Teknisk rapport

För projektarbete i kursen TSEA83 (grupp 34).

May 13, 2022

Revision 1

Innehållsförteckning

1. Inledning

Detta är en teknisk rapport om det projektarbete vi utfört i kursen datorkonstruktion (TSEA83). Projektet handlar om att konstruera en dator på FPGA-kortet NEXYS3 som kan exekvera program och hantera in- och utdata.

2. Introduktion till datorn

När vi designade datorn ville vi ha en von Neumann arkitektur, en tvåstegs pipeline och diverse kringutrustning för att kunna spela vår version av spelet asteroider. Då det är en fungerande dator kan man såklart programmera den till att göra andra saker också. Kringutrustningen som används är en joystick, en VGA-skärm samt en USB kabel för UART kommunikation. I följande kapitel kommer vi redogöra för hur spelet fungerar, hur man programmerar FPGA kortet samt hur man assemblerar och laddar upp program till datorn.

2.1. Spelet asteroider

Spelet asteroider som vi tagit fram går ut på att undvika att krocka in i asteroider som flyger på skärmen med ens eget rymdskepp. Skeppet styrs via en joystick och desto längre man överlever desto fler poäng får man. Poängen visas på en 7-segment display som finns på NEXYS3-kortet. En nackdel för den spelare med höga poäng är att spelet också kommer gå fortare. Följande bild visar hur det kan se ut under en spelomgång.

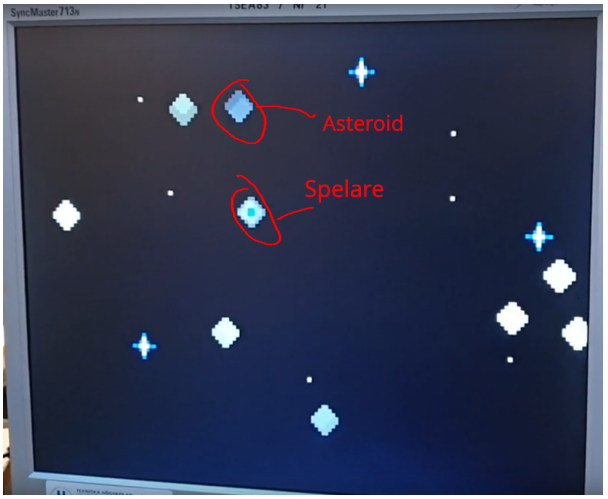

Bild 1: Spelet asteroider körandes på vår datorkonstruktion.

2.2. Initalisering av datorn

För att kunna använda datorn måste vi göra ett par saker först. Dels måste själva konstruktionen programmeras in på NEXYS3-kortet, dels så måste kringutrustning kopplas in korrekt och dels så måste spelet laddas upp. Allt detta beskrivs i följande kapitel.

2.2.1. Inkoppling

Enheten måste kopplas upp korrekt för att man ska kunna spela på den, eller för att ens kunna ladda upp program. Observera att man måste programmera NEXYS3-kortet i steg två nedan för att den visuella återkopplingen som beskrivs i senare steg ska uppstå.

1. Koppla in USB kablar från en dator till “USB PROG” samt “UART” portarna på NEXYS3-kortet. Några LEDs på kortet bör lysa upp.
2. Programmera NEXYS3-kortet enligt instruktionerna under kapitel *2.2.2. Programmering av NEXYS3-kortet*.
3. Koppla in joysticken på JA porten 0-3. Joystickens LED lampor bör börja lysa om kopplingen gjorts korrekt.
4. Koppla in kabeln för VGA. Om VGA-skärmen är på ska en rymdbakgrund visas.

2.2.2. Programmering av NEXYS3-kortet

För att programmera NEXYS3-kortet med rätt hårdvarukonstruktion måste man först navigera till rotmappen för projektet[1] i en terminal. Därifrån går man in i undermappen *./src/*, och kör sedan följande instruktion:

make pipecpu.bitgen && make pipecpu.prog

2.2.3. Programuppladdning

För att få datorn att exekvera någonting måste ett program laddas upp. Detta program måste såklart innehålla vettiga maskininstruktioner. Det lättaste sättet att skapa ett sådant program är att läsa kapitel *2.3. Instruktionsuppsättning*, eller bara ladda upp det färdiga spelet som finns i rotmappen för projektet. Följande steg beskriver en uppladdning av programmet *binary.bin*.

1. Se till att NEXYS3-kortet är korrekt inkopplat och programmerat.
2. Programmera USB-porten som är kopplad till NEXYS3-kortets UART port till att ha baudrate 115200, data bits 8, stop bits 1 och parity none (exempelvis via gtkTerm).
3. Ladda upp valfritt program genom att skicka filen till enheten i Linux, e.x: *cat binary.bin > /dev/ttyUSB0*

Om man då vill ladda upp spelet asteroider ställer man sig i rotmappen i en terminal och skriver: *cat asteroids.bin > /dev/ttyUSB0*. Om nu NEXYS3-kortet dyker upp på port USB0 som då är rätt konfigurerat.

För mer information om hur programminnesladdaren är implementerad, se kapitel *3.3. Programminnesladdaren*.

2.3. Instruktionsuppsättning

Datorn är programmerbar via 34 st olika instruktioner. I projektet inkluderar vi en assembler som kan användas för att göra om mnenomics för instruktioner till maskinkod som kan laddas upp till datorn. I detta kapitel introducerar vi instruktionerna och assemblern.

[1] Se gitlab: https://gitlab.liu.se/da-proj/TSEA83/2022/gr34/projekt

2.3.1. Instruktioner

Samtliga instruktioner följer nedan i *tabell 1*. Det finns även några interna instruktioner som inte är menade att programmeras med direkt, dessa är utmärkta genom att ha kanter i tabellen. Dom instruktionerna används internt för att underlätta exekvering av andra instruktioner.

Hex	Mnemonic	Funktion
0x00	NOP	No operation
0x01	RJMP *offset/etikett*	Hopp till PC+1+*offset/etikett*
0x02	BEQ *offset/etikett*	Hopp om Z = 1
0x03	BNE *offset/etikett*	Hopp om Z = 0
0x04	BPL *offset/etikett*	Hopp om N = 0
0x05	BMI *offset/etikett*	Hopp om N = 1
0x06	BGE *offset/etikett*	Hopp om N xor V = 0
0x07	BLT *offset/etikett*	Hopp om N xor V = 1
0x08	LDI *Rd,Konst*ant	Rd <= Konstant
0x09	LD *Rd,Ra*	Rd <= DM(Ra)
0x0A	STI Rd,*Konstant*	DM(Rd) <= Konstant
0x0B	ST *Rd,Ra*	DM(Rd) <= Ra
0x0C	COPY *Rd,Ra*	Rd <= Ra
0x0D	ADD *Rd,Ra*	Rd <= Rd + Ra, uppd. Z, N, C, V
0x0E	ADDI *Rd,Konst*ant	Rd <= Rd + Konstant, uppd. Z, N, C, V
0x0F	SUB *Rd,Ra*	Rd <= Rd - Ra, uppd. Z, N, C, V
0x10	SUBI *Rd,Konstant*	Rd <= Rd – Konstant, uppd. Z, N, C, V
0x11	CMP *Rd,Ra*	Rd - Ra, uppd. Z, N, C, V
0x12	CMPI *Rd,Konst*	Rd - Konstant, uppd. Z, N, C, Vii
0x13	AND *Rd,Ra*	Rd <= Rd AND Ra, uppd. Z, N, C, V
0x14	ANDI *Rd,Konst*ant	Rd <= Rd AND Konstant, uppd. Z, N
0x15	OR *Rd,Ra*	Rd <= Rd OR Ra, uppd. Z, N, C, V
0x16	ORI *Rd,Konst*ant	Rd <= Rd OR konstant, uppd. Z, N, C, V
0x17	PUSH *Rd*	DM(SP) <= Rd, SP++
0x18	POP Rd	Rd <= DM(SP+1), SP--
0x1B	MUL *Rd,Ra*	Rd <= Rd * Ra (unsigned)
0x1C	MULI *Rd,Konstant*	Rd <= Rd * Konstant (unsigned)
0x1D	MULS *Rd,Ra*	Rd <= Rd * Ra (signed)
0x1E	MULSI *Rd,Konstant*	Rd <= Rd * Konstant (signed)

0x1F	LSLS *Rd*	Rd <= Rd << 1 uppd. N, Z, C
0x20	LSRS *Rd*	Rd <= 1 >> Rd uppd. N, Z, C
0x21	PUSR	DM(SP) <= status_reg
0x22	POSR	status_reg <= DM(SP)
0x23	SUBR *Offset/Etikett*	Gå in i subrutin vid *Offset/Etikett*
0x24	RET	Returnera från en subrutin
0x25	PCR	Poppa stacken till programräknaren
0xFF	Odefinierad/EOF	Odefinierat i assemblern, vid uppladdning till datorn representerar det end-of-file.

Tabell 1: Fullständig instruktionsuppsättning.

2.3.2. Assembler

I projektet medföljer en assembler. Den går att återfinna om man utifrån rotmappen går till *./assembler/*. För att kunna använda assemblern måste den kompileras vilket görs via kommandot *make*. Programkoden för assemblern återfinns i *./assembler/src/*. En del ej assemblerade testprogram är inkluderade och återfinns under *./assembler/tests/*. Även spelet återfinns här i filen *asteroids.asm*.

Hjälp för alla flaggor i assemblern fås om man kör programmet med flaggan *-h*:

```
$ ./asm -h
Syntax: ./asm -i ../assembly.asm -o ./output.bin
-i inputFile (default=./example.asm)
-o outputFile (default=./out.bin)
-t if not used, adds a NOP to the interrupt vector
-m print instructions formatted to terminal
-d print debug information
```

Assemblern har stöd för etiketter och tre olika sätt att skriva konstanter. Exempel på detta ges i testet *labels_and_subroutines.asm*. Det finns 15 programmerbara register som benämns med A-P i assemblyprogrammen. Se kapitel *3.2.3. Registerfilen* för mer info om dessa. Inkluderat i assemblern är även en hel del syntaxkontroller som kan underlätta när man skriver ett program till datorn. Den lägger även till en End-of-File indikator i slutet av programmet åt programminnesladdaren, samt en no-operation (NOP) instruktion i början av filen om inte avbrottsvektorn ska hanteras av programmeraren, något man indikerar med *-t* flaggan. För att till exempel assemblera filen *labels_and_subroutines.asm* hade man då kört följande kommando:

```
$ ./asm -i tests/labels_and_subroutines.asm
```

Den binära filen återfinns då i samma mapp under namnet *out.bin*. Om något fel påträffas under assembleringen skriver assemblern ut detta. Vill man sedan ladda upp den binära filen följer man instruktionerna under *2.2.3. Programuppladdning*.

3. Hårdvaran

Följande kapitel återger detaljerade beskrivningar om hur datorns delar är implementerade i VHDL på FPGA:n. Vi börjar med en överblick av själva CPU:n och går sedan in på olika komponenter.

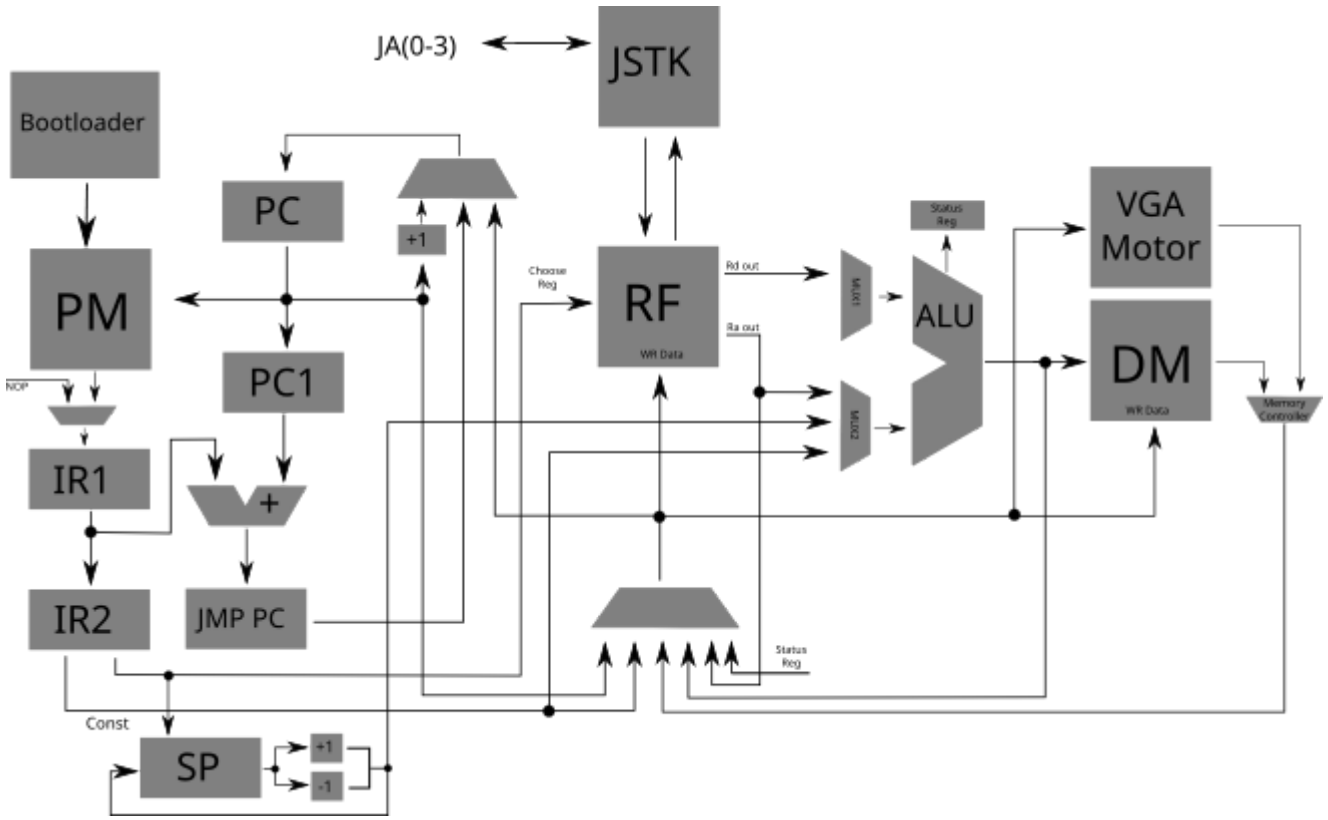

Bild 2: En övergripande bild av CPU:n.

3.1. Central processing unit

Datorn är en tvåstegs pipelinead CPU inspirerad av en som förekommer på en liknande kurs från University of Washington[2]. Den har stöd för UART skrivning och VGA grafik. CPU:n är implementerad i ./src/pipeCPU.vhd.

3.1.1 Pipeline

De två stegen är först ett fetch steg(IR1) och sedan ett decode steg(IR2). IR1 används för hopp och branch instruktioner medans IR2 används för ALU, registerfilen samt stackpekaren. Vid hopp och subrutiner läser IR1 ej in från programminnet utan lägger in specifika instruktioner manuellt; Vid en hoppinstruktion läggs en NOP in, en så kallad hopp-nop, och vid ett subrutinshopp läggs en PUSR in för att spara statusregistret. Branchinstruktioner använder sig också av hopp-nop, oavsett om de branchar eller inte, då flaggorna inte har hunnit sättas än. Detta gör att det ibland blir onödiga NOPs men

[2] http://www.columbia.edu/~yc3096/cad/ee477Final.pdf

alternativet var att lägga till NOPs manuellt i programkoden eller att designa om CPU:n från grunden.

3.1.2 Programräknare

CPU:n har även tre programräknarregister. Det första heter PC och innehåller raden i programminnet som ska läsas ut. Efter PC kommer PC1. PC1 används som ett mellansteg för att värdet ska kunna användas senare och tar alltid PCs värde en klockcykel senare. Det tredje registret kallas JMP PC. JMP PC används för hopp i programminnet då den adderar en konstant från IR2 till det sparade värdet i PC1. Om ett hopp ska utföras skrivs PC:s värde över med JMP PC, annars ignoreras det.

3.1.3 Stackpekare

Stackpekaren håller koll på vart i minnet stacken har sin topp. Den räknas upp/ner automatiskt vid vissa instruktioner, t.ex vid pop, push, subr, m.fl. Stackpekaren pekar på den adress i minnet som man skriver till nästa gång då något ska skrivas till stacken. Detta medför att när något skrivs kan det ske utan problem, men när något ska läsas måste stackpekaren flyttas innan läsningen. Eftersom stackpekaren inte hinner uppdateras innan inläsningen sker, korrigeras värdet i ALU:n för de instruktioner som behöver det för att läsning ska ske korrekt.

3.1.4 Databuss

Databussen kopplar ihop flera olika komponenter, nämligen: IR2, PC, DM, ALU, RF och statusregistret. Vilken signal som går igenom bestäms av en MUX som använder OP-koderna från IR2 för att välja. Bussens huvudsakliga syfte är att skriva och läsa till/från register och dataminnet/spriteminnet men används även för att spara statusregistret och programräknare vid subrutinshopp.

3.1.5 Avbrott

Avbrott fungerar med hjälp av subrutinshopp. Vi har en signal i CPU:n som heter *interrupt*. När den ställs hög så kommer exekveringen hoppa till position noll i programminnet (det som även kallas *avbrottsvektor* i denna rapport). Vi sparar även några avbrottsspecifika flaggor i CPU:n för att hantera specialfallet subrutin vid avbrott. Dessa används dels för att se till att vi inte hoppar till någon offset som annars är kutym för våra hopp, utan att vi hoppar till positionen noll. Samt dels att den returadress vi sparar för subrutinen blir korrekt. När vi väl landat i avbrottsvektorn antas att programmeraren därifrån hoppar till någon plats där själva avbrottet hanteras. Viktigt är att man inte hoppar till en subrutin, då man går ur själva avbrottet genom att använda instruktionen *ret*. Under tiden som ett avbrott hanteras kan inte nya avbrott tas emot. Själva *interrupt* signalen antas sättas låg av det som sätter den hög.

3.2. Minnen

Datorn har tre huvudsakliga minnen. Programminnet, dataminnet och registerfilen. Rent hårdvarumässigt är dessa minnen väldigt lika varandra, men deras användningsområden skiljer sig. En kort beskrivning av samtliga följer nedan.

3.2.1. Programminne

Programminnet är det minne som innehåller programmet som ska exekveras av datorn. Implementationen återfinns från rotmappen i *./src/MEM/PROG_MEM.vhd*. Detta minne är 1024 adresser stort och 32 bitar brett, varje rad innehåller alltså en instruktion. Då vi tillåter programmering av datorn via UART (åtminstone uppladdning av program) är detta minne skrivbart, men bara av programminnesladdaren. Därmed är enda gången något skrivs till programminnet också innan ett program har laddats in. Detaljer om hur uppladdningen sker kan läsas i kapitel *3.3. Programminnesladdaren*. Skrivning till minnet sker synkront medan läsning sker asynkront. Följande bild ger en översyn av programminnets signaler.

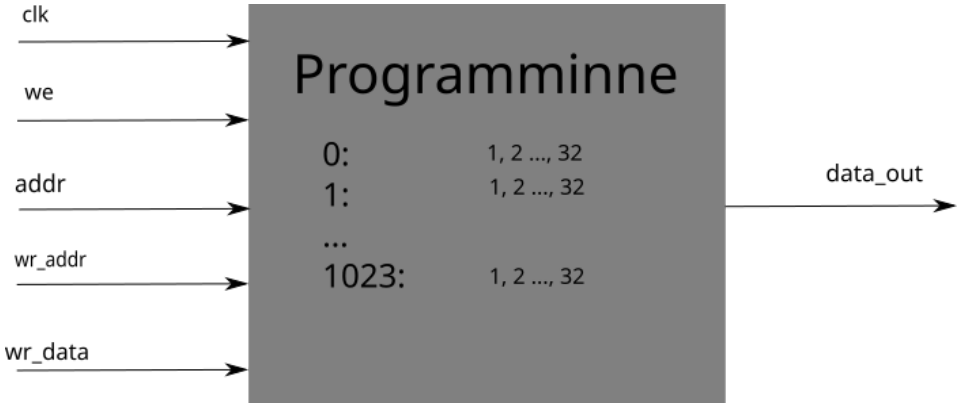

Bild 3: Programminnets olika signaler.

De signaler som inte har med skrivning att göra används direkt i CPU:n. Det vill säga *addr* och *data_out*. Allt som däremot har med skrivning att göra är kopplat mot programminnesladdaren. Värt att notera är att första adressen i minnet är reserverat till avbrottsvektorn. När ett avbrott sker landar man på adress 0 för att förhoppningsvis tas vidare till en subrutin som hanterar avbrottet. Innehållet i programminnet beskrivs i tabell 2 nedan.

Adresser	**Kommentar**
0	Avbrottsvektor
1 - 1023	Maskininstruktioner

Tabell 2: Adresserna i programminnet.

3.2.2. Dataminne

Dataminnet är mer intressant för programmeraren av datorn. Implementationen återfinns från rotmappen i *./src/MEM/DATA_MEM.vhd*. Detta minne går att nå via CPU:n med instruktionerna som i *tabell 1* (kapitel *2.3.1. Instruktioner*) refererar till DM i kommentaren. Minnet används som ett *Random Access Memory* till det program som exekveras. Minnet läser asynkront men skriver synkront. Följande bild ger en översyn över dataminnets signaler.

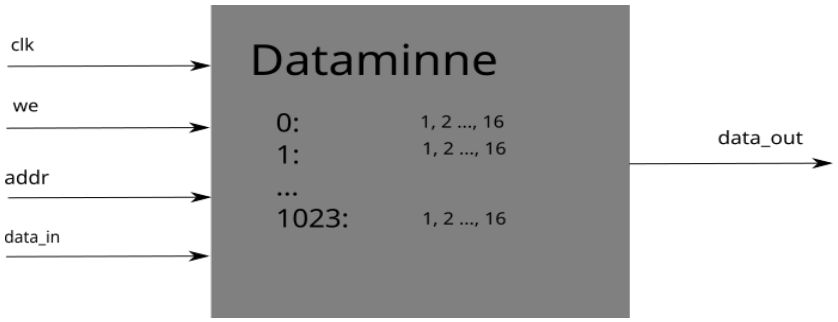

Bild 4: Dataminnets olika signaler.

Innehållet i dataminnet beskrivs i tabell 3 nedan. Då vi vill kunna nå spriteminnet via CPU:n för att kunna avläsa metadata för sprites samt ändra denna metadata minnesmappar vi en del av dataminnet till spriteminnet. Mer om detta går att läsa i kapitlet *Minnesmappning* nedan.

Adresser	**Kommentar**
0 - 255	Stack
256 - 1023	Programmerbart minne
64512 - 65535	Mappat till spriteminnet

Tabell 3: Adresserna i dataminnet.

Minnesmappning

För att kunna komma åt spriteminnet från CPU:n är en del av dataminnet minnesmappat till spriteminnet. Detta innebär att när man försöker nå eller skriva till någonting över adressen 64511 i dataminnet mappas man egentligen till spriteminnet. Detta görs via en adresskontrollerare som beskrivs i följande bild.

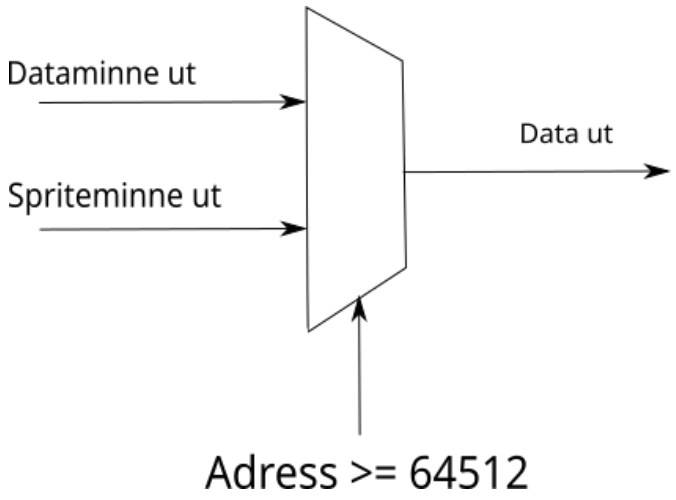

Bild 5: Minneskontrolleraren.

Det vill säga; vad som skickas till *Data ut* beror egentligen på adressen som används. På samma sätt fungerar det vid skrivning till minnet, vilket då tillåter oss att göra ändringar i spriteminnet också.

3.2.3. Registerfilen

Registerfilen innehåller samtliga 16 register och återfinns från rotmappen i *./src/MEM/REG_FILE.vhd*. Varje register är 16 bitar brett. Skrivning till registerfilen sker synkront och läsning asynkront. Det finns två saker som måste tas till hänsyn när man jobbar med registerfilen. Dels är det att 14 stycken utav dessa 16 är fria att användas av programmeraren. De sista två registren används av joysticken. Mer information om det återfinns i kapitlet *3.5. Joystick*. Det andra är att registerfilen också tar in signaler för att kunna mata ut värden på 7-segment displayen som finns på NEXYS3-kortet. Följande bild ger en översyn över registerfilens signaler.

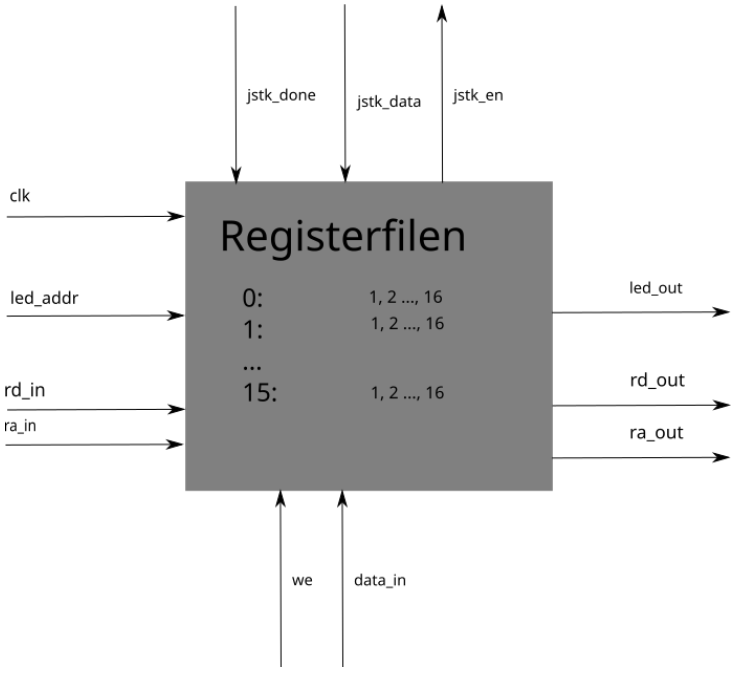

Bild 6: Översyn över registerfilens signaler.

Registerfilen tar in vilket register som ska visas på *led_addr* signalen, och skriver sedan ut registrets innehåll på *led_out* signalen. Detta minne har också två ut-data signaler. Beroende på vilket register (0-15) som vill läsas av i *rd_in* och *ra_in* signalerna kommer motsvarande registers innehåll skrivas ut till *rd_out* och *ra_out*. Däremot kan man bara skriva *data_in* värdet till registret som anges i *rd_in* då *we* signalen är hög. Se tabell 4 nedan för beskrivning av innehållet i registerfilen.

Adresser	**Kommentar**
0 - 13	Programmerbara register
14	Joystick data
15	Joystick data

Tabell 4: Adresserna i registerfilen.

3.3. Programminnesladdaren (bootloader)

För att kunna ladda in program till programminnet används en komponent som här benämns *programminnesladdare* (även känt som *bootloader*). Denna komponent återfinns från rotmappen i *./src/MEM/PROG_LOADER.vhd*. Följande bild beskriver komponenten i grova drag.

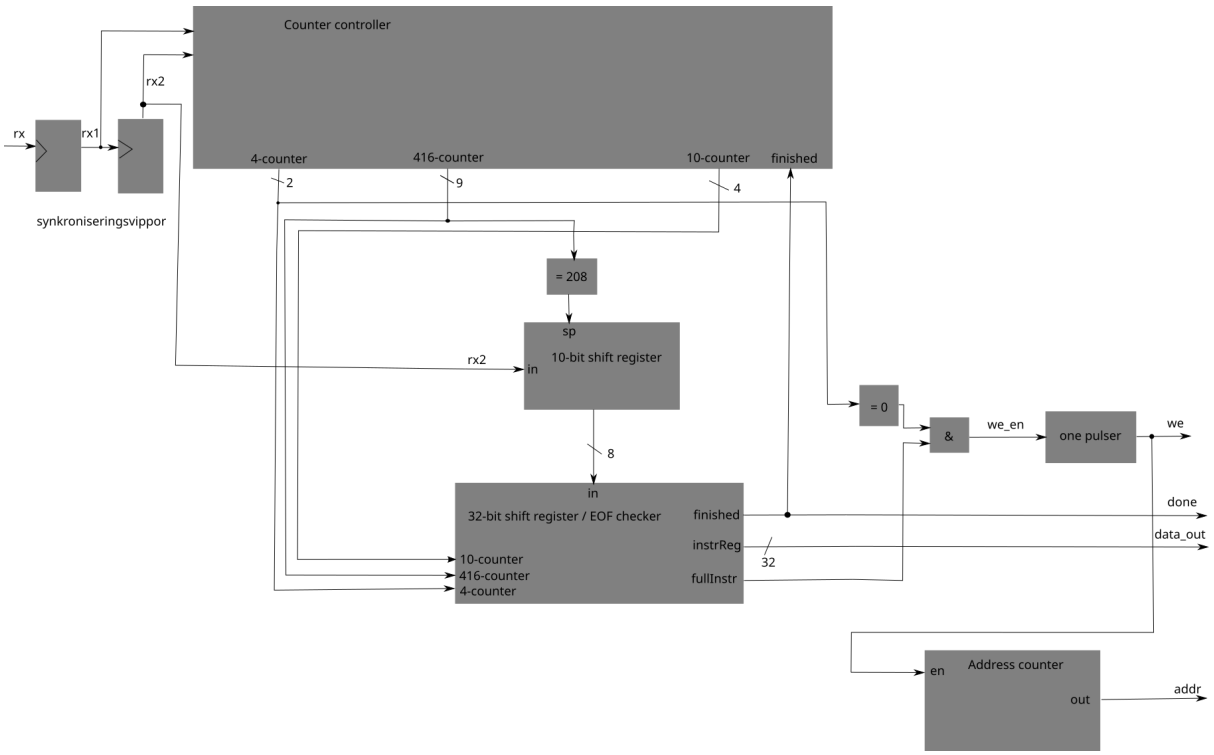

Bild 7: En aningen simplifierad version av programminnesladdaren.

Denna laddare väntar vid uppstart av datorn på att ett program skickas till NEXYS3-kortet via UART. När den ser att en startbit infinner sig på insignalen börjar 416 räknaren att räkna upp. Då vi kommit till 208 avläser vi biten och shiftar in den i ett 10-bitars shiftregister. När vi läst av 10 bitar, inklusive stoppbiten, sparar vi den mottagna byten till ett register som ska innehålla hela instruktionen. Då nästa startbit infinner sig gör vi samma procedur igen tills vi mottagit alla 32 bitar för den fullständiga instruktionen. När hela instruktionen är mottagen skriver vi denna till programminnet och ökar adress-räknaren till nästkommande skrivning. Vi fortsätter läsa av UART på samma sätt tills vi får en instruktion vars OP-kod är 0xFF. Detta indikerar till programminnesladdaren att vi har kommit till slutet av programmet (någonting som assemblern lägger till i slutet av varje assemblerat program). Vi kommer då sätta adressen *done* hög vilket säger till CPU:n att den har tillåtelse att börja exekvera det laddade programmet.

3.4. Arithmetic-logic unit

Följande bild 8 visar en övergripande bild av ALU:n.

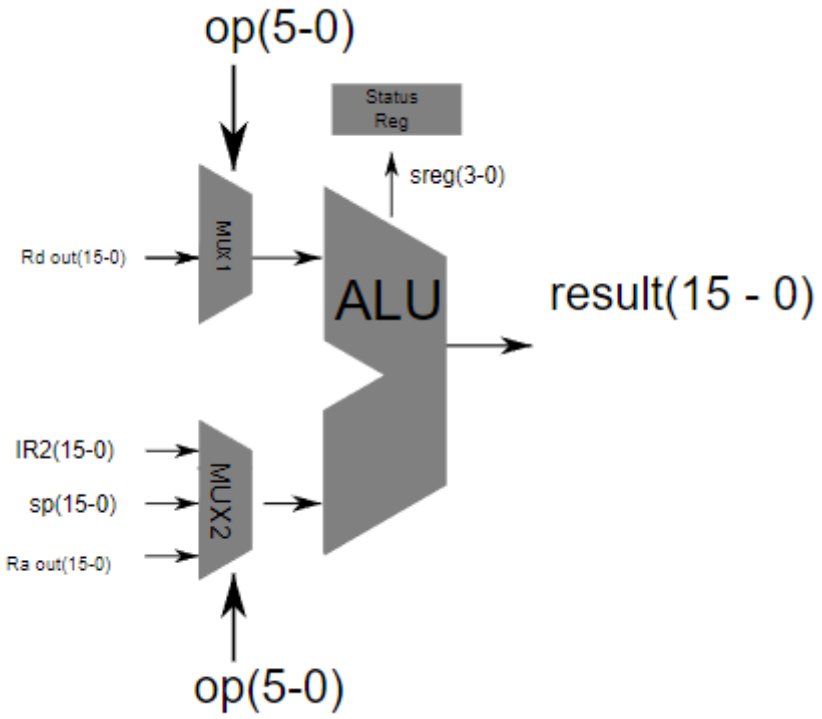

Bild 8: En övergripande bild av ALU:s olika komponenter.

Inmatning till ALU:n styrs av två muxar som väljer sin utdata utifrån OP-koden från IR2 registeret. De olika ingångarna till MUX2 är värdet i ett register, en konstant från IR2 eller stackpekaren, dessa är beskrivna i kapitlet *3.1.4. Databuss.* Tillsammans med MUX2 finns MUX1 som får indata utifrån ett valt register. De instruktioner som inte kräver någon hantering av ALU:n, till exempel LDI, skickas genom ALU:n utan att ändras. Det är MUX2 som register IR2 är kopplat till därav är det inmatningen till MUX2 som skickas rakt igenom ALU:n.

Instruktioner som hanteras av ALU:n kan även sätta dom följande fyra olika flaggorna:
Z - Sätts då ett resultat blir noll.
V - Sätts då en operation resulterade i overflow (bitarna går från hexadecimalt x”0111” -> x”1000” eller x”0111”->x”1000”).
N - Sätts då den högsta biten är satt som indikerar att talet är negativt.
C - Sätts då ett resultat ger en carry, alltså då bitarna går x”1111” -> x”0000” eller tvärtom
De instruktioner som påverkar flaggor och hanteras av ALU:n ses under kapitel *2.3. Instruktionsuppsättning*. Status registret används av CPU:n för att utföra hopp och finns beskrivet under kapitel *3.1. Central processing unit*.

3.5. Joystick

Joysticken är en PMODjstk från Diligent. Joysticken kommunicerar med FPGA:n via SPI protokollet som fungerar genom att det skickas 40 bitar (eller 5 bytes) från och till joysticken parallellt. Dessa bitar genereras då SCLK på joysticken blir hög och data läses då SCLK går låg. MOSI är data till joysticken som styr de 2 LED:s som finns på den. MISO är data från joysticken och som kan ses i bild 9 kommer de mest signifikanta bitarna i varje byte först. För x-värdet kommer först bit 7 till 0 och i den andra byten kommer bit 9 och 8 på de två minst signifikanta bitarna i byten. Sedan kommer y-värdet på samma sätt fast för byte 3 och 4, sist kommer byte 5 med knapparna på de tre lägsta bitarna.

	B1.7	...	B1.1	B1.0	B2.7	...	B2.1	B2.0	B3.7	...	B3.0	B4.7	...	B4.1	B4.0	B5.7	...	B5.2	B5.1	B5.0
MOSI	1	0...0	L2	L1	0	0...0	0	0	0	0...0	0	0	0...0	0	0	0	0...0	0	0	0
MISO	x7	...	x1	x0	0	0...0	x9	x8	y7	...	y0	0	0...0	y9	y8	0	0...0	K2	K1	K0

Bild 9: Bitarna från/till joysticken bild[3]

Joystickkomponenten för FPGA:n har utdatan *data_out* och *done* och indatan *enable*. *Enable* styr när kommunikation med joysticken ska startas, *data_out* är bitarna given av joysticken i form av 22 bitar uppdelat enligt x&y&btns. *Done* sätts när all data från joysticken har tagits emot av FPGA:n. Samtidigt skickar FPGA:n till joysticken att LED lamporna ska blinka varje gång data har skickats eller tagits emot. *Done* ligger hög tills att *enable* går låg, nu kan *enable* sättas hög igen och en ny överföring kan börja.

FPGA:n styr kommunikationen internt och skickar sedan ut resultatet från joysticken på register 14 och 15. Registrena är uppdelade enligt tabell 5.

bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15	en	-	-	btn	btn	btn	x9	x8	x7	x6	x5	x4	x3	x2	x1	x0
14	-	-	-	-	-	-	y9	y8	y7	y6	y5	y4	y3	y2	y1	y0

[3] *Petter Källström, Pmodjstk manual på TSEA83 kurshemsida.*

Tabell 5: En står för joystick enable, btn för de olika knapparna och x,y koordinaten samt vilken bit i koordinaten.

Hårdvarukonstruktionen följer enligt bild 10 där allt börjar med att SCLK COUNTER får *enable* och börjar räkna. När räknaren når utsatt tal räknas 40 counter up. När *fall_edge_onepulse* går hög kommer en läsning av MISO ske och då *rise_edge_onepuls* sker kommer en skrivning av MOSI ske. SCLK styrs av *SCLK_counter* och är hög under varannan räkning så att det matas 2000 cykler då *sclk* är låg och sedan 2000 cykler där den är hög. Vilka bitar som ska skiftas in och vad de ska tilldelas styrs av bit controller, bit controller skiftar in/ut alla bitar från joysticken. Vid varje läsning kommer bitarna från MISO skiftas in till *data_out* som sedan går ut ur komponenten med enligt x&y&btns. Där x,y är joystickens x respektive y värde och btns är joystickens knappar. Vad som skrivs till joysticken bestäms även här och kommer styra de olika LED lamporna. När 40 räknaren har nått 40 är läsning och skrivning klar, då sätts en *done* flagga hög för att indikera att datat som har kommit på *data_out* är fullständigt.

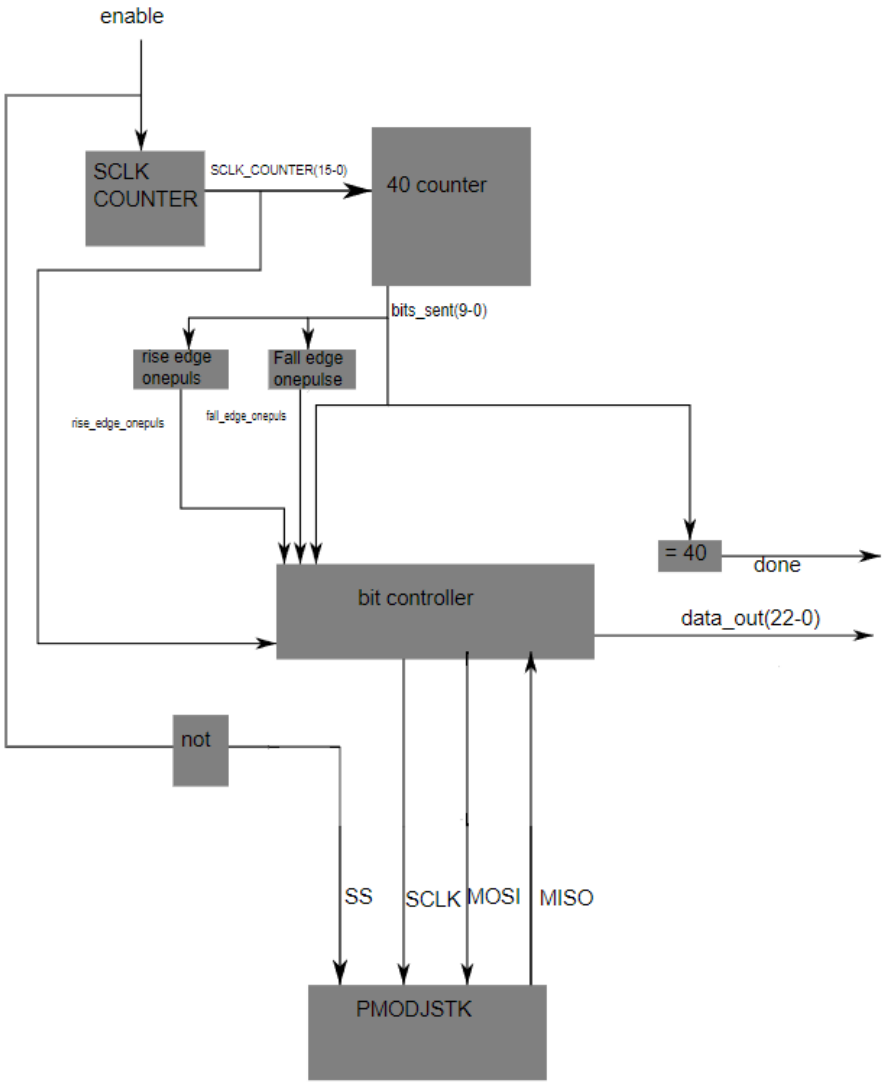

Bild 10: Joystickens övergripliga konstruktion i FPGA datorn

3.6. Grafik

Följande bild 11 visar en övergripande vy av VGA-motorn.

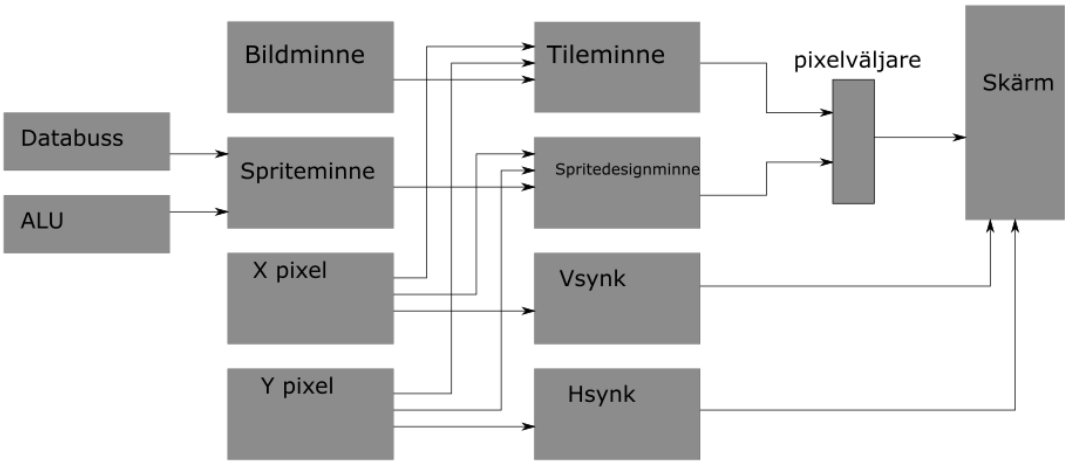

Bild 11: En övergripande vy av VGA motorn.

Det finns stöd för användning av både sprites och tiles i datorkonstruktionen. Bakgrunden består av tiles. För att minska storleken på minnena används pixlar som är 8 fysiska pixlar stora på skärmen. Då skärmen har en upplösning på 1280x1024 fysiska pixlar ger det därmed att det är 160x128 upplösning på det som visas. Vsync och Hsync är tajmat enligt instruktioner på den grafiska laborationen[4] som genomfördes i kursen TSEA83.

Bildminnet lagrar var och vilken tile som ritas ut på bakgrunden, medan tileminnet lagrar hur tilesen ser ut. Tileminnet lagrar tiles som är 8x8 pixlar stora. Det går inte att ändra bildminnet från assembly-koden, därmed är bakgrunden hårdkodad i hårdvaran. För att välja vilken tilepixel som ska användas finns en process i VGA-motorn som tar in Xpixel, Ypixel samt Bildminnet. Denna process beräknar därefter vilken sprite i Bildminnet som ska läsas och skickar denna 4 bitars information till pixelväljaren.

Spriteminnet kan ändras. De högsta adresserna i dataminnet är kopplade till spriteminnet, därmed ändras spriteminnet på samma sätt som dataminnet. Dessa addresser är hex FC00 till hex FC1F. När “write enable” (we) är satt skrivs bitarna från bussen till specificerad plats i spriteminnet. Varje sprite använder två minnesplatser i spriteminnet där den första innehåller x-koordinaten och den andra innehåller sprite-typ samt y-koordinat. Spelet har plats för 10 sprites, det finns dock utrymme att utöka till 15 sprites. Informationen med spritesens utseende är lagrade i spriteDesignMinne. Vardera sprite har ett kombinatoriskt nät som ger vilken pixel som ska ritas ut för vardera x- och y-koordinat. Detta använder sig av både x-, y-pixel, samt spriteMinne och spriteDesignMinne. Alla sprites pixel-data skickas därefter till pixelväljaren.

Pixelväljaren tar in den pixel som tilesen ger samt pixeln från vardera sprite. Pixelväljaren börjar med att kontrollera om den första spriten ger en nolla (ingen information) och antingen väljer denna pixel eller går vidare till den andra spritens pixel data. Om ingen sprite ska ritas på denna position så skrivs bakgrunden ut, vilket är pixeln från tile-processen. Därefter

[4] https://www.isy.liu.se/edu/kurs/TSEA83/pdf/lab_vga.pdf

mappas den resulterade pixelns data med olika olika färger. Pixeldatan är 4 bitar vilket resulterar i att maximalt 16 olika färger kan användas.

Då pixelväljaren läser alla sprites pixlar för respektive koordinat kollar den även om det sker kollision. Den kollar om spriten på första platsen (rymdskeppet) försöker rita ut en pixel samtidigt som en annan sprite, och om så fallet har en kollision skett och en process skriver till sista platsen i spriteminnet.

3.7. Leddriver

Leddrivern används för att driva 7-segment displayen som används i spelet för att visa ens nuvara poäng. Den är även användbar vid felsökning av program genom att tillåta skrivning från CPU:n till *verkligheten* genom register N i registerfilen. Hårdvarumässigt är den inte förändrad från labben. Man tar in ett värde som man vill visa och en räknare används för att egentligen loopa över hela talet som ska skrivas ut på displayen. Detta då en styrsignal används för att välja vilken av siffrorna på displayen man ska skriva till.

4. Slutsatser

Det mesta i projektet uppnådde det vi planerade för i designspecifikationen. Framförallt så nådde vi de krav som var satta på hårdvaran. Det vill säga kraven med att använda en pipelinad processor, UART, joystick och grafisk output via VGA. Ytterligare ett moment som fungerade bra med projektet var uppdelningen av arbetet där vardera gruppmedlem arbetade med separata delar parallellt. Detta resulterade i att projektet blev klart i tid och till stor del utan stress mot slutet.

Det fanns flera oväntade moment med projektet. Ett av de största var att vi hade tajming problem på grund av långa kritiska vägar i kombinatoriska nät. Detta löstes genom att optimera bort ett par av dessa vägar vilket gjorde det möjligt att sänka klockan till 50 MHz. Ett annat problem var tidsbrist vilket ledde till att vi fick tänka om över vilket spel vi ville skapa. Det var bland annat planerat att spelet skulle inkludera skjutande av asteroider. Men då just spelprogrammering inte kändes som fokus i kursen proriterades detta bort.

Det finns flera lärdomar under detta projekt. En av dessa är att det ofta är värt att investera tid i mjukvara eller struktur som förenklar arbetet senare. Exempel på detta är UART som med god marginal tog in på den tid det tog att implementera senare i utvecklingen av spelet. Ytterligare exempel på detta är kompilatorn som omvandlar assembly- till maskinkod. För mer hårdvaruspecifika lärdomar är det att ha längsta vägen i åtanke under alla utvecklingsmoment. Det har även varit lärorikt att få djupare förståelse i hur filerna utöver VHDL-koden är uppbyggda. Dessa filer är exempelvis make- samt .do-filer.