# Teknisk rapport
# Grafisk räknare

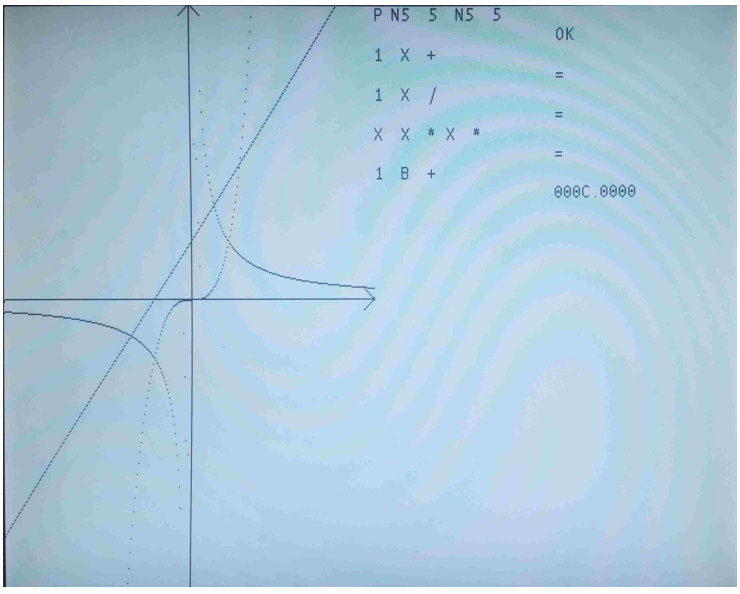
23 maj 2016

**Sammanfattning**

## 1 Inledning

Bakgrund, syfte, källor!?

## 2 Apparaten

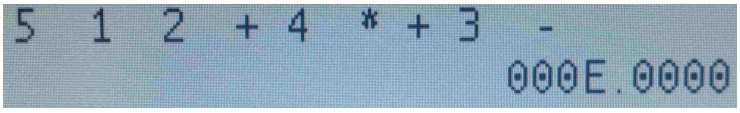



Figur 1: Bild på grunkan **TODO: Low quality because sshfs is slow. Change to HQ when done**

Apparaten är en hexadecimal grafräknare med reverse polish-notation.

Börja gärna med en bild på bygget och redgör för hur apparaten används. Detta blir en kombinerad presentation av konstruktionen och användarhandledning.

## 2.1 Beräkningar och inmatning




Figur 2: Beräkningsexempel **TODO: Low quality because sshfs is slow. Change to HQ when done**

En vanlig beräkning genomför genom att man skriver ett uttryck och avslutar med enter.

Siffror skrivs in med numpad eller sifferraden, tillsammans med knapparna `A-F`. För att få negativa tal börjar man med ett `N` innan siffrorna. Decimalen fås med punkt. Avsluta siffor med mellanslag. Notera att endast siffror upp till 32 bitars tvåkomplement stöds, med jämn uppdelning mellan heltalsdelen och decimaldelen. Med andra ord går heltalsdelen från $-8000$ och $+7FFF$, medan decimaldelen går mellan 0000 och $FFFF$.

Operander matas in via knapparna på numpaden. Efter dessa behövs inget mellanslag.

## 2.2 Grafritning

En funktion för uppritning matas in likadant som en vanlig beräkning, med undantag för att en eller flera siffror ersätts med `X`. Som standard kommer den inmatade funktionen ritas mellan $-10$ och 10 i både X- och Y-led.

Vill man ändra utritningsområdet så görs det genom att man matar in

```
P Xmin Xmax Ymin Ymax
```

där Xmin, Xmax, Ymin och Ymax skrivs in som vanliga siffror. P räknas som operator och ska alltså inte ha något mellanslag efter sig. När den fjärde siffran matats in så töms grafen och nästa funktion som matas in kommer att ritas på det nya utritningsområdet.

# 3 Teori

Här kan ett videoprojekt beskriva videoformatet, ett MIDI-projekt redogöra för MIDI-standarden ...

### 3.1 Reverse polish

Reverse polish är en notation för aritmetiska uttryck utan parenteser. Även känt som postfixnotation, eftersom operatorn hamnar efter operanderna. Exempelvis skrivs $1 + 2$ som $12+$. Har man flera operatorer så skrivs operatorn direkt efter andra operanden, exempelvis $1-2+3$ blir $12-3+$. Man läser alltså från vänster till höger. När det kommer en operand så appliceras den på de två värdena som kom innan, och resultatet läggs tillbaka på operandens plats. Ett mer avancerat exempel med parenteser är $5 + ((1 + 2) * 4) - 3$ som blir $512 + 4 * +3-$.[1]

## 4 Hårdvaran

Börja med ett översiktligt blockschema, gå vidare till mera detaljerade blockschemor. Vill man rita figurer och blockschemor på datorn så är Inkscape ett bra alternativ. Handritade figurer och blockschemor är acceptabelt, bara de är snygga och väl läsbara, MEN dessa ska då vara inscannade, dvs avfotograferade figurer och blockschemor är INTE godtagbart. Beskriv sedan hur de olika blocken fungerar. Tänk på läsbarheten och växla mellan figurer och text. Den här typen av text är ganska grafisk", då nästan varje mening syftar in i en figur. Var därför noga med att text och figurer stämmer överens.

### 4.1 Översikt

### 4.2 CPU

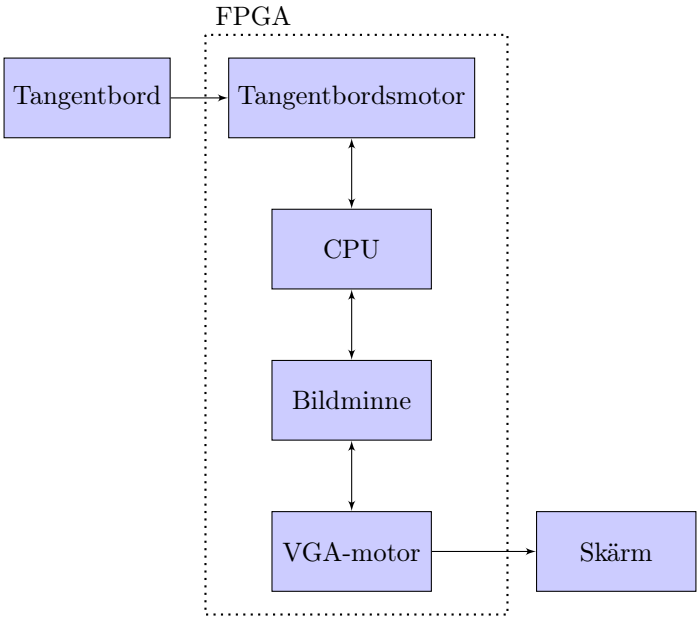
CPU:n är baserad på icke-pipelinade Björn Lindskogs-datorn. Huvudminnet, och de allra flesta register inklusive de generella, är 32 bitar långa.

#### 4.2.1 mPC

Microprogramräknaren. Microninstruktioner har fyra bitar att modifiera räknaren med, vilket sker på positiv klockflank. Koderna gör som följer:

| **Kod** | **Funktion** |
|---|---|
| `0001` | mPC := ADR |
| `0010` | mPC := K2(<Given mod>) |
| `0011` | mPC := K1(<Given Instruktion>) |
| `1000` | mPC := ADR if flag_X == `1` else mPC++ |
| `1001` | mPC := ADR if flag_N == `1` else mPC++ |
| `1010` | mPC := ADR if flag_Z == `1` else mPC++ |
| `1011` | mPC := ADR if flag_C == `1` else mPC++ |
| `1100` | mPC := ADR if flag_V == `1` else mPC++ |
| `Others` | mPC++ |

[1] Källa: `https://en.wikipedia.org/wiki/Reverse_Polish_notation`




Figur 3: Översiktligt blockschema.

K2 och K1 är minnen som binder en given mods eller instruktions binärrepresentation till raden i microminnet som har hand om dess del i hämtfasen respektive exekveringsfasen.

#### 4.2.2 IR

Instruktionsregistret, 32 bitar, håller i en kopia av instruktionen från huvudminnet som körs för tillfället. Den kan vidare delas upp i OP, GRx och MM som är den instrukionens instruktionskod, GRx-argument samt valt mod.

Man kan i microkoden skriva och läsa till och från IR genom att ange busskoden `0001` i från-buss respektive till-bus fältet. Att skriva till IR påverkar inte huvudminnet, och därmed inte den egentliga programkoden.

#### 4.2.3 PC

Programräknaren, håller koll på instruktion i p_mem som ska exkaveras. Registret är 22 bitar långt, då detta är fullt tillräckligt för alla rimligt stora program, samt att addressfältet hos en instruktion inte har plats för större tal på sin addressdel: de sista 22 bitarna.

PCsig, bit nummer 18 i microinstruktionen, kan påverka PC. Ett värde på 0 håller PC oförändrad, medan ett värde på 1 inkrementerar den.

PC kan skrivas till via bussen. En från-buss-signal på `0011` skriver de 22 lägsta bitarna av DATA_BUS till PC. Den kan även läsas ifrån genom att ange

samma busskod på till-buss delen i microminnet.

#### 4.2.4 ASR

Addressregistret används i microprocessorn för att peka ut addressen till raden som ska passera som argument. Detta sätts i hämtfasen och beror på vilket mod den dåvarande instruktionen körs med.

Liksom PC är registret av samma anledning 22 bitar långt.

ASR kan läses från och skrivas till via busskoden `0100`.

#### 4.2.5 GRx

De åtta generella registrena: GR0 till och med GR7.

Dessa register är 32 bitar långa och används för att främst hålla i temporära värden åt användaren.

Vektorn g_reg håller i de åtta registren, och ett av de väljs genom signalen GRx. Denna är bitar 26 till och med 24 hos programinstruktionen som körs för tillfället.

Att skriva eller läsa till det valda generella registret görs via busskuden `1000`. Det är den som valts i GRx som då interageras med.

#### 4.2.6 p_mem

Vårt programminne, 32 bitar långt. Då programminne och dataminne är samma minne så syftas här på det delade minnet. Genom att skriva eller läsa härifrån via busskoden `0010` så kan huvudminnet läsas eller modifieras. Raden i minnet som opereras på utpekas ASR, som sätts automatiskt i hämtfasen.

Det är värt att notera att programminnet initialt sätts till konstanten p_mem_c, där då programkod skrivs in.

#### 4.2.7 save_at_p/save_at_b/data_out_picmem/data_out_bitmap

Signaler för att skriva till bild och bitmapminne. Via busskoden `0111` skickas de sista 8 bitarna i `DATA_BUS` via `data_out_picmem` och `ASR` till `save_at_p`. Via busskoden `0110` skickas den sista biten i `DATA_BUS` via `data_out_bitmap` och `ASR` till `save_at_b`. Samtidigt aktiveras en (aktivt hög) write-enable-signal, `web` vid skrivning till bitmapminnet och `wep` vid skrivning till bildminnet. Dessa nollställs inte, utan används bara för att undvika odefinierade signaler innan första skrivningen.

#### 4.2.8 AR/ALU

I processen titulerad AR finns koden som utgör enhetens ALU.

Här modifieras ackumulatorregistret, AR, (32 bitar), och de fem flaggorna. En rad olika operationer kan utföras på positiv klockflank beroende på koden hos ALU-signalen; de fem första bitarna i mikrominnesinstruktionen som för tillfället körs.

Vid matematiska operationer tolkas AR och bussens värde som ett signerat tvåkomplementstal.

AR:s värde kan senare läggas på bussen genom att ange koden `0101` i mikrokoden. Den kan dock ej skrivas till direkt.

Nedan följer en tabell över alla ALU-operationer: deras koder, funktion, och modifierade flaggor.

| **Kod** | **Funktion** | **Flaggor påverkade** |
|---|---|---|
| `00000` | `<Ingenting>` | |
| `00001` | AR := BUS | |
| `00010` | AR := BUS' | |
| `00011` | AR := 0 | |
| `00100` | AR := AR+BUS | `X C V N Z` |
| `00101` | AR := AR-BUS | `X C V N Z` |
| `00110` | AR := AR AND BUS | `C V N Z` |
| `00111` | AR := *signed_to_real*(AR) (16 skift vänster) | |
| `01000` | AR := *real_to_signed*(AR) (16 skift höger) | |
| `01001` | AR := AR ASR BUS | `X C V N Z` |
| `01010` | AR := AR ASL BUS | `X C V N Z` |
| `01011` | AR := AR+BUS (Ekvivalent med `00100`) | `X C V N Z` |
| `01100` | AR := AR-BUS (Ekvivalent med `00101`) | `X C V N Z` |
| `01101` | AR := AR*BUS (fixed point) | |
| `01110` | `<Undefined>` | |
| `01111` | AR := AR LSR BUS | `X C N Z` |
| `10000` | AR := AR LSL BUS | `X C N Z` |
| `Others` | `<Undefined>` | |

TODO: Hälfen av det här gör ju identiska saker eller används aldrig (antingen i assembly eller inte ens i mikrokod)

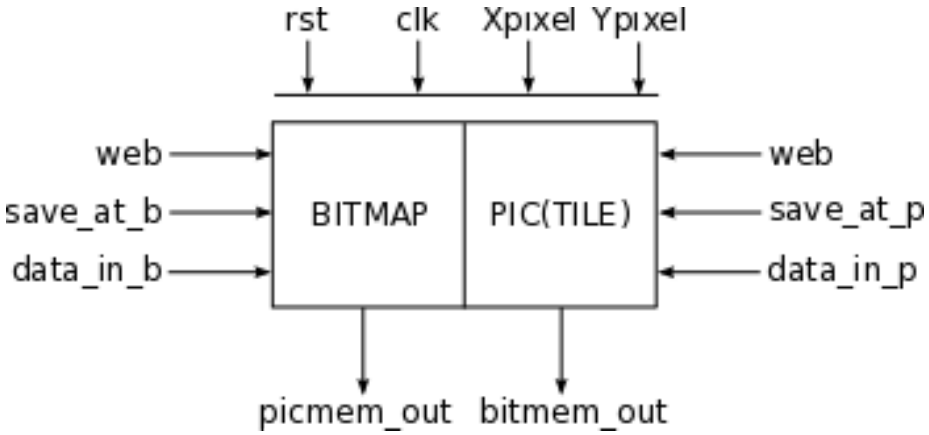
## 4.3 Tangentbordsmotor

## 4.4 Bildminne

Bildminnet är uppdelat i två kolumner, där ena hälften använder tiles och andra hälften använder en bitmap. Bitmapminnet innehåller 320x480 bitars blockram för en svartvit representation av alla pixlar på vänstra skärmhalvan. Bildminnet för tiles innehåller 40x30 rader.

Varje del har en write_enable-signal, en save_at-signal och en data_in-signal. När write_enable ettställs ersätts innehållet på adressen save_at med data_in. Ingen kontroll av att adresserna ligger inom minnet sker, utan det är upp till programmeraren.

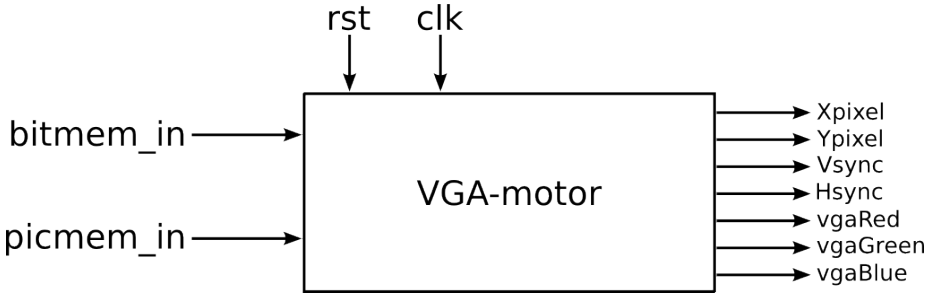
`Xpixel` och `Ypixel` används för att bestämma om `picmem_out` eller `bitmem_out` ska uppdateras från respektive minne, för att undvika overflow i minnena. Dessa två signaler innehåller alltså värdet för pixeln som ska ritas ut, beroende på vilken hälft av skärmen vi befinner oss i.




Figur 4: Blockschema över bildminnet

### 4.5 VGA-motor

TODO: Behövs båda?

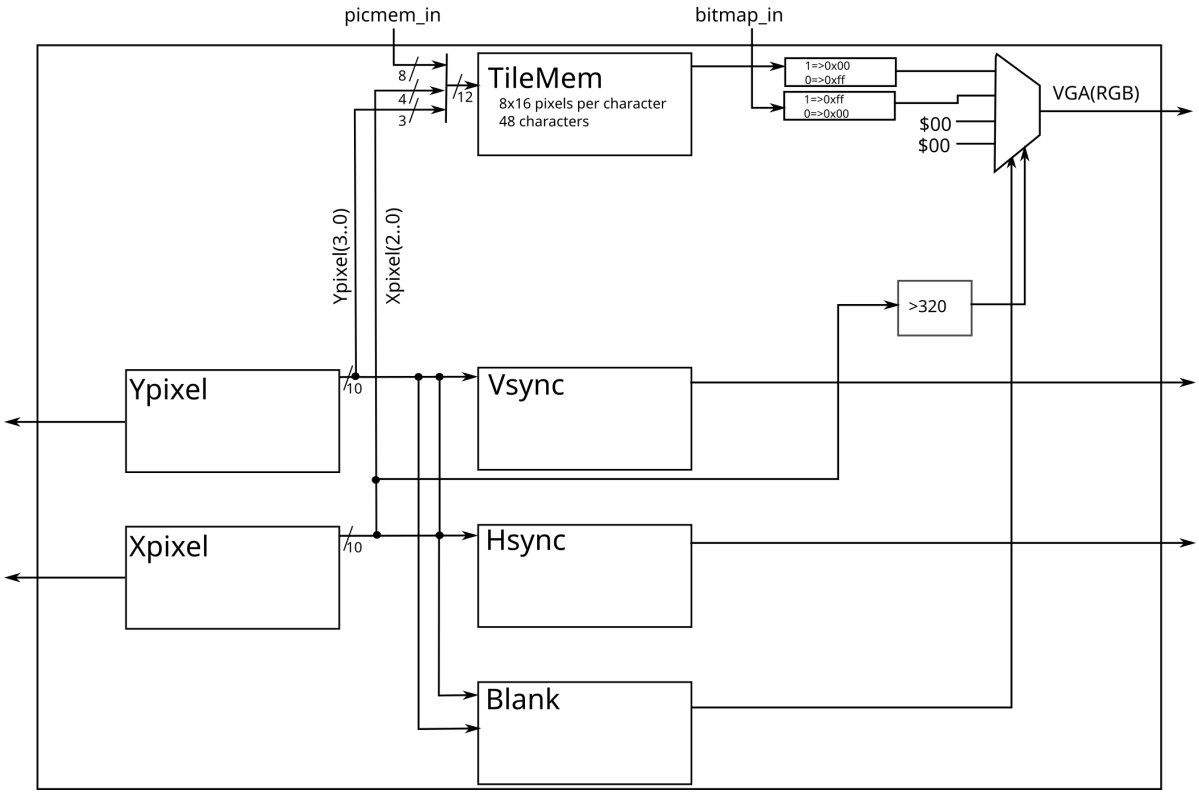



Figur 5: Översiktligt blockschema över VGA-motorn

VGA-motorn är ansvarig för att ta informationen från bildminnet och översätta detta till giltig VGA-output i form av RGB-signaler och H/V-sync för 640x480 pixlar i 60Hz. Den använder en klockfrekvens på 25Mhz. Denna används för att räkna upp `Xpixel` och `Ypixel`. Pixelräknarna används sedan för att generera `Vsync`, `Hsync` och `Blank`.

För varje pixel måste också ett utvärde beräknas. På vänster hälft (`Xpixel` $<$ 320) av skärmen innehåller `bitmem_in` information om ifall pixeln är svart (nolla) eller vit (etta). På höger hälft (`Xpixel` $>$ 320) innehåller `picmem_in` information om vilken tile som ska ritas upp. Se 4.4 för hur dessa uppdateras.

VGA-motorn har ett tileminne, `tileMem`, som innehåller 48 tecken (A-Ö, 0-9, samt ett par specialtecken) på 8x16 bitar vardera. Med hjälp av `picmem_in` som väljer ut en tile, de sista fyra bitarna av `Xpixel` och de sista tre bitarna av `Ypixel` pekas en pixel i tileminnet ut. I `tileMem` är en svart pixel en etta och




Figur 6: Detaljerat blockschema över VGA-motorn

en vit pixel en nolla.

En svart pixel fyller alla bitar i `vgaRed`, `vgaGreen` och `vgaBlue` med nollor, medans en vit pixel fyller dem med ettor. Om man ska läsa från `tileMem` eller `bitmem_in` bestäms även här av `Xpixel`. En blanksignal ger samma resultat som en svart pixel.

# 5 Mjukvara

Använd de här tabellerna för ”subrutiner”

| GRx | Input |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| Others | Används ej till input |

| GRx | Output |
|---|---|
| 0 | Resultat |

## 5.1 Hjälpmedel

### 5.1.1 Sillescript

Är ett relativt avancerat korsassembler-språk, vars kompilator är skriven i Python. I detta den stora majoriteten av koden är skriven.

Förutom att innehålla översättningar av alla instruktionsnamn till deras respektive binärkod finns även labels, kommentarer, if-satser (med support för else), while-satser, include-statements, och lite annat syntaktiskt socker.

Nedan följer ett utdrag ur koden som används för att byta ut ”X” i ett funktionsuttryck mot ett visst värde.

```
# Modify function: Replace X'es
load$ 0,0 # Loop counter
while 0 < &inputLength # GR0 < number of input elements

      # Put the value &nextInputType is
      #   pointing to on GR2.
      load~ 2,&nextInputType

      if 2=$2 # If value is X
        # Replace the X in the function with proper Xval
        store~ 1,&nextInput
      end if

      ...

      # Increment loop counter
      add$ 0,1

      ...
```

```
end while

# Labels this row as inputLength. Start value: 8.
inputLength: sli 8
```

Syntax som bär uppmärksammas:

| **Syntax** | **Beskrivning** |
|---|---|
| # | Kommentar. |
| : | Labeldefinition. ”Döper” raden det är skrivet på till labelnamnet som ges innan kolonet. |
| Jämförelser | Vänsterledet avser alltid ett GRx. Siffran indikerar vilken av de 8. Högerledet är mer flexibelt, och kan ändras genom att använda tecknen nedan. Finns support för =, **!**, > och <. |
| $ | Indikator. Säger att instruktionen skall körs med omedelbart mod. Om tecknet används i en jämförelse så indikerar det att följande värde är en literal. |
| ~ | Indikator. Säger att instruktionen skall körs med indirekt mod. |
| & | Indikator. Säger att det som följer är ett labelnamn. Detta namn översätts under kompilering till en korrekt address. |
| Ingen indikator | Säger att instruktionen skall körs med direkt mod. Om högerledet i en jämförelse saknar sådant tecken så indikerar det att värdet är en adress, och värdet sparat på adressen ska användas i jämförelsen. |

Nedan följer en lista på instruktioner som språket stödjer. Alla tar en mod (given via indikator), ett GRx (även om det inte används), och en address, literal, eller label.

- load
- store
- add
- addf
- sub
- subf
- and
- asl
- asr
- jmp
- lsr
- lsl
- storep
- rc
- storeb
- sli
- include

Sli (”**s**et **li**ne”) och include är speciella, och har inte samma syntax som resten av instruktionerna; båda tar endast ett argument, och har varken support för moder, labels, eller indikatorer. Detta då de körs i kompileringstid, snarare än som maskinkod.

Sli följs av ett värde, och sätter raden som instruktionen befinner sig på till binärrepresentationen av det värdet.

Include följs av ett filnamn, och kopierar innehållet av den filen till filen som kompileras. En fil kan som mest inkluderas en gång. Flera inklusioner ger varningar och ignoreras.

För att faktiskt kompilera en Sillescript-fil så kan följande konsolkommando användas (då man sitter i projektets `sillescript2`-mapp):

```
python compiler.py filename
```

Detta skriver ut kompilerad output i konsolen, som sedan är ämnat att kopieras in i `cpu.vhd`-filen, på rätt plats.

Pythonscriptet `compileAndLoad.py` kan dock göra detta åt dig; den kompilerar en fil och skriver sedan in resultatet i `cpu.vhd`.

Notera att detta är i allra högsta grad specialanpassad för detta projekt, så omstrukturering av mappar, eller större ändringar i `cpu.vhd` kan få scriptet att sluta funka. Körs via:

```
python compileAndLoad.py filename
```

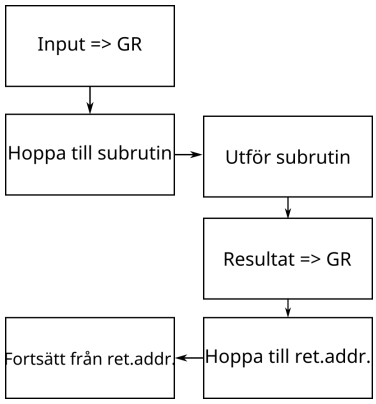
Scriptet sätter även en begränsning för när PM ska laddas med information från programminnet. Krävs för att det inte ska gå out of range när `ASR` sätts till något högre än storleken på `p_mem`. Används inte `compileAndLoad.py` så bör denna begränsning sättas för hand.

## 5.2 Testbänk

Testbänken har förutom att generera en klocka på 100MHz även support för att skriva ut VGA-outputen tillsammans med tidsangivelser till en textfil. Denna kan sedan simuleras i en VGA-simulator.

## 5.3 ”Subrutiner”

En subrutinsliknande struktur används för flera delar av programmet. Detta görs genom att ladda in input, däribland en returadress, till GR-register. Därefter hoppar den som vill kalla på ”subrutinen” till ”subrutinens” label. ”Subrutinen” använder inputen för att utföra sin uppgift. Om returvärden behövs så sparar den det till ett eller flera förutbestämda GR. Sedan hoppar den till den specificerade returadressen. ”Subrutiner” tar ingen hänsyn till att spara innehållet i register, utan det är anroparens uppgift att spara undan om det behövs. Dock används inga platser i dataminnet utanför dess egna block (TODO: Är detta sant?). Det finns i de flesta fall demoprogram för subrutiner vid namn `subrutinsnamn_demo`.




Figur 7: Användandet av en ”subrutin”.

## 5.4 Omvandla inmatning till tal

För att omvandla en inmatad följd av siffror till ett enda tal i minnet använder vi koden i `text2hex`. Detta program väntar tills en giltig tangent registreras av tangentbordsmotorn (dvs. instruktionen `RC` läser in något annat värde än x$FF$) och avgör sedan om inmatningen är en operand (x0 - x$F$), operator ($+$, $-$, $*$, $/$, $=$), en särskild modifierare ($P$, $N$, $R$) eller en särskild karaktär såsom $X$ eller mellanslag.

För att konstruera ett tal av en sifferföljd skiftas inmatningen ett lämpligt antal steg till vänster (16 steg för ett $4 \times 4$ fixpunktstal) så att inmatningen hamnar på de första fyra bitarna av heltalsdelen. Det tal som redan ligger i minnet skiftas sedan fyra steg till vänster och de två talen adderas. Se bilaga A för ett exempel på denna process. Inmatning av decimaler fungerar på stort sett samma sett, men den nya siffran skiftas istället in till höger om punkten.

Den särskilda modifieraren $N$ ($N$egative) kan användas för att ange att inmatningen ska tolkas som ett negativt tal, dvs inmatningsföljden ”`N5`” betyder att värdet $-5$ ska sparas.

## 5.5 Calc

Koden i `calc` är huvudloopen i programmet, som inte hanterar enskilda inmatade siffror utan behandlar istället de hela tal, operatorer och modifierare som skickas vidare från `text2hex` (se sektion 5.4). Här avgörs det huruvida inmatningen ska läggas på evalueringsstacken (om tecknet är operand, operator, eller variabel $X$), tolkas som en plotparameter (om inmatningen inleds med $P$), eller är en särskild karaktär såsom $R$ (som rensar skärmen).

Detta bestäms i `calc` som sedan går vidare till lämplig ’subrutin’ såsom `evaluate` eller `plot`. Dessa subrutiner återvänder sedan till `calc` som återställer relevanta variabler och register för nya inmatningar.

## 5.6 Beräkningar

### 5.6.1 Reverse polish

| GRx | Input |
|---|---|
| 0 | Pekare till lista med värden |
| 1 | Pekare till lista med typer |
| 2 | Storlek på listor |
| 3 | Returadress |
| Others | Används ej till input |

| GRx | Output |
|---|---|
| 0 | Resultat av beräkning |
| 1 | Beräkning lyckades |

Listan som pekare GR0 pekar på innehåller alla värden för beräkningen i omvänd polsk notation. Operander kommer att sparas som sin adress i tileminnet, alltså 43=+, 44=-, 45=* och 46=/. För att särskilja operander från vanliga siffror används opvektorn som GR1 pekar på. Om det står 0 så är innehållet på samma index i värdesvektorn en siffra, om det står något annat så tolkas det som en operand. I övrigt följer evalueringen algoritmen från Wikipedia. Stacken har en maxstorlek på 16 rader, därefter har den odefinierat resultat.

När beräkningen är slutförd ligger resultatet i GR0 och en nolla i GR1. Ligger det någonting annat i GR1 så har beräkningen gått fel (felaktig input).

### 5.6.2 Eval_fn

Byter ut ”X” i ett uttryck mot nya värden, baserat på vilken pixel i X-led som nu ritas, och räknar sedan ut Y-värdet av funktionen vid den pixelkolumnen.

| GRx | Input |
|---|---|
| 0 | Pekare till lista med värden |
| 1 | Pekare till lista med typer |
| 2 | Storlek på listor |
| 3 | Nuvarande X-pixel (heltal) |
| 4 | Minsta X-värde på intervall (fixed-point) |
| 5 | Största X-värde på intervall (fixed-point) |
| 6 | Returadress |
| Others | Används ej till input |

| GRx | Output |
|---|---|
| 0 | Resultat av beräkning: Y-värde (fixed-point) |
| 1 | 0 om lyckad beräkning, 1 annars |

Värdelista tolkas på samma sätt som i *Reverse polish*. Typlistan är också likadan, förutom skillnaden att en 2:a *inte* tolkas som att en operator befinner sig på samma plats i värdelistan. Istället tolkas en 2:a som att ett ”X” finns på den platsen, och i värdelistan sätts då motsvarande X-värde in innan dessa automatiskt matas vidare till *Reverse polish*. Samma stackbegränsningar gäller.

Minsta och största X-värden avser axelvärden efter skalning; om till exempel en graf ska ritas för X-värden mellan $-3.5$ och 8.47, så anges dessa två värden. Därefter är det bara att ange motsvarande pixel. Rutinen antar att graffönstret är 320 pixlar brett, och given X-pixel förväntas därmed vara på intervallet 0 till 319.

När beräkningen är slutförd ligger resultatet i GR0 och en nolla i GR1. Ligger det någonting annat i GR1 så har beräkningen gått fel (felaktig input).

### 5.6.3 Division

| GRx | Input |
|---|---|
| `0` | Täljare |
| `1` | Nämnare |
| `2` | Returadress |
| `Others` | Används ej till input |

| GRx | Output |
|---|---|
| `2` | Resultat från division |

Tar in ett en nämnare och en täljare och utför divisionen. Detta görs genom att subtrahera nämnaren från täljaren och räkna antalet subtraktioner. Försöker förbättra precisionen genom att skifta täljaren så långt till vänster som möjligt innan beräkningen, men kan ändå ge varierande bra resultat. Kommer ta mycket lång tid att utföra för vissa tal (små nämnare), upp till flera millisekunder. Efter beräkningen laddas resultatet till GR2 och programmet hoppar tillbaka till returadressen från inputen. En division med noll kommer ge noll som resultat.

## 5.7 Grafer

### 5.7.1 DrawAxis

| GRx | Input |
|---|---|
| `0` | X-pixel |
| `1` | Y-pixel |
| `2` | Returadress |
| `Others` | Används ej till input |

| GRx | Output |
|---|---|
| | Ingen output |

Ritar grafaxlar med origo angivet som pixelkoordinater.

### 5.7.2 Utritning: plot

| GRx | Input |
|---|---|
| 0 | Minsta X-värde på intervall (fixed-point) |
| 1 | Största X-värde på intervall (fixed-point) |
| 2 | Minsta Y-värde på intervall (fixed-point) |
| 3 | Största Y-värde på intervall (fixed-point) |
| 4 | Pekare till lista med värden |
| 5 | Pekare till lista med typer |
| 6 | Storlek på listor |
| 7 | Returadress |

| GRx | Output |
|---|---|
| | Ingen output |

Argument följer samma logik som för *Eval_fn*.

Plot börjar med att beräkna skärmkoordinater för origo, och drawAxis tar sedan hand om den utritningen.

Därefter, för varje pixel i X-led så passeras argument till Eval_fn, och motsvarande y-värde ritas ut. Detta skalas och översätts till ett pixelvärde baserat på given indata.

## 5.8 IO

### 5.8.1 print_num

| GRx | Input |
|---|---|
| 0 | Nummer att skriva ut |
| 1 | Adress i bildminnet att skriva på |
| 2 | Returadress |
| Others | Används ej till input |

| GRx | Output |
|---|---|
| | Ingen output |

Tar in ett fixed point-tal och skriver detta till specificerad adress med eventuellt minustecken framför. Skriver på formen ”0123.4567” i hexadecimala tal. Tar alltså upp 9 tiles för positiva tal och 10 för negativa. Efter utskrivningen hoppar programmet tillbaka till returadressen från inputen.

### 5.8.2 reset_bitmap

Skriver över alla pixlar i bitmapminnet med vitt. Är ingen subrutin, utan inkluderas där den ska användas.

#### 5.8.3 TODO

# 6 Slutsatser

# 7 Referenser

# Bilaga A Exempel på inmatning

Antag att användaren matar in sifferföljden "*55*". Alltså förväntar sig användaren att talet x53 ska sparas i minne. Antag också att varje ny karaktär sparas till GR0, och det slutgiltiga talet ska sparas till GR1.

Vi observerar nu dessa register för att se hur x53 sparas.

Inledningsvis:

```
GR0: 0000 0000 0000 0000.0000 0000 0000 0000
GR1: 0000 0000 0000 0000.0000 0000 0000 0000
```

Inmatning: GR0 := 5

```
GR0: 0000 0000 0000 0000.0000 0000 0000 0101
GR1: 0000 0000 0000 0000.0000 0000 0000 0000
```

GR0 skiftas vänster 16 steg

```
GR0: 0000 0000 0000 0101.0000 0000 0000 0000
GR1: 0000 0000 0000 0000.0000 0000 0000 0000
```

GR1 skiftas vänster 4 steg

```
GR0: 0000 0000 0000 0101.0000 0000 0000 0000
GR1: 0000 0000 0000 0000.0000 0000 0000 0000
```

GR1 := GR1 + GR0

```
GR0: 0000 0000 0000 0101.0000 0000 0000 0000
GR1: 0000 0000 0000 0101.0000 0000 0000 0000
```

Inmatning: GR0 := 3

```
GR0: 0000 0000 0000 0000.0000 0000 0000 0011
GR1: 0000 0000 0000 0101.0000 0000 0000 0000
```

GR0 skiftas vänster 16 steg

```
GR0: 0000 0000 0000 0011.0000 0000 0000 0000
GR1: 0000 0000 0000 0101.0000 0000 0000 0000
```

GR1 skiftas vänster 4 steg

```
GR0: 0000 0000 0000 0011.0000 0000 0000 0000
GR1: 0000 0000 0101 0000.0000 0000 0000 0000
```

GR1 := GR1 + GR0

```
GR0: 0000 0000 0000 0011.0000 0000 0000 0000
GR1: 0000 0000 0101 0011.0000 0000 0000 0000
```

GR1 innehåller nu bitrepresentatinen av x53 för ett $4 \times 4$ fixpunktstal.