En rapport i kursen Datorkonstruktion (TSEA83)

Elliot Norlander, ellno907
Jacob Sjölin, jacsj573
Jennifer Santos, jensa682
Elin Rydebrink, eliry213

Datateknik (D)
Linköpings Universitet, Linköping 2023-05-22 (Version 1)

Innehållsförteckning

1. Inledning

1.1 Bakgrund

I kursen Datorkonstruktion (TSEA83) vid Linköpings universitet får studenterna i uppgift att konstruera en egen processor under en period om cirka tre månader. Studenterna får fritt välja om de ska konstruera en mikrokodad- eller pipeline-processor. Utöver detta ska även ett enklare program konstrueras för att visa funktionaliteten hos processorn. Kraven på dessa program är relativt få, vilket leder till många varierande projekt bland de olika projektgrupperna.

Med detta i åtanke valde gruppen att konstruera en variant av det klassiska spelet Snake. Snake är ett tvådimensionellt spel där användaren styr en orm på en förutbestämd kvadratisk spelplan. Målet med spelet är att styra in ormen i så många poäng som möjligt. Varje gång spelaren styr in ormen i ett poäng ökar ormens längd med en enhet. På detta sätt blir ormen längre och längre desto fler poäng spelaren får. Således blir spelet svårare och svårare. Spelet har sitt ursprung i spelet "Blockade" som kom ut 1976[1]. Eftersom spelet inte har någon ensam ägare har det sedan dess kommit ut många varianter av spelet, bland annat för apples "Apple II" år 1978 och nokias "6110" år 1997[2]. Just detta har gjort snake till ett mycket populärt och klassiskt 2D-spel.

1.2 Syfte

Syftet med projektet är först och främst att studenten ska bekanta sig med en dators konstruktion och funktionalitet i detalj, däribland in- och utdata, olika minnestyper samt bildgenerering med VGA. Det finns också ett bredare syfte med projektet. Studenterna får inte bara vanan av att arbeta i grupp under längre tid, men också erfarenhet kring hur det är att konstruera en teknisk produkt från start till slut, hur man hanterar komplexa problem samt vikten av god kommunikation.

[1] "Gaming Hitsory". Hämtad 21/5-2023, https://www.arcade-history.com/?n=blockade-model-807-0001&page=detail&id=287

[2] "Snake (Video Game Genre)". Hämtad 21/5-2023. https://en.wikipedia.org/wiki/Snake_(video_game_genre)

2. Apparaten

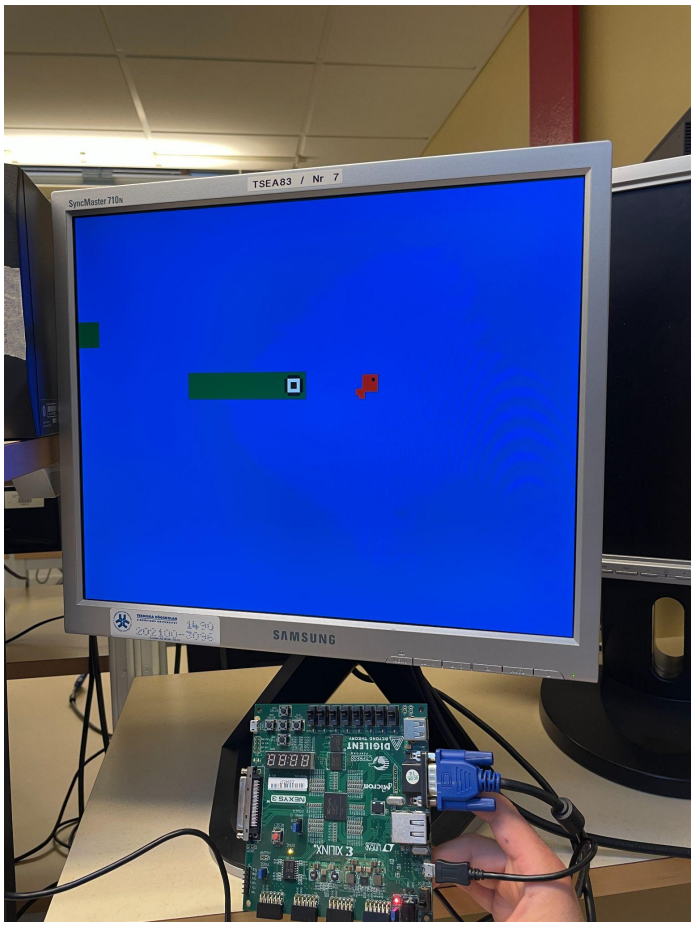


Bild 1, översiktsbild.

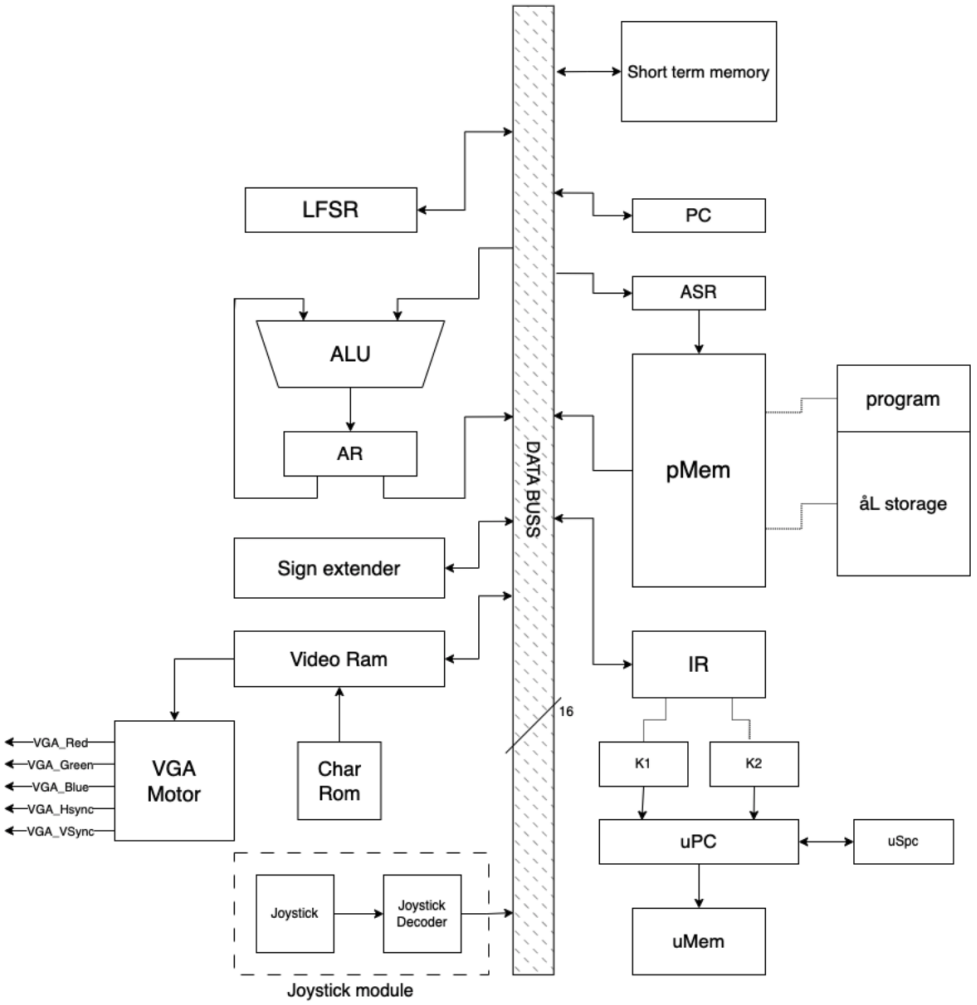
Apparaten består i huvudsak av tre delar. En joystick, en VGA-skärm samt ett FPGA-kort av typen Digilent Nexys 3. Kortet och skärmen sammankopplas med en VGA-kabel och joysticken pluggas in i porten JA på Nexys 3. En konventionell dator sammankopplas med hjälp av en USB-MicroUSB kabel. Denna pluggas in i porten USBProg på Nexys 3. Då strömmen slagits på är kortet redo att användas.

För att programmera kortet används terminalen hos den konventionella datorn. Genom att navigera till lokaliseringen för spelfilerna och sedan gå vidare till uprogCPU kan kortet programmeras genom att först skriva “module add courses/TSEA83”, sen “make proj.bitgen” och till sist ”make proj.prog”. Spelet startar då och visas på skärmen.

För att styra ålens rörelse används joysticken. Genom att styra in ålen i en fisk erhåller spelaren ett poäng, i vilket ålen blir längre. Då spelaren kolliderar med sig själv, eller en av de yttre väggarna, avslutas spelet. För att starta om används kommandot “make proj.prog”. Spelet startar då om.

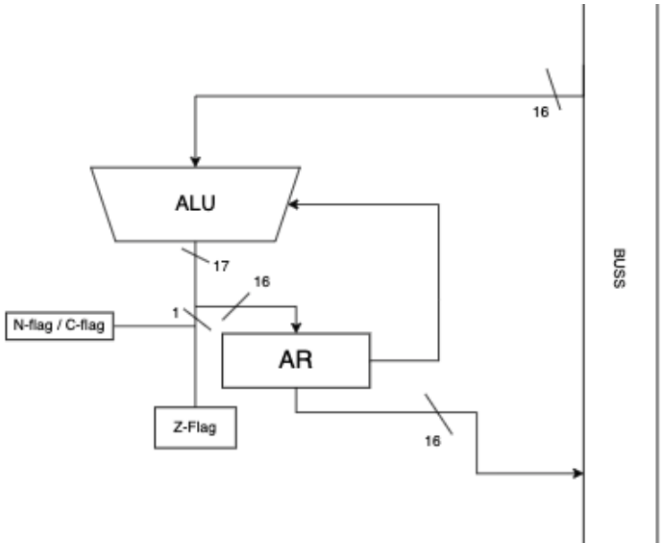
3. Hårdvaran

Konstruktionen består av ett flertal olika komponenter. Däribland en ALU, en VGA-motor och ett programminne. Processorn är mikrokodad, vilket innebär att ett mikrominne har använts för att styra hur datorns centrala delar ska fungera. En tydligare bild av den övergripande konstruktionen kan fås av figur blockschema 1.



Blockschema 1, övergripande blockschema över konstruktionen.

3.1 Aritmetisk logisk enhet (ALU)



Blockschema 2, blockschema över aritmetisk logisk enhet

Den aritmetiska enheten ger datorn möjlighet att utföra en rad olika aritmetiska operationer. Dessa specificeras i tabell 1.

Operation	Förklaring
Add, A, B	A = A + B
Sub, A, B	A = A - B
And, A, B	A = A & B
Or, A, B	A =A | B
lsr, A	Logic shift right, A

Tabell 1, översikt över implementerade operationer

Enhetens konstruktion beskrivs av en relativt enkel case-do sats. Beroende på vilken kombination av signaler som registreras från porten operator (OP) utför enheten olika typer av operationer. Till exempel medför OP-koden “010” att addition kommer utföras. Resultatet går ut genom den 17 bitar breda R-porten, därefter maskas den mest signifikanta biten ut och resterande 16 bitar lagras på AR-registret. Den utmaskade biten används sedan för att beräkna N-flaggan.

ALU-enheten ansvarar också för tilldelning av flaggor i form av en Z, N, C och V flagga. Dessa beskrivs mer konkret i tabell 2.

Benämning	Förklaring
Z-flagga	Aktiv då den senaste ALU operationen resulterade i noll.
N-flagga	Aktiv då den senaste ALU operationen resulterade i ett negativt resultat.
C-flagga	Aktiv då den senaste ALU operationen resulterade i ett tal av större storlek än den tillåtna bitbredden.
V-flagga	Aktiv då den senaste ADD- eller SUB-operationen resulterade i ett tal av större storlek än den tillåtna bitbredden.

Tabell 2, översikt över implementerade flaggor.

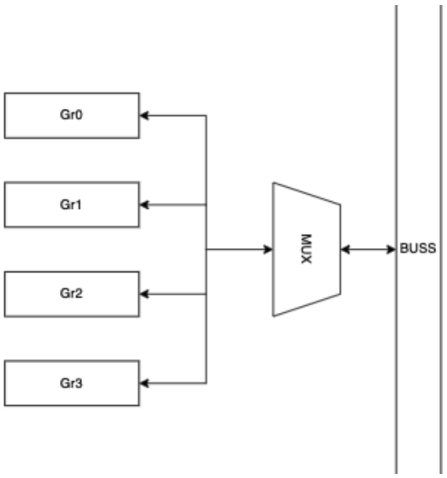
För att beräkna flaggorna tillämpas en hel del logik. Z-flaggan tilldelas ett när hela R är noll. N-flaggan tilldelas värdet av den näst mest signifikanta biten av R. På samma sätt tilldelas C-flaggan värdet av den mest signifikanta biten hos R. V-flaggan beräknas med hjälp av ett flertal logiska operationer där de mest signifikanta bitarna behandlas.

3.1.1 Sign extender

Sign extender används då relativa hopp ska utfärdas i programminnet. Vid ett relativt hopp används de sista åtta bitarna i instruktionen för att hänvisa vart mikroprogramräknaren ska hoppa, detta används främst i hopp-instruktioner. De relativa hoppen kräver att man ska kunna hoppa fram och tillbaka, därmed måste tvåkomplementstal användas. Sign extendern är lokaliserad i CPUn och tar in de 8 sista bitarna i instruktionsregistret. De åtta bitarna skrivs över till ett eget register, och värdet på den åttonde biten skrivs till bit 9 till 16. Det betyder att om instruktionsregistet innehåller ett två komplements tal kan detta hanteras av CPUn eftersom sign extendern omvandlar ett 8-bitars tal till ett 16-bitar tal, där den åttonde biten avgör om det är ett negativt tal eller inte. Tillgång ges av en specifik TB-kod.

3.2 Korttidsminne

Processorns korttidsminne sköts i form av en två bits multiplexer följt av fyra 16 bitars register. Se blockschema 3.



Blockschema 3, Korttidsminne och mux

Multiplexern styrs med hjälp av mikrokod. Fältet “Grx” i mikroinstruktionen bestämmer vilken av de fyra in- eller utgångarna som ska vara aktiva. På så sätt kan registren snabbt skriva till samt läsa från bussen.

3.3 Programminne

Programminnet används för att lagra instruktionsregister och är ett 16x1024 brett minne. Vilken instruktion som ska avläsas bestäms av programräknaren. Programminnet har två portar, en för CPUn och en för VGA-Motorn. CPUn har tillgång till vilken instruktion som läses av mikroprogramräknaren, och kan även skicka information till programminnet för att manipulera register. Video-RAM använder bara utdata från programminnet för att läsa av specifika register.

De första 240 registren i programminnet är förprogrammerad programkod som används för spelets funktionalitet. Resterande rader används för att beskriva ormen, i början av spelet dyker en liten förprogrammerad ål upp på skärmen. Vid varje förändring på skärmen ändras ormens data av VGA-motorn, därför tillägnas resterande register till lagring av ormen.

3.3.1 Programräknare

Programräknaren hanteras som en egen modul i CPUn och kommer att öka med ett varje gång bit P från mikroinstruktionen är hög. Räknaren kan också tilldelas ett värde om FB tilldelas programräknarens FB-kod.

3.4 Mikrominne

I det 32x48 breda mikrominnet sparas alla mikroinstruktioner. Instruktionerna hanterar hur information ska förflyttas i hårdvaran där varje instruktion är 32 bitar bred. Det finns totalt 16 operander som använder 1 till 8 mikroinstruktioner, och avslutet markeras med SEQ ”0011”. Varje instruktion är uppdelad på följande sätt:

	ALU	TB	FB	NOP	P	LC	SEQ	myADR
BIT:	31-29	28-25	24-21	20	19	18-17	16-13	12-0

3.4.1 Mikroprogramräknare

Mikroprogramräknaren (uPC) hittas i CPUn och beroende på vilket värde SEQ blir tilldelat i mikroinstruktionen kommer uPC bli tilldelad olika värden. Följande är exempel på olika SEQ-koder:

SEQ	uPC
0000	uPC räknas upp
0001	Tilldelas K1_reg
1111	Halt

3.5 Loop Counter

Loop Counter används som en räknare, och beroende på vilket kod som mikroinstruktionen tilldelar kommer LC att anta olika värden. LC används av specifika mikroinstruktioner som behöver iterera ett förutbestämt antal gånger, då hjälper LC att hålla koll på hur många iterationer som kvarstår innan det är dags att avbryta.

3.6 Databuss

Databussen används för att skicka information från ett ställe till ett annat. Mikroinstruktionen tilldelar ett värde till TB och CPUn kommer att skicka matchande information till bussen. Fältet FB säger vilken modul som ska läsa av bussen och därmed tilldelas informationen som finns på bussen.

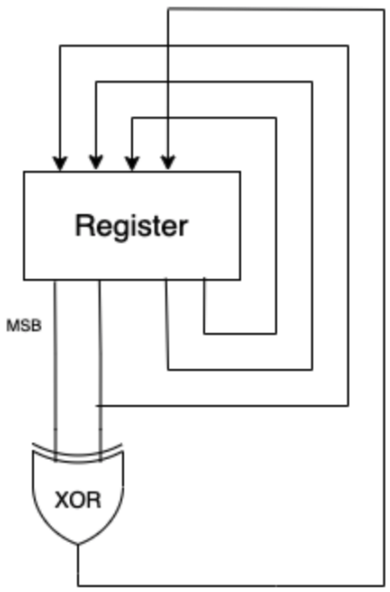
3.7 K1

Varje operand har en plats i den 13x16 breda K1 modulen, och dess placering i K1 är det instruktionsregistret hänvisar till. I varje placering sparas operandens startadress, som finns lokaliserad i mikrominnet. K1 används som toppmodul, för att skydda mikrominnet, så att instruktionsregistret inte kan använda mikroinstruktioner felaktigt.

3.8 K2

För att veta vilken information som ska skrivas/avläsas används K2 för att bestämma hur adresseringen ska ske. Det finns fyra olika typer, och dessa lagras i K2 modulen som är 24x4 bred. K2 modulens register hänvisar till vart i mikrominnet som de olika typerna av adressering sker. De fyra olika typerna är direkt, indirekt, relativ, och absolut adressering.

3.9 Slumpgenerator (LFSR)



Blockschema 4, konstruktion av LFSR.

En slumpgenerator har konstruerats. Detta är nödvändigt för att kunna lägga till poäng på brädet på ett slumpmässigt sätt. En slumpgenerator av typen Linear Feedback Shift Register (LFSR) har därför konstruerats, mer specifikt ett typ av LFSR som använder exclusive-or (XOR) för att operera.

Inledningsvis består registret av ett förutbestämt värde av längd fyra, sedan utförs en XOR operation på de två mest signifikanta bitarna i registret och resultatet av denna operation sparas undan i en signal vars namn är feedback. Feedback placeras på den minst signifikanta biten i registret och således flyttas de tidigare tre minst signifikanta bitarna ett steg åt vänster och den tidigare mest signifikanta blir därmed irrelevant. Denna operation körs kontinuerligt när programmet är igång, vilket ger tillgång till ett nytt slumptal varje klockcykel.

3.10 VGA-Motor

Vga-motorn har flera olika komponenter som samverkar. I bildminnet ligger beskrivningar av hur de olika tiles som används i spelet ser ut. Bilderna består av 8x8 pixlar men representeras av 32x32 pixlar på VGA-monitorn. Bildminnet har 4 tiles som är: ål-huvud, ål-kropp, bakgrund och en fisk som ålen ska fånga.

För att bestämma vart på skärmen varje tile ska befinna sig används ett video RAM minne. Det är ett minne som hela tiden uppdateras beroende på var ålen befinner sig på skärmen vid olika tillfällen. Var varje tile, som ålen består av, har för position på skärmen finns sparat längst ner i programminnet och uppdateras kontinuerligt under spelets gång. Positionen för tiles ligger sparade i ett 16 bitars tal där bit 10 till 6 beskriver X-positionen och bit 5 till 2 beskriver Y-positionen. De sista två bitarna talar om för programmet vilken tile som ska vara placerad på den positionen. För att talet ska bli lika lång som den övriga programkoden så finns det fem nollor i början av talet för att fylla ut. Se ett exempel i tabell 3

Utfyllnad	X-position	Y-position	Tile typ
00000	01000	0110	01

Tabell 3, ålens positioner.

Video RAM tar in all denna information och använder den för att placera ut tiles på skärmen. Plats (0, 0) på skärmen är längst upp i vänstra hörnet och (20, 15) är längst ner till höger. VGA-motorn ritar upp varje pixel på skärmen. Koden för VGA-motorn som användes i detta projekt är hämtad från en laboration som utfördes i kursen TSEA83 några månader tidigare.

3.11 Joystick

Huvuddelen av koden till joystickmodulen kommer från ett kompendium som erhålls från kurshemsidan för TSEA83. Egen kod skrivits för att joysticken ska fungera med resten av koden. I dagsläget fungerar inte joysticken enligt förväntan då signalen “direction”, som ska tala om för programkoden i vilken riktning ålen rör sig i, konstant har samma värde. Även efter timmar av felsökning så har inte detta problem kunnat lösas och ålen kan nu endast röra sig i en riktning.

Tanken med koden är att signalen “direction” ska uppdateras konstant och ändra värde om joysticken förflyttades tillräckligt långt i någon riktning. Joysticken har två riktningar: X och Y, samt en signal för knapparna som finns på joysticken.

4. Slutsatser

Projektarbetet anses i allmänt vara lyckat. En mikroprogrammerad processor med fungerande register, logik och I/O har konstruerats, vilket var det översiktliga målet. Ett snake-liknande spel har också konstruerats med framgångsrikt resultat. Problem har givetvis förekommit, framförallt med avseende på I/O och avläsningen av joystick.

4.1 Vad som gick bra

Generellt anses projektet vara lyckat. Målet var att konstruera en mikroprogrammerad processor - vilket har uppfyllts med gott resultat. Konstruktionen av instruktionshantering, mikrokod samt ALUn upplevdes som strömlinjeformad. Detta är en av projektets starka sidor, då det implementerade systemet fungerar väldigt bra. Mer generellt upplever gruppens medlemmar att samarbetet mellan medlemmarna fungerade bra. I många fall arbetade gruppmedlemmarna med sina egna komponenter, för att sedan föra ihop arbetet med hjälp av git. Detta fungerade mycket bra, då problem som uppstod under vägen lätt kunde spåras.

4.2 Problem

Som tidigare nämnt uppstod en hel del problem under projektets gång. Många av dessa var små och lättlösta, men vissa problem var större och mer komplexa. Ett exempel på detta var när det upptäcktes att instruktionsbredden var för kort. Ursprungligen användes 8-bitars instruktioner. Detta ändrades till 16 bitar då detta krävdes för att få tillgång till hoppinstruktioner som BRA och BNE. Detta ledde i sin tur att databussens bredd behövde utökas. Detta skapade nya mindre problem, vilket saktade ner arbetet markant.

Ytterligare ett problem uppstod då implementationen av joysticksavkodaren skulle genomföras. Innan detta hade gruppen ett för det mesta fungerande projekt, men under implementationen av joysticken skapades ett signalproblem i VGA-porten. Detta gjorde att skärmen inte visade spelet utan istället var svart. Detta är ett problem som förblivit olöst.

4.3 Vad borde vi gjort annorlunda?

Efter projektets avslutning kan gruppen dra slutsatsen att en hel del förändringar hade varit önskvärda. Framförallt önskar gruppen att man konstruerat och implementerat sin egen kompilator. Detta hade signifikant kortat ner implementeringen av spelet. Framförallt hade det underlättat felsökningen av programmet då detta hade kunnat göras i en IDE såsom VS code, för att sedan kompileras till programkod. I den valda arbetsgången behövde gruppen felsöka koden genom modelsim, vilket var mycket tidskrävande.

Vidare önskar gruppen också att de hade implementerat absoluta hopp istället för relativa. Ett relativt hopp betyder i sig att programräknaren hoppar ett förutbestämt antal rader i positiv eller negativ riktning. Detta betyder att programmeraren själv måste hålla reda på samt beräkna hur långt räknaren ska hoppa för att komma till rätt del av programkoden. En

konsekvens av detta är att hoppen måste räknas om varje gång programmeraren lägger till eller tar bort en rad i programkoden, vilket ödslar en hel del tid.

Ur ett processmässigt perspektiv hade gruppen också gjort en del ändringar. Inför projektet skapades en översiktlig designspecifikation. Där inkluderades en lista med preliminära instruktioner som skulle implementeras. När det sedan var dags att implementera dessa följde gruppen listan utan att reflektera om alla instruktioner skulle behövas. Totalt implementerades ungefär 25 instruktioner, men spelet använder endast 16 av dessa. Det betyder att en hel del tid slösades på att implementera onödiga instruktioner. Gruppen hade föredragit att först skapa spelet för att se vilka instruktioner som skulle behövas och sedan implementerat just denna uppsättning.

4.4 Lärdomar

Projektet har varit mycket lärorikt ur ett flertal aspekter. Primärt har deltagarna lärt sig mer kring datorer, deras inre komponenter, hur de är konstruerade samt samspelet mellan dem. Innan projektet hade deltagarna ingen kunskap kring hur till exempel en VGA-signal fungerade, eller hur man konstruerar en fungerande ALU. Under projektets gång har gruppdeltagarna dock kontinuerligt tagit sig an nya koncept och utmaningar..

Gruppmedlemmarna har inte bara fått tekniska lärdomar, de har också lärt sig en hel del branschnödvändiga kunskaper. Projektet har varit ett bra tillfälle för gruppmedlemmarna att arbeta i större projekt i git samt lösa eventuella problem relaterade till versionshantering. Vidare har gruppmedlemmarna också skapat sig erfarenhet kring längre grupparbeten, kommunikationen inom dessa samt vikten av att ha en god och lättläst kodstruktur.

4.5 Kontentan

Projektet var i stora drag väldigt lyckat. Gruppmedlemmarna har utvecklat sin förståelse kring datorns inre byggstenar, samt skapat sig kompetensen och erfarenheten kring hur det är att konstruera dessa.

5. Referenser

1. “Gaming Hitsory”. Hämtad 21/5-2023,
https://www.arcade-history.com/?n=blockade-model-807-0001&page=detail&id=287

2. “Snake (Video Game Genre)”. Hämtad 21/5-2023.
https://en.wikipedia.org/wiki/Snake_(video_game_genre)

6. Appendix

Tabell över implementerade och använda instruktioner

Benämning	Funktion
LOAD. Rd, Ra	Rd <= MEM(Ra)
STORE. Rd, Ra	MEM(Rd) <= Ra
ADDI. Rd, const	Rd <= Rd+const, uppdatera flag
SUBI, Rd, const	Rd <= Rd - const, uppdatera flag
ANDI. Rd, const	Rd <= Rd AND const, uppdatera flag
CMPI. Rd, const	Rd-const, uppdate (Z,N,C,V)
ORI. Rd, const	Rd <= Rd | const
BRA. offset	Hopp till PC+1+offset
BNE. offset	Hopp om Z = 0
BMI. offset	Hopp om N = 1
BEQ. offset	Hopp om Z = 1
BGE. offset	Hopp om N = 0
HALT	STOP PC
Get_Random	Get random value from LFSR
Get_Joystick	Get new direction from joystick