

XYZ-Pilot

1 Inledning

Den här tekniska rapporten behandlar ett projektarbete i kursen TSEA83: Datorkonstruktion på Linköpings Universitet.

1.1 Bakgrund

Kursen TSEA83 läses av studenter på civilingenjörsprogrammet datateknik och avser att ge studenter kunskap om hur datorer fungerar i dess minsta beståndsdelar. Förutom projektarbetet består kursen av en förberedande laborationsdel som lägger behandlar de grundläggande kunskaperna i processorkonstruktion och VHDL-kod.

1.2 Syfte

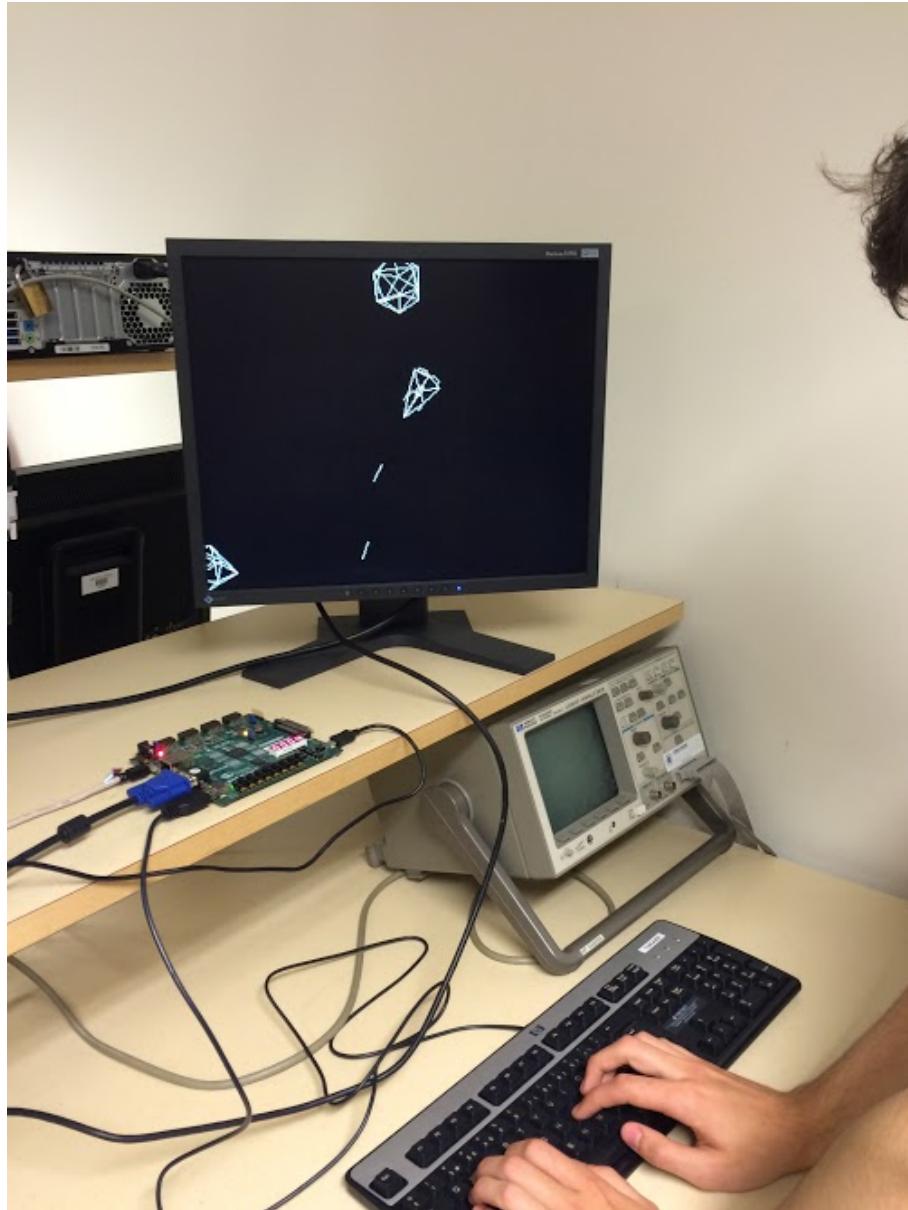
Denna rapport syftar att ge en förståelse för vår konstruktions arkitektur och tekniska detaljer samt vilka lösningar vi tillämpat för att nå vår färdiga produkt. Läsaren ska efter att ha tagit del av rapporten ha förståelse för apparatens olika komponenter och de grundläggande teorier som krävs för detta. Rapporten förklrar också hur apparaten används för att fungera.

1.3 Källor

TODO

2 Apparaten

Vår apparat använder ett tangentbord för input och visar sedan spelet på en skärm med hjälp av VGA. Programkoden, som innehåller instruktioner för processorn, laddas in via UART. Det program vi valt att realisera för apparaten är ett rymdspel som går ut på att skjuta sönder asteroider för poäng. Poängen man får syns hexadecimalt på 7-segments-displayen. Figur 1 visar en bild som beskriver hur konstruktionen ser ut.

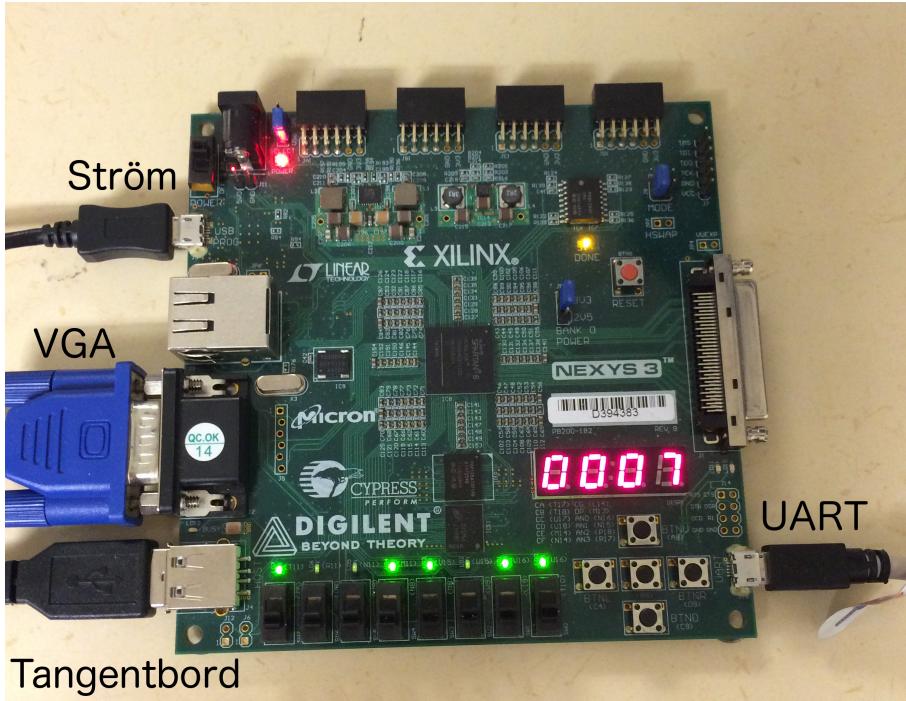


Figur 1: Med hjälp av tangentbordet är det möjligt att styra skeppet på skärmen och skjuta asteroider

Konstruktionen byggs genom att koppla in VGA från skärmen, UART och strömsladd från valfri USB-port på en dator samt en tangentbordssladd i FPGAn enligt figur 2. För att starta programmet behöver du vara inne i mappen XYZ-Pilot/assemblers och skriva in kommandot:

```
./assemble.sh game.asm
```

Detta skript programmerar hårdvaran och laddar in mjukvaran via UART. Inom kort borde skeppet synas på skärmen och tangentbordet kan användas för att spela.



Figur 2: Bilden beskriver de sladdar som ska kopplas in i kortet

3 Teori

3.1 3D-grafik

3D-objekt representeras oftast som en mängd vektorer som beskriver hörnpunkter i objektet, linjer som binder ihop dessa vektorer och ytor som binder ihop dessa linjer. I vårt projekt finns inga ytor och för enkelhetens skull beskriver vi inte linjer som två index till punkter utan direkt som två vektorer.

I moderna GPUer används 4-dimensionella vektorer istället för 3D vilket möjliggör rotation, skalning och translation av en vektor i en enda matrismultiplikation. Från början tänkte vi också göra det vilket är anledningen till att våra vektorer är 4-dimensionella.

3.2 Bresenhams linjealgoritm

Bresenhams linjealgoritm utvecklades på IBM under tidigt 1960-tal och hade den stora fördelen att dess implementation endast kräver heltalsaddition, subtraktion och shiftande av bitar. Eftersom algoritmen är så pass snabb och simpel finns den realiseras i många existerande grafikprocessorer och vilket är varför vi valt att använda den i vårt projekt.

Algoritmen bygger på att räkna ut felet som uppstår jämfört motstående axel när algoritmen rör sig från början till slut. Blir felet tillräckligt stort korrigeras det och felet minskar då med en enhet. Felet ökar med absolutvärdet av kvoten mellan komponenternas längd för varje passerad pixel i algoritmens ursprungliga form, som då kräver decimaltalsaritmetik. En implementation av detta följer nedan.

```

func drawLine(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    e = -1
    de = abs(dy / dx)

    x = x0
    y = y0

    while x < x1 do
        fill(x, y)
        e += de
        if e >= 0 then
            y += 1
            e -= 1
        endif
        x += 1
    endwhile
endfunc

```

I och med att det är mer kostsamt och ineffektivt att använda decimaltalsaritmetik än heltalsaritmetik var en viktig egenskap med just Bresenhams algoritm att den går att skriva om för detta. En annan begränsning med algoritmen i dess grundutförande är att den endast fungerar för linjer som rör sig med riktningen av den sista oktanten och därfor ser den algoritmen som är implementerad i vår hårdvara aningen mer komplicerad ut än grundfallet. Vad som sker i hårdvaran går att göra enklare genom att använda transformering av in- och utdata beroende på vilken oktant linjen rör sig i istället för att använda if-satser.

```

func drawLine(x0, y0, x1, y1)
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)
    x = 0
    y = 0
    i = 0

    x_incr = 1 if (x1 - x0) > 0 else -1
    y_incr = 1 if (y1 - y0) > 0 else -1

    if dx > dy then
        len = dx
        D = 2 * dy - dx
    else
        len = dy
        D = 2 * dx - dy
    endif

    while i < len do
        fill(x, y)
        if D > 0 then
            if dx > dy then
                y += y_incr
                D -= 2 * dx
            else
                x += x_incr
                D -= 2 * dy
            endif
        endif

        if dx > dy then
            x += x_incr
            D += 2 * dy
        else
            y += y_incr
            D += 2 * dx
        endif

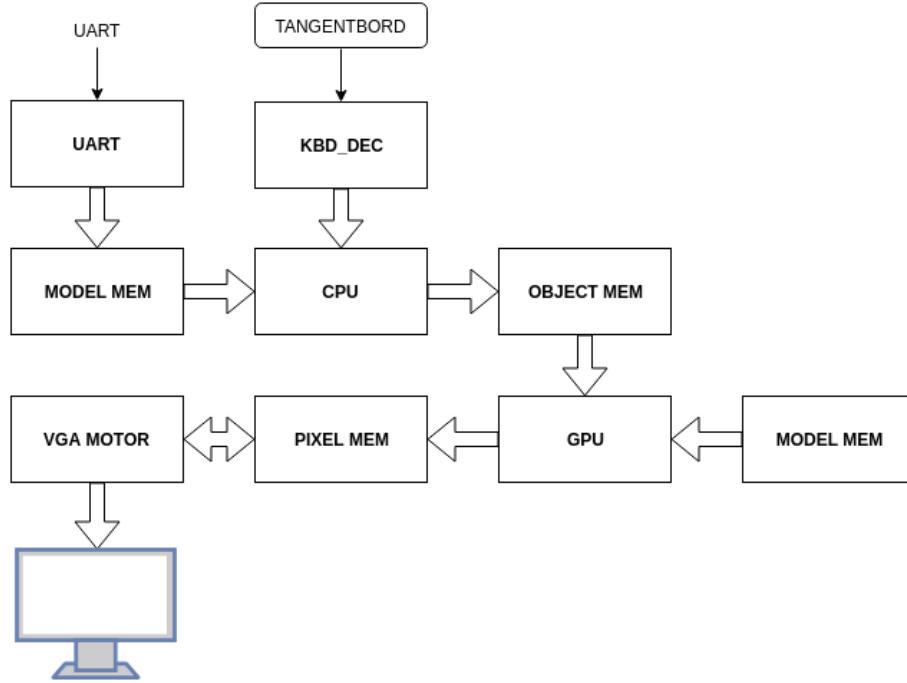
        i += 1
    endwhile

    fill(x, y)
endfunc

```

4 Hårdvaran

Vår konstruktion består i grunden av en hårdvaruaccelererad grafikprocessor för 3-dimensionella objekt och modeller som utökas med en centralprocessor som kan manipulera objekten som renderas och även utföra aritmetik på tal och vektorer. Programmet som körs på apparaten programmeras vid start via UART. En översikt av konstruktionen ses i blockschemat i figur 3.



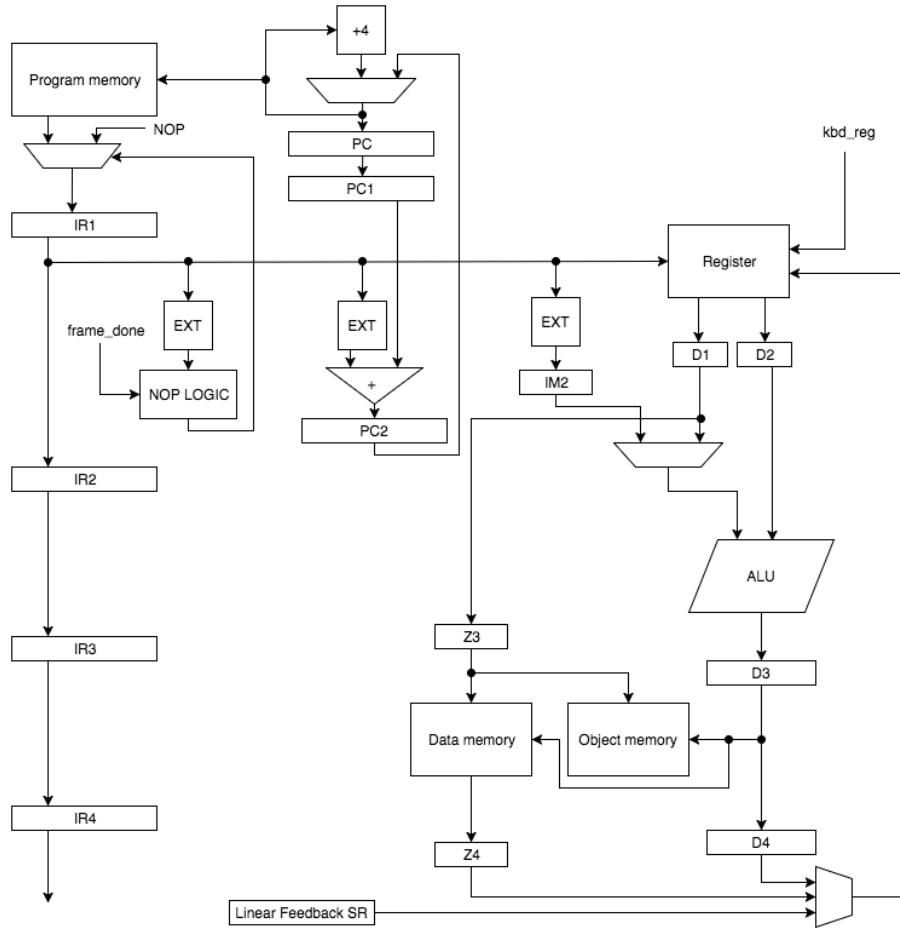
Figur 3: Översiktligt blockschema över konstruktionen

4.1 Centralprocessor

För att manipulera det som ritas med hjälp av grafikenheten används en processor som med hjälp av instruktioner i programminnet bestämmer de värden som finns i objektminnet. Processorn, som är naivt pipelinad, läser av instruktioner från programminnet och utför beräkningar som får spelet att uppdateras. Alla minnen och register är 64 bitar breda för att vektorer ska få plats på en adress. Ett översiktligt blockschema finns i Figur 4.

För att spara programminne används extra hårdvara som automatiskt kör tre NOPar mellan varje instruktion. Denna krets ger även möjligheten för processorn att ”vänta” på att VGA-motorn ska rendera klart en bild. En av processorns instruktioner får NOP-logiken att ständigt skapa NOPar i processorn, något som kommer att pågå tills VGA-motorn skickar en signal när den har renderat en bild.

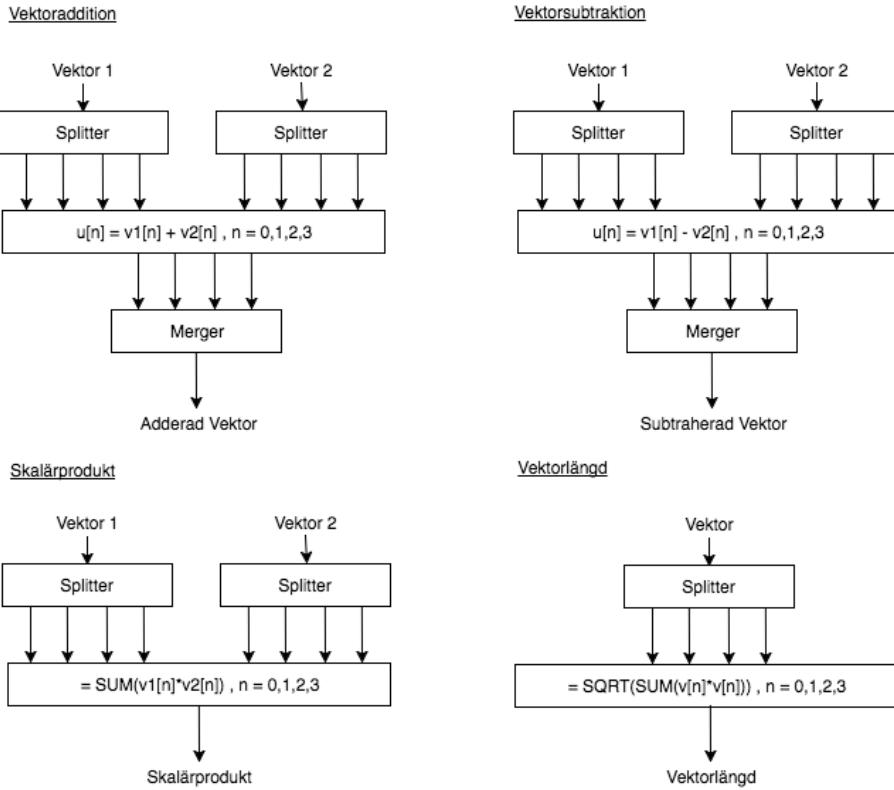
Det sista registret bland registerfilerna används för att spara information om vilka knappar på tangentbordet är nedtryckta. Varje gång det finns en instruktion i IM4 som inte skriver till något register passar processorn på att uppdatera detta register från det register som finns i tangentbordsavkodaren. Detta innebär att de nedtryckta tangenterna ständigt hålls uppdaterade eftersom majoriteten av instruktionerna som körs är NOPar.



Figur 4: Överskådligt blockschema över kretsen

4.2 Vektorinstruktioner

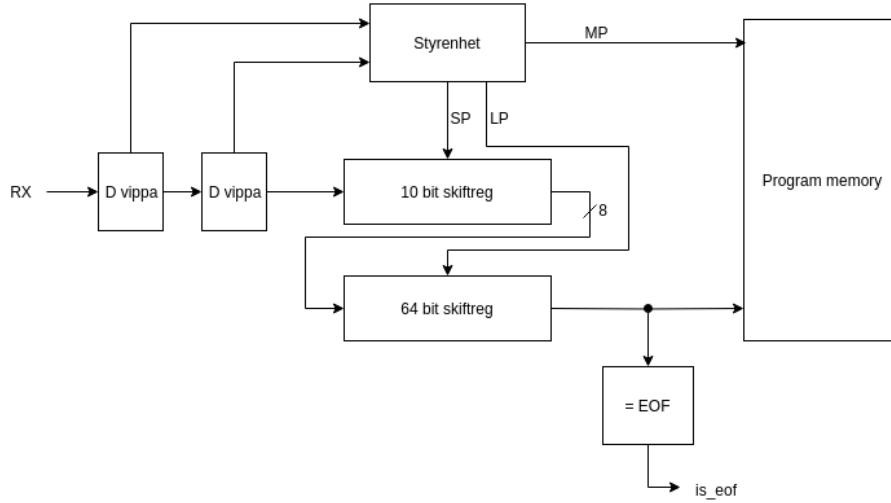
Förutom vanliga instruktioner har processorn även stöd för instruktioner som hanterar vektorer. Bland annat används addition och subtraktion av vektorer, samt skalärprodukt och vektorlängd. Detta är möjligt tack vare separata komponenter som används i ALUn. Innan en vektor ska användas vid beräkning delas den först upp i element som sedan används på olika sätt beroende på vilken instruktion som körs. När beräkningar är gjorde sätts elementen ihop igen för en resulterande vektor. En enkel illustration i figur 5 beskriver hur detta fungerar. Notera att hårdvara för kvadrering krävs för beräkning av vektorlängd. Algoritmen för denna vhdl-kod har tagits från vhdlguru.blogspot.se.



Figur 5: Illustration över hur vektorinstruktioner fungerar

4.3 UART

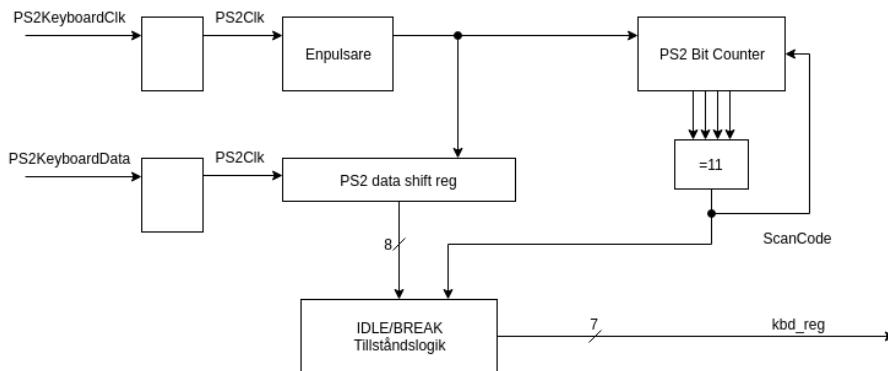
För att snabbt och enkelt ladda in programkod i programminnet används UART. Eftersom varje instruktion i programminnet är 64 bitar lång krävs det alltså 8 bytes för att ladda in en hel instruktion. Detta görs genom två olika skiftregister, ett 10-bitars skiftregister som innehåller de 10 senaste bitarna och ett 64-bitars skiftregister som innehåller de 8 senaste bytes som laddats in. När 8 bytes laddats in (en hel instruktion) i UART-komponenten laddas dessa in i programminnet samtidigt som minnets nuvarande adress inkrementeras med ett. Allt detta styrs av en styrenhet bestående av tre räknare, en som räknar varje klockpuls för varje bit, en som räknar alla bitar för varje byte och en som räknar alla bitar för varje instruktion. När dessa räknare nått sitt maxvärde sätts respektive laddpulser (sp, lp, mp) som får skiftregistren och programminnet att uppdateras. Den sista "instruktionen" består enbart av ettor och får UART-komponenten att sluta arbeta. Figur 6 visar ett enkelt blockschema som beskriver kretsen.



Figur 6: Blockschema för UART kretsen

4.4 Tangentbordsavkodare

För att använda ett tangentbord används en komponent som hanterar de PS/2-signaler som uppstår vid tangentnedtryck. Kretsen beskrivs i figur 7. En enkel tillståndsmaskin används för att ta reda på vilken tangent som är nedtryckt. Tillståndsmaskinen består av två tillstånd, IDLE och BREAK. Vid IDLE undersöks om nuvarande byte i shiftregistret stämmer med den kod för relevanta tangenter (tangenten sätts som nedtryckt) eller om koden är lika med den som föregår den sekvens som uppstår när en tangent släpps (tillståndet sätts till BREAK). Vid BREAK undersöks om nuvarande byte i shiftregistret stämmer med relevanta tangenter varvid tangenten sätts som uppsläppt. Tillståndet övergår alltid till IDLE vid BREAK.



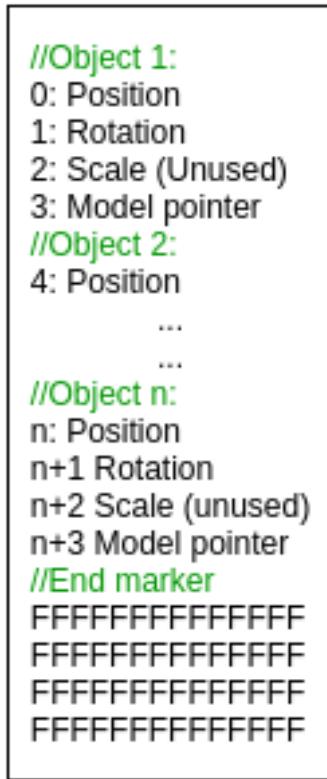
Figur 7: Blockschema för tangentbordsavkodaren

4.5 Objektminne

För att rendera modellerna på skärmen lagras olika objekt med specificerad modell, position och rotation. Objekten i minnet kan manipuleras med hjälp av processorinstruktioner och på så sätt ändra dess antal och egenskaper.

Formatet består av 4 ord per objekt där varje ord representerar en egenskap av position, rotation, skala, och en pekare till modellminnet i denna ordning. Notera att egenskapen för skala inte används i den nuvarande implementationen av grafikprocessorn utan finns kvar för

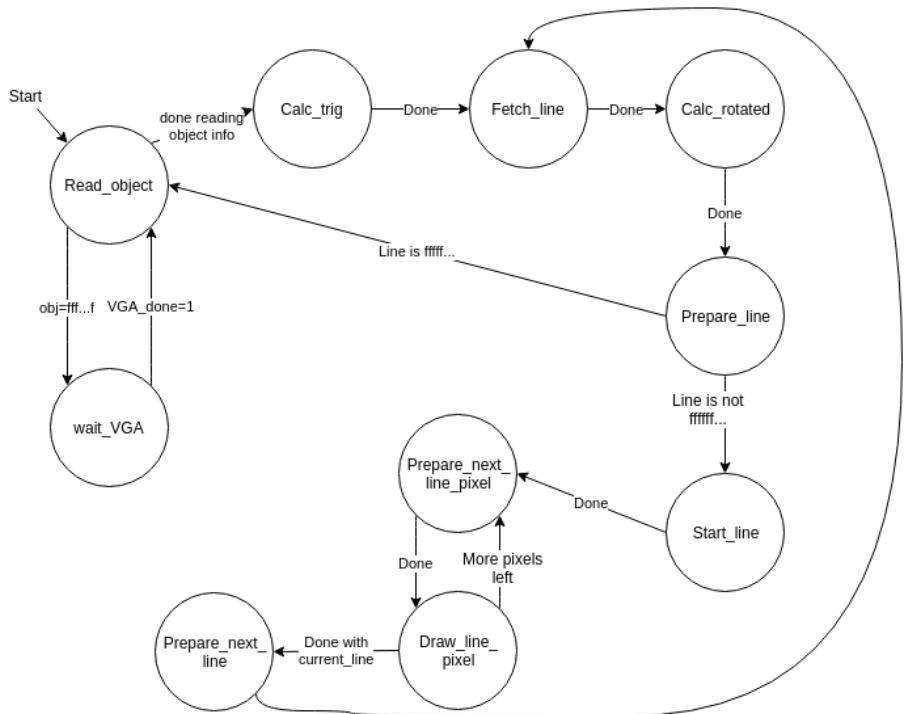
kompatibilitetsskäl. Efter alla objekt finns ett ord bestående av enbart höga bitar. Strukturen på objektminnet kan ses i figur 8.



Figur 8: Objektminnets struktur

4.6 Grafikprocessor

Alla objekt definierade i objektminnet renderas varje bildskärmsuppdatering från dess 3-dimensionella representation till 2-dimensionella koordinater på skärmen med hjälp av apparatens grafikprocessor. Detta sker i ett antal steg som kan ses i figur 9. Grafikprocessorn kommer, 60 gånger i sekunden, läsa in alla objekt i objektminnet och omvandla deras information till pixelminnet. Vid uppdatering av ett objekt kommer grafikprocessorn först läsa in den information som finns i objektminnet. Därefter räknas trigonometriska värden ut från objektets rotation. Nu kommer grafikenheten, för varje linje i objektets modell, bestämma vilka pixlar mellan start- och slutpunkt som ska sättas som "vit" i pixelminnet.



Figur 9: Översiktligt state-diagram för GPU

4.6.1 Inläsning av modeller

TODO

4.6.2 Beräkning av trigonometriska funktioner

Objektets rotation används för att beräkna olika trigonometriska värden som används i beräkningen av linjernas koordinater med avseende på rotationen. Det sker med hjälp av ett sinus lookup table där varje kombination av sin och cos för x,y,z vinkelns beräknas sekventiellt.

4.6.3 Inläsning av linjer

Varje linje består av två vektorer, en start och en slutvektor. Dessa står i modellminnet och en modell avslutas med en linje som har FFFF... som både start och slutvektor. Om den inlästa inte är en slutmarkör går GPU vidare till nästa state, annars går den till att läsa nästa linje. (egentligen sker den kollen efter beräkningen av den roterade vektorn).

4.6.4 Beräkning av linjens roterade position

Den nyligen inlästa linjen roteras med hjälp av en rotationsmatris (figur 10) som beräknas med de värden som tidigare bestämdes för objektets rotation. Egentligen beräknas inte rotationsmatrisen utan resultatet av rotationsmatrisen multiplicerat med positionsvektorn beräknas direkt. Eftersom att GPU inte hanterar djup beräknas bara X och Y elementen i den nya positionen. Den roterade vektorn kan ses i figur 11.

Beräkningen sker sekventiellt för både start och slutvektorn för linjen. För att beräkna varje element beräknas först de trigonometriska funktionerna ihop två och två. Är det 3 trig funktioner i en delberäkning sparas resultatet av de två första i en buffer som sedan multipliceras med den

$$\begin{pmatrix} \cos(c) & -\sin(c) & 0 \\ \sin(c) & \cos(c) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(c) & 0 & \sin(b) \\ 0 & 1 & 0 \\ 0 & 0 & \cos(b) \end{pmatrix} \cdot \begin{pmatrix} \cos(b) & 0 & \sin(b) \\ 0 & 1 & 0 \\ -\sin(b) & 0 & \cos(b) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Figur 10: Matriserna som ska multipliceras för att rotera med eulervinklar (a, b, c)

$$\begin{pmatrix} x\cos(b)\cos(c) + y(\cos(c)\sin(a)\sin(b) - \cos(a)\sin(c)) + z(\cos(a)\cos(c)\sin(b) + \sin(a)\sin(c)) \\ x\cos(b)\sin(c) + z(\cos(a)\sin(b)\sin(c) - \cos(c)\sin(a)) + y(\cos(a)\cos(c) + \sin(a)\sin(b)\sin(c)) \\ z\cos(a)\cos(b) + y\sin(a)\cos(b) - x\sin(b) \end{pmatrix}$$

Figur 11: Matriserna som ska multipliceras för att rotera med eulervinklar

tredje. När en delberäkning är klar multipliceras den med ett av elementen i positionsvektorn och resultatet sparas i en accumulator. När ett helt element i slutvektorn har beräknats sparas det och nästa element beräknas. När processen är klar för både start och slutvektorn går GPU:n vidare till att rita ut linjerna. Ett blockschema för processen kan ses i figur ??

4.6.5 Utritning av linje

När linjens början och slut bestämts i skärmkoordinater används Bresenhams linjealgoritm för att rita ut den. Först förbereds algoritmens begynnelsevillkor för att sedan stega igenom den axel för linjen som är längst och fylla de positioner som stegas igenom i pixelminnet. En detaljerad beskrivning av algoritmen finns i teorikapitlet.

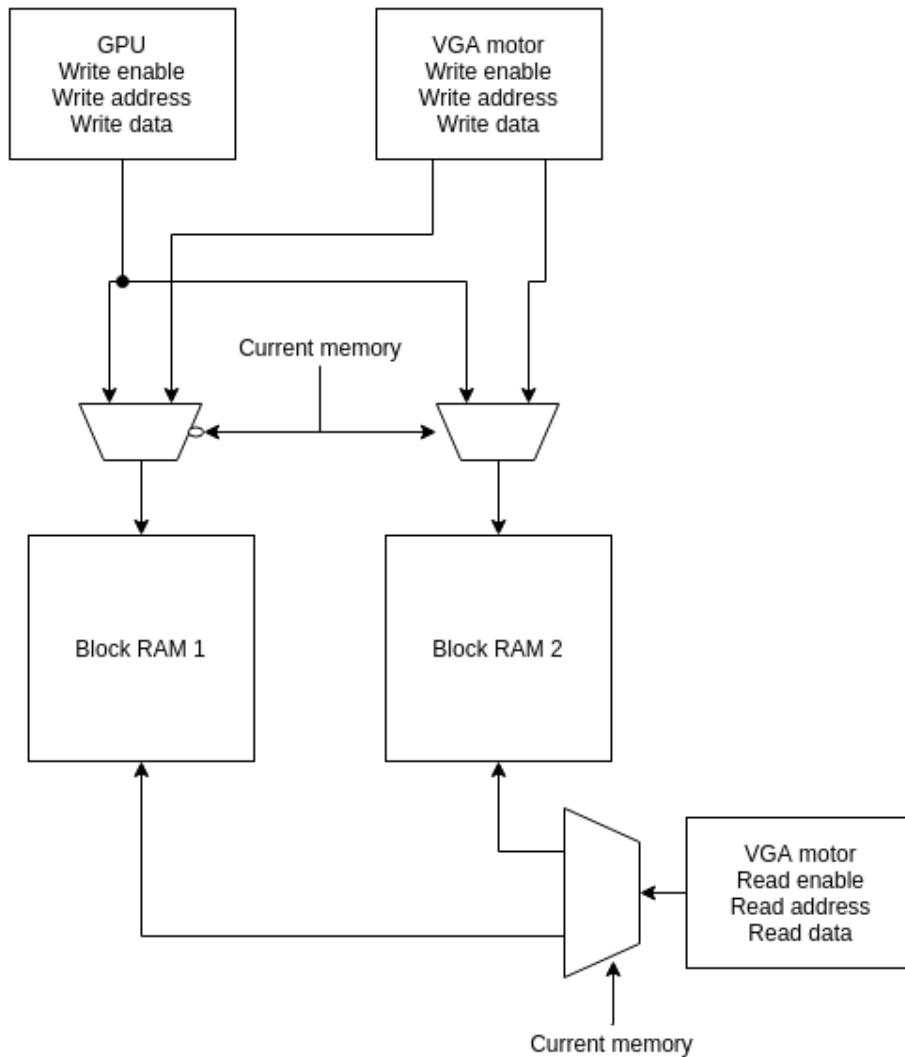
4.6.6 Väntan

När alla objekt ritats ut väntar grafikprocessorn med att börja om utritningen tills vga-motorn signalerat att en bild skickats till bildskärmen.

4.7 Pixelminne

Pixelminnet fungerar som kommunikationsbrygga mellan GPU:n och VGA motorn. Eftersom att GPU:n inte kommer ihåg vilka linjer som har ritats ut så måste skärmen någon gång rensas från den förra bilden. Vi valde att låta VGA motorn sköta det eftersom att den ändå stegar igenom varje pixel i bilden. Ett problem med den lösningen är att GPU:n inte kan rita sin bild samtidigt som VGA motorn rensar då delar av bilden i så fall skulle rensas bort innan de ritas ut. Därför valde vi att dubbelbuffra pixelminnet.

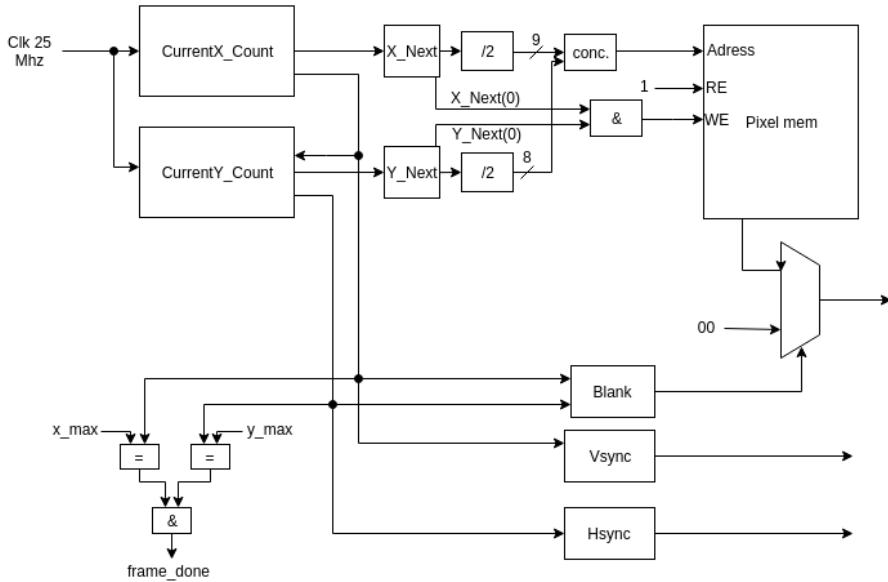
Pixelminnet består av två identiska block RAM, ett som GPU:n skriver till och ett som VGA motorn läser och rensar. När VGA motorn är färdig med en bild signalerar den till pixelminnet att det ska byta RAM som läses och skrivs. Då läser VGA motorn från det RAM som GPU:n skrev till vid förra bilduppdateringen och GPU:n får ett nyrensat minne att skriva i.



Figur 12: Blockschema över pixelminnet

4.8 VGA motorn

VGA-motorn är den komponent som stegar igenom varje pixel på skärmen för att sätta ett bildvärde på dem. En pixel kan vara svart eller vit. Denna information fås från pixelminnet som VGA-motorn har konstant kommunikation med. Eftersom det tar en klockpuls för att läsa från pixelminnet bestämmer VGA-motorn vilken position på skärmen som den borde vara vid nästa klockpuls för att synka positionen på skärmen med det värde som fås från pixelminnet. Bilden som ritas ut är 320x240 som skalas upp till 640x480. Detta innebär att varje "storpixel" i pixelminnet beskriver vilken färg som 4 pixlar på skärmen ska få. För att få ut rätt adress i pixelminnet vid läsning divideras x- och y-position enkelt med 2. När den sista pixeln för en "storpixel" sätts på skärmen skickar VGA-motorn en signal till pixelminnet för att nollställa just den "storpixeln". När en hel bild har uppdaterats sänds en signal till konstruktionens CPU och GPU för att informera att de ska fortsätta jobba. En detaljerad beskrivning syns i figur ??.



Figur 13: Blockschema över VGA motorn

5 Mjukvara

5.1 Assembler

För att lättare kunna skapa program till apparaten har vi designat ett program för att tolka assemblykod och konvertera det till maskinkod som sen kan laddas in via UART och exekveras. Genom att också förse assemblern med vissa mer avancerade programstatser blir det väldigt enkelt att skapa större program.

5.1.1 Labels

En av de enklaste funktionerna i en assembler är att kunna namnge hoppadresser och på så sätt slippa ändra hoppadresser varje gång som kod läggs till i mitten av programmet.

5.1.2 Variabler

Det är möjligt att definiera variabler längst med programmet som det reserveras en minnesplats till i arbetsminnet. Variabler kan antingen användas som konstanter eller ändras allt eftersom med hjälp av samma syntax.

5.1.3 Registeralias

Eftersom variabler endast kan användas i instruktionerna LOAD och STORE blir det många instruktioner som använder så kallade magiska nummer för att referera till de register som används. Lösningen som används i vår assembler stödjer därför möjligheten att sätta alias för register. Genom att också uppdatera dessa alias så fort man laddar in ett värde med LOAD blir programstrukturen betydligt mer lättöverskådlig.

5.1.4 Kommentarer

På grund av begränsningar i tolkens funktionalitet kan bara en typ av statement användas per rad, och det gäller även för kommentarer. Rader som börjar med tecknet för kommentarer hamnar inte i den genererande programkoden och kan istället användas för att förklara koden.

5.1.5 Loopar

Eftersom mycket i det program som vi **PATRIK F:R FAN DU KAN JU INTE BARA DRA UTAN ATT AVSLUTA MENINGEN. HUR I HELVETE SKA JAG VETA VAD SOM SKA STÅ HÄR. DET KAN JU BLI MISSFÖRSTÅND!**

5.2 Spellogiken