Osservatorio Astrofisico di Arcetri
C.N.R. – C.A.I.S.M.I

Progetto Fasti

Un Assembler per lo SVB1

C. Baffa, V. Biliotti

Rapporto Interno di Arcetri N° **1/2000**
Firenze, Aprile 2000

Sommario.

Nel presente rapporto interno vengono descritti sia il linguaggio che il compilatore (lo **svb1asm***) dello assembler per il sistema di generazione di sequenze per* Fasti*, lo* SVB1*.*

Lo SVB1 *è un disegno concettuale che è stato realizzato tramite la tecnologia dei componenti programmabili. Nella corrente versione è implementato tramite due chip Xilix XC95288.*

Chapter 1

Introduzione

Il gruppo infrarosso di Arcetri sta lavorando ad una elettronica *leggera* per l'acquisizione dati con rivelatori bidimensionali infrarossi, Fasti. L'idea base di Fasti è che deve essere basata più su standard industriali e *disegni concettuali* che su devices specifici, per la facilità di sviluppo e per evitare una prematura obsolescenza.

In quest'ottica, uno dei componenti cruciali che abbiamo dovuto sviluppare è il generatore di sequenze, lo SVB1[1]. Lo SVB1 è un *disegno concettuale* che è stato realizzato tramite la tecnologia dei componenti programmabili. In questa prima versione è implementato tramite due chip Xilix XC95288.

Lo SVB1 si comporta come una periferica, ed è completamente controllato da un canale seriale, ora realizzato tramite una RS232. All'interno vi è della logica schematizzabile come un microprocessore specializzato nella produzione di sequenze. Nel seguito daremo il dettaglio del linguaggio di programmazione utilizzato. Il numero di linee che ogni unità di SVB1 può pilotare è 8.

Tramite il compilatore **svb1asm** si possono tradurre dei *programmi* di generazione di forme d'onda in un formato direttamente comprensibile dallo SVB1. Il programma **svb1asm** fornisce gli usuali strumenti di un'assembler, come la definizione di simboli e label, la possibilità di più aree disgiunte di dati e di programmi, il controllo della sintassi, la generazione di un listato con la diagnostica ed il compilato. Può inoltre essere utilizzato come sottoprogramma di un programma di acquisizione completo, per riprogrammare lo SVB1 al bisogno.

Per la comodità di sviluppo dei programmi è stato sviluppato un particolare ambito (syntax-file) per lo editor `vim`. Questo modo d'uso permette il riconoscimento della sintassi in fase di editing, facilitando cisì grandemente lo sviluppo del codice. Tale syntax-file viene riportato in appendice.

In appendice riportiamo il listato ottenuto compilando il programma che genera le forme d'onda per il controllo dei chip Nicmos3, ad esempio per gli

strumenti Arnica e LongSP.

Chapter 2

La struttura dello SVB1

Lo SVB1 è composto, dal pundo di vista funzionale, da tre parti:

- *Seqser*, un nucleo di controllo delle operazioni. Questa porzione, controllata direttamente da un'interfaccia RS232, permette di far partire o fermare la generazione di sequenze, di programmare e rileggere le memorie dati e programmi, e di leggere i registri di loop e di program counter della porzione di generazione di sequenze.

- *Seqproc*, una porzione di generazione di sequenze. Questa parte può essere pensata come un *microprocessorespecializzato nella generazione di sequenze*. Tale approccio garantisce il massimo di flessibilità e di riusabilità nella produzione di forme d'onda. Questo generatore è costituito dalla logica di esecuzione (*CPU*), da un'area di memoria per le maschere generatrici delle forme d'onda (*memoria dati*), da una memoria di *programma* e da quattro registri specializzati nel controllo dei loop, i registri A, B, C e D.

- una porzione per lo output ininterrotto delle maschere, che genera cos le forme d'onda richieste. Questa porzione è realizzata tramite una memoria FIFO (First Input First Output) *intelligente*, che comincia le proprie operazioni quando è piena a metà, e quando è completamente piena, ferma le operazioni della porzione di generazione delle sequenze. Quest'ultima situazione è quella più comune, mentre in situazioni realistiche, non accade mai che la memoria FIFO si vuoti. Il clock di uscita è fissato ora alla stessa frequenza di esecuzione dello *Seqproc*, ma tale scelta non è vincolante, e si possono usare delle demoltipliche o una frequenza esterna.

Lo SVB1 è realizzato tramite due chip Xilix XC95288 e due memorie, ma vogliamo sottolineare che, come la maggior parte di Fasti, esso è in realtà un *disegno concettuale*, e con l'evolversi della tecnologia potrà essere implementato con componenti più moderni, probabilmente in un unico chip, e potrà arricchirsi di funzionalità che ora mancano.

metacomando	valore	descrizione
`wrp`	0x4000	Scrive un byte nella memoria di programma
`wrd`	0x1000	Scrive un byte nella memoria dati
`rdp`	0x6000	Legge un byte dalla memoria di programma
`rdd`	0x8000	Legge un byte dalla memoria dati
`rdra`	0xa000	Legge il registro A
`rdrb`	0xa001	Legge il registro B
`rdrc`	0xa002	Legge il registro C
`rdrd`	0xa003	Legge il registro D
`rdpl`	0xa004	Legge lo LSB del Program Counter
`rdph`	0xa005	Legge lo MSB del Program Counter
`reset`	0x300080	Software reset dello SVB1
`stop`	0x300040	Ferma la generazione di sequenze
`step`	0x300020	Esegue un solo passo di programma
`go`	0x30001X	Inizia il programma alla pagina X (0-3)
`rdsl`	0xb008	Legge lo LSB del registro di autotest
`rdsh`	0xb009	Legge lo MSB del registro di autotest

Elenco dei metacomandi dello *Seqser*

Chapter 3

La struttura dettagliata di *Seqproc*

Per chiarificare meglio il modello mentale che vede nella porzione centrale di SVB1, lo *Seqproc*, un *microprocessore specializzato nella generazione di sequenze*, occorre analizzare un poco più in dettaglio la struttura di questa parte.

Vi sono due aree di memoria separate, una per le maschere di output e l'altra pione centrale di SVB1, lo *Seqproc*, un *microprocessore specializzato nella generazione di sequenze*, occorre analizzare un poco più in dettaglio la struttura di questa parte.

Vi sono due aree di memoria separate, una per le maschere di output e l'altra per le istruzioni da eseguire (*il programma*). Entrambe sono memorie a 16 bit e sono lunghe rispettivamente 256 e 4×4096 bytes. La memoria programmi è organizzata in 4 pagine da 4096 bytes, non si possono eseguire salti tra una pagina e l'altra. Il *microprocessore* ha istruzioni e dati a 16 bit, mentre l'indirizzamento è a byte. Il *microprocessore* ha la capacità di indirizzare solo word allineate, mentre dall'esterno, tramite l'accesso seriale a *Seqser*, si pu accedere alla memoria a livello di byte.

Le maschere di output sono composte da un byte (bit 15-8) che contiene il dato da presentare sulle 8 line di uscita dei clock da generare, e da un byte (bit 7-0) che agisce come contatore di ripetizione, cioè indica quante volte, in termini di cicli elementari da 0.1 μsec, il byte alto va ripetuto, ottenendo in questo modo un ulteriore livello di loop, (loop implicito).

La memoria di programma è composta da 4 pagine, lunghe 4096 bytes. In lettura/scrittura dall'esterno tutto lo spazio di indirizzamento è accessibile, mentre dalle istruzioni di salto e di loop è accessibile solo la pagina delle quattro che viene selezionata dalla meta-istruzione `Go` **n** dello *Seqser*, che fa partire la generazione di sequenze dalla pagina **n**, e permette cos di scegliere, senza riprogrammazioni, tra quattro sequenze diverse ed indipendenti.

I quattro registri del *microprocessore* sono registri specializzati per i

loop. Essi agiscono come contatori in decremento e permettono di effettuare il salto solo se il contenuto del registro **non** è zero. Il decremento viene effettuato durante il salto. I quattro registri (A, B, C e D), sono del tutto equivalenti, ma una buona pratica di programmazione assegna al registro A il ruolo di contatore dei loop più interni, e via via ai registri B, C e D, i loop progressivamente più esterni.

Per una più agevole modifica del tempo di integrazione, si può anche posizionare la routine di ritardo in una locazione fissa, ad esempio alla fine di una pagina, semplificando le operazioni e rendendo meno prono agli erori questa fase.

Il tempo di integrazione viene ottenuto con un loop che mantiene una maschera costante per il tempo stabilito. Con un clock elemetare di 0.1 μsec, il tempo massimo di integrazione è di circa 30 ore per una singola integrazione e poco più di 7 minuti di intervallo per le integrazioni a campionamento multiplo, con un massimo di 255 campioni.

Chapter 4

Gli elementi dell'assembler

Per poter utilizzare in maniera comoda e flessibile lo SVB1, abbiamo disegnato un semplice linguaggio assembler e scritto un compilatore che verificasse la sintassi e la correttezza dei parametri e lo traducesse nei metacomandi necessari per la programmazione.

Il linguaggio risultante ha quattro direttive ed 11 istruzioni, è quindi relativamente diretto ed agevole da implementare. Possono essere definite delle label, che indicano la locazione di dati (istruzione `data`) o posizioni di programma. In quest'ultimo caso, le label devono trovarsi all'inizio di righe vuote o contenenti solo commenti.

4.1 Le direttive

L'assembler dello SVB1 comprende quattro direttive che servono per modificare il comportamento delprogramma assemblatore.

direttiva	argomento	descrizione
`equ`	16 bit	Definisce il valore di un simbolo
`org`	12 bit	Stabilisce la locazione della prossima istruzione
`orgd`	8 bit	Stabilisce la locazione del prossimo dato
`data`	16 bit	Definisce un dato e ne alloca l'area di memoria

Elenco delle direttive dello assembler

La prima è l'istruzione `equ` che definisce il valore di simbolo che può essere usato in tutte le occasioni dove si può usare un numero. All'inizio il programma ha già definiti solo i simboli *zero* ed *uno*, di significato ovvio. Il valore può essere definito come un numero, una label o in generale un'espressione che possa essere valutata come un numero. Nella tabella è riportata la lungezza, in bit dell'argomento. La sintassi è la seguente:

[simbolo] `equ` [valore]

Vi sono due direttive che permettono di specificare in quale zona della memoria saranno posizionati gli elementi che seguono, sia per la memoria dati che per la la memoria istruzioni. Le due istruizioni sono `orgd`, per la memoria dati e `org` per la memoria di programma. Le due istruzioni accettano un valore numerico, una label o in generale un'espressione che possa essere valutata come un numero. La sintassi è la seguente:

`org` [valore]

per la memoria di programma e per la memoria dati:

`orgd` [valore]

L'ultima direttiva`data`, permette di definire una maschera per la generazione delle forme d'onda. Questa direttiva prevede l'uso di una label che permette di indirizzare il dato dichiarato. I dati (o maschere di output) sono a 16 bit, e sono composte da un byte (alto, bit 15-8) che contiene il dato da presentare sulle 8 line di uscita dei clock da generare, e da un byte (basso, bit 7-0) che agisce come contatore di ripetizione, cioè indica quante volte, in termini di cicli elementari da 0.1 μsec, il byte alto va ripetuto. Particolarmente utile per la definizione dei dati è l'operatore '**@**' ($A@B = 256\dot{A} + B$), che permette di definire separatamente i due byte, di output e di ripetizione. La sintassi è la seguente:

[simbolo] `data` [valore]

oppure, più comodamente:

[simbolo] `data` [output] @ [ripetizione]

4.2 Le istruzioni

Vi sono 11 istruzioni diverse, raggruppate in cinque classi differenti. Questo set di istruzioni é ridotto, ma sufficientemente flessibile per una descrizione comoda delle forme d'onda. A titolo di esempio, in appendice riportiamo le forme d'onda complete per gli strumenti Arnica e LongSP, dotati di rivelatore Nimos3. Tutte le istruzioni sono dotate di argomento. Nella tabella è riportata la lungezza, in bit dell'argomento.

istruzione	valore	argomento	descrizione
`outm`	0xf000	8 bit	Output di una maschera
`loada`	0x4000	8 bit	Carica un valore nel registro A
`loadb`	0x5000	8 bit	Carica un valore nel registro B
`loadc`	0x6000	8 bit	Carica un valore nel registro C
`loadd`	0x7000	8 bit	Carica un valore nel registro D
`loopa`	0x8000	12 bit	Loop con decremento del registro A
`loopb`	0x9000	12 bit	Loop con decremento del registro B
`loopc`	0xa000	12 bit	Loop con decremento del registro C
`loopd`	0xb000	12 bit	Loop con decremento del registro D
`jump`	0x3000	12 bit	Salto incondizionato
`rest`	0x1000	12 bit	Restart condizionato delle operazioni

Elenco delle istruzioni dello assembler

La prima istruzione, `outm` [valore], è l'istruzione fondamentale che permette di generare le forme d'onda richieste. Il valore che viene inviato alla memoria FIFO di uscita è formato da due byte. Il byte più significativo contiene la maschera da porre in uscita, mentre il byte più basso contiene il tempo, espresso in passi di clock, per il quale la maschera in questione deve essere presente in uscita. Il formato è:

`outm` [valore]

Le successive quattro istruzioni formano il gruppo delle istruzioni di caricamento dei registri. Vi sono infatti quattro registri (A, B, C e D), che sono specializzati per l'esecuzione di loop. Le istruzioni `loadX` permettono di caricare questi registri con valori appropriati. I registi sono a 8 bite e possono quindi essere caricati con valori compresi tra 0 e 255. Il valore da caricare può essere specificato solo in modo esplicito, tramite un valore numerico o un'espressione che dia un risultato numerico, e non con un altro registro o una locazione di memoria. Il formato è:

`loada` [valore]
`loadb` [valore]
`loadc` [valore]
`loadd` [valore]

Le quattro istruzioni per l'esecuzione di loop formano il terzo gruppo di istruzioni. Esse permettono di eseguire dei loop, anche annidati fino a quattro livelli. Queste istruzioni verificano se il contenuto del registro corrispondente è diverso da zero, ed in tal caso, eseguono un salto alla locazione specificata da [valore], solitamente una label, decrementando al tempo stesso il contenuto del registro. Anche questa istruzione ammette solo argomenti espliciti. Il formato è:

`loopa` [valore]
`loopb` [valore]

`loopc` [valore]
`loopd` [valore]

Le due istruzioni successive (`jump` e `rest`), eseguono entrambe dei salti incondizionati, ma con un'importante differenza: l'istruzione `rest` dopo aver eseguito il salto, ferma la generazione di sequenze se é stato ricevuto un metacomado `go`, permettendo così di avere degli arresti sincroni con la fine delle integrazioni. Entrambi i comandi accettano solo argomenti espliciti, solitamente delle label, e possono saltare ad un punto qualunque della pagina corrente dello spazio di memoria di programma, selezionata dall'istruzione `Go` **n**. Il formato è:

`jump` [valore]
`rest` [valore]

4.3 Le espressioni

Tutte le istruzioni dello **svb1asm** richiedono un argomento numerico. Tale argomento deve essere *immediato* nel senso che può essere formato da una espressione complessai, che includa anche label o indirizzi, ma non può consistere in un riferimento ad una locazione di memoria.

Tutte le espressioni vengono valutate come interi ed il risultato può avere 8, 12 o 16 bit significativi. I calcoli sono eseguiti utilizzando gli interi di default, quindi solitamente a 32 bit, che vengono poi mascherati secondo la necessità.

L'ordine di precedenza nella valutazione delle espressioni è quello usuale. Si possono utilizzare le parentesi e i segni di meno aritmetico (`-`) e NOT logico (`!`). Esiste anche l'operatore non standard `@`, disegnato per la definizione delle maschere, e che permette di *montare* due bytes in un unica word a 16 bit ($A@B = 256\dot{A} + B$). In tabella elenchiamo le operazioni permesse con l'ordine di precedenza.

operatore	priorità	descrizione
`( e )`	1	parentesi
`!`	2	negazione bit a bit (inversione)
`-`	2	negazione aritmetica
`/`	3	divisione intera
`*`	3	moltiplicazione intera
`-`	4	sottrazione
`+`	4	addizione
`&`	5	AND bit a bit (AND aritmetico)
`|`	6	OR bit a bit (AND aritmetico)
`^`	6	XOR bit a bit (OR esclusivo aritmetico)
`@`	7	$A@B = 256\dot{A} + B$

Appendix A

Esempio di un programma per lo svb1asm

Per avere un quadro più chiaro del funzionamento di SVB1 e della sintassi dello **svb1asm**, diamo di seguito il listato del programma che genera le forme d'onda per gli array Nicmos3 degli strumenti di Arcetri (Arnica e LongSP).

```
0000 0000 0000 ; arnica_base.asm 30-3-2000
0001 0000 0000 ; Arnica Waveforms for SVB1 serializer.
0002 0000 0000 ;
0003 0000 0000 ; taken directly from arnica's nprogra.c
0004 0000 0000 ;
0005 0000 0000
0006 0000 0000 ; some symbols
0007 0020 0000 BASE            equ     #20             ; standard no operation pattern
0008 0001 0000 MXCLK           equ     #01             ; hex mask for xclock
0009 0021 0000 XCLK            equ     MXCLK | BASE    ; xclock
0010 0022 0000 XSYNC           equ     #02 | BASE      ; xsync
0011 0004 0000 MYCLK           equ     #04             ; hex mask for yclock
0012 0024 0000 YCLK            equ     MYCLK | BASE    ; yclock
0013 0028 0000 YSYNC           equ     #08 | BASE      ; ysync
0014 0010 0000 MFRESET         equ     #10             ; hex mask for detector reset
0015 0030 0000 FRESET          equ     MFRESET | BASE  ; hex mask for detector reset
0016 0020 0000 ADCST           equ     #20             ; hex mask for ADC start
0017 0000 0000 MADCST          equ     BASE & ! ADCST  ; hex mask for ADC start
0018 0060 0000 EOF             equ     #40 | BASE      ; End of frame
0019 000a 0000 mul             equ     10              ; steps in a millisecond
0020 0000 0000
0021 0000 0000 ; here we define the global output masks and symbols
0022 0000 0000                 orgd    zero            ; start of output masks
0023 0000 20ff dummy255        data    BASE @ 255      ; 255 * noop
0024 0002 20c8 smallnoharm     data    BASE @ (20*mul) ; the old small_no_harm
0025 0004 2064 BASE10          data    BASE @ (10*mul) ; base delay for TINT (10usecs)
0026 03e8 0000 TINT            equ     1000            ; base time in milliseconds
0027 0002 0000 TINTC           equ     2               ; TINT / .5 sec
0028 00fa 0000 TINTB           equ     250             ; 500  msecs
0029 00c8 0000 TINTA           equ     200             ; 2000 usecs
0030 0001 0000 nreset          equ     1               ; how many reset to perform
0031 0000 0000
0032 0000 0000 ; here we define the form3 and form4 masks
0033 0006 2128 rdxclk          data    XCLK                  @ 4 * mul
```

```
0008 2f0a rdxcxsycys      data    XCLK|XSYNC|YCLK|YSYNC @ 1 * mul
000a 2e0a rdxsycys        data         XSYNC|YCLK|YSYNC @ 1 * mul
000c 29c8 rdxcys20        data    XCLK|            YSYNC @ 20* mul
000e 270a rdxcxsyc        data    XCLK|XSYNC|YCLK       @ 1 * mul
0010 260a rdxsyc          data         XSYNC|YCLK       @ 1 * mul
0012 21c8 rdxc20          data    XCLK                  @ 20* mul
0014 6014 rdeof           data    EOF                   @ 2 * mul
0000 0000 ; here we define the form1 and form2 masks
0016 3128 rsxclk          data    FRESET|XCLK                  @ 4 * mul
0018 3f0a rsxcxsycys      data    FRESET|XCLK|XSYNC|YCLK|YSYNC @ 1 * mul
001a 3e0a rsxsycys        data    FRESET|     XSYNC|YCLK|YSYNC @ 1 * mul
001c 39c8 rsxcys20        data    FRESET|XCLK|           YSYNC @ 20* mul
001e 370a rsxcxsyc        data    FRESET|XCLK|XSYNC|YCLK       @ 1 * mul
0020 360a rsxsyc          data    FRESET|     XSYNC|YCLK       @ 1 * mul
0022 31c8 rsxc20          data    FRESET|XCLK                  @ 20* mul
0024 3014 rsnoeof         data    FRESET                       @ 2 * mul
0000 0000
0000 0000 ; we define the init masks
0026 200a initbase        data    BASE         @ 1 * mul
0028 250a initxcyc        data    XCLK|YCLK    @ 1 * mul
0000 0000
0000 0000 ; here we define the rst64 masks
002a 3114 rresxclk        data    FRESET | XCLK @ 2 * mul
002c 3014 rres            data    FRESET       @ 2 * mul
002e 30c8 rres20          data    FRESET       @ 20 * mul
0000 0000
0000 0000 ; here we define the read64 masks
0030 210a rxclk           data    BASE | XCLK   @ 1 * mul
0032 200a rbase           data    BASE          @ 1 * mul
0034 210a rxclkadcst      data    MADCST | XCLK @ 1 * mul
0036 000a rbaseadcst      data    MADCST        @ 1 * mul
0038 20c8 rbase20         data    BASE          @ 20 * mul
0000 0000 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
0000 0000                 org     zero            ; start of code
0000 f000                 outm    dummy255        ; just to fill out fifo
0002 0000 ; loop to empty detector shift register
0002 0000 empty
0002 4080                 loada   128             ; to have 128 clock out
0004 0000 emptya
0004 f026                 outm    initbase
0006 f028                 outm    initxcyc        ; we clock out x and y sync
0008 8004                 loopa   emptya
000a 0000 ; we jump to real start
000a 300c                 jump    startframe
000c 0000 ; Here we define the read and reset inner loops, just in case
000c 0000 ; sometimes subroutines will be implemented.
000c 0000 ;; the old rst64 64 reset cycles (128 pixels)
000c 0000 ;rst64
000c 0000 ;                loada   64              ; To have 64 row clock
000c 0000 ;rst64a
000c 0000 ;                outm    rresxclk
000c 0000 ;                outm    rres            ; loop with reset enabled
000c 0000 ;                loopa   rst64a
000c 0000 ;                                        ; tail
000c 0000 ;                outm    rres20          ; only a delay
000c 0000 ;; end of form
000c 0000 ;
000c 0000 ;; the old read64, 64 read cycles (128 pixels)
000c 0000 ;read64
000c 0000 ;                loada   64              ; to have 64 row clock
000c 0000 ;                outm    rxclk           ; to increment counter
000c 0000 ;                outm    rxclkadcst      ; start conversion
```

```
0096 000c 0000 ;                outm    rbase           ; to increment counter
0097 000c 0000 ;                outm    rbaseadcst      ; start conversion
0098 000c 0000 ;                loopa   read64
0099 000c 0000 ;                                        ; tail
0100 000c 0000 ;                outm    rbase20         ; only a delay
0101 000c 0000 ;; end of form
0102 000c 0000
0103 000c 0000 ; Here start the real frame clock
0104 000c 0000 startframe
0105 000c 0000 ; We implement variable number of reset
0106 000c 5001                  loadb   nreset          ; how many reset?
0107 000e 0000 ; We start the first line of reset (the old form1)
0108 000e 0000 startreset
0109 000e 0000 form1
0110 000e f016                  outm    rsxclk          ; small delay
0111 0010 f018                  outm    rsxcxsycys      ; Y and X line init
0112 0012 f01a                  outm    rsxsycys
0113 0014 f01c                  outm    rsxcys20        ; X and Y clock and stabilize
0114 0016 0000
0115 0016 0000                  ;call    rst64
0116 0016 0000 ; the old rst64 64 reset cycles (128 pixels)
0117 0016 0000 ;rst64
0118 0016 4040                  loada   64              ; To have 64 row clock
0119 0018 0000 rst64a2
0120 0018 f02a                  outm    rresxclk
0121 001a f02c                  outm    rres            ; loop with reset enabled
0122 001c 8018                  loopa   rst64a2
0123 001e 0000                                          ; tail
0124 001e f02e                  outm    rres20          ; only a delay
0125 0020 0000
0126 0020 0000 ; end of form
0127 0020 0000
0128 0020 0000 ; Bulk (127) row in read mode
0129 0020 0000 form2
0130 0020 f016                  outm    rsxclk          ; small delay
0131 0022 f01e                  outm    rsxcxsyc        ; X line init
0132 0024 f020                  outm    rsxsyc
0133 0026 f022                  outm    rsxc20          ; X and Y clock and stabilize
0134 0028 0000
0135 0028 0000                  ;call    rst64
0136 0028 0000 ; the old rst64 64 reset cycles (128 pixels)
0137 0028 0000 ;rst64
0138 0028 4040                  loada   64              ; To have 64 row clock
0139 002a 0000 rst64a1
0140 002a f02a                  outm    rresxclk
0141 002c f02c                  outm    rres            ; loop with reset enabled
0142 002e 802a                  loopa   rst64a1
0143 0030 0000                                          ; tail
0144 0030 f02e                  outm    rres20          ; only a delay
0145 0032 0000
0146 0032 0000 ; end of form
0147 0032 0000
0148 0032 0000 ;       loop on number of reset
0149 0032 900e                  loopb   startreset      ; to perform more than one reset
0150 0034 0000
0151 0034 0000 ; We are at last line: We do not output EOF
0152 0034 0000 form5
0153 0034 f024                  outm    rsnoeof         ; no EOF output
0154 0036 0000
0155 0036 0000 ; We start the first line of read. (the old form3)
0156 0036 0000 startread
0157 0036 0000 form3
```

```
0158 0036 f006                  outm    rdxclk          ; small delay
0159 0038 f008                  outm    rdxcxsycys      ; Y and X line init
0160 003a f00a                  outm    rdxsycys
0161 003c f00c                  outm    rdxcys20        ; X and Y clock and stabilize
0162 003e 0000
0163 003e 0000                  ;call    read64
0164 003e 0000 ; the old read64, 64 read cycles (128 pixels)
0165 003e 0000 ;read64
0166 003e 4040                  loada   64              ; to have 64 row clock
0167 0040 0000 read64a1
0168 0040 f030                  outm    rxclk           ; to increment counter
0169 0042 f034                  outm    rxclkadcst      ; start conversion
0170 0044 f032                  outm    rbase           ; to increment counter
0171 0046 f036                  outm    rbaseadcst      ; start conversion
0172 0048 8040                  loopa   read64a1
0173 004a 0000                                          ; tail
0174 004a f038                  outm    rbase20         ; only a delay
0175 004c 0000
0176 004c 0000 ; end of form
0177 004c 0000
0178 004c 0000 ; Bulk (127) row in read mode
0179 004c 0000 form4
0180 004c f006                  outm    rdxclk          ; small delay
0181 004e f00e                  outm    rdxcxsyc        ; X line init
0182 0050 f010                  outm    rdxsyc
0183 0052 f012                  outm    rdxc20          ; X and Y clock and stabilize
0184 0054 0000
0185 0054 0000                  ;call    read64
0186 0054 0000 ; the old read64, 64 read cycles (128 pixels)
0187 0054 0000 ;read64
0188 0054 4040                  loada   64              ; to have 64 row clock
0189 0056 0000 read64a2
0190 0056 f030                  outm    rxclk           ; to increment counter
0191 0058 f034                  outm    rxclkadcst      ; start conversion
0192 005a f032                  outm    rbase           ; to increment counter
0193 005c f036                  outm    rbaseadcst      ; start conversion
0194 005e 8056                  loopa   read64a2
0195 0060 0000                                          ; tail
0196 0060 f038                  outm    rbase20         ; only a delay
0197 0062 0000 ; end of form
0198 0062 0000
0199 0062 0000 ; We are at last line: We output EOF
0200 0062 0000 form7
0201 0062 f014                  outm    rdeof           ; EOF output
0202 0064 0000
0203 0064 0000 ; Here we start TINT
0204 0064 0000
0205 0064 0000 tint
0206 0064 6002                  loadc   TINTC           ; TINT / .5 seconds
0207 0066 0000 tintc
0208 0066 50fa                  loadb   TINTB           ; each unit is worth 2 msec
0209 0068 0000 tintb
0210 0068 40c8                  loada   TINTA           ; each unit is worth 10 usecs
0211 006a 0000 tinta
0212 006a f004                  outm    BASE10          ; base delay - 10 usec
0213 006c 806a                  loopa   tinta           ; inner  loop
0214 006e 9068                  loopb   tintb           ; medium loop
0215 0070 a066                  loopc   tintc           ; outer  loop
0216 0072 0000
0217 0072 0000 ; here we do the second read
0218 0072 0000
0219 0072 0000
```

```
0220 0072 0000 ; We start the first line of read. (the old form3)
0221 0072 0000 startreadb
0222 0072 0000 form3b
0223 0072 f006                 outm    rdxclk          ; small delay
0224 0074 f008                 outm    rdxcxsycys      ; Y and X line init
0225 0076 f00a                 outm    rdxsycys
0226 0078 f00c                 outm    rdxcys20        ; X and Y clock and stabilize
0227 007a 0000
0228 007a 0000                 ;call    read64
0229 007a 0000 ; the old read64, 64 read cycles (128 pixels)
0230 007a 0000 ;read64
0231 007a 4040                 loada   64              ; to have 64 row clock
0232 007c 0000 read64a3
0233 007c f030                 outm    rxclk           ; to increment counter
0234 007e f034                 outm    rxclkadcst      ; start conversion
0235 0080 f032                 outm    rbase           ; to increment counter
0236 0082 f036                 outm    rbaseadcst      ; start conversion
0237 0084 807c                 loopa   read64a3
0238 0086 0000                                         ; tail
0239 0086 f038                 outm    rbase20         ; only a delay
0240 0088 0000
0241 0088 0000 ; end of form
0242 0088 0000
0243 0088 0000 ; Bulk (127) row in read mode
0244 0088 0000 form4b
0245 0088 f006                 outm    rdxclk          ; small delay
0246 008a f00e                 outm    rdxcxsyc        ; X line init
0247 008c f010                 outm    rdxsyc
0248 008e f012                 outm    rdxc20          ; X and Y clock and stabilize
0249 0090 0000
0250 0090 0000                 ;call    read64
0251 0090 0000 ; the old read64, 64 read cycles (128 pixels)
0252 0090 0000 ;read64
0253 0090 4040                 loada   64              ; to have 64 row clock
0254 0092 0000 read64a4
0255 0092 f030                 outm    rxclk           ; to increment counter
0256 0094 f034                 outm    rxclkadcst      ; start conversion
0257 0096 f032                 outm    rbase           ; to increment counter
0258 0098 f036                 outm    rbaseadcst      ; start conversion
0259 009a 8092                 loopa   read64a4
0260 009c 0000                                         ; tail
0261 009c f038                 outm    rbase20         ; only a delay
0262 009e 0000 ; end of form
0263 009e 0000
0264 009e 0000 ; We are at last line: We output EOF
0265 009e 0000 form7b
0266 009e f014                 outm    rdeof           ; EOF output
0267 00a0 0000
0268 00a0 0000
0269 00a0 0000 ; Final stage: we go back
0270 00a0 100c                 rest    startframe
0271 00a2 0000
```

A.1 Vim syntax-file

Riportiamo il file che permette il riconoscimento della sintassi dello **svb1asm** durante una fase di editing, facilitando così lo sviluppo.

```
" Vim syntax file
```

```
" Language: SVB Assembler
" Maintainer: Kevin Dahlhausen <ap096@po.cwru.edu>
" Last change: 1999 Jun 14

" Remove any old syntax stuff hanging around
syn clear

syn case ignore


" storage types
syn match asmType "\.long"
syn match asmType "\.ascii"
syn match asmType "\.asciz"
syn match asmType "\.byte"
syn match asmType "\.double"
syn match asmType "\.float"
syn match asmType "\.hword"
syn match asmType "\.int"
syn match asmType "\.octa"
syn match asmType "\.quad"
syn match asmType "\.short"
syn match asmType "\.single"
syn match asmType "\.space"
syn match asmType "\.string"
syn match asmType "\.word"

syn match asmLabel "[a-z_][a-z0-9_]*:"he=e-1
syn match asmIdentifier "[a-z_][a-z0-9_]*"

" Various #'s as defined by GAS ref manual sec 3.6.2.1
" Technically, the first decNumber def is actually octal,
" since the value of 0-7 octal is the same as 0-7 decimal,
" I prefer to map it as decimal:
syn match decNumber "0\+[1-7]\=[\t\n$,; ]"
syn match decNumber "[1-9]\d*"
syn match octNumber "0[0-7][0-7]\+"
syn match hexNumber "0[xX][0-9a-fA-F]\+"
syn match hexNumber "#[0-9a-fA-F]\+"
syn match binNumber "0[bB][0-1]*"


syn match asmSpecialComment ";\*\*\*.*"
syn match asmComment ";.*"hs=s+1

syn match asmInclude "\.include"
syn match asmCond "\.if"
syn match asmCond "\.else"
syn match asmCond "\.endif"
syn match asmMacro "\.macro"
syn match asmMacro "\.endm"

syn match asmDirective          "loop[a-d]"
syn match asmDirective          "load[a-d]"
syn match asmDirective          "outm"
syn match asmDirective          "jump"
syn match asmDirective          "rest"

syn match asmDirective "\.[a-z][a-z]\+"
syn match asmMacro     "org"
syn match asmMacro     "orgd"
syn match asmMacro     "data"
```

```
syn match asmMacro     "equu"


syn case match

if !exists("did_asm_syntax_inits")
  let did_asm_syntax_inits = 1

  " The default methods for highlighting.  Can be overridden later
  hi link asmSection Special
  hi link asmLabel Label
  hi link asmComment Comment
  hi link asmDirective Statement

  hi link asmInclude Include
  hi link asmCond PreCondit
  hi link asmMacro Macro

  hi link hexNumber Number
  hi link decNumber Number
  hi link octNumber Number
  hi link binNumber Number

  " My default color overrides:
  hi asmSpecialComment ctermfg=red
  hi asmIdentifier ctermfg=lightcyan
  hi asmType ctermbg=black ctermfg=brown

endif

let b:current_syntax = "asm"

" vim: ts=8
```

Contents

Bibliography

[1] V.Biliotti, "Il generatore di sequenze SVB1 per Fasti", 2000, Rapporto Interno dell"Osservatorio di Arcetri, **N^ 2/2000**

[2] C.Baffa, "Fasti un controller veloce per l'Infrarosso", 1998, Memo del Gruppo Infrarosso di Arcetri.