

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

**Matita come Supporto per Specifiche
Eseguibili: Formalizzazione Interattiva
dei Microcontroller a 8 bit Freescale**

Tesi di Laurea in Sistemi Operativi

Tesi di Laurea di:

COSIMO ARNDT OLIBONI

Relatore:

Dott. CLAUDIO SACERDOTI COEN

III Sessione

Anno Accademico 2006-2007

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

Matita come Supporto per Specifiche Eseguibili: Formalizzazione Interattiva dei Microcontroller a 8 bit Freescale

Tesi di Laurea in Sistemi Operativi

Tesi di Laurea di:

COSIMO ARNDT OLIBONI

Relatore:

Dott. CLAUDIO SACERDOTI COEN

Parole chiave: formalizzazione interattiva;
microcontroller a 8 bit; processori;
Matita; ambiente di dimostrazione assistita;
Freescale; Motorola;
HC05; HC08; HCS08; RS08.

III Sessione

Anno Accademico 2006-2007

Introduzione

Questa tesi descrive la formalizzazione, in forma di specifica eseguibile, di tutti i microcontroller Freescale basati su CPU a 8 bit (HC05, HC08, HCS08 e RS08). Come strumento è stato adottato l'ambiente di dimostrazione assistita Matita che, grazie al suo approccio interattivo alla formalizzazione, ha significativamente agevolato (rispetto ad altri ambienti non interattivi) il lavoro, permettendo di ampliare la portata dell'obiettivo: realizzare una macchina virtuale che simula in maniera completa il comportamento della memoria e della CPU di un'intera famiglia di microcontroller.

Il lavoro di formalizzazione realizzato estende un precedente esperimento condotto dal Dott. Claudio Sacerdoti Coen sullo specifico microcontroller MC9RS08KA2 ed esce dall'ambito delle semplici proof of concept, profilandosi come fase iniziale di un più ampio progetto. La formalizzazione del livello di esecuzione costituisce, infatti, la base per la verifica di correttezza di programmi assembler ed uno dei moduli logici richiesti per il funzionamento di un compilatore certificato multiplatforma (vedi progetto CompCert).

La natura eseguibile della specifica, all'interno del panorama delle formalizzazioni realizzate dagli anni '80 ad oggi, si pone come un elemento di novità. Il supporto all'esecuzione, fornito solitamente dai simulatori di tipo tradizionale che accompagnano i processori commerciali, si unisce alla logica del sistema formale, aprendo la strada a vari tipi di sperimentazione (e.g.: verifica di proprietà dei programmi) che completeranno il progetto Freescale.

L'ambiente di formalizzazione utilizzato (Matita) si è trasformato, mediante l'estrazione in linguaggio OCAML delle definizioni realizzate, in un compilatore per un linguaggio con tipi dipendenti (e.g.: DML, CAYENNE, EPIGRAM). L'adozione sistematica di questi tipi all'interno di una specifica hardware introduce un nuovo e più stringente livello di correttezza all'interno di un ambito

che necessita della minor ambiguità possibile. La formalizzazione realizzata costituisce, quindi, anche un esempio di documentazione tecnica pubblica non ambigua che, facendo uso di un linguaggio simile a quello matematico, permette una condivisione dei dettagli implementativi più efficiente dell'eventuale disponibilità del codice sorgente di un simulatore open source.

La macchina virtuale parametrica realizzata, contrariamente a tutti gli altri lavori di formalizzazione pubblicati, non si limita a simulare l'esecuzione di un singolo dispositivo hardware. Attraverso un'estesa esplorazione di tutte le componenti logiche e comportamentali comuni, è stato possibile riunire in unica formalizzazione la specifica eseguibile di ben 229 modelli diversi di microcontroller con CPU a 8 bit.

L'estrazione del codice OCAML risente naturalmente della natura parametrica delle definizioni realizzate (e.g.: controlli interni di tipo pattern match/case per determinare quale comportamento reale associare ad ogni entità logica parametrica) ma, tramite valutazione parziale, è comunque possibile ricavare, a partire dalla macchina virtuale parametrica generale, tutte le possibili (più compatte ed efficienti) macchine virtuali specifiche per ogni modello.

La presente tesi si compone di due capitoli:

- nel primo capitolo, di tipo compilativo, verranno introdotti molti concetti utili ad una migliore comprensione dei vantaggi (spesso trascurati o poco noti) offerti dalla formalizzazione software e (in particolar modo) hardware;
- nel secondo capitolo verranno invece descritte alcune delle caratteristiche più importanti (e.g.: problemi affrontati, scelte implementative, utilizzo dei tipi dipendenti) del lavoro di formalizzazione realizzato.

In appendice sarà anche possibile trovare la descrizione di alcuni primi esperimenti di verifica condotti facendo uso della specifica eseguibile.

Ringraziamenti

Colgo l'occasione per ringraziare il mio relatore Dott. Claudio Sacerdoti Coen che, durante i mesi dedicati alla realizzazione di questa tesi, mi ha fattivamente assistito e consigliato.

Desidero inoltre ringraziare la ditta Freescale Semiconductor Inc. che ha gentilmente e sollecitamente fornito l'attrezzatura utilizzata (compilatore CodeWarrior e USB Spider08) per condurre i test sul microcontroller HCS08.

Indice

	Introduzione	i
	Indice	v
1	Il Perché della Formalizzazione Software e Hardware	1
	1.1 Formalizzazione e Metodi Formali	1
	1.1.1 Spazio di Applicazione dei Metodi Formali	2
	1.1.2 Esempio di Metodi Formali applicati all'Hardware	4
	1.1.3 Formalizzazione	5
	1.2 Dimostrazione di Correttezza	7
	1.3 Disastri e Sicurezza	9
	1.3.1 Metodi Formali e Design	11
	1.3.2 Requisiti di Sicurezza e Standard	15
	1.4 Formalizzazione Assistita	19
	1.4.1 First-Order Highly-Automated Provers	20
	1.4.2 Higher-Order Tactic-Based Provers	22
2	Formalizzazione in Matita dei Microcontroller Freescale ...	29
	2.1 I Microcontroller a 8 bit Freescale	29
	2.1.1 Struttura della Formalizzazione	34
	2.2 Livello 1	35
	2.2.1 Tuple	35
	2.2.2 Option e Option Map	36
	2.2.3 Aritmetica	39

2.3	Livello 2	41
2.3.1	Rappresentazione degli ISA	41
2.3.2	Decodifica	45
2.3.3	Compilazione	46
2.4	Livello 3	50
2.4.1	Caratteristiche Comuni	50
2.4.2	Memoria a Funzione	51
2.4.3	Memoria ad Albero di Byte	52
2.4.4	Memoria ad Albero di Bit	55
2.4.5	Astrazione di Accesso	56
2.5	Livello 4	57
2.5.1	CPU	57
2.5.2	Stato	61
2.5.4	Instanziamento e Reset dello Stato	63
2.6	Livello 5	65
2.6.1	Modalità di Indirizzamento	65
2.6.2	Unificazione delle Logiche	67
2.6.3	Ciclo di Esecuzione	70
	Appendice: Test Svolti	77
A.1	Test stringReverse	78
A.2	Test countingSort	82
	Bibliografia	87

Capitolo 1

Il Perché della Formalizzazione Software e Hardware

1.1 Formalizzazione e Metodi Formali

“Formal methods are the use of mathematical techniques in the design and analysis of computer hardware and software; in particular, formal methods allow properties of a computer system to be predicted from a mathematical model of the system by a process related to calculation.” [Rus93]

La formalizzazione è un sottoinsieme del più vasto insieme dei metodi formali e, per questo motivo, la prima sezione si apre con una definizione del termine “metodi formali”, attraverso cui si può arrivare ad una migliore definizione del termine “formalizzazione”.

La definizione fornita da Rushby è inserita in un contesto orientato alla certificazione di sistemi critici ma sintetizza efficacemente alcuni punti generali riguardo ai metodi formali:

- il loro campo di applicazione spazia dal software all’hardware;
- operano attraverso tecniche matematiche;
- si prefiggono di predire alcune proprietà specifiche di un sistema, quali l’assenza di deadlock nell’accesso ad una o più risorse condivise, l’esistenza di un lower bound o upper bound al tempo di esecuzione, ed altre ancora.

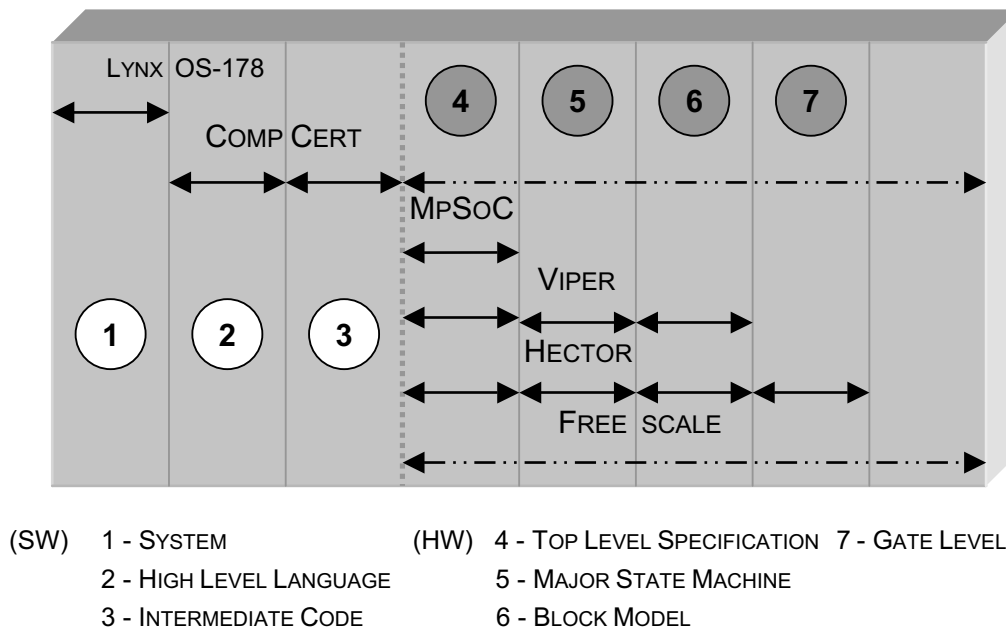


Figura 1.1.1: Spazio di applicazione dei metodi formali

Solo il primo punto è strettamente inerente sia ai metodi formali che alla formalizzazione ed il suo approfondimento contribuisce a dare una precisa collocazione alla formalizzazione dei microcontroller Freescale.

1.1.1 Spazio di Applicazione dei Metodi Formali

I lavori di formalizzazione ed applicazione dei metodi formali possono essere visti come elementi componibili di una catena che, idealmente, estende l'analisi di un sistema a partire dal software fino al particolare hardware designato per la sua esecuzione (Figura 1.1.1). La tabella riporta alcuni esempi che saranno citati in questa tesi, ordinati in senso discendente lungo la scala che porta dal software all'hardware. Nello specifico sono raffigurate:

- un'applicazione dei metodi formali ristretta unicamente al primo e più generale livello software dei sistemi operativi (LynxOS-178);
- una formalizzazione interattiva che, a partire dal linguaggio C minor, copre tutti i livelli fino a produrre un compilatore certificato (CompCert). La freccia tratteggiata indica che i livelli di formalizzazione hardware assumono la forma di un'unica specifica eseguibile;

- un'applicazione dei metodi formali ristretta unicamente al primo e più generale livello hardware di interazione fra componenti (MpSoC);
- una formalizzazione non interattiva che copre tutti i livelli hardware, mediante trattazione separata ed indipendente (Viper);
- una formalizzazione non interattiva che copre quasi tutti i livelli hardware, mediante trattazione separata ed indipendente, fermandosi prima dell'ultimo livello di architettura delle porte logiche (Hector);
- la formalizzazione interattiva descritta in questa tesi che, a partire dalle specifiche operative dei microcontroller Freescale, copre tutti i livelli fino a produrre una macchina virtuale. La freccia tratteggiata indica, come nel caso CompCert, che i livelli di formalizzazione hardware assumono la forma di un'unica specifica eseguibile.

Ampliando il campione all'intero panorama dei lavori realizzati si può introdurre una prima suddivisione: software e hardware, separati idealmente dal terzo livello (il codice intermedio). Quest'ultimo sembra costituire una barriera ideale che solo pochissime applicazioni dei metodi formali riescono a superare. Non si può nemmeno sperare, purtroppo, di comporre i lavori esistenti ed ottenere dei modelli che, spaziando fra l'hardware già sottoposto a formalizzazione, permettano di tradurre le proprietà software in invarianti di esecuzione. Eterogeneità di approccio ed inconciliabilità degli strumenti utilizzati contribuiscono alla separazione in due mondi distinti.

Un'ulteriore suddivisione distingue i lavori secondo l'uso o meno di strumenti che supportino, oltre alla formalizzazione, la successiva produzione di codice eseguibile.

Il lavoro di questa tesi può essere quindi classificato come:

- ristretto al campo hardware;
- finalizzato alla produzione di una specifica eseguibile.

Gli esempi che seguiranno, a partire dalla prossima sezione, si concentreranno prevalentemente sull'hardware, affiancati da concetti e definizioni che, al contrario, avranno sempre validità estesa all'intero spazio di applicazione dei metodi formali qui considerato.

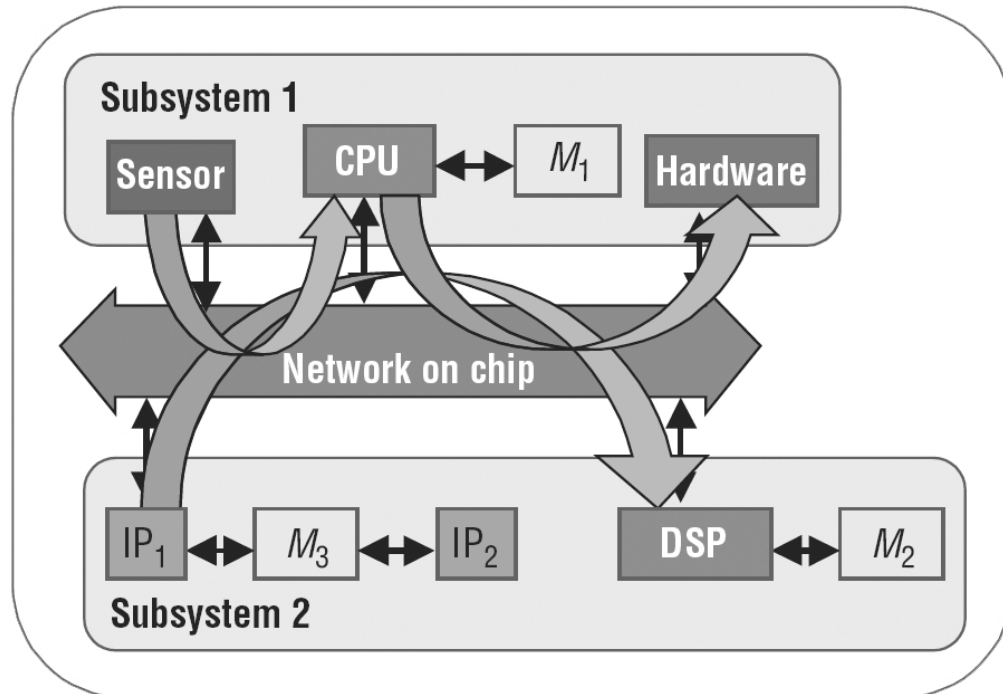


Figura 1.1.2: Interazione tramite Network on Chip in un MpSoC [RJE03]

1.1.2 Esempio di Metodi Formali Applicati all'Hardware

Un esempio di impiego di metodi formali per determinare l'esistenza di un upper bound al tempo di esecuzione è l'analisi delle performance di un Multi Processor System on Chip (MpSoC). Gli MpSoC sono dei sistemi multicore integrati, in cui numerosi core specializzati ed eterogenei tra loro (e.g.: CPU, sensori hardware, interfacce di periferiche, DSP di encoding/decoding) sono connessi tra loro tramite un protocollo interno di comunicazione simile, per comportamento e complessità, ad una intranet: il Network on Chip (Figura 1.1.2).

I singoli componenti (i core specializzati) sono dispositivi hardware il cui sviluppo ha superato da tempo la fase iniziale di test e debug, le cui proprietà locali hanno già subito un sufficiente livello di indagine e certificazione. Il problema principale resta, quindi, il comportamento globale determinato dalle performance del Network on Chip, per il cui studio le tradizionali tecniche di simulazione possono rivelarsi inadeguate.

La simulazione, basata sull'analisi dei risultati di un campione rappresentativo di corse, ha bisogno di riuscire a definire dei pattern di verifica che evidenzino i comportamenti limite del sistema. La ricerca di questi pattern, a causa del grado di complessità del sistema stesso, non può essere affidata alla composizione di pattern già noti o all'identificazione manuale da parte degli sviluppatori. I metodi formali offrono uno strumento più sofisticato, flessibile e dotato di scalabilità:

“When simulation falls short, formal approaches become more attractive, offering systematic verification based on well-defined models. Formal analysis guarantees full performance corner-case coverage and bounds for critical performance parameters.” [RJE03]

Si conclude qui il breve accenno ai metodi formali (per un approfondimento dell'argomento si rimanda a [RJE03]) i cui requisiti possono essere ora confrontati con quelli più forti richiesti dalla formalizzazione.

1.1.3 Formalizzazione

“Numerous definitions of implementation relations have been proposed. The one we choose has a simple intuitive definition. We take two nondeterministic agents, called the specification and the realization. We say that the realization implements the specification if, for any stimulus on which the specification produces a response, the response of the realization to this stimulus is always one of the responses that we could see from the specification.” [Bea93]

Un microprocessore è un sistema di calcolo che svolge la funzione di agente reale di esecuzione di software. Il termine “agente reale” (realizzazione) indica una delle molte macchine fisiche (hardware) capaci di eseguire in maniera equivalente lo stesso linguaggio (software) descritto da una macchina astratta (specifica). Punto cruciale di questa definizione è l'interpretazione della relazione di equivalenza tra la macchina astratta descritta dalla specifica e la macchina reale implementata. Entrambe le macchine sono analizzate come delle black-box, dove il confronto avviene unicamente tra l'insieme degli input e degli output possibili.

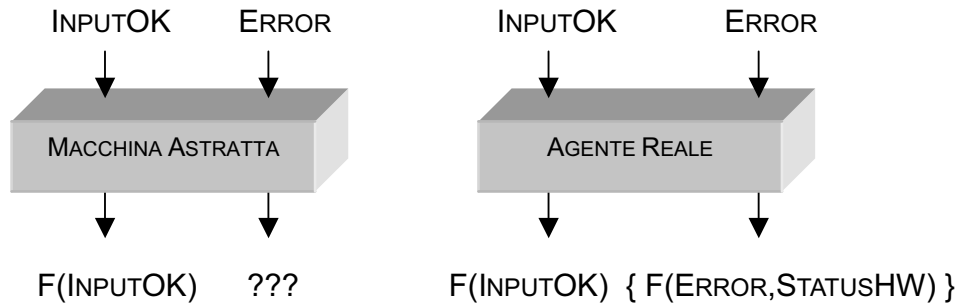


Figura 1.1.3: Comportamento non deterministico in presenza di errore

È importante notare che, contrariamente a quanto ci si aspetterebbe, le macchine sono ipotizzate non deterministiche, come in realtà avviene. Nelle specifiche gli input “corretti” ottengono una risposta univoca (deterministica), ma gli input “scorretti” o non sono trattati, o sono semplicemente descritti come output indefinito; negli agenti reali la risposta all’errore dipende, oltre che dall’input, anche dallo stato interno dell’hardware (e.g.: valori memorizzati nei flip-flop) invisibile all’esterno della black-box, determinando un output apparentemente non deterministico (Figura 1.1.3).

È evidente come questa interpretazione assuma validità più generale e si adatti correttamente anche alla descrizione del software. Si considerino, ad esempio, i singoli moduli di un sistema che dialogano tra loro tramite delle interfacce: in fase di implementazione possono verificarsi, a causa di sottospecificazione a livello di design, dei comportamenti non deterministici dovuti a componenti interne (e.g.: variabili locali, strutture dati locali).

Beatty introduce questa definizione trattando la formalizzazione del microprocessore Hector, di cui non erano disponibili le specifiche hardware, e sceglie di inserire dentro la black-box tutto il gate-level (la descrizione del microprocessore come architettura composta di porte logiche elementari). Questo approccio di netta separazione, nelle formalizzazioni non eseguibili a più livelli, può essere considerato come uno standard e richiederebbe sostanzialmente di esibire almeno due distinte formalizzazioni:

- una formalizzazione che partendo dall’Instruction Set Architecture (ISA) si fermi prima del gate-level;
- una formalizzazione del gate-level, che faccia uso di strumenti derivati da quelli usati in fase di design della circuiteria (e.g.: HDL).

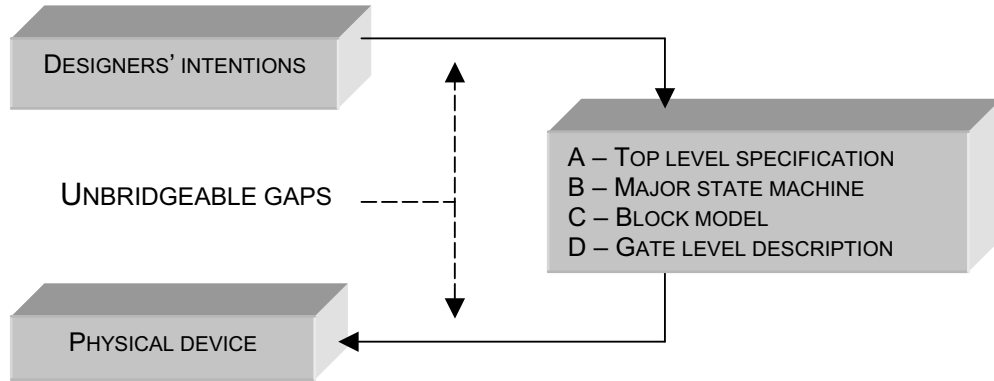


Figura 1.2: Diagramma della dimostrazione del Viper [Mac91]

1.2 Dimostrazione di Correttezza

“All known methods contributing to correctness have shortcomings that make it impossible to establish correctness beyond reasonable doubt. That is, establishing correctness is a matter of belief, not proof.” [AbrZel]

Purtroppo, anche disponendo di entrambe le formalizzazioni, bisogna prendere atto di una intrinseca impossibilità di completezza: non esistono, infatti, solo gli errori logici, dovuti ad un errore di design, ma anche quelli esclusivamente fisici. Un esempio famoso è quello del microprocessore Intel 80486 che, nel 1991, si rivelò soggetto a surriscaldamento in alcune sue componenti con conseguente malfunzionamento dei transistor coinvolti [Bea93].

Parlando di incompletezza delle tecniche di formalizzazione, a prescindere dalla scoperta o meno di bug nei microprocessori in commercio, il caso principale cui fare riferimento è quello del microprocessore Viper, il primo (e l'unico) ad essere venduto insieme ad una “dimostrazione di correttezza”.

Ciò suscitò molto clamore, controversie legali (concluse purtroppo senza arrivare a nessun pronunciamento definitivo sulla validità legale delle dimostrazioni matematiche) e la contestazione di Donald MacKenzie (Figura 1.2), più volte ripresa da tutti coloro che si sono occupati di dimostrazioni di correttezza:

“To say that the design of a device was ‘correct’ does not imply that the device would do what its designers intended: it means only that it is a correct implementation of the formal specification. Even the most detailed description of a device was still an abstraction from the physical object. Gaps unbridgeable by formal logic and mathematical proof must remain between VIPER’s level A description and the designers’ intentions, and between level D and the actual chip.” [Mac91]

Come conseguenza di un’affermazione di tale portata ci si potrebbe chiedere quale senso dare ai successivi lavori di formalizzazione gate-level condotti tra gli anni ‘90 ed oggi. La domanda non è retorica e la risposta riassume in sé l’obiettivo principale della formalizzazione condotta ad ogni livello: compilazione, specifiche, realizzazione hardware.

Definizione: *La formalizzazione ha come obiettivo il sottoporre software e hardware ad un’analisi il più approfondita possibile, in modo da:*

- *fornire uno strumento di razionalizzazione utilizzabile parallelamente o congiuntamente a tutte le fasi di sviluppo, dal design fino al debug;*
- *aumentare il nostro grado di confidenza, specialmente nell’ambito di sistemi critici [BoyYu92].*

Il termine “sistema critico” è definito molto chiaramente all’interno dei disclaimer che, sempre a partire dagli anni ’90, tutti i produttori di software e hardware hanno introdotto sistematicamente, alla stregua di uno standard, nelle licenze di utilizzo dei loro prodotti:

“XXX products are not authorized for use as critical components in life support devices or systems without the express written approval of XXX. As used herein:

1. *Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and (c) whose failure to perform when properly used in accordance with instructions for use provided in the labelling, can be reasonably expected to result in a significant injury of the user.*

2. *A critical component in any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.*
3. *XXX assumes no liability for incidental, consequential or special damages or injury that may result from improper use.”*

Sistemi critici sono tutti quegli apparati il cui malfunzionamento può causare il ferimento o il decesso di una o più persone ed ai quali, purtroppo, noi affidiamo quotidianamente, e più o meno consapevolmente, la nostra vita: aerei, apparecchiature mediche, centrali nucleari, armamenti. È legittimo chiedersi a questo punto quali ulteriori e più forti garanzie siano richieste al software e hardware utilizzati per il loro regolare funzionamento.

Per rispondere a questa domanda la prossima sezione esplorerà una serie di clamorosi disastri fino ad arrivare agli stringenti standard di sicurezza richiesti dall'industria avionica che, purtroppo, ancora oggi non richiede esplicitamente l'uso della formalizzazione nella certificazione dei sistemi critici adottati.

1.3 Disastri e Sicurezza

“When software fails, it invariably fails catastrophically.” [Rus93]

L'affermazione di Rushby può sembrare esagerata, irrealistica ed allarmista, ma rispecchia ciò che avviene nella realtà. L'errore software o hardware è per sua natura sottile, difficile da isolare, impossibile da escludere e, nel momento cruciale in cui si rivela, può condurre senza rimedio alla cessazione del funzionamento. In un sistema critico ciò equivale alla catastrofe. I seguenti esempi sono tratti da [Bea93, Jac94, Rus93].

1) Disastri medici

Nel 1986, negli ospedali americani era già in funzione una nuova apparecchiatura per il trattamento a raggi X che, in sostituzione ai tradizionali controlli di sicurezza meccanici, equipaggiava software di controllo: il Therac-25.

Un sistema di autodiagnostica controllava il tasso di irraggiamento, sostituendo il quotidiano controllo dei relè di sicurezza da parte dei tecnici degli ospedali. Il software di controllo non malfunzionò mai ma, a causa di un errore di design, si rivelò criptico e assolutamente inadeguato alla gestione del sistema:

- segnalava con la stessa modalità i malfunzionamenti lievi (variazioni inferiori all'1% nell'irraggiamento) e quelli critici (dose letale);
- permetteva all'operatore di decidere arbitrariamente se proseguire o meno nella fase di irraggiamento, saltando la necessaria fase di reinizializzazione e ricalibratura dell'apparecchio.

Il secondo difetto fu certamente quello decisivo che causò nell'arco di una decina d'anni 6 decessi. Come se non bastasse, altri tipi di dispositivi medici causarono, nel periodo dal 1983 al 1987, incidenti gravi. La Food and Drug Administration (FDA) riporta, per quel periodo, una stima totale di almeno un centinaio di morti dovuti all'utilizzo di software e hardware difettoso all'interno di sistemi critici o indispensabili al supporto della vita umana.

2) Disastri aerei

Un Boeing 767 precipitò nel 1991, causando 233 vittime, a causa di un'errata attivazione dell'invertitore di spinta di un motore. Quattro Airbus A320 precipitarono, nel periodo dal 1988 al 1994, a causa della mancata risposta dei controlli, causando un totale di 98 vittime.

L'attribuzione delle cause di questi incidenti è controversa ed alcune fonti sostengono l'assoluta bontà delle apparecchiature, facendo ricadere tutta la responsabilità sull'errore umano.

L'unico punto oggettivo a proposito è, come enunciato all'inizio del paragrafo, la difficoltà nell'isolare un errore software o hardware. Basti pensare alla differenza che intercorre fra l'individuare una anomalia all'interno di un sistema in funzione e l'indagare fra i rottami di un aereo ormai precipitato.

3) Disastri economici

Nel periodo dal 1989 al 1992 il microprocessore Intel80486, già in commercio,

rivelò una serie di bug (compreso un errore di calcolo in virgola mobile) che costrinsero la casa produttrice al ritiro e alla sostituzione gratuita dei microprocessori incriminati.

Nel 1994 la centrale nucleare canadese di Darlington adottò un nuovo tipo di controllo software per l'arresto di emergenza. Giunti quasi al momento di entrare in funzione, i tecnici si trovarono di fronte all'imposizione, da parte dell'ente di sorveglianza, di certificare il programma di controllo tramite formalizzazione.

Ci vollero dei mesi per analizzare al livello richiesto le 2.500 righe di codice, con un conseguente ritardo dell'entrata in funzione dell'impianto che costò 20 milioni di dollari al mese, più un costo, per la sola verifica, di 4 milioni di dollari. Uno dei costruttori dichiarò alla fine, esasperato, che da allora in poi si sarebbe affidato esclusivamente a controlli manuali.

Questi due incidenti evidenziano un punto molto importante riguardo alla formalizzazione: la sua adozione, nel caso si voglia o si debba farne uso, deve avvenire fin dalle prime fasi di design. Per i produttori di hardware impiegato in sistemi a criticità massima ciò è uno standard, ma i benefici economici per i produttori in genere non sarebbero affatto indifferenti: il risparmio è stimato in almeno 100.000 dollari per ogni errore rilevato durante le fasi iniziali di design.

La prossima sezione analizzerà in dettaglio le motivazioni a favore di questa scelta.

1.3.1 Metodi Formali e Design

“Many observers believe that formal policy models have their maximum benefit in removing inconsistencies, ambiguities, and contradictions in the natural language policy statement. The process of formalizing the policy aids in clarifying the policy. This process then has the secondary benefit of making a clearer statement of policy to the implementers.” [AbrZel]

L'utilizzo di metodi formali comporta, indipendentemente dal momento di adozione all'interno del processo di design, due sostanziali vantaggi: l'utilizzo di uno strumento privo delle ambiguità tipiche del linguaggio naturale ed una maggior chiarezza e completezza di descrizione dell'obiettivo.

L'impatto sul design, al contrario, dipende pesantemente dal momento di adozione scelto. A tal proposito si riscontrano due diverse posizioni: sole fasi finali (Late-Lifecycle) e sole fasi iniziali (Early-Lifecycle):

“However, formal methods are relatively expensive to apply at present and must compete on cost and effectiveness with other methods for quality control and assurance, so some selectivity in their application is inevitable in most projects. The crudest form of selection amounts to a dichotomy: we can prefer to apply formal methods late in the lifecycle, or early. Both positions have strong advocates. Late-lifecycle advocates observe that it is the running program code (or gate layout in the case of hardware) that determines the behaviour of the system; unless the code (or gate layout) has been verified, formal methods haven't done anything that is real. It is easy to see the flaw in this argument: what good does it do to verify the code against its detailed specification if that specification could be wrong? [...] Early-lifecycle advocates take this argument to its conclusion and claim that the early lifecycle is the source of the most dangerous and expensive faults, and that the later stages of the lifecycle are adequately served by conventional methods.” [Rus93]

Rushby difende convintamente la scelta Early-Lifecycle, sostenendo che per le fasi finali le tecniche tradizionali (e.g.: validazione attraverso simulazione) sono uno strumento adeguato. Presenta, inoltre, una serie di benefici per le fasi iniziali, articolati in una precisa serie di punti.

1) Comunicazione

La comunicazione tra tecnici provenienti da discipline diverse è molto più intensa durante le fasi iniziali. Discipline diverse adottano, trattando gli stessi problemi, modelli mentali e linguaggi differenti. I metodi formali possono fornire uno strumento di comunicazione non ambigua e di unificazione delle diverse prospettive settoriali.

2) Astrazione

I metodi formali incoraggiano l'utilizzo di un linguaggio basato su insiemi e quantificatori. In tal modo si può, senza perdita di precisione, astrarre dalla specifica rappresentazione degli oggetti e descriverne le proprietà senza ricorrere a cicli e procedure di ricerca.

3) Completezza

I metodi formali, diversamente dalla maggior parte dei linguaggi di programmazione, richiedono una trattazione sistematica di tutti i possibili casi da elaborare ed una precisa istanziazione delle entità coinvolte.

4) Correttezza

I metodi formali consentono l'utilizzo di tipi dipendenti, un importantissimo aiuto nell'evitare banali errori di distrazione o di sintassi. Il semplice controllo di tipo, nel caso di tipi sufficientemente stretti, rileva in anticipo moltissimi errori destinati, contrariamente, a propagarsi fino alla fase di debug.

5) Semplicità

È molto più semplice validare un algoritmo finché la sua descrizione mantiene un elevato grado di astrazione, piuttosto che condurre una complessa analisi sulla sua implementazione. La successiva fase di implementazione deve comunque fare uso di strumenti che forzino il codice o l'hardware a non violare le specifiche di design.

6) Risparmio

L'applicazione anticipata dei metodi formali, a fronte di un apparente aumento di spesa, garantisce invece un sensibile risparmio per almeno quattro motivi:

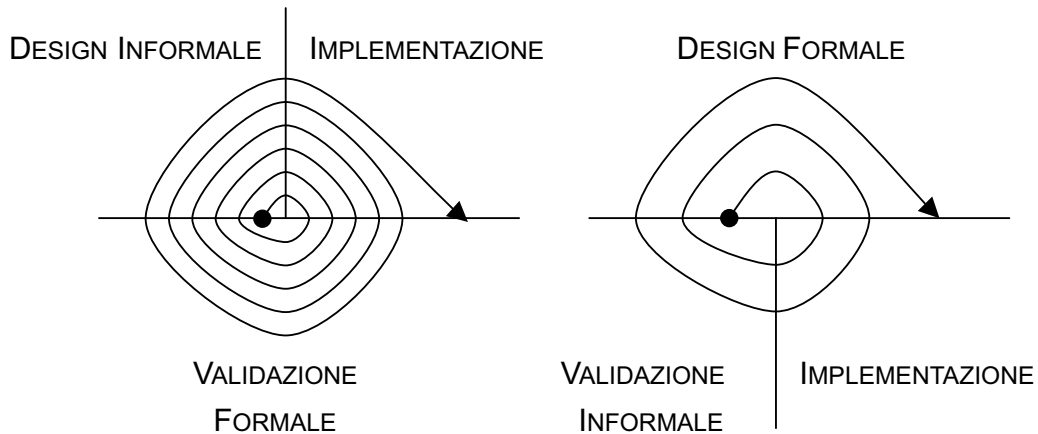


Figura 1.3.1: Iterazione tra design, implementazione e validazione

- le specifiche prodotte durante le fasi iniziali sono molto più concise e la loro modellizzazione richiede un ridotto sforzo formalizzativo;
- ogni errore rilevato in questa fase potrà essere corretto in anticipo, riducendo il numero delle iterazioni tra design, implementazione e validazione (Figura 1.3.1);
- la difficoltà di formalizzare delle soluzioni azzardate o particolarmente complicate è una forte spinta all'adozione di soluzioni modulari o quantomeno semplici;
- la possibilità di verificare in anticipo proprietà anche complesse del modello può rendere meno rischiosa l'adozione di soluzioni innovative, altrimenti viste come incognite non accettabili.

7) Trasparenza

L'ultimo punto, forse il più importante dal punto di vista degli utenti, riguarda la trasparenza. La specifica formale prodotta durante le prime fasi può essere vista anche, previa traduzione, come un'ottima specifica pubblica che accompagni il prodotto al momento della commercializzazione. Il produttore, infatti, non svelerebbe nessun dettaglio implementativo ai concorrenti e fornirebbe agli utenti, dal punto di vista operativo, un manuale d'utilizzo preciso e completo:

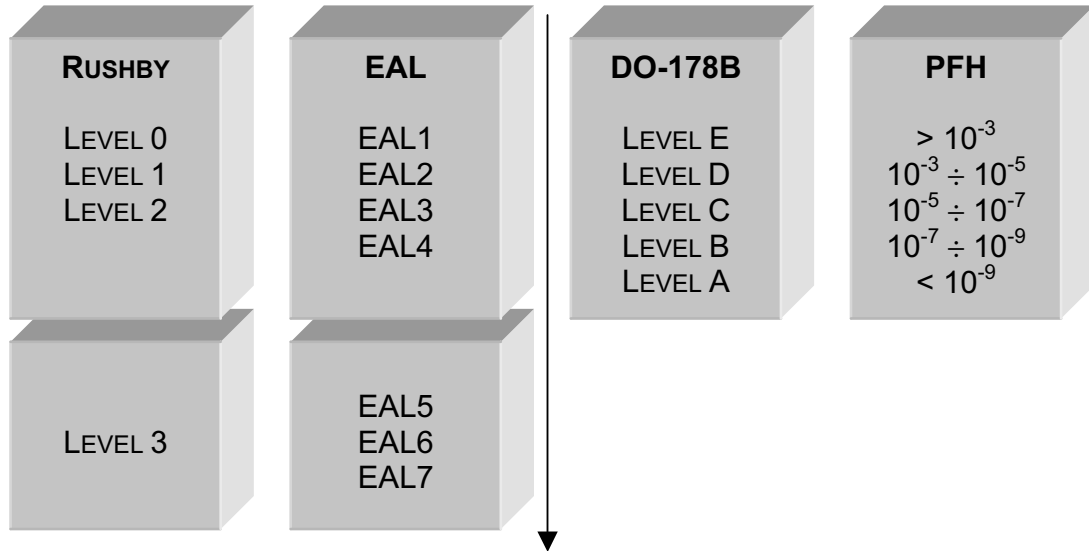
“Moreover, simulators are kept secret because (since they are actually implementations) they contain implementation details. Any company would be foolish to reveal a solid piece of its design to its competitors. A specification structured as a set of assertions lacks these disadvantages. It lacks implementation details. Thus there is no need to conceal it from competitors. It can be structured so as to specify only intended behaviour, and leave unintended behaviour unspecified. Thus there is no need to conceal it from customers.” [Bea93]

1.3.2 Requisiti di Sicurezza e Standard

“Computer systems are increasingly taking on roles where their failure could have catastrophic results, for example in medical care and in the control systems of aircraft and nuclear power stations. How can such systems be known to be safe? Certainly, they can be tested. But for a system of any complexity, the number of all possible combinations of external inputs and internal states is too large for even the most highly automated testing to be comprehensive.” [Mac91]

Prima di entrare nel vivo di quest’ultimo paragrafo è necessario formulare un breve riepilogo dei principali punti finora emersi riguardo alla formalizzazione:

- la normale manutenzione non può prevenire i difetti logici dei controlli software e hardware dei sistemi critici (vedi Therac-25);
- la simulazione non può esplorare tutte le possibili combinazioni di input e stati interni di un sistema anche se di ridotta complessità (vedi MpSoC);
- la ridondanza, pratica comune nell’avionica, non offre garanzie certe nel caso dei sistemi critici: i sistemi ausiliari possono soffrire tutti dello stesso difetto o essere tutti sottoposti alla medesima sequenza di input fatale (vedi Airbus A320) [Bro07];
- quegli errori che, in un sistema a bassa criticità, possono essere tollerati, non possono in nessun caso essere tollerati in un sistema a criticità massima: mentre un aereo precipita non ci sarà mai tempo di attendere un eventuale reset del sistema malfunzionante (vedi Boeing 767);



Scala progressiva di misura del grado di confidenza nel sistema

Figura 1.3.2: Confronto delle metriche [Bro07, Rus93, Sub03, Wik07, Wik08]

- la correttezza del software risente superiormente del gap fra intenzioni del designer e specifiche di alto livello (vedi sezione 1.2);
- la correttezza dell'hardware risente inferiormente del gap fra gate-level e livello fisico (vedi sezione 1.2);
- la correttezza, a causa dei limiti superiore ed inferiore (componente umana e fisica), non è un problema ben posto, ma solo un percorso di successivi raffinamenti di certificazione [BoyYu92].

Gli standard di sicurezza sono l'ineludibile risposta al problema pratico di garantire la nostra sicurezza, e ogni standard si basa su delle metriche che, in questo caso, si occupano di misurare il grado di confidenza che possiamo riporre in un sistema.

Diverse metriche di sicurezza rilevano diversi aspetti dello stesso problema e la trattazione della sicurezza e degli standard si conclude con una breve descrizione e confronto (Figura 1.3.2) della metrica PFH, del DO-178B, dell'EAL e della scala di automazione della formalizzazione proposta da Rushby.

1) Probability of Failure per Hour (PFH)

Il termine “fallimento” indica la proprietà di un sistema di non svolgere più il servizio richiesto, indipendentemente dalla durata dell’interruzione di servizio.

Un tipico valore è 10^{-7} PFH che l’industria avionica potrebbe interpretare nel seguente modo: 10^7 ore di servizio ininterrotto distribuite, per esempio, in una flotta di 100 veicoli che volino 3.000 ore all’anno per un periodo totale di 33 anni [Rus93].

2) DO-178B

È un documento, sviluppato in campo avionico, che descrive un insieme di direttive e tecniche finalizzate alla produzione di componenti con uno specifico grado di affidabilità. I target di rischio formano una scala graduata:

<i>Level A - Catastrophic</i>	<i>Failure may cause a crash</i>
<i>Level B - Hazardous</i>	<i>Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the plane due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers</i>
<i>Level C - Major</i>	<i>Failure is significant, but has a lesser impact than a hazardous failure (for example, leads to passenger discomfort rather than injuries)</i>
<i>Level D - Minor</i>	<i>Failure is noticeable, but has a lesser impact than a major failure (for example, causing passenger inconvenience or a routine flight plan change)</i>
<i>Level E - No Effect</i>	<i>Failure has no impact on safety, aircraft operation, or crew” [Wik08]</i>

Il documento tratta principalmente di una combinazione di pratica consolidata e stretta collaborazione con enti del campo addetti alla certificazione e, pur ricorrendo ad alcune pratiche derivate dal campo dei metodi formali, non menziona mai esplicitamente la formalizzazione.

3) Evaluation Assurance Level (EAL)

È una scala graduata non ristretta al campo avionico, che in base al soddisfacimento di precisi requisiti (Common Security Criteria), assegna un livello di testabilità. Da notare la testabilità (tipo e modalità dei test superati) non implica il conseguimento automatico di un determinato grado di sicurezza.

<i>EAL7</i>	<i>Formally Verified Design and Tested</i>
<i>EAL6</i>	<i>Semi Formally Verified Design and Tested</i>
<i>EAL5</i>	<i>Semi Formally Design and Tested</i>
<i>EAL4</i>	<i>Methodically Designed, Tested and Reviewed</i>
<i>EAL3</i>	<i>Methodically Tested and Checked</i>
<i>EAL2</i>	<i>Structurally Tested</i>
<i>EAL1</i>	<i>Functionally Tested” [Wik07]</i>

I primi tre livelli (7-5) menzionano esplicitamente la formalizzazione e, a causa di questo preciso requisito, pochissimi dispositivi hardware hanno raggiunto un livello superiore al quarto (e.g.: Java Card Machine certificata EAL7).

Per lo stesso motivo, nonostante numerosi sistemi operativi siano già stati certificati DO-178B Level A (e.g.: LynxOS-178), nessuno di loro è finora riuscito a conseguire la certificazione EAL7. Facendo sempre riferimento all'avionica, bisogna comunque tener presente che alcuni semplici accorgimenti (e.g.: sequenzializzazione dell'esecuzione, isolamento dei processi), applicati all'interno di sistemi operativi già certificati DO-178B Level A, garantiscono un livello di affidabilità molto elevato ai sistemi di controllo in dotazione agli aerei moderni (vedi progetto Joint Unmanned Combat Air System “J-UCAS”).

4) Scala di automazione della formalizzazione proposta da Rushby

L'ultima scala proposta è, nell'ambito di questa tesi sulla formalizzazione interattiva, forse la più interessante: si concentra esclusivamente sull'utilizzo di strumenti di dimostrazione automatica.

- “Level 3 Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking*
- Level 2 Use of formalized specification languages with some mechanized support tool*
- Level 1 Use of concepts and notations from discrete mathematics*
- Level 0 No use of formal methods” [Rus93]*

Il paragrafo seguente, a conclusione del capitolo, descriverà brevemente le principali tipologie di strumenti automatici di dimostrazione utilizzati nell’ambito della formalizzazione hardware e software.

1.4 Formalizzazione Assistita

“Formal verification is the use of mathematical techniques to verify properties of a system description. A particular style of formal verification that has shown considerable promise in recent years is the use of general purpose automated reasoning systems to model systems and prove properties of them. Every such reasoning system requires considerable assistance from the user, which makes it important that the system provide convenient ways for the user to interact with it.” [KauMoo97]

La formalizzazione di software e hardware, introdotta nel capitolo precedente, fa uso intensivo di strumenti di dimostrazione automatizzata appartenenti a due principali famiglie:

- 1) ambienti di dimostrazione fortemente automatizzati basati su logica del primo ordine (First-Order Heavily-Automated Provers);
- 2) ambienti di dimostrazione basati su tattiche e su logica di ordine superiore (Higher-Order Tactic-Based Provers).

Kaufmann e Moore, gli ideatori di ACL2, introducono il loro ambiente di dimostrazione automatizzata con un’affermazione apparentemente contraddittoria:

ogni sistema di ragionamento automatizzato richiede una considerevole assistenza ed interazione da parte dell'utente. Le successive sezioni, contestualmente alla breve descrizione delle due famiglie di strumenti, chiariranno questo punto.

1.4.1 First-Order Heavily-Automated Provers

“The theorem prover is fully automatic in the sense that once a proof attempt has started, the system accepts no advice or directives from the user. The only way the user can interfere with the system is to abort the proof attempt. However, on the other hand, the theorem prover is interactive: the system may gain more proving power through its data base of lemmas, which have already been formulated by the user and proved by the system. [...] Typically, the checking of difficult theorems by Nqthm requires extensive user interaction. [...] The user first formalizes the problem to be solved in the logic. [...] The user then leads the theorem prover to a proof of the goal theorem by proving lemmas that, once proved, control the search for additional proofs. Typically, the user first discovers a hand proof, identifies the key steps in the proof, formulates them as a sequence of lemmas, and gets each checked by the prover.” [BoyYu96]

A questa famiglia di ambienti di dimostrazione fortemente automatizzati appartengono, per esempio, Nqthm e ACL2 (A Computational Logic for Applicative Common Lisp), accomunati dalle seguenti caratteristiche:

- si basano su logica del primo ordine priva del quantificatore universale ed esistenziale;
- durante la ricerca della dimostrazione non interagiscono direttamente con l'utente e lavorano in modo totalmente automatizzato;
- hanno bisogno, durante la ricerca della dimostrazione, di accedere ad un database di lemmi creati ad hoc dall'utente;
- non producono una registrazione della dimostrazione;
- consentono l'estrazione sotto forma di codice LISP.

Questi strumenti sono stati usati in numerose formalizzazioni di microprocessori, fra cui:

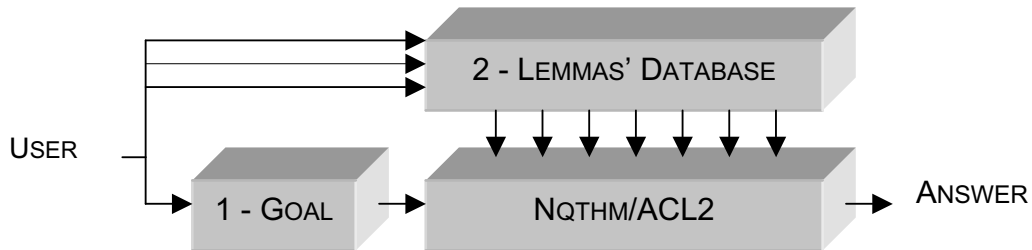


Figura 1.4.1: Interazione dell'utente con Nqthm/ACL2

- Nqthm → FM9001 [HunBro92];
- Nqthm → Motorola MC68020 [BoyYu92];
- ACL2 → Motorola Complex-Arithmetic-Processor (CAP) [BKM96];
- ACL2 → AMD K86 Floating-Point Kernel [BKM96].

L'interazione dell'utente col sistema, come descritto da Boyer e Yu, si compone di due fasi distinte: una di formalizzazione del problema nella logica del sistema ed una di euristica, alla ricerca di un'opportuna sequenza e formulazione dei lemmi da inserire nel database per riuscire a dimostrare il goal (Figura 1.4.1).

La seconda fase di euristica è la più delicata in quanto:

- il sistema affronta in modo totalmente automatizzato la ricerca dell'esistenza di almeno una dimostrazione del goal fornito dall'utente;
- la ricerca, seppur automatizzata, può fallire nel produrre un risultato entro un tempo accettabile.

Un'opportuna formulazione e sequenza di lemmi intermedi, quindi, fornisce al sistema una traccia che gli consente di produrre un risultato in tempo accettabile, senza esplorare tutti i possibili rami di dimostrazione. Tale procedura non può essere automatizzata e rientra sempre nel campo dell'euristica, richiedendo all'utente una profonda conoscenza del sistema (dettagli implementativi compresi) ed un'elevata manualità nel campo dimostrativo, ottenibile unicamente attraverso una prolungata esperienza.

L'elevata competenza dimostrativa richiesta, unita all'impossibilità di automazione del processo, è - dettaglio non trascurabile - anche un forte ostacolo all'adozione di questo tipo di strumenti da parte dell'industria:

“Why then is industry apparently so reluctant to adopt these techniques? Part of the answer is that the skills required to use our tools are different than those ordinarily found in a hardware design team. [...] However, another major reason our tools are not more widely used is that they are not integrated into the design process.” [BKM96]

L’impiego di questi sistemi comporta, nonostante tutto, anche dei vantaggi, il più importante dei quali deriva proprio dall’elevato grado di automazione: una volta completato con successo il database dei lemmi, la dimostrazione risulta resistente a modifiche marginali. Ogni microprocessore commerciale non è mai un’entità isolata ma fa parte di una famiglia di dispositivi che condividono gran parte della circuiteria di base:

“A considerable amount of strategic planning is required to co-opt the Nqthm and ACL2 proof heuristics to prove interesting theorems. However, the style of proof of these efforts has an important benefit: proof robustness. [...] For example, when the verified processor FM8501 was redesigned to increase its word size the Nqthm proof of the modified processor correctness theorem worked with minimal human assistance.” [Wil97]

1.4.2 Higher-Order Tactic-Based Provers

“Coq proofs are developed interactively using a number of tactics as elementary proof steps. The sequence of tactics used constitutes the proof script. Building such scripts is surprisingly addictive, in a videogame kind of way, but reading and reusing them when specifications change is difficult.” [Ler06]

A questa famiglia di ambienti di dimostrazione basati su tattiche appartengono, per esempio, HOL (Higher Order Logic), PVS (Prototype Verification System), Coq e Matita, accomunati dalle seguenti caratteristiche:

- si basano su logica di ordine superiore;
- durante la ricerca della dimostrazione interagiscono direttamente con l’utente che sceglie, volta per volta, quali tattiche applicare.

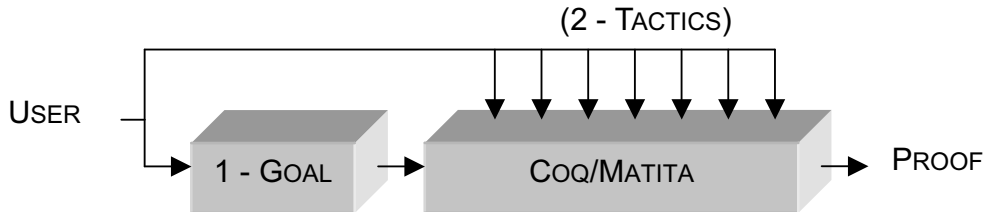


Figura 1.4.2a: Interazione dell'utente con Coq/Matita

Questi strumenti sono stati usati in numerose formalizzazioni di microprocessori, fra cui:

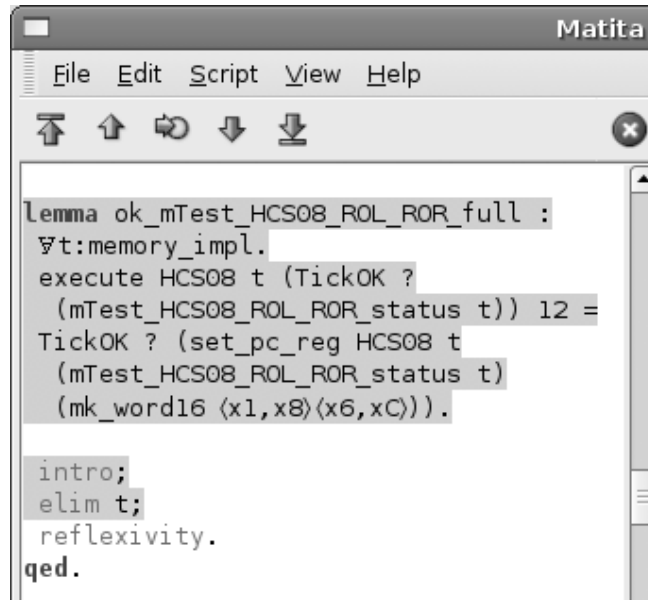
- PVS → Rockwell AAMP5 [Wil97];
- HOL → VIPER [Mac91];
- HOL → ARM3 [Fox01];
- HOL → ARM6 [Fox02];
- HOL, Coq → ATM network chip [JCC97].

L'interazione dell'utente col sistema durante la fase di dimostrazione è sempre diretta e l'insieme delle tattiche applicate, riunite in uno script, costituisce la prova (Figura 1.4.2a). L'automazione del sistema, quindi, si sposta dalla ricerca della dimostrazione alla verifica della corretta applicazione delle tattiche e alla chiusura di tutti i rami di dimostrazione.

Matita, lo strumento utilizzato in questa tesi, presenta insieme a Coq quattro ulteriori caratteristiche: interattività, riduzione, estrazione di codice e tipi dipendenti. Esse hanno contribuito profondamente a semplificare lo sforzo richiesto dalla formalizzazione e meritano una speciale sottolineatura, prima di affrontare, nel prossimo capitolo, i dettagli implementativi.

1) Interattività

Oltre alla generale capacità di eseguire uno script di dimostrazione, l'ambiente interattivo consente di visualizzare, passo per passo, il risultato dell'esecuzione di ogni singola tattica e di controllare lo stato di ogni ramo di dimostrazione ancora attivo. Viene così semplificato moltissimo il compito dell'utente che, come asserisce Leroy, può sperimentare ed imparare con lo stesso approccio immediato di un videogioco (Figure 1.4.2b e 1.4.2c). Questo stile intuitivo, unito alla capacità



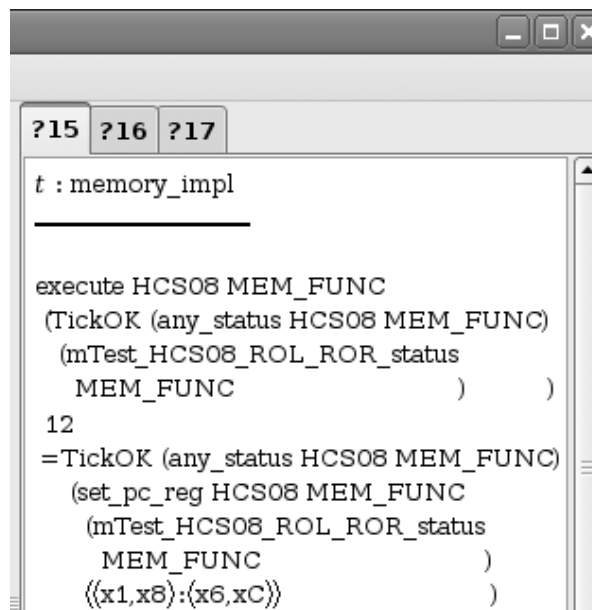
```

Matita
File Edit Script View Help
[Navigation icons]
Lemma ok_mTest_HCS08_ROL_ROR_full :
  ∀t:memory_impl.
  execute HCS08 t (TickOK ?
    (mTest_HCS08_ROL_ROR_status t)) 12 =
  TickOK ? (set_pc_reg HCS08 t
    (mTest_HCS08_ROL_ROR_status t)
    (mk_word16 ⟨x1,x8⟩⟨x6,xC⟩)).

intro;
elim t;
reflexivity.
qed.

```

Figura 1.4.2b: Esecuzione passo per passo in Matita



```

?15 ?16 ?17
t : memory_impl
-----
execute HCS08 MEM_FUNC
(TickOK (any_status HCS08 MEM_FUNC)
 (mTest_HCS08_ROL_ROR_status
  MEM_FUNC          ) )
12
=TickOK (any_status HCS08 MEM_FUNC)
(set_pc_reg HCS08 MEM_FUNC
 (mTest_HCS08_ROL_ROR_status
  MEM_FUNC          )
 ⟨⟨x1,x8⟩:⟨x6,xC⟩⟩ )

```

Figura 1.4.2c: Visualizzazione di uno dei rami di dimostrazione aperti in Matita

di riduzione del sistema, ha consentito di portare a termine molte semplici dimostrazioni e verifiche di proprietà basate sullo svolgimento per casi e sull'applicazione, ad ogni ramo di dimostrazione, della sola riflessività.

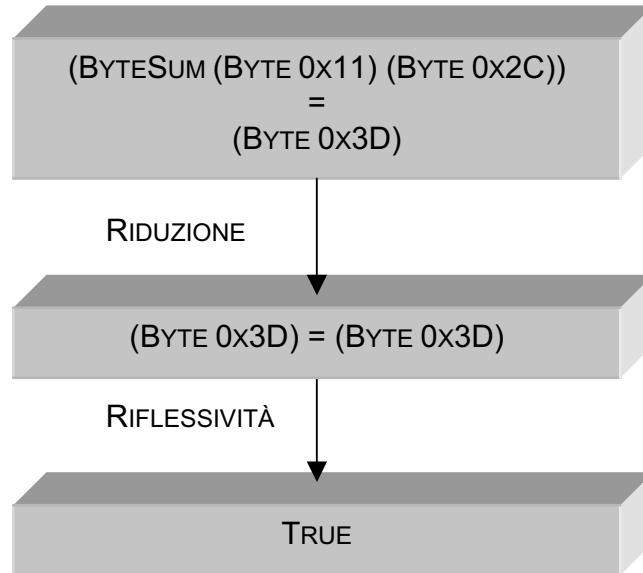


Figura 1.4.2d: Esempio di dimostrazione per riflessività in Matita

2) Riduzione

Il sistema può effettuare al suo interno delle riduzioni che, nel caso di domini finiti e sufficientemente piccoli (e.g.: byte, modalità di indirizzamento, pseudocodici, opcode), consentono di ricondurre le dimostrazioni a semplici applicazioni di riflessività (Figura 1.4.2d). Nel caso di domini più grandi (e.g.: word a 16/32 bit, liste di byte) questa tecnica può eccedere la capacità di memorizzazione del sistema e si può ricorrere all'estrazione di codice. Il lemma si trasforma così in un predicato espresso in un linguaggio di programmazione e, tramite compilazione ed esecuzione, se ne può verificare la correttezza.

3) Estrazione di codice

Il sistema utilizza un linguaggio funzionale molto simile al lambda calcolo tipato e permette di esportare quasi automaticamente le sue definizioni in linguaggio OCAML. L'estrazione di codice ha reso possibile:

- eseguire semplici verifiche per casi che eccedono la capacità di memorizzazione del sistema (vedi i test enunciati in appendice);

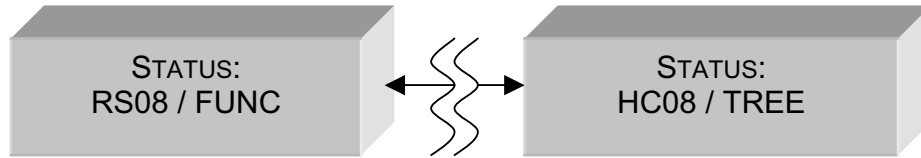


Figura 1.4.2e: Esempio di tipi dipendenti incompatibili in Matita

- ottenere, alla fine del lavoro di formalizzazione sui microcontroller Freescale, una specifica eseguibile che consente di compilare ed eseguire il relativo codice macchina in modo simile ad un simulatore tradizionale.

La traduzione può richiedere, comunque, un intervento manuale per eliminare lo “useless-code” non individuato automaticamente [BCDG00]:

“Automatic extraction of Caml code from Coq specifications is another feature of the Coq environment on which we depended crucially and which turned out to work satisfactorily. The major difficulty for extraction is dealing with specifications that use dependent types intensively: the resulting Caml code is not necessarily typeable in Caml, and it can contain large, costly proof computations that do not contribute to the final result and are eliminated by techniques akin to slicing.” [Ler06]

4) Tipi dipendenti

La specifica eseguibile prodotta ha assunto la forma di una macchina virtuale parametrizzata, in grado di eseguire codice specifico di tutti i diversi modelli di microcontroller a 8 bit Freescale. I parametri operativi (e.g.: CPU, modello di chip, implementazione della memoria) sono stati implementati come tipi dipendenti. Ad ogni entità parametrizzabile, cioè, vengono indissolubilmente associati, al momento dell’istanziamento, dei valori che contribuiscono a determinare il tipo stesso dell’entità.

Uno stato interno, per esempio, una volta istanziato come CPU di tipo RS08 ed implementazione della memoria *FUNC* risulterà compatibile solo con altri stati istanziati coi medesimi parametri (Figura 1.4.2e):

“Dependent types are one of the main features provided by Coq but not HOL. These are types that can be dependent on a term. They are used throughout the Coq specifications; for example, to represent words, communications ports and lists with associated lengths. [...] The dependent types provided by the calculus of construction enforce great precision in writing specifications. The dependent lists type, for example, is very suitable for describing multiple inputs or outputs of a device. [...] The length of a dependent list is part of its type. Thus, when handling several kinds of lists with different lengths, certain inconsistencies related to these lengths can be detected at type checking time. [...] Dependent types can only be simulated in HOL. This is done by giving type restrictions as predicate assumptions. [...] Whilst this superficially looks like a dependent type, the type checker does not recognize the type as being dependent: a list of length 8 has the same type as a list of length 16. Type checking cannot pick up dependent type errors. This only occurs much later when proof is performed as part of the verification process. With real dependent types a term just cannot be created unless such errors have been removed: the proofs must be given as part of the type checking process.” [JCC97]

Una misura concreta di quanto l’interattività, la riduzione, l’estrazione di codice ed infine i tipi dipendenti abbiano facilitato questo lavoro di formalizzazione si può basare anche sul solo confronto della dimensione in linee di codice della formalizzazione e del tempo richiesto per la sua produzione.

La specifica eseguibile completa, considerando la sola parte di definizione, consta di circa 9.300 righe prodotte in meno di due mesi, seguiti da meno di una settimana di debug complessivo.

Capitolo 2

Formalizzazione in Matita dei Microcontroller Freescale

2.1 I Microcontroller a 8 bit Freescale

“A microcontroller can be defined as a complete computer system including a CPU, memory, a clock oscillator, and I/O on a single integrated circuit chip. When some of these elements such as the I/O or memory are missing, the integrated circuit would be called a microprocessor. [...] The smallest microcontrollers are used for such things as converting the movements of a computer mouse into serial data for a personal computer. Very often microcontrollers are embedded into a product and the user of the product may not even know there is a computer inside.” [Fre07]

La definizione citata si trova all’inizio di molti dei manuali tecnici relativi ai microcontroller a 8 bit Freescale e ne sintetizza efficacemente le caratteristiche generali:

- sono veri e propri computer completi dotati di CPU e memoria autonome;
- includono uno o più dispositivi di I/O per comunicare con altre periferiche;
- sono talmente piccoli da poter essere inseriti, in modo quasi invisibile, all’interno di altri prodotti.

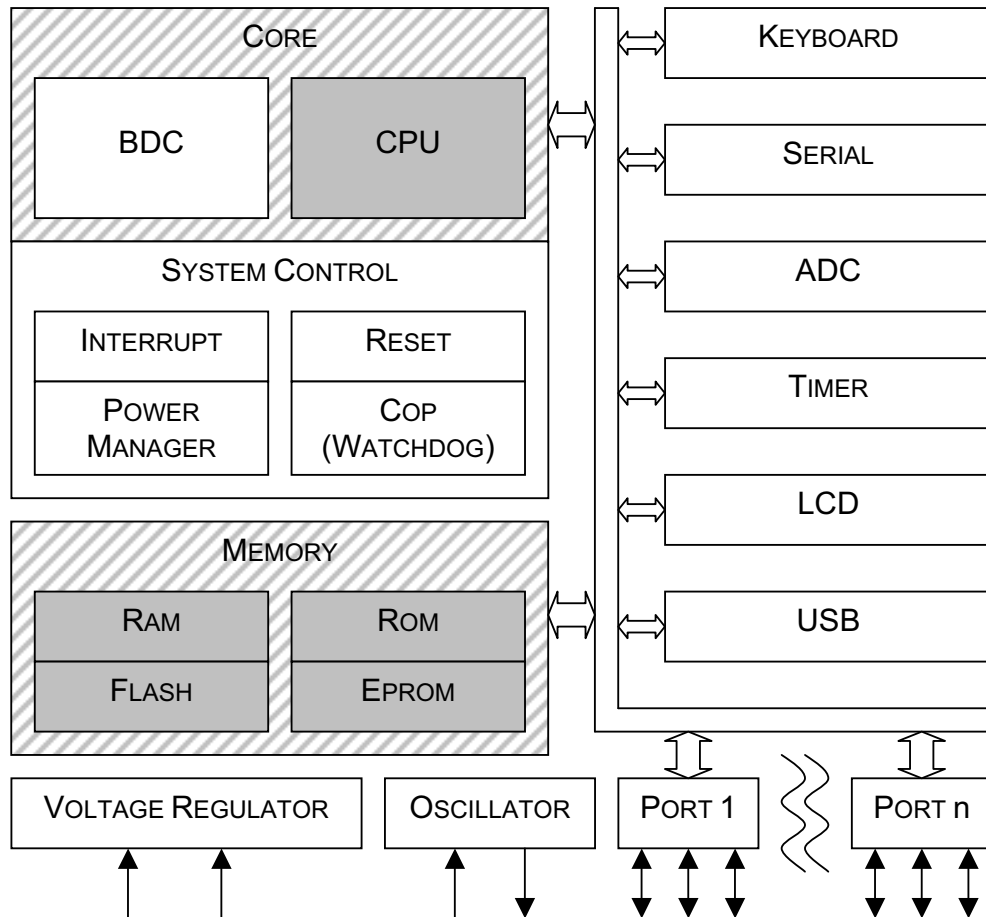


Figura 2.1a: Schema di un microcontroller [Fre07]

La loro struttura interna (Figura 2.1a) è abbastanza articolata e comprende, considerando le diverse dotazioni di tutti i modelli, le seguenti componenti (evidenziate in grigio quelle sottoposte a formalizzazione):

- la CPU e, solo per le CPU HCS08 e RS08, il Background Debug Controller (BDC) che costituiscono il nucleo del microcontroller;
- il controllo di esecuzione del sistema, che gestisce gli interrupt, le diverse modalità di consumo e il reset automatico tramite countdown. Se quest'ultimo controllo è abilitato, il programma in esecuzione è tenuto a segnalare periodicamente, pena reset del sistema, di non essere entrato in stallo (e.g.: condizione di loop infinito, routine di gestione dell'errore che fallisce e richiama se stessa). Il Computer Operating Properly Watchdog

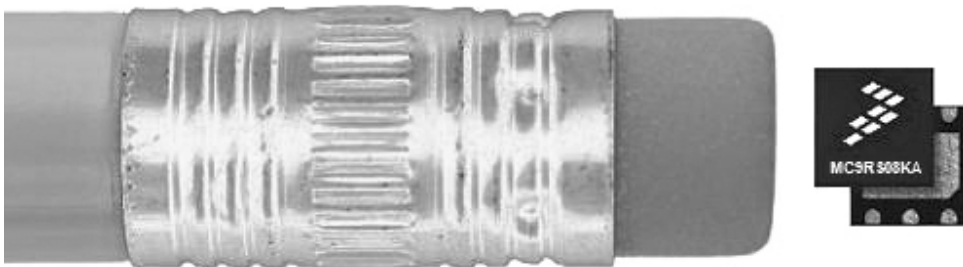


Figura 2.1b: Microcontroller MC9RS08KA ingrandito del 400% [Fre07]

(COP), similmente, effettua un reset automatico nel caso che il voltaggio non rientri entro i limiti di tolleranza o che la CPU effettui operazioni non consentite (e.g.: tentativo di eseguire un opcode non definito, accesso ad una locazione di memoria non esistente);

- vari tipi di memoria, RAM (da pochi byte fino ad un massimo di qualche Kb) e ROM/EPROM/FLASH (sempre compresa entro i 64Kb);
- uno o più dispositivi di I/O di tipo tastiera, seriale, ADC, multichannel timer, display LCD, USB, ecc...;
- un regolatore della tensione in ingresso, a seconda delle diverse modalità di consumo;
- un oscillatore per adattare dinamicamente il clock interno e quello in uscita verso altri dispositivi;
- una o più porte di comunicazione connesse ai piedini esterni.

Altre importanti caratteristiche hanno ulteriormente contribuito ad una significativa diffusione di questi microcontroller:

- sono molto piccoli, fino a meno di 5mm di lato per il modello MC9RS08KA (Figura 2.1b);
- richiedono un ridotto consumo energetico, compreso tra 10mW (a pieno regime) e 10 μ W (in fase di standby);
- operano correttamente all'interno di un range di temperatura molto esteso, compreso tra -40°C e +125°C;
- hanno un costo ridotto, compreso tra 0,43\$ (MC9RS08KA) e 9\$.

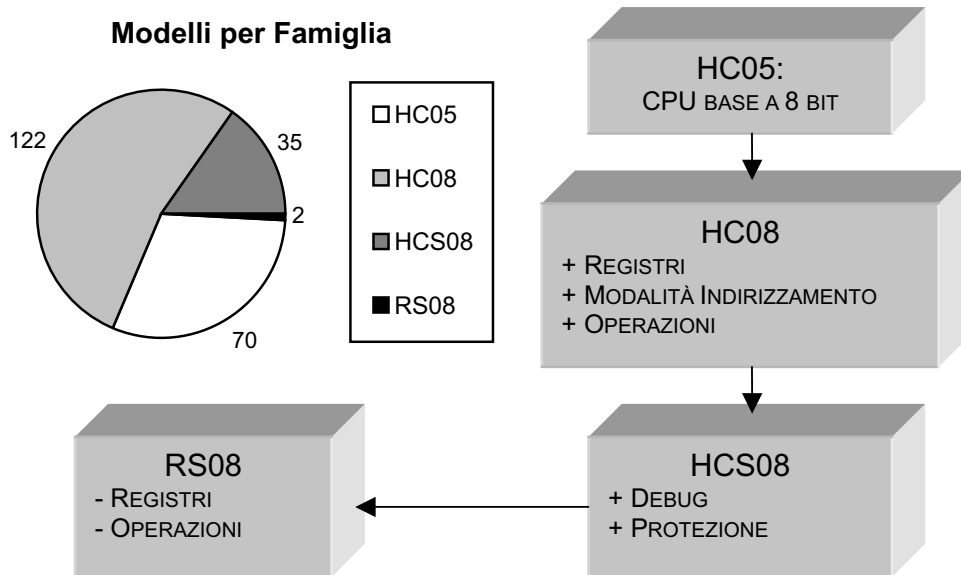


Figura 2.1c: Famiglie di CPU e grafico modelli per famiglia

I vari microcontroller a 8 bit Freescale possono essere raggruppati in quattro grandi famiglie a seconda della CPU: HC05, HC08, HCS08 e RS08. All'interno di ogni famiglia si distinguono a loro volta diversi modelli (229 in totale), a seconda delle diverse dotazioni di memoria, dispositivi di I/O, frequenze di clock, porte di comunicazione e piedinatura esterna (Figura 2.1c).

1) HC05

Sono i modelli di base, verso cui le altre famiglie (RS08 esclusi) mantengono una totale retrocompatibilità a livello di CPU: registri ed ISA.

Vengono identificati tramite sigle dal formato MC68HC(7)05xxxx, dove xxxx contraddistingue il modello specifico e 7 la sostituzione della self-check ROM con uno specifico firmware di bootstrap.

2) HC08

È una famiglia molto numerosa (oltre 100 modelli) che equipaggia una CPU

arricchita da nuovi registri, nuove istruzioni e nuove modalità di indirizzamento.

Vengono identificati tramite sigle dal formato MC68HC(9)08xxxx, dove xxxx contraddistingue il modello specifico e 9 la presenza di memoria FLASH al posto della più economica EPROM.

3) HCS08

La CPU HCS08 si distingue dalla HC08 per un'unica caratteristica: oltre alle tre tradizionali modalità operative (run, wait, stop) se ne aggiunge una di debug (background). Questa possibilità di esecuzione passo per passo pilotando direttamente dall'esterno la CPU, evitando quindi le tradizionali piattaforme di emulazione, si chiama in-circuit debug.

L'in-circuit debug, in presenza della memoria FLASH montata su tutti i modelli, pone un serio problema di sicurezza. Cosa può impedire ai produttori concorrenti di clonare il software di controllo attraverso gli strumenti di debug? Per questo motivo viene introdotto anche un controllo di sicurezza nell'accesso alla memoria.

Un apposito bit di protezione, se attivato, blocca tutti i tentativi di lettura e scrittura verso le aree critiche (RAM e FLASH) eseguiti da codice che non risieda a sua volta in un'area critica. Questo bit di protezione, a completare il meccanismo, si trova alla fine della memoria FLASH. La protezione quindi, una volta attivata, può essere disabilitata solo sovrascrivendo per intero il codice contenuto della memoria FLASH, vanificando così ogni tentativo di copia non autorizzata.

Vengono identificati tramite sigle dal formato MC9S08xxxx, dove xxxx contraddistingue il modello specifico.

4) RS08

Questa famiglia comprende due soli modelli di tipo ultra-low-end. Sono cioè realizzati col preciso obiettivo di minimizzarne il costo semplificando al massimo l'hardware (e.g.: registri, operazioni ISA). La CPU RS08, derivata dalla HCS08 di

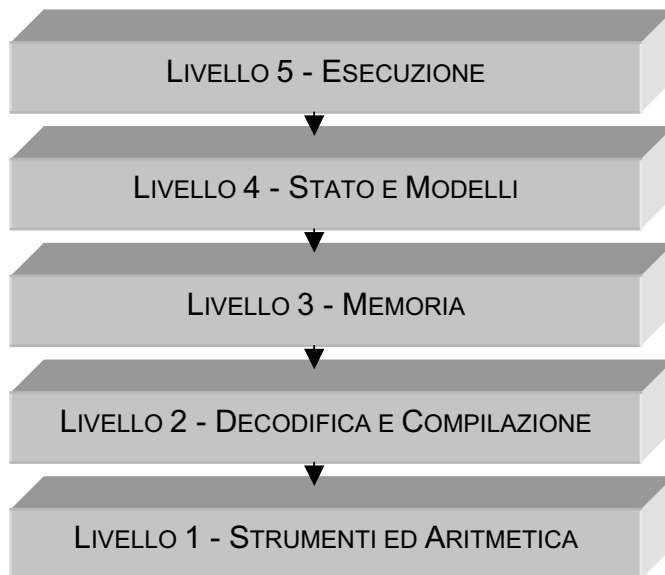


Figura 2.1.1: Livelli di formalizzazione realizzati

cui mantiene tutte le caratteristiche di debug e sicurezza, adotta una complessa serie di soluzioni ibride che ne massimizzano l'efficienza a discapito della retrocompatibilità con le altre famiglie.

Vengono identificati tramite sigle dal formato MC9RS08xxxx, dove xxxx contraddistingue il modello specifico.

Termina qui la breve panoramica sulle caratteristiche generali dei microcontroller a 8 bit Freescale e si può, prima di scendere nei dettagli della formalizzazione, introdurre una schema generale.

2.1.1 Struttura della Formalizzazione

Il lavoro di tesi realizzato può essere suddiviso in cinque livelli disposti secondo una precisa gerarchia che rispecchia anche l'ordine di sviluppo e di compilazione delle definizioni contenute in ogni modulo (Figura 2.1.1): strumenti ed aritmetica (livello 1), decodifica e compilazione (livello 2), memoria (livello 3), stato e modelli (livello 4) ed esecuzione (livello 5).

Obiettivo: *La formalizzazione realizzata ha come obiettivo il produrre una specifica eseguibile di tutti i microcontroller Freescale basati su CPU a 8 bit (HC05, HC08, HCS08 e RS08), sotto forma di una macchina virtuale parametrizzabile a seconda del modello, astraendo da tutte le componenti hardware tranne la memoria (livelli 3 e 4) e la CPU (livelli 2, 4 e 5).*

2.2 Livello 1

Il primo livello comprende i primi quattro moduli realizzati (*extra*, *exadecim*, *byte8*, *word16*) che servono a introdurre gli strumenti di tipo sintattico (tuple, opzioni e option map) e l'aritmetica, necessari ai livelli successivi. Gli esempi che seguiranno faranno uso della sintassi di Matita, simile al λ -calcolo tipato, per il cui approfondimento si rimanda a [BerCas04 e Mat08].

2.2.1 Tuple

La classica implementazione delle tuple (coppie di coppie, di tipo dipendente, dotate di uno o più livelli di annidamento) non consente un'agevole manipolazione dei dati. È stata preferita un'implementazione ad hoc, sempre tramite tipi dipendenti, creando un tipo distinto per ogni dimensione di tupla.

[Implementazione delle tuple di lunghezza 3]

```
inductive TUPLA3TIPO (T1:Type) (T2:Type) (T3:Type) : Type :=
```

```
TUPLA3CONSTRUTTORE: T1 → T2 → T3 → TUPLA3TIPO T1 T2 T3.
```

```
definition PRIMOGETTER :=
```

```
λT1,T2,T3:Type.λp:TUPLA3TIPO T1 T2 T3.match p with [TUPLA3CONSTRUTTORE x _ _ => x ].
```

```
definition SECONDOGETTER :=
```

```
λT1,T2,T3:Type.λp:TUPLA3TIPO T1 T2 T3.match p with [TUPLA3CONSTRUTTORE _ x _ => x ].
```

```
definition TERZOGETTER :=
```

```
λT1,T2,T3:Type.λp:TUPLA3TIPO T1 T2 T3.match p with [TUPLA3CONSTRUTTORE _ _ x => x ].
```

Questa scelta non comporta invece, a livello di controllo statico di tipo, nessun miglioramento rispetto all'implementazione tradizionale. Per esempio, [3] e [4] sono tipi differenti esattamente allo stesso modo di [1] e [2].

[1] **COPPIA**_{TIPO} T₁ (**COPPIA**_{TIPO} T₂ T₃) [2] **COPPIA**_{TIPO} T₁ (**COPPIA**_{TIPO} T₂ T₄)
 [3] **TUPLA3**_{TIPO} T₁ T₂ T₃ [4] **TUPLA3**_{TIPO} T₁ T₂ T₄

Le tuple, indipendentemente dall'implementazione scelta, sono molto utili per la manipolazione di vettori di dimensione fissa di elementi di tipo eterogeneo. Le liste, infatti, consentono di raggruppare solo elementi di tipo omogeneo e richiedono l'utilizzo di un apposito predicato per controllarne la dimensione (controllo a run-time di tipo).

Esempi classici di utilizzo delle tuple all'interno di una formalizzazione hardware possono essere:

- *<PortaHW, Input fornito, Output restituito>* per incapsulare i parametri e il risultato di un'operazione ;
- *<Pseudocodice, Modalità di indirizzamento, Opcode, Cicli macchina>* per incapsulare tutte le informazioni sull'esecuzione di un'istruzione da parte della CPU.

2.2.2 Option e Option Map

La gestione dell'errore, nelle implementazioni classiche (e.g.: linguaggio C), può essere una pericolosa fonte di ambiguità. Basta considerare il seguente esempio di output di una funzione da interi ad interi:

- [-1, -20] = Codice di Errore 1-20;
- [0, *MAX_UNSIGNED_INT*] = Risultato corretto;
- [-21, -(*MAX_UNSIGNED_INT*+1)] = Risultato indefinito.

Un qualsiasi errore di design, di gestione dell'overflow o di semplice distrazione può far produrre alla funzione, invece del risultato corretto previsto, un valore che sarà interpretato come codice di errore o, peggio ancora, privo di interpretazione (Figura 2.2.2a). Correggere un errore di questo tipo, sempre che si riesca ad individuarlo in tempo, richiede tempi di debug molto lunghi.

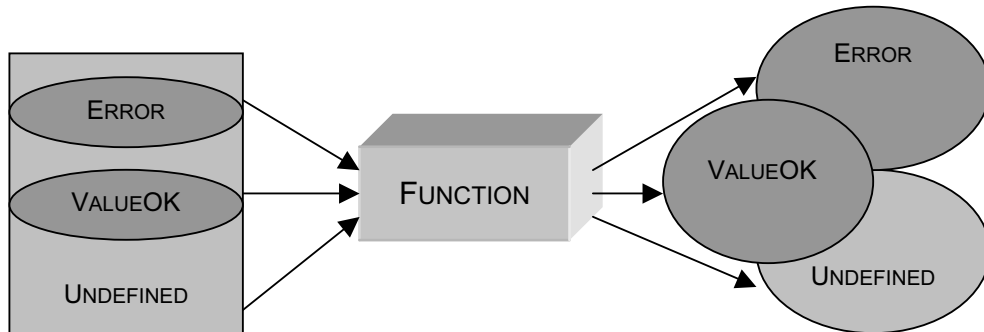


Figura 2.2.2a: Gestione dell'errore C-like

Per rimediare a questa possibile ambiguità bisogna adottare un approccio diverso che garantisca una netta divisione fra l'insieme dei codici di errore e l'insieme degli output possibili. Si è quindi scelto di adottare sistematicamente l'opzione, tecnica che consente, tra l'altro, di segnalare in modo ancora più versatile (Figura 2.2.2b):

- la generica assenza di un risultato corretto (opzione singola, *None*);
- una o più condizioni di errore di classi diverse (opzione multipla, *NoneErrorClassn*).

Un esempio significativo di utilizzo dell'opzione multipla può essere lo stato di una CPU:

- *StatusError* <Stato, Codice di errore> in caso di un errore di esecuzione;
- *StatusSuspended* <Stato, Modalità di sospensione> per segnalare che la CPU è entrata in una modalità di sospensione dell'esecuzione;
- *StatusOK* <Stato> in caso di corretta esecuzione.

[Implementazione dell'opzione singola e multipla]

```
inductive MULTIOPTIONTIPO (T:Type) : Type :=
  | NONEERRORCLASS1COSTRUTTORE: T → ErrorCode → MultiOptionTIPO T
  | NONEERRORCLASS2COSTRUTTORE: T → ErrorCode → MultiOptionTIPO T
  | SOMECOSTRUTTORE: T → MultiOptionTIPO T.
```

```
inductive SINGLEOPTIONTIPO (T:Type) : Type :=
  | NONECOSTRUTTORE: SingleOptionTIPO T
  | SOMECOSTRUTTORE: T → SingleOptionTIPO T.
```

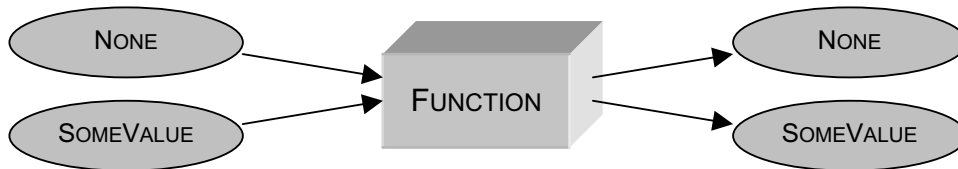


Figura 2.2.2b: Gestione dell'errore con opzione singola

L'uso sistematico dell'opzione comporta, purtroppo, un significativo appesantimento del codice: ogni accesso ad un valore opzionale prevede la gestione obbligatoria di tutte le possibilità. Come rimedio, al fine di ottenere anche una migliore leggibilità del codice, è stata adottata, nel caso dell'opzione singola, la tecnica dell'option map.

[Implementazione dell'option map]

```
definition OPTIONMAP :=
  λT1,T2:Type.λp:SINGLEOPTIONTIPO T1.λf:T1 → SINGLEOPTIONTIPO T2.match p with
  [ NONECOSTRUTTORE => NONECOSTRUTTORE ?
  | SOMECOSTRUTTORE x => (f x) ].
```

Il programmatore, automatizzando la gestione del caso di assenza di risultato (*None*), può scrivere meno codice (sbagliando meno) e concentrarsi unicamente sui casi validi. Basta confrontare, per esempio, due frammenti equivalenti di codice: [5] apre tutti i rami di analisi, mentre [6] fa uso dell'option map.

```
[5] match (...exp...) with
  [ NONECOSTRUTTORE => NONECOSTRUTTORE ?
  | SOMECOSTRUTTORE x1 => match (...exp con x1...) with
  [ NONECOSTRUTTORE => NONECOSTRUTTORE ?
  | SOMECOSTRUTTORE x2 => SOMECOSTRUTTORE ? (...exp con x1 e/o x2...) ]]
```

```
[6] OPTIONMAP ?? (...exp...)
  (λx1.OPTIONMAP ?? (...exp con x1...))
  (λx2.SOMECOSTRUTTORE ? (...exp con x1 e/o x2...))
```

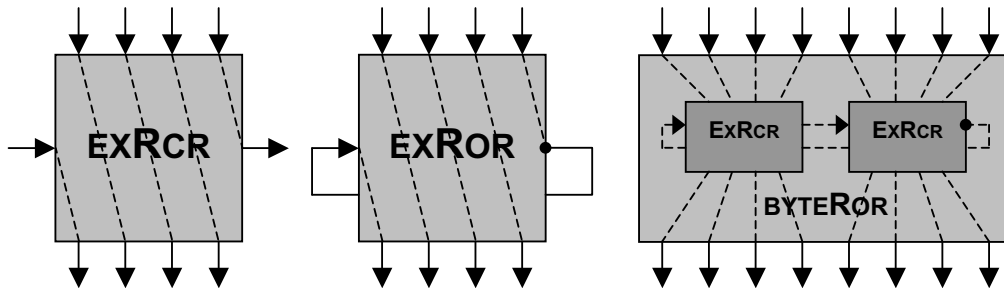


Figura 2.2.3: Descrizione modulare delle funzioni *exRcr*, *exRor* e *byteRor*

2.2.3 Aritmetica

Nell'implementazione dell'aritmetica sono state fatte le seguenti scelte:

- rappresentare i numeri tramite notazione posizionale finita in base esadecimale;
- aggiungere ai normali operatori (e.g.: somma, moltiplicazione) anche tutte le operazioni previste dall'ISA (e.g.: or, complemento, scorrimento, most significant bit, confronto);
- realizzare tre diversi insiemi di operatori (esadecimali, byte e word), a seconda della dimensione in cifre esadecimali, dell'input.

1) Rappresentazione posizionale esadecimale

La notazione posizionale, rispetto alla rappresentazione di Peano (in base uno), è molto più compatta. Il numero 10 per esempio, $S(S(S(S(S(S(S(S(0))))))))$ nella rappresentazione di Peano, si riduce a 0xA in base esadecimale.

Ogni operatore applicato ad una cifra esadecimale può essere considerato come un passo elementare di computazione. Il numero di input possibili, 16 per un operatore unario e 16x16 per un operatore binario, consente un'implementazione a mappa di valori, dove ad ogni input possibile viene associato il corrispondente output precalcolato [7].

Gli operatori applicati ai byte e alle word possono essere ottenuti combinando in modo modulare e non ricorsivo, similmente ai circuiti, gli operatori applicati agli esadecimali. La complessità computazionale può essere resa così lineare nella dimensione, in cifre esadecimali, dell'input [8] (Figura 2.2.3).

[7] definition **EXADECIMALROR** := $\lambda e:\text{Exadecimal.match } e$ with

```
[ 0x0 => 0x0 | 0x1 => 0x8 | 0x2 => 0x1 | 0x3 => 0x9
| 0x4 => 0x2 | 0x5 => 0xA | 0x6 => 0x3 | 0x7 => 0xB
| 0x8 => 0x4 | 0x9 => 0xC | 0xA => 0x5 | 0xB => 0xD
| 0xC => 0x6 | 0xD => 0xE | 0xE => 0x7 | 0xF => 0xF ]
```

[8] definition **BYTEROR** :=

```
 $\lambda b:\text{Byte.match (EXADECIMALROR (HigherDigit } b) \text{ false)}$  with
[ Couple newHighDigit highCarry =>
  match (EXADECIMALROR (LowerDigit  $b$ ) highCarry) with
  [ Couple newLowDigit lowCarry => match lowCarry with
  [ true => MakeByte ((EXADECIMALOR 0x8 newHighDigit) newLowDigit)
  | false => MakeByte (newHighDigit newLowDigit) ]]].
```

2) Operatori ed ISA

Gli operatori applicati alle cifre esadecimali, ai byte ed alle word sono stati integrati con l'ISA delle varie CPU e, nel loro insieme, corrispondono alla dotazione standard delle ALU a 8 e 16 bit (e.g.: Intel 8088):

- Logici (*AND*, *OR*, *XOR*, *NOT*, Complemento a due - *NEG*)
- Rotazione (Shift right - *SHR*, Shift left - *SHL*, Rotate right - *ROR*, Rotate left - *ROL*, Rotate with carry right - *RCR*, Rotate with carry left - *RCL*)
- Confronto (Equal - *EQ*, Less than - *LT*, Less than or equal - *LE*, Greater than - *GT*, Greater than or equal - *GE*)
- Aritmetici (Somma, Moltiplicazione, Divisione)
- Miscellanei (Most significant bit - *MSB*, Decimal adjust addition - *DAA*)

Solo due casi hanno richiesto, a causa delle specifiche incomplete fornite dai manuali tecnici, una particolare attenzione all'implementazione hardware specifica: la divisione e *DAA*. Il problema, output indefinito in caso di input errato (vedi sezione 1.1.3), è stato risolto ricorrendo all'in-circuit debug.

La CPU reale è stata sottoposta a tutte le possibili combinazioni di input ($\{\text{registri}\} \times \{\text{flag}\}$), compresi quelli errati, ricavando delle tabelle di I/O complete. In base a quest'ultime si è potuto ricavare, sotto forma di funzione, la specifica operativa completa delle due istruzioni.

2.3 Livello 2

Il secondo livello comprende sette moduli (*aux_bases*, *opcode*, *table_HC05*, *table_HC08*, *table_HCS08*, *table_RS08*, *translation*) che servono a formalizzare la decodifica degli opcode (fetch-decode-execute) e la compilazione degli pseudocodici.

2.3.1 Rappresentazione degli ISA

La famiglia delle CPU formalizzate è di tipo CISC e comprende:

- un totale di 91 istruzioni (paragonabile alle 115 dell'Intel 8088);
- un totale di 30 differenti modalità di indirizzamento, distribuite in modo fortemente asimmetrico fra le varie istruzioni.

[Frammento di implementazione della lista che descrive la CPU HCS08]

<i>Pseudocodice</i>	<i>Modalità</i>	<i>Opcode</i>	<i>Cicli</i>
...			
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_IMM ₁	(Byte 0xAB)	2
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_DIR ₁	(Byte 0xBB)	3
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_DIR ₂	(Byte 0xCB)	4
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_IX ₂	(Byte 0xDB)	4
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_IX ₁	(Byte 0xEB)	3
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_IX ₀	(Byte 0xFB)	3
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_SP ₂	(Word 0x9EDB)	5
; Tupla4 ???? (AnyOP HCS08 ADD)	MODE_SP ₁	(Word 0x9EEB)	4
...			
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o0)	(Byte 0x10)	5
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o1)	(Byte 0x12)	5
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o2)	(Byte 0x14)	5
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o3)	(Byte 0x16)	5
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o4)	(Byte 0x18)	5
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o5)	(Byte 0x1A)	5
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o6)	(Byte 0x1C)	5
; Tupla4 ???? (AnyOP HCS08 BSETn)	(MODE_INH _{DIRn} o7)	(Byte 0x1E)	5
...			

		Bit Manipulation		Branch	Read-Modify-Write		
		DIR	DIR	REL	DIR	INH	INH
MSB	LSB	0	1	2	3	4	5
0		BRSET ₀ ⁵ ₃ DIR ₂	BSET ₀ ⁵ ₂ DIR ₂	BRA _{REL} ³ ₂	NEG _{DIR} ⁵ ₁	NEGA _{INH} ³ ₁	NEGX _{INH} ³ ₁
1		BRCLR ₀ ⁵ ₃ DIR ₂	BCLR ₀ ⁵ ₂ DIR ₂	BRN _{REL} ³ ₂			
2		BRSET ₁ ⁵ ₃ DIR ₂	BSET ₁ ⁵ ₂ DIR ₂	BHI _{REL} ³ ₂		MUL _{INH} ¹¹ ₁	

Figura 2.3.1a: Frammento di tabella opcode/pseudocodice HCS08 [Fre07]

Si è scelto di formalizzare gli ISA delle quattro CPU sotto forma di quattro distinte liste di tuple. Ogni tupla può essere identificata tramite due chiavi primarie (interpretabili come altrettante invarianti):

- 1) una combinazione univoca di pseudocodice e modalità di indirizzamento;
- 2) un opcode univoco.

La presenza di queste due chiavi primarie ha consentito di implementare cinque controlli di correttezza totalmente automatici: il rispetto delle due invarianti e l'assenza di pseudocodici, modalità e opcode non definiti all'interno delle tabelle opcode/pseudocodice presenti nelle specifiche (Figura 2.3.1a). Per comprendere quanto questi controlli possano essere utili basti considerare che, al termine della stesura manuale delle liste, hanno rilevato almeno una decina di errori sfuggiti ad un tradizionale controllo umano ripetuto tre volte.

I seguenti esempi illustrano i cinque tipi di errore rilevati automaticamente:

- le tuple [12] e [13] violano la prima invariante;
- le tuple [13] e [14] violano la seconda invariante;
- la tupla [15] contiene elementi esplicitamente vietati da [9] [10] e [11].

[9] definition **NOTIMPLEMENTEDPSEUDO**HCS08 := [... ; **SHA** ; ...].

[10] definition **NOTIMPLEMENTEDMODE**HCS08 := [... ; (**MODE_INH**_{TINY} **0x1**) ; ...].

[11] definition **NOTIMPLEMENTEDOPCODE**HCS08 := [... ; **0x31** ; ...].

[12] Tupla4 ???? (AnyOP HCS08 **ADD**) **MODE_IMM**₁ (Byte **0xAB**) 2

[13] Tupla4 ???? (AnyOP HCS08 **ADD**) **MODE_IMM**₁ (Byte **0xBB**) 3

[14] Tupla4 ???? (AnyOP HCS08 **ADD**) **MODE_DIR**₁ (Byte **0xBB**) 3

[15] Tupla4 ???? (AnyOP HCS08 **SHA**) (**MODE_INH**_{TINY} **0x1**) (Byte **0x31**) 4

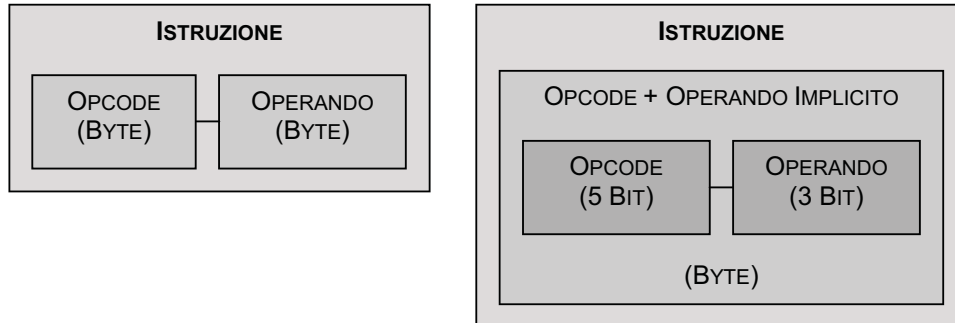


Figura 2.3.1b: Modalità di indirizzamento esplicita ad 1 byte ed implicita a 3 bit

Anche in questo caso si può notare l'utilità e la versatilità dei tipi dipendenti (1) che, insieme ai tipi algebrici (2), hanno consentito di:

- 1) creare i quattro diversi ISA delle CPU, partendo da un unico tipo base che comprende tutti gli opcode della famiglia, sotto forma di phantom type (associazione ad un tipo base di uno o più attributi caratterizzanti);

[Implementazione dei quattro ISA come phantom type]

```
inductive INSTRUCTIONSET (m:McuType) : Type :=
  AnyOP : Opcode → InstructionSet m.
```

- 2) creare delle modalità di indirizzamento ibride (modalità più operando implicito - *INHERENT*), molto utilizzate in questa famiglia di microcontroller per esigenze di compattezza del codice (Figura 2.3.1b).

[Frammento di implementazione delle modalità di indirizzamento (implicite)]

```
inductive INSTRUCTIONMODE : Type :=
  ...
  | MODE_INH_NONE:   InstructionMode
  | MODE_INH_A:     InstructionMode
  | MODE_INH_DIRn:  Octal → InstructionMode
  | MODE_INH_TINY:  Exadecimal → InstructionMode
  | MODE_INH_SHORT: Bitrigesimal → InstructionMode
  ...
```

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
CLR <i>opr8a</i>	DIR	3F	dd	5	r/wpp
CLRA	INH (A)	4F		1	p
CLR _X	INH (X)	5F		1	p
CLR _H	INH (H)	8C		1	p
CLR <i>opr8,X</i>	IX1	6F	ff	5	r/wpp
CLR ,X	IX	7F		4	r/wp
CLR <i>opr8,SP</i>	SP1	9E6F	ff	6	pr/wpp

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
LDA <i>#opr8i</i>	IMM	A6	ii	2	pp
LDA <i>opr8a</i>	DIR	B6	dd	3	r/pp
LDA <i>opr5a</i>	SRT	Cx / Dx		3	r/p

Figura 2.3.1c: Modalità di indirizzamento implicite [Fre07]

Alcuni dettagli riguardanti le modalità di indirizzamento ibride meritano, al fine di evidenziare le scelte implementative fatte, un'ulteriore approfondimento.

1) Operandi impliciti di tipo registro

Le specifiche prevedono, nel caso di istruzioni con un solo operando di tipo registro, pseudocodici diversi. L'aggiunta di pochi pseudocodici in più all'ISA, dettaglio apparentemente non problematico, viola in realtà uno dei principi ispiratori seguiti in questa formalizzazione: creare entità parametrizzabili che unifichino le logiche della CPU.

Si è quindi scelto di creare apposite modalità di indirizzamento a registro implicito che rendono ogni pseudocodice (entità logica) totalmente ortogonale a ai suoi operandi.

Un esempio può essere costituito dall'operazione di azzeramento (*CLR*) applicata ai registri A, X e H (Figura 2.3.1c). L'insieme delle relative coppie pseudocodice e modalità di indirizzamento cambia, all'interno della formalizzazione, di formato:

$$\begin{array}{l} \textit{FormatoSpecifica} \qquad \qquad \qquad \textit{FormatoFormalizzazione} \\ \langle \{CLR_A, CLR_X, CLR_H\}, INH_{NONE} \rangle \rightarrow \langle CLR, \{INH_A, INH_X, INH_H\} \rangle \end{array}$$

2) Operandi impliciti di tipo valore

Alcune istruzioni prevedono come operando un valore inferiore al byte, integrato all'interno dello stesso opcode. La principale conseguenza operativa è l'impossibilità di rispettare la normale sequenza di fetch-decode dell'opcode, seguita dal caricamento di uno o più operandi (execute).

Per evitare, durante la fase execute, di accedere una seconda volta all'opcode per estrarne i bit che costituiscono l'operando implicito, si è scelto di utilizzare tipi algebrici combinati con apposite basi:

- base 8 → operando implicito di 3 bit;
- base 16 → operando implicito di 4 bit;
- base 32 → operando implicito di 5 bit.

Un esempio può essere costituito dall'operazione di caricamento (*LDA*) applicata, con la modalità di indirizzamento *INH_{SHORT}*, ad operandi a 5 bit (Figura 2.3.1c). Le coppie pseudocodice e modalità di indirizzamento implicita sono, all'interno della formalizzazione, in corrispondenza biunivoca con gli opcode:

$$\langle LDA, \{INH_{SHORT} (00_{32} \div 1F_{32})\} \rangle \leftrightarrow \langle \{0xC0 \div 0xDF\} \rangle.$$

2.3.2 Decodifica

Le combinazioni di pseudocodice e modalità di indirizzamento superano (nelle CPU HC(S)08) la dimensione di una singola tabella di decodifica opcode/pseudocodice, fissata in 256 (i valori esprimibili da un singolo byte).

La CPU reale risolve questo problema distinguendo un particolare valore (0x9E) che svolge la funzione di selettore di pagina (precode) fra due diverse tabelle. Questo meccanismo è efficiente e consente agli sviluppatori delle CPU di ottenere una codifica completa (precode più opcode):

- contenuta in un singolo byte per la maggior parte degli pseudocodici;
- aperta a future aggiunte di nuovi opcode all'interno delle tabelle esistenti;
- aperta a future aggiunte, tramite nuovi precode, di nuove tabelle.

La codifica completa di un opcode può quindi, nel caso delle CPU formalizzate, avere una dimensione variabile di tipo byte o word. Combinando un

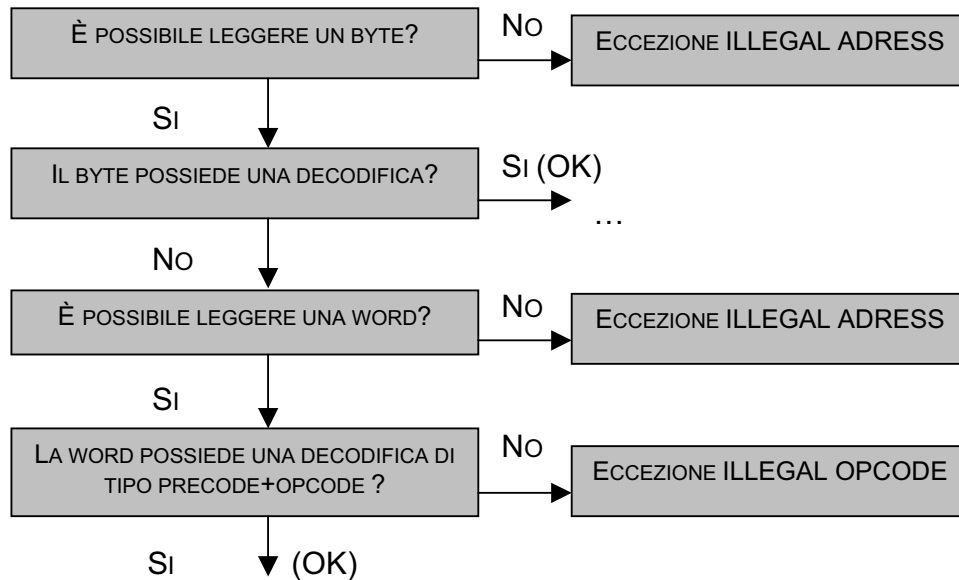


Figura 2.3.2: Algoritmo di fetch-decode per la CPU HCS08

apposito algoritmo con la possibilità di inserire valori di tipo *ByteOrWord* all'interno delle liste che descrivono le quattro CPU si possono unificare, astruendo la distinzione fra precode e opcode, tutte le distinte tabelle di decodifica opcode/pseudocodice (Figura 2.3.2).

2.3.3 Compilazione

La compilazione effettua una traduzione “al volo” (on the fly) di un sorgente scritto nell'ISA di una delle quattro CPU in una lista di byte. Il compilato (la lista) può essere:

- caricato in memoria, partendo da una qualsiasi locazione;
- combinato con altre liste di byte che rappresentano i dati;

Attraverso una serie di controlli statici, effettuati direttamente o tramite tipi dipendenti, si impedisce al sorgente di violare le seguenti invarianti:

- 1) ogni sorgente deve comprendere istruzioni appartenenti all'ISA di un'unica CPU;

- 2) ogni combinazione di pseudocodice e modalità di indirizzamento deve comparire all'interno della lista che definisce l'ISA scelto;
- 3) ogni modalità di indirizzamento deve essere usata unicamente insieme ad operandi di tipo appropriato (e.g.: nessuno, implicito, byte, word, due byte).

Il primo controllo è una conseguenza diretta dell'implementazione della funzione *compile*. Al posto di una generica lista di byte, viene restituita una lista di phantom byte, associati cioè ad una CPU. Le due medesime istruzioni [16] e [17] per esempio, una volta compilate, si riducono alle due liste [18] e [19] composte dai medesimi valori ma, a causa dei loro tipi differenti, non concatenabili.

[16] (Compile **HC08** ? ADD (CheckIMM₁ 0xFF) I)

[17] (Compile **HCS08** ? ADD (CheckIMM₁ 0xFF) I)

[18] (List (**PHANTOMBYTE HC08**))_{TIPO} [0xAB ; 0xFF]

[19] (List (**PHANTOMBYTE HCS08**))_{TIPO} [0xAB ; 0xFF]

Il terzo controllo viene effettuato direttamente dal costruttore degli operandi. Il tipo dipendente *OperandCheck* associa automaticamente ad ogni modalità di indirizzamento uno o più operandi la cui correttezza di tipo è controllata staticamente.

[Frammento di implementazione del costruttore degli operandi]

inductive **OPERANDCHECK** : InstructionMode → Type :=

...

| CheckINH_{NONE}: OperandCheck MODE_INH_{NONE}

| CheckIMM₁: Byte → OperandCheck MODE_IMM₁

| CheckIMM₂: Word → OperandCheck MODE_IMM₂

| CheckIMM₁toDIR₁: Byte → Byte → OperandCheck MODE_IMM₁toDIR₁

| CheckINH_{DIR_n}: ∀ o:Octal.Byte → OperandCheck (MODE_INH_{DIR_n} o)

| CheckINH_{TINY}: ∀ e:Exadecimal.OperandCheck (MODE_INH_{TINY} e)

| CheckINH_{SHORT}: ∀ t:Bitrigesimal.OperandCheck (MODE_INH_{SHORT} t)

...

... @ (COMPILE HCS08 MODE_IMM₁ ADD (CHECKIMM₁ 0xFF) I) @ ...

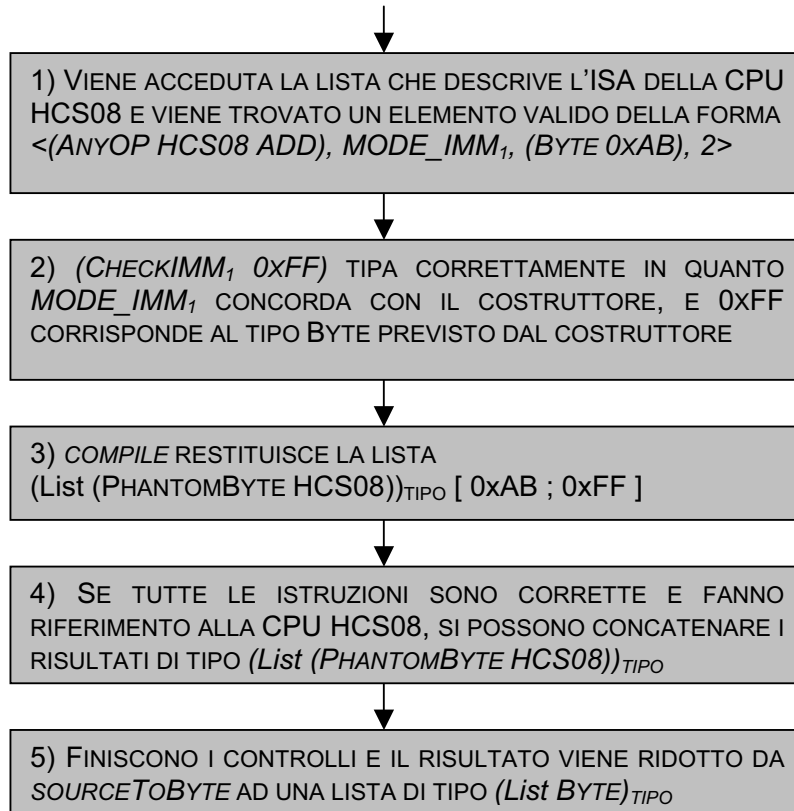


Figura 2.3.3a: Compilazione del sorgente [..., *ADD #0xFF*, ...] per HCS08

Il meccanismo di compilazione al volo, nel suo insieme, comprende cinque fasi distinte (esemplificate in Figura 2.3.3a):

- 1) una funzione controlla l'esistenza della combinazione di pseudocodice e modalità di indirizzamento e restituisce, tramite opzione, uno o due byte di traduzione (controllo diretto);
- 2) il costruttore della modalità di indirizzamento controlla la correttezza degli operandi (controllo tramite tipo dipendente);
- 3) *compile* combina la traduzione dell'opcode e degli operandi in una lista di phantom byte, dipendenti dalla CPU scelta;
- 4) le liste prodotte tramite *compile* vengono concatenate, controllando la coerenza di scelta della CPU (controllo tramite tipo dipendente);
- 5) *sourceToByte* riduce a semplice lista di byte il compilato.

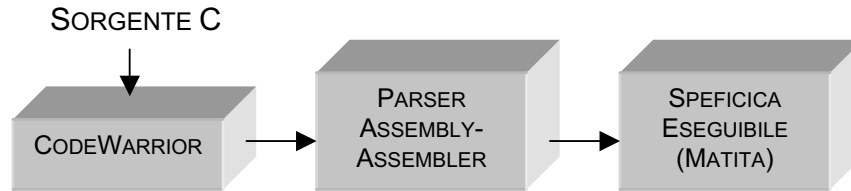


Figura 2.3.3b: Automazione del collegamento tra CodeWarrior e Matita

Questo approccio consente di inserire all'interno delle dimostrazioni, facendo uso di una notazione molto simile a quella di un compilatore, programmi sintatticamente corretti [20] e dati [21].

[20] definition **EXAMPLESOURCE** : Word \rightarrow (List Byte) :=

```

λparam:Word.
let m := HCS08 in SOURCETOBYTE m (
(* LDHX #param *)      (COMPILE m ? LDHX (CheckIMM2 param) I) @
(* STHX 4,SP *)        (COMPILE m ? STHX (CheckSP1 0x04) I) @
(* BRA *+52 *)         (COMPILE m ? BRA (CheckIMM1 0x32) I) @
(* TSX *)              (COMPILE m ? TSX CheckINHNONE I) @
(* LDA 2,X *)          (COMPILE m ? LDA (CheckIX1 0x02) I) @
(* ADD #0x00 *)        (COMPILE m ? ADD (CheckIMM1 0x00) I) @
(* PSHA *)             (COMPILE m ? PSHA CheckINHNONE I) @
(* LDA 1,X *)          (COMPILE m ? LDA (CheckIX1 0x01) I) @
(* ADC #0x01 *)        (COMPILE m ? ADC (CheckIMM1 0x01) I) @
... )
  
```

[21] definition **EXAMPLEDATA** :=

```

[ 0x11 ; 0xAF ; 0xD2 ; 0xBF ; 0x0B ; 0xBC ; 0x08 ; 0xE5 ; 0xFE ; 0xAC ;
  0xFC ; 0x94 ; 0x34 ; 0x88 ; 0x28 ; 0xDB ; 0xD3 ; 0x3D ; 0x38 ; 0x6A ;
  0x74 ; 0x49 ; 0x7E ; 0x6F ; 0x4D ; 0x3D ; 0x3A ; 0xC2 ; 0xD3 ; 0x36 ].
  
```

L'aggiunta di un parser completerebbe una catena ideale di automazione che unisce il compilatore C prodotto dalla Freescale (CodeWarrior) alla specifica eseguibile prodotta (Figura 2.3.3b). Al momento attuale è ancora necessario prelevare l'output del compilatore C Freescale ed inserirlo manualmente all'interno di Matita, come è stato fatto per i test enunciati in appendice.

2.4 Livello 3

Il terzo livello comprende cinque moduli (*memory_struct*, *memory_func*, *memory_trees*, *memory_bits*, *memory_abs*) che servono a formalizzare tre diverse implementazioni della memoria: a funzione, ad albero di byte e ad albero di bit. Un apposito modulo di astrazione consente, infine, di oscurare ai livelli superiori l'implementazione scelta.

2.4.1 Caratteristiche Comuni

Tutte e tre le implementazioni di memoria associano un attributo ad ogni unità minima di memorizzazione (byte o bit). L'attributo descrive le caratteristiche di accesso del particolare tipo di memoria:

- *MEM_READ_ONLY* nel caso di memoria ROM/EEPROM/FLASH;
- *MEM_READ_WRITE* nel caso di memoria RAM;
- *MEM_OUT_OF_BOUND* nel caso di memoria assente.

L'associazione, per motivi di efficienza di implementazione, non avviene tramite coppie di tipo *<Attributo, Valore>* ma prevede due distinti descrittori: uno per gli attributi ed uno per i valori. Può essere infatti necessario:

- impostare il tipo di intere regioni di memoria senza modificarne i valori contenuti (e.g.: istanziazione della memoria del microcontroller, abilitazione/disabilitazione in scrittura di blocchi di memoria FLASH);
- azzerare, sempre durante l'istanziazione della memoria del microcontroller, intere regioni.

Ogni accesso in lettura e scrittura, a causa della possibile assenza di unità di memorizzazione, restituisce un valore di tipo opzione. La scrittura invece, a causa della presenza di memoria abilitata in sola lettura, può avere successo ma lasciare inalterati i contenuti della memoria.

Lo spazio di indirizzamento è, in tutti i modelli, di 64Kb (indirizzi a 16 bit) e comprende: RAM, *MEM_READ_ONLY* in cui risiede il programma in esecuzione e locazioni adibite a memory-mapped I/O.

2.4.2 Memoria a Funzione

L'implementazione a funzione della memoria (*FUNC*) associa ad ogni indirizzo una funzione descrittrice:

- degli attributi (di tipo *Word* \rightarrow *MemoryType*);
- dei valori (di tipo *Word* \rightarrow *Byte*).

La memoria a funzione non utilizza, né simula, array di elementi. Le modifiche effettuate ad un descrittore provocano, al posto della modifica di un elemento, l'incapsulamento dell'intero descrittore all'interno di una nuova funzione descrittrice. Gli accessi in lettura, come succede utilizzando degli array, non provocano nessuna modifica al descrittore.

L'implementazione è molto semplice e presenta, all'inizio, un'estrema compattezza. L'intera memoria iniziale (64Kb di zeri) viene descritta attraverso la funzione *zeroMemory* che si limita a restituire 0x00 indipendentemente dall'indirizzo passato.

[Implementazione della memoria a funzione]

```
definition ZEROMEMORY :=  $\lambda$ _:Word.0x00.
```

```
definition MEMORYWRITE :=
 $\lambda$ mem:Word  $\rightarrow$  Byte. $\lambda$ check:Word  $\rightarrow$  MemoryType. $\lambda$ address:Word. $\lambda$ value:Byte.
match (check address) with
[ MEM_READ_ONLY => Some ? mem
| MEM_READ_WRITE =>
Some ? ( $\lambda$ x.match (wordEq x address) with [ true => value | false => (mem x) ])
| MEM_OUT_OF_BOUND => None ? ].
```

```
definition MEMORYREAD :=
 $\lambda$ mem:Word  $\rightarrow$  Byte. $\lambda$ check:Word  $\rightarrow$  MemoryType. $\lambda$ address:Word.
match (check address) with
[ MEM_READ_ONLY => Some ? (mem address)
| MEM_READ_WRITE => Some ? (mem address)
| MEM_OUT_OF_BOUND => None ? ].
```

Si pone invece, considerando l'utilizzo che segue l'instanziamento, un serio problema di dimensione del descrittore. Ogni successivo accesso in scrittura, se ha successo, ne aumenta di un passo elementare (un match annidato) la dimensione. La complessità computazionale derivante, anche in sola lettura, può diventare lineare nel numero di modifiche effettuate.

Quattro accessi in scrittura, anche non sequenziali, alla stessa locazione di memoria 0x100, per esempio, aumentano di quattro passi elementari la dimensione del descrittore. Il descrittore [22] è equivalente a quello derivante dalla sola ultima scrittura [23], ma il sistema è costretto a tenere traccia anche delle tre precedenti modifiche.

```
[22] (λx1.match (wordEq x1 0x0100) with [ true => 0x04 | false => (
  (λx2.match (wordEq x2 0x0100) with [ true => 0x03 | false => (
    (λx3.match (wordEq x3 0x0100) with [ true => 0x02 | false => (
      (λx4.match (wordEq x4 0x0100) with [ true => 0x01 | false =>
        (OLDMEMORY x4) ]) x3) ]) x2) ]) x1) ])
```

```
[23] (λx.match (wordEq x 0x0100) with [ true => 0x04 | false => (OLDMEMORY x) ])
```

2.4.3 Memoria ad Albero di Byte

L'implementazione ad albero di byte della memoria (*TREE*) associa ad ogni indirizzo un elemento all'interno di un albero perfettamente bilanciato (di $16^4 = 64\text{Kb}$ elementi):

- di attributi (di tipo $Tree_{16} (Tree_{16} (Tree_{16} (Tree_{16} MemoryType)))$);
- di valori (di tipo $Tree_{16} (Tree_{16} (Tree_{16} (Tree_{16} Byte)))$).

Gli alberi sono B-Tree perfettamente bilanciati di profondità 4 (Figura 2.4.3) nella cui implementazione:

- i nodi interni contengono sempre 16 puntatori. Non sono necessarie chiavi in quanto l'ordine logico dei puntatori ([0x0-F]) costituisce la chiave di accesso al livello sottostante. L'indirizzo 0x15AC, per esempio, si traduce nella sequenza di ricerca (partendo dalla radice): chiave 0x1 al livello 1, chiave 0x5 al livello 2, chiave 0xA al livello 3, chiave 0xC al livello 4;
- le foglie contengono sempre un solo elemento.

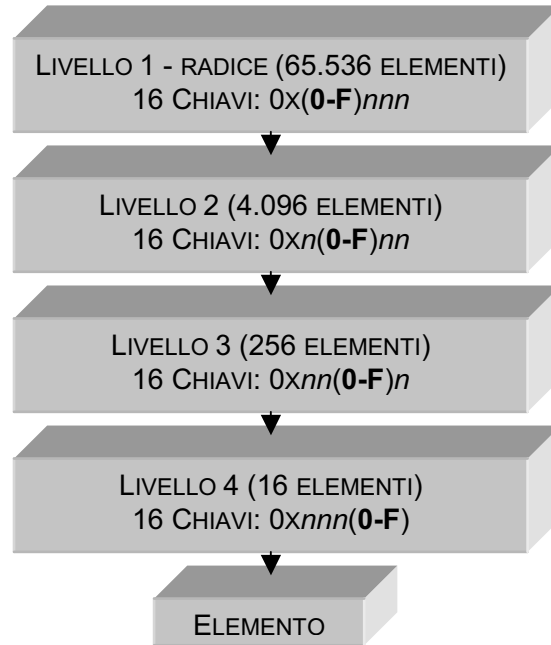


Figura 2.4.3: Diagramma del B-Tree di 64Kb elementi suddiviso in 4 livelli

Per la formalizzazione dei nodi e delle foglie è stato utilizzato l'apposito tipo dipendente *Tree16*, che prevede la presenza di 16 argomenti non opzionali e di tipo omogeneo. In tal modo si avranno:

- al livello 4 elementi di tipo (*Tree₁₆ xxx*);
- al livello 3 elementi di tipo (*Tree₁₆ (Tree₁₆ xxx)*);
- al livello 2 elementi di tipo (*Tree₁₆ (Tree₁₆ (Tree₁₆ xxx))*);
- una radice di tipo (*Tree₁₆ (Tree₁₆ (Tree₁₆ (Tree₁₆ xxx)))*).

[Implementazione degli alberi]

inductive **TREE16_T** (T:Type) : Type :=

TREE16_C: T → T → T → T → T → T → T → T → T →
T → T → T → T → T → T → T → T → T → **TREE16_T** T.

definition **GETCHILD** :=

λchild:Exadecimal.λT:Type.λnode:**TREE16_T** T.
match node with [**TREE16_C** e00 e01 ... e15 =>
match child with [0x0 => e00 | 0x1 => e01 ... | 0xF => e15]].

L'intera memoria iniziale viene descritta attraverso l'albero completo *zeroMemory* e ogni successivo accesso in lettura o scrittura non ne altera mai struttura, ordinamento e bilanciamento.

Il principale vantaggio di questa implementazione, rispetto alla memoria a funzione, è una complessità computazionale costante: 4 accessi per la lettura e 7 per la scrittura.

[Implementazione della memoria ad albero di byte]

definition **ZEROMEMORY** :=

```
let lev4 := (Tree16 ? 0x0 0x0 ... 0x0) in
let lev3 := (Tree16 ? lev4 lev4 ... lev4) in
let lev2 := (Tree16 ? lev3 lev3 ... lev3) in
let lev1 := (Tree16 ? lev2 lev2 ... lev2) in
lev1.
```

definition **TREEVISIT** :=

```
λT:Type.λroot:Tree16 (Tree16 (Tree16 (Tree16 T))).λaddress:Word.
match address with
[ MakeWord highByte lowByte =>
GETCHILD (LowerDigit lowByte) ?
GETCHILD (HigherDigit lowByte) ?
GETCHILD (LowerDigit highByte) ?
GETCHILD (HigherDigit highByte) ? root) ].
```

definition **TREEUPDATE** :=

```
λT:Type.λdata:Tree16 (Tree16 (Tree16 (Tree16 T))).λaddress:Word.λvalue:T.
match address with
[ MakeWord highByte lowByte =>
let lev2 := (GETCHILD (HigherDigit highByte) ? root) in
let lev3 := (GETCHILD (LowerDigit highByte) ? lev2) in
let lev4 := (GETCHILD (HigherDigit lowByte) ? lev3) in
SETCHILD (HigherDigit highByte) ? root
SETCHILD (LowerDigit highByte) ? lev2
SETCHILD (HigherDigit lowByte) ? lev3
SETCHILD (LowerDigit lowByte) T lev4 value) ].
```

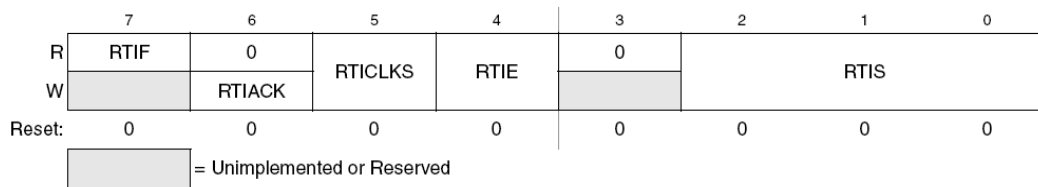


Figura 2.4.4: Esempio di byte usato per il memory-mapped I/O [Fre07]

2.4.4 Memoria ad Albero di Bit

La memoria ad albero di bit (*BITS*) è quasi identica a quella della memoria ad albero di byte, tranne che per il tipo dell'unità minima di memorizzazione: un insieme di 8 bit (booleani) a cui possono essere associati attributi indipendenti.

Quest'ultima implementazione si avvicina in modo ancora più preciso, rispetto alle due precedenti, al comportamento reale della memoria dei microcontroller formalizzati. La comunicazione con i vari dispositivi di controllo integrati (e.g.: seriale, tastiera, ADC, timer, USB) avviene in modalità memory-mapped I/O. In molti casi, alcuni bit delle parole di controllo sono abilitati in sola lettura o addirittura disabilitati (Figura 2.4.4).

[Implementazione della memoria ad albero di bit]

definition **ZEROMEMORY** :=

```

let byte := (Tree8 ? false false ... false) in
let lev4 := (Tree16 ? byte byte ... byte) in
let lev3 := (Tree16 ? lev4 lev4 ... lev4) in
let lev2 := (Tree16 ? lev3 lev3 ... lev3) in
let lev1 := (Tree16 ? lev2 lev2 ... lev2) in
lev1.

```

Dal punto di vista implementativo l'unica differenza consiste nella granularità di accesso agli alberi descrittivi dei valori e degli attributi e la complessità computazionale degli accessi in lettura e scrittura resta costante.

2.4.5 Astrazione di Accesso

Le operazioni inerenti alla memoria richieste dai livelli successivi sono:

- 1) istanziiazione di *zeroMemory* (tutta la memoria azzerata) e istanziiazione di *outOfBoundMemory* (tutta la memoria non installata);
- 2) impostazione degli attributi di una regione di memoria o di un singolo bit;
- 3) caricamento di una lista di byte (sorgente compilato o dati);
- 4) lettura e scrittura di un byte o di un singolo bit.

I primi tre insiemi di operazioni vengono richiamati, secondo l'ordine in cui appaiono, unicamente durante l'istanziiazione di un microcontroller. L'ultimo insieme di operazioni viene invece utilizzato, durante la successiva emulazione del ciclo di fetch-decode-execute.

Queste operazioni sono implementate sotto forma di multiplexer con tipo dipendente (e.g.: *memoryRead_{ABS}*) e permettono ai livelli successivi di astrarre, una volta scelto il modello di memoria, dai dettagli implementativi (e.g.: *memoryRead_{FUNC}*, *memoryRead_{TREE}*, *memoryRead_{BITS}*).

[Implementazione dell'astrazione di accesso]

definition **AUX_{MEM}** :=

```

λt:MemoryImplementaion.match t with
  [ FUNC => Word → Byte
  | TREE => Tree16 (Tree16 (Tree16 (Tree16 Byte)))
  | BITS => Tree16 (Tree16 (Tree16 (Tree16 (Tree8 Boolean)))) ].

```

definition **MEMORYREAD_{ABS}** : (Πt.AUX_{MEM} t → aux_{CHK} t → Word → Option Byte) :=

```

λt:MemoryImplementation.match t with
  [ FUNC => λmem:(AUXMEM FUNC).λcheck:(auxCHK FUNC).λaddr:Word.
    MEMORYREADFUNC mem check addr
  | TREE => λmem:(AUXMEM TREE).λcheck:(auxCHK TREE).λaddr:Word.
    MEMORYREADTREE mem check addr
  | BITS => λmem:(AUXMEM BITS).λcheck:(auxCHK BITS).λaddr:Word.
    MEMORYREADBITS mem check addr ].

```

REGISTER HC05	HIGHER BYTE	LOWER BYTE							
A - ACCUMULATOR	X	X	X	X	X	X	X	X	X
X - INDEX		X	X	X	X	X	X	X	X
SP - STACK POINTER	0	0	0	0	0	0	0	0	0
PC - PROGRAM COUNTER	0	0	0	X	X	X	X	X	X

REGISTER HC(S)08	HIGHER BYTE	LOWER BYTE							
A - ACCUMULATOR	X	X	X	X	X	X	X	X	X
H:X - INDEX		X	X	X	X	X	X	X	X
SP - STACK POINTER	X	X	X	X	X	X	X	X	X
PC - PROGRAM COUNTER	X	X	X	X	X	X	X	X	X

Figura 2.5.1a: Registri delle CPU HC05, HC08 e HCS08 [Fre07]

2.5 Livello 4

Il quarto livello comprende due moduli (*status*, *model*) che servono a formalizzare l'istanziamento e la manipolazione della CPU e dello stato.

2.5.1 CPU

Ogni CPU formalizzata (HC05, HC08, HCS08 e RS08) è implementata sotto forma di record che comprende:

- registri di tipo word (16 bit) o byte (8bit);
- flag di tipo booleano (1 bit).

L'implementazione dei registri ha richiesto un'attenzione particolare per il registro PC, il registro SP.

Il registro PC, in tutti i microcontroller, è strettamente connesso alla dimensione della memoria. Se quest'ultima è inferiore ai 64Kb, i bit superiori del registro vengono disabilitati in scrittura ed impostati a 0 (Figura 2.5.1a e 2.5.1b),

impedendogli di assumere valori esterni allo spazio di indirizzamento della memoria. Una memoria di 8Kb, per esempio, è associata allo spazio di indirizzamento $[00000000.00000000_2, 00011111.11111111_2]$ e prevede la disabilitazione dei tre bit superiori.

Il registro SP, nella famiglia HC05, è strettamente connesso alla dimensione dello stack. Il vincolo a non violare lo spazio di indirizzamento, come per il caso del registro PC, determina la disabilitazione in scrittura dei bit superiori. Il limite inferiore però, in questo caso, è sempre maggiore di 0 e, per questo motivo, alcuni bit devono essere impostati ad 1. Uno stack compreso nella regione $[0x00C0, 0x00FF]$, per esempio, è associato allo spazio di indirizzamento $[00000000.11000000_2, 00000000.11111111_2]$ e prevede la disabilitazione dei dieci bit superiori, impostandone gli ultimi due ad 1.

Questo comportamento, variabile da modello a modello all'interno della stessa famiglia, è stato formalizzato associando a questi due registri dei descrittori aggiuntivi, che consentono la protezione in scrittura dei bit superiori (maschera) e l'impostazione ad un valore fisso (parte fissa).

Il registro PC richiede la sola maschera (ad esempio 0x1FFF per disabilitare i tre bit superiori), mentre il registro SP richiede maschera e parte fissa (ad esempio 0x003F per disabilitare i dieci bit superiori e 0x00C0 per impostarne gli ultimi due ad 1):

$$PC_{NEW} = (Value \& PC_{MASK})$$

$$SP_{NEW} = ((Value \& SP_{MASK}) | SP_{FIX})$$

[Frammento di implementazione della CPU HC05 (registri)]

```
record CPU05 : Type := {
  A_Reg05:   Byte;
  X_Reg05:   Byte;
  SP_Reg05:  Word;
  SP_Mask05: Word;
  SP_Fix05:  Word;
  PC_Reg05:  Word;
  PC_Mask05: Word;
  ... }.
```

REGISTER RS08	HIGHER BYTE							LOWER BYTE							
A – ACCUMULATOR	X							X							
PC - PROGRAM COUNTER	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X
SPC - SHADOW PROGRAM COUNTER	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X
MAPPED REGISTER RS08	HIGHER BYTE							LOWER BYTE							
(D[X] → 0x000E) - INDEXED DATA	X							X							
(X → 0x000F) – INDEX	X							X							
(PS → 0x001F) - PAGE SELECT	X							X							

Figura 2.5.1b: Registri della CPU RS08 [Fre07]

La CPU RS08 possiede delle caratteristiche che la distinguono nettamente dalle altre:

- tre registri memory-mapped X, D[X] e PS (Figura 2.5.1b);
- una RAM di paging indicizzata tramite il registro PS;
- modalità di indirizzamento implicite che consentono l'accesso diretto ai registri memory-mapped.

I registri X, D[X] e PS corrispondono a delle locazioni di memoria accessibili tramite le modalità di indirizzamento implicite *TINY* e *SHORT* (Figura 2.5.1c), consentendo una codifica efficiente dell'accesso, esprimibile tramite un byte.

Il registro X svolge la stessa funzione di indice prevista nella altre CPU e, accedendo al registro D[X] (ossia l'offset puntato da X), si può simulare la modalità di indirizzamento indicizzata, altrimenti assente.

Il registro PS determina la posizione, all'interno dell'intero spazio di indirizzamento della RAM, della finestra di paging. In tal modo, combinando l'uso del paging e della modalità di indirizzamento *DIR₁* (diretta ad un byte di offset), si può simulare, accedendo alla finestra di paging, la modalità di indirizzamento *DIR₂* (diretta a due byte di offset):

$$Address_{REAL} = ((PS \ll 8) | (Address_{PAGING} \& 0x003F))$$

L'obiettivo principale di queste scelte di design è la compattezza del codice, punto critico a causa della ridotta memoria equipaggiata da questi microcontroller.

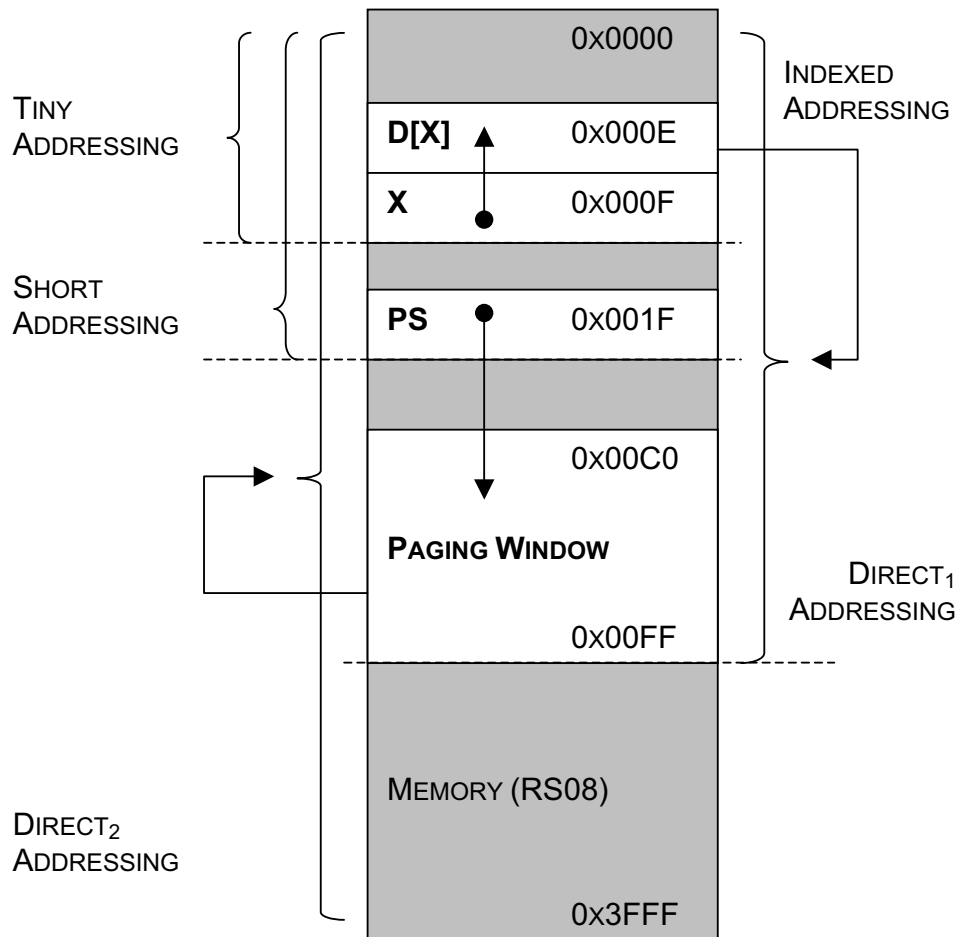


Figura 2.5.1c: Funzionamento della memoria dei modelli RS08 [Fre07]

All'interno della formalizzazione, i registri X e PS sono stati implementati come normali registri all'interno del record che descrive la CPU RS08 e vengono acceduti in lettura e scrittura intercettando gli accessi alle loro corrispondenti locazioni di memoria.

Il registro D[X] invece, dipendente dal valore del registro X e dal contenuto della memoria, non è stato inserito all'interno della CPU RS08. Il suo comportamento, assieme a quello del paging, è interpretabile piuttosto come una modalità di indirizzamento simulata, assimilabile all'accesso alla memoria formalizzato nel livello successivo.

2.5.2 Stato

Lo stato complessivo del microcontroller, tenendo conto dell'obiettivo di astrazione introdotto all'inizio (vedi sezione 2.1.1), è costituito da:

- una CPU, scelta fra le quattro che compongono la famiglia dei microcontroller a 8 bit Freescale;
- una memoria, scelta fra le tre implementazioni realizzate;
- un descrittore dello stato di esecuzione della CPU (run, wait o suspended) e dell'eventuale stato di avanzamento di esecuzione.

Lo stato è quindi un'entità dipendente dal tipo di CPU e dall'implementazione della memoria e, per la sua formalizzazione, è stato scelto un record dipendente.

[Frammento di implementazione dello stato (elementi dipendenti)]

```
record STATUS (mcu:McuType) (t:MemoryImplementaion) : Type := {
CPU_DESC:    match mcu with
              [ HC05 => CPU05   | HC08 => CPU08
              | HCS08 => CPU08   | RS08 => CPURS ];
Mem_DESC:    auxMEM t;
Check_DESC:  auxCHK t;
... }.
```

Nel livello successivo, come è già stato accennato trattando la formalizzazione dell'ISA, verrà associato ad ogni pseudocodice un'unica logica di esecuzione. Ciò può essere considerato come l'obiettivo a cui tende l'intero impianto di formalizzazione e richiede, a questo livello, degli appositi strumenti che consentano di manipolare in maniera uniforme i registri ed i flag di qualsiasi stato.

Sono stati quindi realizzati dei getter e dei setter che, in caso di assenza dell'elemento richiesto, segnalano l'errore (grave in questo caso, in quanto di formalizzazione) tramite un'opzione.

Un esempio di errore di questo tipo può essere l'esecuzione, da parte di una CPU HC05, di un salto condizionato basato sul flag di overflow. Questo flag, presente nelle CPU HC(S)08 ma non in quella in esecuzione, risulta essenziale per

determinare l'esecuzione del salto e la sua assenza può derivare unicamente da un bug della formalizzazione (e.g.: errata traduzione di un opcode, errata associazione della CPU allo stato).

Ai normali setter sono stati affiancati anche dei setter deboli che, in caso di assenza dell'elemento richiesto, non segnalano alcun errore. Un esempio di utilizzo di setter debole può essere la modifica del flag di overflow in seguito all'esecuzione di una somma da parte di una CPU HC05. La logica della somma considera questa modifica come un semplice effetto collaterale opzionale (a seconda della CPU) e il setter debole consente di astrarre da questo dettaglio.

L'implementazione dei getter e dei setter assume la forma di multiplexer con tipo dipendente (e.g.: *getX_Reg*, *setX_Reg*) e permette di astrarre dalla CPU specifica (e.g.: *X_Reg08*, *setX_RegHC08*).

[Implementazione di getter, setter e setter debole]

```
definition GETX_REG : (Πm.Πt.Status m t → Option Byte) :=
  λm:McuType.λt:MemoryImplementation.λs:Status m t.
  match m with
  [ HC05 => λcpu.Some ? (X_REG05 cpu) | HC08 => λcpu.Some ? (X_REG08 cpu)
  | HCS08 => λcpu.Some ? (X_REG08 cpu) | RS08 => λcpu.None ? ]
  (CPUDESC m t s).
```

```
definition SETX_REGHC08 : (CPU08 → Byte → CPU08) :=
  λcpu:CPU08.λregx':Byte.
  match cpu with [ MakeCPU08 regA _ regH ... => MakeCPU08 regA regx' regH ... ].
```

```
definition SETX_REG (Πm.Πt.Status m t → Byte → Option (Status m t)) :=
  λm:McuType.λt:MemoryImplementation.λs:Status m t.λregx':Byte.
  optionMap ?? (match m with
  [ HC05 => Some ? SETX_REG05 | HC08 => Some ? SETX_REG08
  | HCS08 => Some ? SETX_REG08 | RS08 => None ? ])
  (λf.Some ? (setCPU m t s (f (CPUDESC m t s) regx'))).
```

```
definition SETWEAKX_REG (Πm.Πt.Status m t → Byte → Status m t) :=
  λm:McuType.λt:MemoryImplementation.λs:Status m t.λregx':Byte.
  match (SETX_REG m t s regx') with [ None => s | Some s' => s' ].
```

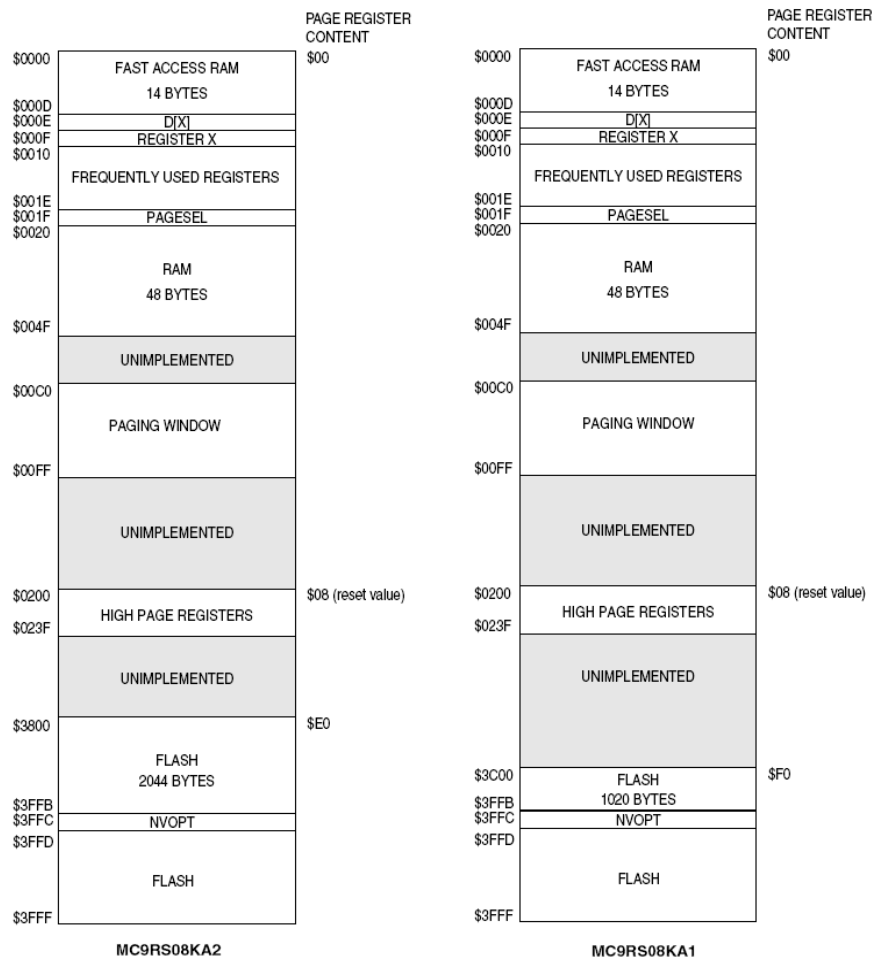


Figura 2.5.3: Memoria dei modelli RS08 [Fre07]

2.5.3 Instanziazione e Reset dello Stato

Per istanziare correttamente la memoria di uno stato, bisogna anche associare ad ogni regione di memoria il corretto attributo che ne descriva la modalità di accesso: *READ_ONLY* per la ROM/EPROM/FLASH, *READ_WRITE* per la RAM e *OUT_OF_BOUND* quando non è installata (Figura 2.5.3).

Ciò è stato implementato tramite delle liste che, modello per modello, elencano tutte le regioni da inizializzare (associate cioè ad un attributo diverso da *OUT_OF_BOUND*) rispetto ad una memoria *outOfBoundMemory* (tutta non installata):

< *InizioRegione*, *FineRegione*, *Attributo* >

[Implementazione della lista che descrive i modelli RS08]

```

definition MEMORYTYPERS08 :=
  λm:McuParamModelRS08.match m with
  [ MC9RS08KA1 =>
    [ Tupla3 ???      0x0000      0x000E      MEM_READ_WRITE
    ; Tupla3 ???      0x0010      0x001E      MEM_READ_WRITE
    ; Tupla3 ???      0x0020      0x004F      MEM_READ_WRITE
    ; Tupla3 ???      0x00C0      0x00FF      MEM_READ_WRITE
    ; Tupla3 ???      0x0200      0x023F      MEM_READ_WRITE
    ; Tupla3 ???      0x3C00      0x3FFF      MEM_READ_ONLY ]
  | MC9RS08KA2 =>
    [ Tupla3 ???      0x0000      0x000E      MEM_READ_WRITE
    ; Tupla3 ???      0x0010      0x001E      MEM_READ_WRITE
    ; Tupla3 ???      0x0020      0x004F      MEM_READ_WRITE
    ; Tupla3 ???      0x00C0      0x00FF      MEM_READ_WRITE
    ; Tupla3 ???      0x0200      0x023F      MEM_READ_WRITE
    ; Tupla3 ???      0x3800      0x3FFF      MEM_READ_ONLY ]
  ].

```

Il termine istanziazione può essere paragonato, dal punto di vista hardware, all'accensione o al riavvio (causato da una interruzione o sbalzo eccessivo di corrente) e comporta la costruzione ex novo di uno stato, CPU e memoria comprese. I valori di alcuni registri e di alcuni flag, però, vengono descritti nei manuali tecnici, come non deterministici all'inizio dell'esecuzione, introducendo un problema: come simulare il non determinismo all'interno di una formalizzazione totalmente deterministica?

La soluzione adottata prevede semplicemente di spostare il non determinismo al di fuori dell'istanziamento, aggiungendo ai parametri in ingresso un insieme di valori da associare ad ogni entità non deterministica. In questo modo, inserendo l'istanziamento all'interno di un predicato che quantifichi universalmente rispetto ad ogni parametro non deterministico, si può simulare in maniera corretta il non determinismo:

$$\forall \text{NonDeterministico}_1. \dots \forall \text{NonDeterministico}_n. \\
(\text{Istanziamento} \dots \text{NonDeterministico}_1 \dots \text{NonDeterministico}_n)$$

Un altro meccanismo, molto usato in questi microcontroller, è il reset che corrisponde, dal punto di vista hardware, ad un riavvio (non causato da una interruzione o sbalzo eccessivo di corrente) che non cancelli la memoria.

Un microcontroller adibito alla gestione di un timer, per esempio, potrebbe comportarsi nel seguente modo:

- 1) aggiornare il timer;
- 2) programmare il gestore interno per provocare, in mancanza di una segnalazione qualsiasi che il sistema non sia entrato in stallo, un reset entro esattamente un 1ms;
- 3) entrare in loop infinito ed attendere il reset.

Questo meccanismo, dal punto di vista della formalizzazione, è equivalente all'istanziamento di un nuovo stato, al cui interno venga inserita la memoria estratta dallo stato precedente al reset.

2.6 Livello 5

L'ultimo livello comprende due moduli (*load_write*, *multivm*) che servono a formalizzare la transizione da stato a stato.

2.6.1 Modalità di Indirizzamento

L'esecuzione di una qualsiasi istruzione deve poter:

- astrarre dal tipo di CPU (registri e flag);
- astrarre dalla modalità di indirizzamento (accesso agli operandi).

L'astrazione dal tipo di CPU non richiede, a questo livello, nessun accorgimento specifico in quanto le quattro CPU costituiscono una vera e propria famiglia: gli pseudocodici comuni a più CPU implementano logiche omogenee, le cui lievi differenze sono totalmente oscurate tramite l'utilizzo dei getter e dei setter (specialmente quelli deboli).

L'astrazione dalle 30 diverse modalità di indirizzamento, invece, richiede l'utilizzo di un apposito livello di astrazione (e.g.: *multiModeLoad*,

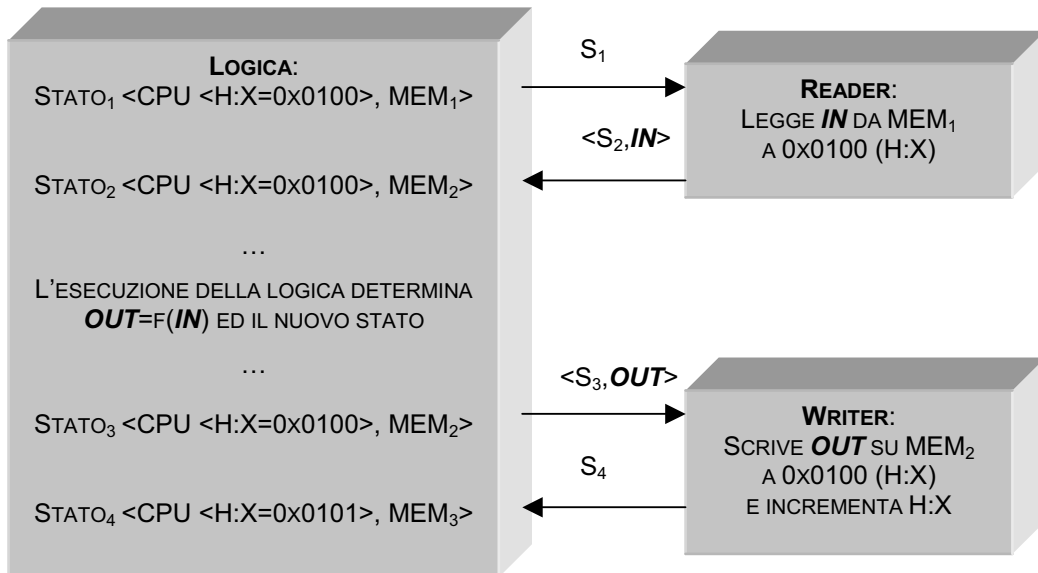


Figura 2.6.1: Modalità di indirizzamento $MODE_IX_0^{++}$

multiModeWrite) che nasconda alle logiche:

- la natura degli operandi (registro, locazione di memoria o implicito);
- la presenza di eventuali effetti collaterali (scrittura su un operando di tipo registro, post incremento del registro indice, accesso ad una locazione di memoria tramite paging).

Il problema della gestione degli effetti collaterali, punto molto delicato per le possibili modifiche allo stato, è comunque facilmente risolto facendo uso della seguente procedura (Figura 2.6.1):

- 1) la logica richiama il loader passando lo stato₁;
- 2) il loader restituisce l'operando ed uno stato₂;
- 3) la logica richiama il writer passando il risultato e lo stato₃;
- 4) il writer restituisce lo stato₄.

L'implementazione assume la forma di multiplexer con tipo dipendente (e.g.: *multiModeWrite*) e permette di astrarre dalla modalità di indirizzamento specifica (e.g.: *setA_Reg.*, *modeDIR₁Write*) e, mediante un filtro, (e.g.: *memoryFilterRead*) dal meccanismo di paging dei modelli RS08 (vedi sezione 2.5.1).

[Implementazione del writer]

```

definition MULTIMODEWRITE :
  (ΠbyteFlag.Πm.Πt.Status m t → Word → InstructionMode →
  (match byteFlag with [ true => Byte | false => Word ]) →
  Option (Couple (Status m t) Word)) :=
  λbyteFlag:Boolean.λm:McuType.λt:MemoryImplementation.
  match byteFlag with
  (* scrittura di un byte *)
  [ true => λs:Status m t.λcurPC:Word.λi:InstructionMode.λvalue:Byte.
  match i with
  | MODE_INHA => Some ? (Couple ?? (SETA_REG m t s value) curPC)
  | MODE_DIR1 => MODEDIR1WRITE true m t s curPC value
  | ... ]
  (* scrittura di una word *)
  | false => λs:Status m t.λcurPC:Word.λi:InstructionMode.λvalue:Word.
  match i with
  | MODE_INHA => None ?
  | MODE_DIR1 => MODEDIR1WRITE false m t s curPC value
  | ... ].

```

La dimensione degli operandi (byte o word) viene determinata dalla logica, ma non tutte le combinazioni di dimensione e modalità sono ammesse. Il controllo di validità, a causa dell'irregolarità delle combinazioni ammissibili, non è purtroppo automatizzabile ed è implementato attraverso una griglia.

Il loader ed il writer, nel caso di una combinazione non ammessa, segnalano l'errore (grave in questo caso, in quanto di formalizzazione) tramite un'opzione.

2.6.2 Unificazione delle Logiche

Alcune istruzioni possono essere raggruppate, dal punto di vista dell'esecuzione, in specifici sottoinsiemi dell'ISA caratterizzati da:

- 1) un'unica logica applicata a parametri diversi;
- 2) logiche ed effetti collaterali diversi, applicati ad un'unica tipologia di operandi e modalità di indirizzamento.

In entrambi i casi si è scelto di formalizzare un unico nucleo (proprietà del sottoinsieme) a cui le istruzioni forniscono, come argomento, i parametri o le logiche (proprietà degli elementi). Questa scelta ha consentito di modularizzare e snellire significativamente il livello di esecuzione, individuando i seguenti sottoinsiemi:

- somma e sottrazione, con e senza riporto;
- incremento, decremento e complemento;
- scorrimento destro e sinistro, con e senza riporto;
- salto condizionato ed incondizionato;
- lettura, scrittura e salto con maschera di bit.

Un esempio di unica logica applicata a parametri diversi è costituito dall'insieme di istruzioni di salto condizionato. Il nucleo si occupa della gestione del caricamento dell'offset del salto e, nel caso che la condizione si verifichi, della modifica del program counter. L'istruzione si limita, invece, a controllare il verificarsi o meno della condizione.

[Implementazione dell'unificazione del salto condizionato]

definition **EXECUTEANYBRANCH** :=

```

λm:McuType.λt:MemoryImplementation.λs:Status m t.
λi:InstructionMode.λcurPC:Word.λCONDITION:Boolean.
optionMap ?? (MULTIMODELOAD true m t s curPC i)
(λS_M_PC.match S_M_PC with
 [ Tupla3 status1 operand nextPC => match CONDITION with
   (* true => newPC = nextPC + operand | false => newPC = nextPC *)
 [ true => Some ? (Couple ?? status1 (branchPC m t status1 nextPC operand))
 | false => Some ? (Couple ?? status1 nextPC) ]]).

```

definition **EXECUTEBHCS** :=

```

λm:McuType.λt:MemoryImplementation.λs:Status m t.
λi:InstructionMode.λcurPC:Word.
optionMap ?? (GETH_FLAG m t s)
(* salta se il flag H è 1 => condition = HOP *)
(λHOP. EXECUTEANYBRANCH m t s i curPC HOP).

```

Un esempio di logiche ed effetti collaterali diversi, applicati ad un'unica tipologia di operandi, è costituito dall'insieme di istruzioni di incremento, decremento e complemento. Il nucleo si occupa di tutti gli aspetti dell'esecuzione (caricamento dell'operando, scrittura del risultato, modifica dei flag ed effetti collaterali) e, in particolare, per la logica e la modifica dei flag di carry e di overflow utilizza le funzioni fornite dall'istruzione.

[Implementazione dell'unificazione dell'incremento]

```

definition EXECUTECOM_DEC_INC_NEGAUX :=
  λm:McuType.λt:MemoryImplementation.λs:Status m t.
  λi:InstructionMode.λcurPC:Word.λFUNCM:Byte → Byte.
  λFUNCV:Boolean → Boolean → Boolean.λFUNCC:Boolean → Byte → Boolean.
  optionMap ?? (MULTIMODELOAD true m t s curPC i)
  (λS_M_PC.match S_M_PC with [ Tupla3 status1 MOP _ =>
    (* R = funcM(M) *)
    let ROP := (FUNCM MOP) in
    optionMap ?? (MULTIMODEWRITE true m t status1 curPC i ROP)
    (λS_PC.match S_PC with [ Couple status2 nextPC =>
      (* C = funcC(C,R) *)
      let status3 := (SETC_FLAG m t status2 (FUNCC (GETC_FLAG m t status2) ROP)) in
      (* Z = nR7&nR6&nR5&nR4&nR3&nR2&nR1&nR0 *)
      let status4 := (SETZ_FLAG m t status3 (byteEq ROP 0x00)) in
      (* N = R7 *)
      let status5 := (SETWEAKN_FLAG m t status4 (byteMSB ROP)) in
      (* V = funcV(M7,R7) *)
      let status6 := (SETWEAKV_FLAG m t status5
        (FUNCV (byteMSB MOP) (byteMSB ROP))) in
      (* newPC = nextPC *) Some ? (Couple ?? status6 nextPC) ])).

```

```

definition EXECUTEINC :=
  λm:McuType.λt:MemoryImplementation.λs:Status m t.
  λi:InstructionMode.λcurPC:Word.
  EXECUTECOM_DEC_INC_NEGAUX m t s i curPC
  (* funcM = successore *) byteSucc
  (* funcV = nM7&R7 *) (λM7.λR7.(boolNot M7) boolAnd R7)
  (* funcC = C *) (λCOP.λROP.COP).

```

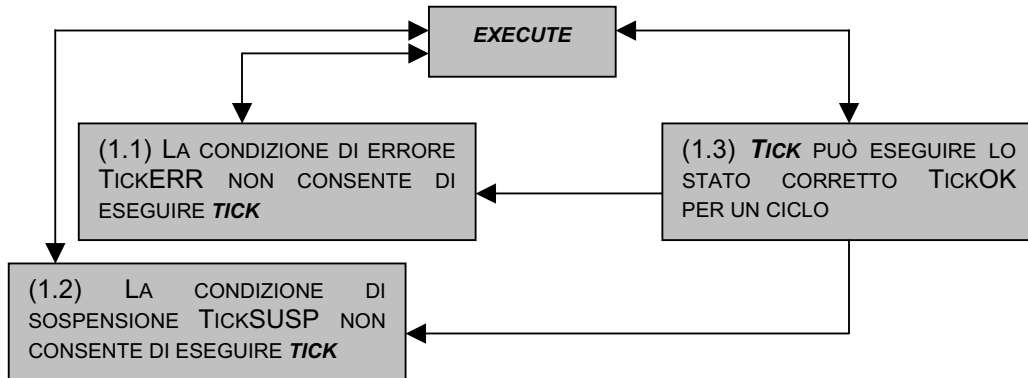


Figura 2.6.3a: Algoritmo della funzione *execute*

2.6.3 Ciclo di Esecuzione

L'esecuzione, obiettivo finale di questa formalizzazione, è implementata come funzione da stato a stato e presenta due principali caratteristiche:

- distingue tre diverse modalità di esecuzione *TickERR*, *TickSUSP* e *TickOK*;
- non esegue (come in altre formalizzazioni) le istruzioni in modo atomico, ma adotta come passo elementare di esecuzione il tick (un ciclo di clock).

La prima modalità di esecuzione *TickERR* serve a descrivere l'intero insieme di errori a run-time (e.g.: eccezione illegal address o illegal opcode, bug di formalizzazione) che possono verificarsi. La seconda modalità di esecuzione *TickSUSP* serve a descrivere l'intero insieme delle modalità di arresto (e.g.: run, wait) in cui la CPU può entrare.

Entrambe le modalità fanno riferimento ad un tipo stato che necessita di un intervento esterno dalla CPU per riprendere la corretta esecuzione (e.g.: intervento del COP per gestire l'eccezione, interrupt che risveglia la CPU, sviluppatore che rintraccia ed elimina il bug). Per questo motivo la funzione *execute* congela qualsiasi stato diverso da *TickOK*, facendo avanzare il contatore dei tick senza però richiamare la funzione *tick* (il ciclo fetch-decode-execute).

Qualsiasi teorema che faccia uso di *execute* può, in questo modo, richiedere l'esecuzione di un numero arbitrariamente grande di cicli senza preoccuparsi di una eventuale perdita di informazioni su un qualsiasi evento intermedio, di arresto o di errore, che abbia interrotto il normale processo di esecuzione.

La scelta del tick come passo elementare di esecuzione presenta, inoltre, due vantaggi:

- la possibilità immediata di associare, una volta stabilita la frequenza di esecuzione della CPU, anche un tempo reale all'esecuzione. Ciò può essere indispensabile per la formulazione di teoremi che riguardino una proprietà di non sfioramento di una specifica deadline temporale;
- la possibilità futura di estendere la formalizzazione a modelli con una architettura a pipeline, in cui l'esecuzione contemporanea di più istruzioni richiede necessariamente una granularità di esecuzione a livello di clock.

Il seguente esempio di implementazione di *execute* è anche tradotto, per una migliore comprensione, sotto forma di algoritmo (Figura 2.6.3a).

[Implementazione dell'esecuzione]

```
let rec EXECUTE (m:McuType) (t:MemoryImplementation)
    (s:tickResult (Status m t)) (n:nat) on n :=
match s with
(* (1.1) se la CPU è in condizione di errore ci resta *)
| TICKERR s' error => TICKERR ? s' error
(* (1.2) se la CPU è in modalità di sospensione ci resta *)
| TICKSUSP s' susp => TICKSUSP ? s' susp
(* (1.3) esecuzione corretta fino ad esaurimento dei cicli richiesti *)
| TICKOK s' => match n with
[ O => TICKOK ? s' | S n' => EXECUTE m t (TICK m t s') n' ]].
```

La funzione *tick*, richiamata da *execute*, suddivide il ciclo di fetch-decode-execute in due distinte fasi:

- fetch e decodifica (*fetch*);
- esecuzione (*tickExecute*).

La prima fase *fetch* incorpora l'intero meccanismo di decodifica trattato nel secondo livello e, insieme alle informazioni derivate dalla decodifica (pseudocodice, modalità di indirizzamento, cicli di esecuzione), anche l'avanzamento del program counter derivante dalle operazioni di lettura effettuate.

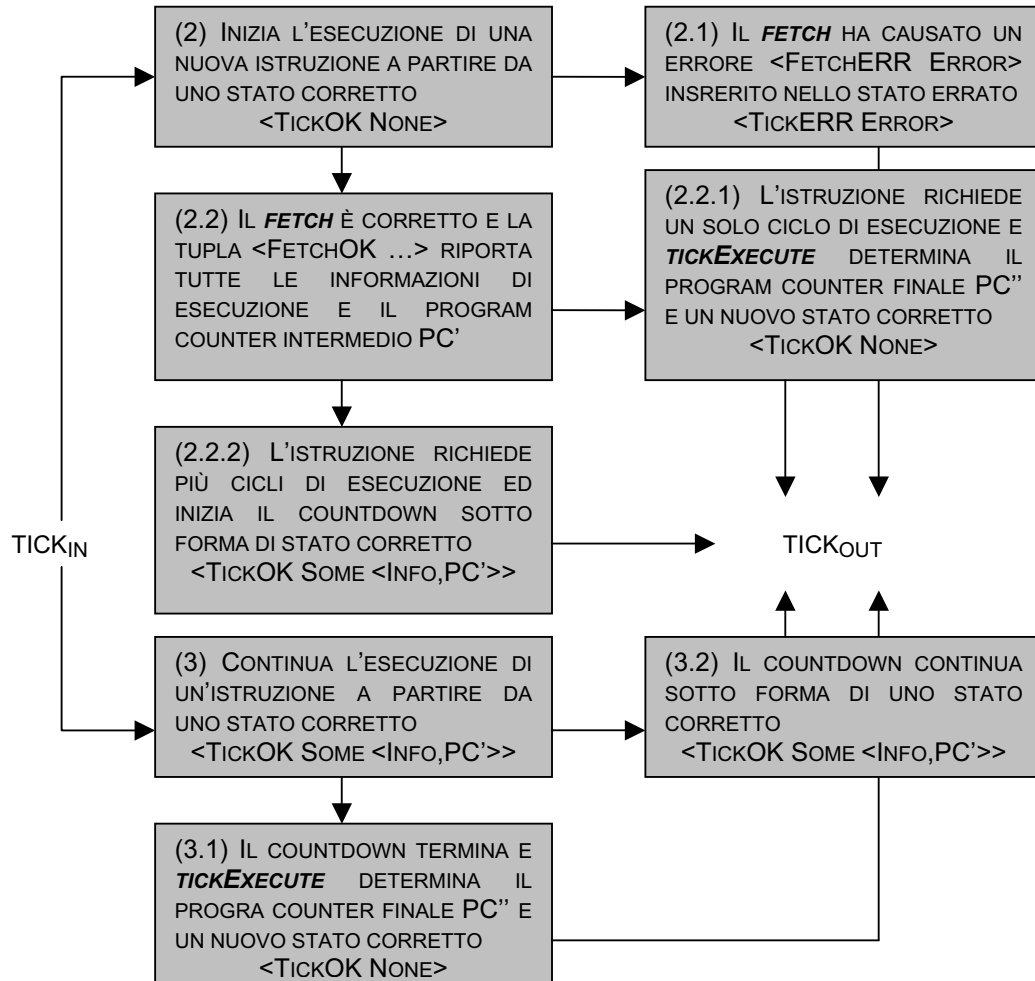


Figura 2.6.3b: Algoritmo della funzione *tick*

La seconda fase *tickExecute* incorpora l'intera unificazione delle logiche trattata nelle sezioni precedenti e, utilizzando l'avanzamento del program counter derivante dagli accessi in lettura e scrittura agli operandi (automatizzato dal loader e dal writer), completa il processo di fetch-decode-execute.

Dal punto di vista temporale, la prima fase corrisponde all'esecuzione del primo ciclo dell'istruzione e la seconda fase corrisponde all'esecuzione dell'ultimo ciclo dell'istruzione. La funzione *tick* infatti, pur esibendo verso l'esterno una granularità di esecuzione a livello di tick, mantiene internamente lo stesso stato fino all'esecuzione atomica allo scadere del countdown. L'esecuzione di un'istruzione che prevede più cicli, per esempio, comporta un fetch-decode

(al primo ciclo), seguito da un countdown intermedio privo di qualsiasi effetto sullo stato (cicli intermedi) e concluso dall'esecuzione (ultimo ciclo).

Il firmware interno all'unità di esecuzione delle CPU reali suddivide l'esecuzione in più passi (e.g.: caricamento del primo/secondo operando, scrittura del risultato) ma, in assenza di pipelining, si è preferito astrarre dal meccanismo di esecuzione reale ed escludere gli stati intermedi dalla formalizzazione.

Il seguente esempio di implementazione di *tick* è anche tradotto, per una migliore comprensione, sotto forma di algoritmo (Figura 2.6.3b).

[Implementazione dell'avanzamento del tick]

```

definition TICK :=
λm:McuType.λt:MemoryImplementation.λs:Status m t.
let optInfo := (getCLK_Desc m t s) in match optInfo with
  (* (2) e' il momento del fetch *)
  [ None => match (FETCH m t s) with
    (* (2.1) se ci sono errori nel fetch/decode non c'è avanzamento di esecuzione *)
    [ FETCHERR err => TICKERR ? s err
    (* (2.2) nessun errore nel fetch *)
    | FETCHOK fetchInfo curPC => match fetchInfo with
      [ Tupla4 pseudo mode _ totClk =>
        match (byteEq 0x01 totClk) with
          (* (2.2.1) manca un solo ciclo: execute è effettuata *)
          [ true => TICKEXECUTE m t s pseudo mode curPC
          (* (2.2.2) mancano piu' cicli: execute è rimandata *)
          | false => TICKOK ? (setCLK_Desc m t s
            (Some ? (Tupla5 ????? 0x01 pseudo mode totClk curPC))) ]]]
    (* (3) il fetch e' gia' stato eseguito: e' il turno di execute? *)
    | Some info => match info with [Tupla5 curClk pseudo mode totClk curPC =>
      match (byteEq (byteSucc curClk) totClk) with
        (* (3.1) si *)
        [ true => TICKEXECUTE m t s pseudo mode curPC
        (* (3.2) no, avanzamento di curClk *)
        | false => TICKOK ? (setCLK_Desc m t s
          (Some ? (Tupla5 ????? (byteSucc curClk) pseudo mode totClk curPC))) ]]].

```

MODULO	DEFINIZIONI	DIMOSTRAZIONI	TOTALE
EXTRA	99	61	160
EXADECIM	1.467	320	1.787
BYTE8	266	251	517
WORD16	198	66	264
LIVELLO 1	2.030	698	2.728
AUXBASES	82	0	82
OPCODE	442	0	442
TABLEHC05	348	27	375
TABLEHC08	445	63	508
TABLEHCS08	457	58	515
TABLERS08	366	23	389
TRANSLATION	186	0	186
LIVELLO 2	2.326	171	2.497
MEMORYSTRUCT	595	18	613
MEMORYFUNC	52	0	52
MEMORYTREES	192	28	220
MEMORYBITS	192	0	192
MEMORYABS	220	0	220
LIVELLO 3	1.251	46	1.297
STATUS	894	65	959
MODEL	602	0	602
LIVELLO 4	1.496	65	1.561
LOADWRITE	862	0	862
MULTIVM	1.280	20	1.300
LIVELLO 5	2.142	20	2.162
LIVELLI 1-5	9.245	1.000	10.245

Figura 2.6.3c: Dimensione (in righe di codice matita) della formalizzazione

Il capitolo si conclude con una breve considerazione sulla dimensione e composizione della formalizzazione realizzata (Figura 2.6.3c).

La dimensione dei cinque livelli è pressappoco equivalente, con una netta predominanza della componente di specifica (definizioni) sulla componente di dimostrazione (lemmi ed assiomi): la componente di specifica (il 90%), racchiude in sé tutte le informazioni necessarie al funzionamento della macchina virtuale *multivm*; la componente di dimostrazione (il 10%), invece, serve a verificare la

	Code (in Coq)	Code (in Caml)	Specifi- cations	Theorems	Proofs	Other	Total
Data structures, proof auxiliaries	268	70	-	514	933	489	2274
Integers, floats, memory model, values	221	18	851	1159	2872	449	5570
Cminor semantics	-	-	257	-	-	21	278
Instruction recognition, reassociation	693	-	-	186	462	99	1440
RTL semantics and type reconstruction	194	-	196	253	673	94	1410
RTL generation	279	-	-	858	1626	224	2987
Optimizations over RTL	1235	-	-	633	1447	325	3640
LTL semantics	-	-	354	61	151	56	622
Register allocation	800	521	-	1907	4897	375	8500
Linear semantics	-	-	238	20	34	48	340
Linearization	133	-	-	412	749	129	1423
Mach semantics	-	-	494	366	710	96	1666
Layout of activation records	116	-	-	469	987	156	1728
PPC semantics	9245	-	479	6	9	33	527
PPC generation	407	-	-	700	1705	127	2939
Compiler driver, Cminor parser, PPC printer	32	704	-	43	98	61	938
Total	4378	1313	2869	7587	17353	2782	36282

Figura 2.6.3d: Dimensione (in righe di codice) del progetto CompCert [Ler06]

la correttezza degli operatori realizzati e dei dati inseriti.

Da un confronto con il progetto CompCert, ristretto al livello dell'assembly del PowerPC Motorola (Figura 2.6.3d), si può notare che:

- la descrizione dei microcontroller Freescale, a causa della loro natura CISC, è molto più complessa della descrizione del PowerPC, di tipo RISC (9.245 righe contro 479 righe);
- la percentuale della componente di specifica, nel progetto CompCert, è identica (90%).

La preponderanza della componente di specifica può essere facilmente interpretata considerando che la formalizzazione realizzata non è un'entità a sé stante, ma costituisce la fase iniziale (sviluppo di una specifica completa funzionante) del neonato progetto Freescale che, nelle fasi successive, comporterà anche lo sviluppo di un opportuno impianto dimostrativo.

L'obiettivo iniziale (il completo supporto all'esecuzione), una volta raggiunto, ha comunque consentito la verifica di correttezza di frammenti reali di codice (vedi appendice) tramite semplice verifica dello stato finale ottenuto tramite computazione.

Appendice: Test Svolti



Figura A: USB Spyder08 [Fre07]

Per lo svolgimento dei due teoremi è stata adottata la seguente procedura:

- 1) compilazione con CodeWarrior (il compilatore C Freescale) di un breve frammento di codice che implementa l'algoritmo scelto;
- 2) estrazione del compilato assembler e delle locazioni di memoria corrispondenti alle variabili;
- 3) esecuzione passo passo (in-circuit debug) su un microcontroller reale per mezzo dello strumento di debug USB Spyder08 (Figura A);
- 4) rilevazione dello stato iniziale/finale e dei tempi di esecuzione del microcontroller reale;
- 5) formulazione di un test che comprenda il compilato assembler, lo stato e la memoria iniziali/finali e un upper bound o tight bound del tempo di esecuzione;
- 6) verifica (per particolari input) del test tramite esecuzione all'interno di Matita o (nel caso l'esecuzione ecceda la capacità di rappresentazione del sistema) all'esterno, ricorrendo all'estrazione di codice.

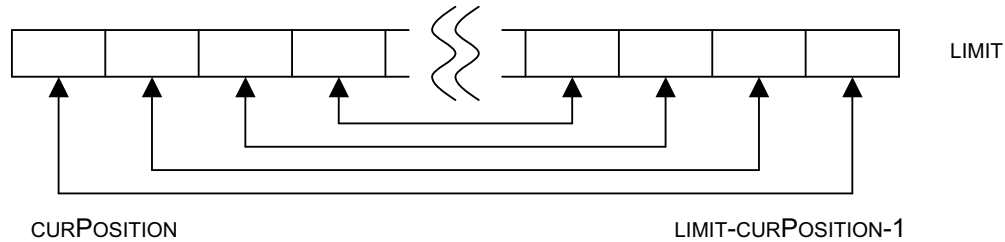


Figura A.1: Algoritmo *stringReverse*

A.1 Test stringReverse

Il test *stringReverse* verifica la correttezza di una funzione che inverte l'ordine degli elementi di un generico array di byte (Figura A.1). L'implementazione adottata è stata scelta in base ai seguenti criteri:

- ottenere un compilato assembler di dimensione minima;
- garantire un tempo di esecuzione lineare nella dimensione dell'array;
- testare un ampio insieme di pseudocodici, manipolazione dello stack e chiamata a subroutine comprese.

È stato quindi necessario, per rispettare i precedenti criteri, introdurre i seguenti vincoli sulla dimensione dell'array *data*:

- deve essere pari;
- deve avere dimensione *SIZE* inferiore a 3Kb, la dimensione massima allocabile staticamente in un microcontroller dotato di 4Kb di RAM.

[Implementazione dell'algoritmo stringReverse]

```
static unsigned char DATA[SIZE] = {...};           /* [1] */

void SWAP(unsigned char *a, unsigned char *b)      /* [2] */
{ unsigned char tmp = *a; *a = *b; *b = tmp; return; } /* [2] */

void STRINGREVERSE(void)
{ unsigned int curPosition = 0, limit = 0;
  for(limit = SIZE; curPosition < (limit/2); curPosition++) /* [3] */
  { SWAP(&DATA[curPosition], &DATA[limit-curPosition-1]); } /* [3] */
  return; }
```

Usando il compilatore CodeWarrior è stato ottenuto un compilato assembler di cui solo le sezioni [1-3] sono rilevanti per la verifica del test.

[1]		(0x18FC)	PSHA
(0x0100-0x0CFF) ...		(0x18FD)	LDX ,X
[2]		(0x18FE)	PULH
(0x18BE)	PSHX	(0x18FF)	AIX #-1
(0x18BF)	PSHH	(0x1901)	TXA
(0x18C0)	LDHX 5,SP	(0x1902)	ADD #0x00
(0x18C3)	LDA ,X	(0x1904)	PSHH
(0x18C4)	LDHX 1,SP	(0x1905)	TSX
(0x18C7)	PSHA	(0x1906)	STA 3,X
(0x18C8)	LDA ,X	(0x1908)	PULA
(0x18C9)	LDHX 6,SP	(0x1909)	ADC #0x01
(0x18CC)	STA ,X	(0x190B)	LDX 3,X
(0x18CD)	LDHX 2,SP	(0x190D)	PSHA
(0x18D0)	PULA	(0x190E)	PULH
(0x18D1)	STA ,X	(0x190F)	BSR *-81
(0x18D2)	AIS #2	(0x1911)	AIS #2
(0x18D4)	RTS	(0x1913)	TSX
[3]		(0x1914)	INC 2,X
(0x18E0)	LDHX # SIZE /* inizio */	(0x1916)	BNE *+4
(0x18E3)	STHX 4,SP	(0x1918)	INC 1,X
(0x18E6)	BRA *+52	(0x191A)	TSX
(0x18E8)	TSX	(0x191B)	LDA 3,X
(0x18E9)	LDA 2,X	(0x191D)	PSHA
(0x18EB)	ADD #0x00	(0x191E)	PULH
(0x18ED)	PSHA	(0x191F)	LSRA
(0x18EE)	LDA 1,X	(0x1920)	TSX
(0x18F0)	ADC #0x01	(0x1921)	LDX 4,X
(0x18F2)	PSHA	(0x1923)	RORX
(0x18F3)	LDA 4,X	(0x1924)	PSHA
(0x18F5)	SUB 2,X	(0x1925)	PULH
(0x18F7)	STA ,X	(0x1926)	CPHX 2,SP
(0x18F8)	LDA 3,X	(0x1929)	BHI *-65
(0x18FA)	SBC 1,X	(0x192B)	... /* fine */

Usando lo strumento di debug USB Spyder08 sono stati rilevati gli stati iniziali/finali e i tempi di esecuzione per un campione di array *data* di diverse dimensioni.

[Tabella dei registri e dei flag degli stati stringReverseStatus iniziale e finale]

STATO	A	H:X	SP	PC	CCR (V.H.N.Z.C)
INIZIALE	0x00	0x0D4B	0x0D4A	0x18E0	0.0.0.1.0
FINALE	0x00	SIZE/2	0x0D4A	0x192B	0.0.0.1.0

[Tabella dei cicli di esecuzione assoluti, relativi e parametrizzati rispetto a SIZE]

SIZE	CLOCK _I	CLOCK _F	#CLOCK	ORDINE DI GRANDEZZA
8	178.806	179.480	674	$674 = 42 + 5 * (8/512) + 79 * 8$
16	178.806	180.112	1.306	$1.306 = 42 + 5 * (16/512) + 79 * 16$
32	178.806	181.376	2.570	$2.570 = 42 + 5 * (32/512) + 79 * 32$
64	178.806	183.904	5.098	$5.098 = 42 + 5 * (64/512) + 79 * 64$
128	178.806	188.960	10.154	$10.154 = 42 + 5 * (128/512) + 79 * 128$
256	178.806	199.072	20.266	$20.266 = 42 + 5 * (256/512) + 79 * 256$
511	178.806	219.138	40.332	$40.332 = 42 + 5 * (511/512) + 79 * 511$
512	178.806	219.301	40.495	$40.495 = 42 + 5 * (512/512) + 79 * 512$
514	178.806	219.459	40.653	$40.653 = 42 + 5 * (514/512) + 79 * 514$
1.024	178.806	259.754	80.948	$80.948 = 42 + 5 * (1.024/512) + 79 * 1.024$

Dall'ultima colonna della tabella si ricava subito la formula che descrive il tight bound del tempo di esecuzione:

$$T(n) = 42 + 5 * (n/512) + 79 * n$$

Prima di formulare il test è necessario definire le seguenti funzioni ausiliarie per la manipolazione delle stringhe:

- *strlen*: *List Byte* → *Word*
restituisce la lunghezza di una stringa di byte sotto forma di word;
- *boundedStrlen*: *List Byte* → *Word* → *Boolean*
restituisce true se la lunghezza della stringa di byte è minore o uguale al parametro word fornito;
- *reverse*: *List Byte* → *List Byte*
restituisce una stringa di byte i cui elementi sono invertiti rispetto a quella fornita.

Test stringReverse:

L'esecuzione del sorgente C stringReverse

- *su un microcontroller con CPU HC(S)08*
- *restringendo l'analisi alle sole sezioni di codice [1-3]*
- *per un qualsiasi array DATA di dimensione pari ed inferiore ai 3Kb*
- *per una qualsiasi implementazione di memoria formalizzata*
- *definito $N = \text{strlen}(\text{DATA})$*

richiede un tempo lineare $T(N) = 42 + 5(N/512) + 79*N$ e ha come effetto collaterale (azzerando tutta la RAM tranne l'array DATA) la transizione dallo stato stringReverseStatus iniziale allo stato stringReverseStatus finale e la trasformazione del contenuto dell'array DATA descritta dalla funzione reverse.*

[Implementazione del test stringReverse]

lemma **STRINGREVERSE** :=

λ t:MemoryImplementation. λ string>List Byte.

(* (1) stringa deve avere una lunghezza ≤ 3.072 *)

(BOUNDEDSTRLEN string 0x0C00) \wedge

(* (2) string deve avere lunghezza pari *)

((andByte (LowerByte (STRLEN string)) 0x01) = 0x00) \wedge

(* (3) match di esecuzione con tight bound *)

(match EXECUTE HCS08 t

(TickOK ? (STRINGREVERSESTATUS t 0x0D4B (STRLEN string) string))

(* tight bound $42+5*(n/512)+79*n$ *)

($42+((\text{STRLEN string})/512)+ 79*(\text{STRLEN string})$) with

[TickERR s _ => None ?

(* azzeramento di tutta la RAM tranne data *)

| TickSUSP s _ => None ?

| TickOK s => Some ? (setMem HCS08 t s

(loadFromSource t (getMem HCS08 t s) ZEROBYTES 0x0D00))] =

Some ? (setPC_Reg HCS08 t

(STRINGREVERSESTATUS t (getFirst ?? (wordShr (STRLEN string)))

(STRLEN string) (REVERSE string)) 0x192B)).

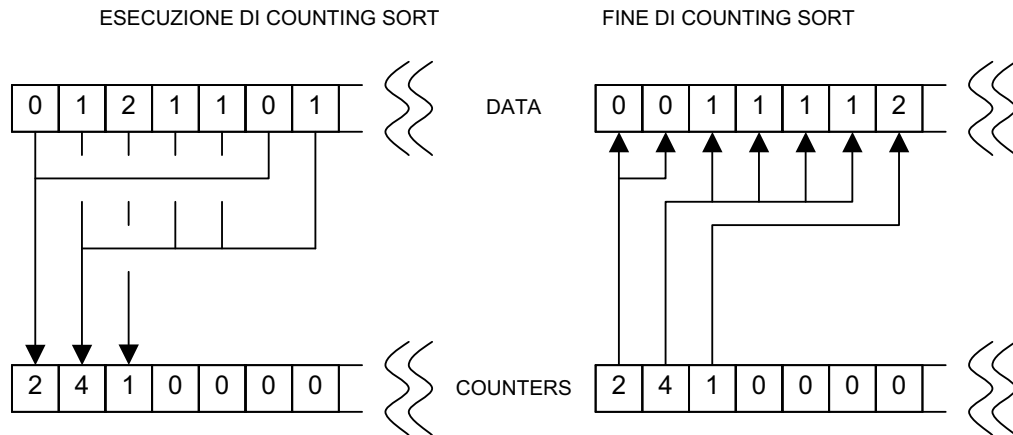


Figura A.2: Algoritmo *countingSort*

A.2 Test countingSort

Il test *countingSort* verifica la correttezza di una funzione che ordina gli elementi di un generico array di byte secondo l'algoritmo standard counting sort (Figura A.2). L'implementazione adottata è stata scelta in base ai medesimi criteri descritti per il precedente test e richiede lo stesso vincolo di dimensione dell'array *data* inferiore ai 3Kb.

[Implementazione dell'algoritmo countingSort]

```
static unsigned char DATA[SIZE] = { ... };          /* [1] */
static unsigned int counters[256] = { 0, ..., 0 }; /* [2] */
```

```
void COUNTINGSORT(void)
{ unsigned int index = 0;                          /* [3] */
  unsigned int curPosition = 0;                    /* [4] */
  for(; index < SIZE; index++)                      /* [5] */
  { counters[DATA[index]]++; }                     /* [5] */
  for(index = 0; index < 256; index++)             /* [6] */
  { while(counters[index]--)                       /* [6] */
    { DATA[curPosition++] = index; } }           /* [6] */
  return; }
```

Usando il compilatore CodeWarrior è stato ottenuto un compilato assembler di cui solo le sezioni [1-6] sono rilevanti per la verifica del test.

[1]		(0x18F8)	LDHX 3,SP	
(0x0100-0x0CFF) ...		(0x18FB)	STA 256,X	
[2]		(0x18FE)	AIX #1	
(0x0D00-0x0EFF) 0x0000, ...		(0x1900)	STHX 3,SP	
[3]		(0x1903)	TSX	
(0x0F4C-0x0F4D) 0x0000		(0x1904)	LDX 1,X	
[4]		(0x1906)	LSLX	
(0x0F4E-0x0F4F) 0x0000		(0x1907)	LDA 1,SP	
[5]		(0x190A)	ROLA	
(0x18C8)	BRA *+31	<i>/* inizio */</i>	(0x190B)	PSHA
(0x18CA)	LDHX 1,SP		(0x190C)	PULH
(0x18CD)	LDA 256,X		(0x190D)	PSHX
(0x18D0)	LSLA		(0x190E)	LDHX 3328,X
(0x18D1)	CLR X		(0x1912)	PSHX
(0x18D2)	ROLX		(0x1913)	PSHH
(0x18D3)	ADD #0x00		(0x1914)	AIX #-1
(0x18D5)	PSHA		(0x1916)	PSHH
(0x18D6)	TXA		(0x1917)	PSHA
(0x18D7)	ADC #0x0D		(0x1918)	PULH
(0x18D9)	PSHA		(0x1919)	PSHX
(0x18DA)	PULH		(0x191A)	LDX 5,SP
(0x18DB)	PULX		(0x191D)	PULA
(0x18DC)	INC 1,X		(0x191E)	STA 3329,X
(0x18DE)	BNE *+3		(0x1921)	PULA
(0x18E0)	INC ,X		(0x1922)	STA 3328,X
(0x18E1)	TSX		(0x1925)	PULH
(0x18E2)	INC 1,X		(0x1926)	PULX
(0x18E4)	BNE *+3		(x01927)	CPHX #0x0000
(0x18E6)	INC ,X		(0x192A)	PULH
(0x18E7)	LDHX 1,SP		(0x192B)	BNE *-54
(0x18EA)	CPHX #SIZE		(0x192D)	TSX
(0x18ED)	BCS *-35		(0x192E)	INC 1,X
[6]		(0x1930)	BNE *+3	
(0x18EF)	TSX	(0x1932)	INC ,X	
(0x18F0)	CLR 1,X	(0x1933)	LDHX 1,SP	
(0x18F2)	CLR ,X	(0x1936)	CPHX #0x0100	
(0x18F3)	BRA *+16	(0x1939)	BNE *-54	
(0x18F5)	TSX	(0x193B)	STOP	
(0x18F6)	LDA 1,X		<i>/* fine */</i>	

Usando lo strumento di debug USB Spyder08 sono stati rilevati gli stati iniziali/finali e i tempi di esecuzione per un campione di array *data* di diverse dimensioni.

[Tabella dei registri e dei flag degli stati countingSortStatus iniziale e finale]

STATO	A	H:X	SP	PC	CCR (V.H.N.Z.C)
INIZIALE	0x00	0x0F4C	0x0F4B	0x18C8	0.0.0.1.0
FINALE	0xFF	0x0100	0x0F4B	0x193B	0.0.0.1.0

[Tabella dei cicli di esecuzione assoluti, relativi e parametrizzati rispetto a SIZE]

SIZE	CLOCK _I	CLOCK _F	#CLOCK	ORDINE DI GRANDEZZA
0	183.941	209.557	25.616	$25.616 < 25.700 + 0*150$
8	183.941	210.794	26.853	$26.853 < 25.700 + 8*150$
16	183.941	212.018	28.077	$28.077 < 25.700 + 16*150$
32	183.941	214.466	30.525	$30.525 < 25.700 + 32*150$
64	183.941	219.362	35.421	$35.421 < 25.700 + 64*150$
128	183.941	229.154	45.213	$45.213 < 25.700 + 128*150$
256	183.941	248.742	64.801	$64.801 < 25.700 + 256*150$
512	183.941	287.914	103.973	$103.973 < 25.700 + 512*150$
1.024	183.941	366.258	182.317	$182.317 < 25.700 + 1.024*150$
2.048	183.941	522.946	339.005	$339.005 < 25.700 + 2.048*150$
3.072	183.941	679.634	495.693	$495.693 < 25.700 + 3.072*150$

Dall'ultima colonna della tabella si ricava subito la formula che descrive l'upper bound del tempo di esecuzione:

$$T(n) \leq 25.700 + 150*n$$

Per formalizzare il tempo di esecuzione limitato superiormente è stato necessario inserire alla fine della sezione [6] del compilato assembler l'istruzione assembler *STOP*. La differenza fra il tempo di esecuzione effettivo e l'upper bound, in questo modo, diventa irrilevante (ma quantificabile) al fine della computazione realizzata da parte della macchina virtuale.

Una volta entrata in modalità di sospensione, la CPU formalizzata non effettua nessuna transizione di stato e il tick avanza "a vuoto" consentendo, una volta raggiunto il limite temporale superiore, di verificare la correttezza dell'ultimo stato di esecuzione effettiva.

Test countingSort:

L'esecuzione del sorgente C countingSort

- *su un microcontroller con CPU HC(S)08*
- *restringendo l'analisi alle sole sezioni di codice [1-6]*
- *per un qualsiasi array DATA di dimensione inferiore ai 3Kb*
- *per una qualsiasi implementazione di memoria formalizzata*
- *definito $N = \text{strlen}(\text{DATA})$*

*richiede un tempo lineare $T(N) \leq 25.700 + 150 * N$ e ha come effetto collaterale (azzerando tutta la RAM tranne l'array DATA) la transizione dallo stato countingSortStatus iniziale allo stato (di sospensione) countingSortStatus finale e la trasformazione del contenuto dell'array DATA descritta dalla funzione order.*

[Implementazione del test countingSort]

lemma **COUNTINGSORT** :=

λ t:MemoryImplementation. λ string>List Byte.

(* (1) stringa deve avere una lunghezza ≤ 3.072 *)

(**BOUNDEDSTRLEN** string 0x0C00) \wedge

(* (2) match di esecuzione con upper bound *)

(match **EXECUTE** HCS08 t

(TickOK ? (**COUNTINGSORTSTATUS** t 0x00 0x0F4C (**STRLEN** string) string))

(* upper bound $25.700 + 150 * n$ *)

($25700 + 150 * (\text{STRLEN string})$) with

[TickERR s _ => None ?

(* azzeramento di tutta la RAM tranne data *)

| TickSUSP s _ => Some ? (setMem HCS08 t s (loadFromSource t (getMem HCS08 t s) **ZEROBYTES** 0x0D00))

| TickOK s => None ?

] =

Some ? (setPC_Reg HCS08 t

(**COUNTINGSORTSTATUS** t 0xFF 0x0100 (**STRLEN** string) (**ORDER** string) 0x193C)).

Bibliografia

- [AbrZel] MARSHALL D. ABRAMS AND MARVIN V. ZELKOWITZ, *Belief in Correctness*.
<http://www.cs.umd.edu/users/mvz/pub/correctness-belief.pdf>
- [Bea93] DEREK LEE BEATTY, *A Methodology for Formal Hardware Verification with Application to Microprocessors*, PhD thesis, Carnegie-Mellon University, School of Computer Science, 1993.
<http://citeseer.ist.psu.edu/beatty93methodology.html>
- [BCDG00] STEFANO BERARDI, MARIO COPPO, FERRUCCIO DAMIANI AND PAOLA GIANNINI, “Type-Based Useless-Code Elimination for Functional Programs”, PLI Workshop – SAIG’00, 2000.
<http://www.di.unito.it/~damiani/papers/saig00.html>
- [BerCas04] YVES BERTOT AND PIERRE CASTERAN, *Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer Verlag, 2004
- [BoyYu92] ROBERT S. BOYER AND YUAN YU, “A Formal Specification of Some User Mode Instructions for the Motorola 68020”, *Technical Report TR-9204*, Computer Sciences Department, University of Texas, 1992.
<http://citeseer.ist.psu.edu/boyer92formal.html>
- [BoyYu96] ROBERT S. BOYER AND YUAN YU, “Automated Proofs of Object Code for a Widely Used Microprocessor”, *Journal of the ACM*, vol. 43, n. 1, 1996.
<http://citeseer.ist.psu.edu/article/boyer96automated.html>

-
- [BKM96] BISHOP C. BROCK, MATT KAUFMANN AND J. STROTHER MOORE, “ACL2 Theorems about Commercial Microprocessors”, *Proceedings of Formal Methods in Computer-Aided Design*, 1996.
<http://citeseer.ist.psu.edu/brock96acl.html>
- [Bro07] PETER BROOKER, “Air Traffic Management Safety Challenges”, *2nd Institution of Engineering and Technology Systems Safety Conference*, 2007.
<http://dspace.lib.cranefield.ac.uk/handle/1826/1966>
- [Fox01] ANTHONY J. C. FOX, “A HOL Specification of the ARM Instruction Set Architecture”, *Technical Report*, University of Cambridge, Computer Laboratory, 2001.
<http://citeseer.ist.psu.edu/fox01hol.html>
- [Fox02] ANTHONY J. C. FOX, “Formal Verification of the ARM6 Micro-Architecture”, *Technical Report*, University of Cambridge, Computer Laboratory, 2002.
<http://citesser.ist.psu.edu/fox02formal.html>
- [Fre07] FREESCALE SEMICONDUCTOR, INC., 2007.
<http://www.freescale.com>
- [HunBro92] WARREN A. HUNT AND BISHOP C. BROCK, “A Formal HDL and its Use in the FM9001 Verification”, *Mechanized Reasoning and Hardware Design*, 1992.
<http://citeseer.ist.psu.edu/warren92formal.html>
- [Jac94] JONATHAN JACKY, *Safety-Critical Computing: Hazards, Practices, Standards and Regulation*, 1994.
<http://staff.washington.edu/join/pubs/safety-critical.html>
- [JCC97] LINE JAKUBIEC, SOLANGE COUPET-GRIMAL AND PAUL CURZON, “A Comparison of the Coq and HOL Proof Systems for Specifying Hardware”, *Theorem Proving in Higher Order Logics: Short Presentations*, 1997.
<http://citeseer.ist.psu.edu/jakubiec97comparison.html>
- [KauMoo97] MATT KAUFMANN AND J. STROTHER MOORE, “An Industrial Strength Theorem Prover for a Logic Based on Common Lisp”, *Transactions on Software Engineering*, vol. 23, n. 4, 1997.
<http://citeseer.ist.psu.edu/kaufmann97industrial.html>

-
- [Ler06] XAVIER LEROY, “Formal Certification of a Compiler Back-End, or Programming a Compiler with a Proof Assistant”, *Proceedings of the POPL 2006 Symposium*, 2006.
<http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf>
- [Mac91] DONALD MACKENZIE, “The Fangs of the VIPER”, *Nature*, vol. 352, n. 8 August 1991.
- [Mat08] MATITA INTERACTIVE THEOREM PROVER, *Documentation*.
<http://matita.cs.unibo.it/documentation.shtml>
- [RJE03] KAI RICHTER, MAREK JERSAK AND ROLF ERNST, “A Formal Approach to MpSoC Performance Verification”, *IEEE Computer*, vol. 36, n. 4, 2003.
<http://citeseer.ist.psu.edu/richter03formal.html>
- [Rus93] JOHN RUSHBY, “Formal Methods and the Certification of Critical Systems”, *Technical Report SRI-CSL-937*, Computer Science Laboratory, SRI International, 1993.
<http://citeseer.ist.psu.edu/article/rushby93formal.html>
- [Sub03] ARUN SUBBARAO, *Secure Operating Systems Becoming Commercial Necessity*, 2003.
<http://www.iappliancweb.com/story/oeg20031205s0037.htm>
- [Wik07] WIKIPEDIA THE FREE ENCYCLOPAEDIA, *Evaluation Assurance Level*, 2007.
http://en.wikipedia.org/wiki/Evaluation_Assurance_Level
- [Wik08] WIKIPEDIA THE FREE ENCYCLOPAEDIA, *DO-178B*, 2008.
<http://en.wikipedia.org/wiki/DO-178B>
- [Wil97] MATTHEW WILDING, “Robust Computer System Proofs in PVS”, *LFM’97: Fourth NASA Langley Formal Methods Workshop*, 1997.
<http://citeseer.ist.psu.edu/wilding97robust.html>