INAF - Osservatorio Astrofisico di Arcetri
INAF - I.R.A.

Progetto Fasti

Manuale d'uso del Software - I
L'assembler e l'emulatore

C. Baffa, E. Giani

Rapporto Interno di Arcetri N° **1/2004**
Firenze, Maggio 2004

Sommario.

Per lo sviluppo e l'uso dell'elettronica di controllo per rivelatori infrarossi bidimensionali Fasti*sono stati sviluppati diversi strumenti software. Questo rapporto ne descrive l'uso e fornisce le ulteriori informazioni necessarie per il test e l'utilizzo di* Fasti.

Il presente rapporto interno è il primo di una serie in cui vengono descritti tutti gli strumenti software di Fasti. *Nel seguito descriveremo sia il linguaggio che il compilatore (***svb1asm***) dello assembler per il sistema di generazione di sequenze per* Fasti, *lo* SVB1, *che l'uso dell'emulatore software dello* SVB1, **EmuSvb**. *Nei successivi rapporti verranno descritti il programma di visualizzazione delle forme d'onda,* **GsvbPlot**, *il programma di visualizzazione di un flusso ininterrotto di dati* **FreeRun**, *il programma di controllo remoto* **GClient**, *il programma di controllo generale,* **Ftest**, *e lo standard di scrittura delle forme d'onda per nics.*

Capitolo 1

Introduzione

Il gruppo infrarosso di Arcetri ha realizzato una elettronica *leggera* per l'acquisizione dati con rivelatori bidimensionali infrarossi, Fasti. La caratteristica di Fasti è l'appoggiarsi più su standard industriali e *disegni concettuali* che su devices specifici, per la facilità di sviluppo e per evitare una prematura obsolescenza. La struttura dettagliata di Fasti si può trovare in diversi documenti[1, 3, 4].

Uno dei componenti cruciali che abbiamo sviluppato è il generatore di sequenze, lo SVB1[5]. Lo SVB1 è un *disegno concettuale* che è stato realizzato tramite la tecnologia dei componenti programmabili. Nella versione per NICS è implementato tramite due chip Xilix XC95288. All'interno vi è della logica schematizzabile come un microprocessore specializzato nella produzione di sequenze. Abbiamo definito uno *pseudoassembler* per la programmazione di tali sequenze, per agevolare il loro sviluppo. Tramite il compilatore **svb1asm** si possono tradurre questi *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.

Per facilitare ulteriormente lo sviluppo di forme d'onda adeguate, sono stati sviluppati anche due programmi ausiliari: un programma di emulazione software del generatore di sequenze, **EmuSvb**, ed un programma che permette di esplorare gaficamente i segnali prodotti, **GsvbPlot**.

I segnali generati dal rivelatore vengono convertiti da quattro ADC a 16 bit, posti su due schede. I dati risultanti vengono trasmessi tramite quattro bus seriali veloce ad una scheda buffer che si occupa del colloquo con il calcolatore di controllo e, quando richiesto, della sottrazione tra prima e seconda lettura. I dati vengono infine letti dal PC di controllo[6] tramite una scheda NI PCI-DIO32HS, per cui è stato sviluppato appositamente un device driver per Linux[7].

Il programma che permette il controllo di Fasti, sia in fase di test che in fase di utilizzo regolare è **Ftest**. Questo programma ha due modi principali

di funzionamento, in modo interattivo ed in modo *demone*. Nel primo caso permette un controllo assai dettagliato dell'elettronica, consentendo sia operazioni elementari di test che acquisizioni, display e acquisizioni continue. Nel secondo caso agisce come un programma demone di Unix, eseguendo i comandi che riceve tramite una socket di rete. Questo è anche il modo usuale di funzionamento al telescopio, rendendo Fasti un *device di rete*, con gli ovvi benfici in termini di versatilità. Il programma **GClient** permette il test di questo tipo di funzionamnento ed un limitato uso di Fasti in questa modalità.

Il presente rapporto descrive gli elementi base dell'assembler sviluppato per lo SVB1, l'uso dell'assembler e dell'emulatore. I restanti programmi verranno descritti in rapporti successivi.

Capitolo 2

L'Assembler dello SVB1

Lo SVB1 è composto, dal punto di vista funzionale, da tre parti: *Seqser*, un nucleo di controllo delle operazioni, *Seqproc*, una porzione di generazione di sequenze, ed una una porzione per lo output ininterrotto delle maschere. *Seqproc* può essere pensato come un microprocessore *specializzato 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.

Per una descrizione dettagliata del funzionamento di SVB1 si rimanda all'opportuna documentazione[5, 2] da cui riprendiamo alcune parti relative all'assembler, aggiungendo le descrizioni delle modifiche intervenute.

2.1 La struttura dettagliata di *Seqproc*

Per comprendere meglio il funzionamento dello assembler per lo SVB1, analizziamo un poco più in dettaglio la struttura della porzione di generazione delle sequenze, *Seqproc*, pensandolo come un microprocessore di tipo tradizionale.

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[1]. 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 seriale, si puó accedere alla memoria a livello di byte.

[1] nella versione attuale è attiva solo la prima pagina

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 0.125 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. (Nella versione attuale è attiva solo la prima pagina)

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 **prima** della sottrazione **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. Nel file listato prodotto dal programma **svb1asm**, l'estensione dei loop è evidenziata per comodià di programmazione.

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 0.125 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.

2.2 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 cinque direttive e 12 istruzioni, è quindi relativamente diretto ed agevole da implementare. Possono essere definite delle etichette simboliche (comando `equ`) aventi un valore numerico e 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.

I commenti sono preceduti dal carattere punto e virgola (`;`), tutto quanto segue questo carattere viene ignorato dal compilatore.

2.3 Le direttive

L'assembler dello SVB1 comprende cinque direttive che servono per modificare il comportamento del programma 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
`nmask`	16 bit	Definisce i segnali a logica negativa

Elenco delle direttive dello assembler

La prima è l'istruzione `equ` che definisce il valore di un simbolo che può essere usato in tutte le occasioni dove si può usare un numero. All'inizio, il programma ha già definiti i simboli *zero* ed *uno*, di significato ovvio, *TRUE*=1 e *FALSE*=0. 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, mentre per la memoria dati:

`orgd` [valore]

La 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]

Per comodità di programmazione vi è una direttiva, `nmask` che definisce quali segnali siano da considerarsi a logica negativa. Dal punto di vista operativo, questo viene realizzato tramite un operazione di `XOR` tra tutti i valori in uscita e la maschera definita tramite questa istruzione. L'uso di questa istruzione permette di negare un segnale tramite una sola operazione, senza un pesante editing. Un altro vantaggio consiste nella possibilità di definire delle maschere per i vari segnali, e costruire quindi le parole in uscita componendole con la somma o con un `OR`, indipendentemente dal loro stato di logica negativa o positiva. Per esemplificare, se devo definire una maschera con i valori `SOCX` e `READ`, con `SOCX` a logica negativa, posso usare il comando `netmask SOCX` e scrivere `READ + SOCX`, invece di `READ & (!SOCX)` nella istruzione e `maschera | SOCX` *per tutte le istruzioni* per cui `SOCX` non è asserito. Il formato è:

`netmask` [valore]

2.4 Le istruzioni

Vi sono 12 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
`nop`	0x30XX	nessuno	Nessuna operazione

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, **diminuito di 1**. Quindi indicando un intervallo di 9 si ottengono 10 passi da .125 μsec. 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 bit 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 `stop`, 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]

L'ultima istruzione (`nop`) è un istruzione *segnaposto*, lo SVB1 non compie nessuna operazione, lasciando lo stato del sistema invariato. È stata introdotta per poter avere dei programmi modificabili durante il funzionamento, tramite la sostituzione di `nop` con un comando diverso (ad esempio un `jump`). Viene implementata direttamente dall'assembler generando un salto incondizionato all'istruzione successiva. Il formato è:

```
nop
```

2.5 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 complessa, 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	$A@B = 256\dot{A} + B$

Capitolo 3

L'uso del programma assemblatore

Il programma **svb1asm** permette il passaggio da uno pseudoprogramma scritto utilizzando i codici mnemonici dell'assembler e i simboli che vengono definiti dall'utente per generare un file *oggetto* cioè direttamente eseguibile dal generatore di sequenze, SVB1. L'uso è semplice, occorre generare un file assembler (`<nome>.asm`) e chiederne la compilazione con il comando:

`svb1asm  <nome>`

Non va specificata l'estensione (`.asm`). Il programma genera i messaggi diagnostici e due files, un file di listato (`<nome>.lst`) che comprende le informazioni di compilazione in un formato adatto alla lettura ed un file oggetto (`<nome>.obj`) direttamente eseguibile dallo SVB1.

Abbiamo descritto precedentemente gli elementi disponibili per costruire il programma assembler. Diamo ora un esempio di un semplice programma, che genera un'onda quadra asimmetrica di 0.4MHz su tutte le linee in uscita.

Il sorgente del programma è il seguente:

```
;  flip
;  this program flip all lines at 0.4Mhz
;  for a second
        orgd    #0
mark    data    #FF @ #0A
space   data    #00 @ 8

        org     0

start
        loadc   19
lc
        loadb   99
```

```
lb
        loada   199
la
        outm    mark
        outm    space
        loopa   la
        loopb   lb
        loopc   lc
;
        outm    mark
        rest    start
```

Il programma, dopo tre linee di commenti, definisce la locazione di origine dei dati, con il comando `orgd`. Di seguito vengono definite due maschere, una con le linee tutte ad uno ed una con le linee tutte a zero. La prima ha una durata di 11 cicli elementari (`#0A` vale 10 in esadecimale) la seconda di 9 cicli, ciascuno di 0.125 μsec.

Il programma procede poi a definire l'indirizzo di inizio delle istruzioni, comando `org`. Seguono poi le istruzioni che caricano i registri C, B ed A con i valori che complessivamente permettono 400.000 iterazioni. Ricordiamo che il valore massimo contenuto in un registro è 255, che implica 256 ripetizioni. Interallacciate ai `load` ci sono le label per avre i tre cicli di loop annidati.

Seguono le due istruzioni (`outm`) che producono l'ouput delle due metá della semionda (in questo caso asimmetrica). Notiamo come l'istruzione `outm` richieda un valore precedentemente memorizzato, e **non** accetti un valore immediato.

Seguono le tre istruzioni di loop, che fanno riferimento alle labels precedentemente definite. Il tempo totale è di un secondo, come precedentemente calcolato.

Le ultime due istruzioni pongono rispettivamente le linee tutte ad un valore alto e fanno ripartire la generazione di sequenze, se, nel frattempo, non è stato ricevuto un comando `stop`.

La compilazione di questo programma procede come di seguito:

```
darkstar-97> svb1asm flip
  svb1asm $Revision: 1.5 $
 input file flip.asm
first pass
loop pass
second pass
 0000 0000 0000 ---- ;  flip
 0001 0000 0000 ---- ;  this program flip all lines at 0.4Mhz
 0002 0000 0000 ---- ;  for a second
```

```
<...analisi del programma rimossa...>

 Total errors 0  Total Warnings 0

darkstar-98>
```

Come già ricordato, il programma genera due files, un file di listato (`<nome>.lst`) che comprende le informazioni di compilazione in un formato adatto alla lettura ed un file oggetto (`<nome>.obj`) direttamente eseguibile dallo SVB1. Nel nostro caso il file listato è:

```
0000 0000 0000 ---- ;  flip
0001 0000 0000 ---- ;  this program flip all lines at 0.4Mhz
0002 0000 0000 ---- ;  for a second
0003 0000 0000 ----          orgd    #0
0004 0000 ff0a ---- mark     data    #FF @ #0A
0005 0001 0008 ---- space    data    #00 @ 8
0006 0000 0000 ----
0007 0000 0000 ----          org     0
0008 0000 0000 ----
0009 0000 0000 ---- start
0010 0000 6013 ----          loadc   19
0011 0001 0000 C--- lc
0012 0001 5063 C---          loadb   99
0013 0002 0000 CB-- lb
0014 0002 40c7 CB--          loada   199
0015 0003 0000 CBA- la
0016 0003 f000 CBA-          outm    mark
0017 0004 f001 CBA-          outm    space
0018 0005 8003 CBA-  loopa   la
0019 0006 9002 CB--  loopb   lb
0020 0007 a001 C---  loopc   lc
0021 0008 0000 ---- ;
0022 0008 f000 ----          outm    mark
0023 0009 1000 ----  rest    start
```

Osserviamo come insieme alle informazioni di compilazioe, cioè gli indirizzi di memoria e i relativi valori, il compilatore evidenzi l'estensione dei loop, identificati dal registro di controllo, per una più agevole identificazione dei flussi di esecuzione.

Per completezza elenchiamo le prime righe del file eseguibile (`.obj`). Tale file, in formato ascii, può essere direttamente letto dai programmi che

controllano Fasti, cioè **Ftest** e **GClient**, agevolando la sperimentazione di differenti forme d'onda.

```
54 10 00 00
54 10 01 00
54 10 00 0a
54 10 01 ff
54 10 02 08
54 10 03 00
54 40 00 13
54 40 01 60
 . . .
```

Nel nostro caso il file `obj` generato è lungo 26 linee di 4 bytes ciascuna, noi ne mostriamo le prime 8, che sono relative alla porzione dati ed alla prima istruzione del programma.

Il formato del file obj è un poco verboso. Ad ogni istruzione e ad ogni dato definito in memoria, corrispondono due bytes. Per ogni byte viene generata una riga che comprende 4 elementi: Il primo è sempre il comando di trasmissione (54_{16}, in ascii 'T'), il secondo è il comando di scrittura (10_{16} per i dati e 40_{16} per i codici eseguibili), il terzo è l'indirizzo, ed il quarto il byte da inviare. Il file così composto ha il grosso vantaggio che può essere inviato direttamente allo SVB1 tramite canale seriale senza alcuna ulteriore modifica.

Capitolo 4

L'emulatore

Al fine di verificare l'effettivo funzionamento di una forma d'onda e quindi facilitarne lo sviluppo, sono stati sviluppati due programmi, **EmuSvb** e **GsvbPlot**. Il programma **EmuSvb** emula dettagliatamente il funzionamento di *Seqproc*, la parte di generazione delle sequenze di SVB1 e produce in uscita un file (di estensione .wvf) che contiene il risultato di questa generazione.

L'uso del programma **EmuSvb** è semplice, nel formato Unix standard il comando è:

```
emusvb [-s|t] [-d] <inputfile>
```

Il file di ingresso (`<inputfile>`, da indicarsi senza specificarne l'estensione (.obj), è il file oggetto generato dal programma assemblatore. Questo file viene *eseguito* da **EmuSvb**, simulando i comportamenti dello SVB1.

I due argomenti `[-s|t]` sono mutuamente esclusivi. Se viene indicato `-t` (il default) allora, il file di uscita è minimale e le forme d'onda vengono descritte con il minimo numero di maschere, concatenando istruzioni diverse che danno lo stesso output. Specificando invece `-s` nel file in uscita sono presenti gli output di tutte le istruzioni `outm`. La differenza sarà chiarita meglio quando più avanti sarà descritto il formato del file di uscita.

L'opzione di debug `[-d]` permette di avere un *trace* dell'esecuzione, cioè l'indiczione delle istruzioni man mano che vengono eseguite. Questa opzione è assai utile per ricostruire il flusso dell'esecuzione e risolvere problemi di debug altrimenti difficilmente rintracciabili.

4.1 Il formato del file di uscita.

Il formato dei files in uscita è piuttosto semplice. Questi sono dei files ASCII, con estensione `.wvf` (waveform), con due colonne. La prima colonna indica il tempo corrente in unità di cicli di clock, ora .125μsec, la seconda la maschera di uscita, **dopo** tale istante di tempo. Entrambi i numeri sono in formato decimale.

È possibile avere dei commenti, tutte le righe che cominciano con il carattere '`#`' sono ignorate ai fini della forma d'onda. Se in un commento compare la parola '`labelN`' con $0 \leq N < 8$, quanto segue viene utilizzato dal programma **GsvbPlot** come etichetta per la linea N.

Se nella directory di lavoro è presente il file `emusvb.tpl` tutte le righe che cominciano con il carattere '`#`' sono copiate in cima al file di uscita. Questa caratteristica permetet di definire agevolmenet le label delle linne in uscita. Riportiamo quello relativo a NICS:

```
; file defining line labels
# label1 Pixel
# label2 Xsync
# label3 Line
# label4 Ysync
# label5 Reset
# label6 Socx
# label7 FirstRead
# label8 Read
```

Se è attiva l'opzione `-ti` (il default), allora nel file di uscita maschere consecutive dello stesso valore vengono riunite in una sola. Se al contrario è attiva l'opzione `-s` , nel files di uscita vengono riportate **tutte** le maschere in uscita. Questa caratteristica è particolarmente utile nel caso si programmi dei tempi di integrazione. Riportiamo un frammento preso da un file di forme d'onda per chip Hawaii:

```
integra
tint
                loadc   TINTC           ; TINT  seconds
tintc
                loadb   TINTB           ; each unit is worth 4 msec
tintb
                loada   TINTA           ; each unit is worth 20 usecs
tinta
                outm    BASE20          ; base delay - 20 usec
                loopa   tinta           ; inner  loop
                loopb   tintb           ; medium loop
                loopc   tintc           ; outer  loop

; End of integration loop : we jump back
                jump   integrb
```

Con i valori di default (`TINTA=199 TINTB=250`) i e con l'opzione `-s` , questo frammento genera 50000 righe *per ogni unità in* `TINTC`! Da qui la necessità di ridurre consistentemente il file di output.

4.2 Una forma d'onda di esempio

Per comprendere meglio l'uso del programma emulatorei ed il senso delle suo opzioni, verrà dato un esempio di sessione.

Si consideri un file di forme d'onda di esempio:

```
0000 0000 0000 ---- ;  this is a test 4
0001 0000 0000 ----         orgd    #0
0002 0000 ff0a ---- mark    data    #FF @ #0A
0003 0001 000a ---- space   data    #00 @ 10
0004 0002 0109 ---- dat0    data    #01 @ 9
0005 0003 0209 ---- dat1    data    #02 @ 9
0006 0004 0409 ---- dat2    data    #04 @ 9
0007 0005 0809 ---- dat3    data    #08 @ 9
0008 0006 1009 ---- dat4    data    #10 @ 9
0009 0007 2009 ---- dat5    data    #20 @ 9
0010 0008 4009 ---- dat6    data    #40 @ 9
0011 0009 8009 ---- dat7    data    #80 @ 9
0012 0000 0000 ----
0013 0000 0000 ---- t4
0014 0000 0000 ----         org     0
0015 0000 0000 ----
0016 0000 0000 ---- start
0017 0000 4002 ----         loada   2
0018 0001 0000 A--- la
0019 0001 f002 A---         outm    dat0
0020 0002 f003 A---         outm    dat1
0021 0003 f004 A---         outm    dat2
0022 0004 f005 A---         outm    dat3
0023 0005 f006 A---         outm    dat4
0024 0006 f007 A---         outm    dat5
0025 0007 f008 A---         outm    dat6
0026 0008 f009 A---         outm    dat7
0027 0009 8001 A---         loopa   la
0028 000a 0000 ---- ;
0029 000a f000 ----         outm    mark
0030 000b f000 ----         outm    mark
0031 000c 1000 ----         rest    start
```

Il programma mostrato definisce 8 maschere che sono attive per 10 cicli di clock ed hanno ciascuna un solo bit attivo. Il programma vero e proprio consiste di un loop eseguito tre volte (nella riga 0017 viene caricato il numero 2 nel registro di loop), ed in ciascuna iterazione vengono poste in uscita le 8 maschere corrispondenti ai diversi bit.

Usciti dal loop, una maschera con tutti i bit alti viene posta in uscita due volte di seguito. Segue un restart, che permette la riesecuzione del programma, se non è stato inviato un comando `stop` allo SVB1.

L'esecuzione del programma **EmuSvb** dà come risultato:

```
darkstar-78> emusvb test4

  emusvb $Revision: 1.5 $
 input  file test4.obj
```

```
 output file test4.wvf
Processed 262 time slices  (at 8Mhz total time is 3.275000e-05 sec )
Correctness mask is AA00
Clocked pixels 6,  Converted pixels 4

darkstar-79>
```

Il programma riporta quanti cicli di output sono stati generati (in questo caso solo 262) ed il tempo di esecuzione. Segue la firma digitale (*Correctness mask*) di questo programma. Questa è maschera che viene generata dallo SVB1 in base alle forme d'onda in uscita, per verificare, durante il funzionamento, la corretta esecuzione dei programmi. Tale valore è leggibile dallo SVB1 tramite la seriale e viene pubblicato nello stato del sistema dal programma **Ftest**.

Le ultime due voci indicano quanti pixel sono stati mossi (sulla base delle transizioni della linea `0x1` e quante conversioni eseguite (transizioni positiva della linea `0x20`.

Se invece si abilita l'opzione `[-d]` si ha un *trace* del flusso di istruzioni. Riportiamo lo output per il programma sopra riportato:

```
darkstar-155> emusvb -d test4
  emusvb $Revision: 1.5 $
 option 100
 debug output enabled

 input  file test4.obj
 output file test4.wvf
0001 DWRITE  [0a]
0000 DWRITE  [ff]
0003 DWRITE  [0a]
0002 DWRITE  [00]
<porzione di output soppressa>
0017 PWRITE  [00]
0016 PWRITE  [f0]
0019 PWRITE  [00]
0018 PWRITE  [10]

0000 LOADA   [02]
0002 OUTM    [01] @ [09]
0004 OUTM    [02] @ [09]
0006 OUTM    [04] @ [09]
0008 OUTM    [08] @ [09]
000a OUTM    [10] @ [09]
000c OUTM    [20] @ [09]
000e OUTM    [40] @ [09]
0010 OUTM    [80] @ [09]
0012 LOOPA   [02]
0002 OUTM    [01] @ [09]
0004 OUTM    [02] @ [09]
0006 OUTM    [04] @ [09]
0008 OUTM    [08] @ [09]
000a OUTM    [10] @ [09]
000c OUTM    [20] @ [09]
000e OUTM    [40] @ [09]
0010 OUTM    [80] @ [09]
```

```
0012 LOOPA   [02]
0002 OUTM    [01] @ [09]
0004 OUTM    [02] @ [09]
0006 OUTM    [04] @ [09]
0008 OUTM    [08] @ [09]
000a OUTM    [10] @ [09]
000c OUTM    [20] @ [09]
000e OUTM    [40] @ [09]
0010 OUTM    [80] @ [09]
0012 loopA   [02]
0014 OUTM    [ff] @ [0a]
0016 OUTM    [ff] @ [0a]
0018 REST    [0]
Processed 262 time slices  (at 8Mhz total time is 3.275000e-05 sec )
Correctness mask is AA00
Clocked pixels 6,  Converted pixels 4

darkstar-156>
```

Nello output vi sono tre colonne: la prima indica l'indirizzo, di memoria o di programma, la seconda l'operazione eseguita, la terza l'eventuale argomento.

Il programma comincia con l'esecuzione del file di ingresso, immagazzina cioè i dati (istruzione `DWRITE`) ed il programma (istruzione `PWRITE`) nelle diverse locazioni. Successivamente **EmuSvb** passa ad eseguire il programma immagazzinato, a partire dalla locazione 0 (nel nostro caso esegue `LOADA [02]`. Come abbiamo descritto prima il corpo del loop (l'emissione di otto maschere) viene eseguito tre volte. **EmuSvb** segnala il comportamento dei loop mostrando l'istruzione in maiuscolo (`0012 LOOPA [02]`) se il loop è ancora attivo (il registro non è zero). Se il flusso del programma esce dal loop (il registro è a zero), l'istruzione viene mostrata in minuscolo (`0012 loopA [02]`). A questo punto il programma, uscito dal loop esegue due istruzioni consecutive di output con tutti i bit ad uno, e torna a rieseguire il programma.

Appendice A

Forme d'onda per Nics

Riportiamo in appendice il file di programmazione per gli array Hawaii 1024×1024, come quello utilizzato su Nics. Tale file (nics_3) comprende solo le forme d'onda necessaria alle acquisizioni normali, senza riportare quelle che vengono utilizzate tra una misura scientifica e l'altra per meglio cancellare gli effetti di memoria (forma d'onda nics_4, riportato da un rapporto successivo).

```
; nics_3.asm 20-12-2002
; Nics Waveforms for SVB1 serializer.
; Integration time loop at 0x80
; implementation of mask for lines in negative logic
; half wave for x (pixel) and y (line) clock lines
;
; WARNING: all repetition index means value+1:
;  0 means 1 iteration, 2 means 3 iterationis, 255 (the max)
;    means 256 iterations
; This apply on BOTH loop and output mask, so (LINE @ 0) means LINE for 1
; clock cycle
; $Log: fastimanual.tex,v $
; Revision 1.5  2004/05/26 13:20:06  baffa
; Some addenda for wvf format.
;
; Revision 1.4  2004/05/26 08:15:24  egiani
;
; Correzioni.
;
; Revision 1.3  2004/05/25 08:56:56  baffa
; Alcune aggiunte al formato dei files in uscita.
;
; Revision 1.2  2004/05/20 13:05:38  baffa
; Added some parts on emusvb
;
; Revision 1.1  2004/05/20 07:44:48  baffa
; Manual for Fasti software. Part I - svb1asm and emusvb
;
; Revision 1.21  2004/03/26 15:10:21  baffa
; We implemented a 4 shells integration time loops.
;
; <removed>
;
```

```
; some symbols
;
ZERO            equ     #0              ; all void
BASE            equ     ZERO            ; standard no operation pattern
PIXEL           equ     #01             ; xclock
XSYNC           equ     #02             ; xsync (LSYNC)
LINE            equ     #04             ; yclock
YSYNC           equ     #08             ; ysync (FSYNC)
RSTX            equ     #10             ; Reset of all Quadrants
SOCX            equ     #20             ; ADC start (Start Of Conversion)
PREAD           equ     #40             ; Flag of first read for fasti/buffer
READ            equ     #80             ; power to output buffers
mul             equ     8               ; step (1 millisecond)
ROWS            equ     511             ; Number of rows in a quadrant
ROWS4           equ     127             ; Number of rows in a quadrant/4
ROWS2           equ     255             ; Number of rows in a quadrant/2
ROWSL           equ     1               ; Number of loop to complete a quadrant
COL4            equ     127             ; Number of columns in a quadrant/4
COL2            equ     255             ; Number of columns in a quadrant/2
COLL            equ     1               ; Number of loop to complete a quadrant
shold           equ     4               ; sample and hold time 625ns
pixwait         equ     2*mul-3         ; Pixel set time. Now 2us, 380 Mhz

; mask for lines in negative logic
                nmask   XSYNC + YSYNC + RSTX + SOCX
; here we define the global output masks and symbols
                orgd    zero            ; start of output masks
dummy255        data    BASE @ 255      ; 255 * noop
smallnoharm     data    BASE @ (20*mul-1); the old small_no_harm
BASE20          data    BASE @ (20*mul-1); base delay for TINT (20usecs)
;
; values for init and frame reset
VOID3           data    BASE @ 2        ; BASE for 375ns
VOID2           data    BASE @ 1        ; BASE for 250ns
VOID1           data    BASE @ 0        ; BASE for 125ns
VOID3L          data    (BASE+LINE) @ 2 ; BASE for 375ns
VOID2L          data    (BASE+LINE) @ 1 ; BASE for 250ns
VOID1L          data    (BASE+LINE) @ 0 ; BASE for 125ns
LINE1           data    LINE @ 0        ; LINE for 125ns
LINE1R          data    LINE+READ @ 0   ; LINE+READ for 125ns
RES5            data    RSTX @ 4        ; RSTX for 625ns
RES5L           data    (RSTX+LINE) @ 4 ; RSTX for 625ns
YINIT           data    YSYNC @ 7       ; Y sync for 1 usec
YINITF          data    YSYNC+PREAD+READ @ 7 ; Y sync+first_read for 1 usec
YINITR          data    YSYNC+READ @ 7  ; Y sync+read for 1 usec
PRINIT          data    PREAD+READ @ 7       ; first_read for 1 usec
YINITY          data    YSYNC+LINE @ 7  ; Y sync+line for 1 usec
XINIT           data    XSYNC @ 2       ; X sync for 375ns
XSYNC5          data    (XSYNC+READ) @ 4; X sync+read for 625ns
XSYNC24         data    (XSYNC+READ) @ pixwait ; X sync+read for 1 pixel
XSYNC1P         data    (XSYNC+READ+PIXEL) @ 0; X sync+read+pixel for 125ns
PIXEL3          data    (PIXEL+READ) @ 2; Xclock+read for 375ns
PIXEL1          data    (PIXEL+READ) @ 0; Xclock+read for 125ns
READ3           data    READ  @ 2       ; READ for 375ns
READ8           data    READ  @ 7       ; READ for 1 usec
SOCX2           data    (SOCX+READ) @ shold ; SOCX+read for shold
READ1           data    READ  @ 1       ; READ for 125ns
READ24          data    READ  @ pixwait ; READ for 3000ns
SOCX2P          data    (SOCX+READ+PIXEL) @ shold ; SOCX+read for shold
READ1P          data    (READ+PIXEL) @ 1      ; READ for 125ns
READ24P         data    (READ+PIXEL) @ pixwait; READ for 3000ns
```

```
;
XSYNC5L         data    (XSYNC+READ+LINE) @ 4 ; X sync+read for 625ns
XSYNC24L        data    (XSYNC+READ+LINE) @ pixwait ; X sync+read for 1 pixel
PIXEL3L         data    (PIXEL+READ+LINE) @ 2 ; Xclock+read for 375ns
XSYNC1LP        data    (XSYNC+READ+LINE+PIXEL) @ 1 ; X sync+read+pixel for 125ns
PIXEL1L         data    (PIXEL+READ+LINE) @ 0 ; Xclock+read for 125ns
READ3L          data    (READ+LINE)  @ 2      ; READ for 375ns
READ8L          data    (READ+LINE)  @ 8      ; READ for 1 usec
SOCX2L          data    (SOCX+READ+LINE) @ shold  ; SOCX+read for shold
READ1L          data    (READ+LINE)  @ 1      ; READ for 125ns
READ24L         data    (READ+LINE)  @ pixwait; READ for 3000ns
SOCX2PL         data    (SOCX+READ+PIXEL+LINE) @ shold ; SOCX+read for shold
READ1PL         data    (READ+PIXEL+LINE) @ 1 ; READ for 125ns
READ24PL        data    (READ+PIXEL+LINE) @ pixwait; READ for 3000ns
;
;
TINT            equ     1000            ; base time in milliseconds
TINTC           equ     0               ; TINT secs
TINTB           equ     0               ; 1000 msecs
TINTA           equ     0               ; 4000 usecs
TINTD           equ     199             ; 4000 usecs
nreset          equ     63              ; how many resets to perform

; Global start, page 0
                org     0
start
                outm    smallnoharm     ; 20 usec of void

;
; We perform some reset
startr
                loadc   nreset          ; number of reset to perform
resl0
; init of frame (wf10)
                outm    VOID2           ; all quiescent (250 nsec)
                outm    YINIT           ; Y sync (1 usec)
                outm    VOID2           ; all quiescent (250 nsec)
                outm    XINIT           ; X sync ( 375ns)
                outm    VOID2           ; all quiescent (250 nsec)

; reset loop (loop on 512 row, 256 times wf70)
                loada   ROWS2           ; init of row counter
                outm    VOID1           ; all quiescent (125 nsec)
resl2
                outm    LINE1           ; Y clock (125 nsec)
                outm    VOID3L          ; all quiescent (375 nsec)
                outm    RES5L           ; Reset (625 nsec)
                outm    VOID1L          ; all quiescent (125 nsec)
                outm    VOID3           ; all quiescent (375 nsec)
                outm    RES5            ; Reset (625 nsec)
                outm    VOID1           ; all quiescent (125 nsec)
; end of inner loop
                loopa   resl2

; loop on many resets
                loopc   resl0

; Reading frame: first read
; init of frame (wf10)
                outm    READ1           ; READ   (1usec)
                outm    PRINIT          ; first read (1 usec)
                outm    YINITF          ; Y sync + first read (1 usec)
```

```
                outm    PRINIT          ; first read (1 usec)
                outm    READ8           ; READ   (1usec)
; begin of loop on row (256*2)
                loadc   ROWS2           ; init of row counter
read2
; yshift (wf20)
                outm    READ1           ; only read
                outm    READ1L          ; Y clock (125 nsec)
;
; init of an odd row (wf50)
                outm    XSYNC24L        ; XSYNC+READ (1 pixel)
                outm    READ8L          ; READ   (1usec)
                outm    READ24PL        ; READ (3000 ns)
                outm    READ24L         ; READ (3000 ns)
;
; begin of loop on columns (512 colums, 256 the waveform)
                loada   COL2            ; init of row counter
read4o
;
; we read an odd pixel (wf50):
                outm    READ24L         ; READ (3000 ns)
                outm    SOCX2L          ; SOCX+READ (250 ns)
                outm    READ1L          ; READ (125 ns)
; we read an even pixel (wf50):
                outm    READ24PL        ; READ (3000 ns)
                outm    SOCX2PL         ; SOCX+READ (250 ns)
                outm    READ1PL         ; READ (125 ns)

;
; end of inner pixel loop
                loopa   read4o
                outm    READ24L         ; READ (3000 ns)
                outm    READ1PL         ; To shift xsync out of
                outm    READ1L          ; the shift register
                outm    READ24L         ; READ (3000 ns)
;
; init of a even row (wf50)
                outm    XSYNC24         ; XSYNC+READ (1 pixel)
                outm    READ8           ; READ   (1 usec)
                outm    READ24P         ; READ (3000 ns)
                outm    READ24          ; READ (3000 ns)
;
; begin of loop on columns (512 colums, 256 the waveform)
                loada   COL2            ; init of row counter
read4e
;
; we read an odd pixel (wf50):
                outm    READ24          ; READ (3000 ns)
                outm    SOCX2           ; SOCX+READ (250 ns)
                outm    READ1           ; READ (125 ns)
; we read an even pixel (wf50):
                outm    READ24P         ; READ (3000 ns)
                outm    SOCX2P          ; SOCX+READ (250 ns)
                outm    READ1P          ; READ (125 ns)

;
; end of inner pixel loop
                loopa   read4e
                outm    READ24          ; READ (3000 ns)
                outm    READ1P          ; To shift xsync out of
                outm    READ1           ; the shift register
                outm    READ24          ; READ (3000 ns)
```

```
; end of inner row loop
                loopc   read2

;
;  integration time
;  the integration time loop is in a standar place so
;  we can tune it easily

                jump   integra
; we jump here from integration delay loop
integrb


; Reading frame: second read
; we have the same timings of first read (I tried to ....)
; init of frame (wf10)
                outm    VOID2           ; all quiescent (250 nsec)
                outm    READ1           ; only read  (125 usec)
                outm    READ8           ; READ   (1 usec)
                outm    YINITR          ; Y sync (1 usec)
                outm    READ8           ; READ   (1 usec)
                outm    READ8           ; READ   (1 usec)
; begin of loop on row (256*2)
                loadc   ROWS2           ; init of row counter
read3
; yshift (wf20)
                outm    READ1           ; only read
                outm    READ1L          ; Y clock (125 nsec)
;
; init of an odd row (wf50)
                outm    XSYNC24L        ; XSYNC+READ (1 pixel)
                outm    READ8L          ; READ   (1 usec)
                outm    READ24PL        ; READ (3000 ns)
                outm    READ24L         ; READ (3000 ns)
;
; begin of loop on columns (512 colums, 256 the waveform)
                loada   COL2            ; init of row counter
read5o
;
; we read an odd pixel (wf50):
                outm    READ24L         ; READ (3000 ns)
                outm    SOCX2L          ; SOCX+READ (250 ns)
                outm    READ1L          ; READ (125 ns)
; we read an even pixel (wf50):
                outm    READ24PL        ; READ (3000 ns)
                outm    SOCX2PL         ; SOCX+READ (250 ns)
                outm    READ1PL         ; READ (125 ns)

;
; end of inner pixel loop
                loopa   read5o
                outm    READ24L         ; READ (3000 ns)
                outm    READ1PL         ; To shift xsync out of
                outm    READ1L          ; the shift register
                outm    READ24L         ; READ (3000 ns)
;
; init of a even row (wf50)
                outm    XSYNC24         ; XSYNC+READ (1 pixel)
                outm    READ8           ; READ   (1 usec)
                outm    READ24P         ; READ (3000 ns)
                outm    READ24          ; READ (3000 ns)
;
```

```
; begin of loop on columns (512 colums, 256 the waveform)
                loada   COL2            ; init of row counter
read5e
;
; we read an odd pixel (wf50):
                outm    READ24          ; READ (3000 ns)
                outm    SOCX2           ; SOCX+READ (250 ns)
                outm    READ1           ; READ (125 ns)
; we read an even pixel (wf50):
                outm    READ24P         ; READ (3000 ns)
                outm    SOCX2P          ; SOCX+READ (250 ns)
                outm    READ1P          ; READ (125 ns)

;
; end of inner pixel loop
                loopa   read5e
                outm    READ24          ; READ (3000 ns)
                outm    READ1P          ; To shift xsync out of
                outm    READ1           ; the shift register
                outm    READ24          ; READ (3000 ns)
; end of inner row loop
                loopc   read3


                outm    smallnoharm     ; 20 usec of void
                rest    0

;  to have an accessible location of shold and pixwait
                org     #78
                loada   shold
                loada   pixwait

;
;  integration time
;  we have the integration delay loop in a fixed location
;  for ease of modification
                org     #80

; Here we start TINT

integra
tint
                loadc   TINTC           ; TINT  seconds
tintc
                loadb   TINTB           ; each unit is worth 4 msec
tintb
                loada   TINTA           ; each unit is worth 20 usecs
tinta
                loadd   TINTD           ; each unit is worth 20 usecs
tintd
                outm    BASE20          ; base delay - 20 usec
                loopd   tintd           ; base   loop
                loopa   tinta           ; inner  loop
                loopb   tintb           ; medium loop
                loopc   tintc           ; outer  loop

; End of integration loop : we jump back
                jump   integrb

                end
```

Appendice B

Vim syntax-file

Riportiamo il file di configurazione per l'editor Vim che permette il riconoscimento della sintassi dello **svb1asm** durante una fase di editing, facilitando così lo sviluppo.

```
" Vim syntax file
" Language: SVB Assembler
" Maintainer: C.Baffa
" 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          "nop"

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


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
```

Indice

Bibliografia

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

[2] Baffa, C., C., Biliotti, V., Progetto Fasti - Un assembler per lo SVB1, Rapporto interno dell'Osservatorio Astrofisico di Arcetri, **N°1/2000.**

[3] Baffa, C., The Fasti Project, Memorie della Società Astronomica Italiana, 2003, **74**, p165.

[4] Baffa, C., Biliotti, V., Checcucci, A., Gavrioussev, V., Gennari, S., Giani, E., Lisi, F., Marcucci, G., Sozzi., M., "The Fasti Project", ADASS XII ASP Conference Series, Vol. **295**, 2003, Payne, H., Jedrzejewski, R., Hook, R. Eds., p.355

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

[6] Checcucci, A., Baffa, C., Giani, E., "Progetto Fasti – Fasti Global Controller, Rapporto interno dell'Osservatorio Astrofisico di Arcetri, **N°3/2001**.

[7] Giani, E., Checcucci, A., Baffa, C., "Progetto Fasti – Driver Linux per la scheda di acquisizione NI PCI-DIO32HS, Versione 2, Rapporto interno dell'Osservatorio Astrofisico di Arcetri, **N°3/2002**.