Designspecifikation 4ss-troids

TSEA83, Grupp 34

Anton Wegeström - antwe841
Anton Östman - antos931
Jakob Österberg - jakos322
Rasmus Wallin – raswa151

Version 1.2

2022-03-04

Innehållsförteckning

Sammanfattning

I denna designspecifikation beskrivs en 2-stegs pipeline-processor med nödvändig kringutrustning. Tanken är att man ska kunna köra spelet asteroids på processorn där användaren kontrollerar ett rymdskepp via en joystick och spelplanen visas på en skärm via VGA. Senare i projektet ska UART inladdning av program läggas till, samt om tid finns så ska ljud implementeras.

Processorn

Processorn använder både ett dataminne och ett programminne. Dess huvudsakliga uppgift är att räkna ut vart sprites befinner sig och hantera I/O mellan spelaren och spelet. Registerfilen består av 16st register som vardera är 16 bitar breda, vilket också blir processorns naturliga ordbredd.

Instruktionsset

Mnemonic	**Funktion**
NOP	No operation
RJMP *offset*	Hopp till PC+1+*offset*
BEQ *offset*	Hopp om Z = 1
BNE *offset*	Hopp om Z = 0
BPL *offset*	Hopp om N = 0
BMI *offset*	Hopp om N = 1
BGE *offset*	Hopp om N xor V = 0
BLT *offset*	Hopp om N xor V = 1
LDI *Rd,Konst*	RD <= konst
LD *Rd,Ra*	RD <= MEM(Ra)
STI *Rd,Konst*	MEM(Rd) <= Konst
ST *Rd,Ra*	MEM(Rd) <= Ra
COPY *Rd,Ra*	Rd <= Ra
ADD *Rd,Ra*	Rd <= Rd + Ra, uppd. Z, N, C, V
ADDI *Rd,Konst*	RD <= Rd + konst, uppd. Z, N, C, V
SUB Rd,Ra	Rd <= Rd - Ra, uppd. Z, N, C, V
SUBI Rd,Konst	Rd <= Rd – Konst, uppd. Z, N, C, V
CMP *Rd,Ra*	Rd-Ra, uppd. Z, N, C, V
CMPI *Rd,Konst*	Rd-konst, uppd. Z, N, C, V
AND Rd,Ra	Rd <= Rd AND Ra, uppd. Z, N, C, V
ANDI *Rd,Konst*	Rd <= Rd AND konst, uppd. Z, N
OR *Rd,Ra*	Rd <= Rd OR Ra, uppd. Z, N, C, V
ORI *Rd,Konst*	RD <= Rd OR konst, uppd. Z, N, C, V
PUSH *Rd*	DM(SP) <= Rd, SP++
POP Rd	Rd <= DM(SP+1), SP--
ADC *Rd,Ra*	Rd <= Rd + Ra + C

SBC *Rd,Ra*	Rd <= Rd – Ra – C
MUL *Rd,Ra*	Rd <= Rd * Ra (unsigned)
MULI Rd,Konst	Rd <= Rd * Konst (unsigned)
MULS *Rd,Ra*	Rd <= Rd * Ra (signed)
MULSI Rd,Konst	Rd <= Rd * Konst (signed)
LSLS *Rd,Ra*	Rd <= Rd << Ra uppd. N, Z, C
LSRS *Rd,Ra*	Rd <= Rd >> Ra uppd. N, Z, C

Eftersom vi vill ha plats med åtminstone 32st instruktioner, och vi kan tänkas behöva lägga till fler, används 6 bitar för att identifiera vilken instruktion som ska utföras. Instruktionsregistret delas upp enligt nedan beroende på typen av instruktion (se tabell ovan) och är 26 bitar brett, för att få plats med en 16 bitars konstant.

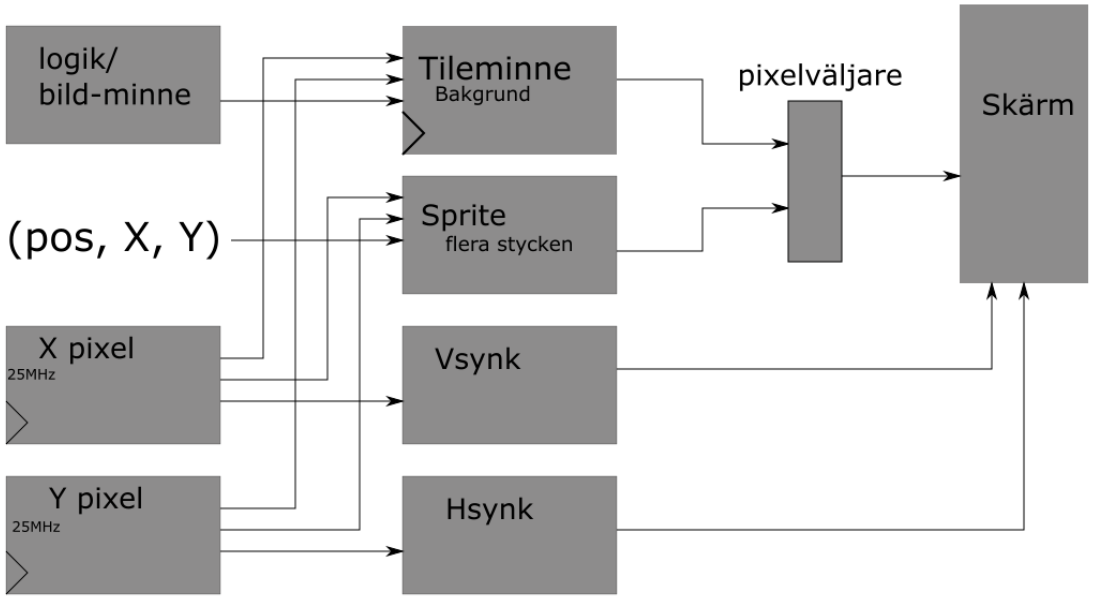
```
   Op.      Rd     Ra       Oanvänt
[000000][0000][0000][00000000000]
   Op.      Rd              Konstant
[000000][0000][0000000000000000]
```

Start av processor

Till en början har vi ett program laddat från start i programminnet. Vi vill senare i projektet använda oss av en boot-loader för att kunna ladda in program via UART. Detta blir en separat enhet från CPU:n kopplad till programminnet som bara hanterar inladdning av ett program.

Grafik

Vi har valt att ha upplösningen 640x480 med 4 bitar till färger. Vi har tänkt använda oss av sprites för spelobjekt och tiles för bakgrunden. Ett abstrakt blockschema ses i figur 1 nedan.



Figur 1: Blockschema över grafikdelen.

Själva designen för sprites och tiles kommer finnas i två separata hårdkodade ROM minnen. Dessa måste inte förändras under exekvering, det är bara positioneringen av sprites och vilka sprites som finns i spelet som kommer förändras, vilket kommer gå att nå via minnesmappning av primärminnet, se figur 2 längst ner. Animering av bakgrund sker genom att vi byter ut vilka tiles som finns vart på skärmen. Detta kan också vara hårdkodat och måste inte nås av CPU:n då tilesen inte ändras under spelets gång, och animeringen kan fortgå från start till slut.

Pixelväljaren kommer att välja om det är bakgrunden som kommer skrivas till skärmen eller en sprite. Den kommer alltid föredra att skriva ut en sprite. Vi kommer använda värdet ‘0’ för NULL, vilket innebär att det inte finns någon färg på positionen för spriten.

I grafikminnen finns färgerna som ska skrivas ut, vilket skickas till skärmen av pixelväljaren.

I/O

Vi kommer använda en Joystick (Pmod JSTK) för att bestämma spelarens position. Kommunikation sker över SPI och vi måste koppla FPGA:n mot joystickens MOSI/MISA/SCK/CS pins. UART kommer användas för att ladda över program till FPGA:n i ett senare skede, detaljer för kopplingen till FPGA:n tar vi från labb 3. Detaljer för VGA tas från labb 4 (samt se grafikdelen ovan). En stereohögtalare kommer användas för att spela ljud relaterat till spelet, implementeras om vi hinner med övriga delar av projektet. Kommunikationen sker då över I2S protokollet på pins kopplade mellan högtalaren och FPGA:n. I/O-enheterna kopplas till de högsta registren i CPU:ns registerfil.

Minnesanvändning

Till vår konstruktion vill vi ha ett bildminne, ett tileminne, fem olika sprites, programminne och dataminne. Tileminnet adresseras med 13 bitar vilket ger plats för 32st 8*8px stora tiles. Bredden på tileminnet är 4 bitar som representerar varje pixels färg. Det tillhörande bildminnet adresseras också med 13 bitar, där vi använder 4800 adresser för att peka ut tiles i tileminnet, vilket gör att minnet har bredden 13 bitar.

Våra fem olika sprites ska representera tre olika storlekar av asteroider, spelaren och skott från spelaren. Asteroiderna är dels den största som är 64*64px och adresseras med 10 bitar, dels den mellersta som är 32*32px och adresseras med 8 bitar och dels en minsta som är 16*16px och adresseras med 6 bitar. Spelaren är 32*32px och adresseras med 8 bitar och ett skott från spelaren är 8*8px och adresseras med 4 bitar.

Program- och data-minnet adresseras båda med 16 bitar men programminnet är 26 bitar brett (likt en instruktion) medan dataminnet är 16 bitar. Därmed sätter vi alltså PC till 16 bitar. Vi måste också ha ett statusregister som behöver rymma fem flaggor, detta register är 5 bitar brett. Om vi mot förmodan får slut på programminne kan vi använda två register för att adressera ett 32 bitars minne. De minnen som CPU:n skriver eller läser ifrån är: PC, dataminne, programminne och registerfilen. Stackpekaren är ett eget register som bara går att ändra via pop/push instruktioner, en startposition hårdkodas. Stacken återfinns i dataminnet.

Vi använder minnesmappning för att komma åt minnet där vi sparar sprites position. Samtliga sprites skickas sedan till *VGA motorn* för att målas ut på skärmen. Se figur 2 under ***Blockschema för designen*** för en tydligare skiss.

Programmering

Vi programmerar för hand till en början, det vill säga i maskinkod. När projektet börjar ta form så skapas en assembler som gör om mnenomics till maskinkod.

Timing

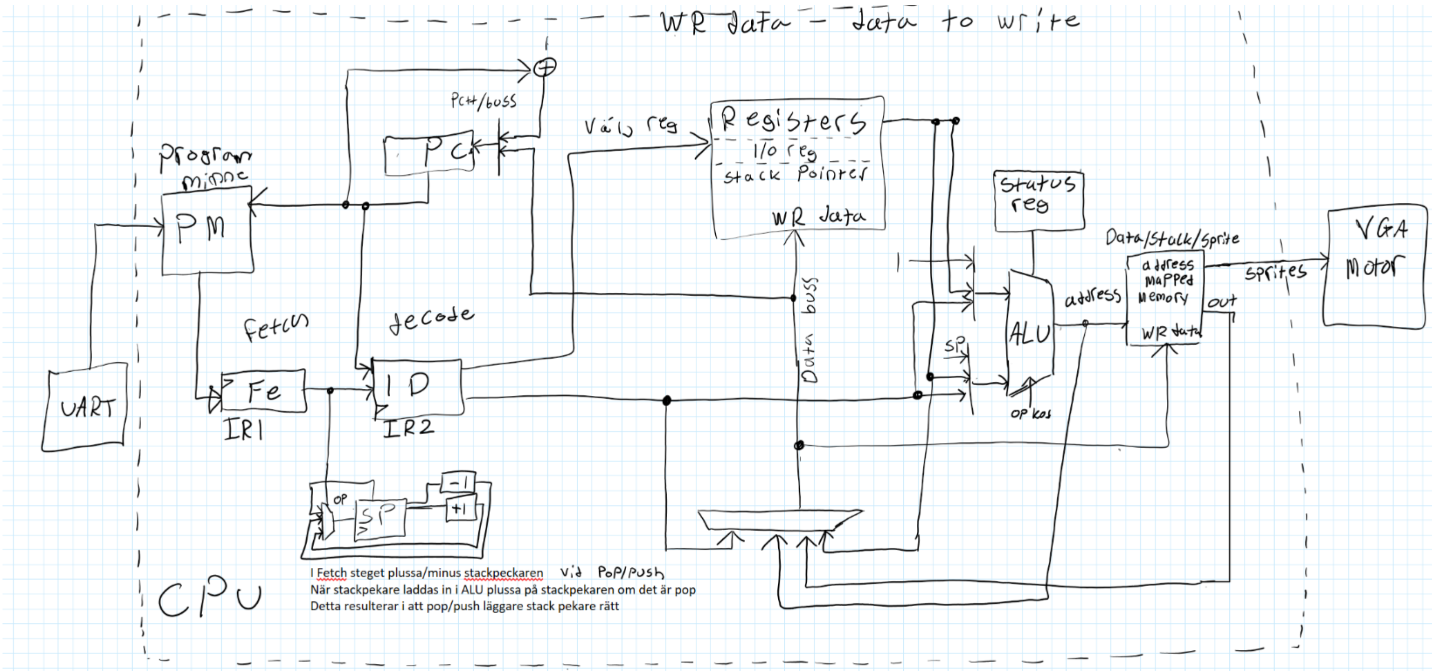
Vi väntar tills skärmen uppdaterar (håller koll på vsync pulser) och skickar en puls som indikerar på att spel loopen ska utföras igen. På detta sätt kan vi synka spelet mot skärmens uppdateringsfrekvens.

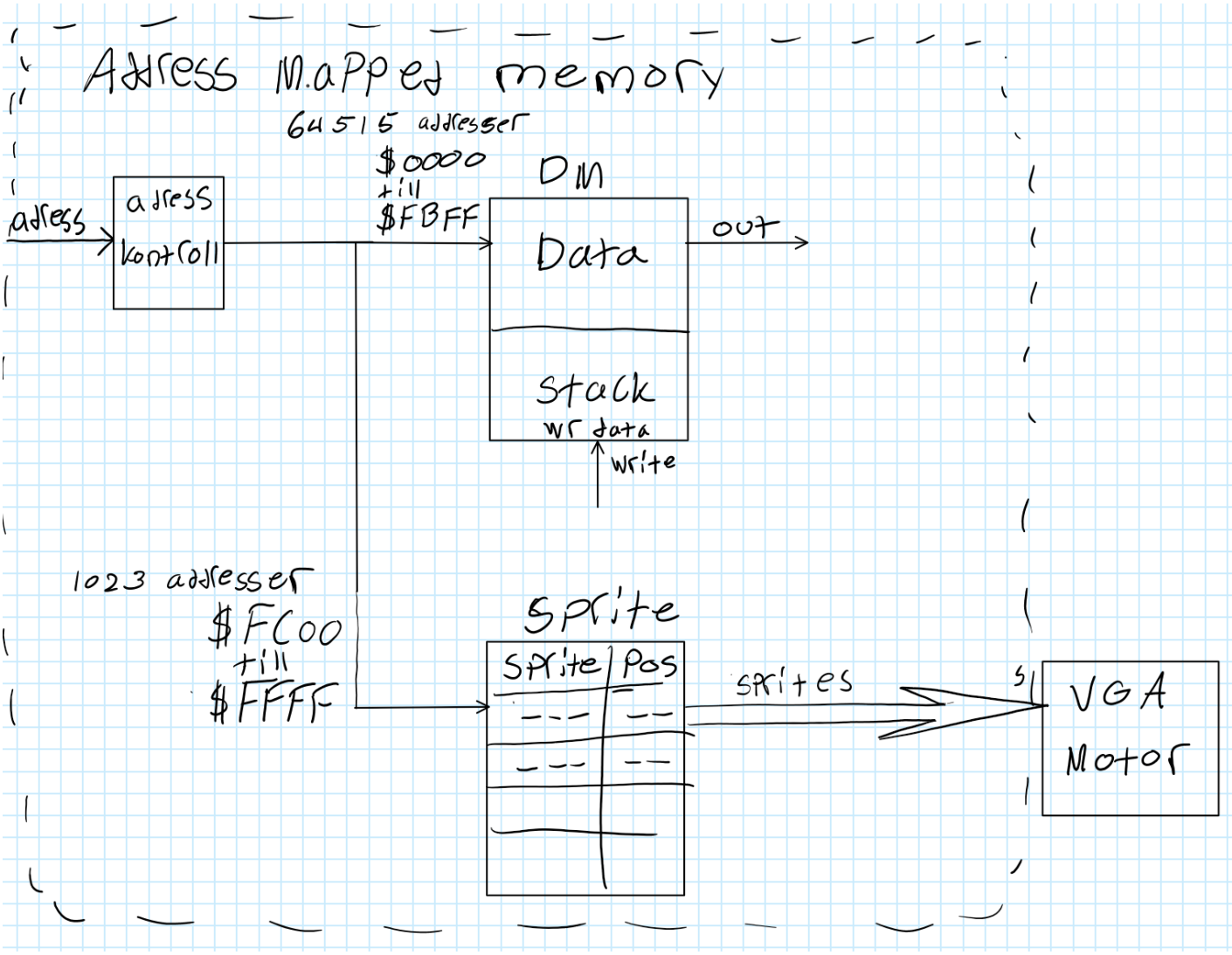
Halvtidsmål

Vi vill kunna rita ut någonting på en skärm och i simulering visa upp en fungerande processor.

Blockschema för designen

Nedan visas ett blockschema över CPU:n och dess kringliggande hårdvara.


WR data – data to write
Pc++/buss
Väln reg
Registers
I/O reg
stack Pointer
WR data
Program minne
PM
PC
Status reg
Data/Stack/Sprite
VGA Motor
address Mapped Memory
WR data
sprites
out
address
ALU
SP
op kod
Data buss
Fetch
decode
UART
Fe
IR1
ID
IR2
op
SP
-1
+1
CPU
I Fetch steget plussa/minus stackpeckaren vid Pop/Push
När stackpekare laddas in i ALU plussa på stackpekaren om det är pop
Detta resulterar i att pop/push läggare stack pekare rätt



Figur 2: Blockschema över CPU:n.