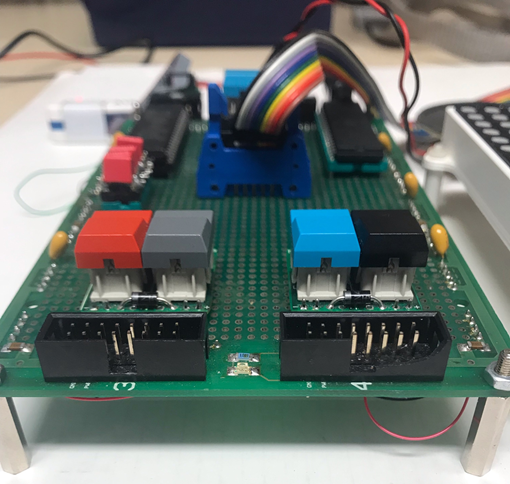
Tetris

Mikrodatorprojekt - Grupp 1

Linköpings Univesitet



Albin Ceder, Daniel Mastell, Jesper Mjörnman, Elin Thorgren

Sammanfattning

Rapporten handlar om skapandet av spelet Tetris i koden Assembly. Detta gjordes i en grupp om fyra personer med hjälp av olika hårdvaror, som till exempel två 8x8 LED-skärmar, ett styrdon och ett par tryckknappar. I slutändan blev resultatet ett välfungerande Tetris där lärdommar tagits med inför framtiden.

Innehållsförteckning

[1. Inledning 1](#_Toc8111840)

[1.1 Syfte 1](#_Toc8111841)

[2. Kravspecifikation: 2](#_Toc8111842)

[2.1 Skallkrav 2](#_Toc8111843)

[2.2 Börkrav 2](#_Toc8111844)

[3. Komponenter 3](#_Toc8111845)

[3.1 Kopplingsschema 4](#_Toc8111846)

[3.2 Display 5](#_Toc8111847)

[3.2.1 SPI 6](#_Toc8111848)

[3.3 ATmega16A 7](#_Toc8111849)

[3.4 Dubbelknappar 8](#_Toc8111850)

[3.5 Resetknapp 8](#_Toc8111851)

[3.6 Högtalare 8](#_Toc8111852)

[4. Tillvägagångssätt och planering 9](#_Toc8111853)

[4.1 Planeringsschema 9](#_Toc8111854)

[5. Analys av koden 10](#_Toc8111855)

[5.1 Översikt av kodstruktur (processor 1) 10](#_Toc8111856)

[5.2 Figurernas logik 12](#_Toc8111857)

[5.3 Rotationslogik 13](#_Toc8111858)

[5.4 Radhantering för full rad 15](#_Toc8111859)

[5.5 Översikt av kodstruktur (processor 2) 17](#_Toc8111860)

[6. Diskussion 18](#_Toc8111861)

[6.1 Lärdomar 18](#_Toc8111862)

[Källförteckning 19](#_Toc8111863)

[Bilaga 1 20](#_Toc8111864)

[Bilaga 2 58](#_Toc8111865)

# 

# Inledning

Tetris är ett ryskt spel från 1984 med ett väldigt simpelt koncept: från toppen av en skärm faller block uppbyggda av fyra rutor där det finns 7 olika figurer. Dessa 7 figurer slumpas fram. Ordet Tetris kommer från det grekiska prefixet *tetra* vilket betyder 4 som i sin tur är antalet rutor per block.

Med hjälp av ett styrmedel som styr åt vänster och höger samt en möjlighet att rotera figuren ska du på bästa sätt placera blocket. När blocket landar stannar den där och en ny figur skapas på toppen av skärmen. På bästa sätt betyder att lägga dem på vågrät rad, där en fylld rad raderar alla rutor på just den raden. Detta ger dig poäng och en chans att fortsätta samla mer poäng genom att göra samma sak igen, bygga upp rader. Tetris är ett evighetsspel i grunden. Så länge du lyckas skapa vågräta rader som sedan tas bort är det bara att fortsätta men så fort en horisontell rad fylls av rutor är det över. Det är Tetris i sin enkelhet men sedan dess har många varianter av spelet gjorts med nya funktioner. Inte för så länge gjordes ett Tetris där spelare mötte andra spelare, Tetris 99. Talet 99 i spelets namn betyder att spelaren möter upp till 99 andra spelare där målet är att ta sig längre än de andra motståndarna. Det är möjligt att sabotera för andra genom att sätta upp kombinationer, vilket betyder att skapa flera vågräta rader i ett svep, med en figur. Detta fyller snabbare upp andra spelares skärmar vilket kan förstöra deras planer eller bara stressa dem.

Detta projekt går tillbaka till grunden av Tetris, innan alla finesser hade skapats. Det väsentliga är det enda som finns där målet är att se vem som är bäst på att se mönster, ha snabb reaktionsförmåga och kunna planera flera steg fram. Med två LED-skärmar, ett par knappar och lite Assembler-kod ska det enfärgade retrospelet Tetris få liv igen.

## 1.1 Syfte

Varför Tetris valdes till detta projekt är enkelt, det verkade som ett fullständigt möjligt spel att göra inom en period på 6 veckor och passade även bra med de nya 8x8 LED-skärmarna som fanns att använda sig av. Syftet med detta projekt är att se de olika byggstenarna som behövs för att göra ett fungerande Tetris.

# 2. Kravspecifikation:

Det fanns en del krav för att få godkänt på projektet, uppdelat i skallkrav och börkrav. Skallkraven är de krav som anses nödvändiga för projektets basfunktioner,

## 2.1 Skallkrav

* Möjligheten till att kunna rotera figuren, samt att kunna åka åt både höger håll samt vänster håll.
* Vid skapandet av en horisontell rad ska raden tas bort och figurerna ska flyttas ner.
* Själva blocken ska slumpas och skifta mellan 7 olika varianter av figurer.
* Vid radrensning och förlust ska det spelas ett enkelt ljud.

De utökade kraven, eller börkraven, var mer krav som skulle göra spelet bättre men som inte var väsentliga för spelet Tetris.

## 2.2 Börkrav

* En poängtavla som sköter hur många poäng som sparats, samt håller reda på vem som genom tiderna samlat mest poäng i samma försök.
* Färg på de olika blocken var också ett börkrav. De senare generationerna av Tetris består av olika sorters färger men i början var spelet enfärgat. På det sättet som koden sparade positionen av blocken var det inte lätt att låta varje figur ha sin egna färg så beslutet blev att ha Tetris i en och samma färg.
* Dynamisk hastighet var det sista börkravet. Beroende på hur långt spelaren kom i spelet, alltså hur länge användaren har kört, ändras också hastigheten. Så desto längre tid som går desto svårare/snabbare blir spelet.

# Komponenter

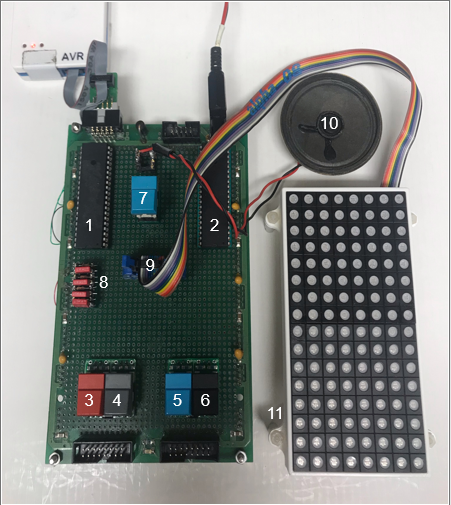
De komponenter som har använts under detta projekt i numrerad ordning är:

Bild 1. Överblick över konstruktionen.

1. ATmega16A (spellogik)
2. ATmega16A (ljud)
3. Avstudsad knappmodul (Rotation)
4. Avstudsad knappmodul (Rotation)
5. Avstudsad knappmodul (Vänsterflytt)
6. Avstudsad knappmodul (Högerflytt)
7. Reset-knapp
8. Motstånd
9. HC595
10. Högtalare
11. LED 8x16 matris

Arbetsfördelningen mellan de två processorerna har under detta projekt blivit något ojämn, där den ena processorn förfogar över majoriteten av koden gällande spellogik och hårdvara medan den andra processorn istället besitter kod över ljudet och har i uppgift att styra högtalaren.

Den första avstudsade knappmodulen har i uppgift att sköta rotationen av spelfigurerna. Den första knappen är tillägnad moturs-rotation och den andra medurs-rotation.

Den andra knappmodulen styr istället den horisontella förflyttningen kring spelfigurerna i spelet, den första för vänster respektive andra för högerförflyttning. Den vertikala förflyttningen av figurerna har valts bort utan sker istället naturligt, likt gravitation, kodmässigt.

Det finns en extern reset-knapp som är kopplad mot ett motstånd för att fungera korrekt och har varit av stor betydelse under projektets gång. Den har främst kommit till användning gällande felsökning av buggar, som har skett mjukvarumässigt, för att snabbt starta en ny spelsession i hopp om att försöka återskapa och lokalisera hur ett fel har uppstått.

HC595 är en seriell styrning som möjliggör kommunikation mellan ena processorn och två seriekopplade skärmar. Skärmarna husas av en 3D-printad plastbehållare som håller ihop dem och tillsammans skapar en fullgod spelplan för Tetris.

Högtalaren är tyst till dess att en hög flank från den andra processorn sker. Processorn styr också hur högtalaren ska låta.

## 3.1 Kopplingsschema

Bild 2. Kommunikationen mellan komponenterna.

Bild 2 innehåller kopplingsschemat över komponenterna. Där kan det tydligt ses att den övre ATmega16A processorn är den som är direkt kopplad till majoriteten av komponenterna som används i detta projekt. Den har alltså ansvaret för att kommunicera med båda avstudsade knappmodulerna, för såväl rotationen som den horisontella förflyttningen, men också till displayen och till sist den externa reset-knappen, som i sin tur är kopplad mot ett motstånd.

Den undre ATmega16A processorn är den som, tidigare nämnts, sköter ljudet i projektet och är därför direkt uppkopplad för kommunikation mot högtalaren.

Processorerna kommunicerar sinsemellan genom två vardera direkt kopplade pinnar. Detta gör att det är enkelt att identifiera när under spelets gång som det vill spelas ett visst ljud. Genom att skicka en hög flank till en specifik pin, kommer det i sin tur ge en hög flank hos den undre processorn. Genom detta tillvägagångssätt går det att veta just vilken pinne som fått en hög flank och också vilken melodi som önskas att spelas upp genom högtalaren.

Värt att nämna är att det befinner sig två stycken JTAG:s på kopplingsschemat trots att det endast fanns en i verkligheten. Detta är för tydlighetens skull, då endast en processor kunde programmeras i taget. Detta görs genom att använda Daisy Chain i programmet Atmel Studio som har använts till projektet vilket också är där koden har skrivits. För att komma åt den första processorn anges att det inte är några enheter före, men att det är en enhet efter, men också att det inte är några bitar före, men fyra bitar efter. För att nå den andra processorn är processen omvänd.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

## 3.2 Display

Bild 3. Skärmfunktion över en 8x8 RGB LED-matris.

Spelplanen som valdes är egentligen uppbyggd av två stycken seriekopplade 8x8 RGB   
LED-matriser. Dessa två tillsammans bildar en total spelyta om 8x16. Det bildar en önskvärd storlek för projektet då spelfigurerna som mest är tre tända dioder i höjdled. Detta gör det möjligt att stapla flera figurer på varandra innan att spelplanen tar slut.

Nollorna i figuren ovan är tända dioder, vilket kommer att representera de olika figurerna, medan ettorna representerar släckta dioder som indikerar på ledigt spelutrymme.

Då skärmen är av RGB har den möjlighet att representera olika färger beroende på vilka värden som anges för respektive bit. RGB är en förkortning för färgerna rött, grönt och blått och används för att möjliggöra ett spektrum av olika färgkombinationer.

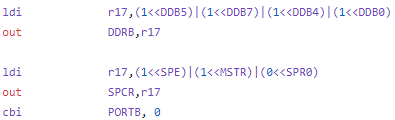
För att matriserna ska fungera korrekt används två olika speciella drivkretsar. Den första benämns source och tillgodoser ström radvis medan den andra benämns sink och jordar kolumnvis. Dessa drivkretsar är en del i konstruktionen av kretsarna 2981 respektive 2803. 2981 är en source-krets som innehåller åtta source-kretsar och 2803 är en sink-krets som består av åtta sink-kretsar. I konstruktionen används en 2981 och två 2803. Dessa är i sin tur seriekopplade genom varsina HC595 skiftregister och bildar tillsammans ett 24-bits skiftregister som möjliggör seriell styrning. Ordningen som bitarna skickas ut till HC595-kretsen är anod, röd, grön och blå och sker genom användning av SPI.

## 3.2.1 SPI

Bild 4. Illustrering av SPI.

För att kunna påbörja kommunikationen mellan den första processorn och displayen används Serial Peripheral Interface (SPI) som är ett synkront seriellt skiftregister.

SPI bussen använder sig av fyra olika logiska signaler: Serial Clock (SCLK), Master Output Slave Input (MOSI), Master Input Slave Output (MISO) och Slave Select (S̅S̅).



Kod 1. Kodexempel över initieringen av SPI.

Initieringen av SPI startar genom att tilldela en master, vilket är första ATmega16A processorn. När detta sker sätts både klockhastigheten och en hög flank skickas till S̅S̅. Först när den höga flanken i S̅S̅ sker är bussen mottaglig för dataöverföring.

Bild 5. Skiftning av bitar.

Dataöverföringen består i detta fall av totalt åtta bitar, då det används två skärmar. Bitarna skiftas ut en i taget undertiden S̅S̅ ligger i viloläge och mellan varje skiftning sker en mindre fördröjning. När väl alla åtta bitar har skiftats ut på SPDR skickas det en låg flank till S̅S̅ och då är överföringen klar.

Mottagandet av dataöverföringen sker hos en slave, vilket är displayen. Detta sker bytevis i samband med ett avbrott då SPIF-flaggan är satt i SPSR.

## 3.3 ATmega16A

Två ATmega16A processorer användes till projektet. Processor 1 stod för utskrivning till en 8x16 matris, hantering av indata från knappar samt spellogik. Processor 2 var kopplad till en högtalare som processorn skickade signaler till och ljud kunde därefter komma ut ur högtalaren. ATmega16A har tillgång till 32 register som är 8 bitar långa.

ATmega16A har fyra portar: A, B, C och D, enligt bild 6. Varje port innehåller åtta pinnar som är numrerade 0 – 7 och kan användas som in- eller utgångar av logiska signaler som bestäms via initialisering. Det finns även pinnar på processorn som inte tillhör någon port. De pinnarna är 9 – 13 samt 30 – 32. Av de som används så är de kopplade från:

* Pin 10 och pin 30 till jord (GND)
* Pin 11 och pin 31 till ström (VCC)

För de övriga pinnarna kan de användas till processorns interna funktioner. Pinnarna för PORT A kan ta emot analoga signaler till den inbyggda A/D-omvandlaren. PORT D kan hantera avbrottssignaler. I detta projekt används de interna funktionerna däremot inte. För processor 1 hanterar för övrigt PORT A indata från tryckknappsmodulen som sköter horisontell förflyttning, PORT B hanterar kommunikation med display, PORT C hanterar JTAG och SPI och PORT D hanterar kommunikation till Processor 2 samt indata från tryckknappsmodul som sköter rotation. För processor 2 hanterar PORT A indata från processor 1 och PORT C hanterar JTAG samt skrivning till högtalaren.

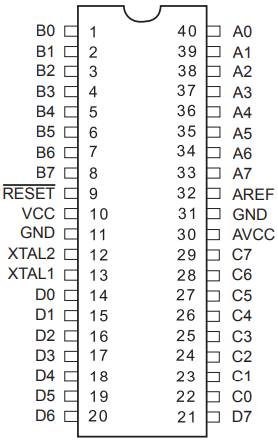


Bild 6. Atmega16A processor. (Atmel, 2014)

## 3.4 Dubbelknappar

Bild 7. Tryckknappsmodul.  
(Vanheden, 2019)

Det är de avstudsade tryckknappsmodulerna som används för att spela det egengjorda Tetris. Med två 2x1 tryckknappsmoduler med inbyggt motstånd finns det totalt fyra knappar att använda varav två sköter den vertikala flyttningen och en av dem sköter rotationen. En av knapparna är inte inkopplad och inte heller tänkt att användas. För konstruktionen av kretskortet kopplades tre pinnar från processor 1 till knapparnas positiva utgångar markerade P på bild 7. Dessa ger en positiv utsignal vid nedtryckning av knapp.

## 3.5 Resetknapp

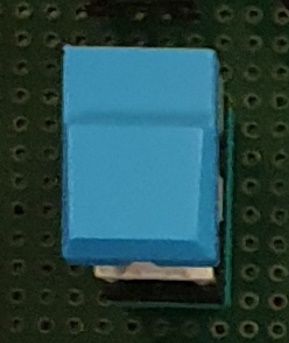
Resetknappen monterades för att under arbetets gång kunna felsöka bra, men även som en spelfunktion för att ge användaren en trevlig spelupplevelse. Enkelknappen, bild 8, installerades så att en positiv flank skickas när knappen är intryckt och processor 1 programmerades utefter detta. Knappen som användes har inget inbyggt motstånd och kopplades därav till ett motstånd för att inte kortsluta hela kretskortet.

Bild 8. Enkel tryckknapp

## 3.6 Högtalare

Högtalaren används till spelet för att generera ett retroljud om en rad försvinner och tre på varandra följande ljudsignaler om spelaren förlorar. Högtalaren är kopplad till jord (GND), men även till pinne 23 på processor 2. I processor 2 sköts logiken för vilket ljud som ska spelas enligt kod på bild 9. Där skickas även ett okänt antal signaler ut till högtalaren som omarbetas till ljudsignaler möjliga att höras.

Bild 9. Högtalare

# Tillvägagångssätt och planering

I början av projektet var tempot lugnt för att ta sin tid att läsa datablad och skaffa sig en riklig grund. Processorerna började därefter kopplas på kretskortet och möjligheten till SPI skapades genom att koppla ihop rätt pinnar med varandra. När det blev dags för kodning ökade tempot. En prick på displayen började snart röra sig vertikalt ner med hjälp av en avbrottsfunktion. Knappar kopplades in för att förflytta pricken horisontellt och med lite spellogik hade ett väl fungerande Tetris skapats.

Det gick därefter till att skapa figurer och hitta ett sätt att rotera dessa. Under tiden detta gjordes arbetades det även på högtalaren för att skapa ljud när raden rensades eller vid förlust. Efter två veckor hade spelet tagit sin form och efter lite slutlig buggtestning ansågs det klart och redo för redovisning.

## 4.1 Planeringsschema

I tabell 1 nedan visas en sammanfattning av planeringsschemat över hur veckorna såg ut i form av vilket arbete som utfördes.

Tabell 1. Planeringsschema

|  |  |
| --- | --- |
| Vecka | Arbetsuppgift |
| v.5 | Krav specialisering inlämnad (Fredag) |
| v.6 | Läsa datablad och börja koppla |
| v.7 | Gravitation och Tetris med en prick |
| v.8 | Figurer |
| v.9 | Rotation och högtalare |
| v.10 | Rotation och buggar |
| v.11 | Skrivning av redovisning och buggar |
| v.12 | Redovisning (Måndag) |

# 5. Analys av koden

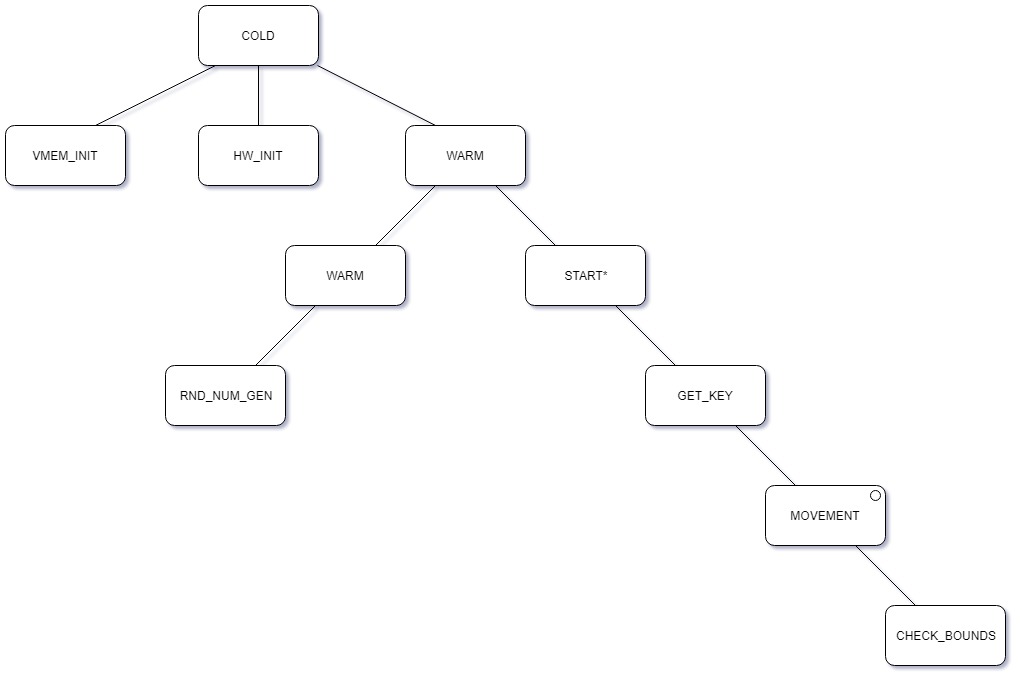
I följande kapitel kommer väsentliga delar av koden att granskas och förklaras. För att kunna förstå vissa delar av följande kapitel måste ett fåtal saker förklaras:

* Displayen består av 8x16 lysdioder. När väggar och tak förekommer syftar det till gränserna på displayen, det vill säga är något utanför gränsen ryms det inte på displayen.
* Bilderna som representerar en spelplan illustrerar endast undre halvan av spelplanen (en av de båda 8x8 displayerna).

## 5.1 Översikt av kodstruktur (processor 1)

Bild 10 är en representation över processerna som processor 1 hanterar, exklusive avbrottsrutiner vilka beskrivs nedan i bild 11. I startprocessen COLD hanteras vissa initieringar som att sätta stacken. Därefter kallas VMEM\_INIT vilket initierar videominnet. Denna ordning är viktig för att videominnet inte ska bråka med gravitationen samtidigt som rensning sker. HW\_INIT initierar hårdvaran och initierar därmed avbrott via intern klocka och SPI:n. Från START kallas kontinuerligt subrutinen GET\_KEY som inväntar ett knapptryck för hantering av figurernas flyttrutiner.

Bild 10. JSP-diagram över processor 1 (exklusive avbrottsrutiner)



Vid ett specifikt knapptryck kallar rutinen GET\_KEY på ett specifikt MOVEMENT som i sin tur tillkallar CHECK\_BOUNDS för kollisionshantering innan MOVEMENT får lov att utföra sin flytt. Alltså granskas kollisionerna innan förflyttning har skett vilket underlättar vid minneshanteringen.

Avbrotten kommer kontinuerligt där GRAVITY**-**avbrottet sköter hur figuren förflyttas nedåt medan MUX**-**avbrottet sköter displayen. GRAVITY, se bild 11, består av ett flertal subrutiner och funktioner som måste tillkallas i ordning innan avbrottet är över.

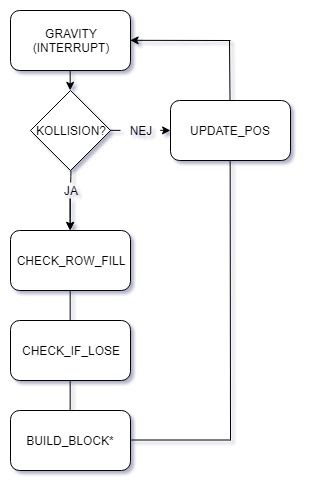


Bild 11. Diagram över avbrottsrutinen GRAVITY

Avbrottet sker i 5 steg:

1. Undersök om kollision. Uppfylls kollisionsvillkoret avbryts avbrottet.
2. Undersök om en rad är fylld. Är en rad fylld startas rutinen för radborttagning.
3. Undersök om förlust. Når figuren ”taket”? Om ja har förlust skett och spelet spelar upp ett förlustljud och stannar klockan för GRAVITY (spelet stannar och visar hur du förlorade).
4. Bygg ett nytt block om kollision har skett.
5. Uppdatera positionerna. Eftersom figuren faller ned genom att ändra dess Y**-**koordinater kommer figurens förra plats ligga kvar i minnet. På grund av detta måste den gamla platsen rensas ur minnet och den nya istället sparas. Detta sker oavsett om det uppstått kollision eller ej.

Det andra avbrottet som sker klockvis är MUX. Detta avbrott sköter utskrift till LED**-**matrisen genom att skifta ut information via SPI. Sättet som muxningen sker på är inte koordinatvis utan istället radvis. Alltså sköts utskriften genom att skicka ut en rad per display för varje MUX**-**avbrott. Informationen för utskrift sköts genom att:

1. Hämta MUXCOUNTER för att se vilken rad som är aktuell för utskrift.
2. Hämta information ur VMEM, som innehåller 16 rader med information, genom att hämta raden MUXCOUNTER och MUXCOUNTER + 7. Genom detta får båda displayerna information som kan skrivas ut samtidigt.
3. Skifta ut information till displayerna.
4. Öka MUXCOUNTER om MUXCOUNTER ≠ 8 annars sätt MUXCOUNTER = 0

Muxningsmetoden som används är väldigt snabb och resulterar därför i att muxningen inte stör andra rutiner som körs när avbrottet sker. Ljusstyrkan på dioderna blir också starkare av detta samt att spelhastigheten kan manipuleras utan hänsyn till muxnings-hastigheten.

## 5.2 Figurernas logik

Tetris består av figurer av fyra punkter vilket innebär att det på något sätt måste sparas koordinater för dessa prickar. Det sätt som valdes är att spara det som ett koordinatplan bestående av tre rader med åtta kolumner. Detta plan definierar hur figuren kommer att se ut och går att direkt implementera i videominnet (VMEM). Därför sparas koordinaterna på ett mer sparsamt sätt med förlusten av individuell kontroll av punkter. Koordinaterna för y sparas simplare och mer hanterbart, nämligen genom att spara ett heltal som representerar ett radnummer (1–16). Koordinaterna för x sparas istället i en frigiven byte per x-koordinat, där det sparas åtta ”punkter” med information, se bild 12.

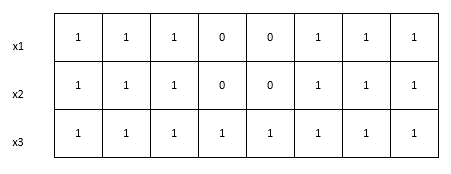
Denna minnesrad beskriver de punkter, i en av figurens rader, som är aktiva respektive inaktiva (nollor aktiva, ettor inaktiva). En figur byggs sedan upp av tre stycken bytes där varje x beskriver en rad i figuren likt nedan:

Bild 13. Tre stycken x-minnen som bygger upp ett koordinatplan och definierar en fyrkant.

Bild 12. Minnet för en x-byte (sparar koordinaterna för en rad)

Koordinatplanet ovan, bild 13, skulle alltså då rita ut en fyrkant baserat på var   
x-koordinaternas respektive y-värde är. Denna metod innebär att muxningen kan ske genom att muxa en hel rad åt gången vilket ökar muxningens hastighet avsevärt.

## 5.3 Rotationslogik

För att kunna rotera figurerna måste två olika problem överkommas: ska rotationerna vara hårdkodade eller generella? Hur ska rotationer hanteras när kollision sker efter rotation? Eftersom koordinaterna sparas som rader är det komplicerat att manipulera enskilda koordinater för att kunna skapa en generell algoritm för rotation. Detta medför att rotationerna sparas hårdkodade i minnet. Kollisionshanteringen genomförs i tre steg: Är det något direkt under figuren? Är figuren vid en vägg? Är figuren instängd av upptagna punkter i spelplanen?

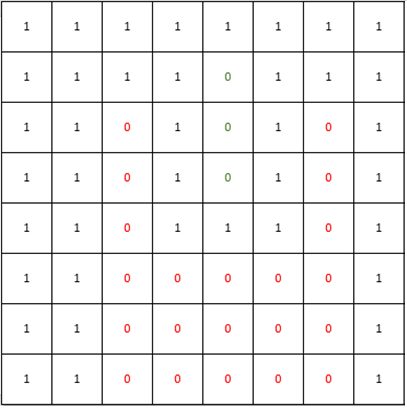
Första villkoret för rotation är väldigt simpel; kontrollera om det är upptaget under figuren. Detta görs eftersom rotationspunkten resulterar i att den blir minst en bredare i x-led eller minst en högre i y-led vid rotation. Kollisionschecken undersöker alla punkter i figuren för att se om det är någon plats undertill som är upptagen.

Bild 14. Illustrerar hur kollision ses. Röda är upptagna, gröna är aktiva figuren

Bild 14 är en illustration av en halv spelplan (8x8) där grön representerar figuren som för närvarande spelas och röd representerar färdigspelade figurer. Rotationen sker alltid runt koordinaten i mitten eller x2 i koordinatminnet. Först görs kollisionscheck för alla punkter i figuren och om någon ger ett sant retur-värde, att figuren då befinner sig direkt ovanför en upptagen plats, avbryts rotationsprocessen och ingenting händer. Denna lösning för detta problem är väldigt primitiv men resultatet blir ett mer pålitligt rotationssystem.

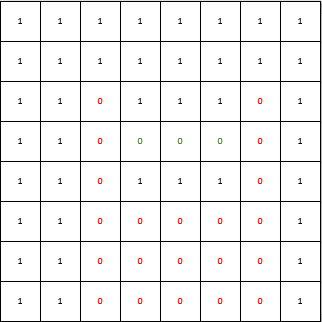
Liknande metod används vid rotation av en instängd figur likt nedan, se bild 15, men istället görs först en kontroll för att se vilken rotation som är aktuell. Om det är så att figuren kommer att ”lägga sig ned” kontrolleras om det är fritt ett steg till höger respektive vänster innan rotationen genomförs. Om kontrollen returnerar falskt kommer rotationen avbrytas och inget kommer att hända. I bild 14 kommer rotation att genomföras eftersom längden kommer omvandlas till bredden likt nedan, bild 15.

Bild 15. Resultat efter att rotation har kallats i bild 14

Samma teknik används vid väggarna för att se om rotationen kommer göra att figuren hamnar utanför. Roteras figuren vid väggen kommer figuren, om det krävs, att flytta ett steg inåt (bort från väggen). För att detta ska fungera utan problem undersöks det om det finns plats att flytta in figuren efter rotation. Om plats inte finns kommer rotationen avbrytas och ingenting händer.

## 5.4 Radhantering för full rad

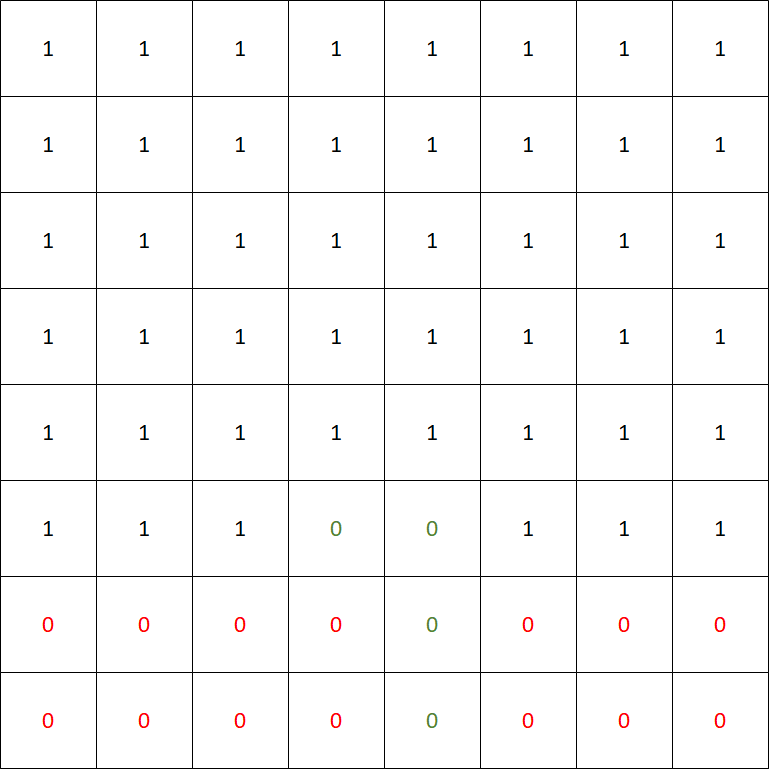
När ett block har landat kallas en subrutin som kontrollerar om en rad har blivit fylld. Om en rad har blivit fylld resulterar det i att den eller de rader som är fyllda kommer att försvinna ur videominnet VMEM. Enligt spelreglerna för Tetris ska detta resultera i poängökning, men det sker inte i denna versionen av Tetris på grund av varierande anledningar där tidsbrist är den främsta.

Bild 16. Figur (grön) landar och fyller upp 2 rader

Bild 16 illustreras hur en L-figur precis har landat. Vid nästa avbrott som hanterar GRAVITY kommer en kollisionscheck genomföras och i detta fall köra rutinen för radborttagning och skicka en signal till processor 2 som spelar upp ett ljud. Denna algoritm kör rensning från översta raden nedåt. Eftersom alla rader ovanför den borttagna måste tas bort kommer koden att iterera över alla rader som ska undersökas, se bild 17.

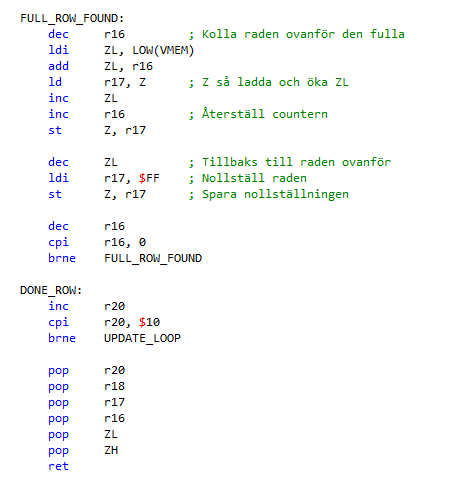


Bild 17. Kodsnutt som hanterar radflytt och borttagning

Rutinen sköter radhanteringen genom att ladda in raden ovanför den som kommer att försvinna följt av att raden under skrivs över med värdet från raden ovan. På detta sätt sparas tid då raderna som är fyllda inte behöver nollställas utan istället blir direkt överskrivna. Efter detta backas Z-pekaren tillbaka till den rad som nyligen ”flyttats” ned och nollställer därefter den nyligen flyttade radens gamla position. När detta är gjort minskas registret r16, som i detta fall räknas som en loopcounter, för att sedan flytta ned de rader som är kvar att flytta på. När detta är klart ramlar koden ner i DONE\_ROW som ser till att genomföra operationen för alla rader på spelplanen. Detta sker eftersom det kan vara fler än endast en rad som blivit fylld. Efter rutinen kört klart kommer spelplanen att se ut likt Bild 18.

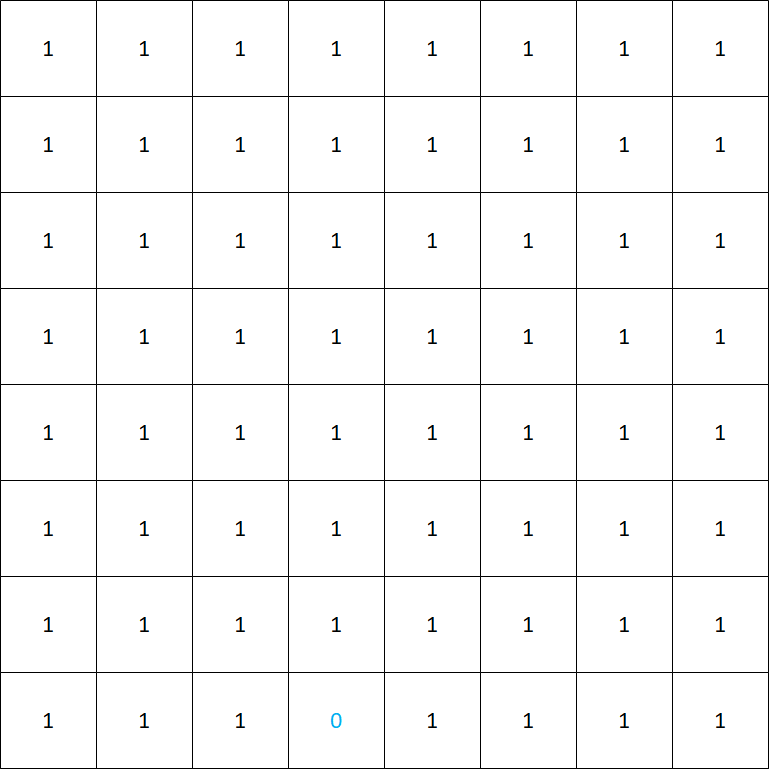


Bild 18. Resultat av bild 15 efter radborttagning. Blå representerar biten som är kvar av den spelade figuren efter rensning.

Eftersom två av raderna i bild 17 var fyllda försvinner dessa efter rutinen i bild 18 och kvar blir då den, i bild 18, blå biten som landade ovanpå raderna. När GRAVITY**-**avbrottet är genomfört kommer sedan rutinen BUILD\_BLOCK köras så att spelet fortsätter.

## 5.5 Översikt av kodstruktur (processor 2)

Bild 19. JSP för processor 2

Processor 2 sköter endast ljudhantering genom att få in signaler på två av sina ben. Dessa två insignaler bestämmer huruvida det ska spelas ett ljud för radborttagning eller om det ska spelas ett ljud för förlust. Processerna som används är väldigt simpla och använder sig endast av olika delayer för att sätta hög respektive låg signal på den pinne som är kopplad till högtalaren. Bild 19 illustrerar ett JSP**-**diagram som tydliggör hur processorn hanterar sina requests. Cold initierar stacken och in- respektive utgångar. Rutinen START loopas och inväntar en signal från en av två pinnar. När en signal uppfångats verifierar rutinen vilken pinne som skickade signalen och väljer nästa subrutin beroende på vad som ska spelas. Valen som görs är att spela ett ljud för förlust eller att spela ett ljud när en full rad är funnen.

# 6. Diskussion

Projektet har varit givande och har gjort att vi har fått använda oss av kunskaper från tidigare kurser till att förverkliga detta projekt. Vi lyckades uppnå skallkraven men under projektets gång insåg vi att tiden var knapp. Av den anledningen uppfyllde vi inte våra börkrav. Det var en del hinder längs vägen som hindrade arbetsflödet. Det första ordentliga hindret som vi stötte på vid kommunikationen mellan processorn och skärmen. Efter en del sökning inom diverse datablad utan att komma någonstans fick vi den hjälpen som behövdes för att få skärmen att lysa. Strax efter kunde vi bestämma hur 8x16 LED-displayen skulle lysa.

Då det tog en del tid att sätta upp skärmen var vi ivriga att börja koda för att komma ikapp med planeringen. Resultatet av detta var att vi av misstag vände displayen upp-och-ned som upptäcktes först efter att vi implementerat spellogiken. Konsekvenserna av detta misstag var att styrningen blev inverterad. Vid högerförflyttning innebar detta kodmässigt ett logiskt skift vänster och vice versa. Detta ledde till ett vägval, antingen att skriva om majoriteten av spellogiken eller att acceptera det misstag vi gjort. Vi beslutade oss att fortsätta på den logik vi hade då vi trodde att det inte skulle påverka arbetet i allt för stor mening.

Nästa hinder som dök upp var vid rotationen av figurerna. På det sätt vi sparar koordinaterna gjorde det nästintill omöjligt att finna en generell lösning. Resultatet blev att hårdkoda alla figurernas samtliga rotationer, vilket ledde till en hel del extra kod. På grund av det tidigare misstaget var även rotationerna inverterade kodmässigt vilket ökade komplexiteten.

## 6.1 Lärdomar

Vid reflektion över projektet finns det en del lärdomar att ta med sig för framtida bruk.

* Planera mer genomgående från början med hänsyn till förseningar.
* Reflektera över de val man gör, periodvis granska arbetet
* Stressa inte över att planeringen inte följs till takt.
* Fördela arbetet mer noga från start.

# Källförteckning

Atmel (2014). *Pinout ATmega16A*. [bild]. [2019-04-07]

<http://www.isy.liu.se/edu/kurs/TSIU02/5_Atmega16.pdf>

Vanheden, Avstudsad (2019) *Modul.* [bild]. [2019-04-02]

<https://docs.isy.liu.se/pub/VanHeden/DataSheets/avstudsad_tryckknappsmodul_2x1.pdf>

Vanheden, Ledmatris (2019) *Kopplingschema*. [text]. [2019-04-01]

<https://docs.isy.liu.se/pub/VanHeden/DataSheets/ledmatris.pdf>

# Bilaga 1

.org 0

rjmp COLD

.org INT0addr

rjmp ROTATE

.org OVF1addr

rjmp GRAVITY

.org OVF0addr

rjmp MUX

; --------------------------

; |--- VARIABLE LAYOUT ---|

; ---------------------------

.def MUXCOUNTER = r19

.def LOOPCOUNTER = r21

.def BOOLEAN = r18

.equ BLOCK\_SIZE = 3

; -------------------------

; |--- MEMORY LAYOUT ---|

; -------------------------

.dseg

VMEM: .byte 16

LINE: .byte 1 ; Sparar vilken rad vi är på för att MUX:a rätt

POSX: .byte BLOCK\_SIZE

POSY: .byte BLOCK\_SIZE

SEED: .byte 1

FIGURE: .byte 1 ; 1 = I | 2 = L1 | 4 = L2 | 8 = SQUARE | 10 = Z1 | 20 = Z2 | 40 = PYRAMID

ROT: .byte 1 ; 0 = NO rotation, 1 = 1 rotation, 2 = 2 rotations, 3 = 3 rotations, 4 = 4 rotations (Back to 0)

ROTP: .byte 1 ; Rotationspunkt -> rotera kring och kompensera i x-led för rotationen

; ------------------------

; |--- CODE SEGMENT ---|

; ------------------------

.cseg

COLD:

ldi r16, HIGH(RAMEND)

out SPH, r16

ldi r16, LOW(RAMEND)

out SPL, r16

clr MUXCOUNTER

ldi ZH, HIGH(LINE)

ldi ZL, LOW(LINE)

ldi r16, $01

st Z, r16

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

clr r16

st Z, r16

ldi ZH, HIGH(ROTP)

ldi ZL, LOW(ROTP)

ldi r16, $10

st Z, r16

rcall VMEM\_INIT

;rcall GAME\_OVER

call HW\_INIT

;rcall VMEM\_INIT

WARM:

rcall BUILD\_BLOCK

START:

rcall GET\_KEY

rjmp START

GET\_KEY:

sbic PINA, 0

call MOV\_LEFT

sbic PINA, 1

rcall MOV\_RIGHT

ret

;-----------------------------

;--- VMEM initieras med värden

;--- "Nollställer spelplanen"

VMEM\_INIT:

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

clr r16

ldi r17, $FF

VMEM\_SET:

st Z+, r17

inc r16

sbrs r16, 4

rjmp VMEM\_SET

ret

GAME\_OVER:

push ZH

push ZL

push r19

push r16

;GG: .db $C3, $DF, $DF, $DF, $D3, $DB, $DB, $C3, $C3, $DF, $DF, $DF, $D3, $DB, $DB, $C3

clr r16

clr r19

GG\_SET\_DISP: ; FUNGERAR NU

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

ldi r16, $C3

st Z+ , r16

ldi r16, $DF

st Z+, r16

st Z+, r16

st Z+, r16

ldi r16, $D3

st Z+, r16

ldi r16, $DB

st Z+, r16

st Z+, r16

ldi r16, $FF

st Z+, r16

inc r19

cpi r19, 2

brne GG\_SET\_DISP

pop r16

pop r19

pop ZL

pop ZH

ret

;-----------------------------

;--- MOVEMENT - LEFT

;--- USES: Z, r16, r17,

MOV\_LEFT:

push ZH

push ZL

push r16

push r17

push r22

push BOOLEAN

push LOOPCOUNTER

clr BOOLEAN

ldi LOOPCOUNTER, BLOCK\_SIZE

ldi r22, $01

rcall BORDER\_CHECK ; Check borders before movement

rcall BLOCKED\_LEFT

MOVING\_L:

dec LOOPCOUNTER

sbrc BOOLEAN, 0

rjmp END\_MOVL

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

add ZL, LOOPCOUNTER

ld r16, Z

com r16

lsr r16

com r16

st Z, r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

add ZL, LOOPCOUNTER

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

com r16

lsl r16

or r17, r16

lsr r16

com r16

and r17, r16

st Z, r17

END\_MOVL:

cpi LOOPCOUNTER, 0

brne MOVING\_L

ldi ZH, HIGH(ROTP)

ldi ZL, LOW(ROTP)

ld r16, Z

sbrs BOOLEAN, 0

lsr r16

st Z, r16

rcall WAIT\_RELEASE

pop LOOPCOUNTER

pop BOOLEAN

pop r22

pop r17

pop r16

pop ZL

pop ZH

ret

;-----------------------------

;--- MOVEMENT - RIGHT

;--- USES: Z, r16, r17

MOV\_RIGHT:

push ZH

push ZL

push r16

push r17

push r22

push BOOLEAN

push LOOPCOUNTER

clr BOOLEAN

ldi LOOPCOUNTER, BLOCK\_SIZE

clr r22

rcall BORDER\_CHECK

rcall BLOCKED\_RIGHT

MOVING\_R:

dec LOOPCOUNTER

sbrc BOOLEAN, 0

rjmp END\_MOVR

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

add ZL, LOOPCOUNTER

ld r16, Z

com r16

lsl r16

com r16

st Z, r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

add ZL, LOOPCOUNTER

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

com r16

lsr r16

or r17, r16

lsl r16

com r16

and r17, r16

st Z, r17

END\_MOVR:

cpi LOOPCOUNTER, 0

brne MOVING\_R

ldi ZH, HIGH(ROTP)

ldi ZL, LOW(ROTP)

ld r16, Z

sbrs BOOLEAN, 0

lsl r16

st Z, r16

rcall WAIT\_RELEASE

pop LOOPCOUNTER

pop BOOLEAN

pop r22

pop r17

pop r16

pop ZL

pop ZH

ret

WAIT\_RELEASE:

sbic PINA, 0

rjmp WAIT\_RELEASE

sbic PIND, 2

rjmp WAIT\_RELEASE

sbic PINA, 1

rjmp WAIT\_RELEASE

ret

BORDER\_CHECK:

push LOOPCOUNTER

push r16

push r17

ldi LOOPCOUNTER, 0

CHECKING\_BORDER:

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

add ZL, LOOPCOUNTER

ld r16, Z

cpi r22, $01

breq CHECKING\_L

CHECKING\_R:

mov r17, r16

andi r17, $80 ; Border check

sbrs r17, 7 ; - Bättre eftersom generell lösning

ldi BOOLEAN, 1

rjmp END\_BORDER

CHECKING\_L:

andi r16, $01

sbrs r16, 0

ldi BOOLEAN, 1

END\_BORDER:

inc LOOPCOUNTER

cpi LOOPCOUNTER, BLOCK\_SIZE

brne CHECKING\_BORDER

pop r17

pop r16

pop LOOPCOUNTER

ret

;-------------------------------------

;--- CHECK IF BLOCKED BY BITS

;--- USES: Z, r16, r17

BLOCKED\_RIGHT:

push ZH

push ZL

push r16

push r17

push r20

push LOOPCOUNTER

clr LOOPCOUNTER

LOOP\_R:

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

add ZL, LOOPCOUNTER

ld r16, Z

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

add ZL, LOOPCOUNTER

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

com r16

mov r20, r16

add r17, r16

lsl r16

lsl r20

and r16, r17

cp r16, r20

breq END\_BRCHECK

ldi BOOLEAN, 1

END\_BRCHECK:

inc LOOPCOUNTER

cpi LOOPCOUNTER, BLOCK\_SIZE

brne LOOP\_R

pop LOOPCOUNTER

pop r20

pop r17

pop r16

pop ZL

pop ZH

ret

;-------------------------------------

;--- CHECK IF BLOCKED BY BITS

;--- USES: Z, r16, r17

BLOCKED\_LEFT:

push ZH

push ZL

push r16

push r17

push r20

push LOOPCOUNTER

clr LOOPCOUNTER

LOOP\_L:

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

add ZL, LOOPCOUNTER

ld r16, Z

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

add ZL, LOOPCOUNTER

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

com r16

mov r20, r16

add r17, r16

lsr r16

lsr r20

and r16, r17

cp r16, r20

breq END\_BLCHECK

ldi BOOLEAN, 1

END\_BLCHECK:

inc LOOPCOUNTER

cpi LOOPCOUNTER, BLOCK\_SIZE

brne LOOP\_L

pop LOOPCOUNTER

pop r20

pop r17

pop r16

pop ZL

pop ZH

ret

;-------------------------------------

;--- MUX

;--- USES: Z, r16, r17, MUXCOUNTER (r19)

MUX:

push ZH

push ZL

push r16

push r17

push r18

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, MUXCOUNTER

ld r16, Z

subi ZL, -8

ld r18, Z

ldi ZH, HIGH(LINE)

ldi ZL, LOW(LINE)

ld r17, Z

SPI\_LCD:

//BLUE

out SPDR, r17

rcall Wait\_Transmit

//GREEN

out SPDR, r17

rcall Wait\_Transmit

//RED

out SPDR, r17

rcall Wait\_Transmit

//KOLUMN (Satta värden visas ej)

;ld r18, Z

out SPDR, r16

rcall Wait\_Transmit

//BLUE

out SPDR, r17

rcall Wait\_Transmit

//GREEN

out SPDR, r17

rcall Wait\_Transmit

//RED

out SPDR, r17

rcall Wait\_Transmit

//KOLUMN (Satta värden visas ej)

;ld r18, Z

out SPDR, r18

rcall Wait\_Transmit

rjmp SEND\_BITS

Wait\_Transmit:

sbis SPSR,SPIF

rjmp Wait\_Transmit

ret

SEND\_BITS:

sbi PORTB, 0

cbi PORTB, 0

inc MUXCOUNTER

sbrc MUXCOUNTER, 3

clr MUXCOUNTER

cpi r17, $80

brne NOT\_0

ldi r17, $01

rjmp END\_MUX

NOT\_0:

lsl r17

END\_MUX:

st Z, r17

ldi ZH, HIGH(SEED)

ldi ZL, LOW(SEED)

ld r17, Z

inc r17

st Z, r17

pop r18

pop r17

pop r16

pop ZL

pop ZH

reti

;-------------------------------------

;--- GRAVITY - DROPS THE BLOCKS

;--- USES: Z, r16, r17, LOOPCOUNTER, r20

GRAVITY:

push ZH

push ZL

push r16

push r17

push r20

push LOOPCOUNTER

ldi LOOPCOUNTER, BLOCK\_SIZE

clr r20

call CHECK\_COLLISION

sbrc r20, 0 ; BOOLEAN

rjmp END\_GRAV

FALLING:

dec LOOPCOUNTER

ldi ZL, LOW(POSX)

ldi ZH, HIGH(POSX)

add ZL, LOOPCOUNTER

ld r16, Z

ldi ZL, LOW(POSY)

ldi ZH, HIGH(POSY)

add ZL, LOOPCOUNTER

ld r17, Z

inc r17

st Z, r17

ldi ZL, LOW(VMEM)

ldi ZH, HIGH(VMEM)

add ZL, r17

ld r17, Z

and r17, r16

st Z, r17

call UPDATE\_POS

cpi LOOPCOUNTER, 0

brne FALLING

END\_GRAV:

pop LOOPCOUNTER

pop r20

pop r17

pop r16

pop ZL

pop ZH

reti

;-------------------------------------

;--- UPDATE VMEM WITH NEW COORDINATES

;--- (ERASE OLD COORDINATES)

;--- USES: Z, r16, r17, r18

UPDATE\_POS:

push ZH

push ZL

push r16

push r17

push r18

UPDATING\_POS:

ldi ZL, LOW(POSX)

ldi ZH, HIGH(POSX)

add ZL, LOOPCOUNTER

ld r16, Z

ldi ZL, LOW(POSY)

ldi ZH, HIGH(POSY)

add ZL, LOOPCOUNTER

ld r17, Z

dec r17

ldi ZL, LOW(VMEM)

ldi ZH, HIGH(VMEM)

add ZL, r17

ld r17, Z

com r16

or r17, r16

st Z, r17

END\_UPDATE:

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

;-------------------------------------

;--- CHECK IF COLLISION

;--- IF YES -> BUILD NEW BLOCK

;--- USES: Z, r17, r18, r19

CHECK\_COLLISION:

push ZH

push ZL

push r17

push r18

push r19

push r16

push LOOPCOUNTER

clr LOOPCOUNTER

CHECKING\_COLL:

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

add ZL, LOOPCOUNTER

ld r18, Z+

ldi r16, $FF

cpi LOOPCOUNTER, BLOCK\_SIZE-1

breq C1

ld r16, Z

C1:

ldi ZL, LOW(POSY)

ldi ZH, HIGH(POSY)

add ZL, LOOPCOUNTER

ld r17, Z

cpi r18, $FF

breq NOT\_BOTTOM

cpi r17, $0F

breq HIT

NOT\_BOTTOM:

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

inc r17

add ZL, r17

ld r17, Z

com r16

add r17, r16

mov r19, r17

com r18 ; $EF -> $10 etc

or r19, r18

cp r17, r19

brne HIT

CHECK\_LOOP:

inc LOOPCOUNTER

cpi LOOPCOUNTER, BLOCK\_SIZE

brne CHECKING\_COLL

rjmp END\_CHECK

HIT:

sbi PORTD, 0

ldi r20, $01

rcall CHECK\_ROW\_FILLED

rcall CHECK\_IF\_LOST

rcall BUILD\_BLOCK

END\_CHECK:

cbi PORTD, 0

pop LOOPCOUNTER

pop r16

pop r19

pop r18

pop r17

pop ZL

pop ZH

ret

CHECK\_IF\_LOST:

push ZH

push ZL

push r16

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

inc ZL

ld r16, Z

cpi r16, $FF

breq END\_LOSS\_CHECK

LOST:

sbi PORTD, 3

;rcall GAME\_OVER

rcall VMEM\_INIT

END\_LOSS\_CHECK:

cbi PORTD, 3

pop r16

pop ZL

pop ZH

ret

;------------------------------------------

;--- CHECK IF ROW IS FILLED

;--- IF YES -> DELETE ROW, ROWS ABOVE FALL

;--- USES: Z, r16, r17, r18, r20

CHECK\_ROW\_FILLED:

push ZH

push ZL

push r16

push r17

push r18

push r20

ldi r20, 1

ldi ZH, HIGH(VMEM)

UPDATE\_LOOP:

ldi ZL, LOW(VMEM)

ld r18, Z ;(Sparar värdet)

add ZL, r20

ld r17, Z

cpi r17, $00

brne DONE\_ROW

mov r16, r20 ; Intern loop counter för FULL\_ROW\_FOUND

FULL\_ROW\_FOUND:

dec r16 ; Kolla raden ovanför den fulla

ldi ZL, LOW(VMEM)

add ZL, r16

ld r17, Z ; Z så ladda och öka ZL

inc ZL

inc r16 ; Återställ countern

st Z, r17

dec ZL ; Tillbaks till raden ovanför

ldi r17, $FF ; Nollställ raden

st Z, r17 ; Spara nollställningen

dec r16

cpi r16, 0

brne FULL\_ROW\_FOUND

sbi PORTD, 1 ; Spela ljud

DONE\_ROW: ; |

cbi PORTD, 1 ; |

inc r20

cpi r20, $10

brne UPDATE\_LOOP

pop r20

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

BUILD\_BLOCK:

push ZH

push ZL

push r16

push r17

push LOOPCOUNTER

clr LOOPCOUNTER

ldi ZH, HIGH(ROTP)

ldi ZL, LOW(ROTP)

ldi r16, $10

st Z, r16

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

clr r16

st Z, r16

ldi ZH, HIGH(SEED)

ldi ZL, LOW(SEED)

ld r16, Z

;com r16

mov r17, r16

com r17

MOD\_2:

lsr r17

inc LOOPCOUNTER

cpi LOOPCOUNTER, 5

brne MOD\_2

rol r17

com r17

sub r16, r17

rol r16

;eor r16, r17

sbrs r16, 6

rcall BUILD\_BLOCK\_Z2

sbrs r16, 5

rcall BUILD\_BLOCK\_Z1

sbrs r16, 4

rcall BUILD\_BLOCK\_PYRAMID

sbrs r16, 3

rcall BUILD\_BLOCK\_L1

sbrs r16, 2

rcall BUILD\_BLOCK\_L2

sbrs r16, 1

rcall BUILD\_BLOCK\_I

sbrs r16, 0

rcall BUILD\_BLOCK\_SQUARE

pop LOOPCOUNTER

pop r17

pop r16

pop ZL

pop ZH

ret

; ----------------------------------------

; -- BLOCKMINNE FÖR SKAPANDE AV BLOCKEN --

; -- USES: Z, r16, r17, LOOPCOUNTER --

; ----------------------------------------

BUILD\_BLOCK\_I:

push ZH

push ZL

push r16

push r17

push LOOPCOUNTER

clr LOOPCOUNTER

ldi r17, 1

ldi ZL, LOW(FIGURE)

st Z, r17

clr r17

BUILDING\_I:

ldi r16, $EF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

add ZL, LOOPCOUNTER

st Z, r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

add ZL, LOOPCOUNTER

st Z, r17

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, LOOPCOUNTER

st Z, r16

inc LOOPCOUNTER

inc r17

cpi LOOPCOUNTER, BLOCK\_SIZE

brne BUILDING\_I

FINISHED\_BUILD\_I:

pop LOOPCOUNTER

pop r17

pop r16

pop ZL

pop ZH

ret

BUILD\_BLOCK\_L1:

push ZH

push ZL

push r16

push r17

ldi r16, 2

ldi ZH, HIGH(FIGURE)

ldi ZL, LOW(FIGURE)

st Z, r16

clr r16

ldi r16, $E7

clr r17

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r16

ldi r17, $EF

st Z+, r17

st Z, r17

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

st Z+, r16

st Z+, r17

st Z, r17

clr r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

st Z+, r16

inc r16

st Z+, r16

inc r16

st Z, r16

pop r17

pop r16

pop ZL

pop ZH

ret

BUILD\_BLOCK\_L2:

push ZH

push ZL

push r16

push r17

ldi r17, $04

ldi ZL, LOW(FIGURE)

st Z, r17

clr r17

clr r17

clr r18

ldi r16, $CF

clr r17

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r16

ldi r17, $EF

st Z+, r17

st Z, r17

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

st Z+, r16

st Z+, r17

st Z, r17

clr r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

st Z+, r16

inc r16

st Z+, r16

inc r16

st Z, r16

pop r17

pop r16

pop ZL

pop ZH

ret

BUILD\_BLOCK\_SQUARE:

push ZH

push ZL

push r16

push r17

ldi r16, 8

ldi ZH, HIGH(FIGURE)

ldi ZL, LOW(FIGURE)

st Z, r16

clr r16

clr r17

clr r18

ldi r16, $E7

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r16

st Z+, r16

ldi r17, $FF

st Z, r17

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

st Z+, r16

st Z+, r16

st Z, r17

clr r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

st Z+, r16

inc r16

st Z+, r16

inc r16

st Z, r16

pop r17

pop r16

pop ZL

pop ZH

ret

BUILD\_BLOCK\_PYRAMID:

push ZH

push ZL

push r16

push r17

ldi r16, $10

ldi ZH, HIGH(FIGURE)

ldi ZL, LOW(FIGURE)

st Z, r16

clr r16

ldi r16, $EF

ldi r17, $CF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r16

st Z+, r17

st Z, r16

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

st Z+, r16

st Z+, r17

st Z, r16

clr r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

st Z+, r16

inc r16

st Z+, r16

inc r16

st Z, r16

pop r17

pop r16

pop ZL

pop ZH

ret

BUILD\_BLOCK\_Z1:

push ZH

push ZL

push r16

push r17

ldi r16, $20

ldi ZH, HIGH(FIGURE)

ldi ZL, LOW(FIGURE)

st Z, r16

clr r16

clr r17

clr r18

ldi r16, $EF

ldi r17, $E7

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r16

st Z+, r17

ldi r16, $F7

st Z, r16

ldi r16, $EF

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

st Z+, r16

st Z+, r17

ldi r17, $F7

st Z, r17

clr r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

st Z+, r16

inc r16

st Z+, r16

inc r16

st Z, r16

pop r17

pop r16

pop ZL

pop ZH

ret

BUILD\_BLOCK\_Z2:

push ZH

push ZL

push r16

push r17

ldi r16, $40

ldi ZH, HIGH(FIGURE)

ldi ZL, LOW(FIGURE)

st Z, r16

clr r16

clr r17

clr r18

ldi r16, $F7

ldi r17, $E7

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r16

st Z+, r17

ldi r16, $EF

st Z, r16

ldi r16, $F7

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

st Z+, r16

st Z+, r17

ldi r17, $EF

st Z, r17

clr r16

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

st Z+, r16

inc r16

st Z+, r16

inc r16

st Z, r16

pop r17

pop r16

pop ZL

pop ZH

ret

; --------------------------------------------

; -- ROTATIONSMINNE FÖR SKAPANDE AV BLOCKEN --

; -- USES: Z, r16, r17, LOOPCOUNTER --

; -- 1 = I | 2 = L1 | 4 = L2 | 8 = SQUARE | --

; -- | 10 = Z1 | 20 = Z2 | 40 = PYRAMID --

; --------------------------------------------

ROTATE:

push ZH

push ZL

push r16

push r17

push BOOLEAN ;Rensa några register som inte används

clr BOOLEAN

ldi ZH, HIGH(ROTP)

ldi ZL, LOW(ROTP)

ld r17, Z

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

subi ZL, -1 ; För hårt villkor?

ld r16, Z

cpi r16, $0F

breq END\_ROTATE

call BLOCKED\_RIGHT ; ROTATIONSKRAV

sbrc BOOLEAN, 0 ; |

rjmp END\_ROTATE ; |

call BLOCKED\_LEFT ; |

sbrc BOOLEAN, 0 ; |

rjmp END\_ROTATE ; |

; |

sbrc r17, 0 ; |

rcall BORDER\_L ; |

sbrc BOOLEAN, 0 ; |

rjmp END\_ROTATE ; |

; |

sbrc r17, 7 ; |

rcall BORDER\_R ; |

sbrc BOOLEAN, 0 ; |

rjmp END\_ROTATE ; |

ldi ZH, HIGH(FIGURE)

ldi ZL, LOW(FIGURE)

ld r17, Z

sbrc r17, 0

rcall ROTATE\_I

sbrc r17, 1

rcall ROTATE\_L1

sbrc r17, 2

rcall ROTATE\_L2

sbrc r17, 3 ;SQUARE

rjmp END\_ROTATE

sbrc r17, 4

rcall ROTATE\_PYRAMID

sbrc r17, 5

rcall ROTATE\_Z1

sbrc r17, 6

rcall ROTATE\_Z2

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

ld r17, Z

inc r17

mov r16, r17

lsr r16

sbrc r16, 1

clr r17

st Z, r17

END\_ROTATE:

; call WAIT\_RELEASE

pop BOOLEAN

pop r17

pop r16

pop ZL

pop ZH

reti

BORDER\_R:

clr BOOLEAN

rcall MOV\_LEFT

rcall BLOCKED\_LEFT

sbrc BOOLEAN, 0

rcall MOV\_RIGHT

ret

BORDER\_L:

clr BOOLEAN

rcall MOV\_RIGHT

rcall BLOCKED\_RIGHT

sbrc BOOLEAN, 0

rcall MOV\_LEFT

ret

COMPENSATE:

push ZH

push ZL

push r22

ldi ZH, HIGH(ROTP)

ldi ZL, LOW(ROTP)

ld r17, Z

com r20

com r23

ldi r22, $10

cp r17, r22

breq END\_COMP

cp r17, r22

brlo COMP\_R

COMP\_L:

lsl r20

lsl r22

lsl r23

cp r17, r22

brne COMP\_L

rjmp END\_COMP

COMP\_R:

lsr r20

lsr r22

lsr r23

cp r17, r22

brne COMP\_R

END\_COMP:

com r20

com r23

pop r22

pop ZL

pop ZH

ret

ROTATE\_I:

push ZH

push ZL

push r16

push r17

push r18

push r19

push r20

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

ld r18, Z

sbrc r18, 0

rjmp ROT\_I\_2

ROT\_I\_1:

ldi r20, $C7

rcall COMPENSATE

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z

st Z+, r17

st Z+, r20

st Z, r17

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

com r16

ld r17, Z

or r17, r16

st Z+, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r16

st Z, r17

rjmp END\_ROTI

ROT\_I\_2:

ldi r20, $EF

ldi ZH, HIGH(ROTP)

ldi ZL, LOW(ROTP)

ld r17, Z

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

subi ZL, -1 ; Bara figur på andra raden i minnet

ld r16, Z

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

com r16

ld r17, Z

and r17, r20

st Z+, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

and r17, r20

st Z, r17

ldi ZH, HIGH(POSX) ; Spara nya värden i POSX efter VMEM?

ldi ZL, LOW(POSX)

st Z+, r20

st Z+, r20

st Z, r20

END\_ROTI:

pop r20

pop r19

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

ROTATE\_L1:

push ZH

push ZL

push r16

push r17

push r18

push r19

push r20

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

ld r18, Z

cpi r18, 0

breq ROT\_L1\_1

cpi r18, 1

breq ROT\_L1\_2

rjmp ROT\_CL1

ROT\_L1\_1:

ldi r20, $C7

ldi r23, $F7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r23

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r17

st Z+, r20

st Z, r23

rjmp END\_ROTL1

ROT\_L1\_2:

ldi r20, $EF

ldi r23, $CF

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

and r17, r20

st Z+, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r23

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r20

st Z+, r20

st Z, r23

rjmp END\_ROTL1

ROT\_CL1:

cpi r18, 3

breq ROT\_L1\_4

ROT\_L1\_3:

ldi r20, $C7

ldi r23, $DF

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r23

st Z+, r20

st Z, r17

rjmp END\_ROTL1

ROT\_L1\_4:

ldi r20, $EF

ldi r23, $E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z+, r17

ld r17, Z

and r17, r20

st Z, r17

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r23

st Z+, r20

st Z, r20

END\_ROTL1:

pop r20

pop r19

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

ROTATE\_L2:

push ZH

push ZL

push r16

push r17

push r18

push r19

push r20

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

ld r18, Z

cpi r18, 0

breq ROT\_L2\_1

cpi r18, 1

breq ROT\_L2\_2

rjmp ROT\_CL2

ROT\_L2\_1:

ldi r20, $C7

ldi r23, $F7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

;and r17, r23

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r23

st Z+, r20

st Z, r17

rjmp END\_ROTL2

ROT\_L2\_2:

ldi r20, $EF

ldi r23, $E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z+

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z+, r17

ld r17, Z

and r17, r23

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r20

st Z+, r20

st Z, r23

rjmp END\_ROTL2

ROT\_CL2:

cpi r18, 3

breq ROT\_L2\_4

ROT\_L2\_3:

ldi r20, $C7

ldi r23, $DF

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

;and r17, r23

st Z+, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r23

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r17

st Z+, r20

st Z, r23

rjmp END\_ROTL2

ROT\_L2\_4:

ldi r20, $EF

ldi r23, $CF;$E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

;or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z, r17

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r23

st Z+, r20

st Z, r20

END\_ROTL2:

pop r20

pop r19

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

ROTATE\_PYRAMID:

push ZH

push ZL

push r16

push r17

push r18

push r19

push r20

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

ld r18, Z

cpi r18, 0

breq ROT\_PYRAMID\_1

cpi r18, 1

breq ROT\_PYRAMID\_2

rjmp ROT\_CL3

ROT\_PYRAMID\_1:

ldi r20, $C7

ldi r23, $EF

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z+, r17

ld r17, Z

or r17, r16

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r23

st Z+, r20

st Z, r17

rjmp END\_ROTP

ROT\_PYRAMID\_2:

ldi r20, $EF

ldi r23, $E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r23

st Z+, r17

ld r17, Z

and r17, r20

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r20

st Z+, r23

st Z, r20

rjmp END\_ROTP

ROT\_CL3:

cpi r18, 3

breq ROT\_PYRAMID\_4

ROT\_PYRAMID\_3:

ldi r20, $C7

ldi r23, $EF

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z+, r17

ld r17, Z

or r17, r16

and r17, r23

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r17

st Z+, r20

st Z, r23

rjmp END\_ROTP

ROT\_PYRAMID\_4:

ldi r20, $EF

ldi r23, $CF

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

and r17, r20

st Z+, r17

ld r17, Z

or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r19

and r17, r20

st Z, r17

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r20

st Z+, r23

st Z, r20

END\_ROTP:

pop r20

pop r19

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

ROTATE\_Z1:

push ZH

push ZL

push r16

push r17

push r18

push r19

push r20

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

ld r18, Z

sbrc r18, 0

rjmp ROT\_Z1\_2

ROT\_Z1\_1:

ldi r20, $CF

ldi r23, $E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r18, Z+

ld r19, Z

com r16

com r18

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r18

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r23

st Z+, r20

st Z, r17

rjmp END\_ROTZ1

ROT\_Z1\_2:

ldi r20, $EF

ldi r23, $E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r23

st Z+, r17

ld r17, Z

;or r17, r16

com r20

lsr r20

com r20

and r17, r20

st Z, r17

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

mov r19, r20

com r20

lsl r20

com r20

st Z+, r20

st Z+, r23

st Z, r19

END\_ROTZ1:

pop r20

pop r19

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

ROTATE\_Z2:

push ZH

push ZL

push r16

push r17

push r18

push r19

push r20

ldi ZH, HIGH(ROT)

ldi ZL, LOW(ROT)

ld r18, Z

sbrc r18, 0

rjmp ROT\_Z2\_2

ROT\_Z2\_1:

ldi r20, $F3

ldi r23, $E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r18, Z+

ld r19, Z

com r16

com r18

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r23

st Z+, r17

ld r17, Z

or r17, r18

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

st Z, r17

ldi r17, $FF

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

st Z+, r23

st Z+, r20

st Z, r17

rjmp END\_ROTZ1

ROT\_Z2\_2:

ldi r20, $F7

ldi r23, $E7

rcall COMPENSATE

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

ld r16, Z+

ld r19, Z

com r16

com r19

ldi ZH, HIGH(POSY)

ldi ZL, LOW(POSY)

ld r17, Z

ldi ZH, HIGH(VMEM)

ldi ZL, LOW(VMEM)

add ZL, r17

ld r17, Z

or r17, r16

and r17, r20

st Z+, r17

ld r17, Z

or r17, r19

and r17, r23

st Z+, r17

ld r17, Z

com r20

lsl r20

com r20

and r17, r20

st Z, r17

ldi ZH, HIGH(POSX)

ldi ZL, LOW(POSX)

mov r19, r20

com r20

lsr r20

com r20

st Z+, r20

st Z+, r23

st Z, r19

END\_ROTZ2:

pop r20

pop r19

pop r18

pop r17

pop r16

pop ZL

pop ZH

ret

HW\_INIT:

ldi r17,(1<<DDB5)|(1<<DDB7)|(1<<DDB4)|(1<<DDB0) ; Set MOSI, SCK, SS, PB0 output, all others input

out DDRB,r17

ldi r17,(1<<SPE)|(1<<MSTR)|(0<<SPR0) ; Enable SPI, Master, set clock rate fck/4

out SPCR,r17

cbi PORTB, 0

ldi r16, (1 << CS01) ; MUX

out TCCR0, r16

ldi r16, (1 << CS11 | 0 << CS10 | 0 << CS12| 1 << WGM12) ; fclk / 8 Gravity

out TCCR1B, r16

ldi r16, (1 << TOIE0 | 1 << OCIE1B)

out TIMSK, r16

ldi r16,(1<<ISC01)|(0<<ISC00)

out MCUCR, r16

ldi r16,(1<<INT0)

out GICR, r16

ldi r17, $3D

ldi r16, $09

out OCR1AH, r17

out OCR1AL, r16

ldi r16, $0B

out DDRD, r16

clr r16

out DDRA, r16

sei

ret

GG: .db $C3, $DF, $DF, $DF, $D3, $DB, $DB, $C3, $C3, $DF, $DF, $DF, $D3, $DB, $DB, $C3

# Bilaga 2

INIT:

clr r16

out DDRA, r16

ldi r16, $FF

out DDRC, r16

START:

sbic PINA, 1 ; PIN 39

call FULL\_ROW

sbic PINA, 2 ; PIN 38

call LOST

END\_START:

rjmp START

FULL\_ROW:

ldi r16, $10

SOUND\_LOOP1:

sbi PORTC, 1

call DELAY1

cbi PORTC, 1

call DELAY1

dec r16

cpi r16, 0

brne SOUND\_LOOP1

ret

DELAY1:

push r18

push r17

ldi r18, 10

delayYttreLoop1:

ldi r17,$7D

delayInreLoop1:

dec r17

brne delayInreLoop1

dec r18

brne delayYttreLoop1

pop r17

pop r18

rcall WAIT

ret

LOST:

ldi r16, $64

SOUND\_LOOP2:

sbi PORTC, 1

call DELAY2

cbi PORTC, 1

call DELAY2

dec r16

cpi r16, 0

brne SOUND\_LOOP2

rjmp LOST2

DELAY2:

push r18

push r17

ldi r18, 10

delayYttreLoop2:

ldi r17,$DF

delayInreLoop2:

dec r17

brne delayInreLoop2

dec r18

brne delayYttreLoop2

pop r17

pop r18

ret

LOST2:

call NO\_SOUND

ldi r16, $7D

SOUND\_LOOP3:

sbi PORTC, 1

call DELAY3

cbi PORTC, 1

call DELAY3

dec r16

cpi r16, 0

brne SOUND\_LOOP3

rjmp LOST3

DELAY3:

push r18

push r17

ldi r18, 10

delayYttreLoop3:

ldi r17,$EF

delayInreLoop3:

dec r17

brne delayInreLoop3

dec r18

brne delayYttreLoop3

pop r17

pop r18

ret

LOST3:

call NO\_SOUND

ldi r16, $FF

SOUND\_LOOP4:

sbi PORTC, 1

call DELAY4

cbi PORTC, 1

call DELAY4

dec r16

cpi r16, 0

brne SOUND\_LOOP4

ret

DELAY4:

push r18

push r17

ldi r18, 10

delayYttreLoop4:

ldi r17,$Ff

delayInreLoop4:

dec r17

brne delayInreLoop4

dec r18

brne delayYttreLoop4

pop r17

pop r18

rcall WAIT

ret

WAIT:

sbic PINA,1

rjmp WAIT

sbic PINA,2

rjmp WAIT

ret

NO\_SOUND:

ldi r18, 6

ldi r19, 19

ldi r20, 174

L1: dec r20

brne L1

dec r19

brne L1

dec r18

brne L1

ret