Design af en Multiprocesser 3D-Engine i SystemC til en FPGA

Thomas Christensen
Gustav Hvilsted

Kgs. Lyngby 2003
IMM-THESIS-2003-27

Design af en Multiprocesser 3D-Engine i SystemC til en FPGA

Thomas Christensen
Gustav Hvilsted

Kgs. Lyngby 2003

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

Abstract

This thesis deals with the further development of an existing 3D graphic system, Hybris, which were originally developed in software by Ph.D. Hans Holten-Lund.

There is a short summary of the theory behind the functions used in this thesis, and the changes that are made to that theory. This includes comments on how the system can be optimized for speed.

The current system builds upon a combined hardware-software system, which were developed in SystemC by Santiago Estaban Zorita. The sorting and calculation functions, which were in HAHL's original design, are transformed into equivalent hardware functions using SystemC. Zorita's SystemC source code is optimized for speed. The hardware units that are implemented and developed further, are part of the 3D system that sorts triangles and transforms these into pixels, which can be shown on a monitor.

The system in this thesis is targeted for a development board consisting of a Xilinx FPGA and a DDR-SDRAM module, which are the planned hardware platform in the future. Since the DDR-SDRAM hasn't been used before, a new interface is developed. This interface allows for different internal units to access the memory.

The *BackEnd* part of the system is expanded into a multiprocessor system, and general principles on how to create this system are explained.

Software simulators are developed to be able to evaluate the speed and functionality of the system. The simulators are used to draw selected objects in order to estimate the speed of the system. The simulations show that the developed system is stable, but has a few flaws. It is shown that the speed of the developed system is faster than the previous systems.

Finally some ideas, on how to further develop the design in the future, are described.

Resumé

Denne rapport omhandler videreudviklingen af et 3D grafik system, Hybris, der oprindeligt er udviklet i software af Ph.D. Hans Holten-Lund.

Der er en kort opsummering, af teorien bagved de funktioner der bruges og de ændringer, der er foretaget af teorien. Herunder en kort gennemgang af hvordan systemet kan optimeres til at opnå en bedre hastighed.

Denne udgave af systemet bygger videre på en kombineret software og hardware udgave, der er udviklet af Santiago Estaban Zorita i SystemC. Sorterings og beregnings funktioner fra HAHL's oprindelige softwareudgave er ændret til tilsvarende hardware funktioner i SystemC. Zoritas SystemC kode er optimeret for at opnå en bedre hastighed. De hardware funktioner, der er implementeret og videreudviklet, udgør den del af 3D systemet, som sorterer trekanter, og derefter omdanner disse til pixels, der kan vises på en skærm.

Systemet udvikles efter, at det på et tidspunkt skal implementeres i et udviklingsboard med en Xilinx FPGA og et DDR-SDRAM hukommelses modul. Da DDR-SDRAM hukommelse ikke har været brugt før, er der udviklet et interface til at kommunikere med denne, hvilket inkluderer en controller, der giver mulighed for, at flere forskellige enheder bruger hukommelsen.

BackEnd delen af systemet er udvidet til et multiprocessor system. Herunder er der også gennemgået nogle generelle principper for, hvordan dette opnås.

Der er udviklet software simulatorer til at vurdere systemets funktionalitet og hastighed. Vha. dem er der tegnet udvalgte figurer for at få data til at vurdere systemets hastighed. Dette viser, at der er opnået et velfungerende system, der dog har enkelte småfejl. Desuden vises det, at der er opnået store forbedringer i hastigheden, sammenlignet med de tidligere systemer.

Til sidst beskrives de udvidelser, der er tænkt på, men som ikke er implementeret i dette projekt.

Forord

Denne rapport er udarbejdet i forbindelse med et eksamensprojekt ved civilingeniør uddannelsen på DTU. Projektet er udført ved Institut for Matematisk Modellering i Computer Science and Engineering gruppen.

Projektet er udført med Hans Holten-Lund og Steen Pedersen som vejledere.

Vi vil gerne benytte lejligheden til at takke vores vejledere for gode råd og kommentarer til vores ideer under udarbejdelsen af dette projekt. Derudover vil vi også gerne takke Jens Sparsø for hans kommentarer omkring synkronisering og kommunikation mellem processer.

Endelig en stor tak til M.Sc. Morten Felsvang, M.Sc. Kim Sune Hansen og stud. polyt. Henrik Christensen for korrekturlæsning og kommentarer.

1 Indledning

Grundlaget for dette projekt er et eksisterende 3D grafik system, Hybris[1]. De oprindelige system modeller er lavet i software. Senere er den del af systemet, som tegner trekanter, implementeret i hardware for at øge hastigheden.

Dette projekt tager udgangspunkt i den seneste hardware version, som er implementeret i form af en HDL[2] model. Denne skal videreudvikles til at have en højere ydelse, målt ved antallet af producerede billeder per sekund. Derudover skal der implementeres ekstra funktioner i HDL, for at tilføje eksisterende software algoritmer i HDL modellen. Dette tjener 2 formål; en bedre brudflade imellem software og hardware, og tage endnu et skridt mod det overordnede mål; at få implementeret et komplet 3D grafik system i hardware.

Hardwareplatformen i de tidligere projekter er en Xilinx FPGA, hvorfor denne fortsat skal danne grundlag for projektet, i form af en ny og større udgave, implementeret på et udviklingsboard. Derfor skal de muligheder og begrænsninger dette giver undersøges.

Ændringerne realiseres ved at opdele projektet i faser, for at strukturere projektforløbet.

Først gennemgås den nødvendige teori for beregningerne, for at få et overblik over, hvilke krav og betingelser der er. Herunder bliver der gennemgået nogle ændringer af forudsætningerne for, hvordan systemet implementeres. Der opstilles nogle generelle ideer til, hvordan hardware af denne type optimeres.

Det undersøges, om der kan findes en bedre snitflade mellem software og hardware, dvs. det undersøges, om der kan overføres færre informationer per trekant, i forhold til det eksisterende system. Dette medfører at der implementeres ekstra funktioner til behandling af datastrukturer, herunder et system til håndtering af eksterne hukommelser.

De eksisterende hardware funktioner analyseres, med det formål at reducere tidsforbruget per datastruktur. Derudover implementeres der et multiprocessor system, for at opnå parallelle beregninger.

For at give et retvisende billede af systemets funktionalitet, udvikles der en simulerings-platform, som er i stand til at opsamle statistisk information til at dokumentere systemets ydeevne. Dette danner grundlag for sammenligninger mellem forskellige udgaver af systemet. Derudover udføres der hastighedsmålinger for enkeltdele af systemet, til at understrege forbedringerne, men også til at identificere eventuelle flaskehalse.

[1] Udviklet af Hans Holten-Lund [1, 2]
[2] Hardware Description Language

1.1 Læsevejledning

Rapporten er opdelt i sektioner; først generel teori og baggrundsinformation vedrørende 3D grafik systemer. Derefter følger tre kapitler om, hvordan systemet er implementeret i hardware. Endelig er der et kapitel, der viser det implementerede designs formåen.

Det forventes, at læseren har et grundlæggende kendskab til 3D grafik, hvorfor ikke alle grafik begreber er forklaret i teksten. Teorien bag Hybris systemet er ikke gengivet i denne rapport, og der optræder derfor mange henvisninger til denne, hvorfor det vil være en fordel for forståelsen af dette projekt, at kende de bagvedliggende principper.

De udviklede systemer er med engelsk brugerflade, og kommenteringen af kildekode er ligeledes engelsk. Dette er gjort, for at give udenlandske studerende m.fl. en mulighed for at forstå, bruge og videreudvikle systemet. Dette gør, at der rapporten igennem er brugt engelske ord for de enkelte variable, hvilket måske virker lidt afbrydende i læsningen, men sikrer en direkte reference til kildekoden. Tilsvarende er der brugte engelske termer for tekniske udtryk.

Den udviklede kode ligger på den medfølgende cd, og af bilag 1 ses en komplet indholdsfortegnelse for cd'en.

1.2 Hybris baggrund

Hybris projektet startede i midten af 90'erne, hvor det havde sin spæde start under navnet HPGA – High Performance Graphics Architecture eller Hybrid Parallel Graphics Architecture. Det blev udviklet som et eksamensprojekt af HAHL [1], med det formål at konstruere en effektiv skalerbar 3D-engine.

Med baggrund i et gennemtænkt grunddesign blev HPGA systemet videre udviklet i et Ph.D. projekt, til at inkludere en komplet 3D-engine i software, inklusiv de nødvendige programmer til at indlæse modeller fra VRML[3] filer. Ph.D. projektet blev ligeledes lavet af HAHL [2], og dette projekt tager udgangspunkt i projektets resultater og teori. I mellemtiden skiftede designet navn til det mere PR-venlige "Hybris".
Grunddesignet har vist sig at være så effektivt, at den udviklede software-renderer kunne hamle op med relativt kraftige hardware-accelererede løsninger – først med nVidia's GeForce 4 og ATI's Radeon 7500 serier kan scener, baseret på VRML, renderes hurtigere med hardware-acceleration end softwaren kan.

I takt med udviklingen af Hybris systemet har det også været et ønske at lave noget af renderingen i hardware. Det har med tiden resulteret i flere eksamensprojekter, sideløbende med HAHL's Ph.D. projekt.
Disse projekter har alle implementeret den del af Hybris systemet, som "tegner" de trekanter, der skal vises på skærmen. Dette inkluderer beregning af pixels, og beslutning om hvilke pixels der skal tegnes, hvis flere trekanter overlapper.

[3] Virtual Reality Modelling Language

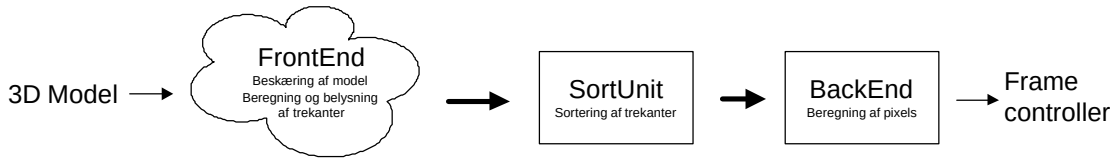
De første projekter blev designet med nogle relativt små FPGA kredse[4], og programmeret i hardware sproget VHDL.

Senere er hardware programmeringssproget SystemC lanceret, som giver mulighed for at udvikle koden på et højere abstraktionsniveau, dog uden at miste forbindelsen til hardwaren. Dette kan gøres, da SystemC bygger på C++, og består af en mængde biblioteker, som kan inkluderes i et almindeligt C++ design. Kode skrevet med det formål, at det skal være muligt at syntetisere, kan derpå oversættes på samme måde som VHDL kan[5].

Dette resulterede i et eksamensprojekt udført af Santiago Estaban Zorita[3], som oversatte den VHDL kode, der var lavet af Henrik A. Sørensen [4].
Denne SystemC kode har været det direkte grundlag for dette projekt, som til dels har gået ud på at parallelisere og effektivisere denne kode.

1.3 Hybris designet generelt

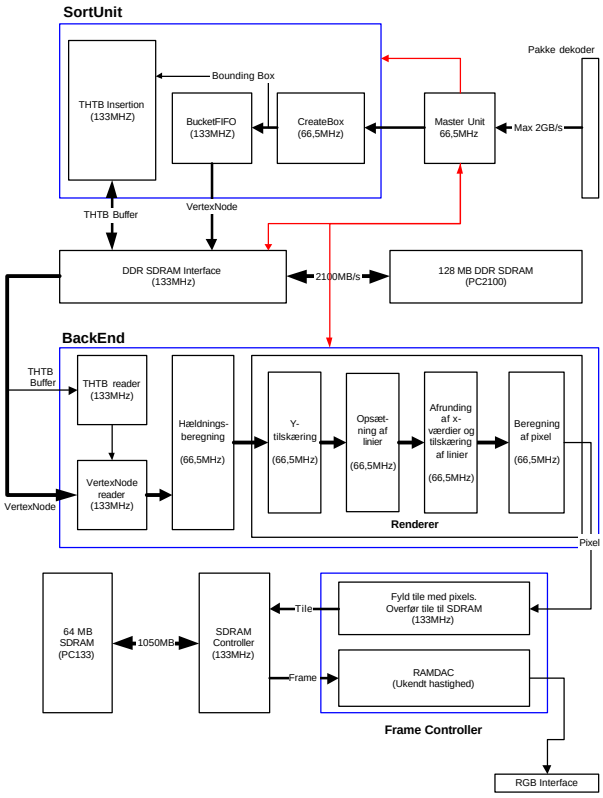
Hybris designet er lavet med henblik på at være skalerbart, hvilket det da også har vist sig at være. Det består af tre hoved dele, som vist på figur 1.
FrontEnd er den mest komplicerede, da det er denne, der arbejder med de komplette scener, og sørger for at transformere objekterne fra modeller til trekanter, hvor der er beregnet belysning, størrelse og placering af samtlige trekanter (kendt som ”Transform & Lightning”).
Objekterne i Hybris er partitioneret i mindre delobjekter, og *FrontEnd* beregner placeringen af hvert delobjekt, og er i stand til at fjerne de dele, som ikke vil være synlige på skærmen.
De objekter, der ikke fjernes, bliver overført til *SortUnit*. Det sker ved at gennemgå objektet for trekanter og sende disse til *SortUnit*, hvorved *SortUnit* og *BackEnd* kun skal regne med trekanter.



Figur 1: Hybris systemets tre hovedblokke

SortUnit har til opgave at finde ud af, hvor store trekanterne er, og derpå fordele dem ud i de del-områder af skærmen, som trekanterne rammer. Sorteringen skal sikre, at sidste del, *BackEnd*, har optimale arbejdsbetingelser, så den kan tegne en del af skærmen færdig, før den skal tegne næste del af skærmen.
En mere detaljeret udgave af systemet ses af figur 2, og kan med fordel benyttes under læsningen af denne rapport, da det giver et overblik af Hybris systemets generelle opbygning.

[4] Små, set i forhold til de FPGA kredse, der kan købes i dag
[5] SystemC kode bliver oversat til VHDL med de, i dette projekt, brugte værktøjer, se appendiks A



Figur 2: Komplet Hybris Diagram. Kan ses i fuld størrelse i bilag 2

Da det i dag er muligt at købe FPGA kredse, som er 10-20 gange større og kraftigere end for bare få år siden, er det nu muligt at flytte endnu mere af Hybris systemet ned i hardware. Det har betydet, at dette projekt har flyttet *SortUnit* ned i hardware, således at det nu kun er *FrontEnd*, som er en ren softwaredel.

Til at opsamle de pixels Hybris systemet genererer, vil der sidde en *Frame Controller*, der vil overføre dem til en frame-buffer, som siden vil blive overført til computer-skærmen. Denne del er endnu ikke implementeret i dette projekt.

1.4 Udviklingsmiljø

Som hardware platform benyttes et udviklingsboard, der er udviklet af Avnet [5]. Boardet har et passende sæt af features, som projektet kan drage nytte af. Nogle af disse er:

- PCI-X bus, 64bit/100MHz
- Hukommelse, DDR-SDRAM PC2100
- Xilinx Virtex II XC4000 FPGA
    - 2,1Mbit intern hukommelse i form af BlockRAM moduler
    - 4 millioner gates
    - 912 I/O ben
    - 120 multipliers
- Udvidelsesporte på kortet, som bl.a. kan bruges til ekstra hukommelsesmodul
- Pris ca. 21000 dkr.

Hardware sproget er SystemC. Det benyttes, da den eksisterende kode er lavet i SystemC. Eftersom det er et nyt sprog, kompileres koden først til VHDL og syntetiseres derefter mod en chip. Til at kompilere SystemC koden til VHDL bruges Synopsys´ SystemC compiler, og til at syntetisere VHDL koden er Xilinx egne ISE værktøjer benyttet. Der er flere værktøjer til rådighed, men Xilinx værktøjerne giver en komplet pakke med mulighed for at inkludere små cores[6], der på forhånd er lavet af Xilinx, sådan at alt ikke skal laves helt fra bunden.

Udviklingsrutiner og værktøjer er beskrevet i appendiks A.

1.5 Sammenfatning

I dette kapitel er grænserne for projektet ridset op, herunder hvilke målsætninger der er, og hvordan de opnås. Herunder er også en kort historisk baggrund, som beskriver de tidligere projekter, der danner grundlag for dette projekt.
Det beskrives kort, hvordan 3D modellen behandles og hvordan dette projekt fokuserer på den sidste del af beregningen af et 3D objekt: Sorteringen af trekanter, og omdannelsen til punkter på en skærm, så det former et 3D objekt for det menneskelige øje.

Til sidst er der en kort beskrivelse af den hardware platform, der stilles til rådighed for projektet.

[6] Xilinx Core Generator indeholder en lang række core funktioner, som kan tilpasses det enkelte design

2 Design analyse

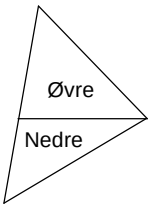
I dette kapitel introduceres algoritmerne for, hvordan en trekant opbygges, sådan at den får den korrekte udformning på skærmen. Desuden beskrives enkelte fejl, der er fundet i Hybris systemet. Endelig følger en forklaring af, hvordan data og kode fra de forrige projekter ønskes analyseret.

2.1 Opbygningen af en trekant

Hele teorien til beregning af de variable, som skal bruges til at tegne en trekant korrekt, er udarbejdet af HAHL, og er dokumenteret i [1, 2]. Her følger en beskrivelse af de dele af teorien, som bruges i dette projekt.

Trekanten bygges op udfra 3 koordinat sæt, et for hvert hjørne. Koordinatsættene består hver af x, y, z og en farveværdi, og skal tage udgangspunkt i skærmens koordinater.

Når trekantens punkter navngives bruges top, midte og bund. Dette er alene regnet efter y-aksen. Trekanterne deles efter en vandret streg gennem deres midtpunkt. Ligger 2 punkter i samme y-koordinat vælges et af dem til midten. Herudfra kan en trekant deles i 3 kategorier; dem der kun har noget over delingen, dem der kun har noget under delingen og dem der har noget begge steder af delingen. På figur 3 ses definitionen af øvre og nedre trekanter.



Figur 3: Øvre og nedre trekant

Hele beregningsmodellen tager udgangspunkt i et start-punkt, og alle ændringer sker udfra dette startpunkt, hvor de beregnede hældningskoefficienter bruges til at beregne, hvor næste punkt eller linie skal tegnes.
Der vælges altid en start-værdi øverst i trekanten, men beregningsmodellen skal også være i stand til at håndtere trekanter med flere toppunkter, dvs. en trekant uden en øvre trekant.
Ud fra det første punkt, *Start*, beregnes hældningsværdier for x, z og farve. Disse værdier bruges til at interpolere første pixel-placering i hver linie, og til at beregne bredden af hver linie. Der vil altid blive interpoleret ned langs den side af trekanten, der går fra top til bund. På denne måde er det nok med et sæt hældningsværdier. Der skal desuden beregnes et linie-z-differentiale og et linie-farvedifferentiale, så det er muligt at bestemme farveværdien og z-positionen på hver pixel langs en linie. Dermed bliver det muligt at lave glidende farveovergange.

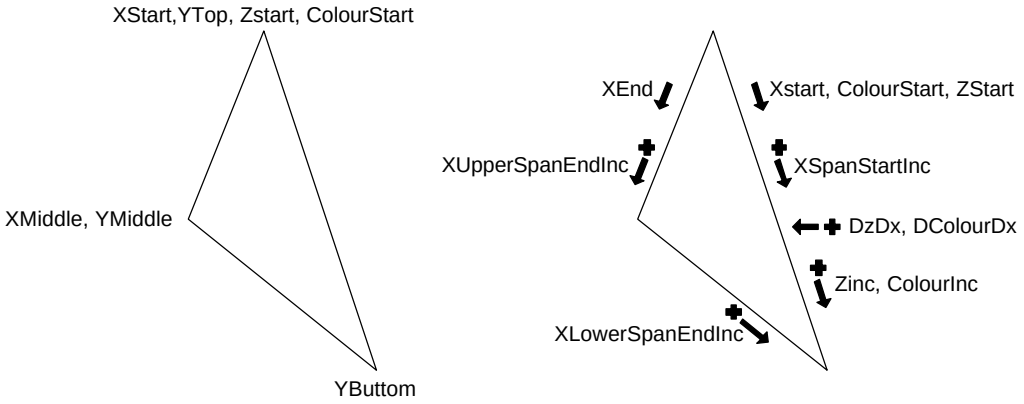
Når disse værdier er beregnet, kan trekanten tegnes ved at begynde i start-positionen og tegne alle pixels i linien, ved at tage udgangspunktet, og lægge de beregnede linie-hældnings værdier

til. Efter at linien er tegnet, kan næste linies start-værdi bestemmes, ved blot at addere med de beregnede hældningsværdier i y-retningen.

De variable der danner grundlag for beregningen af trekanten, kan ses af tabel 1, og på figur 4 er det illustreret, hvordan de enkelte variable er placeret i trekanten. På venstre figur er startværdierne illustreret og på højre figur er alle hældningsværdierne placeret sammen med de værdier de adderes med. Det er illustreret, hvilken side de forskellige værdier følger.

Variabel	Beskrivelse
XStart	X-koordinat for trekantens toppunkt / den aktuelle linies startpunkt
YTop	Y-koordinat for trekantens toppunkt
XMiddle	X-koordinat for det hjørne der ligger mellem top og bund
YMiddle	Y-koordinat for det hjørne der ligger mellem top og bund
YButtom	Y-koordinat for det hjørne der ligger i bunden af trekanten
XEnd	X-koordinat for den aktuelle linies slutpunkt
XUpperSpanEndInc	Ændringsværdi for x langs med den korte side
XSpanStartInc	Ændringsværdi for x langs med den lange side
XLowerSpanEndInc	Ændringsværdi for XEnd efter midterpunktet er passeret
ZStart	Z-værdien i trekantens toppunkt
ColourStart	Farveværdien i trekantens toppunkt
ZInc	Ændringsværdi for z langs med start siden
ColourInc	Ændringsværdi for farven langs med start siden
DzDx	Ændringsværdi for z langs med x-aksen
DColourDx	Ændringsværdi for farven langs med x-aksen
Valid	Flag som angiver, at trekanten skal tegnes
Last	Flag som angiver, at det er sidste trekant i dette sæt

Tabel 1: Oversigt over alle variabler i en trekant.



Figur 4: Principdiagram for trekantvariabler

Da XMiddle = XStart + n·XUpperSpanEndInc, hvor n er antallet af linier fra YTop til YMiddle, virker XMiddle umiddelbart overflødig, men der sker små afrundingsfejl, der gør, at dette punkt godt kan flytte sig lidt, hvorefter linien fra midten til bunden vil ligge forskudt. Dette undgås ved at trække x-værdien på plads med XMiddle, når hjørnet nås.

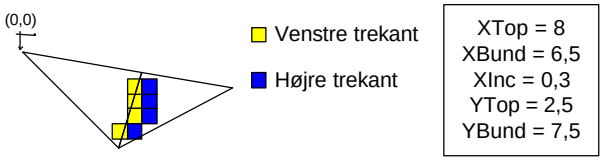
2.2 Afrunding til skærmkoordinater

Når de beregnede koordinater skal gå fra decimaltals præcision til værdier, der passer præcist med en pixel på skærmen, skal der ske en afrunding. Her kan der ikke rundes tilfældigt op og ned, da dette kan give ”døde” pixels imellem trekanterne. Derfor er følgende regler vedtaget [2a]:

- X-koordinater i trekantens venstre side rundes altid op
- X-koordinater i trekantens højre side rundes altid ned

Der er dog en undtagelse, da en koordinat, der ikke skal afrundes, vil give sammenfald af højre og venstre punkt. Dette undgås ved at højrekoordinaten i disse tilfælde reduceres med 1. Tilsvarende gælder det for y-koordinaterne, at top koordinaten rundes op, og bundkoordinaten rundes ned, og når bundkoordinaten er hel, reduceres den med 1. Dette giver ikke nogen regel for midten, men da linierne skal følges ad, for at der ikke skal opstå døde pixels, skal midtkoordinaten både rundes op og ned, sådan at der opstår 2 værdier for midtpunktet. På den måde vil en vilkårlig linie altid få sin startværdi rundet op og slutværdi rundet ned.

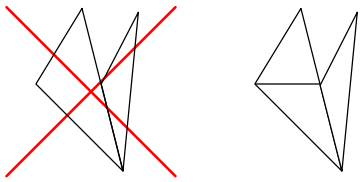
Afrundingerne gør, at 2 trekantssider, der deler koordinatsæt, vil komme til at ligge helt tæt op af hinanden i skærmkoordinater, men de vil ikke dele nogle pixels. Af figur 5 ses et eksempel på, hvordan grænsen mellem to trekanter tegnes.



Figur 5: Afrundingseksempel

En trekant, der ligger vinkelret på skærmen, sådan at den skal ses som en streg, vil altid blive afrundet sådan, at den ikke ses. Det skyldes, at linier med samme start og slutkoordinat vil blive afrundet, sådan at start koordinaten kommer til at være en større end slutkoordinaten og derfor får den enkelte linie i stregen en effektiv længde på –1. Det samme vil ske for den første linie i en trekant, som altid vil have samme start og slut koordinat.

For at alle trekanter skal ligge tæt op af hinanden, samtidigt med at der bruges afrundinger, er det nødvendigt, at alle trekanter er arrangeret sådan, at deres hjørner altid møder hjørner fra de omkring liggende trekanter. En trekant må ikke have et hjørne midt på en anden trekants side. Dette er illustreret på figur 6.



Figur 6: Hjørne side eksempel

Som det også kan ses af figur 6, er det nemt at løse problemet, ved blot at dele den ene trekant i 2 mindre, hvilket gøres i *FrontEnd*.

2.3 Ændring af variable

For at minimere antallet af bit der sendes igennem systemet, analyseres alle variable og datastrukturer.

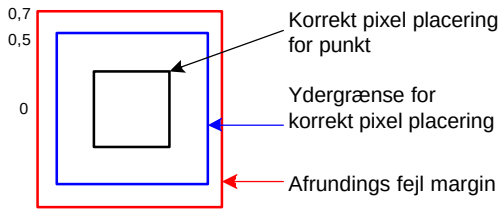
2.3.1 Trekant beskrivelse

I HAHL's design beskrives de tre punkter i en trekant med en datastruktur, der indeholder informationer, end dem som *SortUnit* og *BackEnd* skal bruge. Strukturen består af følgende data:

```
typedef struct _RMPCSvertex
{
  Real tx,ty,tz;        /* Transformed-Clipped 3D vertex. */
  Real px,py,pz;        /* Projected vertex (screen coords). */
  Real si;              /* Shaded-Clipped vertex color. */
  WORD tu,tv;           /* Transformed-Clipped texture coords. */
} RMPCSvertex;
```

Real er defineret som **float**[7], mens WORD er en **short**[8]**.** Dvs. at et punkt fylder 256 bit.
Hardware delen af Hybris har kun brug for information vedrørende punktets koordinater på skærmen og dets farve, hvilket er de informationer, som er fremhævet i ovenstående struktur.

Disse informationer skal omdannes til et fixed point format[9], hvilket gøres i softwaren vha. funktionen FP2INT. Denne omdanner en **float** værdi til en værdi med 12 betydende cifre efter kommaet, hvilket garanterer, at afrundingsfejlen maksimalt er 1/4096. Det betyder, at den maksimale afrundingsfejl fra punkt (0, 0) til (2047, 2047) vil være 0,7, da diagonalen svarer til 2894 pixels. Det kan betyde, at en meget stor trekant kan risikere at forskubbes med én i alle retninger, inkl. farve – se figur 7.



Figur 7: 12 bit til decimaler kan resultere i pixelfejl

For at bringe størrelsen af en punktbeskrivelse ned, skæres x og y koordinaten ned til 23 bit, og z værdien til 24 bit. Farveværdien skæres ned til en 8 bit værdi, hvilket giver mulighed for

[7] 32 bit på en 32 bit platform
[8] 16 bit på en 32 bit platform
[9] Der afsættes 11 bit til heltallet og 12 bit til decimaler, hvilket giver en maksimal opløsning på 2048x2048

256 nuancer. Da denne farve værdi bruges til at danne gråtoner[10], er 256 nuancer rigeligt til at snyde det menneskelige øje, da vi ikke kan differentiere mellem mere end maksimalt 100 gråtoner [6, 7]. Internt i Hybris systemet arbejdes dog med 20 bit, hvoraf de 12 bit er til decimaler, så glidende farveovergange sikres.

Det betyder, at et punkt kan repræsenteres med 78 bit, og en trekant med 234 bit, hvilket har stor betydning for antallet af trekanter, som kan modtages pga. begrænset båndbredde.

Hvert punkt repræsenteres, i hardwaredelen af Hybris, med følgende struktur:

```
struct FPGAVertex
{
      sc_uint< 23 >     x, y;
      sc_uint< 24 >     z;
      sc_uint< 8 >      c;
};
```

Som det fremgår, er alle koordinater repræsenteret som positive heltalsværdier, og det betyder, at alle koordinater, der afleveres til Hybris hardwaren skal være positive[11].
En samlet beskrivelse af trekanten, består af 3 af ovenstående datastrukturer, og kaldes for en *VertexNode.*

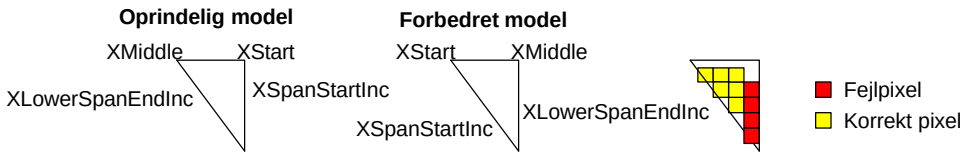
2.3.2 Interne variable

Datastrukturen til *Renderer* indeholdt to ens x-værdier, XLeft og XRight [2b], som er identiske. Fejlen stammer fra en tidligere udformning af systemet, hvor der har været forskel på de 2. Den ene bliver derfor fjernet i dette design, og den tilbageværende får det mere sigende navn XStart, da der er tale om x-værdien for det første punkt.

For trekanter, der kun består af den nedre del, opstår der en afrundingsfejl i Hybris systemet, da XSpanStartInc er øget med 1. I skærmkoordinater svarer dette til $1/(2^{12})$ svarende til 0,000244, da det er fixed point tal med 12 decimaler. XSpanStartInc øges, da der ikke er nogen øvre trekant til at give en værdi for XUpperSpanEndInc, denne sættes derfor til samme værdi som XSpanStartInc. Det giver problemer, i forhold til den sammenligning, der afgør fra hvilken side trekanten tegnes[12]. Det er undgået ved at øge XSpanStartInc med 1. Problemet opstår, når XStart er et heltal, så vil første punkt blive rundet et helt tal ned, mens næste punkt bliver rundet ned til XStart, da det nu er øget ganske lidt. Dette giver en trekant, der har en ekstra linie i højre side, se figur 8. Problemet kan let løses ved at reducere XUpperSpanEndInc i stedet for at øge XSpanStartInc, da XUpperSpanEndInc ikke bruges til andet end at afgøre, hvilken vej trekanten skal tegnes.
En anden løsning på problemet er at bytte om på XStart og XMiddle, og tilsvarende bytte om på XSpanStartInc og XLowerSpanEndInc, derefter sætte XUpperSpanEndInc til XLowerSpanEndInc. På denne måde skal der altid tegnes fra venstre mod højre, og da XLowerSpanEndInc er mindre end XspanStartInc, vil trekanten blive genkendt til at skulle tegnes i denne retning. Fordelen ved denne løsning er, at der spares en algebraisk operation,

[10] Det gøres ved at lade alle 3 RGB-farver have samme værdi, hvilket giver gråtoner.
[11] Da negative tal er et spørgsmål om fortolkning af MSB, vil det være muligt at overføre "negative" tal. De vil dog blive fortolket som positive af Hybris og dermed give uforudsete beregningsfejl
[12] Afgørelsen af, i hvilken retning trekanten tegnes, forklares i afsnit 5.1

hvilket vil øge hastigheden og spare plads i hardware, da der er tale om et 24 bit tal, hvilket giver en del logik pga. carry kontrol. Ombytningen af de variable fremgår af figur 8. Denne løsning vælges, da det giver den hurtigste løsning hardwaremæssigt.



Figur 8: Fejl ved nedre trekant

Til sidst bliver de flag, som styrer skift af tiles analyseret. Det resulterer i, at flagene *LastTri* og *NewTile* bliver lagt sammen, da der ikke er nogen grund til at signalere slutningen af en tile, og, i den efterfølgende datastruktur, starten af næste. De er i stedet erstattet af flaget *Last*, som signalerer, at datastrukturen er sidste del i billedet. Samtidigt med skal der i outputdelen implementeres en tæller, som tæller *Last* flag; og på den måde ved, hvornår der er tegnet et komplet billede.
En bit sparet er ikke meget, men det giver den fordel, at det ikke er nødvendigt at tage hensyn til begge flag igennem systemet, da der for hvert flag skal ske en håndtering af, hvilken datastruktur flagene skal sendes ud med. Dette er kun et problem i de tilfælde, hvor en datastruktur bliver til flere.

2.4 Optimering af beregninger

For at opnå et hurtigt design udføres forskellige analyser af det hardware, der skal udvikles. Udgangspunktet for analyserne er inddelt i nedenstående punkter:

- Optimering af tilstandsmaskiner til at foretage færrest mulige beregninger
- Restrukturering af beregninger
- Parallelisering
- Hurtigere udveksling af data
- Ændring af beregningsalgoritmer
- Optimering efter specifikke hardware funktioner
- Pipeline design
- Multiprocessor opbygning

Tilstandsmaskinerne er et godt udgangspunkt for en optimering, da de typisk kun producerer en datastruktur per gennemløb, og derfor er interessante at reducere, til færrest mulige trin, eventuelt helt fjerne. Det gøres ved at sørge for, at det kun er beregninger, som er afhængige af deres egne resultater, som placeres i tilstandsmaskiner. Det forsøges så vidt muligt at få dem til at regne i et trin, hvilket ikke altid kan lade sig gøre, pga. at det sløver designet for meget ned, og så må der indsættes flere trin i tilstandsmaskinen, som gør det ud for en slags pipeline, dog med den begrænsning, at der kun arbejdes på en datastruktur af gangen.

Når tilstandsmaskinerne er reduceret, skulle de resterende beregninger gerne kunne placeres i pipelines før og efter tilstandsmaskinerne.

Her gælder det om at strukturere beregningerne således, at de kan forløbe parallelt med hinanden, dvs. at alle operander til en beregning helst skal udregnes samtidigt, sådan at beregningen ikke skal vente på, at de bliver beregnet en efter en. Samtidigt med skal register forbruget i en pipeline holdes nede. Det gør, at placeringen af hver beregning i pipelinen skal være sådan, at hvis de indgående operander har flere bit end resultatet, og operanderne ikke bruges til andre beregninger, skal resultatet beregnes så tidligt som muligt, for at undgå ekstra bit i pipelinen. Gælder det omvendte for bit forholdet, skal beregningen foregå så sent som muligt.

Udveksling af data mellem tilstandsmaskiner og pipeline enheder skal hver gang tilstands-maskinerne kan levere data. Dette er dog ikke helt simpelt, da en pipeline altid modtager og leverer en datastruktur for hver periode, mens tilstandsmaskinen i nogle tilfælde kan følge med til at levere en datastruktur hver periode, men i andre tilfælde kun hver anden eller sjældnere. Dette kan løses ved, at tilstandsmaskinen sætter et flag, der viser, at datastrukturen er ugyldig, når den ikke er færdig med at regne på en given datastruktur. F.eks. hvis en tilstandsmaskine bruger 2 perioder til en beregning. Derfor vil output i den ene periode være en tilfældig datastruktur med *Valid* flaget sat falskt. For at denne fremgangsmåde ikke skal overbelaste enheder efter en pipeline, indsættes der en sortering til sidst i en pipeline, som sorterer alle ugyldige datastrukturer fra. På den måde bliver datastrukturer, som i løbet af beregningen er blevet erklæret ugyldige, også fjernet og arbejdet for de efterfølgende enheder bliver reduceret yderligere.

For leveringen af data til en tilstandsmaskine er situationen mere kompliceret, da der ikke kan opstilles en nem beregning for om den er færdig i løbet af 1 eller 10 perioder. En mulighed er handshakes, men de kræver flere perioder og er svære at integrere med en pipeline. FIFO[13] buffere derimod er en langt bedre løsning, da de samtidigt med tjener andre formål, udover at de kan modtage eller levere en datastruktur for hver periode, giver de også mulighed for kun at modtage eller levere når den tilhørende enhed ønsker det. Dog kræver det, at enheden tjekker om bufferen er hhv. fuld eller tom, alt efter om enheden ønsker at skrive eller læse. Desuden giver FIFO buffere en mere flydende beregning, da de kan ophobe flere datastrukturer, og dermed reducerer behovet for at stoppe beregningerne, pga. manglende datastrukturer eller en enhed, der er i gang med en krævende beregning og derfor ikke kan aftage datastrukturer.

Udover at ændre på beregningsrækkefølgen, kan der ændres på, hvad der udregnes. F.eks. når en tilstandsmaskine styres af en tæller, som tæller fra a til b, kan det tit være en fordel at udregne b-a i den foranliggende pipeline og dermed kun sende forskellen videre. Det giver en mindre bit bredde i den mellemliggende FIFO buffer, hvilket sparer logik.
Det er altid interessant at reducere bit bredden af datastrukturerne, men det er forskelligt hvor stor gevinsten er. Ved bus systemer, hukommelse og FIFO buffere er databredderne inddelt i blokke, mens de for pipeline registre kan skaleres friere. Derfor skal datastrukturer, for de enkelte enheder, undersøges og sammenlignes med de blokke, de skal placeres i. Når en datastruktur bruger flere blokke, hvoraf sidste blok indeholder få bit, vil det give forbedringer at spare de bit, der er i sidste blok og dermed blokken.

Til sidst i designfasen af de enkelte enheder skal det overvejes, hvordan deres beregninger kører hurtigst på den planlagte hardware platform. Dette kan gøres udfra en viden om, hvad

[13] First In First Out

der er af specielle enheder på chippen, men også udfra nogle generelle betragtninger. F.eks. vil det tidligere nævnte eksempel med en løkke, der styres af en tæller, der går fra b-a til 0, i stedet for fra a til b give den fordel, at der skal foretages en sammenligning med 0 for at tjekke, om løkken er nået til enden, hvilket er langt hurtigere end at sammenligne 2 tal.
Udover overvejelser om hvordan de enkelte beregninger passes bedst muligt ned i chippen, skal det også overvejes om nogle beregninger eller if-sætninger kan gøres simplere udfra viden om, hvad de indgående variable kan have af værdier. F.eks. vil viden om, at variablen X indeholder et ulige tal kunne reducere følgende regnestykke således:

$$X_1 = ((X + 1) >> 1) - 1 = X >> 1$$

Denne form for optimering kræver et godt overblik over koden og samtidigt med en kommentar i koden om, hvad betingelsen er for denne. Dertil kommer, at en sådan optimering nemt kan indføre fejl, hvis betingelsen for denne rettes et andet sted i koden.

Der er lavet pipelines på 2 måder; den traditionelle måde, hvor det er analyseret, hvor det er hensigtsmæssigt at placere registrene og vha. Xilinx compilerens Registerbalancing.
Registerbalancing går ud på, at der under opbygningen blot placeres et passende antal registre på udgangen af den ønskede pipeline, og derefter fordeler compileren dem selv, dvs. den rykker dem ind imellem udregningerne. Umiddelbart burde dette give det bedste resultat, da compileren kan regne efter, for hver eneste logikenhed, hvor registrene bedst placeres. Dette fungerer desværre ikke helt problemfrit endnu, men ideen er god, og den gør det noget nemmere at fejlsøge i SystemC koden, da mellemregninger ikke er forsinket i forhold til hinanden, når der simuleres i C++.

Når hver enhed er optimeret, kan det overvejes at lægge flere kopier af hver enhed ind i designet for at opnå flere samtidige beregninger af datastruktur. Dette behandles i kapitel 5.3.1.

2.5 Sammenfatning

I dette kapitel er der givet en forklaring af principperne for, hvordan en trekant struktureres og beregnes udfra 3 koordinatsæt, som består af x, y, z og farvekoordinater. Herunder en beskrivelse af, hvordan koordinater, der er beregnet som kommatal, konverteres til heltal, for at svare til et entydigt punkt på skærmen, og samtidigt med sikre, at der ikke opstår punkter midt i figuren, der ikke tegnes.
For at sikre den højeste hastighed, er der opstillet nogle ideer til, hvordan data og kode, fra de forrige projekter, skal analyseres og ændres. Derudover er der beskrevet 2 fejl, som er fundet i det oprindelige Hybris system; den ene har kun betydning for hastigheden, mens den anden vil føre til fejl i en bestemt type af trekanter.

3 Udveksling af data i systemet

Dette kapitel omhandler de komponenter, som transporterer data til og fra Hybris systemet, og som ikke er en del af Hybris systemets *SortUnit* eller *BackEnd*.
Der indledes med en beskrivelse af, hvordan data skal afleveres til det udviklede system. Efterfølgende beskrives den enhed, der kontrollerer at data kommer korrekt ind i *SortUnit*, og som sørger for, at *SortUnit* og *BackEnd* arbejder med de korrekte dele af den eksterne hukommelse.
Derefter følger en beskrivelse af de eksterne hukommelser og de enheder, der er udviklet, eller tænkt udviklet, til at kommunikere med disse.
Til sidst følger et afsnit, som generelt beskriver, hvad der er gjort for at adskille enheder, der arbejder med forskellige frekvenser.

3.1 Datapakker til FPGA

Det nuværende design indeholder ikke noget færdigt interface mod specifikke eksterne datakilder, men er designet til at have et simpelt interface mod en nødvendig pakkedekoder. Der tages udgangspunkt i, at data vil ankomme via den PCI-X bus[14], der sidder på udviklingskortet. Da data på PCI-X bussen sendes i pakker af 2^n x 64 bit, er der valgt at benytte et 256 bit interface til Hybris systemet, da det derved er muligt at overføre én trekant per periode.
Hybris systemet skal i den nuværende udgave kun bruge datapakker på 234 bit, hvilket derved giver et spild på 22 bit per pakke, men disse bit kan eventuelt benyttes i en fremtidig udvidelse.

3.1.1 Implementering

Hybris systemets indgang er desuden udstyret med 3 styresignaler, som har til formål at signalere om en gyldig pakke er klar, og om Hybris systemet er parat til at tage imod pakken. Endelig er der et signal, der fortæller, om datapakken er en kommando pakke, eller en pakke med en trekant. Det samlede interface ses i tabel 2.

Det konstruerede interface giver mulighed for at tilføje en pakke dekoder umiddelbart foran Hybris systemet, som håndterer kommunikationen med PCI-bussen, eller et hvilket som helst andet interface, som der måtte benyttes.
Sidegevinsten ved dette design er, at et hurtigere interface nemt kan kobles på systemet, uden at en eneste ændring af Hybris systemet er nødvendigt, eftersom der skal sidde en pakke-dekoder foran indgangen. Eventuelt kan det tænkes, at AGP8 bussen vil være en fremtidig mulighed[15].

LegalPackage og *VertexPresent* signalerne er ført ind som selvstændige signaler for at tillade, at alle 256 bit i pakken bliver brugt til data. I et scenario, hvor en PCI bus bruges, skal disse

[14] PCI-X bussen har en båndbredde på maksimalt 528MB, med 64bit bus og 66MHz datafrekvens. Yderligere PCI specifikationer kan findes på PCI-SIG's hjemmeside www.pcisig.com
[15] AGP8-bussen tilbyder en båndbredde på 2.1GB per AGP8 bus, og der tillades flere separate busser [8]

dog maskeres ind i datapakken. Bit 0 til 233 er reserveret til de tre koordinatsæt, og bit 255 må heller ikke benyttes, da denne bruges til at signalere, at sidste trekant i billede er modtaget.

Signal	I/O	Bredde (bit)	Kommentar
LegalPackage	I	1	Højt, når en gyldig datapakke er modtaget og parat til Hybris systemet
Saturated	O	1	Højt, når Hybris systemet ikke kan modtage nye data
VertexPresent	I	1	Højt, når den modtagne pakke er en trekant, lavt hvis det er en kommando pakke
VertexPackage	I	256	Indgående datapakke

Tabel 2: Hybris systemets indgangs-interface. I/O signalerne er set fra Hybris systemets synspunkt

3.2 Master Unit

For at styre, at både *SortUnit* og *BackEnd* kan køre samtidigt, er det nødvendigt med en enhed, som kan holde øje med, hvornår de to enheder er færdige med hhv. at skrive og læse trekanter fra den fælles eksterne hukommelse.
Dette gøres med en fælles proces, som kommunikerer med de to enheder. Denne proces kaldes for *Master Unit* og foruden kommunikationen med de to enheder, modtager *Master Unit* også data fra pakke dekoderen, vha. det i tabel 2 beskrevne interface.

Master Unit har til opgave at styre disse tre kommando signaler, således at frame skift sker korrekt:

- Flush
- ChangeBanks
- Go

Når Hybris systemet starter vil *Master Unit* vente på at modtage enten en kommando pakke eller en Vertex pakke. Hvis det er en kommando pakke, vil denne blive forsøgt dekodet[16]. Er det derimod en Vertex pakke, vil *Master Unit* lade denne passere igennem til *SortUnit*, som vil begynde at indsætte trekanten i de korrekte tiles.
Det forventes, at alle datapakker er Vertex pakker, indtil en pakke med et *LastVertex* signal modtages. Dette signal er defineret til at være bit 255 i data pakken, og skal derfor være 0, med mindre det er sidste trekant i et billede.

Efter at *LastVertex* er modtaget sendes en *Flush* kommando til *THTB Insertion*, dog ikke før alle enheder i *SortUnit* ikke længere indeholder data. Det sker for at sikre, at alle trekanter er sendt til den eksterne hukommelse. Efter, at en flush er udført, ventes der på, at *BackEnd* bliver færdig med at indlæse trekanter fra den eksterne hukommelse. Når alle trekanter er indlæst, vil *Master Unit* sætte *ChangeBanks* højt. Derpå vil de to TPT tabeller[17] blive skiftet, og *TPT Reset Engine* vil gå i gang med at nulstille den tabel, som *THTB Insertion* senere skal

[16] Kommando dekoderen er endnu ikke implementeret, så pakken vil blive ignoreret i den nuværende implementering
[17] Se 4.3

bruge. Den anden tabel vil, efter en periode, stå klar til *BackEnd*, som vil gå i gang med at indlæse nye trekanter efter at have modtaget en *Go* kommando, hvilket sendes af *Master Unit* som næste kommando.
Efter at *Go* kommandoen er udstedt, går Master Unit tilbage til Idle tilstanden, og nye pakker kan sendes igen.

På bilag 3 ses algoritmen som *Master Unit* benytter.

Den opløsning, som systemet skal arbejde med, bestemmes af *Master Unit*. Hele systemet er designet til at kunne arbejde med opløsninger på op til 2048 x 2048 pixels, svarende til 64 x 64 tiles. Da ingen i dag benytter en så høj opløsning, og det desuden ikke er muligt for standard skærme at vise en sådan opløsning, er der indbygget mulighed for at angive, hvilken opløsning der ønskes at arbejde med. Denne skal angives i antal tiles i x og y retningen, da det ikke er muligt at skære midt i en tile. Opløsnings information bliver videregivet til de relevante enheder, som kan bruge informationen til at bortkaste de data, som pga. begrænset opløsning, ikke skal tegnes.
Vilkårlige opløsninger supporteres, og det tjekkes ikke, om opløsningen har et 4:3, 16:9 eller andet format.
Da *Master Unit* endnu ikke kan fortolke kommando pakker, er det ikke muligt at arbejde med en dynamisk opløsning. Det skal pt. gøres i *Master Unit*'s kildekode.

3.3 DDR-SDRAM

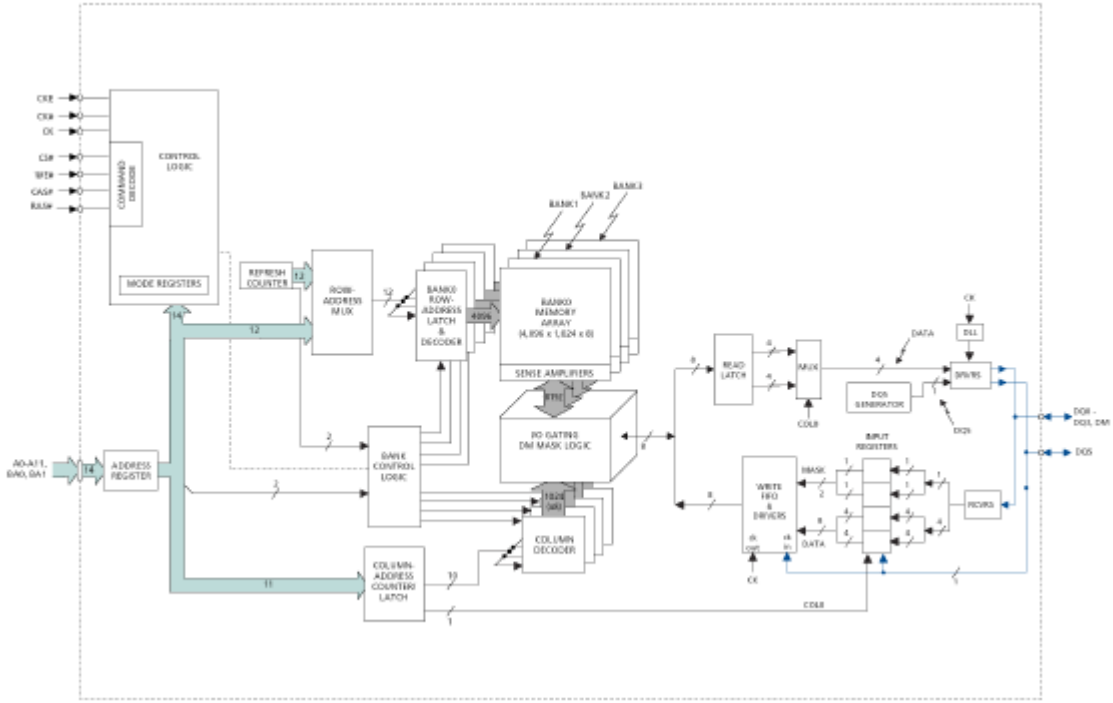
Som ekstern hukommelse anvendes det DDR-SDRAM modul, som sidder på udviklingsboardet.
Adressering og signalering af SDRAM og DDR-SDRAM er næsten identisk, DDR-SDRAM overfører blot 2 sæt data per periode, modsat SDRAM som kun overfører et sæt data, så beskrivelsen dækker principielt begge typer.
De reelle forskelle mellem de to typer nævnes i afsnit 3.4.

3.3.1 Beskrivelse af DDR-SDRAM hukommelsen

DDR-SDRAM er en hukommelsestype, der på en forholdsvis simpel - og billig - måde kan gemme store mængder data.
Data er dog ikke direkte tilgængelige, da adresse bussen er på 12 bit, hvilket betyder, at adressen deles i 2, så en adressering tager 2 perioder. Desuden er hukommelsen delt op i 4 data-banker, hvor der i hver bank yderligere er en række-søjle opdeling af hukommelses områderne, hvilket ses illustreret på figur 9.
Adressering sker ved at sende adressen over to perioder, hvor 4 styresignaler bruges til at fortælle hukommelsesmodulets styrekredse, om det er en række eller søjleadresse, der sendes.
Efter en afsending af en række eller søjle adresse, skal der ventes et givent antal perioder, inden den ønskede række/kolonne er tilgængelig. Da det tager omkring 20 ns at indeksere i hukommelsen, skal der efter hver adressering ventes 2,5 eller 3 perioder[18], før næste adresse kan sendes, forudsat at der benyttes en frekvens på 133MHz.

[18] Antallet af perioder, der skal ventes, kaldes for CAS (Column Address Strobe)

Når der først er åbnet til det ønskede data-område, skal der skrives eller læses i et givent antal perioder[19], hvor der er mulighed for at overføre i 1, 2 eller 4 perioder.
Data ind og udlæses 64 bit ad gangen, og det sker ved begge flanker af clock-signalet, hvorved 2 dataoverførsler opnås på én periode.
De 4 styresignaler kan bruges til at sende andre informationer end adresser, hvorfor disse kaldes for kommandoen.



Figur 9: Funktionelt blokdiagram. © Micron[9]

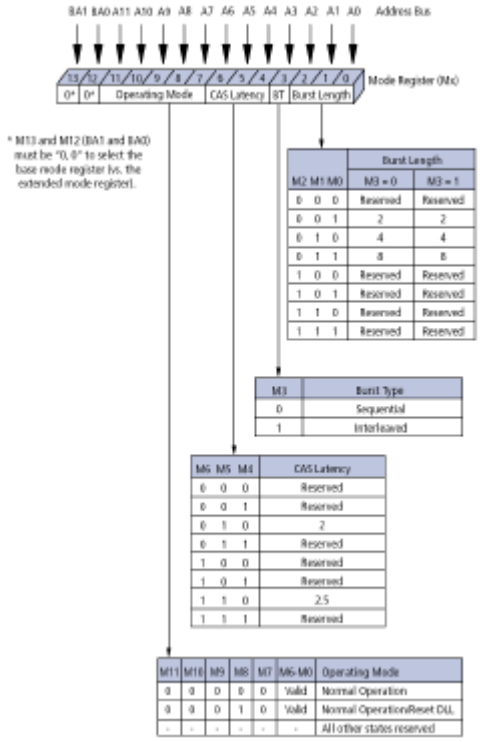
Grunden til at data overføres, med en dobbelt frekvens af resten af signalerne, skyldes analoge elektriske forhold. Med en længere periodetid giver dette mindre krav til signalernes stig og fald tider, hvilket gør signalerne mere tolerante over for støj fra andre signalkilder, som der typisk er mange af i digitale miljøer. Et mindre strømforbrug opnås desuden som "sidegevinst", da mange af signalerne kun skifter halvt så hyppigt som datasignalet.
Da data overføres dobbelt så hurtigt, er de noget mere følsomme over for støj, hvilket er mindst lige så kritisk som korruption af de andre signaler. For at imødegå dette, har alle datasignalerne et jordsignal som nabo-leder, hvilket sænker risikoen for støj markant [10].

Da hukommelsen er dynamisk, dvs. at de lagrede data efter relativt kort tid vil forsvinde, er det nødvendigt periodisk at genopfriske hukommelsen. Dette skal gøres af DDR-SDRAM controlleren, som skal sørge for at sende en opfrisknings kommando i intervaller, der ikke overstiger 15,625µs. Overholdes dette ikke, er der risiko for datatab[20].
Da hukommelsen på udviklingsboardet er beregnet til at køre med en frekvens på 133MHz, betyder det, at en 12 bit tæller kan bruges til at styre opfriskningen.

[19] Antallet af overførsler kaldes "Burst Length", og er 2,4 eller 8, da der kan overføres 2 gange på en periode
[20] Det er dog muligt at sende flere "REFRESH" kommandoer af sted på én gang, således at controllerens refresh interval kan øges – denne mulighed benyttes dog ikke i det nuværende design

3.3.2 Dataoverførsel

Data overføres via en 64 bit bidirektionel bus, hvor kommandoen til hukommelsen bestemmer, hvem der sender, og hvem der modtager. Hukommelsen skal bruge tre kommandoer for at kunne bestemme overførselstype og retning.
Først skal hukommelsen have en kommando, der angiver størrelsen af datapakken, som skal overføres, og på hvilken måde denne skal vælges. Dette gøres ved at sende et ”Mode Register” til hukommelsen, hvor disse info overføres via adresse bussen. Figur 10 herunder beskriver opsætningen af registeret. ”Mode Register” skal kun sendes, hvis hukommelsen skal sende med en anden pakkelængde, eller vælge dataraekkefølge anderledes. Hvis der kun er forskel på, hvilken vej data skal flyde, skal ”Mode Register” ikke ændres.
Der er endnu et mode register, ”Extended Mode Register”, som primært bruges, når hukommelsen skal nulstilles, efter at strømmen er tilsluttet. Dette er ikke forklaret yderligere her, hvorfor der henvises til [11].
Implementeringen, i dette design, bruger kun burst-længder på 4 eller 8, og sekventiel data overførsel, men principielt er alle overførselstyper understøttet.



Figur 10: ”Mode Register” diagram. © Micron [9]

Den anden kommando, hukommelsen skal modtage, angiver den række, som der ønskes data fra eller til.
Sidste kommando indeholder information om hvilken vej data skal flyde, foruden anden halvdel af adressen.
I tabel 3 ses, hvilke kommanoder der kan sendes. De kommandoer, der bruges i dette design, er markeret med fed. Nogle af kommandoerne kan have flere options, men de er ikke beskrevet her. I stedet henvises til [11].

Funktion	Kommando			
	CS#	RAS#	CAS#	WE#
Deselect /NOP	H	X	X	X
No Operation /NOP	L	H	H	H
ACTIVE	L	H	H	H
READ	L	L	L	H
WRITE	L	H	L	L
BURST TERMINATE	L	H	H	L
PRECHARGE	L	L	H	L
AUTO/SELF REFRESH	L	L	L	H
LOAD MODE REGISTER	L	L	L	L

Tabel 3: DDR-SDRAM kommandoer

Når en læsning eller skrivning er tilendebragt efterlades hukommelsescellen åben, og vil derfor ”spærre” for adgang til resten af hukommelsen i samme bank. Derfor skal en yderligere kommando sendes til hukommelsen, der fortæller at cellen skal lukkes. Dette kan dog også gøres automatisk, ved at en bestemt bit i adressen er høj (A10), når læse/skrive kommandoen sendes, hvorved hukommelsesbanken automatisk lukkes efter en dataoverførsel.

Da de 4 banker i hukommelsen fungerer uafhængigt af hinanden, er det muligt at udnytte båndbredden næsten fuldt ud, da det er muligt at sætte næste dataoverførsel op, inden den igangværende overførsel er færdig.
Eftersom hukommelsen nominelt kører med 133MHz, kan en overførselshastighed på næsten 2100MB per sekund opnås[21].

3.3.3 Implementering af DDR-SDRAM controller

Controlleren i dette design består af to enheder. *DDR controller*, som styrer datastrømmen mellem FPGA og ekstern hukommelse, og *DDR interface*, som styrer de enkelte enheders adgang til hukommelsen.
Controlleren bygger på et Xilinx design [12], og er kun ændret på 2 punkter, hvilket fremgår af tabel 4.
Xilinx interfacet benytter et 128 bit interface, for at kunne sende og modtage data hurtigt nok til at aflevere 2 pakker á 64 bit på en periode.

Signal	VHDL fil ændret	Ændring
u_addr	addr_latch.vhd	Udvidet til 23 bit, for at kunne adresser alle 128MB. Før var adressebussen kun 22 bit
u_ref_ack	controller.vhd	Ændret til at være høj, når controlleren ikke er i Idle tilstand, så det nu kan detekteres om controlleren er *busy*

Tabel 4: Ændringer i Xilinx designet

Xilinx designet er desværre ikke i stand til at styre adgangen til hukommelsen intelligent, så det er ikke muligt at sætte adresser op, mens data overføres. Dette betyder at båndbredden, ifølge Xilinx, falder til ca. 570MB/sek. [12], eller ca. 27%.

[21] Af samme årsag kaldes denne type hukommelse også for PC2100

Af hensyn til muligheden for at teste systemet i SystemC, er der lavet en kombineret DDR-SDRAM/Xilinx Controller emulator, som signal og timingmæssigt opfører sig korrekt, således at software simuleringerne giver et korrekt billede, af den tid, det tager at overføre data.

Da der er flere dele af Hybris systemet, der gemmer og henter data i hukommelsen, er det nødvendigt med et internt interface, der kan bestemme, hvilken enhed der har råderet over hukommelsen. Til det formål er der udviklet et generelt interface, som har 5 indgange, hvoraf den ene er prioriteret.
Da Xilinx's interface ikke har en "refresh counter", er interfacet derfor udstyret med en sådan. Skematisk kan hele interfacet ses på bilag 4.

Hver indgang har en I/O buffer, hvor data ind eller ud placeres. Derved kan interfacet bruges til at adskille eventuelle forskellige clock-domæner, se afsnit 3.7.
Modulet har en prioriteret indgang (Port 1), som kan bruges af en enhed, som skal bruge data inden for kort tid, derfor vil forespørgsler fra denne indgang blive serviceret før de andre fire. De andre indgange vil få adgang til bussen efter tur.
"Refresh counteren" har dog førsteret til hukommelsen, for at sikre at DDR-SDRAM hukommelsen bliver genopfrisket.

Den prioriterede indgang var oprindelig tænkt til frame controller modulet, men da den, forholdsvis sent i designfasen, blev ændret til at bruge den eksterne SDRAM, bliver denne ikke brugt, da vigtigheden af de andre enheder kan sidestilles.
Da modulet er lavet, og eventuelt kan bruges i fremtidige udvidelser, er designet bibeholdt.
De fem indgange fungerer uafhængigt af hinanden, og de blokke, der kommunikerer med DDR interfacet, behøver derfor ikke at tage hensyn til de andre blokke, der også kommunikerer med hukommelsen.

Interfacet imellem *DDR Interface* og de eksterne blokke består af et simpelt interface, som gør det yderst nemt at tilslutte enheder. Der er ikke nogen decideret kommunikationsprotokol, som skal overholdes af de tilkoblede moduler. Der er kun et simpelt *REQ* signal, som sættes højt, når en data transaktion ønskes gennemført. Dette skal være højt indtil *Busy* bliver høj, hvilket betyder at transaktionen er påbegyndt. *Busy* kan sænkes igen, når transaktionen er færdig.

Interface signalerne er beskrevet i tabel 5. Det bemærkes at data, ved en skrivning, skal skrives til I/O bufferen før *REQ* sættes højt, ellers vil data ikke være klar på udgangen af I/O bufferen.

Signal	Bredde	I/O	Beskrivelse
In_data	128	I	Databus
Out_data	128	O	Databus
In_addr	22	I	Adressebus
In_REQ	1	I	Request. Sættes høj, når en overførsel ønskes påbegyndt
In_Longburst	1	I	Høj, hvis 512 bit ønskes overført, og lav hvis 256 bit ønskes overført
In_read	1	I	Høj, hvis en læsning ønskes foretaget, lavt hvis en skrivning ønskes foretaget
RegEN	1	I	Signal, om at I/O bufferen skal clockes
Out_busy	1	O	Høj, mens en overførsel foretages

Tabel 5: DDR-SDRAM Interface. I/O signaler er set fra interfacets side

Der er, i det implementerede design, tilkoblet 4 enheder. 2 indlæser data til hukommelsen og deles om to banker, mens 2 andre deles om de to sidste banker. *BankSelect* signalet bruges til at skifte bankerne ved billedskift, så de fire tilkoblede enheder ikke skal tage hensyn til, hvilken bank de skriver til, men kan bruge én fast adresserings model. Af samme årsag er adresse bussen kun 22 bit, hvilket giver mulighed for at adressere 64MB.

Det er valgt at lade *SortUnit* bruge *AddressSpace B* (Port 4..5), mens *BackEnd* bruger *AddressSpace A* (Port 1..3).

Den indbyggede tilstandsmaskine håndterer kommunikationen, og fremgår af bilag 5. Implementeringen sikrer, at ingen enhed – bortset fra den prioriterede – kan få eksklusiv adgang til hukommelsen ved at beholde *REQ* signalet højt, da *REQ* signalet for næste port undersøges, når en data overførsel er tilendebragt
Tilstandsmaskinen sørger desuden for at udstede en REFRESH kommando inden for den garanterede tid, uanset om flere enheder vil serviceres.

3.3.4 Initialisering af DDR-SDRAM

I tilstandsmaskinen er der integreret en funktion til at initialisere DDR hukommelsen, da denne skal foretages én gang ved opstart, for at DDR hukommelsen virker korrekt.

Følgende trin skal gennemløbes for at initialisere korrekt:

- Spændinger skal være stabile, og korrekte, og CKE skal være lav
- 200µs skal gå, før kommandoer må sendes til controlleren
- Send PRECHARGE ALL kommando
- Send LOAD (extended) MODE REGISTER kommando, med DLL = 1
- Send LOAD MODE REGISTER med korrekte data og Reset DLL = 1
- Vent 200 perioder
- Send PRECHARGE ALL
- Send AUTO REFRESH (2 gange)
- Send LOAD MODE REGISTER med korrekte data og Reset DLL = 0

Tilstandsmaskinen gennemgår disse trin umiddelbart efter reset, på nær de første punkter, som det ikke er muligt at udføre, da spændingerne allerede er stabile, før FPGA'en kan benyttes[22]. Dermed kan det ikke garanteres, at det I/O ben på Xilinx chippen, som er bundet til hukommelsens CKE ben, er lavt. Det betyder, at I/O signalerne DQ og DQS på DDR-SDRAM kredsen ikke umiddelbart kan garanteres at være i "High-Z" efter initialisering[23]. En WRITE kommando efter initialiseringen vil da kunne resultere i, at både DDR-SDRAM hukommelse og controller prøver at styre signalerne, hvilket i værste fald kan medføre uoprettelige systemfejl. Det formodes, at udviklerne af hardwaren er bekendt med dette problem, og det derfor ikke er noget reelt problem, som det er nødvendigt at tage hensyn til.

[22] DC spændingen skal nå et stabilt niveau på mellem 200µs og 50ms [13]
[23] Jfr. JEDEC specifikationerne, vedrørende initialisering [11]

3.4 SDRAM

3.4.1 Forskelle mellem SDRAM og DDR-SDRAM

DDR-SDRAM er en videreudvikling af SDRAM. Det betyder, at de to hukommelsestyper er meget ens, og den største forskel er, at DDR-SDRAM overfører 128 bit per periode modsat 64 bit for SDRAM.

Pga. det nære slægtskab mellem de to typer, benytter de stort set samme instruktionssæt, dog er de elektriske specifikationer noget anderledes, hvorfor de to typer ikke er direkte kompatible.

De forskelle, som der skal tages højde for i et digitalt design[24], ses i tabel 6. Da forskellene ikke er større, kan udviklingstiden sænkes betragteligt, da det er nok at lave og teste et interface til den ene hukommelsestype. Når denne er verificeret, kan forskellene i tabel 6, udnyttes til at justere designet til at virke med den anden hukommelsestype. Test af dette nye design skulle da kunne overstås hurtigt, da signaltimingen er ens for modellerne, bortset fra de signaler, som har med dataoverførslen at gøre.

Punkt	Forskel
Mode register	SDRAM har ikke det udvidede ”Extended Mode Register” register. ”Mode Register” er derimod ens for de to typer. Der er dog visse kommandoer, som ikke kan benyttes af begge typer
Burst-længde	DDR-SDRAM kan lave burst på 2, 4 eller 8 datablokke, mens SDRAM kan lave 1, 2, 4, 8 eller full-page
CAS	SDRAM har et setup delay på 2 eller 3 perioder, hvor DDR-SDRAM har 2,5 eller 3

Tabel 6: Forskelle mellem SDRAM og DDR-SDRAM [9, 14]

3.4.2 SDRAM controller

Som nævnt, er der ikke mange forskelle på DDR-SDRAM og SDRAM, hvilket kan udnyttes til at omdanne DDR-SDRAM controlleren til en SDRAM controller. Denne viden kan udnyttes til at lave et næsten identisk interface ind mod de interne Hybris moduler, hvorved der kun skal udvikles ind imod et fælles interface. Forskellene på de to interface-modeller vil være adressebredden, som kun skal være på 21 bit, eftersom der kun er 64MB til rådighed. Desuden skal databussen kun være 64 bit, da der kun overføres 64 bit per periode. Det samlede interface fremgår af tabel 7.

[24] De hardwaremæssige designforskelle er noget større, men udeladt her

Signal	Bredde	I/O	Beskrivelse
In_data	64	I	Databus
Out_data	64	O	Databus
In_addr	21	I	Adressebus
In_REQ	1	I	Request. Sættes høj, når en overførsel ønskes påbegyndt
In_Longburst	1	I	Høj, hvis 256 bit ønskes overført, og lav hvis 128 bit ønskes overført.
In_read	1	I	Høj, hvis en læsning ønskes foretaget, lav hvis en skrivning ønskes foretaget
RegEN	1	I	Signal, om at I/O bufferen skal clockes
Out_busy	1	O	Høj, mens en overførsel foretages

Tabel 7: SDRAM Interface. I/O signaler er set fra interfacets side

Da det er tiltænkt, at SDRAM modulet skal bruges af frame-controlleren, er det kun nødvendigt med to porte; en til at skrive tiles ud til hukommelsen og en som kan bruges af frame-controlleren, til at danne billeder til skærmen. Dermed bliver 3 af DDR-SDRAM-interfacets porte overflødige og kan med fordel fjernes, så det kun tager en periode at gennemgå portene.

3.5 Hukommelses modeller

3.5.1 DDR-SDRAM

Da både *BackEnd* og *SortUnit* skal benytte DDR hukommelsen, er det valgt at afsætte 2 banker til hver, hvilket giver hver enhed 64MB hukommelse, som udnyttes ens. Disse to banker byttes ved billed-skift, så der kontinuerligt kan både læses og skrives data til hukommelsen.

Da DDR-SDRAM adresseres i spring af 128 bit, er det nødvendigt med 23 bit for at adressere alle 128MB, da:

$$128\text{MB} = 1024\text{Mbit} = 1.073.741.824\text{bit} = 128 \times 2^{23} \text{ bit.}$$

64MB adresseringen kræver da 22 bit, plus en bit til at indikere, hvilke hukommelses banker, der ønskes benyttet. Sidste bit skjules for enhederne, hvorved de kun skal nulstille deres adresse-tællere ved billedskift.
Da det kun er DDR interfacet, der bruger den sidste bit, udelades den i den videre model, som beskriver, hvordan 64MB blokken udnyttes.

Hver THTB buffer fylder 512 bit i hukommelsen, så de to mindst betydende adresse bit kan derfor fjernes, eftersom det vides, at de altid vil være 0. Dette giver, at en THTB buffer kan adresseres med 20 bit.

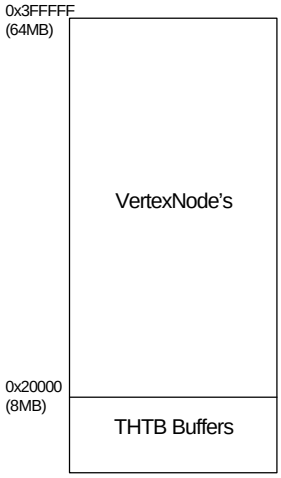
Hver trekant fylder 256 bit, hvilket, med argumentationen ovenover, giver 21 adressebit for en VertexNode.

Som det forklares senere, kan der være 23 VertexNode referencer i hver THTB buffer. Hovedparten af THTB buffere vil være fyldte, hvis en scene indeholder mange trekanter. Derfor vil der være 23 gange så mange VertexNode's som THTB buffere, men da en THTB buffer fylder det dobbelte af en VertexNode vil pladsforholdet mellem de to strukturer være 1/11,5.
Det betyder, at det optimale forhold til de to strukturere vil være:

VertexNode:	$64MB \cdot (1 - 1/11.5)$	=	58,4MB
THTB Buffer:	$64MB \cdot (1/11.5)$	=	5,57MB

Det er derfor valgt at afsætte 8 MB til THTB buffere og 56MB til VertexNode data, for at runde af til 2^n [25].

THTB bufferne er blevet placeret i den første del af hukommelsen, hvilket illustreres af figur 11. Dette medfører en yderligere reduktion på 3 bit i det nødvendige antal bit, hvilket dermed sænker det nødvendige antal bit til 17.



Figur 11: Hukommelses model

17 bit giver mulighed for 131072 THTB buffere, men adresse 0x1FFFF kan ikke bruges, da denne adresse bruges af *SortUnit* til at indikere, at et tabel-indeks ikke indeholder gyldige data. Derfor er der reelt plads til 131071 THTB buffere, svarende til 3.014.633 VertexNode referencer, hvilket, i rigelig grad, dækker behovet, da den store Buddha [2c, 15] figur kun indeholder 1.09 millioner trekanter, og kun omkring halvdelen efter culling.

De resterende 56MB giver plads til 1.835.008 VertexNode's, hvilket dækker behovet.

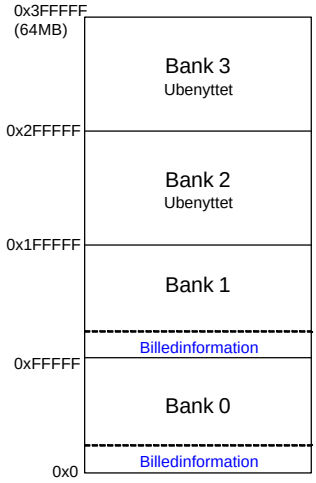
Hvis en VertexNode senere skal indeholde mere information, vil det være nødvendigt at benytte pakker af 512 bit. Det vil halvere antallet af mulige VertexNode's, men med godt 900000 VertexNode's vil der stadig være nok til at kunne tegne selv meget komplekse

[25] Nedrunding, af THTB buffer størrelsen ville medføre, at der ville være plads til flere VertexNode's, end THTB bufferne ville kunne holde styr på

objekter. THTB buffer pladsen kan reduceres til 4MB, hvilket vil give mere plads til VertexNode's, da bufferne stadig vil kunne referere til ca. 1,5 millioner trekanter.

3.5.2 SDRAM

Som tilfældet med DDR-SDRAM skal SDRAM hukommelsen også deles i to, da der er to enheder, der ønsker at benytte den samtidigt. Hukommelsen deles som i DDR-SDRAM tilfældet, dvs. der laves 2 identiske blokke, som skifter ved hvert frame skift, som vist på figur 12.



Figur 12: Illustration af SDRAM opdeling

Hukommelses kravet for en blok i SDRAM er, at der skal være plads til et billede på 2048x2048 punkter, hvilket svarer til 4.194.304 billedpunkter. Da det nuværende design kun understøtter nuancer af en farve, svarer det til, at der skal afsættes 4MB til billedpunkterne, og da SDRAM gemmer i pakker á 64 bit, vil der være 8 pixels i hver pakke, hvilket giver en 100% pladsudnyttelse af hukommelsen.
Med denne fordeling af hukommelsen, er det ikke nødvendigt at benytte mere end to hukommelsesbanker á 16MB. Derfor vil der være 32MB ubundet hukommelse, som eventuelt kan benyttes efter behov i senere udvidelser. Der skal blot tages i betragtning, at der skal være en garanteret båndbredde til rådighed for frame-controlleren, så den kan generere 60 eller flere skærmbilleder per sekund.

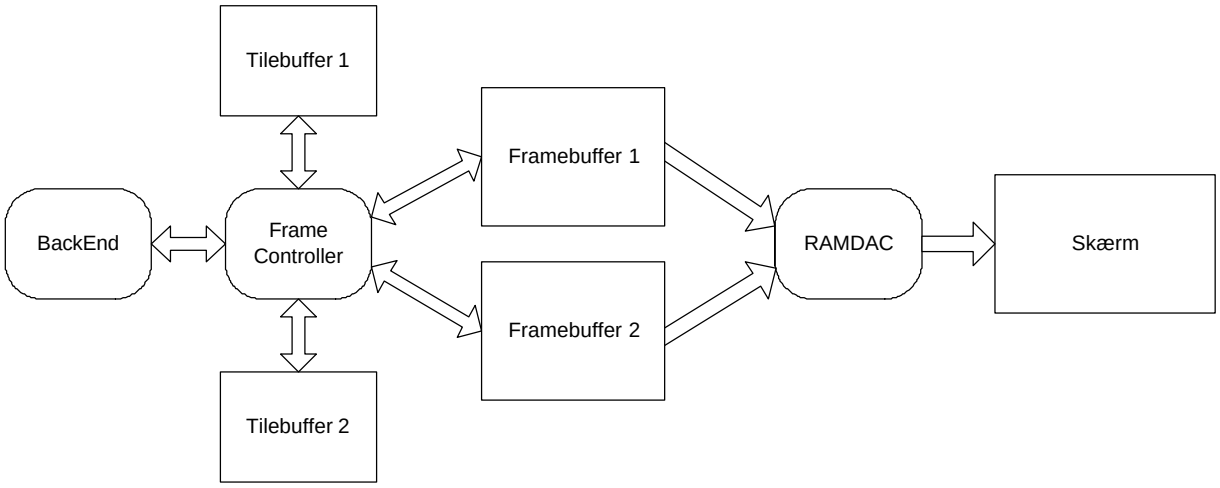
Hvis der udvides med farver, vil der kun være plads til 2 pixels i hver pakke, hvis det ønskes at gemme alle 3 farver samlet. Derfor skal der afsættes 16MB til et skærmbillede, hvilket vil sige, at der spildes 25% af hukommelsen.
Modellen skal til gengæld ikke laves om, da der i forvejen er 16MB til hver frame. Det er da kun nødvendigt at ændre på, hvor de enkelte pixels ligger placeret i hukommelsen.

3.6 Framecontroller

Frame controllerens opgave er:

- Sørge for at data fra pixel beregningen ender i den rigtige tile buffer
- Kopiere tile bufferen til frame bufferen
- Sørge for at der bliver skrevet til den rigtige frame buffer
- Udlæse frames til skærmen

Framecontrolleren tænkes at bestå af 2 tile buffere, 2 frame buffere og en controller, som vist på figur 13, hvilket svarer til de tidligere implementeringer.



Figur 13: Framecontroller

Af de 2 tile buffere benyttes den ene til at gemme farve og z-værdier for den tile, som *BackEnd* er i gang med at beregne. Samtidigt kan den anden buffer, som indeholder en færdigberegnet tile, udlæses til frame bufferen.

Frame bufferen er bygget op på samme måde, sådan at der er en buffer, som er ved at blive fyldt med tiles, samtidigt med at den anden kontinuerligt udlæses til skærmen.

Under kopiering fra tile buffer til frame buffer, skal controlleren også håndtere de hukommelses adresser, der skrives til. Dette er ikke helt simpelt, da data i frame bufferen, skal være ordnet sådan, at hver hele linie på skærmen ligger i en sammenhængende klump i den eksterne hukommelse. På denne måde undgås det, at skærmopdaterings enheden skal være intelligent. Problemet er, at hver tile kun er 32 pixels bred, og derfor skal der, for hver linie i tilen, laves et spring i antallet af hukommelses adresser. Springets størrelse afgøres af bredden af skærmens opløsning.

3.7 Clock domæner

DDR-SDRAM båndbredden er proportional med frekvensen. Derfor er det vigtigt, at DDR-SDRAM modulet kører hurtigst muligt. Da hukommelses modulet, på udviklings kortet, kan køre med 133MHz, er det ønskeligt også at benytte denne frekvens.
De udviklede moduler kan til gengæld ikke alle køre med denne frekvens, hvorfor det er nødvendigt at have flere clock-domæner.
Korrekt signal overgang mellem domæner er vigtig, da det ellers kan betyde signalkorruption, og i dette projekt foregår overgange mellem de to domæner med FIFO buffere[26], hvor indgang og udgang har hvert sit clock-signal. FIFO bufferne er Xilinx Cores, og beskrives ikke yderligere her. Det skal blot konstateres, at disse buffere bruges til adskillelse af clock-domæner.
Eneste undtagelse fra denne adskillelse er *Master Unit*, der i denne implementering tænkes at køre med 66,5MHz. Denne kommunikerer både med 133MHz og 66,5MHz enheder, men under udviklingen er der taget hensyn til dette, og kommunikation er baseret på handshaking.
66,5MHz vælges, da det svarer til en halvering af hukommelsens frekvens, og samtidigt med giver det en frekvens, som er et rimeligt mål for alle beregningsenheder.
I implementeringen er den langsomme frekvens en halvering af den hurtige, og de er dermed synkrone. Det er ikke undersøgt, om designet vil virke med asynkrone perioder.

[26] DDR-SDRAM interfacets I/O buffer bruges til at adskille *VertexNode Reader* og DDR-SDRAM interfacet

3.8 Sammenfatning

Dette kapitel beskrev, hvordan det udviklede system tænkes at kommunikere med en fremtidig pakke dekoder. Der er udviklet et generelt interface, således at det udviklede system ikke er afhængig af en specifik bus.
Dette interface styres af *Master Unit*, som også kontrollerer, at de to Hybris enheder bruger de korrekte hukommelsesblokke.

Det er beskrevet, hvordan DDR-SDRAM virker signalmæssigt, og hvordan et interface, udviklet af Xilinx, er implementeret. Desuden beskrives det interne DDR-SDRAM interface, der er udviklet, og som tillader op til 5 enheder at hente og gemme data i hukommelsen, uden at hver enhed skal tage hensyn til de andre.

Forslag til implementering af en SDRAM controller er ligeledes beskrevet, og det bliver foreslået, hvordan det udviklede DDR-SDRAM interface kan ændres. Dette kan gøres forholdsvis nemt, da SDRAM og DDR-SDRAM kommandoerne er kompatible.

De hukommelsesmodeller, som senere bruges af *SortUnit* og *BackEnd*, beskrives, og det udledes hvor mange bit, der skal bruges af de forskellige enheder, for at kunne adressere data.

Der er foreslået, hvad en fremtidig *Frame controller* skal indeholde af enheder, for at kunne arbejde sammen med *BackEnd*.

Endelig følger et kort afsnit, som beskriver, hvordan FIFO buffere, designet af Xilinx, bruges til at separere moduler med forskellige frekvenser.

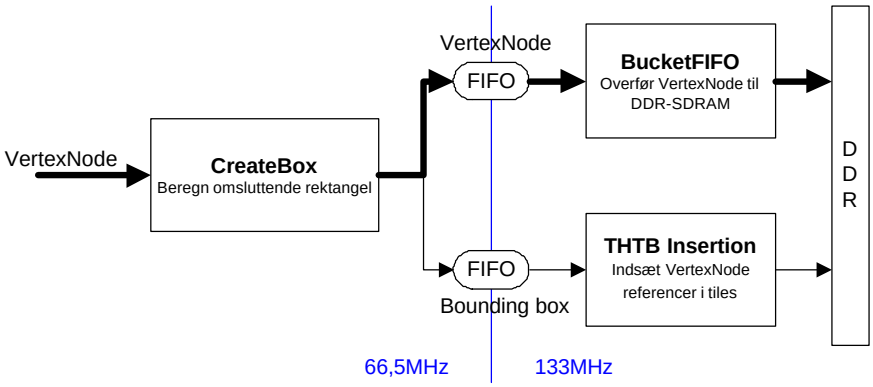
4 SortUnit

I dette kapitel beskrives den midterste del af Hybris systemet, *SortUnit*.
Enheden skal sortere de indkommende trekanter ud i de såkaldte tiles, som Hybris opdeler en skærm i. Der er tre hoveddele i *SortUnit*:

- *CreateBox* udregner, hvilke dele af skærmen en trekant dækker
- *BucketFIFO* overfører en trekant til den eksterne hukommelse
- *THTB Insertion*, som indsætter referencer til trekanter i de lister, som refererer til de enkelte dele af skærmen

Hver del beskrives i dette kapitel, og i figur 14 ses en figur, som viser sammenhængen mellem de tre dele.

I afsnittet om *CreateBox* beskrives hvorfor en opdeling i tiles er nødvendig. Det beskrives desuden, hvordan denne opdeling er lavet i hardware.
Derpå følger et kort afsnit om, hvordan *BucketFIFO* er implementeret.
THTB Insertion enheden beskrives, og det forklares, hvorfor der er implementeret en cache.
Til sidst beskrives de parallelt kørende processer i *THTB Insertion*.



Figur 14: SortUnit

4.1 CreateBox

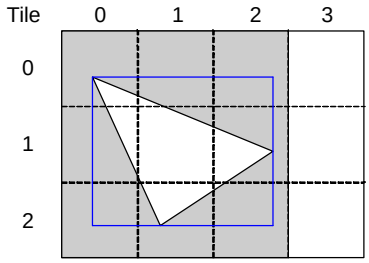
4.1.1 Omsluttende rektangel

Da Hybris supporterer en opløsning på op til 2048 x 2048 billedpunkter, vil det, uden opdeling af skærmen, betyde at der skal kunne adresseres direkte til alle billedpunkter, hvilket vil kræve en 4MB hukommelse til billedpunkterne. Dertil kommer z-bufferen, som er 32 bit dyb, hvilket kræver yderligere 16MB hukommelse.
Det vil i alt give et hukommelseskrav på 20MB, som skal kunne adresseres inden for én periode.

Det er endnu urealistisk med så store on-chip hukommelser[27], og ingen gængse hukommelsestyper kan adresseres direkte inden for én periode. Derfor må andre metoder tages i betragtning, for at kunne bruge den hukommelse, der er til rådighed.

Løsningen, i forbindelse med Hybris systemet, har været at inddele skærmen i mindre områder [2d]. Disse områder – kaldet en tile – er et kvadratisk udsnit på 32x32 pixels, eller i alt 1024 pixels.
Herved sænkes hukommelseskravet dramatisk til 1024 bytes til skærmbufferen, og 4KB til z-bufferen, eller i alt 5KB for én tile. Da der er noget mere hukommelse til rådighed end 5KB i Xilinx chippen, åbner det for muligheden af at have flere samtidige tiles ad gangen, med en betragtelig hastighedsforbedring til følge.

Med en tile på 32x32 pixel, opdeles skærmen i 64x64 områder (4096 i alt), og som antydet bliver hver trekant sorteret i forhold til, hvilke områder trekanten dækker over.
Hybris er bygget op omkring en algoritme, der benytter en ”ikke-helt-nøjagtig” sortering. Algoritmen er simpel og går ud på at danne et rektangel uden om trekanten. Dette rektangel skal netop omslutte trekanten, hvorfor rektanglet da også kaldes et omsluttende rektangel, eller Bounding Box på engelsk.
Som det fremgår af figur 15, vil det omsluttende rektangel også ramme tiles, som trekanten ikke ligger i. I disse tiles vil der også blive lagt en reference til trekanten, selvom denne er overflødig.
En nøjagtig sortering er mulig, men kræver nogle flere beregninger, og er ikke implementeret, men er beskrevet i [2d]. Desuden er problemet med overflødige referencer ikke stort, da en trekant for det meste ligger inden for én tile [17].



Figur 15: Omsluttende rektangel

Beregningen, af det omsluttende rektangel, følger denne algoritme:

- Sorter de tre punkter i trekanten efter stigende y-værdi, og udtræk top og bund værdi
- Sorter de tre punkter efter stigende x-værdi og udtræk venstre og højre værdi. Nu kendes det omsluttende rektangel
- Indsæt reference til trekanten i alle de tiles, som det omsluttende rektangel rammer

De første 2 punkter er uafhængige og kan udføres samtidigt i en pipeline, hvorimod 3. punkt kræver en løkke, hvilket umuliggør en pipeline.

[27] Intels Itanium 2, har 32KB on-chip L1 hukommelse og 6MB L3 hukommelse[16], og er den største kendte on-chip hukommelse, hos et kommercielt produkt. Hverken ATI eller nVidia oplyser cache-størrelser

Resultatet af dette er at splitte algoritmen op i to dele; en pipeline og en løkke, som indsætter referencer til trekanten i de berørte tiles.

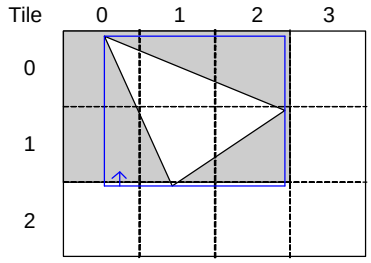
I den aktuelle implementering er det gjort, og de første to trin udføres i *CreateBox* processen, mens sidste trin udføres af *InsertVertexNode* processen.
Sidstnævnte er en del af den større *THTB Insertion* proces.

Beregningen af det omsluttende rektangel er oprindeligt en del af en større funktion, som desuden beregner trekantens hældningsværdier. Denne forklares nærmere i afsnit 5.1.

4.1.2 Implementering

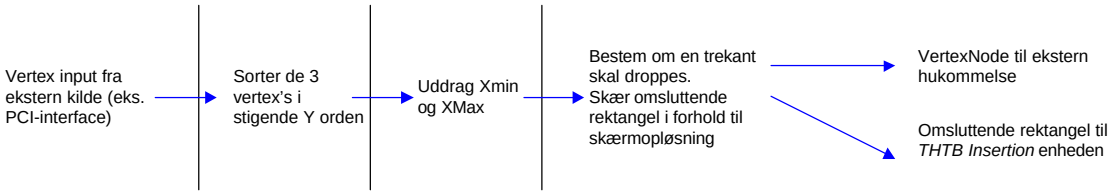
CreateBox processen er udført som en pipelinet tretrins proces, som udfører de beregninger, der er beskrevet i afsnit 5.1.
Opsplitningen i tre dele skyldes, at den første sortering i y-aksen er en tidskrævende proces, hvilket tvinger denne sortering ind i sit eget trin, da der ønskes en frekvens på 66,6MHz i denne proces.

Da der, senere i beregningsforløbet, regnes med hele pixel-værdier, afrundes x og y værdierne efter de samme regler, før rektanglet beregnes. Dette illustreres af figur 16, som viser, at det skraverede område ikke vil blive en del af det omsluttende rektangel, på trods af at trekanten ligger i tile (1,2), men den vil ikke blive tegnet pga. afrunding, hvilket betyder en mindre belastning af *BackEnd*. Brugen af korrekte afrundingsregler i beregningen kan derved hindre nogle af de overflødige tile indsættelser.



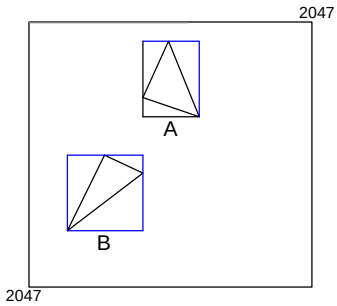
Figur 16: Omsluttende rektangel dækker ikke skraveret område pga. afrunding

De tre trin i pipelinen udfører de beregninger, som ses af figur 17. Enheden modtager en datapakke, også kaldet en VertexNode, fra en ekstern datakilde. Pakken fylder 234 bit, og indeholder de tre punkter, der udgør trekanten. Pakkens opbygning ses af figur 19. Først sorteres data i forhold til y-koordinaten. Herefter uddrages Xmin og Xmax, og sidste trin afgør, om trekanten er stor nok til overhovedet at skulle tegnes. Trekanten droppes, hvis den fylder under én linie. Hvis den derimod ikke droppes signaleres til de to næste enheder, at datapakken skal videre, og at der skal indsættes referencer til den i de tiles, som det omsluttende rektangel dækker.



Figur 17: Beregning af omsluttende rektangel i 3-trins pipeline

For at undgå overflødige indsættelser, pga. en begrænset skærmopløsning, bliver det omsluttende rektangel skåret til i forhold til skærmens opløsning. Eventuelt droppes trekanten, hvis den ligger helt uden for skærmens synsfelt. Af figur 18 ses to situationer, som vil medføre en beskæring af det omsluttende rektangel. Rektangel A vil blive beskåret, da det ligger uden for skærmens opløsning, mens rektangel B ligger uden for skærmens opløsning, og vil blive fjernet helt.



Figur 18: Beskæring og bortkastning af omsluttende rektangel

Det omsluttende rektangel overføres til *THTB Insertion* enheden, som udfører den egentlige indsættelse i de forskellige tiles, mens trekanten samtidigt overføres til *BucketFIFO*, som har til opgave at overføre trekanten til den eksterne hukommelse.

4.2 BucketFIFO

BucketFIFO har udelukkende til opgave at overføre trekanten til den eksterne hukommelse. Trekanten ankommer fra *CreateBox* til en FIFO buffer, som er placeret imellem de to enheder af to årsager; adskillelse af to clock-domæner og for at have en bufferzone mellem *CreateBox* og *BucketFIFO*.

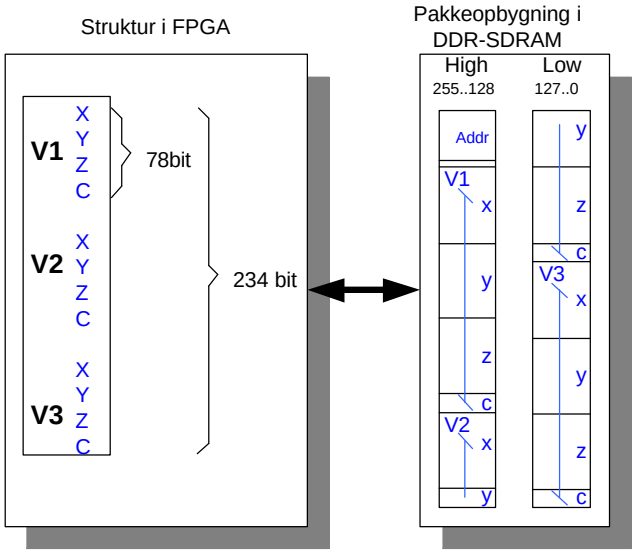
Når data kommer ud af FIFO bufferen, bestemmes den adresse, hvor trekanten skal placeres. Adressen starter med at være 0x20000, og tælles en op for hver pakke[28]. Dette sikrer, at *BucketFIFO* og *THTB Insertion* vælger samme adresse til trekanten, da *THTB Insertion* benytter samme adressegenerering.

Eftersom en VertexNode kun fylder 234 bit, er der 22 uudnyttede bit. I dette design indsættes adressen i dette område. Dette er gjort af hensyn til, at det skal være nemmere at fejlsøge på systemet, da informationen kan bruges til at detektere, om en pakke er blevet placeret korrekt i

[28] Der tilføjes desuden et nul til adressen, for at lave den fysiske adresse, som DDR-SDRAM hukommelsen skal bruge. Se afsnit 3.5

hukommelsen. Informationen bruges ikke af Hybris systemet, og kan derfor udelades i en senere udvidelse, hvis der er brug for de 22 bit.

Af figur 19 fremgår det, hvordan en VertexNode afleveres til den eksterne hukommelse, og det ses, at der er tale om en simpel opsplitning af pakken i to 128 dele á 128 bit.



Figur 19: VertexNode opbygning og opsplitning i den eksterne hukommelse

4.3 THTB Insertion

THTB Insertion modtager det omsluttende rektangel via en FIFO buffer, som sidder imellem *CreateBox* og *THTB Insertion*.
Det omsluttende rektangel modtages i form af 4 ekstrema værdier, som bruges til at gennemføre en løkke, der indsætter referencer til trekanter i de tiles, som ligger inden for ekstrema værdierne. Løkken er simpel, og enhedens virkemåde kan opsummeres i følgende pseudokode, hvor Ymin, Ymax, Xmin og Xmax er de 4 ekstrema værdier:

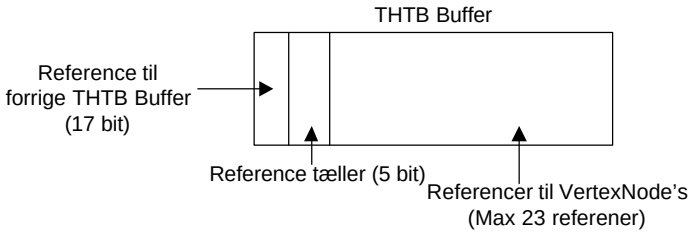
```
For y = Ymin til Ymax:
        For x = Xmin til Xmax:
                Indsæt trekant reference i tile (y, x)
```

For at få denne løkke til at køre hurtigst muligt, er der lavet flere under enheder, som hver laver en lille del af indsættelsen. Trekant referencen bliver indsat i en buffer, som der er plads til i FGPA'ens interne hukommelse. Til at styre denne buffer, samt 127 andre samtidigt, er der udviklet et cache system, som kan kontrollere, at der bliver skrevet til de korrekte buffere. I de efterfølgende afsnit beskrives hele dette system.

På bilag 6, bilag 7 og bilag 8 ses algoritmer og diagrammer over hele *THTB Insertion* enheden.

4.3.1 THTB buffer

Som i det oprindelige Hybris design indsættes referencer i blokke af en vis størrelse. Størrelsen af disse blokke – THTB buffere – er bestemt ud fra hardwaremæssige betragtninger. I softwareudgaven af Hybris er en THTB buffer 4KB stor, hvilket svarer til 63 trekanter og en header[29]. Headeren indeholder blandt andet information om hvor mange referencer, der er i bufferen, og en adresse på den forrige THTB buffer, som også er tilknyttet samme tile.
I hardware implementeringen er princippet med buffere bibeholdt, hvorfor navnet er det samme, til trods for at de ikke længere indeholder THNode referencer, men nu referencer til VertexNode strukturer.



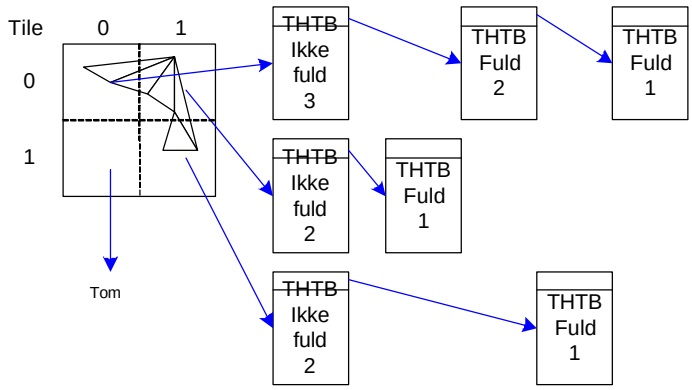
Figur 20: THTB buffer

Størrelsen på THTB bufferen i hardware udgaven er valgt udfra hensynet til, at et cache-miss skal kunne klares uden at sløve processen meget. Buffer størrelser på 512 bit er valgt, da det er den største datamængde, der kan overføres i et *burst* af DDR-SDRAM controlleren. Et kortere *burst* kunne eksempelvis nøjes med at overføre 256 bit, men da dette kun sparer to perioder, ud af 14, er det valgt ikke at bruge den mulighed.
Med 512 bit er der plads til 24 VertexNode referencer, men da der også skal være plads til en reference til forrige THTB buffer og en tæller, som skal holde øje med antallet af VertexNode referencer i THTB bufferen, er der plads til 23 referencer. Dette giver et overhead på 29 bit, svarende til 6%, da der skal afsættes 5 bit til tælleren og 17 bit til THTB buffer referencen. Opbygningen af THTB bufferen ses på figur 20.

Når en reference indsættes, tælles *Reference tælleren* op. Bliver denne 23, kan der ikke indsættes flere referencer. Derfor skal der oprettes en ny THTB buffer, hvor de næste referencer kan indsættes. For ikke at miste de data, der ligger i den fyldte THTB buffer, skal THTB referencen udfyldes med en reference til den gamle THTB buffer. Princippet kan ses af figur 21.

Som det fremgår, linkes der fra den nye buffer til den gamle buffer. Dvs. når trekanterne bliver indlæst igen af *BackEnd*, vil det ikke ske i samme rækkefølge, som de blev indlæst – bortset fra de tilfælde, hvor der maksimalt er 23 trekanter i en tile. I det nuværende design betyder denne ”omvendte” linkning ikke noget. Men skal designet senere udvides til også at indeholde a-værdier, skal trekanterne behandles i den korrekt rækkefølge, for at følge OpenGL standarden. I afsnit 7.1.4 er et forslag, til de ændringer, der er nødvendige, for at linke korrekt.

[29] Software udgaven indsætter en komplet kopi af en THNode, á 512bit, i hver THTB buffer, for at udnytte Pentium III CPU'ens cache optimalt



Figur 21: Linkede lister

4.3.2 Cache

Pga. begrænset on-chip hukommelse er det nødvendigt at begrænse de datamængder, som ligger i chippen. Dette gøres typisk ved kun at have de mest brugte data liggende i den interne hukommelse, hvilket kendes under det engelske udtryk *cache*.

Der findes flere teknikker til at bestemme hvilke data, der skal blive liggende i cache og hvilke, der kan ligge i en ekstern hukommelse, uden at systemets ydeevne falder drastisk. *THTB Insertion* enheden bruger den mest simple form for cache struktur – direct mapped cache[18]. Denne form for cache tildeler et lager område i cache strukturen på en logisk meget simpel måde, hvorfor denne type cache kan fungere meget hurtigt.

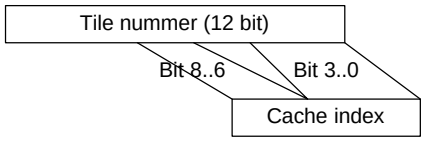
Da der er 4096 tiles, og der er tildelt lagerplads til 128 THTB-buffere, er der umiddelbar grund til at tro, at der vil være mange cache-miss situationer, som vil kræve en cache udskiftning, hvilket kunne foranledige én til at tro, at en mere avanceret n-way set-associate cache struktur ville give en betydelig bedre performance. Men software-delen af Hybris systemet opdeler sine objekter i partitioner således, at de optræder rimeligt grupperet [2a], og dermed også inden for et relativt begrænset område. Dermed øges chancen for et cache-hit, selv med en cache, der kun kan indeholde en brøkdel af den samlede datamængde.

TPT Tabel

Cache systemet er bygget op omkring to tabeller, foruden en cache hukommelse. En stor tabel med plads til 4096 THTB pointere, således at informationen om, hvor de enkelte THTB buffere er placeret i hukommelsen ikke mistes, og en lille tabel med plads til 128 THTB pointere, som bruges til at huske adressen på de THTB buffere, der ligger i cache.
Førstnævnte tabel kaldes for TPT – ”THTB Pointer Table”. TPT tabellen er opdelt således, at de øverste 6 bit i TPT indekseringsadressen, TPT_Address, svarer til y-tile indekset, og de nederste 6 bit svarer til x-tile indekset. Det gør, at en 12 bit tæller kan bruges til at løbe tabellen igennem, således at skærmen tegnes korrekt, dvs. i ”læseretningen”. Dette benyttes i *BackEnd*, se afsnit 5.2.1.

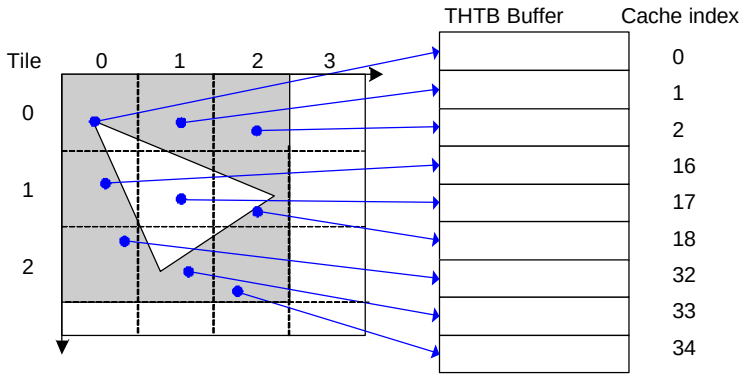
TPT tabellens pointere er alle på 18 bit, hvoraf kun de 17 er nødvendige for at kunne adressere THTB bufferne. Den 18 bit betydning er ikke defineret, men blot med af praktiske årsager, da der benyttes Xilinx BlockRAM moduler til implementeringen, og denne er i dette tilfælde på 18 bit. Da betydningen af denne bit ikke er defineret, bliver denne maskeret væk i de enheder, der benytter TPT tabellen.

Hver pointer peger på den sidste THTB buffer i hukommelsen, og denne THTB buffer peger videre på den forrige og så videre, indtil der ikke er flere THTB buffere i tilen.
De 4096 tiles svarer til 2^{12} indekses, og med plads til 128 THTB buffere i cache hukommelsen skal hver cache index deles af 32 tiles.
Til bestemmelse af indekseringsmetoden er det udnyttet, at data er grupperet, så cache systemet er inddelt på en måde, således det er muligt at cache et samlet område af tiles på samme tid. Det er gjort ved at tage udgangspunkt i TPT indekseringsadressen, og tage 4 bit fra x-delen og 3 bit fra y-delen af denne og sætte dem sammen til et cache index. Dette ses illustreret af figur 22.



Figur 22: Cache indeksering

Med denne implementering vil hver 16. tile ramme samme cache index, dvs. at eksempelvis tile (0, 0) og (16, 0) vil ramme samme cache index. Fordelen er, at et kontinuerligt område på 16x8 tiles[30] kan være i cache hukommelsen på samme tid (se figur 22), hvilket giver et rimeligt cache-hit niveau, som ses af afsnit 6.6.
I figur 23 ses det, hvordan en tilfældig trekant vil fordele sig i cache hukommelsen, og det bemærkes, at alle hele trekanter kan være der samtidigt.



Figur 23: Indsættelse i THTB Buffers

[30] 512x256 pixel, svarende til 3% af skærmen med en opløsning på 2048x2048, eller 43% af skærmen med en opløsning på 640x480

THTB RAM tabel
Den anden tabel, kaldet THTB RAM tabellen, indeholder DDR-SDRAM adresser på alle de THTB buffere, som ligger i cache hukommelsen. Tabellens adresser er på 18 bit, hvor de 17 er afsat til THTB buffer adressen, og den 18. bit benyttes til at indikere om indholdet af et index er legalt. Hvis det ikke er, skal det fortolkes som et tomt index, hvor der ikke ligger en THTB buffer. I kildekoden, og i teksten herunder, kaldes denne bit for *InCache* flaget.

Datastruktur i cache
Selve cache strukturen, hvor data gemmes, er stor nok til at indeholde 128 komplette THTB buffere. Da hver VertexNode reference er 21 bit, skal det være muligt at skrive 21 bit ad gangen. For at have en så hurtig cache som muligt er der brugt BlockRAM moduler med 2 uafhængige porte, som kan adressere og skrive samtidigt. Det betyder, at en reference kan indsættes samtidigt med, at THTB bufferens *reference tæller* øges.
En cache opdatering kan derved udføres på to perioder, da det er nødvendigt for *THTB Insertion* enheden at vide hvor mange referencer, der er i en eksisterende THTB buffer, før den kan placere den næste reference korrekt. De to perioder gør dette:

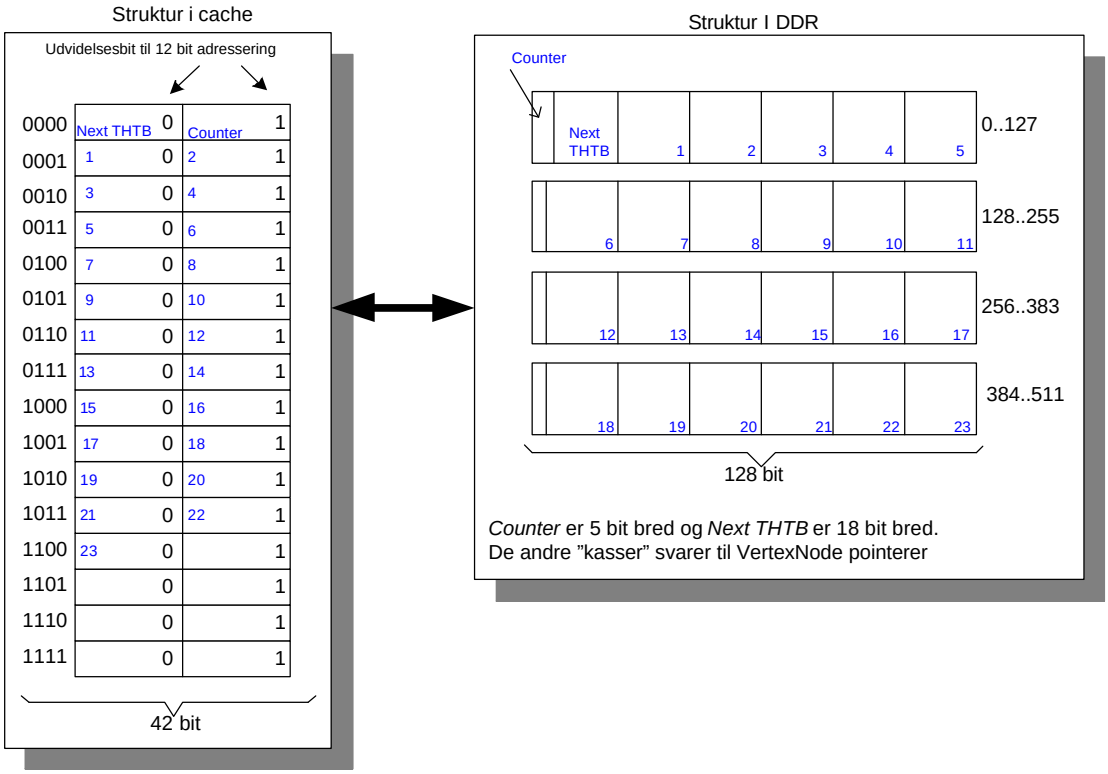
- Indekser headeren, dvs. reference tælleren
- Skriv den nye VertexNode reference i adresse (*reference tæller* + 2), og øg *reference tæller* med en

Sidstnævnte punkt kan udføres på én periode, da begge porte benyttes til at skrive.

Reference tælleren er kun på 5 bit. I og med den kun skal tælle til 23, kunne en 5 bit port være nok, men BlockRAM kredsene kan ikke have både en 5 bit og 21 bit port samtidigt, men kun porte, hvor bredden af port B er et 2^n multiplum af Port A's bredde[31].
Derfor er der valgt en port bredde på hhv. 21 og 42 bit, da der også skal tages hensyn til, at hele THTB bufferen skal kunne tømmes og fyldes hurtigst muligt til den eksterne hukommelse, for ikke at sinke systemet for meget ved et cache-miss.

De to port størrelser kan lade sig gøre, da de første 42 bit i hver THTB buffer afsættes til headeren, dvs. *reference tæller* og reference til den forrige THTB buffer. Disse 42 bit overskrives altid, når tælleren øges, dog med en datablok, hvor NextTHTB referencen er den samme som den eksisterende, så denne information ikke mistes. 21 bit porten benyttes til at indsætte VertexNode referencerne.
Størrelse og placering af de enkelte THTB buffere i cache strukturen er valgt, udfra ønsket om ikke at skulle bruge addition til at finde THTB bufferen. Eftersom en THTB buffer skal bruge 25 adresser á 21 bit eller 13 á 42 bit, er der afsat 16 styks 42 bit pladser, hvilket giver uudnyttede pladser, men til gengæld kan THTB bufferen lokaliseres direkte udfra cache indekseringen ved at tilføje 4 eller 5 bit, som vist i figur 24.
Samlet betyder det, at en THTB buffer fylder 672 bit i cache hukommelsen, men skal pakkes til under 512 bit i DDR-SDRAM hukommelsen, hvilket der er lavet en enhed til.
Figur 24 viser også, hvordan THTB bufferen er pakket i den eksterne hukommelse.

[31] Hvor n er 1..3



Figur 24: THTB buffer

Cache opdatering
Cache opdatering skal ske, hvis en af følgende 3 kriterier er opfyldt:

- *InCache* flaget er ikke sat
- TPT tabellens hukommelses reference er ikke den samme som THTB bufferens hukommelses reference
- Den nuværende THTB buffer indeholder 23 referencer

Første kriterium er kun opfyldt, første gang en ny tile vælges, og kræver ikke kommunikation med hukommelsen, da det kun er nødvendigt at nulstille THTB headeren, før data kan indlæses.

Andet kriterium kan til gengæld betyde, at data både skal skrives til og læses fra hukommelsen. Hvis det er første gang, en tile bliver benyttet, er det nok kun at skrive den eksisterende THTB buffer til hukommelsen, og derefter nulstille THTB headeren.

Sidste kriterium udløser en skrivning til hukommelsen, da bufferen er fyldt.

Når en cache opdatering skal foretages, bliver signalet *ChangeCache* sat højt, således at det er muligt at stoppe for VertexNode indlæsningen.

4.4 Parallelt kørende processer

For at få *THTB Insertion* til at fungere hurtigst muligt, er der tre styre processer, der kører parallelt, således at en samlet proces ikke skal bruge tid på at gennemgå alle situationer. De tre processer er:

- InsertVertexNode – Indsætter VertexNode referencer i de nødvendige tiles
- Cache Control – Opdaterer cache hukommelsen, i tilfælde af cache-miss
- THTB Output – Henter og skriver THTB buffere til ekstern hukommelse

4.4.1 InsertVertexNode

Processen er en tilstandsmaskine med 4 tilstande, som sekventielt indsætter VertexNode referencer i alle de tiles, som omslutnings-rektanglet dækker.
Når processen ikke kører, låses *Cache Control*, da denne ikke må starte, selv om TPT tabellens hukommelses adresse eventuelt ikke stemmer overens med THTB tabellens hukommelses adresse, da det vil medføre overflødig trafik til den eksterne hukommelse.
InsertVertexNode sørger for at indeksere i TPT tabellen, men opdatering af tabellen varetages af *Cache Control.*
Indsættelse af en adresse tager 2 perioder, da det er nødvendigt at bruge en periode på at indeksere i TPT tabellen, og derefter en periode på at opdatere indholdet af cache hukommelsen. Indledningsvis tager det 2 perioder at hente det omsluttende rektangel fra FIFO bufferen.

4.4.2 Cache Control

Processen startes, når *ChangeCache* flaget sættes, med mindre processen er låst. Først testes *InCache* flaget. Hvis dette er nul, ligger der ingen data i dette cache index, og det er derfor kun nødvendigt at tildele en adresse til den nuværende tile. Desuden sættes *InCache* flaget højt.

Hvis *InCache* flaget er sat, skal data skrives til den eksterne hukommelse, hvilket signaleres til *THTB Output*, som så vil sørge for skrivningen.
Når cache strukturen bliver frigivet af *THTB Output* processen, kan 3 situationer opstå, baseret på en sammenligning mellem TPT_Table og THTB_RAM. Disse er:

- TPT_Table er nul
- TPT_Table og THTB_RAM er ens
- TPT_Table og THTB_RAM er ikke ens

Første situation betyder, at tilen ikke har været indekseret før, og derfor vil processen tildele en adresse til THTB bufferen, og skrive denne til begge tabeller.

Anden situation betyder, at en THTB buffer er fyldt, og derfor er *ChangeCache* flaget sat, pga. en fyldt buffer. En ny THTB header skrives med en reference til den gamle adresse. Derved opstår de linkede lister.

Den sidste situation kræver en læsning fra den eksterne hukommelse, da den aktuelle tile før har været indekseret, og dens THTB buffer skal derfor indlæses. Dette signaleres til *THTB Output* processen, som derpå vil indlæse data.

Når *Cache Control* processen har opdateret cache indholdet, vil *ChangeCache* signalet automatisk blive sænket, og *InsertVertexNode* processen vil fortsætte.

Som processen er implementeret, vil en fyldt THTB buffer ikke blive skrevet med det samme, men først blive overført til den eksterne hukommelse, næste gang cache strukturen bliver indekseret; enten når en ny reference skal tilføjes, eller når en anden tile skal bruge cache hukommelsen. Dette er med til at sikre mindst mulig trafik til den eksterne hukommelse.

4.4.3 THTB Output

Processen styrer data ind og udlæsningen mellem THTB Cache og ekstern hukommelse, og kan startes af enten *Cache Control* processen eller af *Master Unit*.

Cache Control kan bede om ind eller udlæsning af data til dens cache, hvilket gøres med *RAM_REQ* signalet. *THTB Output* svarer igen ved at sætte både *Busy* og *Not_Ready* høje. En ny forespørgsel vil ikke blive behandlet, før *Not_Ready* bliver lav.

Når en udlæsning til ekstern hukommelse startes, vil processen starte med at bygge datapakker, som sendes til DDR interfacets I/O buffer. Dette sker relativt langsomt, da der kun kan udlæses 63 bit ad gangen, svarende til 3 VertexNode referencer.
Derfor kan en komplet udlæsning tage op til 9 perioder. For at bringe udlæsningstiden ned testes det, om der er under 12 referencer i bufferen. Er dette tilfældet bygges kun to pakker, svarende til 256 bit eller en kort burst. Derved spares der 4 perioder på pakke-opbygning, og yderligere 2 på kun at sende en lille datapakke.
Når *THTB Output* processen er færdig med at læse fra cache hukommelsen, sætter den sit *Busy* signal lavt, hvilket indikerer, at cache hukommelsen bliver frigivet til *Cache Control* enheden, som derved genstartes, parallelt med at *THTB Output* stadig arbejder. Derfor holdes *Not_Ready* signalet stadig højt, som først sættes lavt, når dataudlæsningen er helt færdig.
Ved at have disse to busy signaler kan flere VertexNode referencer indsættes før en cache udlæsning er tilendebragt. Dog kun så længe, at der er cache-hit; ellers skal der ventes, indtil udskiftningen er tilendebragt, så næste cache udskiftning kan foretages. Under optimale forhold vil et cache-miss kun koste tiden, det tager at udlæse data fra cache hukommelsen, hvilket med en lille pakke er 8-9 perioder – afhængigt af om indholdet af TPT tabellen er nul eller ej. Hvis der er to cache-miss i træk vindes stadig de 3 perioder som *Cache Control* kan køre, før en udskiftningsforespørgsel igen foretages.

Ved indlæsning af en THTB buffer indlæses data fra den eksterne hukommelse først, hvilket betyder, at begge ”busy” signaler forbliver høje under hele udskiftningen, da cache bliver opdateret, som sidste led.
Der vil altid blive læst en ”stor” pakke, da det på forhånd ikke vides, hvor mange VertexNode referencer, der ligger i THTB bufferen som skal indlæses.
Dog skrives kun de nødvendige data til cache, når reference antallet er kendt, for at nedbringe cache-miss straf tiden.

Master Unit kan starte en udlæsning, *flush*, af samtlige data i cache. Dette vil ske, når alle trekanter i et billede er indlæst, og det derfor er nødvendigt at tømme cache hukommelsen, for at kunne starte et nyt billede.
Ved en flush gennemgås alle *InCache* flag i THTB RAM tabellen, og THTB bufferen bliver overført til den eksterne hukommelse, hvis flaget er sat. Efter en komplet flush, er samtlige THTB buffere overført til hukommelsen, og *Master Unit* kan starte et billedskift.

Foruden de allerede beskrevne betingelser for en cache opdatering, indgår *Busy* signalet fra *THTB Output* også i *ChangeCache* flaget for at forhindre, at indsætningsrutinen starter, før en cache opdatering er tilendebragt.

4.4.4 Slave processer

Foruden de nævnte processer, er der to slave-processer, *THTB Controller* og *TPT Reset Engine*, som aktiveres af andre processer. De er slave-processer i det de ikke kan låse andre processer og de kan ikke ”beslutte” et videre forløb, men kan udelukkende styre en datastrøm, på grundlag af situationen ved proces-start.

THTB Controller styrer opdateringen af THTB pakkerne. Dvs. indsættelse af en VertexNode reference på korrekt adresse, og forøgelse af THTB bufferens *reference tæller*. Desuden sidder der en tilstandsmaskine, som nulstiller alle 128 *InCache* flag ved billedskift, dvs. når *ChangeBanks* går høj.

TPT Reset Engine, aktiveres ved billedskift og sørger for at nulstille indholdet af den ene TPT tabel, mens indholdet af den anden tabel ikke ændres. Den uberørte tabel indekseres af *BackEnd*, mens den nulstillede tabel er til rådighed for *THTB Insertion*. Det tager 512 perioder at nulstille en tabel, da det ikke er muligt at lave en almindelig asynkron reset af BlockRAM modulerne. I stedet udnyttes det, at modulerne kan have to uafhængige porte, hvor den ene er 18 bit bred, og den anden er 144 bit bred. 18 bit porten benyttes af *THTB Insertion* til opdatering af THTB buffer adresser, mens 144 bit porten benyttes til at nulstille tabellen.

4.5 Sammenfatning

I dette kapitel er *SortUnit*’s tre hoveddele beskrevet; disse er *CreateBox*, *BucketFIFO* og *THTB Insertion*.

CreateBox danner det omsluttende rektangel, der benyttes af *THTB Insertion* til at indsætte VertexNode referencer. Desuden sorterer *CreateBox* den indkommende trekants punkter, således at de tre punkters indbyrdes placering i y-aksen kendes. Det er desuden beskrevet, hvordan *CreateBox* kan fjerne trekanter, der ikke kan ses på skærmen.

Det er beskrevet, hvordan *BucketFIFO* overfører VertexNode referencer til den eksterne hukommelse.

Der er designet et komplet cache system, som har til formål at reducere belastningen af den eksterne hukommelse, da THTB bufferne ellers skulle hentes fra og læses til den eksterne hukommelse kontinuerligt. Der er designet et system med plads til 128 THTB buffere.

Til at styre denne cache, er der designet nogle parallelt kørende processer. Disse har til opgave at indsætte VertexNode's i de korrekte THTB buffere. De styrer desuden cache opdateringen, så de nødvendige THTB buffere befinder sig i den interne hukommelse, mens resten vil blive placeret i den eksterne hukommelse.

5 BackEnd

I dette kapitel analyseres og forklares de algoritmer, der bruges i software og hardware for at tegne en trekant.
Herefter følger selve implementeringen, som omfatter konvertering af eksisterende software funktioner til hardware og optimering af eksisterende hardware funktioner.
Efterfølgende gives en analyse af principperne for, hvordan systemet bedst muligt ændres til et multiprocessor system, og herefter hvordan det er implementeret i det udviklede system.
Til sidst er der opstillet nogle betingelser for tile størrelser.

5.1 Analyse af algoritmer

Som udgangspunkt for hardware implementeringen, er software-udgaven af beregningsfunktionen analyseret[32], for derved at opnå en hurtig beregning, specielt set i lyset af, at der skal beregnes hældningskoefficienter, hvilket kræver divisioner, som i hardware er en tidskrævende proces.

Software implementeringen er lavet i C++, og giver mulighed for en relativ simpel oversættelse til SystemC, men dette er ikke fornuftigt, da der derved ikke tages hensyn til eventuelle paralleliseringsmuligheder.
Da softwaren arbejder med 32 bit hel og decimaltal, vil en direkte oversættelse være umulig, da decimaltal ikke understøttes direkte i hardware. I stedet bruges et fixed-comma format, hvor der er afsat 12 bit til decimaler, da tidligere data-analyser har vist, at dette er tilstrækkeligt for at opnå et visuelt korrekt billede [19]. Dertil skal tilføjes 11 bit til heltal, da opløsningen er på 2048 pixel. Derved kan en bit bredde på 23 bit benyttes. Desuden kræves det, at et punkt har en ”hel” farve, hvilket betyder, at 8 bit kan beskrive farven i hvert punkt. Ved at bruge tal, med konstant kommaplacering, kan tallene manipuleres, som om de var heltal, hvilket mindsker beregningskompleksiteten meget.

Z-koordinaten er skåret ned til 24 bit af hensyn til, at den samlede trekantsbeskrivelse dermed kommer ned under 256 bit, hvilket i forbindelse med en PCI bus er en stor fordel, da der dermed kan overføres markant flere trekanter.
Renderer regner dog med en 32 bit z-værdi, af hensyn til at eventuelle fremtidige udvidelser skal kunne implementeres nemt.

Sammenlagt betyder det, at en trekant kan beskrives med 234 bit, hvilket betyder, at en trekant kan overføres i en 256 bit stor datapakke. I en eventuel opsætning, hvor pakkerne sendes over en PCI bus, kan de resterende 22 bit bruges til at overføre systeminformation, eller trekant beskrivelsen kan eventuelt udvides med gennemsigtighedsinformation i form af a-værdier. Udvides der med rigtig farveinformation eller 32 bit z-buffer (3 x 8 bit ekstra per trekant), skal en større datapakke benyttes, hvilket går ud over det antal trekanter, som kan overføres.

[32] Funktionen InsertTriangle i Hans Holten-Lund’s program

Software-funktionen beslutter, hvordan hældningskoefficienterne skal beregnes, ved at teste om det er en normal trekant, eller om der mangler en øvre eller nedre trekant. Ud fra disse tests beregnes derpå hældninger. Desuden bruges viden om, hvor den lange trekant-side er, til at placere divisionerne korrekt.

Analysen af funktionen afslørede, at funktionen kan opsplittes i følgende trin:

1. Sortering af punkter i trekant
2. Udtræk trekantens ekstrema (bounding box)
3. Afgør om trekanten kan ses; hvis ikke så smid trekanten væk
4. Bestem, hvilke divisioner, der skal udføres (6 eller 7)
5. Divider
6. Kombiner resultaterne til en datastruktur

Alle trin sender udelukkende data videre til næste trin, hvilket åbner muligheden for at udføre alle 6 beregninger samtidigt, dog på 6 forskellige datastruktur. Det giver mulighed for at beregne et output per periode.

I trin 4 skal divisionerne bestemmes, og som det fremgår skal op til 7 divisioner beregnes. Linie-hældningskoefficienterne beregnes altid, og har følgende formler:

Dybde ændring per x-pixel:

$$\mathrm{DzDx} = \frac{\left(\mathrm{Y_{Buttom}} - \mathrm{Y_{Top}}\right)\left(\mathrm{V2.z} - \mathrm{V1.z}\right) - \left(\mathrm{Y_{Middle}} - \mathrm{Y_{Top}}\right)\left(\mathrm{V3.z} - \mathrm{V1.z}\right)}{\left(\mathrm{V2.x} - \mathrm{V1.x}\right)\left(\mathrm{Y_{Buttom}} - \mathrm{Y_{Top}}\right) - \left(\mathrm{V3.x} - \mathrm{V1.x}\right)\left(\mathrm{Y_{Middle}} - \mathrm{Y_{Top}}\right)}$$

Farve ændring per x-pixel:

$$\mathrm{DColourDx} = \frac{\left(\mathrm{Y_{Buttom}} - \mathrm{Y_{Top}}\right)\left(\mathrm{V2.c} - \mathrm{V1.c}\right) - \left(\mathrm{Y_{Middle}} - \mathrm{Y_{Top}}\right)\left(\mathrm{V3.c} - \mathrm{V1.c}\right)}{\left(\mathrm{V2.x} - \mathrm{V1.x}\right)\left(\mathrm{Y_{Buttom}} - \mathrm{Y_{Top}}\right) - \left(\mathrm{V3.x} - \mathrm{V1.x}\right)\left(\mathrm{Y_{Middle}} - \mathrm{Y_{Top}}\right)}$$

Divisoren er her 20 bit bred, mens den i de andre tilfælde er 11 bit.
De sidste 5 divisioner afhænger af trekantens udseende, og det beregnes, hvor meget de variable ændrer sig fra linie til linie. Udregningerne er:

Hældning af den ene trekantsside:

$$\mathrm{dx1} = \begin{cases} \dfrac{\mathrm{V2.x} - \mathrm{V1.x}}{\mathrm{Y_{Middle}} - \mathrm{Y_{Top}}}, & \mathrm{Y_{Top}} \le \mathrm{Y_{Middle}} \\ \dfrac{\mathrm{V3.x} - \mathrm{V1.x}}{\mathrm{Y_{Buttom}} - \mathrm{Y_{Middle}}}, & \text{ellers} \end{cases}$$

Hældning af den anden side:

$$dx2 = \begin{cases} \dfrac{V3.x - V1.x}{Y_{Buttom} - Y_{Top}}, & Y_{Top} \le Y_{Middle} \\ \dfrac{V3.x - V2.x}{Y_{Buttom} - Y_{Middle}}, & \text{ellers} \end{cases}$$

Dybde-hældningen:

$$ZInc = \begin{cases} \dfrac{V3.z - V1.z}{Y_{Buttom} - Y_{Top}}, & Y_{Top} \le Y_{Middle} \\ \dfrac{V3.z - V2.z}{Y_{Buttom} - Y_{Middle}}, & Y_{Top} > Y_{Middle}, V2.x > V1.x \\ \dfrac{V3.z - V1.z}{YB_{Buttom} - Y_{Middle}}, & \text{ellers} \end{cases}$$

Farvehældningen:

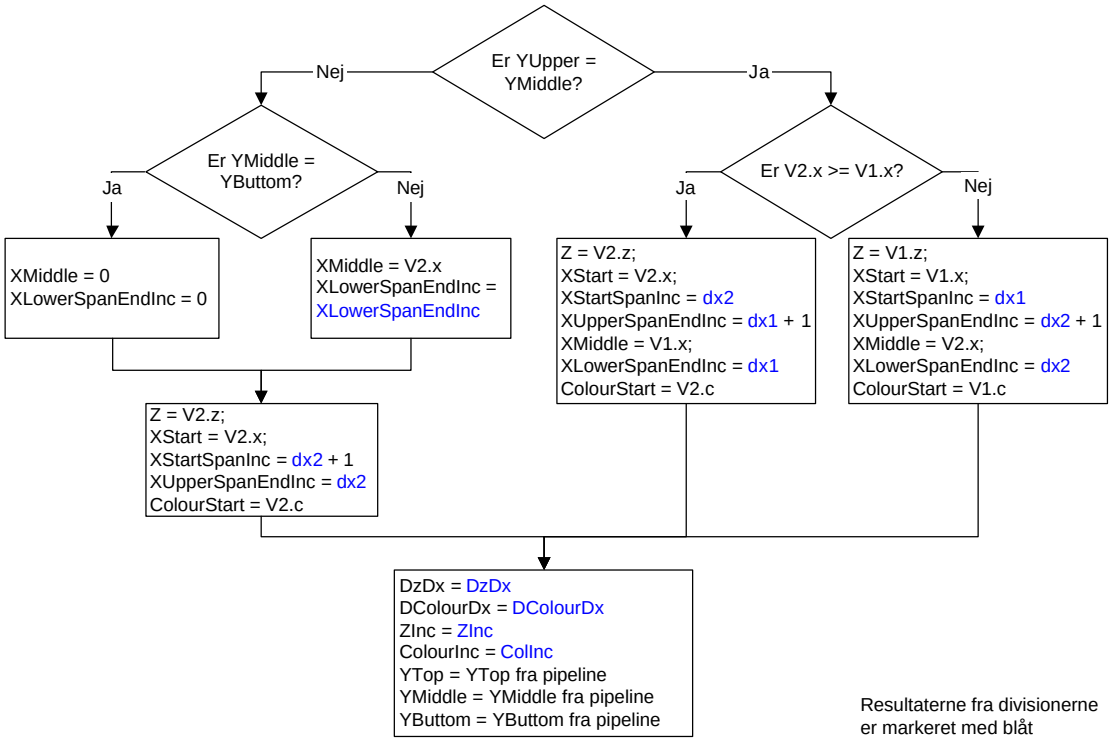
$$ColourInc = \begin{cases} \dfrac{V3.c - V1.c}{Y_{Buttom} - Y_{Top}}, & Y_{Top} \le Y_{Middle} \\ \dfrac{V3.c - V2.c}{Y_{Buttom} - Y_{Middle}}, & Y_{Top} > Y_{Middle}, V2.x > V1.x \\ \dfrac{V3.c - V1.c}{Y_{Buttom} - Y_{Middle}}, & \text{ellers} \end{cases}$$

Hældningen fra midten og nedefter:

$$XLowerSpanEndInc = \begin{cases} \dfrac{V3.x - V2.x}{Y_{Buttom} - Y_{Middle}}, & Y_{Top} \le Y_{Middle}, Y_{Middle} < Y_{Buttom} \\ \dfrac{V3.x - V3.x}{Y_{Buttom} - Y_{Middle}}, & Y_{Top} \le Y_{Middle}, Y_{Middle} \ge Y_{Buttom} \\ Dx1, & Y_{Top} > Y_{Middle}, V2.x > V1.x \\ Dx2, & Y_{Top} > Y_{Middle}, V2.x \le V1.x \end{cases}$$

> Som det ses, er det kun første betingelse, der kræver en division. Anden betingelse er opfyldt, hvis der ikke er nogen øvre trekant, mens de to sidste opfyldes, hvis der ikke er nogen nedre trekant.

Efter divisionerne er udført, indsættes resultaterne i en datastruktur sammen med de relevante start-værdier, således at de næste moduler kan tegne trekanten korrekt. Værdierne indsættes på baggrund af sammenligninger, som er vist i figur 25.



Figur 25: Endelig indsættelse i datastruktur til *BackEnd*

Trekantens linier skal tegnes fra venstre mod højre eller omvendt. Det afgøres ved sammenligningen:

$$XSpanStartInc < XUpperSpanEndInc$$

Er dette opfyldt, tegnes trekanten fra venstre mod højre.

Herefter skal trekanterne tilpasses den tile, de skal tegnes i. Dette sker ved klipning. Under klipningen, skal startværdierne ændres, sådan at de passer med den nye x eller y-værdi; dette gøres ved at addere dem med ændringsværdierne. F.eks. hvis en trekant skal klippes 2 linier fra toppen, vil følgende udtryk opstå:

$$XStartKlippet = XStart + 2 \cdot XSpanStartInc$$
$$XEndKlippet = XStart + 2 \cdot XUpperSpanEndInc$$

Bemærk, at XStart bruges til at beregne begge værdier, da trekantens x-værdi tager udgangspunkt i samme koordinatsæt.

Det samme sker, når x-koordinaten klippes, her vil blot benyttes de ændringsværdier, der er tilknyttet x-linierne.
Den første klipning, der foretages, er til y-aksen, sådan at trekanten ikke har en top eller bund uden for den aktuelle tile.

Herefter beregnes start og slutpunkter for hver enkelt linie i trekanten, dette gøres ved hjælp af de tilklippede start og slut værdier for x og de tilhørende differentiel værdier. Beregningen følger denne løkke:

```
for Y = YStart til YMiddle
    XStart =XStart + XSpanStartInc
    XEnd = XEnd + XUpperSpanEndInc
```

Dette gælder dog kun for den øvre trekant. Når midten passeres justeres XEnd ind ved, at den får værdien gemt i XMiddle, og differentiel værdien XUpperSpanEndInc byttes ud med XLowerSpanEndInc, og løkken fortsætter til Y = YButtom.

For at passe linierne ind i den aktuelle tile afrundes endepunkterne og linien klippes til, sådan at de ikke har x-værdier der overskrider x-værdierne for den aktuelle tile.

Til sidst skal hver pixel beregnes, dette gøres ved at køre følgende løkke:

```
for X = XStart til XEnd
    ColourStart =ColourStart + DColourDx
    ZStart = ZStart + DzDx
```

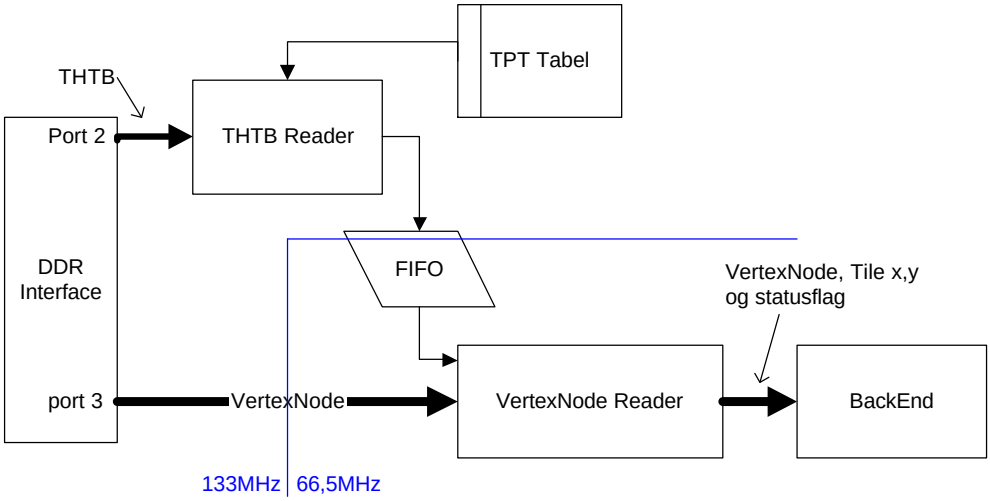
Når trekanten skal gemmes, skal det sikres, at den ikke overskriver en trekant, der ligger tættere på kameraet. Det gøres ved at sammenligne den netop beregnede trekants z-værdier med dem, der er gemt for de forrige trekanter. Det gøres på pixel basis, sådan at hver pixels z-værdi sammenlignes med den z-værdi, der er gemt for en eventuel tidligere beregnet pixel.

Dette er fremgangsmåden for at konstruere hele trekanten fra 3 koordinatsæt. Den aktuelle implementering i hardware, kræver dog en del mellemregninger og hjælpevariabler for at fungere korrekt.

5.2 Implementering

5.2.1 Indlæsning fra ekstern hukommelse

Der er lavet to enheder til at indlæse data til *BackEnd*. Den ene enhed, *THTB Reader*, har til opgave at indlæse THTB bufferne og uddrage alle VertexNode referencerne og overføre dem til *VertexNode Reader*, som er den anden enhed, der har til opgave at indlæse VertexNode's, og overføre dem til *CreateTHNode*, som vil regne på pakkerne. Figur 26 viser, hvordan de to enheder fungerer.
Som det fremgår arbejdes der med to clock-domæner. Den ene FIFO buffer adskiller *THTB Reader* og *VertexNode Reader*. Adskillelsen mellem *VertexNode Reader* og *DDR Interface* fremgår ikke af tegningen, men der er taget hensyn til dette.



Figur 26: Principdiagram over THTB / VertexNode reader

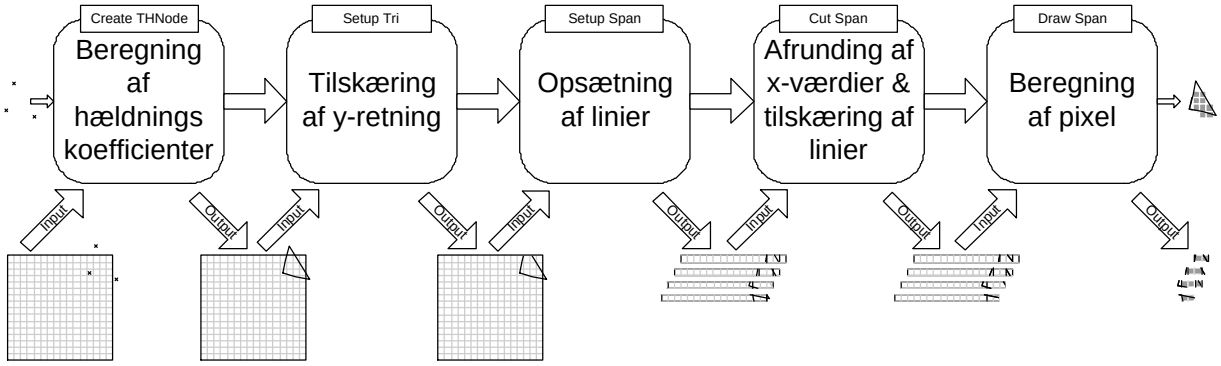
THTB Reader benytter den ene TPT tabel til at lokalisere første THTB buffer i en tile; eventuelle linkede buffere bliver lokaliseret ved at aflæse *NextTHTB* adressen i THTB bufferens header.
For at *BackEnd* kan vide, hvilken tile, der aktuelt arbejdes med, bliver der af *THTB Reader* tilføjet information om, hvornår det er sidste trekant i en tile. Dette gøres ved at tilføje *LastVertexInTile* flaget, som vil være højt, når sidste trekant i tilen indlæses. Hvis tilen er tom sendes signalet også af hensyn til, at det skal være muligt for frame-controlleren at nulstille en tile buffer også selv om, at den er tom, af hensyn til at gammel billedinformation skal slettes fra tilen.

I *VertexNode Reader* tilføjes endnu et flag, *LegalData*, som indikerer om en datapakke er gyldig. Det skal ske, da *BackEnd* består af mange pipeline dele, som altid kræver nye data. Derfor er det nødvendigt at tilføje et signal, som indikerer hvilke pakker, der er legale.
BackEnd er desuden så smart, at en ny, men tom tile, kan behandles i løbet af 2 perioder. Derfor er hele indlæsningsmodulet også i stand til at aflevere en sådan indikation med 2 perioders mellemrum. På bilag 8 og bilag 9 ses de to enheders algoritmer.

Da den absolutte placering af en tile mistes i løbet af beregningerne i *Renderer*, er det vigtigt, at der kommer et korrekt antal *LastVertexInTile* indikationer igennem *Renderer*, så det kan garanteres, at frame-controlleren afleverer tilen korrekt i SDRAM hukommelsen. Det forventes, at alle enheder starter med tile 0 efter en reset. Dette princip er overholdt i implementeringen, hvorfor der tages hensyn til den aktuelle skærmopløsning, så et korrekt antal *LastVertexInTile* signaler bliver afleveret til *Renderer*.

5.2.2 Endelig udformning af BackEnd

Den grundige analyse af *BackEnd* enheden har resulteret i en opdeling, som ses af figur 27. I toppen af hver enhed er det navn, som bruges i koden, angivet. Det eneste, der er sket med opdelingen, i forhold til de tidligere udgaver, er, at linieopsætningsenheden er blevet delt i 2, dvs. en tilstandsmaskine del og en pipeline del. Desuden er der tilføjet en enhed til hældningsberegning, som ikke tidligere har været implementeret i hardware.



Figur 27: Funktionsdiagram for *BackEnd*

Fra hukommelsen kommer der et komplet sæt trekant koordinater. Først kommer de igennem hældningsberegningen, hvor alle hældninger bliver beregnet, herefter passerer datastrukturen videre til *y-tilskæring*, her bliver de tilpasset, sådan at den del af trekantens vertikale linier, der ligger uden for den igangværende tile, fjernes. Næste trin er opsætning af linier, hvor trekanten bliver delt op i linier, hvorved datastrukturen bliver til lige så mange datastrukturer, som der er linier. Efter dette når datastrukturerne *afrunding af x-værdier og tilskæring af linier*, hvor x-værdierne bliver afrundet til skærmkoordinater og linierne tilpasset i x-retningen. Det gøres ved, at den del af en trekantlinie, der ligger udenfor den i gangværende tile fjernes. Til sidst er *beregning af pixel*, hvor hver pixels farve og z-værdi beregnes. Z-værdien bruges til at tjekke, om den igangværende pixel ligger tættere på kameraet, end den pixel der eventuelt er gemt i bufferen. Begge værdier gemmes derfor til bufferen. Farveværdien afrundes dog til 8 bit, så den kan omsættes til et standard RGB signal. Datastrukturerne mellem hver enhed ses af bilag 10.

5.2.3 Hældningsberegning

Som antydet i afsnit 5.1 er implementeringen lavet som en pipeline. Yderligere er beregningen splittet op i to processer. Første proces udfører punkt 1 til 3 og ligger i det modul, der hedder *CreateBox*. I afsnit 4.1.2 beskrives disse trin.
Trin 4 til 6 udføres af *hældningsberegning* og beskrives her.

Implementeringen er opdelt i to af hensyn til, at den eksterne hukommelse udgør en flaskehals, da der er en begrænset båndbredde til rådighed, hvorfor det er et mål at have så små strukturer til at passere hukommelsen, som overhovedet muligt.
Da VertexNode strukturen, der ankommer fra Hybris systemets software-del, er 234 bit, mens datastrukturen efter *hældningsberegningen* fylder 323 bit, er det klogest at lade den ubearbejdede VertexNode passere gennem hukommelsen, hvorved det største throughput opnås. Men da sorterings funktionen bruges til at beregne, hvilke tiles trekanten dækker, skal sorteringen være placeret før *THTB Insertion*. Ydermere kan sorteringen fjerne trekanter, som ligger uden for skærmens synsfelt, hvorved mængden af overflødig trafik nedbringes.

Hældningsberegningen består af to beregningstrin, foruden 30 trin, som udelukkende bruges, til at få timet resultatet af divisionen med datastrømmen.

Første trin beregner data til divisions-modulerne. Divisions modulerne er lavet med Xilinx's Core Generator, og har en forsinkelse, der kan beregnes således, hvor restbredden[33] er 3:

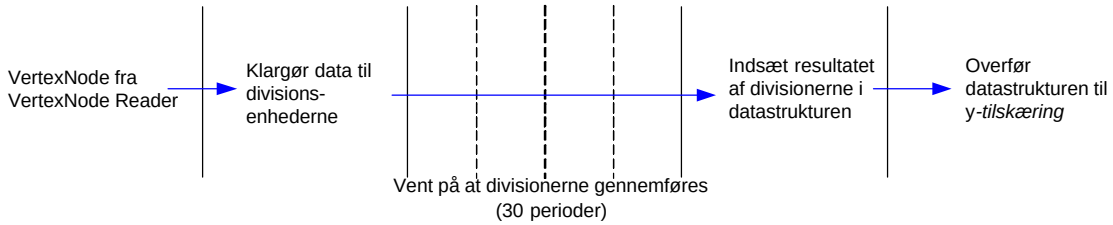
$$\text{Latency} = \text{Dividend bredde} + \text{Rest bredde} + 3$$

Da dividenderne alle er valgt til at have 24 bit, giver det en beregnings tid på 30 perioder, men da der bruges en pipeline proces, kan hele processen aflevere en datastruktur per periode.
En division med en 24 bit tæller giver beregningsfejl, da der i udtrykket for beregning af DzDx er en tæller, der principielt er 35 bit stor (11 bit · 24 bit), og der skal klippes 11 bit fra. Det vil give et problem med store trekanter, som spænder over mange tiles i y-retningen, som ligger tæt på skærmen, da de har både en stor z-værdi og et stort spænd i y-retningen, hvorved overløbsfejl kan forekomme. Desuden vil 20 bit til divisoren også give tilsvarende afrundingsfejl.
Men, som simuleringsresultaterne i afsnit 9.14 viser, er problemet ikke stort, da alle tests viser, at figurerne bliver tegnet uden store visuelle fejl.

Før divisionerne bestemmes, afrundes x og y værdierne.

Til sidst, i *hældningsberegning*, sidder et beregningstrin, som indsætter divisions-resultaterne korrekt i datastrukturen. Dernæst overføres datastrukturen til *y-tilskæring*, som begynder at beregne hvilke pixels, der skal tegnes.

Den samlede proces ses illustreret i figur 28.



Figur 28: Flow diagram for *hældningsberegning*

5.2.4 Det tidligere design af Renderen

Det design, der danner udgangspunktet for de sidste 4 enheder af *BackEnd*, kaldet *Renderer*, er skrevet af Santiago i SystemC [3]. Det indeholdt 3 enheder og ses af figur 29.

Figur 29: Det tidligere design af *Renderer*

[33] I andre konfigurationer er latency anderledes, og fremgår af Xilinx's Core Genrator

Tilskæringen af y-retningen foregår i følgende trin:

- Bestemmelse af, om trekanten tegnes fra venstre mod højre eller omvendt
- Beregning af minimum og maksimum x og y-koordinater for tilen
- Opsætning af startværdier og differentielværdier for x, afhængig af tegneretning
- Beregning af eventuel klipning af toppen af trekanten
- Test for om trekantens nedre del skal klippes
- Beregn hukommelses adresse for første linie
- Beregning af eventuel klipning af bunden

Tilskæringen er bygget som en lang beregning, der foregår på én periode. Kommunikationen, med de andre enheder, styres af en handshake funktion. Handshake funktionen gør, at det tager 5 perioder at sende en datastruktur igennem beregningsenheden til trods for, at beregningen kun tager en periode.

Linieopsætningen sker i en stor tilstandsmaskine, som består af 4 trin, foruden handshakes.

- Indlæs data
- Tjek om det er sidste trekant i tilen, ellers send datastruktur
- Hvis det var sidste trekant, laves en speciel datastruktur og sendes
- Beregn næste datastruktur, dette foregår efter ideerne skitseret i afsnit 5.1

Den datastruktur, som tilstandsmaskinen producerer for hvert gennemløb, sendes igennem følgende beregningsforløb:

- Afrunding af liniens endepunkter
- Klipning af liniens endepunkter
- Beregning af linielængde
- Beregning af start hukommelses adresse
- Test af om linien skal tegnes

Indlæsning af en datastruktur tager 4 perioder. Herefter tager hver iteration, i tilstandsmaskinen, 4 perioder, da der skal laves et handshake med den efterfølgende enhed for hver datastruktur, der er beregnet.

Datastrukturen fortsætter over i pixel beregningsenheden, som fungerer som en tilstandsmaskine på 4 trin foruden handshakes:

- Indlæsning af datastruktur, og test af om linien skal tegnes
- Vente trin
- Sammenligning af ny og gammel z-værdi
- Beregning af ny farve og z-værdi, og tælle linielængden en ned. Når linielængden når 0, hoppes til genindlæsning af datastruktur, ellers hoppes til ventetrin.

Linielængden bruges til at styre, hvor mange gange beregningen skal køres.

Der kan tegnes en pixel på 3 perioder, hertil kommer så handshakes, som gør, at indlæsning tager 3 perioder.

5.2.5 Generelle ændringer for alle enheder i Renderer

Det oprindelige design af *Renderer* er videreudviklet med det formål at forbedre hastigheden.
Da der bruges meget tid på handshakes, er de elimineret ved at indføre FIFO buffere. For at kontrollere dem er der blevet tilføjet styresignaler til de enkelte enheder:

- *FIFOFull*: Er højt, når bufferen er fuld
- *FIFOEmpty*: Er højt når bufferen er tom
- *FIFORead*: Når signalet er højt, vil FIFO bufferen sende en datastruktur ud ved starten af næste periode
- *FIFOWrite*: Når signalet er højt, gemmer FIFO bufferen en datastruktur ved starten af næste periode

FIFOFull signalet bliver ledt videre til alle enheder, sådan at de ikke laver en ny datastruktur, hvis der ikke er plads til det i bufferen. I en pipeline er dette simpelt, da registrene ikke henter nye data ind, når *FIFOFull* er lavt. I tilstandsmaskinerne er det mere besværligt, da disse automatisk regner videre, men dette er hindret ved, at de, når flaget er sat, til slut sletter den udregnede datastruktur ved at overskrive den med den forrige.
Omvendt har FIFO bufferne også et flag, der viser, at de ikke indeholder data. I disse tilfælde skal der ikke læses data fra dem, da det vil give uforudsigelige resultater. Derfor er der tilføjet kontrollogik, som styrer læse flaget til FIFO bufferen, og samtidigt med styrer, om den indlæsende tilstandsmaskine blot gentager sig selv uden at læse data, og dermed blot sender datastrukturen ud med *Valid* flaget sat falskt.
Yderligere er der også læse og skrive signaler til FIFO bufferne. Læse flaget styres altid af den efterfølgende tilstandsmaskine og sættes kun, når tilstandsmaskinen er parat til en ny datastruktur. Skrive flaget styres altid af en pipeline, og er en simpel or-funktion mellem *Valid* og *Last* flagene.

Der er valgt at bruge registerbalancing til pipeline enhederne. Af timing analyserne i bilag 11 kan det ses, at det kan gøres bedre med en manuel pipeline opbygning, men da resultatet er så godt, at det ikke sløver de resterende dele af *Renderer* ned, er registerbalancing valgt, da det gør designet nemmere at udvikle og fejlfinde.
Tilstandsmaskinerne er ændret, sådan at beregningsstadierne sætter læse flagene til FIFO bufferne. Det gør, at der ikke skal bruges en periode på dette.
I de enkelte enheder er der implementeret optimeringer, som bliver gennemgået herefter. For at få et indblik i, hvordan hver enhed behandler data, henvises til bilag 12, hvor der er procesdiagrammer for hver enkelt enhed.

5.2.6 Y-tilskæring

Y-tilskæring enheden var oprindeligt lavet til at være en pipeline, men den var omgivet af en handshake struktur, som ikke muliggjorde at udnytte pipeline ideen i dette.
Dobbeltvariablerne XRight og XLeft blev udskiftet med XStart. Dette gjorde en del af pipelinen overflødig.

Alle tre y-værdier bliver ikke sendt videre til næste trin, som tidligere. I stedet bliver det beregnet, hvor mange linier der er i hhv. øvre og nedre trekant. Dette gør, at der kun skal sendes 2 værdier videre. Da højden af de 2 dele ikke kan være større end højden af en tile, er der ingen grund til, at de har en 11 bit opløsning, der svarer til højden af skærmen i fuld opløsning, derfor spares der yderligere 6 bit per værdi.
Denne optimering har dog gjort det nødvendigt med en test, for om en trekant kun består af en nedre del. Da denne type skal have sat højden af den nedre halvdel til 0, og højden, af den øvre halvdel til den aktuelle højde af trekanten. Dette virker ulogisk, men er nødvendigt, for at linieopsætningen virker korrekt.
Udover det sættes *Low* flaget sådan, at linieopsætningsenheden er klar over, at det kun er en nedre trekant. XEnd sættes til værdien af XMiddle i stedet for XStart, da der her skal startes med en linie på flere pixels, i stedet for et hjørne, hvor de 2 værdier er ens. Denne ændring har desuden den fordel, at der ikke bruges en iteration i tilstandsmaskinen på at beregne en ugyldig datastruktur, som blot kastes væk.
Beregningen af start og slutværdi i x-retningen af tilen er fjernet, og i stedet sendes tilens x-koordinat videre. Det kan gøres, da de 2 værdier ikke bruges i denne enhed, og de skal ikke bruges i tilstandsmaskinen i den efterfølgende enhed. Dette giver en bit besparelse på 12 bit, eftersom der kun sendes en værdi videre, og da denne kun skal angive, hvilken tile der er anvendt, og ikke en skærmkoordinat, spares der yderligere 6 bit.
Sammenlægningen af *LastTri* og *NewTile* flagene har ikke betydning for enheden, da de blot bliver sendt videre, hvilket også er tilfældet med det nye *Last* flag.

I alt er datastrukturen i overgangen fra *y-tilskæring* til *opsætning af linier*, gået fra 372 bit til 330 bit, hvilket er en besparelse på 42 bit. Dette giver en besparelse i den mellemliggende FIFO buffer, da denne nu kan være i 5 BlockRAM moduler, hvor den ellers ville fylde 6.

5.2.7 Opsætning og tilskæring af linier

Opsætning og tilskæring af linier var tidligere en stor tilstandsmaskine med handshakes, men er nu delt i 2. Først en tilstandsmaskine og derefter en pipeline. Derudover er alle handshakes fjernet.
Tilstandsmaskinen foretager beregninger af nye start og slut punkter med fixedpoint præcision. Dertil kommer nye startværdier af farve, z og tile buffer adresse. De nævnte er eneste værdier, som afhænger af forrige resultat, og er derfor afhængig af en tilstandsmaskine.
I tilstandsmaskinen er alle beregninger flyttet ind under den tilstand, de hører til. Dette kan give en lille gevinst, da der ikke skal tages hensyn til nogle variable, som ikke kan ændre sig i visse tilstande. Det har den fordel, at koden bliver lidt nemmere at forstå, da funktionerne bliver kaldt, hvor de bliver brugt. Koden er sandsynligvis lavet sådan i sin tid, i et forsøg på at få noget af koden til at køre som en pipeline, hvilket er svært, når det styres af handshakes.

Udover at pipeline delen er blevet fjernet fra den tidligere tilstandsmaskine, er der også ændret lidt på de enkelte tilstande. Tilstandsmaskinen indeholder nu 3 tilstande. En reset tilstand, som nulstiller de nødvendige variable; dette er nødvendigt efter, at handshakes er fjernet, da de tidligere sikrede, at en enhed ventede, indtil der var korrekte data. Herefter kommer en indlæsningstilstand. Denne tilstand er ændret lidt for, at den kan fange trekanter, der kun består af en linie. De er fjernet i sorteringen, men under *y-tilskæringen* kan en trekant blive klippet sådan, at det kun er den sidste linie af trekanten, som ender i en tile, og der skal tages hensyn til disse. Ved at teste dette, allerede under indlæsningen, undgås det, at der spildes en

iteration på finde ud af, at trekanten er tegnet færdig. I samme tilstand er der også tilføjet en test for, om trekanten er gyldig. For begge disse tests gælder det, at der ikke skiftes til en anden tilstand. Dette gør, at der kan indlæses en ny trekant ved næste periode.
Den sidste tilstand, som er tilbage, er beregningstilstanden. Her er der fjernet nogle midlertidige variable. Dette påvirker ikke funktionaliteten af koden, men det gør den lidt nemmere at læse.

Da højden af øvre og nedre trekant modtages fra forrige enhed, i stedet for y-koordinaterne, skal testene, for om midten og bunden er nået, tilpasses dette. I stedet for at teste en tæller mod en y-variabel, skal det nu testes, om tælleren har nået nul. Dette har en sidegevinst, da det er langt hurtigere, og kræver mindre logik, at teste om en værdi er nul, i forhold til at teste 2 værdier mod hinanden.
Ved skiftet, mellem beregning af øvre og nedre trekant, tildeles tælleren højden af den nedre trekant, hvor der før blot blev talt videre.
Last flaget gemmes ved indlæsningen, hvorefter det sættes falskt. I den datastruktur, der regnes på, når den sidste linie sendes, tilføjes den gemte værdi igen. På denne måde kan en datastruktur indeholde trekantdata og *Last* flaget samtidigt.

Herefter sendes datastrukturen videre over i en pipeline, hvor X-værdierne afrundes til skærmkoordinater og linierne afskæres, sådan at de passer til den tile, de skal tegnes i.
Her er der tilføjet de beregninger af start og slut koordinaterne for x-aksen af den aktive tile, som blev fjernet fra *y-tilskæring* enheden.
Skrive flaget til den efterfølgende FIFO buffer styres af *Valid* og *Last* flaget. En af dem skal være sande, for at linien bliver skrevet i FIFO bufferen. På den måde skal der ikke spildes tid i pixel beregningsenheden på linier, som er kasseret, fordi de opnåede linielængder på -1.

Bit bredden af den datastruktur, der sendes videre til *beregning af pixel* enheden, er kun reduceret med en bit.

5.2.8 Beregning af pixel

Ligesom for de andre funktioner er handshakes fjernet. Derudover er tilstandsmaskinen bibeholdt, men den er ændret fra at kunne beregne en pixel på 3 perioder til at beregne en pixel hver periode. Dette er realiseret, uden at det går ud udover frekvensen, ved at lave et enkelt pipeline trin ved outputtet fra tilstandsmaskinen, som foretager sammenligningen med den z-værdi, der er gemt i bufferen.
Tilstandsmaskinen er reduceret til 4 tilstande, hvor der før var 10 tilstande. Igen er der tilføjet en reset tilstand pga. de manglende handshakes. Indlæsningstilstanden er bibeholdt, men er tilføjet en test for, om linien kun består af en pixel, eftersom der, i dette tilfælde, ønskes en ny linie indlæst ved næste periode.
I beregningsstadiet er de fleste midlertidige variabler fjernet, da de ikke er nødvendige. Det gør koden nemmere at forstå, men compileren ville nok optimere de midlertidige variabler væk alligevel, så der er ingen hastighedsgevinst at hente. Der er tilføjet en test for, om *Last* flaget er sat lige inden, der skiftes til indlæsnings tilstanden, da der hoppes til Last tilstanden direkte fra beregningen.
Last tilstanden er ændret fra at være en tom venteperiode til at sætte *Last* flaget som output, og gøre klar til indlæsning af næste datastruktur.

Til sidst er der lavet en enkelt trins pipeline i udgangen fra tilstandsmaskinen, som sammenligner den beregnede z-værdi med den, der ligger i z-bufferen. Resultatet bliver til skrive signalet for tile bufferen. Dette blev tidligere gjort i en separat tilstand i tilstandsmaskinen. Det har været nødvendigt at tilføje et *enable* signal til denne sammenligning, for at skriveflaget ikke skal blive sat ved en fejl, når der ikke er data at regne på, eller når *Last* flaget bliver håndteret.
I alt er *pixel beregningsenheden* gået fra at kunne tegne en pixel på 3 perioder til nu at beregne en pixel per periode. Dog vil datastrukturer med *Last* flaget sat bruge en periode ekstra, hvor der ikke beregnes en pixel.

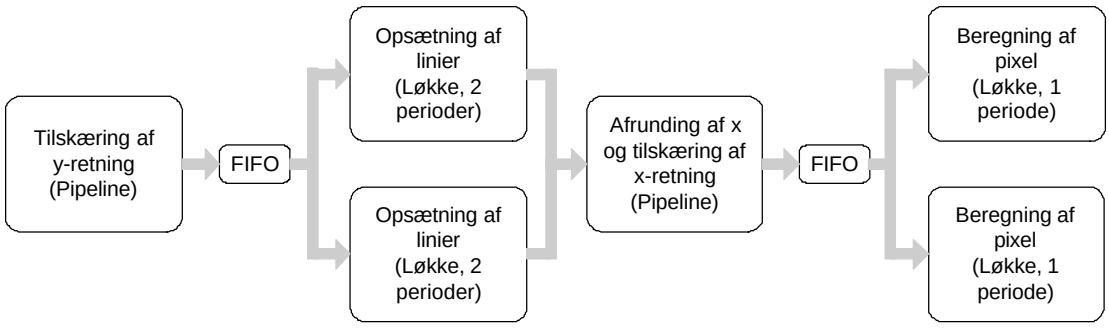
5.3 Multiprocessor Renderer

Den beskrevne *Renderer* fungerer som et single processor system, da der er én enhed af hver type. En måde, at øge hastigheden på, er at indsætte flere eksemplarer af beregningsenhederne. Det nemmeste er at indsætte 2 eksemplarer af hele *Renderer* og lade dem regne på hver sin tile. På den måde vil der kunne beregnes næsten dobbelt så mange tiles på samme tid. Denne metode er nem og hurtig, men der bruges dobbelt så meget plads i chippen. Derfor skal mulighederne for at kopiere enkeltelementer undersøges.

I det nedenstående er det hele tiden antaget, at de foranliggende rutiner er optimeret sådan, at de kan give det nødvendige antal input.

5.3.1 Metoder til dimensionering af multiprocessor system

Først analyseres systemet. Alle pipeline dele er nemme at have med at gøre, da de altid accepterer et input per periode og giver et output per periode. Derfor skal det så vidt muligt tilstræbes, at enhederne før kan levere en datastruktur for hver periode og, at enhederne efter også kan acceptere en datastruktur for hver periode. For at opfylde dette må de omkring liggende tilstandsmaskiner undersøges. Det første, der kigges på, er om tilstandsmaskinerne leverer et output for hver periode, dette kan opnås, hvis de kan afslutte en beregning i hver periode. Dette er dog ikke altid muligt, når der er komplicerede beregninger, og en høj frekvens skal overholdes. Kan tilstandsmaskinerne levere en datastruktur for hver periode, kan de fylde pipelinen ud, men kan de f.eks. kun levere en datastruktur hver anden periode, kan der med fordel indsættes 2 ens tilstandsmaskiner parallelt med hinanden. Hvis de skiftes til at levere data til pipelinen, vil de være i stand til at fylde pipelinen op. Et eksempel på dette kan ses af figur 30, hvor der er indsat 2 tilstandsmaskiner til opsætning af linier, hvor de hver bruger 2 perioder.
Når det er sikret, at pipelinen altid har data, skal det også sikres, at den kan komme af med dem igen. Eftersom den efterfølgende tilstandsmaskine for det meste vil bruge mere end en periode på at regne datastrukturen færdig, er dette ikke umiddelbart opnået. Igen kan det nødvendige hastighed opnås ved at indsætte flere ens enheder, der regner på hver sin datastruktur. Dette er også med i figur 30, hvor der er indsat 2 tilstandsmaskiner til beregning af pixels.



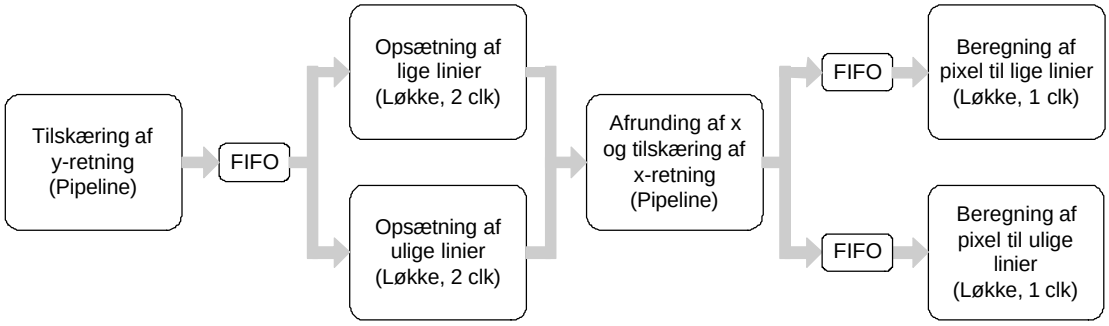
Figur 30: Eksempel på multiprocessor *Renderer*

Figur 30 har en flaskehals, da tilskæring af y-retningen ikke kan komme af med en datastruktur for hver periode. Her kan der indsættes en ekstra kopi af de efterfølgende enheder for at kompensere for dette.

Lige meget hvordan antallet af tilstandsmaskiner, der sættes efter en pipeline, vælges, vil der altid opstå flaskehalse, da der indgår mange forskellige størrelser af trekanter. Det er derfor meget forskelligt, om der skal laves 2 eller 10 linier for hver trekant, der kommer ind. Tilsvarende er det også meget forskelligt hvor mange pixels, der er i hver linie. Det kompenseres der delvist for ved at indsætte FIFO buffere efter hvert pipeline led. Størrelsen på disse må afgøres udfra nogle forsøg med forskellige typer af billeder for at se, hvor meget de bliver fyldt. Derudover kan hardware også sætte nogle betingelser op for hvor store eller små, det er hensigtsmæssigt at lave FIFO bufferne, da de typisk implementeres i BlockRAM moduler, som ikke kan bruges til andet samtidigt med, at de er FIFO buffere, og de derfor ligeså godt kan fyldes op, selvom der reelt godt kunne bruges en mindre buffer.

Ovenstående er en teoretisk gennemgang af, hvordan pipelines og tilstandsmaskiner optimeres bedst muligt, i forhold til hinanden. Der er dog et lille problem med denne teori, når der skal tegnes trekanter. Det antages normalt, at hvis 2 pixels skal tegnes i præcis samme x, y, z position, skal det være den pixel, der blev beregnet først, som ender på skærmen. Dette giver problemer i den ovenfor foreslåede teori, da en pixel i en lille trekant godt kan overhale en pixel i en stor trekant, selvom den lille trekant er indlæst sidst. Dette kan ske i *opsætning af linier*, hvis 1. enhed er ved at regne på øverste del af den store trekant, mens 2. enhed indhenter den lille trekant, som deler en pixel med nederste del af den store trekant, her vil den lille trekants linier komme ind i den efterfølgende pipeline før de nederste linier i den store trekant. Det vil derfor blive en pixel fra den lille trekant, der bliver tegnet, og ikke den fra den store trekant, som egentligt skulle have være tegnet.
Der er flere måder at undgå dette på. En ville være at give hver trekant et unikt ID, men dette er ikke særligt hensigtsmæssigt, da der er mange trekanter, og det ville give 20-25 bit, som skal med igennem alle pipelines, hvilket koster ekstra registre. Dertil kommer, at der samtidigt med z, også skal gemmes et ID nummer i tile bufferen, og dette gør denne endnu større, så denne løsning er ikke brugbar.
En anden løsning er at dele tilen i lige og ulige linier [2e]. 1. linieopsætningsenhed håndterer alle linier, der ligger på en lige y-koordinat, 2. enhed håndtere alle linier, der ligger på en ulige y-koordinat. Hvis de stadig skal dele den efterfølgende pipeline, vil dette kræve en ekstra bit til at indikere, om linien hører til den ulige eller lige tile. Og en ekstra FIFO buffer sådan, at

der er en separat buffer til hhv. lige og ulige linier. Herefter skal der være en pixel beregningsenhed til hhv. lige og ulige linier. Denne opsætning er vist i figur 31.

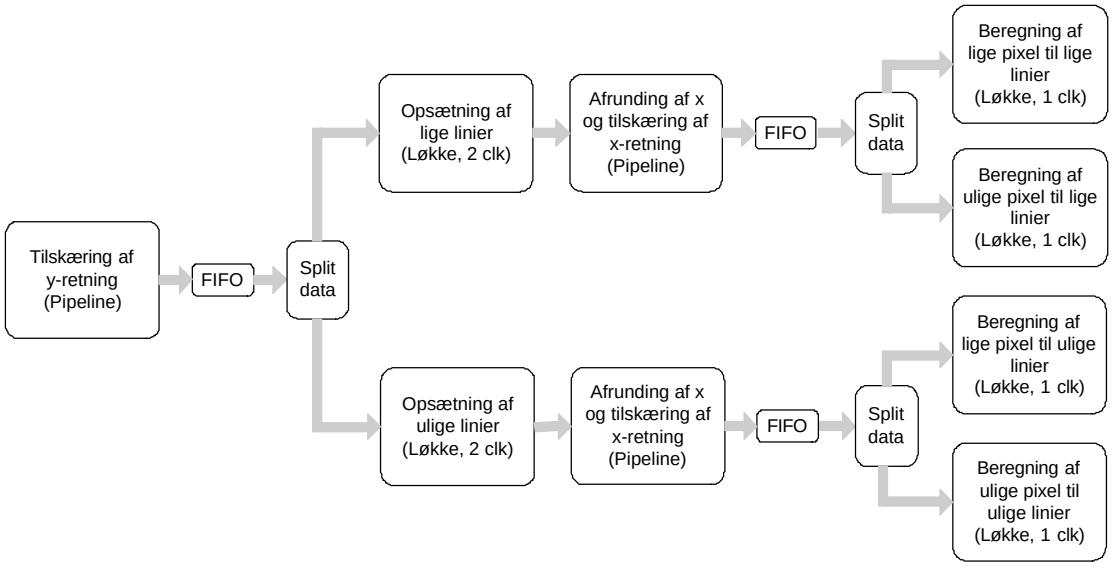


Figur 31: Multiprocessor *Renderer* med opdeling i lige og ulige linier

Metoden har en ulempe, da der ikke altid er et lige antal linier i en trekant. Derfor vil der, for nogle trekanter, være et beregningstrin, hvor den ene *opsætning af linier* ikke laver noget. Dette er svært at komme udenom, med mindre der indlægges 2 FIFO buffere i stedet for 1 efter *y-tilskæring*. Dette er dog en pladskrævende løsning, da denne FIFO buffer er 340 bit bred.
Ligesom der kan implementeres 2 *opsætning af linier*, kan hver linie også beregnes af flere pixel beregnings enheder. Dette giver de samme problemer, da reglen om at ældste pixel skal tegnes først, også gælder her, og der også kan opstå spildte beregningstrin.
Ved anvendelse af flere *beregning af pixel* enheder opdeles tile bufferen i flere dele, sådan at hver beregningsenhed har sin egen buffer.

5.3.2 Implementering af multiprocessor system

Den løsning, der er implementeret, afviger lidt fra det tidligere beskrevne, da der er opnået en enhed til opsætning af linier, der kan beregne en linie for hver periode. Der er dog fortsat 2 enheder til opsætning af linier, men de deles ikke om en fælles enhed til afrunding. Her har de en hver, og yderligere er der tilføjet 2 ekstra enheder til beregning af pixels, sådan at hver linie bliver beregnet af 2 enheder. Disse er også delt efter lige ulige princippet, sådan at den ene enhed tager alle pixels, der ligger på en lige x-koordinat, og den anden tager alle, der ligger på en ulige. Systemet ses af figur 32.



Figur 32: Den implementerede multiprocessor *Renderer*

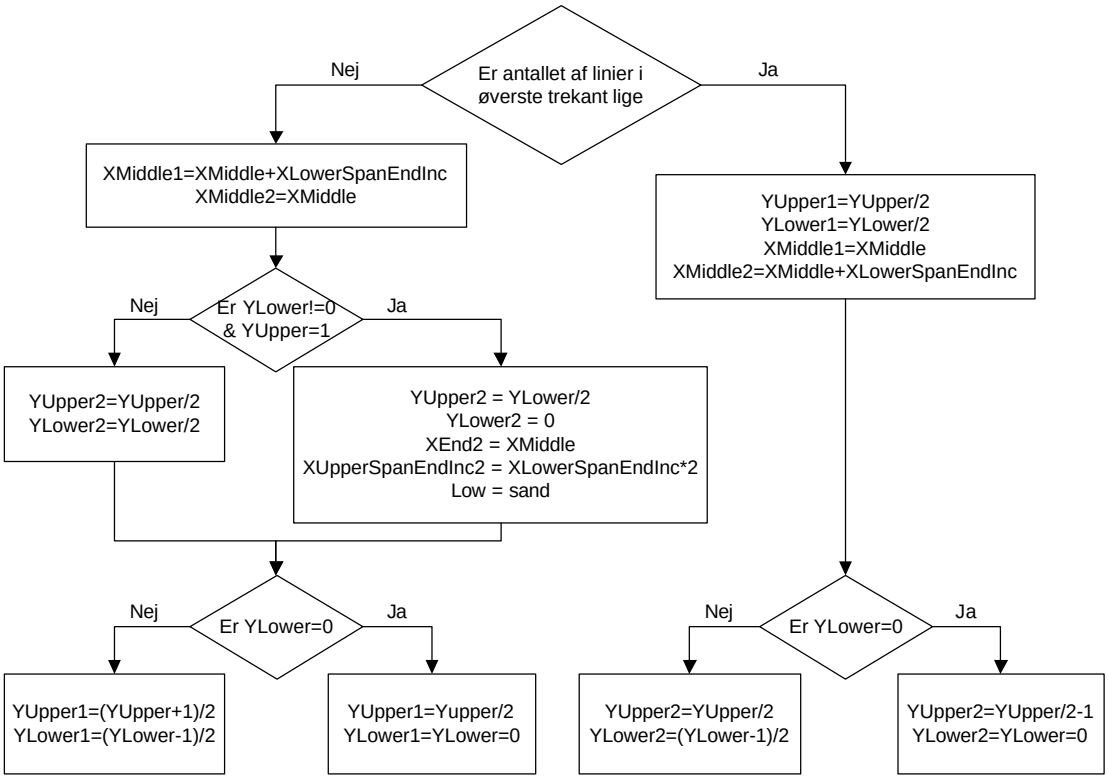
Denne pyramide struktur er valgt udfra et synspunkt om, at for hver tilstandsmaskine stiger antallet af datastrukturer, der skal regnes på. F.eks. for en trekant på 3 linier og 6 pixels, vil der være en beregning med y-tilskæring, 3 linieopsætningsberegninger og 6 pixel beregninger.

Det er forsøgt at bibeholde de oprindelige 4 enheder fra single processorudgaven af systemet, sådan at der ikke skal rettes i mere end én fil, når der rettes en fejl, eller der tilføjes funktionalitet. Der er kun ændret nogle få ting, men de er implementeret sådan, at enhederne stadig vil fungere i singleprocessor udgaven eller i udgaver med et andet antal enheder.
For at enhederne stadig skal fungere i singleprocessor udgaven, er der tilføjet nogle enheder til opsplitning af data. Deres funktion er at rette start og forøgelses værdier til sådan, at de passer til, at det kun er hver anden linie eller pixel, der skal beregnes. Dette gøres ved at fordoble alle ændringsværdier og derefter beregne en ny startværdi for den af enhederne, som ikke skal regne på første datastruktur. F.eks. når en trekant kommer fra y-tilskæringen, vil startkoordinaterne passere videre til den ene enhed, mens den anden enhed vil få et sæt startværdier, der er øget med ændringsværdierne.
Hvilken enhed, der får hvilket sæt data, afgøres af om trekanten eller linien starter i en lige eller ulige del.
Enhederne er identisk opbygget, dog er de forskellige variabelnavne tilpasset, og enheden før *opsætning af linier* har fået tilføjet kontrollogik til at styre om trekanten, der skal tegnes, starter under midten. Dette kan ske, hvis den er blevet klippet af *y-tilskæring*, eller hvis den kun består af en nedre trekant. Beregningerne for *split data* enheden før *opsætning af linier* ses af tabel 8, og beregningerne for *split data* enheden før *beregning af pixel* enheden ses af tabel 9. Her refererer navne med 1 og 2 til første og anden enhed. Det er samme beregning uanset om trekanten starter på en lige eller ulige enhed. Bemærk, at der i koden bruges out1 og out2, som er direkte bundet til hver deres enhed. Hvilken der får første eller anden datastruktur, styres ud fra bit 6 i tile buffer adressen, da denne afslører om, det er en lige eller ulige linie. Flagene bliver sendt direkte igennem, på nær *Valid* flaget for den anden enhed, da dette skal sættes falskt, hvis trekanten kun består af en linie.

Data til første enhed	Data til anden enhed
XStart1 = XStart	XStart2 = XStart + XSpanStartInc
XEnd1 = XEnd	XEnd2 = XEnd + XUpperSpanEndInc
XSpanStartInc1 = XSpanStartInc · 2	XSpanStartInc2 = XSpanStartInc ˣ 2
XUpperSpanEndInc1 = XUpperSpanEndInc · 2	XUpperSpanEndInc2 = XUpperSpanEndInc ˣ 2
XLowerSpanEndInc1 = XLowerSpanEndInc · 2	XLowerSpanEndInc2 = XLowerSpanEndInc ˣ 2
ZStart1 = Zstart	ZStart2 = ZStart + ZInc
ZInc1 = ZInc · 2	ZInc2 = ZInc ˣ 2
DzDx1 = DzDx	DzDx2 = DzDx
ColourStart1 = ColourStart	ColourStart2 = ColourStart + ColourInc
ColourInc1 = ColourInc · 2	ColourInc2 = ColourInc ˣ 2
DColourDx1 = DColourDx	DColourDx2 = DColourDx
X_RAM1 = X_RAM / 2	X_RAM2 = X_RAM / 2
TileX1 = TileX	TileX2 = TileX
Flags1 = Flags	Flags2 = Flags

Tabel 8: Beregninger i split data for opsætning af linier

I tabel 8 mangler YUpper, YLower og XMiddle variablerne. Det skyldes, at de deles ud på nogle ekstra specialtilfælde, da der er forskel på, om antallet af linier i den øvre trekant er lige eller ulige, og sammen med informationen om, at trekanten starter i en lige eller ulige linie, afgør dette hvilken enhed, der skal tegne linien, hvor XMiddle skal sættes ind. Dertil kommer, at YUpper og YLower skal splittes, alt efter hvilken værdi YUpper og YLower har. Udregningerne fremgår af figur 33.



Figur 33: Proces diagram for beregningen af antal linier i øvre og nedre trekant

Som det ses, giver summen af de 2 nye ikke altid summen af den gamle. Dette skyldes, at *opsætning af linier* enheden laver en ekstra linie, dvs. når YUpper er 2 laver den 3 linier. For YLower bliver der tegnet det rigtige antal linier, men 2. trekants XMiddle linie bliver trekantens første linie i nedre trekant, og derfor skal YLower være reduceret med en.

Data til første enhed	**Data til anden enhed**
SpanWidth1 = SpanWidth / 2	SpanWidth2 = (SpanWidth –1) / 2
ZStart1 = Zstart	Zstart2 = ZStart + DzDx
DzDx1 = DzDx · 2	DzDx2 = DzDx · 2
ColourStart1 = ColourStart	ColourStart2 = ColourStart + DColourDx
DColourDx1 = DColourDx	DColourDx2 = DColourDx
Flags1 = Flags	Flags2 = Flags

Tabel 9: Beregninger i split data for beregning af pixels

Der er en undtagelse fra tabel 9; *Valid* flaget, for datastrukturen til anden enhed, vil blive sat falskt, hvis linien har længden nul, og dermed kun består af en pixel. Derudover mangler beregningen af tile buffer adressen, som skal tage hensyn til i hvilken retning, trekanten tegnes. Udregningerne for tile buffer adressen er vist i tabel 10.

Tegnes fra venstre mod højre	**Tegnes fra højre mod venstre**
X_RAM1 = X_RAM / 2	X_RAM1 = X_RAM /2
X_RAM2 = (X_RAM + 1) / 2	X_RAM2 = (X_RAM – 1) / 2

Tabel 10: Beregning af tile buffer adresse i split data for beregning af pixels

Da *split data* enhederne ikke indeholder nogen form for hukommelse, og de læser fra en FIFO buffer, skal det sikres, at de 2 enheder, som *split data* enheden sender data videre til, begge er klar til at læse data, inden der læses fra bufferen. Dette gøres ved, at deres *FIFOenable* flag begge skal være høje, før der foretages en læsning. Samtidigt skal enhederne være klar over, hvis der ikke er indlæst data, selv om de har bedt om det. Dette løses ved, at hver enhed har den anden enheds *FIFOenable* ført ind. Skal enhederne bruges i en single processer udgave, sættes denne indgang blot høj.

Et ekstra problem, ved denne implementering er, at de interne BlockRAM skal anvendes i klumper på 18kbit. Dette gør, at der, ved den anvendte tile størrelse på 32x32 pixels, opstår et stort spild, da der skal være en separat buffer for hver pixel beregningsenhed. Dette skyldes, at der skal kunne skrives en pixel inkl. z-værdi per beregningsenhed per periode til tile bufferen. Samtidigt med skal der udlæses en z-værdi svarende til den næste datastruktur.
Spildet per z-buffer vil være:

$$18\text{kbit} - (16 \cdot 16\text{pixels}) \cdot 32\text{bit} = 10\text{kbit}$$

og per farve buffer er det større, selv om der placeres 2 buffere i hver BlockRAM:

$$18\text{kbit} - (16 \cdot 16\text{pixels}) \cdot 2 \cdot 8\text{bit} = 14\text{kbit}$$

Det ses, at spildet er mere end 50%, hvilket giver mulighed for at ændre tile størrelsen; størrelsen er dog fastholdt.

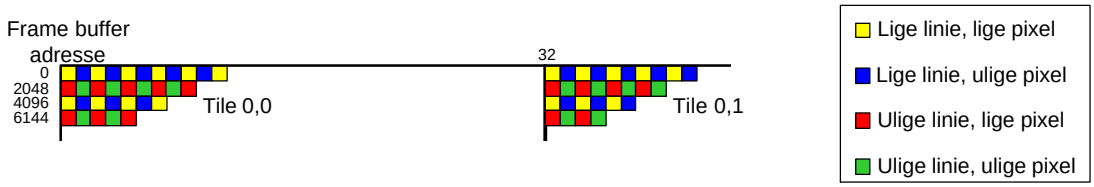
5.3.3 Tile størrelse

Størrelsen af den tile, der regnes på, afhænger af mange forskellige faktorer. De kan deles i 2 grupper, dem der har gavn af en mindre tile, og dem der har gavn af en større tile.
Den vigtigste grund til at bruge en lille tile er, at det sparer på de interne BlockRAM moduler. Derudover bliver flere af variablerne undervejs skåret ned, sådan at de passer i en tile, dvs. at en forøgelse af tile størrelsen vil give en større bit bredde på disse variable. Dette betyder større beregningsenheder og større FIFO buffere.

Modsat er der mange argumenter for at vælge en større tile størrelse, det vil give færre læsninger fra hukommelsen, da en trekant skal læses ind en gang for hver tile, den er med i. Dette vil give færre klipninger i både x og y retningen, da der vil være færre trekanter, der dækker flere tiles. Færre klipninger vil give færre beregninger i hver enhed bortset fra pixel beregningsenhederne, da de altid skal beregnes en gang for hver pixel. Overhead for de dobbelte linieopsætning og pixel beregningsenheder vil også reduceres, da færre klipninger giver færre trekanter med ulige linieantal og færre linier med ulige pixel antal. Derudover giver det færre tiles, og dermed færre skift, hvor hver eneste pixel skal nulstilles, da det er samme antal pixels, som skal udlæses og nulstilles, er der umiddelbart ikke nogen besparelse ved dette, men da der vil være færre helt tomme tiles, vil der ikke være så mange perioder, hvor pixel beregningsenheden venter på, at tilen bliver tømt og nulstillet.

5.3.4 Framecontroller udlæsning med multiprocessor system

Når farveværdierne skal læses over i en frame buffer, skal der tages hensyn til denne opdeling af tile bufferen, opdelingen er illustreret på figur 34. Det kan dog udnyttes, at de kan sættes i par af 2 og 2, da samme adresse i buffer for lige og ulige pixels svarer til 2 pixels, der ligger efter hinanden. På den måde kan der udlæses 16 bit per periode, dette er dog stadig ikke tilstrækkeligt.



Figur 34: De enkelte pixels placering i frame bufferen

Styringen af udlæsningen til frame bufferen er ikke implementeret, men der foreligger følgende principper for, hvordan det skal gøres: For at udnytte båndbredden til frame bufferens hukommelse bedst muligt, og for hurtigst muligt at gøre tile bufferen klar til et skift, skal der læses fra alle 4 tile buffere på en gang. Det gøres, ved at have et sæt midlertidige registre, der kan holde de læste værdier, og da hukommelsesbåndbredden udnyttes bedst ved at skrive større mængder af data på en gang, er det ønskeligt at skrive 256 bit ad gangen. Dette skal benyttes, da der ved en tom tile vil være en skifte tid, svarende til den tid, det tager at kopiere tile bufferen over i frame bufferen. Den aktuelle belastning ses af nedenstående udregning, som også danner grundlag for den forsinkelse, der er brugt under de performance tests, som er beskrevet i kapitel 6.

Udlæsningstiden for 256 bit fra tile buffer til frame buffer tager, hvor setup tiden er medregnet, 14 perioder. En tile består af 1024 pixels, hvilket svarer til 8192 bit. Det vil tage 448 perioder at udlæse som bevirker, at der skal gå 448 perioder mellem hvert tile skift. Dette kan give anledning til forsinkelser i beregningerne, da der ved tomme tiles vil opstå ventetider, da disse kun tager 2 perioder at beregne.

5.4 Sammenfatning

I dette kapitel er principperne for opbygningen af en trekant fra kapitel 2 oversat til funktioner, som kan implementeres i hardware. De algoritmer, der kun var tilgængelige i software versioner, er analyseret og implementeret i hardware, mens den resterende del af *BackEnd* er optimeret udfra de ideer, der er givet i afsnit 2.4.
For at få en bedre performance, er der en forklaring af, hvilke principper, der kan bruges, til at forbedre hastigheden for en multiprocessor udgave af *Renderer* markant, men uden at frådse med den logik der er til rådighed. Dette har resulteret i en pyramide struktur for *Renderer*, hvor der efter hver pipeline enhed, er lavet en forgrening ud til 2 ens enheder.
Ændringen af *Renderer*, til en multiprocessor udgave, har givet nogle andre betingelser for den efterfølgende tile buffer, derfor er der en kort diskussion af størrelsen af denne.

6 Funktionalitet

I dette kapitel beskrives de metoder, der bruges til at undersøge funktionaliteten af det udviklede system.

Først beskrives grundlaget for en visuel test af systemet, og derefter de benyttede simuleringsværktøjer, som er udviklet i forbindelse med projektet.
Til sidst måles systemets performance, diskuteres og sammenlignes med andre systemer.

6.1 Valideringsmetoder

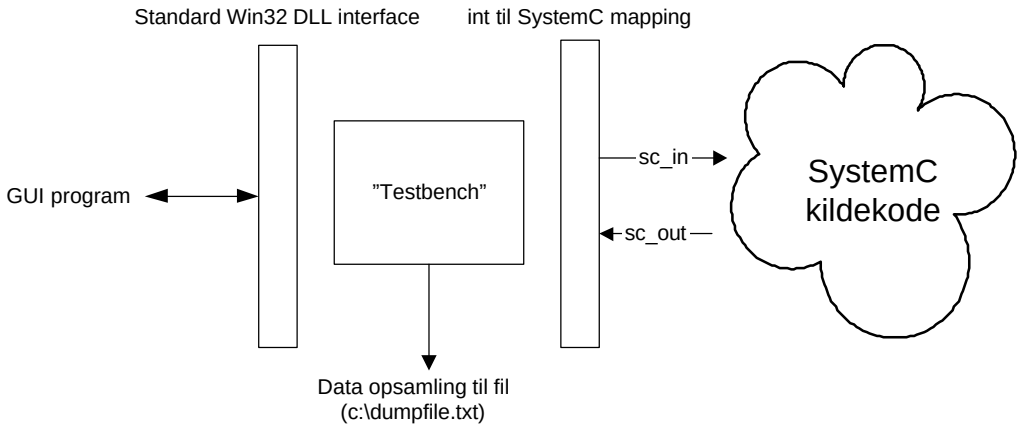
Den nemmeste måde at teste systemet på, er at se om de figurer, der bliver tegnet, er korrekte. Her skal der holdes øje med, at der ikke opstår huller i objekterne. De kan opstå og forsvinde, alt efter hvilken vinkel eller zoom faktor figuren betragtes med. Men det kan være svært at se pixel fejl, eller trekanter der forsvinder. Derudover vil overlap mellem trekanter ikke blive fanget på denne måde. Umiddelbart sker der ikke noget ved, at der opstår fejl, så længe de ikke kan ses, men hvis de medfører ekstra beregninger, medfører de en dårligere frame rate. F.eks. vil overlappende trekanter betyde, at de samme pixels skal beregnes flere gange. Overlappende pixels kan detekteres visuelt ved at indføre lidt ekstra i pixel beregningen, der gør, at når en pixel skrives til en placering, hvor der allerede er en pixel, gives den berørte pixel en iøjnefaldende farve. Dette kan ikke bruges på objekter, der har overlap i z-retningen, da de vil have overlappende pixels, som skal tegnes. Et brugbart eksempel kunne være et rundt objekt, f.eks. en kugle.
Problemet med disse visuelle tests er, at de kun kan give et svagt hint om, hvor fejlen er.

6.2 Hybris simulator

Igennem udviklingsforløbet har det været nødvendigt at udvikle en testplatform, som hurtigt kan teste funktionaliteten af det udviklede system, for at undgå relativt langsomme eksisterende systemer.
Det har i projektforløbet resulteret i flere forskellige simuleringsprogrammer, som alle har været baseret på en fælles platform, hvilket har lettet udviklingen af nye simulatorer. Den fælles platform har udviklet sig i takt med behovet for hurtig validering af SystemC koden, og principperne er forklaret i dette afsnit, så eventuelle fremtidige projekter kan drage nytte af simuleringsmiljøet.

Af de udviklede simulatorer kan 2 af disse bruges med den færdige udgave af SystemC koden, og i appendiks B er der en brugervejledning til disse simulatorer.
De to simulatorer er *Hybris Simulator*, som er udviklet til at teste og simulere det komplette Hybris system, og *Renderer Simulator*, som er blevet brugt til at teste Hybris systemets renderingsdel.

Simulator programmet består af 2 dele; en ”testbench”, som kommunikerer med Hybris koden, og et Windows program, der bruges til at styre simuleringsforløbet og vise udvalgte data fra Hybris systemet. Sammenhængen mellem simulatoren og SystemC koden ses på figur 35.



Figur 35: Sammenhæng mellem simulator og SystemC koden

6.2.1 Testbench

Testbench består af én C++ fil, som har tre hovedfunktioner:

- Binding af SystemC kodens signaler til simulatorens datastruktur
- Udtrækning af ønskede data til en dataopsamlings fil
- Kommunikation med *Hybris Simulator* eller *Renderer Simulator*, via et DLL interface

Simulatorens datastruktur består i princippet af to **struct** blokke. En struktur til de data der skal ind i SystemC koden, og en struktur til de data der ønskes ud af systemet. De to blokke bliver hhv. hentet fra og afleveret til simulatorprogrammerne via DLL interfacet.

Data opsamlingen sker ved, at en fil åbnes, når *testbench* startes. *Testbench* vil, efter hver periode, aflevere de ønskede data til en tekst fil, så det er muligt at holde øje med, hvordan data ændrer sig.

Endelig indeholder *testbench* et DLL interface, som består af 4 funktioner. Dette interface er forklaret i tabel 11.

Funktion	Beskrivelse
GetSignals(unsigned int *Mapped)	Afleverer opsamlede data til Hybris Simulator Desuden skrives data til data-opsamlings filen
SetSignals(unsigned int *Mapped)	Skriver nye data til SystemC koden
ClockSystem(int ClockCount)	Simulerer en enkelt periode i SystemC systemet. Hvis ClockCount er nul, ændres clock-signalet ikke
StopSystem()	Kaldes, lige inden Hybris Simulator lukkes, så testbench kan lukke data-opsamlings filen

Tabel 11: DLL interface

Årsagen til, at simulator programmet er delt i to, er, at *Hybris Simulator* og *Renderer Simulator* er programmer med grafiske brugerinterfaces (GUIs), og er udviklet i Borland C++

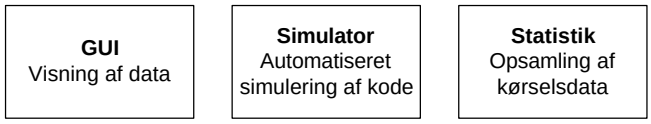
Builder 6, på grund af de gode muligheder for at udvikle grafiske applikationer hurtigt[34]. *Testbench* er udviklet i Microsoft Visual Studio .NET, og er en del af det projekt, som også inkluderer hele SystemC koden.
Der er en *testbench* og tilhørende grafisk simulator til hver af de udviklede simulatorer, og dermed også selvstændige projektfiler. De to *testbench* projekter hedder hhv. ”Hybris Simulator” og ”RendererTest”, og ligger på cd’en i biblioteket ”TestSuite”. Sammenhængen mellem de vigtigste SystemC og C++ filer ses af bilag 13. Med blåt er skrevet de navne, som de forskellige dele af systemet bliver instantieret med. Bilaget dækker begge simulatorer, dog består *RendererTest* kun af den del af træet, som har med ”TileRenderer” at gøre, hvilket er indikeret med gult.

6.2.2 Simuleringsprogrammerne

Begge simulatorer bygger på en fælles platform således, at de begge har en række fælles funktioner. Grundstenen, i begge simulatorer, er en ”MasterClass”, som håndterer disse funktioner:

- Binding og kommunikation med *testbench*, via DLL interfacet
- Indlæsning af eventuelle simuleringsdata
- Oprettelse af Simulatorens datablokke
- Korrekt afslutning af program

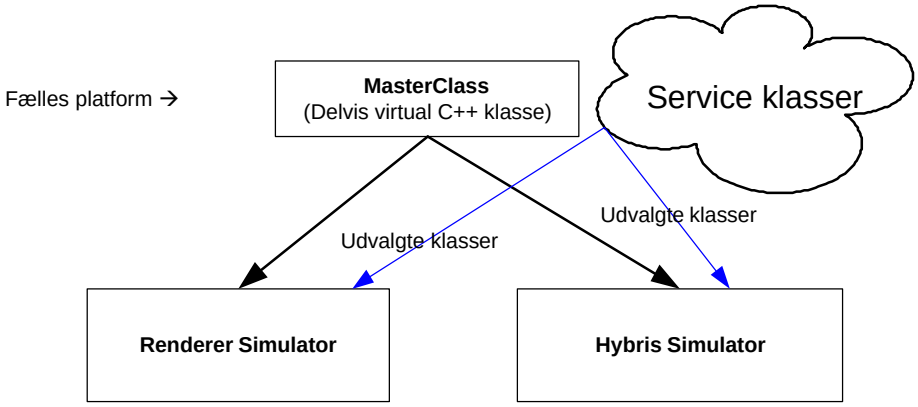
Desuden er der udviklet en række serviceklasser, som kan deles i 3 grupper, som vist på figur 36. Disse klasser giver mulighed for at lave et automatiseret simuleringsforløb, automatiseret dataopsamling, og endelig giver de mulighed for at få aktuelle SystemC data visualiseret. Statistik delen samler løbende diverse simuleringsdata, såsom hvor mange trekanter, der er tegnet, hvor lang tid der er brugt, og hvor effektivt det foregår. Dermed kan der opsamles realistiske måledata, til baggrund for en performance vurdering af systemet.



Figur 36: Service klasser

Begge simulatorer bruger et udvalg af disse service klasser til at vise data og til at køre en simulering. Funktionaliteten af de enkelte klasser er beskrevet i bruger vejledningen. Illustrativt kan opbygningen af simulatorerne ses på figur 37.

[34] Desuden skulle det meste af koden kunne kompileres direkte til en Linux platform, vha. Borland’s Kylix program. Dog skal DLL interfacet ændres, da dette kun kan bruges på en Win32 platform



Figur 37: Simulatorernes fælles platform

Begge simulatorer bruger, ud over den fælles platform, GUI klasserne til at vise data, mens det kun er *Hybris Simulator*, som benytter simulator og statistik modulerne.

Simuleringsdelen kan blandt andet bruges til at tegne frames automatisk. Data kommer fra simuleringsfiler, der indeholder alle de trekanter, som en scene består af, og programmet er i stand at simulere pakke dekoderen, således at det udviklede Hybris system kan simuleres med en række komplette scener.
Simuleringsfilerne kan laves med en lettere modificeret udgave af HAHL's *Hybris Demo* program[35]. Der er i programmet tilføjet kode, som kan udtrække samtlige trekanter, i en scene, til en fil, hvor data bliver lagt i et format som *Hybris Simulator* kan overføre til hardware Hybris systemet uden yderligere modifikation. Dvs. at alle koordinater bliver oversat til fixed point tal, inden de bliver skrevet.
I filerne gemmes desuden information om, hvor mange trekanter scenen består af og hvor mange af disse, der bliver afleveret til sorterings rutinen, så det kan beregnes, hvor stor en procentdel af trekanterne, en tegnet scene udgør.

6.3 Simulering af systemet

Ovennævnte simulatorer er brugt for at sikre, at systemet er brugbart. Det er til dels sket ved at sende udvalgte trekanter igennem *Renderer*, og ved at sende en række objekter igennem hele systemet, og sammenligne dem med de samme objekter tegnet af software udgaven.
Trekanterne, der er sendt gennem *Renderer*, er lavet ud fra principper om, at de skal aktivere alle dele af koden for at teste, om de opstillede beregningsprincipper fungerer som forventet. Det er dog ikke muligt at teste alle tænkelige typer af trekanter, da det vil give millioner af kombinationer. Pga. dette er der i udviklingsforløbet maksimalt testet med 2 trekanter per forgrening i koden, hvor antallet af tests er reduceret ved, at én trekant kan indeholde flere specialtilfælde. Denne form for test fanger de fleste fejl, men ikke alle, da der er kombinationer, som er svære at forudsige, der vil give anledning til fejl.
Rendering af objekter tester hele systemet, og giver en god indikation af, om der er fejl i systemet. Der er desværre enkelte pixelfejl i hvert billede, som det ikke er lykkedes at finde. Det skyldes, at pixelfejl på et helt billede kan være meget omstændige at finde, da det først

[35] Det modificerede program, inkl. kildekode findes på cd'en

skal lokaliseres, hvor fejlen sker. Derefter skal der simuleres ”baglæns” for at se, hvor fejlen opstår.
Billederne, der er dannet med hhv. software udgaven og det udviklede system ses af bilag 14.
Der er ikke udført simuleringer, hvor der testes for pixeloverlap, da dette ikke var muligt at nå inden for tidsrammen af dette projekt.

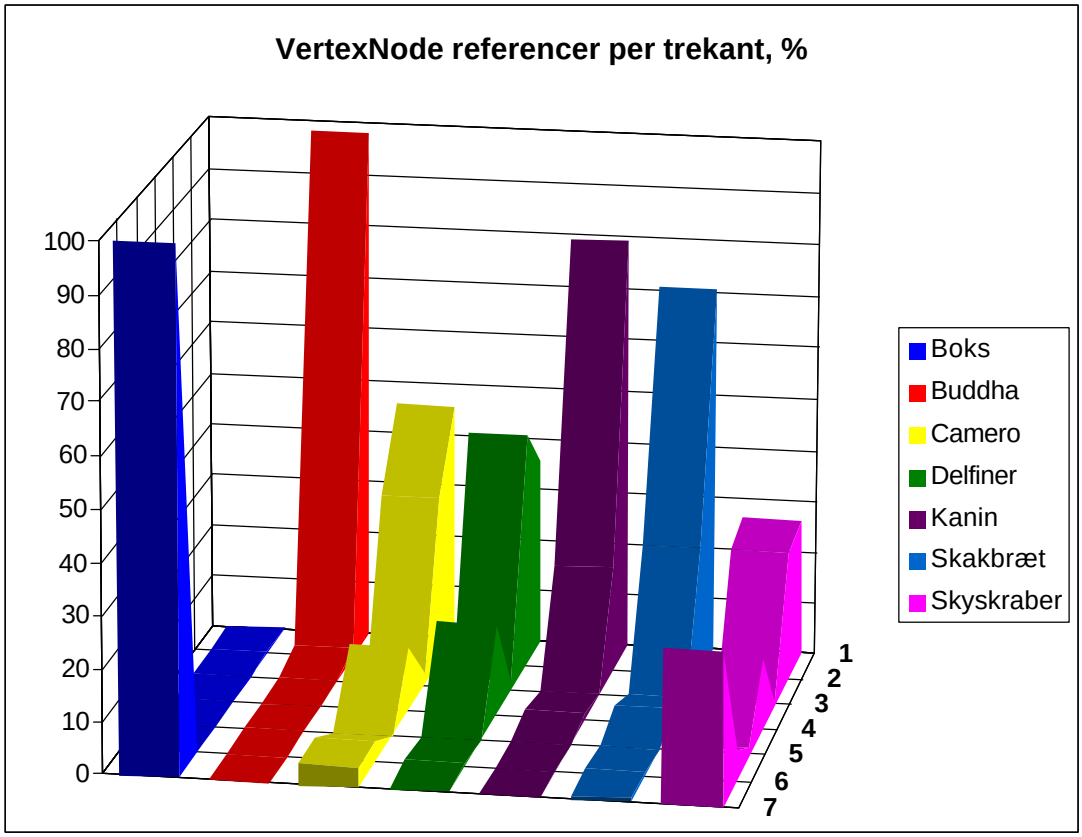
6.4 Simulering

I de næste afsnit belyses det hvor hurtigt, og effektivt det udviklede system er.
Alle simuleringer er udført med simuleringsfiler, som er trukket ud fra *Hybris Demo* programmet. Simuleringerne er valgt ud fra ønsket om at belaste det udviklede system så forskelligt som muligt, således at det belyses, hvordan systemet arter sig med objekter af forskellig størrelse og kompleksitet.
Det er valgt at simulere med en opløsning på 640x480, da det er denne opløsning, som *Hybris Demo* startes med, og det dermed formodes, at denne er brugt som standardopløsning til tests i tidligere projekter. Der er dog lavet en enkel simulering i 2048x2048, for at belyse cache systemets performance i fuld opløsning. Simulatoren har ingen ventetid og afleverer derfor trekanter i det tempo, Hybris systemet kan følge med til.

Kompleksiteten, af de benyttede simuleringsfiler, fremgår af tabel 12, og i figur 38 illustreres det, hvordan størrelsen af trekanterne fordeler sig i de enkelte figurer. Som det ses simuleres der med figurer, hvor alle trekanter dækker over mere end en tile, og med store komplekse figurer, hvor stort set samtlige trekanter ligger inden for én tile. Bilag 15 viser måleresultaterne fra simuleringerne.

Tile overlap	Figur						
	Boks	**Buddha**	**Camero**	**Delfiner**	**Kanin**	**Skakbræt**	**Skyskraber**
1	0	228120	786	291	18881	2365	463
2	0	4228	555	367	4734	825	432
3	0	0	38	14	0	12	2
4	0	58	212	142	278	102	235
5	0	0	0	0	0	0	19
6	0	0	68	5	0	24	102
7 eller mere	6	0	69	0	0	22	517
Trekanter	6	232406	1728	819	23893	3350	1770

Tabel 12: Tile overlap i simulator figurer, og deres kompleksitet



Figur 38: Fordeling af tile overlap for simuleringsobjekter

6.4.1 Beregning af DDR modulets reelle ydeevne

For at få et indtryk af Hybris systemets ydelse er en beregning af dets svageste led foretaget. Svageste led er DDR-SDRAM modulet, som godt nok har en båndbredde på 2,1GB/s, men da det benyttede Xilinx interface ikke kan udnytte det mere end 27%, er der kun omkring 500MB/s til rådighed. Men, som Hybris modellen på bilag 2 viser, skal der være mindst 2GB/s til rådighed for at sikre, at hukommelsen ikke kan blive til en flaskehals, eftersom *SortUnit* i teorien kan aflevere en VertexNode på 4 perioder[36].

Det er undersøgt, hvor lang tid en datatransaktion tager i alle tænkelige scenarier. Da der kun kan skrives eller læses med burst-længder af 4 eller 8, svarer dette til 4 mulige overførsler. Hver af disse 4 overførsler tager konstant tid. De fire tider er målt med *Hybris Simulator*, og resultatet ses af tabel 13.
Da undersøgelsen skal angive den reelle datamængde, der som minimum kan overføres, er der i tiden indberegnet en periode til at sætte ”Mode Register”, eftersom ingen af de fire moduler skriver eller læser ens, og derfor forventes at skulle ændre DDR modulets setup. *THTB Insertion* og *CreateBox* skriver dog begge 256 bit i de situationer, hvor *THTB Insertion* har cache-miss med en THTB buffer, der kun er halvt fyldt. Dette indregnes dog ikke i denne

[36] 33,25millioner VertexNodes i sekundet á 256 bit ved 133MHz

worst-case beregning. Der er desuden lagt endnu en periode til, som bruges af interfacet til at skifte port.

Retning	Datamængde (bit)	Tid (perioder)	Effektiv båndbredde
Read	256	10	405MB/s
	512	12	676MB/s
Write	256	12	338MB/s
	512	14	579MB/s

Tabel 13: Overførselstider for de 4 porte

Worst-case vil i følge tabel 13 være en skrivning af 256 bit. Dertil skal indregnes 3 perioder, som skal bruges til at undersøge om de andre porte venter på en overførsel. I værste fald vil der kunne skrives 256 bit på 15 perioder, hvilket giver 284MB/s med en frekvens på 133MHz, eller 13,5% af båndbredden. Det betyder, at der i princippet kan skrives 8,6 millioner trekanter i sekundet, men eftersom der også skal skrives og læses THTB buffere, falder antallet.

I normale situationer, hvor alle 4 porte bliver serviceret, vil dette forekomme:

- Port 2 vil læse 512 bit – 12 perioder
- Port 3 vil læse 256 bit – 10 perioder
- Port 4 vil skrive 256 bit – 12 perioder
- Port 5 vil enten læse 512 bit eller skrive 256/512 bit – 12 eller 14 perioder

En komplet runde vil da tage 46-48 perioder, og vil involvere hhv. læsning og skrivning af en VertexNode. Dvs. en VertexNode kan som minimum overføres på 48 perioder. Tager det under 48 perioder at sortere en trekant i *SortUnit,* kan DDR-SDRAM interfacet være en flaskehals.
Med 133MHz giver det 2,77 millioner trekanter i sekundet, som kan passere gennem hukommelsen, og med en ønsket frame rate på 25 billeder betyder det, at en scene ikke må indeholde mere end 110000 trekanter.

I best-case vil scenariet se anderledes ud, da de 4 porte ikke vil blive benyttet lige meget. I best-case vil en VertexNode fylde én tile, og der vil kun forekomme cache-miss med fyldte THTB buffere, hvilket betyder, at der skal skrives 23 VertexNode's for hver gang, *THTB Insertion* vil skrive data. *BackEnd* vil tilsvarende læse fyldte THTB buffere, og derfor også kun læse en THTB buffer for hver 23. VertexNode.

Mønsteret vil da se således ud:

- Der skrives 23 VertexNode's – 254 perioder
- Der læses 23 VertexNode's – 208 perioder
- Der skrives en THTB buffer – 14 perioder
- Der læses en THTB buffer – 12 perioder
- Der vil være 44 perioder, hvor Port 2 og 5 undersøges forgæves

Der er taget hensyn til, at "Mode Register" kun ændres, når der læses eller skrives en THTB buffer.

I alt vil der blive overført 23 VertexNode's på 488 perioder. Det svarer til 6,3 millioner overførte trekanter per sekund. Med en ønsket frame rate på 25 billeder/s giver det 250738 trekanter per scene. Det ses, af afsnit 6.9, at Buddha figuren kan tegnes 24 gange i sekundet. Det vil sige, at det udviklede interface bliver udnyttet 88%, svarende til 5,5 millioner læste VertexNode's.

I situationer, hvor en trekants-indsættelse vil medføre cache-miss, eller hvor der arbejdes med store trekanter, vil en indsættelse i cache tage så lang tid, at båndbredden ikke kan udnyttes, hvorved flaskehalsen flyttes bort fra DDR-SDRAM interfacet.
Det samme gør sig gældende i *BackEnd*, hvor tegning af store trekanter vil medføre en pause i indlæsningen af trekanter, hvorved båndbredden igen ikke udnyttes.

6.4.2 SDRAM performance

Som udgangspunkt er det et krav, at den fremtidige frame controller har en garanteret båndbredde, som minimum svarer til 60 billeder i sekundet, da et 60Hz signal er alment accepteret, som den mindste frekvens vi kan opfatte som "rolig"[37]. Med en teoretisk opløsning på 2048x2048 er kravet, at der er 240MB til rådighed per sekund til at danne billeder. I mindre opløsninger kan denne båndbredde udnyttes til at have højere billedfrekvenser. Eksempelvis kan 240MB/s bruges til at have en billedfrekvens på 130Hz med en opløsning på 1600x1200, som er det maksimale, en normal skærm kan vise i dag.
Her følger en beregning af, hvad der er muligt med det tænkte design. Der tages udgangspunkt i, at den brugte Xilinx DDR-SDRAM controller bliver ændret til også at være SDRAM controller. For at få maksimal udbytte af båndbredden sendes kun datapakker á 256 bit, dvs. med et burst på 8. "Mode Register" skal derfor ikke ændres, da læse/skrive retning angives af kommandoen, hvorved denne ikke skal tages i betragtning i beregningerne. De forventede data transaktionstider ses af tabel 14, og det fremgår, at skrivning giver den laveste båndbredde på 312MB/s. Indregnes en ekstra periode til at undersøge den anden port, falder båndbredden til hhv. 289MB/s og 338MB/s. Dette ligger over DDR-SDRAM controllerens minimumsbåndbredde, hvilket indikerer, at en mere intelligent DDR-SDRAM controller mangler, hvis DDR-SDRAM hukommelsen skal udnyttes bedre.

Retning	Datamængde (bit)	Tid (perioder)	Effektiv båndbredde
Read	256	11	368MB/s
Write	256	13	312MB/s

Tabel 14: Forventede overførselstider for SDRAM

Hvis der skal være 240MB/s til rådighed til frame-controllerens billedvisning, betyder det, at der skal læses 71% af tiden. De resterende 29% af tiden kan bruges til at skrive. Der vil da kunne skrives 90MB/s, som svarer til ca. 92000 tiles, eftersom en tile fylder 1KB. Derfor kan der maksimalt blive dannet 22 frames per sekund, i højeste opløsning. Med en opløsning på 640x480, som det nuværende design benytter, er der båndbredde til 306 billeder i sekundet. Med farver vil ovenstående falde til en fjerdedel.

[37] Frekvenser på over 75Hz er nødvendige for, at billedet ikke skal blinke, hvis det ses ud af øjenkrogen [20]

6.4.3 Tile performance

Som det antages i afsnit 5.3.4, tager det 448 perioder at overføre en tile til SDRAM hukommelsen. Det er forudsat, at der ikke skal ventes på andre transaktioner.
Det betyder, at der maksimalt kan overføres 296875 tiles i sekundet, svarende til en datamængde på 289MB/s. I højeste opløsning er der 4096 tiles, hvilket svarer til, at der maksimalt kan overføres 72 billeder i sekundet. Med en opløsning på 640x480 vil der kunne overføres 989 billeder.
Som det fremgår af SDRAM hukommelsens performance tal, er der kun båndbredde til ca. 1/3 af det mulige antal tile overførsler; men eftersom 22 billeder i sekundet, i fuld opløsning, er nok til at give et indtryk af flydende bevægelse, så kan Xilinx's DDR-SDRAM controller godt benyttes, som grundlag for en SDRAM controller.

6.5 DDR-SDRAM Performance

Til at afgøre hvor *SortUnit* og *BackEnd* belaster hukommelsen, er dette simuleret. Da en simulering starter med at fylde hukommelsen med en hel frame, inden *BackEnd* starter, er der simuleret over en længere periode på 10 frames, for at have data, der med rimelig sikkerhed viser belastningen imellem de to enheder.
Måleresultaterne ses af bilag 15, og i tabel 15 er de vigtigste resultater trukket frem. Af tabellen fremgår det hvor mange perioder, der er simuleret over, og hvor meget DDR-SDRAM interfacet har været benyttet. Det ses, at delfin simuleringen kun belaster interfacet i ca. 15% af tiden. Det skyldes, at figuren ikke fylder meget, hvorfor den hurtigt overføres til, og læses fra, hukommelsen. Eftersom det tager ca. 450 perioder at skifte tile buffer i udgangen af *BackEnd*, vil der opstå megen ventetid, når objekterne ikke er særligt komplekse, hvilket ses af den lave udnyttelse af interfacet. Modsat er det med store figurer såsom kaninen og Buddha figuren. Disse figurer indeholder mange trekanter per tile, og *BackEnd* vil da ikke stoppe op for at vente på en ny tile buffer så tit, hvilket medfører, at belastningen på interfacet vil stige, hvilket da også fremgår. Da det kan tage op imod 48 perioder at indlæse en trekant, skal der kun være ca. 10 trekanter i en tile, før *BackEnd* kan skifte tile buffer hurtigere, end interfacet kan levere alle trekanter til en tile.

DDR-SDRAM interface	**Figur**		
	Delfiner	**Kanin**	**Skyskraber**
Perioder	1416983	5923221	5262803
Samlet belastning af interface	14,77%	73,05%	21,33%
Belastning af interface fra THTB Reader	5,85%	2,87%	4,84%
Belastning af interface fra VertexNode Reader	50,77%	43,43%	63,01%
Belastning af interface fra CreateBox	36,49%	49,76%	14,45%
Belastning af interface fra THTB Insertion	6,90%	3,94%	17,71%
Gns. perioder per udlæst VertexNode	94,40	22,34	52,98
Gns. antal perioder DDR-SDRAM interface er busy, ved udlæsning af VertexNode	7,08	7,09	7,12
Belastning af interface, stammende fra BackEnd (perioder)	56,62%	46,30%	67,84%
Belastning af interface, stammende fra BackEnd (dataoverførsler)	62,28%	52,43%	74,65%

Tabel 15: DDR-SDRAM interface belastning. Simulering over 10 frames

Det fremgår af tabel 15, at interfacet typisk er optaget i lidt over 7 perioder for en indlæsning. Til dette tal skal lægges to perioder, hvor interfacet skifter fra og til sin *idle* tilstand. I disse to perioder er *busy* signalet ikke højt, og simulatoren tæller dermed ikke de to perioder med. Det reelle tal er da godt 9 perioder i gennemsnit for en indlæsning. Sammenholdt med tabel 13 indikerer det, at ”Mode Register” ikke bliver ændret så tit som i worst-case tilfældet, og som det fremgår af tallene i tabellen, så er der en del flere VertexNode transaktioner end THTB buffer transaktioner. Derfor sættes ”Mode Register” kun sjældent. Dette fremgår ikke af tabellen, men det gør sig gældende for alle fire porte, hvorfor de i gennemsnit er ca. en periode hurtigere til at overføre, end worst-case tallene i tabel 13 viser, hvilket gør den reelle båndbredde 7-10% højere.

Da en VertexNode kun skal skrives en gang til hukommelsen, men læses flere gange, hvis den fylder mere end én tile, vil *BackEnd* kræve mindst lige så mange transaktioner som *SortUnit*. Som det fremgår af tabellen, vil et objekt med store trekanter, eksempelvis skyskraberen, skubbe balancen meget. Buddha figuren har næsten identisk fordeling af belastningen, eftersom en trekant kun sjældent skal tegnes i flere tiles.
Det fremgår desuden, at *SortUnit* bruger længere tid på at indlæse trekanter. Dette er dog ikke overraskende idet, det tager længere tid at skrive, end at læse.

6.6 Cache performance

Cache strukturen blev designet ud fra en formodning om, at en simpel cache struktur var effektiv nok til at begrænse trafikken til den eksterne hukommelse. Derfor er der udført simuleringer, som viser cache systemets formåen. Udvalgte resultater ses af tabel 16, mens alle måleresultaterne ses af bilag 15. Simuleringerne er lavet i *Hybris Simulator*, og der er simuleret én frame.

	Figur						
	Skyskraber	**Skakbræt**	**Buddha**	**Kanin**	**Camero**	**Delfiner**	**Boks**
Perioder	70161	48903	3164195	330983	33727	13749	2950
Trekanter ind	1780	3669	503146	24912	1757	826	6
Trekanter til hukommelse	1770	3350	232406	23893	1728	819	6
Trekanter fjernet i CreateBox	10	319	270740	1019	29	7	0
Trekanter fjernet i software	1	319	270661	633	26	1	0
Trekanter fjernet, forskel	-9	0	-79	-386	-3	-6	0
Fyldte THTB buffere	364	157	10228	1176	96	17	0
Cache transaktioner til hukommelse	1895	307	10362	1460	553	136	90
Gns. antal busy-perioder for overførsel, port 4	9,27	9,05	9,04	9,05	9,13	9,04	9,00
Gns. antal busy-perioder for overførsel, port 5	10,44	11,04	11,98	11,68	10,41	10,57	9,00
Port 5 overførsler i forhold til Port 4 overførsler, %	107,06%	9,16%	4,46%	6,11%	32,00%	16,61%	1500%
Indsatte VertexNode referencer	11035	4810	236808	29461	4121	1665	206
Gns. antal VertexNode referencer pr trekant	6,23	1,44	1,02	1,23	2,38	2,03	34,33
Cache hit, %	88,66%	93,95%	95,62%	95,17%	90,12%	91,83%	56,31%

Tabel 16: Resultater fra simulering til måling af cache performance

Det fremgår af tabellen, hvor mange trekanter der indlæses, hvor mange der bliver fjernet af *CreateBox*, og hvor mange trekanter der blev fjernet i software. Som det fremgår, er forskellen, mellem det implementerede system og softwareudgaven, ikke stor. Der bliver fjernet lidt flere trekanter i den implementerede udgave, og som det fremgår af billederne på bilag 14, kan der ses manglende trekanter på figurerne.
Antallet af fyldte THTB buffere fremgår ligeledes. Når dette sammenholdes med antallet af datatransaktioner, mellem cache og hukommelse, ses det, at en stor del af trafikken skyldes fyldte buffere. For figurer med store trekanter; skyskraberen, skakbrættet, cameroen og boksen, er der dog mange transaktioner, der skyldes et cache-miss.
I tallet for cache transaktioner er antallet af transaktioner i forbindelse med flush medregnet, så op mod 128 af disse skrivninger skyldes flush.
Som det ses, tager det i gennemsnit godt 9 perioder at skrive en VertexNode til hukommelsen. Dertil skal indregnes de to perioder, som simulatoren ikke opfanger, hvorfor den reelle tid er godt 11 perioder. De figurer, som har de højeste snit, er også dem med lavest cache-hit, eftersom dette betyder en større cache udskiftning og dermed en øget trafik med halvt fyldte THTB buffere. Da boksen kun indeholder 6 trekanter, bliver alle trekanter overført til hukommelsen, inden cache hukommelsen bliver tømt, hvorved ”Mode Register” aldrig ændres. Registeret ændres heller ikke, når cache hukommelsen skal tømmes, da alle THTB buffere er under halvt fyldte, hvilket medfører, at disse skrives med et kort burst.

Buddha figuren derimod, har en gennemsnitlig skrive tid på 13,98 perioder. Dette skyldes, at det næsten udelukkende er fyldte buffere, der overføres til hukommelsen, hvorved ”Mode Register” næsten altid skal ændres.
Der er opstillet et tal, der viser antallet af skrivninger fra cache enheden, i forhold til antallet af trekanter. Det fremgår, at de komplekse figurer har langt færre skrivninger, end dem med store trekanter. Dette skyldes igen, at store trekanter udløser flere cache-miss situationer. Helt galt går det med boksen, hvor cache hukommelsen har 15 gange flere transaktioner, end antallet af overførte trekanter. Anderledes går det for Buddha figuren, hvor antallet af cache transaktioner kun udgør 4,5%. Det svarer til 22,42 VertexNode skrivninger for hver THTB buffer skrivning, eller tæt på en optimal overførsel.

Til sidst i tabel 16 fremgår det, hvor høj cache-hit raten er. Det fremgår, at den i alle tilfælde er over 50%. De figurer, som ikke har mange store trekanter, har alle en cache-hit rate på over 90%, så det må konkluderes, at den implementerede cache er ganske effektiv.

Da det tager 11-12 perioder at skrive en VertexNode til hukommelsen, kan det beregnes, hvor mange tiles en trekant må fylde, før cache systemet bliver en flaskehals. Da indsættelse af én reference tager 4 perioder og efterfølgende referencer 2 perioder, kan der indsættes 4 eller 5 referencer, forudsat at der ikke er cache-miss, før det tager længere tid at indsætte i cache, end det tager at overføre en VertexNode. Er der tit cache-miss, vil det tage længere tid at opdatere cache hukommelsen, end det tager at skrive en VertexNode. Det er derfor vigtigt med en høj cache-hit rate.

6.6.1 Cache performance med fuld opløsning

Cache systemet er eneste enhed i det udviklede system, der kan blive langsommere, hvis en større opløsning benyttes, da cache hukommelsen vil kunne rumme en mindre og mindre procentdel, af det samlede antal tiles[38]. I 2048x2048 vil der være plads til 3% af skærmens område. Derfor er det undersøgt, hvordan en større skærmopløsning vil influere på systemets performance. Den store Buddha figur er benyttet som sammenligningsgrundlag, og i tabel 17 ses resultaterne af simuleringen. Måleresultatet ses af bilag 15.

Som det fremgår, er der praktisk talt ingen ændring i cache systemets performance, til trods for, at der laves næsten dobbelt så mange VertexNode referencer. Det tager lidt kortere tid at overføre en THTB buffer via port 5, hvilket skyldes, at der sker flere overførsler med 256 bit, da der er flere tiles, som kun er halvfyldte, grundet den højere opløsning.

[38] De andre enheder i systemet bliver ikke ramt, da de arbejder på tiles og kender principielt ikke antallet af tiles i en skærm, kun en fremtidig Frame Controller vil kende til antallet af tiles

	Figur		
	Buddha, 640x480	**Buddha, 2048x2048**	**Forskel**
Perioder	3164195	5894855	2730660
Trekanter ind	503146	527032	23886
Trekanter til hukommelse	232406	431775	199369
Trekanter fjernet i createbox	270740	95257	-175483
Trekanter fjernet i software	270661	94168	-176493
Trekanter fjernet, forskel	-79	-1089	-1010
Fyldte THTB buffere	10228	19852	9624
Cache transaktioner til hukommelse	10362	21467	11105
Gns. antal busy-perioder for overførsel, port 4	9,04	9,05	0,01
Gns. antal busy-perioder for overførsel, port 5	11,98	11,87	-0,11
Port 5 overførsler i forhold til port 4 overførsler, %	4,46%	4,97%	0,51%
Indsatte vertexnode referencer	236808	465180	228372
Gns. antal vertexnode referencer pr trekant	1,02	1,08	0,06
Cache hit, %	95,62%	95,47%	-0,15%

Tabel 17: Cache performance, fuld opløsning

6.6.2 Worst-case frame skift tid

Den tid, et frame skift kan tage, er ikke stabil, da *Master Unit* skal vente på, at *SortUnit* bliver færdige med at skrive trekanter og BackEnd bliver færdig med at læse igangværende frame. Her beregnes et estimat for worst-case tiden, hvor det antages, at *BackEnd* lige akkurat bliver færdig samtidigt med *SortUnit*, således at *Master Unit* ikke skal vente på *BackEnd*. Hvis dette ikke er tilfældet, skal den tid, *BackEnd* bruger, adderes til nedenstående worst-case tid.

Den længste tid opnås, når følgende er opfyldt:

- Alle 128 THTB buffere i cache hukommelsen indeholder mindst 12 VertexNode referencer
- 4 VertexNode pakker i *CreateBox*'s pipeline
- Alle VertexNode pakker skal spænde over hele skærmen, dvs. 4096 tiles
- *CreateBox*'s og *THTB Insertions* FIFO buffere skal være fyldt med VertexNode pakker
- *BackEnd* skal indlæse data, bestående af tiles med én trekant

Dette vil belaste DDR-SDRAM interfacet med 100%, dvs. det tager i gennemsnit 48 perioder at gennemføre en transaktionsrunde, se afsnit 6.4.1.

Port 4 vil kun skulle skrive 20 gange, hvilket betyder, at det, efter 20 runder, kun vil tage 36 perioder at gennemføre en transaktionsrunde.
Da de indkommende trekanter alle fylder 4096 tiles, vil der være et cache-miss på 100%, forudsat at forrige trekant også fyldte hele skærmen. Det betyder, at *THTB Insertion* vil foretage 8192 transaktioner per trekant, eller 163840 i alt, eftersom den korrekte THTB buffer skal genindlæses.
Eftersom der skal indsættes 20 VertexNode referencer i alle tiles, vil der være 8 indsættelser per tile, der vil udløse en skrivning med et kort burst, hvorfor der spares 2 perioder per skrivning, eller 65536 i alt, som skal trækkes fra.
Dertil kommer de 128 skrivninger, som en komplet flush vil bevirke. Worst-case bliver da:

De 20 første transaktioner:

$$20 \cdot 48\,\text{perioder} = 960\,\text{perioder}$$

De efterfølgende 163820 transaktioner:

$$163820 \cdot 36\ \text{perioder} = 5897520\ \text{perioder}$$

De sidste 128 transaktioner, som følge af en flush:

$$128 \cdot 36\,\text{perioder} = 4608\,\text{perioder}$$

Til sidst tager det yderligere 512 perioder at nulstille TPT tabellen, før de næste trekanter kan indlæses.
Alt i alt vil det tage 5838064 perioder, eller 44ms, når der arbejdes med en frekvens på 133MHz.

Dette, tænkte eksempel, viser, at selv i en worst-case situation, vil det være muligt at gennemføre et frame skift på relativ kort tid. Arbejdes der derimod med en opløsning på 640 x 480 pixels, og med samme setup, vil der være 6000 cache-miss, hvilket resulterer i, at det i alt tager 432560 perioder at gennemføre et frame skift, hvilket svarer til 3,3ms.
I normale situationer vil der typisk kun være trekanter, der fylder 1 eller 2 tiles, og vil måske resultere i et cache-miss, og med kun 3 – 4 nodes i FIFO bufferen. Derfor vil et typisk frame skift maksimalt tage 10000 perioder, eller 75µs.

6.7 Sammenligning mellem Santiagos design og det nye design

Systemets performance kan opgøres på mange måder. Der er desværre en del flaskehalse i systemet. Derfor regnes der på, hvor hurtigt systemet kan køre, hvis disse flaskehalse fjernes. Det gøres, for at få et indblik i, hvor hurtigt systemet egentligt er blevet i forhold til tidligere.

6.7.1 Forskel i software, hardware snitflade

I det tidligere design fyldte datapakken, der blev overført til *BackEnd,* 333 bit [3]. I det nuværende design fylder pakken kun 234 bit. Dette giver en besparelse på 30%. Den nuværende struktur kan overføres med data pakker af 256 bit via en PCI bus, hvorimod det

tidligere design kræver pakker af 512 bit. Det betyder, at de nuværende design kan modtage dobbelt så mange pakker som før.
Arrangeres datapakkerne så de passer med 32 bit datablokke, skal det forrige design benytte 11 datapakker og det nuværende 8. Det betyder, at der kan overføres 27% flere trekanter.

I dette projekt er det første gang, at en trekant kun skal overføres én gang til hardware per billede. Dette skyldes, at *SortUnit* håndterer fordelingen af trekanter til tiles, hvor en trekant før blev overført et antal gange, svarende til det antal, som trekantens omsluttende rektangel dækker.

6.7.2 Hastighedsforskelle for Renderer

Det er beregnet, hvor mange perioder hhv. Santiagos design og det udviklede design bruger i *Renderer*'s tilstandsmaskiner. Først er det udregnet hvor mange perioder, der bruges fra input, til første output er klar. Herefter, hvor lang tid der går, inden næste output kommer, og hvor lang tid det tager at gøre klar til indlæsning af næste datastruktur. For *beregning af pixel* er der også et tal for en ikke gyldig linie. Til sidst er der også en beregning af, hvor lang tid det tager, at få et *Last* flag igennem. Tallene ses af tabel 18.
Pipelinen for *tilskæring af linier* er udeladt, da den vil sende data ud med samme hastighed, som *opsætning af linier* kan levere dem, og dermed ikke forsinke processen beregnings-mæssigt.

	Y-tilskæring		**Opsætning af linier**		**Beregning af pixel**	
	Tidligere	Udviklet	Tidligere	Udviklet	Tidligere	Udviklet
Første output	5	1	7	1	6	1
Andet output	NA	NA	4	1	3	1
Genindlæsning	NA	NA	0	0	1	0
Ikke gyldig	NA	NA	4	1	4	0
Last flag	5	1	8	1	7	2

Tabel 18: Tabel over tidsforbrug i de enkelte tilstandsmaskiner

Tallene, fra tabel 18, bruges til at give et estimat på hvor lang tid, det tager at tegne en trekant med 4 linier på hhv. 0, 1, 2 og 3 pixels. Første søjle viser hvor lang tid, der bruges i be*regning af pixel* enheden, hvis *opsætning af linier* kan følge med. Dette er dog ikke tilfældet for det tidligere design, da handshaket mellem *opsætning af linier* og *beregning af pixel* får *opsætning af linier* til at hænge i 2 perioder, og derfor ikke kan nå at have data klar, næste gang *beregning af pixel* er klar til data. Dette er medtaget i andet data sæt for det tidligere design. Alle tallene ses af tabel 19. Når der med parentes er angivet *multi*, skyldes det, at den angivne periode ligger parallelt med den ovenstående, og derfor ikke skal tælles med i det samlede resultat.

Antal pixels	**Tidligere**	**Tidligere med 2 Perioder venten**	**Forbedret Single processor**	**Udviklet Multi processor**
0	4	4	0	0
1	7	9	1	1
2	10	10	2	1 (*multi*)
3	13	13	3	2
Total	34	36	6	3

Tabel 19: Antal perioder forbrugt per linie til pixel beregning i en trekant på 4 linier

Der ses en meget tydelig forbedring. Noget af den er opnået ved at fjerne handshakes, men der er også hentet en del ved at optimere tilstandsmaskinerne til at levere data hver periode. Det, at tilstandsmaskinerne ikke skal bruge tid på ugyldige data, forbedrer også situationen.

For den beregnede trekant er forbedringen en faktor 6 med single processor udgaven og en faktor 12 med multiprocessor udgaven. Dette tal gælder kun for den beregnede trekant, men det fremgår, at linier tegnes langt hurtigere af det nye system. Derfor vil forbedringen også ses på større trekanter, men den vil variere alt efter, hvordan trekanten er udformet, men det kan ikke ske, at det nye system er langsommere end det forrige.

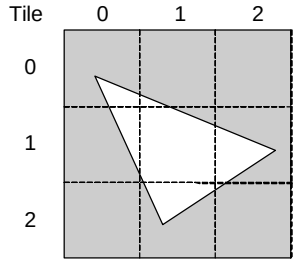
Udover antallet af trekanter per tidsenhed, er det interessant, hvor lang tid det tager, fra input til output. Dette måles fra trekanten kommer ind i *Renderer* til første pixel gemmes i tile bufferen, og indtil den sidste pixel gemmes i tilebufferen. Latency er udregnet for den tidligere udgave, og for den udviklede med samme trekant som i foregående analyse. Udregningen forudsætter at FIFO bufferne er tomme. Resultatet ses af tabel 20.

	Tidligere	**Forbedret single processor**	**Udviklet dual processor**
Til første pixel er tegnet	24	13	13
Til anden pixel er tegnet	31	14	14
Til sidste pixel er tegnet	48	19	16

Tabel 20: Latency for tidligere og nuværende *Renderer*

Igen er det udviklede design hurtigere, på trods af, at der er implementeret flere pipelines. Dog er latency forholdsvis lav for begge systemer, sammenlignet med, hvor lang tid det tager at beregne en frame, som består af flere hundrede trekanter. Derfor har det ingen betydning for den generelle hastighed. Dette viser, at hastigheden for pipeline enhederne kan øges ved at indsætte ekstra pipeline registre, uden at det betyder noget for systemets ydelse.

Pga. bounding box metoden til at sortere trekanter, vil der ved store trekanter være et stort overhead, da en trekant, der dækker flere tiles, vil få opsat linier uanset om trekanten skal tegnes i den tile, som den tilhørende omsluttende rektangel dækker. F.eks. vil tile (2,0) i figur 39 resultere i at der opsættes ca. 11 linier.



Figur 39: Bounding box, der vil være overhead ved tile (2,0)

I begge udgaver vil dette give overhead i form af ekstra linier, der skal sættes op, dog vil den gamle udgave bruge ekstra perioder, da den er flere perioder om at opsætte en linie. Dertil

kommer, at den nye version vil fjerne de beregnede linier i afrundingsenheden, og vil derfor ikke give overhead i *beregning af pixel* enheden.

6.8 Begrænsninger i BackEnd

Den maksimale hastighed *BackEnd* kan opnå, er 4 pixels per periode. Det vil dog sjældent ske i praksis, da det kræver, at der ikke er nogle linier med et ulige antal af pixels.

Største begrænsning for det antal trekanter, der kan tegnes per sekund, i *BackEnd*, er det antal trekanter, som kan hentes ind fra hukommelsen. I bedste tilfælde tager det 5 perioder, hvilket giver mulighed for at tegne 20 pixels per trekant ved fuld udnyttelse af *BackEnd*. Worst-case, hvor der ikke stalles pga. manglende data i FIFO buffere, er når trekanten indeholder 10 linier, med en pixel i hver, hvor den første linie ikke vil blive tegnet pga. afrunding. Det vil kun give 9 pixels per trekant. Færrest pixels opnås, når der sendes trekanter med kun en pixel. Resultaterne er opsummeret i tabel 21.

	Best case	**Uden tomme FIFO buffere**	**Worst case**
Pixels per trekant	20	9	1
Mpixels per sekund v. 66,6MHz	266	125,4	66
Mpixels per sekund v. 85MHz	340	161,5	85
Mpixels per sekund v. 95MHz	380	180,5	95

Tabel 21: Fillrate for de forskellige udgaver af *BackEnd*

Ønskes en trekant på 9 pixels beregnet vil det, tage 5 perioder i worst-case, mens det i best-case kan gøres på 3 perioder, da der kan tegnes 4 pixels per periode.

6.8.1 Udnyttelsesgrad for multiprocessor delen

For at få en idé om, hvorvidt de opstillede betragtninger for multiprocessor delen er korrekte, er forskellige simuleringer udført, hvorfra der er opsamlet statistik.
Det ønskes vurderet, om det er en god idé med flere enheder til at håndtere linier og flere enheder til at håndtere pixels. Målingerne er foretaget med følgende betingelser: Antallet af pixels er målt før z-sammenligningen, sådan at alle beregnede pixels tæller med. Perioder, hvor der ikke beregnes nogen pixels, er udeladt, for at manglende performance, i udlæsning af tiles til frame bufferen ikke skal påvirke resultatet. Samtidigt med reduceres påvirkningen fra hukommelsens performance også, men kan dog ikke reduceres helt.
For hver tile skift vil der være overhead, da der ikke kan startes på en ny tile, før alle enheder er færdige med den nuværende tile.

Uddrag af måledata er samlet i tabel 22 og gennemgås punktvis, og alle måledata ses af bilag 15. Figurer, med mindre end 10.000 trekanter, vil blive betegnet som figurer med få trekanter, og figurer med flere vil blive betegnet, som figurer med mange trekanter.
Udnyttelsesgraden ses direkte af, hvor mange pixels der i gennemsnit skrives per periode. For figurer med mange trekanter er denne lav, mens det for figurer med få trekanter ligger over 85%, hvilket svarer til 3,4 pixels per periode.
Yderligere ønskes belastningsfordelingen mellem de enkelte enheder undersøgt. Fordelingen mellem lige og ulige linier vil give gode resultater for stort set alle figurer, da det er de

færreste figurer, som kun indeholder pixels i hver anden linie i en tile. Dog vil figurer med trekanter, bestående af 2 linier hvoraf den ene bliver fjernet pga. afrunding, aldrig kunne udnytte denne opdeling, da der kun kan behandles en datastruktur per periode. Derfor vil de 2 *opsætning af linier* enheder aldrig være aktive samtidigt. Ved større trekanter vil det dog aldrig være helt lige fordelt, hvilket fremgår af tallene. Det er målt, hvor tit det kun er det ene sæt af *beregning af pixel* enheder, der leverer data. Sammenlignes det med hvor mange gange, der leveres data i alt, kan overhead aflæses direkte. Dette tal ses af rækken ”Kun lige eller ulige linie aktiv”. Tallet er påvirket af, at udlæsningen af datastrukturer fra hukommelsen ikke kan følge med *BackEnd*, hvorved der ikke opnås beregninger af en trekant hver periode. Kunne dette undgås ville tallene være bedre, men det er kun for de figurer med mange små trekanter, hvor det er et problem. For de resterende er der et overhead på under 15%.
Belastningen af de 4 *beregning af pixel* enheder fremgår af tabel 22. Her er der fokuseret på, hvor stort et overhead der er for de *beregning af pixel* enheder, der regner på samme linie. Dette er gjort ved at måle, hvor tit de begge udlæser pixels, i forhold til det totale antal udlæsninger. Igen er det figurer med små trekanter, der ikke har en god udnyttelse, ellers ligger de resterende figurer over 75% i udnyttelsesgrad. Her vil der altid være et overhead, da alle linier med et ulige antal pixels vil give overhead.
Målingen, for hvor tit der kun er en enhed aktiv, viser, at det for figurer, med få trekanter, sker i under 5% af skrivningerne, hvilket igen viser at multiprocessor delen har sin berettigelse.
Målingen for 4 samtidigt aktive enheder bekræfter også, at multiprocessor strukturerne bliver udnyttet, da det i over 65% af tilfældene sker for figurer med få trekanter.

	Boks	**Buddha**	**Camero**	**Delfiner**	**Kanin**	**Skak-bræt**	**Sky-skraber**
Trekanter	206	236808	3121	1665	29641	4810	2792
Pixels	48251	218443	163121	63006	188490	111037	750789
Gns. Pixels per trekant	234,23	0,92	52,27	37,84	6,40	23,08	268,907
Skrivninger	13122	179145	45892	17358	75527	31269	199925
Gns. Pixels per skrivning	3,68 91,93%	1,22 30,48%	3,55 88,86%	3,63 90,74%	2,50 62,39%	3,55 88,78%	3,76 93,88%
Kun lige eller ulige linie aktiv	12,78%	95,40%	5,74%	3,62%	38,15%	4,93%	2,78%
Begge lige aktive	84,71%	8,67%	79,94%	83,51%	43,80%	80,38%	89,14%
Begge ulige aktive	95,78%	8,66%	81,25%	83,10%	43,92%	79,65%	89,17%
1 aktiv	0,67%	81,05%	1,48%	0,96%	22,97%	2,43%	1,18%
2 aktive	12,74%	16,68%	9,25%	6,26%	28,53%	9,12%	4,06%
3 aktive	4,79%	1,55%	21,61%	21,62%	24,48%	19,36%	12,81%
4 aktive	81,79%	0,72%	67,66%	71,16%	24,03%	69,08%	81,95%
Minimum 2 aktive	99,33%	18,95%	98,52%	99,04%	77,03%	97,57%	98,82%
Minimum 3 aktive	86,59%	2,27%	89,27%	92,78%	48,50%	88,45%	94,76%

Tabel 22: Performance data for multiprocessor *Renderer*

Sammenligningerne viser, at udnyttelsesgraden af multiprocessor enhederne er meget afhængige af hvilken figur, der tegnes. Generelt ses det, at figurer med mange små trekanter opnår en dårligere udnyttelse, end dem med store, hvilket også måtte forventes, da

multiprocessor fordelene kun opnås, når der er flere pixels i en trekant. Det er tydeligt, at der gennemsnitligt opnås en stor udnyttelsesgrad for multiprocessor enhederne, og det kan derfor godt betale sig at bruge lidt ekstra plads i chippen på multiprocessor enhederne.
Om det er det rigtige antal af enheder, der er valgt, er svært at afgøre, men umiddelbart vil det koste for meget logik at indsætte ekstra *opsætning af linier* enheder, eftersom at opsplitningen af data til dem vil blive omfattende pga. de øvre og nedre trekanter. Derimod kan det overvejes at indsætte flere *beregning af pixel* enheder. Det kan forholdsvis let lade sig gøre, men det kræver en større opdeling af tile bufferne og giver dermed et større BlockRAM forbrug.
Skal der bruges flere BlockRAM moduler, vil det være mere interessant, at indsætte en komplet kopi af *Renderer*, sådan at der regnes på 2 forskellige tiles samtidigt.

6.9 Sammenligning med tidligere udgaver

I tabel 23 ses de frame rates, som simuleringsfigurerne renderes med i både HAHL's *Hybris Demo* program og i det udviklede system. Figuren med skakbrættet får desværre *Hybris Demo* til at gå ned, så her kendes frame raten ikke. Beregningerne er foretaget udfra, at dual clock systemet arbejder med en frekvens på 133MHz.
Det ses, at der er tale om en væsentlig forbedring i forhold til softwareversionen, og der opnås som minimum en forbedring på faktor 5.

	Figur (frames/s)						
	Boks	**Buddha**	**Camero**	**Delfiner**	**Kanin**	**Skak-bræt**	**Sky-skraber**
Hybris Demo	25	3-5	35-40	18	31-34	NA	30-40
Udviklet system	711	24	542	771	200	602	208
Forbedrings-faktor	28	5	13,5	43	6	NA	5

Tabel 23: Hybris Demo frame rate i 640x480. Målt på PC m. Pentium 4, 1,7GHz

De resultater, der er tilgængelige fra tidligere udgaver, er opgivet uden skærmopløsning [2f], hvilket tilføjer en vis usikkerhed til sammenligningen, men resultaterne er vist i tabel 24. Tallene viser en forbedring, men der kan ikke konkluderes noget konkret, da der ikke eksisterer et korrekt sammenlignings grundlag.

Implementering	**Buddha (frames/s)**	**Kanin (frames/s)**
Single CPU	2,5	19
Dual CPU	3,9	29
FPGA	1	12
ASIC (Simuleret)	16	NA
nVidia GeForce 2 GTS	1,6	20
Udviklet system (simuleret)	24	200

Tabel 24: Performance for forskellige implementeringer af Hybris,

og et kommercielt produkt i form af et nVidia GeForce 2 GTS

6.10 Syntese

Det foregående er udelukkende baseret på hvor mange perioder, det vil tage at beregne trekanter, men for at få det omsat til frames per sekund skal det også undersøges med hvilken frekvens, det udviklede system kan bruges.
For at finde klokfrekvensen for systemet er det syntetiseret mod en Xilinx XC2V4000-4ff1152, som er den langsomste af de 3 mulige udgaver af chippen. Det gav et hardwareforbrug af Xilinx chippen, som ses af tabel 25.

	Brugt	I alt i chip	Forbrug i %
Slices	13137	23040	57
Slice Flip Flops	20205	46080	43
4 input LUTs	14333	46080	31
Bonded IOBs	993	824	120
BlockRAMs	28	120	23
MULT18X18s	34	120	28
Global Clocks	4	16	25

Tabel 25: Chipforbrug til hele systemet

Som det ses ,er forbruget af IO ben større end det antal, der er til rådighed. Derfor er det ikke muligt at gennemføre en Place & Route og finde ud af om, der kan laves en programmerings fil til chippen uden fejl.
Der bruges for mange IO ben, da der ikke er udviklet en framecontroller. Derfor bruges der ben til både z-buffer og farve buffer for 4 *beregning af pixel* enheder, hvilket svarer til 380 ben[39]. Ændres dette vil der være en udgang til SD-RAM og en til en DA-converter til videosignalet. Desuden bruger indgangen til systemet 259 ben[40], som i stedet kobles op mod en pakkedekoder, der skal indeholde et PCI interface. Sammenholdes dette, er det muligt at få antallet af ben reduceret, sådan at systemet kan passes ned i chippen.
Eftersom der ikke er implementeret nogen framecontroller, er antallet af BlockRAM ikke korrekt. Framecontrolleren skal bruge 6 BlockRAM moduler, derfor er det rigtige antal 34. Mængden af logik, der skal bruges til framecontrolleren, er det ikke muligt at komme med et estimat for. Forbruget af resourcerne på chippen er lavt, og der er derfor gode muligheder for at implementere ekstra funktioner i fremtidige udgaver.

Af tabel 26 ses det at den forventede frekvens på 66,5MHz ikke er opnået. Af timing analysen på bilag 16, ses det, at det er *CreateBox*, som ikke er hurtig nok. Denne enhed består af en pipeline, og det er derfor muligt at hæve hastigheden, ved blot at indsætte et ekstra pipeline trin.

Beskrivelse	Tid	Frekvens
Minimum periode	19,432ns	51,461MHz
Minimum setup tid	11,704ns	85,441MHz
Maximum holde tid	11,248ns	88,905MHz

Tabel 26: Tider fra syntetisering af systemet

[39] 4 tilebuffers med hver 1 z koordinat ind og ud, 2 tilebuffer adresser ud, og en farve ud, dertil 3 kontrol signaler
[40] 256 databen og 3 kontrol signaler

6.10.1 Syntese af Renderer

Ligeledes bliver *Renderer* syntetiseret mod samme chip, og hardware forbruget ses af tabel 27.

	Brugt	I alt i chip	Forbrug i %
Slices	4203	23040	18
Slice Flip Flops	5969	46080	12
4 input LUTs	5588	46080	12
Bonded IOBs	702	824	85
BlockRAMs	19	120	15
MULT18X18s	18	120	15
Global Clocks	2	16	12

Tabel 27: Chipforbrug til multiprocessor del af *Renderer*

For *Renderer* er det muligt at lave en Place & Route og generere en programmerings fil til chippen.

Af tabel 28 ses det, at der er opnået en hastighed på 81,255 MHz for systemet, dette passer fint til dual clock systemet, da det kun er krævet, at *Renderer* kører med 66,5MHz. Ifølge timing analysen, se bilag 11, ligger begrænsningen i overgangen mellem *opsætning af linier* og *tilskæring af linier*, hvilket er en overgang til en pipeline, som er lavet med Register balancing. Med en manuel opbygget pipeline kan det derfor forventes, at hastigheden kan hæves yderligere, da det fremgår, at der er koblet flere logikenheder efter hinanden uden pipeline registre imellem.

Beskrivelse	Tid	Frekvens
Minimum periode	12,307ns	81,255MHz
Minimum setup tid	11,811ns	84,667MHz
Maximum holde tid	10,399ns	97,305MHz

Tabel 28: Tider fra syntetisering af multiprocessor *Renderer*

6.11 Sammenfatning

I dette kapitel er der kort opridset nogle betingelser for, hvordan systemets funktionalitet kan vurderes.

Den primære metode, til at vurdere funktionaliteten af systemet, er vha. de udviklede simulator programmer. Visuelle tests viser, at de simulerede figurer bliver tegnet med enkelte pixelfejl. Derudover viser simuleringerne, at en simpel cache struktur er tilstrækkelig, da et højt cache-hit generelt opnås.

Beregninger for DDR-SDRAM hukommelsen viser, at den kan være en flaskehals i systemet, hvis det tager de andre enheder under 48 perioder at processsere en trekant. Beregningerne viser også, at der overføres mellem 2,8 og 6,3 millioner trekanter fra *SortUnit* til *BackEnd* per sekund, hvis der arbejdes med 133MHz.
Simuleringerne viser, at det er muligt at udnytte det udviklede DDR-SDRAM interface ganske effektivt.

Sammenligninger mellem det tidligere SystemC design af *Renderer*, og det udviklede single og multiprocessor design viser tydelige forbedringer på flere hundrede procent.
Det er ikke muligt at udlede noget omkring hastighedsforbedringer i form af framerates, i forhold til tidligere systemer, men en sammenligning mellem en ren software udgave og det udviklede system viser, at hardware versionen har en langt højere framerate. Denne er dog ikke helt korrekt, da systemet ikke har opnået den forventede frekvens på 66,5MHz, men der er forslag til, hvordan dette løses.

Systemet er syntetiseret mod Xilinx FPGA chippen og det har vist, at der er god plads til fremtidige udvidelser.

7 Konklusion

7.1 Udvidelsesmuligheder for systemet

Det udviklede system virker, men under udviklingsarbejdet er der opstået ideer til design forbedringer, det ikke har været muligt at implementere, inden for tidsrammen af dette projekt. I dette afsnit beskrives de ideer, der ikke er blevet implementeret, men som fremtidige projekter vil kunne have gavn af. Nogle er allerede nævnt i teksten.

7.1.1 Mulige ændringer i datapakken

Sent i projektforløbet stod det klart, at datapakken fra *FrontEnd* til *SortUnit* kan reduceres yderligere, da det viste sig, at det kun er heltalsværdierne, der bruges i y-retningen. Derfor kan et fremtidigt projekt fjerne de overflødige data fra pakken, som vil give 36 bit yderligere, eller i alt 58 ledige bit til ekstra data. Det vil give plads til, at z-værdien øges til de oprindelige 32 bit og samtidigt give plads til eksempelvis a-værdier. I stedet for a-værdier, kunne data pakken udvides til at indeholde farve information, foruden 32 bit z-værdier. Det kræver dog, at der skæres til 3x6 bit farver, men det har ikke den store betydning for det visuelle indtryk[41], hvis blot disse udvides til at have højere præcision internt i systemet, så glidende farveovergange kan opnås.

7.1.2 Forslag til udvidelse af THTB Insertion til parallel cache

Da indsættelse af én VertexNode i cache tager 4 perioder, og *THTB Insertion* enheden kun kan køre med dobbelt hastighed af *CreateBox* enheden, vil der reelt kun blive behandlet en VertexNode på 2 perioder, set fra *CreateBox* enheden. For at fjerne denne begrænsning, og derved opnå, at én VertexNode kan blive behandlet per periode, så kan *THTB Insertion* enheden relativt nemt opdeles i to, hvorved der teoretisk kan indsættes i to tiles per periode.

Ideen er at lade den ene enhed tage sig af de tiles, der ligger på lige linier (lige y-koordinater), mens den anden enhed tager sig af ulige linier. Forslaget ses skematisk vist i bilag 17.

Da trekanterne er grupperet, og sjældent fylder mere end en tile, vil det typisk betyde, at den ene enhed modtager trekanter, og den anden intet laver. Det kan løses ved at indsætte FIFO enheder imellem *CreateBox* og *THTB Insertion* enhederne. Disse buffere skal være store nok til at kunne indeholde nok trekanter til det gennemsnitlige antal trekanter, der vil være placeret i én tile, før der er overlap. Er dette ikke opfyldt, vil den dobbelte cache ikke øge indsættelses-hastigheden nok til, at *CreateBox* kan aflevere én trekant per periode.

I *CreateBox* skal der desuden tilføjes logik, som skal afgøre, om et omsluttende rektangel skal afleveres til begge enheder, eller kun en. Derved kan det undgås at fylde en FIFO buffer med overflødige data.

[41] TFT skærme, til bærbare computere, benytter 6 bit farver af tekniske årsager

Som det fremgår af bilag 17, er udvidelses forslaget udvidet til også at omfatte 3 hukommelses moduler, hvilket betyder, at en næsten fuld udnyttelse af den tilgængelige båndbredde kan sikres.
For at udnytte båndbredden skal der designes en controller, som er mere effektiv. Det gøres eventuelt ved at indbygge noget logik i controlleren, som holder hukommelsen åben, indtil et andet område af hukommelsen skal adresseres. Derved kan der spares meget tid ved ikke at skulle åbne og lukke op for de forskellige områder af hukommelsen.
Controlleren skal desuden være i stand til at sætte adresser op, mens en datatransaktion er i gang.
Da VertexNode og THTB buffere er helt adskilt, er det ikke nødvendigt at skifte læse/skrive retning og ændre på hukommelsens "Mode Register" på de to moduler, der kun indeholder VertexNode's. Disse data skal kun sættes efter et frame skift, hvilket ikke koster noget tidsmæssigt, da de interne moduler skal bruge et vist antal perioder på at gøre klar til en ny frame.
Hukommelses-modulet med THTB buffere kan deles af både *BackEnd* og *SortUnit*, eftersom kravet til båndbredde og hukommelses-størrelse er ca. 1/10 af kravet til VertexNode hukommelsen. Derfor vil det ikke sænke systemet, at flere enheder har adgang til modulet.

Med udgangspunkt i de eksisterende enheder, dvs. 133MHz DDR-SDRAM og 66,5/133MHz enheder, vil det give følgende maksimale data overførselsmængder:

CreateBox kan behandle én VertexNode per periode, eller 66,5 millioner VertexNode's i sekundet.

Da én VertexNode fylder 256 bit, vil kravet til båndbredden være 2029MB per sekund, hvilket vil kræve, at 96,6 procent af båndbredden kan udnyttes, hvilket bør være muligt med en effektiv hukommelsesstyring.

Båndbreddekravet til THTB bufferne er ca. 10% af ovenstående, eller godt 200MB per sekund. Selv den eksisterende ueffektive controller kan som minimum overføre 280MB, hvilket principielt set er nok. Dog vil en mere effektiv controller give lavere adgangstider, og derved kunne sikre, at *THTB Insertion* enhederne ikke sinkes så længe som nu.

Med maksimal udnyttelse af 2 komplette *BackEnd* enheder kan der opnåes en teoretisk fill-rate på 532 millioner pixels per sekund, såfremt *BackEnd* kan tegne alle trekanter uden at stalle.

Hvis denne modifikation kunne lade sig gøre, ville Hybris systemet klare sig godt i sammenligning med kommercielle systemer, som det fremgår af tabel 29.
Fill-rate tallene i tabellen er nedjusteret, så alle grafikkort kører med samme frekvens, dvs. 66,5MHz internt og 133MHz til hukommelsen, hvilket gør tallene direkte sammenlignelige. På nær GeForce 4 chippen, så kører alle kortene i tabel 29 med 8 pixel pipelines, hvilket betyder at fill-rate tallene burde være ens, når de kører med ens periode tid, men ATI's og nVidia's tal er formodentligt afrundede.

Chip	Normal frekvens (Core/Memory)	Fill-rate (mill)	Båndbredde (GB)
Hybris med dual Renderer	66,5/133	**532**	6,3
nVidia GeForce 4, TI4600	300/325	280	4,3
nVidia GeForce FX5800	**500/500**	530	4,3
ATI Radeon 9700	325/310	530	**8,5**

Tabel 29: Performance sammenligning. ATI og nVidia tal stammer fra Toms Hardware Guide [21]

7.1.3 Omsluttende rektangel

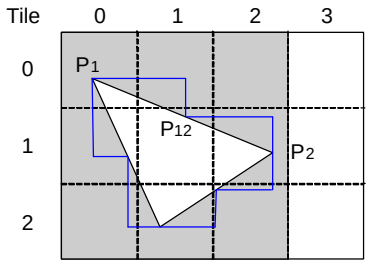
Da omsluttende rektangel metoden er en grov måde at sortere trekanter på, kunne cache systemet med fordel forbedres til kun at indsætte VertexNode referencer i de tiles, som trekanten dækker [2b]. Det vil spare hukommelses båndbredde, da der skal indlæses færre datastrukturer i *BackEnd*. Derudover giver en stor trekant meget overhead i *BackEnd* for de tiles som trekantens omsluttende rektangel dækker, men trekanten ikke selv dækker, da *BackEnd* først får sorteret et sådan tilfælde fra efter linieopsætningen, dvs. at en trekant der i tilen ved siden af dækker f.eks. halvdelen af linierne, vil spilde 8 perioder i *opsætning af linier* enheden i *BackEnd*.

Da den korrekte sortering kræver, at trekantens hældninger kendes, vil det være nødvendigt at flytte eller kopiere *hældningsberegningen* ind i *SortUnit*. Hvis dette ikke er ønskværdigt, eventuelt for at spare logik, kan en alternativ metode bruges, som skitseres her.

Midten af trekantens sider kan nemt beregnes, da denne er lig med:

$$P_{12} = \left(\frac{P_1 x + P_2 x}{2} \ , \ \frac{P_1 y + P_2 y}{2} \right)$$

Midten af de 2 andre sider findes på tilsvarende måde. Regneoperationen vil kun kræve en addition i hardware, da divisionen udføres ved at bortkaste LSB.
Med de tre midtpunkter kan det oprindelige omsluttende rektangel forbedres til det, som fremgår af figur 40. Herved undgås det at indsætte VertexNode referencer i nogle af de overflødige tiles.

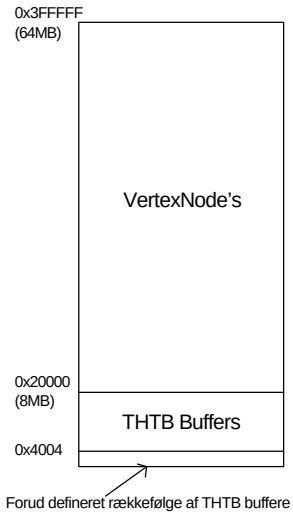


Figur 40. Omsluttende grænse

De tre midtpunkter kan beregnes i et 4. pipeline trin i *CreateBox* enheden, hvorved throughput ikke vil ændres. I *THTB Insertion* enheden skal *InsertVertexNode* processen ændres, så den kun indsætter i de tiles, som den nye "omsluttende grænse" rammer.
Metoden kan forbedres ved at dublere beregningen til at beregne et nyt midtpunkt imellem P_1 og P_{12}, og tilsvarende for de andre punkter. Dette kan gentages et vilkårligt antal gange, hvorved metoden giver mulighed for en vilkårlig præcis omsluttende grænse.

7.1.4 Korrekt linkede lister

Som beskrevet i afsnit 4.3, er der et problem med linkede lister i THTB bufferne. De kan relativt nemt ændres til at linke korrekt, således at første trekant i en tile, også vil være den, der først læses af *BackEnd*. Her følger et forslag, som uden de store ændringer af det nuværende design, vil give korrekt linkede lister.
For at vide, hvor den første THTB buffer i hver tile er placeret, skal den første del af THTB buffer området, i den eksterne hukommelse, reserveres til en fast placering af den første buffer i hver tile – se figur 41.



Figur 41: Hukommelses model med korrekt linkede lister

Indsættelsen af en pointer til næste THTB buffer skal nu placeres i den gamle THTB buffer. For at undgå ekstra trafik til hukommelsen skal det ske, når bufferen bliver dannet. Det betyder til gengæld, at der opstår henvisninger til THTB buffere, som muligvis ikke eksisterer. For at undgå dette problem, skal der gemmes information om, hvornår en THTB buffer linker korrekt.

TPT tabellen kan bruges til at sikre dette. Eftersom den ikke skal modificeres, vil TPT tabellen altid pege på sidste THTB buffer i en tile, hvilket kan udnyttes af *THTB Reader*. Når denne indlæser en THTB buffer, som har samme adresse som TPT tabellens adresse, vides det, at der ikke er flere buffere i tabellen, derfor skal *NextTHTB* adressen ignoreres.

THTB Reader skal ikke længere finde første THTB buffer, der hvor TPT tabellen peger, men i stedet finde den på den reserverede adresse i den eksterne hukommelse.

Dette forslag vil kunne gennemføres ved at ændre i *Cache Control* og *THTB Reader* processerne, og forslaget vil sandsynligvis ikke sænke systemets ydelse, da det ikke er introduceret ny kompliceret logik.

7.1.5 Mulige forbedringer i BackEnd

Selv om en stor del af *BackEnd* er blevet optimeret i dette projekt, er det stadig muligt at gøre den bedre. De fleste er små variabelforbedringer, og det formodes at de vil reducere størrelsen af *BackEnd*.
TPT Reset Engine kan fjernes, hvis *THTB Reader* nulstiller hvert index i TPT tabellen. Dette skal gøres ved at hvert index nulstilles perioden efter, at det er læst, hvilket der er tid til, da der går mindst 12 perioder imellem hver læstning fra TPT tabellen.
Fra *y-tilskæringen* og videre til *opsætning af linier*, sendes en fuld tile buffer adresse, men i *y-tilskæringen* er adressen kun beregnet til første pixel i en linie, for derefter at blive tilpasset i *tilskæring af linier*. Derfor kunne det være en fordel at overføre trekantens første y-koordinat i stedet, hvilket også vil give en fordel i *opsætning af linier*, som derefter kan nøjes med at hæve denne værdi med 1. I *tilskæring af linier* skal y-koordinaten blot rykkes 5 pladser til venstre, hvorved den oprindelige tile buffer adresse opstår. Denne optimering vil spare 5 bit i FIFO bufferen mellem *y-tilskæring* og *opsætning af linier*, og vil spare bit i additionen i linieopsætningen.

Da afrundingen af x-koordinaterne altid gør, at den første linie i den øvre trekant aldrig bliver tegnet, er der ingen grund til at bruge tid på at beregne den i *opsætning af linier*. Følgende 2 udregninger kan indsættes i *y-tilskæring*, i de tilfælde hvor den øvre trekant starter i den tile, der er ved at blive beregnet.

$$XStart = XStart + XSpanStartInc$$
$$XEnd = XEnd + XUpperSpanEndInc$$

Dette vil spare en periode i linieopsætningsenheden for alle de trekanter, som sendes igennem og starter i den aktuelle tile.

Mellem *tilskæring af linier* og *beregning af pixel* overføres der en linielængde, som er på 12 bit, da linierne maximalt kan være 32 pixels lange, er dette spild af plads, og det koster samtidigt med ekstra tid i pixelberegningen at reducere en 12 bit variabel med én sammenlignet med en 5 bit, som det burde være. Eneste fare er, at der i linieafrundingen beregnes forskellen mellem XStart og XEnd, som i nogle tilfælde bliver negativ, derfor skal der regnes med en bit ekstra i linieafrundingen, men da de tilfælde, hvor længden bliver negativ, bliver sorteret væk af linieafrundingen, giver dette ikke problemer i pixelberegningen.

7.1.6 Ændring af tile buffer størrelse

Som det er antydet, er der et stort spild ved den valgte tile buffer på 32x32 pixels. I den aktuelle opsætning, med 4 *beregning af pixel* enheder, bruges der 4 BlockRAM moduler til z-buffer og 2 til pixel buffer. Der kan ligge 2 pixel buffere i hver blok, da hver blok har 2 ind og udgange. Z skal læses og skrives i samme periode, og kræver derfor en hel blok per buffer. 32x32 z-værdier svarer til 32kbit, og skal deles ud på 4 *beregning af pixel* enheder, hvilket giver 8kbit per blok. Da der er 18kbit i en blok, giver dette et stort spild. Det er derfor oplagt at

bruge en tile størrelse, der er dobbelt så stor. Det vil give 16kbit, som ikke giver fuld udnyttelse af blokken, men eftersom en del af beregningerne kan laves med skifte operationer, i stedet for multiplikationer, når længden af tilens sider er 2^n, bør denne størrelse vælges. Det giver mulighed for enten at øge tilens bredde til 64 pixels eller højden til 64 linier.

Både x og y passerer kun en FIFO buffer som tile koordinater. Derfor kan dette ikke afgøre, på hvilken led tilen skal øges. Derimod er der stor forskel på hvor mange gange, der skal skæres, alt efter hvilken løsning, der vælges. Ved en forøgelse af y spares der maksimalt en skæring per trekant, medmindre den dækker mere end 4 tiles i y-retningen. Ved en forøgelse i x-retningen spares der maksimalt en skæring per linie i trekanten. Da en skæring medfører en ekstra datastruktur, som skal beregnes, og det er den sidste del af systemet, der er mest belastet, bør længden af x-aksen i en tile ændres til 64.

7.1.7 Forslag til projekter

I sin nuværende form mangler 2 hoveddele, før Hybris systemet kan køre i hardware:

- Packet Decoder
- Frame Controller

Førstnævnte skal bruges til at kommunikere med det system, som afleverer trekanter fra Hybris systemets *FrontEnd*, og ideen er at udnytte det PCI-X interface, der sidder på udviklingsboardet. IMM råder allerede nu over en VHDL soft-core, der kan bruges. Den udnytter dog ikke hele PCI-X interfacets båndbredde, men vil kunne bruges. Til denne core mangler dog stadig et interface ind til *Master Unit*, og der skal udvikles en effektiv software driver.
Frame controlleren skal tage data fra *BackEnd* og overføre dette til SDRAM hukommelsen. Den skal også indeholde et RAMDAC modul, som kan udlæse billeder til en RGB kompatibel skærm.
Til hjælp for udviklingen af dette, skal der laves et SDRAM interface. I afsnit 3.4 er der et forslag til hvordan, dette kan udføres udfra det eksisterende DDR-SDRAM interface. Til udvikling af RAMDAC delen kan HAHL's eksisterende modul bruges, eventuelt med modifikationer, så andre opløsninger end 640x480 kan vises.

Ud over de stillede forslag kan fremtidige projekter eksempelvis også bestå i:

- Udvikling af en kommando dekoder
- Forbedring af cache systemet, eventuelt med udvikling af cache prefetch mekanismer, eller udnyttelse af ledige BlockRAM moduler
- Indsættelse af cache i forbindelse med indlæsning af data fra DDR-SDRAM, da en del trekanter optræder i flere tiles, og det vil derfor være fordelagtigt at kunne cache nogle af de indlæste VertexNode's
- Forbedring af DDR-SDRAM controller, så båndbredden udnyttes bedre

7.1.8 Kendte fejl i det nuværende system

Der er enkelte kendte fejl i det udviklede system, der ikke er rettet. Det ses af figurerne på bilag 14. Det har ikke været muligt at lokalisere fejlen, inden for tidsrammen for dette projekt, men med en ihærdig indsats kan de elimineres.
Trekanter må ikke starte i negative koordinater, hvilket er en kendt fejl i tidligere udgaver af Hybris systemet, dertil kommer, at trekanter heller ikke må starte præcis i x eller y koordinaten 0.

7.2 Opsummering

Det oprindelige Hybris system er videreudviklet, og der er opnået et design, som på flere punkter er hurtigere end den eksisterende 3D-engine.

Hele systemet er optimeret udfra en ide om at prioritere hastighed højt, men samtidigt med holde ressourceforbruget i udviklingsboardets Xilinx FPGA på et rimeligt niveau. Dette indebærer, at datastrukturerne fra de forrige projekter er gennemgået, med henblik på at optimere dem, hvilket har medført, at der kan overføres 27% flere trekanter igennem snitfladen mellem software og hardware.

I tråd med ideen om at implementere størstedelen af Hybris systemet i hardware er sorteringsdelen og beregning af trekanternes hældning implementeret i hardware.
For at gemme trekanter, fra sorteringsdelen, benyttes et DDR-SDRAM modul, der sidder på udviklingsboardet. Da dette modul har en begrænset båndbredde, er der udviklet et cache system, for at reducere antallet af datatransaktioner mellem FPGA og ekstern hukommelse.
Til at håndtere de udviklede enheders uafhængige datatransaktioner til og fra hukommelsen er der udviklet et effektivt interface, der bygger ovenpå en eksisterende Xilinx DDR-SDRAM controller.

HDL funktionerne fra et tidligere projekt er optimeret. Det har betydet, at alle enheder i *Renderer* nu kan levere en datastruktur i hver periode, i forhold til det tidligere design, hvor enhederne brugte mellem 3 og 7 perioder per datastruktur.
Mulighederne for at opnå multiprocessering, ved at dublere udvalgte enheder af *Renderer*, er undersøgt, og har resulteret i et design, der kan levere 4 pixels per periode.

Til at evaluere det udviklede system, er der opbygget en simuleringsplatform, hvor det er muligt at fejlfinde og simulere Hybris systemet ved hjælp af et Microsoft Windows™ program.
Beregninger og simuleringer viser, at det udviklede hukommelsessystem kan overføre mellem 2,7 og 6,3 millioner trekanter imellem *SortUnit* og *BackEnd* per sekund. Generelt opnås der en høj cache-hit rate, selv om der benyttes en simpel cache model.
Simuleringer har vist, at det udviklede system er en faktor 5 hurtigere end software udgaven af Hybris.

Det udviklede system er ikke implementeret i udviklingsboardet, da der stadig mangler et businterface mod software delen af Hybris, og en framecontroller til at modtage og vise det genererede objekt på en skærm.

8 Litteraturliste

[1] Hans Holten-Lund: Fast Rendering Techniques for Real-Time 3D Image Synthesis in an Interactive Environment. Department of Information Technology, Technical University of Denmark. August 1995.

[2] Hans Holten-Lund: Design for Scalability in 3D Computer Graphics Architectures. IMM DTU, http://www.imm.dtu.dk. Juli 2001.

2a: Kapitel 3.2
2b: Kapitel 3.5
2c: Kapitel 3.3
2d: Kapitel 2.4
2e: Kapitel 3.6
2f: Kapitel 4.6

[3] Santiago Estaban Zorita: Modelling a 3D-Graphics Engine using SystemC. IMM DTU, http://www.imm.dtu.dk. 2002.

[4] Henrik Ahrendt Sørensen: SoC design-eksperimenter med en stor FPGA. Institut for Informationsteknologi, Danmarks Tekniske Universitet. Februar 2001.

[5] Avnet Avenue: Product brief, Xilinx Virtex™-II Development Kit. 2002.

[6] Werks Offset A/S: Gråtoner, http://www.werk.dk/gr_02c.htm#Gråtoner

[7] The SSU Observatory: AIP Tutorial 1, http://www.phys-astro.sonoma.edu/observatory /documentation/aip1.html. Februar 2001.

[8] Intel: AGP v 3.0 Interface Specification. September 2002.

[9] Micron Technologies Inc.: 128Mb: x4, x8, x16 DDR SDRAM. Oktober 2002

[10] Bent Poul Jensen: Dimensionering og Konstruktion af Digitale Systemer. BeeGs Forlag. 1998

[11] JEDEC: JEDEC Standard, Double Data Rate (DDR) SDRAM Specifikation, JESD79 (Release 2). JEDEC Solid State Technology Association, www.jedec.org. Maj 2002.

[12] Xilinx Inc.: Application Note, XAPP200. Synthesizable DDR SDRAM Controller, version 2.4. www.xilinx.com. Juli 2002.

[13] Xilinx Inc.: Virtex™-II Platform FPGAs: DC and Switching Characteristics.. www.xilinx.com. December 2002.

[14] Micron Technologies Inc.: 256Mb: x4, x8, x16 SDRAM. Januar 2003.

[15] Stanford Computer Graphics Laboratory. The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/. 2003.

[16] Intel: Intel Processor Roadmap. http://www.intel.com/procucts/roadmap. 2003.

[17] Thomas Gleerup: ASIC for 3D Graphics Pipeline Back-End. Department of Information Technology, Technical University of Denmark. Januar 1999.

[18] David A. Patterson og John L. Hennessy: Computer Organization & Design, The Hardware / Software Interface, Second Edition. Morgan Kaufmann Publishers, Inc. 1998. Siderne 545 - 579 .

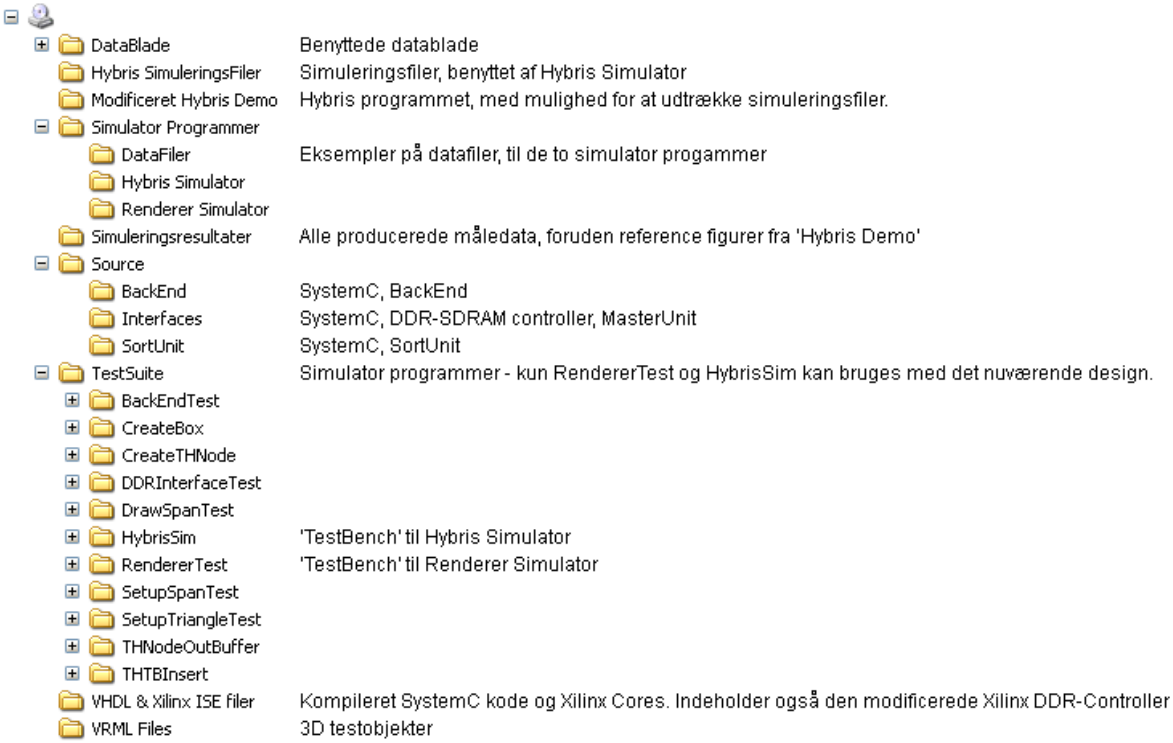
[19] Martin Lütken: Hardware realisering af 3D-grafik-pipeline. Institut for Informationsteknologi, Danmarks Tekniske Universitet. Juli 1997. Kapitel 4.

[20] Dustin D. Brand: Human Eye Frames Per Second. http://amo.net/NT/02-21-01FPS.html. Februar 2001.

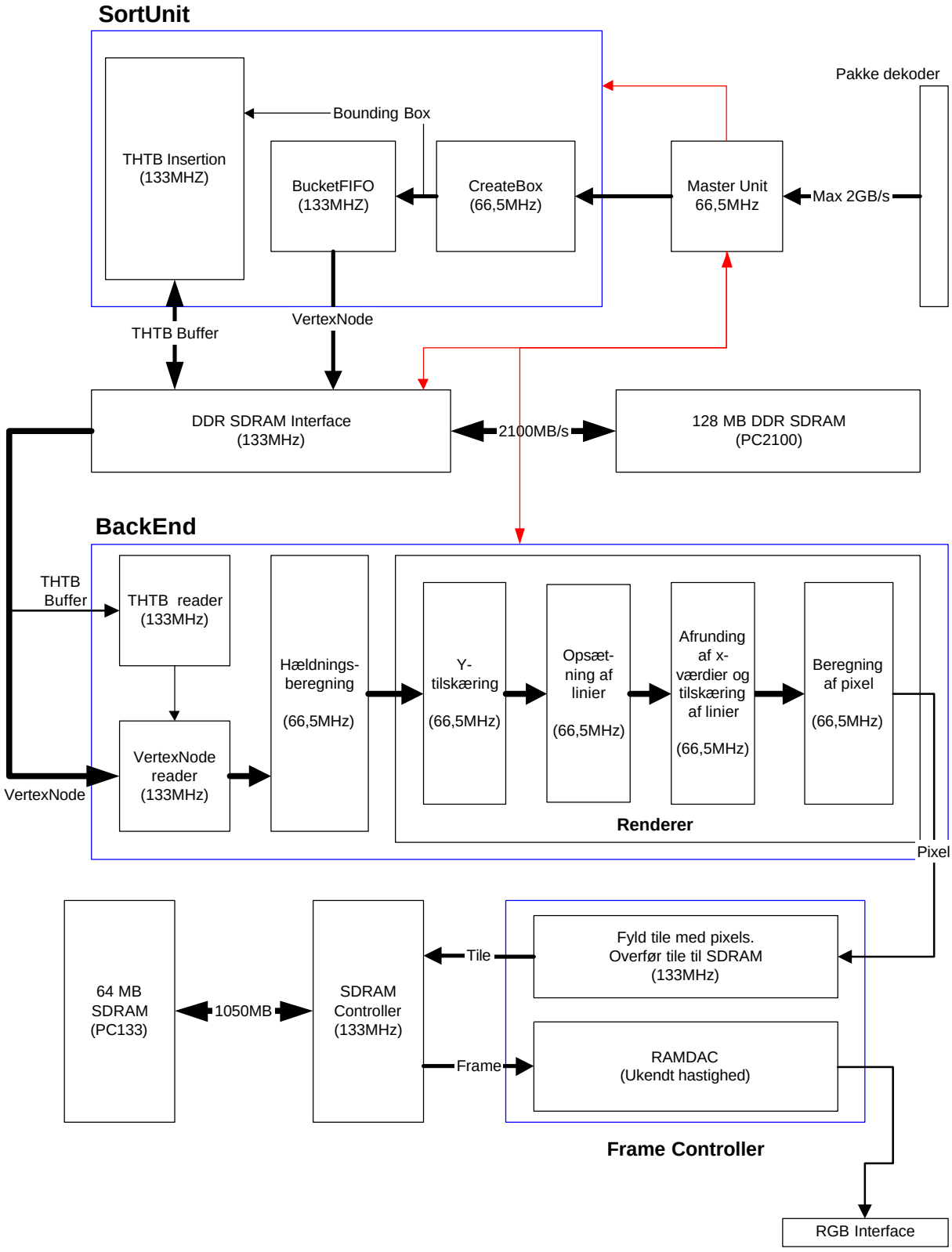
[21] Tom's Guides Publishing LLC: Tom's Hardware Guide, http://www6.tomshardware.com/graphic/20021118/index.html. November 2002.

9 Bilag

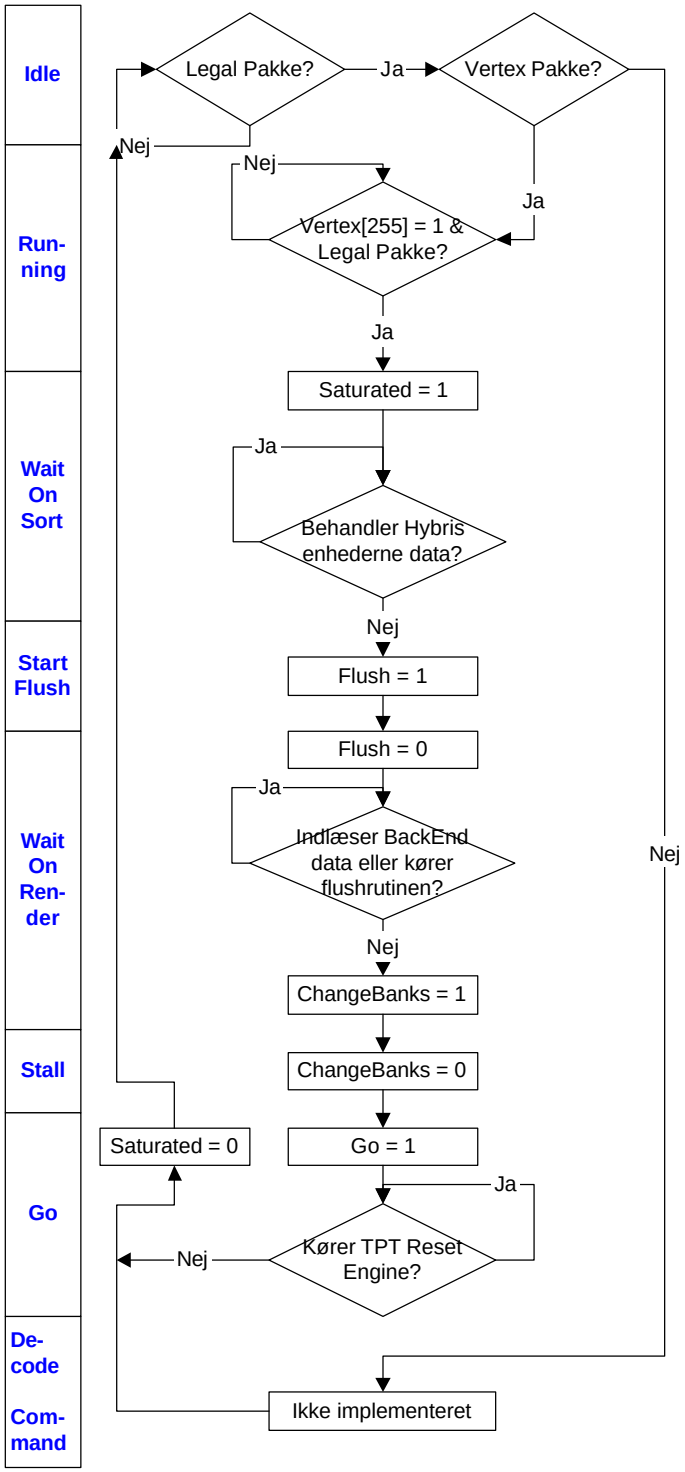
9.1 Bilag 1: Indhold af cd



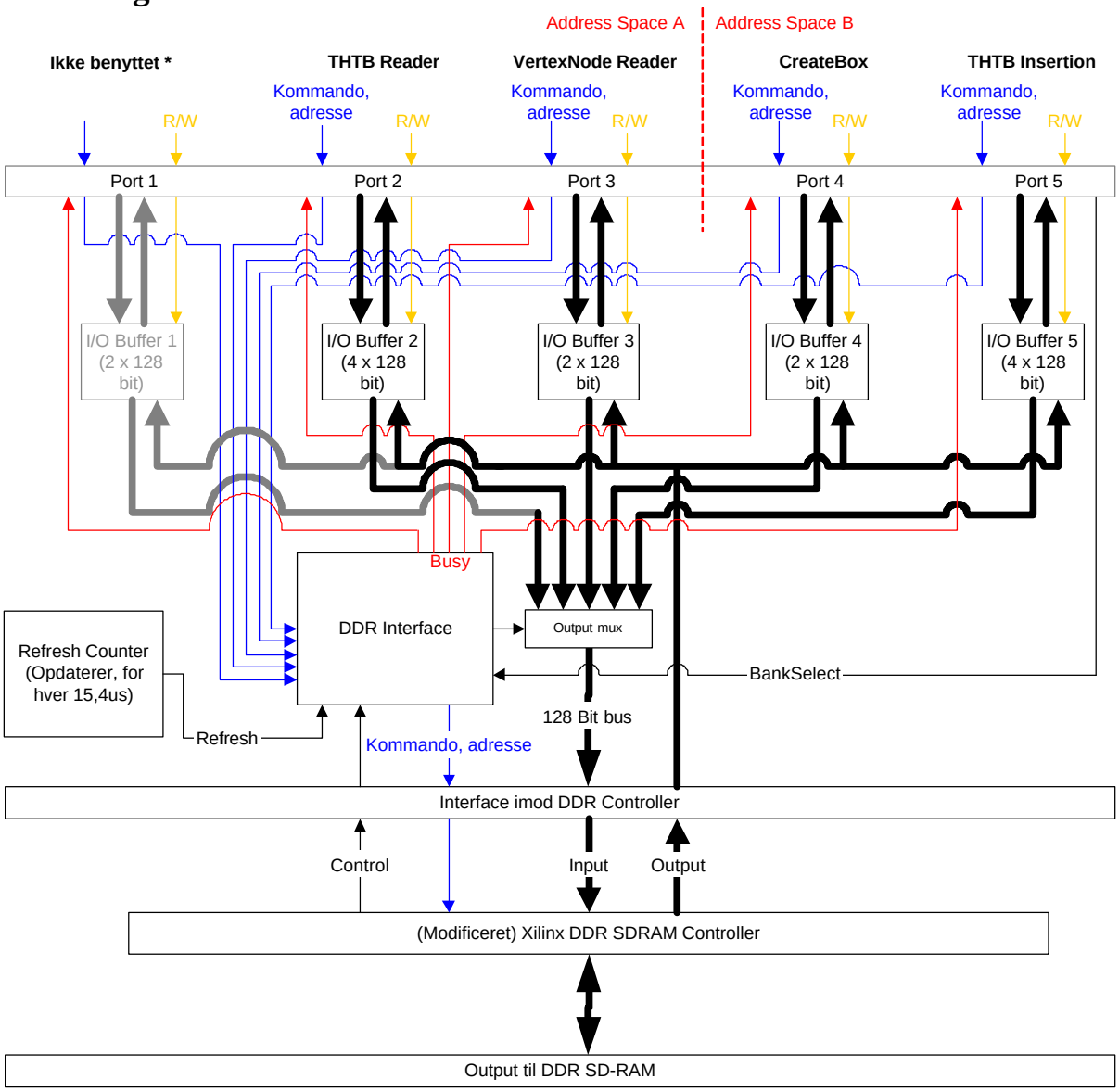
9.2 Bilag 2: Komplet Hybris figur



9.3 Bilag 3: Master unit algoritme

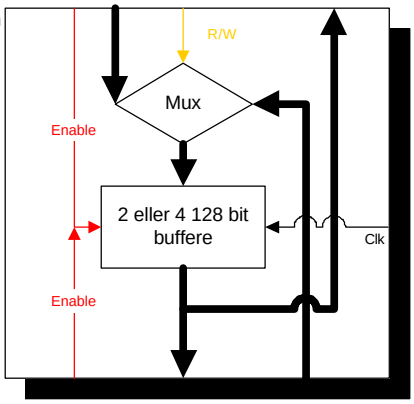


9.4 Bilag 4: DDR interface

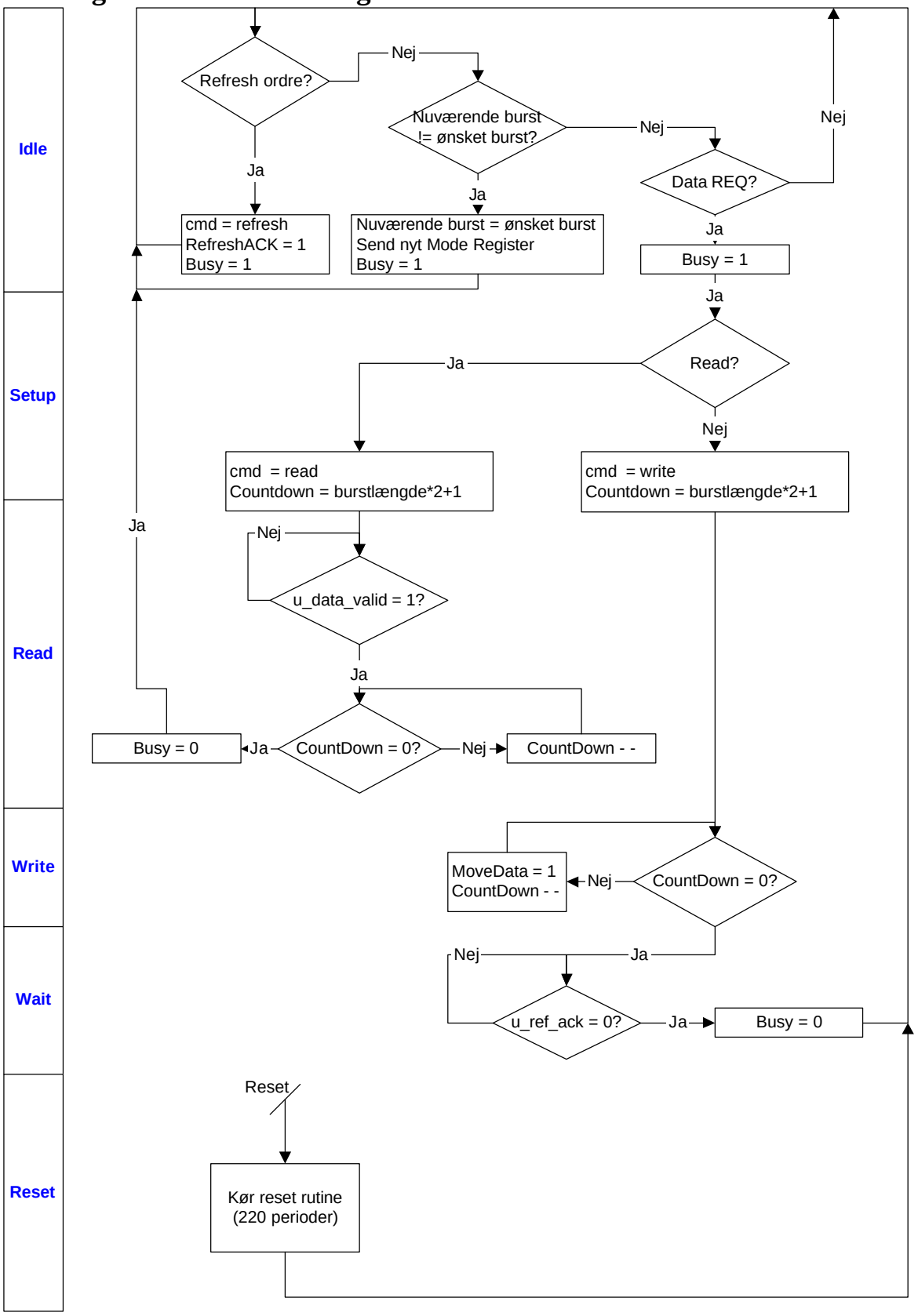


*
Prioriteret port. Oprindeligt designet til en frame controller, men ikke benyttet i det endelige design.

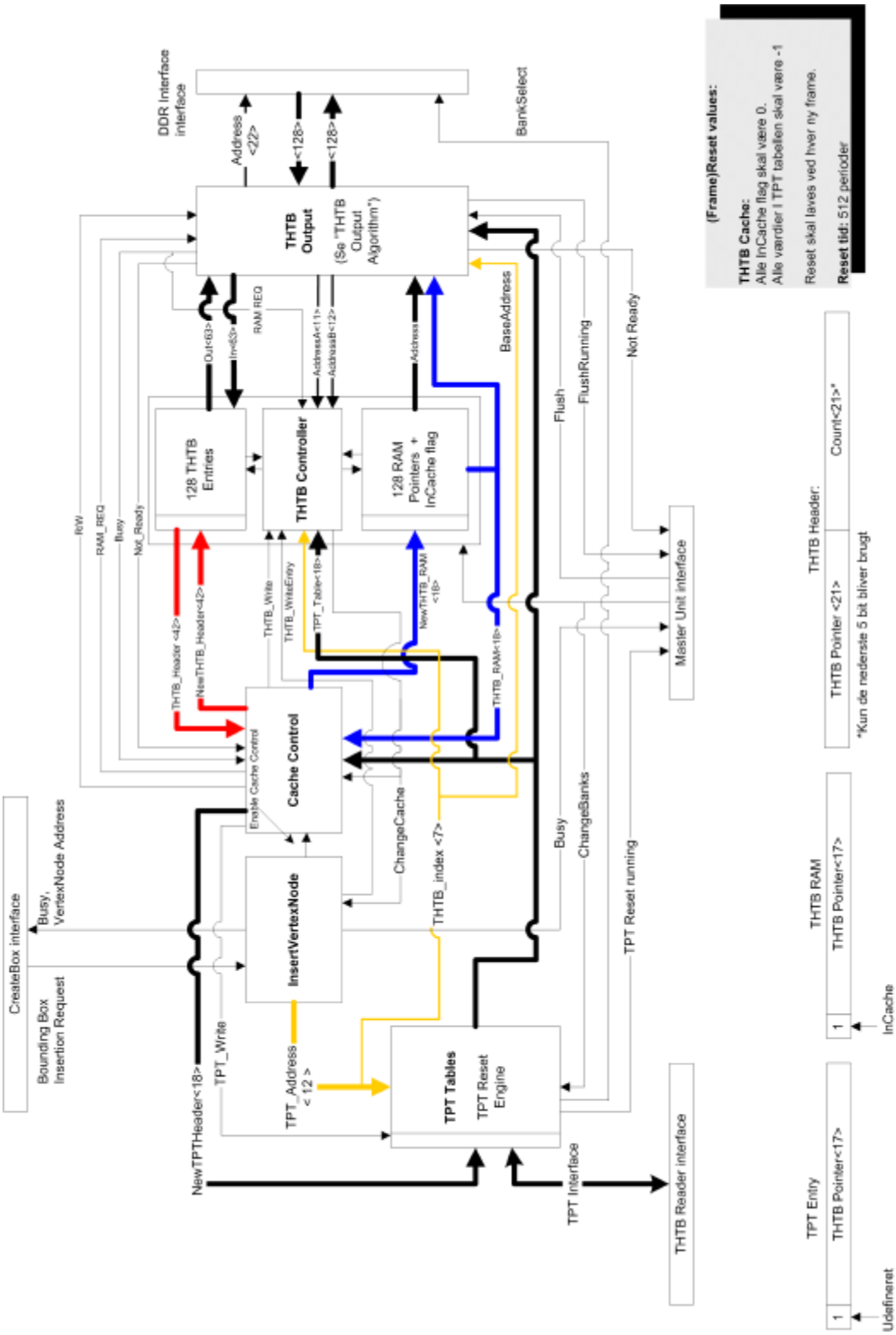
I/O Buffer diagram



9.5 Bilag 5: DDR interface algoritme



9.6 Bilag 6: THTB Insertion, diagram



9.7 Bilag 7: THTB Insertion, algoritmer

InsertVertexNode

```
StartVertexInsertion:
        Hvis THNodeInsertionReq = 1
                Busy = 1
                Xcur = Xmin, Ycur = Ymin
                TPT_Address = Xmin +32Ymin
                NextCase  = InsertVertex
```

```
InsertVertex:
        InsertRunning = 1
        Hvis ChangeCache = 0
                THTB_WriteEntry = 1
                NextCase = IncXY
                NewTHTB_Header = VertexNode Address
        ellers
                NextCase = InsertTH
                THTB_WriteEntry = 0
```

```
IncXY:
        THTB_WriteEntry = InsertRunning = 0
        Hvis sidste x og y i omsluttende rektangel:
                NextCase = StartVertexInsertion
                InsertRunning = 0
        ellers
                Øg x,y
                NextCase = InsertVertex
```

```
Stall:                                          ← Vent på output fra FIFO buffer

        NextCase  = InsertVertex
```

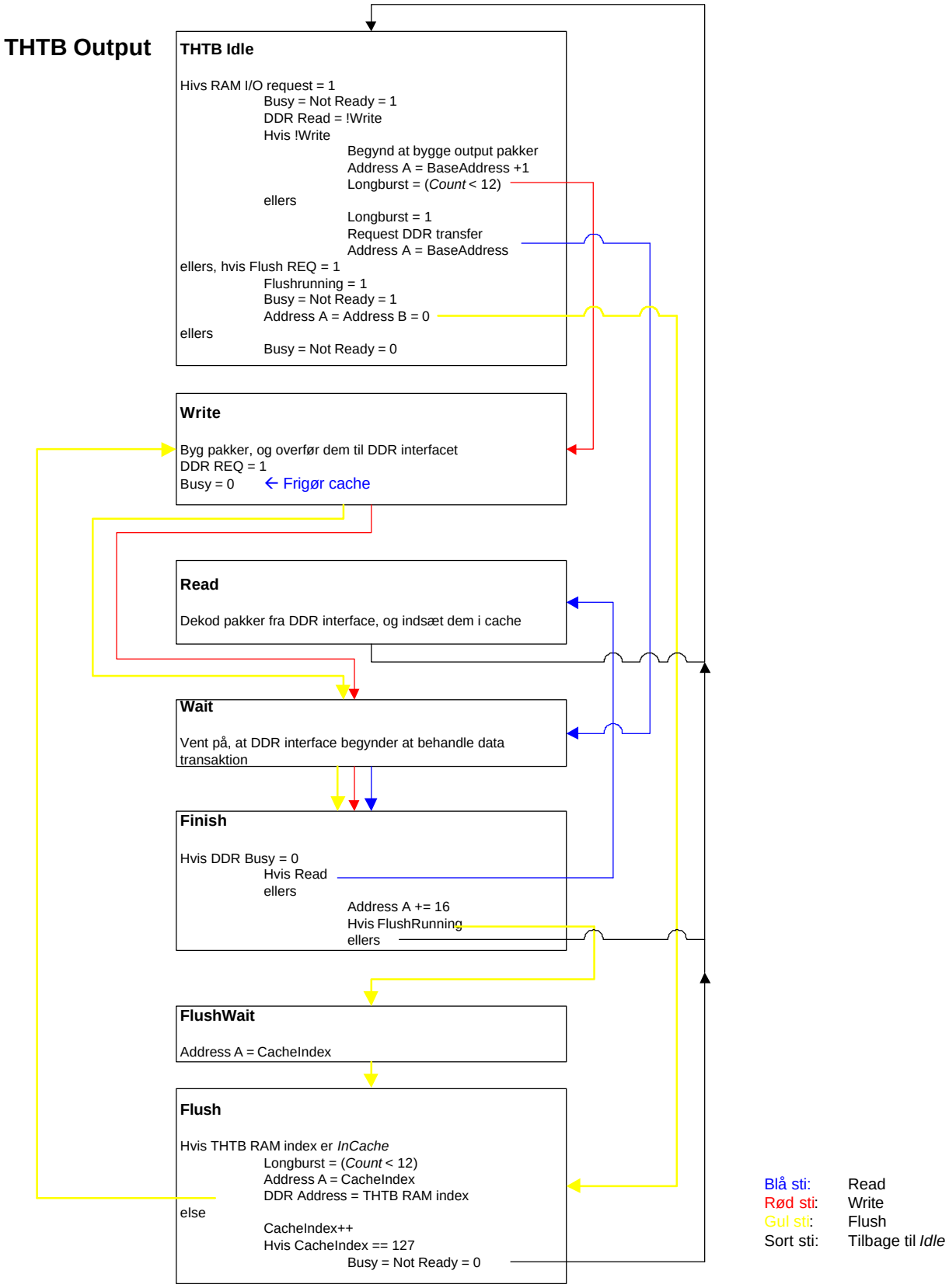
Cache Control

```
SenseInCache:
        Hvis ChangeCache = 1 & InsertRunning = 1
                NextCase = StartNew
        ellers
                NextCase = SenseInCache
```

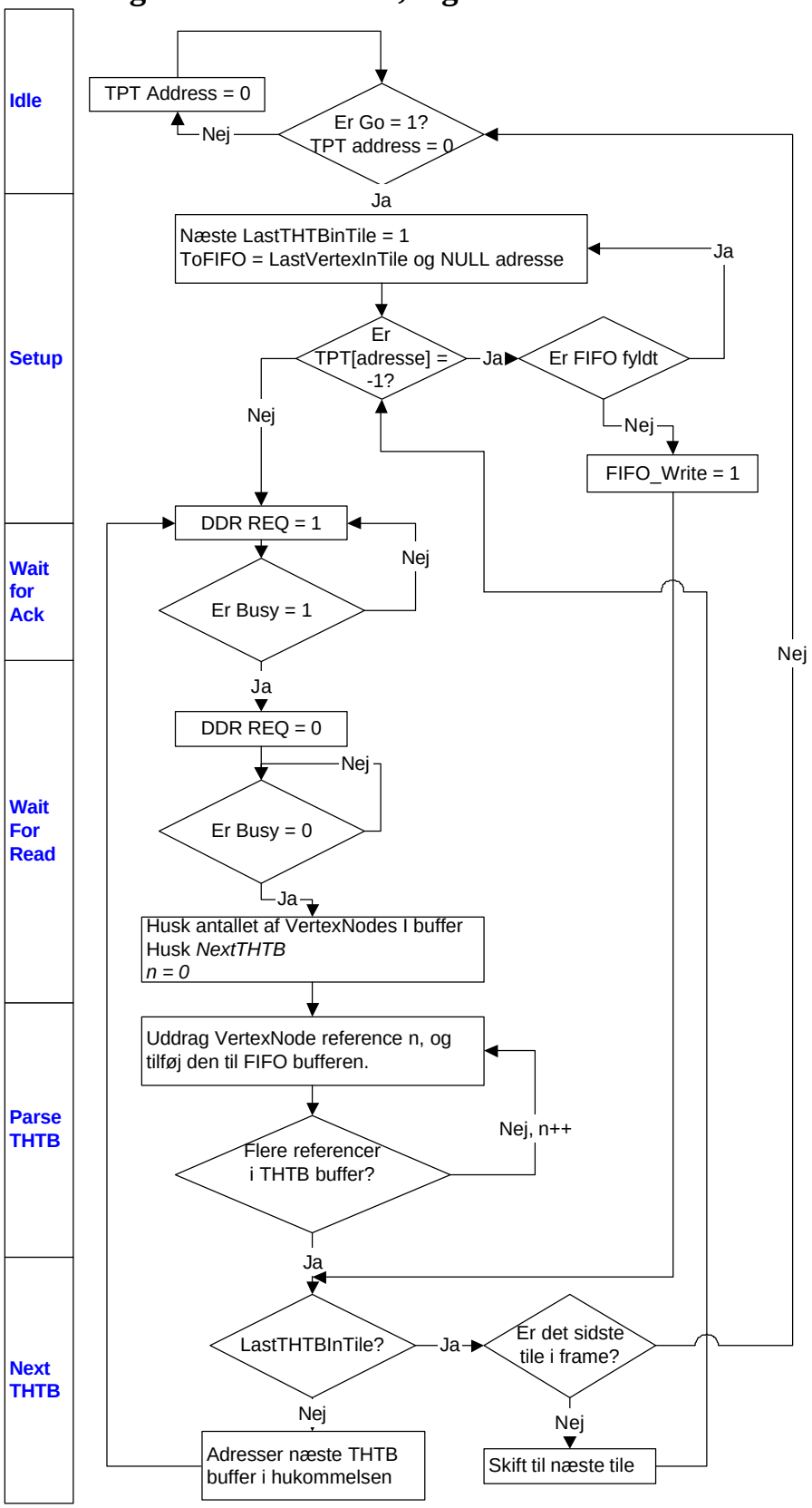
```
StartNew:
        Hvis THTB_RAM.InCache = 1
                WriteNodeToDDR = 1
                NextCase = WaitForWritten
                DontReadFromDDR = (THTB_RAM == TPT_Table.Address) ← THTB Fuld
        ellers
                NewTHTB_Header = 0
                NewTHTB_RAM = NewNextTHTB + InCache
                NewTPTHeader = NewNextTHTB
                TPTWrite = THTB_Write = 1
                NextCase = SenseInCache
```

```
WaitForWritten:
        TPTWrite = THTB_Write = 0
        Hvis busy = 0
                WriteNodeToDDR = false
                Hvis TPT_Table == NULL  ←Tile har ikke været brugt før, så drop læsning
                        NewTHTB_Header = 0
                        NewTHTB_RAM = NewNextTHTB + InCache
                        NewTPTHeader = NewNextTHTB
                        TPTWrite = THTB_Write = 0
                        NextCase = SenseInCache
                ellers
                        ReadNodeFromDDR = !DontReadFromDDR
                        RAMAddress = TPT_Table
                        NextCase = WaitForRead
```

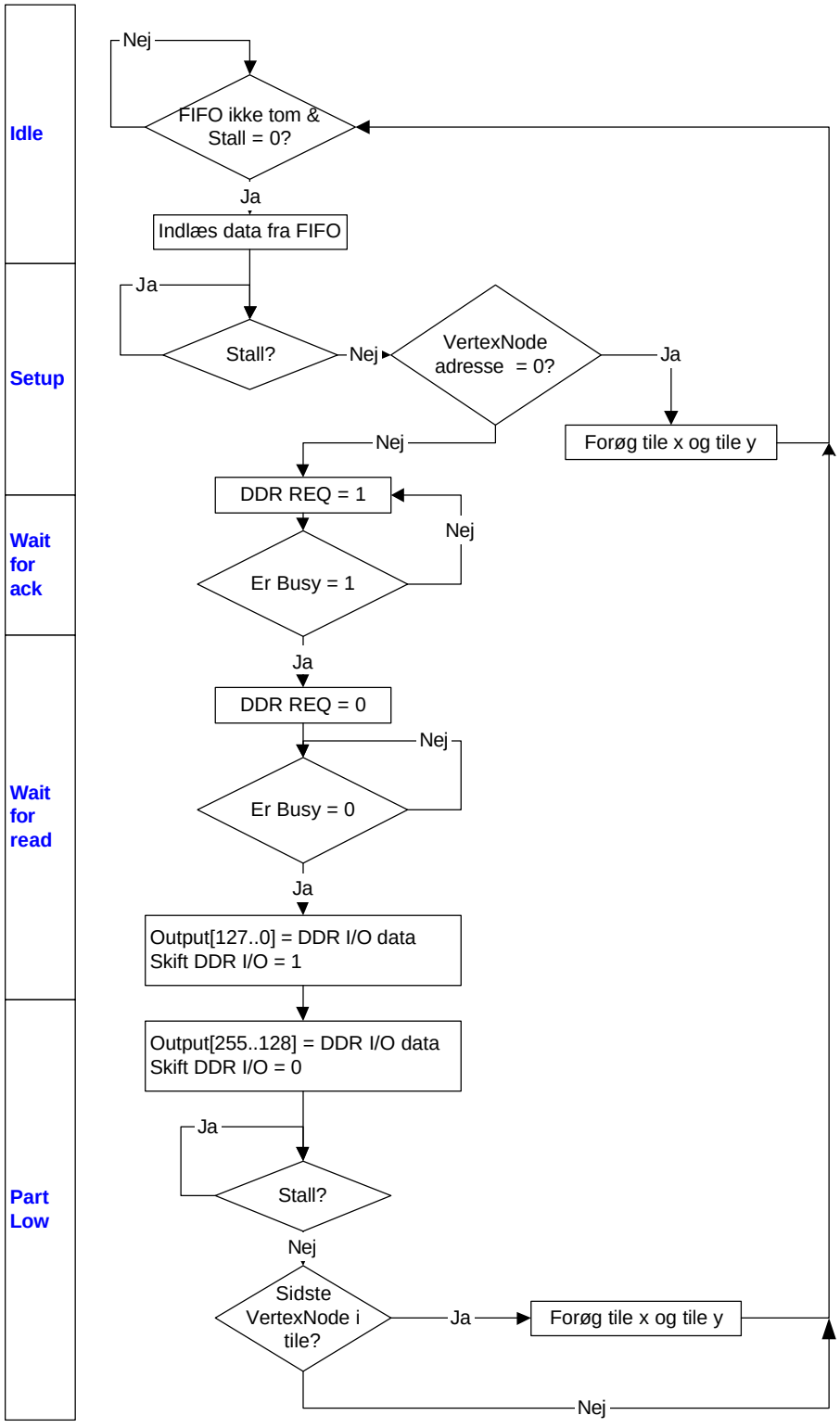
```
WaitForRead:
        Hvis not_ready = 0 eller DontReadFromDDR = 1
                ReadNodeFromDDR = 0
                Hvis THTBHeader.Count == 23← Max antal VertexNode referencer i THTB
                        NewTPTHeader = NewNextTHTB
                        NewTHTB_RAM = NewNextTHTB + TPT_Address
                        NewTHTB_Header = TPT_Table & Count = 0
                        TPTWrite = 1
                ellers
                        NewTHTB_RAM = TPT_Table + InCache
                        NewTHTB_Header = THTB_Header
                        TPTWrite = 0
                NextCase = SenseInCache
```


THTB Output
THTB Idle
Hivs RAM I/O request = 1
Busy = Not Ready = 1
DDR Read = !Write
Hvis !Write
Begynd at bygge output pakker
Address A = BaseAddress +1
Longburst = (Count < 12)
ellers
Longburst = 1
Request DDR transfer
Address A = BaseAddress
ellers, hvis Flush REQ = 1
Flushrunning = 1
Busy = Not Ready = 1
Address A = Address B = 0
ellers
Busy = Not Ready = 0
Write
Byg pakker, og overfør dem til DDR interfacet
DDR REQ = 1
Busy = 0
← Frigør cache
Read
Dekod pakker fra DDR interface, og indsæt dem i cache
Wait
Vent på, at DDR interface begynder at behandle data transaktion
Finish
Hvis DDR Busy = 0
Hvis Read
ellers
Address A += 16
Hvis FlushRunning
ellers
FlushWait
Address A = CacheIndex
Flush
Hvis THTB RAM index er InCache
Longburst = (Count < 12)
Address A = CacheIndex
DDR Address = THTB RAM index
else
CacheIndex++
Hvis CacheIndex == 127
Busy = Not Ready = 0
Blå sti: Read
Rød sti: Write
Gul sti: Flush
Sort sti: Tilbage til Idle

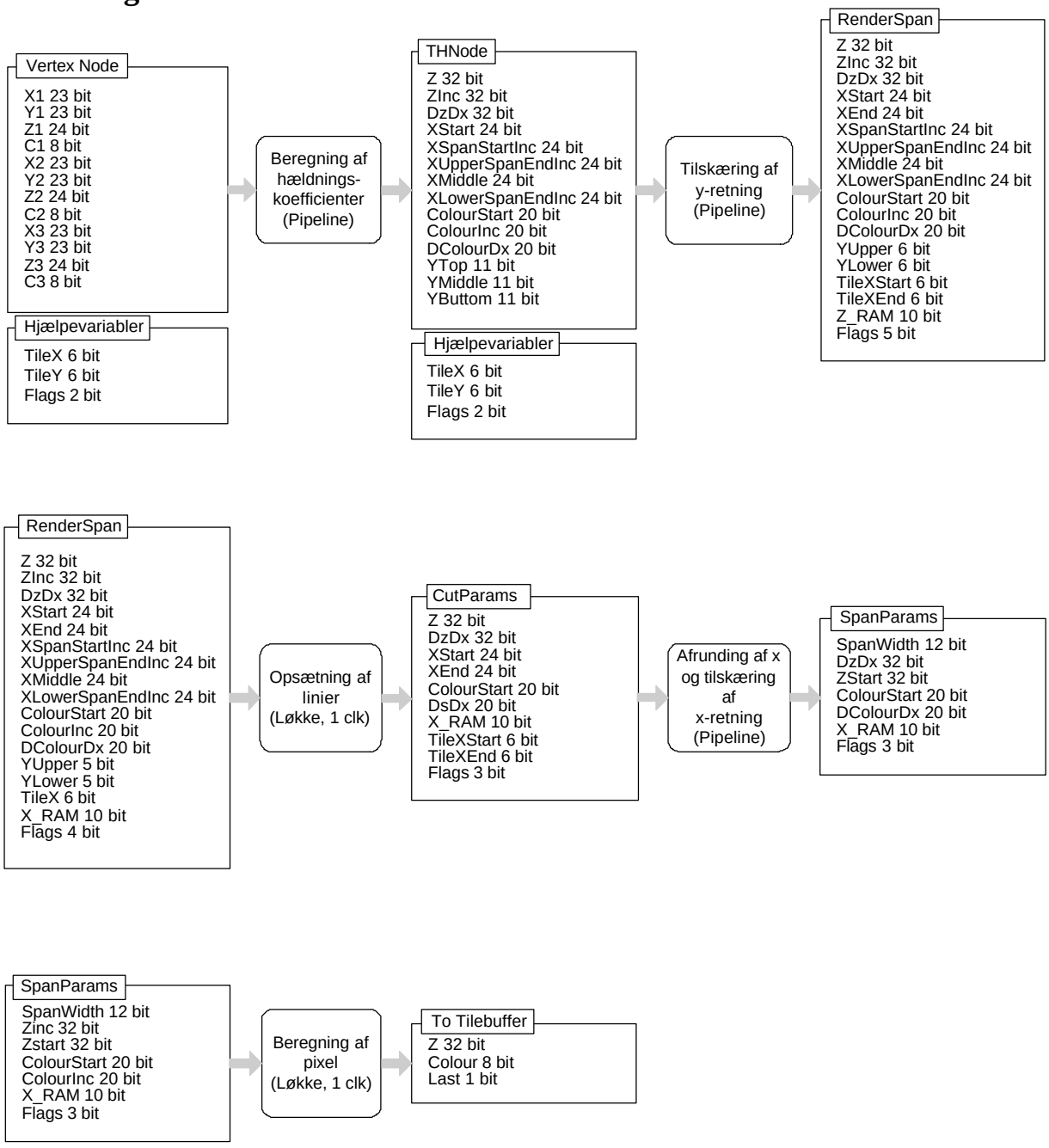
9.8 Bilag 8: THTB Reader, algoritme



9.9 Bilag 9: VertexNode Reader, algoritme



9.10 Bilag 10: Variable i BackEnd



9.11 Bilag 11: Timing analyse for Renderer

```
================================================================================
*                           Final Report                                       *
================================================================================
Final Results
RTL Output File Name               : scm_mtilerender.ngr
Top Level Output File Name         : scm_mtilerender
Output Format                      : NGC
Optimization Criterion             : Speed
Keep Hierarchy                     : NO
Macro Generator                    : macro+

Design Statistics
# IOs                              : 704

Macro Statistics :
# Registers                        : 1634
#      1-bit register              : 1611
#      10-bit register             : 8
#      129-bit register            : 4
#      171-bit register            : 2
#      32-bit register             : 8
#      323-bit register            : 1
# Shift Registers                  : 329
#      3-bit shift register        : 329
# Adders/Subtractors               : 76
#      10-bit adder                : 2
#      10-bit adder carry out      : 2
#      10-bit addsub               : 4
#      11-bit subtractor           : 3
#      12-bit addsub               : 6
#      12-bit subtractor           : 8
#      20-bit adder                : 13
#      24-bit adder                : 9
#      32-bit adder                : 13
#      32-bit subtractor           : 5
#      5-bit adder                 : 2
#      5-bit adder carry out       : 2
#      5-bit subtractor            : 7
# Multipliers                      : 9
#      20x11-bit multiplier        : 1
#      20x12-bit multiplier        : 2
#      24x11-bit multiplier        : 3
#      32x11-bit multiplier        : 1
#      32x12-bit multiplier        : 2
# Comparators                      : 17
#      11-bit comparator equal     : 1
#      11-bit comparator greater   : 1
#      11-bit comparator less      : 1
#      11-bit comparator lessequal : 2
#      11-bit comparator not equal : 1
#      12-bit comparator greatequal: 4
#      12-bit comparator less      : 2
#      24-bit comparator less      : 1
#      32-bit comparator less      : 4

Cell Usage :
# BELS                             : 8496
#      GND                         : 5
#      LUT1                        : 217
#      LUT1_D                      : 2
#      LUT1_L                      : 22
#      LUT2                        : 1280
#      LUT2_D                      : 3
#      LUT2_L                      : 112
#      LUT3                        : 744
#      LUT3_D                      : 29
#      LUT3_L                      : 99
#      LUT4                        : 2449
#      LUT4_D                      : 67
```

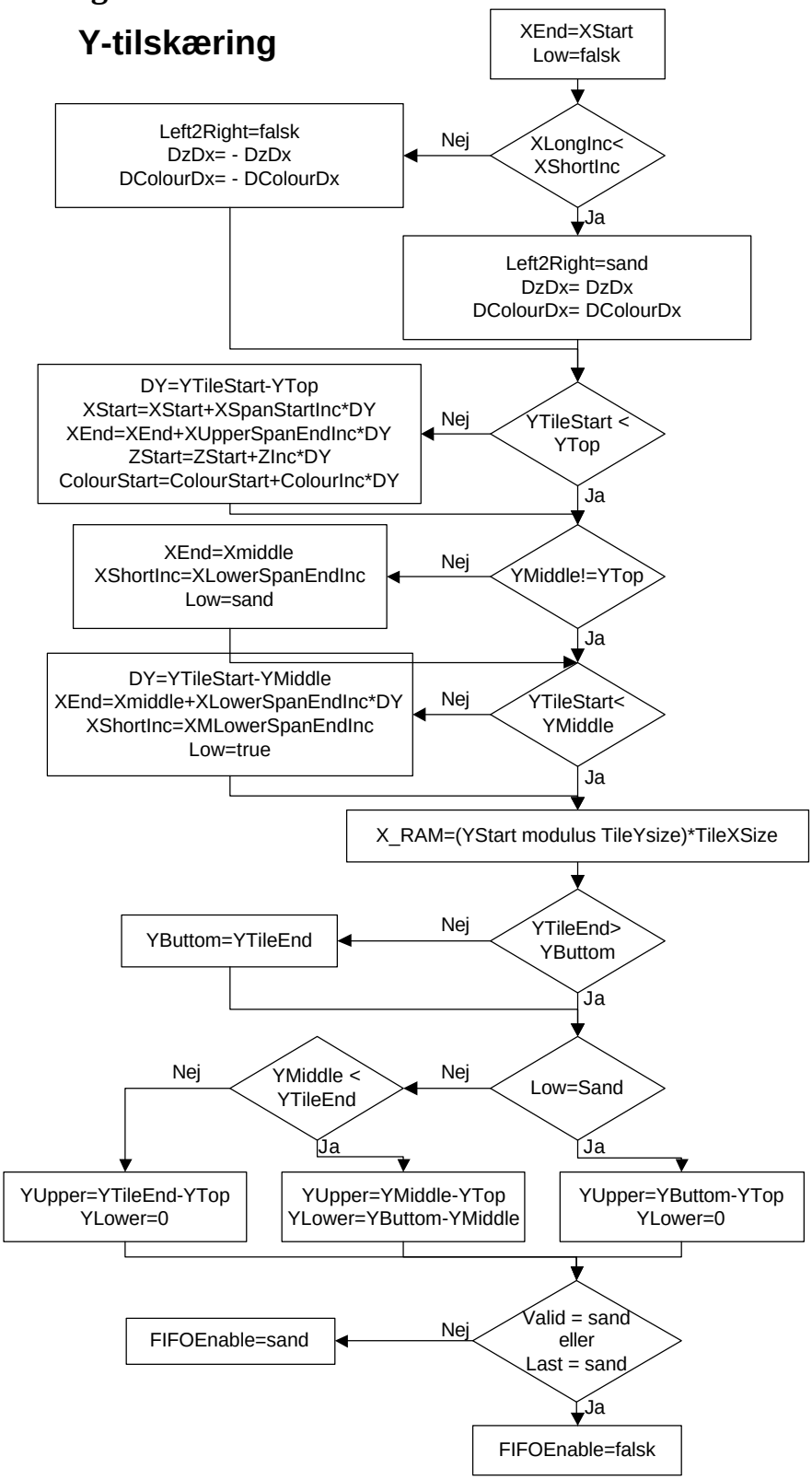
```
#      LUT4_L                      : 235
#      MUXCY                       : 1608
#      MUXF5                       : 231
#      VCC                         : 5
#      XORCY                       : 1388
# FlipFlops/Latches                : 5969
#      FDC                         : 24
#      FDCE                        : 360
#      FDE                         : 3205
#      FDP                         : 1
#      FDPE                        : 4
#      FDRE                        : 94
#      FDSE                        : 4
#      LD                          : 2277
# RAMS                             : 19
#      RAMB16_S36_S36              : 17
#      RAMB16_S4_S4                : 2
# Shifters                         : 329
#      SRL16E                      : 329
# Clock Buffers                    : 2
#      BUFGP                       : 2
# IO Buffers                       : 702
#      IBUF                        : 452
#      OBUF                        : 250
# MULTs                            : 18
#      MULT18X18                   : 18
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v4000ff1152-4

 Number of Slices:                     4203  out of  23040    18%
 Number of Slice Flip Flops:           5969  out of  46080    12%
 Number of 4 input LUTs:               5588  out of  46080    12%
 Number of bonded IOBs:                 702  out of    824    85%
 Number of BRAMs:                        19  out of    120    15%
 Number of MULT18X18s:                   18  out of    120    15%
 Number of GCLKs:                         2  out of     16    12%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
drawspanevenodd_ker899091:o        | NONE(*)(drawspanevenodd_scs_sspanstage_116_0)| 128   |
drawspanoddodd_ker899091:o         | NONE(*)(drawspanoddodd_scs_sspanstage_126_0)| 128   |
drawspanoddeven_ker899091:o        | NONE(*)(drawspanoddeven_scs_sspanstage_105_0)| 128   |
drawspaneveneven_ker899091:o       | NONE(*)(drawspaneveneven_scs_sspanstage_126_0)| 128   |
setupspanodd_setupspan__n081221:o  | NONE(*)(setupspanodd_setupspan_sco_bfiforead_sig_0)| 1
|
setupspaneven_setupspan__n08131:o  | NONE(*)(setupspaneven_setupspan_scs_blast_0)| 1     |
setupspaneven_setupspan__n081221:o | NONE(*)(setupspaneven_setupspan_sco_bfiforead_sig_0)| 1
|
drawspanoddodd__n00791:o           | NONE(*)(drawspanoddodd_scs_bvdatapathin_126_0)| 127   |
drawspanoddeven__n00791:o          | NONE(*)(drawspanoddeven_scs_bvdatapathin_3_0)| 127   |
drawspaneveneven__n00791:o         | NONE(*)(drawspaneveneven_scs_bvdatapathin_126_0)| 127   |
drawspanevenodd__n00791:o          | NONE(*)(drawspanevenodd_scs_bvdatapathin_6_0)| 127   |
scs_reset                          | BUFGP                  | 1253  |
scs_clock                          | BUFGP                  | 4040  |
setupspanodd_setupspan__n08131:o   | NONE(*)(setupspanodd_setupspan_scs_blast_0)| 1     |
-----------------------------------+------------------------+-------+
(*) These 12 clock signal(s) are generated by combinatorial logic,
and XST is not able to identify which are the primary clock signals.
```

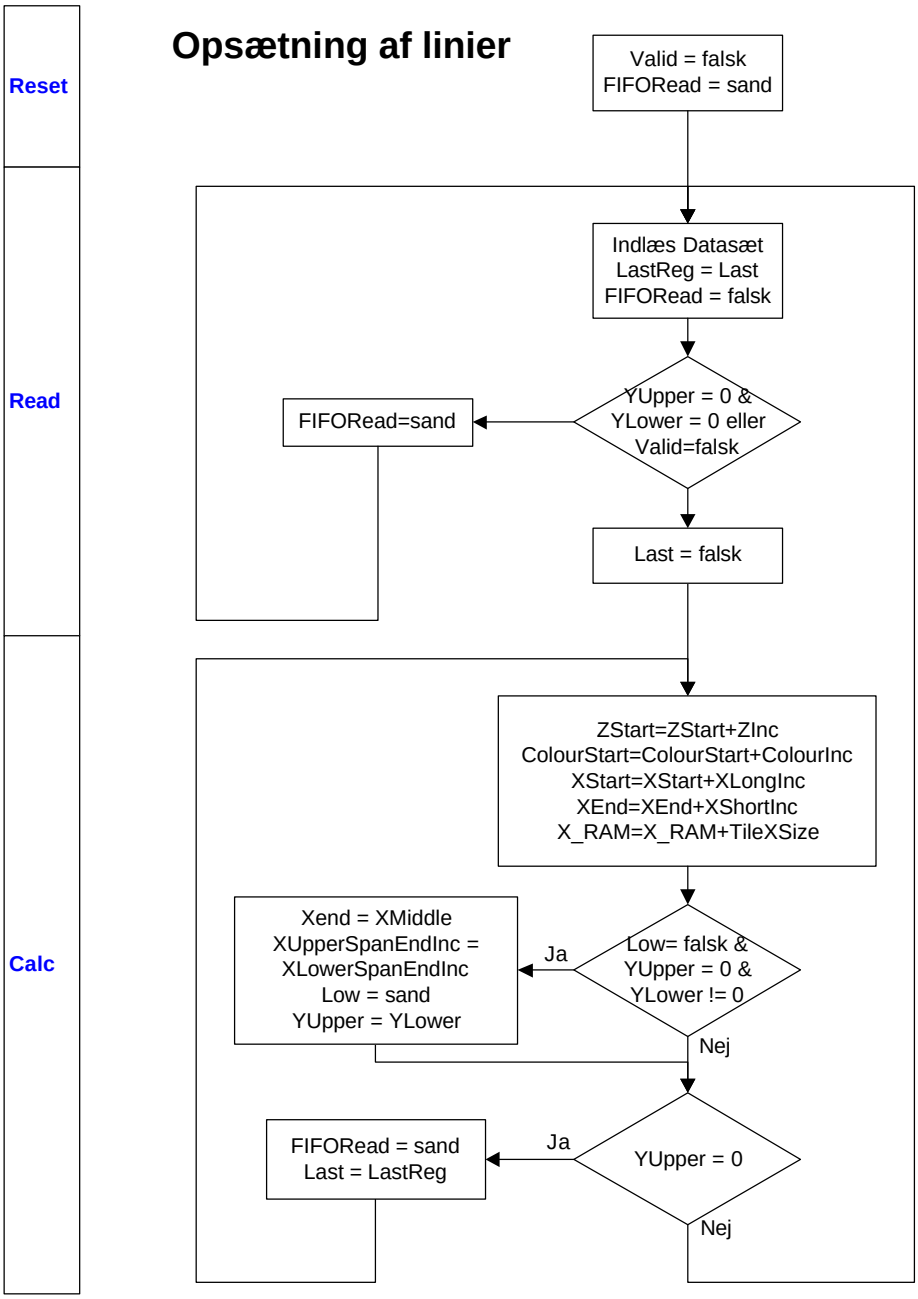
```
Please use the CLOCK_SIGNAL constraint to specify the clock signal(s) generated by
combinatorial logic.

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 12.307ns (Maximum Frequency: 81.255MHz)
   Minimum input arrival time before clock: 11.811ns
   Maximum output required time after clock: 10.399ns
   Maximum combinational path delay: 10.277ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'scs_clock'
Delay:               12.307ns (Levels of Logic = 18)
  Source:            setupspanodd_cutspan_mmux__n0025_i7_result1_frb
  Destination:       setupspanodd_cutspan_scs_sspanparamsstage2_1_brb1
  Source Clock:      scs_clock rising
  Destination Clock: scs_clock rising

  Data Path: setupspanodd_cutspan_mmux__n0025_i7_result1_frb to
setupspanodd_cutspan_scs_sspanparamsstage2_1_brb1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDE:c->q             12   0.568   1.470  setupspanodd_cutspan_mmux__n0025_i7_result1_frb
(setupspanodd_cutspan_mmux__n0025_i7_result1_frb)
     LUT4_D:I3->O          1   0.439   0.000  setupspanodd_cutspan_maddsub__n0024_inst_lut3_01
(setupspanodd_cutspan_maddsub__n0024_inst_lut3_0)
     MUXCY:s->o            1   0.298   0.000  setupspanodd_cutspan_maddsub__n0024_inst_cy_494
(setupspanodd_cutspan_maddsub__n0024_inst_cy_494)
     XORCY:ci->o           6   1.274   1.170  setupspanodd_cutspan_maddsub__n0024_inst_sum_370
(setupspanodd_cutspan__n0957<1>)
     LUT2_L:I1->LO         1   0.439   0.000
setupspanodd_cutspan_mcompar__n0094_inst_lut2_3941
(setupspanodd_cutspan_mcompar__n0094_inst_lut2_394)
     MUXCY:s->o            1   0.298   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_407
(setupspanodd_cutspan_mcompar__n0094_inst_cy_407)
     MUXCY:ci->o           1   0.053   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_408
(setupspanodd_cutspan_mcompar__n0094_inst_cy_408)
     MUXCY:ci->o           1   0.053   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_409
(setupspanodd_cutspan_mcompar__n0094_inst_cy_409)
     MUXCY:ci->o           1   0.053   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_410
(setupspanodd_cutspan_mcompar__n0094_inst_cy_410)
     MUXCY:ci->o           1   0.053   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_411
(setupspanodd_cutspan_mcompar__n0094_inst_cy_411)
     MUXCY:ci->o           1   0.053   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_412
(setupspanodd_cutspan_mcompar__n0094_inst_cy_412)
     MUXCY:ci->o           1   0.053   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_413
(setupspanodd_cutspan_mcompar__n0094_inst_cy_413)
     MUXCY:ci->o           1   0.053   0.000  setupspanodd_cutspan_mcompar__n0094_inst_cy_414
(setupspanodd_cutspan_mcompar__n0094_inst_cy_414)
     MUXCY:ci->o           2   0.053   0.790  setupspanodd_cutspan_mcompar__n0094_inst_cy_415
(setupspanodd_cutspan_mcompar__n0094_inst_cy_415)
     LUT4_D:I3->O         12   0.439   1.470  lut_459177065 (setupspanodd_cutspan__n0973<1>)
     LUT4:i1->o            1   0.439   0.408  setupspanodd_cutspan_mmux__n0072_i11_result1
(setupspanodd_cutspan__n0072<0>)
     LUT2_L:I1->LO         1   0.439   0.000  setupspanodd_cutspan_msub__n0035_inst_lut2_4171
(setupspanodd_cutspan_msub__n0035_inst_lut2_417)
     MUXCY:s->o            1   0.298   0.000  setupspanodd_cutspan_msub__n0035_inst_cy_430
(setupspanodd_cutspan_msub__n0035_inst_cy_430)
     XORCY:ci->o           1   1.274   0.000  setupspanodd_cutspan_msub__n0035_inst_sum_306
(setupspanodd_cutspan__n0035<1>)
     FDCE:d                    0.370          setupspanodd_cutspan_scs_sspanparamsstage2_1_brb1
    ----------------------------------------
    Total                     12.307ns (6.999ns logic, 5.308ns route)
                                       (56.9% logic, 43.1% route)

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

```
Timing constraint: Default OFFSET IN BEFORE for Clock 'scs_clock'
Offset:              11.811ns (Levels of Logic = 8)
  Source:            scs_reset
  Destination:       drawspanfifoiodd/bu164
  Destination Clock: scs_clock rising

  Data Path: scs_reset to drawspanfifoiodd/bu164
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     BUFGP:i->o         2292   0.994   2.329  scs_reset_bufgp (scs_reset_bufgp)
     LUT3:i0->o            8   0.439   1.270  ker842771 (n84279)
     LUT4:i1->o            1   0.439   0.000  drawspanevenodd_ker8987015_f (n186125)
     MUXF5:i0->o           1   0.436   0.408  drawspanevenodd_ker8987015 (choice1547)
     LUT4:i2->o            2   0.439   0.790  drawspanevenodd_ker8987034
(drawspanevenodd_n89872)
     LUT3_D:I1->O          3   0.439   0.981  predrawodd__n00311 (scs_fifo3readodd)
     begin scope: 'drawspanfifoiodd'
     LUT4:i0->o            1   0.439   0.408  bu13 (n2)
     LUT4:i0->o            9   0.439   1.321  bu37 (n53)
     FDRE:ce                   0.240          bu164
    ----------------------------------------
    Total                     11.811ns (4.304ns logic, 7.507ns route)
                                       (36.4% logic, 63.6% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'scs_reset'
Offset:              7.189ns (Levels of Logic = 1)
  Source:            setuptri_sco_bnextstalled_0
  Destination:       scs_bnextstalled
  Source Clock:      scs_reset falling

  Data Path: setuptri_sco_bnextstalled_0 to scs_bnextstalled
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               1   0.674   0.408  setuptri_sco_bnextstalled_0
(setuptri_sco_bnextstalled_0)
     OBUF:i->o                 6.107          scs_bnextstalled_obuf (scs_bnextstalled)
    ----------------------------------------
    Total                     7.189ns (6.781ns logic, 0.408ns route)
                                       (94.3% logic, 5.7% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'scs_clock'
Offset:              10.399ns (Levels of Logic = 3)
  Source:            drawspanoddodd_scs_estate_ffd4
  Destination:       sco_btilechange
  Source Clock:      scs_clock rising

  Data Path: drawspanoddodd_scs_estate_ffd4 to sco_btilechange
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:c->q              7   0.568   1.219  drawspanoddodd_scs_estate_ffd4
(drawspanoddodd_scs_estate_ffd4)
     LUT4:i0->o            7   0.439   1.219  _n00169 (choice1509)
     LUT3:i1->o            1   0.439   0.408  _n00191 (sco_btilechange_obuf)
     OBUF:i->o                 6.107          sco_btilechange_obuf (sco_btilechange)
    ----------------------------------------
    Total                     10.399ns (7.553ns logic, 2.846ns route)
                                       (72.6% logic, 27.4% route)

--------------------------------------------------------------------------------
Timing constraint: Default path analysis
Delay:               10.277ns (Levels of Logic = 3)
  Source:            scs_reset
  Destination:       sco_btilechange

  Data Path: scs_reset to sco_btilechange
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
```

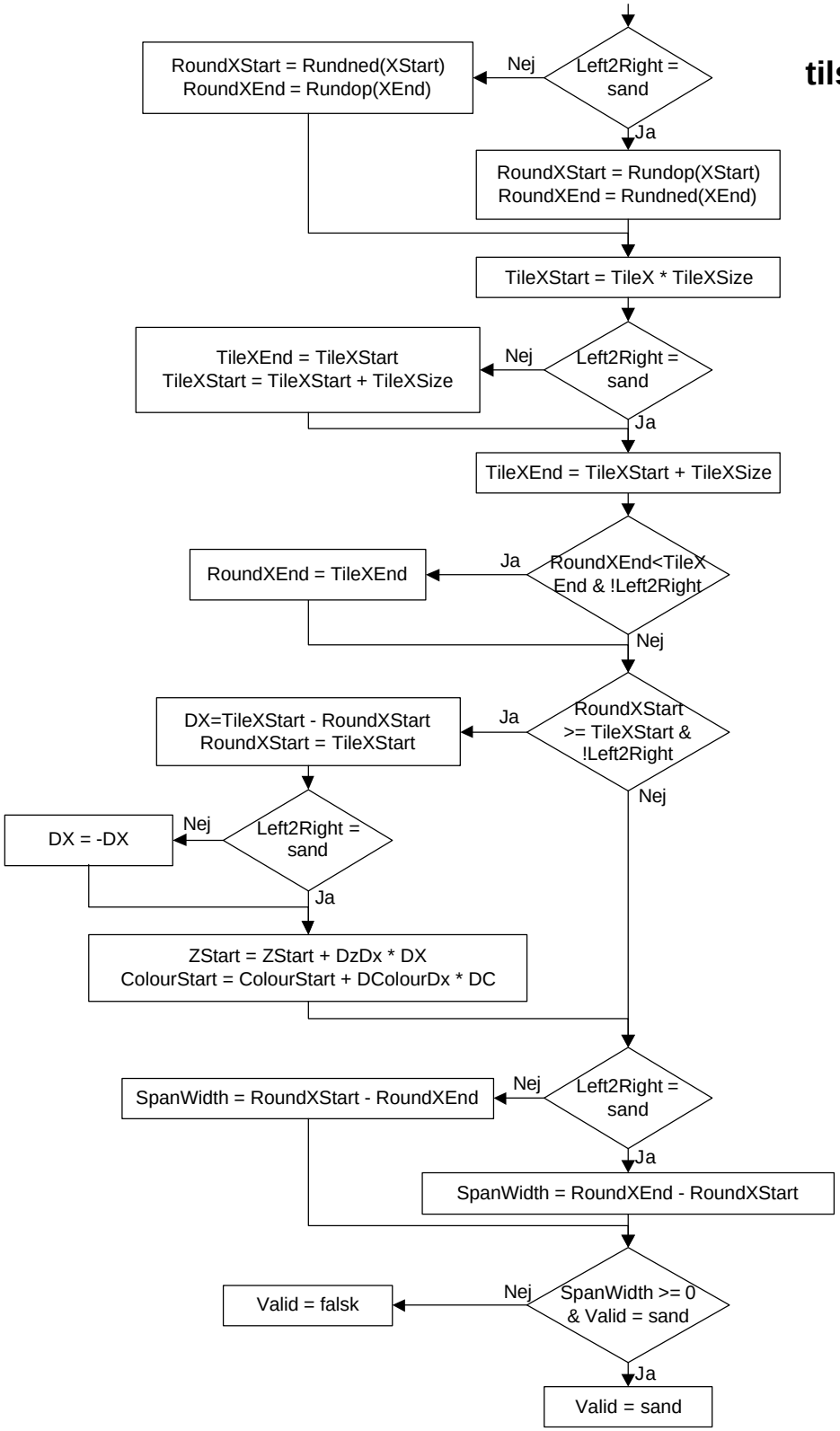
```
     BUFGP:i->o         2292   0.994   2.329  scs_reset_bufgp (scs_reset_bufgp)
     LUT3:i2->o            1   0.439   0.408  _n00191 (sco_btilechange_obuf)
     OBUF:i->o                 6.107          sco_btilechange_obuf (sco_btilechange)
    ----------------------------------------
    Total                     10.277ns (7.540ns logic, 2.737ns route)
                                       (73.4% logic, 26.6% route)

=========================================================================
CPU : 341.61 / 342.13 s | Elapsed : 341.00 / 342.00 s

-->

Total memory usage is 322296 kilobytes
```

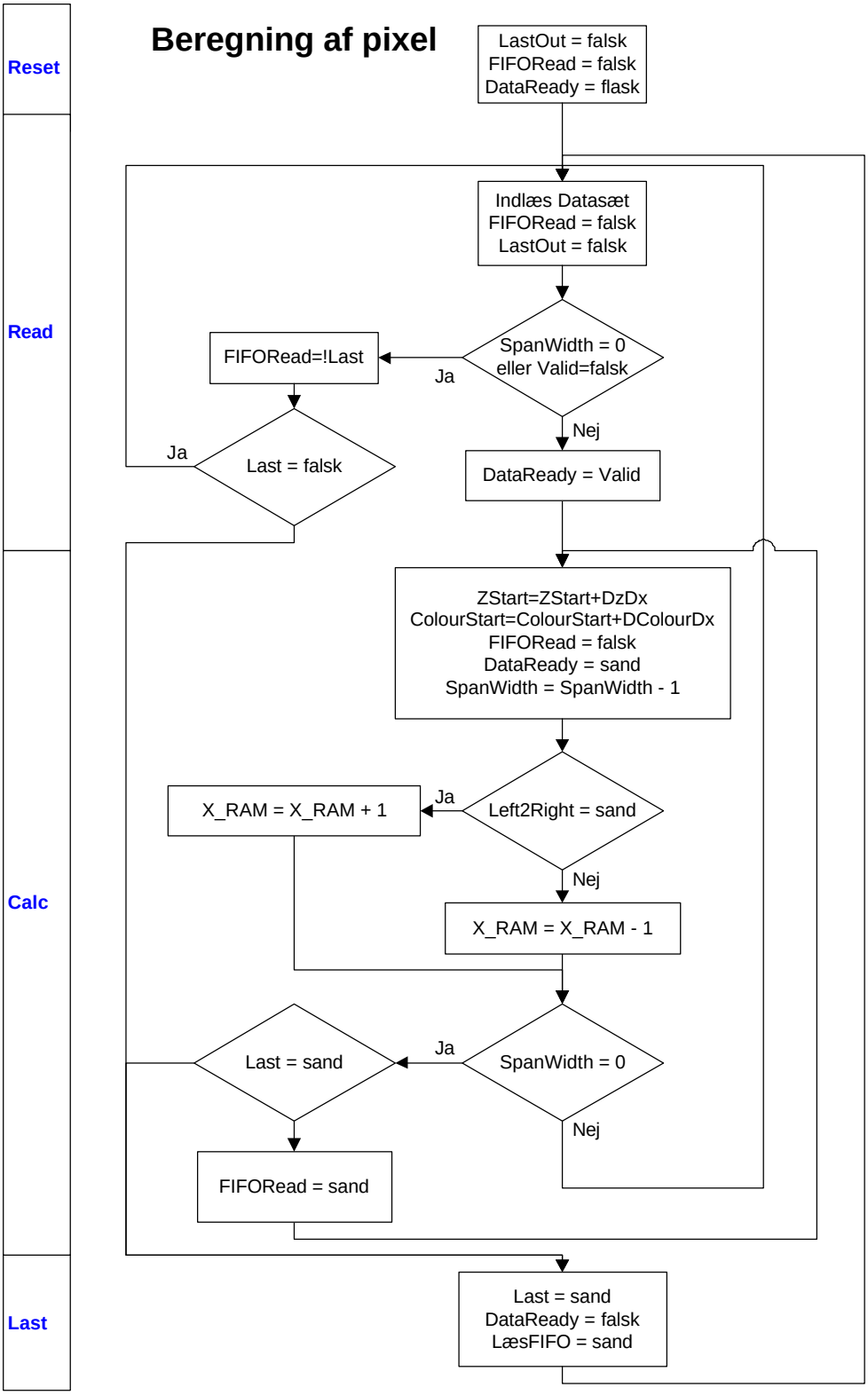
9.12 Bilag 12: Algoritmer for enheder i Renderer




Reset
Read
Calc
Opsætning af linier
Valid = falsk
FIFORead = sand
Indlæs Datasæt
LastReg = Last
FIFORead = falsk
YUpper = 0 &
YLower = 0 eller
Valid=falsk
FIFORead=sand
Last = falsk
ZStart=ZStart+ZInc
ColourStart=ColourStart+ColourInc
XStart=XStart+XLongInc
XEnd=XEnd+XShortInc
X_RAM=X_RAM+TileXSize
Low= falsk &
YUpper = 0 &
YLower != 0
Ja
Xend = XMiddle
XUpperSpanEndInc =
XLowerSpanEndInc
Low = sand
YUpper = YLower
Nej
YUpper = 0
Ja
FIFORead = sand
Last = LastReg
Nej

Afrunding og tilskæring af linier



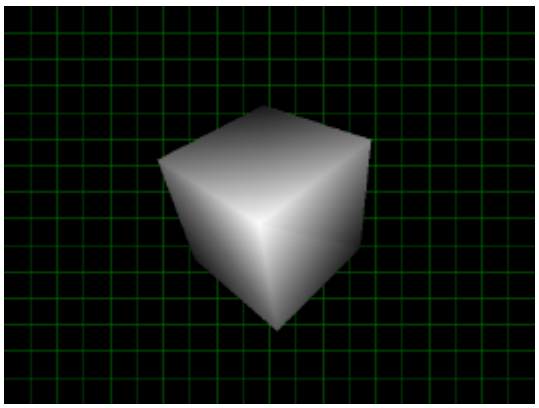

Beregning af pixel
Reset
Read
Calc
Last
LastOut = falsk
FIFORead = falsk
DataReady = flask
Indlæs Datasæt
FIFORead = falsk
LastOut = falsk
SpanWidth = 0
eller Valid=falsk
Ja
FIFORead=!Last
Ja
Last = falsk
Nej
DataReady = Valid
ZStart=ZStart+DzDx
ColourStart=ColourStart+DColourDx
FIFORead = falsk
DataReady = sand
SpanWidth = SpanWidth - 1
Left2Right = sand
Ja
X_RAM = X_RAM + 1
Nej
X_RAM = X_RAM - 1
SpanWidth = 0
Ja
Last = sand
Nej
FIFORead = sand
Last = sand
DataReady = falsk
LæsFIFO = sand

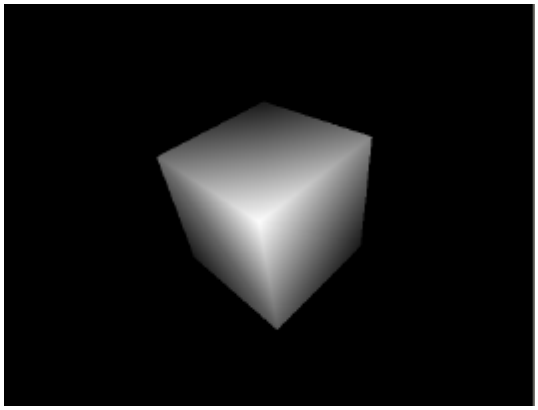
9.13 Bilag 13: C++ afhængighedstræ

9.14 Bilag 14: Reference og simulerings billeder af figurer fra Hybris software

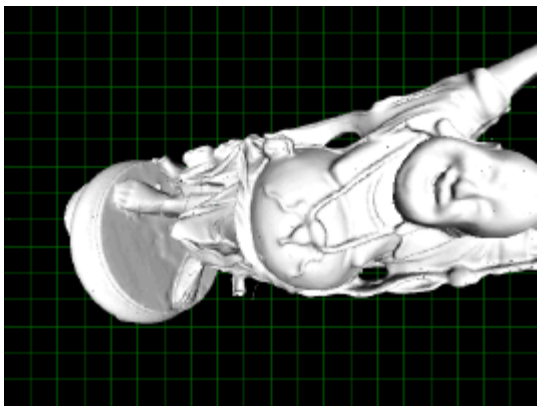
I venstre kolonne ses de simulerede objekterne, og til højre de tilsvarende reference modeller, genereret med *Hybris Demo*. Alle billeder kan findes på cd'en.

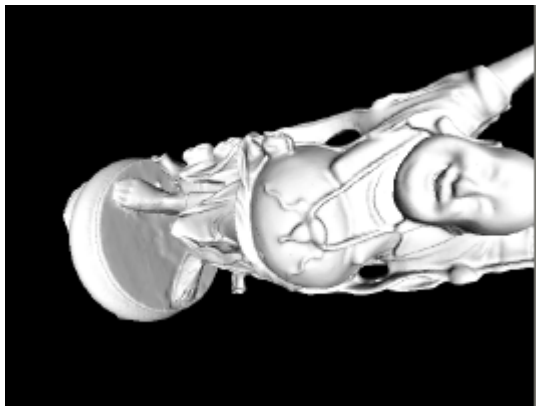
Boks: 640x480.



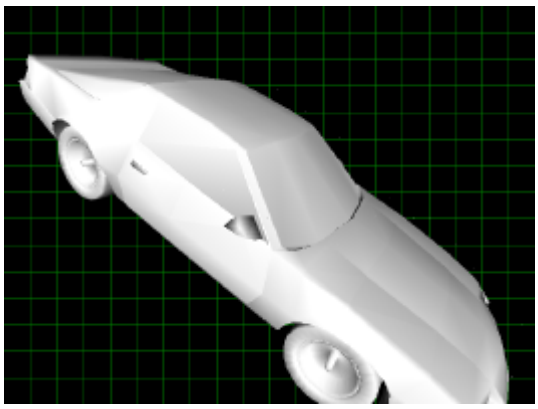
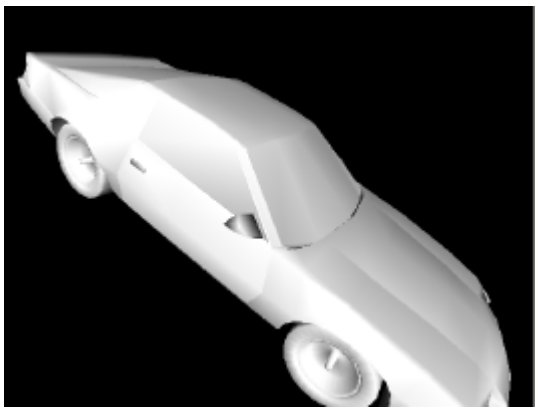


Buddha: 640x480.

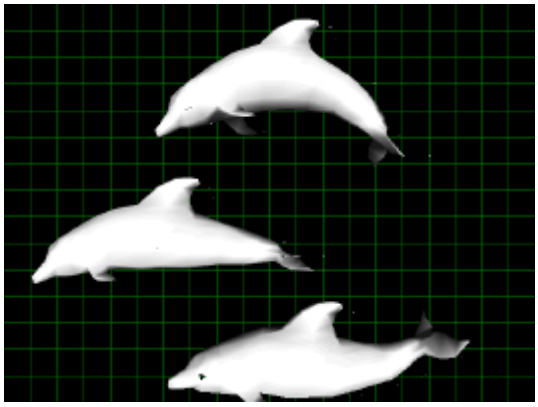




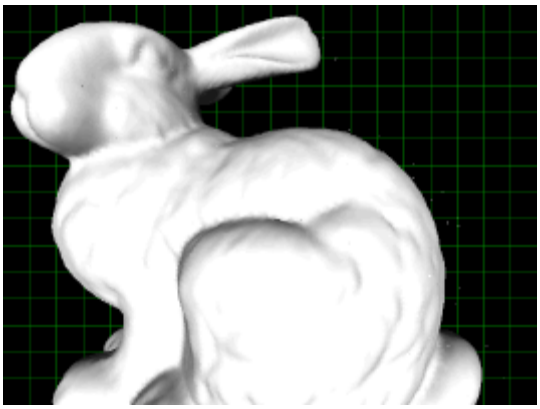
Camero: 640x480

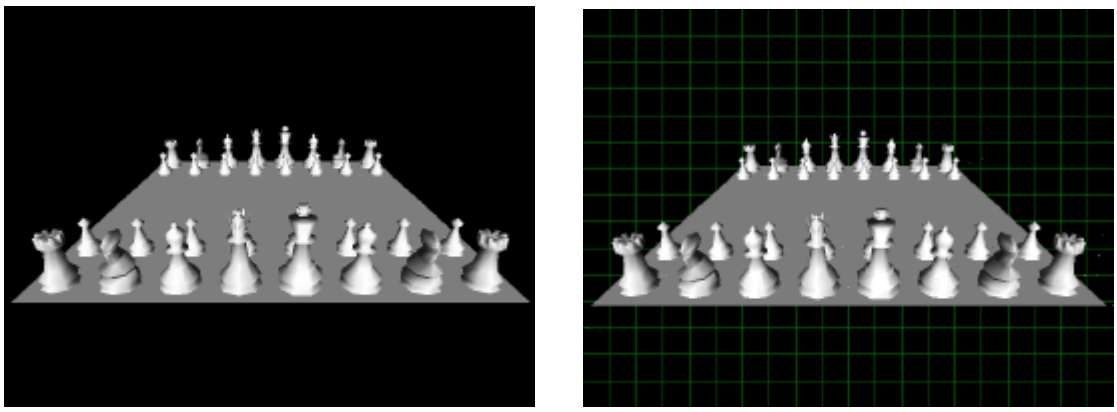
Delfiner: 640x480. Den manglende trekant, i nederste delfin, skyldes at det udviklede system klipper lidt for mange trekanter i forhold til software udgaven.



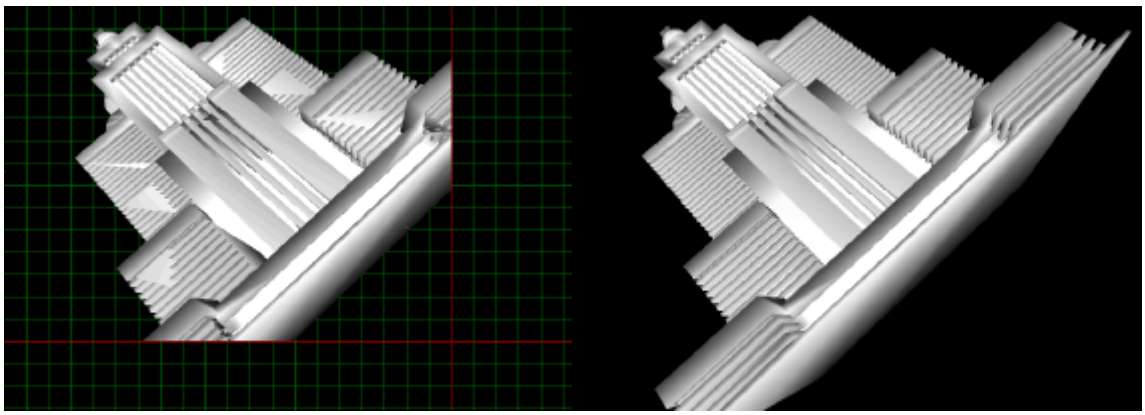

Kanin: 640x480




Skatbræt: 640x480



Skyskraber: Referencebillede i 800x584, men klippet til 640x480 af udviklet Hybris system, til at illustrere at klipning i *SortUnit* virker.



9.15 Bilag 15: Måleresultater

Sort Unit statistik, Simulering 1 frame

	Boks	Buddha	Camero	Delfiner	Kanin	Skakbræt	Sky-skraber	Buddha
	640x480	640x480	640x480	640x480	640x480	640x480	640x480	Fullscreen
Perioder	2950	3164195	33727	13749	330983	48903	70161	5894855
Trekanter ind	6	503146	1757	826	24912	3669	1780	527032
Trekanter ud	6	232406	1728	819	23893	3350	1770	431775
Trekanter, slettet	0	270740	29	7	1019	319	10	95257
Trekanter, forventet slettet	0	270661	26	1	633	319	1	94168
Forskel	0	-79	-3	-6	-386	0	-9	-1089
Perioder per trekant/ind	491,6666667	6,28882074	19,19578828	16,64527845	13,28608703	13,32869992	39,41629213	11,18500395
Perioder per vertes, efter CreateBox	491,6666667	13,6149454	19,51793981	16,78754579	13,85271837	14,59791045	39,63898305	13,65260842
Cache index's benyttet	72	117	124	91	128	104	128	128
VertexNode referencer indsat	206	236808	4121	1665	29461	4810	11035	465180
Cache-hit	116	226446	3714	1529	28038	4519	9784	444120
Cache-hit, %	56,31067961	95,62430323	90,12375637	91,83183183	95,16988561	93,95010395	88,66334391	95,47272024
Fyldte THTB buffere	0	10228	96	17	1176	157	364	19852
Write count	18	10245	283	45	1295	187	1123	20932
Read count, inkl THTB full	0	10228	242	17	1213	173	1008	20259
Read count, excl THTB full	0	0	146	0	37	16	644	407
DDR transaktioner, port 5	90	10362	553	136	1460	307	1895	21467
Port 4 busy	54	2101894	15782	7400	216297	30326	16414	3906528
Port 5 busy	810	124142	5754	1437	17058	3389	19780	254857
Port 4 busy, % total perioder	1,830508475	66,42744837	46,7933703	53,82209615	65,34988202	62,01255547	23,39476347	66,27012878
Port 5 busy, % total perioder	27,45762712	3,923335951	17,06051531	10,45166921	5,153739014	6,930045192	28,19230057	4,323380304
Port 4, gns. busy tid	9	9,044060825	9,133101852	9,035409035	9,052735111	9,052537313	9,273446328	9,047601181
Port 5, gns. busy tid	9	11,98050569	10,40506329	10,56617647	11,68356164	11,03908795	10,43799472	11,87203615
Port 5, % i forhold til port 4	1500	5,906196982	36,45925738	19,41891892	7,886378452	11,17522918	120,5068844	6,523874909
Hybris saturated, perioder	0	975593	12520	4270	127339	17682	30764	2211240
Port 5 / Port 4 skrivninger, %	1500	4,458576801	32,00231481	16,60561661	6,110576319	9,164179104	107,0621469	4,971802443
Hybris saturated, % af total tid	0	30,83226539	37,12159398	31,05680413	38,47297293	36,15729096	43,84772167	37,51135524
Gns tiles pr trekant	34,33333333	1,018940991	2,384837963	2,032967033	1,233038965	1,435820896	6,234463277	1,077366684

DDR-SDRAM / Controller statistik

Simulering: 10 Frames, 640x480

	Delfiner	Kanin	Skyskraber
	640x480	640x480	640x480
Perioder	1416983	5923221	5262803
DDR Busy, perioder	209347	4326736	1122761
Port 2 busy, perioder	12244	124193	54305
Port 3 busy, perioder	106279	1879201	707396
Port 4 busy, perioder	76382	2152897	162233
Port 5 busy, perioder	14442	170445	198827
DDR Interface, udnyttelse, %	14,77413632	73,04701277	21,33389754
Port 2 busy, % af samlet busy	5,848662747	2,870362324	4,836737293
Port 3 busy, % af samlet busy	50,76690853	43,43230093	63,00503847
Port 4 busy, % af samlet busy	36,48583452	49,75799309	14,44946876
Port 5 busy, % af samlet busy	6,8985942	3,939343653	17,70875547
Cache index's benyttet	91	128	128
Transkaktioner port 2	1228	13320	5471
Transkaktioner port 3	15011	265193	99344
Transkaktioner port 4	8469	238984	17769
Transkaktioner port 5	1365	13759	17820
Port 2/3 load	0,081806675	0,05022757	0,055071268
Port 4/5 load	0,161176054	0,057572892	1,002870167
Gns tid for læsning af trekant	94,39630937	22,33551036	52,97554961
Gns busy per læsning	7,080074612	7,086163662	7,120671606
Gns stall per periode	87,31623476	15,2493467	45,854878
DDR interface, stall % af samlet	92,49962773	68,27400159	86,55856964
BackEnd, belastning, perioder	56,61557128	46,30266325	67,84177577
BackEnd, belastning, antal	62,28282131	52,42538437	74,65243155

BackEnd statistik

Simulering: 2 frames, 640 x 480

A = Lige linie, lige pixel, B = Lige linie, ulige pixel, C = Ulige linie, lige pixel, D = Ulige linie, ulige pixel

	Skakbræt	Boks	Buddha	Kanin	Camero	Delfiner	Skyskraper
Objects In Scene	6576	12	1087716	69451	3640	1689	3692
Objects After Cull	3669	6	503146	24912	1757	826	1780
Objects To Be Deleted	319	0	270661	633	26	1	1
Frames Completed	2	3	2	2	2	2	2
Clock Count	247673	158129	8163451	930193	254333	168813	646051
Clock Count (half)	123836	79064	4081725	465096	127166	84406	323025
Clocks last frame	199188	155598	4999672	599628	221022	155482	576302
Triangle size is 1 tile	6578	0	458465	37804	1587	695	941
Triangle size is 2 tiles	2240	0	8463	9475	1127	904	891
Triangle size is 3 tiles	36	0	0	0	79	32	4
Triangle size is 4 tiles	289	0	116	556	454	357	483
Triangle size is 5 tiles	0	0	0	0	0	0	38
Triangle size is 6 tiles	72	0	0	0	145	13	204
Triangle size is 7+ tiles	66	18	0	0	141	0	1048
Tiles processed	391	446	340	303	342	332	311
Triangles processed	4915	282	237023	29529	4175	1745	11079
WE1	23902	11919	34852	45767	35856	15461	74295
WE2	23925	11903	34871	45777	35870	15464	74327
WE3	24191	11295	34622	45920	35409	15494	74408
WE4	24193	11317	34902	45895	35447	15486	74306
LastTriangle 1	300	300	300	300	300	300	300
LastTriangle 2	300	300	300	300	300	300	300
LastTriangle 3	300	300	300	300	300	300	300
LastTriangle 4	300	300	300	300	300	300	300
KnownZ1	19932	3429	2124003	180084	17475	4881	178022
KnownZ2	19718	3269	2113192	180332	16226	4422	178077
KnownZ3	18440	4263	2125953	179520	18514	5417	180267
KnownZ4	18634	4125	2111786	177080	17037	4688	180229
ChangeTile Count	300	300	300	300	300	300	300
StallCount	60311	75355	19616	3941	77647	64285	201212
Ingen	92567	65942	3902580	389569	81274	67048	123100
D	156	49	36219	4381	192	33	567
C	181	30	36363	4212	204	34	534
CD	327	1529	12833	5751	1245	193	1591
B	210	3	36194	4397	145	36	619
BD	605	50	1525	3330	678	175	1073
BC	428	0	609	1671	493	154	1389
BCD	1479	134	716	4598	2453	945	6364
A	213	6	36419	4355	140	64	630
AD	415	2	578	1629	504	119	1409
AC	623	31	1472	3451	615	176	1050
ACD	1497	172	682	4584	2537	934	6478
AB	454	60	12873	5716	709	269	1610
ABD	1567	155	689	4655	2435	945	6307
ABC	1512	168	685	4650	2491	929	6468
ABCD	21602	10733	1288	18147	31051	12352	163836

BackEnd statistik, fortsat

	Skakbræt	Boks	Buddha	Kanin	Camero	Delfiner	Skyskraper
Trekanter	4810	206	236808	29461	3121	1665	2792
Pixels	111037	48251	218443	188490	163121	63006	750789
Gns pixel pr. trekant	23,084615	234,22816	0,9224477	6,3979498	52,26562	37,841441	268,907235
Total	31269	13122	179145	75527	45892	17358	199925
Gns pixel pr. skrivning	3,551025	3,6771071	1,2193642	2,4956638	3,5544539	3,6297961	3,75535326
Ovenstående i procent	88,78%	91,93%	30,48%	62,39%	88,86%	90,74%	93,88%
A or B	2,80%	0,53%	47,72%	19,16%	2,17%	2,13%	1,43%
C or D	2,12%	12,25%	47,68%	18,99%	3,58%	1,50%	1,35%
Kun lige eller ulige aktiv	4,93%	12,78%	95,40%	38,15%	5,74%	3,62%	2,78%
D i %	0,50%	0,37%	20,22%	5,80%	0,42%	0,19%	0,28%
C i %	0,58%	0,23%	20,30%	5,58%	0,44%	0,20%	0,27%
CD i %	1,05%	11,65%	7,16%	7,61%	2,71%	1,11%	0,80%
B i %	0,67%	0,02%	20,20%	5,82%	0,32%	0,21%	0,31%
BD i %	1,93%	0,38%	0,85%	4,41%	1,48%	1,01%	0,54%
BC i %	1,37%	0,00%	0,34%	2,21%	1,07%	0,89%	0,69%
BCD i %	4,73%	1,02%	0,40%	6,09%	5,35%	5,44%	3,18%
A i %	0,68%	0,05%	20,33%	5,77%	0,31%	0,37%	0,32%
AD i %	1,33%	0,02%	0,32%	2,16%	1,10%	0,69%	0,70%
AC i %	1,99%	0,24%	0,82%	4,57%	1,34%	1,01%	0,53%
ACD i %	4,79%	1,31%	0,38%	6,07%	5,53%	5,38%	3,24%
AB i %	1,45%	0,46%	7,19%	7,57%	1,54%	1,55%	0,81%
ABD i %	5,01%	1,18%	0,38%	6,16%	5,31%	5,44%	3,15%
ABC i %	4,84%	1,28%	0,38%	6,16%	5,43%	5,35%	3,24%
ABCD i %	69,08%	81,79%	0,72%	24,03%	67,66%	71,16%	81,95%
Begge ulige aktive	79,65%	95,78%	8,66%	43,80%	81,25%	83,10%	89,17%
Begge lige aktive	80,38%	84,71%	8,67%	43,92%	79,94%	83,51%	89,14%
1 draw	2,43%	0,67%	81,05%	22,97%	1,48%	0,96%	1,18%
2 draw	9,12%	12,74%	16,68%	28,53%	9,25%	6,26%	4,06%
3 draw	19,36%	4,79%	1,55%	24,48%	21,61%	21,62%	12,81%
4 draw	69,08%	81,79%	0,72%	24,03%	67,66%	71,16%	81,95%
Min 3 aktive	88,45%	86,59%	2,27%	48,50%	89,27%	92,78%	94,76%
Min 2 aktive	97,57%	99,33%	18,95%	77,03%	98,52%	99,04%	98,82%
min 1 aktiv	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
Frekvens (MHz)	120	120	120	120	120	120	120
Framerate	602,44593	771,21814	24,001575	200,12408	542,93238	771,79352	208,22416
Skrivninger / clock	0,3139647	0,1686654	0,0716627	0,2519129	0,4152709	0,2232799	0,69382025

9.16 Bilag 16: Timing analyse for systemet

```
=========================================================================
*                          Final Report                                 *
=========================================================================
Final Results
RTL Output File Name               : scm_fpga.ngr
Top Level Output File Name         : scm_fpga
Output Format                      : NGC
Optimization Criterion             : Speed
Keep Hierarchy                     : NO
Macro Generator                    : macro+

Design Statistics
# IOs                              : 1015

Macro Statistics :
# Registers                        : 2916
#      1-bit register              : 2849
#      10-bit register             : 8
#      11-bit register             : 7
#      12-bit register             : 4
#      128-bit register            : 5
#      129-bit register            : 4
#      16-bit register             : 1
#      171-bit register            : 2
#      2-bit register              : 7
#      256-bit register            : 1
#      32-bit register             : 8
#      323-bit register            : 1
#      4-bit register              : 1
#      6-bit register              : 2
#      64-bit register             : 1
#      7-bit register              : 3
#      78-bit register             : 12
# Counters                         : 1
#      17-bit up counter           : 1
# Shift Registers                  : 870
#      3-bit shift register        : 331
#      31-bit shift register       : 143
#      32-bit shift register       : 12
#      4-bit shift register        : 256
#      5-bit shift register        : 128
# Multiplexers                     : 1
#      24-bit 4-to-1 multiplexer   : 1
# Adders/Subtractors               : 126
#      10-bit adder                : 2
#      10-bit adder carry out      : 2
#      10-bit addsub               : 4
#      11-bit adder                : 10
#      11-bit subtractor           : 12
#      12-bit addsub               : 6
#      12-bit subtractor           : 8
#      15-bit subtractor           : 1
#      20-bit adder                : 13
#      21-bit adder                : 2
#      24-bit adder                : 10
#      24-bit subtractor           : 7
#      32-bit adder                : 13
#      32-bit subtractor           : 12
#      4-bit adder                 : 1
#      5-bit adder                 : 4
#      5-bit adder carry out       : 2
#      5-bit subtractor            : 7
#      6-bit adder                 : 6
#      7-bit adder                 : 3
#      9-bit adder                 : 1
# Multipliers                      : 15
#      20x11-bit multiplier        : 1
#      20x12-bit multiplier        : 2
#      24x11-bit multiplier        : 5
#      32x11-bit multiplier        : 1
#      32x12-bit multiplier        : 2
```

```
#      32x32-bit multiplier        : 4
# Comparators                      : 49
#      11-bit comparator equal     : 3
#      11-bit comparator greater   : 10
#      11-bit comparator less      : 2
#      11-bit comparator lessequal : 2
#      11-bit comparator not equal : 1
#      12-bit comparator greatequal: 4
#      12-bit comparator less      : 2
#      17-bit comparator equal     : 2
#      23-bit comparator greatequal: 2
#      23-bit comparator greater   : 3
#      24-bit comparator less      : 1
#      32-bit comparator less      : 4
#      5-bit comparator equal      : 1
#      5-bit comparator less       : 1
#      6-bit comparator equal      : 6
#      6-bit comparator greater    : 4
#      6-bit comparator not equal  : 1

Cell Usage :
# BELS                             : 23797
#      BUF                         : 16
#      GND                         : 16
#      LUT1                        : 576
#      LUT1_L                      : 1
#      LUT2                        : 2184
#      LUT2_D                      : 3
#      LUT2_L                      : 47
#      LUT3                        : 2917
#      LUT3_D                      : 8
#      LUT3_L                      : 145
#      LUT4                        : 7189
#      LUT4_D                      : 12
#      LUT4_L                      : 226
#      MUXCY                       : 5279
#      MUXCY_D                     : 4
#      MUXF5                       : 230
#      VCC                         : 16
#      XORCY                       : 4928
# FlipFlops/Latches                : 20205
#      FD                          : 430
#      FD_1                        : 336
#      FDC                         : 133
#      FDC_1                       : 14
#      FDCE                        : 10305
#      FDCP                        : 39
#      FDCPE                       : 17
#      FDE                         : 5700
#      FDE_1                       : 2
#      FDP                         : 17
#      FDP_1                       : 3
#      FDPE                        : 28
#      FDPE_1                      : 3
#      FDR                         : 3
#      FDR_1                       : 2
#      FDRE                        : 94
#      FDSE                        : 4
#      LD                          : 3075
# RAMS                             : 28
#      RAMB16_S18                  : 1
#      RAMB16_S1_S2                : 1
#      RAMB16_S36_S36              : 19
#      RAMB16_S4_S4                : 2
#      RAMB16_S4_S9                : 5
# Shifters                         : 1025
#      SRL16E                      : 713
#      SRL16E_1                    : 2
#      SRLC16E                     : 310
# Clock Buffers                    : 4
#      bufg                        : 2
#      BUFGP                       : 2
# IO Buffers                       : 993
```

```
#      IBUF                        : 521
#      ibufg_sstl2_i               : 2
#      iobuf_sstl2_ii              : 64
#      OBUF                        : 381
#      obuf_sstl2_i                : 23
#      obuft_sstl2_ii              : 2
# DLLs                             : 2
#      clkdll                      : 2
# MULTs                            : 34
#      MULT18X18                   : 34
# Others                           : 3
#      scm_blockram_tpt_table      : 2
#      scm_fifo256ddr              : 1
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v4000ff1152-4

 Number of Slices:                    13137  out of  23040    57%
 Number of Slice Flip Flops:          20205  out of  46080    43%
 Number of 4 input LUTs:              14333  out of  46080    31%
 Number of bonded IOBs:                 993  out of    824   120% (*)
 Number of BRAMs:                        28  out of    120    23%
 Number of MULT18X18s:                   34  out of    120    28%
 Number of GCLKs:                         4  out of     16    25%

WARNING:Xst:1336 -  (*) More than 100% of Device resources are used


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
tilerender_setupspanodd_setupspan__n081221:o|
NONE(*)(tilerender_setupspanodd_setupspan_sco_bfiforead_sig_0)| 1     |
tilerender_setupspaneven_setupspan__n08131:o|
NONE(*)(tilerender_setupspaneven_setupspan_scs_blast_0)| 1     |
tilerender_setupspanodd_setupspan__n08131:o|
NONE(*)(tilerender_setupspanodd_setupspan_scs_blast_0)| 1     |
tilerender_setupspaneven_setupspan__n081221:o|
NONE(*)(tilerender_setupspaneven_setupspan_sco_bfiforead_sig_0)| 1     |
readerunit_thtbreader_currentcase_ffd4:q| NONE                  | 22    |
readerunit_thtbreader__n02311:o    | NONE(*)(readerunit_thtbreader_out_addr_3_0)| 17    |
thtbunit_thetpt__n00491:o          | NONE(*)(thtbunit_thetpt_out_resetinprogress_0)| 1     |
thtbunit_thethtb_output__n034538:o | NONE(*)(thtbunit_thethtb_output_ram_busy_0)| 1     |
readerunit_thtbreader__n023236:o   | NONE(*)(readerunit_thtbreader_tofifo_16_0)| 22    |
ddrinterface__n218859:o            | NONE(*)(ddrinterface_u_data_i_125_0)| 128   |
thtbunit_thcontrol__n012529:o      | NONE(*)(thtbunit_thcontrol_out_newtptheader_15_0)| 17    |
thtbunit_thethtb_output__n0347:o   | NONE(*)(thtbunit_thethtb_output_out_addr_18_0)| 18    |
ddrinterface__n218759:o            | NONE(*)(ddrinterface_addr_20_0)| 23    |
thtbunit_thethtb_output__n034644:o | NONE(*)(thtbunit_thethtb_output_ram_notready_0)| 1     |
thtbunit_thcontrol__n0126:o        | NONE(*)(thtbunit_thcontrol_out_newthtb_ram_15_0)| 18    |
ddrinterface__n2186:o              | NONE(*)(ddrinterface_nextcountdown_0_0)| 2     |
createth__n02231_2:o               | NONE(*)(createth_div1/bu70)| 1733  |
createth__n02231_3:o               | NONE(*)(createth_div7/bu235)| 1283  |
createth__n02231_4:o               | NONE(*)(createth_div6/bu142)| 1283  |
createth__n02231_6:o               | NONE(*)(createth_div4/bu58)| 1283  |
createth__n02231_5:o               | NONE(*)(createth_div5/bu367)| 1283  |
createth__n02231_1:o               | NONE(*)(createth_div2/bu4575)| 1733  |
tilerender_drawspanoddeven__n00791:o|
NONE(*)(tilerender_drawspanoddeven_scs_bvdatapathin_126_0)| 127   |
tilerender_drawspanoddeven_ker3584441:o|
NONE(*)(tilerender_drawspanoddeven_scs_sspanstage_126_0)| 128   |
```

```
tilerender_drawspaneveneven__n00791:o|
NONE(*)(tilerender_drawspaneveneven_scs_bvdatapathin_126_0)| 127   |
tilerender_drawspaneveneven_ker3584441:o|
NONE(*)(tilerender_drawspaneveneven_scs_sspanstage_126_0)| 128   |
tilerender_drawspanoddodd__n00791:o| NONE(*)(tilerender_drawspanoddodd_scs_bvdatapathin_126_0)|
127   |
tilerender_drawspanoddodd_ker3584441:o|
NONE(*)(tilerender_drawspanoddodd_scs_sspanstage_126_0)| 128   |
tilerender_drawspanevenodd__n00791:o|
NONE(*)(tilerender_drawspanevenodd_scs_bvdatapathin_126_0)| 127   |
tilerender_drawspanevenodd_ker3584441:o|
NONE(*)(tilerender_drawspanevenodd_scs_sspanstage_126_0)| 128   |
ddrinterface_busy:q                | NONE                  | 8     |
sci_breset                         | BUFGP                 | 1228  |
clockhalf                          | BUFGP                 | 5416  |
clockfull                          | NONE                  | 2656  |
clockfull                          | ddrinterface_ddrctrl_u_clk_dlls_dll_int:clk2x| 230   |
thtbunit_thcontrol_insertcase_ffd4:q| NONE                  | 1     |
thtbunit_thethtb_resetinprogress:q | NONE                  | 7     |
createth__n02231:o                 | NONE(*)(createth_div3/bu1317)| 1283  |
bucketunit_sortboxfifo_out_data_127__n00011:o| NONE(*)(bucketunit_sortboxfifo_out_data_1_0)|
128   |
thtbunit_thcontrol__n01231:o       | NONE(*)(thtbunit_thcontrol_dontreadfromddr_0)| 1     |
thtbunit_thcontrol_out_busy__n00011:o| NONE(*)(thtbunit_thcontrol_out_busy_0)| 1     |
thtbunit_thcontrol_incramaddr__n00011:o| NONE(*)(thtbunit_thcontrol_incramaddr_0)| 1     |
bucketunit_sortboxfifo_out_addr_21__n00011:o| NONE(*)(bucketunit_sortboxfifo_out_addr_21_0)| 21
|
thtbunit_thcontrol_thnodeaddressnext_0__n00011:o|
NONE(*)(thtbunit_thcontrol_thnodeaddressnext_7_0)| 21    |
thtbunit_thcontrol_nextx_1__n00011:o| NONE(*)(thtbunit_thcontrol_nextx_3_0)| 6     |
readerunit_vertexreader__n00641:o  | NONE(*)(readerunit_vertexreader_vertex123_84_0)| 234   |
thtbunit_thethtb_output__n0351:o   | NONE(*)(thtbunit_thethtb_output_out_portb_0_0)| 63    |
thtbunit_thcontrol__n01221:o       | NONE(*)(thtbunit_thcontrol_ram_write_0)| 1     |
thtbunit_thcontrol__n01241:o       | NONE(*)(thtbunit_thcontrol_ram_req_0)| 1     |
thtbunit_thethtb_output__n0350:o   | NONE(*)(thtbunit_thethtb_output_out_addressb_sig_8_0)| 12
|
thtbunit_thethtb_output__n0348:o   | NONE(*)(thtbunit_thethtb_output_out_addressa_sig_6_0)| 11
|
thtbunit_thethtb_output__n0349:o   | NONE(*)(thtbunit_thethtb_output_nextcacheindex_5_0)| 7
|
ddrinterface__n2185185:o           | NONE(*)(ddrinterface_u_addr_20_0)| 23    |
thtbunit_thcontrol__n0121:o        | NONE(*)(thtbunit_thcontrol_nexty_3_0)| 6     |
-----------------------------------+-----------------------+-------+
(*) These 46 clock signal(s) are generated by combinatorial logic,
and XST is not able to identify which are the primary clock signals.
Please use the CLOCK_SIGNAL constraint to specify the clock signal(s) generated by
combinatorial logic.

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 19.432ns (Maximum Frequency: 51.461MHz)
   Minimum input arrival time before clock: 11.704ns
   Maximum output required time after clock: 11.248ns
   Maximum combinational path delay: 11.225ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'readerunit_thtbreader_currentcase_ffd4:q'
Delay:               2.552ns (Levels of Logic = 1)
  Source:            readerunit_thtbreader_nextthtbnext_0_0
  Destination:       readerunit_thtbreader_nextthtbnext_0_0
  Source Clock:      readerunit_thtbreader_currentcase_ffd4:q falling
  Destination Clock: readerunit_thtbreader_currentcase_ffd4:q falling

  Data Path: readerunit_thtbreader_nextthtbnext_0_0 to readerunit_thtbreader_nextthtbnext_0_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
```

```
     LD:g->q               4   0.674   1.069  readerunit_thtbreader_nextthtbnext_0_0
(readerunit_thtbreader_nextthtbnext_0_0)
     LUT3:i2->o            1   0.439   0.000  readerunit_thtbreader_mmux__n0075_result1
(readerunit_thtbreader__n0075)
     LD:d                      0.370          readerunit_thtbreader_nextthtbnext_0_0
    ----------------------------------------
    Total                      2.552ns (1.483ns logic, 1.069ns route)
                               (58.1% logic, 41.9% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'createth__n02231_2:o'
Delay:               7.200ns (Levels of Logic = 26)
  Source:            createth_div1/bu383
  Destination:       createth_div1/bu5430
  Source Clock:      createth__n02231_2:o rising
  Destination Clock: createth__n02231_2:o rising

  Data Path: createth_div1/bu383 to createth_div1/bu5430
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:c->q             1   0.568   0.408  bu383 (bu383)
     LUT4:i0->o           29   0.439   2.238  bu5285 (n401)
     LUT4:i2->o            1   0.439   0.000  bu5289 (n46234)
     MUXCY:s->o            1   0.298   0.000  bu5290 (n46185)
     MUXCY:ci->o           1   0.053   0.000  bu5296 (n46184)
     MUXCY:ci->o           1   0.053   0.000  bu5302 (n46183)
     MUXCY:ci->o           1   0.053   0.000  bu5308 (n46182)
     MUXCY:ci->o           1   0.053   0.000  bu5314 (n46181)
     MUXCY:ci->o           1   0.053   0.000  bu5320 (n46180)
     MUXCY:ci->o           1   0.053   0.000  bu5326 (n46179)
     MUXCY:ci->o           1   0.053   0.000  bu5332 (n46178)
     MUXCY:ci->o           1   0.053   0.000  bu5338 (n46177)
     MUXCY:ci->o           1   0.053   0.000  bu5344 (n46176)
     MUXCY:ci->o           1   0.053   0.000  bu5350 (n46175)
     MUXCY:ci->o           1   0.053   0.000  bu5356 (n46174)
     MUXCY:ci->o           1   0.053   0.000  bu5362 (n46173)
     MUXCY:ci->o           1   0.053   0.000  bu5368 (n46172)
     MUXCY:ci->o           1   0.053   0.000  bu5374 (n46171)
     MUXCY:ci->o           1   0.053   0.000  bu5380 (n46170)
     MUXCY:ci->o           1   0.053   0.000  bu5386 (n46169)
     MUXCY:ci->o           1   0.053   0.000  bu5392 (n46168)
     MUXCY:ci->o           1   0.053   0.000  bu5398 (n46167)
     MUXCY:ci->o           1   0.053   0.000  bu5404 (n46166)
     MUXCY:ci->o           1   0.053   0.000  bu5410 (n46165)
     MUXCY:ci->o           1   0.053   0.000  bu5416 (n46164)
     MUXCY:ci->o           0   0.053   0.000  bu5422 (n46163)
     XORCY:ci->o           1   1.274   0.000  bu5429 (n46187)
     FDCE:d                    0.370          bu5430
    ----------------------------------------
    Total                      7.200ns (4.554ns logic, 2.646ns route)
                               (63.2% logic, 36.8% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'createth__n02231_3:o'
Delay:               7.200ns (Levels of Logic = 26)
  Source:            createth_div7/bu329
  Destination:       createth_div7/bu3567
  Source Clock:      createth__n02231_3:o rising
  Destination Clock: createth__n02231_3:o rising

  Data Path: createth_div7/bu329 to createth_div7/bu3567
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:c->q             1   0.568   0.408  bu329 (bu329)
     LUT4:i0->o           29   0.439   2.238  bu3422 (n365)
     LUT4:i2->o            1   0.439   0.000  bu3426 (n31825)
     MUXCY:s->o            1   0.298   0.000  bu3427 (n31776)
     MUXCY:ci->o           1   0.053   0.000  bu3433 (n31775)
     MUXCY:ci->o           1   0.053   0.000  bu3439 (n31774)
     MUXCY:ci->o           1   0.053   0.000  bu3445 (n31773)
     MUXCY:ci->o           1   0.053   0.000  bu3451 (n31772)
```

```
     MUXCY:ci->o           1   0.053   0.000  bu3457 (n31771)
     MUXCY:ci->o           1   0.053   0.000  bu3463 (n31770)
     MUXCY:ci->o           1   0.053   0.000  bu3469 (n31769)
     MUXCY:ci->o           1   0.053   0.000  bu3475 (n31768)
     MUXCY:ci->o           1   0.053   0.000  bu3481 (n31767)
     MUXCY:ci->o           1   0.053   0.000  bu3487 (n31766)
     MUXCY:ci->o           1   0.053   0.000  bu3493 (n31765)
     MUXCY:ci->o           1   0.053   0.000  bu3499 (n31764)
     MUXCY:ci->o           1   0.053   0.000  bu3505 (n31763)
     MUXCY:ci->o           1   0.053   0.000  bu3511 (n31762)
     MUXCY:ci->o           1   0.053   0.000  bu3517 (n31761)
     MUXCY:ci->o           1   0.053   0.000  bu3523 (n31760)
     MUXCY:ci->o           1   0.053   0.000  bu3529 (n31759)
     MUXCY:ci->o           1   0.053   0.000  bu3535 (n31758)
     MUXCY:ci->o           1   0.053   0.000  bu3541 (n31757)
     MUXCY:ci->o           1   0.053   0.000  bu3547 (n31756)
     MUXCY:ci->o           1   0.053   0.000  bu3553 (n31755)
     MUXCY:ci->o           0   0.053   0.000  bu3559 (n31754)
     XORCY:ci->o           1   1.274   0.000  bu3566 (n31778)
     FDCE:d                    0.370          bu3567
    ----------------------------------------
    Total                      7.200ns (4.554ns logic, 2.646ns route)
                               (63.2% logic, 36.8% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'createth__n02231_4:o'
Delay:               7.200ns (Levels of Logic = 26)
  Source:            createth_div6/bu329
  Destination:       createth_div6/bu3567
  Source Clock:      createth__n02231_4:o rising
  Destination Clock: createth__n02231_4:o rising

  Data Path: createth_div6/bu329 to createth_div6/bu3567
                                 Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:c->q             1   0.568   0.408  bu329 (bu329)
     LUT4:i0->o           29   0.439   2.238  bu3422 (n365)
     LUT4:i2->o            1   0.439   0.000  bu3426 (n31825)
     MUXCY:s->o            1   0.298   0.000  bu3427 (n31776)
     MUXCY:ci->o           1   0.053   0.000  bu3433 (n31775)
     MUXCY:ci->o           1   0.053   0.000  bu3439 (n31774)
     MUXCY:ci->o           1   0.053   0.000  bu3445 (n31773)
     MUXCY:ci->o           1   0.053   0.000  bu3451 (n31772)
     MUXCY:ci->o           1   0.053   0.000  bu3457 (n31771)
     MUXCY:ci->o           1   0.053   0.000  bu3463 (n31770)
     MUXCY:ci->o           1   0.053   0.000  bu3469 (n31769)
     MUXCY:ci->o           1   0.053   0.000  bu3475 (n31768)
     MUXCY:ci->o           1   0.053   0.000  bu3481 (n31767)
     MUXCY:ci->o           1   0.053   0.000  bu3487 (n31766)
     MUXCY:ci->o           1   0.053   0.000  bu3493 (n31765)
     MUXCY:ci->o           1   0.053   0.000  bu3499 (n31764)
     MUXCY:ci->o           1   0.053   0.000  bu3505 (n31763)
     MUXCY:ci->o           1   0.053   0.000  bu3511 (n31762)
     MUXCY:ci->o           1   0.053   0.000  bu3517 (n31761)
     MUXCY:ci->o           1   0.053   0.000  bu3523 (n31760)
     MUXCY:ci->o           1   0.053   0.000  bu3529 (n31759)
     MUXCY:ci->o           1   0.053   0.000  bu3535 (n31758)
     MUXCY:ci->o           1   0.053   0.000  bu3541 (n31757)
     MUXCY:ci->o           1   0.053   0.000  bu3547 (n31756)
     MUXCY:ci->o           1   0.053   0.000  bu3553 (n31755)
     MUXCY:ci->o           0   0.053   0.000  bu3559 (n31754)
     XORCY:ci->o           1   1.274   0.000  bu3566 (n31778)
     FDCE:d                    0.370          bu3567
    ----------------------------------------
    Total                      7.200ns (4.554ns logic, 2.646ns route)
                               (63.2% logic, 36.8% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'createth__n02231_6:o'
Delay:               7.200ns (Levels of Logic = 26)
  Source:            createth_div4/bu329
  Destination:       createth_div4/bu3567
```

```
  Source Clock:      createth__n02231_6:o rising
  Destination Clock: createth__n02231_6:o rising

  Data Path: createth_div4/bu329 to createth_div4/bu3567
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:c->q             1   0.568   0.408  bu329 (bu329)
     LUT4:i0->o           29   0.439   2.238  bu3422 (n365)
     LUT4:i2->o            1   0.439   0.000  bu3426 (n31825)
     MUXCY:s->o            1   0.298   0.000  bu3427 (n31776)
     MUXCY:ci->o           1   0.053   0.000  bu3433 (n31775)
     MUXCY:ci->o           1   0.053   0.000  bu3439 (n31774)
     MUXCY:ci->o           1   0.053   0.000  bu3445 (n31773)
     MUXCY:ci->o           1   0.053   0.000  bu3451 (n31772)
     MUXCY:ci->o           1   0.053   0.000  bu3457 (n31771)
     MUXCY:ci->o           1   0.053   0.000  bu3463 (n31770)
     MUXCY:ci->o           1   0.053   0.000  bu3469 (n31769)
     MUXCY:ci->o           1   0.053   0.000  bu3475 (n31768)
     MUXCY:ci->o           1   0.053   0.000  bu3481 (n31767)
     MUXCY:ci->o           1   0.053   0.000  bu3487 (n31766)
     MUXCY:ci->o           1   0.053   0.000  bu3493 (n31765)
     MUXCY:ci->o           1   0.053   0.000  bu3499 (n31764)
     MUXCY:ci->o           1   0.053   0.000  bu3505 (n31763)
     MUXCY:ci->o           1   0.053   0.000  bu3511 (n31762)
     MUXCY:ci->o           1   0.053   0.000  bu3517 (n31761)
     MUXCY:ci->o           1   0.053   0.000  bu3523 (n31760)
     MUXCY:ci->o           1   0.053   0.000  bu3529 (n31759)
     MUXCY:ci->o           1   0.053   0.000  bu3535 (n31758)
     MUXCY:ci->o           1   0.053   0.000  bu3541 (n31757)
     MUXCY:ci->o           1   0.053   0.000  bu3547 (n31756)
     MUXCY:ci->o           1   0.053   0.000  bu3553 (n31755)
     MUXCY:ci->o           0   0.053   0.000  bu3559 (n31754)
     XORCY:ci->o           1   1.274   0.000  bu3566 (n31778)
     FDCE:d                    0.370          bu3567
    ----------------------------------------
    Total                      7.200ns (4.554ns logic, 2.646ns route)
                               (63.2% logic, 36.8% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'createth__n02231_5:o'
Delay:               7.200ns (Levels of Logic = 26)
  Source:            createth_div5/bu329
  Destination:       createth_div5/bu3567
  Source Clock:      createth__n02231_5:o rising
  Destination Clock: createth__n02231_5:o rising

  Data Path: createth_div5/bu329 to createth_div5/bu3567
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:c->q             1   0.568   0.408  bu329 (bu329)
     LUT4:i0->o           29   0.439   2.238  bu3422 (n365)
     LUT4:i2->o            1   0.439   0.000  bu3426 (n31825)
     MUXCY:s->o            1   0.298   0.000  bu3427 (n31776)
     MUXCY:ci->o           1   0.053   0.000  bu3433 (n31775)
     MUXCY:ci->o           1   0.053   0.000  bu3439 (n31774)
     MUXCY:ci->o           1   0.053   0.000  bu3445 (n31773)
     MUXCY:ci->o           1   0.053   0.000  bu3451 (n31772)
     MUXCY:ci->o           1   0.053   0.000  bu3457 (n31771)
     MUXCY:ci->o           1   0.053   0.000  bu3463 (n31770)
     MUXCY:ci->o           1   0.053   0.000  bu3469 (n31769)
     MUXCY:ci->o           1   0.053   0.000  bu3475 (n31768)
     MUXCY:ci->o           1   0.053   0.000  bu3481 (n31767)
     MUXCY:ci->o           1   0.053   0.000  bu3487 (n31766)
     MUXCY:ci->o           1   0.053   0.000  bu3493 (n31765)
     MUXCY:ci->o           1   0.053   0.000  bu3499 (n31764)
     MUXCY:ci->o           1   0.053   0.000  bu3505 (n31763)
     MUXCY:ci->o           1   0.053   0.000  bu3511 (n31762)
     MUXCY:ci->o           1   0.053   0.000  bu3517 (n31761)
     MUXCY:ci->o           1   0.053   0.000  bu3523 (n31760)
     MUXCY:ci->o           1   0.053   0.000  bu3529 (n31759)
     MUXCY:ci->o           1   0.053   0.000  bu3535 (n31758)
```

```
     MUXCY:ci->o           1   0.053   0.000  bu3541 (n31757)
     MUXCY:ci->o           1   0.053   0.000  bu3547 (n31756)
     MUXCY:ci->o           1   0.053   0.000  bu3553 (n31755)
     MUXCY:ci->o           0   0.053   0.000  bu3559 (n31754)
     XORCY:ci->o           1   1.274   0.000  bu3566 (n31778)
     FDCE:d                    0.370          bu3567
    ----------------------------------------
    Total                      7.200ns (4.554ns logic, 2.646ns route)
                               (63.2% logic, 36.8% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'createth__n02231_1:o'
Delay:               7.200ns (Levels of Logic = 26)
  Source:            createth_div2/bu383
  Destination:       createth_div2/bu5430
  Source Clock:      createth__n02231_1:o rising
  Destination Clock: createth__n02231_1:o rising

  Data Path: createth_div2/bu383 to createth_div2/bu5430
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:c->q             1   0.568   0.408  bu383 (bu383)
     LUT4:i0->o           29   0.439   2.238  bu5285 (n401)
     LUT4:i2->o            1   0.439   0.000  bu5289 (n46234)
     MUXCY:s->o            1   0.298   0.000  bu5290 (n46185)
     MUXCY:ci->o           1   0.053   0.000  bu5296 (n46184)
     MUXCY:ci->o           1   0.053   0.000  bu5302 (n46183)
     MUXCY:ci->o           1   0.053   0.000  bu5308 (n46182)
     MUXCY:ci->o           1   0.053   0.000  bu5314 (n46181)
     MUXCY:ci->o           1   0.053   0.000  bu5320 (n46180)
     MUXCY:ci->o           1   0.053   0.000  bu5326 (n46179)
     MUXCY:ci->o           1   0.053   0.000  bu5332 (n46178)
     MUXCY:ci->o           1   0.053   0.000  bu5338 (n46177)
     MUXCY:ci->o           1   0.053   0.000  bu5344 (n46176)
     MUXCY:ci->o           1   0.053   0.000  bu5350 (n46175)
     MUXCY:ci->o           1   0.053   0.000  bu5356 (n46174)
     MUXCY:ci->o           1   0.053   0.000  bu5362 (n46173)
     MUXCY:ci->o           1   0.053   0.000  bu5368 (n46172)
     MUXCY:ci->o           1   0.053   0.000  bu5374 (n46171)
     MUXCY:ci->o           1   0.053   0.000  bu5380 (n46170)
     MUXCY:ci->o           1   0.053   0.000  bu5386 (n46169)
     MUXCY:ci->o           1   0.053   0.000  bu5392 (n46168)
     MUXCY:ci->o           1   0.053   0.000  bu5398 (n46167)
     MUXCY:ci->o           1   0.053   0.000  bu5404 (n46166)
     MUXCY:ci->o           1   0.053   0.000  bu5410 (n46165)
     MUXCY:ci->o           1   0.053   0.000  bu5416 (n46164)
     MUXCY:ci->o           0   0.053   0.000  bu5422 (n46163)
     XORCY:ci->o           1   1.274   0.000  bu5429 (n46187)
     FDCE:d                    0.370          bu5430
    ----------------------------------------
    Total                      7.200ns (4.554ns logic, 2.646ns route)
                               (63.2% logic, 36.8% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'ddrinterface_busy:q'
Delay:               10.377ns (Levels of Logic = 6)
  Source:            ddrinterface_movedatamask_3_0
  Destination:       ddrinterface_longburst_0
  Source Clock:      ddrinterface_busy:q rising
  Destination Clock: ddrinterface_busy:q rising

  Data Path: ddrinterface_movedatamask_3_0 to ddrinterface_longburst_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               9   0.674   1.321  ddrinterface_movedatamask_3_0
(ddrinterface_movedatamask_3_0)
     LUT2:i1->o           33   0.439   2.242  ddrinterface_busymask<3>1 (out_busy2)
     LUT2:i0->o            1   0.439   0.408  readerunit_thtbreader_currentcase_xx_ffd3_sw0
(n607448)
     LUT4:i3->o            2   0.439   0.790  readerunit_thtbreader_currentcase_xx_ffd3
(in_req2)
```

```
     LUT3:i2->o            1   0.439   0.408  ddrinterface__n0867104 (choice6450)
     LUT3:i0->o            8   0.439   1.270  ddrinterface__n0867123 (ddrinterface__n0867)
     MUXF5:s->o            1   0.699   0.000  ddrinterface__n076329 (ddrinterface__n0763)
     LD:d                      0.370          ddrinterface_longburst_0
    ----------------------------------------
    Total                    10.377ns (3.938ns logic, 6.439ns route)
                                      (37.9% logic, 62.1% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'clockhalf'
Delay:               19.432ns (Levels of Logic = 83)
  Source:            bucketunit_sortbox_in2vertex1_0
  Destination:       bucketunit_sortbox_xmax_10
  Source Clock:      clockhalf rising
  Destination Clock: clockhalf rising

  Data Path: bucketunit_sortbox_in2vertex1_0 to bucketunit_sortbox_xmax_10
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDE:c->q              7   0.568   1.219  bucketunit_sortbox_in2vertex1_0
(bucketunit_sortbox_in2vertex1_0)
     LUT2_L:I0->LO         1   0.439   0.000  bucketunit_sortbox_mcompar__n0104_inst_lut2_6921
(bucketunit_sortbox_mcompar__n0104_inst_lut2_692)
     MUXCY:s->o            1   0.298   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_687
(bucketunit_sortbox_mcompar__n0104_inst_cy_687)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_688
(bucketunit_sortbox_mcompar__n0104_inst_cy_688)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_689
(bucketunit_sortbox_mcompar__n0104_inst_cy_689)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_690
(bucketunit_sortbox_mcompar__n0104_inst_cy_690)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_691
(bucketunit_sortbox_mcompar__n0104_inst_cy_691)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_692
(bucketunit_sortbox_mcompar__n0104_inst_cy_692)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_693
(bucketunit_sortbox_mcompar__n0104_inst_cy_693)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_694
(bucketunit_sortbox_mcompar__n0104_inst_cy_694)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_695
(bucketunit_sortbox_mcompar__n0104_inst_cy_695)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_696
(bucketunit_sortbox_mcompar__n0104_inst_cy_696)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_697
(bucketunit_sortbox_mcompar__n0104_inst_cy_697)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_698
(bucketunit_sortbox_mcompar__n0104_inst_cy_698)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_699
(bucketunit_sortbox_mcompar__n0104_inst_cy_699)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_700
(bucketunit_sortbox_mcompar__n0104_inst_cy_700)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_701
(bucketunit_sortbox_mcompar__n0104_inst_cy_701)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_702
(bucketunit_sortbox_mcompar__n0104_inst_cy_702)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_703
(bucketunit_sortbox_mcompar__n0104_inst_cy_703)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_704
(bucketunit_sortbox_mcompar__n0104_inst_cy_704)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_705
(bucketunit_sortbox_mcompar__n0104_inst_cy_705)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_706
(bucketunit_sortbox_mcompar__n0104_inst_cy_706)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_707
(bucketunit_sortbox_mcompar__n0104_inst_cy_707)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0104_inst_cy_708
(bucketunit_sortbox_mcompar__n0104_inst_cy_708)
     MUXCY:ci->o          96   0.053   2.305  bucketunit_sortbox_mcompar__n0104_inst_cy_709
(bucketunit_sortbox__n0104)
     LUT3:i0->o            1   0.439   0.408  bucketunit_sortbox_mmux__n0050_i22_result1
(bucketunit_sortbox__n0157<0>)
```

```
     MUXCY:di->o           1   0.462   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_687
(bucketunit_sortbox_mcompar__n0105_inst_cy_687)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_688
(bucketunit_sortbox_mcompar__n0105_inst_cy_688)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_689
(bucketunit_sortbox_mcompar__n0105_inst_cy_689)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_690
(bucketunit_sortbox_mcompar__n0105_inst_cy_690)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_691
(bucketunit_sortbox_mcompar__n0105_inst_cy_691)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_692
(bucketunit_sortbox_mcompar__n0105_inst_cy_692)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_693
(bucketunit_sortbox_mcompar__n0105_inst_cy_693)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_694
(bucketunit_sortbox_mcompar__n0105_inst_cy_694)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_695
(bucketunit_sortbox_mcompar__n0105_inst_cy_695)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_696
(bucketunit_sortbox_mcompar__n0105_inst_cy_696)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_697
(bucketunit_sortbox_mcompar__n0105_inst_cy_697)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_698
(bucketunit_sortbox_mcompar__n0105_inst_cy_698)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_699
(bucketunit_sortbox_mcompar__n0105_inst_cy_699)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_700
(bucketunit_sortbox_mcompar__n0105_inst_cy_700)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_701
(bucketunit_sortbox_mcompar__n0105_inst_cy_701)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_702
(bucketunit_sortbox_mcompar__n0105_inst_cy_702)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_703
(bucketunit_sortbox_mcompar__n0105_inst_cy_703)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_704
(bucketunit_sortbox_mcompar__n0105_inst_cy_704)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_705
(bucketunit_sortbox_mcompar__n0105_inst_cy_705)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_706
(bucketunit_sortbox_mcompar__n0105_inst_cy_706)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_707
(bucketunit_sortbox_mcompar__n0105_inst_cy_707)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0105_inst_cy_708
(bucketunit_sortbox_mcompar__n0105_inst_cy_708)
     MUXCY:ci->o          51   0.053   2.260  bucketunit_sortbox_mcompar__n0105_inst_cy_709
(bucketunit_sortbox__n0105)
     LUT4_L:I0->LO         1   0.439   0.000  bucketunit_sortbox_mcompar__n0106_inst_lut2_6921
(bucketunit_sortbox_mcompar__n0106_inst_lut2_692)
     MUXCY:s->o            1   0.298   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_687
(bucketunit_sortbox_mcompar__n0106_inst_cy_687)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_688
(bucketunit_sortbox_mcompar__n0106_inst_cy_688)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_689
(bucketunit_sortbox_mcompar__n0106_inst_cy_689)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_690
(bucketunit_sortbox_mcompar__n0106_inst_cy_690)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_691
(bucketunit_sortbox_mcompar__n0106_inst_cy_691)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_692
(bucketunit_sortbox_mcompar__n0106_inst_cy_692)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_693
(bucketunit_sortbox_mcompar__n0106_inst_cy_693)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_694
(bucketunit_sortbox_mcompar__n0106_inst_cy_694)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_695
(bucketunit_sortbox_mcompar__n0106_inst_cy_695)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_696
(bucketunit_sortbox_mcompar__n0106_inst_cy_696)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_697
(bucketunit_sortbox_mcompar__n0106_inst_cy_697)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_698
(bucketunit_sortbox_mcompar__n0106_inst_cy_698)
```

```
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_699
(bucketunit_sortbox_mcompar__n0106_inst_cy_699)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_700
(bucketunit_sortbox_mcompar__n0106_inst_cy_700)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_701
(bucketunit_sortbox_mcompar__n0106_inst_cy_701)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_702
(bucketunit_sortbox_mcompar__n0106_inst_cy_702)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_703
(bucketunit_sortbox_mcompar__n0106_inst_cy_703)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_704
(bucketunit_sortbox_mcompar__n0106_inst_cy_704)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_705
(bucketunit_sortbox_mcompar__n0106_inst_cy_705)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_706
(bucketunit_sortbox_mcompar__n0106_inst_cy_706)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_707
(bucketunit_sortbox_mcompar__n0106_inst_cy_707)
     MUXCY:ci->o           2   0.053   0.000  bucketunit_sortbox_mcompar__n0106_inst_cy_708
(bucketunit_sortbox_mcompar__n0106_inst_cy_708)
     MUXCY:ci->o          15   0.053   1.632  bucketunit_sortbox_mcompar__n0106_inst_cy_709
(bucketunit_sortbox__n0106)
     LUT3_D:I1->O         18   0.439   1.812  bucketunit_sortbox_ker4027481
(bucketunit_sortbox_n402750)
     LUT3:i2->o            0   0.439   0.000  bucketunit_sortbox_mmux__n0057_i8_result
(bucketunit_sortbox__n0057<14>)
     MUXCY:di->o           1   0.462   0.000  bucketunit_sortbox_msub_x1max_inst_cy_2
(bucketunit_sortbox_msub_x1max_inst_cy_2)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_msub_x1max_inst_cy_3
(bucketunit_sortbox_msub_x1max_inst_cy_3)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_msub_x1max_inst_cy_4
(bucketunit_sortbox_msub_x1max_inst_cy_4)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_msub_x1max_inst_cy_5
(bucketunit_sortbox_msub_x1max_inst_cy_5)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_msub_x1max_inst_cy_6
(bucketunit_sortbox_msub_x1max_inst_cy_6)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_msub_x1max_inst_cy_7
(bucketunit_sortbox_msub_x1max_inst_cy_7)
     MUXCY:ci->o           1   0.053   0.000  bucketunit_sortbox_msub_x1max_inst_cy_8
(bucketunit_sortbox_msub_x1max_inst_cy_8)
     MUXCY:ci->o           0   0.053   0.000  bucketunit_sortbox_msub_x1max_inst_cy_9
(bucketunit_sortbox_msub_x1max_inst_cy_9)
     XORCY:ci->o           1   1.274   0.000  bucketunit_sortbox_msub_x1max_inst_sum_10
(bucketunit_sortbox_x1max<10>)
     FDE:d                     0.370          bucketunit_sortbox_xmax_10
    ----------------------------------------
    Total                     19.432ns (9.796ns logic, 9.636ns route)
                                       (50.4% logic, 49.6% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'clockfull'
Delay:               13.235ns (Levels of Logic = 7)
  Source:            thtbunit_thetpt_out_tptselect_sig
  Destination:       thtbunit_thethtb_thethtbcache/b13
  Source Clock:      clockfull rising
  Destination Clock: clockfull rising

  Data Path: thtbunit_thetpt_out_tptselect_sig to thtbunit_thethtb_thethtbcache/b13
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:c->q             17   0.568   1.753  thtbunit_thetpt_out_tptselect_sig
(thtbunit_thetpt_out_tptselect_sig)
     LUT3:i0->o            6   0.439   1.170  thtbunit_thetpt_mmux_out_dataout_i13_result1
(thtbunit_in_tpt_table<4>)
     LUT4:i0->o            1   0.439   0.408  thtbunit_thcontrol__n00464 (choice6487)
     LUT2:i0->o            2   0.439   0.790  thtbunit_thcontrol__n004610 (choice6491)
     LUT4:i0->o            4   0.439   1.069  thtbunit_thcontrol_ker4195561
(thtbunit_thcontrol_n419558)
     LUT4:i3->o           44   0.439   2.253  thtbunit_thcontrol_ker4195181
(thtbunit_thcontrol_n419520)
     LUT2:i0->o            3   0.439   0.981  thtbunit_thcontrol_out_thtb_write1
(thtbunit_out_thtb_write)
```

```
     LUT4:i2->o            6   0.439   1.170  thtbunit_thethtb_mmux__n0022_result1
(thtbunit_thethtb__n0022)
     begin scope: 'thtbunit_thethtb_thethtbcache'
     RAMB16_S4_S9:web          0.000          b13
    ----------------------------------------
    Total                     13.235ns (3.641ns logic, 9.594ns route)
                                       (27.5% logic, 72.5% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'createth__n02231:o'
Delay:               7.200ns (Levels of Logic = 26)
  Source:            createth_div3/bu329
  Destination:       createth_div3/bu3567
  Source Clock:      createth__n02231:o rising
  Destination Clock: createth__n02231:o rising

  Data Path: createth_div3/bu329 to createth_div3/bu3567
                             Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:c->q             1   0.568   0.408  bu329 (bu329)
     LUT4:i0->o           29   0.439   2.238  bu3422 (n365)
     LUT4:i2->o            1   0.439   0.000  bu3426 (n31825)
     MUXCY:s->o            1   0.298   0.000  bu3427 (n31776)
     MUXCY:ci->o           1   0.053   0.000  bu3433 (n31775)
     MUXCY:ci->o           1   0.053   0.000  bu3439 (n31774)
     MUXCY:ci->o           1   0.053   0.000  bu3445 (n31773)
     MUXCY:ci->o           1   0.053   0.000  bu3451 (n31772)
     MUXCY:ci->o           1   0.053   0.000  bu3457 (n31771)
     MUXCY:ci->o           1   0.053   0.000  bu3463 (n31770)
     MUXCY:ci->o           1   0.053   0.000  bu3469 (n31769)
     MUXCY:ci->o           1   0.053   0.000  bu3475 (n31768)
     MUXCY:ci->o           1   0.053   0.000  bu3481 (n31767)
     MUXCY:ci->o           1   0.053   0.000  bu3487 (n31766)
     MUXCY:ci->o           1   0.053   0.000  bu3493 (n31765)
     MUXCY:ci->o           1   0.053   0.000  bu3499 (n31764)
     MUXCY:ci->o           1   0.053   0.000  bu3505 (n31763)
     MUXCY:ci->o           1   0.053   0.000  bu3511 (n31762)
     MUXCY:ci->o           1   0.053   0.000  bu3517 (n31761)
     MUXCY:ci->o           1   0.053   0.000  bu3523 (n31760)
     MUXCY:ci->o           1   0.053   0.000  bu3529 (n31759)
     MUXCY:ci->o           1   0.053   0.000  bu3535 (n31758)
     MUXCY:ci->o           1   0.053   0.000  bu3541 (n31757)
     MUXCY:ci->o           1   0.053   0.000  bu3547 (n31756)
     MUXCY:ci->o           1   0.053   0.000  bu3553 (n31755)
     MUXCY:ci->o           0   0.053   0.000  bu3559 (n31754)
     XORCY:ci->o           1   1.274   0.000  bu3566 (n31778)
     FDCE:d                    0.370          bu3567
    ----------------------------------------
    Total                      7.200ns (4.554ns logic, 2.646ns route)
                                       (63.2% logic, 36.8% route)

--------------------------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'readerunit_vertexreader__n00641:o'
Delay:               2.602ns (Levels of Logic = 1)
  Source:            readerunit_vertexreader_vertex123_69_0
  Destination:       readerunit_vertexreader_vertex123_69_0
  Source Clock:      readerunit_vertexreader__n00641:o falling
  Destination Clock: readerunit_vertexreader__n00641:o falling

  Data Path: readerunit_vertexreader_vertex123_69_0 to readerunit_vertexreader_vertex123_69_0
                             Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               5   0.674   1.119  readerunit_vertexreader_vertex123_69_0
(readerunit_vertexreader_vertex123_69_0)
     LUT4:i3->o            1   0.439   0.000  readerunit_vertexreader__n0061<69>1
(readerunit_vertexreader__n0061<69>)
     LD:d                      0.370          readerunit_vertexreader_vertex123_69_0
    ----------------------------------------
    Total                      2.602ns (1.483ns logic, 1.119ns route)
                                       (57.0% logic, 43.0% route)
```

```
--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'readerunit_thtbreader__n02311:o'
Offset:              3.019ns (Levels of Logic = 2)
  Source:            thtbunit_thetpt_blocktpt1
  Destination:       readerunit_thtbreader_out_addr_3_0
  Destination Clock: readerunit_thtbreader__n02311:o falling

  Data Path: thtbunit_thetpt_blocktpt1 to readerunit_thtbreader_out_addr_3_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<1>    3   0.000   0.981  thtbunit_thetpt_blocktpt1
(thtbunit_thetpt__n0046<19>)
     LUT3:i1->o            2   0.439   0.790  thtbunit_thetpt_mmux_out_dataoutb_i16_result1
(out_dataoutb<1>)
     LUT4:i1->o            1   0.439   0.000  readerunit_thtbreader__n0077<3>1
(readerunit_thtbreader__n0077<3>)
     LD:d                  0.370          readerunit_thtbreader_out_addr_3_0
    ----------------------------------------
    Total                  3.019ns (1.248ns logic, 1.771ns route)
                                   (41.3% logic, 58.7% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'thtbunit_thcontrol__n012529:o'
Offset:              9.484ns (Levels of Logic = 6)
  Source:            thtbunit_thetpt_blocktpt0
  Destination:       thtbunit_thcontrol_out_newtptheader_15_0
  Destination Clock: thtbunit_thcontrol__n012529:o falling

  Data Path: thtbunit_thetpt_blocktpt0 to thtbunit_thcontrol_out_newtptheader_15_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<4>    2   0.000   0.790  thtbunit_thetpt_blocktpt0
(thtbunit_thetpt__n0046<4>)
     LUT3:i1->o            6   0.439   1.170  thtbunit_thetpt_mmux_out_dataout_i13_result1
(thtbunit_in_tpt_table<4>)
     LUT4:i0->o            1   0.439   0.408  thtbunit_thcontrol__n00464 (choice6487)
     LUT2:i0->o            2   0.439   0.790  thtbunit_thcontrol__n004610 (choice6491)
     LUT4:i0->o            4   0.439   1.069  thtbunit_thcontrol_ker4195561
(thtbunit_thcontrol_n419558)
     LUT4:i3->o           44   0.439   2.253  thtbunit_thcontrol_ker4195181
(thtbunit_thcontrol_n419520)
     LUT4:i0->o            1   0.439   0.000  thtbunit_thcontrol__n0065<15>1
(thtbunit_thcontrol__n0065<15>)
     LD:d                  0.370          thtbunit_thcontrol_out_newtptheader_15_0
    ----------------------------------------
    Total                  9.484ns (3.004ns logic, 6.480ns route)
                                   (31.7% logic, 68.3% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'thtbunit_thethtb_output__n0347:o'
Offset:              3.399ns (Levels of Logic = 2)
  Source:            thtbunit_thetpt_blocktpt0
  Destination:       thtbunit_thethtb_output_out_addr_5_0
  Destination Clock: thtbunit_thethtb_output__n0347:o falling

  Data Path: thtbunit_thetpt_blocktpt0 to thtbunit_thethtb_output_out_addr_5_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<3>    3   0.000   0.981  thtbunit_thetpt_blocktpt0
(thtbunit_thetpt__n0046<3>)
     LUT3:i1->o            6   0.439   1.170  thtbunit_thetpt_mmux_out_dataout_i14_result1
(thtbunit_in_tpt_table<3>)
     LUT4:i2->o            1   0.439   0.000  thtbunit_thethtb_output__n0259<5>1
(thtbunit_thethtb_output__n0259<5>)
     LD:d                  0.370          thtbunit_thethtb_output_out_addr_5_0
    ----------------------------------------
    Total                  3.399ns (1.248ns logic, 2.151ns route)
                                   (36.7% logic, 63.3% route)

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

```
Timing constraint: Default OFFSET IN BEFORE for Clock 'ddrinterface__n218759:o'
Offset:              2.889ns (Levels of Logic = 3)
  Source:            in_addr1<20>
  Destination:       ddrinterface_addr_20_0
  Destination Clock: ddrinterface__n218759:o falling

  Data Path: in_addr1<20> to ddrinterface_addr_20_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:i->o             1   0.825   0.408  in_addr1_20_ibuf (in_addr1_20_ibuf)
     LUT4:i1->o            1   0.439   0.408  ddrinterface__n0772<20>_sw0 (n593146)
     LUT3:i0->o            1   0.439   0.000  ddrinterface__n0772<20> (ddrinterface__n0772<20>)
     LD:d                      0.370          ddrinterface_addr_20_0
    ----------------------------------------
    Total                      2.889ns (2.073ns logic, 0.816ns route)
                                       (71.8% logic, 28.2% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'thtbunit_thcontrol__n0126:o'
Offset:              11.676ns (Levels of Logic = 7)
  Source:            thtbunit_thetpt_blocktpt0
  Destination:       thtbunit_thcontrol_out_newthtb_ram_15_0
  Destination Clock: thtbunit_thcontrol__n0126:o falling

  Data Path: thtbunit_thetpt_blocktpt0 to thtbunit_thcontrol_out_newthtb_ram_15_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<4>    2   0.000   0.790  thtbunit_thetpt_blocktpt0
(thtbunit_thetpt__n0046<4>)
     LUT3:i1->o            6   0.439   1.170  thtbunit_thetpt_mmux_out_dataout_i13_result1
(thtbunit_in_tpt_table<4>)
     LUT4:i0->o            1   0.439   0.408  thtbunit_thcontrol__n00464 (choice6487)
     LUT2:i0->o            2   0.439   0.790  thtbunit_thcontrol__n004610 (choice6491)
     LUT4:i0->o            4   0.439   1.069  thtbunit_thcontrol_ker4195561
(thtbunit_thcontrol_n419558)
     LUT4:i3->o           44   0.439   2.253  thtbunit_thcontrol_ker4195181
(thtbunit_thcontrol_n419520)
     LUT3:i0->o           17   0.439   1.753  thtbunit_thcontrol_out_tpt_write1
(thtbunit_out_tpt_write)
     LUT4:i1->o            1   0.439   0.000  thtbunit_thcontrol__n0066<15>1
(thtbunit_thcontrol__n0066<15>)
     LD:d                      0.370          thtbunit_thcontrol_out_newthtb_ram_15_0
    ----------------------------------------
    Total                     11.676ns (3.443ns logic, 8.233ns route)
                                       (29.5% logic, 70.5% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'ddrinterface_busy:q'
Offset:              10.178ns (Levels of Logic = 8)
  Source:            thtbunit_thetpt_blocktpt1
  Destination:       ddrinterface_longburst_0
  Destination Clock: ddrinterface_busy:q rising

  Data Path: thtbunit_thetpt_blocktpt1 to ddrinterface_longburst_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<1>    3   0.000   0.981  thtbunit_thetpt_blocktpt1
(thtbunit_thetpt__n0046<19>)
     LUT3:i1->o            2   0.439   0.790  thtbunit_thetpt_mmux_out_dataoutb_i16_result1
(out_dataoutb<1>)
     LUT4:i0->o            1   0.439   0.408  readerunit_thtbreader__n005339_sw0_sw0 (n631679)
     LUT4:i3->o            1   0.439   0.408  readerunit_thtbreader__n005339_sw0 (n630423)
     LUT4:i3->o            3   0.439   0.981  readerunit_thtbreader__n005339
(readerunit_thtbreader__n0053)
     LUT4:i0->o            2   0.439   0.790  readerunit_thtbreader_currentcase_xx_ffd3
(in_req2)
     LUT3:i2->o            1   0.439   0.408  ddrinterface__n0867104 (choice6450)
     LUT3:i0->o            8   0.439   1.270  ddrinterface__n0867123 (ddrinterface__n0867)
     MUXF5:s->o            1   0.699   0.000  ddrinterface__n076329 (ddrinterface__n0763)
     LD:d                      0.370          ddrinterface_longburst_0
```

```
    ----------------------------------------
    Total                     10.178ns (4.142ns logic, 6.036ns route)
                                       (40.7% logic, 59.3% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'clockhalf'
Offset:              11.517ns (Levels of Logic = 8)
  Source:            sci_breset
  Destination:       tilerender_drawspanfifoieven/bu170
  Destination Clock: clockhalf rising

  Data Path: sci_breset to tilerender_drawspanfifoieven/bu170
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     BUFGP:i->o         2596   0.994   2.329  sci_breset_bufgp (sci_breset_bufgp)
     LUT3:i2->o            8   0.439   1.270  tilerender_ker4497771 (tilerender_n449779)
     LUT4:i2->o            1   0.439   0.000  tilerender_drawspaneveneven_ker35840515_f
(n632861)
     MUXF5:i0->o           1   0.436   0.408  tilerender_drawspaneveneven_ker35840515
(tilerender_drawspaneveneven_choice384)
     LUT4:i2->o            1   0.439   0.408  tilerender_drawspaneveneven_ker35840534
(tilerender_drawspaneveneven_n358407)
     LUT3:i1->o            4   0.439   1.069  tilerender_predraweven__n00311
(tilerender_scs_fifo3readeven)
     begin scope: 'tilerender_drawspanfifoieven'
     LUT4:i0->o            1   0.439   0.408  bu13 (n2)
     LUT4:i0->o            9   0.439   1.321  bu37 (n53)
     FDRE:ce                   0.240          bu170
    ----------------------------------------
    Total                     11.517ns (4.304ns logic, 7.213ns route)
                                       (37.4% logic, 62.6% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'clockfull'
Offset:              11.704ns (Levels of Logic = 7)
  Source:            thtbunit_thetpt_blocktpt0
  Destination:       thtbunit_thethtb_thethtbcache/b13
  Destination Clock: clockfull rising

  Data Path: thtbunit_thetpt_blocktpt0 to thtbunit_thethtb_thethtbcache/b13
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<4>    2   0.000   0.790  thtbunit_thetpt_blocktpt0
(thtbunit_thetpt__n0046<4>)
     LUT3:i1->o            6   0.439   1.170  thtbunit_thetpt_mmux_out_dataout_i13_result1
(thtbunit_in_tpt_table<4>)
     LUT4:i0->o            1   0.439   0.408  thtbunit_thcontrol__n00464 (choice6487)
     LUT2:i0->o            2   0.439   0.790  thtbunit_thcontrol__n004610 (choice6491)
     LUT4:i0->o            4   0.439   1.069  thtbunit_thcontrol_ker4195561
(thtbunit_thcontrol_n419558)
     LUT4:i3->o           44   0.439   2.253  thtbunit_thcontrol_ker4195181
(thtbunit_thcontrol_n419520)
     LUT2:i0->o            3   0.439   0.981  thtbunit_thcontrol_out_thtb_write1
(thtbunit_out_thtb_write)
     LUT4:i2->o            6   0.439   1.170  thtbunit_thethtb_mmux__n0022_result1
(thtbunit_thethtb__n0022)
     begin scope: 'thtbunit_thethtb_thethtbcache'
     RAMB16_S4_S9:web          0.000          b13
    ----------------------------------------
    Total                     11.704ns (3.073ns logic, 8.631ns route)
                                       (26.3% logic, 73.7% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'thtbunit_thcontrol_insertcase_ffd4:q'
Offset:              6.535ns (Levels of Logic = 12)
  Source:            thtbunit_thetpt_blocktpt0
  Destination:       thtbunit_thcontrol_out_thtb_writeentry_sig_0
  Destination Clock: thtbunit_thcontrol_insertcase_ffd4:q rising

  Data Path: thtbunit_thetpt_blocktpt0 to thtbunit_thcontrol_out_thtb_writeentry_sig_0
                                Gate     Net
```

```
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<1>    3   0.000   0.981  thtbunit_thetpt_blocktpt0
(thtbunit_thetpt__n0046<1>)
     LUT3:i1->o            5   0.439   1.119  thtbunit_thetpt_mmux_out_dataout_i16_result1
(thtbunit_in_tpt_table<1>)
     LUT4:i2->o            1   0.439   0.408  thtbunit_thethtb_mcompar__n0026_inst_lut4_181
(thtbunit_thethtb_mcompar__n0026_inst_lut4_18)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thethtb_mcompar__n0026_inst_cy_753
(thtbunit_thethtb_mcompar__n0026_inst_cy_753)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thethtb_mcompar__n0026_inst_cy_754
(thtbunit_thethtb_mcompar__n0026_inst_cy_754)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thethtb_mcompar__n0026_inst_cy_755
(thtbunit_thethtb_mcompar__n0026_inst_cy_755)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thethtb_mcompar__n0026_inst_cy_756
(thtbunit_thethtb_mcompar__n0026_inst_cy_756)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thethtb_mcompar__n0026_inst_cy_757
(thtbunit_thethtb_mcompar__n0026_inst_cy_757)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thethtb_mcompar__n0026_inst_cy_758
(thtbunit_thethtb_mcompar__n0026_inst_cy_758)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thethtb_mcompar__n0026_inst_cy_759
(thtbunit_thethtb_mcompar__n0026_inst_cy_759)
     MUXCY:ci->o           1   0.053   0.408  thtbunit_thethtb_mcompar__n0026_inst_cy_760
(thtbunit_thethtb__n0026)
     LUT4:i1->o            4   0.439   1.069  thtbunit_thethtb__n002435
(thtbunit_in_changecache)
     LUT2:i0->o            2   0.439   0.000  thtbunit_thcontrol_insertcase_xx_ffd31
(thtbunit_thcontrol_insertcase_xx_ffd3)
     LD:d                      0.370          thtbunit_thcontrol_out_thtb_writeentry_sig_0
    ----------------------------------------
    Total                      6.535ns (2.550ns logic, 3.985ns route)
                               (39.0% logic, 61.0% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock
'bucketunit_sortboxfifo_out_data_127__n00011:o'
Offset:              1.217ns (Levels of Logic = 1)
  Source:            bucketunit_sortboxfifo_thefifo
  Destination:       bucketunit_sortboxfifo_out_data_1_0
  Destination Clock: bucketunit_sortboxfifo_out_data_127__n00011:o falling

  Data Path: bucketunit_sortboxfifo_thefifo to bucketunit_sortboxfifo_out_data_1_0
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_fifo256ddr:dout<1>    1   0.000   0.408  bucketunit_sortboxfifo_thefifo
(bucketunit_sortboxfifo_fromfifo<1>)
     LUT4:i1->o            1   0.439   0.000  bucketunit_sortboxfifo__n0011<1>1
(bucketunit_sortboxfifo__n0011<1>)
     LD:d                      0.370          bucketunit_sortboxfifo_out_data_1_0
    ----------------------------------------
    Total                      1.217ns (0.809ns logic, 0.408ns route)
                               (66.5% logic, 33.5% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'thtbunit_thcontrol__n01231:o'
Offset:              5.027ns (Levels of Logic = 11)
  Source:            thtbunit_thetpt_blocktpt0
  Destination:       thtbunit_thcontrol_dontreadfromddr_0
  Destination Clock: thtbunit_thcontrol__n01231:o falling

  Data Path: thtbunit_thetpt_blocktpt0 to thtbunit_thcontrol_dontreadfromddr_0
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<1>    3   0.000   0.981  thtbunit_thetpt_blocktpt0
(thtbunit_thetpt__n0046<1>)
     LUT3:i1->o            5   0.439   1.119  thtbunit_thetpt_mmux_out_dataout_i16_result1
(thtbunit_in_tpt_table<1>)
     LUT4:i3->o            1   0.439   0.408  thtbunit_thcontrol_mcompar__n0082_inst_lut4_181
(thtbunit_thcontrol_mcompar__n0082_inst_lut4_18)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thcontrol_mcompar__n0082_inst_cy_753
(thtbunit_thcontrol_mcompar__n0082_inst_cy_753)
```

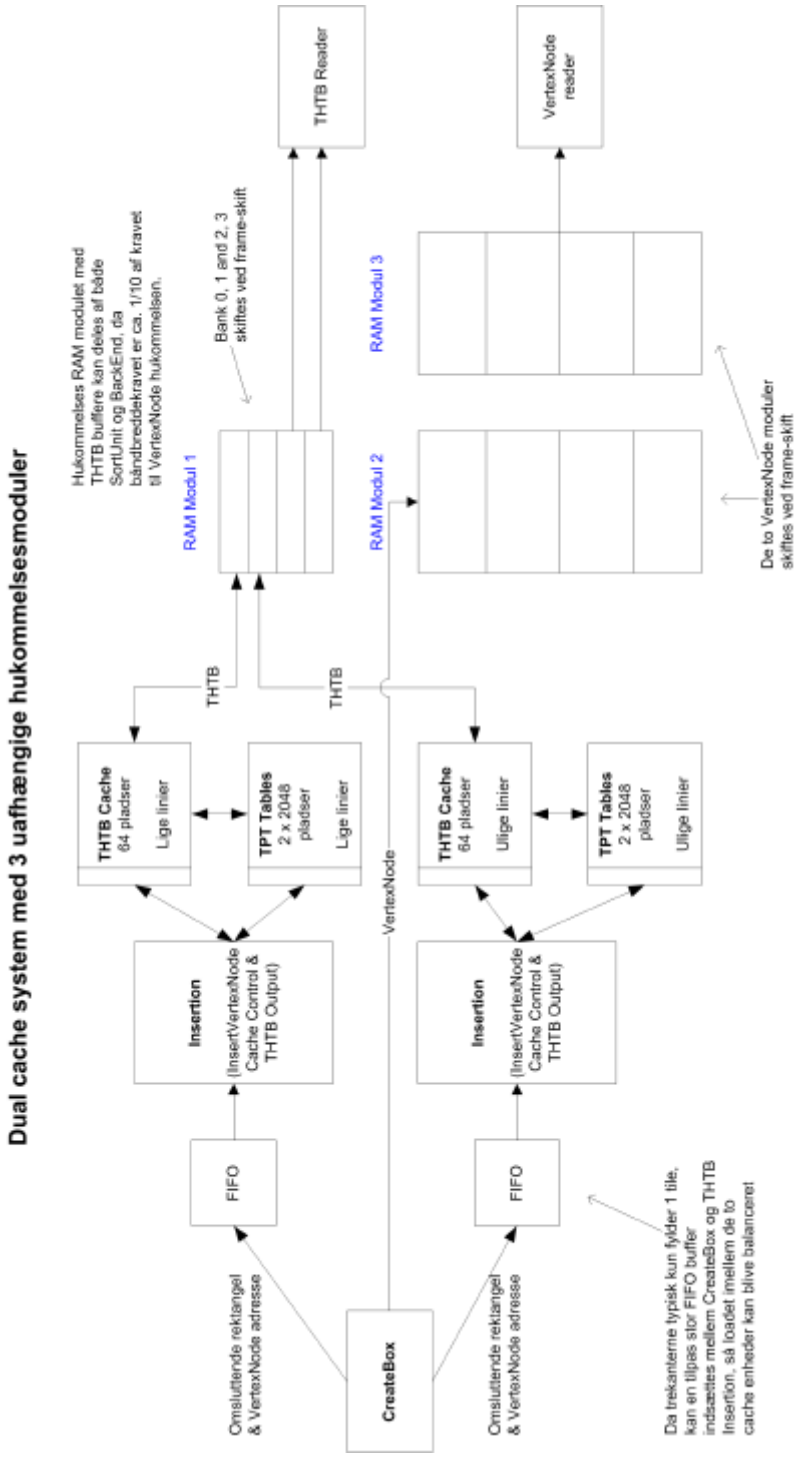
```
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thcontrol_mcompar__n0082_inst_cy_754
(thtbunit_thcontrol_mcompar__n0082_inst_cy_754)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thcontrol_mcompar__n0082_inst_cy_755
(thtbunit_thcontrol_mcompar__n0082_inst_cy_755)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thcontrol_mcompar__n0082_inst_cy_756
(thtbunit_thcontrol_mcompar__n0082_inst_cy_756)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thcontrol_mcompar__n0082_inst_cy_757
(thtbunit_thcontrol_mcompar__n0082_inst_cy_757)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thcontrol_mcompar__n0082_inst_cy_758
(thtbunit_thcontrol_mcompar__n0082_inst_cy_758)
     MUXCY:ci->o           1   0.053   0.000  thtbunit_thcontrol_mcompar__n0082_inst_cy_759
(thtbunit_thcontrol_mcompar__n0082_inst_cy_759)
     MUXCY:ci->o           1   0.053   0.408  thtbunit_thcontrol_mcompar__n0082_inst_cy_760
(thtbunit_thcontrol__n0082)
     LUT3:i2->o            1   0.439   0.000  thtbunit_thcontrol__n00631
(thtbunit_thcontrol__n0063)
     LD:d                      0.370          thtbunit_thcontrol_dontreadfromddr_0
    ----------------------------------------
    Total                      5.027ns (2.111ns logic, 2.916ns route)
                               (42.0% logic, 58.0% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET IN BEFORE for Clock 'thtbunit_thcontrol__n01241:o'
Offset:              7.551ns (Levels of Logic = 6)
  Source:            thtbunit_thetpt_blocktpt0
  Destination:       thtbunit_thcontrol_ram_req_0
  Destination Clock: thtbunit_thcontrol__n01241:o falling

  Data Path: thtbunit_thetpt_blocktpt0 to thtbunit_thcontrol_ram_req_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
    scm_blockram_tpt_table:douta<4>    2   0.000   0.790  thtbunit_thetpt_blocktpt0
(thtbunit_thetpt__n0046<4>)
     LUT3:i1->o            6   0.439   1.170  thtbunit_thetpt_mmux_out_dataout_i13_result1
(thtbunit_in_tpt_table<4>)
     LUT4:i0->o            1   0.439   0.408  thtbunit_thcontrol__n00464 (choice6487)
     LUT2:i0->o            2   0.439   0.790  thtbunit_thcontrol__n004610 (choice6491)
     LUT3:i0->o            3   0.439   0.981  thtbunit_thcontrol__n004639
(thtbunit_thcontrol__n0046)
     LUT3:i2->o            1   0.439   0.408  thtbunit_thcontrol_ker4196671
(thtbunit_thcontrol_n419669)
     LUT4:i2->o            1   0.439   0.000  thtbunit_thcontrol__n00641
(thtbunit_thcontrol__n0064)
     LD:d                      0.370          thtbunit_thcontrol_ram_req_0
    ----------------------------------------
    Total                      7.551ns (3.004ns logic, 4.547ns route)
                               (39.8% logic, 60.2% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'thtbunit_thethtb_output__n034538:o'
Offset:              7.640ns (Levels of Logic = 3)
  Source:            thtbunit_thethtb_output_ram_busy_0
  Destination:       thtbunit_thetpt_blocktpt1
  Source Clock:      thtbunit_thethtb_output__n034538:o falling

  Data Path: thtbunit_thethtb_output_ram_busy_0 to thtbunit_thetpt_blocktpt1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q             118   0.674   2.327  thtbunit_thethtb_output_ram_busy_0
(thtbunit_thethtb_output_ram_busy_0)
     LUT4:i2->o            4   0.439   1.069  thtbunit_thcontrol_ker4195561
(thtbunit_thcontrol_n419558)
     LUT4:i3->o           44   0.439   2.253  thtbunit_thcontrol_ker4195181
(thtbunit_thcontrol_n419520)
     LUT4:i0->o            0   0.439   0.000  thtbunit_thetpt__n00351 (thtbunit_thetpt__n0035)
    scm_blockram_tpt_table:wea        0.000          thtbunit_thetpt_blocktpt1
    ----------------------------------------
    Total                      7.640ns (1.991ns logic, 5.649ns route)
                               (26.1% logic, 73.9% route)

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

```
Timing constraint: Default OFFSET OUT AFTER for Clock 'thtbunit_thcontrol__n012529:o'
Offset:              1.903ns (Levels of Logic = 1)
  Source:            thtbunit_thcontrol_out_newtptheader_0_0
  Destination:       thtbunit_thetpt_blocktpt1
  Source Clock:      thtbunit_thcontrol__n012529:o falling

  Data Path: thtbunit_thcontrol_out_newtptheader_0_0 to thtbunit_thetpt_blocktpt1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               2   0.674   0.790  thtbunit_thcontrol_out_newtptheader_0_0
(thtbunit_thcontrol_out_newtptheader_0_0)
     LUT2:i1->o            0   0.439   0.000  thtbunit_thetpt_da1<0>1 (thtbunit_thetpt_da1<0>)
    scm_blockram_tpt_table:dina<0>        0.000          thtbunit_thetpt_blocktpt1
    ----------------------------------------
    Total                       1.903ns (1.113ns logic, 0.790ns route)
                                        (58.5% logic, 41.5% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'thtbunit_thethtb_output__n034644:o'
Offset:              5.022ns (Levels of Logic = 2)
  Source:            thtbunit_thethtb_output_ram_notready_0
  Destination:       thtbunit_thetpt_blocktpt1
  Source Clock:      thtbunit_thethtb_output__n034644:o falling

  Data Path: thtbunit_thethtb_output_ram_notready_0 to thtbunit_thetpt_blocktpt1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               7   0.674   1.219  thtbunit_thethtb_output_ram_notready_0
(thtbunit_thethtb_output_ram_notready_0)
     LUT3:i2->o           42   0.439   2.251  thtbunit_thcontrol_ker4196271
(thtbunit_thcontrol_n419629)
     LUT4:i2->o            0   0.439   0.000  thtbunit_thetpt__n00351 (thtbunit_thetpt__n0035)
    scm_blockram_tpt_table:wea        0.000          thtbunit_thetpt_blocktpt1
    ----------------------------------------
    Total                       5.022ns (1.552ns logic, 3.470ns route)
                                        (30.9% logic, 69.1% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'ddrinterface_busy:q'
Offset:              9.660ns (Levels of Logic = 2)
  Source:            ddrinterface_movedatamask_4_0
  Destination:       out_busy1
  Source Clock:      ddrinterface_busy:q rising

  Data Path: ddrinterface_movedatamask_4_0 to out_busy1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               9   0.674   1.321  ddrinterface_movedatamask_4_0
(ddrinterface_movedatamask_4_0)
     LUT2:i1->o            5   0.439   1.119  ddrinterface_busymask<4>1 (out_busy1_obuf)
     OBUF:i->o                 6.107          out_busy1_obuf (out_busy1)
    ----------------------------------------
    Total                       9.660ns (7.220ns logic, 2.440ns route)
                                        (74.7% logic, 25.3% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'clockhalf'
Offset:              11.248ns (Levels of Logic = 4)
  Source:            bucketunit_sortbox_thtb_fifo/bu211
  Destination:       out_saturated
  Source Clock:      clockhalf rising

  Data Path: bucketunit_sortbox_thtb_fifo/bu211 to out_saturated
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDPE:c->q             8   0.568   1.270  bu211 (full)
     end scope: 'bucketunit_sortbox_thtb_fifo'
     LUT2:i0->o            6   0.439   1.170  bucketunit_sortbox__n00871 (out_stall)
     LUT4:i0->o            1   0.439   0.408  masterunit_out_saturated5 (choice2324)
```

```
     LUT4:i0->o            1   0.439   0.408  masterunit_out_saturated18 (out_saturated_obuf)
     OBUF:i->o                 6.107          out_saturated_obuf (out_saturated)
    ----------------------------------------
    Total                     11.248ns (7.992ns logic, 3.256ns route)
                                       (71.1% logic, 28.9% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'clockfull'
Offset:              10.645ns (Levels of Logic = 6)
  Source:            thtbunit_thetpt_out_tptselect_sig
  Destination:       thtbunit_thetpt_blocktpt1
  Source Clock:      clockfull rising

  Data Path: thtbunit_thetpt_out_tptselect_sig to thtbunit_thetpt_blocktpt1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:c->q             17   0.568   1.753  thtbunit_thetpt_out_tptselect_sig
(thtbunit_thetpt_out_tptselect_sig)
     LUT3:i0->o            6   0.439   1.170  thtbunit_thetpt_mmux_out_dataout_i13_result1
(thtbunit_in_tpt_table<4>)
     LUT4:i0->o            1   0.439   0.408  thtbunit_thcontrol__n00464 (choice6487)
     LUT2:i0->o            2   0.439   0.790  thtbunit_thcontrol__n004610 (choice6491)
     LUT4:i0->o            4   0.439   1.069  thtbunit_thcontrol_ker4195561
(thtbunit_thcontrol_n419558)
     LUT4:i3->o           44   0.439   2.253  thtbunit_thcontrol_ker4195181
(thtbunit_thcontrol_n419520)
     LUT4:i0->o            0   0.439   0.000  thtbunit_thetpt__n00351 (thtbunit_thetpt__n0035)
    scm_blockram_tpt_table:wea        0.000          thtbunit_thetpt_blocktpt1
    ----------------------------------------
    Total                     10.645ns (3.202ns logic, 7.443ns route)
                                       (30.1% logic, 69.9% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'thtbunit_thcontrol__n01231:o'
Offset:              4.872ns (Levels of Logic = 2)
  Source:            thtbunit_thcontrol_dontreadfromddr_0
  Destination:       thtbunit_thetpt_blocktpt1
  Source Clock:      thtbunit_thcontrol__n01231:o falling

  Data Path: thtbunit_thcontrol_dontreadfromddr_0 to thtbunit_thetpt_blocktpt1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               4   0.674   1.069  thtbunit_thcontrol_dontreadfromddr_0
(thtbunit_thcontrol_dontreadfromddr_0)
     LUT3:i1->o           42   0.439   2.251  thtbunit_thcontrol_ker4196271
(thtbunit_thcontrol_n419629)
     LUT4:i2->o            0   0.439   0.000  thtbunit_thetpt__n00351 (thtbunit_thetpt__n0035)
    scm_blockram_tpt_table:wea        0.000          thtbunit_thetpt_blocktpt1
    ----------------------------------------
    Total                      4.872ns (1.552ns logic, 3.320ns route)
                                       (31.9% logic, 68.1% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'thtbunit_thcontrol_nextx_1__n00011:o'
Offset:              2.383ns (Levels of Logic = 1)
  Source:            thtbunit_thcontrol_nextx_2_0
  Destination:       thtbunit_thetpt_blocktpt1
  Source Clock:      thtbunit_thcontrol_nextx_1__n00011:o falling

  Data Path: thtbunit_thcontrol_nextx_2_0 to thtbunit_thetpt_blocktpt1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               8   0.674   1.270  thtbunit_thcontrol_nextx_2_0
(thtbunit_thcontrol_nextx_2_0)
     LUT3:i2->o            0   0.439   0.000  thtbunit_thetpt_mmux_ad1_i9_result1
(thtbunit_thetpt_ad1<2>)
    scm_blockram_tpt_table:addra<2>        0.000          thtbunit_thetpt_blocktpt1
    ----------------------------------------
    Total                      2.383ns (1.113ns logic, 1.270ns route)
                                       (46.7% logic, 53.3% route)
```

```
--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'thtbunit_thcontrol__n0121:o'
Offset:              2.383ns (Levels of Logic = 1)
  Source:            thtbunit_thcontrol_nexty_2_0
  Destination:       thtbunit_thetpt_blocktpt1
  Source Clock:      thtbunit_thcontrol__n0121:o falling

  Data Path: thtbunit_thcontrol_nexty_2_0 to thtbunit_thetpt_blocktpt1
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:g->q               8   0.674   1.270  thtbunit_thcontrol_nexty_2_0
(thtbunit_thcontrol_nexty_2_0)
     LUT3:i2->o            0   0.439   0.000  thtbunit_thetpt_mmux_ad1_i3_result1
(thtbunit_thetpt_ad1<8>)
    scm_blockram_tpt_table:addra<8>        0.000          thtbunit_thetpt_blocktpt1
    ----------------------------------------
    Total                      2.383ns (1.113ns logic, 1.270ns route)
                                       (46.7% logic, 53.3% route)

--------------------------------------------------------------------------------
Timing constraint: Default path analysis
Delay:               11.225ns (Levels of Logic = 5)
  Source:            in_legalpackage
  Destination:       out_saturated

  Data Path: in_legalpackage to out_saturated
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:i->o            10   0.825   1.370  in_legalpackage_ibuf (in_legalpackage_ibuf)
     LUT3:i2->o            2   0.439   0.790  masterunit_ker3528631 (masterunit_n352865)
     LUT4:i3->o            1   0.439   0.408  masterunit_out_saturated16 (choice2329)
     LUT4:i3->o            1   0.439   0.408  masterunit_out_saturated18 (out_saturated_obuf)
     OBUF:i->o                 6.107          out_saturated_obuf (out_saturated)
    ----------------------------------------
    Total                     11.225ns (8.249ns logic, 2.976ns route)
                                       (73.5% logic, 26.5% route)

==========================================================================
CPU : 806.96 / 807.47 s | Elapsed : 807.00 / 808.00 s

-->

Total memory usage is 937912 kilobytes
```

9.17 Bilag 17: Cache udvidelse



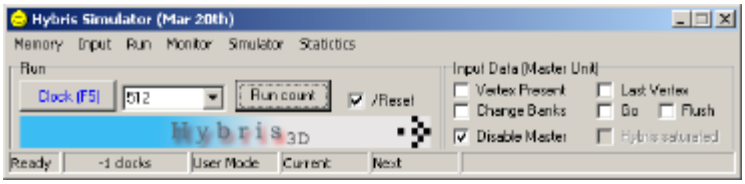
10 Appendiks A: Udvikling i SystemC

SystemC har været nemt at gå til og bruge i udviklingen af hardware renderen, det har dog vist sig at det også har sine ulemper.
Den klare gevinst er, at der kan udvikles testprogrammer i C++, som gør det nemmere at teste de udviklede hardware rutiner. For hardware renderen kunne der nemt sendes data ind i systemet og output kunne vises som både tal og trekanter.
Derimod har der været problemer med SystemC compileren, som desværre af og til genererede VHDL kode, som ikke kunne compiles; i de fleste tilfælde kunne VHDL koden rettes til, sådan at den kunne compile. Dette er dog ikke meningen, da det er uhensigtsmæssigt at VHDL koden skal rettes, til hver gang den er genereret. I de fleste tilfælde har det dog været muligt at tilrette SystemC koden, sådan at den blev compilet til korrekt VHDL kode.
Umiddelbart virker det lidt klodset at SystemC koden først skal laves til VHDL kode, men når SystemC compileren ikke er bedre, end den er, er det en fordel, da det giver mulighed for at gennemse den oversatte kode, der dog ikke er optimeret, men det giver udviklerne mulighed for at se, om det hardware, der blev implementeret i SystemC, blev oversat det, der var forventet, når koden blev compilet til VHDL.
Når det udviklede system skal testes, kan det godt anbefales at compile det med alle 3 compilere, da de afslører forskellige fejl. Først bruges en normal C++ compiler, som vil afsløre generelle fejl i C++-koden. Hovedparten af disse fejl, kan SystemC compileren også fange, men det er nemmere at finde ud af, hvor fejlen er opstået, da de gængse C++ compilere er langt foran mht. fejlbeskeder. Derimod er SystemC compilerens fejlbeskeder nærmest håbløse, da den kommer ud med en fejl og en linieangivelse, som det ikke er lykkedes at finde ud af, hvorfra den har, da den har ikke nogen direkte sammenhæng med den kode, der forsøges compileret. Men med lidt gætteri er det muligt at finde fejlen og rette den.
Fordelen ved SystemC compileren frem for den normale C++ compiler er at den kan fange hardware relaterede fejl, såsom signaler der mangler i sensitivity lister. Den finder også forskelle i bit vektor længder, f.eks. hvis det forsøges at sammenligne en bit vektor på 4 bit med en på 5 bit.
Til sidst compiles VHDL koden. VHDL compileren finder fejl, såsom signaler, der ikke ændrer sig. Den finder også signaler, som ikke anvendes eller tildeles en værdi, før de bruges. Nogle af disse fejl opstår desværre, når SystemC koden oversættes til VHDL, da der oprettes en hel del ekstra midlertidige variable. Her må fejlene sorteres efter reelle fejl og fejl der er opstået under kompileringen.
Når koden er oversat tilfredsstillende af alle compilere kan den egentlige test begynde. Startes testen før dette, ender det hurtigt med, at der skal bruges megen tid på at finde fejl, som en af compilerne kunne opdage.
Generelt er der mange fordele ved SystemC, men det har bestemt også sine bagdele. Til denne opgave har det været godt pga. de gode muligheder for at lave en ’testbench’ progammer i, som kunne vise trekanterne visuelt.

11 Appendiks B: Simulator brugervejledning

11.1 Hybris Simulator

Programmet startes ved at åbne programmet 'HybrisSim', hvorefter *Hybris Simulator* startes – se figur 42.



Figur 42: Hybris Simulator

I bunden ses diverse status information:

- Det fremgår, om simulatoren er optaget, eller klar til brug.
- Antallet af simulerede perioder.
- Om simulatoren kører med manuelt input (*User Mode*), eller i simulator mode (*Master Mode*).
- De felter, hvor der i figur 42, står 'Current' og 'Next' viser *Master Unit*'s tilstand. Det tomme felt vil vise forskellige statusinformationer.

Run:
Herfra køres systemet. **Clock** kører systemet i én periode, hvor der er tale om systemets hurtige frekvens.
Run Count kører i et specificeret antal perioder.
Hybris systemet kan resettes, når **/Reset** er lav.

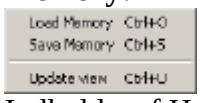
Input Master:
Herfra kan Hybris systemets *Master Unit* kobles ud, hvorefter *SortUnit* og *BackEnd* kan køre uafhængigt af hinanden. Er denne koblet ud, kan de tre styresignaler kontrolleres manuelt. **Vertex Present** og **Last Vertex** bruges til at signalere til Hybris systemet, at der er en datapakke klar. Vær opmærksom på at datapakker ignoreres af Hybris systemet, hvis systemet er mættet (**Hybris Saturated**).

Programmet er udviklet løbende i takt med behovet for simulering, og er derfor at betragte som et hjælpeværktøj. Stabiliteten og funktionaliteten svarer til, hvad der kan forventes af et tidligt beta-program. Der er nogle bugs i systemet, som ikke er rettet. Blandt andet "tabes" den første trekant i en simulering, hvis der benyttes dual-clock. Der er nogle variabelnavne, der optræder med navne, som ikke stemmer helt overens med Hybris systemets navne. Dette skyldes igen, at programmet er udviklet løbende og der optræder derfor midlertidige variabelnavne i nogle menuer og monitorer.

Systemkrav:
Microsoft Windows 2000/XP.
En simulation af Hybris systemet kræver ca. 150MB hukommelse.
Pentium Pro eller nyere (Pentium 4, Athlon XP eller nyere CPU anbefales).
Memory dump kræver 128MB plads, og screen dump kræver 16MB.

11.1.1 Beskrivelse af menuer og vinduer:

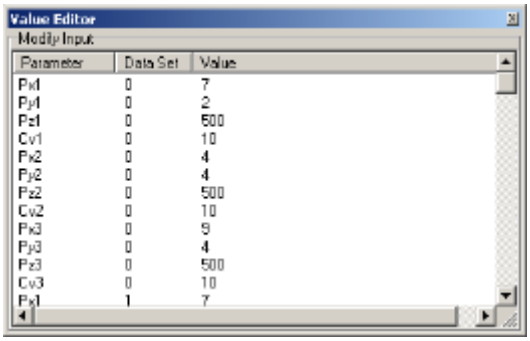
Memory:



Indholdet af Hybris systemets hukommelse og cache kan gemmes eller hentes vha. denne menu.

Input:

I denne menu, kan **Value Editor** vinduet fremkaldes. Det kan bruges til at redigere i, hvilke data pakker, der afleveres til Hybris systemet. Ved at højre klikke i input-vinduet, kan data redigeres (Se figur 43). Der kan ikke tilføjes nye data. Dette skal gøres i data-filen (C:\HybrisData.txt). Vær opmærksom på, at kun hele pixel og farveværdier kan benyttes her. Der tilføjes automatisk 12 decimaler i 'testbench' modulet.



Figur 43: Input data editor.

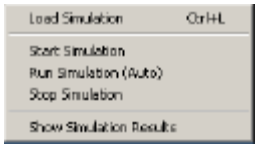
Run:

I denne menu gives mulighed for at køre systemet. Foruden **Run** og **Run Count** kan systemet også køres i nul perioder. Det giver mulighed for at se, hvordan systemet eventuelt vil opføre sig, hvis et input signal ændres.
Double Clock benyttes til at simulere systemet med en samlet system-clock, eller med 2, som det er tiltænkt.

Monitor:

I denne menu gives der mulighed for at fremkalde alle monitor vinduer, eller kun dem, der har med *SortUnit* eller *BackEnd* at gøre.

Simulator:

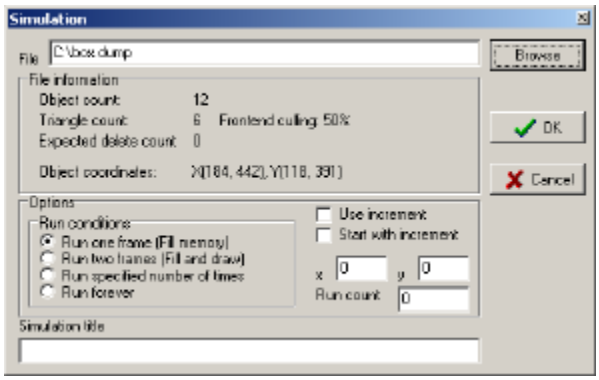


Simulatoren kan startes herfra. Først skal en simulator fil åbnes.
Der vises et dialog vindue, som vist på figur 44. Her kan en simuleringsfil vælges. Det vil fremgå, hvor mange trekanter, der er i filen (**Triangle Count**), og hvor mange trekanter, det kan forventes at *CreateBox* skal fjerne.
Forskellige simuleringsmodeller kan vælges:

- Run one frame (Fill memory). Simulerer en indlæsning af objektet. Kan benyttes til at teste *SortUnit*
- Run two frames (Fill and draw). Simulerer en indlæsning og tegning af objektet. Tester både *SortUnit* og *BackEnd*
- Run specified number of times. Indlæser objektet til Hybris et defineret antal af gange. Antallet bestemmes af tallet i **Run Count**.
- Run forever. Kører simuleringen, indtil den afbrydes manuelt.

Foruden kørselsbetingelserne, kan objektet flyttes, vha. **Use increment.** x og y bestemmer, hvor meget objektet skal flyttes for hver indlæsning. Forskydningen skal angives i fixed-point format, dvs. at objektet skubbes 1 pixel, hvis 4096 angives.
Ønskes det, at starte med en forskydning kan det vælges med **Start with increment**.

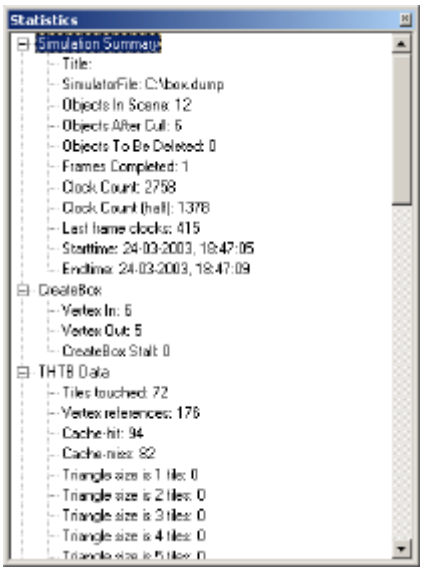
Når simuleringen er indlæst, kan den startes med **Start Simulation** eller **Run Simulation (Auto)** i simulator-menuen. Sidstnævnte kører simuleringen automatisk, mens den første kræver manuel kørsel vha. **Clock** eller **Run count**. Når simuleringen startes tilkobles Hybris systemets *Master Unit*, og statistik værdierne nulstilles.
Efter en simulering gemmes statistik resultaterne, og kan ses vha. **Show Simulation Results**.



Figur 44: Simulator vinduet

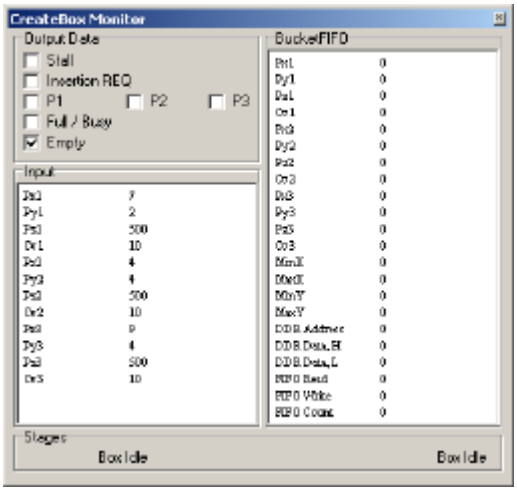
Statictics:

Statistik vinduet kan fremkaldes, vha. denne menu. Desuden kan statistik tællerne nulstilles i denne menu. På figur 45 ses statistik vinduet, hvor opsamlet data vises. Data kan gemmes ved at højre klikke.

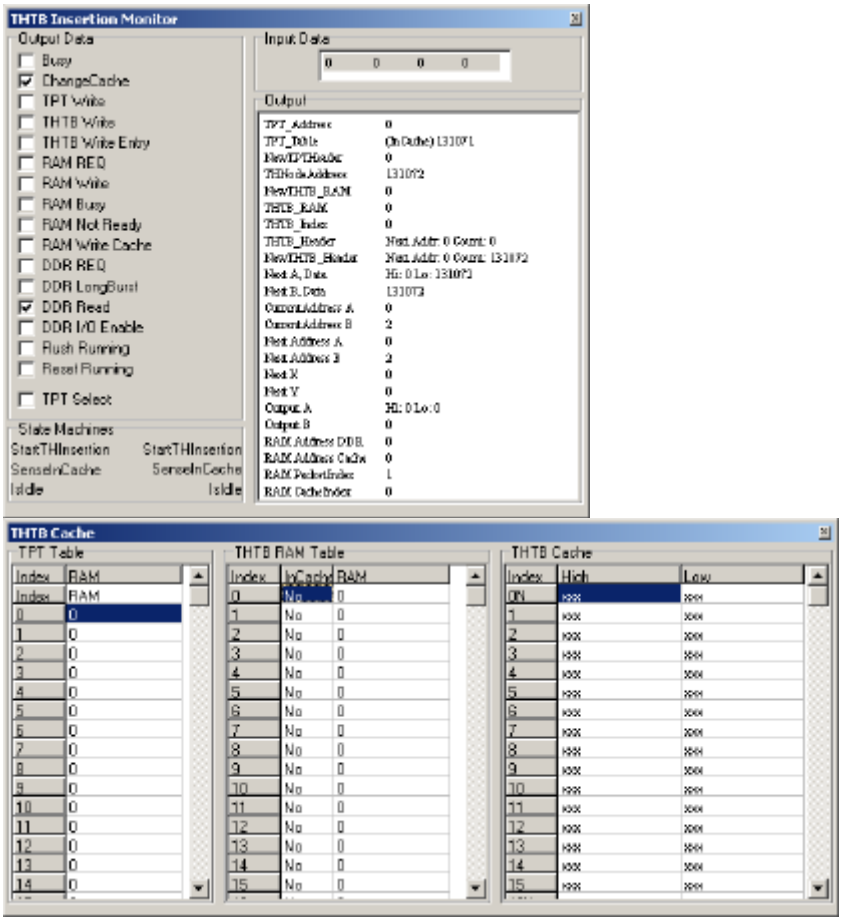


Figur 45: Statistik opsamling

Monitorer:

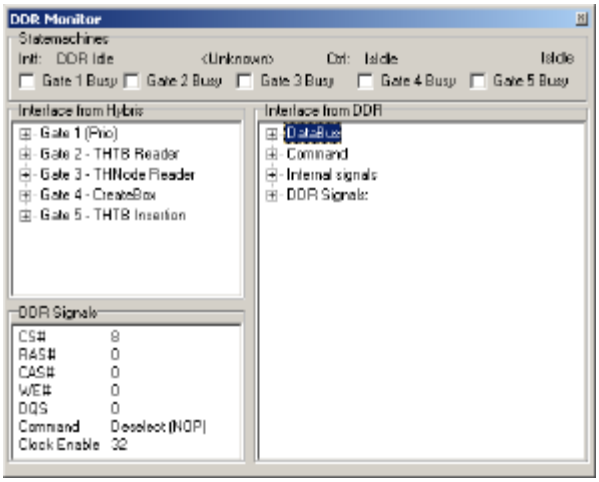


CreateBox viser information fra *CreateBox* og *BucketFIFO*. **Input** vinduet viser den indgående data pakke, mens **Output** vinduet viser de data *CreateBox* afleverer.
Output Data vinduet viser udvalgte status signaler.
Nederst kan *BucketFIFO*'s tilstand ses.

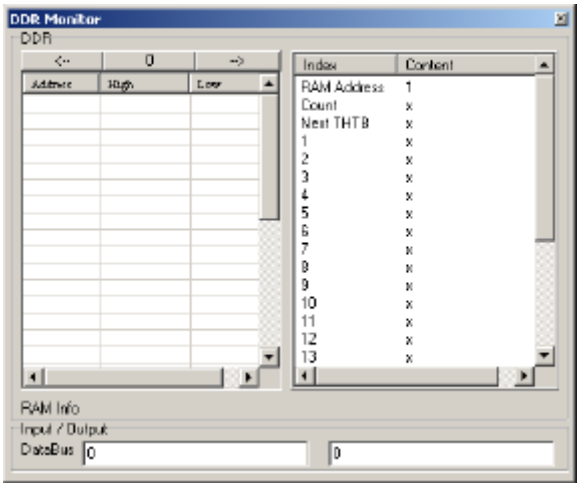


THTB Insertion Monitor og **THTB Cache** viser information fra *THTB Insertion* enheden.
Output Data viser udvalgte statussignaler, og **Output** viser udvalgte variable.
I **Input Data** ses det omsluttende rektangel, som *THTB Insertion* arbejder med.

I **State Machines** vises de tre parallelt kørende processers tilstande. Fra toppen og nedefter vises tilstandene fra hhv. *InsertVertexNode*, *Cache Control* og *THTB Output*.
I cache monitoren, ses indholdet af cache systemet, inkl. de to tilhørende tabeller. Indholdet opdateres løbende, men kan opdateres manuelt, ved at højreklikke i vinduet.

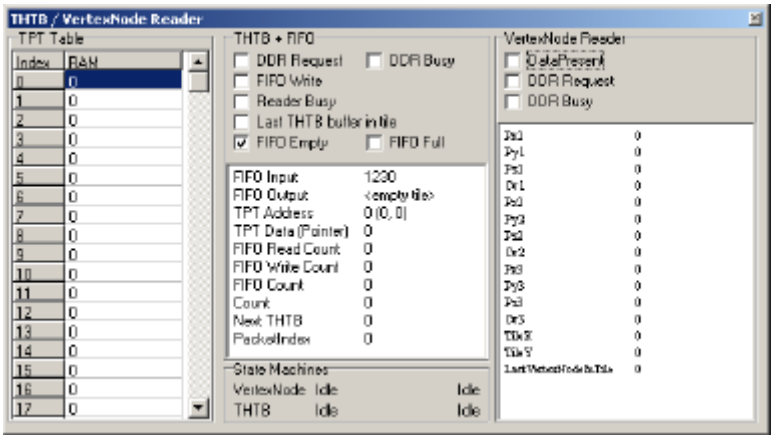


Denne monitor viser udvalgte data fra DDR-SDRAM controller og interface.
De to tilstandsmaskiner for det udviklede interface ind imod Hybris modulerne er placeret til venstre, mens tilstandsmaskinen for DDR-SDRAM controlleren (Xilinx controlleren/DDR-SDRAM simulator) ses til højre.
I **Interface from Hybris** kan DDR-SDRAM interfacet ind imod de interne enheder overvåges. Port 1 optræder, men bliver ikke benyttet af Hybris systemet.
I **Interface from DDR** kan DDR-SDRAM controlleren overvåges, og i **DDR Signals** ses den aktuelle kommando til den eksterne hukommelse.

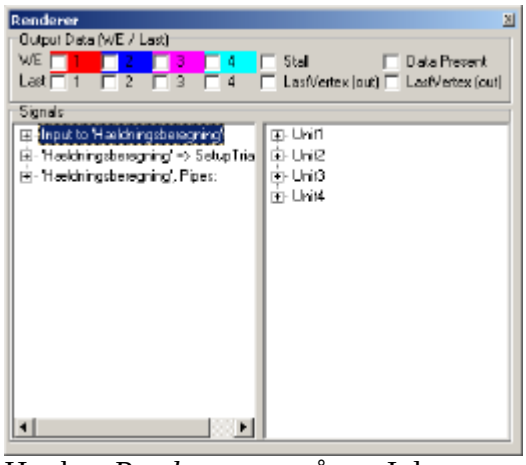


I denne monitor kan alle 128MB hukommelse overvåges. I panelet til venstre ses indholdet af hukommelsen. **High** og **Low** viser 64 bit tal.
Adressen kan vælges ved at indtaste ønsket adresse i øverste venstre felt, og adressen aktiveres ved at trykkke på det felt, hvor der står '0'. Ved at højre klikke, kan der springes direkte til enten THTB buffer eller VertexNode områderne, i alle 4 hukommelses banker.
Panelet til højre oversætter hukommelsesindholdet til enten en VertexNode pakke eller en THTB buffer, afhængigt af, hvor hukommelsen overvåges.

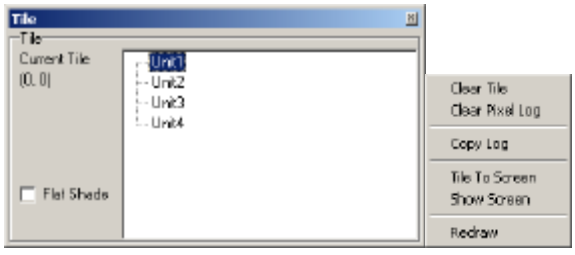
Nederst ses det data sæt, som vil blive overført fra eller til hukommelsen i hhv. første og anden del af en periode (2 x 64 bit).



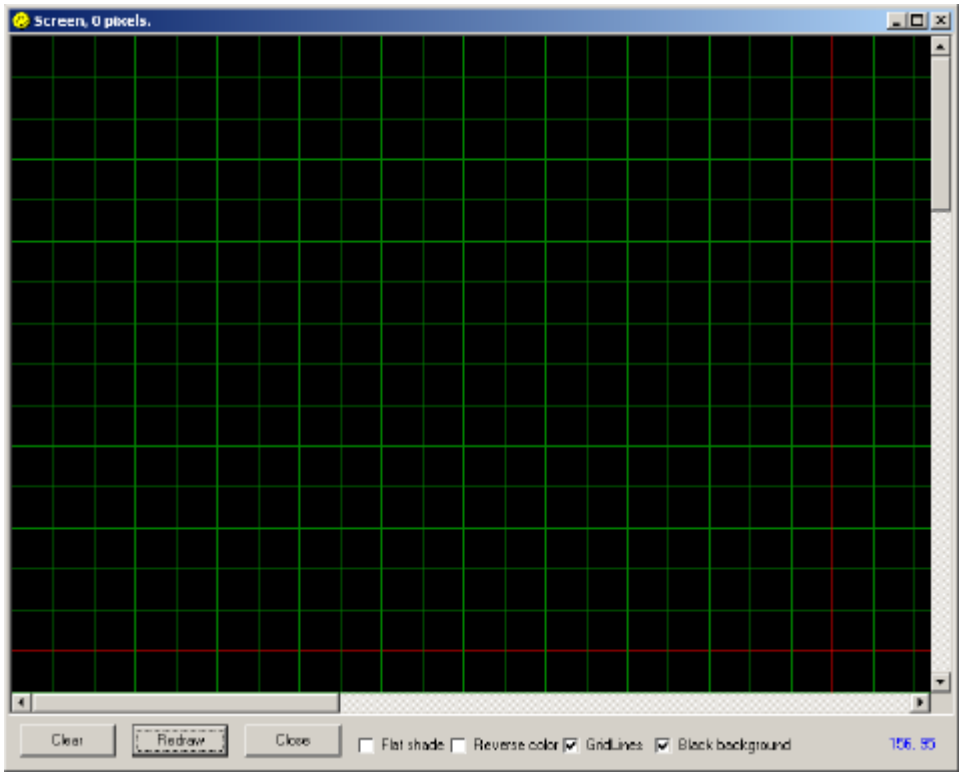
Denne monitor viser udvalgte data fra *BackEnd*'s data indlæsningsmoduler.
Til venstre ses den TPT tabel, hvorfra første THTB buffer i en tile kan findes.
De to indlæsningsmodulers tilstandsmaskiner tilstande vises. Panelet til højre viser, den VertexNode pakke der er indlæst.



Her kan *Renderer* overvåges. I det venstre panel kan det ses, hvilke data, der indlæses til *Hældningsberegning* enheden (**Input to Renderer)**; den datastruktur, som overføres til *Tilskæring af y-retning* (**Create VertexNode => SetupTriangle),** og hvor i *Hældningsberegning* enheden, der er data.
Til højre kan nogle af *Renderer* enhedens interne variable og FIFO buffere overvåges.



Den aktuelle tile vises i denne monitor. Til højre kan det overvåges, hvad *Renderer* afleverer, og til venstre vil tilen blive tegnet. Vælges **Flat Shade** tegnes der i sort/hvid, ellers benyttes gråtoner, da farveværdien fra *Renderer* benyttes til alle tre farver (rød, grøn, blå).
Ved at højre klikke ses den menu, der vises til højre for tile monitoren.
Her kan tilen slettes (det sker automatisk, når *ChangeTile* flaget fra *Renderer* modtages[42]), og den kan overføres til "skærmen". Skærmen ses ved at vælge **Show Screen**, hvorefter vinduet på figur 46 ses.
Skærmen kan gengive den fulde opløsning på 2048x2048. De røde streger indikerer den opløsning, som systemet arbejder med (640x480). Nederst til højre kan det ses, hvor musemarkøren befinder sig, og holdes musen stille vil en tekst fremkomme, som fortæller, hvilken tile musen er over, og hvor mange pixels, der er i denne tile.
Billedet kan gemmes ved at højreklikke, og kan gemmes i et format, som Windows understøtter.



Figur 46: Skærm

Data fil

Ved opstart af simulator programmet vil programmet forsøge at indlæse en tekst fil med input data. Denne består af linier med data, der skal laves således:

Variabelnavn = tal

Der differentieres ikke mellem store og små bogstaver, og ukendte variabelnavne ignoreres.

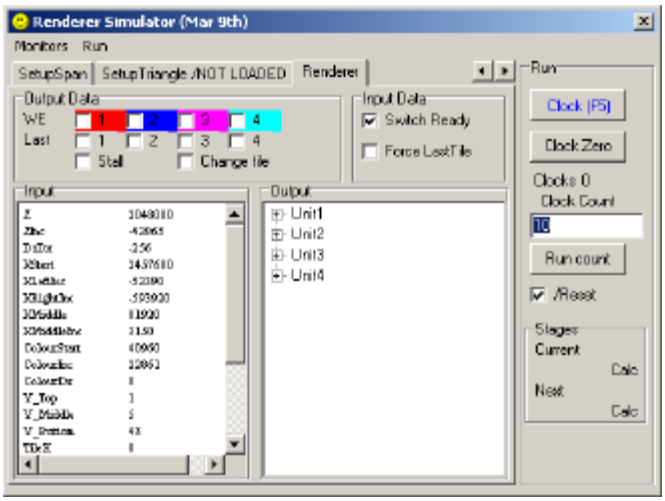
[42] Simulatoren forventer at Hybris systemet arbejder med en opløsning på 640x480.

Udover variabelnavne kan der indlægges styrekoder, som kan benyttes efter behov.
Disse er:

- [Initial]. Alle variabler efter denne kode vil automatisk blive brugt som input værdi til Hybris systemet.
- [/Initial]. Alle variable efter denne kode, vil ikke blive brugt som input værdier.
- [Data]. Bruges til at adskille data sæt. Hver gang denne kode optræder, vil **Value Editor** give variablen et nyt data sæt nummer.

I starten af tekstfilen skal der stå identifikator, så simulatoren kan vide sig sikker på, at det er en korrekt data fil. Denne skal være [HYBRIS_DATA] for *Hybris Simulator* og [SETUPTRIANGLE] for *Renderer Simulator*.
På den vedlagte cd ligger to datafiler, der viser eksempler på datafiler.

11.2 Renderer Simulator:



Figur 47: Renderer Simulator

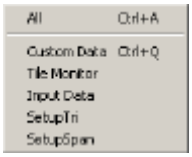
Renderer Simulator kan bruges til at overvåge Hybris systemets *Renderer*. Systemkravene til denne simulator er langt mindre krævende end den komplette simulator. Der kræves omkring 20MB hukommelse til at køre programmet, og det kan afvikles på alle maskiner med en Pentium Pro processor eller nyere.

Når programmet startes, ses flere faneblade. I den endelige version er det kun fanebladet **Renderer**, der kan benyttes, og kan ses af figur 47.
Da *Hybris Simulator* og *Renderer Simulator* bygger på samme platform er megen af funktionaliten den samme, hvorfor denne ikke gennemgås her, med mindre der er forskelle.
Renderer Simulator benytter filen C:\SetupTriangle.txt som datafil.

I **Input Data** kan input flagene til *Renderer* manuelt sættes. **Force LastTile** vil fortælle *Renderer*, at den modtager den sidste trekant i en tile, og **Switch Ready** indikerer, om tile bufferen er parat til at modtage data.

11.2.1 Menuer og vinduer

Monitors:

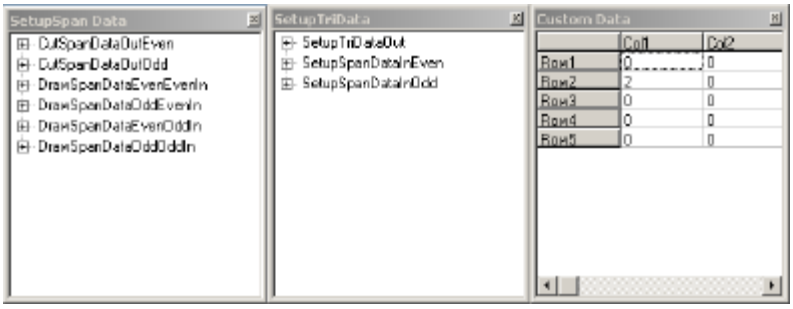


De forskellige monitorer i *Renderer Simulator* kan kaldes frem i denne menu.
Tile Monitor og **Input Data** er de samme som i *Hybris Simulator*, mens de tre sidste fremgår af figur 48

Run:

Herfra kan systemet køres.

Vinduer:



Figur 48: Monitorer i *Renderer Simulator*

Der er to data monitorer, der viser udvalgte data. Disse er **SetupSpan Data** og **SetupTri Data**, og de overvåger udvalgte data, der løber imellem de interne enheder i *Renderer*.

Custom Data kan bruges til at vise data, der ikke er nærmere defineret. En datastruktur med plads til 1024 heltal kan mappes ind i **Custom Data**. Data[0] svarer til (Col1, Row1) og Data[1] svarer til (Col1, Row2) osv. Antallet af rækker og kolonner angives i data filen, hvor de også skal navngives. Data kan bruges til både input og outputsignaler, og kan bindes efter behov i ’testbench’ delen.