FPGA System Spesifikasjon

Contents

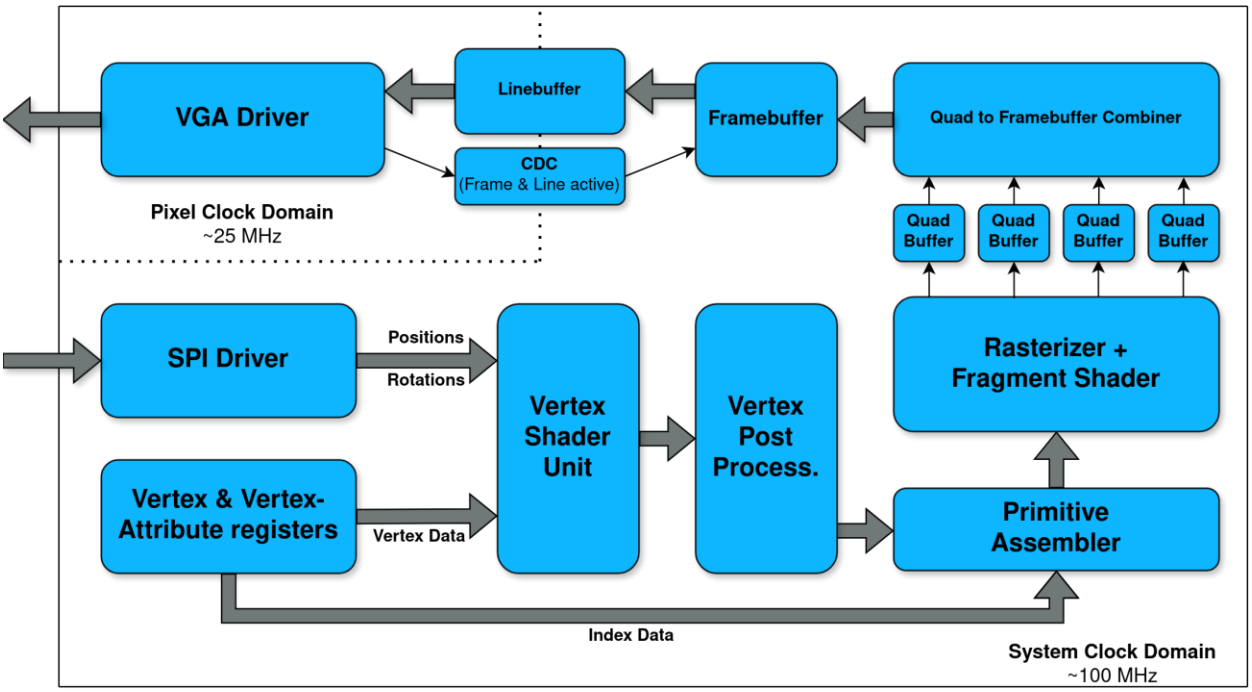
Systemoversikt 2
FPGA-MCU Kommunikasjon 3
SPI Driver 3
Dataprotokoll 3
Framebuffer 3
VGA-Driver 4
VGA signaler 4
Render pipeline 4
Vertex og vertex-attributt lagring 4
Vertex shader stage 4
Matrise-matrise multiplikator 4
Matrise-vektor multiplikator 5
Vertex post processor 5
Primitive assembler 5
Rasterizer + Fragment shader 5
Barycentric coordinate calculation 5

Systemoversikt

Systemet består hovedsakelig av fire deler:

1. Kommunikasjon mellom FPGA-en og MCU-en (SPI Driver)
2. Framebufferen
3. VGA Driveren
4. Render pipelinen

Følgende er et diagram som viser systemet, delsystemene og kommunikasjonen mellom dem.



Systemer har to klokker: en for VGA driveren og en for systemet. Hvert klokkesignal er drevet fra en felles 100 MHz klokke, og benytter to av FPGA-ens 6 MMCM enheter for å generere de deriverte klokkesignalene. De forskjellige klokkedomenene er adskilt med Dual FF Synchronizers.

FPGA-MCU Kommunikasjon

SPI Driver

Dataprotokoll

Dataprotokollen brukt for å kommunisere data mellom MCU-en og FPGA-en er vist i figuren under. For hvert dataframe blir DATA_FRAME_NEW satt lav, og en byte blir sendt om gangen over SPI bussen.

[[FIGUR SOM VISER DATAPROTOKOLL]]

Første byte representerer antall entities som skal rendres til skjermen, følgelig kalt NUM_ENTETIES. Følgelige 3 bytes representerer kameraets yaw og pitch vinkler. De 3 bytene er splittet inn i to 12-bit fixed-point tall på formen Q1.11, der MSB er et sign bit. Det betyr at hver vinkel representeres av et tall i mengden [−1,0.99951171875], som korresponderer direkte til mengden [-π, π]. Så følger NUM_ENTITIES gjenntagelser av følgelige enkoding:

1. – 8. byte: Entity posisjon i x, y, z, hver med 18 bits på formen Q7.11 + 10 bits for forskjellige entity flags; flag posisjonene er de 10 mest signifikante bitsene.

9. – 11. byte: Entity rotasjon som pitch, yaw, roll, hver representert på formen Q1.7.

Flag-bitene kan dekodes som følger:

[[FIGUR SOM VISER DEKODING AV FLAG BITS]]

Framebuffer

Framebufferen er delen av systemet som skal lagre pixel verdiene som skal skrives til skjermen. Det markerer slutten på Render Pipelinen. Framebufferen til systemet skal ta i bruk en dobbel framebuffer. Dette gjør det mulig å skrive til en buffer og samtidig lese fra et tidligere frame sin framebuffer. Dette er ideelt da endelig oppnådd framerate til systemet er ukjent. Intern struktur til Framebufferen er vist i figuren under.

[[FIGUR SOM VISER FRAMEBUFFER SYSTEMET]]

Framebufferen skal være dual-ported, slik at det er mulig å lese fra og skrive til bufferen simultant. En intern state velger hvilket av de to interne buffer-ene som skal leses fra og skrives til. Staten endres når VGA Driveren signaliserer at den starter et nytt frame, altså når *VGA_Frame_o* signalet går høyt, men kun når signalet fra render pipelinen som forteller at current frame er ferdig rendret, signalisert med signalet *Render_Pipeline_done_o*.

Hver buffer er satt opp som arrays av 6-bit tall med dybde lik 640x480. Dette passer direkte med display oppløsningen vi ønsker å bruke. Dekodingen av de 6-bitene er vist under.

[[FIGUR FOR MAPPING FOR PIXEL BITS TIL BETYDNING]]

De to mest signifikante bit-ene representerer luminansen til pixel-ene. Altså har hver pixel 4 forskjellige luminansverdier: fullstendig belyst, trekvart belyst, halvt belyst og en kvart belyst. Mappingen følger følgende bit-verdier.

[[FIGUR FOR MAPPING FRA BIT-VERDI TIL LUMINANS]]

Argumentet for å ikke ha en verdi for ikke belyst er fordi vi da får ambient belysning gratis, da alt som ikke blir direkte belyst får en liten belysning på omtrent en kvart av full belysning.

De minst signifikante bit-ene representerer så en index inn i et 12-bit farge-lookup-table. Dette lar oss bruke arbitrære 12-bit farger uten å måtte ta høyde for en dobbelt så stor framebuffer.

VGA-Driver

VGA signaler

Render pipeline

Vertex og vertex-attributt lagring

Vertex data og vertex attributt data lagres i BRAM, da extern DDR ram ikke er tilgjengelig.

Vertex shader stage

Databredden brukt for å representere posisjonen til en vertex er 18 bit, på formen Q8.10, men hver del av kretsen, eksempelvis en matrise matrise multiplikator, vil lagre intermediate verdier som dobbel databredde, altså Q16.20.

Matrise-matrise multiplikator

Modulen tar inn to 4x4 matriser *A* og *B*, hver med databredde på 18 bit fixed-point tall, et input data valid signal *i_dv*, samt klokke *clk* og reset signal *rstn*, og sender ut en 4x4 matrise *C* også med 18-bit fixed-point tall, et output data valid signal *o_dv* og et ready signal *o_ready*.

Matrise-vektor multiplikator

Modulen tar inn en 4x4 matriser *A* og en 4-dimensjonal vektor *x*, hver med databredde på 18 bit fixed-point tall, et input data valid signal *i_dv*, samt klokke *clk* og reset signal *rstn*, og sender ut en 4-dimensjonal vektor *y* også med 18-bit fixed-point tall og et output data valid signal *o_dv*.

Vertex post processor

Primitive assembler

Rasterizer + Fragment shader

Barycentric coordinate calculation

Gitt de tre punktene som definerer en trekant v_1, v_2, v_3 og et punkt p, kan det barysentriske koordinatet λ finnes ved følgende algoritme

$$v_{10} = v_1 - v_0, v_{21} = v_2 - v_1, v_{p0} = p - v_0$$

$$d_{00} = v_{10} * v_{10}, d_{01} = v_{10} * v_{21}, d_{11} = v_{21} * v_{21}, d_{20} = v_{p0} * v_{10}, d_{21} = v_{p0} * v_{21}$$

$$det = (d_{00} * d_{11} - d_{01} * d_{01})$$

$$\lambda_y = \frac{(d_{11} * d_{20} - d_{01} * d_{21})}{det}$$

$$\lambda_z = \frac{d_{00} * d_{21} - d_{01} * d_{20}}{det}$$

$$\lambda_x = 1 - \lambda_y - \lambda_z$$

[[TODO: Make documentation in latex so I don't have to use Word equations]]