**Designskiss**

**Ice Age**

Version: 3.2

David Schützer – davsc542

Hannes Tuhkala – hantu447

Olav Övrebö – olaov121

# Inledning

Vi tänkte skapa spelet/pusslet **Ice Age** där en spelare ska ta sig från start till slut genom att glida på is och krocka med stenar så att spelaren till slut kommer till målet. Vi tar upp analys över spelet och hur konstruktionen av hur vår dator ska se ut.

# Analys

## Spelplanens design

Spelplanen kommer att bestå av 16x16 rutor där varje ruta kommer att innehålla 2 bitar med information – alltså vad det är för slags ruta.

## Lösning för spelare

Vi kommer att använda oss av tillståndsmaskiner för att bestämma vilket läge spelaren befinner sig i. De olika tillstånden spelaren kan befinna sig i är *moving* och *still*. När spelaren är på is är tillståndet *moving* och när spelaren står på fast mark eller har krockat med en sten är tillståndet *still*. Spelaren rör sig en ruta åt gången.

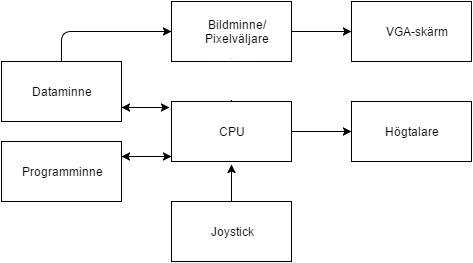
## Kollision

För att möjliggöra inkrementell utveckling av spelet kommer kollisionsdetektering hanteras separat för mark och sten, sådan att karaktären stannar om nuvarande ruta är mark eller om nästa ruta är en sten. Vi kommer med en statebaserad check kolla varje ruta innan spelaren rör sig in i den jämns med kollisionsmatrisen.

# Konstruktion

## Blockschema

Dataminnet och programminnet ska i och för sig ligga i CPU:n, men illustreras (se figur 1) nedan separata då de rent strukturellt skiljer sig så starkt från övriga CPU:n.

****

**Figur 1: Övergripande blockschema för konstruktionen.**

## CPU design

Vi tänkte bygga en pipelinad CPU som har separata minnen för data respektive program, likt den CPU:n vi använde i lab 2. Vi planerar att använda oss av fyra olika adresseringsmoder: Direkt, Indirekt, Omedelbar respektive Relativ adressering.

Det kommer finnas mindre än 16 olika instruktioner som vår CPU stödjer vilket ger att OP koden blir 4 bitar lång. Processorn använder sig av 16 bitars ord bredd och kommer vara laddat från start med ett program.

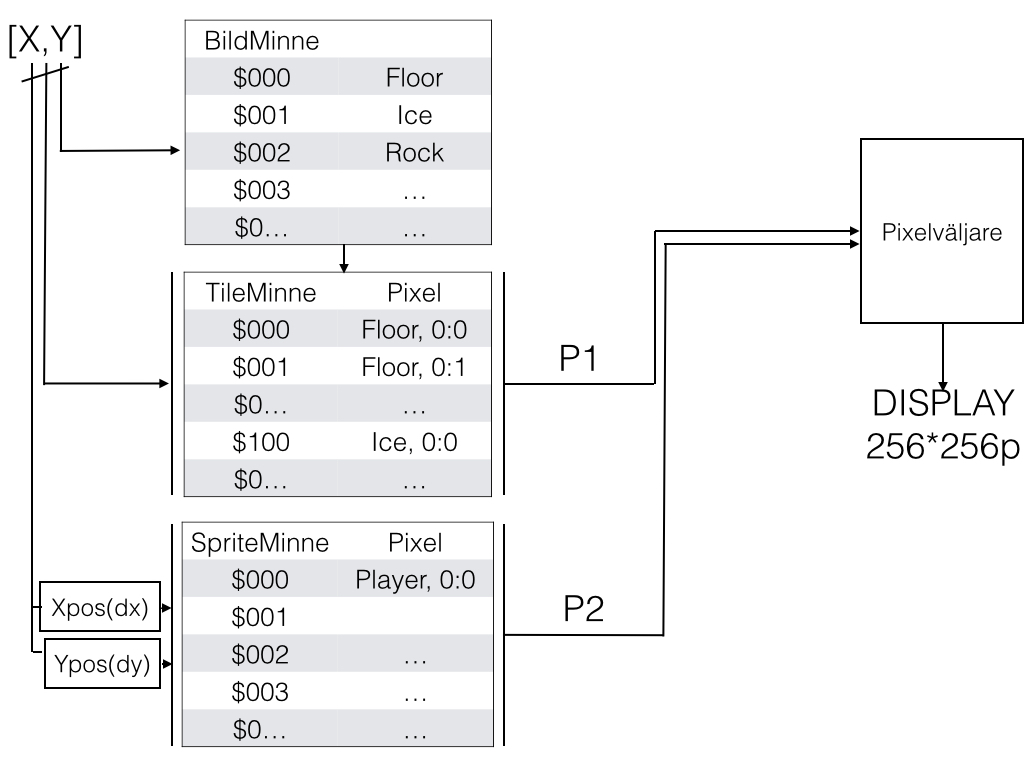
Se bifogad fil (CpuBlockSchema.png).

## Grafik

Vi kommer att använda oss av tiles och sprites. Varje tile kommer bestå av 16\*16 pixlar, och spelplanen kommer bestå av 16\*16 tiles. Detta ger oss upplösningen 256\*256 och vi har stöd för 256 antal färger. Vi kommer att använda oss av ett bildminne som ritar ut spelplanen under varje tick, och sedan lägga på spelaren över bakgrunden. Vi kommer använda oss av tiles för att skapa pixelbilder. Allt på skärmen utanför de 256\*256 pixlar som används för spelet kommer att mörkläggas.

Varje tile i minnet kräver 256 byte, vilket ger 3\*256 bytes för tileminnet. Eftersom vi endast använder en sprite så kommer spriteminnet att gå på 256 bytes.

Koordinater för spelarposition och layout för spelplanen lagras i förbestämda adresser i dataminnet. Pixelväljare, likt CPUn, ges tillgång till dataminnet, vilket innebär att vi (sannolikt med nop-inmatning) kommer behöva motarbeta simultan write och read i dataminnet i CPUn (då block-ram tillåter simultan access för upp till två operationer). Minne för Tiles bör ta 64 bytes, då vi har 16\*16 = 256 tiles om två bitar vardera (tre sorter tiles kräver två bitar för att lagra tillstånd). För strukturell beskrivning av den grafiska komponenten se figur 2.

Koordinater för spelarpositionen lagras i en adress i minnet, och skickas av CPUn vidare till bildminnekoordinater för spelarpositionen tas ut z

**Figur 2: Blockschema beskrivandes grafikenhet.**

## Ljud

VI tänkte ha ett separat minne där vi har musiken/ljudet i .wav format, alltså binärt. Sedan körs en loop på minnet för att slippa använda för mycket av det. Eftersom att ljud är ett bör-krav och förmodligen kan beskrivas med kod så reserverar vi inget specifikt minne för ljud.

## Input

Input, som tas från joystick ska genom hårdvara synkroniseras och läggas in i ett minnesregister. Detta samplas sedan på assembly-nivå, när programmet befinner sig i ett tillstånd där indata tillåts. Detta närmar sig busy waiting, men i en situation där (annat än uppspelning av eventuellt ljud) inget arbete på mjukvarunivå skall utföras förrän det att input fås. Vi konverterar de 10 bitars tal för x och y till ett trebitars tal beskrivande det state joysticken är i.

# Komponenter

De komponenter vi kommer att använda oss av i vårt projekt är följande:

* Nexys3-kort
* VGA-skärm
* Joystick
* Stereohögtalare

5. I/O-enheter

* För projektet krävs en inputenhet och en eller två outputenheter.
* Den inputenhet som krävs är en joystick med vilken användaren förmår styra spelet. För att fylla våra behov behöver joysticken ha minst fyra distinkta riktningar, utöver ett neutralt viloläge. För att läsa av dessa totala fem tillstånd kommer vi alltså behöva åtminstone tre bitars ord från joysticken, vilken vi avser ta in som avbrottsprocess, som skiftar när joysticken går från neutralt läge till att peka i någon riktning. Då spelet om det inväntar input inte ändras, är hastighet inte kritiskt för läsningen av joystick, och således lär det onödigt att dedicera ett register åt endast joysticken; mest sannolikt är att vi kopplar in joysticken till en minnesadress.
* Den outputenhet som vi otvivelaktigt kommer behöva är en VGA-skärm att använda som display, sådan att spelaren kan se banan. För denna kommer vi använda 8bitars färg, med en uppdaterings-frekvens om 60 Hz. Skärmen skall även ha en upplösning om minst 640 \* 480 pixlar. Till denna kommer vi även behöva en VGA-sladd. Data för skärmen ligger i minnesceller.
* Slutligen kan vi komma att behöva en högtalarenhet. Detta är helt beroende på om vi uppnår våra ska-krav om ljudimplementation.

# Milstolpe

Efter halva projekttiden så tänkte vi ha en simpel version av vår CPU som fungerar och att vi kan visa en bild på VGA-skärmen.