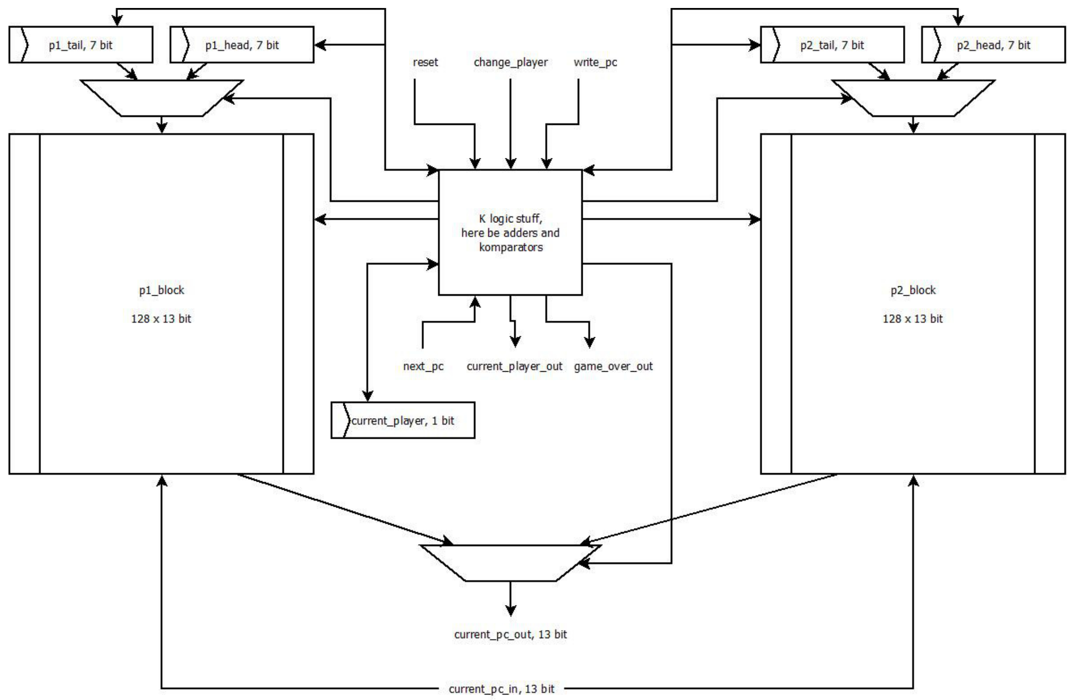
Vår dator använder en 13 bitars buss, det skulle därmed vara trevligt om indatat ifrån vår värddator skulle vara 13 bitar detta med. Då vi använder Anders Nilssons FBART vilken arbetar i 8 bitar skulle det vara trevligt att slå samman två sändningar till en. Det gör vi med modulen i figur 7. Modulen 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.



Figur 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.

A Warriors

A.1 Little bomber

```
;name Little bomber
;author Jesper Tingvall
;description Bombs the memory backwards with DAT 0, 0. Smaller than a dwarf

JMP  LOOP,     0
BOMB DAT 0,    0
LOOP MOV BOMB, <BOMB
     DJN LOOP, BOMB
```

A.2 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.3 Carpet bomber

```
;name carpet bomber
;author Jonas Hietala

step    EQU -31

start   ADD #step,scan
scan    JMZ start,-100

bombit  MOV bomb, @scan
        MOV loop, <scan
        MOV move, <scan
        SPL @scan
        JMP start

move    MOV bomb, <bomb
loop    DJN move, bomb
bomb    DAT #0,#move
```

A.4 Core Cleaner

```
;name Cleaner
;description A core cleaner. Will split bomb then dat bomb the whole memory except
    ourselves
;author Jonas Hietala

size    EQU     -9

start   MOV     #size, target   ; setup bomb ptr
infect  MOV     split, <target  ; split bomb
        JMN     infect, target

        MOV     #size, target   ; setup bomb ptr
kill    MOV     bomb, <target   ; dat bomb
        JMN     kill, target
        JMP     start, 0        ; do everything over again

split   SPL     0, 0
bomb    DAT     10, 10
target  DAT     0, 0

end
```

A.5 Dwarf scout

```
;name Dwarf Scout
;description Will dwarf bomb, and flee if it spots any changes.
;author Jonas Hietala

dist        EQU start-160               ; how for ahead of us will the carpet be laid
csize       EQU 8                       ; carpet size
copydist    EQU 3783                    ; when we flee, how far?
length      EQU last-start
bombstep    EQU 3

start       MOV watcher, watch          ; reset watch vector
            MOV #csize, len1
laycarp     MOV carpet,<watch           ; lay a carpet to watch
len1        DJN laycarp,#csize

            MOV bombloc,bombjmp         ; restore jmp vector for bomber
            SPL bomber                  ; split out a dwarf bomber

scan        MOV watcher, watch          ; reset watch vector
            MOV #csize, len2            ; reset cmp for loop
scancarp    CMP carpet, <watch          ; check for intruders
            JMP evacuate                ; something happened!!
len2        DJN scancarp, #csize        ; keep checking the length of carpet
            JMP scan

evacuate    MOV last, bombjmp           ; kill our bomber, we're running out of time!
            MOV #0, last                ; reset last pointer
            MOV #last + copydist, new   ; reset new pointer
```

```
            MOV #length, len3           ; reset prog length
            MOV @last, @new             ; copy over whole program
copy        MOV <last, <new
len3        DJN copy, #length

new         SPL @0,last + copydist      ; and split there (acts as a jump)

bomber      MOV #last+1,bomb            ; setup bomb vector
dobomb      ADD #bombstep, bomb         ; add in bombstep
            MOV bomb, @bomb             ; bomb
bombjmp     JMP dobomb                  ; continue to bomb

bombloc     JMP -2                      ; bomb jmp backup
bomb        DAT #0,#last+1              ; bomb

carpet      DAT #237,#986               ; our inique id to lay down
watcher     DAT #dist,#dist             ; where to place our carpet
watch       DAT #0,#0                   ; where are we watching?

last        DAT #0
```

A.6 Stone of ages

```
;name Stone of ages
;author Jonas Hietala
;description Will sparsely bomb the memory and then fall back to a core cleaner.

bombstep    EQU 37
size        EQU -15


;try for a quick kill
bomber      ADD #bombstep, dst
;            MOV bomb, @dst
            MOV split, @dst
            SLT dst, #2*bombstep
            JMP bomber
            JMP clean

clean       MOV #size, dst
infect      MOV split, <dst
            JMN infect, dst

bashmem     MOV #size, dst
kill        MOV bomb, <dst
            JMN kill, dst

            JMP bashmem

split       SPL 0, 0
bomb        DAT 10, 10
dst         DAT #bombstep, #bombstep
```

A.7 Jumper

```
;name Jumper
;author Jesper Tingvall
;description Proof of concept replicator, creates a copy of itself and kills itself.
    The constants get corrupted after a while, causing the replicator to kill itself.

START JMP CPY,  #0
SIZ   DAT #12,  #12
      ADD STEP, PTR
CPY   MOV <SIZ, <PTR
      JMN CPY,  SIZ
      ADD DAT7, @PTR
      ADD DAT1, PTR
      SPL @PTR, 0
      ADD STEP, PTR
PTR   DAT #50,  #50
STEP  DAT #8,   #8
DAT1  DAT #1,   #1
DAT7  DAT #12,  #12
```

A.8 Jumper gate

```
;name Jumper-gate
;author Jesper Tingvall
;description Proof of concept replicator, creates a copy of itself and becomes a gate
    after copying. Might work, might not...

START JMP CPY,  #0
SIZ   DAT  #13, #13
      ADD STEP, PTR
CPY   MOV <SIZ, <PTR
      JMN CPY,  SIZ
      ADD DAT7, @PTR
      ADD DAT1, PTR
      SPL @PTR, 0
      ADD STEP, PTR
      JMP 0,    <-2
PTR   DAT  #50, #50
STEP  DAT  #7,  #7
DAT1  DAT  #1,  #1
DAT7  DAT  #12, #12
```

A.9 Inseminator

```
;name Inseminator
;author Jesper Tingvall
;description This one gets all lovey-dovey with the enemy warrior, inseminate that
    code with our PCs!

START ADD STEP,  PTR
      JMZ START, @PTR
      SPL @PTR,  #0
      JMP START, 0
PTR   DAT #5,    #5            ; Pointer
STEP  DAT #5,    #5            ; Data to add to pointer
```

A.10 Kopimi

```
;name Kopimi
;author Jesper Tingvall
;description Scans the memory after code and creates a copy of it!
;              a
;            a C a
;          a a a a a
;        a a a a a a a
;      a a a a a a a a a

START SUB STEP,   PTR
      JMZ START,  @PTR
COPY  MOV @PTR,   <SPAWN
    JMN COPY,   <PTR
      SPL @SPAWN, #2
      JMP START,  3
PTR   DAT #-50,   #-50      ; Scan Pointer
SPAWN DAT #4000,  #4000     ; Copy Pointer
STEP  DAT #1,     1
```

A.11 Replicator

```
;name Replicator
;description Watch stargate dawg
;author Jonas Hietala

step    EQU 417
init    EQU 1337
size  EQU 9

        JMP start           ; boot jump as we can't specify PC in the middle T.T

src     DAT 0               ; src pointer
start   MOV #size, src       ; setup src pointer
copy    MOV @src, <dst      ; copy self
        DJN copy, src
        SPL @dst            ; throw a pc there

        ADD #step, dst      ; space out a bit
        JMP start           ; make a new copy, yay!

dst     DAT #0, #init       ; dst pointer

end
```

A.12 Scanner

```
;name Scanner 1 / Stealth Bomber
;author Jesper Tingvall
;description Scans the memory until it finds something != 0, then bomb it with DAT 100
     100!

scanA ADD  #5,   ptrA
```

```
ptrA  JMZ scanA, 100    ; scan 1 location every 2 cycles
      MOV BOMB,  @ptrA  ; Attack
      JMP scanA, 0
BOMB  DAT 100,   100
```

A.13 Vampire

```
;name Vampire
;author Jesper Tingvall
;description Proof of concept Vampire, bombs memory forward with JMP instructions that
     traps PCs in a cage that DAT 0 0 bombs the memory backwards. Vampin' is slow a f
    ***.

JMP     VAMP,           0
        DAT 0,          0
CAGE1   MOV -1,         <-1
CAGE2   DJN CAGE1,      -2
VAMP    ADD STEP,       PTR
        SUB STEP,       BOMB
        MOV BOMB,       @PTR
        JMP VAMP,       0
PTR     JMP 150,        150
STEP    DAT 5,          5
BOMB    JMP CAGE1-148,  0
```

A.14 Suck on this

```
;name Suck on this!
;description A clever vampire which will make you bomb for it, and it will start
    bombing itself as well. Will also change stuff in memory to screw your code.
;author Jonas Hietala

decoy   EQU 1337                ; fill mem with random numbers
step    EQU 211
mem     EQU start - 2000        ; we will change B op in memory with DJN from here

start   SPL 0                   ; process generator, necessary for we will go into
vamp    ADD stepp, ptr          ; add in offset
  MOV ptr, @ptr           ; lay our fang
        DJN vamp, <mem          ; DJN bomb

ptr     JMP trap, ptr

trap    SPL 1, -100             ; suck the life out of them
        MOV bomb, <trap         ; make them bomb for us
        JMP trap                ; forever...

bomb    DAT #decoy,#-decoy      ; lay out decoys to defeat anti-vampires
stepp   DAT #step,#step
```

A.15 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
```

A.16 Imp worm

```
;name Earthworm Jim
;description Will setup a length 3 imp worm. Gogo!
;author Jonas Hietala

start   SPL second
        SPL third
        JMP imp
third   JMP imp+1
second  SPL fourth
        JMP imp+2
fourth  JMP imp+3
imp     MOV 0,1

end
```

A.17 Vampire bomber

```
;name Vampire bomber
;author Jesper Tingvall
;description Same as vampire but splits in start to cage, making this DAT 0 0 and JMP
    at the same time per default.

        SPL VAMP, 0
        JMP CAGE1, 0
        DAT 0,0
CAGE1   SPL 0,<-4
  MOV -2, <-1
CAGE2   JMP CAGE1,<-6
VAMP    ADD STEP, PTR
        SUB STEP, BOMB
        MOV BOMB, @PTR
        JMP VAMP, 0
PTR     JMP 150,150
STEP    DAT 5,5
BOMB    JMP CAGE1-148, 0
```

A.18 Vampire bomber gate

```
;name Vampire bomber gate
;author Jesper Tingvall
;description Same as vampire bomber but also splits to a gate.

        SPL VAMP,      0
        SPL GATE,      0
        JMP CAGE1,     0
        DAT 0,         0
CAGE1   MOV -1,        <-1
CAGE2   DJN CAGE1,     -2
VAMP    ADD STEP,      PTR
        SUB STEP,      BOMB
        MOV BOMB,      @PTR
        JMP VAMP,      0
PTR     JMP 150,       150
STEP    DAT 5,         5
BOMB    JMP CAGE1-148, 0
        DAT 0,         0
        DAT 0,         0
        DAT 0,         0
        DAT 0,         0
GATE    JMP 0,         <-2
```

A.19 Vampire bomber gate replicator

```
;name Vampire bomber gate replicator
;author Jesper Tingvall
;description Same as vampire bomber gate but this creates a replica of the vampire
   bomber part (srsly, this is getting silly - I call this code bloatware)

      SPL VAMP,      0
      SPL GATE,      0
      SPL CPY,       0
      JMP CAGE1,     0
SIZ   DAT #29,       #29
CPY   MOV <SIZ,      <PTR2
      JMN CPY,       SIZ
      ADD DAT7,      @PTR2
      ADD DAT1,      PTR2
      SPL @PTR2,     0          ; cage
      ADD DAT3,      PTR2
      SPL @PTR2,     0          ; vampire

      ADD DATX,      PTR2
    SPL @PTR2,     0
PTR2  DAT  #4000+8192, #4000+8192
STEP2 DAT  #21,      #21
DAT1  DAT  #14,      #14
DAT7  DAT  #12,      #12
      DAT -50,       -50
CAGE1 MOV -1,        <-1
CAGE2 DJN CAGE1,     -2
VAMP  ADD STEP,      PTR
```

```
      SUB STEP,      BOMB
      MOV BOMB,      @PTR
      JMP VAMP,      0
PTR   JMP 150,       150
STEP  DAT 5,         5
BOMB  JMP CAGE1-148, 0
      DAT 0,         0
DAT3  DAT 2,         2
DATX  DAT 76,        76
      DAT 0,         0
GATE  JMP 0,         <-2
```

A.20 The big maker

```
;name The big maker
;description A vampire which spawns dwarfs and an imp.
;author Jonas Hietala

bombstep    EQU 4

firstdwarf  EQU 2431
dwarfstep   EQU 793

pitbomb     EQU dwarf - 100
djnbomb     EQU vamp - 1
impstart    EQU decoy + 10

            JMP boot

; description of a dwarf
dwarf       ADD #bombstep, bomb
            MOV bomb, @bomb
            JMP dwarf
bomb        DAT #1,#1

; an imp!
imp         MOV 0,1

; decoy
            DAT #1,#1
            DAT #1,#1
            DAT #1,#1
            DAT #1,#1
            DAT #1,#1
            DAT #1,#1

; launch out 3 dwarfs
boot        MOV bomb, <dwarfcp1     ; launch out a dwarf
            MOV bomb-1, <dwarfcp1
            MOV bomb-2, <dwarfcp1
            MOV bomb-3, <dwarfcp1

            MOV bomb, <dwarfcp2     ; launch out a dwarf
            MOV bomb-1, <dwarfcp2
            MOV bomb-2, <dwarfcp2
```

```
            MOV bomb-3, <dwarfcp2

            MOV bomb, <dwarfcp3     ; launch out a dwarf
            MOV bomb-1, <dwarfcp3
            MOV bomb-2, <dwarfcp3
            MOV bomb-3, <dwarfcp3

; launch an imp
            MOV imp, impstart
            SPL @0, impstart

; add processes to dwarfs
dwarfcp1    SPL @0, firstdwarf
dwarfcp2    SPL @0, firstdwarf + dwarfstep
dwarfcp3    SPL @0, firstdwarf + 2 * dwarfstep

            SPL 0                   ; process generator

; vampire bomber
vamp        MOV fang, @fang
            ADD fangstep, fang
            DJN vamp, <djnbomb
fang        JMP pit, fang

fangstep    DAT #-3*bombstep, #-3*bombstep

pit         SPL 1, pitbomb          ; trap processes here
            MOV decoy, <pit         ; make them bomb for us
            JMP pit

decoy       DAT #1,#1

end
```

B VHDL

B.1 MARC.vhd

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

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MARC is
    Port (  clk : in std_logic;
            reset_a : in std_logic;

            uCount_limit : in std_logic_vector(7 downto 0);
            fbart_in : in std_logic;

            -- VGA output
            red : out std_logic_vector(2 downto 0);
            grn : out std_logic_vector(2 downto 0);
            blu : out std_logic_vector(1 downto 0);
            HS : out std_logic;
            VS : out std_logic;

            -- Test upstart load without fbart
            --tmp_has_next_data : in std_logic;
            --tmp_IN : in std_logic_vector(12 downto 0);
            --tmp_request_next_data : out std_logic;

            -- Output flags etc
            reset_out : out std_logic;
            game_started_out : out std_logic;
            active_player_out : out std_logic_vector(1 downto 0);
            pad_error : out STD_LOGIC_VECTOR(2 downto 0);
            game_over_out : out std_logic;
            alu1_o : out stD_LOGIC_VECTOR(7 downto 0);
            player_victory_out : out std_logic_vector(1 downto 0);

            -- Hex display output
            ca,cb,cc,cd,ce,cf,cg,dp : out  STD_LOGIC;
            an : out  STD_LOGIC_VECTOR (3 downto 0)
    );
end MARC;


architecture Behavioral of MARC is

    --------------------------------------------------------------------------
    -- COMPONENTS
    --------------------------------------------------------------------------

    component Microcontroller
        Port (  clk : in std_logic;
                reset_a : in std_logic;
                buss_in : in std_logic_vector(7 downto 0);
```

```
            uCount_limit : in std_logic_vector(7 downto 0);

            PC_code : out std_logic_vector(1 downto 0);
            buss_code : out std_logic_vector(2 downto 0);

            ALU_code : out std_logic_vector(2 downto 0);
            ALU1_code : out std_logic_vector(1 downto 0);
            ALU2_code : out std_logic;

            memory_addr_code : out std_logic_vector(2 downto 0);

            memory1_write : out std_logic;
            memory2_write : out std_logic;
            memory3_write : out std_logic;

            memory1_read : out std_logic;
            memory2_read : out std_logic;
            memory3_read : out std_logic;

            OP_code : out std_logic;
            M1_code : out std_logic_vector(1 downto 0);
            M2_code : out std_logic_vector(1 downto 0);

            ADR1_code : out std_logic_vector(1 downto 0);
            ADR2_code : out std_logic_vector(1 downto 0);

            FIFO_code : out std_logic_vector(1 downto 0);

            game_code : out std_logic_vector(1 downto 0);

            Z : in std_logic;
            N : in std_logic;
            both_Z : in std_logic;

            new_IN : in std_logic;
            game_started : in std_logic;
            shall_load : in std_logic;
            game_over : in std_logic;

            current_instr : out std_logic_vector(3 downto 0)
    );
end component;

component ALU
    Port (  clk : in std_logic;

            alu_operation : in std_logic_vector(1 downto 0);
            alu1_zeroFlag_out : out std_logic;
            alu1_negFlag : out std_logic;
            alu_zeroFlag : out std_logic;

            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 component;

component Memory_Cell
    Port (  clk : in std_logic;
            read : in std_logic;
            write : in std_logic;

            reset : in std_logic;

            address_in : in std_logic_vector(12 downto 0);
            address_out : out std_logic_vector(12 downto 0);

            data_in : in std_logic_vector(12 downto 0);
            data_out : out std_logic_vector(12 downto 0)
    );
end component;

component Memory_Cell_DualPort
    Port (  clk : in std_logic;
            read : in std_logic;
            write : in std_logic;

            active_player : in std_logic_vector(1 downto 0);

            address_in : in std_logic_vector(12 downto 0);
            address_out : out std_logic_vector (12 downto 0);

            data_in : in std_logic_vector(7 downto 0);
            data_out : out std_logic_vector(7 downto 0);

            address_gpu : in std_logic_vector(12 downto 0);
            data_gpu : out std_logic_vector(7 downto 0);
            read_gpu : in std_logic
    );
end component;

component FBARTController
    Port (  request_next_data : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            clk : in  STD_LOGIC;
            control_signals : out  STD_LOGIC_VECTOR (2 downto 0);
            buss_out : out  STD_LOGIC_VECTOR (12 downto 0);
            has_next_data : out  STD_LOGIC;
            rxd : in std_logic;
            padding_error_out : out STD_LOGIC_VECTOR(2 downto 0)
        );
 end component;

 component PlayerFIFO is
     Port ( current_pc_in : in  STD_LOGIC_VECTOR (12 downto 0);
            current_pc_out : out  STD_LOGIC_VECTOR (12 downto 0);
            current_player_out : out STD_LOGIC;
            game_over_out : out STD_LOGIC;
            next_pc : in  STD_LOGIC;
```

```
            write_pc : in STD_LOGIC;
            change_player : in STD_LOGIC;
            clk : in std_logic;
            reset : in std_logic
        );
 end component;

  component vga is
    Port (  rst : in  STD_LOGIC;
            clk : in  STD_LOGIC;
            data_gpu : in  STD_LOGIC_VECTOR (7 downto 0);
            address_gpu : out  STD_LOGIC_VECTOR (12 downto 0);
    border_color : in std_logic_vector (7 downto 0);
            red : out  STD_LOGIC_VECTOR (2 downto 0);
            grn : out  STD_LOGIC_VECTOR (2 downto 0);
            blu : out  STD_LOGIC_VECTOR (1 downto 0);
            HS : out  STD_LOGIC;
            VS : out  STD_LOGIC
         );
    end component;

component MARCled is
    Port ( clk,rst : in  STD_LOGIC;
        ca,cb,cc,cd,ce,cf,cg,dp : out  STD_LOGIC;
        an : out  STD_LOGIC_VECTOR (3 downto 0);

        game_started : in std_logic;
        current_instr : in std_logic_vector(3 downto 0);

        game_over : in std_logic;
        active_player : in std_logic_vector(1 downto 0)
    );
end component;

-------------------------------------------------------------------------
-- DATA SIGNALS
-------------------------------------------------------------------------

-- Module data signals
signal ALU_in : std_logic_vector(12 downto 0);
signal ALU1_out : std_logic_vector(12 downto 0);
signal ALU2_out : std_logic_vector(12 downto 0);

signal ALU_operation : std_logic_vector(1 downto 0);
-- 00 hold
-- 01 load main buss
-- 10 +
-- 11 -

signal memory1_data_in : std_logic_vector(7 downto 0);
signal memory2_data_in : std_logic_vector(12 downto 0);
signal memory3_data_in : std_logic_vector(12 downto 0);

-- Combined to one for now
signal memory_address_in : std_logic_vector(12 downto 0);
signal memory_address : std_logic_vector(12 downto 0);
```

```vhdl
----------------------------------------------------------------------
-- FLOW CONTROL
----------------------------------------------------------------------

-- Flow control signals
signal main_buss : std_logic_vector(12 downto 0) := "1X1X1X1X1X1X1";

-- Registers
signal PC : std_logic_vector(12 downto 0);
signal ADR1 : std_logic_vector(12 downto 0);
signal ADR2 : std_logic_vector(12 downto 0);

-- Memory outputs register values
signal OP : std_logic_vector(7 downto 0);
signal M1 : std_logic_vector(12 downto 0);
signal M2 : std_logic_vector(12 downto 0);

signal IN_reg : std_logic_vector(12 downto 0);

----------------------------------------------------------------------
-- CONTROL SIGNALS
----------------------------------------------------------------------

signal PC_code : std_logic_vector(1 downto 0);

signal buss_code : std_logic_vector(2 downto 0);

signal ALU_code : std_logic_vector(2 downto 0);
signal ALU1_code : std_logic_vector(1 downto 0);
signal ALU2_code : std_logic;

signal alu1_operand : std_logic_vector(12 downto 0);
signal alu2_operand : std_logic_vector(12 downto 0);

signal memory_addr_code : std_logic_vector(2 downto 0);

signal memory1_write : std_logic;
signal memory2_write : std_logic;
signal memory3_write : std_logic;

signal memory1_read : std_logic;
signal memory2_read : std_logic;
signal memory3_read : std_logic;

signal OP_code : std_logic;
signal M1_code : std_logic_vector(1 downto 0);
signal M2_code : std_logic_vector(1 downto 0);

signal ADR1_code : std_logic_vector(1 downto 0);
signal ADR2_code : std_logic_vector(1 downto 0);

signal FIFO_code : std_logic_vector(1 downto 0);

signal game_code : std_logic_vector(1 downto 0);
```

```
----------------------------------------------------------------------------
-- STATUS SIGNALS
----------------------------------------------------------------------------

signal Z : std_logic := '0';
signal N : std_logic := '0';
signal both_Z : std_logic := '0';

signal active_player : std_logic_vector(1 downto 0);
-- Are we executing code as playing?
signal game_started : std_logic := '0';
-- Sould we load?
signal load : std_logic := '0';
-- Did the play stop? (Will not be game over after reset)
signal game_over : std_logic := '0';
signal player_victory : std_logic_vector(1 downto 0);

signal new_IN : std_logic := '0';
signal reset : std_logic := '0';

----------------------------------------------------------------------------
-- FBART SIGNALS
----------------------------------------------------------------------------
signal fbart_request_next_data :  STD_LOGIC := '0';         -- Generate this when
    we read from FBART into BUSS
signal fbart_control_signals :   STD_LOGIC_VECTOR (2 downto 0);
signal rxd : std_logic := '1';

----------------------------------------------------------------------------
-- FIFO SIGNALS
----------------------------------------------------------------------------
signal fifo_out : std_logic_vector(12 downto 0) := "0010XXXXXXXXX";

signal fifo_current_player : STD_LOGIC;
signal fifo_game_over :std_logic;

signal fifo_next_pc : std_logic := '0';
signal fifo_write_pc : std_logic := '0';
signal fifo_change_player : std_logic := '0';

----------------------------------------------------------------------------
-- VGA SIGNALS
----------------------------------------------------------------------------

signal data_gpu : std_logic_vector (7 downto 0);
signal data_gpu_out : std_logic_vector (7 downto 0);
signal address_gpu : std_logic_vector (12 downto 0);

signal border_color : std_logic_vector(7 downto 0) := "00100111";

----------------------------------------------------------------------------
-- HEX DISPLAY
----------------------------------------------------------------------------

signal current_instr : std_logic_vector(3 downto 0);
```

```
begin

    ------------------------------------------------------------------------
    -- TEST SIGNALS
    ------------------------------------------------------------------------

    active_player_out <= active_player;
    reset_out <= reset;
    game_started_out <= game_started;
    game_over_out <= game_over;
    alu1_o <= ALU1_out(12 downto 5);

    ------------------------------------------------------------------------
    -- TEMP AND TESTING
    ------------------------------------------------------------------------

    --new_IN <= tmp_has_next_data;
    --IN_reg <= tmp_IN;
    --tmp_request_next_data <= fbart_request_next_data;

    ------------------------------------------------------------------------
    -- COMPONENT INITIATION
    ------------------------------------------------------------------------

    micro: Microcontroller
        port map (  clk => clk,
                    reset_a => reset_a,
                    buss_in => main_buss(7 downto 0),
                    uCount_limit => uCount_limit,

                    PC_code => PC_code,
                    buss_code => buss_code,

                    ALU_code => ALU_code,
                    ALU1_code => ALU1_code,
                    ALU2_code => ALU2_code,

                    memory_addr_code => memory_addr_code,

                    memory1_write => memory1_write,
                    memory2_write => memory2_write,
                    memory3_write => memory3_write,

                    memory1_read => memory1_read,
                    memory2_read => memory2_read,
                    memory3_read => memory3_read,

                    OP_code => OP_code,
                    M1_code => M1_code,
                    M2_code => M2_code,

                    ADR1_code => ADR1_code,
                    ADR2_code => ADR2_code,

                    FIFO_code => FIFO_code,
```

```
                game_code => game_code,

                Z => Z,
                N => N,
                both_Z => both_Z,
                new_IN => new_IN,
                game_started => game_started,
                shall_load => load,
                game_over => game_over,

                current_instr => current_instr
    );

alus: ALU
    port map (  clk => clk,
                alu_operation => ALU_operation,

                alu1_zeroFlag_out => Z,
                alu1_negFlag => N,
                alu_zeroFlag => both_Z,

                alu1_operand => alu1_operand,
                alu2_operand => alu2_operand,
                alu1_out => ALU1_out,
                alu2_out => ALU2_out
    );

memory1: Memory_Cell_DualPort
    port map (  clk => clk,
                read => memory1_read,
                address_in => memory_address_in,
                address_out => memory_address,
                write => memory1_write,
                data_in => memory1_data_in,
                data_out => OP,
                data_gpu => data_gpu,
                read_gpu => '1',
                address_gpu => address_gpu,
                active_player => active_player
    );

memory2: Memory_Cell
    port map (  clk => clk,
                read => memory2_read,
                address_in => memory_address_in,
                --address_out => memory2_address_out,
                write => memory2_write,
                data_in => memory2_data_in,
                data_out => M1,
                reset => reset
    );

memory3: Memory_Cell
    port map (  clk => clk,
                read => memory3_read,
```

```
                address_in => memory_address_in,
                --address_out => memory3_address_out,
                write => memory3_write,
                data_in => memory3_data_in,
                data_out => M2,
                reset => reset
    );

fbart: FBARTController
    port map (  clk => clk,
                request_next_data  => fbart_request_next_data,
                reset => reset,
                control_signals => fbart_control_signals,

                -- Commented when testing
                buss_out => IN_reg,
                has_next_data => new_IN,

                rxd => rxd,
                padding_error_out => pad_error
    );

fifo: PlayerFIFO
    port map (  current_pc_in => main_buss,         -- Always connected to
        main_buss
                current_pc_out => fifo_out,
                current_player_out => fifo_current_player,
                game_over_out => fifo_game_over,
                next_pc => fifo_next_pc,
                write_pc => fifo_write_pc,
                change_player => fifo_change_player,
                clk => clk,
                reset => reset
    );

 GPU: vga
        port map (  clk => clk,
                    rst => '0',
                    data_gpu => data_gpu_out,
                    address_gpu => address_gpu,
                    border_color => border_color,
                    red => red,
                    grn => grn,
                    blu => blu,
                    HS => HS,
                    VS => VS
     );

 hexdisplay : MARCled
    port map (
            clk => clk,
            rst => reset,
            ca => ca,
            cb => cb,
            cc => cc,
            cd => cd,
```

```
            ce => ce,
            cf => cf,
            cg => cg,
            dp => dp,
            an => an,
            game_started => game_started,
            current_instr => current_instr,
            game_over => game_over,
            active_player => active_player
);

-------------------------------------------------------------------------
-- CLOCK EVENT
-------------------------------------------------------------------------

process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            rxd <= '1';
        else
            rxd <= fbart_in;
        end if;

    if reset = '1' then
        border_color <= "00100111";
    elsif player_victory = "01" then
        border_color <= "00000111";
    elsif player_victory = "10" then
        border_color <= "00111000";
    end if;

    -- Set victory status
    if reset = '1' then
        player_victory <= "00";
    elsif game_over = '1' and active_player = "01" then
        player_victory <= "10";
    elsif game_over = '1' and active_player = "10" then
        player_victory <= "01";
    end if;

        -- Set load status
        if reset = '1' then
            load <= '0';
        elsif game_code = "11" then
            load <= '1';
        end if;

        -- Game started, will only set at certain times
        if reset = '1' or game_over = '1' then
            game_started <= '0';
        elsif game_code = "01" then
            game_started <= '1';
        end if;

        -- Game over, will only check at certain times
```

```
        if reset = '1' then
            game_over <= '0';
        elsif game_code = "10" then
            game_over <= fifo_game_over;
        end if;

        -- Sync reset
        if reset_a = '1' then
            reset <= '1';
        elsif reset = '1' then
            reset <= '0';
        end if;

        -------------------------------------------------------------------------
        -- REGISTRY MULTIPLEXERS
        -------------------------------------------------------------------------

        if reset_a = '1' then
            PC <= "0000000000000";
        elsif PC_code = "01" then
            PC <= main_buss;
        elsif PC_code = "10" then
            PC <= PC + 1;
        elsif PC_code = "11" then
            PC <= (others => '0');
        end if;

    if reset_a = '1' then
        ADR1 <= (others => '0');
    elsif ADR1_code = "01" then
        ADR1 <= main_buss;
    elsif ADR1_code = "10" then
        ADR1 <= M1;
    elsif ADR1_code = "11" then
        ADR1 <= ALU1_out;
    else
        ADR1 <= ADR1;
    end if;

    if reset_a = '1' then
        ADR2 <= (others => '0');
    elsif ADR2_code = "01" then
        ADR2 <= main_buss;
    elsif ADR2_code = "10" then
        ADR2 <= M2;
    elsif ADR2_code = "11" then
        ADR2 <= ALU2_out;
    else
        ADR2 <= ADR2;
    end if;

    end if;
end process;

-------------------------------------------------------------------------
-- MEMORY MULTIPLEXERS
```

```
--------------------------------------------------------------------------

with memory_addr_code select
    memory_address_in <= main_buss when "001",
                         ALU1_out when "010",
                         ALU2_out when "011",
                         ADR1 when "100",
                         ADR2 when "101",
                         PC when "110",
                         memory_address when others;

with OP_code select
    memory1_data_in <= main_buss(7 downto 0) when '1',
                       OP when others;

with M1_code select
    memory2_data_in <= main_buss when "01",
                       ALU1_out when "10",
                       ALU2_out when "11",
                       M1 when others;

with M2_code select
    memory3_data_in <= main_buss when "01",
                       ALU1_out when "10",
                       ALU2_out when "11",
                       M2 when others;

--------------------------------------------------------------------------
-- ALU MULTIPLEXERS
--------------------------------------------------------------------------

-- This works like follows:
-- alu operand is the in data to the alu

-- ALU_code is the control signal which says what the ALU should do

-- ALU_code has these commands:
-- 000     nothing
-- 001     load
-- 010     add
-- 011     sub
-- 100     +1
-- 101     -1
-- 110      = 0

-- ALUx_code states the source
-- 00      M1
-- 01      buss
-- 10      M2
-- 11      mem_addr

alu1_operand <= "0000000000000" when ALU_code = "110" else -- = 0
                "0000000000001" when ALU_code = "100" or ALU_code = "101" else --
                    +1 or -1
                M1 when ALU1_code = "00" else
                M2 when ALU1_code = "10" else
```

```
                memory_address when ALU1_code = "11" else
                main_buss;

alu2_operand <= "0000000000000" when ALU_code = "110" else -- = 0
                "0000000000001" when ALU_code = "100" or ALU_code = "101" else --
                    +1 or -1
                M1 when ALU2_code = '0' else
                M2;

-- 00 hold
-- 01 load main buss
-- 10 +
-- 11 -
ALU_operation <= "01" when ALU_code = "001" else -- load
                 "10" when ALU_code = "010" else -- +
                 "11" when ALU_code = "011" else -- -
                 "10" when ALU_code = "100" else -- +1
                 "11" when ALU_code = "101" else -- -1
                 "01" when ALU_code = "110" else
                 "00"; -- hold

------------------------------------------------------------------------
-- FIFO Handling
------------------------------------------------------------------------

fifo_write_pc <= '1' when FIFO_code = "01" else
                 '0';

fifo_next_pc <= '1' when FIFO_code = "11" else
                '0';

fifo_change_player <= '1' when FIFO_code = "10" else
                      '0';

active_player <= "00" when load = '0' else
                 "01" when fifo_current_player = '0' else
                 "10";

------------------------------------------------------------------------
-- FBARt Handling
------------------------------------------------------------------------

fbart_request_next_data <= '1' when buss_code = "110" else '0';

------------------------------------------------------------------------
-- BUSS MEGA-MULTIPLEXER
------------------------------------------------------------------------

with buss_code select
    main_buss <= PC when "000",
                "00000" & OP when "001",
                M1 when "010",
                M2 when "011",
                ALU1_out when "100",
                fifo_out when "101",
                IN_reg when "110",
```

```
                "0000000000000" when others;
    ----------------------------------------------------------------------------
    -- COLOR HANDLING
    ----------------------------------------------------------------------------

    data_gpu_out <= "11011111" when PC = address_gpu and active_player = "01" else
                "11111011" when PC = address_gpu and active_player = "10" else
                "01110111" when active_player = "01" and (address_gpu = ADR1 or
                    address_gpu = ADR2) else
                "01111110" when active_player = "10" and (address_gpu = ADR1 or
                    address_gpu = ADR2) else
                data_gpu;

    player_victory_out <= player_victory;


end Behavioral;
```

B.2 microcontroller.vhd

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

entity Microcontroller is
    Port(
        -- Our clock
        clk : in std_logic;

        -- Asynced reset
        reset_a : in std_logic;

        -- Buss input
        buss_in : in std_logic_vector(7 downto 0);

        uCount_limit : in std_logic_vector(7 downto 0);

        -- Control codes
        PC_code : out std_logic_vector(1 downto 0);

        buss_code : out std_logic_vector(2 downto 0);

        ALU_code : out std_logic_vector(2 downto 0);
        ALU1_code : out std_logic_vector(1 downto 0);
        ALU2_code : out std_logic;

        memory_addr_code : out std_logic_vector(2 downto 0);

        memory1_write : out std_logic;
        memory2_write : out std_logic;
        memory3_write : out std_logic;

        memory1_read : out std_logic;
        memory2_read : out std_logic;
        memory3_read : out std_logic;

        OP_code : out std_logic;
        M1_code : out std_logic_vector(1 downto 0);
        M2_code : out std_logic_vector(1 downto 0);

        ADR1_code : out std_logic_vector(1 downto 0);
        ADR2_code : out std_logic_vector(1 downto 0);

        FIFO_code : out std_logic_vector(1 downto 0);

        game_code : out std_logic_vector(1 downto 0);

        -- Status signals
        Z : in std_logic;
        N : in std_logic;
        both_Z : in std_logic;
```

```
        new_IN : in std_logic;
        game_started : in std_logic;
        shall_load : in std_logic;
        game_over : in std_logic;

        current_instr : out std_logic_vector(3 downto 0)
    );
end Microcontroller;

architecture Behavioral of Microcontroller is

    -- Microcode lives here
    subtype DataLine is std_logic_vector(46 downto 0);
    type Data is array (0 to 255) of DataLine;

    -- game FIFO IR ADR1 ADR2 OP M1 M2 mem1 mem2 mem3 mem_addr ALU1 ALU2 ALU buss PC
        uPC  uPC_addr
    --  00   00  0   00  00   0  00 00  00   00   00    000     00   0   000 000  00
        00000 00000000
    signal mem : Data := (
        -- Startup, check if we're in game
        "00000000000000000000000000000000000011000110011", -- jmpS GAME(33)
        "00000000000000000000000000000000000101000000001", -- jmpO +0(01)

        -- Clear memory contents
        "00000000000000000000000000110000000000000000000", -- ALU = 0
        "00000000010101000000000000000100010000000000000", -- ALU1 -> buss, buss -> OP
            , buss -> M1, buss -> M2, buss -> PC
        "00000000000000000000110000000000000000000000000", -- PC -> mem_addr
        "00000000000000101010000000000000000000000000000", -- OP -> mem, M1 -> mem, M2
             -> mem
        "00000000000000000000000000100000000000000000000", -- ALU++
        "00000000000000000000000000000100010001100001001", -- ALU1 -> buss, buss -> PC
            , jmpZ LOADP(09)
        "00000000000000000000000000000000000001000000100", -- jmp CLRMEM(04)
        --"00000000000000000000000000000000000000000000000", -- nothing, skip clearing
             memory

        "11000000000000000000000000000000000001000001011", -- shall_load, jmp POLL(0b)
        "00000000000000000000000000000000000001000001001", -- jmp -1(09)

        -- Load in program to memory
        "00000000000000000000000000000110000000000000000", -- IN -> buss

        -- Load program 1
        "00110000000000000000000000110000000000000000000", -- ALU = 0, fifo_next
        "00010000000000000000000000000100010000000000000", -- ALU1 -> buss, buss ->
            FIFO, buss -> PC

        "00000000000000000000000000000000000010000010000", -- jmpIN F1NUM(10)
        "00000000000000000000000000000000000001000001110", -- jmp -1(0e)
        "00000000000000000000000010001110000000000000000", -- IN -> buss, buss -> ALU1
        "00000000000000000000000000000000000010000010011", -- jmpIN F1OP(13)
        "00000000000000000000000000000000000001000010001", -- jmp -1(11)
        "00000000010000000000000000000110000000000000000", -- IN -> buss, buss -> OP
        "00000000000000000000000000000000000010000010110", -- jmpIN F1M1(16)
```

```
"00000000000000000000000000000000000001000010100", -- jmp -1(14)
"00000000000100000000000000000110000000000000000", -- IN -> buss, buss -> M1
"00000000000000000000000000000000000010000011001", -- jmpIN F1M2(19)
"00000000000000000000000000000000000001000010111", -- jmp -1(17)
"00000000000001000000110000000110000000000000000", -- IN -> buss, buss -> M2,
    PC -> mem_addr
"00000000000000101010000000000000000000000000000", -- OP -> mem, M1 -> mem, M2
     -> mem
"00000000000000000000000000101000100000000000000", -- ALU--, PC++
"00000000000000000000000000000000000001100011110", -- jmpZ LOAD2(1e)
"00000000000000000000000000000000000001000010001", -- jmp F1ROW(11)

-- Load program 2
"00100000000000000000000000000000000000000000000", -- change_player
"00000000000000000000000000000000000010000100001", -- jmpIN F2PC(21)
"00000000000000000000000000000000000001000011111", -- jmp -1(1f)
"00010000000000000000000000000110010000000000000", -- IN -> buss, buss -> FIFO
    , buss -> PC
"00000000000000000000000000000000000010000100100", -- jmpIN F2NUM(24)
"00000000000000000000000000000000000001000100010", -- jmp -1(22)
"00000000000000000000000010001110000000000000000", -- IN -> buss, buss -> ALU1
"00000000000000000000000000000000000010000100111", -- jmpIN F2OP(27)
"00000000000000000000000000000000000001000100101", -- jmp -1(25)
"00000000010000000000000000000110000000000000000", -- IN -> buss, buss -> OP
"00000000000000000000000000000000000010000101010", -- jmpIN F2M1(2a)
"00000000000000000000000000000000000001000101000", -- jmp -1(28)
"00000000000100000000000000000110000000000000000", -- IN -> buss, buss -> M1
"00000000000000000000000000000000000010000101101", -- jmpIN F2M2(2d)
"00000000000000000000000000000000000001000101011", -- jmp -1(2b)
"00000000000001000000110000000110000000000000000", -- IN -> buss, buss -> M2,
    PC -> mem_addr
"00000000000000101010000000000000000000000000000", -- OP -> mem, M1 -> mem, M2
     -> mem
"00000000000000000000000000101000100000000000000", -- ALU--, PC++
"00000000000000000000000000000000000001100110010", -- jmpZ LEND(32)
"00000000000000000000000000000000000001000100101", -- jmp F2ROW(25)
"01000000000000000000000000000000000001100000000", -- game_started, jmpZ 0(00)


-- Game sequence
"00100000000000000000000000000000000000000000000", -- change_player
"00110000000000000000000000000000000000000000000", -- fifo_next
"00000000000000000000000000001010100000000000000", -- FIFO -> buss, buss -> PC
"00000000000000000000110000000000000000000000000", -- PC -> mem_addr
"00000000000000010101000000000000000000000000000", -- mem -> OP, mem -> M1,
    mem -> M2
"00010000000000000000000000000001000001000111001", -- OP -> buss, buss -> IR,
    jmp AMOD(39)

-- Calculate adress mode for A operand
"00000000000000000000000000000000001000001001010", -- jmpAimm BMOD(4a)
"00000000000000000000000000001000000000000000000", -- M1 -> ALU1
"00000000000000000000000010010000000000000000000", -- ALU1 += PC
"00000000001000000000000000000000000000000000000", -- ALU1 -> M1
"00000000000000000000000000000000001000101001010", -- jmpAdir BMOD(4a)
"00000000000000000000001000000010000000000000000", -- M1 -> buss, buss ->
```

```
    mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000100000000000000000011001001001000101", -- M2 -> buss, buss -> M1,
    jmpApre APRE(45)
"00000000000000000000000000001000000000000000000", -- M1 -> ALU1
"00000000000000000000000110010000000000000000000", -- ALU1 += mem_addr
"00000000001000000000000000000000000000000000000", -- ALU1 -> M1
"00000000000000000000000000000000000001001001010", -- jmp BMOD(4a)

"00000000000000000000000000001000000000000000000", -- M1 -> ALU1
"00000000000000000000000000101000000000000000000", -- ALU--
"00000000001010000000000000000000000000000000000", -- ALU1 -> M1, ALU1 -> M2
"00000000000000000010000000000000000000000000000", -- M2 -> mem
"00000000000000000000000000000000000001001000001", -- jmp AOFF(41)

-- Calculate adress mode for B operand
"00000000000000000000110000000000000000000000000", -- PC -> mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000000000000001001101011101", -- jmpBimm INSTR(5d)
"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000000010010000000000000000000", -- ALU1 += PC
"00000000000010000000000000000000000000000000000", -- ALU1 -> M2
"00000000000000000000000000000000001010001011101", -- jmpBdir INSTR(5d)
"00000000000000000000001000000011000000000000000", -- M2 -> buss, buss ->
    mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000000000000001010101011000", -- jmpBpre BPRE(58)
"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000000110010000000000000000000", -- ALU1 += mem_addr
"00000000000010000000000000000000000000000000000", -- ALU1 -> M2
"00000000000000000000000000000000000001001011101", -- jmp INSTR(5d)

"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000000000101000000000000000000", -- ALU--
"00000000000010000000000000000000000000000000000", -- ALU1 -> M2
"00000000000000000010000000000000000000000000000", -- M2 -> mem
"00000000000000000000000000000000000001001010100", -- jmp BOFF(54)


-- 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

"00000101000000000000000000000000000000100000000", -- 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
```

```
"00000000000000000000000000000000000001011000001", -- jmp END(c1)

-- MOV  Move A to B
"00000000000000000000000000000000001000001100101", -- jmpAimm IMOV(65)
"00000000000000000000100000000000000000000000000", -- ADR1 -> mem_addr
"00000000000000010101000000000000000000000000000", -- mem -> OP, mem -> M1, mem ->
    M2
"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000101010000000000000000000000000000", -- OP -> mem, M1 -> mem, M2 ->
    mem
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

-- If A immediate, move A to B op specified by B mem address
"00000000000001000000101000000010000000000000000", -- ADR2 -> mem_addr, M1 -> buss,
    buss -> M2
"00000000000000000010000000000000000000000000000", -- M2 -> mem
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

-- ADD  Add A to B
"00000000000000000000000000000000001000001110001", -- jmpAimm IADD(71)
"00000000000000000000100000000000000000000000000", -- ADR1 -> mem_addr
"00000000000000000101000000000000000000000000000", -- mem -> M1, mem -> M2
"00000000000000000000101001001000000000000000000", -- ADR2 -> mem_addr, M1 -> ALU1,
    M2 -> ALU2
"00000000000000000101000000000000000000000000000", -- mem -> M1, mem -> M2
"00000000000000000000000001010000000000000000000", -- ALU1 += M1, ALU2 += M2
"00000000001011000000000000000000000000000000000", -- ALU1 -> M1, ALU2 -> M2
"00000000000000001010000000000000000000000000000", -- M1 -> mem, M2 -> mem
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000001000000001000000000000000000", -- M1 -> ALU1, mem -> M2
"00000000000000000000000100010000000000000000000", -- ALU1 += M2
"00000000000010000000000000000000000000000000000", -- ALU1 -> M2
"00000000000000000010000000000000000000000000000", -- M2 -> mem
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

-- SUB  Sub A from B
"00000000000000000000000000000000001000010000000", -- jmpAimm ISUB(80)
"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000101000000000000000000000000000", -- mem -> M1, mem -> M2
"00000000000000000000100001001000000000000000000", -- ADR1 -> mem_addr, M1 -> ALU1,
    M2 -> ALU2
"00000000000000000101000000000000000000000000000", -- mem -> M1, mem -> M2
"00000000000000000000000001011000000000000000000", -- ALU1 -= M1, ALU2 -= M2
"00000000001011000000101000000000000000000000000", -- ALU1 -> M1, ALU2 -> M2, ADR2
    -> mem_addr
"00000000000000001010000000000000000000000000000", -- M1 -> mem, M2 -> mem
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000000000011000000000000000000", -- ALU1 -= M1
"00000000000010000000000000000000000000000000000", -- ALU1 -> M2
"00000000000000000010000000000000000000000000000", -- M2 -> mem
```

```
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

-- JMP  Jump to A
"000100000000000000000000000000010000001011000001", -- M1 -> buss, buss -> FIFO, jmp
    END(c1)

-- JMPZ Jump to A if B zero
"00000000000000000000000000000000001001110001011", -- jmpBimm IJMPZ(8b)
"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000000000000000000001110001110", -- jmpZ DOJMPZ(8e)
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)
"00000000000000000000000000000000000001010000111", -- jmp JMP(87)

-- JMPN Jump to A if B non-zero
"00000000000000000000000000000000001001110010010", -- jmpBimm IJMPN(92)
"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000000000000000000001110111111", -- jmpZ ADDPC(bf)
"00000000000000000000000000000000000001010000111", -- jmp JMP(87)

-- CMP If A eq B skip next instr
"00000000000000000000000000000000001000010100010", -- jmpAimm ICMP(a2)
"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000010101000000000000000000000000000", -- mem -> OP, mem -> M1, mem ->
    M2
"00000000000000000000100001001000000000000000000", -- ADR1 -> mem_addr, M1 -> ALU1,
    M2 -> ALU2
"00000000000000000101000000000000000000000000000", -- mem -> M1, mem -> M2
"00000000000000000000000000000000000000000000000", -- ALU1 -= M1, ALU2 -= M2
"00000000000000000000000000000000000100010011101", -- jmpE CMPOP(9d)
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

"00000000000000000000000010001001000000000000000", -- OP -> buss, buss -> ALU1
"00000000000000010000000000000000000000000000000", -- mem -> OP
"00000000000000000000000010011001000000000000000", -- ALU1 -= OP
"00000000000000000000000000000000000001110111110", -- jmpZ SKIP(be)
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000001000000001000000000000000000", -- mem -> M2, M1 -> ALU1
"00000000000000000000000100011000000000000000000", -- ALU1 -= M2
"00000000000000000000000000000000000001110111110", -- jmpZ SKIP(be)
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

-- SLT if A is less than B skip next instr
"00000000000000000000000000000000001000010110000", -- jmpAimm ISLT(b0)
"00000000000000000000100000000000000000000000000", -- ADR1 -> mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000100011000000000000000000", -- ALU1 -= M2
"00000000000000000000000000000000000011110111110", -- jmpN SKIP(be)
```

```
"00000000000000000000000000000000000001010111111", -- jmp ADDPC(bf)

"00000000000000000000000000001000000001010101011", -- M1 -> ALU1, jmp SLTCMP(ab)

-- DJN Decr B, if not zero jmp to A
"00000000000000000000000000000000001001110111010", -- jmpBimm IDJN(ba)
"00000000000000000000101000000000000000000000000", -- ADR2 -> mem_addr
"00000000000000000001000000000000000000000000000", -- mem -> M2
"00000000000000000000000100001000000000000000000", -- M2 -> ALU1
"00000000000000000000000000101000000000000000000", -- ALU--
"00000000000010000000000000000000000000000000000", -- ALU1 -> M2
"00000000000000000010000000000000000000000000000", -- M2 -> mem
"00000000000000000000000000000000000001110111111", -- jmpZ ADDPC(bf)
"00010000000000000000000000000010000001011000001", -- M1 -> buss, buss -> FIFO, jmp
    END(c1)

"00000000000000000000110100001000000001010110100", -- M2 -> ALU1, PC -> mem_addr,
    jmp DODJN(b4)


-- SPL Place A in process queue
"00000000000000000000000000000000100000000000000", -- PC++
"00010000000000000000000000000000000000000000000", -- PC -> buss, buss -> FIFO
"00010000000000000000000000000010000001011000001", -- M1 -> buss, buss -> FIFO, jmp
    END(c1)


"00000000000000000000000000000000100001010111111", -- PC++, jmp ADDPC(bf)

-- Keep the PC for next round
"00000000000000000000000000000000100000000000000", -- PC++
"00010000000000000000000000000000000000000000000", -- PC -> buss, buss -> FIFO

"10000000000000000000000000000000000000000000000", -- check_gameover
"00000000000000000000000000000000000010100000000", -- jmpC 0(00)
--"00000000000000000000000000000000001111100000000", -- uPC = 0
"00000000000000000000000000000000000001011000010", -- jmp DELAY(c2)

      others => (others => '0')
  );

  -- Synced reset
  signal reset : std_logic;

  signal uCount_limit_sync : std_logic_vector(7 downto 0);

  -- Current microcode line to process
  signal signals : DataLine;

  -- Controll the behavior of next uPC value
  signal uPC_addr : std_logic_vector(7 downto 0);
  signal uPC_code : std_logic_vector(4 downto 0);

  signal IR_code : std_logic;

  signal uCounter : std_logic_vector(26 downto 0);
```

```
    -- Registers
    signal IR : std_logic_vector(7 downto 0);
    signal uPC : std_logic_vector(7 downto 0);

    -- Split up IR
    alias OP_field is IR(7 downto 4);
    alias A_field is IR(3 downto 2);
    alias B_field is IR(1 downto 0);

    -- Instruction code decodings
    signal op_addr : std_logic_vector(7 downto 0);
    signal A_imm : std_logic;
    signal A_dir : std_logic;
    signal A_pre : std_logic;
    signal B_imm : std_logic;
    signal B_dir : std_logic;
    signal B_pre : std_logic;
begin

    current_instr <= OP_field;

    ------------------------------------------------------------------------
    -- RETRIEVE SIGNALS
    ------------------------------------------------------------------------

    signals <= mem(conv_integer(uPC));

    uPC_addr <= signals(7 downto 0);
    uPC_code <= signals(12 downto 8);

    PC_code <= signals(14 downto 13);
    buss_code <= signals(17 downto 15);

    ALU_code <= signals(20 downto 18);
    ALU2_code <= signals(21);
    ALU1_code <= signals(23 downto 22);

    memory_addr_code <= signals(26 downto 24);

    memory3_read <= signals(27);
    memory3_write <= signals(28);

    memory2_read <= signals(29);
    memory2_write <= signals(30);

    memory1_read <= signals(31);
    memory1_write <= signals(32);

    M2_code <= signals(34 downto 33);
    M1_code <= signals(36 downto 35);
    OP_code <= signals(37);

    ADR2_code <= signals(39 downto 38);
    ADR1_code <= signals(41 downto 40);
```

```vhdl
IR_code <= signals(42);
FIFO_code <= signals(44 downto 43);

game_code <= signals(46 downto 45);

------------------------------------------------------------------------
-- ENCODINGS
------------------------------------------------------------------------

-- OP code address decoding
with OP_field select
    op_addr <=      "01011110" when "0000", -- DAT 5e
  "01011111" when "0001", -- MOV 5f
  "01101000" when "0010", -- ADD 68
  "01110111" when "0011", -- SUB 77
  "10000111" when "0100", -- JMP 87
  "10001000" when "0101", -- JMPZ 88
  "10001111" when "0110", -- JMPN 8f
  "10010101" when "0111", -- CMP 95
  "10100111" when "1000", -- SLT a7
  "10110001" when "1001", -- DJN b1
  "10111011" when "1010", -- SPL bb
  "11111111" when others;


A_dir <= '1' when A_field = "00" else '0';
A_imm <= '1' when A_field = "01" else '0';
A_pre <= '1' when A_field = "11" else '0';

B_dir <= '1' when B_field = "00" else '0';
B_imm <= '1' when B_field = "01" else '0';
B_pre <= '1' when B_field = "11" else '0';

------------------------------------------------------------------------
-- ON CLOCK EVENT
------------------------------------------------------------------------

process (clk)
begin
    if rising_edge(clk) then

        if reset_a = '1' then
            reset <= '1';
        elsif reset = '1' then
            reset <= '0';
        end if;

    uCount_limit_sync <= uCount_limit;

        ------------------------------------------------------------------------
        -- SIGNAL MULTIPLEXERS
        ------------------------------------------------------------------------

        -- Update uPC
        if reset_a = '1' then
            uPC <= "00000000";
```

```
        elsif uPC_code = "00001" then
            uPC <= op_addr;
        elsif uPC_code = "00010" then
            uPC <= uPC_addr;
        elsif uPC_code = "00011" and Z = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "00100" and new_IN = '1' then
            uPC <= uPC_addr;
        --elsif uPC_code = "00101" and '0' & uCounter >= '0' & uCount_limit_sync &
             "0000011000000000000" then
        elsif uPC_code = "00101" and '0' & uCounter >= '0'
& uCount_limit_sync(7 downto 6) & "00"
& uCount_limit_sync(5 downto 4) & "00"
& uCount_limit_sync(3 downto 2) & "00"
& uCount_limit_sync(1 downto 0) & "00"
& "00000000000" then

            uPC <= uPC_addr;
        elsif uPC_code = "00110" and game_started = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "00111" and N = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "01000" and both_Z = '1' then
            uPC <= uPC_addr;
        -- elsif uPC_code = "01001" and  = '1' then
        elsif uPC_code = "01001" then -- Deprecated
            uPC <= uPC_addr;
        elsif uPC_code = "01010" and game_over = '1' then
            uPC <= uPC_addr;

        elsif uPC_code = "10000" and A_imm = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "10001" and A_dir = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "10010" and A_pre = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "10011" and B_imm = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "10100" and B_dir = '1' then
            uPC <= uPC_addr;
        elsif uPC_code = "10101" and B_pre = '1' then
            uPC <= uPC_addr;

        elsif uPC_code = "11111" then
            uPC <= "00000000";
        else
            uPC <= uPC + 1;
        end if;

        -- Update uCounter
        if reset_a = '1' then
            uCounter <= (others => '0');
        elsif uPC = "00000000" then
            uCounter <= (others => '0');
        else
            uCounter <= uCounter + 1;
```

```
            end if;

            if reset = '1' then
                IR <= "00000000";
            elsif IR_code = '1' then
                IR <= buss_in;
            end if;

        end if;
    end process;

end Behavioral;
```

B.3 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.4 MemoryCell.vhd

```
-------------------------------------------------------------------------------------
-- Course:      TSEA43
-- Student:     Jesper Tingvall
-- Design Name: MARC
-- Module Name: Memory Cell
-- Description: This will behave like a 8192 x 13 bit memory.
-------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity Memory_Cell is
    Port (  clk         : in std_logic;
            reset       : in std_logic;  -- This does nothing at the moment, would be
                nice if one could reset the whole memory with this signal but I don't
                know how to tell VHDL that.
            read        : in STD_LOGIC;
            write       : in  STD_LOGIC;
            address_in  : in  STD_LOGIC_VECTOR (12 downto 0);
            address_out : out  STD_LOGIC_VECTOR (12 downto 0);  -- Connect address_in
                to address_out via a multiplexer
            data_in     : in  STD_LOGIC_VECTOR (12 downto 0);
            data_out    : out  STD_LOGIC_VECTOR (12 downto 0)); -- Connect data_in to
                data_out via a multiplexer
end Memory_Cell;

architecture Behavioral of memory_cell is
    signal address_sync     : STD_LOGIC_VECTOR (12 downto 0);
    signal data_sync        : STD_LOGIC_VECTOR (12 downto 0);

    type ram_block_type is array (0 to 1023) of std_logic_vector(12 downto 0);

    signal ram_block_0 : ram_block_type := (others => (others => '0'));
    signal ram_block_1 : ram_block_type := (others => (others => '0'));
    signal ram_block_2 : ram_block_type := (others => (others => '0'));
    signal ram_block_3 : ram_block_type := (others => (others => '0'));
    signal ram_block_4 : ram_block_type := (others => (others => '0'));
    signal ram_block_5 : ram_block_type := (others => (others => '0'));
    signal ram_block_6 : ram_block_type := (others => (others => '0'));
    signal ram_block_7 : ram_block_type := (others => (others => '0'));

begin

    address_out <= address_sync;
    data_out <= data_sync;

    -- Write or read data to the correct memory block.
    PROCESS(clk) begin
        if(rising_edge(clk)) then

            address_sync <= address_in;
            data_sync <= data_in;
```

```
if (address_sync(12 downto 10) = "000") then
     if(write='1') then
          ram_block_0(to_integer(unsigned( address_sync(9 downto 0) ))
              ) <= data_sync;
     end if;
     if (read='1') then
          data_sync <= ram_block_0(to_integer(unsigned(address_sync(9
              downto 0))));
     end if;

elsif (address_sync(12 downto 10) = "001") then
     if(write='1') then
          ram_block_1(to_integer(unsigned( address_sync(9 downto 0) ))
              ) <= data_sync;
     end if;
     if (read='1') then
          data_sync <= ram_block_1(to_integer(unsigned(address_sync(9
              downto 0))));
     end if;

elsif (address_sync(12 downto 10) = "010") then
     if(write='1') then
          ram_block_2(to_integer(unsigned( address_sync(9 downto 0) ))
              ) <= data_sync;
     end if;
     if (read='1') then
          data_sync <= ram_block_2(to_integer(unsigned(address_sync(9
              downto 0))));
     end if;

elsif (address_sync(12 downto 10) = "011") then
     if(write='1') then
          ram_block_3(to_integer(unsigned( address_sync(9 downto 0) ))
              ) <= data_sync;
     end if;
     if (read='1') then
          data_sync <= ram_block_3(to_integer(unsigned(address_sync(9
              downto 0))));
     end if;

elsif (address_sync(12 downto 10) = "100") then
     if(write='1') then
          ram_block_4(to_integer(unsigned( address_sync(9 downto 0) ))
              ) <= data_sync;
     end if;
     if (read='1') then
          data_sync <= ram_block_4(to_integer(unsigned(address_sync(9
              downto 0))));
     end if;

elsif (address_sync(12 downto 10) = "101") then
     if(write='1') then
          ram_block_5(to_integer(unsigned( address_sync(9 downto 0) ))
              ) <= data_sync;
     end if;
     if (read='1') then
```

```
                    data_sync <= ram_block_5(to_integer(unsigned(address_sync(9
                        downto 0))));
                end if;

            elsif (address_sync(12 downto 10) = "110") then
                if(write='1') then
                    ram_block_6(to_integer(unsigned( address_sync(9 downto 0) ))
                        ) <= data_sync;
                end if;
                if (read='1') then
                    data_sync <= ram_block_6(to_integer(unsigned(address_sync(9
                        downto 0))));
                end if;

            else
                if(write='1') then
                    ram_block_7(to_integer(unsigned( address_sync(9 downto 0) ))
                        ) <= data_sync;
                end if;
                if (read='1') then
                    data_sync <= ram_block_7(to_integer(unsigned(address_sync(9
                        downto 0))));
                end if;
            end if;
        end if;
    END PROCESS;

end Behavioral;
```

B.5 MemoryCellDualPort.vhd

```
----------------------------------------------------------------------------------
-- Course:      TSEA43
-- Student:     Jesper Tingvall
-- Design Name: MARC
-- Module Name: Memory Cell Dual Port
-- Description: This will behave like a 8192 x 8 bit memory for our OP + addressing
    modes. It will automagickally calculate some pretty colors for our code. Since this
     is a dual port memory can our GPU ask it whenever it wants what colors a address
    holds.
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity Memory_Cell_DualPort is
    Port ( clk             : in  STD_LOGIC;
           read            : in  STD_LOGIC;
           write           : in  STD_LOGIC;
           active_player   : in  STD_LOGIC_VECTOR(1 downto 0);  -- So we can color
               our code in pretty colors!
           address_in      : in  STD_LOGIC_VECTOR(12 downto 0);
           address_out     : out STD_LOGIC_VECTOR(12 downto 0);  -- Connect
               address_in to address_out via a multiplexer
           data_in         : in  STD_LOGIC_VECTOR(7 downto 0);
           data_out        : out STD_LOGIC_VECTOR(7 downto 0); -- Connect data_in to
               data_out via a multiplexer
           address_gpu     : in  STD_LOGIC_VECTOR(12 downto 0); -- This is not
               delayed
           data_gpu        : out STD_LOGIC_VECTOR(7 downto 0); -- This is not delayed
           read_gpu        : in  STD_LOGIC);
end Memory_Cell_DualPort;

architecture Behavioral of Memory_Cell_DualPort is
    signal address_sync         : STD_LOGIC_VECTOR (12 downto 0);
    signal data_sync            : STD_LOGIC_VECTOR (7 downto 0);

    signal calculated_color : STD_LOGIC_VECTOR (7 downto 0);

    type ram_block_type is array (0 to 1023) of std_logic_vector(15 downto 0);




    signal ram_block_0 : ram_block_type := (others => (others => '0'));
    signal ram_block_1 : ram_block_type := (others => (others => '0'));
    signal ram_block_2 : ram_block_type := (others => (others => '0'));
    signal ram_block_3 : ram_block_type := (others => (others => '0'));
    signal ram_block_4 : ram_block_type := (others => (others => '0'));
    signal ram_block_5 : ram_block_type := (others => (others => '0'));
    signal ram_block_6 : ram_block_type := (others => (others => '0'));
    signal ram_block_7 : ram_block_type := (others => (others => '0'));
```

```
begin

    address_out <= address_sync;
    data_out <= data_sync;

    -- Colorifier
    -- COLOR IS LIKE THIS BB GGG RRR
    -- This determines the color depending on what player put what there and what kind
         of OP code it is (data / non data)
    calculated_color <=     "00000000" when active_player = "00" else
                              -- DAT 0 0, black!

                            "00000010" when active_player = "01" and data_sync(7
                                downto 4) = "0000" else    -- Player 1 data
                            "00000101" when active_player = "01" and (data_sync(7
                                downto 4) = "0010" or data_sync(7 downto 4) = "0011")
                                else    -- Player 1 aritmetic
                            "00000111" when active_player = "01" and (data_sync(7
                                downto 4) = "0100" or data_sync(7 downto 4) = "0101" or
                                 data_sync(7 downto 4) = "0110" or data_sync(7 downto
                                4) = "1001")  else    -- Player 1 Jumps
                            --"00000110" when active_player = "01" and (data_sync(7
                                downto 4) = "0111" or data_sync(7 downto 4) = "1000")
                                else    -- Player 1 Compares

                            "00000110" when active_player = "01" else    -- Player 1
                                misc

                            "00010000" when active_player = "10" and data_sync(7
                                downto 4) = "0000" else    -- Player 2 data
                            "00101000" when active_player = "10" and (data_sync(7
                                downto 4) = "0010" or data_sync(7 downto 4) = "0011")
                                else    -- Player 2 aritmetic
                            "00111000" when active_player = "10" and (data_sync(7
                                downto 4) = "0100" or data_sync(7 downto 4) = "0101" or
                                 data_sync(7 downto 4) = "0110" or data_sync(7 downto
                                4) = "1001")  else    -- Player 2 Jumps
                            "00110000" when active_player = "10" else    -- Player 2
                                misc
                            "11111111";

    -- Use this if we want even prettier colors!
    --calculated_color <= active_player & data_sync(7 downto 4) & "00";

    PROCESS(clk) begin
        if(rising_edge(clk)) then

            address_sync <= address_in;
            data_sync <= data_in;

            -- Write and read OP

            if (address_sync(12 downto 10) = "000") then
                if(write='1') then
                    ram_block_0(to_integer(unsigned( address_sync(9 downto 0) ))) <=
                        data_sync & calculated_color; -- First OP then color!
```

```
    end if;
    if (read='1') then
        data_sync <= ram_block_0(to_integer(unsigned(address_sync(9 downto
             0))))(15 downto 8);
    end if;


elsif (address_sync(12 downto 10) = "001") then
    if(write='1') then
        ram_block_1(to_integer(unsigned( address_sync(9 downto 0) ))) <=
            data_sync & calculated_color;
    end if;
    if (read='1') then
        data_sync <= ram_block_1(to_integer(unsigned(address_sync(9 downto
             0))))(15 downto 8);
    end if;


elsif (address_sync(12 downto 10) = "010") then
    if(write='1') then
        ram_block_2(to_integer(unsigned( address_sync(9 downto 0) ))) <=
            data_sync & calculated_color;
    end if;
    if (read='1') then
        data_sync <= ram_block_2(to_integer(unsigned(address_sync(9 downto
             0))))(15 downto 8);
    end if;


elsif (address_sync(12 downto 10) = "011") then
    if(write='1') then
        ram_block_3(to_integer(unsigned( address_sync(9 downto 0) ))) <=
            data_sync & calculated_color;
    end if;
    if (read='1') then
        data_sync <= ram_block_3(to_integer(unsigned(address_sync(9 downto
             0))))(15 downto 8);
    end if;


elsif (address_sync(12 downto 10) = "100") then
    if(write='1') then
        ram_block_4(to_integer(unsigned( address_sync(9 downto 0) ))) <=
            data_sync & calculated_color;
    end if;
    if (read='1') then
        data_sync <= ram_block_4(to_integer(unsigned(address_sync(9 downto
             0))))(15 downto 8);
    end if;


elsif (address_sync(12 downto 10) = "101") then
    if(write='1') then
        ram_block_5(to_integer(unsigned( address_sync(9 downto 0) ))) <=
            data_sync & calculated_color;
    end if;
```

```
        if (read='1') then
            data_sync <= ram_block_5(to_integer(unsigned(address_sync(9 downto
                 0))))(15 downto 8);
        end if;


    elsif (address_sync(12 downto 10) = "110") then
        if(write='1') then
            ram_block_6(to_integer(unsigned( address_sync(9 downto 0) ))) <=
                data_sync & calculated_color;
        end if;
        if (read='1') then
            data_sync <= ram_block_6(to_integer(unsigned(address_sync(9 downto
                 0))))(15 downto 8);
        end if;


    else
        if(write='1') then
            ram_block_7(to_integer(unsigned( address_sync(9 downto 0) ))) <=
                data_sync & calculated_color;
        end if;
        if (read='1') then
            data_sync <= ram_block_7(to_integer(unsigned(address_sync(9 downto
                 0))))(15 downto 8);
        end if;

    end if;


    -- Read GPU
    if (address_gpu(12 downto 10) = "000") then
        if (read_gpu='1') then
            data_gpu <= ram_block_0(to_integer(unsigned(address_gpu(9 downto
                0))))(7 downto 0);
        end if;

    elsif (address_gpu(12 downto 10) = "001") then
        if (read_gpu='1') then
            data_gpu <= ram_block_1(to_integer(unsigned(address_gpu(9 downto
                0))))(7 downto 0);
        end if;

    elsif (address_gpu(12 downto 10) = "010") then
        if (read_gpu='1') then
            data_gpu <= ram_block_2(to_integer(unsigned(address_gpu(9 downto
                0))))(7 downto 0);
        end if;

    elsif (address_gpu(12 downto 10) = "011") then
        if (read_gpu='1') then
            data_gpu <= ram_block_3(to_integer(unsigned(address_gpu(9 downto
                0))))(7 downto 0);
        end if;

    elsif (address_gpu(12 downto 10) = "100") then
```

```
                if (read_gpu='1') then
                    data_gpu <= ram_block_4(to_integer(unsigned(address_gpu(9 downto
                        0))))(7 downto 0);
                end if;

            elsif (address_gpu(12 downto 10) = "101") then
                if (read_gpu='1') then
                    data_gpu <= ram_block_5(to_integer(unsigned(address_gpu(9 downto
                        0))))(7 downto 0);
                end if;

            elsif (address_gpu(12 downto 10) = "110") then
                if (read_gpu='1') then
                    data_gpu <= ram_block_6(to_integer(unsigned(address_gpu(9 downto
                        0))))(7 downto 0);
                end if;

            else
                if (read_gpu='1') then
                    data_gpu <= ram_block_7(to_integer(unsigned(address_gpu(9 downto
                        0))))(7 downto 0);
                end if;
            end if;

        end if;
    END PROCESS;

end Behavioral;
```

B.6 PlayerFIFO.vhd

```
----------------------------------------------------------------------------------
-- Course:      TSEA43
-- Student:     Jesper Tingvall
-- Design Name: MARC
-- Module Name: Player FIFO
-- Description: Two FIFOs we can store our player PC in.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity PlayerFIFO is
     Port ( current_pc_in : in  STD_LOGIC_VECTOR (12 downto 0);
            current_pc_out : out  STD_LOGIC_VECTOR (12 downto 0);
            current_player_out : out STD_LOGIC; -- Current player (0 is player 1, 1 is
                 player 2)
            game_over_out : out STD_LOGIC;  -- One player (or both) is dead
            next_pc : in  STD_LOGIC;       -- Removes the first PC in the current
                players FIFO and write it to current_pc_out.
            write_pc : in STD_LOGIC;       -- Add current_pc_in last to the current
                players FIFO, if it's full then this does nothing.
            change_player : in STD_LOGIC; -- Flips current player.
            clk : in std_logic;
            reset : in std_logic);
end PlayerFIFO;

architecture Behavioral of PlayerFIFO is
    signal p1_head : std_logic_vector(6 downto 0) := "0000000";
    signal p1_tail : std_logic_vector(6 downto 0) := "0000000";

    signal p2_head : std_logic_vector(6 downto 0) := "0000000";
    signal p2_tail : std_logic_vector(6 downto 0) := "0000000";

    type pc_block_type is array (0 to 127) of std_logic_vector(12 downto 0);  -- This
        will synthetisize to a dual port memory, we only need a single port memory but
        VHDL doesn't seem to understand that we only use one of the pointers (head and
        tail) and not both during a cp. This might be solved by some multiplexers and
        black magick.
    signal p1_block : pc_block_type := (others => (others => '0'));
    signal p2_block : pc_block_type := (others => (others => '0'));

    signal current_player : STD_LOGIC;

    -- Signals to current_pc_out MUX
    signal p1_head_pc : STD_LOGIC_VECTOR (12 downto 0);
    signal p2_head_pc : STD_LOGIC_VECTOR (12 downto 0);

    signal block_to_read : STD_LOGIC := '0';


begin
    current_player_out <= current_player;
```

```
current_pc_out <=   --"0000000000000" when game_over = '1' else
                    p1_head_pc when block_to_read = '0' else
                    p2_head_pc when block_to_read = '1';

game_over_out <= '1' when (p1_head = p1_tail) or (p2_head = p2_tail) else
                 '0';

process(clk)
begin
    if rising_edge(clk) then

        if (reset = '1') then
            current_player <= '0';
            p1_head <= "0000000";
            p1_tail <= "0000000";
            p2_head <= "0000000";
            p2_tail <= "0000000";
            block_to_read <= '0';

        else
            if next_pc = '1' then
                if (current_player = '0') then
                    if (p1_head /= p1_tail) then            -- Read player 1 PC
                        p1_head_pc <= p1_block(to_integer(unsigned(p1_head)));
                            -- Put out the top PC
                        p1_head <= p1_head + 1;
                            -- And increase out head
                        block_to_read <= '0';
                            -- Multplexer signals
                    end if;
                elsif (current_player = '1') then
                    if (p2_head /= p2_tail) then            -- Read player 2 PC
                        p2_head_pc <= p2_block(to_integer(unsigned(p2_head)));
                            -- Put out the top PC
                        p2_head <= p2_head + 1;
                            -- And increase out head
                        block_to_read <= '1';
                            -- Multplexer signals
                    end if;
                end if;

            elsif (write_pc = '1') then
                if (current_player = '1') then                  -- Current player
                    2
                    if (p2_tail + 1 /= p2_head) then            -- This can maybe
                        be done better without needing an extra aritmetic unit
                        p2_block(to_integer(unsigned(p2_tail))) <= current_pc_in;
                              -- Increase tail and write current_pc_in
                        p2_tail <= p2_tail + 1;
                    end if;

                elsif (current_player = '0') then               -- Current player
                    1
                    if (p1_tail + 1 /= p1_head) then
                        p1_block(to_integer(unsigned(p1_tail))) <= current_pc_in;
```

```
                                -- Increase tail and write current_pc_in
                            p1_tail <= p1_tail + 1;
                        end if;
                    end if;
                end if;

                -- Flip player
                if change_player = '1' then
                    current_player <= not current_player;
                end if;

            end if;

        end if;
    end process;
end Behavioral;
```

B.7 FBARTController.vhd

```
--------------------------------------------------------------------------------
-- Course:      TSEA43
-- Student:     Jesper Tingvall
-- Design Name: MARC
-- Module Name: FBART Controller
-- Description: Controller for a FBART, It takes two 8 bits transmissions and put them
     together to a single 13 bits (it throws away 3 bits) transmission.
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity FBARTController is
    Port (
           -- Starts next data transmission.
           request_next_data : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           clk : in  STD_LOGIC;

           -- Fbart serial data input
           rxd : in std_logic;

           -- The 3 thrown away bits, might be usefull if we want to send other things
                than data to MARC (special commands).
           control_signals : out  STD_LOGIC_VECTOR (2 downto 0);
           buss_out : out  STD_LOGIC_VECTOR (12 downto 0);

           -- We have 13 bits ready to be read!
           has_next_data : out  STD_LOGIC;

           -- Debug features, the 3 thrown away bits should be 000, otherwise
               something is wrong with out timing. These bits will be the error if it
               happends.
           padding_error_out : out STD_LOGIC_VECTOR(2 downto 0));
end FBARTController;

architecture Behavioral of FBARTController is
    component fbartrx
        port(   clk : in std_logic;      -- System clock input
                reset : in std_logic;    -- System reset input
                rxd : in std_logic;      -- Receiver data input
                rxrdy : out std_logic;   -- Receiver ready output
                rd : in std_logic;       -- Read received data input
                rts : out std_logic;     -- Request To Send output
                d : out std_logic_vector(7 downto 0));  -- Data bus
    end component;

    signal register1 : STD_LOGIC_VECTOR(7 downto 0);
    signal register2 : STD_LOGIC_VECTOR(7 downto 0);
    signal state : STD_LOGIC_VECTOR(2 downto 0);
    signal padding_error : STD_LOGIC_VECTOR(2 downto 0) := "000";

    signal rxrdy : std_logic := '0';        -- Receiver ready output
```

```vhdl
    signal rd : std_logic := '1';      -- Read received data input
    signal rts : std_logic := '0';         -- Request To Send output
    signal d : std_logic_vector(7 downto 0);    -- Data bus
    signal inverted_reset : std_logic := '0';
begin
    padding_error_out <= padding_error;
    buss_out <= register1(4 downto 0) & register2;

    control_signals <= register1(7 downto 5);
    inverted_reset <= not reset;  -- FBART reset is inverted (why would you do that!?)

    process(clk)
    begin
        if rising_edge(clk) then

            -- Reset our FBART
            if (reset='1') then
                state <= "000";
                has_next_data <= '0';
                padding_error <= "000";
                register1 <= "00000000";
                register2 <= "00000000";

            else
                -- Detect if we have a padding error
                if (register1(7 downto 5) = "000") then
                    if padding_error = "000" then
                        padding_error <= "000";
                    end if;
                else
                    padding_error <= register1(7 downto 5);
                end if;

                -- Wait for request_next_data request
                if state = "000" then
                    if request_next_data = '1' then
                        state <= "001";
                        rd <= '1';
                        has_next_data <= '0';

                        -- We don't want junk data in out memory, better to have DAT 0
                             0 as default.
                        register1 <= "00000000";
                        register2 <= "00000000";
                    else
                        state <= "000";
                    end if;



                -- Wait for first 8 bits of data
                elsif state = "001" then
                    if rxrdy = '1' then
                        state <= "010";
                        rd <= '0';          -- Request data read!
                    else
```

```
                    state <= "001";
                end if;

            -- Read 1st data
            elsif state = "010" then
                rd <= '1';
                register1 <= d;
                state <= "011";

            -- Wait for second 8 bits of data
            elsif state = "011" then
                if rxrdy = '1' then
                    state <= "100";
                    rd <= '0';          -- Request data read!
                else
                    state <= "011";
                end if;

            -- Read 2st data
            elsif state = "100" then
                rd <= '1';
                register2 <= d;
                state <= "000";         -- Ready for next transmission
                has_next_data <= '1';
            else
            end if;
        end if;
    end if;
end process;



fbart: fbartrx
  port map (    clk => clk,
                reset => inverted_reset,
                rxd => rxd,
                rxrdy => rxrdy,
                rd => rd,
                rts => rts,
                d => d
  );


end Behavioral;
```

B.8 vga.vhd

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

entity vga is
    Port ( rst : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           data_gpu : in  STD_LOGIC_VECTOR (7 downto 0);
           address_gpu : out  STD_LOGIC_VECTOR (12 downto 0);
           border_color : in std_logic_vector (7 downto 0);
           red : out  STD_LOGIC_VECTOR (2 downto 0);
           grn : out  STD_LOGIC_VECTOR (2 downto 0);
           blu : out  STD_LOGIC_VECTOR (1 downto 0);
           HS : out  STD_LOGIC;
           VS : out  STD_LOGIC);
end vga;

architecture Behavioral of vga is
    component colorpixSender
    Port (
            rst : in std_logic;
            clk : in std_logic;
            indata : in  STD_LOGIC_VECTOR (7 downto 0);
            border_color : in std_logic_vector (7 downto 0);
            colorpix : out  STD_LOGIC_VECTOR (7 downto 0);
            address : out  STD_LOGIC_VECTOR (12 downto 0));
    end component;

    component vgaController
    Port (
            rst : in std_logic;
            clk : in std_logic;
            colorpix : in std_logic_vector (7 downto 0);
            red : out  STD_LOGIC_VECTOR (2 downto 0);
            grn : out  STD_LOGIC_VECTOR (2 downto 0);
            blu : out  STD_LOGIC_VECTOR (1 downto 0);
            HS : out  STD_LOGIC;
            VS : out  STD_LOGIC );
    end  component;

-------------
-- SIGNALS --
-------------
signal colorpix: std_logic_vector (7 downto 0);


begin
    colordata: colorpixSender
    port map (  clk => clk,
                rst => rst,
                indata => data_gpu,
                colorpix => colorpix,
                border_color => border_color,
                address => address_gpu
    );
```

```
    vga_con: vgaController
    port map (  clk => clk,
                rst => rst,
                colorpix => colorpix,
                red => red,
                grn => grn,
                blu => blu,
                HS => HS,
                VS => VS
    );


end Behavioral;
```

B.9 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;
```

B.10 vgaController.vhd

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

entity vgaController is                              -- the vga_controller for 640
    _480_60Hz
port(
    rst : in std_logic;
    clk : in std_logic;
    colorpix : in std_logic_vector(7 downto 0);      -- 8 colorpixel bits from
        colorpixSender
    red : out std_logic_vector(2 downto 0);
    grn : out std_logic_vector(2 downto 0);
    blu : out std_logic_vector(1 downto 0);
    HS : out std_logic;                              -- HS = '1' for pixels sync
        enabled
    VS : out std_logic                               -- VS = '1' for pixels sync
        enabled
);
end vgaController;

Architecture Behavioral of vgaController is

------------------------------------------------------------------------
-- CONSTANTS
------------------------------------------------------------------------

-- maximum value for the horizontal pixel counter
constant HMAX  : std_logic_vector(10 downto 0) := "01100100000"; -- 800
-- maximum value for the vertical pixel counter
constant VMAX  : std_logic_vector(10 downto 0) := "01000001101"; -- 525
-- total number of visible columns
constant HLINES: std_logic_vector(10 downto 0) := "01010000000"; -- 640
-- value for the horizontal counter where front porch ends
constant HFP   : std_logic_vector(10 downto 0) := "01010001000"; -- 648
-- value for the horizontal counter where the synch pulse ends
constant HSP   : std_logic_vector(10 downto 0) := "01011101000"; -- 744
-- total number of visible lines
constant VLINES: std_logic_vector(10 downto 0) := "00111100000"; -- 480
-- value for the vertical counter where the front porch ends
constant VFP   : std_logic_vector(10 downto 0) := "00111100010"; -- 482
-- value for the vertical counter where the synch pulse ends
constant VSP   : std_logic_vector(10 downto 0) := "00111100100"; -- 484
-- polarity of the horizontal and vertical synch pulse
-- only one polarity used, because for this resolution they coincide.
constant SPP   : std_logic := '0';


------------------------------------------------------------------------
-- SIGNALS
------------------------------------------------------------------------

signal hcounter : std_logic_vector(10 downto 0) := (others => '0');     -- counter for
```

```
     horizontal timing
signal vcounter : std_logic_vector(10 downto 0) := (others => '0');     -- counter for
     vertical timing
signal pixel_cnt : std_logic_vector(1 downto 0) := "00";

begin
    process(clk)
    begin
      if(rising_edge(clk)) then
            if rst = '1' then                                         --reset all the
                counters
                hcounter <= "00000000000";
                vcounter <= "00000000000";
                pixel_cnt <= "00";
            end if;

            if pixel_cnt = "11" then                                  -- pixel_cnt in 25MHz
                pixel_cnt <= "00";
                -----------------
                -- Main starts --
                -----------------
                if hcounter = HMAX then                               -- increase h_counter
                    hcounter <= "00000000000";                        -- reset h_counter
                        when HMAX
                    if vcounter = VMAX then                           -- increase v_counter
                        vcounter <= "00000000000";                    -- reset v_counter
                            when VMAX
                    else
                        vcounter <= vcounter + 1;
                    end if;
                else
                    hcounter <= hcounter + 1;
                end if;

                if hcounter >= HFP and hcounter < HSP then            -- generate and
                    refresh HS signal during display area
                    HS <= SPP;                                        -- HS <= '0'
                else
                    HS <= not SPP;                                    -- HS <= '1'
                end if;

                if vcounter >= VFP and vcounter < VSP then            -- generate and
                    refresh VS signal during display area
                    VS <= SPP;                                        -- VS <= '0'
                else
                    VS <= not SPP;                                    -- VS <= '1'
                end if;

                if hcounter < HLINES and vcounter < VLINES then
                    if vcounter < 476 then                            -- send color pixels
                        to vga port within display area
                        red <= colorpix (2 downto 0);
                        grn <= colorpix (5 downto 3);
                        blu <= colorpix (7 downto 6);
                    else
                        red <= "000";
```

```
                        grn <= "000";
                        blu <= "00";
                    end if;

                else
                    red <= "000";
                    grn <= "000";
                    blu <= "00";
                end if;
            else
                pixel_cnt <= pixel_cnt + 1;
           end if;
        end if;
    end process;
end Behavioral;
```

B.11 MARCled.vhd

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MARCled is
    Port (
            clk,rst : in  STD_LOGIC;
            ca,cb,cc,cd,ce,cf,cg,dp : out  STD_LOGIC;
            an : out  STD_LOGIC_VECTOR (3 downto 0);

            game_started : in std_logic;
            current_instr : in std_logic_vector(3 downto 0);

            game_over : in std_logic;
            active_player : in std_logic_vector(1 downto 0)
         );
end MARCled;

architecture Behavioral of MARCled is
    signal segments : STD_LOGIC_VECTOR (6 downto 0);
    signal counter_r : std_logic_vector(17 downto 0) := (others => '0');

    alias led_state is counter_r(17 downto 16);
    --alias led_state is counter_r(1 downto 0);
    signal scroll_counter : std_logic_vector(26 downto 0) := (others => '0');
    signal scroll_offset : std_logic_vector(5 downto 0) := (others => '0');
    signal current_char : std_logic_vector(5 downto 0) := (others => '0');

    -- 00   show MARC
    -- 01   show player 1 victory
    -- 10   show player 2 victory
    -- 11   show instruction mnemonic
    signal display_state : std_logic_vector(1 downto 0) := "00";

    signal instr_start : std_logic_vector(5 downto 0) := (others => '0');
    signal instr_char : std_logic_vector(5 downto 0) := (others => '0');

    subtype DataLine is std_logic_vector(6 downto 0);

    type IdData is array (0 to 7) of DataLine;
    signal id : IdData := (
       "0110000", -- M (E)
       "0001000", -- A
       "1111010", -- r
```

```
   "0110001", -- C

   -- Blanking period
   "1111111", --
   "1111111", --
   "1111111", --
   "1111111" --
);

type PlayerData is array (0 to 42) of DataLine;
signal player1_victory : PlayerData := (
   "0001000", -- A
   "1110001", -- L
   "1110001", -- L
   "1111111", --
   "1001100", -- Y
   "0000001", -- O
   "1000001", -- U
   "1111010", -- r
   "1111111", --
   "0110001", -- C
   "0000001", -- O
   "1000010", -- d
   "0110000", -- E
   "1111111", --
   "0001000", -- A
   "1111010", -- r
   "0110000", -- E
   "1111111", --
   "1100000", -- b
   "0110000", -- E
   "1110001", -- L
   "0000001", -- O
   "1101010", -- n
   "0100000", -- G
   "1111111", --
   "0111001", -- T
   "0000001", -- O
   "1111111", --
   "1000001", -- U
   "0100100", -- S
   "1111111", --
   "0011000", -- P
   "1110001", -- L
   "0001000", -- A
   "1001100", -- Y
   "0110000", -- E
   "1111010", -- r
   "1111111", --
   "1001111", -- 1
   "1111111", --
   "1111111", --
   "1111111", --
   "1111111" --
);
```

```
signal player2_victory : PlayerData := (
   "0001000", -- A
   "1110001", -- L
   "1110001", -- L
   "1111111", --
   "1001100", -- Y
   "0000001", -- O
   "1000001", -- U
   "1111010", -- r
   "1111111", --
   "0110001", -- C
   "0000001", -- O
   "1000010", -- d
   "0110000", -- E
   "1111111", --
   "0001000", -- A
   "1111010", -- r
   "0110000", -- E
   "1111111", --
   "1100000", -- b
   "0110000", -- E
   "1110001", -- L
   "0000001", -- O
   "1101010", -- n
   "0100000", -- G
   "1111111", --
   "0111001", -- T
   "0000001", -- O
   "1111111", --
   "1000001", -- U
   "0100100", -- S
   "1111111", --
   "0011000", -- P
   "1110001", -- L
   "0001000", -- A
   "1001100", -- Y
   "0110000", -- E
   "1111010", -- r
   "1111111", --
   "0010010", -- 2
   "1111111", --
   "1111111", --
   "1111111", --
   "1111111" --
);

type InstrData is array (0 to 43) of DataLine;
signal instr : InstrData := (
   "1111111", --
   "1000010", -- d
   "0001000", -- A
   "0111001", -- T

   "1111111", --
   "0001001", -- m (N)
   "0000001", -- O
```

```
        "1100011", -- v (u)

        "1111111", --
        "0001000", -- A
        "1000010", -- d
        "1000010", -- d

        "1111111", --
        "0100100", -- S
        "1000001", -- U
        "1100000", -- b

        "1111111", --
        "0000011", -- J
        "0001001", -- m (N)
        "0011000", -- P

        "1111111", --
        "0000011", -- J
        "0001001", -- m (N)
        "0010010", -- Z

        "1111111", --
        "0000011", -- J
        "0001001", -- m (N)
        "1101010", -- n

        "1111111", --
        "0110001", -- C
        "0001001", -- m (N)
        "0011000", -- P

        "1111111", --
        "0100100", -- S
        "1110001", -- L
        "0111001", -- T

        "1111111", --
        "1000010", -- d
        "0000011", -- J
        "1101010", -- n

        "1111111", --
        "0100100", -- S
        "0011000", -- P
        "1110001", -- L

        others => (others => '1')
    );

    signal one : std_logic_vector(6 downto 0) := "1001111"; -- 1
    signal two : std_logic_vector(6 downto 0) := "0010010"; -- 2
begin
    ca <= segments(6);
    cb <= segments(5);
    cc <= segments(4);
```

```
cd <= segments(3);
ce <= segments(2);
cf <= segments(1);
cg <= segments(0);
dp <= '1';

    segments <= id(conv_integer(current_char(2 downto 0))) when display_state = "
        00" else
                player1_victory(conv_integer(current_char(5 downto 0))) when
                    display_state = "01" else
                player2_victory(conv_integer(current_char(5 downto 0))) when
                    display_state = "10" else
    one when led_state = "00" and display_state = "11" and active_player = "01"
        else
    two when led_state = "00" and display_state = "11" and active_player = "10"
        else
                instr(conv_integer(instr_char));

with current_instr select
    instr_start <=  "000000" when "0000", -- DAT
                    "000100" when "0001", -- MOV
                    "001000" when "0010", -- ADD
                    "001100" when "0011", -- SUB
                    "010000" when "0100", -- JMP
                    "010100" when "0101", -- JMZ
                    "011000" when "0110", -- JMN
                    "011100" when "0111", -- CMP
                    "100000" when "1000", -- SLT
                    "100100" when "1001", -- DJN
                    "101000" when "1010", -- SPL
                    "000000" when others;

process(clk) begin
    if rising_edge(clk) then

        counter_r <= counter_r + 1;

        -- Update scroll counter
        if rst = '1' then
            scroll_counter <= (others => '0');
        elsif scroll_counter(26) = '1' then
        --elsif scroll_counter(2) = '1' then
            if led_state = "11" then
                scroll_counter <= (others => '0');

                -- Artificial max
                if scroll_offset = "100111" then
                    scroll_offset <= (others => '0');
                else
                    scroll_offset <= scroll_offset + 1;
                end if;
            end if;
        -- Check resets...
        elsif (display_state = "00" and (game_started = '1' or game_over = '1'))
            or
              ((display_state = "01" or display_state = "10") and game_over = '0')
```

```
                 or
               (display_state = "11" and game_over = '1') then

            scroll_counter <= (others => '0');
            scroll_offset <= (others => '0');
        else
            scroll_counter <= scroll_counter + 1;
        end if;

        current_char <= led_state + scroll_offset;
        instr_char <= led_state + instr_start;

        -- Change of display state
        if rst = '1' then
            display_state <= "00";
        elsif display_state = "00" and game_over = '1' then
            display_state <= active_player;
        elsif display_state = "00" and game_started = '1' then
            display_state <= "11";
        elsif (display_state = "01" or display_state = "10") and game_over = '0'
            then
            display_state <= "00";
        elsif display_state = "11" and game_over = '1' then
            display_state <= active_player;
        end if;

        -- Set current led display to output
        case led_state is
            when "00" => an <= "0111";
            when "01" => an <= "1011";
            when "10" => an <= "1101";
            when others => an <= "1110";
        end case;
    end if;
end process;

end Behavioral;
```

C Script

C.1 Assembler

```perl
#!/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;
}
```

```perl
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);
```

```perl
$b_op = "0" if !$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",
```

```perl
);

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;
}
```