Politecnico di Torino

POLITECNICO DI TORINO

Corso di Laurea Magistrale in
INGEGNERIA ELETTRONICA

Tesi di Laurea Magistrale

CAPACITÀ DI VERIFICA CON AUTO-TEST PER UN MICROCONTROLLORE INTEGRATO TRAMITE UVM E FIRMWARE

Relatore:
Prof. Maurizio MARTINA

Candidato:
Lorenzo OTTINO

Correlatori:
Ing. Paola BALDRIGHI
Ing. Alessandro Giuliano LOCARDI
Ing. Laura MAESTRI

Anno Accademico 2022-2023

Abstract

I sensori basati sulla tecnologia *System-in-Package* (SiP) sono ampiamente diffusi nel mercato dei Sistemi Embedded ed è previsto che le loro applicazioni crescano ulteriormente con la pervasività della tecnologia nella società. Nel corso degli anni la complessità della progettazione dei Sensori MEMS è aumentata a seguito di una serie di fattori, quali lo *scaling* tecnologico, i requisiti di prestazione, la gestione di sistemi sempre più a basso consumo e l'integrazione di funzionalità diversificate. Le soluzioni integrate facenti uso dei Sensori MEMS comprendono lo sviluppo e l'interfacciamento fra parti analogiche e parti digitali, per le quali viene richiesto di assolvere le computazioni desiderate su una varietà di differenti sorgenti di informazione. Il *Front-End* analogico è responsabile di rilevazione, trasduzione e filtraggio delle grandezze fisiche in ingresso provenienti dai Sensori MEMS. I segnali acquisiti sono convertiti in codici digitali e opportunamente processati. I protocolli di comunicazione come l'I2C o l'SPI consentono la trasmissione dei dati tra interfacce e Microcontrollore. Quest'ultimo gestisce le informazioni servendosi di elementi di memoria e di un core, imposta i parametri di configurazione e genera i risultati richiesti dall'applicazione dell'utente.

Ogni singolo elemento di una architettura di tale complessità richiede una serie di controlli fondamentali, al fine di dimostrare la correttezza delle operazioni sotto differenti condizioni operative. I Team di Ingegneri devono affrontare le sfide della verifica sin dalle prime fasi del *Design Cycle*, partendo dalla descrizione RTL sino all'implementazione su Silicio. La verifica impatta in modo considerevole lo sviluppo del prodotto, arrivando a occupare il 70 % del tempo complessivo. Di conseguenza, la verifica è un aspetto cruciale per il *time-to-market* e per i guadagni di una azienda.

L'obiettivo di questo lavoro di tesi è quello di mostrare la fattibilità della verifica della corretta integrazione di un Microcontrollore all'interno di un ASIC per un Sensore MEMS di tipo SiP, dispiegando le tecniche e gli strumenti di verifica industriale disponibili allo stato dell'arte. Il lavoro è stato condotto presso l'azienda *STMicroelectronics*, all'interno del Team di *Digital Design* del Gruppo MEMS di Cornaredo (MI). Partendo dallo studio architetturale, è stato definito il flusso di verifica da perseguire.

La tesi si focalizza sulla Verifica Digitale tramite la creazione di Firmware di test funzionali in linguaggio C, implementando algoritmi di *self-checking*. L'intento è la verifica del corretto comportamento dell'IP core integrato nel progetto mentre si interfaccia con gli elementi di memoria, ossia la RAM dati e i banchi di registri. Ciascun test è gestito all'interno dell'ambiente UVM affinché questi possano risultare auto-consistenti e automatizzati.

I risultati ottenuti sono significativi, in quanto i test ideati non solo verificano le funzionalità del Microcontrollore associate a come il core si interfaccia con le risorse hardware, ma si sono anche dimostrati utili ai fini della portabilità per la fase di *Final Test* sul dispositivo fisico su Silicio. In questo modo si ha la garanzia di avere a disposizione un flusso di verifica digitale auto-consistente a partire dal livello di astrazione RTL sino alla produzione su Silicio prima del *tape-out*.

Ai miei nonni

Ringraziamenti

Ringrazio il mio Relatore, il Prof. Maurizio Martina, per avermi dato l'opportunità di svolgere la tesi sotto la sua supervisione e per la sua continua disponibilità.

Ringrazio l'Ing. Paola Baldrighi, l'Ing. Alessandro Giuliano Locardi e l'Ing. Laura Maestri per la professionalità e competenza con le quali mi hanno seguito durante lo svolgimento dell'attività di tesi. È stata una importante opportunità di crescita personale della quale sarò sempre grato nei loro confronti.
Ringrazio tutti i membri del Team di Digital Verification e Digital Design per avermi dato l'opportunità di lavorare su un dispositivo di loro competenza e per avermi reso consapevole dell'importanza del lavoro di squadra all'interno di progetti complessi.

Ringrazio tutte le persone che mi hanno supportato e che hanno reso questi anni di intenso studio più leggeri, dai miei amici di lunga data e il mio ex coinquilino, ai compagni di treno e i colleghi di laboratorio.

Ringrazio i miei genitori per avermi dato la possibilità di continuare a studiare e formarmi e per aver sempre creduto in me, a loro va il mio grazie più affettuoso.

Indice

Elenco Figure

Introduzione

In questo lavoro di tesi l'obiettivo è verificare attraverso metodologia UVM la corretta integrazione di un microcontrollore all'interno di un dispositivo MEMS. L'attività si è tenuta all'interno del team di progettazione digitale appartenente alla divisione *Analog, MEMS and Sensors* (AMS) della sede di Cornaredo (MI) dell'azienda *STMicroelectronics*.

Dato un design che integra un sensore MEMS con un ASIC elettronico comprensivo di parte analogica e digitale, lo scopo è la verifica *top level* della parte digitale dell'elettronica presente nell'ASIC. Ciò che occorre verificare è la corretta interazione tra gli elementi digitali del design, in particolare il corretto comportamento e interazione tra gli elementi di memoria e l'IP core del microcontrollore, controllandone la correttezza nell'interfacciamento e nella connettività.

Per adempiere allo scopo di questo lavoro di tesi è stata sfruttata la capacità del microcontrollore di eseguire algoritmi: sulla base delle esigenze del designer di riferimento e del *Product Engineer* è stato programmato opportunamente il core al fine di verificare al tempo stesso la corretta esecuzione del firmware e le funzionalità del microcontrollore integrato nel design, ottenendo delle routine adattabili anche per la fase di *Final Test*.

Di seguito si delineano gli argomenti trattati nei vari capitoli in cui si articola l'elaborato di tesi.

Nel **Capitolo 1** viene introdotto il dispositivo oggetto di studio. Si descrive ad alto livello il sensore, definendo le parti che concorrono a costituire il sistema. Si introducono i sensori MEMS, il front-end analogico, i convertitori analogico-digitali dei segnali in ingresso e le catene DSP di elaborazione degli stessi. Oltre alle interfacce e ai protocolli di trasmissione si analizza il microcontrollore integrato, il quale è il fulcro attorno a cui gravita il lavoro di tesi, costituito da un IP core già certificato che si interfaccia con un banco di registri, una memoria codice e una memoria dati. In particolare si pone l'attenzione sull'interfacciamento tra questi blocchi e sulle due modalità di accesso agli elementi di memoria, ossia tramite processore e tramite interfaccia I2C/SPI. Siccome la funzionalità dei protocolli di comunicazione con l'esterno è già stata validata durante le fasi di verifica precedenti l'integrazione del microcontrollore, l'attività di verifica si è concentrata sul *checking* della connettività e del comportamento degli elementi interni del microcontrollore. Il flusso di verifica perseguito è sviluppato in dettaglio nei capitoli successivi.

Nel **Capitolo 2** si descrive l'approccio alla verifica digitale mediante la metodologia UVM. Si introduce il paradigma del linguaggio e le sue caratteristiche, focalizzandosi sulle motivazioni che spingono un ingegnere a utilizzare questo standard. Viene presentato l'ambiente UVM dispiegato per la verifica del dispositivo, mostrando nel dettaglio la suddivisione in vari blocchi e sotto-blocchi che cooperano per eseguire i test sviluppati. Si descrivono le funzionalità di ciascun componente e le loro peculiarità. Viene esposto il flusso attraverso il quale appositi componenti programmano il dispositivo e ricevono i risultati per essere processati e ricavarne informazioni utili ai fini di controllo e di verifica.

Nel **Capitolo 3** viene trattata la prima parte dell'attività pratica. Vengono illustrati i test applicati sulla memoria dati del microcontrollore, ciascuno dei quali composto da una parte in ambiente UVM SystemVerilog e una parte implementata sotto forma di algoritmi in linguaggio C. Si motiva la creazione di test cosiddetti *self-checking*, come questi sono stati organizzati e quale è stato il processo di sviluppo degli stessi a partire dalla loro descrizione in pseudocodice e diagrammi di flusso. I test prevedono l'esecuzione di algoritmi compilati a partire da programmi scritti in linguaggio C per un processore progettato da *STMicroelectronics*, con architettura e *Instruction Set* di tipo RISC, il tutto gestito in ambiente UVM per l'esecuzione delle simulazioni e il *reporting* formattato dei risultati verificati. Per ogni test ideato, si menzionano i risultati attesi, si descrive nel dettaglio la loro esecuzione attraverso l'approccio simulativo e si riportano i risultati ottenuti, dimostrando il corretto interfacciamento tra il core e la memoria dati. In particolare, tra gli obiettivi della verifica digitale a livello RTL vi è stato quello di sviluppare dei test che siano portabili anche per la fase di *Final Test*. Questo risultato è stato rilevante in quanto ha permesso di sfruttare gli stessi algoritmi anche su Silicio, rendendo possibile una verifica auto-consistente con il progetto ad alto livello.

Nel **Capitolo 4** viene descritta la seconda parte dell'attività di tesi, riguardante i test dei banchi di registri. Si tratta dei registri interni del microcontrollore e dei registri di sistema. Operativamente, per ciascuna delle due porzioni del design viene implementato uno script Python per il *parsing* delle specifiche e vengono creati dei test UVM e dei test C. Anche in questo caso la metodologia utilizzata è quella di *self-checking*. A partire da un file di specifiche fornito dal designer di riferimento, si è proceduto con lo sviluppo di test che fossero in grado di verificare la correttezza dei requisiti di accesso a ciascun byte del banco di registri. In particolare, le specifiche fornite prevedono un insieme di combinazioni di policy di accesso che regolano l'interfacciamento tra il core e i vari registri, di conseguenza i test devono essere in grado di provare che le policy siano rispettate sotto ogni possibile condizione operativa delle istruzioni eseguite dal core. I banchi di registri racchiudono al loro interno un insieme variegato di registri, ciascuno avente finalità anche molto diverse. Vista la maggiore complessità dovuta a un ampio insieme di casi possibili da prendere in esame per la verifica, i test progettati sono resi il più possibile modulari, in modo da suddividere la complessità e trattando ciascun caso con un algoritmo che opera sul core dedicato a una modalità di accesso specifica. Dato un unico test UVM per ciascun banco di registri, sulla base dei parametri impostati si è

in grado di far eseguire al core un algoritmo di test associato ad una policy di accesso differente.
Anche in questo caso, alla descrizione di ciascun test corrisponde l'esposizione dettagliata dei risultati ottenuti, i quali permettono di determinare la presenza di indirizzi di memoria associati a dei byte il cui comportamento è disallineato rispetto alle specifiche fornite.

Nelle **Conclusioni** si mettono in luce gli obiettivi raggiunti in quanto a completezza della verifica funzionale ed efficacia dei test, si mettono in rilievo le competenze acquisite e si sottolinea la portabilità dei test ideati.

Capitolo 1
Descrizione del Dispositivo

1.1 Panoramica e Caratteristiche principali

Il dispositivo sul quale è incentrato il lavoro di tesi è un sensore MEMS progettato da *STMicroelectronics*, il cui scopo è la misura di grandezze fisiche. In questo modo il dispositivo è in grado di fornire dei dati utili al sistema nel quale è impiegato, sulla base di un tasso di acquisizione dei dati di misura assegnato. Il dispositivo comunica con l'esterno servendosi di appositi pin capaci di supportare, nel caso in esame, i protocolli I2C/SPI (vedasi **Appendice A**). I campi di utilizzo di questo dispositivo sono i più disparati e spaziano dall'uso in dispositivi mobili di varia natura come *joypad*, *smartphone* e *tablet*, a quello per apparecchiature indossabili o per la navigazione all'interno di stabili [1, 2].

I sensori presenti nel dispositivo sono sistemi MEMS realizzati direttamente sul Silicio mediante tecniche di microlavorazione meccanica dette *Micromachining* [2]. Il chip su cui sono integrati questi sensori è a sua volta integrato all'interno di un System-in-Package, nel quale il chip è disposto al di sopra di un ASIC (*Application Specific Integrated Circuit*) e si interfaccia con esso attraverso dei fili di *bonding*, come illustrato in **Figura 1.1**. L'ASIC è in grado di ricevere e processare i segnali elettrici provenienti dai sensori. Esso è costituito da diverse entità che possono essere riassunte come segue.

- La grandezza fisica che è stata rilevata dal sensore viene dapprima trasformata in un segnale tempo-continuo attraverso il front-end analogico [1, 2];
- il segnale ottenuto viene convertito da un blocco ADC in modo da essere rappresentato sotto forma di valori numerici [1, 2];
- il segnale digitale viene processato da una catena DSP che fornisce i dati filtrati a un blocco di interfaccia [1, 2];

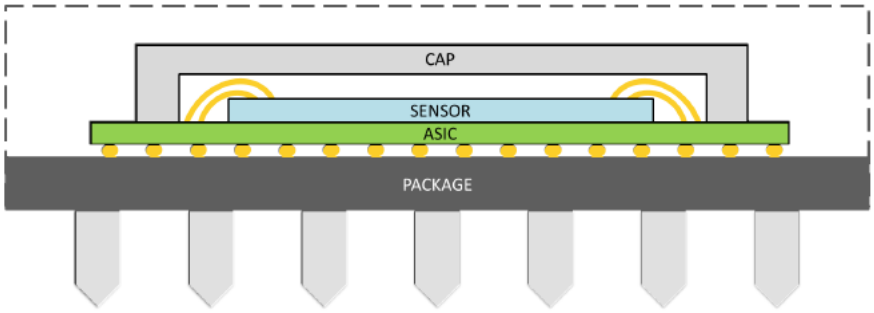


Figura 1.1: *Rappresentazione di un System-in-Package che integra un chip di sensori MEMS e un ASIC elettronico [1].*

- il microcontrollore, integrato all'interno dell'ASIC, comunica attraverso l'interfaccia seriale dalla quale è possibile programmare il microcontrollore stesso dall'esterno e leggere le informazioni di interesse agli indirizzi desiderati e accessibili, descritti nella sua *register map*. Internamente presenta un banco di registri, tra i quali figurano dei registri *General Purpose* di uscita, particolarmente utili ai fini della verifica, all'interno dei quali è possibile salvare valori di interesse per i test svolti, come descritto più avanti nell'elaborato. Esistono anche dei registri di controllo, dei registri di stato e un insieme di registri adibiti al salvataggio dei valori digitali in uscita dalle catene dei sensori, questi ultimi ricevuti come ingressi di dato dal microcontrollore. Il microcontrollore ha anche la possibilità di interfacciarsi con un banco di registri esterni, la maggior parte dei quali sono dedicati al salvataggio dei parametri di configurazione impostabili dall'interfaccia seriale, mentre alcuni di essi hanno lo scopo di abilitare il controllo dei *pad* esterni attraverso il core.

In **Figura 1.2** è raffigurato lo schema di massima dell'ASIC nel quale è integrato il microcontrollore oggetto della verifica digitale.

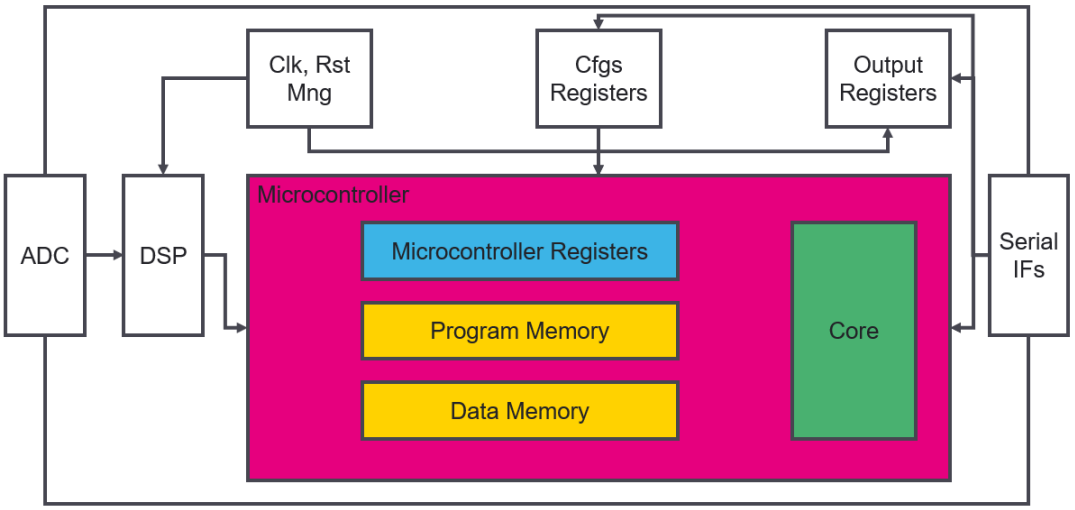


Figura 1.2: *Architettura della sezione digitale del progetto integrato. La parte evidenziata è quella oggetto di verifica.*

Nei paragrafi successivi si descrivono nel dettaglio le parti costituenti il dispositivo che sono state sino qui introdotte, suddividendo la trattazione in dominio meccanico, analogico e digitale, con particolare enfasi sul dominio digitale, in quanto gli sforzi profusi in questo lavoro di tesi si sono rivolti alla verifica della corretta integrazione del microcontrollore nel design digitale.

Nota
Il lavoro svolto in questa tesi è *device independent*, ciò significa che può essere trasferito su altri dispositivi aventi un microcontrollore da integrare al loro interno.
Pertanto, le descrizioni che seguono sono da considerarsi come un punto di riferimento e sono finalizzate a rappresentare un generico dispositivo MEMS in cui è integrato un microcontrollore.

1.2 Dominio meccanico

I sensori MEMS integrati su Silicio sono una tecnologia avanzata che sfrutta le proprietà elettromeccaniche del materiale di cui è composta. Il vantaggio di un uso su larga scala di tale tecnologia sta nel ridotto costo per unità che permette di rendere i costi di produzione competitivi sul mercato. Presenta inoltre delle dimensioni e un consumo di potenza dinamica ridotti se paragonati a quelli di un sensore standard [2].

Non essendo lo studio dei sensori MEMS in sé il fulcro del lavoro di tesi, basti la descrizione generale che segue sui principi di funzionamento di alcuni dei sensori in tecnologia MEMS più diffusi.

1.2.1 Accelerometro MEMS

L'accelerometro MEMS è un sistema massa-molla in grado di produrre una variazione capacitiva a seguito dell'applicazione di una forzante esterna. La misura dell'accelerazione è possibile a partire da tale variazione, che è dovuta a sua volta alla variazione della distanza tra le armature (elettrodi) integrate nel sistema elettromeccanico. Le armature solidali con il *bulk* in Silicio rimangono fisse, mentre quelle solidali con la massa meccanica si muovono con essa in caso di sollecitazione. Per ottenere variazioni rilevabili sono realizzate delle strutture modellizzabili come un insieme di condensatori in parallelo, ciascuno di dimensioni micrometriche. La struttura è realizzata in modo tale da rilevare le variazioni di capacità in modo differenziale. Così facendo si ha la cancellazione di offset sistematici e la reiezione dei disturbi dovuti a spostamenti ortogonali rispetto alla direzione presa come riferimento per la misura dell'accelerazione lineare [1, 2].

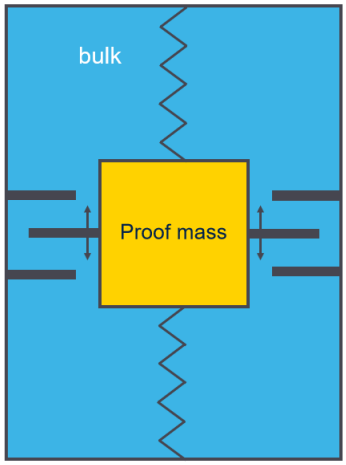


Figura 1.3: *Modello massa-molla dell'accelerometro MEMS con schema di massima delle armature fisse e mobili che provocano variazioni di capacità differenziali quando la massa è sollecitata.*

1.2.2 Giroscopio MEMS

Il giroscopio MEMS è costituito da un sistema massa-molla che misura l'accelerazione di *Coriolis*. Considerando l'applicazione di una forzante elettrostatica (generata da un sistema di attuazione capacitivo a partire da un segnale in tensione) lungo l'asse X, si genera un moto vibrazionale di attuazione che mette in rotazione la massa intorno all'asse Z ortogonale al piano di massa. Questo genera una forza apparente, detta forza di *Coriolis*, diretta lungo l'asse Y e proporzionale al modulo della velocità di rotazione. Tale forza induce uno spostamento dell'elemento meccanico lungo l'asse Y, il quale viene rilevato per mezzo di un sistema capacitivo che traduce lo spostamento in una variazione di capacità [1, 2].

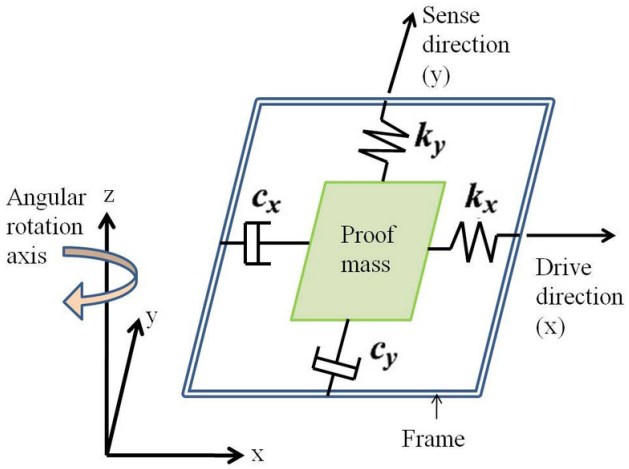


Figura 1.4: *Modello del giroscopio MEMS [3]. Il moto esterno di attuazione è diretto lungo X, mentre il moto risultante di trasduzione è diretto lungo Y; il moto rotazionale incognito è diretto lungo Z.*

1.2.3 Sensore di Pressione MEMS

Il sensore di pressione capacitivo MEMS è un sistema in grado di produrre una variazione capacitiva a seguito dell'applicazione di una pressione esterna, la quale determina una differenza rispetto a una pressione di riferimento. Nella sua concezione più semplice, il sensore è costituito da due superfici planari conduttive disposte in parallelo e separate da uno spazio vuoto o da un materiale dielettrico. I due piatti paralleli sono anche detti elettrodi, uno è metallico e solidale con il substrato rigido posto al di sotto di esso, l'altro invece è in Silicio ed è sensibile alle variazioni di pressione. Il comportamento di quest'ultimo elettrodo è analogo a quello di un diaframma: esso si deforma a causa della differenza di pressione che si instaura tra il mondo esterno e la cavità interna di riferimento. L'effetto che si ha è una variazione della distanza tra i due elettrodi, la quale provoca una variazione della capacità. Pertanto, la pressione esterna può essere misurata grazie al gradiente di capacità che si viene a creare [4].

Esistono anche sensori di pressione di tipo piezoresistivo, basati sulla variazione di resistenza del materiale sollecitato. Tuttavia, i sensori capacitivi sono più performanti rispetto ai piezoresistivi, dal momento che presentano una minore sensibilità alla temperatura, una sensibilità alla pressione più elevata e un consumo di potenza più basso [4].

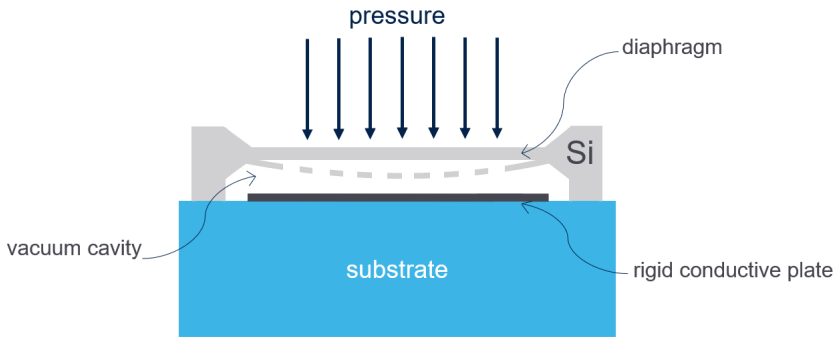


Figura 1.5: *Modello di base di un sensore di pressione capacitivo MEMS.*

I sensori MEMS possono essere utilizzati non solo singolarmente, ma anche in modo sinergico, sfruttando l'integrabilità all'interno di un unico package.

Una delle combinazioni che si possono trovare è quella che utilizza accelerometro e giroscopio: essa permette di misurare l'accelerazione lineare e la velocità angolare di un oggetto, al fine di determinarne la posizione nello spazio istante per istante. I dati provenienti da entrambi i sensori consentono di raggiungere una maggiore precisione rispetto all'uso del solo accelerometro.

Un'altra possibile applicazione è quella che sfrutta l'accelerometro assieme a un sensore di temperatura: il dato in temperatura in questo caso consente di compensare le non idealità degli elementi che compongono il sistema di trasduzione, introducendo degli offset all'interno del sistema di filtraggio dei segnali, cambiandone la risposta e quindi il valore del dato in uscita [5].

1.3 Front-End analogico, Front-End digitale e Interfacce di comunicazione

1.3.1 Analog Front-End

L'interfaccia analogica è responsabile della conversione della variazione di capacità in una variazione di tensione. Quando si ha una differenza di capacità non nulla, ossia quando il sensore è soggetto a una forzante esterna, è possibile rilevare la differenza di carica accumulata sulle capacità e convertirla in un segnale in tensione tramite un amplificatore operazionale con ingresso e uscita differenziali, configurato come integratore di carica [2]. La forzante esterna può essere modellizzata come un gradino di tensione sulla base delle proprietà elettromeccaniche del sensore, per cui ad una sollecitazione meccanica del materiale piezoelettrico corrisponde l'instaurazione di una differenza di potenziale nel materiale stesso. L'amplificatore fornisce in uscita una tensione proporzionale alla variazione di capacità dovuta al gradino di tensione che si crea al nodo in ingresso.

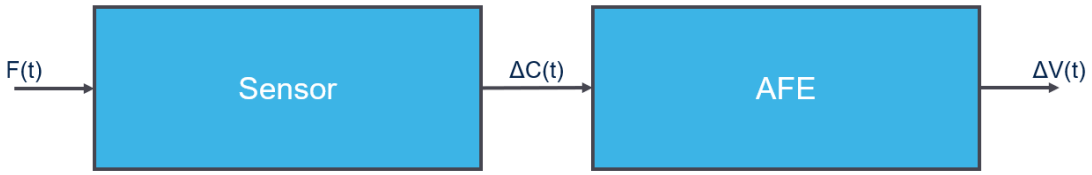


Figura 1.6: *Schema a blocchi della catena di trasduzione della grandezza fisica $F(t)$ rilevata dal sensore (descritto al **Paragrafo 1.2**). L'Analog Front-End è in grado di generare una corrispondente variazione di tensione continua nel tempo grazie a uno stadio amplificatore fully-differential con uscita in tensione.*

1.3.2 Convertitori ADC e Catene DSP

I dati in tensione provenienti dall'*Analog Front-End* costituiscono gli ingressi del sistema di acquisizione integrato nel dispositivo. Il sistema è stato progettato in modo tale da poter gestire l'acquisizione e il processamento di tutte le informazioni in ingresso tramite l'utilizzo di opportune risorse hardware: esistono infatti diverse catene DSP, ognuna di esse adibita rispettivamente al processamento dei dati provenienti da uno specifico sensore. I segnali acquisiti sono instradati verso un blocco di conversione analogico-digitale, grazie al quale è possibile convertire l'informazione contenuta nei segnali di tempo-continuo in un insieme di livelli di tensione discreti. In questo modo le elaborazioni successive possono avvenire interamente nel dominio digitale, come mostrato in **Figura 1.7**.
Considerando ad esempio dei sensori inerziali, essi possono fornire informazioni lungo

più assi simultaneamente, di conseguenza i dati che occorre raccogliere devono essere sincronizzati e gestiti in funzione di come sono dispiegate le risorse hardware. Nel caso di un solo ADC disponibile, è necessario un meccanismo di *time interleaving* per l'acquisizione nel tempo dell'informazione proveniente da ciascun asse, effettuata su canali differenti del medesimo ADC. Invece nel caso di più moduli ADC è possibile far operare più catene in parallelo, ciascuna avente a monte il proprio convertitore analogico-digitale, separando sin dall'inizio i vari *stream* di dati.

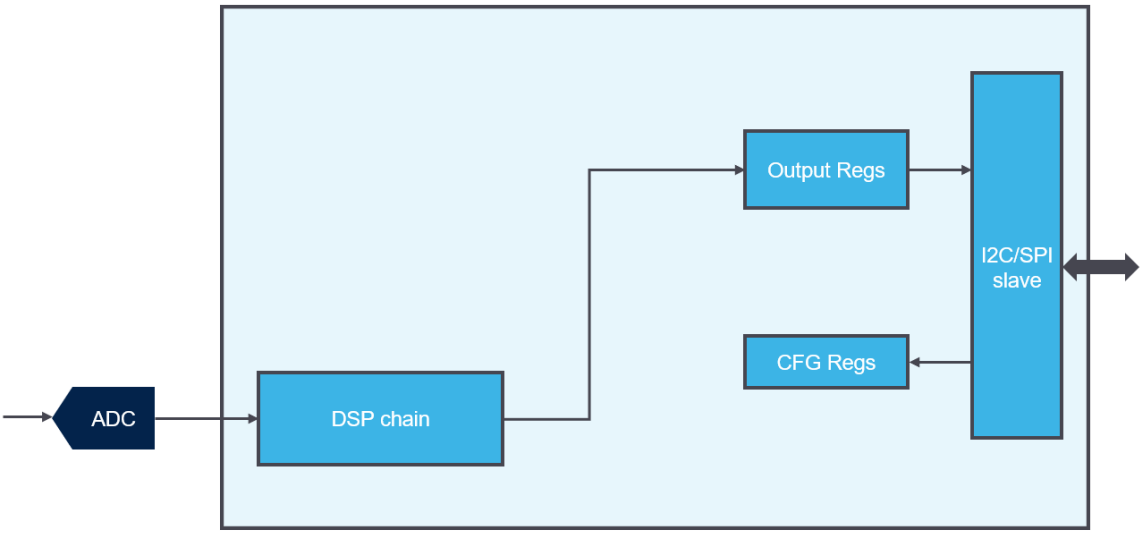


Figura 1.7: *Digital Front-End di un dispositivo MEMS.*

I dati digitali, rappresentati all'ingresso del *Digital Front-End* secondo la dinamica di uscita dell'ADC, vengono elaborati attraverso una catena di filtri digitali, i quali applicano una serie di operazioni di filtraggio e medie sulla base delle caratteristiche del sensore e delle modalità operative [2]. L'informazione elaborata è un dato digitale rappresentato su N bit, i quali vengono salvati in appositi registri di uscita (indicati nello schema a blocchi come *Output Regs*) [1]. Da questi registri è possibile leggere mediante interfaccia I2C/SPI. Tale operazione è opportunamente temporizzata da un segnale di *data ready*, il quale avvisa dell'avvenuta elaborazione di un dato ogni volta in cui esso viene asserito [2]. I registri di configurazione (*CFG Regs*) invece sono accessibili tramite interfaccia e possono essere programmati per definire i parametri di funzionamento del blocco digitale a seconda delle esigenze e delle necessità applicative.

Le catene di elaborazione digitale possono includere un livello di complessità tale da potersi adattare a diverse condizioni di funzionamento. Ad esempio, se l'utente desidera le massime prestazioni, le catene rimangono attive continuamente per poter elaborare i nuovi campioni in ingresso. Differentemente, nel caso in cui l'applicazione richiedesse una minimizzazione dei consumi di potenza del dispositivo, le catene possono operare in base a dei cicli di attivazione e spegnimento, limitando al tempo stesso la frequenza massima selezionabile per l'aggiornamento del dato.

I segnali campionati in uscita variano con un determinato *Output Data Rate*, parametro configurabile dall'utente in base alle esigenze di velocità di acquisizione e adattabile per il corretto rilevamento di quantità variabili in modo più lento o più rapido.
È da mettere in evidenza che nelle modalità a basso consumo l'utilizzo di corrente si riduce in funzione dell'ODR, in quanto il dispositivo MEMS è progettato per soddisfare vincoli stringenti di ottimizzazione del consumo di potenza. Questo aspetto è di fondamentale importanza nel mondo dei dispositivi mobili a basso consumo: a spese di un aumento di complessità del design per implementare diverse modalità di funzionamento, è possibile beneficiare di migliori risultati ottenibili in termini di flessibilità e riduzione dei consumi. Esiste un intervallo discretizzato di ODR impostabili che spazia da un insieme di valori minimi nell'ordine di decine di hertz, validi solamente per la modalità a basso consumo, a un insieme di valori massimi nell'ordine di qualche migliaia di hertz, utilizzabili solamente quando è necessario massimizzare le prestazioni. I valori dell'intervallo sono derivati agevolmente a partire da un *prescaler* di valore pari a una potenza di due.

1.3.3 Interfacce I2C e SPI

Le interfacce di comunicazione del dispositivo soddisfano i seguenti requisiti:

- ottimizzazione del numero di bus necessari alla trasmissione;
- compromesso tra complessità del protocollo e velocità di trasmissione delle informazioni.

Avere a disposizione due protocolli consente maggiore flessibilità nella scelta della modalità di trasferimento dei dati. In particolare il protocollo I2C fornisce prestazioni inferiori rispetto all'SPI, ed è caratterizzato da velocità di trasferimento che spaziano da 100 kbps in *Standard Mode* a 5 Mbps in *Ultra Fast Mode* [6]. Il protocollo SPI permette di raggiungere velocità superiori, che nel caso del dispositivo in esame sono compatibili con la massima frequenza di clock. La trasmissione tramite interfaccia I2C prevede l'uso di due bus, invece quella SPI avviene su tre oppure su quattro fili.

Per la descrizione dettagliata dei due protocolli si rimanda all'**Appendice A**.

1.4 Architettura del Microcontrollore integrato

La parte che segue fornisce una presentazione degli elementi del design su cui si è focalizzato il lavoro di tesi, come già affermato nella esposizione degli obiettivi a inizio elaborato. Il microcontrollore integrato nel design è un sistema dotato di un core *General Purpose* avente come target applicazioni in cui occorre soddisfare vincoli stringenti di area e di potenza. Al fine di generare il codice eseguibile con istruzioni aventi formato comprensibile dal core, esiste una *Toolchain* apposita fornita assieme alla *Intellectual Property* del core. In questo modo colui il quale è incaricato di creare programmi *custom* è in grado di compilare e caricare codice eseguibile, con l'unico vincolo di dover produrre un codice compilato di dimensione inferiore a una lunghezza massima data dalle dimensioni della memoria codice.
Il microcontrollore integra una memoria dedicata allo *storage* delle istruzioni del programma eseguibile, ossia la *Program Memory*, la quale occupa uno spazio pari a *DIM_PM* kB, e una memoria per il salvataggio dei dati, ossia la *Data Memory*, avente una dimensione di *DIM_DM* kB. Quest'ultima memoria può contenere i dati necessari alla corretta esecuzione del programma caricato, come ad esempio le variabili globali definite dall'utente.
Inoltre esistono dei registri detti *Microcontroller Registers*, entro i quali sono identificabili dei registri *General Purpose* per la memorizzazione di valori di uscita prodotti dal programma, una serie di registri di configurazione e di stato e i registri per il caricamento dei dati di ingresso provenienti dalle catene di elaborazione delle uscite dei sensori.
Il microcontrollore si interfaccia al resto del dispositivo tramite dei registri detti *System Registers* in cui sono presenti informazioni sulle configurazioni delle risorse esterne al design digitale del microcontrollore.
Gli elementi di memoria sono mappati all'interno di una *memory map*, la quale è fornita come file di specifiche del dispositivo sotto forma di un file Excel. Il file è organizzato il pagine, a ognuna delle quali ci si può riferire a seconda delle operazioni che si desidera effettuare. L'entità elementare di memoria è il singolo byte, a ogni byte corrisponde un indirizzo fisico in memoria.
In generale, ciascuna entità elementare di memoria è identificabile attraverso due caratteristiche:

- la posizione all'interno dello spazio di indirizzamento del core, ossia un *address* fisico univoco al quale ci si può riferire a partire da un offset (indirizzo relativo a una *page*), oppure per intero (indirizzo riferito a tutto lo spazio di indirizzamento);

- la modalità di accesso (lettura/scrittura), la quale può differire se si accede tramite interfaccia o tramite core. Esistono infatti dei registri che possono essere sia letti sia scritti da interfaccia ma sono solamente leggibili dal core; oppure dei registri a cui si può accedere solo da interfaccia e risultano invece "invisibili" al core, eccetera.

Ai fini del lavoro di verifica è stato necessario tenere conto della policy di accesso fornita dalle specifiche, in particolare si è fatto riferimento alle policy di accesso previste per il core, in quanto il corretto funzionamento dell'interfaccia è già stato verificato e pertanto

non è stato oggetto del lavoro di tesi. Sostanzialmente si vuole integrare nel design un IP core e degli elementi di memoria che si desidera mostrino le funzionalità richieste come ci si attende, cercando di identificare possibili bachi del design a seguito dell'esecuzione dei programmi, come scritture/letture errate e violazioni delle policy di accesso.

Di seguito si riportano le partizioni interne (*pages*) degli indirizzi attraverso le quali il core può interfacciarsi con gli elementi di memoria sopra introdotti. La *memory map* è servita per definire i *boundaries* d'azione per i test che sono stati creati al fine di raggiungere l'obiettivo del lavoro di tesi.

- La *Data Memory* ha una dimensione di *DIM_DM* kB ed è mappata agli indirizzi `[0x00000 ÷ DM_end]`; per i test di verifica prodotti si rimanda al **Capitolo 3**.

- I *Microcontroller Registers* occupano *DIM_UC_REGS* byte e si trovano agli indirizzi `[START_uc_regs ÷ END_uc_regs]`; per la verifica di questo banco di registri si veda il **Capitolo 4**.

- I *System Registers* occupano uno spazio di *DIM_UC_REGS* byte e sono mappati agli indirizzi `[START_sys_regs ÷ END_sys_regs]`; per la verifica della corretta interazione del processore con questa porzione del design si faccia riferimento al **Capitolo 4**.

- La *Program Memory* è definita su *DIM_PM* kB ed è mappata nell'intervallo `[PM_start ÷ PM_end]`. Per la sua verifica è bastato osservare che i programmi progettati venissero eseguiti come atteso nei vari test prodotti, a dimostrazione del fatto che il codice delle istruzioni da eseguire viene caricato correttamente a partire dalla memoria codice verso il core. In questo modo è stato possibile dedurre che l'interfacciamento tra lo stadio di *Instruction Fetch* del core e l'accesso in lettura alla memoria avviene correttamente. Tuttavia, siccome per i programmi caricati non è dato sapere in fase di sviluppo quale dimensione essi possano avere dopo la compilazione, non è stato possibile testare tramite core tutte le locazioni disponibili. Per avere un test esaustivo analogo ai precedenti e che fosse portabile al variare di DIM_PM, il core sarebbe dovuto essere abilitato alla scrittura nella memoria codice, cosa proibita *by design* per evitare di sovrascrivere il codice eseguibile caricato da interfaccia in fase di configurazione del microcontrollore. Di conseguenza, il test delle locazioni della memoria codice non occupate dall'eseguibile può solamente essere effettuato tramite scritture da interfaccia. Tale modalità di test esula dallo scopo di questa tesi, in quanto si vuole validare la connettività del core con gli elementi di memoria con l'esecuzione di codice di *self-checking* (**Paragrafo 3.1**); basti quindi per questa sezione il corretto interfacciamento in lettura tra il core e le istruzioni caricate verificato durante l'esecuzione del firmware dei vari test.

Il microcontrollore è in grado di eseguire sino a TOT_ALGOS algoritmi *General Purpose*, ciascuno dei quali ha un *prescaler* impostabile per configurare l'*execution rate* desiderato. Esiste un *mini OS* che viene caricato durante la fase di inizializzazione ed è in grado di gestire in modo automatizzato le configurazioni di abilitazione degli algoritmi, la loro

velocità di esecuzione e le rispettive funzioni di inizializzazione degli algoritmi, le quali vengono invocate automaticamente nel caso in cui fossero abilitati gli algoritmi a cui esse fanno riferimento.
Avere a disposizione una porzione di codice a sé che viene caricata indipendentemente da quello che desidera caricare l'utilizzatore ha una serie di vantaggi in termini di affidabilità e uniformità d'uso: l'utente, che in questo caso è l'ingegnere verificatore, si deve solo preoccupare di quali algoritmi abilitare tramite interfaccia, concentrando gli sforzi sullo sviluppo dei singoli algoritmi che desidera implementare.

Essenzialmente il flusso che porta all'esecuzione di codice nel core del microcontrollore si articola nei seguenti passaggi.

- Il programma scritto dall'utente e compilato dalla *Toolchain* viene caricato nella *Program Memory* tramite operazioni di scrittura da interfaccia seriale.
- Viene acceso il clock del core al fine di avviare il caricamento del *mini OS* e gestire le fasi preliminari di inizializzazione sopra descritte, che comprendono anche l'allocazione delle risorse della memoria dati. Questa fase è altresì detta fase di *boot* del microcontrollore, terminata la quale il core viene rimesso in *sleep mode*.
- L'esecuzione degli algoritmi che sono stati abilitati può avvenire solamente a condizione che il sensore sia acceso e a seguito della generazione di un dato valido. Questo evento scatena un segnale di *interrupt*, il quale funge da *trigger* per il microcontrollore, il clock si riattiva ed è possibile avviare l'esecuzione del programma caricato dall'utente.

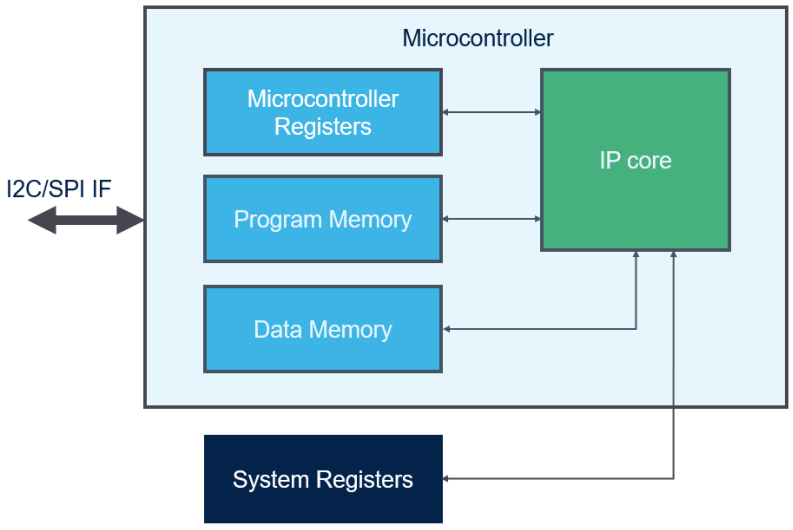


Figura 1.8: *Diagramma a blocchi del microcontrollore oggetto dei test di verifica.*

Come indicato in **Figura 1.8**, le opzioni di accesso possibili agli indirizzi di memoria sono due e sono spiegate qui sotto.

- Lettura/scrittura tramite INTERFACCIA (*User Interface*).
  Questa modalità consente di caricare dall'esterno il codice compilato *off-line* all'interno della memoria codice, di impostare gli ODR e i registri di configurazione, di leggere i risultati prodotti dall'esecuzione degli algoritmi quali informazioni di variabili di interesse trasmesse in uscita tramite i registri *General Purpose* oppure i *flag* di stato e i segnali di *interrupt* che codificano particolari condizioni verificatesi durante l'esecuzione del programma, le quali potranno quindi essere adeguatamente decodificate e controllate.
  In questa tesi, l'interfaccia viene usata per gestire l'esecuzione di test dinamici mediante approccio simulativo in ambiente UVM, per cui occorre effettuare le scritture desiderate per programmare il microcontrollore e le letture dei dati che si desidera controllare tramite *post-processing*, ossia una volta che gli algoritmi sono completati e il clock del microcontrollore si disattiva. Dopo le letture dei risultati vengono fatte delle scritture finali da interfaccia al fine di spegnere il dispositivo e di terminare la simulazione.
  L'ipotesi di partenza di un uso corretto dell'interfaccia è quella secondo la quale l'interfaccia con protocollo I2C/SPI funziona correttamente a priori. Infatti essa è già stata validata in precedenza ed è stata sfruttata per gestire le simulazioni funzionali in ambiente UVM nell'attività di verifica svolta.

- Lettura/scrittura tramite CORE.
  Questa modalità è stata il fulcro del lavoro di verifica svolto in questa tesi. L'uso di questo approccio consente di verificare la funzionalità del design tramite *self-checking*, per cui il core, eseguendo gli algoritmi di test, è in grado di verificare il comportamento del core stesso quando deve accedere alle risorse hardware di memoria. L'obiettivo della tesi è infatti quello di verificare la corretta integrazione del design del micocontrollore, di conseguenza i test creati si sono focalizzati sulle letture e scritture attraverso il core per riferirsi alle porzioni di memoria mostrate nel diagramma. Per ulteriori dettagli si rimanda ai capitoli dedicati alla descrizione dei test.

Nei prossimi paragrafi si analizzano in dettaglio gli elementi che compongono il design del microcontrollore integrato nel progetto digitale del dispositivo. In particolare per ogni entità si evidenziano le funzionalità di maggiore rilievo e le modalità di interazione possibili.

1.4.1 IP core

Il processore è stato progettato da un altro team di *STMicroelectronics* ed è un sistema il cui funzionamento interno è già stato verificato e certificato, di conseguenza non necessita di ulteriori test per accertare la corretta esecuzione delle istruzioni macchina dei programmi *custom* che si desidera caricare.
Si tratta di un IP core *General Purpose* e appartiene a una famiglia di processori usati in soluzioni proprietarie per applicazioni *DSP-oriented*. La versione integrata nel design oggetto di studio è quella base ed è in grado di soddisfare vincoli stringenti di consumo di potenza e area. L'architettura del processore è a 32 bit e supporta una dimensione dati variabile da un minimo di 1 byte a un massimo di 4 byte. Una caratteristica importante è la lunghezza variabile delle istruzioni: l'*Instruction Set* del processore permette di raggiungere un'alta densità del codice eseguibile, ottimizzando lo spazio occupato nella memoria codice. Le istruzioni più lunghe includono già al loro interno dei valori costanti, detti *immediate*, sui quali l'istruzione deve agire per compiere le operazioni richieste, evitando di caricarli dalla memoria dati e di conseguenza riducendo la latenza.
L'architettura è composta da vari stadi di *pipeline* ed è di tipo *Harvard*, caratteristica che permette concorrenzialità nell'accesso ai dati e nel *fetching* delle istruzioni.

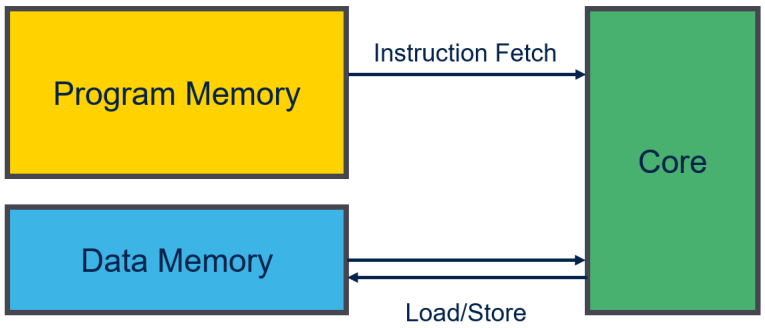


Figura 1.9: *Rappresentazione generale di una architettura di tipo Harvard.*

Il sistema presenta dei percorsi separati con i quali il core accede ai dati e alle istruzioni, caratteristica che migliora le prestazioni in fase di esecuzione del programma.
Il core, essendo di tipo RISC, lavora su un *Instruction Set* ridotto, costituito da un insieme di istruzioni atomiche semplici. Il formato delle istruzioni è standardizzato, cosa che semplifica lo stadio di decodifica. Il fatto che si abbia a disposizione un insieme semplificato di istruzioni rende il design interno nel suo complesso più lineare.
Infine, l'indirizzamento dei dati in memoria è basato su due singole istruzioni, la `LOAD` e la `STORE`, le quali semplificano il metodo di accesso.

Siccome il core è già stato certificato, si omette la rappresentazione dell'architettura interna dettagliata, in quanto lo studio del suo funzionamento esula dagli scopi di questa tesi. Basti riferirsi a esso come una *black box* capace di caricare dalla memoria codice

le istruzioni con la temporizzazione desiderata, di eseguire le computazioni richieste da tutte le istruzioni definite nell'*Instruction Set Architecture* (ISA), di salvare i risultati ottenuti nel *Register File* interno al core nel caso di variabili temporanee, oppure di scrivere tali valori verso l'esterno, da *flag* di stato per la generazione di segnali *interrupt* a dati da riportare in RAM o nei registri di uscita *General Purpose*.

Software Development Kit

Assieme all'IP core e al suo *Architecture Manual*, lo sviluppatore ha a disposizione un *Software Development Kit* (SDK) progettato appositamente per i microprocessori della famiglia alla quale il core appartiene.
Si tratta di un package software che racchiude al suo interno un insieme di *utilities* per lo sviluppatore. In particolare è messa a disposizione una *Toolchain* e un insieme di librerie di sistema.

La *Toolchain* è basata sulla *GNU Compiler Collection* (GCC) ed è costituita dal seguente *stack* software: il Compilatore, l'Assemblatore, il Linker e il Debugger. Essa è in grado di operare su macchine a 32-bit basate su un sistema Linux e produce un codice eseguibile per una piattaforma *target*, nello specifico il processore integrato nel microcontrollore avente architettura di tipo RISC. Il meccanismo con il quale il codice viene generato è detto *cross-compiling*.

Nel SDK sono integrate un insieme di librerie *runtime*, tra cui la *Standard C Library* (`libc.a`), le librerie di supporto per le computazioni aritmetiche, etc. In fase di *Linking* viene integrato nell'eseguibile finale solamente il codice binario delle funzioni di libreria che sono utilizzate dagli algoritmi sviluppati.
Per la spiegazione del processo di compilazione si rimanda al **Paragrafo 3.3.3**.

Va menzionato il fatto che il SDK supporta lo sviluppatore firmware anche nei vari livelli del design flow *target*: terminata la fase di verifica del design a livello RTL, in fase di prototipazione è possibile sfruttare, tra gli strumenti presenti, anche il Debugger. Una scheda di prototipazione infatti include una serie di funzionalità aggiuntive oltre alla disponibilità del chip su Silicio, quali una serie di periferiche e una interfaccia dedicata al *debugging*, le quali possono essere usate in modo sinergico per rimuovere eventuali errori nel codice oppure per effettuare dei test funzionali sul *layout* fisico della scheda.

1.4.2 Memoria Codice e Memoria Dati

Memoria Codice

La memoria codice consente il salvataggio dei byte caricati tramite interfaccia. Si tratta di un insieme di byte a cui corrisponde una codifica in binario delle istruzioni *assembly*, delle costanti e delle variabili globali del programma.
Ogni riga che compone questa memoria contiene un numero di *word* pari a *ROW_WORDS* e ogni *word* è composta da 32 bit.
Contrariamente a quanto riportato nella descrizione dei registri nel file di specifiche, in questo caso non è possibile reperire informazioni sulle singole porzioni di memoria contenute nella memoria codice. Viene dato solamente l'intervallo di indirizzi entro i quali possono essere conservate le istruzioni di programma. Le risorse disponibili vengono occupate in funzione di come il *bootloader* decide di disporre le varie porzioni che caratterizzano il codice eseguibile.

Memoria Dati

La memoria dati a livello RTL è costituita da una *word* per riga. Lo riempimento con le variabili globali di programma viene gestito dal *bootloader*, il quale si serve delle informazioni contenute nel *Linker Script* per allocare le risorse. Queste vengono scritte a partire dall'indirizzo più basso della RAM. Tutte le locazioni da specifiche sono accessibili sia in lettura sia in scrittura, di conseguenza il core può sovrascrivere il contenuto delle *word* già allocate in fase di *boot* del programma riferendosi direttamente agli indirizzi associati a quelle righe. Questo tipo di programmazione firmware non richiede livelli "intermedi" di software ed è al più basso livello possibile. Lo sviluppatore deve conoscere a fondo l'architettura del sistema, in modo da accedere alle varie risorse tramite gli indirizzi fisici dei vari registri. In questa tesi il metodo seguito consiste proprio nell'utilizzare questo tipo di approccio, detto anche approccio *bare metal*, che non prevede l'ausilio di una API intermedia né di un Sistema Operativo. Le uniche funzionalità tipiche di un OS sono quelle implementate dalla porzione di codice che viene automaticamente caricata in fase di *boot* del microcontrollore, come descritto al **Paragrafo 1.4**.

Analogamente a quanto già rilevato per la memoria codice, anche per la memoria dati non viene fornita dal designer una descrizione dettagliata di ogni porzione indirizzabile. Si ha solamente conoscenza della posizione nella *register map* degli indirizzi associati alla memoria dati. In questo caso la gestione delle risorse è assegnata al core, il quale è responsabile del salvataggio dei dati necessari al funzionamento del programma al suo interno. Oppure vengono scritti dei valori legati alla inizializzazione di variabili che appartengono a specifiche sezioni del codice eseguibile quali la `.bss` e la `.data`, per cui in tal caso il responsabile della scrittura di tali dati in specifiche posizioni della RAM è il *bootloader*.

1.4.3 *Microcontroller Registers* e *System Registers*

Le descrizioni che seguono servono a introdurre le principali funzionalità dei registri. Molte di queste funzionalità sono state sfruttate nella fase di test per gestire l'evoluzione dei test stessi, come ad esempio i segnali di *interrupt*, i registri di gestione degli algoritmi e i registri di uscita. Tuttavia va messo in evidenza il fatto che i test prodotti sono facilmente riutilizzabili al variare del microcontrollore da integrare in un design digitale: al variare della *register map* infatti, varierebbero solamente i valori dei registri salvati nelle variabili del programma, le quali invece agirebbero allo stesso modo. Il metodo utilizzato per rendere i test dei registri il più possibile portabili è dettagliato nel paragrafo dedicato al processo di *parsing* dei file di specifiche (**Paragrafo 4.1**).

Microcontroller Registers

I Registri interni al microcontrollore presentano un insieme di finalità differenti e sono organizzati come segue.

- Registri dedicati all'abilitazione e gestione degli *interrupt*. Per ogni algoritmo da eseguire deve essere abilitata la generazione di un segnale interno di *interrupt*. Questi segnali determinano l'invio di una *Interrupt Request* (IRQ) per l'esecuzione della routine associata all'i-esimo algoritmo. I segnali di *flag* a associati ai segnali di *interrupt* sono salvati in un apposito registro di stato e possono essere instradati verso le linee di *interrupt* esterne (ogni linea è associata a un *pad*). Il *routing* avviene attraverso dei registri di controllo, per cui tutti i bit asseriti in un registro di controllo vengono instradati sul medesimo *pad*.

- Un insieme di registri *General Purpose* detti *dual registers*, i quali possono essere utilizzati per implementare un meccanismo di sincronizzazione tra la *User Interface* e le operazioni del core. Ad esempio, su alcuni di essi il core è in grado di impostare dei generici valori di *flag*, mentre la *User Interface* può solamente leggerli e resettarli. Altri registri invece lavorano in modo opposto.

- Un sotto-banco di registri *General Purpose* di uscita, i quali possono essere letti da interfaccia per ricavare i risultati delle operazioni del core.

- Un sotto-banco di registri dedicati alla definizione degli *execution rate* (ODR) dei possibili algoritmi implementabili nel microcontrollore.

- Alcuni registri per configurare il microcontrollore (segnale di reset, segnale di abilitazione del clock, etc.) e per abilitare gli algoritmi.

- Un sotto-banco adibito al salvataggio dei dati di misura derivanti dalle catene dei sensori.

Ogni sotto-banco di registri in generale è caratterizzato da delle modalità di accesso differenti, le quali sono state il *focus* dei test che sono stati sviluppati.

System Registers

I Registri di Sistema sono un insieme di registri che si riferiscono più specificatamente alle configurazioni nel dominio dei sensori, ossia alle risorse esterne al design digitale del microcontrollore.

- Registri per abilitare il controllo dei *pad* esterni da parte del core.
- Registri di controllo per configurare i sensori attraverso la *User Interface*. Ad esempio per selezionare le modalità operative delle catene, gli ODR dei sensori e i segnali di temporizzazione di core e interfacce.
- Registri di controllo per configurare i sensori attraverso il core sfruttando funzionalità dedicate all'utente.
- Registri di stato che informano riguardo allo stato dei sensori e dell'interfaccia di comunicazione.
- Registri di *timestamp* che informano su quando è stato asserito l'ultimo *data ready*, ossia quando è stato salvato l'ultimo dato in uscita.
- Un sotto-banco di registri *General Purpose* per scrivere da interfaccia delle configurazioni generiche sul core.

Anche in questo caso a seconda delle funzionalità di ciascun registro vi possono essere *policy* di accesso diverse. Inoltre in questo caso molti registri sono indicati come accessibili ma non vengono utilizzati, ossia nessun identificativo è stato loro assegnato, né hanno delle funzionalità associate. Visto che tali registri sono fisicamente esistenti, ossia un insieme di *flip-flop* li definisce all'interno del design, è stato necessario testarli ugualmente assieme a tutti gli altri.

Capitolo 2
Ambiente UVM e Verifica Digitale

2.1 Motivazioni all'utilizzo e Caratteristiche fondamentali

La Universal Verification Methodology è uno standard IEEE che rappresenta lo stato dell'arte nell'approccio alla verifica digitale. La potenza di tale metodologia consiste principalmente in tre aspetti [7].

- La drastica riduzione del costo di verifica: l'ingegnere verificatore è coadiuvato dagli strumenti a disposizione all'interno della *UVM class library*, grazie ai quali si ha un aumento della produttività.

- La riusabilità: è possibile avvalersi di *Verification IPs* (VIPs) creati in precedenza e configurarli per nuovi *Designs Under Test* (DUTs) aventi le medesime interfacce.

- L'interoperabilità: lo standard UVM permette di uniformare la costruzione degli ambienti di verifica, consentendo una comunicazione agevole tra team di verifica differenti.

L'UVM si basa sul SystemVerilog (SV), un linguaggio di descrizione e di verifica dell'hardware, pertanto eredita da esso il paradigma orientato agli oggetti, la *Constrained Random Verification*, l'uso delle *Assertions*, la *Functional Coverage*, la programmazione a *Threads* e la comunicazione *Inter-Process*.
Le caratteristiche appena citate rendono il processo di verifica più efficiente e più facilmente riproducibile per testare design dotati di parecchie funzionalità [2, 7].

2.2 Ambiente UVM per la verifica del dispositivo

2.2.1 *Building Blocks* di un testbench UVM

Un generico *testbech* deve essere in grado di:

- generare degli stimoli;
- applicare gli stimoli sia al design sia a un modello di riferimento;
- catturare e raccogliere le risposte del DUT;
- effettuare comparazioni con i risultati del *golden model* di riferimento;
- verificare la correttezza delle risposte sulla base dei comportamenti attesi;
- tenere conto della percentuale di *coverage* delle funzionalità verificate.

Lo standard UVM mette a disposizione un insieme di strumenti per la *verification* capaci di sincronizzarsi e cooperare tra di loro per compiere tutte le operazioni sopra elencate. Il testbench UVM a disposizione del team di verifica per il dispositivo oggetto di studio prevede il dispiegamento di una serie di istanze, dettagliate nei paragrafi successivi [1, 2, 7] e rappresentate graficamente in **Figura 2.1**.

2.2.2 Descrizione del testbench UVM

L'entità a più alto livello, indicata nello schema come `tb_top`, è il modulo all'interno del quale sono definite tutte le connessioni e le istanze dei seguenti blocchi: DUT, interfacce, modelli e ambiente UVM.

Integrazione dell'istanza del DUT

Il DUT è l'istanza del design; nel caso in esame è costituito da un complesso insieme di moduli descritti a livello RTL in VHDL o in Verilog e interagenti tra di loro.
La trasmissione delle informazioni tra il DUT e l'ambiente di verifica avviene attraverso l'uso di interfacce. I percorsi dei segnali del design sono mappati all'interno di una *Virtual Interface*, grazie alla quale è possibile accedere sia ai segnali fisici interni, sia ai *pad* esterni. Sono anche definite una serie di interfacce associate ai vari componenti di verifica inclusi nel SVE (*SystemVerilog Environment*), le quali permettono di tradurre i segnali digitali in uscita dal DUT in transazioni UVM di pacchetti di informazioni comprensibili allo standard e viceversa.

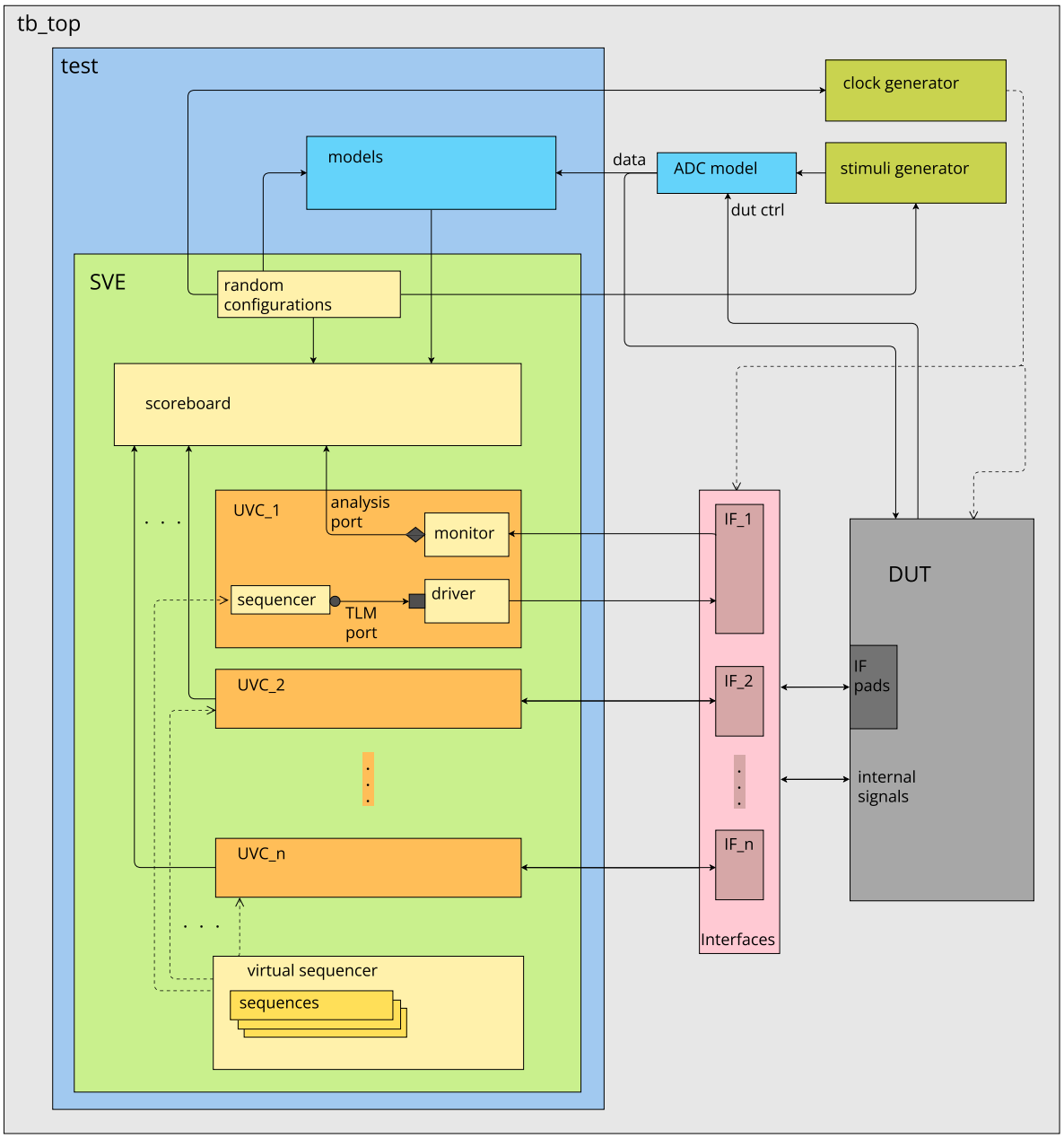


Figura 2.1: *Schema a blocchi dell'ambiente di test.*

L'Ambiente di Verifica

Il *SystemVerilog Environment* (SVE) è una classe che al suo interno istanzia tutti i componenti di verifica caratterizzanti la simulazione UVM. In particolare, sono istanziati gli oggetti a più alto livello, ossia la classe di randomizzazione, i componenti di verifica, il sequencer e lo scoreboard. La creazione di questo ambiente dipende dalla tipologia di test che si desidera eseguire. Possono essere definiti diversi test, all'interno dei quali può variare l'implementazione di alcune entità dell'ambiente a seconda delle funzionalità

che il test desidera verificare. Se i test agiscono su un DUT differente da quello per cui sono stati ideati, è possibile sfruttare la riusabilità dei componenti di verifica qualora le interfacce del design rimangano invariate (nel caso di studio, la comunicazione I2C o SPI).

Il *Random Generator*

Sulla base delle configurazioni decise dall'ingegnere verificatore, un generatore randomico provvede alla generazione dei parametri di simulazione. Esso opera grazie a un *engine* matematico capace di calcolare dei parametri di estrazione randomica a partire da un seme (*seed*) noto. Il *seed* è un numero intero e può essere scelto manualmente oppure selezionato direttamente dal *tool*. Grazie alla conoscenza del *seed* è possibile fare dei confronti tra i file di *log* dei risultati di simulazioni differenti, in quanto i parametri randomici possono essere ricondotti allo stesso albero binario che genera tali possibili valori.
Tutti i test contengono al loro interno dei parametri randomici. Avviata l'esecuzione di un test, la randomizzazione dei parametri avviene prima della fase di configurazione e può essere condizionata da alcuni vincoli (*constraint*) definiti sulla base dei test che si desidera attuare. I parametri di estrazione randomica possono essere visualizzati sulla console del programma di simulazione, i quali devono essere coerenti con la definizione dei *constraint* nel caso in cui questi siano sono stati definiti.

Lo *Stimuli Generator* e i Modelli

Sulla base dei parametri della randomizzazione, vengono generati gli stimoli da fornire al DUT grazie a un blocco detto *stimuli generator*, il quale emula il comportamento delle grandezze rilevabili dal sensore. Si tratta di un generatore di segnali, mediante il quale è possibile selezionare singolarmente gli stimoli e impostarne i parametri caratteristici (frequenza, ampiezza di picco, pendenza dei transitori, etc.). Ad esempio, possono essere generate onde quadre, sinusoidali, a dente di sega, triangolari, treni di impulsi, etc.

Lo standard UVM permette di integrare all'interno dell'ambiente di verifica anche un insieme di modelli; questi ultimi possono essere implementati ad esempio in linguaggio MatLab e servono a emulare il comportamento dei sensori a monte del design digitale. In questo modo è possibile avere a disposizione a livello RTL dei dati in ingresso ragionevoli e compatibili con quelli che si potrebbero avere nel caso reale.
I segnali provenienti dal generatore di stimoli vengono campionati dal modello del convertitore analogico-digitale, pilotato dal DUT, il quale instrada i dati sia verso i modelli inclusi nel test sia verso il DUT.

Per la gestione della temporizzazione del DUT sono definiti dei processi dedicati alla selezione della frequenza di funzionamento, alla generazione del segnale di reset esterno e alla generazione del segnale di clock.
Tali processi hanno lo scopo di emulare la parte analogica del design che implementa la

generazione del segnale di clock all'interno del sistema fisico. In **Figura 2.1** i processi in questione sono da ascrivere al blocco *Clock Generator* e sono caratterizzati da un insieme di parametri, quali frequenza, *jitter*, *duty cycle*, tempo di assestamento, etc.

Gli UVCs

Gli *Universal Verification Components* (UVCs) sono un insieme di blocchi di verifica riutilizzabili, i quali permettono all'ambiente SV di scambiare le informazioni durante l'esecuzione dei test. In altre parole, tali componenti permettono di fornire gli stimoli al DUT o di monitorarne il comportamento, raccogliendo i dati in uscita. A seconda di come sono implementati, gli UVC possono essere di due tipi: passivi o attivi. Gli UVC attivi interagiscono attivamente con il DUT grazie al *driver*, il quale converte i pacchetti di transazioni UVM in un insieme di segnali digitali da fornire in qualità di stimoli di ingresso al DUT. Gli UVC passivi invece non hanno il *driver* ma possono solamente raccogliere i dati provenienti dal DUT mediante il *monitor*, in modo da renderli disponibili per le operazioni eseguite dalle altre entità dell'ambiente di verifica.
Gli UVC presenti sono stati sviluppati dal team di verifica al fine di comunicare con gli *slave* I2C/SPI e si comportano da *master* in quanto:

- forniscono il segnale di temporizzazione;
- forniscono il comando di dato oppure attendono in lettura che il dispositivo trasmetta i dati in uscita.

In questa tesi, come ipotesi di partenza si considerano gli UVC integrati nell'ambiente di verifica come già funzionanti, consentendo:

- attraverso i driver, il corretto trasferimento delle informazioni delle sequenze UVM ideate in ciascun test e il DUT;
- attraverso i monitor, la raccolta dei dati provenienti dal DUT da instradare verso lo scoreboard implementato per lo specifico test.

Il *Sequencer* e le sequenze UVM

Il *sequencer* è un componente di verifica che permette di eseguire le sequenze di istruzioni atte al pilotaggio del dispositivo, dalle quali dipende il controllo dell'evoluzione dei test. Esiste una organizzazione gerarchica, secondo la quale il componente a più alto livello, detto *Virtual Sequencer*, sincronizza i sequencer a più basso livello integrati negli UVC. Questi ultimi a loro volta instradano i pacchetti di istruzioni verso i driver mediante le porte TLM (*Transaction Level Modeling*).

In generale, una sequenza UVM deve assolvere i seguenti compiti:

- accensione del dispositivo;
- attesa dei *trigger event*;

- comando di operazioni di scrittura e lettura, grazie ai quali gli UVC, muovono le linee a più basso livello, servendosi del protocollo implementato per eseguire scritture o letture agli indirizzi ricevuti;

- invio dei dati letti sulle linee verso lo scoreboard, per il *checking* della correttezza dei dati.

Per comunicare attraverso l'interfaccia del DUT vengono eseguiti dei task di lettura e scrittura. Essi possono essere personalizzati in funzione dell'intervallo di indirizzi a cui si desidera accedere, al numero di byte di dato e alle opzioni sul protocollo.
I segnali fisici corrispondenti a ciascun pacchetto atomico di lettura/scrittura sono visualizzabili nella sezione *Waveform* del simulatore. Il comportamento dei segnali sui *pad* di interfaccia corrisponde a quello descritto nel protocollo che viene utilizzato, per cui le linee che vengono mosse dall'UVC *master* presentano un andamento coerente con i *timing* riportati all'**Appendice A**.

La creazione di sequenze *custom* viene descritta nel dettaglio nei paragrafi dedicati dei capitoli sui test.

Lo *Scoreboard*

Si tratta dell'entità che si occupa del controllo delle funzionalità del DUT mediante la collezione dei pacchetti provenienti dagli UVCs e dei dati provenienti dai modelli. I dati vengono raccolti attraverso la definizione di collegamenti TLM detti *analysis port*. Quando un pacchetto è disponibile, esso viene automaticamente campionato.

Le classi di Test

L'ambiente UVM appena descritto viene generato a partire da una classe a più alto livello, indicata nello schema in **Figura 2.1** come `test`.
Si ha la classe test di base, che estende direttamente dalla classe `uvm_test` della *UVM Standard Library*, all'interno della quale sono definiti gli elementi di base per il corretto funzionamento di un test UVM. All'interno del test di base sono anche definite le fasi UVM di test (riportate per completezza in **Appendice B**).

Si possono definire altre classi che estendono dalla classe base; ognuna di esse è associata a un test specifico e permette, all'interno del SVE, l'utilizzo dei componenti definiti appositamente per quei test. In questo modo è possibile richiamare:

- la classe di randomizzazione utilizzata;

- la classe scoreboard specifica implementata per il controllo della *feature* da testare;

- la sequenza responsabile dell'invio di scritture e letture attraverso l'interfaccia e, più in generale, dell'evoluzione delle operazioni di test. Tali transazioni possono avvenire sulla base di particolari condizioni sui segnali di controllo, come combinazioni di fronti di salita e di discesa, etc.

Esistono vari tipi di test, sviluppati nel corso degli anni dagli ingegneri del team di verifica digitale, i quali ereditano gli attributi e i metodi della classe base. Ciascun test si premura di verificare una specifica porzione del design e delle specifiche funzionalità.

Durante il lavoro di tesi, al fine di acquisire dimestichezza con il complesso ambiente di verifica e con le principali funzionalità del design digitale, sono stati avviati alcuni test già preesistenti. In particolare si è fatto uso del test della catena di interfacciamento e del test del microcontrollore. In questo modo è stato possibile acquisire le seguenti informazioni.

- Prendere atto del corretto funzionamento e interfacciamento della catena digitale, verificando che i protocolli I2C e SPI implementati nel design evolvono correttamente in funzione delle transazioni di dati e indirizzi. Se tale test non avesse funzionato, non sarebbe stato fattibile proseguire, in quanto l'interfaccia è un elemento fondamentale per i test successivi che si focalizzano sugli elementi interni del design.

- Studiare il comportamento del microcontrollore. Il test in questione permette di:

  · eseguire gli algoritmi definiti in C e processati dal core, acquisendo i segnali desiderati (valori di ingressi e uscite), al fine di poterli stampare e post-processare in ambiente UVM;

  · eseguire in parallelo il modello SystemVerilog degli algoritmi implementati in C;

  · i risultati di entrambe le elaborazioni vengono comparati all'interno dello scoreboard, il quale riporta dei messaggi di errore nel caso fossero presenti dei *mismatch*.

Test della catena
La prova della catena di elaborazione DSP si articola nei seguenti passaggi.

- Controllo dei dati in uscita dai sensori, confrontandoli con quelli del modello MatLab incluso. Tale controllo si effettua mediante la procedura di lettura dei registri di uscita delle interfacce I2C e SPI.

- Controllo della procedura di configurazione del dispositivo, la quale avviene tramite operazioni di scrittura da interfaccia verso i registri di configurazione.

Lo studio del comportamento di tale test è servito per osservare, in fase di simulazione, l'evoluzione dei comandi di lettura e scrittura dei due protocolli di comunicazione, constatando che essi corrispondessero a quelli definiti nel test.
Come già anticipato, una volta appurato che i dati provenienti dalla catena DSP sono congruenti, si può procedere con gli altri test di verifica.

Test del microcontrollore
Il test in questione, oltre a svolgere sul design le operazioni già definite nel test della catena, si occupa di programmare il microcontrollore.
L'attenta analisi di tale test ha permesso di verificare l'evoluzione dei segnali di interesse in

fase di simulazione e alcune modifiche dello stesso hanno consentito di osservare evoluzioni differenti dello stesso test e di constatare che le nuove evoluzioni corrispondessero a quella desiderata. Sostanzialmente, il test in oggetto è servito per acquisire manualità nella gestione dell'ambiente, sia lato UVM (scoreboard, classe di randomizzazione con la definizione di nuovi *constraint*, modifica di letture e scritture della catena, etc.), sia lato microcontrollore, ridefinendo gli algoritmi preesistenti e modificando in parallelo il modello SV. Nello specifico, gli algoritmi C riportano sui registri di uscita i valori letti dai registri di dato dei sensori in ingresso. Apportando semplici modifiche aritmetiche, ad esempio operazioni di *shifting*, è stato possibile visualizzarne gli effetti sui segnali di uscita in fase di simulazione. Gli algoritmi di prova preesistenti sono molto semplici e hanno permesso di prendere dimestichezza con un ambiente di test che in realtà è assai più complesso.

Mentre il test di base del microcontrollore prevede l'utilizzo in parallelo di algoritmi in C e modelli SV, l'obiettivo dei test sviluppati nell'attività di tesi ha riguardato una tipologia totalmente differente e nuova di test. Si è trattato di implementare test di *self-checking*, i quali non prevedono un modello SV da eseguire in parallelo, né l'utilizzo dell'interfaccia per l'esecuzione del test dei valori nelle porzioni di memoria oggetto di verifica. In questo caso è il core stesso che, mediante l'esecuzione di algoritmi appositamente sviluppati (che rimpiazzano i precedenti), è responsabile dell'applicazione di una metodologia di controllo sulle porzioni di memoria del design. L'ambiente SV invece si "limita" a collezionare i dati in uscita dal microcontrollore, alla loro decodifica e al *reporting* dei risultati di simulazione.

Test della RAM e Test dei banchi di registri
I test in questione costituiscono il fulcro dell'attività di tesi.
Per la descrizione dettagliata delle classi di test implementate per la verifica della memoria dati e dei banchi di registri si rimanda ai capitoli successivi, dedicati interamente al lavoro di tesi.

Capitolo 3
Test della RAM del Microcontrollore

Il primo elemento del design su cui si sono concentrati gli sforzi per la creazione di test è la memoria dati del microcontrollore. La RAM in questione è stata progettata in linguaggio VHDL ed è stata integrata all'interno del progetto digitale dal team di Design. L'obiettivo dell'attività è stato quello di verificare i seguenti aspetti caratterizzanti:

- Il corretto interfacciamento della memoria dati con il core. Tutte le locazioni a disposizione che sono state mappate all'interno del design occorre che risultino accessibili sia in lettura sia in scrittura dal core del microcontrollore. Nel caso di comportamento anomalo, l'algoritmo in esecuzione deve restituire opportunamente le informazioni di interesse, in modo che esse vengano post-processate in fase di *checking* in ambiente UVM.

- Utilizzare una metodologia che possa testare non solo gli accessi ma anche il corretto *toggling* dei bit memorizzati all'interno delle singole celle di memoria. L'obiettivo è verificare l'assenza di anomalie durante scritture e letture alternate multiple, in modo tale che anche il comportamento dinamico della memoria possa essere validato. Trattandosi di test applicati su design a livello RTL, non si può fare riferimento alla disposizione fisica corrispondente delle righe e locazioni della RAM rispetto alla loro descrizione in VHDL. Tuttavia, tra gli obiettivi vi è anche quello di produrre dei test portabili anche per la fase di *Final Test*. In altre parole, si vuole mettere in atto l'applicazione di una metodologia che verifichi l'assenza di difetti di produzione delle celle fisiche della ram, per cui lo stato di ciascuna cella non sia influenzato dallo stato di quelle adiacenti, per quanto riguarda sia la loro disposizione per righe sia per colonne.

3.1 Motivazioni dell'utilizzo di test *self-checking*

Lo sviluppo dei test di questa attività di tesi si serve di una particolare metodologia, la quale consiste nel costruire dei test che non prevedono l'implementazione di un modello e di un meccanismo di controllo nello scoreboard. Grazie agli algoritmi caricati all'interno del core infatti, è possibile controllare direttamente la correttezza del design. Il *checking* viene eseguito *run time* nelle routine degli algoritmi e i risultati vengono salvati in modo codificato sui registri di uscita. Terminata l'esecuzione nel core, i risultati vengono letti

da interfaccia e instradati verso lo scoreboard. Quest'ultimo ha il compito di decodificare le informazioni e di riportare i risultati nel file di *log*.

L'uso di tale metodologia serve essenzialmente per avere delle *routine* portabili anche per la fase di *Final Test*, nella quale il programma viene eseguito direttamente sull'hardware in Silicio, con la possibilità quindi di avere una verifica della correttezza della funzionalità del design fisico.

3.2 Processo di ideazione dei test e metodologia seguita

Partendo dalle informazioni a disposizione, ricavate dal manuale tecnico del microcontrollore, è stato possibile pianificare lo sviluppo di opportuni test per la memoria dati. Per applicare dei test esaustivi e fattibili, si è tenuto conto delle seguenti risorse hardware:

- Registri di uscita *General Purpose*. Tali registri possono essere utilizzati per riportare all'ambiente UVM l'informazione sui dati processati dal core che esegue il test di *self-checking*. Siccome il numero totale di registri messi a disposizione per tale scopo è limitato, lo sviluppo e l'organizzazione dei test è stata fortemente influenzata da questo aspetto, come riportato nel **Paragrafo 3.3**.

- Segnali di *interrupt*. Nello specifico, il design include in tutto due segnali che fungono da segnali di *interrupt*; il loro uso in modo combinato può servire per avvisare in fase di simulazione che si sono rilevate anomalie, oppure è possibile utilizzarli come segnali di *trigger* affinché la sequenza UVM modifichi il suo comportamento mandando comandi differenti agli UVC di interfaccia. In generale quindi, è possibile utilizzare segnali di *interrupt* sia da instradare verso i *pad* esterni per informare del fallimento del test al termine della simulazione, sia per modificare dinamicamente l'andamento della simulazione sulla base di condizioni di test che si sono verificate *run time*.

Dal punto di vista del supporto software, ci si è affidati alla *Toolchain* di compilazione messa a disposizione nel progetto, la quale genera il codice eseguibile supportato dallo specifico processore integrato nel design. In particolare si è fatto riferimento ai seguenti file di compilazione:

- uno script bash per l'avvio dei comandi di compilazione;

- il file eseguibile e il suo corrispettivo in *S-format* (vedasi **Paragrafo 3.3.3**);

- il *linker script*, all'interno del quale sono descritte le informazioni che istruiscono il *bootloader* sull'arrangiamento nella memoria delle varie porzioni di codice eseguibile.

- il file *disassembly*, utile per controllare eventuali condizioni che provocano l'uscita anomala dal programma principale, a seguito di anomalie verificatesi durante

l'esecuzione degli algoritmi che non possono essere comprese dalla semplice visualizzazione dei segnali di simulazione. Grazie all'analisi delle istruzioni *assembly*, è possibile risalire a quali istruzioni causano problemi e riscrivere il codice sorgente in una forma tale da evitare la generazione di codice eseguibile contenente istruzioni che comporterebbero l'esecuzione di alcune *system calls* per la gestione delle eccezioni, le quali in ultima istanza determinerebbero la condizione di *abort* del programma.

Il processo di ideazione dei test ha tenuto conto non solo delle peculiarità dell'hardware ma anche dell'organizzazione dell'ambiente UVM. In particolare, ciascun test ha reso necessaria l'implementazione di classi SystemVerilog *ad hoc* che gestissero le informazioni e il flusso di test.

- La classe di randomizzazione con al suo interno i parametri di interesse e i rispettivi (eventuali) *constraint*.

- La sequenza di gestione dei comandi. In questo modo, sulla base di alcune condizioni su segnali di test, è possibile personalizzare i comandi di lettura e scrittura da fornire agli UVC di interfaccia per il pilotaggio dei monitor e dei driver che forniscono e ricevono i segnali digitali da e verso il DUT.

- Lo scoreboard per la decodifca e il controllo dei dati in uscita dal microcontrollore, trasmessi allo scoreboard mediante le *analysis port* sotto forma di pacchetti di tipo *sequence item*, a partire dai task di scrittura implementati nella sequenza per la comunicazione con lo scoreboard e attraverso delle funzioni invocate automaticamente nello scoreboard al fine di catturare i pacchetti ricevuti.

Nei paragrafi successivi sono analizzati nel dettaglio gli algoritmi implementati in C e la loro gestione in ambiente UVM, sottolineando i punti di forza di ciascun approccio. Per ogni verifica effettuata, si riportano i risultati salienti e si dimostra la funzionalità della memoria RAM e l'efficacia dei test prodotti.
L'analisi che segue è effettuata su parametri hardware generici, in quanto il firmware di test è adattabile al variare delle risorse di memoria a disposizione grazie alla sua organizzazione modulare. Inoltre, grazie alla ottimizzazione del codice sorgente, il codice eseguibile risulta compatto e presenta un impatto limitato sulla memoria di programma in cui viene caricato, permettendo il suo riuso all'interno di altri microcontrollori con risorse più limitate. Gli aspetti relativi all'ottimizzazione sono approfonditi nelle **Conclusioni** in cui sono esposte le considerazioni finali sul lavoro svolto.

3.3 Test RAM #1: esecuzione di più algoritmi in sequenza

Il primo test prevede di verificare la RAM da cima a fondo in un'unica esecuzione dello stesso test. L'obiettivo di questo test è determinare se all'interno della RAM ci siano delle locazioni di memoria che, al termine delle operazioni di accesso eseguite, non si comportano come desiderato. Nel caso di comportamenti anomali, si abilitano dei bit di stato e si genera un segnale di *interrupt* verso l'esterno.
L'organizzazione del test è stata concepita sulla base del dimensionamento di memoria dati del microcontrollore oggetto di studio. Tuttavia l'analisi che segue può essere generalizzata a un qualsiasi microcontrollore avente al suo interno una memoria dati da DIM_DM kB generici. Quello che cambierebbe sarebbe il partizionamento che il test compie sulla memoria per raccogliere correttamente le informazioni a valle dell'esecuzione del test. Il design RTL oggetto di test presenta, per la memoria dati, la seguente descrizione:

- i DIM_DM kB di memoria mappati nella RAM sono suddivisi su NUM_ROWS righe, ciascuna delle quali composta da 32 bit;
- la suddivisione per righe in VHDL è coerente con il dimensionamento delle *word* definito nei manuali di riferimento del microcontrollore e del core. Ciascuna riga pertanto contiene al suo interno l'informazione associata a una *word* su 32 bit.

L'accesso ai dati memorizzati nella RAM può avvenire solamente "per righe". In altre parole, per leggere o scrivere meno di 4 byte disponibili in una *word* è necessario accedere a tutti i byte della *word*. Il vincolo di accesso a *burst* ha quindi determinato la modalità di svolgimento del test. Il test di tutti i bit di memoria viene eseguito modificando e controllando i bit riga per riga. Il tutto viene eseguito dal core stesso, rispettando il vincolo originario di avere un test di tipo *self-checking*. Al termine di tutti gli algoritmi lanciati, il controllo viene ceduto all'ambiente UVM per decodificare i risultati da riportare in un file di *log* e la simulazione termina.

L'informazione che si desidera riportare in questo test preliminare riguarda quale o quali tra i blocchi di RAM testati presentano dei fallimenti. In altre parole, nel caso in cui all'interno di un blocco venga rilevato almeno un bit disallineato rispetto al valore atteso, la simulazione deve riportare, sotto forma di segnali di flag, la posizione del blocco di memoria che sbaglia, in modo tale da eseguire successivamente un altro test più approfondito solo sui blocchi critici, agendo questa volta per blocchi isolati e riportando anche le posizioni delle righe che presentano dei bit che falliscono, come descritto nel **Paragrafo 3.4**.

Il test è stato concepito in questo modo per via delle risorse hardware limitate a disposizione. Siccome il numero di registri disponibili per riportare i risultati è relativamente ridotto, non è fattibile salvare anche le posizioni dei bit che falliscono all'interno delle singole righe della RAM.

Ad esempio, si prenda in considerazione il seguente caso di studio, in cui sono scelte delle dimensioni ragionevoli per gli elementi di memoria, in modo da rispecchiare gli spazi dedicati alla memoria dati e ai registri *General Purpose* tipici di un generico microcontrollore.

Si consideri una memoria dati da $DIM\_DM = 4096$ byte, divisa in $NUM\_ROWS = 1024$ da $NUM\_COLS = 4$ byte ciascuna e $NUM\_GP = 16$ registri di uscita da $DIM\_GP\_reg = 2$ byte ciascuno. Il numero totale di bit da testare è pari a

$$NUM\_BITS = NUM\_ROWS \cdot (8 \cdot NUM\_COLS) = 32768 \text{ bit}$$

Per poter mappare la posizione di ogni bit sui registri di uscita, con l'obiettivo di ricavare la posizione del bit che fallisce, è necessario avere a disposizione un numero di registri pari a

$$NUM\_GP\_ideal = \frac{NUM\_BITS}{8 \cdot DIM\_GP\_reg} = 2048$$

$$NUM\_GP\_ideal >> NUM\_GP$$

pertanto la soluzione che salva l'informazione della posizione dei bit che sbagliano non risulta concretamente praticabile. Nella realtà il numero totale di registri disponibili è molto inferiore a quello necessario per implementare una simile soluzione.

Sulla base di queste considerazioni, il test potrebbe tenere conto del salvataggio degli indirizzi associati alle righe che presentano almeno un bit che fallisce. Tuttavia sorgerebbe un altro problema: mappando ogni riga su un bit differente dei registri di uscita, nel caso preso come esempio si raggiungerebbe la saturazione dei bit messi a disposizione dopo aver mappato un numero di righe pari a

$$ROWS\_BLOCK = NUM\_GP \cdot (8 \cdot DIM\_GP\_reg) = 256$$

Da qui nasce la necessità di testare separatamente i blocchi di RAM per evitare di sovrascrivere l'informazione salvata nei registri di uscita. Ciascun blocco, detto *chunk*, è testato da un algoritmo differente. Il numero totale di algoritmi coincide col numero totale di blocchi ed è pari a

$$NUM\_ALGOS = \frac{NUM\_ROWS}{NUM\_GP \cdot (8 \cdot DIM\_GP\_reg)}$$

Facendo nuovamente riferimento alle quantità utilizzate nell'esempio, si avrà $NUM\_ALGOS = 4$.

Si consideri ora il comportamento dell'algoritmo i-esimo. Al termine della sua esecuzione si vuole vengano salvate in uscita le posizioni delle righe associate al *chunk* sotto test.
Sostanzialmente, ogni bit di ciascun registro di uscita contiene l'informazione codificata di una *word* della RAM. L'ordinamento dei bit all'interno del singolo registro presenta lo stesso ordinamento della sequenza di righe corrispondente. I registri di uscita sono

un sotto-banco di celle di memoria consecutive, di conseguenza due registri di uscita consecutivi contengono in tutto la codifica di $(2 \cdot DIM\_GP\_reg) + (2 \cdot DIM\_GP\_reg)$ righe consecutive.

La codifica binaria ha il seguente significato:

- se una *word* presenta al suo interno almeno un bit di *mismatch*, il core scrive '1' nel registro di uscita al bit corrispondente a quella *word*;
- se il test della riga ha esito positivo, il core lascia il bit corrispondente al valore di inizializzazione, ossia '0'.

Il core si occupa della scrittura dei bit, invece l'interfaccia accede in lettura ai registri di uscita attraverso una sequenza di letture multiple di registri consecutivi, grazie alle quali in ultima istanza le informazioni lette vengono instradate verso lo scoreboard per essere opportunamente decodificate. Il meccanismo che sta alla base della codifica è rappresentato graficamente in **Figura 3.1** e si basa su un codice termomentrico, per cui all'aumentare del numero di riga aumenta la posizione del bit di salvataggio.

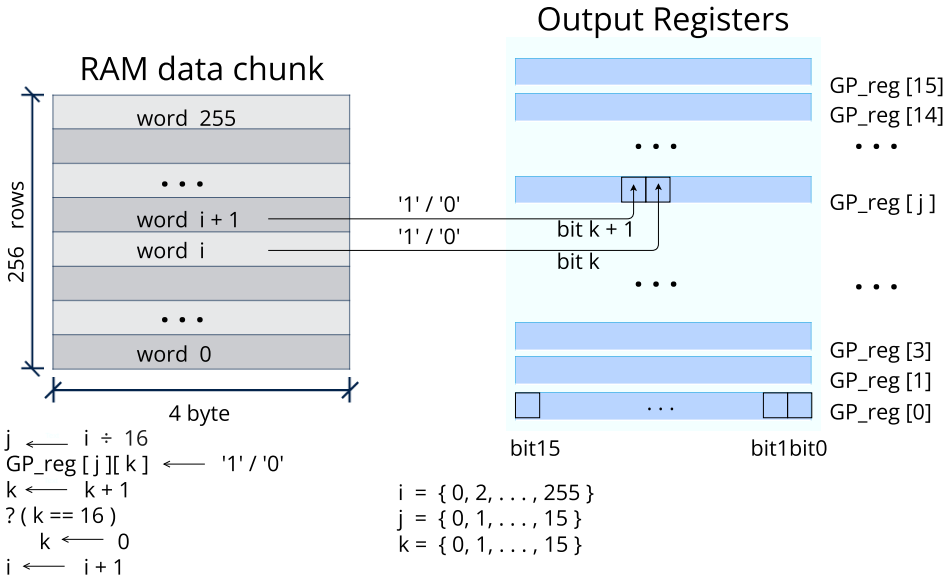


Figura 3.1: *Mapping delle righe della memoria dati sui bit dei registri di uscita.*

3.3.1 Obiettivi del test

L'obiettivo di base di questo test è quello di ottenere una prima stima quantitativa circa la correttezza del design. Il test preliminare esegue tutti gli algoritmi in sequenza, pertanto al termine dell'esecuzione del core l'interfaccia esegue solamente la lettura del registro di stato degli *interrupt* per verificare quali algoritmi presentino come esito:

- $PASSED$ se tutti i bit, ossia tutte le celle, si comportano correttamente;

- *NOT_PASSED* nel caso in cui almeno un bit del blocco fallisce i controlli dopo la sequenza di operazioni eseguite a seguito degli accessi del core nella memoria.

3.3.2 Sviluppo del test C

Il fulcro di questo test si può descrivere come segue.
Per verificare che le righe di memoria siano tutte accessibili, si è optato per due cicli di scrittura alternati da due cicli di lettura, codifica delle uscite e controllo:

- dapprima viene scritta in ogni riga la costante su 32 bit `0xAAAAAAAA`. L'operazione equivale a scrivere il pattern "1010...";
- si effettua un ciclo di lettura e si verifica se il dato letto è uguale al dato atteso, ossia alla costante sopra indicata;
- il secondo ciclo di scrittura inverte ciascun bit della costante precedente, per cui viene scritta su ogni riga la *word* `0x55555555`. L'operazione equivale a scrivere il pattern "0101..."
- si ripete la lettura verificando nuovamente che il nuovo valore atteso sia pari a quanto viene letto a ogni riga.

Gli *step* di base di questo test sono molto semplici, e vengono reiterati allo stesso modo per ogni blocco in cui è stata divisa la memoria. Essenzialmente ogni algoritmo esegue le stesse operazioni, pertanto è bastato definire tre funzioni C che vengono invocate all'interno di ogni algoritmo.

La gestione degli algoritmi avviene all'interno del task *main*, attraverso delle routine predefinite che non sono state modificate. Una volta calcolato quale algoritmo deve essere eseguito, viene automaticamente invocata la funzione che lo implementa. Il funzionamento è schematizzato in **Figura 3.2**.

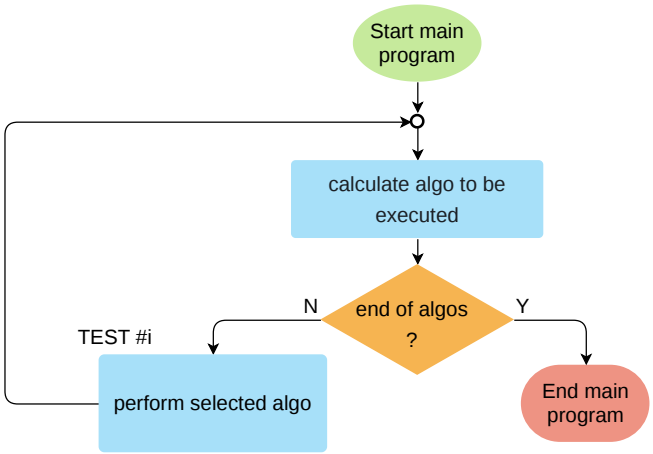


Figura 3.2: *Flusso di gestione della selezione ed esecuzione degli algoritmi di test integrati nel programma.*

Nel *main* del programma viene invocata una funzione che calcola il prossimo algoritmo da eseguire. Esiste un registro interno al microcontrollore adibito alla definizione di quale algoritmo eseguire per cui i bit che vengono settati in questo registro hanno una decodifica decimale che corrisponde al numero dell'algoritmo da eseguire. La velocità con la quale gli algoritmi possono essere eseguiti è definita da una variabile di *prescaler*. Si tratta di valori interi che corrispondono a determinati *Output Data Rate*, per cui le velocità massime di esecuzione si hanno per valori di *prescaler* inferiori. La velocità viene raddoppiata ad ogni incremento unitario del *prescaler* e viene codificata sempre nel medesimo registro grazie a una operazione di *shifting* sui bit. Il registro dell'algoritmo corrente, che è stato impostato dal core, viene confrontato in ambiente UVM per determinare se l'algoritmo è stato abilitato attraverso i parametri definiti nella classe di randomizzazione. In caso positivo, l'algoritmo viene eseguito dal core, altrimenti viene filtrato e si procede con l'iterazione successiva che ripete le operazioni sopra descritte.

Ogni algoritmo agisce su uno specifico blocco della RAM, identificato anche come *chunk*. Nel codice C, per ogni algoritmo è definita una variabile che contiene il numero progressivo del blocco e grazie ad esso è possibile definire l'offset di riga a partire dal quale vengono eseguite le operazioni. Il test di ogni blocco si articola come mostrato in **Figura 3.3**.

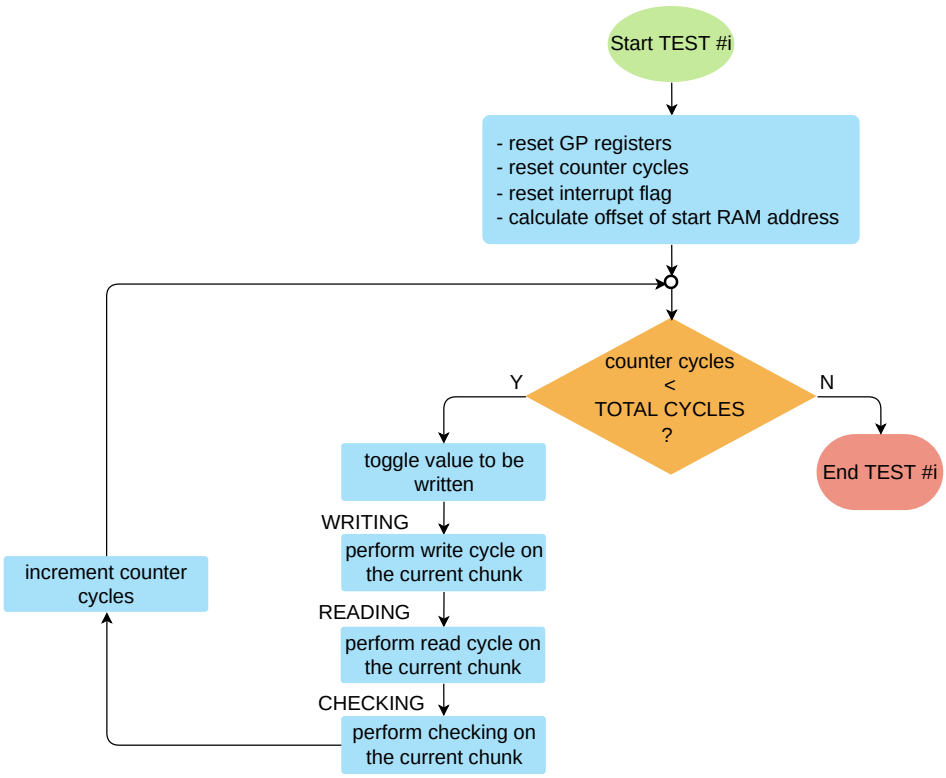


Figura 3.3: *Algoritmo di test di un generico blocco della RAM.*

Inizialmente si resettano i registri *General Purpose* adibiti a uscite per il salvataggio degli indirizzi codificati. Tra le operazioni di inizializzazione preliminari, si sottolinea quella che calcola l'indirizzo di partenza, ossia l'offset del *chunk* corrente:

```
(volatile uint32_t *) ram_address = (volatile uint32_t *) \
  start_ram_address + (uint32_t) (algo_number * DIM_CHUNK);
```

dove *algo_number* è apri al valore zero nel caso di test delle prime $ROWS\_BLOCK$ righe della RAM, pertanto quando viene lanciato il primo algoritmo l'offset di partenza coincide con l'indirizzo di partenza della RAM all'interno della mappa di memoria.

Dopo la fase di inizializzazione dell'algoritmo, il cuore del test è costituito da un ciclo che racchiude al suo interno i vari passi in cui si articola il test. Il ciclo viene rieseguito in totale due volte ($TOTAL\_CYCLES = 2$) per fare in modo che ogni bit abbia la possibilità di commutare stato. Infatti la prima operazione che viene eseguita ad ogni iterazione è il *toggling* della costante che si desidera scrivere:

```
uint32_t ram_value = ~ram_value;
```

Per rendere il codice portabile, tale costante è dichiarata sotto forma di macro in un *header file*, assieme a una serie di altre macro che definiscono ad esempio gli indirizzi estremi della RAM, i valori di alcuni *flag*, etc. Sempre per soddisfare il criterio di portabilità, è stato sviluppato del codice il più possibile modulare, per cui ogni macro-blocco nei diagrammi di flusso, ossia i blocchi indicati con una etichetta a lato, sono stati implementati con delle funzioni. Questo approccio ha permesso di riutilizzare, ove è stato possibile, alcuni moduli anche in altri test, riducendo i tempi di sviluppo e ottimizzando il codice prodotto. I macro-blocchi riportati nel diagramma di cui sopra, sono ampliati nelle pagine successive in **Figura 3.4**, **3.5** e **3.6**.

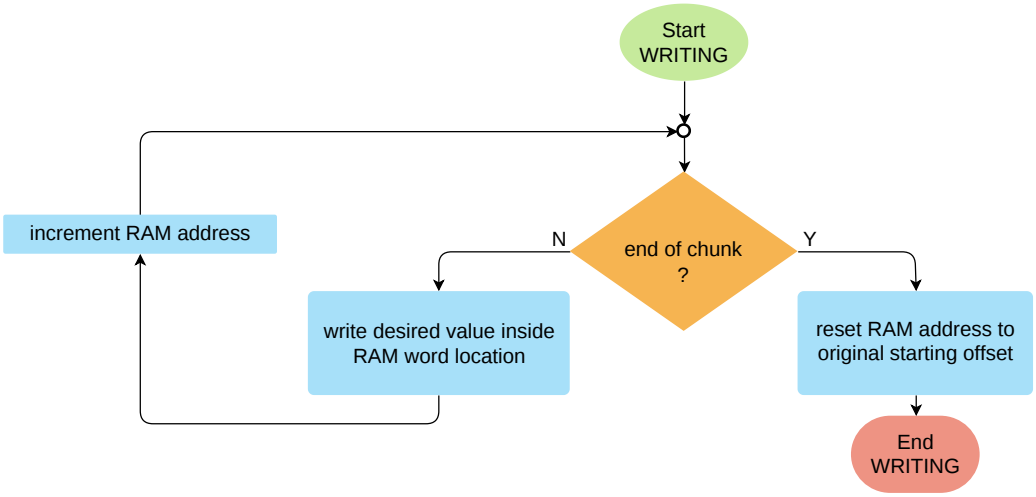


Figura 3.4: *Ciclo di scrittura di un generico blocco della RAM.*

L'operazione base del ciclo di scrittura consiste nel de-referenziare l'indirizzo di riga per scrivere il dato desiderato. Infatti l'unico modo che ha il core di accedere alle risorse hardware è quello di riferirsi a esse tramite indirizzi fisici di memoria. Grazie alla mappa di memoria definita in fase di design, ogni risorsa hardware è identificata dal core attraverso un indirizzo univoco, di conseguenza basta definire una variabile di tipo puntatore per accedere ai registri. Siccome ogni riga della memoria RAM è costituita da 4 byte, un incremento unitario della variabile puntatore a ogni iterazione corrisponde a referenziare la *word* successiva, riferendosi al primo byte di essa.
Le due operazioni appena descritte si traducono nelle seguenti istruzioni C:

```
*((volatile uint32_t *) ram_address) = ram_value;
(volatile uint32_t *) ram_address += (uint32_t) 0x00001;
```

Per quanto concerne il ciclo di lettura, le operazioni eseguite sono descritte in **Figura 3.5**.
In caso di *mismatch* tra valore memorizzato nella riga e valore desiderato, viene calcolato l'offset corrispondente al registro GP in cui occorre asserire il bit relativo alla riga che fallisce. Al primo mismatch trovato viene asserito un flag di stato, passato come parametro *by reference* alla funzione, il quale al termine della fase di lettura verrà testato nella routine di *checking*.
A seconda dell'iterazione corrente, vengono eseguiti gli incrementi e i reset delle variabili della routine di lettura necessari alla corretta computazione della posizione in cui salvare il bit di codifica dell'indirizzo contenente il valore disallineato. Le operazioni C sono le seguenti:

```
1  do {
2      if ((*(volatile uint32_t *)(ram_address) != ram_value)) {
3          *flag_int = NOT_PASSED;
4          num_of_GP_reg = (int) (ram_address - ram_offset)/DIM_GP_reg;
5          *(volatile uint32_t *)(GP_reg_00 + 2*num_of_GP_reg) \
6                                      |= 1 << cnt_pos_bit;
7      }
8      if (cnt_pos_bit == (DIM_GP_reg - 1)){
9          cnt_pos_bit = 0;
10     } else {
11         cnt_pos_bit++;
12     }
13         ram_address += (uint32_t) 0x00001;
14 } while (ram_address < (ram_offset + (uint32_t) DIM_CHUNK));
```

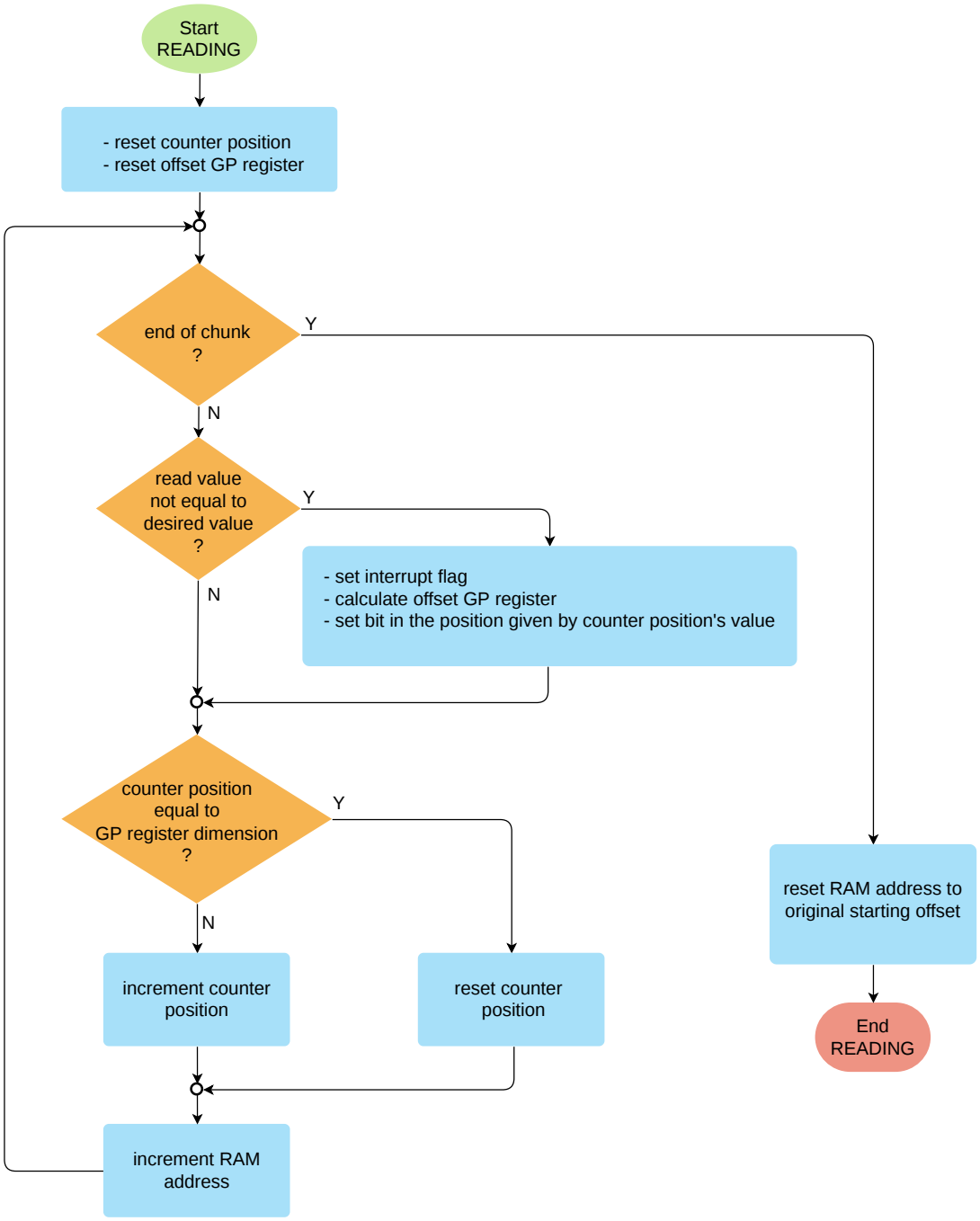


Figura 3.5: *Ciclo di lettura di un generico blocco della RAM, comprensivo di salvataggio codificato delle righe che falliscono.*

Infine, per riportare all'ambiente di verifica l'informazione di quali blocchi presentano almeno un fallimento, si scrivono, all'interno del registro di stato associato ai flag di *interrupt*, i bit le cui posizioni sono associate al numero di *chunk* corrente. L'operazione di *set* avviene dopo aver testato la variabile di *flag* del ciclo di lettura.

L'informazione del registro di stato viene sfruttata in ambiente UVM per riportare nel file di *log* quali blocchi presentano come esito del test *PASSED* oppure *FAILED*. Inoltre, secondo le configurazioni che sono state impostate tramite l'ambiente UVM nella fase precedente l'esecuzione degli algoritmi, tutti i bit del registro sono instradati sul *pad* dedicato al segnale di $INT1$ tramite una *OR* logica. Pertanto, se questa linea di *interrupt* esterna viene stimolata, significa che il test presenta almeno un fallimento.

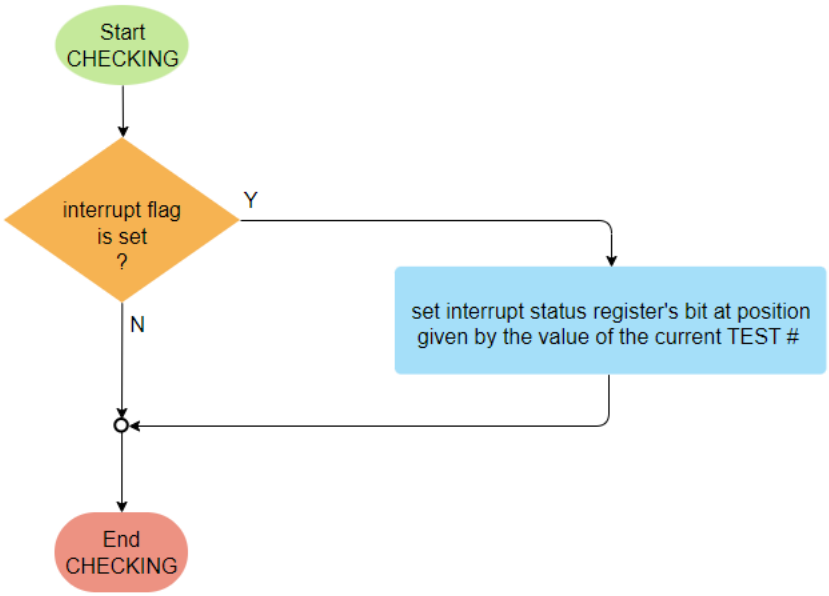


Figura 3.6: *Principio di funzionamento del rilevamento dell'errore.*

3.3.3 Processo di compilazione del test C

Una volta prodotto il codice sorgente, occorre compilarlo attraverso la *Toolchain* messa a disposizione dello sviluppatore seguendo i passaggi indicati in **Figura 3.7**. Il passaggio antecedente la compilazione vera e propria è compiuto dal Preprocessore. Esso effettua una sostituzione letterale, all'interno dei file sorgente, dei valori delle macro dichiarate all'interno degli *header file* e include l'insieme di funzioni C definite in eventuali altri file presenti.
Gli *step* che traducono le istruzioni del codice sorgente in codice binario sono eseguiti dal Compilatore e dall'Assemblatore. Le istruzioni di programma vengono tradotte in una serie di istruzioni elementari in linguaggio *assembly* (estensione `.S`). A questo punto l'Assemblatore traduce gli mnemonici corrispondenti alle varie istruzioni *assembly* in linguaggio macchina e il file ottenuto è detto file oggetto (estensione `.o`). Si tratta di un insieme di byte aventi un significato comprensibile per l'hardware. Tutte le informazioni sono organizzate secondo una *symbol table*. A questo livello, tutti i simboli associati alle funzioni e alle variabili dello sviluppatore sono stati risolti e sono associati a un indirizzo con un offset all'interno del file. Tuttavia mancano ancora una serie di informazioni cruciali per la corretta esecuzione del programma. Infatti le funzioni di libreria non sono

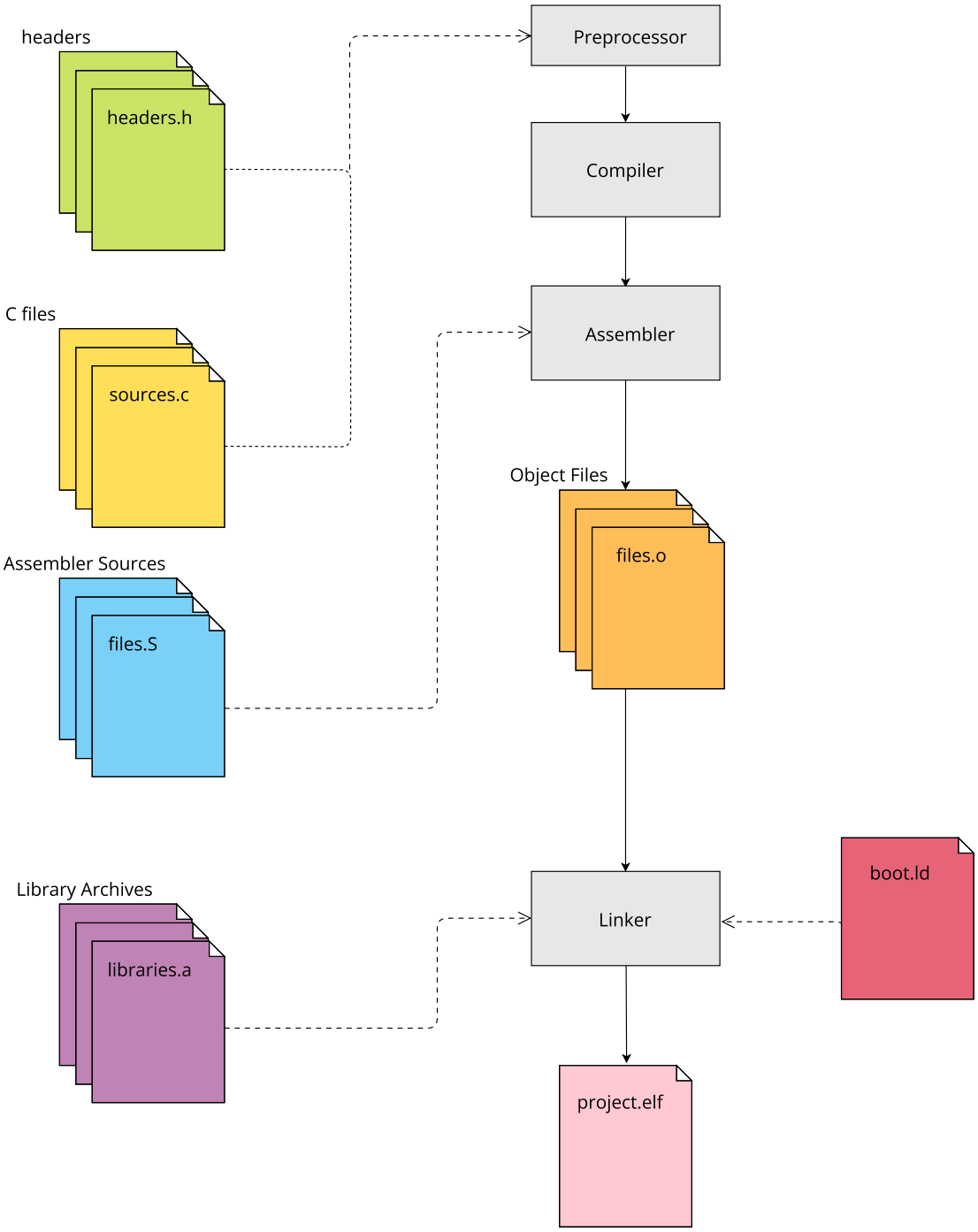


Figura 3.7: *Flusso di compilazione perseguito dalla Toolchain di sviluppo.*

ancora state risolte in quanto manca il codice eseguibile a esse associato.
Grazie al Linker viene aggiunta la parte di inizializzazione che precede l'esecuzione del programma principale, ossia le funzioni del *C runtime environment* (`crt0.o`) e il corpo delle funzioni di libreria che sono invocate nel programma. In questo modo si ottiene un

file all'interno del quale tutti gli elementi sono risolti e corrispondono a degli indirizzi che costituiscono l'immagine eseguibile del programma. Questo file ha estensione `.elf`, che sta per *Executable and Linkable Format*.
Il programma è diviso in un insieme di *Program Segment* (**Figura 3.8**). Grazie alle informazioni contenute nel *linker script* (file con estenzione `.ld`), ogni porzione del file è mappata all'interno della memoria fisica.

```
Section to Segment mapping:
 Segment Sections...
  00     .init .text
  01     .data .bss
```

Figura 3.8: *Informazione sui segmenti contenuti nell'eseguibile stampata dal comando* ***`readelf`****. A ogni segmento corrisponde un sotto-insieme di sezioni; vengono riportate solo le sezioni di dimensione non nulla.*

Tramite il comando `objdump` è possibile ricavare informazioni più dettagliate sul file eseguibile, le quali possono essere salvate su un file di uscita che per convenzione ha estensione `.dis`. In particolare si possono conoscere i dettagli elencati di seguito.

- Sezioni in cui è diviso l'eseguibile da allocare e caricare.
  Si riconoscono:
  * la `.init`, contenente il codice di inizializzazione da eseguire all'accensione, costituito dalla *system call* di `_startup` per inizializzare lo *Stack Pointer* e la `_interrupt_table`. Quest'ultima è un array di puntatori per la gestione delle eccezioni, quali condizioni anomale e *interrupt requests* dall'esterno;
  * la `.text`, contenente il codice eseguibile delle istruzioni del programma `main` e delle funzioni di inizializzazione degli algoritmi;
  * la `.rodata`, contenente le costanti, ossia le variabili accessibili in sola lettura;
  * la `.data`, contenente dalle variabili globali del programma;
  * la `.bss`, contenente le variabili statiche inizializzate a zero.

  Per ciascuna sezione è indicata la dimensione in byte e l'indirizzo di partenza. Esistono due tipi di indirizzi:
  - Il *Virtual Memory Address* (VMA) è l'indirizzo a cui dovrà risiedere la sezione quando il programma è in esecuzione. In altre parole, quando viene invocata una funzione, l'indirizzo a cui viene fatto riferimento è il VMA.
  - Il *Load Memory Address* (LMA) è l'indirizzo riferito alla memoria in cui viene salvata una copia della sezione. Ad esempio, nel caso della sezione `.data`, questa inizialmente viene caricata all'interno della memoria codice. Allo *start* del programma, tale sezione viene copiata nella RAM dati e il programma farà riferimento a essa tramite il VMA. Il contenuto all'interno salvato nella memoria codice invece sarà associato al LMA, che in questo caso quindi differisce dal VMA. Durante l'esecuzione del programma, tutte le modifiche alle variabili globali saranno visibili nella RAM dati per mezzo del VMA della sezione.

La decisione relativa a quali indirizzi le sezioni devono essere mappate viene determinata dal *memory layout* definito nel *linker script*, che può essere riadattato a seconda delle esigenze e al variare delle dimensioni del design. Nel caso in esame, così come per tutti i test successivi, l'unica sezione che viene scritta anche nella memoria dati è la `.data`. Infatti, le righe in cui viene mappata la sezione *.data* contengono i valori di inizializzazione delle variabili globali, per cui le scritture della RAM sovrascrivono tali valori invece di sovrascrivere celle non inizializzate. In ogni caso, la corretta esecuzione del test non viene influenzata dalla decisione del *linker script* sull'allocazione delle risorse.

- *Symbol Table*. Ogni *entry* della tabella contiene l'indirizzo a cui il simbolo è mappato, e la sezione alla quale appartiene. Ciascun simbolo può corrispondere a una funzione di libreria o definita dallo sviluppatore, oppure a una variabile globale.

- *Disassembly* di ogni sezione. Per ciascuna sezione viene indicata la sequenza di istruzioni *assembly* che definiscono il contenuto di tutte le funzioni, ossia il contenuto delle funzioni di inizializzazione e del *main*. Ogni istruzione è riconoscibile dal fatto che, accanto ai byte che la definiscono, è presente lo mnemonico dell'istruzione.

Terminata la fase di *building*, segue la fase di *post-building* dell'eseguibile.
L'ultimo passaggio viene eseguito dal comando `objcopy`. Il comando, adattato per i processori della famiglia alla quale il core appartiene, permette di trasformare l'insieme di caratteri esadecimali del file `.elf` in un insieme di stringhe con il seguente formato:

Tale formato risulta utile all'ambiente SystemVerilog al fine di effettuare il caricamento del codice all'interno del microcontrollore.

3.3.4 Sviluppo del test UVM SystemVerilog

Parallelamente allo sviluppo del programma C, vengono implementate le classi in SystemVerilog all'interno dell'ambiente UVM. Il flusso perseguito è schematizzato in **Figura 3.9** ed è il medesimo anche per tutti i test successivi, salvo una differente implementazione delle classi di randomizzazione, sequenza e scoreboard.

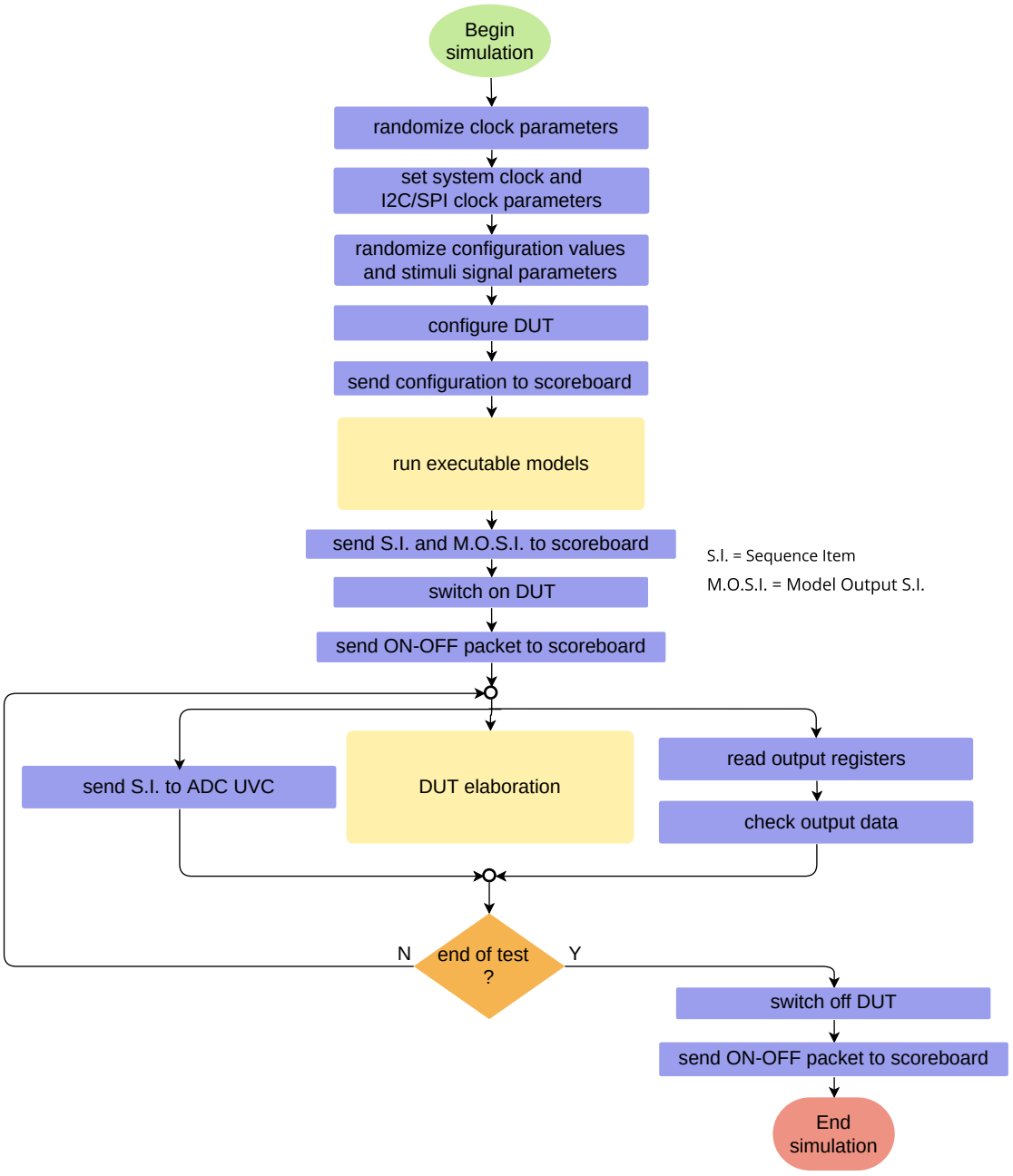


Figura 3.9: *Verification flow eseguito dalla simulazione in ambiente UVM [2, 5].*

Una volta ottenuto il file compilato contenente il codice eseguibile degli algoritmi sviluppati, questo può essere incluso all'interno delle funzioni della classe di randomizzazione assieme al suo *path* all'interno del *File System* per permetterne l'accesso durante la simulazione.
Da questo momento in poi, la gestione del test è demandata all'ambiente UVM SV, dall'impostazione e generazione dei parametri di configurazione per l'esecuzione del microcontrollore, sino all'analisi dei risultati.

Configurazione dei parametri

La sezione dedicata al blocco *Random Generator* è costituita dall'istanza della classe di randomizzazione. Per il test viene utilizzata una nuova classe che estende dalla classe base. All'interno della nuova classe, sono stati definiti una serie di *constraint* sui parametri principali del test, al fine di ottenere il comportamento desiderato.

- Vengono disabilitati i parametri che instradano alcuni segnali di *interrupt* sui *pad* esterni. In questo modo tutti i segnali che non sono di interesse non vengono presi in considerazione e non confliggono con l'instradamento di quelli utili.

- Viene abilitata la variabile che permette un incremento automatico dei pacchetti che si desidera inviare tramite interfaccia.

- Si definisce l'intervallo di valori possibili per l'ODR del microcontrollore. In particolare, si vuole che durante i test il microcontrollore sia sempre acceso al fine di permettere l'esecuzione senza interruzioni degli algoritmi nel core, pertanto tutti i valori della frequenza di funzionamento sono diversi da zero.

- Vengono assegnati gli algoritmi che si vuole vengano abilitati all'esecuzione.

Una volta definiti i *constraint*, si procede alla ridefinizione della funzione di `post_randomize()`. Come si può intuire dal nome, questa funzione viene automaticamente invocata dopo che la randomizzazione è stata eseguita e viene utilizzata per definire dei parametri in funzione della randomizzazione di altri, per sovrascriverne alcuni oppure per stampare a console informazioni utili sulla randomizzazione eseguita. Nel caso in esame, si associa una variabile di prescaler con valore differente sulla base dell'ODR che è stato generato. Dopodiché si assegna a una variabile stringa il percorso del file eseguibile all'interno del *File System*.
Siccome è stato compiuto l'*override* della funzione, per accedere al contenuto del metodo definito nella classe base, occorre includere la parola chiave *super*, in modo da poter riutilizzare le porzioni di codice che non sono state ridefinite.

Sequenza di test

La sequenza per questo primo test della RAM dati estende dalla classe associata alla sequenza per il test della catena introdotta nel **Paragrafo 2.2.2**.
All'interno della nuova classe si ridefiniscono una serie di task necessari alla corretta esecuzione delle operazioni di test.

- Si ridefinisce il task di accensione del microcontrollore. Essenzialmente, quando i sensori sono accesi vengono passati i parametri randomici generati in precedenza alle sequenze che gestiscono il comportamento dei sensori. Dopodiché viene impostato il clock del microcontrollore.

- Si ridefinisce un task che permette l'esecuzione in parallelo di più blocchi di codice, ovvero delle seguenti operazioni:

  · viene gestito il flusso di dati provenienti dai sensori verso i convertitori ADC; tali dati sono fondamentali per permettere l'avvio dell'esecuzione del test in quanto essi assumono la funzione di *trigger event*, come spiegato al **Paragrafo 1.4**.

  · viene avviato il task ideato per la gestione del cuore del test della RAM, identificato come `uc_ram_check()`.

  L'ultimo task introdotto viene definito sempre all'interno della sequenza, come indicato subito sotto.

- Si implementa *ex novo* il task `uc_ram_check()`.
  Nel caso dei test ideati per la memoria dati, tale task non presenta un livello di complessità elevato, a differenza della sequenza implementata per le *policy* dei registri, in cui la presenza di più modalità di accesso possibili ha reso la gestione della sequenza molto più articolata.
  Essenzialmente questo task attende che l'esecuzione del programma caricato nel core termini, testando il segnale di *sleep* interno al microcontrollore.
  Non appena tale segnale viene asserito, tramite le *analysis port* viene inviato allo scoreboard il pacchetto di accensione e spegnimento contenente i parametri di configurazione utili per il controllo dei risultati di simulazione. Dopodiché, tramite la *User Interface* viene eseguita la lettura dei bit di stato desiderati e dei registri di uscita tramite il task di lettura esposto al punto successivo.

- Viene ridefinito il task di lettura per i registri del microcontrollore. Questa funzione permette l'invio di pacchetti tramite interfaccia caratterizzati da:

  · l'indirizzo a cui leggere (*address*);

  · il numero di byte che si desidera leggere a partire da *address*;

  · il tipo di protocollo utilizzato;

  · il bit di *chip select*, il quale deve essere asserito per abilitare la trasmissione verso il DUT.

Il task di lettura prevede un preambolo e una conclusione le cui operazioni sono complementari: prima della generazione del pacchetto, occorre impostare la pagina a cui riferirsi con l'indirizzo. Nel caso in esame, si vuole accedere ai registri di stato appartenenti al banco di registri interni al microcontrollore. Di conseguenza la pagina di riferimento non è più la *default page* del dispositivo, bensì la pagina del microcontrollore. Per accedervi serve abilitare un bit all'interno del registro dedicato nella pagina del dispositivo. Terminate le letture, il task si conclude resettando il medesimo bit.

In **Figura 3.10** viene riportato a titolo esemplificativo un insieme di pacchetti visualizzabili nello spazio dedicato alle *waveform* del *tool* di simulazione. In particolare, si tratta della sequenza di comandi atti a ricavare i dati contenuti nel registro di *INT_STATUS* al termine dell'esecuzione del programma di test. Viene scritto il Byte `0x81` nel registro per la selezione della *page registers*, il cui effetto è quello di abilitare l'accesso alla pagina dei registri interni al microcontrollore, in cui risiede il registro da leggere. Il secondo pacchetto contiene le informazioni della lettura, ossia l'indirizzo a cui leggere, quanti byte leggere e il valore acquisito (in questo caso `0x00`, a indicare che nessun *interrupt* è stato generato, ossia nessuna condizione anomala si è verificata durante il test). Infine l'ultimo pacchetto informa sulla disabilitazione dell'accesso alla pagina, riportando l'indirizzamento dell'interfaccia alla situazione di *default*.
Nel test successivo vengono anche effettuate le letture sui registri *General Purpose*, al fine di rilevare i bit codificanti gli indirizzi di memoria della RAM.
Al di sotto dei pacchetti, i segnali comandati dagli UVC di interfaccia che si muovono in modo coerente con il *timing* del protocollo utilizzato (I2C in questo caso).

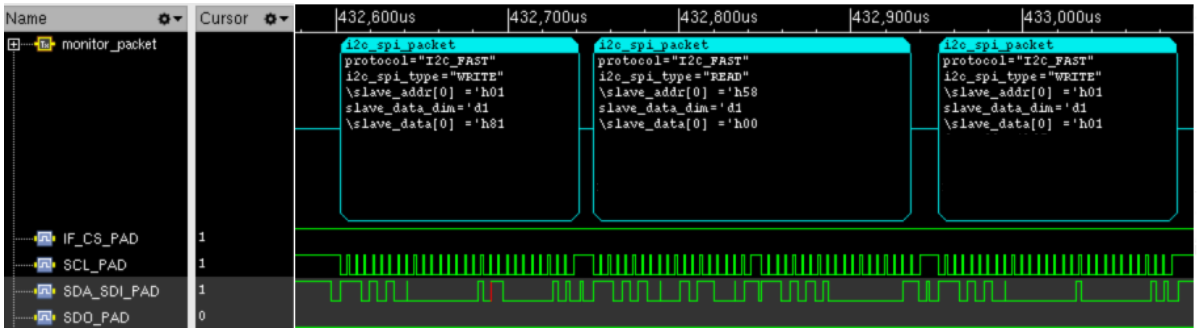


Figura 3.10: *Sequenza di pacchetti contenenti i comandi per accedere alla pagina dei registri del microcontrollore e leggere il registro di stato degli interrupt.*

- Infine, vengono passati ai pacchetti di configurazione i parametri utili allo scoreboard per determinare se i sensori sono accesi e quali algoritmi sono abilitati.

Scoreboard

Le informazioni raccolte attraverso i comandi dati dalla sequenza sono istradate verso lo scoreboard.
All'interno di esso agiscono due funzioni, indicate di seguito.

- La funzione associata alla scrittura dei pacchetti di accensione e spegnimento. All'interno di questa è possibile inizializzare gli attributi dell'istanza scoreboard, quali contatori e variabili di offset (indirizzi di partenza, posizioni in numero di bit per lo *shifting* di alcuni bit di controllo, etc.), il tutto sulla base di quale algoritmo è stato abilitato.

- La funzione di scrittura dei pacchetti ricevuti provenienti dai *monitor* degli UVC di interfaccia.
  Trattasi di una funzione invocata automaticamente ogni qualvolta che un pacchetto viene ricevuto dallo scoreboard attraverso le *analysis port*. La sua implementazione costituisce il fulcro del funzionamento dello scoreboard.
  Questa funzione riporta, tramite un loop sui pacchetti ricevuti, le seguenti informazioni:

  · l'indirizzo del registro letto dall'interfaccia;

  · il dato contenuto all'interno di quel registro.

  Il controllo avviene innanzitutto verificando che la variabile che definisce la modalità di trasferimento associata ai pacchetti sia pari a un valore che indichi di essere in lettura; se la condizione è verificata, per ciascun pacchetto ricevuto si verifica se l'indirizzo sia quello associato al registro di stato. In caso affermativo, sulla base di quali bit sono stati impostati a '1', viene emesso un messaggio di errore che riporta quale o quali blocchi della RAM che sono stati testati contengono al loro interno dei valori inattesi. Siccome ogni algoritmo è mappato su un *chunk* differente, l'*output* di simulazione informa su quali algoritmi hanno fallito. In questo modo, eseguendo il test descritto al paragrafo successivo, vengono fatti eseguire singolarmente solo gli algoritmi che agiscono sulle porzioni di RAM che falliscono, per rilevare gli indirizzi delle righe che sbagliano.

3.3.5 Simulazioni e risultati ottenuti

La simulazione segue i passaggi indicati di seguito.
Inizialmente viene programmata la memoria codice, scrivendo da interfaccia i byte associati al codice eseguibile il cui percorso nel *File System* è mappato nella classe di randomizzazione e su cui è stato eseguito il *parsing*.
Terminata la configurazione del microcontrollore, avviene il *boot*, le catene dei sensori vengono attivate e le operazioni del core partono a seguito del *trigger event* che segnala la presenza di un dato in ingresso elaborato dalle catene dei sensori.
Il task di `body()` della sequenza associata al test della RAM invece lancia più task in

parallelo, per cui si occupa di gestire i dati dei sensori e la sequenza di operazioni da effettuare sul microcontrollore quando questo è in esecuzione. In questo caso si leggono i registri di stato degli *interrupt* al segnale di *sleep* e si spegne il dispositivo, terminando la simulazione.

Le simulazioni hanno dimostrato la correttezza del design, in quanto nessuna scrittura e lettura all'interno delle celle della RAM ha riportato problemi. In **Figura 3.11** si riporta a titolo esplicativo una porzione dell'evoluzione delle *waveform* che tracciano l'evoluzione del contenuto di alcune righe della memoria dati.

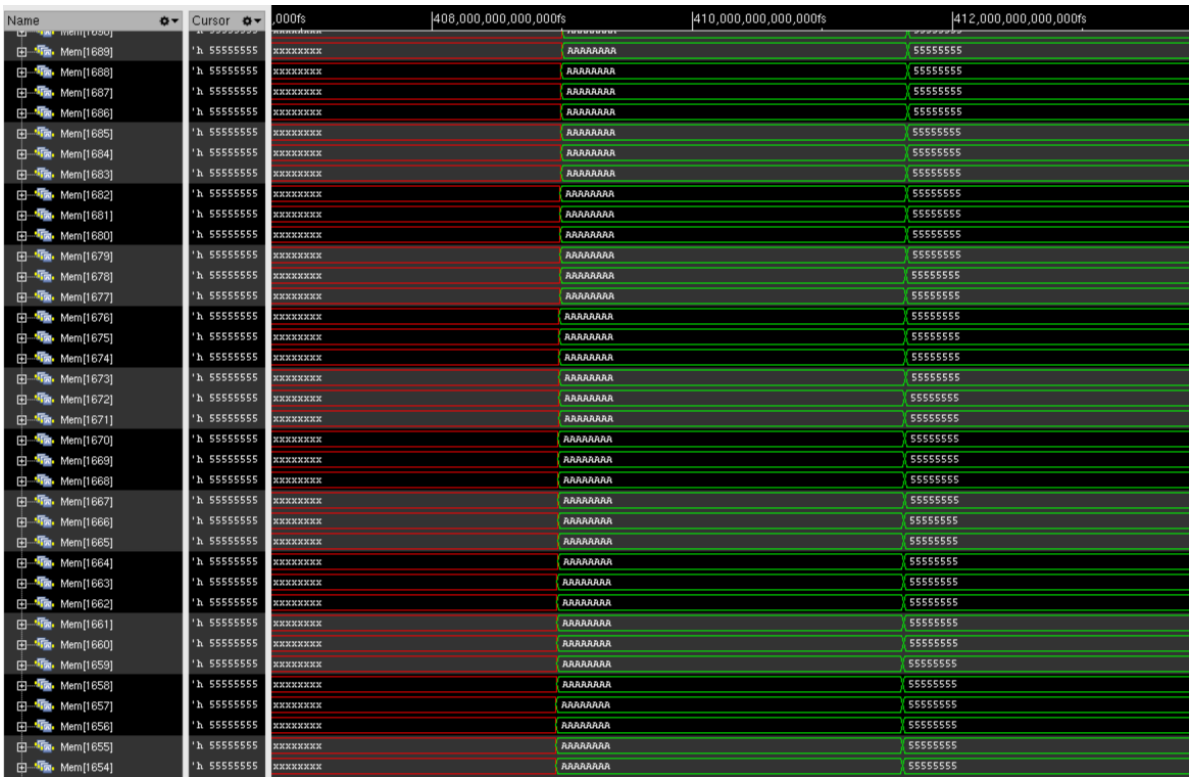


Figura 3.11: *Dettaglio dell'evoluzione di alcune righe della memoria dati.*

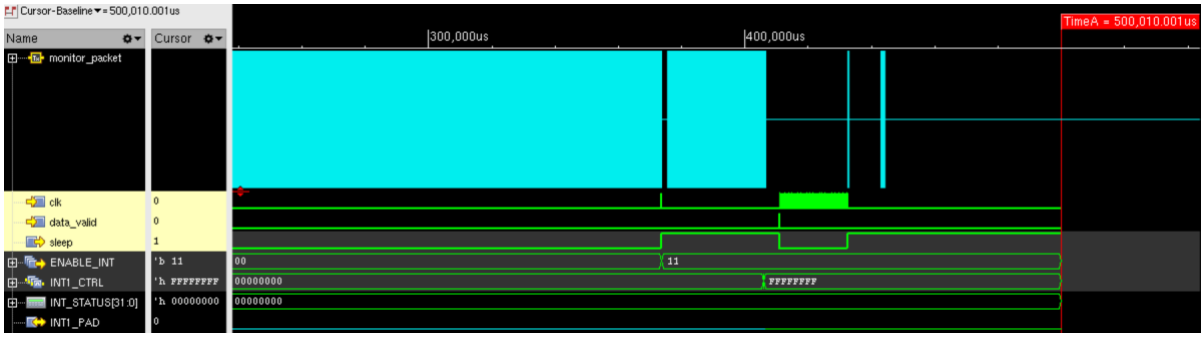
Lato UVM è stato verificato che la sequenza operasse correttamente, in quanto tutte le letture richieste sui registri di uscita sono avvenute rispettando le tempistiche e l'ordine richiesto, come indicato il **Figura 3.12**.
Si possono riconoscere le varie fasi in cui si articola la simulazione:

- Il monitor degli UVC di interfaccia raccoglie una sequenza iniziale di pacchetti di scrittura per la configurazione del microcontrollore.
- Terminata la fase di configurazione, viene abilitato il clock per inizializzare le variabili globali del programma caricato e le variabili di gestione degli algoritmi. Vengono anche abilitate le linee di *interrupt* esterne.
- Dopodiché avvengono le scritture della fase di *boot* che terminano con l'abilitazione dei flag di controllo per instradare gli *interrupt* generati dal core verso il *pad* INT1.

- Al *data_valid* il microcontrollore si riaccende e inizia l'esecuzione delle istruzioni del test caricato nella RAM codice.

- Quando le istruzioni sono state tutte eseguite, viene generato il segnale di *sleep*, il clock si spegne e l'interfaccia ri-acquisisce il controllo, inviando comandi di lettura per rilevare il valore dei bit del registro *INT_STATUS*.

- Infine, la sequenza termina la simulazione inviando dei comandi di scrittura che determinano lo *switch off* dell'intero dispositivo.




```
1  UVM_INFO @ 405772013.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic_seq.scoreboard_inst] Algos enable complete
2  UVM_INFO @ 405875280.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic_seq.scoreboard_inst] Algos odr scaler complete
3  UVM_INFO @ 406153867.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic_seq.scoreboard_inst] INT1_CTRL and INT2_CTRL write complete
4  UVM_INFO @ 406491891.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic_seq.scoreboard_inst] Enable int complete
5  UVM_INFO @ 406638988.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic_seq.scoreboard_inst] Microcontroller start complete
6  UVM_INFO @ 432596850.50 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst] sleep
        asserted
7
8  UVM_INFO @ 432935415.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] No errors have been found during RAM data reading by algo           0 testing
       RAM chunk #          0
9  UVM_INFO @ 432935415.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] No errors have been found during RAM data reading by algo           1 testing
       RAM chunk #          1
10 . . .
11 UVM_INFO @ 432935415.00 ns: uvm_test_top.sve_top.scoreboard_inst [scoreboard_inst] RAM is working as
       expected. All the rows have been tested correctly.
12
13 UVM_INFO @ 433092241.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
       switch off
```

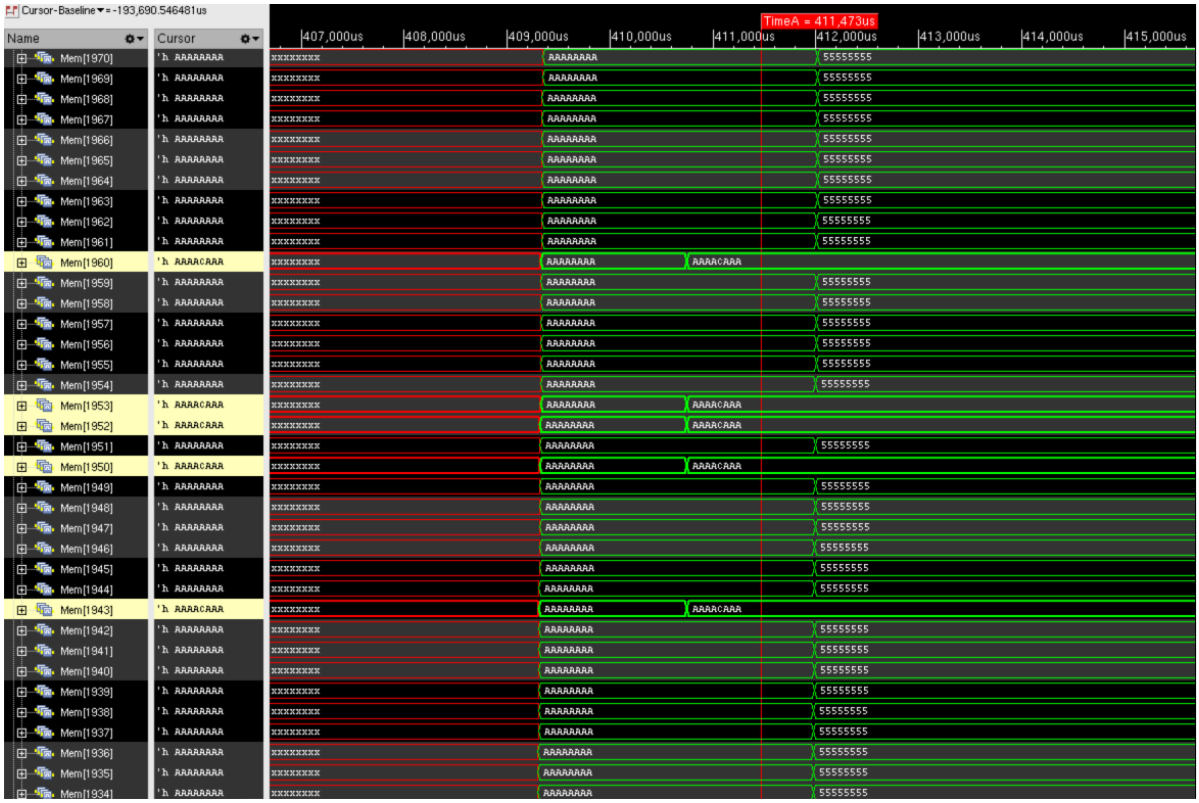
Figura 3.12: *Sequenza temporale delle operazioni di base del test e messaggi di log corrispondenti alle varie fasi di simulazione.*

Per completezza, al fine di avere la certezza della correttezza del test implementato, sono state svolte due ulteriori operazioni.

- È stata invertita la condizione di test al fine di far fallire apposta il test. In questo modo si è verificato, come ci si aspettava, che tutte le righe fossero disallineate rispetto al dato atteso.
  Lato segnali si è osservato che i registri *General Purpose* di uscita avessero al loro interno tutti i bit asseriti a '1', a significare che il core avesse rilevato correttamente i comportamenti inattesi.
  Lato UVM è stato possibile constatare la correttezza del meccanismo di decodifica implementato e la sincronizzazione dei pacchetti ricevuti attraverso le *analysis port* dallo scoreboard contenenti le informazioni da decodificare.

- Tenuto conto dei tempi di simulazione richiesti, è stato eseguito manualmente il *forcing* di alcune celle della RAM scegliendole in modo puramente arbitrario tra quelle presenti. Tramite lo script *tcl* fornito in ingresso al simulatore, viene effettuato il *run* sino a *tot_run* µs, ovvero fermandosi prima che il core abbia eseguito la seconda scrittura sulle righe su cui si desidera agire. A quel punto la simulazione si ferma ed è possibile agire da interfaccia grafica, oppure tramite comandi *tcl* da console, per modificare i byte in esadecimale contenuti nelle celle di memoria. Dopodiché la simulazione viene fatta ripartire e prosegue sino a che il test non spegne il dispositivo.
  Anche in questo caso il test si comporta correttamente, riportando le righe che sbagliano all'interno del file di *log*, ossia le righe su cui è stato forzato un valore diverso da quello atteso dal test.

In **Figura 3.13** sono visibili le righe su cui è stato applicato il *force* per la modifica di alcuni bit e i corrispettivi messaggi di errore riportati dopo la decodifica eseguita dallo scoreboard.




```
1 UVM_INFO @ 414514635.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
      scoreboard_inst] No errors have been found during RAM data reading by algo   0
      testing RAM chunk #        0
2 UVM_INFO @ 414514635.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
      scoreboard_inst] No errors have been found during RAM data reading by algo   1
      testing RAM chunk #        1
3 UVM_INFO @ 414514635.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
      scoreboard_inst] No errors have been found during RAM data reading by algo   2
      testing RAM chunk #        2
4 UVM_ERROR @ 414514635.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Bit     3 of INT_STATUS has been set to 1. Unexpected values have
      been detected during RAM data reading by algo      3 testing RAM chunk #        3.
      Please enable this algo alone to see the rows that do not match.
5 UVM_INFO @ 414514635.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
      scoreboard_inst] No errors have been found during RAM data reading by algo   4
      testing RAM chunk #        4
6 . . .
```

Figura 3.13: *Esempio di simulazione che fallisce apposta per validare il funzionamento del testbench.*

3.4 Test RAM #2: verifica a blocchi con algoritmi isolati

Questo secondo test serve per avviare l'esecuzione degli algoritmi singolarmente a seconda delle esigenze. In particolare, l'esecuzione di questo test può essere effettuata nel caso in cui il primo test presentasse dei fallimenti per specifici sotto-blocchi. Se un *chunk* non contiene alcun *mismatch* e il registro di stato non ha asserito il bit di *flag* corrispondente, significa che il test ha avuto successo, pertanto nel lancio della prossima simulazione non serve ri-testarlo. Viceversa, se un sotto-blocco fallisce, le informazioni sulle righe che falliscono vanno riferite al designer; una volta apportate le modifiche, il *chunk* di memoria modificato va nuovamente verificato. In tal caso basta istruire il core a far eseguire soltanto l'algoritmo associato al sotto-blocco, per cui è stata implementata anche questa seconda possibilità per soddisfare questa esigenza. Nella classe di randomizzazione, al posto di utilizzare il *constraint* che abilita NUM_ALGOS, viene invocata una funzione di assegnazione apposita, nella quale impostare il parametro di abilitazione con un bit randomico tra le scelte possibili, oppure impostando il bit desiderato corrispondente all'algoritmo che si vuole abilitare.

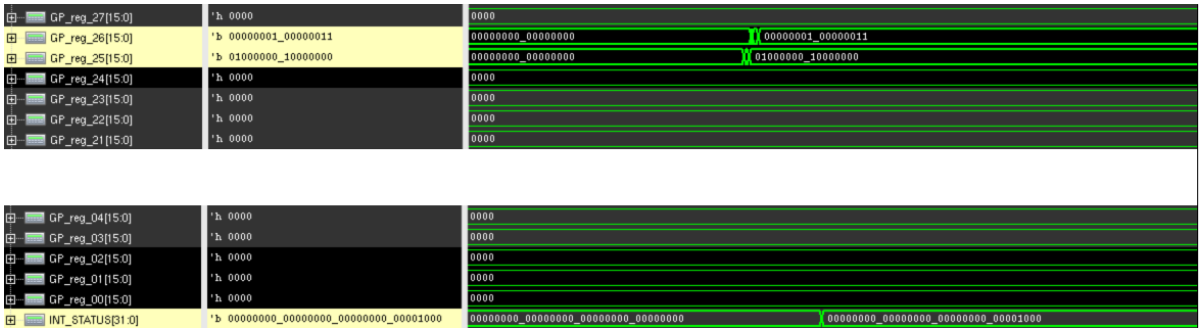
3.4.1 Obiettivi del test

In questo test viene abilitato un singolo algoritmo, per cui i risultati di *chunk* diversi sono ottenuti in simulazioni separate. L'obiettivo ora è generare nei bit dei registri di uscita l'informazione relativa alle righe disallineate, tramite il meccanismo di codifica e decodifica ampiamente descritto nei paragrafi precedenti.
Siccome sono salvati gli indirizzi relativi alle *word* che falliscono, è possibile sfruttare questo test per rilevare dei bachi nel design. Grazie alla conoscenza della posizione alla quale si è verificato l'errore, è possibile risalire alla riga che presenta problemi e informare il designer di riferimento, il quale a sua volta è nella condizione di applicare eventuali correzioni a livello RTL.

Non è stato preso in considerazione il conteggio del numero totale di bit errati in quanto al momento non è un dato interesse, trattandosi di un test sviluppato per avere una informazione preliminare sulla correttezza del design ad alto livello. Nel paragrafi successivi invece il dato sul conteggio dei disallineamenti è cruciale per le finalità di utilizzo di quei test. Avendo dei test portabili per la fase di *Final Test* su Silicio, il numero di bit fisici errati può essere comparato a un valore di soglia scelto dall'ingegnere di *Testing*, il quale deciderà, a seconda del risultato, di mantenere o scartare il pezzo.

3.4.2 Simulazioni e risultati ottenuti

In **Figura 3.14** si riporta il risultato dell'esecuzione del test applicato solo sul blocco che ha riportato degli errori, su cui è stato ripetuto il medesimo *pattern* di bit di *force* applicato al test precedente. Coerentemente con quanto atteso, vengono stampate le posizioni delle righe che contengono i bit forzati, con la posizione del bit che codifica la riga corrispondente all'interno dei segnali di uscita, in questo caso il segnale 25 e 26 sono quelli che entrano in gioco per il salvataggio. Lo scoreboard UVM segnala gli errori stampando le righe che sono state forzate con bit errati.
Le posizioni dei bit codificanti le righe che falliscono sono stampati assieme ai messaggi di errore. Essi corrispondono ai bit asseriti all'interno dei registri *General Purpose* evidenziati nelle *waveform*.




```
UVM_ERROR @ 414505520.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
    .scoreboard_inst] Bit        3 of INT_STATUS has been set to 1. Unexpected values
    have been detected during RAM data reading in chunk #      3
UVM_ERROR @ 417192408.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
    .scoreboard_inst] Mismatch in RAM has been found at row      1943 (bit    7 inside
    GP reg      25)
UVM_ERROR @ 417192408.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
    .scoreboard_inst] Mismatch in RAM has been found at row      1950 (bit   14 inside
    GP reg      25)
UVM_ERROR @ 417192408.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
    .scoreboard_inst] Mismatch in RAM has been found at row      1952 (bit    0 inside
    GP reg      26)
UVM_ERROR @ 417192408.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
    .scoreboard_inst] Mismatch in RAM has been found at row      1953 (bit    1 inside
    GP reg      26)
UVM_ERROR @ 417192408.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
    .scoreboard_inst] Mismatch in RAM has been found at row      1960 (bit    8 inside
    GP reg      26)
```

Figura 3.14: *Effetto del forcing di alcuni bit del terzo blocco di RAM sui registri di uscita, applicando l'esecuzione isolata dell'algoritmo associato al blocco.*

3.5 Test RAM #3: verifica di *mismatching bits* nelle singole celle

3.5.1 Obiettivi del test

Questo nuovo test eseguito sulla memoria dati consiste in una verifica più accurata, ossia in grado di riportare un numero maggiore di informazioni a colui il quale faccia utilizzo del test. In particolare, esso riserva dei registri di uscita per la codifica del conteggio del numero di bit cumulativi che falliscono. In questo caso l'obiettivo è verificare non solo i disallineamenti delle *word* su ciascuna riga, ma applicare una *bitmask* bit per bit al fine di conteggiare il numero totale di *mismatch* cella per cella.
Trattandosi di un test nel quale occorre effettuare le operazioni base di *shifting* e comparazione *bitwise* iterativamente su tutte le *word*, la durata del test ragionevolmente risulterà molto superiore alla tipologia di test applicata in precedenza. Al di là di questo dettaglio, si hanno numerosi vantaggi in termini di portabilità, dal momento che l'informazione sul numero di disallineamenti in fase di *Final Test* è più importante rispetto a quella sulla posizione delle singole righe che sbagliano (informazione che invece veniva riportata dai test iniziali). Infatti, applicando le routine sviluppate sul design fisico in Silicio è possibile capire se la memoria RAM presenta delle difettosità nelle celle, per cui i valori binari al loro interno risulterebbero corrotti e la funzionalità del microcontrollore verrebbe inevitabilmente compromessa. Sebbene l'implementazione delle celle fisiche possa in generale differire da quella a livello RTL su cui il test viene eseguito, l'algoritmo prodotto in C è facilmente adattabile grazie alla sua modularità, come descritto nei *flow chart* del paragrafo successivo.

3.5.2 Sviluppo del test C

In particolare questo test segue i *pattern* riportati qui sotto, i quali permettono la totale copertura di tutti i possibili casi di *toggling*, ossia l'inversione di bit adiacenti per colonne e per righe.

- Scrittura alternata riga per riga delle costanti su 32 bit `0xAAAAAAAA` e `0x55555555`. In questo modo viene coperto il *toggling* delle celle adiacenti poste su righe consecutive nel design RTL.

- Lettura riga per riga e comparazione dei valori letti con quelli attesi nelle varie posizioni. In questo caso il *checking* avviene bit per bit, in modo da salvare in modo incrementale il numero di bit che non soddisfano la condizione di uguaglianza. Per rendere questa informazione più significativa, nella seconda parte di questo test i *mismatch* vengono salvati su un registro di uscita diverso da quello utilizzato a questo *step*, in modo da separare l'informazione sul primo gruppo di scritture-letture-check dal secondo.

- Inversione dei valori delle costanti scritte su ciascuna riga per riscrivere, nell'ordine e in modo alternato, le costanti `0x55555555` e `0xAAAAAAAA`. Così facendo vengono

coperti i *toggling* di celle adiacenti appartenenti alla medesima riga.

- Rilettura riga per riga e controllo: in questo caso eventuali *mismatching bits* vengono salvati su un nuovo registro libero di uscita.

Similmente ai primi due test, sono previste due iterazioni, ciascuna costituite dalle tre fasi di scrittura-lettura-controllo (**Figura 3.15**).

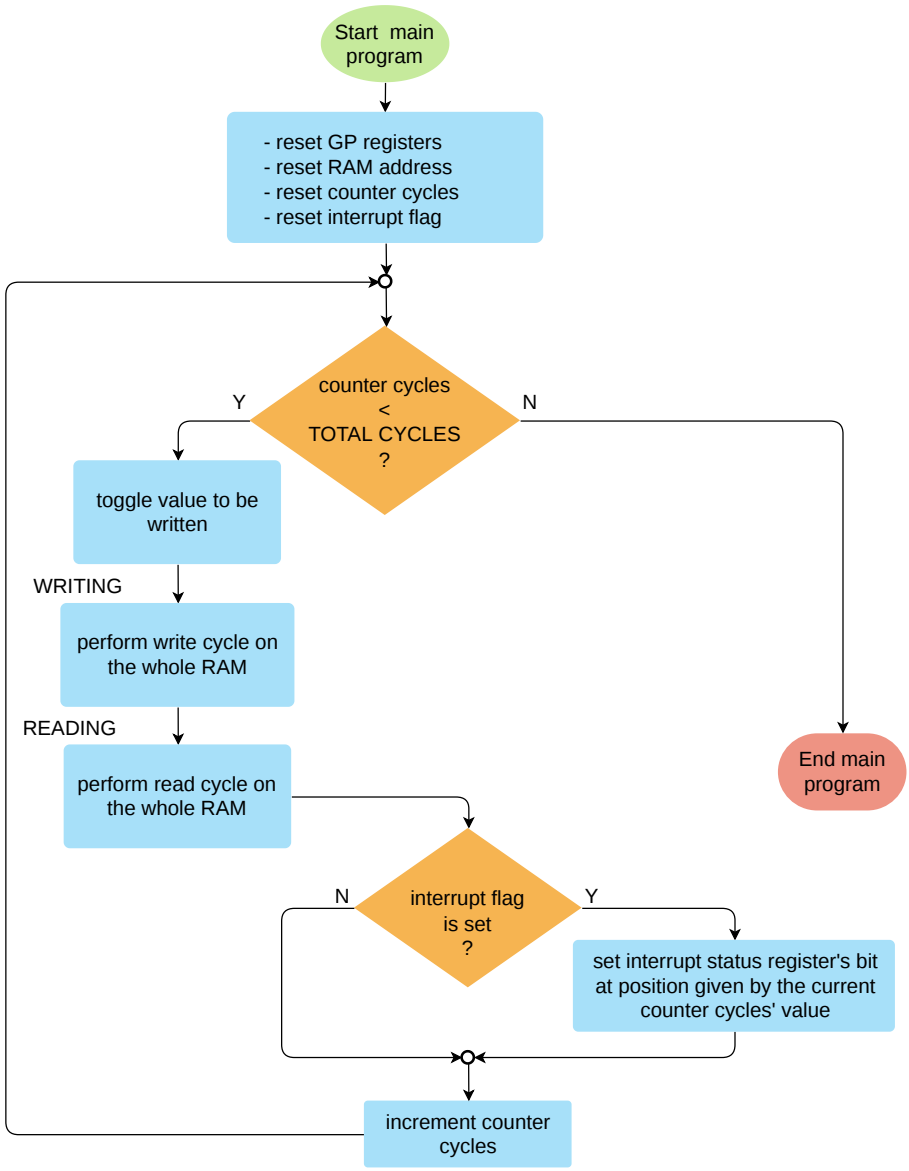


Figura 3.15: *Struttura del programma di test basato su più cicli di scrittura-lettura-controllo.*

Al termine, vengono scritti dei *flag* di *interrupt* nel registro di stato per informare su quale iterazione presenti almeno un fallimento: se vengono conteggiati dei *mismatch* in entrambe le iterazioni, i primi due bit vengono asseriti.

La fase di scrittura (**Figura 3.16**) inverte i bit della costante da scrivere sulla base del numero di riga: se la riga è a un indirizzo pari, nessuna inversione dei bit è effettuata; viceversa, se la riga è dispari, tutti i bit della costante vengono invertiti, in modo da ottenere una matrice costituita da righe con bit invertiti.

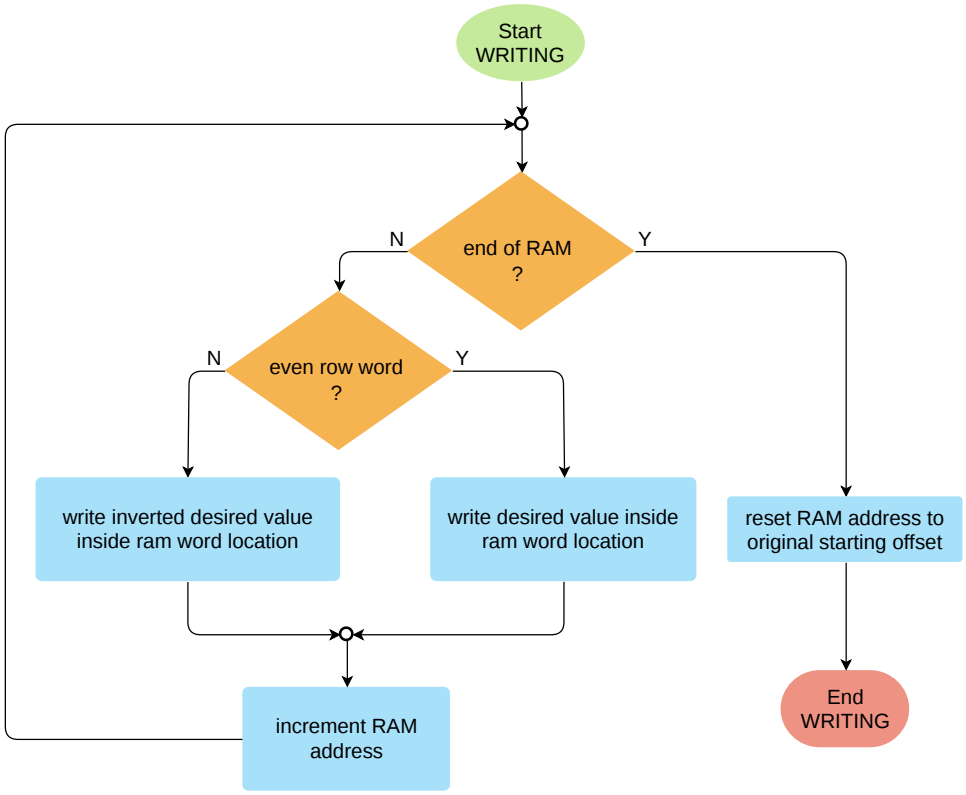


Figura 3.16: *Scrittura di valori a bit invertiti per righe alternate.*

La fase di lettura e controllo (**Figura 3.17**) testa il contatore di ciclo per verificare se la riga è pari o dispari, analogamente al ciclo di scrittura.

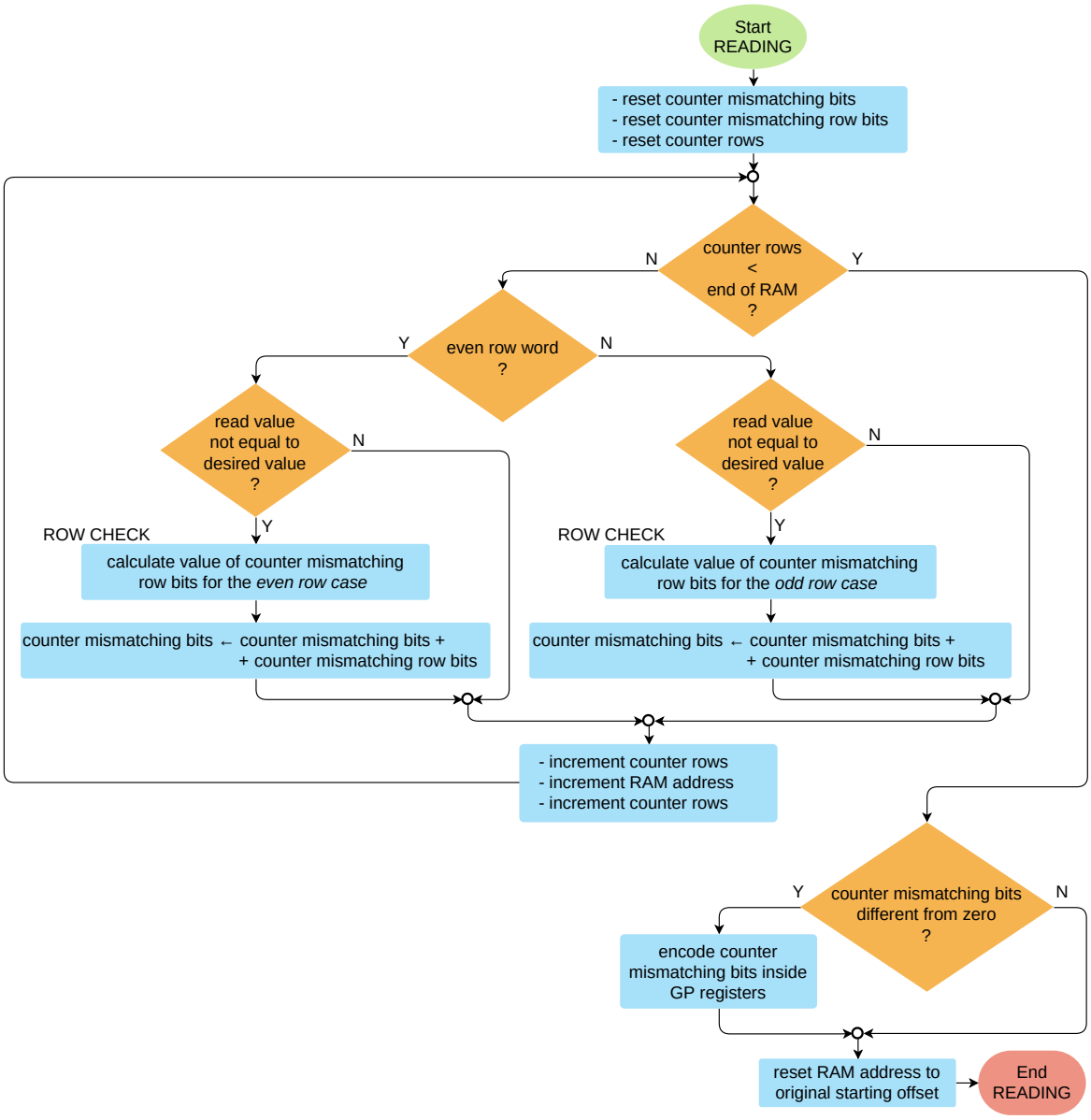


Figura 3.17: *Lettura delle word e conteggio dei bit che falliscono in caso di righe disallineate.*

- Se la riga è pari, si verifica se la *word* letta corrisponde al valore `0xAAAAAAAA`. Qualora il dato letto presenti dei bit disallineati, allora si procede con il conteggio dei bit che falliscono su quella riga. Il conteggio viene spiegato nel dettaglio nel diagramma alla **Figura 3.18** ed è implementato in una funzione apposita identificata come `row_check()`. Il risultato viene utilizzato come incremento della variabile di conteggio dei *mismatch* totali.

- Se la riga è dispari, si compara il valore letto con la costante `0x55555555`. In caso di *mismatch*, si procede in modo analogo al caso precedente, con l'aggiornamento della variabile di conteggio dei bit errati al termine.

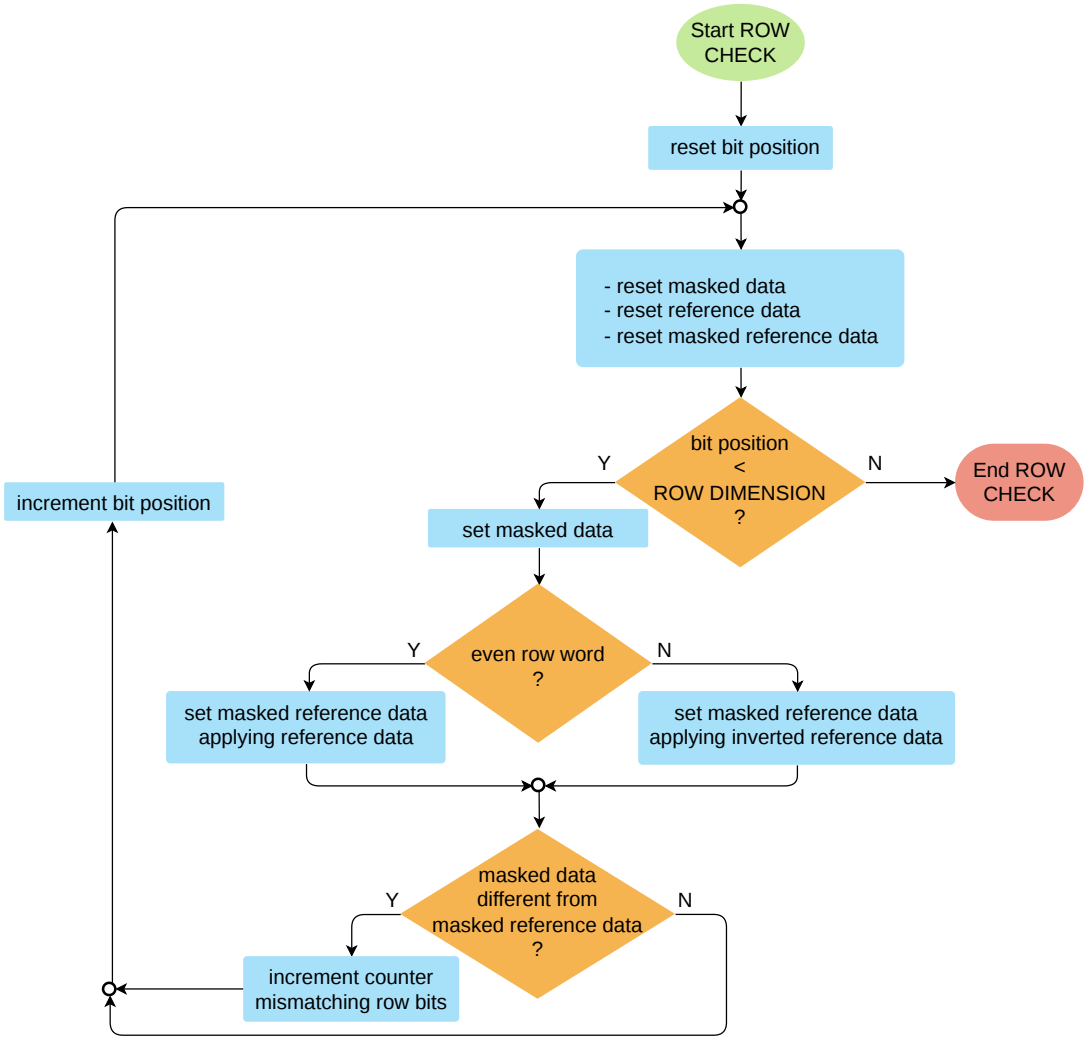


Figura 3.18: *Algoritmo di controllo riga.*

La funzione di `row_check()` si basa sull'applicazione di una maschera che permette di testare un bit per volta sui 32 bit acquisiti. Si definisce la maschera tramite uno *shift*

verso sinistra per selezionare bit che si desidera testare, dopodiché la maschera viene applicata sia al valore letto sia al valore desiderato. In questo modo si mantiene in ciascuna variabile soltanto il bit che si vuole verificare. Se i due bit differiscono, significa che un disallineamento è stato rilevato e si incrementa la variabile di conteggio. La porzione di codice per il controllo di riga è riportata in **Figura 3.19**.

```
static uint32_t row_check(uint32_t row_data, uint8_t choice) {
    uint32_t retval = 0;
    uint32_t mask = DATA_RST;
    uint32_t data_masked = DATA_RST;
    uint32_t data_ref = row_data;
    uint32_t data_ref_masked = DATA_RST;

    for (int i = 0; i < DIM_ROW; i++) {
        mask |= 1 << i;
        data_masked |= (*(ram_address) & mask);

        switch (choice) {
            case FIRST_CHOICE :
                data_ref_masked |= (data_ref & mask);
                if (data_masked != data_ref_masked) {
                    retval++;
                }
                break;
            case SECOND_CHOICE :
                data_ref_masked |= (~data_ref & mask);
                if (data_masked != data_ref_masked) {
                    retval++;
                }
                break;
            default:
                break;
        }
        mask = DATA_RST;
        data_masked = DATA_RST;
        data_ref_masked = DATA_RST;
    }
    return retval;
}
```

Figura 3.19: *Controllo bitwise di tutti i bit di una riga generica. La variabile* `choice` *permette di invertire il valore del bit desiderato contro cui eseguire la comparazione, sulla base della macro-iterazione corrente di scrittura-lettura-controllo.*

Alla seconda iterazione di scrittura-lettura-controllo viene modificata la *word* di riferimento per le scritture alle righe pari e dispari tramite il *toggling* dei bit. In questo modo viene anche coperto il caso di inversione di bit appartenenti alla stessa riga.

Il salvataggio del risultato finale tiene conto della dimensione della variabile di conteggio, definita su 32 bit. Siccome i registri di uscita sono solo su 16 bit, il risultato viene suddiviso su due registri consecutivi nel caso in cui il valore del contatore superi il massimo valore rappresentabile su 16 bit[1]. Inoltre, il controllo viene eseguito due volte e a ciascun controllo è associato un risultato di conteggio, pertanto occorrono in tutto 4 registri di uscita, due per ogni risultato che si vuole salvare.
La decodifica nello scoreboard deve quindi tenere conto di questo aspetto, come riportato al **Paragrafo 3.5.3**. La porzione di codice che segue effettua il controllo dimensionale sul risultato per evitare l'*overflow* sui registri di uscita nei quali avviene il salvataggio.

```
1  /* #define cast_uint16_t(address_reg_name) *((volatile uint16_t *)(address_reg_name))
       */
2  if (cnt_mismatching_bits != 0) {
3      if (num_cycle == 0) {
4          if (cnt_mismatching_bits > MAX_GP_VAL) {
5              cast_uint16_t(GP_reg_00) = MAX_GP_VAL;
6              cast_uint16_t(GP_reg_01) = cnt_mismatching_bits - MAX_GP_VAL; // encode
       MSBs
7          } else {
8              cast_uint16_t(GP_reg_00) = cnt_mismatching_bits;
9              cast_uint16_t(GP_reg_01) = 0;
10         }
11     } else {
12         if (cnt_mismatching_bits > MAX_GP_VAL) {
13             cast_uint16_t(GP_reg_02) = MAX_GP_VAL;
14             cast_uint16_t(GP_reg_03) = cnt_mismatching_bits - MAX_GP_VAL; // encode
       MSBs
15         } else {
16             cast_uint16_t(GP_reg_02) = cnt_mismatching_bits;
17             cast_uint16_t(GP_reg_03) = 0;
18         }
19     }
20     retval = SET_STATUS;
21 }
```

In questo caso il valore massimo rappresentabile in uscita vale `MAX_GP_VAL = 0xFFFF`. Inoltre, se almeno un disallineamento viene rilevato in una delle due macro-iterazioni, viene impostato un bit di *flag* che permette la generazione sul *pad* esterno di un segnale di *interrupt*, qui indicato come variabile di ritorno della funzione che esegue la lettura e il controllo. Analogamente a quanto avveniva nel primo test, viene abilitato il bit nel registro di stato alla posizione corrispondente al numero di ciclo corrente di test. Tutti i bit di tale registro sono instradati con una OR logica verso la linea esterna di *interrupt*, mentre il dato su quale bit viene asserito è sfruttato dall'ambiente UVM per informare in merito a quale o quali iterazioni presentano errori.

[1] È stata utilizzata l'implementazione con variabile di conteggio a 32 bit sulla base del *worst case*, ossia per essere in grado di salvare il numero totale di celle che falliscono nel caso in cui tutte le celle non si comportino in modo corretto. Tuttavia, va tenuto presente che tale eventualità nella pratica è estremamente improbabile si realizzi, in quanto significherebbe che tutto il design risulta errato.

3.5.3 Sviluppo del test UVM SystemVerilog

Configurazione dei parametri

Per questo test, così come per il successivo al **Paragrafo 3.6.3**, la classe di randomizzazione estende da quella definita per il primo test della RAM, in quanto tutte le *features* in essa implementate sono riutilizzate anche in questi test. Viene ridefinita la funzione che assegna gli algoritmi che si vuole vengano abilitati, dal momento che in questo caso basta un solo algoritmo abilitato, all'interno del quale tutta la RAM può essere testata senza saturare le uscite con i risultati.

Sequenza di test

La sequenza, così come la sequenza del test al **Paragrafo 3.6.3**, eredita i task definiti nella sequenza del test iniziale. Viene sovrascritto il task di `uc_ram_check()` per la gestione dell'esecuzione dell'algoritmo C nel microcontrollore e dei risultati di processamento.
Si attende lo spegnimento del microcontrollore, dopodiché, se soltanto il primo algoritmo è stato abilitato dalla randomizzazione, si procede con la lettura del registro di stato degli *interrupt* e dei primi quattro registri di uscita, ossia quelli nei quali sono salvati i conteggi dei *mismatch* delle due iterazioni di scrittura-lettura-controllo.

Scoreboard

Lo scoreboard non eredita alcun metodo dalla classe definita nel test iniziale. Quando le *analysis port* ricevono un pacchetto, si verifica se si tratta di una lettura dei registri di uscita oppure del registro di stato.

- Se l'indirizzo è associato a un registro di uscita, si decodifica in decimale il valore binario in esso contenuto. Se il valore finale risulta essere superiore a una certa soglia, si segnala l'errore. La soglia viene decisa dal *Product Engineer* e serve per il test della RAM nel design fisico su Silicio, al fine di discriminare quali prodotti contengano un numero di difettosità troppo elevato e quali invece ne presentano un numero accettabile.

- Se l'indirizzo è relativo al registro di stato, si testano i suoi bit per segnalare la presenza di errori.

Sulla base del salvataggio in C, in cui si riservano due registri per ogni risultato, la decodifica avviene come riportato nelle istruzioni SystemVerilog seguenti, in cui si considera la decodifica della prima iterazione.

```
    outreg = i2c_spi_packet_inst.slave_data[pkt_n];
    cnt_mismatches_1st_it = cnt_mismatches_1st_it + decode_output(outreg,
    cnt_acquisitions);
    cnt_acquisitions = cnt_acquisitions + 1;
    . . .
    // end foreach
// end function
virtual function int decode_output (bit [`dim_reg-1:0] outreg, int cnt_acquisitions);
        int retval = 0;
        int offset_exp = (`dim_reg/2)*cnt_acquisitions;

        for (int i = 0; i < `dim_reg/2; i++) begin
            retval = retval + (outreg[i] * 2**(offset_exp + i));
        end
        return retval;
endfunction : decode_output
```

L'oggetto indicato come `i2c_spi_packet_inst` estende dalla classe standard `uvm_sequence_item` e contiene al suo interno tutte le informazioni del comando eseguito dalla sequenza sull'interfaccia. La variabile `cnt_acquisitions` vale 0 se il pacchetto ricevuto è associato al primo dei due registri che contengono il risultato della prima iterazione di test, altrimenti vale 1. Si tratta di un peso con il quale è possibile, durante la decodifica, risalire al peso corretto dei bit di quel registro.

3.5.4 Simulazioni e risultati ottenuti

Sono state eseguite una serie di simulazioni atte a verificare la correttezza design e al tempo stesso provare il corretto funzionamento del test sia nel caso di nessun fallimento della RAM sia nel caso in cui vengano rilevati dei disallineamenti del dato letto rispetto a quello atteso.
La verifica è stata eseguita sul design a livello RTL, tuttavia tra le finalità del lavoro vi è anche quella di sfruttare le routine di test anche sul design fisico. Per questo motivo il numero di *mismatch* rilevati nelle celle risulta essere un dato rilevante, che può essere messo a confronto con un valore di soglia massima di errore per permettere all'ingegnere di Test di decidere se mantenere il pezzo oppure di scartarlo.

L'ambiente UVM agisce sul DUT descritto in VHDL o Verilog e l'evoluzione dei segnali in fase di simulazione associati alle varie *entry* della memoria dati è descritta dalle *waveform* in **Figura 3.20**. Si è osservato che tutta la RAM sotto test non contiene disallineamenti, in quanto tutti i bit eseguono il *toggling* coerentemente con le scritture pilotate dal programma di test. Pertanto, tutti gli accessi effettuati dal core avvengono correttamente e la funzionalità della RAM a livello RTL è coperta grazie al tipo di combinazioni di pattern applicati, che permettono di settare e re-settare ogni bit anche in funzione dei valori assunti nelle celle adiacenti della medesima riga e della medesima colonna.

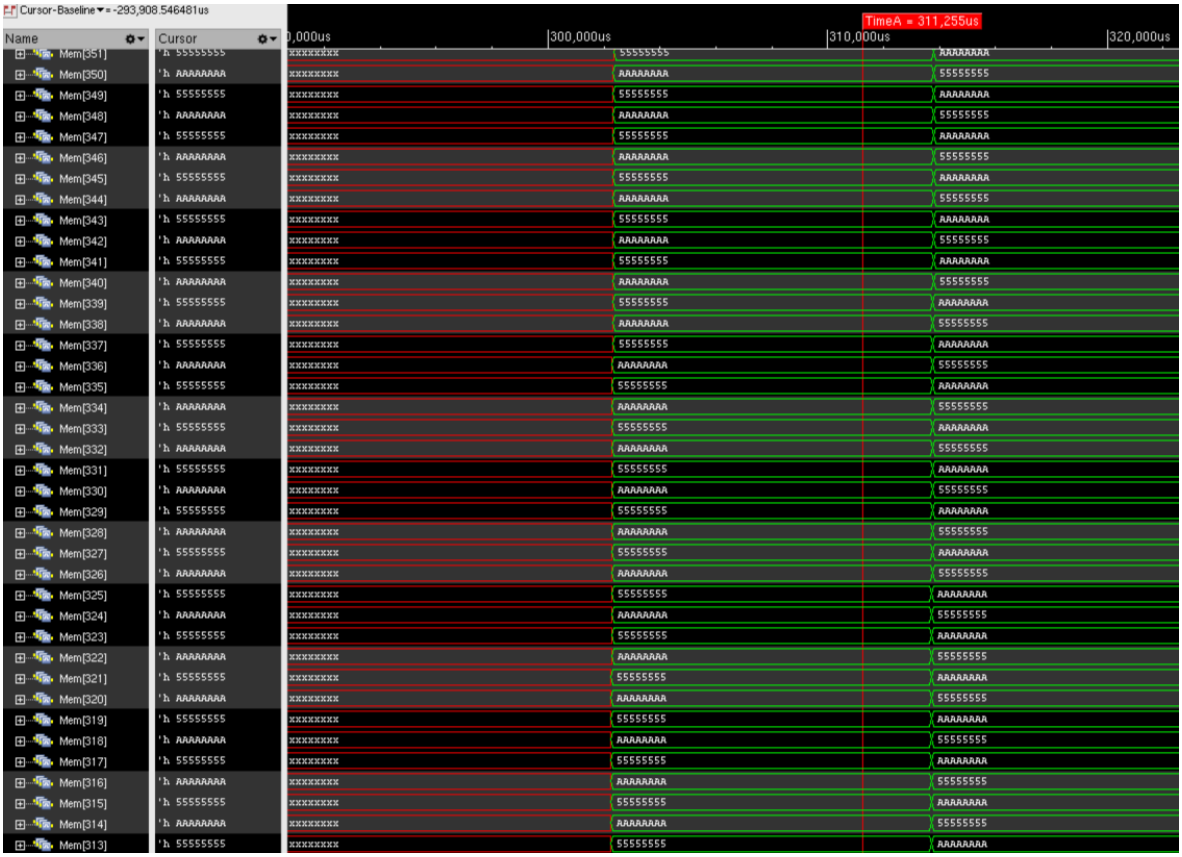
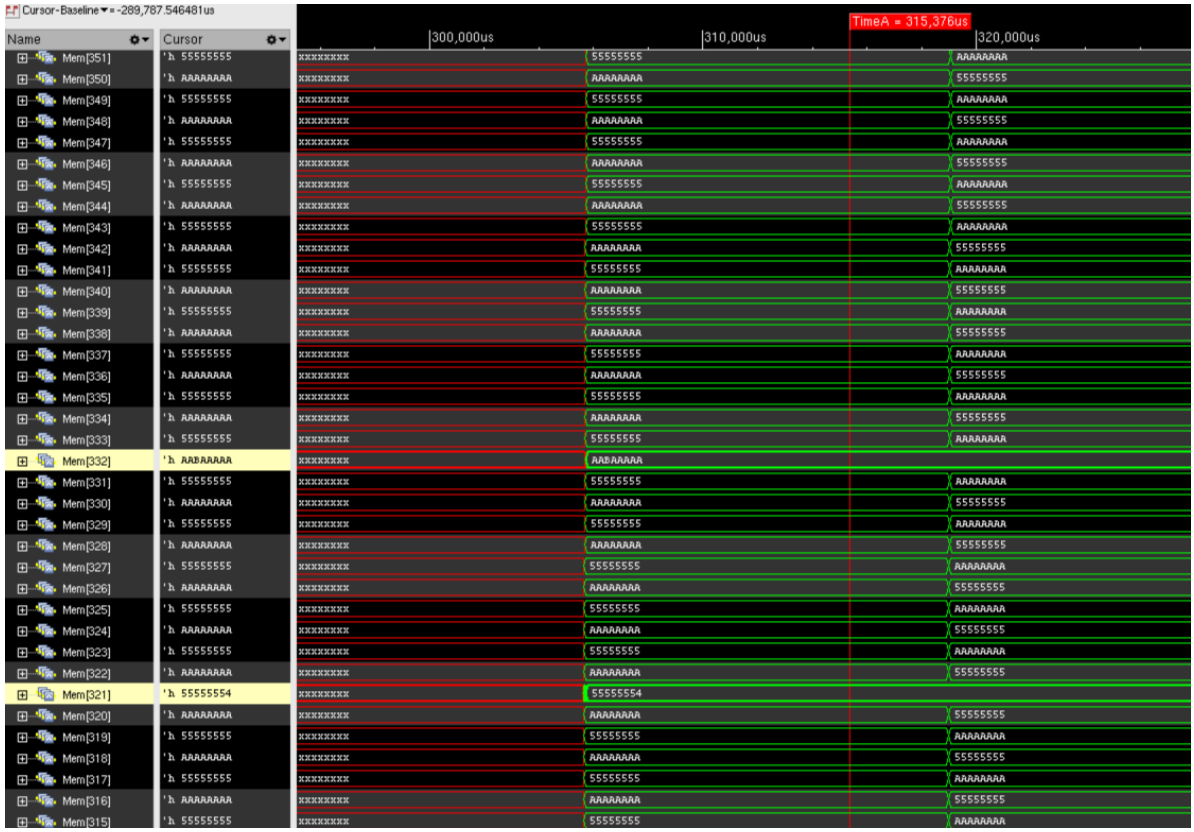


Figura 3.20: *Porzione di RAM le cui locazioni mostrano il pattern di scritture eseguito dal test.*

Il passaggio successivo è stato quello di invertire la condizione di test in modo da far fallire apposta l'algoritmo. Ad ogni riga sono stati individuati i disallineamenti su tutti i bit e sono stati salvati correttamente nelle variabili di conteggio riportate nei registri di uscita.

Dopodiché, è stato eseguito *run time* il *forcing* di alcuni bit scelti in modo arbitrario. In un primo caso, con l'intento di avere un numero di disallineamenti rilevati superiore alla soglia. Così facendo si è dimostrata la corretta funzionalità dell'ambiente UVM, per cui sono stati riportati i messaggi di errore attesi, come riportato in **Figura 3.21**.




```
1 UVM_ERROR @ 331842851.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Bit 0 of INT_STATUS has been set to 1. Unexpected bit values have
       been found during RAM data reading at first iteration
2 UVM_ERROR @ 331842851.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Bit 1 of INT_STATUS has been set to 1. Unexpected bit values have
       been found during RAM data reading at second iteration
3 UVM_ERROR @ 332645049.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Total number of mismatches at first iteration:      2
4 UVM_ERROR @ 332645049.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Total number of mismatches at second iteration:        62
5 UVM_ERROR @ 332645049.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Overall cumulative number of mismatches:       64
6 UVM_ERROR @ 332645049.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] The overall cumulative number of mismatches    64 overcomes the
      threshold of       50.
7 Test status: FAILED
```

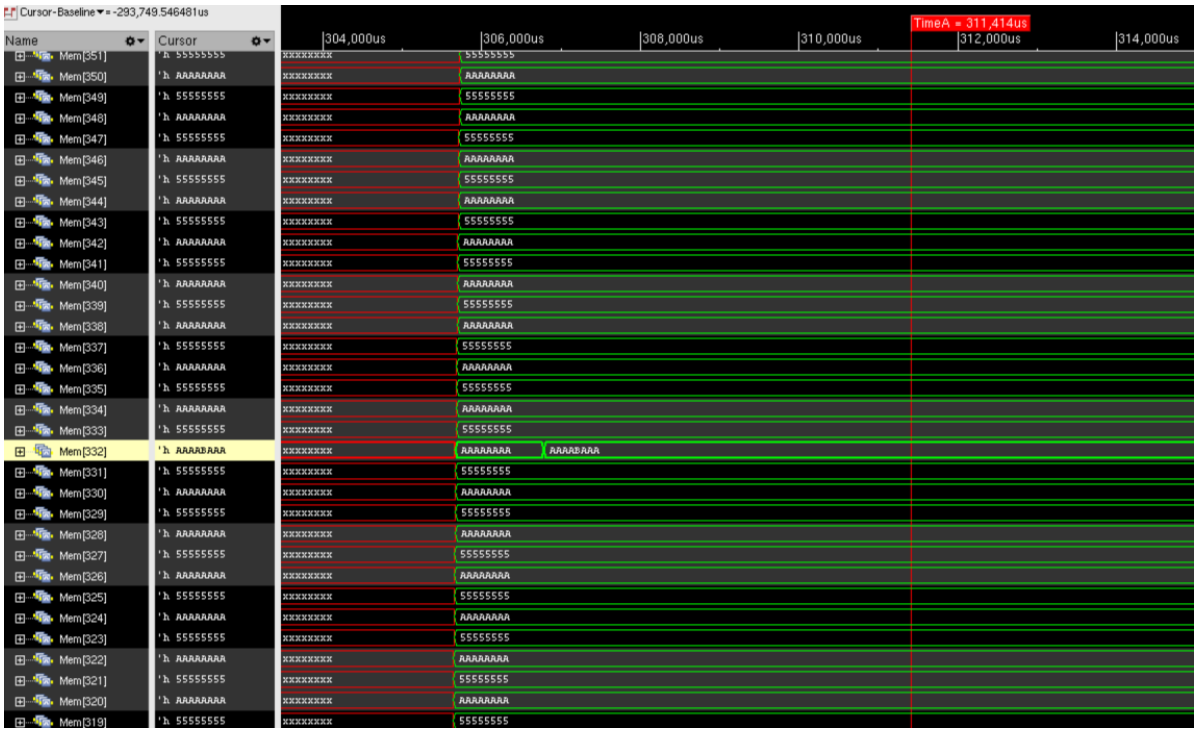
Figura 3.21: *Esempio di test che fallisce a seguito del rilevamento di un numero di disallineamenti superiore al valore di soglia, impostato arbitrariamente al valore di 50 celle massime accettabili.*

In particolare, si è agito su due bit su due righe differenti. L'effetto è stato quello di modificare il valore di un byte all'interno di tali righe, per cui dove prima veniva scritto `0xAA` ora si trova `0xAB`, invece alla riga ove tutti i byte valgono `0x55` ora si ha `0x54`. A causa del *force*, al secondo ciclo di scrittura le righe su cui si è agito rimangono immutate, di conseguenza si conteggia un numero totale di bit disallineati pari a 64:

- al controllo effettuato al primo ciclo di test vengono rilevati due disallineamenti, corrispondenti ai bit che sono stati forzati;

- al controllo effettuato al secondo ciclo di test vengono rilevati 31 + 31 *mismatch*, in quanto le righe forzate non effettuano l'inversione bit a bit della seconda scrittura.

Infine, per completezza si riporta in **Figura 3.22** anche il caso in cui il conteggio dei disallineamenti non supera il valore di soglia.




```
UVM_ERROR @ 510316516.00 ns : uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.
    sve_top.scoreboard_inst] Bit 0 of INT_STATUS has been set to 1. Unexpexted bit
    values have been found during RAM data reading at first iteration
UVM_ERROR @ 510316516.00 ns : uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.
    sve_top.scoreboard_inst] Bit 1 of INT_STATUS has been set to 1. Unexpexted bit
    values have been found during RAM data reading at second iteration
UVM_ERROR @ 511118714.00 ns : uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.
    sve_top.scoreboard_inst] Total number of mismatches at first iteration:        1
UVM_ERROR @ 511118714.00 ns : uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.
    sve_top.scoreboard_inst] Total number of mismatches at second iteration:   31
UVM_ERROR @ 511118714.00 ns : uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.
    sve_top.scoreboard_inst] Overall cumulative number of mismatches:      32
UVM_INFO @ 511118714.00 ns : uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
    .scoreboard_inst] The overall cumulative number of mismatches        32 does not
    overcome the threshold of       50.
Test status: PASSED
```

Figura 3.22: *Esempio di test in cui i mismatch rilevati non superano il valore di guardia e corrispondenti messaggi UVM.*

Si è scelto arbitrariamente di variare un singolo bit al primo ciclo di scrittura-lettura-controllo, in modo che, all'interno della *word* scelta, si avesse un byte con valore `0xAB` invece di `0xAA`. Pertanto, alla prima iterazione il *self-checking* effettuato dal core rileva 1 solo bit che fallisce. Alla seconda iterazione, tutti i bit vengono invertiti, ma a causa del *force* sulla riga, nessun bit viene sovrascritto, pertanto nel secondo ciclo si rilevano 31 bit disallineati (il bit corretto corrisponde a quello che viene forzato già alla prima iterazione). Il numero cumulativo di fallimenti rilevati quindi vale 32, inferiore alla soglia impostata ragionevolmente a 50 *mismatch* massimi.

3.6 Test RAM #4: verifica di *mismatching bits* nelle singole celle *word by word*

3.6.1 Obiettivi del test

L'ultimo test della memoria dati consiste nel verificare non solo il corretto *toggling* dei bit, ma anche il completo reset degli stessi.
Partendo dalla RAM non inizializzata, viene compiuto un ciclo di reset iniziale di tutte le celle. Una volta che tutte le righe della RAM si trovano al valore noto zero, si procede con un ciclo composto dal pattern "scrittura-lettura-controllo-scrittura-lettura-controllo-reset". rispetto al test precedente, qui si agisce su ogni riga in modo separato, eseguendo il ciclo iterativamente *word by word*.
In particolare, su ciascuna riga la prima word da testare è data da `0x55555555`, mentre la seconda scrittura inverte tutti i bit della riga, per cui si ha `0xAAAAAAAA`.
Quando tutte le righe sono state testate, si riporta in uscita il numero di bit che falliscono il test, ossia la somma cumulativa associata a tutte le righe, composta sia dalle prime letture sia dalle seconde.

3.6.2 Sviluppo del test C

La prima operazione eseguita dal core consiste nel resettare tutta la ram. L'operazione avviene de-referenziando ciascuna *word* per inizializzarla a zero.
Il passo successivo prevede l'esecuzione di due cicli annidati.

- Il ciclo più esterno permette l'avanzamento del test da una *word* alla successiva.
- Il ciclo annidato è composto da due iterazioni, agenti entrambe sulla riga sotto test:
  - nel primo caso, il valore di test è pari a `0x55555555`;
  - nel secondo, il valore da testare vale `0xAAAAAAAA`, in modo da coprire tutti i possibili valori assumibili dalle celle della riga.

  Dopo ogni scrittura, in caso di *mismatch* tra valore letto e valore atteso viene invocata una funzione di controllo riga, che si occupa di conteggiare i bit disallineati in modo analogo a quanto avveniva nel test precedente.
  Prima di incrementare il contatore di riga per procedere con la successiva, si applica il reset della *word*, in modo tale che, al termine di tutto il test, tutta la RAM contenga solo celle a zero.

Infine, viene eseguito il salvataggio sui registri di uscita della variabile di conteggio degli errori, effettuando il *boundary checking* tramite la comparazione del risultato su 32 bit con i massimi valori rappresentabili su ciascun registro di uscita da 16 bit, seguendo il medesimo approccio applicato al test precedente.

Nelle pagine successive, in **Figura 3.23** e **3.24**, sono raffigurati i diagrammi di flusso che dettagliano le varie operazioni eseguite dal test che sono appena state descritte.

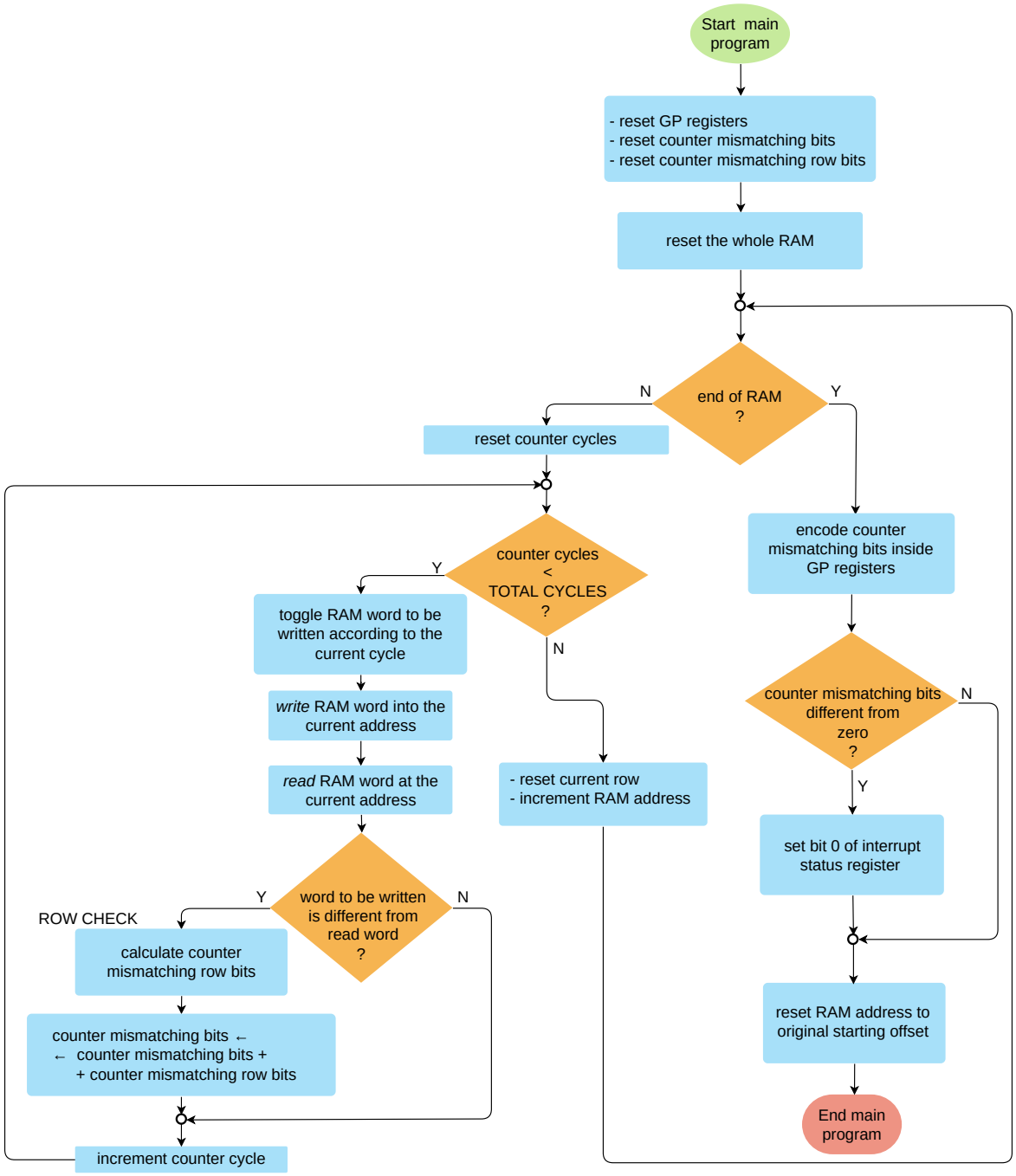


Figura 3.23: *Comportamento del programma di test della RAM che agisce riga per riga.*

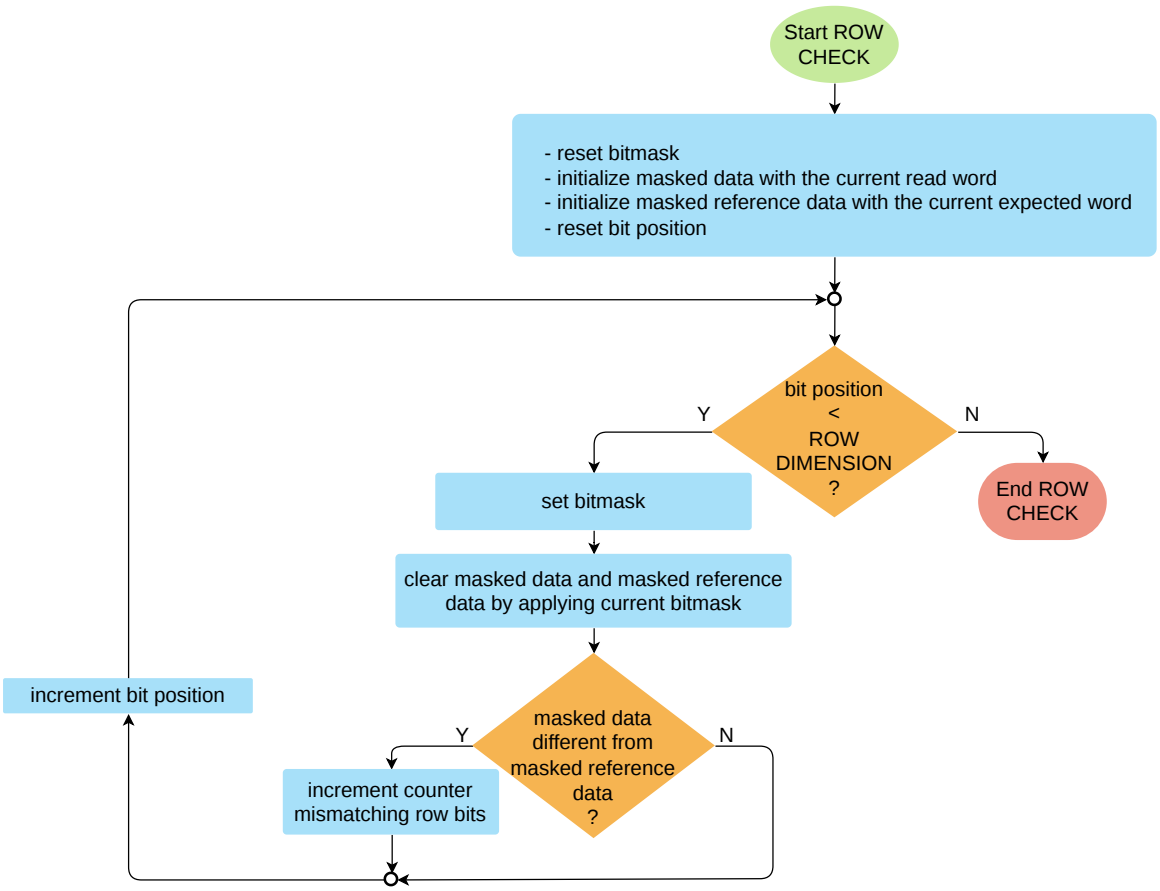


Figura 3.24: *Algoritmo per il controllo dei bit di una singola riga.*

3.6.3 Sviluppo del test UVM SystemVerilog

La classe di randomizzazione e la sequenza sono identiche a quelle del test precedente, per cui possono essere riutilizzate anche in quest'ultimo test, includendole all'interno della nuova classe di test a più alto livello.
Per quanto concerne lo scoreboard, questo eredita il contenuto dello scoreboard definito per il test precedente, dal momento che la funzione di decodifica è la stessa. Invece le funzioni che permettono di processare i pacchetti provenienti dalle *analysis port* sono ridefinite, dal momento che in questo caso si ha una sola variabile di conteggio degli errori e i messaggi da riportare nel file di *log* cambiano di conseguenza.

3.6.4 Simulazioni e risultati ottenuti

Il *tool* di simulazione ha permesso di visualizzare le forme d'onda associate a tutte le righe della ram, constatando l'avvenuta esecuzione dei *pattern* di test. Una idea globale sul comportamento complessivo si può avere da **Figura 3.25** e **Figura 3.26**.

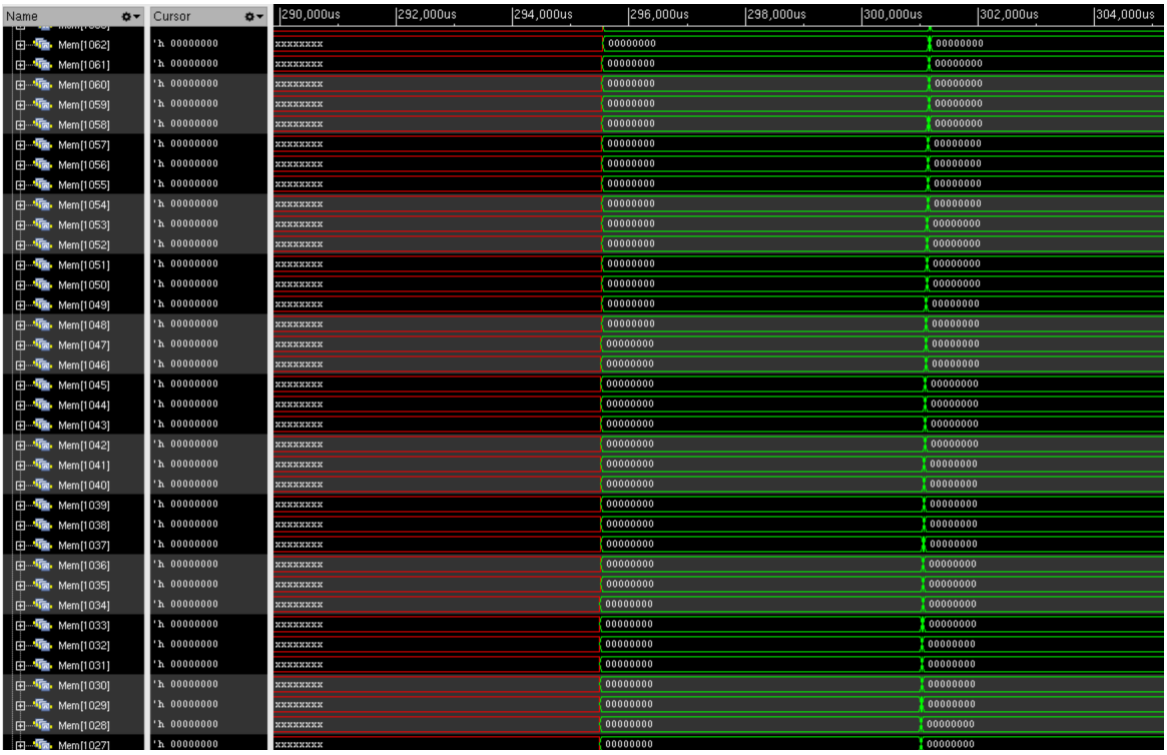


Figura 3.25: *Porzione di RAM in cui per cui viene riportata l'evoluzione della simulazione. Tra il reset iniziale e finale sono applicate le scritture a bit invertiti (qui non visibili per via della scala temporale più piccola), dettagliate meglio nella figura successiva.*

Figura 3.26: *Zoom di dettaglio su alcune righe della RAM, in cui si può appezzare la sequenza di reset-scrittura-lettura-controllo-scrittura-lettura-controllo-reset.*

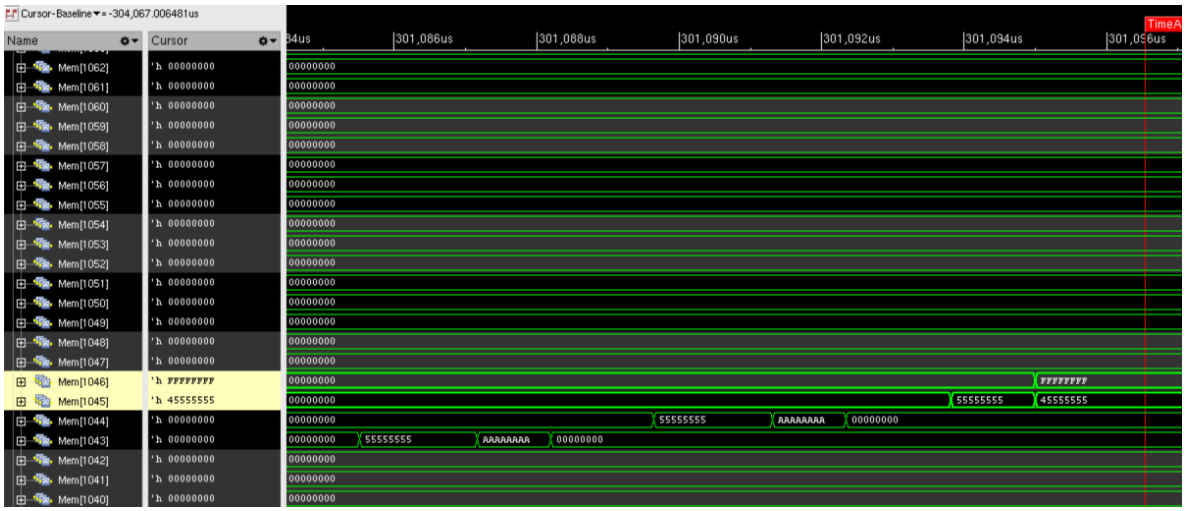
Infine, per provare la funzionalità del test, si forzano dei segnali arbitrariamente e si verifica che il conteggio dei *mismatch* coincida col valore atteso sotto quelle condizioni. Ad esempio, in **Figura 3.27** si effettuano i seguenti *force* da interfaccia grafica:

- alla riga 1045, si inverte un solo bit nell'ultimo byte di riga, risultando pari a `0x45`;
- alla riga successiva, si forzano a '1' tutti i bit della *word*.

Il numero di disallineamenti che ci si attende è pari a 64, ovvero alla somma di:

- per la riga 1045, (1 + 31) bit, in quanto al secondo ciclo di scrittura della riga i bit non si invertono, a causa del *force* sulla *word*;

- per la riga 1046, quando vengono eseguiti i due cicli di scrittura vengono rilevati tutti e 32 i bit errati, 16 bit dal primo ciclo e 16 dal secondo, alle posizioni in cui dovrebbero essere presenti degli zeri invece di tutti uni.




```
1 UVM_ERROR @ 305884482.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Bit 0 of INT_STATUS has been set to 1. Unexpected bit values have
       been found during RAM data reading
2 UVM_ERROR @ 306599020.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] Total number of mismatches:        64
3 UVM_ERROR @ 306599020.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
      .scoreboard_inst] The total number of mismatches     64 overcomes the threshold of
          10.
4 Test status: FAILED
```

Figura 3.27: *Verifica della funzionalità del test tramite operazione di forcing di alcuni bit, per verificare il corretto rilevamento di eventuali errori nel design. In questo esempio, la soglia di errore è impostata a 10 bit.*

Sulla base di tali risultati, è possibile affermare che il test è completo e sia la parte UVM sia la parte firmware funzionano correttamente sia nel caso di design senza errori, sia nel caso in cui vengano rilevati dei *mismatching bits*.

3.7 Sviluppi futuri

Oltre ai test che sono stati ideati vi possono essere implementazioni alternative, sulla base delle esigenze e dei risultati che si vogliono ottenere.

Ad esempio, si prendano in considerazione i primi due test della RAM dati. L'idea originaria, attuata in questo lavoro di tesi, era quella di avere un primo test che eseguisse più algoritmi in sequenza (in base alla dimensione della RAM e al numero di registri di uscita disponibili) e informasse se i sotto-blocchi verificati da ciascun algoritmo avessero passato il test o meno, e di avere un secondo test che permettesse di eseguire ciascun algoritmo in simulazioni separate senza sovrascrivere i registri di uscita per salvare l'informazione su quali righe fossero disallineate. Tuttavia, è anche possibile utilizzare un approccio differente.

Coinvolgendo l'utilizzo *run-time* dell'interfaccia, si è in grado di collassare i due test in uno solo e gli algoritmi dispiegati in uno solo. Un singolo algoritmo verificherebbe ciascun sotto-blocco in un unico test, per cui dopo aver testato il primo *chunk*, i registri di uscita verrebbero letti e decodificati tramite ambiente UVM, dopodiché il controllo ritorna nuovamente al core, il quale ripeterebbe le medesime operazioni di test sul *chunk* successivo, e così via sino al termine della memoria dati. In questo modo, la risorsa limitata costituita dai registri di uscita viene liberata *run-time* senza perderne il contenuto, il quale viene acquisito al termine di ogni algoritmo.

Capitolo 4
Test dei Banchi di Registri

Una volta terminato lo sviluppo dei test applicati al blocco di memoria dati RAM, e verificate le funzionalità e il corretto interfacciamento con il core del microcontrollore, si è proceduto con lo studio e la verifica dei banchi di registri.
Le specifiche messe a disposizione dal team di Design sono racchiuse e ordinate all'interno di pagine specifiche di un file Excel con estensione `.xml`. Ad ogni pagina è associato un banco di registri differente. Il flusso applicato per sviluppare i test a partire dalle specifiche assegnate si può schematizzare come mostrato in **Figura 4.1**.

Date le specifiche in formato strutturato in tabelle Excel, l'obiettivo è riportare le informazioni utili ai fini di verifica affinché possano essere lette e processate correttamente dalle routine degli algoritmi in esecuzione nel core del microcontrollore.

Lo strumento di cui si è fatto uso per effettuare il *parsing* delle pagine Excel è il Python. Grazie alla flessibilità e intuitività di tale linguaggio di programmazione, è possibile trasformare agevolmente l'informazione contenuta nei campi di interesse e convertirla in un formato comprensibile agli algoritmi scritti in linguaggio C.

Tra le informazioni riportate all'interno dei file, sono indicati i requisiti di policy di accesso che il core può attuare su ciascun registro. L'obiettivo è verificare che il core si interfacci correttamente con tutti i registri dei banchi di memoria, in modo da validare, a livello di design RTL, il rispetto delle policy attese. Per approfondire l'implementazione della parte relativa ai test C e UVM si rimanda al **Paragrafo 4.2**.

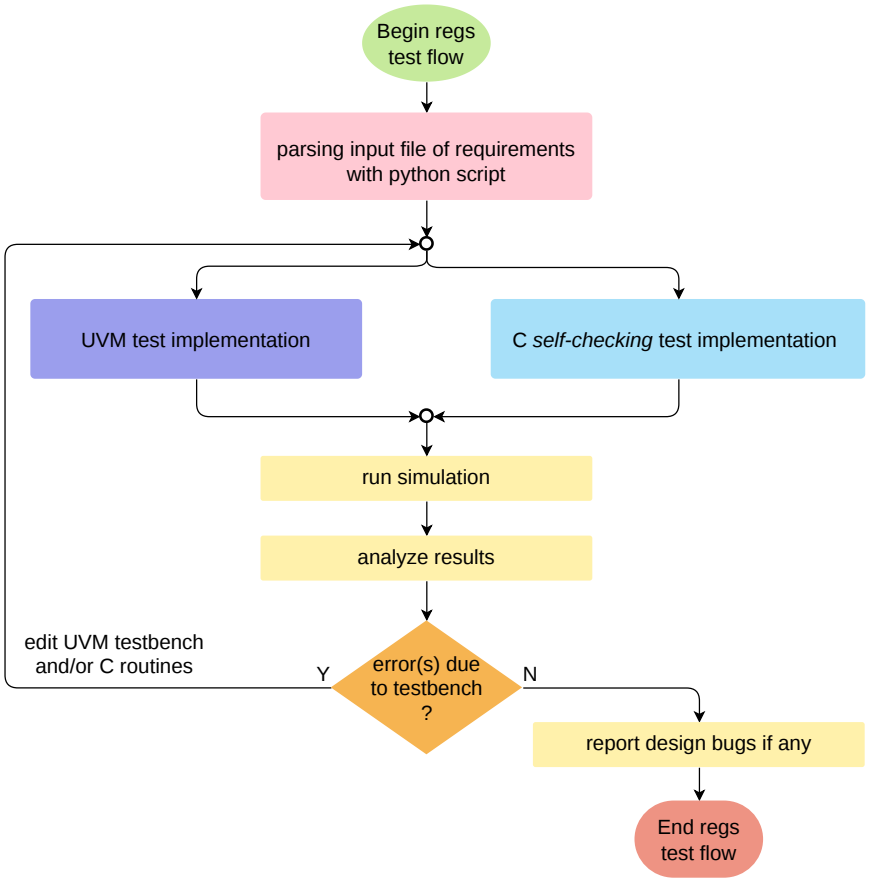


Figura 4.1: *Flusso di sviluppo dei test associati ai banchi di registri.*

4.1 Parsing del file di specifiche

Come primo passo è necessario riportare le specifiche dichiarate nel file a disposizione in un formato compatibile con la sintassi del codice C.
Il file Excel di partenza consiste in una raccolta di *pages*, ciascuna delle quali riporta le specifiche di porzioni differenti della mappa di memoria.

- La pagina principale, detta anche pagina di *default*, è quella associata all'intero dispositivo. In essa sono mappati i registri con un offset tale da poter permettere, quando specificato, le letture e le scritture all'interno degli stessi tramite interfaccia I2C/SPI. Programmando in ambiente UVM in fase di configurazione alcuni registri specifici della *default page*, si è in grado di reindirizzare le transazioni su interfaccia verso registri di pagine differenti della mappa di memoria del dispositivo. Il meccanismo con cui si espleta tale operazione è stato ampiamente spiegato in precedenza al **Paragrafo 3.3.4**.

- La pagina dei registri interni al microcontrollore. In essa sono mappati i registri con l'informazione sull'offset per accedere da interfaccia e con offset per l'accesso

attraverso il core. Oltre ai nomi che identificano ciascun indirizzo dei registri, è anche riportata la policy di accesso. Essenzialmente, si descrive se un registro sia leggibile/scrivibile da interfaccia e se leggibile/scrivibile da core. In generale, un registro che è scrivibile da interfaccia non è scrivibile dal core, e viceversa. Nel caso fosse possibile scrivere da "entrambi i lati", il registro viene indicato come *special* riempiendo un apposito campo dedicato. Esistono altresì alcuni registri che si comportano come registri *dual port*. Si tratta di porzioni di memoria in grado di attuare un protocollo di *handshaking* tra l'interfaccia e il core. Ad esempio, coerentemente con la specifica assegnata, si possono settare i bit di uno di questi registri mediante una operazione di *set* da interfaccia, per poi effettuare il *clear* dal core quando desiderato. Altri registri invece sono in grado di implementare tale meccanismo in modo speculare.

- La pagina dedicata ai registri di sistema esterni al microcontrollore. Le informazioni in essa contenute sono analoghe alla pagina precedente ma sono associate a indirizzi di memoria differenti e sono formattate in maniera leggermente differente, cosa che ha reso necessaria l'implementazione di un nuovo script Python che si adattasse al nuovo file di specifiche e che fornisse ai test UVM e C le informazioni su indirizzi e policy di accesso con il medesimo formato utilizzato per i test dei registri interni. In questo modo è stato possibile sfruttare la riusabilità dei test anche per il banco di registri di sistema.

Il flusso perseguito è riportato nel diagramma in **Figura 4.2**, mentre gli script creati sono riportati per esteso in **Appendice C.1** e **C.2**.

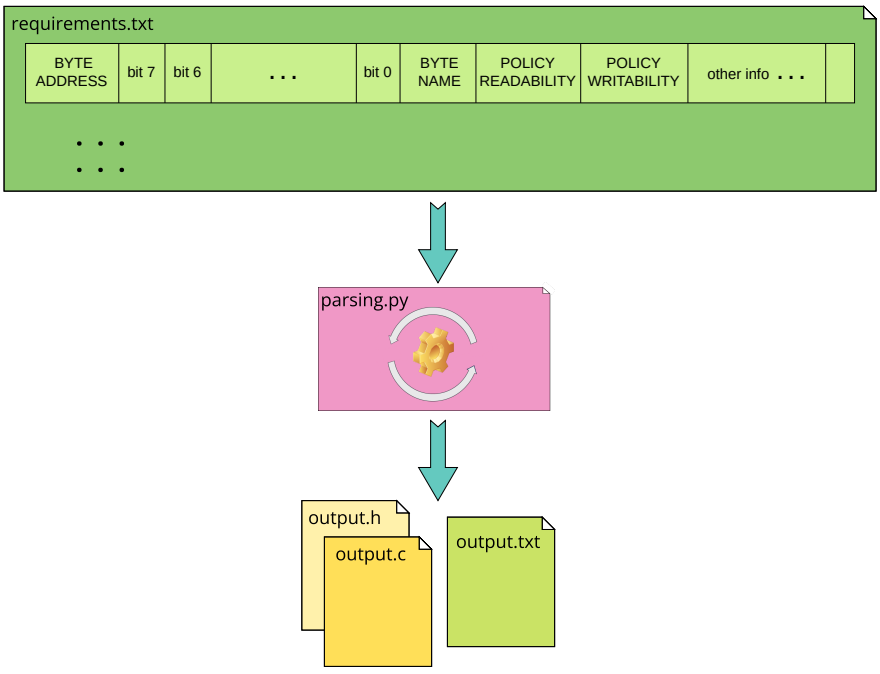


Figura 4.2: *Rappresentazione concettuale del processo di parsing eseguito sul file di specifiche.*

4.1.1 Generazione del file di ingresso e processamento per righe

La sequenza di operazioni da eseguire associate alla generazione del file di stimolo per lo script Python e alla sua elaborazione vengono dettagliate di seguito.

- Il file Excel viene convertito in un file testuale con formato *tab delimiter*, in cui ogni campo di ciascuna riga è separato dal carattere di *tab* `'\t'`. Tale operazione è effettuata attraverso l'interfaccia grafica del programma Excel, per cui attraverso un comando di salvataggio è possibile avviare il processo di conversione in modo automatizzato. Siccome il file `.xml` è costituito da diverse pagine, la conversione viene avviata solamente sulla pagina di interesse, ossia quella dei registri del microcontrollore oppure dei registri di sistema.

- Il file testuale ottenuto costituisce il vero ingresso da fornire allo script Python sviluppato appositamente per il *parsing* dei dati.

- Per ogni riga del file, si controlla se è definito il campo associato all'indirizzo con offset per l'accesso del core. In caso affermativo, si procede a convertire i caratteri letti in un valore numerico in formato esadecimale. La variabile in questione costituisce l'*address* riferito a uno specifico byte di memoria del banco di registri.

- A questo punto si procede con l'analisi dei campi associati alle policy di accesso di lettura e scrittura riferite al "lato" del core. Il risultato della lettura dei caratteri associati alle policy è una stringa che riporta la codifica delle due policy, per cui a ogni policy viene associato un carattere, generando quindi due caratteri.

  - Se il registro è leggibile, si codifica con `'F'`, altrimenti con `'A'`.
  - Se il registro è scrivibile, si codifica con `'F'`, altrimenti con `'A'`.
  - Se il registro è di tipo *clear only*, lo si identifica con `'B'`.
  - Se il registro è di tipo *set only*, lo si identifica con `'C'`.

- Una volta che il *parsing* delle policy del byte alla riga corrente è stato effettuato, occorre codificare anche l'informazione sull'accessibilità dei singoli bit appartenenti a quel byte. Se sulla riga corrente il campo associato a un bit risulta "vuoto", ossia non presenta una identificazione alfanumerica, allora tale bit non è accessibile. Il meccanismo che è stato implementato per codificare tale informazione si basa sulle operazioni *bitwise* con delle maschere.
  Essenzialmente, si applica in modo iterativo una maschera su una variabile numerica, per cui ad ogni iterazione si effettua il *parsing* di un bit, se un bit è accessibile allora si effettua il *set* del bit corrispondente all'interno della variabile di salvataggio. In caso contrario invece il bit rimane al valore di inizializzazione, ossia zero.
  Al termine del ciclo, si converte la variabile processata dapprima in un dato esadecimale, e subito dopo in stringa di caratteri. Quest'ultima rappresenta l'informazione codificata sull'accessibilità.

- Lo *step* finale della routine di lettura delle righe consiste nel salvataggio in una lista delle seguenti informazioni:
  ogni elemento della lista corrisponde ad una riga del file di partenza, ed è costituito da una stringa ottenuta dalla concatenazione, nell'ordine, di
  "accessibilità" + "policy" + "address"

- Siccome l'accesso ai registri da parte degli algoritmi del core può essere effettuato facendo riferimento all'indirizzo associato al primo byte del registro, per evitare l'aggiunta di ulteriori controlli sulla consecutività degli indirizzi nel programma di test, si è fatto in modo di fornire direttamente all'algoritmo un insieme di indirizzi consecutivi l'uno con l'altro. Quando i controlli del programma individuano un byte inesistente (non accessibile), su di esso non viene eseguito alcun test.
  La motivazione per cui si desidera generare le codifiche per indirizzi consecutivi è legata a come gli algoritmi di test C per le policy agiscono: a ogni iterazione, il test tenta di modificare blocchi di 4 byte per volta; dopodiché, le policy di ciascun byte vengono testate separatamente tramite una operazione di *shifting* di una *bitmask* da applicare ad una apposita variabile che contiene l'informazione delle policy di tutta la *word*. Per approfondire il comportamento del test si rimanda al **Paragrafo 4.3.2**.

4.1.2 Generazione dei file di uscita

La parte finale dello script è adibita alla generazione dei file di uscita.
Un file viene scritto in formato testuale e torna utile al codice SystemVerilog nel caso occorresse eseguire delle scritture da interfaccia, le quali possono essere utilizzate al posto delle operazioni di *deposit*, finalizzate a ottenere la completezza del test di alcune policy. Altri due file invece vengono successivamente inclusi all'interno del programma C di test. Si tratta di un *header file* contenente le dichiarazioni delle macro e delle strutture dati generate dal Python, e di un file con estensione `.c` che definisce il contenuto di quelle strutture dati.

Per le codifiche degli indirizzi e delle relative policy si è optato per la seguente soluzione: ciascuna stringa di codifica contenuta nella lista generata dal Python viene scritta come variabile di tipo stringa all'interno della struttura dati del C. In particolare, ci si è serviti di un array di stringhe, per cui ogni elemento dell'array corrisponde alla codifica di un indirizzo con la sua policy. L'array viene riempito in modo sequenziale, per cui due stringhe consecutive corrispondono a due indirizzi consecutivi. Tuttavia, per poter implementare in modo robusto e flessibile questo modo di trasferire le informazioni dal file testuale di partenza al codice C, si deve tenere conto delle seguenti criticità.

- La *register map* del banco di memoria preso in considerazione potrebbe essere troppo grande. In questo caso il rischio di "sforare" la dimensione massima consentita per il numero di elementi in un array diventa concreto. Nel caso in cui questo avvenisse, in fase di compilazione del programma il compilatore genererebbe un errore di *segmentation fault*.

Pertanto, via Python si decide in quanti array dividere la scrittura delle codifiche degli indirizzi. In questo modo si aggiunge flessibilità e si limita la dimensione di ciascun array ad un valore noto, risolvendo il problema tramite un *boundary checking* sulla dimensione assegnata agli array.

- L'uso di un numero variabile di array da applicazione ad applicazione comporta tuttavia un altro problema, ossia quello di riferirsi a ciascuno di essi dinamicamente durante il test.
  Pertanto, si fa uso di un altro array, nel quale il Python salva, in ciascuna cella, la *reference* ad un determinato array. In questo modo, le istruzioni dell'algoritmo in C sono in grado di processare i dati de-referenziando il contenuto di ciascun elemento di questo array.

Al variare dei parametri di dimensionamento degli array in gioco impostati dal Python, ora è possibile accedere ad ogni stringa che è stata scritta utilizzando una variabile di tipo puntatore a stringa, la quale agisce in uno spazio quadridimensionale. Tenuto conto che la dimensionalità di una stringa è pari a due, l'array di stringhe risulta pertanto a tre dimensioni, di conseguenza per accedere ad un singolo elemento dell'array stesso è necessario de-referenziare tre volte un puntatore, il quale sarà tridimensionale. Il concetto è rappresentato in **Figura 4.3**.

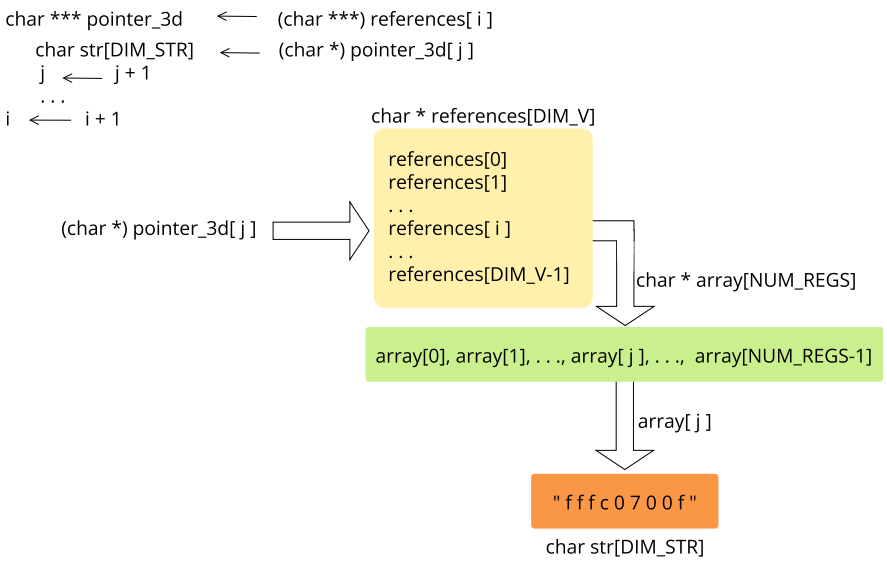


Figura 4.3: *Pseudocodice per l'accesso ai dati generati e relativo schema a blocchi.*

Nota

L'utilizzo di questa modalità di trasposizione delle informazioni si è reso necessario in quanto, nel sistema oggetto di studio e più in generale nei Sistemi Embedded, non è possibile includere nel codice in esecuzione dal core le librerie per la gestione delle funzionalità di I/O. Ad esempio, se si fosse utilizzata la funzione di `fopen()` per aprire il file testuale ed effettuare il *parsing* dell'informazione direttamente nell'algoritmo di test, il compilatore avrebbe restituito un errore.

In **Figura 4.4** si riporta il contenuto del file `.h` di uscita in cui vengono dichiarate le strutture dati generate nel caso in cui si suddivida un generico banco di registri da `NUM_BYTES` byte in 4 array di stringhe. In questo caso, il contenuto presente nelle *entry* dell'array `v_references` è indicato in **Figura 4.5** e viene sfruttato dall'algoritmo di test per accedere iterativamente alle stringhe dei vari array usando un puntatore invece del nome degli array. In questo modo, la flessibilità e la modularità sono massimizzate.

```
#ifndef _UC_REG_INFO_H_
#define _UC_REG_INFO_H_
#include <stdint.h>
#define NUM_BYTES 512
#define NUM_WORDS 128
#define DIM_STR 9 + 1 // take into account '\0'
#define NUM_BYTES_ARRAY 128
#define DIM_V 4
extern const char * v_policy_and_addr_1[NUM_BYTES_ARRAY];
extern const char * v_policy_and_addr_2[128];
extern const char * v_policy_and_addr_3[128];
extern const char * v_policy_and_addr_4[128];
extern const uint32_t v_dims[DIM_V];
extern const char * v_references[DIM_V];
#endif
```

Figura 4.4: *Esempio di file di uscita* `.h` *avendo scelto da Python una suddivisione degli indirizzi di memoria in quattro array di stringhe. In generale, la dimensione dell'ultimo array può essere inferiore alle altre in base al numero di byte rimanenti da mappare.*

```
#include "uc_reg_info.h"
const char * v_policy_and_addr_1[NUM_BYTES_ARRAY] = {
"03ff07000", "00aa07001", "9ffa07002", "00aa07003", "3fff07004", "00aa07005",
"00aa07006", "00aa07007", "00aa07008", "00aa07009", "00aa0700a", "00aa0700b",
"fffb0700c", "fffb0700d", "fffc0700e", "fffc0700f", "ffff07010", "ffff07011",
. . .
};
const char * v_policy_and_addr_2[NUM_BYTES_ARRAY] = {
. . .
}
. . .
const uint32_t v_dims[DIM_V] = {128, 128, 128, 128};
const char * v_references[DIM_V] = {(const char *) v_policy_and_addr_1,
(const char *) v_policy_and_addr_2, (const char *) v_policy_and_addr_3,
(const char *) v_policy_and_addr_4};
```

Figura 4.5: *Estratto del file di uscita* `.c`*, il quale contiene la definizione delle variabili dichiarate all'interno del file di uscita* `.h`*. L'array* `v_dims` *contiene le dimensioni di ciascun array a cui puntano le entry di* `v_references`*.*

Infine, in **Figura 4.6** si riporta una porzione del file di uscita generato per le manipolazioni SV. Nei test sviluppati non se n'è fatto uso, tuttavia se si fosse usata l'interfaccia per leggere i registri sotto test e invertire i valori letti, l'uso delle codifiche contenute in questo file sarebbe servito. Come descritto nel paragrafo dedicato al SV, si è invece optato per l'uso del comando di *deposit*, per cui, anziché utilizzare la *User Interface* nel mezzo del test, ci si è affidati all'accesso diretto via testbench tramite *Virtual Interface*.

```
1 FF   07000
2 AA   07001
3 FA   07002
4 AA   07003
5 FF   07004
6 .  .  .
```

Figura 4.6: *Estratto del file di uscita* `.txt`.

4.2 Metodologia di *self-checking* per la verifica delle modalità di accesso

Così come avviene per i test della RAM, anche per i test delle policy di accesso ai registri si procede tramite auto-test attraverso l'esecuzione del firmware nel core. Il principio consiste nell'accedere alle locazioni di memoria associate al banco di registri sotto test sfruttando gli indirizzi a cui tali porzioni dell'hardware sono state mappate. Una volta de-referenziato il puntatore utilizzato per l'accesso, si procede con l'applicazione di una serie di maschere per verificare che i bit del registro abbiano il valore che ci si attende. Il valore da testare viene scelto dal test sulla base della policy di accesso dichiarata per quel registro, a sua volta codificata nella stringa salvata nella struttura dati generata dal Python. Nel caso in cui vengano rilevati dei bit errati, si codifica l'indirizzo del registro che fallisce e si salva tale informazione sui registri di uscita, con lo stesso meccanismo applicato per i test della RAM.
In questo modo, il test viene totalmente eseguito dal core, l'interfaccia esterna si occupa di decodificare i risultati generati dall'elaborazione nel microcontrollore.

4.2.1 Organizzazione dei test

I test ideati sono costruiti in modo tale da rispettare la seguente suddivisione.

- A livello di codice C, a ogni algoritmo corrisponde il test di una policy di accesso. Per motivi di ottimizzazione del file eseguibile, ogni algoritmo è scritto in un file sorgente separato e viene compilato solo sulla base di un comando fornito dall'utente quando viene lanciato lo script Bash contenete i comandi di compilazione. In **Figura 4.7** vengono mostrati i messaggi relativi a questo processo.

```
Choose which algo shall be compiled. Insert:
'0' for algo_00
'1' for algo_01
'2' for algo_02
'3' for algo_03
'4' for algo_04
'5' for algo_05

4

'4' has been selected, compiling . . .
OK
```

Figura 4.7: *Messaggi dello script di compilazione. Ogni algoritmo corrisponde a un test di una modalità di accesso differente.*

In questo modo lo script, tramite un *case* in Bash, include soltanto i file sorgente necessari alla generazione dell'eseguibile per l'algoritmo i-esimo, escludendo i file che invocano le funzioni associate agli algoritmi per i quali si conosce a priori che l'esecuzione all'interno di quel test non avrebbe luogo. Il risultato è un codice con una dimensione molto più ridotta, limitando l'impatto sulla memoria di programma.

- A livello SystemVerilog esiste un'unica classe di test creata appositamente per gestire tutti i possibili test di questo banco di registri. Al suo interno sono istanziate:

  · La classe di randomizzazione, la quale principalmente permette di decidere il valore della variabile di abilitazione degli algoritmi.

  · La sequenza, il cui task principale permette di eseguire le operazioni desiderate sulla base di quale algoritmo è abilitato. Si tratta di un *case* che testa il valore della variabile di abilitazione degli algoritmi generata dalla randomizzazione, siccome a ogni algoritmo corrisponde un test differente, questo task permette di implementare le operazioni di test per ogni policy di accesso corrispondente.

  · Lo scoreboard, in cui si esegue il controllo sui registri di uscita applicandolo in maniera differente da test a test, sempre grazie a un *case statement* che discrimina le operazioni di decodifica da eseguire sulla base del valore di inizializzazione della variabile di abilitazione degli algoritmi; quest'ultima viene passata allo scoreboard quando viene ricevuto il pacchetto di accensione del dispositivo.

Le operazioni eseguite in ciascun test e i risultati ottenuti sono trattati nei paragrafi seguenti.

4.3 Test della policy READ/WRITE

4.3.1 Obiettivi del test

La prima modalità di accesso da testare è relativa ai byte che nominalmente risultano sia leggibili sia scrivibili dal core.
Il fulcro di questo test consiste in:

- leggere una *word* da 4 byte, indirizzando il primo byte della stessa per eseguire la lettura del registro;

- invertire il valore letto bit a bit;

- scrivere il valore letto al posto della *word* originaria;

- leggere il nuovo valore;

- applicare la maschera per la selezione del byte da testare tra i 4 che sono stati letti;

- applicare la stessa maschera anche al valore della prima lettura;

- applicare la maschera di accessibilità sui bit associati al byte selezionato per entrambe le letture;

- comparare i due risultati, in caso di disallineamento riportare l'errore tramite la codifica dell'indirizzo associato al byte e incrementare il contatore di errore.

Pertanto, l'obiettivo è quello di riportare in uscita l'indirizzo del byte che fallisce rispetto alla policy di accesso che gli è stata assegnata: se anche solo un bit non la rispetta, l'errore viene segnalato. Grazie all'informazione su quale indirizzo si comporta in modo anomalo, dalla *register map* è possibile risalire all'identificatore associato al byte e quindi alle funzionalità implementate dai vari bit, descritte nel manuale tecnico di riferimento. Tramite l'analisi delle forme d'onda di simulazione e i comportamenti descritti nel manuale, si è in grado di motivare le anomalie rilevate, le quali possono essere dovute principalmente a due cause:

* la mancanza di informazioni più dettagliate all'interno del file Excel dei requisiti, come l'assenza di ulteriori campi per indicare il comportamento di alcuni bit all'interno di un byte, per i quali è stata assegnata una policy di accesso che in realtà nel manuale risulta essere differente dal comportamento degli altri bit aventi come policy quella assegnata all'intero byte nella tabella Excel;

* la presenza di un baco nel design, per cui tutti i bit descritti nel manuale di riferimento sono descritti senza perdita di informazione in modo coerente anche all'interno della tabella Excel, ma il comportamento verificato non è conforme a quello atteso sia nella tabella sia nel manuale.

Per quanto concerne invece i bit inaccessibili, ossia quelli indicati nel manuale come *unused* e nella tabella Excel come campi vuoti, questi non causano problemi a livello di errori di test, dal momento che vengono filtrati dalla mascheratura *bitwise* effettuata prima della comparazione.

4.3.2 Sviluppo del test C

Il programma si articola in vari passi, descritti nel diagramma in **Figura 4.8**.
Il primo passo consiste nel leggere le stringhe presenti nella struttura generata dal Python; ogni stringa corrisponde a un byte da testare e contiene un insieme di caratteri che devono essere convertirli in:

- una variabile contenente il valore decimale il cui corrispondente in codice binario è una maschera che discrimina quali bit siano accessibili e quali no; in tutto sono 8 bit, ossia due caratteri da decodificare in esadecimale;

- una variabile contenente i due corrispondenti caratteri ASCII che codificano la policy di accesso da testare;

- una variabile contenente l'indirizzo in esadecimale del byte sotto test.

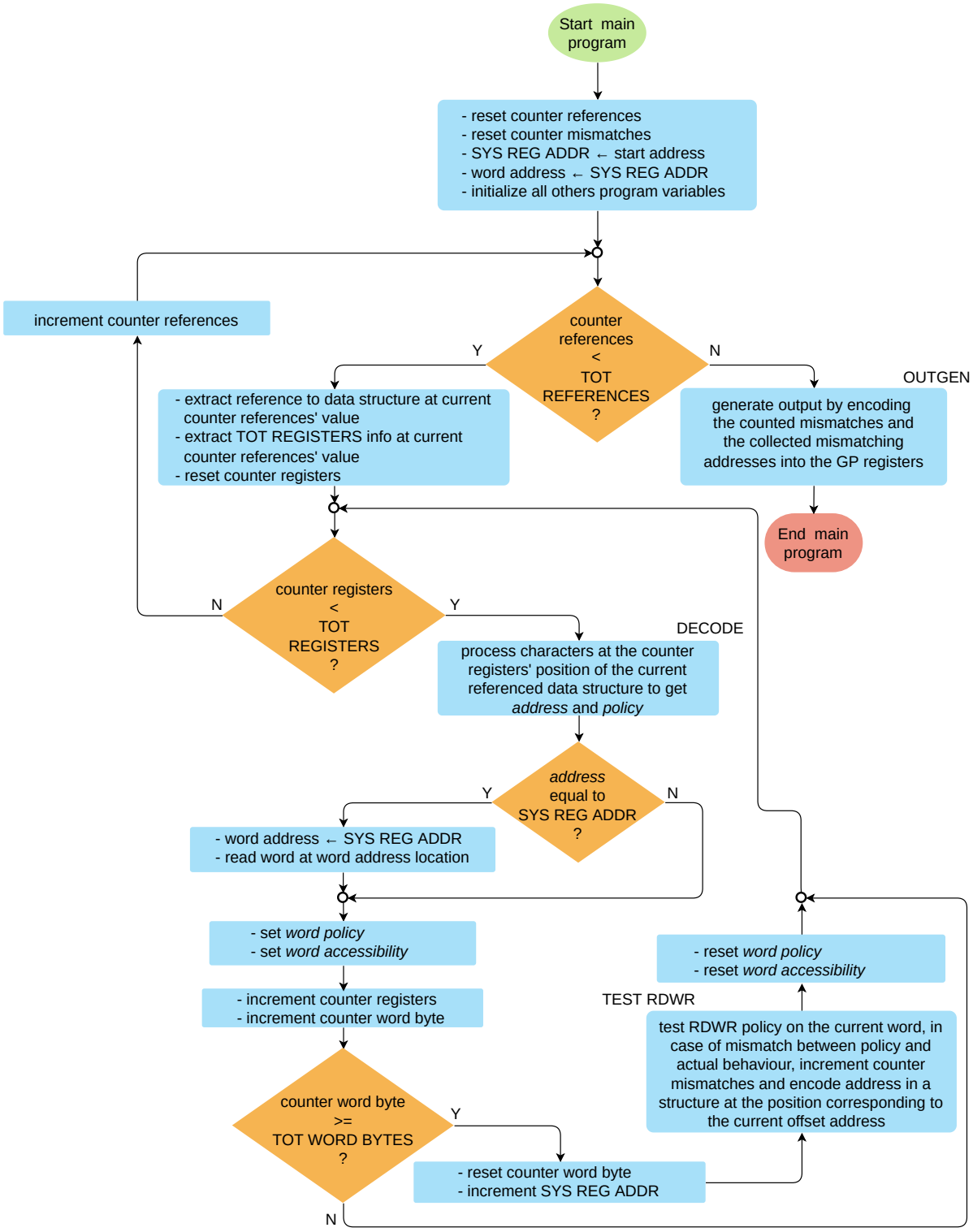


Figura 4.8: *Flusso seguito dal programma per elaborare quanto estrapolato dallo script Python.*

Il meccanismo corrispondente è descritto dalla seguente porzione di codice:

```
// initialize variables
volatile uint32_t *uc_reg_addr = UC_REGS_ADDR_START;
volatile uint32_t *word_addr = uc_reg_addr;
char ***ref_to_v_policy_and_addr = NULL;
. . .

// loop over v_references
for (uint32_t v_ref_cnt = 0; v_ref_cnt < DIM_V; v_ref_cnt++) {
    ref_to_v_policy_and_addr = (char ***) v_references[v_ref_cnt];
    num_regs = v_dims[v_ref_cnt];

    // loop over v_policy_and_addr_x, where x == v_ref_cnt
    while ((v_cnt < num_regs) {
        strcpy(str_policy_addr, (char *) ref_to_v_policy_and_addr[v_cnt]);

    // loop over string @ pos v_cnt inside v_policy_and_addr_x
        for (uint32_t i = 0; i < DIM_STR; i++) {
            car = str_policy_addr[i];
            if (isdigit(car) != 0) {
                val = ((uint8_t) car) - ((uint8_t) '0');
            } else {
                car = tolower(car);
                val = ((uint8_t) car) - ((uint8_t) 'a') + 10;
            }
            v_hex_str_to_int[i] = val;
        if (i < 2) {
                accessibility |= (val << (4*1 - 4*i));
            } else if ((i >= 2) && (i < 4)) {
                policy |= (val << (4*3 - 4*i));
            } else {
                reg_addr |= (val << (4*8 - 4*i));
            }
    } // end for

    // generate masks policy_word, accessibility_word
        if (reg_addr == (uint32_t) uc_reg_addr) {
            word_addr = uc_reg_addr;
            word_reg = *(word_addr); // in perform_rwr
            policy_word |= (policy << 8*cnt_byte);
                    accessibility_word |= (accessibility << 8*cnt_byte);
        } else {
            policy_word |= (policy << 8*cnt_byte);
            accessibility_word |= (accessibility << 8*cnt_byte);
        }
        policy = RST;
        accessibility = RST;
        reg_addr = RST;
        v_cnt++;
        cnt_byte++;

    /* PERFORM TEST on reg # v_cnt */
    . . .
    /* END TEST on reg # v_cnt */
    } // end while
} // end for

// generate output
output_gen(cnt_mismatches, pos_output_reg);
```

Il dettaglio delle operazioni di test appartenenti al blocco indicato come *TEST RDWR* è rappresentato in **Figura 4.9**.

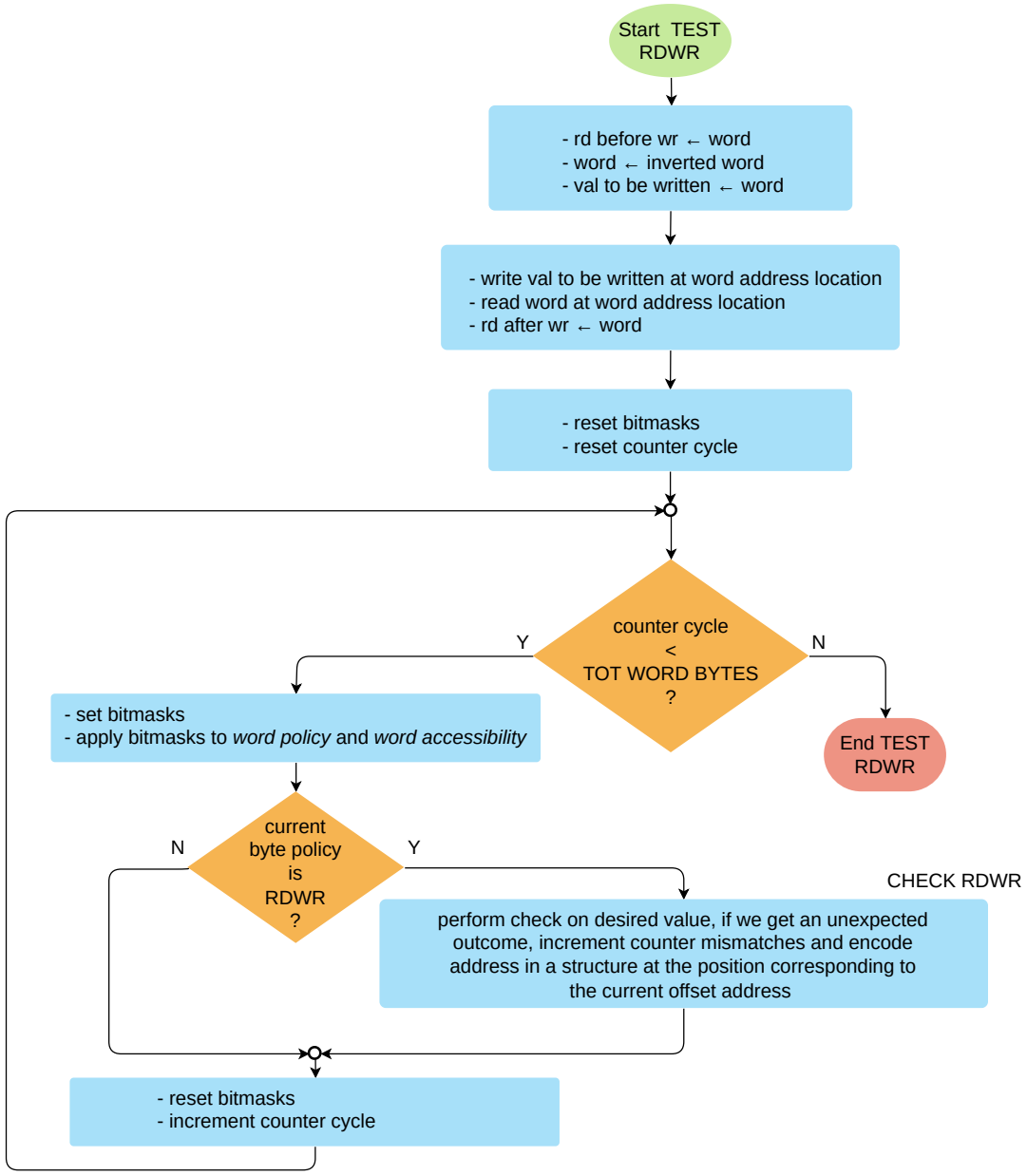


Figura 4.9: *Dettaglio del flusso di auto-test della policy READ/WRITE eseguito dal core.*

Quando sono stati acquisiti 4 byte, al byte successivo si procede con la lettura della prossima *word*, siccome l'indirizzo a esso associato corrisponde al *byte*0 del registro successivo. Dopodiché si genera il valore della maschera di accessibilità e della maschera delle policy per i byte della nuova *word*, decodificando le stringhe a essi associate.
Si consideri la generica iterazione j in cui sono stati acquisiti tutti e 4 i byte di un registro: come riportato nel listato in **Figura 4.10** il test effettua una *read before write*, l'inversione del dato letto, una *read after write*, applica le maschere ed effettua il controllo. In caso di *mismatch*, viene eseguita la codifica dell'indirizzo (funzione `encode_addr()`), con un meccanismo analogo a quello implementato per i test della RAM, con la differenza che i bit di codifica sono scritti all'interno di un array per il salvataggio dei dati di uscita, in modo da permettere il test dei registri stessi di uscita senza sovrascrivere i risultati. Quando tutti i registri sono stati testati, quindi anche i registri di uscita, si procede con la scrittura dei bit di codifica in tali registri a partire dai valori salvati nell'array. Questo avviene mediante la funzione `output_gen()`, a cui si passa il numero totale di errori rilevati e la posizione riferita a quale registro di uscita occorre salvare tale dato; l'array di salvataggio non viene passato in quanto si tratta di una variabile globale e, di conseguenza, già accessibile alla funzione.

```
. . .
/* PERFORM TEST on reg # v_cnt */
if (cnt_byte >= NUM_BYTES_WORD) {
    cnt_byte = 0; // reset
    uc_reg_addr += UC_REGS_ADDR_INCR;
    rd_before_wr = word_reg;
    val_to_be_written = ~rd_before_wr;
    *(word_addr) = val_to_be_written;
    rd_after_wr = *(word_addr);
    for (uint8_t i = 0; i < NUM_BYTES_WORD; i++) {
        mask_check |= (0xFF << 8*i);
        policy_to_check |= ((policy_word & mask_check) >> 8*i);
        access_to_check |= ((accessibility_word & mask_check) >> 8*i);
        if (policy_to_check == RD_WR) {
            val_desired = ((val_to_be_written >> 8*i) & access_to_check);
            if ((val_desired) != ((rd_after_wr & mask_check) >> 8*i)) {
                cnt_mismatches += 1;
                encode_addr((((uint32_t) word_addr) + i));
            }
        }
        mask_check = RST;
        policy_to_check = RST;
        access_to_check = RST;
    }
    policy_word = RST;
    accessibility_word = RST;
}
/* END TEST on reg # v_cnt */
. . .
```

Figura 4.10: *Cuore del test della policy di READ/WRITE.*

4.3.3 Sviluppo del test UVM SystemVerilog

Per quanto concerne il lato UVM, il flusso perseguito è lo stesso descritto nell'introduzione ai test della memoria dati, al **Paragrafo 3.3.4**. Ciò che cambia è l'implementazione delle classi che definiscono il comportamento caratterizzante i nuovi test a livello di tipologia di informazioni processate, mentre la parte di configurazione iniziale del microcontrollore rimane invariata.

Configurazione dei parametri

La classe di randomizzazione estende dalla classe base e ridefinisce i *constraint* già introdotti nel test della RAM al **Paragrafo 3.3.4**. Viene creata la funzione di `algo_assign()` per definire quale algoritmo abilitare; il parametro su cui opera è chiamato `uc_algo_en` (un array multidimensionale per tenere al suo interno in modo leggibile sotto forma di matrice tutti i bit associati agli algoritmi da abilitare) e viene re-impostato prima dell'esecuzione di un nuovo test, sulla base di quale policy si desidera testare. La funzione viene invocata all'interno della `post_randomize()`, ossia dopo che sono stati generati i vincoli ai parametri di simulazione e prima che venga configurato il microcontrollore tramite la programmazione della memoria codice e dei registri di configurazione e che vengano lanciati i modelli per la generazione delle uscite digitali delle catene di elaborazione dei sensori.

Sequenza di test

La sequenza di test estende quella del test della catena ed è costituito dagli stessi task definiti per il test della RAM, con la differenza che viene ridefinito da zero il task `uc_regs_check()`. All'interno di quest'ultimo viene testato il valore del parametro `uc_algo_en`. Nel caso in esame, per la policy $READ/WRITE$ si ha, alla riga zero dell'array multidimensionale, `uc_algo_en[0] = 8'b0000_0001` (corrispondente all'algoritmo 0) e per questa casistica le operazioni da eseguire consistono in:

- attendere lo spegnimento del microcontrollore;
- avviare la lettura dei registri di uscita tramite l'abilitazione all'accesso alla pagina dei registri interni al microcontrollore. Al termine della lettura, l'interfaccia viene riportata all'offset associato agli indirizzi della *default page*.

Scoreboard

La classe scoreboard estende quella di base e ridefinisce la funzione di scrittura dei pacchetti ricevuti attraverso le *analysis port*. In particolare, vengono eseguiti dei controlli differenti sulla base di quale algoritmo risulta abilitato. Nel caso in esame:

- Quando si riceve il pacchetto corrispondente alla lettura del registro utilizzato dal core per salvare il totale dei *mismatch*, si decodifica il numero totale di disallineamenti

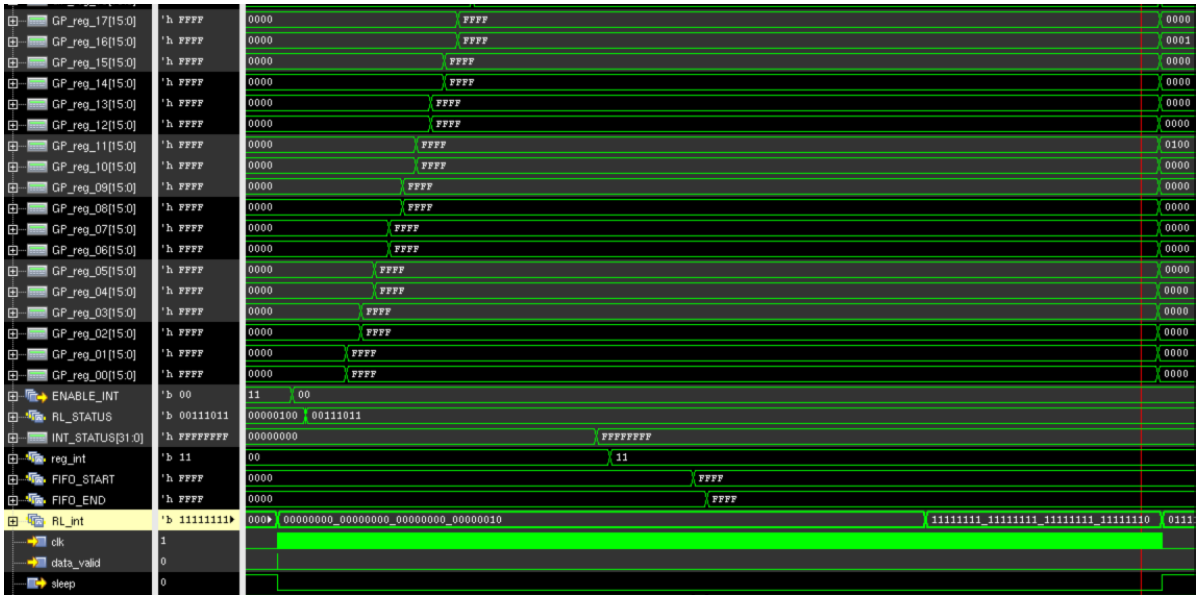
che sono stati salvati nella variabile a 16 bit per il conteggio degli errori. Tale variabile ha una dimensione coincidente con quella dei registri di uscita messi a disposizione, dal momento che risultava privo di utilità avere una variabile su 32 bit. Infatti, siccome vengono conteggiati i byte errati, occorrerebbero in tutto $2^{16} - 1$ byte per saturare la dinamica di uscita, corrispondenti a un banco composto da più di sedicimila registri da 32 bit, un numero molto maggiore del valore tipico di qualche decina o centinaia di registri definiti in un microcontrollore. Pertanto, il *worst case* in cui tutti i registri falliscono viene coperto senza saturare l'uscita.

- Quando si ricevono tutti gli altri pacchetti associati al resto dei registri di uscita, viene applicata la decodifica già applicata nei test della RAM, con la differenza che in questo caso basta decodificare in decimale i bit di un solo registro.

4.3.4 Simulazioni e risultati ottenuti: *Microcontroller Registers*

Le simulazioni hanno permesso di verificare l'andamento delle *waveform* associate ai registri sotto test. In **Figura 4.11** sono raffigurati a titolo dimostrativo alcuni dei registri del banco dei registri interni al microcontrollore aventi policy $READ/WRITE$, disposti tutti al di sopra del segnale di temporizzazione. È possibile apprezzare l'inversione di tutti i bit eseguita dal core, la quale avviene in istanti temporali differenti in funzione della posizione del registro all'interno della *register map*. Al di sotto delle forme d'onda di interesse, si riportano i messaggi di *log* stampati dall'ambiente UVM durante le varie fasi di simulazione, scandite dalla fase di configurazione, dallo *start* dei clock del microcontrollore e dal controllo dei pacchetti di lettura da interfaccia al termine dell'esecuzione del core, terminando con le scritture da interfaccia per lo spegnimento del dispositivo.

In generale, tutti i registri il cui accesso è definito sia in lettura sia in scrittura si comportano correttamente, a parte il registro *RL_int*, evidenziato in giallo paglierino. Quando il core termina l'esecuzione delle istruzioni di auto-test, i registri di uscita *General Purpose* vengono sovrascritti; nel caso in esame, nel *GP_reg_16* viene scritto `0x0001`, corrispondente al numero di byte errati che sono stati rilevati, invece il valore scritto in *GP_reg_11* codifica l'indirizzo errato equivalente a `START_uc_reg + 0xb8`.




```
1  UVM_INFO @ 162423347.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [
       uvm_test_top.sve_top.mss.seq_basic.seq_chain_inst] Algos enable complete
2  UVM_INFO @ 162430900.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [
       uvm_test_top.sve_top.mss.seq_basic.seq_chain_inst] Algos odr scaler complete
3  UVM_INFO @ 162466615.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [
       uvm_test_top.sve_top.mss.seq_basic.seq_chain_inst] INT1_CTRL and INT2_CTRL write
       complete
4  UVM_INFO @ 162502831.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [
       uvm_test_top.sve_top.mss.seq_basic.seq_chain_inst] Enable int complete
5  UVM_INFO @ 162517429.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [
       uvm_test_top.sve_top.mss.seq_basic.seq_chain_inst] Microcontroller start complete
6
7  UVM_ERROR @ 200521583.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
       .scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0xb8 do
       not respect assigned policy requirement
8  UVM_ERROR @ 200802795.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top
       .scoreboard_inst] Policy access violations of registers RD/WR have been found
       during uc registers' test.
9  The overall number of Bytes of the uc registers which does not respect the policy RD/WR
        is            1
10
11 UVM_INFO @ 200818148.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [
       seq_chain_inst] switch off
```

Figura 4.11: *Risultato della simulazione relativa al test di self-checking dei registri interni al microcontrollore con policy READ/WRITE.*

Il comportamento non risulta coerente con quanto riportato nella tabella delle specifiche (**Figura 4.12**), ma è spiegabile attraverso il confronto delle informazioni del *Technical Report*. Il bit in questione è l'INT_A_00, che rimane a '1' anche dopo la scrittura a bit negati. Ciò è spiegabile con la funzionalità che tali bit implementano: se il bit i-esimo viene impostato a '1', significa che è stata generata la IRQ per l'esecuzione dell'algoritmo $i - 1$. Tale bit rimane abilitato sino a che la routine di esecuzione di

quell'algoritmo non viene completata. Siccome viene eseguito l'algoritmo 0, il bit alla posizione 1 di tale registro non viene invertito.
Invece, per quanto riguarda il bit alla posizione zero, esso non viene invertito in quanto quel bit risulta inaccessibile al core e pertanto non viene testato, grazie alle operazioni di *masking* che sono state eseguite sul byte acquisito in fase di test.

Infine, il registro *RL_STATUS* contiene dei bit che non sono stati invertiti, ma nessun errore viene segnalato per quell'indirizzo: questo comportamento è coerente con quanto riportato nel file dei requisiti, in quanto i bit non invertiti non sono accessibili dal core, come indicato in **Figura 4.12**. Di conseguenza, il test funziona correttamente anche per quanto concerne la generazione e l'applicazione delle maschere di accessibilità.

RL_ADDR		7	6	5	4	3	2	1	0
0x04		---	---	GPIO[3]	GPIO[2]	GPIO[1]	GPIO[0]	RAM_FAIL	ORUN
0xb8		INT_A_06	INT_A_05	INT_A_04	INT_A_03	INT_A_02	INT_A_01	INT_A_00	---
0xb9		INT_A_14	INT_A_13	INT_A_12	INT_A_11	INT_A_10	INT_A_09	INT_A_08	INT_A_07
0xba		INT_A_22	INT_A_21	INT_A_20	INT_A_19	INT_A_18	INT_A_17	INT_A_16	INT_A_15
0xbb		INT_ORUN	INT_A_29	INT_A_28	INT_A_27	INT_A_26	INT_A_25	INT_A_24	INT_A_23

name (RL side)	RL_readable	RL_writable
RL_ADDR_RL_STATUS	Yes	Yes
RL_ADDR_RL_INT_A	Yes	Yes
RL_ADDR_RL_INT_B	Yes	Yes
RL_ADDR_RL_INT_C	Yes	Yes
RL_ADDR_RL_INT_D	Yes	Yes

Figura 4.12: *Estratto del file* `.xml` *contenente soltanto gli indirizzi di interesse il cui comportamento risulta critico ai fini dell'esito corretto del test.*

In sintesi, il test verifica correttamente le policy di accesso *READ/WRITE* che sono state fornite per il banco sotto test, tuttavia esiste un bit che non rispetta le specifiche di ingresso, ma il suo comportamento è stato correttamente previsto e dichiarato dal designer all'interno del manuale tecnico del progetto. Pertanto, l'errore rilevato è dovuto a una informazione incompleta nel file `.xml` utilizzato dal test e non a un comportamento imprevisto del design.

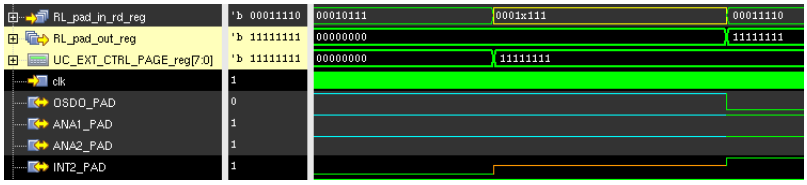
4.3.5 Simulazioni e risultati ottenuti: *System Registers*

All'interno della pagina della *register map* associata ai registri di sistema, i registri che si comportano secondo la policy di accesso *READ/WRITE* sono:

- *UC_EXT_CTRL_PAGE_reg*, i cui bit consentono di abilitare il controllo dei *pad* esterni tramite il core e di abilitare l'accesso ad alcune configurazioni dedicate alla programmazione dei sensori tramite utente;

- *RL_pad_out_reg*, i cui bit consentono di impostare tramite core lo stato logico delle linee sui *pad* esterni.

L'implementazione del test è analoga a quella descritta al **Paragrafo 4.3** e sfrutta le medesime routine di test.




```
1 . . .
2 UVM_INFO @ 155729459.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
      scoreboard_inst] No errors have been found during checking of the policy RD/WR of the system
      registers' test
3 UVM_INFO @ 155941544.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
      switch off
```

Figura 4.13: *Forme d'onda della simulazione associata al test della policy READ/WRITE per i registri di sistema.*

Come si può evincere dal risultato di simulazione, entrambi i registri presentano un comportamento coerente rispetto a quello dichiarato dalla policy.

Per completezza, sono state riportate anche le forme d'onda associate alle linee dei *pad* esterni, il cui comportamento è coerente con quanto riportato dalle specifiche del manuale tecnico di riferimento.
Ad esempio, l'*interrupt* $INT2$ viene asserito dopo che si verificano, nell'ordine, le seguenti condizioni:

- inizialmente si abilita il pilotaggio della linea tramite core scrivendo a '1' il bit corrispondente nel registro $UC\_EXT\_CTRL\_PAGE\_reg$; in questo modo la linea risulta in alta impedenza (colore arancio);
- il segnale $INT2$ viene asserito automaticamente dopo che, tramite il core, viene scritto a '1' il bit corrispondente nel registro $RL\_pad\_out\_reg$.

4.4 Test della policy READ-ONLY

4.4.1 Obiettivi del test

Il test della seconda policy, implementato sull'algoritmo 1, prevede la verifica dei byte aventi la sola lettura come modalità di accesso dal core. Occorre pertanto verificare che il core non sia in grado di accedere in scrittura e modificare tali byte e, al tempo stesso, avere la certezza che i valori presenti a quegli indirizzi siano letti correttamente.
I test sviluppati prevedono l'applicazione delle routine C sia sui valori di inizializzazione di tali registri, sia su bit invertiti inseriti dall'esterno, in modo tale da assicurare la corretta lettura coprendo tutti i possibili valori di ciascun bit.

4.4.2 Sviluppo del test C

La fase di acquisizione delle informazioni codificate nelle stringhe generate dal Python è analoga a quella del test precedente. Il medesimo meccanismo vale anche per i test successivi. La parte che cambia è il cuore del test, ossia le operazioni base la cui combinazione permette una verifica completa della funzionalità *RDONLY*. In particolare, il test prevede l'elaborazione delle stringhe di codifica due volte, intervallate dalle operazioni UVM. Il flusso di test è rappresentato in **Figura 4.14**.

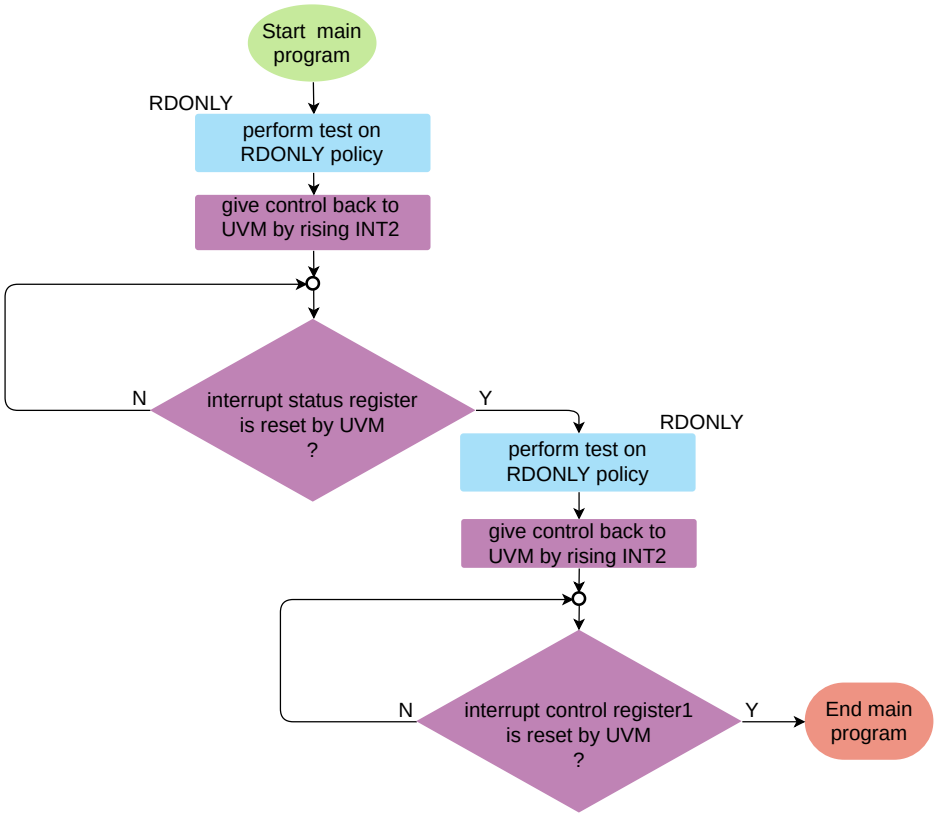


Figura 4.14: *Macro-blocchi del programma di test della policy RDONLY.*

Le operazioni descritte nel diagramma sono implementate dal seguente codice C, dove la routine di `perform_RDO()` permette l'analisi delle stringhe generate dal Python, l'esecuzione delle operazioni di *self-checking* e il salvataggio delle uscite.

```
/* #define cast_uint32_t (address_reg_name) *((volatile uint16_t *) (address_reg_name))
    */
// begin routine associated with algo_01
extern void test_uc_regs_RDONLY(void) {
    // 1st self-check
    perform_RDO(18); // 18 for saving counter of mismatching bytes inside GP_reg_18

    // give control to UVM
    cast_uint32_t(UC_EXT_CTRL_PAGE) |= (1 << 5);
    cast_uint32_t(RL_PAD_OUT_ADD) |= 0xC0;
    while (cast_uint32_t(RL_ADDR_INT_STATUS) == RST) {
    }

    // 2nd self-check
    perform_RDO(17);

    // give control to UVM
    cast_uint32_t(RL_ADDR_INT_STATUS) |= (1 << 0); // to assert INT1
    cast_uint32_t(RL_PAD_OUT_ADD) &= (~0xC0); // clear bit 7 and 6 (de-assert INT2)
    while (cast_uint32_t(RL_ADDR_INT1_CTRL) == RST) {
    }
}// end test_uc_regs_RDONLY
```

Il test consiste nei seguenti passi base.

- Quando una *word* è acquisita, si eseguono le istruzioni che verificano con auto-test che i registri rispettino la policy. Il principio di funzionamento è mostrato in **Figura 4.15** e corrisponde a parte del contenuto della routine di test `perform_RDO()` sopra introdotta.
  - Si esegue una prima lettura all'indirizzo associato alla *word*.
  - Si inverte il valore letto.
  - Si tenta di scrivere il nuovo valore al medesimo indirizzo.
  - Si esegue la seconda lettura.
  - Si applicano le maschere di policy e accessibilità per testare i singoli byte. Se questi differiscono tra una lettura e l'altra, viene rilevato l'errore.
- In caso di *mismatch* tra la policy e il comportamento effettivo, viene incrementato il contatore di errore e viene codificato l'indirizzo all'interno di un array che emula il sotto-banco di registri di uscita, in modo analogo al test precedente.
- Quando tutti i registri sono stati processati, si generano le uscite scrivendo il contenuto dell'array di salvataggio in modo sequenziale sui bit dei registri *General Purpose*. Dopodiché, viene ceduto il controllo all'ambiente UVM e il core rimane in un *loop* di attesa, all'interno del quale non vengono eseguite nuove istruzioni ma viene interrogato in *polling* il contenuto del registro di stato degli *interrupt*, al fine di comparare i bit correnti con il valore che si attende che l'ambiente UVM imposti per far ripartire l'esecuzione delle istruzioni di programma nel core. Per le

operazioni operazioni UVM SystemVerilog si rimanda al **Paragrafo 4.4.3**, in cui si applica il *deposit* sui registri di interesse.

- Quando il core esce dal *loop* di attesa, viene rieseguita la routine di test, salvando il conteggio dei disallineamenti rilevati in un registro di uscita differente da quello utilizzato nel primo caso.
- Al termine della routine di test, sulle uscite sono asseriti i bit di tutti gli indirizzi che hanno sbagliato al primo ciclo di test, oppure al secondo, oppure in entrambi i cicli. I registri non vengono sovrascritti in quanto la scrittura dei bit salvati nell'array verso i registri avviene con una operazione di *bitwise OR*.
- Infine, viene ceduto nuovamente il controllo all'ambiente UVM, con l'obiettivo di effettuare il *release* dei segnali forzati. Tale operazione non è necessaria se viene utilizzato il *deposit* anziché il *force*, in quanto il tal caso l'operazione di *release* è già inclusa nel comando.

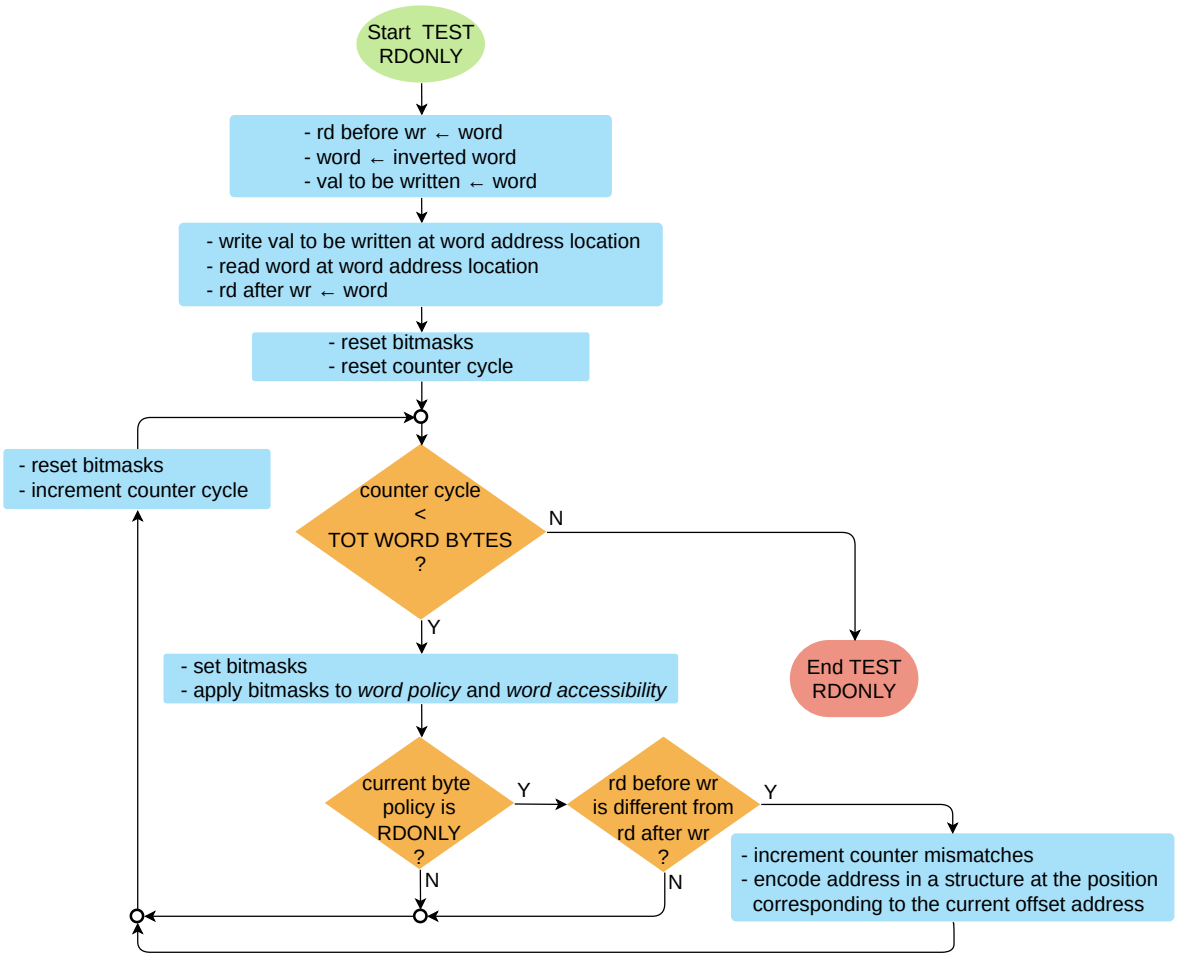


Figura 4.15: *Dettaglio del flusso di auto-test della policy RDONLY eseguito dal core.*

4.4.3 Sviluppo del test UVM SystemVerilog

Configurazione dei parametri

Per questo test, all'interno della classe di randomizzazione, occorre impostare il parametro di abilitazione degli algoritmi al valore `uc_algo_en[0] = 8'b0000_0010`. In questo modo, viene abilitato l'algoritmo 1. La complessità rispetto al test precedente sta nell'implementazione della sequenza di operazioni da eseguire per questo nuovo *test case*.

Sequenza di test

All'interno della sequenza definita per il test dei registri del microcontrollore, viene eseguito il codice associato al *case* della policy, sulla base del numero dell'algoritmo che è stato abilitato. Vengono eseguiti in parallelo i blocchi elencati di seguito.

- Quando vengono asseriti, nell'ordine, i *pad* $INT1$ e $INT2$, vengono effettuati i *deposit* nei registri aventi nominalmente la policy *RDONLY*. I valori che vengono depositati hanno tutti i bit invertiti rispetto al valore corrente acquisito mediante *Virtual Interface*. Il controllo viene ceduto nuovamente al core resettando il registro di stato degli *interrupt*.

- Quando, nell'ordine, viene messo al livello logico '1' il *pad* $INT1$ e viene messo a '0' il *pad* $INT2$, vengono effettuati i *release* di tutti i registri *RDONLY*. Tale operazione non viene mai eseguita in questo test in particolare, dal momento che nel blocco precedente sono stati effettuati dei *deposit* al posto di operazioni di *force*. In questo caso il controllo viene ceduto al core invertendo i bit del registro di controllo *INT1_CTRL*.

- L'ultimo blocco viene eseguito non appena viene asserito il segnale di *sleep*; tale blocco si occupa delle letture da interfaccia dei risultati prodotti.

Scoreboard

Nella classe scoreboard, la funzione che si occupa del controllo dei pacchetti ricevuti esegue le seguenti operazioni:

- per i pacchetti associati ai registri di uscita che contengono la codifica degli indirizzi dei byte che falliscono, viene effettuata la decodifica e si riportano gli indirizzi;

- se il pacchetto è associato a un registro di uscita adibito al salvataggio del numero di disallineamenti del primo ciclo di scrittura-lettura-controllo, si decodifica in decimale il risultato;

- se il pacchetto è associato a un registro di uscita adibito alla memorizzazione del conteggio dei disallineamenti del secondo ciclo, si salva il risultato decodificato in un'altra variabile rispetto al caso precedente.

4.4.4 Simulazioni e risultati ottenuti: *Microcontroller Registers*

In **Figura 4.16** è possibile osservare quanto ottenuto dalla simulazione applicata ai registri interni al microcontrollore.

Partendo dai registri che non presentano criticità, si ha che la policy *RDONLY* viene rispettata sia dai due registri di controllo degli *interrupt*, sia dai registri che accolgono i dati provenienti dai sensori e dal registro *ALGO_TO_RUN*.

Per quanto concerne i registri critici invece, vanno presi in considerazione i due segnali evidenziati in giallo paglierino.

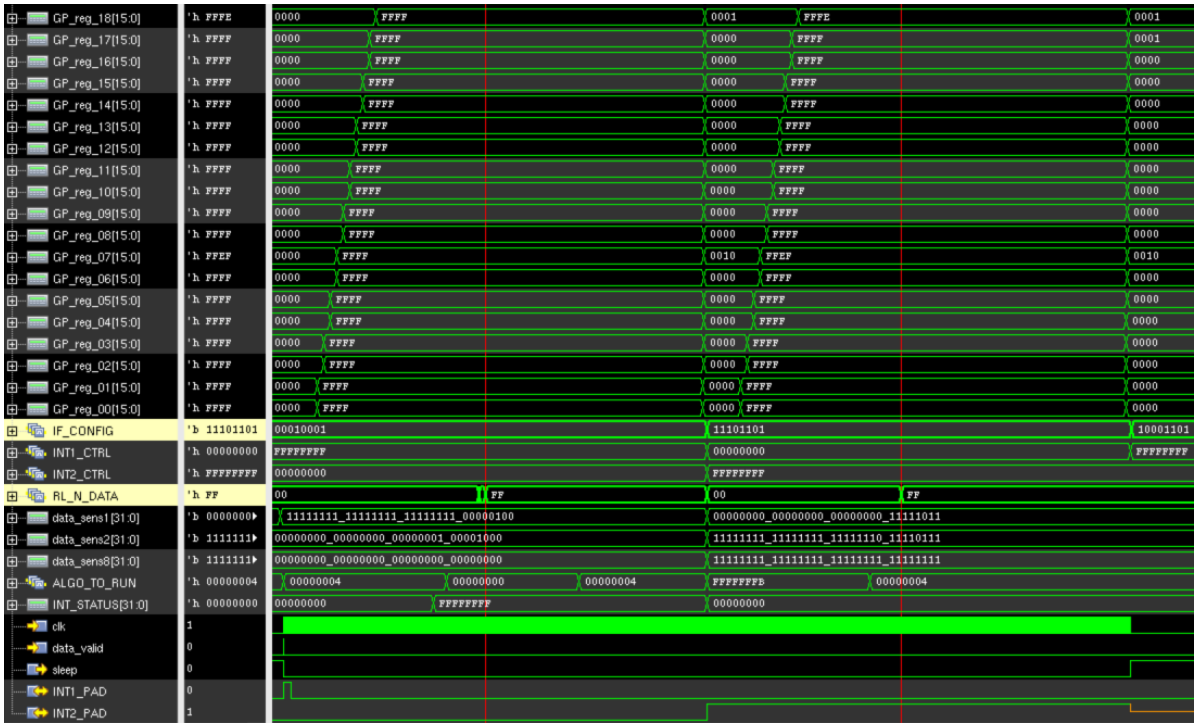


Figura 4.16: *Risultati salienti della simulazione associata al test della policy RDONLY per i registri interni al microcontrollore.*

```
1  . . .
2  UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] INT2 asserted after INT1. Perform deposit on RDONLY regs
3  UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:IF_CONFIG
4  param2 deposit : 11101101
5  UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT1_CTRL
6  param2 deposit : 00000000000000000000000000000000
7  UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT2_CTRL
8  param2 deposit : 11111111111111111111111111111111
9  UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:RL_N_DATA
10 param2 deposit : 00000000
11 UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:data_sens1
12 param2 deposit : 00000000000000000000000011111011
13 UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:data_sens2
14 param2 deposit : 11111111111111111111111011110111
15 . . .
16 param2 deposit : 11111111111111111111111111111111
17 UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:data_sens8
18 param2 deposit : 11111111111111111111111111111111
19 . . .
20 UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:ALGO_TO_RUN
21 param2 deposit : 11111111111111111111111111111011
22 UVM_INFO @ 188480946.01 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT_STATUS
23 param2 deposit : 00000000000000000000000000000000
24
25 UVM_INFO @ 195396527.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] INT1 asserted and INT2 de-asserted. Release signals
26 UVM_INFO @ 195396527.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT1_CTRL
27 param2 deposit : 11111111111111111111111111111111
28 UVM_INFO @ 195401482.82 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] sleep asserted. Core execution terminated
29
30 UVM_ERROR @ 195529136.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0x74 do not respect
       assigned policy requirement
31 UVM_ERROR @ 195860016.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] Policy access violations of registers RDONLY have been found during uc registers
       ' test.
32 The overall number of Bytes of the uc registers which does not respect the policy RDONLY is
        1
33
34 UVM_INFO @ 195889061.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
       switch off
```

Figura 4.17: *Messaggi UVM salienti corrispondenti alle varie fasi di simulazione in cui si articola il test della policy RDONLY per i registri interni al microcontrollore.*

- Il registro *IF_CONFIG*, quando viene manipolato dall'ambiente UVM, non inverte tutti i bit, ma lascia invariato il valore dei primi due. Questo comportamento è stato ottenuto volutamente in fase di sviluppo del test, dal momento che, come indicato nell'estratto del file `.xml` in **Figura 4.18**, i primi due bit sono associati al segnale di reset e al segnale di abilitazione del clock. Affinché il test possa avvenire correttamente, questi bit di configurazione non devono essere modificati durante la simulazione.

- Il byte indicato come *RL_N_DATA*, invece, presenta un comportamento anomalo, non previsto dalla specifica assegnata. Quando il core tenta di invertire i suoi bit, riesce a portare a termine l'operazione, causando la registrazione dell'errore a quell'indirizzo (come riportato nel file di *log*) a seguito della violazione della policy di accesso.

RL_ADDR	7	6	5	4	3	2	1	0
0x02	RAM_TEST@BOOT	---	---	LATCHED[1]	LATCHED[0]	BOOT_FM_RAM	CLK_DIS	RST_N
0x74	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

name (RL side)	RL_readable	RL_writable
RL_ADDR_IF_CONFIG	Yes	ignored
RL_ADDR_FIFO_N_DATA	Yes	ignored

Figura 4.18: *Estratto del file di requisiti relativo alle casistiche critiche.*

Pertanto, si può concludere che il design soddisfa il test di *self-checking* utilizzato per tutti i registri interni al microcontrollore aventi policy *RDONLY*, a parte per il comportamento anomalo di un byte. Tale byte può essere non solo letto ma anche scritto dal core, contrariamente a quanto ci si attende.

4.4.5 Simulazioni e risultati ottenuti: *System Registers*

La maggior parte dei registri associati al banco di registri di sistema hanno come unica modalità di accesso tramite core quella di sola lettura. Anche per questo banco di registri sono state applicate le medesime routine di test implementate per il banco dei registri interni.

In **Figura 4.19** sono stati riportati alcuni dei registri sotto test. In particolare, su quelli evidenziati sono stati rilevati dal *self-test* dei comportamenti anomali rispetto alla policy nominale. In tutti gli altri casi invece, la funzionalità dei registri è stata verificata correttamente.

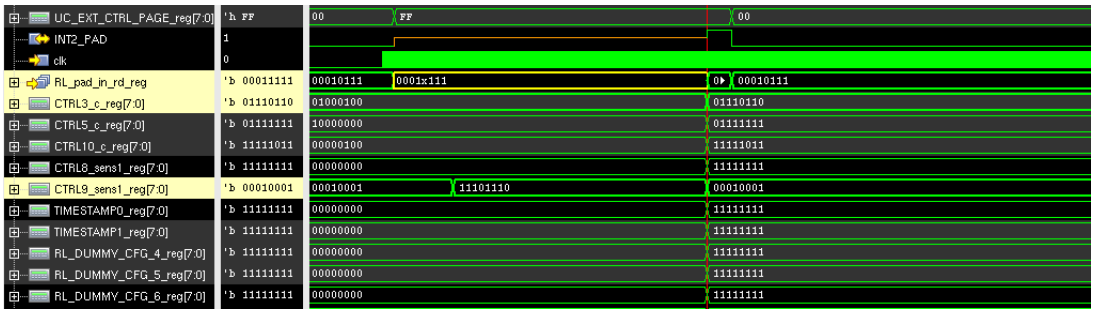


Figura 4.19: *Comportamento di alcuni dei registri con policy RDONLY appartenenti al banco di registri di sistema.*

```
1  . . .
2  UVM_INFO @ 147828556.16 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Bit 0 of INT_STATUS and INT2 asserted. Perform deposit on registers
        having policy RDONLY regs
3  UVM_INFO @ 147828556.16 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:
       registers_inst:sys_regs_inst:RL_pad_in_rd_reg
4  param2 : 11101000
5  UVM_INFO @ 147828556.16 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:
       registers_inst:sys_regs_inst:CTRL3_c_reg
6  param2 : 01110110
7  UVM_INFO @ 147828556.16 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:
       registers_inst:sys_regs_inst:CTRL5_c_reg
8  param2 : 01111111
9  . . .
10 UVM_INFO @ 147828556.16 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:
       registers_inst:sys_regs_inst:RL_DUMMYCFG_6_reg
11 param2 : 11111111
12 . . .
13 UVM_ERROR @ 155091199.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_sys_regs + 0x02 do not respect
       assigned policy requirement
14 UVM_ERROR @ 155106547.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_sys_regs + 0x12 do not respect
       assigned policy requirement
15 UVM_ERROR @ 155113587.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_sys_regs + 0x18 do not respect
       assigned policy requirement
16 UVM_ERROR @ 155352947.00 ns: uvm_test_top.sve_top.scoreboard_inst [scoreboard_inst] Policy access
       violations of registers RDONLY have been found during system registers' test.
17 Moreover, the number of mismatches in the first core test differs from the second one coming after
       interface operations.
18 1st core op: 3
19 2nd core op: 2
20 The behaviour is abnormal with respect to the policy
21 UVM_ERROR @ 155352947.00 ns: uvm_test_top.sve_top.scoreboard_inst [scoreboard_inst] The overall
       number of Policy access Bytes violations found during RDONLY test is 3
22 UVM_INFO @ 155550952.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
       switch off
```

Figura 4.20: *Messaggi UVM associati al test della policy RDONLY sui registri di sistema.*

Analizzando in particolare i fallimenti, si hanno i seguenti comportamenti critici.

- *RL_pad_in_rd_reg* contiene il bit associato all'ingresso del *pad INT*2 e viene automaticamente messo al valore 'x' (*don't care*) quando il core abilita il pilotaggio della linea di *interrupt* e viene resettato quando viene abbassato il *bit*0 di *UC_EXT_CTRL_PAGE_reg*. Tale comportamento risulta coerente con quando descritto nel *Technical Report*, ma non è stato dettagliato nel file dei requisiti utilizzato dal test, di conseguenza nessuna maschera è stata applicata per far fronte a tale eccezione e il test ha rilevato tale incongruenza rispetto alla policy fornita.

- *CTRL*3*_c_reg* ha un comportamento deterministico voluto appositamente dal test, infatti lato ambiente UVM è stata applicata una maschera che mantenesse i valori dei seguenti bit:

  · al livello logico '0', il bit di *software reset* e il bit di *boot*, dal momento che nel manuale tecnico è espressamente indicato che essi non debbano essere asseriti contemporaneamente;

  · al livello logico '0', il bit di selezione della modalità d'uso dell'interfaccia SPI, che rimane quella di *default* a 4 fili;

  - al livello logico '1', il bit che permette l'aggiornamento automatico del contenuto dei registri di uscita.

- $CTRL9\_sens1\_reg$ durante la prima iterazione viene scritto, invertendo tutti i bit al suo interno. Confrontando il manuale tecnico, il registro è dichiarato come accessibile da core se è stato abilitato il bit 0 del registro $UC\_EXT\_CTRL\_PAGE\_reg$ (analizzato al **Paragrafo 4.3.5**). Nel test della policy $RDONLY$, tale bit è stato asserito affinché il core fosse in grado di cedere il controllo all'ambiente UVM attivando il segnale di *interrupt* (nella simulazione tutti i bit del registro $UC\_EXT\_CTRL\_PAGE\_reg$ sono a '1' in quanto durante la verifica della policy il core tenta di effettuare una scrittura su ogni *word* da 32 bit, pertanto all'interno della *word* vengono invertiti i byte aventi policy $READ/WRITE$). Di conseguenza, la scrittura del core ha effetto alla prima iterazione, mentre alla seconda, quando il bit di accesso è disabilitato e dopo che è stato eseguito il *deposit* via testbench UVM, il core non inverte più tutti i bit.
  Si può quindi affermare che la policy $RDONLY$ assegnata al registro $CTRL9\_sens1\_reg$ sia in netto contrasto con quanto dichiarato nel manuale del microcontrollore.

4.5 Test della policy WRITE-ONLY

4.5.1 Obiettivi del test

Per avere un test esaustivo della modalità di accesso in sola scrittura, occorre utilizzare in modo sinergico sia l'esecuzione dell'algoritmo da core, sia la forzatura di bit da ambiente UVM. L'idea di base del funzionamento del test è la seguente: se occorre dimostrare che il core possa solamente scrivere, è prima necessario verificare che il core stesso non sia in grado di leggere i bit sotto esame.

4.5.2 Sviluppo del test C

I passaggi chiave sono i seguenti.

- Test della non leggibilità.
  - Viene invocata la routine di test per la non leggibilità, la quale tenta di leggere i byte aventi policy assegnata *WRONLY* e salva i risultati di lettura in un array alla posizione corrispondente a quella che il registro ha all'interno della *register map*, a meno dell'offset del banco di registri. Se il registro è *WRONLY*, i byte salvati devono avere tutti valore `0x00` indipendentemente dal loro contenuto.
  - Si cede il controllo all'ambiente di verifica, il quale inverte i bit nei registri aventi la medesima policy.
  - Il core riprende il controllo terminato il ciclo di attesa; viene rieseguita la routine di test precedente, in questo caso i valori acquisiti vengono salvati in un nuovo array.
  - Si comparano i risultati: siccome i tentativi di lettura devono acquisire solo tutti bit a zero, se esiste almeno un elemento diverso da zero all'interno degli array, allora avviene la segnalazione abilitando un flag di errore e riportando gli indirizzi che falliscono corrispondenti a quelle posizioni all'interno degli array.
- Se nessun errore è stato rilevato, si procede con il test della sola scrivibilità.
  - Viene invocata una nuova routine di test, la quale effettua la verifica di scrivibilità tramite la scrittura di byte aventi come valore `0x55`.
  - Siccome il core non è in grado di leggere i valori appena inseriti, viene ceduto il controllo all'ambiente UVM, il quale controlla se i valori scritti sono quelli attesi.
  - Terminato il ciclo di attesa, il core riprende il controllo e riesegue la scrittura invertendo i bit, scrivendo `0xAA`.
  - Viene ceduto nuovamente il controllo all'ambiente UVM, il quale effettua il controllo per verificare se ci sono delle discrepanze rispetto al valore atteso.
  - Il core riprende il controllo e il programma termina.

L'evoluzione del programma, costituita dalle due fasi di test appena descritte, è dettagliata in **Figura 4.21**.

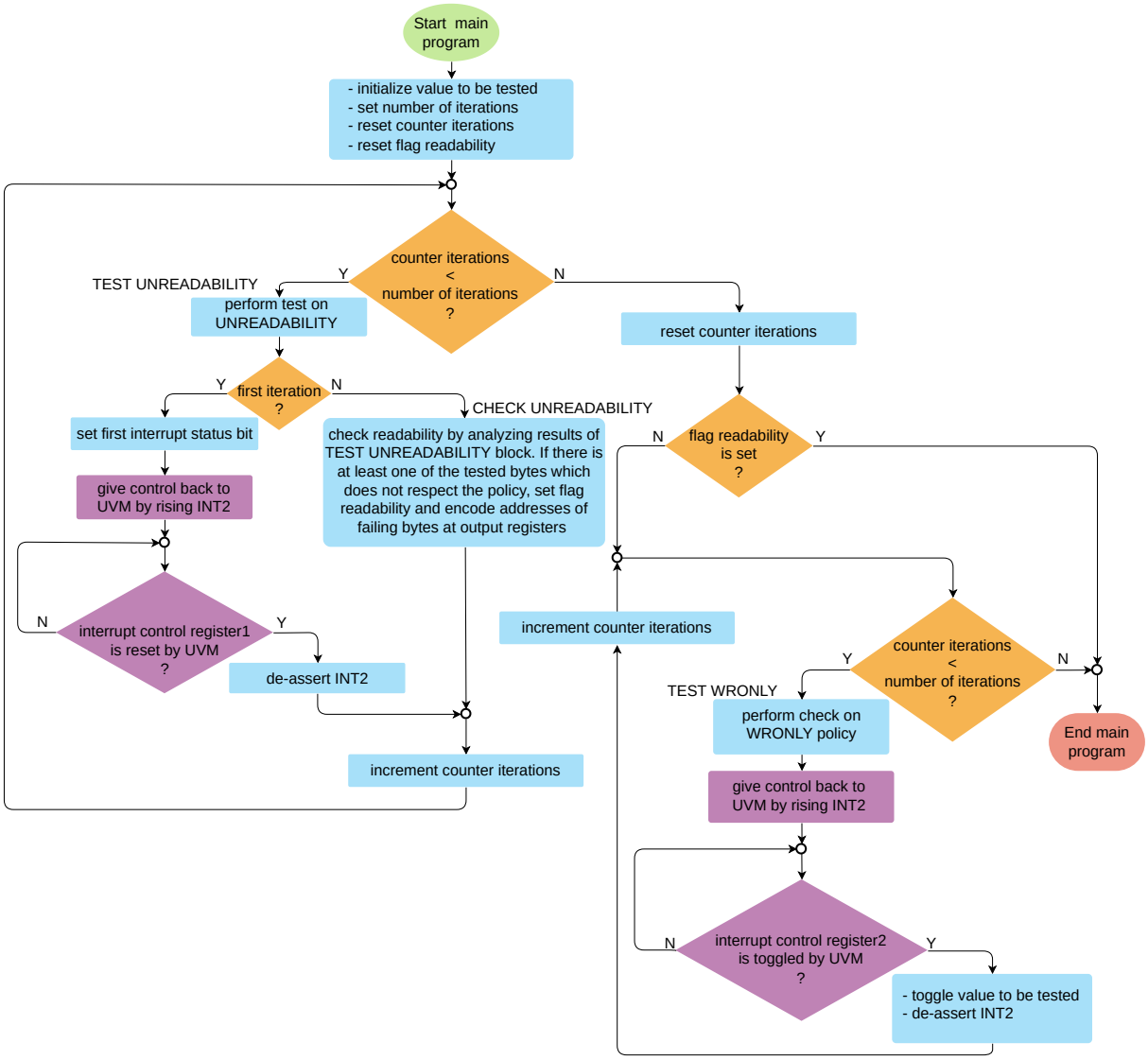


Figura 4.21: *Flusso del programma di self-checking per la policy WRONLY.*

I blocchi indicati come *TEST UNREADABILITY* e *CHECK UNREADABILITY* vengono espansi rispettivamente nel *flow chart* in **Figura 4.22** e nella porzione di codice in **Figura 4.23**.

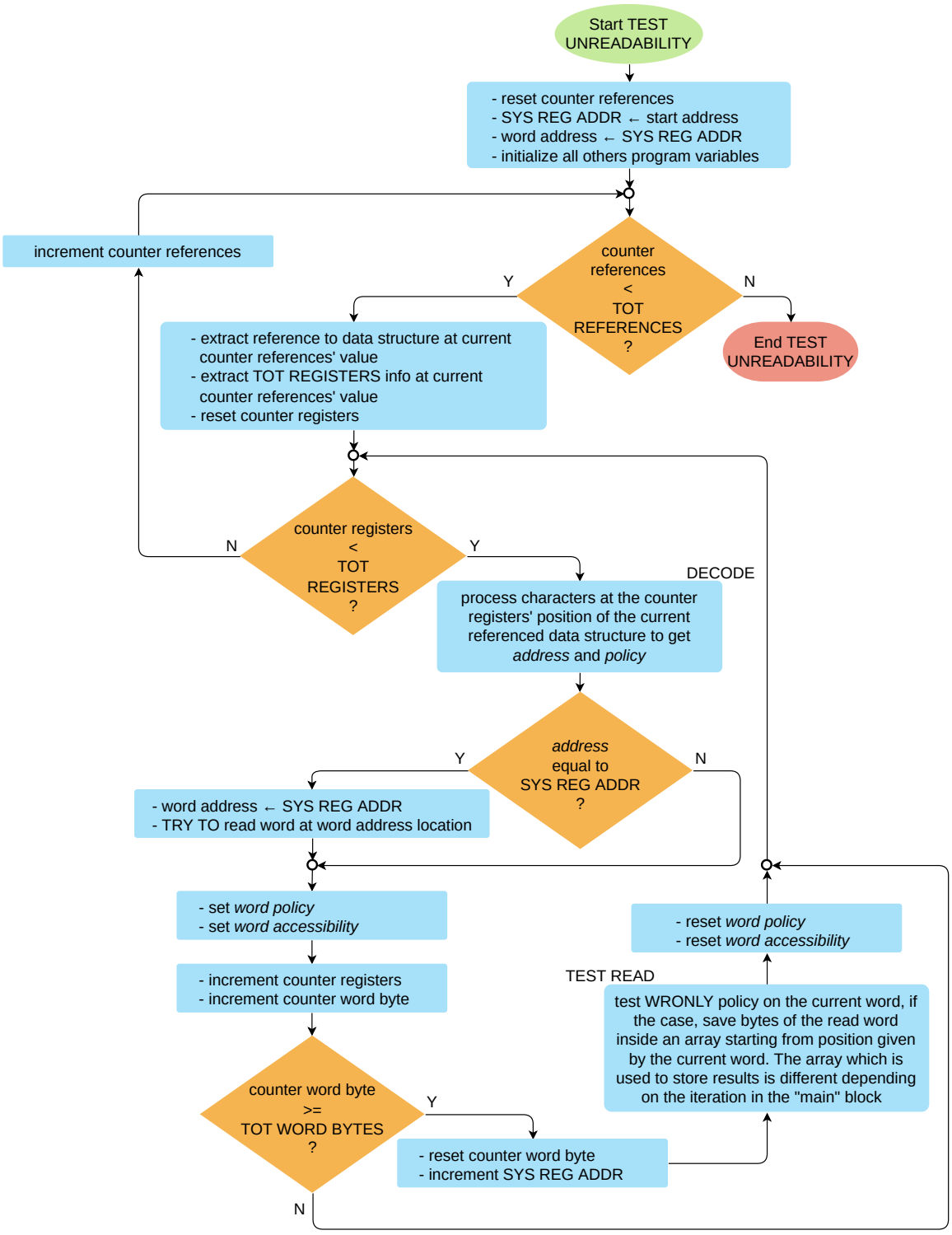


Figura 4.22: *Porzione del test dedicata alla verifica della non leggibilità.*

```
/* global variables */
uint8_t v_bytes_before_wr[NUM_BYTES];
uint8_t v_bytes_after_wr[NUM_BYTES];
. . .
/* test code routines */
. . .
extern uint8_t check_unreadability(void) {
    volatile uint16_t cnt_mismatches = 0;
    uint8_t retval = RST_STATUS;

    for (uint32_t i = 0; i < (sizeof(v_bytes_before_wr)/sizeof(uint8_t)); i++) {
        if ((v_bytes_before_wr[i] != RST ) || (v_bytes_after_wr[i] != RST)) {
          cnt_mismatches++;
          encode_addr((uint32_t) UC_REGS_ADDR_START + i);
        }
    }
    if (cnt_mismatches > 0) {
        retval = SET_STATUS;
    }
    return retval;
}
```

Figura 4.23: *Funzione per il controllo della leggibilità. Se alcuni byte risultano diversi da zero, si invoca la funzione di codifica di indirizzo e al termine si segnala l'errore ritornando un valore di flag.*

I byte negli array di salvataggio alle posizioni non corrispondenti a registri di policy *WRONLY* rimangono al valore zero di inizializzazione, in quanto la lettura viene eseguita solo sui registri sotto test, di conseguenza essi non sono responsabili di eventuali errori funzionali rilevati.

Infine, il blocco indicato come *TEST WRONLY* nella seconda parte del *main* del programma di test della policy, ossia quello per la verifica dell'accesso in scrittura, viene espanso in **Figura 4.24**.

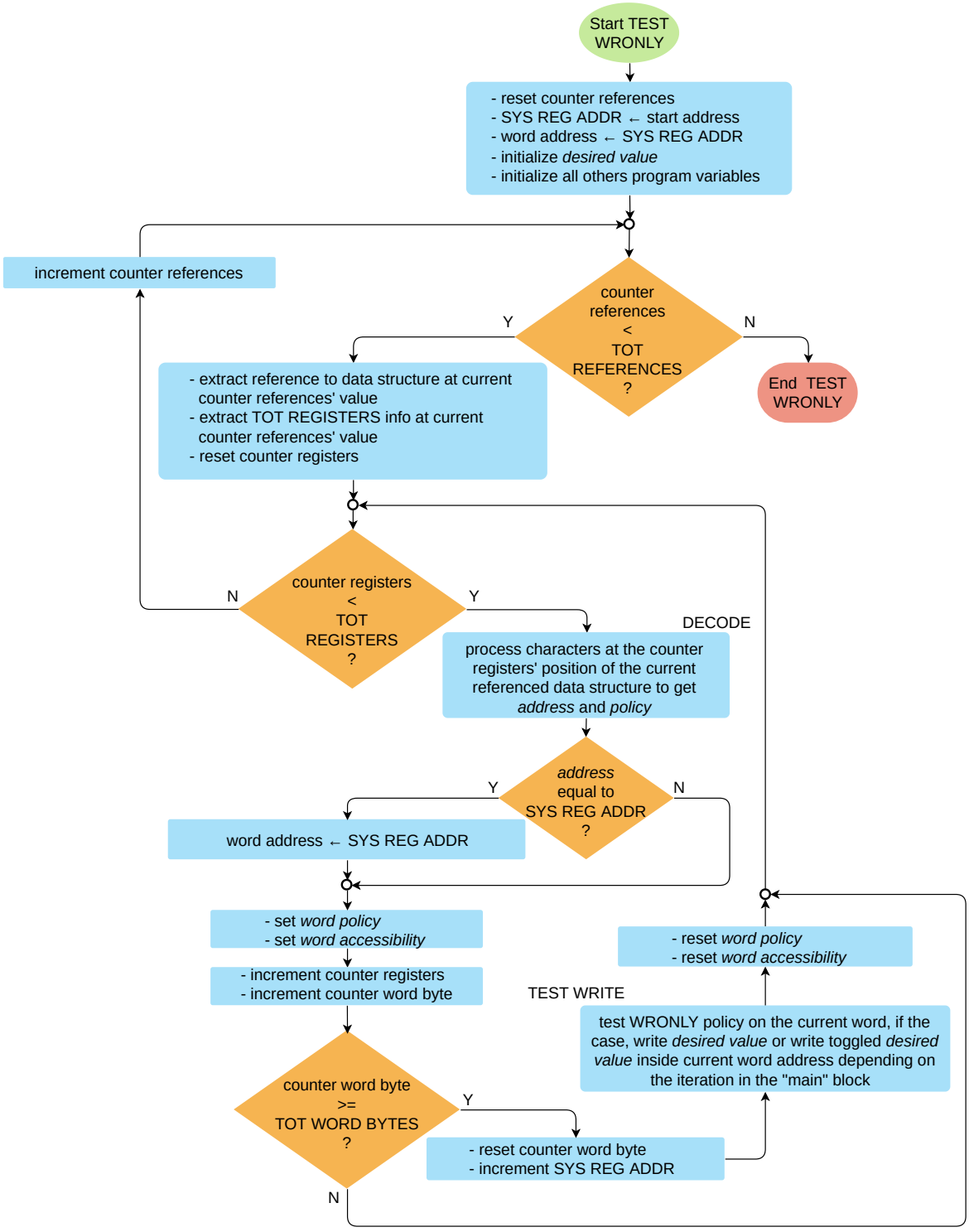


Figura 4.24: *Porzione del test dedicata alla verifica della scrivibilità.*

4.5.3 Sviluppo del test UVM SystemVerilog

Configurazione dei parametri

Il test della policy in oggetto prevede l'abilitazione dell'algoritmo 2, di conseguenza occorre impostare `uc_algo_en[0] = 8'b0000_0100`.

Sequenza di test

La sequenza di operazioni eseguite quando è abilitato l'algoritmo 2 è composta dai seguenti blocchi, ciascuno dei quali entra in gioco quando il core si trova sotto determinate condizioni.

- Quando si hanno, nell'ordine, i fronti di salita associati al *bit*0 del registro di stato degli *interrupt* e al segnale di $INT2$, si invertono tramite una operazione di *deposit* i bit dei registri aventi policy *WRONLY*. Il core esce dal ciclo di attesa quando UVM inverte i bit del registro $INT1\_CTRL$. Inoltre, viene resettato il registro INT_STATUS per le operazioni successive.
- Quando il *bit*0 di INT_STATUS viene resettato e si ha un fronte di salita del segnale $INT2$, si procede con il controllo dei valori scritti dal core. Se ci sono dei disallineamenti, vengono codificati gli indirizzi che falliscono nei registri di uscita tramite l'ambiente di verifica. Il core esce dal ciclo di attesa quando UVM setta i bit del registro $INT2\_CTRL$.
- Quando si ha il fronte di salita del *bit*0 di $INT2\_CTRL$ e viene asserito $INT2$, si procede con il controllo del secondo ciclo di scritture del core, procedendo in modo analogo al caso precedente. Il core esce dal ciclo di attesa quando UVM resetta i bit del registro $INT2\_CTRL$.
- Infine, quando viene asserito il segnale di *sleep*, si procede con le operazioni di lettura da interfaccia dei risultati salvati nei registri di uscita.

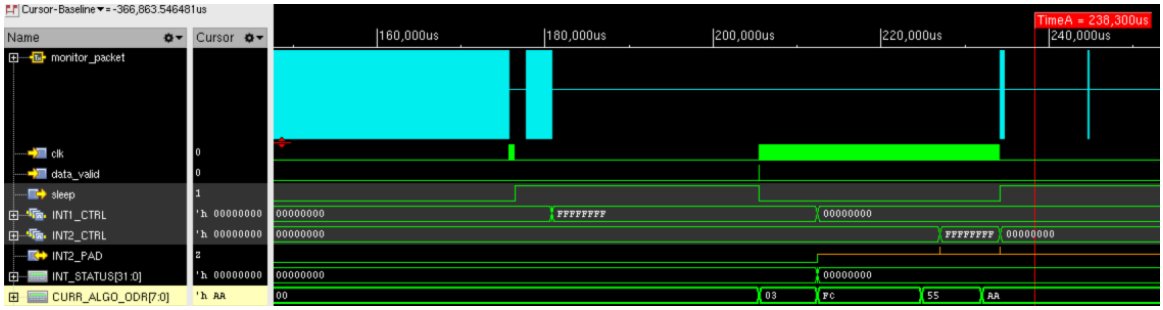
Scoreboard

Per la policy in esame, lo scoreboard si occupa di:

- decodificare gli indirizzi a cui si sono verificati dei *mismatch*;
- estrarre il numero di fallimenti associati a:
  - violazione della policy per via della accessibilità in lettura;
  - violazione della policy per via di valori differenti da quelli attesi nel primo ciclo di scritture da core;
  - violazione della policy per via di valori differenti da quelli attesi nel secondo ciclo di scritture da core.

4.5.4 Simulazioni e risultati ottenuti

Nel caso in esame, esiste solo un byte il cui comportamento segue la policy *WRONLY*, ossia il *CURR_ALGO_ODR*, appartenente al banco dei registri interni al microcontrollore. Come è possibile apprezzare dalla **Figura 4.25**, a inizio esecuzione del programma, esso definisce quale algoritmo deve essere eseguito sulla base degli ODR configurati.




```
1  UVM_INFO @ 180729239.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Algos enable complete
2  UVM_INFO @ 180736792.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Algos odr scaler complete
3  UVM_INFO @ 180772507.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] INT1_CTRL and INT2_CTRL write complete
4  UVM_INFO @ 180808723.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Enable int complete
5  UVM_INFO @ 180823321.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Microcontroller start complete
6
7  UVM_INFO @ 212352557.07 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Bit 0 of INT_STATUS and INT2 asserted : check core non-readability
8  UVM_INFO @ 212352557.07 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:CURR_ALGO_ODR
9  param2 deposit : 11111100
10 UVM_INFO @ 212352557.07 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:int_status
11 param2 deposit : 00000000000000000000000000000000
12 UVM_INFO @ 212352557.07 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT1_CTRL
13 param2 deposit : 00000000000000000000000000000000
14
15 UVM_INFO @ 226942813.31 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Bit 0 of INT_STATUS de-asserted and INT2 assetrted: 1st check core
       writability
16 UVM_INFO @ 226942813.31 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] check 0x55
17 UVM_INFO @ 226942813.31 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT2_CTRL
18 param2 deposit : 11111111111111111111111111111111
19
20 UVM_INFO @ 234110115.75 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] Bit 0 of INT2_CTRL and INT2 asserted: 2nd check core writability
21 UVM_INFO @ 234110115.75 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] check 0xaa
22 UVM_INFO @ 234110115.75 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT2_CTRL
23 param2 deposit : 00000000000000000000000000000000
24
25 UVM_INFO @ 234122824.77 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] sleep asserted. Core execution terminated
26 UVM_INFO @ 234539115.00 ns: uvm_test_top.sve_top.scoreboard_inst [scoreboard_inst] No errors have
       been found during checking of the policy WRONLY inside test of the uc registers.
27 Test status: PASSED
28 UVM_INFO @ 234610400.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
       switch off
```

Figura 4.25: *Andamento della simulazione associata al test della policy WRONLY e messaggi UVM corrispondenti.*

I segnali non evidenziati sono utilizzati per permettere l'interazione tra il core e l'ambiente di verifica. Quando viene ceduto il controllo all'ambiente UVM, viene eseguito il *deposit* dei bit invertiti tramite le seguenti istruzioni SV:

```
    . . .
    int dim = $bits(vif.CURR_ALGO_ODR);
    bit [0:`max_dim_uc_reg-1] uc_reg_sig_n = {<<{~vif.CURR_ALGO_ODR}}; // save inverted bits
    choose_deposit_force_release_uc_reg("CURR_ALGO_ODR", uc_reg_sig_n, dim, 1'b1);
    . . .
// end task uc_regs_check

virtual function void choose_deposit_force_release_uc_reg(string signame, bit [0:`max_dim_uc_reg-1]
     sigval, int dim, bit flag_deposit);
    string path_sig = "tb_top.DUT:fullDigital_inst:top_digital_inst:uc_inst:uc_register_u:";
    string param1 = {path_sig, signame};
    string param2;

    for (int i = 0; i < dim; i++) begin
       param2 = {$sformatf("%b", sigval[i]), param2}; // convert into string each bit preserving
      trailing zeroes
    end
    if (flag_deposit == 1'b1) begin
       $xm_deposit(param1, param2);
    end else if (flag_deposit == 1'b0) begin
       $xm_force(param1, param2);
    end else if (flag_deposit == 1'bZ) begin
       //$xm_release(param1, "1");  // keep changes by setting 1
       $xm_release(param1, "0");
    end
    uvm_report_info(get_full_name(), $psprintf("param1 : %s\nparam2 : %s", param1, param2));
endfunction : choose_deposit_force_release_uc_regs
```

In generale, tutte le volte in cui viene richiesto all'ambiente di verifica di modificare i valori dei registri, si accede all'indirizzo specifico mediante *Virtual Interface* (indicata come `vif`) e la scelta di quale operazione utilizzare tra *force*, *deposit* o *release* viene effettuata invocando la funzione `choose_deposit_force_release()`. In questo modo il test risulta modulare e di facile riutilizzo.

Dopo che il valore del registro sotto test passa dal valore originale di `0x03` a `0xFC`, nessun errore viene rilevato, il core non riesce a leggere i nuovi bit e il risultato di lettura vale `0x00`, rispettando la policy di accesso. Questo risultato deriva dall'elaborazione della funzione di `check_unreadability()` introdotta al **Paragrafo 4.5.2**, in cui si verifica che le letture sono tutte nulle.

Infine, per quanto riguarda le scritture del core, si può osservare il corretto *toggling* di tutti i bit da `0x55` a `0xAA`.

La policy *WRONLY* risulta completamente verificata.

4.6 Test della policy SET

Negli ultimi paragrafi di questo capitolo vengono trattati i test sui *dual registers*. Si tratta di registri la cui funzionalità è quella di due *flip-flop* in cascata, con l'uscita dipendente da due ingressi possibili. Un ingresso è accessibile dall'interfaccia (IF_D), mentre l'altro è accessibile dal core (RL_D).

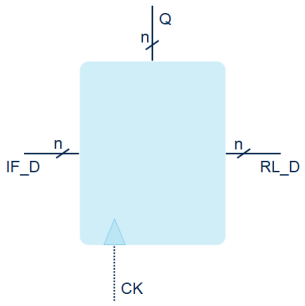


Figura 4.26: *Generica entità dual register.*

I registri di questo tipo sono presenti solamente all'interno del banco dei registri interni al microcontrollore, pertanto i test descritti al **Paragrafo 4.6** e **4.7** sono stati applicati unicamente su quel banco.

Esistono le due seguenti implementazioni dal punto di vista dell'accessibilità via core.

- Policy di SET: il core è solo in grado di settare i bit di uscita ma non di resettarli, mentre l'ambiente esterno è solo in grado di resettare i bit di uscita (vedasi **Paragrafo 4.6.2**).
- Policy di $CLEAR$: il core è solo in grado di "pulire" i bit di uscita ma non di impostarli, mentre l'interfaccia esterna può solo settare tali bit ma non resettarli (vedasi **Paragrafo 4.7**).

Per entrambe le configurazioni, sia lato core sia lato interfaccia, è possibile leggere il valore corrente dell'uscita. Pertanto, oltre a verificare lato core la corretta funzionalità delle policy di SET e $CLEAR$, occorre anzitutto verificare l'accessibilità in lettura a tali registri tramite core (vedasi **Paragrafo 4.6.1**).

4.6.1 Verifica di leggibilità

In questa parte preliminare si descrive il test creato per verificare la leggibilità dei *dual registers*. Il test in questione è applicato contemporaneamente sia ai registri con policy SET sia a quelli con policy $CLEAR$. Se il test fallisse, non avrebbe senso applicare i test successivi, afferenti al comportamento delle policy assegnate, siccome si basano sull'assunto che il core sia in grado di leggere l'uscita correttamente.

Sviluppo del test C

Per dimostrare la corretta funzionalità di accesso in lettura tramite core ai *dual registers*, occorre:

- accedere in lettura tramite core ai registri sotto test contenenti i valori di inizializzazione e salvare i risultati in un array;
- cedere il controllo all'ambiente di verifica, il quale inverte i bit associati alle policy sotto test (tramite scritture da *User Interface* o tramite operazioni di *deposit* da *Virtual Interface*);
- accedere in lettura ai registri tramite core e salvare le letture in un nuovo array;
- comparare le due letture per ciascun byte e riportare nei registri *General Purpose* gli indirizzi che falliscono in forma codificata e il numero totale di disallineamenti trovati.

La condizione da soddisfare, per ogni byte i, è la seguente:

```
v_bytes_before_wr[i] == (~v_bytes_after_wr[i])
```
(4.1)

Il flusso delle operazioni appena citate viene rappresentato in **Figura 4.27**.

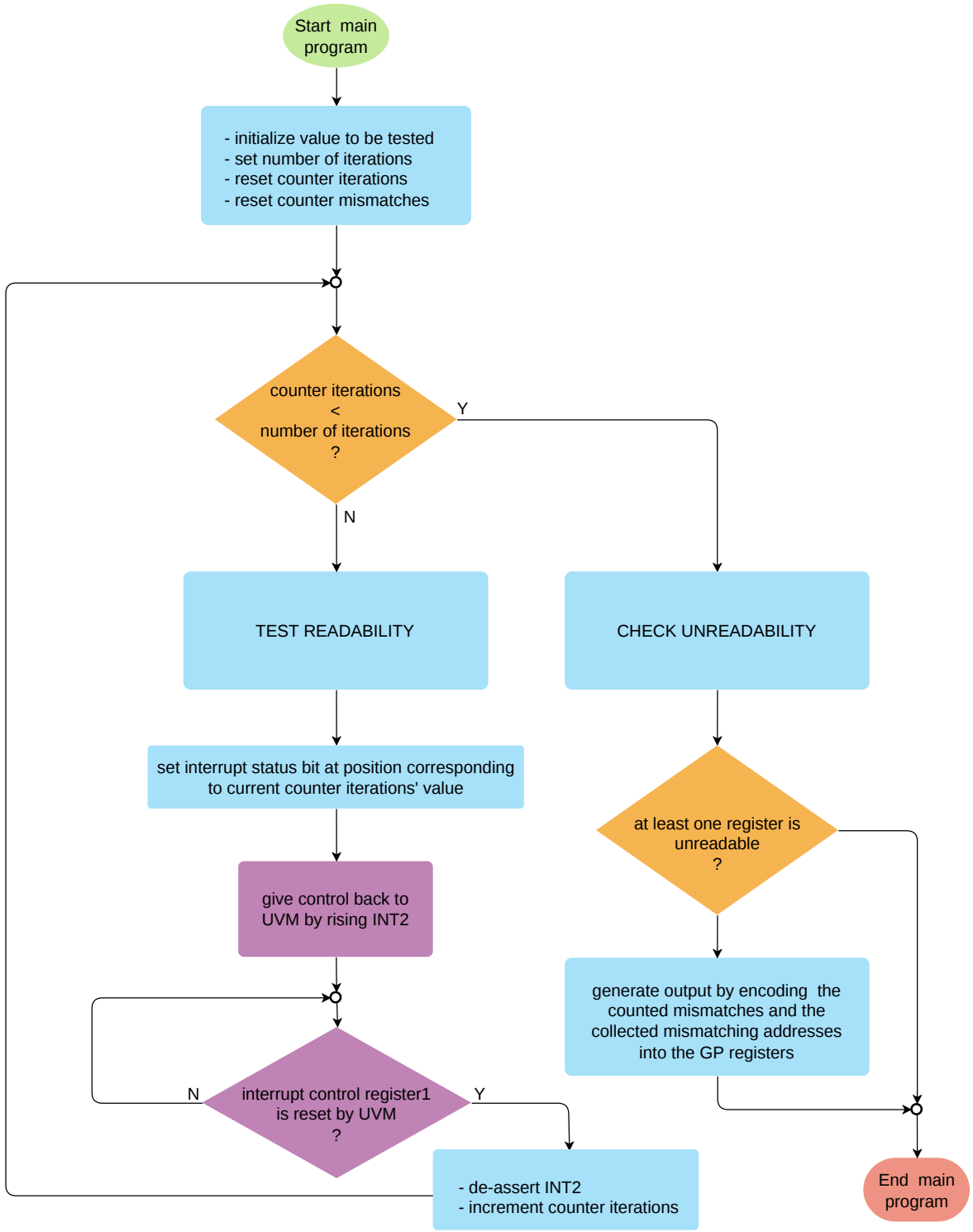


Figura 4.27: *Cuore del programma di self-checking per la verifica della leggibilità.*

Sviluppo del test UVM SystemVerilog

Configurazione dei parametri

Per simulare il test in questione occorre abilitare l'algoritmo 3 tramite `uc_algo_en[0] = 8'b0000_1000`.

Sequenza di test

Una volta avviata l'esecuzione del programma di *self-test* all'interno del core, quest'ultimo cede il controllo all'ambiente UVM in determinate circostanze affinché vengano pilotate le seguenti sequenze di comandi.

- Quando risulta abilitato il *bit*0 del registro di stato degli *interrupt* e viene generato il segnale di $INT2$ sul *pad* esterno, il testbench UVM inverte i bit presenti sulle uscite dei *dual registers* rispetto al valore di inizializzazione.

- Quando viene abilitato il bit1 del registro di stato degli *interrupt* e viene stimolata la linea di $INT2$, il testbench UVM esegue nuovamente una inversione bit a bit delle uscite dei registri sotto test. In questo modo si riportano i valori allo stato logico originario e nella fase finale della simulazione possono essere eseguite le scritture di spegnimento del dispositivo, le quali altrimenti non avverrebbero se alcuni bit di uscita su determinati *dual registers* rimanessero a '1'.

- Quando viene generato il segnale di *sleep* dal design, vengono eseguite le letture dei registri di uscita per decodificare i risultati.

Scoreboard

Lo scoreboard, come di consueto, decodifica gli indirizzi sulla base dei bit abilitati nei registri di uscita e decodifica in decimale il numero di disallineamenti trovati, stampando i risultati ottenuti.

Simulazioni e risultati ottenuti

All'interno del microcontrollore sono presenti i seguenti *dual registers*.

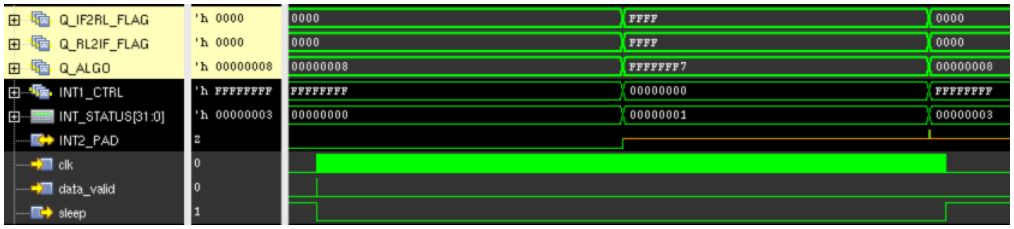
- Policy SET:

  · RL_RL2IF_FLAG, utilizzabile per un meccanismo di *handshaking* per sincronizzare microcontrollore e mondo esterno o in generale per passare dei *flag* di stato dal core all'interfaccia. Il core imposta i bit di uscita tramite la porta di ingresso a esso dedicata, mentre il mondo esterno legge i bit asseriti e li pulisce.

- Policy $CLEAR$:

  · RL_IF2RL_FLAG, il cui comportamento è esattamente duale rispetto al registro precedente;

· *RL_ALGO*, il quale permette di scrivere da interfaccia i bit la cui posizione è associata all'abilitazione delle configurazioni per eseguire gli algoritmi integrati nel programma e permette di "pulire" tali bit tramite core.




```
1  . . .
2  UVM_INFO @ 1495061158.15 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] Bit 0 of INT_STATUS and INT2 asserted. Perform deposit on SET and
       CLEAR regs
3  UVM_INFO @ 1495061158.15 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:Q_RL2IF_FLAG
4  param2_xm : 1111111111111111
5  UVM_INFO @ 1495061158.15 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:Q_IF2RL_FLAG
6  param2 deposit : 1111111111111111
7  UVM_INFO @ 1495061158.15 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:Q_ALGO
8  param2 deposit : 11111111111111111111111111110111
9  UVM_INFO @ 1495061158.15 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT1_CTRL
10 param2 deposit : 00000000000000000000000000000000
11
12 UVM_INFO @ 1510465249.99 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] Bit 1 of INT_STATUS and INT2 asserted. Perform deposit on SET and
       CLEAR regs
13 UVM_INFO @ 1510465249.99 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:Q_RL2IF_FLAG
14 param2 deposit : 0000000000000000
15 UVM_INFO @ 1510465249.99 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:Q_IF2RL_FLAG
16 param2 deposit : 0000000000000000
17 UVM_INFO @ 1510465249.99 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:Q_ALGO
18 param2 deposit : 00000000000000000000000000001000
19 UVM_INFO @ 1510465249.99 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] param1 deposit : tb_top.DUT:fullDigital_inst:top_digital_inst:
       uc_inst:uc_register_u:INT1_CTRL
20 param2 deposit : 11111111111111111111111111111111
21
22 UVM_INFO @ 1511299761.04 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top
       .mss.seq_basic.seq_chain_inst] sleep asserted. Core execution terminated
23 UVM_INFO @ 1513172375.00 ns: uvm_test_top.sve_top.scoreboard_inst [scoreboard_inst] No errors have
       been found during checking of the readability of the uc registers with policy SET or CLR
24
25 UVM_INFO @ 1514556441.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
       switch off
```

Figura 4.28: *Risultato del test di accessibilità in lettura sui dual registers presenti nel design.*

A seguito delle inversioni eseguite sulle uscite, si è osservato che il core ha effettuato correttamente le letture di tali valori, dal momento che la condizione [4.1] ha sempre dato esito positivo e non sono stati riportati bit codificanti indirizzi errati nei registri GP di uscita. Pertanto, tutti i *dual registers* presenti nel design sono accessibili in lettura dal core.

4.6.2 Verifica di sola settabilità dal core

Sviluppo del test C

Il test in questione si occupa di verificare che i registri aventi policy SET siano solo settabili dal core e non resettabili. Le operazioni dettagliate di seguito sono rappresentate nel diagramma di flusso in **Figura 4.29**.

- Si cede il controllo all'ambiente UVM affinché tutti i bit vengano portati a uno stato noto, ovvero vengano messi tutti a zero. Il core rimane in *polling* sul registro $INT1\_CTRL$ sino a che questo registro non viene resettato.
- Il core esegue un ciclo di *self-test* in cui, per ogni registro avente policy SET:
  - vengono scritti tutti i bit a '1';
  - si tenta di riportare tutti i bit a '0';
  - si leggono i bit e si verifica che questi siano rimasti al livello logico '1', validando al tempo stesso sia la settabilità di tutti i bit, sia il fatto che non possano essere resettati dal core.
- Si salvano eventuali fallimenti sui registri di uscita, codificando indirizzi e conteggio dei byte disallineati rispetto ai valori attesi e il programma termina.

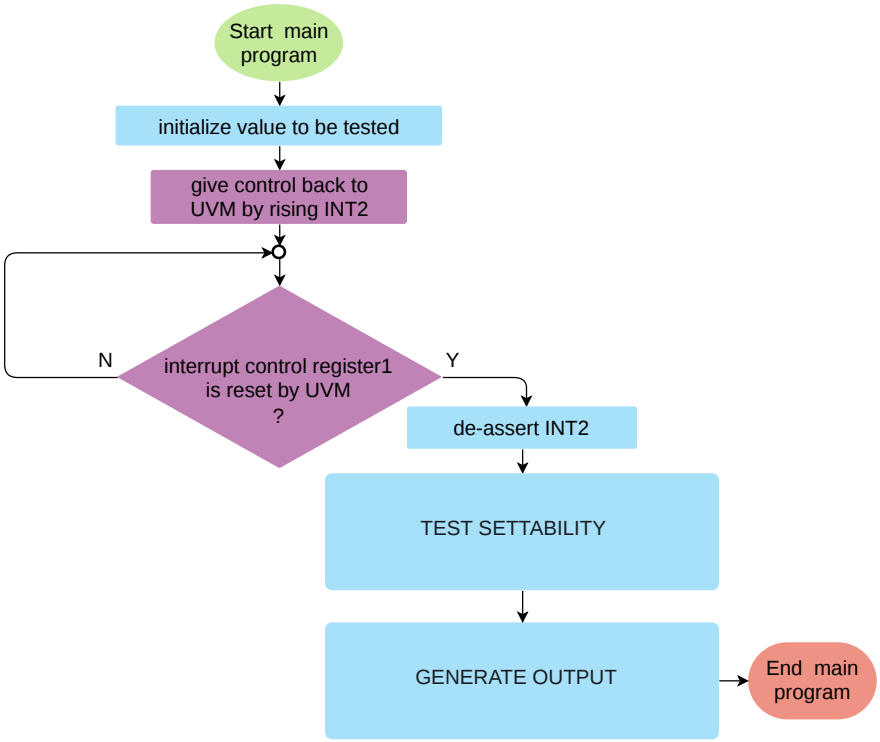


Figura 4.29: *Cuore del programma di self-test eseguito dal core.*

Sviluppo del test UVM SystemVerilog

Configurazione dei parametri

Il test della policy *SET* è implementato all'algoritmo 4 e viene abilitato tramite `uc_algo_en[0] = 8'b0001_0000`.

Sequenza di test

Le operazioni eseguite via testbench UVM consistono in due sequenze.

- Se il core stimola la linea di *interrupt* $INT2$, si esegue un $CLEAR$ iniziale dei registri sotto test, in modo da garantire che il SET di tutti i bit possa avvenire tramite *self-test* via core e che eventuali bit inizializzati a '1' vengano portati a '0'.

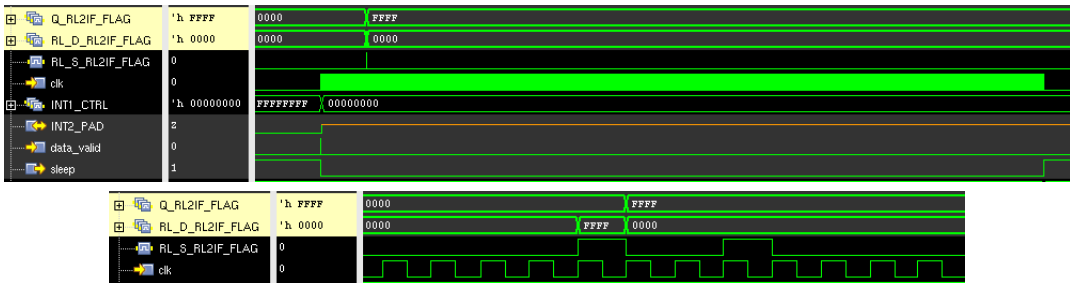
- Se il core asserisce il segnale di *sleep*, si effettuano le consuete letture da interfaccia sui registri di uscita.

Scoreboard

L'implementazione è analoga al caso precedente, con la differenza che i messaggi UVM che vengono generati sono riferiti alla policy di SET.

Simulazioni e risultati ottenuti

Come si può osservare dalla **Figura 4.30**, il registro RL_RL2IF_FLAG viene settato correttamente dal core.




```
. . .
UVM_INFO @ 189880753.72 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
    mss.seq_basic.seq_chain_inst] INT2 asserted. Perform deposit on registers having policy SET from
    core side
UVM_INFO @ 189880753.72 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
    mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:uc_inst:
    uc_register_u:Q_RL2IF_FLAG
param2 : 0000000000000000
UVM_INFO @ 189880753.72 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
    mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:uc_inst:
    uc_register_u:INT1_CTRL
param2 : 00000000000000000000000000000000
UVM_INFO @ 198870075.35 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
    mss.seq_basic.seq_chain_inst] sleep asserted. Core execution terminated
UVM_INFO @ 199145566.00 ns: uvm_test_top.sve_top.scoreboard_inst [scoreboard_inst] No errors have
    been found during checking of the uc registers with policy SET
UVM_INFO @ 199357651.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
    switch off
```

Figura 4.30: *Risultati di simulazione che mostrano la corretta verifica della funzionalità della policy di SET.*

Osservando lo *zoom* di dettaglio, quando il core accede alla porta di ingresso $RL\_D\_RL2IF\_FLAG$, viene automaticamente asserito il bit di *enable* associato alla porta, ossia $RL\_S\_RL2IF\_FLAG$.

- Il bit di abilitazione rimane attivo per un periodo di clock, nel quale viene passato alla porta di ingresso l'insieme di bit che si desidera riportare all'uscita del registro.
- Quando viene passato il valore `0xFFFF`, questo viene correttamente riportato sull'uscita nel periodo di clock successivo.
- Quando alla porta di ingresso si passa il valore `0x0000`, corrispondente al secondo periodo in cui il bit $RL\_S\_RL2IF\_FLAG$ vale '1', l'uscita non viene riportata a zero ma tutti i bit rimangono al livello logico '1'.

La funzionalità di SET è completamente verificata.

4.7 Test della policy CLEAR

4.7.1 Sviluppo del test C

Il flusso di operazioni di test è rappresentato in **Figura 4.31** e si articola nel modo seguente.

- Il core cede il controllo all'ambiente UVM per portare tutti i bit al valore noto '0'.
- Il core riprende il controllo e testa la policy (blocco *TEST CLEARABILITY*). In questa prima iterazione, per ogni registro le operazioni consistono in:
  - lettura dei byte e salvataggio dei valori acquisiti;
  - tentativo di *SET* di tutti i bit;
  - rilettura dei byte sotto test e salvataggio;
  - comparazione delle due letture.
- Se non ci sono disallineamenti tra le letture e il tentativo di *SET* non è andato a buon fine, significa che non ci sono errori funzionali sino a questo punto e si può procedere con la verifica della capacità di *CLEAR* tramite core.
- Il core cede il controllo all'ambiente UVM, per cui viene eseguito un *SET* per ogni registro sotto test, questa volta effettivo.
- Il core riprende il controllo e riesegue la routine di *self-test* precedente, ma in questa nuova iterazione le operazioni ricadono su un *case* differente, per cui per ogni registro sotto test si ha:
  - lettura e salvataggio dei byte acquisiti in un array;
  - esecuzione del *CLEAR* di tutti i bit;
  - rilettura e salvataggio degli elementi in un nuovo array;
  - comparazione elemento per elemento dei due array di salvataggio.
- Prima di terminare il programma, vengono salvati gli indirizzi dei registri che presentano dei fallimenti, codificando la loro posizione all'interno dei registri di uscita a partire dagli elementi contenuti in un array, il quale è stato riempito a seguito delle comparazioni.

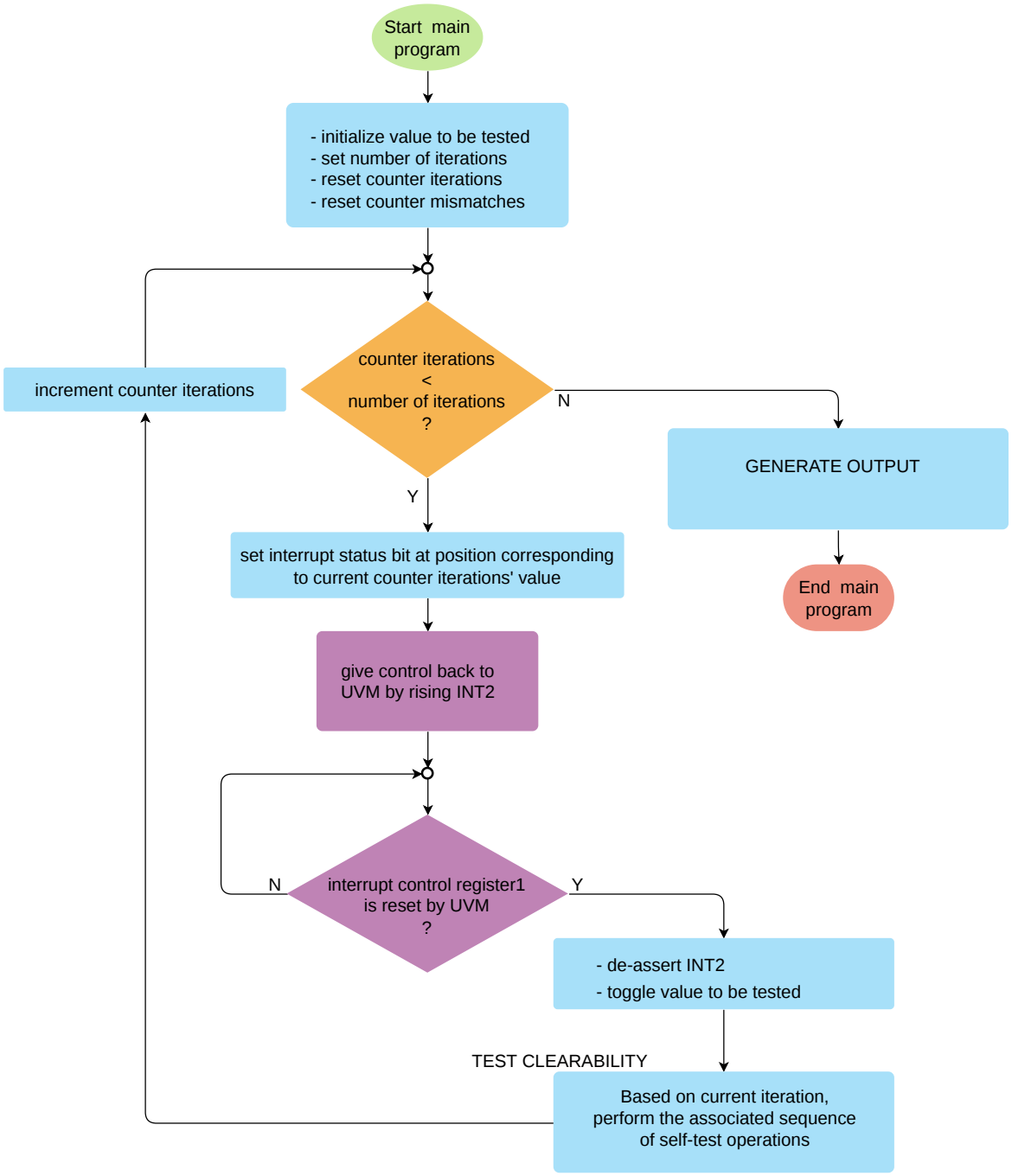


Figura 4.31: *Programma di self-test che agisce sui registri con policy CLEAR.*

4.7.2 Sviluppo del test UVM SystemVerilog

Configurazione dei parametri

Il test della policy $CLEAR$ è implementato all'algoritmo 4 e viene abilitato tramite `uc_algo_en[0] = 8'b0001_0000`.

Sequenza di test

Il comportamento della sequenza consiste nell'esecuzione di tre possibili blocchi di comandi.

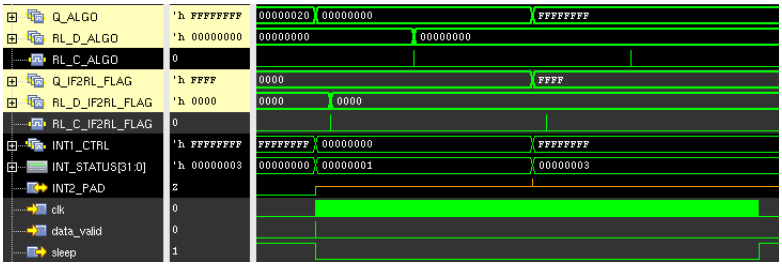
- Quando dal core viene asserito il *bit*0 del registro di stato e viene mandato un segnale di *interrupt* su $INT2$, tutti i registri sotto test vengono inizializzati a zero. Il core esce dal ciclo di attesa non appena $INT1\_CTRL$ viene resettato dal testbench UVM.
- Quando dal core viene asserito il bit1 del registro di stato e viene mandato un segnale di *interrupt* su $INT2$, avviene il SET effettivo delle uscite dei *dual registers* sotto test.
- Quando dal core viene asserito il segnale di *sleep*, sono eseguite le letture per la raccolta dei risultati, che possono essere decodificati dallo scoreboard.

Scoreboard

L'implementazione è analoga al caso precedente, con la differenza che i messaggi UVM che vengono generati sono ora riferiti alla policy di $CLEAR$.

4.7.3 Simulazioni e risultati ottenuti

Differentemente dai *dual registers* aventi policy SET, in questo caso tutti i registri con policy $CLEAR$ non rispettano il comportamento dichiarato nel file dei requisiti.
Come si può osservare dalla **Figura 4.32**, quando tutti i bit delle uscite sono stati impostati a '1' tramite ambiente UVM, la scrittura di tutti i bit a '0' tramite core non sortisce alcun effetto sulle uscite.
Tuttavia, va menzionato il fatto che una parte della funzionalità della policy è verificata con successo: nella prima parte del test, quando il core tenta di operare dei SET nei registri in questione, ossia in corrispondenza dei bit di *enable* a '1' indicati come RL_C_ALGO e $RL\_C\_IF2RL\_FLAG$, sulle uscite Q_ALGO e Q_IF2RL_FLAG nessun bit cambia stato, come ci si attende debba avvenire per la policy di solo $CLEAR$.




```
1  . . .
2  UVM_INFO @ 214918137.06 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:uc_inst:
       uc_register_u:INT1_CTRL
3  param2 : 00000000000000000000000000000000
4  UVM_INFO @ 223260330.53 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:uc_inst:
       uc_register_u:Q_IF2RL_FLAG
5  param2 : 1111111111111111
6  UVM_INFO @ 223260330.53 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:uc_inst:
       uc_register_u:Q_ALGO
7  param2 : 11111111111111111111111111111111
8  UVM_INFO @ 223260330.53 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] param1 : tb_top.DUT:fullDigital_inst:top_digital_inst:uc_inst:
       uc_register_u:INT1_CTRL
9  param2 : 11111111111111111111111111111111
10 UVM_INFO @ 231982664.11 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [uvm_test_top.sve_top.
       mss.seq_basic.seq_chain_inst] sleep asserted. Core execution terminated
11 UVM_ERROR @ 232018798.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0x0c do not respect
       assigned policy requirement
12 UVM_ERROR @ 232018798.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0x0d do not respect
       assigned policy requirement
13 UVM_ERROR @ 232110318.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0x70 do not respect
       assigned policy requirement
14 UVM_ERROR @ 232110318.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0x71 do not respect
       assigned policy requirement
15 UVM_ERROR @ 232110318.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0x72 do not respect
       assigned policy requirement
16 UVM_ERROR @ 232110318.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] One or more bits inside Byte at address START_uc_regs + 0x73 do not respect
       assigned policy requirement
17 UVM_ERROR @ 232258158.00 ns: uvm_test_top.sve_top.scoreboard_inst [uvm_test_top.sve_top.
       scoreboard_inst] Policy access violations of registers of kind CLR have been found during uc
       registers' test.
18 The overall number of Bytes of the uc registers which does not respect the policy CLR is          6
19 UVM_INFO @ 232470243.00 ns: uvm_test_top.sve_top.mss@@seq_basic.seq_chain_inst [seq_chain_inst]
       switch off
```

Figura 4.32: *Risultato di simulazione per la policy CLEAR. L'operazione di reset tramite core non ha effetto su nessun registro.*

Concludendo, sotto tali condizioni, l'operazione di *CLEAR* dei bit tramite core non è effettiva e viene segnalata tramite dei messaggi di errore dal testbench UVM. Non essendo riportate ulteriori informazioni dettagliate nel manuale tecnico di riferimento, occorre analizzare la problematica riferendosi all'implementazione del design RTL, per comprendere se all'interno dei *dual registers* in questione è presente un baco, oppure se la specifica non è stata intesa correttamente rispetto al comportamento effettivo del design. La parte di debug e dell'eventuale modifica dell'implementazione di questa porzione di design è stata demandata al Designer, dal momento che l'obiettivo della verifica digitale consiste, in questo caso, nella verifica del corretto interfacciamento delle entità *dual registers* (viste come *black boxes*) e l'IP core.

Conclusioni

Il lavoro di tesi si è focalizzato sull'implementazione di test funzionali in grado di mettere a frutto la capacità di verifica con auto-test di un microcontrollore integrato, grazie allo sviluppo di firmware di test C e all'utilizzo di un ambiente di verifica UVM.

I test di tipo *self-checking* che sono stati ideati durante l'attività di tesi si sono dimostrati efficaci per la verifica funzionale degli elementi di memoria che si interfacciano con l'IP core.

Infatti, grazie all'applicazione di questi test è stato possibile rilevare delle discrepanze relative al comportamento di alcuni registri del design rispetto al file dei requisiti sulle policy di accesso tramite core. Questo ha permesso di migliorare le specifiche ove queste fossero incomplete o errate, aumentandone il livello di dettaglio e la robustezza.

Inoltre, l'attività ha consentito di maturare preziose competenze nell'utilizzo dei seguenti strumenti.

- La creazione di script Python con l'intento di effettuare il *parsing* di un file di ingresso con le specifiche sulle policy di accesso avente una formattazione tabulare, sfruttando liste, dizionari e conversioni di tipo di dato da stringhe a valori esadecimali o interi. Le informazioni acquisite dalle operazioni di *parsing* sono utilizzate per generare in uscita un codice sorgente C contenente le informazioni di interesse codificate sotto forma di strutture dati C accessibili dal firmware di test in modo iterativo tramite il meccanismo dei puntatori, massimizzando la flessibilità.

- Lo studio e la manipolazione dell'ambiente UVM SystemVerilog, sfruttando il paradigma della programmazione a oggetti, la sincronizzazione di più task in parallelo e l'utilizzo della *User Interface* I2C/SPI e della *Virtual Interface* mediante le istruzioni di *force* o *deposit*. La gestione in ambiente UVM ha reso necessaria l'implementazione di nuove classi per ogni test, in grado di gestire i parametri di configurazione, temporizzare le sequenze di comandi concepiti per interagire con il DUT e costruire i metodi di acquisizione e decodifica dei risultati prodotti dalle routine di test C eseguite all'interno del microcontrollore.

- La creazione di firmware C di auto-test, costituito da istruzioni a basso livello direttamente interagenti con le risorse hardware del microcontrollore sotto test. Questo ha consentito di acquisire maggiore manualità e consapevolezza nel reperimento e nel

confronto delle informazioni di interesse sul microcontrollore nella documentazione tecnica e nell'utilizzo del microcontrollore a scopo di *self-test*.

Un aspetto importante che ha caratterizzato il lavoro di tesi è stato l'implementazione di test ad alta modularità e in grado di soddisfare la portabilità, sia verso il pre-Silicio sia verso il post-Silicio:

- le routine di test C sono adattabili per altri dispositivi in cui è integrato un microcontrollore;
- le routine di test C sono riutilizzabili anche sul chip in Silicio nella fase di *Final Test*, grazie alla loro capacità di rilevare le difettosità fisiche degli elementi di memoria mediante operazioni di accesso in lettura e scrittura eseguite attraverso il core. Tutte le combinazioni di cambiamento di stato logico tra bit adiacenti vengono verificate servendosi di varie combinazioni di *pattern* di test.

Grazie allo script Python, le informazioni su policy e indirizzi sono passate al firmware di test sempre con il medesimo formato, se varia il formato del file di specifiche in ingresso, basta riadattare lo script di *parsing* senza la necessità di modificare il test C. Questo permette di automatizzare il flusso di test in quanto l'interfacciamento tra le istruzioni delle routine C e le strutture dati in ingresso rimane invariato.

Un'altra caratteristica rilevante, intrinseca alla tipologia di test, consiste nella riduzione del tempo di esecuzione rispetto ad un approccio nel quale i *pattern* di test vengano inseriti tramite interfaccia seriale. Il core, sulla base del parallelismo delle *word*, è in grado di operare in parallelo su più bit, mentre l'accesso da interfaccia seriale, a parità di frequenza di funzionamento, sarebbe vincolato dalla trasmissione di 1 bit/ck.

Infine, occorre citare anche l'aspetto correlato all'ottimizzazione del codice. Il codice che viene compilato e caricato nella memoria di programma ha un impatto limitato dal punto di vista dell'occupazione di memoria. In riferimento ai test dei banchi di registri, grazie ad uno script Bash con il quale è possibile decidere da *shell* quali algoritmi compilare, il codice caricato contiene solo le routine di test strettamente necessarie alla verifica della policy di accesso che si desidera testare. Tuttavia, se si desidera caricare l'eseguibile associato alla verifica funzionale di tutte le policy di accesso, si ha che la massima occupazione della memoria codice vale $\sim 9\,\text{kB}$.
Di conseguenza, il codice prodotto può essere utilizzato su qualsiasi microcontrollore che abbia una memoria codice di dimensione almeno pari o superiore a tale valore.

In conclusione, il lavoro compiuto in questa attività di tesi ha contribuito a portare all'interno del Team di Verifica Digitale una metodologia di test innovativa che prima non esisteva, grazie alla quale la funzionalità degli elementi di memoria del design viene verificata dal core stesso tramite *self-checking*. Ciò ha consentito di creare un flusso di verifica auto-consistente dal design RTL sino al chip fisico su Silicio, permettendo di ridurre i tempi di test del programma di produzione nel *Final Test* e di rendere più robuste le specifiche del design da fornire al cliente.

Appendice A

Protocolli di comunicazione I2C e SPI

A.1 Protocollo I2C

Il protocollo I2C (*Inter-Integrated Circuit*) è uno standard di comunicazione *de facto*, che venne inventato da *Philips* (oggi nota come *NXP Semiconductors*) ed è basato sulla sincronizzazione e trasmissione dell'informazione mediante l'ausilio di un bus a due linee, identificate come *Serial Data* (SDA) e *Serial Clock* (SCL) [6].
Queste linee possono diramarsi all'interno del design per interfacciare i diversi blocchi che ne richiedono l'impiego, i quali sono raggruppabili in due categorie.

- Il *Master*[1] è l'entità che genera il segnale di temporizzazione e inizia il trasferimento di informazioni.
- Lo *Slave*[1] è il dispositivo che riceve il segnale di clock dal *master* e con il quale scambia informazioni. Possono esistere più *slave* controllati dallo stesso *master*.

Ciascun blocco può comportarsi sia da trasmettitore sia da ricevitore e si possono avere le seguenti velocità di comunicazione: fino a 100 kbps in *standard mode*, da 100 kbps a 400 kbps in *fast mode*, da 400 kbps a 3.4 Mbps in *high-speed mode* e fino a 5 Mbps in *ultra-fast mode*.

Il trasferimento dei dati avviene in modo seriale e bidirezionale; quando nessun dispositivo trasmette, le linee sono in alta impedenza. I dati trasferiti sono raggruppati su 8 bit e una comunicazione I2C deve sempre iniziare con la *start condition* e terminare con la *stop condition*, entrambe generate dal *master*. Le sequenze in oggetto sono rappresentate in **Figura A.1**.

[1]Questa nomenclatura è desueta ed è stata sostituita dai termini *controller*/*targets*. Tuttavia in questa tesi è stata applicata la vecchia nomenclatura per allineare le descrizioni alla terminologia già presente nel design oggetto di studio.

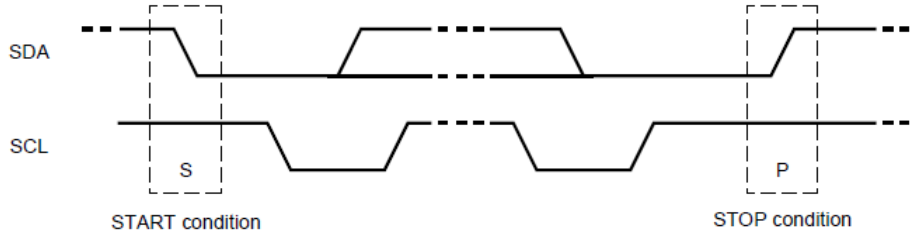


Figura A.1: *Sequenze I2C di START e di STOP [6].*

- *Start condition*: SDA effettua la transizione '1' → '0' mentre SCL rimane a '1'. Il bus viene occupato per la trasmissione dei dati.
- *Stop condition*: SDA effettua la transizione '0' → '1' mentre SCL rimane a '1'. Il bus torna libero dopo il verificarsi di tale condizione.

I byte trasmessi sulla linea SDA devono essere intervallati da un bit di *acknowledge* ACK inviato dal ricevitore per segnalare al trasmettitore che la ricezione è avvenuta correttamente. In questo modo può essere trasmesso un nuovo byte.
In **Figura A.2** è possibile apprezzare un generico trasferimento di dati tramite protocollo I2C. Dopo l'invio della *start condition*, il *master* invia un byte composto da 7 bit di *slave address* e un bit che indica se il dato a quell'indirizzo deve essere scritto (bit a '0') oppure letto (bit a '1'). Il bit successivo di ACK indica la risposta dello *slave*, grazie alla quale si può procedere con il trasferimento del byte di dato.

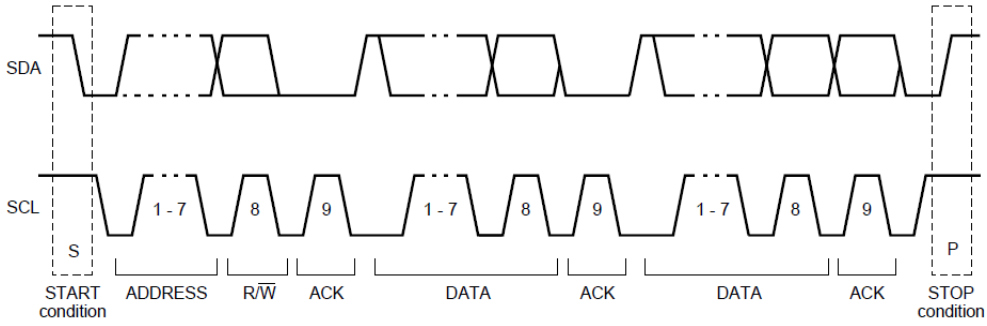


Figura A.2: *Esempio di trasferimento dati I2C [6].*

Operazioni di scrittura
Le operazioni che seguono sono riferite alla scrittura di un singolo byte. Il *timing* di riferimento è rappresentato in **Figura A.3**.

· *start condition*
· *slave address* su 7 bit
· '0' a indicare operazione di scrittura
· ACK dello *slave* a seguito del riconoscimento dell'indirizzo
· *register address*
· ACK dello *slave* a seguito del riconoscimento dell'indirizzo

· scrittura byte di dato nello *slave*
· ACK dello *slave* a seguito della corretta ricezione del dato
· *stop condition*

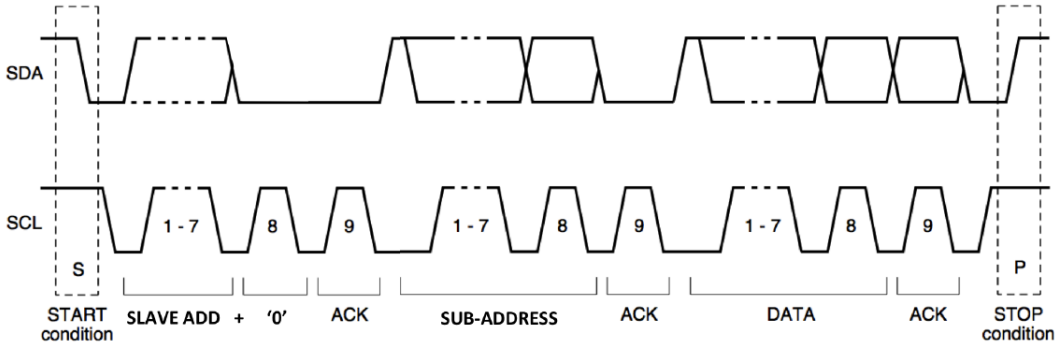


Figura A.3: *Scrittura tramite protocollo I2C [1, 2].*

Operazioni di lettura

Le operazioni che seguono sono riferite alla lettura di un singolo byte. Il *timing* di riferimento è rappresentato in **Figura A.4**.

· *start condition*
· *slave address* su 7 bit
· '0' a indicare operazione di scrittura
· ACK dello *slave* a seguito del riconoscimento dell'indirizzo
· *register address*
· ACK dello *slave* a seguito del riconoscimento dell'indirizzo
· *start condition* (RE-START)
· *slave address* su 7 bit
· '1' a indicare operazione di lettura
· ACK dello *slave* a seguito del riconoscimento dell'indirizzo
· lettura byte di dato dallo *slave*
· NACK del *master* a seguito della corretta ricezione del dato
· *stop condition*

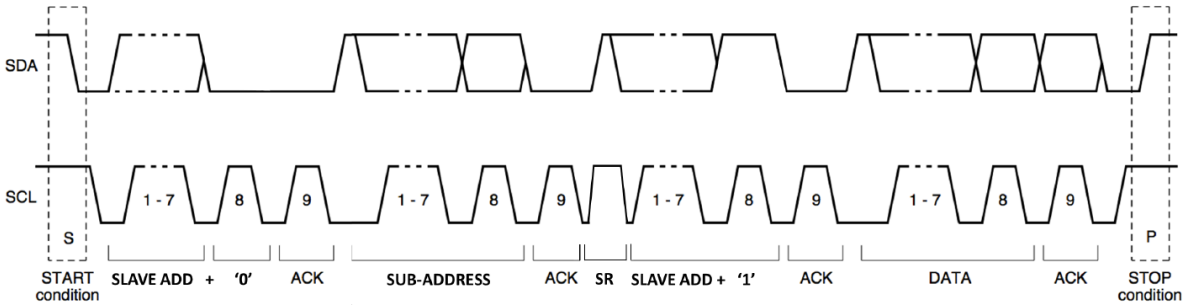


Figura A.4: *Lettura tramite protocollo I2C [1, 2].*

A.2 Protocollo SPI

Il protocollo SPI (*Serial Peripheral Interface*) è una interfaccia di comunicazione seriale sincrona, che venne sviluppata da *Motorola* e si basa sull'utilizzo di quattro linee.

- SPE (*Serial Port Enable*) indica l'avvio della comunicazione se compie una transizione '1' → '0'; compie una transizione '0' → '1' quando la trasmissione termina.

- SPC (*Serial Port Clock*) è il segnale di temporizzazione del protocollo. Esso inizia a scandire i colpi di clock dopo che SPE segnala l'inizio della comunicazione.

- SPDI (*Serial Port Data Input*) è la linea per i dati in ingresso.

- SPDO (*Serial Port Data Output*) è la linea per i dati in uscita.

La trasmissione può avvenire su quattro fili se entrambe le linee di SPDI e SPDO vengono utilizzate, altrimenti su tre fili nel caso in cui la linea SPDO rimanga interdetta e sia la trasmissione sia la ricezione dei dati avvengano sulla linea SPDI.
Esistono quattro differenti modalità di funzionamento in base alla polarità (CPOL) e alla fase (CPHA) del segnale di temporizzazione rispetto ai dati trasmessi.

CPOL	CPHA	
0	0	il clock è in IDLE al livello logico '0' e i dati sono sincronizzati sul fronte di salita del clock
0	1	il clock è in IDLE al livello logico '0' e i dati sono sincronizzati sul fronte di discesa del clock
1	0	il clock è in IDLE al livello logico '1' e i dati sono sincronizzati sul fronte di salita del clock
1	1	il clock è in IDLE al livello logico '1' e i dati sono sincronizzati sul fronte di discesa del clock

In **Figura A.5** si riporta un esempio di trasmissione con protocollo SPI in cui CPOL = 1 e CPHA = 1.

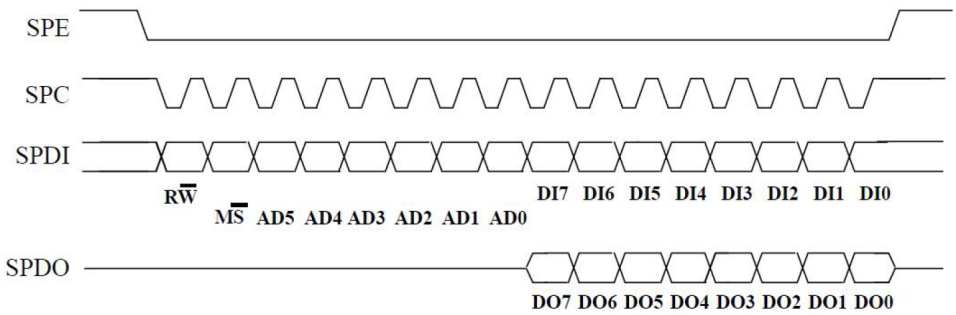


Figura A.5: *Esempio di lettura e scrittura con protocollo SPI [1, 2].*

Le operazioni sono eseguite in sedici periodi del segnale di clock, oppure in multipli di otto nel caso di scritture e letture multiple. Due transizioni '1' $\rightarrow$ '0' della linea SPC definiscono la durata di un bit. I bit trasmessi sono i seguenti:

- il bit R/W indica il tipo di operazione, se vale '0' allora il dato DI[7:0] viene scritto al registro con indirizzo AD[5:0], se vale '1' allora viene letto il dato DO[7:0] all'indirizzo assegnato;
- il bit M/S indica se l'indirizzo dell'operazione successiva a quella corrente rimane invariato (bit a '0'), oppure se l'indirizzo deve essere incrementato (bit a '1');
- AD[5:0] sono i bit che contengono l'indirizzo del registro a cui accedere;
- DI[7:0] sono i bit che contengono il dato su 8 bit da scrivere nel caso di configurazione a quattro fili, altrimenti contengono il dato da leggere o scrivere nel caso di configurazione a tre fili;
- DO[7:0] sono i bit che contengono il dato su 8 bit da leggere nel caso di configurazione a quattro fili.

Appendice B

Fasi di un test UVM

Quando un test UVM viene eseguito, la gestione delle fasi avviene mediante le cosiddette *UVM phases* [7], illustrate in **Figura B.1**.

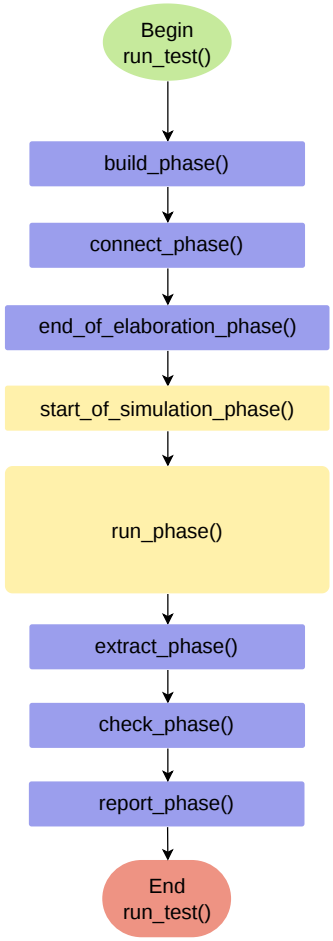


Figura B.1: *Fasi di un generico test UVM.*

Il flusso delle fasi di test viene fatto iniziare all'interno del modulo a più alto livello che costituisce il testbench, anche detto `tb_top`, grazie all'invocazione del metodo di `run_test()`.

- La `build_phase()` permette di costruire i vari componenti e di organizzarli in modo gerarchico. Essa si avvale delle funzionalità messe a disposizione dalla *UVM Factory*, ossia un meccanismo in grado di creare oggetti e componenti configurando dinamicamente la gerarchia e di sostituirne alcuni con altri già registrati senza rompere l'incapsulamento. In questo modo si migliora la flessibilità e la riusabilità dei componenti.

- La `connect_phase()` effettua la connessione dei sotto-componenti creati attraverso connessioni *transaction level*. Ad esempio, i sequencer degli UVC vengono connessi ai rispettivi driver, etc.

- La `end_of_elaboration_phase()` assicura che tutte le connessioni siano state stabilite e che tutte le porte siano accessibili.

- La `start_of_simulation_phase()` inizializza i componenti affinché possano essere pronti a eseguire le operazioni *time-consuming* della fase di simulazione.

- All'interno della `run_phase()` vengono eseguite in parallelo un insieme di *UVM run-time phases* composte da task, funzioni oppure sequenze che vengono eseguite automaticamente rispettando la temporizzazione delle attività da eseguire nei vari stadi di simulazione. Le fasi associate alla `run_phase()` costituiscono il tempo di simulazione del test.

- Al termine della simulazione, la `extract_phase()` estrae i risultati e alcune statistiche di simulazione.

- La `check_phase()` si occupa di validare i dati che sono stati estratti e determina l'esito finale della simulazione.

- La `report_phase()` riporta i risultati finali nel file di *log* e a schermo.

Appendice C

Script Python

C.1 Script di *parsing* per il test dei *Microcontroller Registers*

Di seguito si riporta lo script Python costruito per effettuare il *parsing* del file di specifiche dei registri interni al microcontrollore. Il parametro `MAX_DIM` permette di definire la massima dimensione degli array di salvataggio delle codifiche, quindi in quanti array di stringhe il banco di registri verrà mappato. Prima dell'esecuzione delle operazioni di parsing, il parametro viene controllato affinché rispetti dei valori ragionevoli.

```
#!/usr/bin/python
'''
ASSUMPTIONS:
input file converted from .xml to .txt in the format TAB DELIMITER generated by Excel conversion.
ENCODING values:
 - accessibility:
       the two MSBs (hex values), each bit corresponds to the position of the bit in the reg; whether it is accessible is '1', '0' otherwise
 - read
   - 'as zero' == 'A'
   - 'Yes'     == 'F'
 - write
   - 'ignored'    == 'A'
   - 'Yes'        == 'F'
   - 'clear only' == 'B'
   - 'set only'   == 'C'
'''
import sys
import re
import math

filein = 'UC.txt'
fileout1 = 'uc_reg_info.h'
fileout2 = 'uc_reg_info.c'
fileout3 = 'uc_reg_info.txt'
start_addr = 0x000
end_addr = 0x200
TOT_BYTES = end_addr - start_addr
TOT_WORDS = TOT_BYTES/4
MAX_DIM = 128
DIM_STR = 9
```

```
31
32 def parse_regmap(filein):
33     line = ''
34     linedec = ''
35     l_fields = []
36     field = ''
37     ADDRESS = ''
38     RL_READABLE = ''
39     RL_WRITABLE = ''
40     pos_ADDRESS = 1
41     pos_RL_READABLE = 17
42     pos_RL_WRITABLE = 18
43     pos_end_bit = 3
44     pos_start_bit = 10
45     bit7 = 10
46     rst_mask = 0x0000FFFFF
47     accessibility_mask = 0x1000000000 #1 as MSB to be able to keep leading zero(s)
48     str_policy = '00'
49     base = 16
50     previous_addr = start_addr
51     cnt_addr = 0
52     l_policy_and_addr = []
53     l_addresses_sv = []
54     l_holes_c = []
55     l_holes_sv = []
56
57     try:
58         fid = open(filein, 'rb')
59     except:
60         print('ERROR: Unable to open ' + filein)
61     else:
62         print('Processing ' + filein + ' . . .')
63         line = fid.readline()
64         linedec = line.decode('utf-8')
65         while linedec != '':
66             l_fields = linedec.split('\t')
67             ADDRESS = l_fields[pos_ADDRESS]
68             if re.match('0x*', ADDRESS):
69                 cnt_addr += 1
70                 int_addr = int(ADDRESS, base)
71                 hex_addr = hex(int_addr)[2:]
72                 if int_addr > (previous_addr + 1):
73                     [l_holes_c, l_holes_sv] = fill_holes([previous_addr, int_addr])
74                 previous_addr = int_addr
75                 int_addr = int(hex_addr, base)
76                 hex_policy_and_addr = hex(int_addr)
77                 RL_READABLE = l_fields[pos_RL_READABLE]
78                 RL_WRITABLE = l_fields[pos_RL_WRITABLE]
79                 hex_policy_and_addr = hex(int(hex_policy_and_addr, base) & rst_mask)
80                 #encoding policy
81                 if RL_READABLE == 'Yes':
82                     hex_policy_and_addr = hex(int(hex_policy_and_addr, base) | 0
    x00F000000)
83                     str_policy = 'F' + str_policy[1:]
84                 elif RL_READABLE == 'as zero':
85                     hex_policy_and_addr = hex(int(hex_policy_and_addr, base) | 0
    x00A000000)
86                     str_policy = 'A' + str_policy[1:]
87
88                 if RL_WRITABLE == 'Yes':
89                     hex_policy_and_addr = hex(int(hex_policy_and_addr, base) | 0
    x000F00000)
90                     str_policy = str_policy[:1] + 'F'
91                 elif RL_WRITABLE == 'ignored':
```

```
92                      hex_policy_and_addr = hex(int(hex_policy_and_addr, base) | 0
    x000A00000)
93                      str_policy = str_policy[:1] + 'A'
94                  elif RL_WRITABLE == 'clear only':
95                      hex_policy_and_addr =  hex(int(hex_policy_and_addr, base) | 0
    x000B00000)
96                      str_policy = str_policy[:1] + 'B'
97                  elif RL_WRITABLE == 'set only':
98                      hex_policy_and_addr =  hex(int(hex_policy_and_addr, base) | 0
    x000C00000)
99                      str_policy = str_policy[:1] + 'C'
100                 l_addresses_sv.extend(l_holes_sv)
101                 l_addresses_sv.append(str_policy + '\t07' + hex_addr)
102                 l_holes_sv.clear() # reset
103                 for i in range(pos_end_bit, pos_start_bit+1, 1):
104                     offset_pos = bit7 - i
105                     if l_fields[i] != '---':
106                         accessibility_mask |= (1 << (offset_pos + 28))
107                 hex_policy_and_addr =  hex(int(hex_policy_and_addr, base) |
    accessibility_mask)
108                 accessibility_mask = 0x1000000000 # reset
109                 # insert '7' inside the string
110                 hex_policy_and_addr = hex_policy_and_addr[:8] + '7' +
    hex_policy_and_addr[9:]
111                 l_policy_and_addr.extend(l_holes_c)
112                 l_policy_and_addr.append(hex_policy_and_addr[3:])
113                 l_holes_c.clear()
114             line = fid.readline()
115             linedec = line.decode('utf-8')
116         if previous_addr < (end_addr - 1):
117             [l_holes_c, l_holes_sv] = fill_holes([previous_addr])
118             l_addresses_sv.extend(l_holes_sv)
119             l_policy_and_addr.extend(l_holes_c)
120     finally:
121         fid.close()
122         return [l_policy_and_addr, l_addresses_sv]
123
124 def generate_output(fileout1, fileout2, l_policy_and_addr, l_addresses_sv):
125     flag_end_dims = False
126     l_dims = []
127     l_references = []
128     IDENTIFIER = '_UC_REG_INFO_H_'
129     ptr_to_arr = ''
130     NUM_REGS = len(l_policy_and_addr)
131     DIM_DIFF = NUM_REGS
132     dim_v = 1
133
134     fid1 = open(fileout1, 'w', encoding='utf-8')
135     fid1.write('#ifndef ' + IDENTIFIER + '\n#define ' + IDENTIFIER + '\n')
136     fid1.write('#include <stdint.h>\n')
137     fid1.write('#define NUM_BYTES {}\n'.format(TOT_BYTES))
138     fid1.write('#define NUM_WORDS {}\n'.format(int(TOT_WORDS)))
139     fid1.write('#define DIM_STR {} + 1 // take into account \'\\0\'\n'.format(DIM_STR))
140     if NUM_REGS <= MAX_DIM:
141         fid1.write('#define NUM_REGS ({} + 1)\n'.format(NUM_REGS))
142         dim_v += 1
143     else:
144         fid1.write('#define NUM_REGS {}\n'.format(MAX_DIM))
145         while flag_end_dims == False:
146             DIM_DIFF = DIM_DIFF - MAX_DIM
147             if DIM_DIFF < MAX_DIM:
148                 flag_end_dims = True
149                 if DIM_DIFF > 0:
150                     l_dims.append(DIM_DIFF)
151                     dim_v += 1
```

```
152             else:
153                 l_dims.append(MAX_DIM)
154                 dim_v += 1
155     fid1.write('#define DIM_V {}\n'.format(dim_v))
156     fid1.write('extern const char * m_policy_and_addr_1[NUM_REGS];\n')
157     ptr_to_arr = '(const char *) m_policy_and_addr_1'
158     if flag_end_dims == True:
159         for i in range(0, len(l_dims)):
160             fid1.write('extern const char * m_policy_and_addr_{}[{}];\n'.format(i+2,
    l_dims[i]))
161             l_references.append('(const char *) m_policy_and_addr_{}'.format(i+2))
162     fid1.write('extern const uint32_t v_dims[DIM_V];\n')
163     fid1.write('extern const char * v_references[DIM_V];\n')
164     fid1.write('#endif')
165     fid1.close()
166     fid2 = open(fileout2, 'w', encoding='utf-8')
167     fid3 = open(fileout3, 'w', encoding='utf-8')
168     fid2.write('#include "uc_reg_info.h"\n')
169     fid2.write('const char * m_policy_and_addr_1[NUM_REGS] = { \n')
170     cnt_item = 0
171     cnt_array_by_array = cnt_item
172     off_arr = 0
173     flag_offset = False
174     pos_dim = 0
175     l_dims.insert(0, MAX_DIM)
176     for cnt_item in range(0, len(l_policy_and_addr)):
177         if cnt_item < (len(l_policy_and_addr)-1):
178             if cnt_array_by_array < MAX_DIM-1:
179                 fid2.write('"{}", '.format(l_policy_and_addr[cnt_item]))
180                 fid3.write('{}\n'.format(l_addresses_sv[cnt_item]))
181             else:
182                 off_arr += 1
183                 flag_offset = True
184                 fid2.write('"{}"'.format(l_policy_and_addr[cnt_item]))
185                 fid2.write(' };\n')
186                 fid3.write('{}\n'.format(l_addresses_sv[cnt_item]))
187                 if off_arr < len(l_dims):
188                     fid2.write('const char * m_policy_and_addr_{}[{}] = '.format(
    off_arr+1, l_dims[off_arr]))
189                     fid2.write('{\n')
190                 cnt_array_by_array = 0 # reset
191                 pos_dim += 1
192         else:
193             fid2.write('"{}"'.format(l_policy_and_addr[cnt_item]))
194             fid3.write(l_addresses_sv[cnt_item])
195         cnt_item += 1
196         if flag_offset == False:
197             cnt_array_by_array += 1
198         flag_offset = False
199     fid2.write(' };\n')
200     if flag_end_dims == True:
201         fid2.write('const uint32_t v_dims[DIM_V] = {' + '{}, '.format(l_dims[0]))
202         for pos in range(1, len(l_dims)):
203             if pos < (len(l_dims)-1):
204                 fid2.write('{}, '.format(l_dims[pos]))
205             else:
206                 fid2.write('{}'.format(l_dims[pos]) + '};\n')
207         fid2.write('const char * v_references[DIM_V] = {' + ptr_to_arr + ', ')
208         for pos in range(0, len(l_references)):
209             if pos < len(l_references) - 1:
210                 fid2.write('{}, '.format(l_references[pos]))
211             else:
212                 fid2.write('{}'.format(l_references[pos]) + '};\n')
213     else:
```

```
214         fid2.write('const uint32_t v_dims[DIM_V] = {' + '{}'.format(len(
    l_policy_and_addr)) + '};\n')
215         fid2.write('const char * v_references[DIM_V] = {' + ptr_to_arr + '};\n;')
216     fid2.close()
217     fid3.close()
218
219 def fill_holes(l_params):
220     l_holes_c = []
221     l_holes_sv = []
222     base_addr = l_params[0]
223     if len(l_params) == 1:
224         last_addr = end_addr
225     else:
226         last_addr = l_params[1]
227     for addr in range(base_addr+1, last_addr):
228         l_holes_c.append('00aa07' + hex(addr)[2:])
229         l_holes_sv.append('AA\t07' + hex(addr)[2:])
230     return [l_holes_c, l_holes_sv]
231
232 # entry point of the script
233 if __name__ == "__main__":
234     l_policy_and_addr = [] # for C code manipulation
235     l_addresses_sv = []    # for SV code manipulation
236     if (MAX_DIM < 128) or (MAX_DIM > 255):
237         sys.exit('ERROR: Maximum dimension to be assigned to the array must be >= 128
    and <= 255')
238     else:
239         [l_policy_and_addr, l_addresses_sv] = parse_regmap(filein)
240         generate_output(fileout1, fileout2, l_policy_and_addr, l_addresses_sv)
241         print('The files ' + fileout1 + ' ' + fileout2 + ' ' + fileout3 + ' has been
    created starting from ' + filein)
```

C.2 Script di *parsing* per il test dei *System Registers*

```
1  #!/usr/bin/python
2  '''
3  ASSUMPTIONS:
4  input file converted from .xml to .txt in the format TAB DELIMITER generated by Excel
       conversion
5  ENCODING values:
6   - accessibility:
7         the two MSBs (hex values), each bit corresponds to the position of the bit in
       the reg; whether it is accessible is '1', '0' otherwise
8   - read
9     - 'R' == 'F'
10    - ''  == 'A'
11  - write
12    - 'W'  == 'F'
13    - 'WI' == 'D'
14    - ''   == 'A'
15  - if a register is NOT 'FREE' (a name assigned) but no policy is indicated --> encode
       the policy as 'EE'
16 '''
17 import sys
18 filein = 'REG_MAP_SYS.txt'
19 fileout1 = 'sys_reg_info.h'
20 fileout2 = 'sys_reg_info.c'
21 fileout3 = 'sys_reg_info.txt'
22 start_addr = 0x400
23 end_addr = 0x500
24 MAX_DIM = 50
25 DIM_STR = 9
```

```
26
27 def parse_regmap(filein):
28     d_parse = {}
29     l_fields = []
30     l_policy_and_addr = []
31     l_addresses_sv = []
32     line = ''
33     linedec = ''
34     EXTENDED_ADDRESS = ''
35     ADDRESS = ''
36     hex_addr = ''
37     hex_addr_complete = ''
38     hex_mask_usage = ''
39     hex_policy_and_addr = ''
40     RST_VAL = 0x00
41     mask_usage = RST_VAL
42     pos_extended_address = 0
43     pos_address = 1
44     pos_policy = 2
45     pos_name = 3
46     pos_MSB = 4
47     pos_LSB = 11
48     pos_car = 0
49     num_bits = pos_LSB - pos_MSB
50     base = 16
51     flag_stop = False
52     flag_match = False
53
54     try:
55         fid = open(filein, 'rb')
56     except:
57         print('ERROR: Unable to open ' + filein)
58     else:
59         print('Processing ' + filein + ' . . .')
60         line = fid.readline()
61         linedec = line.decode('utf-8')
62         while (linedec != '') and (not flag_stop):
63             l_fields = linedec.split('\t')
64             EXTENDED_ADDRESS = l_fields[pos_extended_address]
65             for pos_car in range(0, len(EXTENDED_ADDRESS)):
66                 if ((EXTENDED_ADDRESS[pos_car] == '9') and (pos_car == 0)):
67                     ADDRESS = l_fields[pos_address]
68                     ADDRESS = '0x' + ADDRESS
69                     int_addr = int(ADDRESS, base)
70                     hex_addr = hex(int_addr)[2:]
71                     if int_addr <= 0xF:
72                         hex_addr = '0' + hex_addr
73                     hex_addr_complete = '074' + hex_addr
74                     if l_fields[pos_name] == 'FREE':
75                         if l_fields[pos_policy] == '':
76                             hex_policy_and_addr = 'aa' + hex_addr_complete
77                         elif l_fields[pos_policy] == 'R':
78                             hex_policy_and_addr = 'ee' + hex_addr_complete
79                         mask_usage = RST_VAL
80                     else :
81                         if l_fields[pos_policy] == '':
82                             hex_policy_and_addr = 'fa' + hex_addr_complete
83                         elif l_fields[pos_policy] == 'RW':
84                             hex_policy_and_addr = 'ff' + hex_addr_complete
85                         elif l_fields[pos_policy] == 'R':
86                             hex_policy_and_addr = 'fa' + hex_addr_complete
87                         elif l_fields[pos_policy] == 'W':
88                             hex_policy_and_addr = 'af' + hex_addr_complete
89                         elif l_fields[pos_policy] == 'RWI':
90                             hex_policy_and_addr = 'fd' + hex_addr_complete
```

```
91                          for i in range(pos_LSB, pos_MSB-1, -1):
92                              if l_fields[i] == 'VOID':
93                                  mask_usage |= (0 << (num_bits - (pos_LSB - i)))
94                              else:
95                                  mask_usage |= (1 << (num_bits - (pos_LSB - i)))
96                      if mask_usage == RST_VAL:
97                          hex_mask_usage = '00'
98                      else:
99                          hex_mask_usage = hex(mask_usage)[2:]
100                         if mask_usage <= 0xF:
101                             hex_mask_usage = '0' + hex_mask_usage
102                     d_parse.update({hex_policy_and_addr : hex_mask_usage})
103                     mask_usage = RST_VAL # reset
104             line = fid.readline()
105             linedec = line.decode('utf-8')
106             if linedec.split('\t')[0] == '':
107                 flag_stop = True
108     finally:
109         fid.close()
110     return d_parse
111
112 def generate_output(fileout1, fileout2, d_parse):
113     l_dims = []
114     l_references = []
115     ptr_to_arr = ''
116     IDENTIFIER = '_SYS_REG_INFO_H_'
117     NUM_REGS = len(d_parse)
118     DIM_DIFF = NUM_REGS
119     cnt_addr = 0
120     cnt_array_by_array = cnt_addr
121     off_arr = 0
122     dim_v = 1
123     pos_dim = 0
124     flag_end_dims = False
125     flag_offset = False
126
127     fid1 = open(fileout1, 'w', encoding='utf-8')
128     fid1.write('#ifndef ' + IDENTIFIER + '\n#define ' + IDENTIFIER + '\n')
129     fid1.write('#include <stdint.h>\n')
130     fid1.write('#define DIM_STR {} + 1 // take into account \'\\0\'\n'.format(DIM_STR))
131     if NUM_REGS <= MAX_DIM:
132         fid1.write('#define NUM_REGS ({} + 1)\n'.format(NUM_REGS))
133         dim_v += 1
134     else:
135         fid1.write('#define NUM_REGS {}\n'.format(MAX_DIM))
136         while flag_end_dims == False:
137             DIM_DIFF = DIM_DIFF - MAX_DIM
138             if DIM_DIFF < MAX_DIM:
139                 flag_end_dims = True
140                 if DIM_DIFF > 0:
141                     l_dims.append(DIM_DIFF)
142                     dim_v += 1
143             else:
144                 l_dims.append(MAX_DIM)
145                 dim_v += 1
146     fid1.write('#define DIM_V {}\n'.format(dim_v))
147     fid1.write('extern const char * m_policy_and_addr_1[NUM_REGS];\n')
148     ptr_to_arr = '(const char *) m_policy_and_addr_1'
149     if flag_end_dims == True:
150         for i in range(0, len(l_dims)):
151             fid1.write('extern const char * m_policy_and_addr_{}[{}];\n'.format(i+2,
    l_dims[i]))
152             l_references.append('(const char *) m_policy_and_addr_{}'.format(i+2))
153     fid1.write('extern const uint32_t v_dims[DIM_V];\n')
154     fid1.write('extern const char * v_references[DIM_V];\n')
```

```
155     fid1.write('#endif')
156     fid1.close()
157     fid2 = open(fileout2, 'w', encoding='utf-8')
158     fid3 = open(fileout3, 'w', encoding='utf-8')
159     fid2.write('#include "sys_reg_info.h"\n')
160     fid2.write('const char * m_policy_and_addr_1[NUM_REGS] = { \n')
161     l_dims.insert(0, MAX_DIM)
162     for key in d_parse.keys():
163         if cnt_addr < (len(d_parse)-1):
164             if cnt_array_by_array < MAX_DIM-1:
165                 fid2.write('"{}", '.format(d_parse[key] + key))
166             else:
167                 off_arr += 1
168                 flag_offset = True
169                 fid2.write('"{}"'.format(d_parse[key] + key))
170                 fid2.write(' };\n')
171                 if off_arr < len(l_dims):
172                     fid2.write('const char * m_policy_and_addr_{}[{}] = '.format(
    off_arr+1, l_dims[off_arr]))
173                     fid2.write('{\n')
174                 cnt_array_by_array = 0 # reset
175                 pos_dim += 1
176             fid3.write('{}\t{}\n'.format(key[:2].upper(), key[2:]))
177         else:
178             fid2.write('"{}"'.format(d_parse[key] + key))
179             fid3.write('{}\t{}'.format(key[:2].upper(), key[2:]))
180         cnt_addr += 1
181         if flag_offset == False:
182             cnt_array_by_array += 1
183         flag_offset = False
184     fid2.write(' };\n')
185     if flag_end_dims == True:
186         fid2.write('const uint32_t v_dims[DIM_V] = {' + '{}, '.format(l_dims[0]))
187         for pos in range(1, len(l_dims)):
188             if pos < (len(l_dims)-1):
189                 fid2.write('{}, '.format(l_dims[pos]))
190             else:
191                 fid2.write('{}'.format(l_dims[pos]) + '};\n')
192         fid2.write('const char * v_references[DIM_V] = {' + ptr_to_arr + ', ')
193         for pos in range(0, len(l_references)):
194             if pos < len(l_references) - 1:
195                 fid2.write('{}, '.format(l_references[pos]))
196             else:
197                 fid2.write('{}'.format(l_references[pos]) + '};\n')
198     else:
199         fid2.write('const uint32_t v_dims[DIM_V] = {' + '{}'.format(len(d_parse)) +
    '};\n')
200         fid2.write('const char * v_references[DIM_V] = {' + ptr_to_arr + '};\n;')
201     fid2.close()
202     fid3.close()
203
204 # entry point of the script
205 if __name__ == "__main__":
206     d_parse = {}
207     if (MAX_DIM <= 0) or (MAX_DIM > 128):
208         sys.exit('ERROR: Maximum dimension to be assigned to the array must be > 0 and
    <= 128')
209     else:
210         d_parse = parse_regmap(filein)
211         generate_output(fileout1, fileout2, d_parse)
212         print('The files ' + fileout1 + ' ' + fileout2 + ' ' + fileout3 + ' have been
    created starting from ' + filein)
```

Bibliografia

[1] L. Maestri. *Ambiente in System Verilog con metodologia UVM per la verifica della sincronizzazione fra sensori e Application Processor*. Tesi di Laurea Magistrale, Università degli Studi di Pavia, 2013/2014 (cit. on pp. 5, 6, 8, 9, 12, 24, 128, 129).

[2] A. Locardi. *Ambiente UVM a parametri randomici per la verifica della sezione digitale in sensori MEMS*. Tesi di Laurea Magistrale, Università degli Studi di Pavia, 2011/2012 (cit. on pp. 5, 8, 9, 11, 12, 23, 24, 46, 128, 129).

[3] C. Patel and P. McCluskey. «Performance Degradation of the MEMS Vibratory Gyroscope in Harsh Environments». In: vol. 11. Jan. 2011. DOI: 10.1115/IMECE2011-65001 (cit. on p. 9).

[4] A. E. Kubba, A. Hasson, A. I. Kubba, and G. Hall. «A micro-capacitive pressure sensor design and modelling». In: *Journal of Sensors and Sensor Systems* 5 (Mar. 2016), pp. 95–112. DOI: 10.5194/jsss-5-95-2016 (cit. on pp. 9, 10).

[5] D. Bisio. *Studio, modellizzazione real-time e implementazione di un testbench, per la catena di elaborazione digitale dei segnali, per un sensore MEMS*. Tesi di Laurea Magistrale, Politecnico di Torino, 2017/2018 (cit. on pp. 10, 46).

[6] *UM10204 I2C-bus specification and user manual, Rev. 7.0*. NXP Semiconductors. 1 October 2021. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf (cit. on pp. 13, 126, 127).

[7] K. A. Meade and S. Rosenberg. *A Practical Guide to Adopting the Universal Verification Methodology (UVM), Second Edition*. Cadence Design Systems, 2013. ISBN: 978-1-300-53593-5 (cit. on pp. 23, 24, 131).