UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in ingegneria elettronica
e delle tecnologie digitali

Implementazione su FPGA di CPU RISC-V e interfacciamento verso periferiche

FPGA implementation of RISC-V CPUs and interfacing to peripherals

Candidato:
Lorenzo Fabiani

Relatore:
Prof. Giorgio Biagetti

Anno Accademico 2025-2026

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in ingegneria elettronica
e delle tecnologie digitali

Implementazione su FPGA di CPU RISC-V e interfacciamento verso periferiche

FPGA implementation of RISC-V CPUs and interfacing to peripherals

Candidato:
Lorenzo Fabiani

Relatore:
Prof. Giorgio Biagetti

Anno Accademico 2025-2026

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in ingegneria elettronica
e delle tecnologie digitali
Via Brecce Bianche – 60131 Ancona (AN), Italy

"All problems in computer science can be solved by another level of indirection."

— David Wheeler

Abstract

Il paradigma open-source sta rivoluzionando l'architettura dei calcolatori, con il set di istruzioni RISC-V che si sta affermando come standard globale, trovando applicazione dai sistemi embedded fino a piattaforme di largo consumo come i microcontrollori Raspberry Pi. Per comprendere a fondo le potenzialità e le dinamiche di integrazione di questa architettura, il presente lavoro di tesi illustra la progettazione in VHDL di un bridge di comunicazione per sistemi su FPGA. Il progetto si innesta su un'infrastruttura preesistente, originariamente sviluppata per l'acquisizione di segnali elettrocardiografici (ECG) e gestita da un processore soft-core 6502. L'obiettivo centrale è l'ammodernamento del sistema tramite la sostituzione del vecchio core con un moderno processore RISC-V, mantenendo intatte le periferiche e interfacciandosi con il bus di sistema legacy. Tra le varie implementazioni disponibili, la scelta è ricaduta sul soft-core NEORV32, selezionato per la sua eccellente organizzazione strutturale, l'approccio interamente open-source e la flessibilità della sua interfaccia di bus. La metodologia ha previsto una prima fase di ricerca e studio comparativo delle specifiche architettoniche dei due protocolli di comunicazione, seguita dallo sviluppo e dalla sintesi della logica di traduzione (bridge) tramite l'ambiente Xilinx Vivado. I risultati confermano la fattibilità e i vantaggi dell'aggiornamento di architetture legacy verso l'ecosistema RISC-V, aprendo la strada a sviluppi futuri focalizzati sull'ottimizzazione delle latenze del bridge e sull'integrazione di acceleratori hardware dedicati all'elaborazione dei segnali biomedicali.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 evoluzione dei microprocessori

Il microprocessore (o CPU, *Central Processing Unit*) rappresenta il nucleo elaborativo fondamentale della moderna elettronica digitale. A livello puramente architetturale, esso è una complessa rete logica sequenziale composta da unità funzionali ben precise: l'Unità Aritmetico-Logica (ALU) deputata all'esecuzione dei calcoli matematici e logici, l'Unità di Controllo (CU) che orchestra il flusso delle operazioni, e un set di registri interni per la memorizzazione temporanea e rapida dei dati. Il suo funzionamento si basa sul continuo interscambio di informazioni con le memorie e le periferiche esterne tramite i bus di sistema (dati, indirizzi e controllo), seguendo il classico ciclo operativo di *fetch, decode ed execute* delle istruzioni.

Se nei computer di uso generale (*General Purpose*) il microprocessore è concepito per massimizzare il *throughput* e le prestazioni assolute su un'ampia e imprevedibile varietà di carichi di lavoro (sistemi operativi complessi, elaborazione grafica, multitasking intensivo), nel vasto mondo dei sistemi *embedded* il suo ruolo assume connotati radicalmente differenti. Qui, il microprocessore è tipicamente integrato all'interno di un singolo chip chiamato microcontrollore (MCU), che racchiude non solo la CPU, ma anche memorie (RAM per i dati e Flash per il firmware) e periferiche di input/output (timer, ADC, interfacce di comunicazione).

Questa architettura compatta è progettata per eseguire firmware che si interfaccia continuamente con il mondo fisico tramite sensori e attuatori. A differenza del mondo PC, la progettazione *embedded* è dominata da vincoli stringenti noti come SWaP (*Size, Weight, and Power*): l'ingombro fisico, i consumi energetici e la dissipazione termica devono essere ridotti al minimo. A questo si aggiunge la necessità di operare in tempo reale (*real-time*): il sistema deve garantire risposte deterministiche e tempi di latenza prevedibili in risposta a eventi esterni (*interrupt*). Applicazioni critiche, come l'acquisizione e l'elaborazione di segnali biomedicali (ad esempio i tracciati ECG), si basano pesantemente su queste architetture, dove un ritardo nell'elaborazione del segnale comprometterebbe l'intera funzionalità del dispositivo.

Nel corso dei decenni, per far fronte a queste esigenze, la progettazione interna dei microprocessori ha subìto una profonda evoluzione, segnata principalmente dal

passaggio dalle architetture CISC (*Complex Instruction Set Computer*) a quelle RISC (*Reduced Instruction Set Computer*).

Le architetture CISC, storicamente dominanti agli albori dell'informatica (come la famiglia x86 o il vecchio 6502), sono nate in un'epoca in cui la memoria era una risorsa estremamente costosa e lenta. L'obiettivo primario era quindi minimizzare la lunghezza del codice sorgente: le istruzioni CISC sono molto complesse e in grado di eseguire operazioni multi-passaggio (come leggere dalla memoria, sommare e riscrivere) con una singola riga di codice assembly. Questo approccio, tuttavia, sposta la complessità sull'hardware: richiede un'unità di decodifica interna enorme e tempi di esecuzione variabili per ogni istruzione (da 1 a decine di cicli di clock), rendendo difficile l'ottimizzazione e aumentando i consumi.

Con il drastico abbassamento dei costi delle memorie e l'evoluzione delle tecniche di compilazione, l'industria si è progressivamente spostata verso il paradigma RISC. Questa filosofia progettuale inverte il problema: utilizza un set di istruzioni base molto più ridotto e standardizzato. La regola d'oro del RISC è che ogni istruzione sia sufficientemente semplice da poter essere eseguita idealmente in un singolo ciclo di clock. Questo approccio ha permesso di semplificare drasticamente la complessa logica di controllo interna al silicio, lasciando spazio prezioso per implementare un maggior numero di registri generali e, soprattutto, tecniche di *pipelining* avanzate (la sovrapposizione dell'esecuzione di più istruzioni). Il risultato è un'architettura in grado di raggiungere frequenze operative più elevate, consumi energetici nettamente inferiori e, aspetto cruciale per il mondo *embedded*, un'altissima predicibilità nei tempi di esecuzione.

Sebbene il paradigma RISC abbia letteralmente conquistato il mercato dei sistemi *embedded* (si pensi al dominio incontrastato della famiglia ARM Cortex in smartphone, automotive e IoT), questo ecosistema si è storicamente consolidato attorno a modelli di business strettamente proprietari. In questo scenario convenzionale, l'Instruction Set Architecture (ISA) è protetta da rigidi brevetti. L'integrazione di un processore commerciale all'interno di un progetto *System-on-Chip (SoC)* o di un'infrastruttura FPGA richiede la negoziazione di costose licenze e, spesso, il pagamento di royalties per ogni singolo chip prodotto.

Inoltre, i core forniti dai grandi *vendor* sono frequentemente distribuiti sotto forma di *hard-macro* o con codice RTL (*Register Transfer Level*) offuscato, operando di fatto come delle *black-box*. Questo approccio chiuso impone severi limiti all'ingegnere elettronico: impedisce l'ispezione approfondita della logica interna (cruciale per il debug avanzato e la validazione della sicurezza) e vieta categoricamente la modifica dell'ISA. Diventa quindi impossibile, ad esempio, aggiungere un'istruzione hardware customizzata per accelerare un particolare algoritmo di filtraggio digitale di un segnale ECG.

Per superare queste forti limitazioni tecnologiche e commerciali, negli ultimi anni si sta assistendo a una vera e propria rivoluzione guidata dal paradigma dell'hardware *open-source*, il cui indiscusso portabandiera è l'architettura RISC-V. Nata originariamente all'Università della California, Berkeley, e oggi mantenuta dall'ente no-profit RISC-V International, non si tratta di un singolo processore fisico, ma di una specifica ISA aperta e completamente gratuita. Chiunque può scaricare le specifiche tecniche e progettare, simulare e sintetizzare il proprio microprocessore compatibile senza dover pagare alcuna licenza, eliminando di fatto il pericoloso *vendor lock-in*.

La vera forza progettuale di RISC-V, che lo rende particolarmente attrattivo per il mondo *embedded* e per le implementazioni su FPGA, risiede nella sua estrema modularità. L'architettura è costituita da un set di istruzioni base intero e minimale (ad esempio RV32I per i sistemi a 32 bit), a cui i progettisti possono aggiungere estensioni standardizzate solo se strettamente necessarie per l'applicazione target (come l'estensione 'M' per moltiplicazioni e divisioni hardware, o 'F' per l'unità in virgola mobile). Questo permette di ottimizzare al millimetro l'area logica occupata sul silicio o le risorse (LUT e Flip-Flop) sulla FPGA.

Ancora più importante, lo standard RISC-V prevede esplicitamente uno spazio di codifica libero per l'implementazione di istruzioni e acceleratori hardware *custom*. È proprio questa flessibilità senza precedenti, unita alla disponibilità in rete di core RTL *open-source* altamente ingegnerizzati e pronti all'uso, che rende l'ecosistema RISC-V il candidato ideale per l'ammodernamento di sistemi digitali preesistenti. Sostituire un obsoleto processore legacy con un moderno core RISC-V su FPGA non significa solo aggiornare l'hardware, ma abbracciare un'infrastruttura scalabile, totalmente ispezionabile e perfettamente adattabile alle future esigenze dell'elaborazione biomedicale.

1.2 Obiettivo della Tesi

Il presente lavoro di tesi si prefigge l'obiettivo di ammodernare l'architettura di un sistema *embedded* preesistente, implementato su piattaforma FPGA. Nello specifico, il progetto verte sull'integrazione del processore *soft-core* NEORV32, basato sull'Instruction Set Architecture (ISA) RISC-V a 32 bit, in sostituzione di un preesistente microprocessore *legacy* a 8 bit (MOS 6502).

Per rendere possibile e funzionale tale migrazione architetturale, preservando al contempo l'integrità e l'utilizzo delle periferiche originali del sistema, il lavoro di ricerca e sviluppo si è articolato in due macro-aree principali:

- **Progettazione logica di un modulo Bridge**: La prima fase ha riguardato la stesura in linguaggio VHDL di un'interfaccia di comunicazione hardware (Bridge) in grado di fungere da mediatore tra due domini architetturali radicalmente differenti. Il modulo è stato progettato per tradurre le transazioni a

32 bit del bus del NEORV32 (XBUS) in segnali e temporizzazioni compatibili con il bus a 8 bit del sistema *legacy*. Questa fase ha richiesto un'attenta implementazione delle logiche di *routing*, la prevenzione di errori di *aliasing* tramite la traduzione degli indirizzi di memoria e la rigorosa gestione delle latenze di lettura e scrittura.

- **Integrazione del sistema di *Debug* on-chip (JTAG)**: La seconda fase ha avuto come scopo primario l'abilitazione della modalità di *debugging* hardware del processore NEORV32, passaggio fondamentale per poter ispezionare e validare sul campo il corretto funzionamento del *core* e del Bridge stesso. Poiché i pin JTAG fisici dell'FPGA risultano intrinsecamente vincolati dal programmatore della *board* (utilizzati per il caricamento del *bitstream*), si è reso necessario un profondo intervento a livello RTL sul Debug Transport Module (DTM) del processore. L'infrastruttura di ricezione originale è stata interamente riprogettata istanziando specifiche primitive di libreria Xilinx (`BSCANE2`) per intercettare i comandi JTAG dall'infrastruttura di *routing* interna. Contestualmente, si è agito sul meccanismo di sincronizzazione, disaccoppiando il dominio di *clock* del *debugger* da quello della CPU, garantendo così una comunicazione robusta e totalmente indipendente dagli stati operativi a basso consumo (*low-power*) del processore.

1.3 Struttura della Tesi

Il presente elaborato è strutturato in modo da guidare il lettore attraverso l'intero ciclo di vita del progetto, dalla pianificazione metodologica alla validazione finale del sistema integrato:

- **Capitolo 2 - Metodologia di Progettazione e Validazione Preliminare**: introduce il contesto operativo e l'ambiente di sviluppo, definendo le strategie iniziali per l'isolamento delle periferiche e la definizione di un controller custom necessario alla gestione dei segnali.
- **Capitolo 3 - Progettazione e Integrazione del Bridge RISC-V**: vengono analizzate le differenze architetturali tra il protocollo XBUS del core NEORV32 e le specifiche del core 6502. Vengono qui descritte l'implementazione del modulo Bridge in VHDL, la gestione dei cicli di lettura/scrittura e le tecniche di *pipelining* adottate.
- **Capitolo 4 - Validazione Hardware e Risultati di Simulazione**: presenta le fasi di verifica del Bridge tramite simulazioni *RTL*. Viene analizzata la sequenza degli eventi software e la coerenza dei dati in transito, validando il comportamento dell'hardware prima del caricamento su FPGA.

- **Capitolo 5 - Integrazione dell'Interfaccia di Debug JTAG On-Chip**: descrive la complessa attività di riprogettazione del modulo DTM. Viene dettagliato l'utilizzo delle primitive `BSCANE2` per superare i vincoli di accesso JTAG della scheda e l'implementazione del protocollo di *handshake* a quattro fasi per la sincronizzazione tra domini di *clock* indipendenti.
- **Capitolo 6 - Validazione del Sistema Integrato: Esecuzione del Firmware ECG**: rappresenta la conclusione pratica del lavoro. Viene documentata l'esecuzione del firmware ECG originale sulla nuova architettura ibrida e presentata un'analisi comparativa dei tempi di esecuzione tra il sistema basato su 6502 e quello basato su NEORV32, confermando il superamento delle specifiche iniziali.

Capitolo 2

Metodologia di Progettazione e Validazione Preliminare

2.1 Scopo del progetto intermedio e ambiente di sviluppo

Prima di affrontare la complessa integrazione del bridge per il processore RISC-V, è stata condotta una fase di studio e validazione pratica finalizzata a padroneggiare l'ambiente di sviluppo Xilinx Vivado e le logiche della progettazione hardware su FPGA. A tale scopo, è stato definito un progetto intermedio con l'obiettivo di instaurare una comunicazione seriale asincrona tra un PC host e la scheda FPGA.

Il task specifico richiedeva l'invio di caratteri ASCII tramite l'interfaccia seriale USB (sfruttando il Serial Monitor dell'IDE di Arduino) per comandare l'accensione selettiva di un LED RGB montato sulla scheda. In particolare, la ricezione dei caratteri 'R', 'G' o 'B' doveva tradursi nell'attivazione del rispettivo canale cromatico (Rosso, Verde o Blu). Questa prova ha permesso di analizzare nel dettaglio il comportamento delle interfacce di comunicazione preesistenti e di sviluppare logiche di controllo custom basate su Macchine a Stati Finiti (FSM).

2.2 Isolamento delle periferiche e ricablaggio a livello Top-Level

Nel progetto di partenza, basato sul core 6502, le periferiche di I/O — inclusi il modulo UART per la comunicazione seriale e i LED di diagnostica — erano mappate in memoria e interfacciate direttamente al bus del processore. Per permettere alla nuova logica dedicata di gestire la comunicazione seriale e l'accensione del LED, è stato innanzitutto necessario intervenire strutturalmente sul file *top-level* del progetto all'interno dell'ambiente Xilinx Vivado.

Questo intervento si è reso indispensabile per due motivi principali. Il primo riguarda il *routing* dei segnali. Essendo il modulo UART originariamente asservito in via esclusiva alla CPU, è stato necessario intervenire nel *top-level* per scollegare i segnali di interfacciamento della seriale (come il bus dati in ricezione `Do`, il segnale di interrupt `IRQ` e i comandi di lettura/scrittura) dal bus di sistema legacy. Questi

segnali sono stati quindi "ricablati" (re-indirizzati) per connetterli direttamente alle porte in ingresso e in uscita del nuovo controller custom USART RGB, rendendo quest'ultimo l'unico arbitro della comunicazione verso il PC.

Il secondo motivo, altrettanto critico nella progettazione digitale su FPGA, riguarda la gestione degli accessi ai pin fisici di output e il rigoroso rispetto della regola del driver singolo. In VHDL, un netto (collegamento fisico) non può essere pilotato simultaneamente da due sorgenti logiche indipendenti. Lasciare i pin del LED RGB collegati sia all'architettura originale della CPU 6502 sia al nuovo controller avrebbe generato un irrisolvibile errore di sintesi per *multiple drivers*. Pertanto si è proceduto a rimuovere l'assegnazione originaria dei LED, isolandoli per renderli accessibili in via esclusiva al nuovo modulo di controllo.

2.3 Progettazione del controller custom (FSM)

Il cuore del progetto intermedio è costituito dal modulo VHDL USART RGB, progettato per interfacciarsi con i segnali di controllo esposti dal modulo UART preesistente. La logica di controllo è stata modellata come una Macchina a Stati Finiti (FSM) temporizzata dal segnale di *clock* di sistema.

Durante l'analisi dei diagrammi temporali della UART, è emerso un vincolo progettuale fondamentale: la latenza di lettura. Quando la UART riceve un byte, alza il segnale di interrupt IRQ. Tuttavia, asserendo il segnale di abilitazione alla lettura `rd`, il dato non è immediatamente disponibile sul bus dati `Do`, ma richiede un colpo di *clock* aggiuntivo per essere propagato in uscita dai registri interni.

Per gestire correttamente questa latenza, la FSM è stata strutturata su quattro stati:

- **RESET**: Al suo verificarsi, la FSM viene forzata allo stato iniziale e tutti i segnali di controllo vengono azzerati. È esclusivamente in questa fase che i LED vengono spenti di default (tramite logica attiva-bassa, assegnando il vettore logico "111"(riga 8)).
- **IDLE**: Lo stato di attesa. Il controller monitora il segnale `IRQ`. Non appena la UART segnala la disponibilità di un nuovo dato alzando la linea di interrupt, la FSM asserisce il segnale di lettura `rd` e transita nello stato successivo.
- **REQUEST READ**: Questo stato funge da ritardo intenzionale (stadio di pipeline). Il segnale `rd` viene de-asserito, poiché la UART ne ha già campionato il fronte di salita, e il sistema attende un ciclo di clock per garantire la stabilità del dato sul bus.
- **UPDATE LED**: In questo stato, il dato sul bus `Do` è stabile e pronto per essere campionato. Il byte ricevuto viene confrontato con le codifiche esadecimali dei caratteri ASCII attesi (x"52" per 'R', x"47" per 'G', x"42" per 'B'). A seconda

del match, viene pilotato il segnale `rgb_led` con la codifica corretta, per poi riportare la macchina nello stato di IDLE in attesa di un nuovo comando.

Di seguito si riporta l'estratto del codice VHDL relativo all'implementazione del processo sequenziale della FSM:

```
process(clk)
begin
    if rising_edge(clk) then
        if rst = '1' then
            current_state <= IDLE;
            rd <= '0';
            a <= "0";
            rgb_led <= "111"; -- Logica attiva-bassa: LED spento
        else
            case current_state is
                -- STATO 0: Attesa interrupt dalla UART
                when IDLE =>
                    rd <= '0';
                    if irq = '1' then
                        a <= "0";     -- Selezione registro dati
                        rd <= '1';    -- Asserzione segnale di Read
                        current_state <= REQUEST_READ;
                    end if;

                -- STATO 1: Ciclo di attesa per la stabilita' del dato
                when REQUEST_READ =>
                    rd <= '0';
                    current_state <= UPDATE_LED;

                -- STATO 2: Campionamento dato e aggiornamento uscite
                when UPDATE_LED =>
                    if do = x"52" then    -- Carattere 'R'
                        rgb_led <= "011";
                    elsif do = x"47" then -- Carattere 'G'
                        rgb_led <= "101";
                    elsif do = x"42" then -- Carattere 'B'
                        rgb_led <= "110";
                    elsif do = x"30" then -- Carattere '0' (
    Spegnimento)
                        rgb_led <= "111";
                    end if;
                    current_state <= IDLE;

                when others =>
                    current_state <= IDLE;
            end case;
        end if;
    end if;
end process;
```

Listing 2.1: Estratto della FSM per il controllo del LED RGB

A completamento dell'analisi logica, la Figura 2.1 illustra il diagramma temporale che descrive l'esatta sequenza degli eventi durante la ricezione di un carattere. Nel caso di esempio, viene analizzata la ricezione del carattere ASCII 'R' (codifica esadecimale `x"52"`).

Come si evince dal grafico, il sistema parte da una condizione di riposo (stato di `IDLE`), in cui i LED sono spenti e si attende l'asserzione del segnale `IRQ`. Non appena la UART segnala la disponibilità di un nuovo dato alzando la linea di interrupt, la FSM campiona l'evento sul primo fronte di salita utile del *clock* e transita nello stato `REQUEST_READ`, asserendo il comando `rd` per un singolo ciclo. È in questa fase che si apprezza visivamente la latenza di lettura della periferica: il dato sul bus `Do` non è ancora stabile e viene ignorato dalla FSM.

Solo nel ciclo successivo, in corrispondenza del passaggio allo stato `UPDATE_LED`, la UART propaga in uscita il byte `x"52"`. La FSM può quindi campionarlo in modo sicuro, decodificarlo e pilotare conseguentemente il bus `rgb_led` con la configurazione `"011"`, corrispondente all'accensione del canale rosso, per poi ritornare immediatamente in stato di attesa. Questo approccio garantisce la totale sincronizzazione con la periferica originaria del sistema legacy, evitando la lettura di dati spuri.

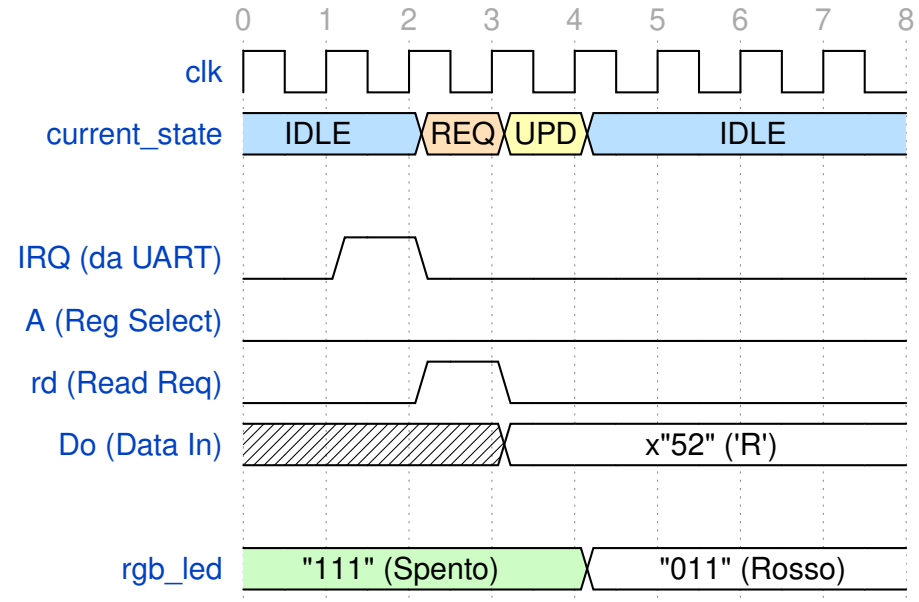

Figura 2.1: Diagramma temporale dell'interazione tra FSM, UART e pilotaggio LED.

Capitolo 3

Progettazione e Integrazione del Bridge RISC-V

3.1 Fase Preliminare: Integrazione e Validazione del Core NEORV32

Al fine di acquisire familiarità con l'architettura e il flusso di lavoro del soft-core RISC-V selezionato, prima di procedere con lo sviluppo del *bridge* si è ritenuto opportuno condurre una fase di test preliminare. L'obiettivo di questo step intermedio è stato l'integrazione del processore NEORV32 all'interno dell'FPGA e l'esecuzione di un semplice applicativo per il controllo di un LED RGB, validando così il corretto funzionamento del sistema hardware e della *toolchain* software.

Il punto di partenza è consistito nel reperimento dei file sorgente del processore, resi disponibili in formato open-source tramite la repository GitHub ufficiale del progetto NEORV32 [1]. Sono stati pertanto importati all'interno dell'ambiente di sviluppo i file VHDL d'interesse, nello specifico i moduli presenti nella directory `core` — incaricati di implementare la CPU e le periferiche interne — unitamente ai *top-level* di test forniti dagli sviluppatori.

Per adattare l'implementazione generica alla scheda FPGA in uso, si è resa necessaria la stesura di un modulo *top-level* personalizzato (*wrapper*) che includesse e istanziasse il modulo di test di base. Questo livello gerarchico aggiuntivo ha permesso di definire il corretto interfacciamento fisico del processore con l'hardware esterno della board. Nello specifico, si è proceduto a:

- **Mappatura della comunicazione seriale:** Le linee UART interne all'FPGA sono state connesse ai pin fisici della scheda, requisito fondamentale per la comunicazione seriale con il PC e per il caricamento del firmware.
- **Interfacciamento I/O:** I segnali di output digitali generati dal processore sono stati instradati verso i pin fisici preposti al pilotaggio del LED RGB integrato sulla board.
- **Gestione dei segnali di temporizzazione:** È stato inserito e configurato il generatore di clock della scheda, collegando in modo opportuno le linee di *clock* e di *reset* all'istanza del processore RISC-V.

Terminata e sintetizzata con successo la configurazione hardware, l'attenzione si è spostata sullo sviluppo software. È stato redatto un elementare programma in linguaggio C avente lo scopo di generare un'alternanza cromatica sul LED RGB. Una volta compilato il codice sorgente, è emersa una specificità del flusso di lavoro del NEORV32: il *bootloader* predefinito del processore, che si pone in ascolto sulla porta UART durante la fase di avvio, non è progettato per accettare direttamente eseguibili in formato standard (come file `.elf` o binari *raw*).

Pertanto, è stato necessario utilizzare un apposito tool software fornito all'interno della repository (denominato `image_gen`). Questo strumento elabora il binario compilato e genera un file eseguibile in un formato specifico accettato dal sistema (tipicamente `neorv32_exe.bin`). A livello tecnico, l'operazione consiste nell'apposizione di un *header* iniziale di 16 byte che include una firma di riconoscimento ("NEO!"), l'indirizzo di base, la dimensione totale dell'immagine e un *checksum*. Grazie a questi metadati, il bootloader può verificare l'integrità e la coerenza del programma trasmesso prima di avviarne l'esecuzione.

Trasmettendo questo specifico eseguibile via UART tramite un emulatore di terminale, il bootloader ha caricato correttamente le istruzioni nella memoria del processore. Il successivo e corretto lampeggio del LED ha fornito un riscontro visivo e funzionale immediato, confermando in via definitiva che l'implementazione del core RISC-V era funzionante e correttamente integrata nell'ecosistema della scheda.

3.2 Analisi comparativa delle interfacce di Bus

Prima di procedere alla stesura dell'architettura hardware del bridge, è fondamentale analizzare in dettaglio i protocolli di comunicazione esposti dai due domini di interfacciamento. Da un lato, il processore RISC-V NEORV32 impiega un bus proprietario a 32-bit denominato XBUS (fortemente ispirato e compatibile con lo standard Wishbone); dall'altro, le periferiche del sistema legacy si aspettano di dialogare con un'interfaccia a 8-bit che emula le tempistiche e la piedinatura del processore MOS 6502. Il compito del bridge sarà proprio quello di mediare tra questi due standard operando una traduzione asimmetrica dei segnali di controllo, degli indirizzi e dei dati.

3.2.1 Il protocollo XBUS del processore NEORV32

L'interfaccia esterna XBUS del NEORV32 è un bus sincrono, governato dal *clock* di sistema, progettato per gestire trasferimenti di memoria mappata. Per la comunicazione con le periferiche esterne, l'interfaccia espone un set di segnali ben definito:

- `xbus_adr_o (32-bit)`: Bus degli indirizzi in uscita.

- `xbus_dat_o (32-bit)` e `xbus_dat_i (32-bit)`: Bus dati separati per le operazioni di scrittura e lettura.
- `xbus_we_o (1-bit)`: Segnale di abilitazione alla scrittura (*Write Enable*). Se asserito, indica un'operazione di scrittura; se basso, una lettura.
- `xbus_sel_o (4-bit)`: *Byte enable*. Indica quali byte del bus dati a 32-bit sono validi durante il trasferimento. Nel caso specifico di questo progetto, dovendo comunicare con un bus a 8-bit, sarà un parametro cruciale per la traduzione.
- `xbus_stb_o (1-bit)`: *Strobe*. Viene asserito per indicare l'inizio di una nuova transazione.
- `xbus_cyc_o (1-bit)`: *Cycle valid*. Rimane asserito per l'intera durata del ciclo di bus, permettendo di raggruppare transazioni multiple.
- `xbus_ack_i (1-bit)` e `xbus_err_i (1-bit)`: Segnali di risposta generati dalla periferica target per segnalare, rispettivamente, il completamento con successo del trasferimento (*Acknowledge*) o un errore sul bus.
- `xbus_cti_o (3-bit)` e `xbus_tag_o (3-bit)`: Segnali di stato aggiuntivi. Il *Cycle Type Identifier* (`cti`) definisce il tipo di accesso (nel nostro caso fissato a `000` per accessi singoli), mentre il `tag` fornisce metadati compatibili con lo standard AXI4 (come la discriminazione tra fetch di istruzioni e dati, o il livello di privilegio).

Il NEORV32 implementa diverse tipologie di trasferimento. L'operazione di base, che verrà intercettata e gestita dal bridge per comunicare con il sistema legacy, è il *Single Access* (Accesso Singolo).

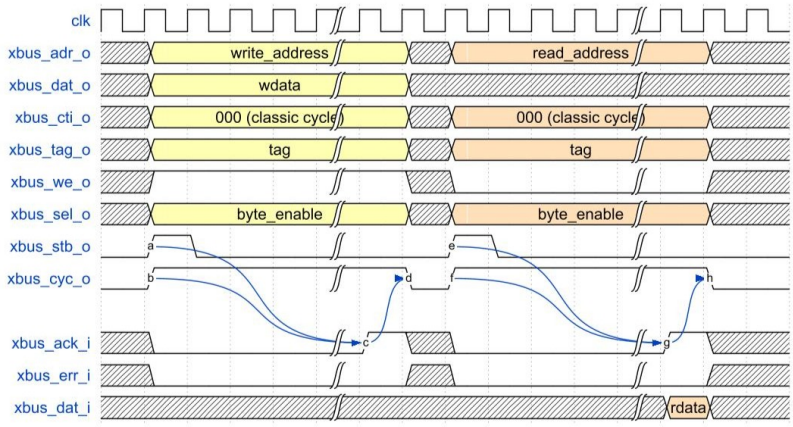

Figura 3.1: Diagramma temporale di un ciclo *Single Access* sull'interfaccia XBUS (scrittura a sinistra, lettura a destra). Riprodotta da: `https://stnolting.github.io/neorv32/#_processor_external_bus_interface_xbus` secondo la licensa BSD-3-Clause.

Come illustrato nella Figura 3.1, durante un ciclo di scrittura *Single Access*, il master (la CPU) posiziona l'indirizzo su `xbus_adr_o` e il dato su `xbus_dat_o`, asserendo contestualmente `xbus_we_o` e le linee `xbus_sel_o`. Il trasferimento viene avviato innalzando il segnale `xbus_cyc_o` per definire l'inizio del ciclo valido e generando un impulso di un singolo *clock* sulla linea `xbus_stb_o`. Da questo istante, il master rimane in attesa: la transazione si considera conclusa unicamente quando la periferica target (in questo caso, il nostro bridge) risponde asserendo `xbus_ack_i`. Al ciclo di *clock* immediatamente successivo alla ricezione dell'Acknowledge, la CPU abbassa `xbus_cyc_o`, terminando l'operazione. Qualora la periferica non risponda entro un tempo prestabilito, il *watchdog* interno del bus genera un errore alzando `xbus_err_i`, abortendo la transazione.

L'operazione di lettura avviene in maniera del tutto speculare: il master mantiene `xbus_we_o` basso e attende l'asserzione di `xbus_ack_i`. Al sopraggiungere di tale segnale, la CPU campiona in sicurezza il dato presente sul bus d'ingresso `xbus_dat_i`.

3.2.2 L'interfaccia legacy del core 6502

Per poter pilotare correttamente le periferiche (come l'UART o i registri ECG) ereditate dal progetto preesistente, è necessario comprendere l'interfaccia del microprocessore che esse si aspettano di incontrare. Nel design originale, il controllo era affidato a un *soft-core* VHDL che emulava il set di istruzioni e il comportamento del MOS 6502.

È importante sottolineare un aspetto fondamentale dell'integrazione hardware: a differenza del chip di silicio originale degli anni '70 — che impiegava un bus dati bidirezionale e un complesso schema di temporizzazione asincrono basato su un clock bifase — il *soft-core* utilizzato per la FPGA adotta un approccio interamente sincrono e linearizzato, ottimizzato per le moderne architetture RTL.

Come visibile nello schema a blocchi in Figura 3.2, l'interfaccia di comunicazione esposta dal core 6502 si presenta come un tipico bus sincrono a 8-bit, notevolmente più snello del suo omologo RISC-V a 32-bit:

- `a_o (15:0)`: Bus degli indirizzi a 16-bit, in grado di indirizzare uno spazio di memoria di 64 KB, tipico delle architetture a 8-bit del passato.
- `d_i (7:0)` e `d_o (7:0)`: Bus dati divisi in lettura e scrittura a 8-bit, che sostituiscono il bus bidirezionale del chip fisico per evitare conflitti logici (tri-state) incompatibili con le architetture interne di routing delle FPGA.
- `rd_o` e `wr_o`: Semplici segnali logici attivi alti che comandano, rispettivamente, la lettura e la scrittura verso le periferiche mappate in memoria.

L'assenza di un segnale di *Acknowledge* esplicito in questa architettura legacy denota la sua natura deterministica: la CPU assume che il dato sia stato scritto, o sia pronto per essere letto, in tempi predefiniti senza attendere un handshaking di

conferma dalla periferica. Questo divario architetturale (protocollo asincrono con handshaking sull'XBUS contro protocollo sincrono cieco sul 6502) è il fulcro del problema di temporizzazione che il bridge dovrà risolvere per garantire l'integrità dei dati.

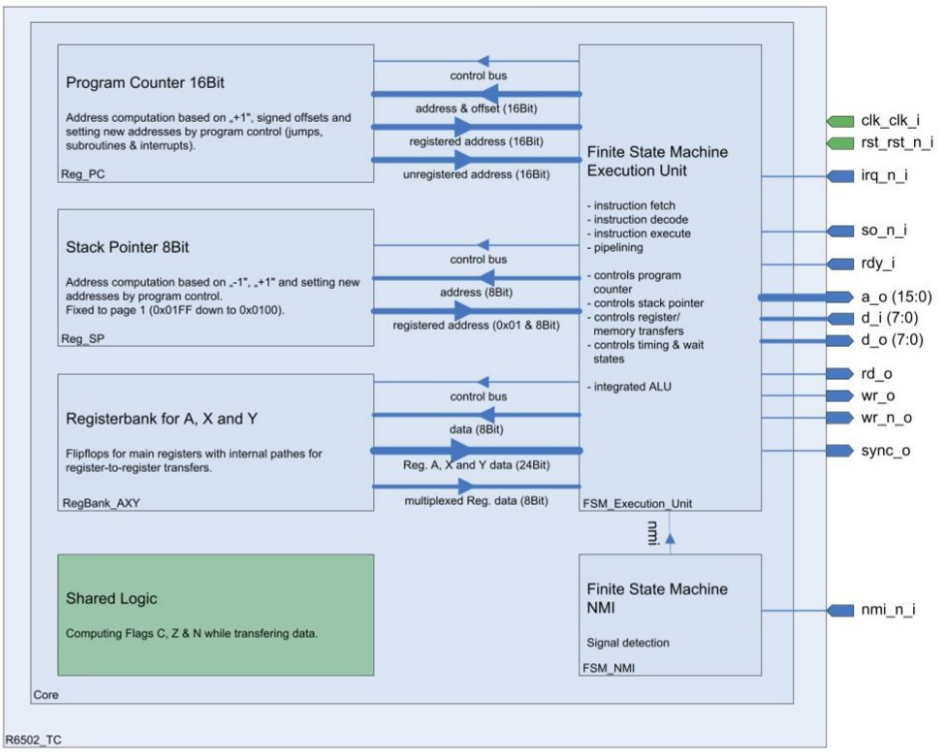

Figura 3.2: Architettura e segnali di interfaccia del *soft-core* 6502 utilizzato nel progetto base. Riprodotta da: `https://opencores.org/websvn/filedetails?repname=cpu6502_true_cycle&path=%2Fcpu6502_true_cycle%2Ftrunk%2Fdoc%2F6502+IP+Core+Specification_V0_7.pdf&rev=0` secondo la licensa BSD-3-Clause.

3.3 Implementazione architetturale del Bridge in VHDL

Il cuore del presente capitolo è costituito dalla progettazione del modulo VHDL `Bridge.vhd`, che funge da interfaccia traduttrice tra il bus a 32-bit (XBUS) del processore RISC-V NEORV32 e le periferiche a 8-bit del sistema legacy, originariamente progettate per dialogare con un core 6502.

3.3.1 Definizione dell'Entity e routing dei segnali

La definizione delle porte (*Port*) del componente non è stata raggruppata per tipologia di bus, ma è stata strutturata in base alla direzione del flusso dati e di controllo rispetto all'unità Bridge stessa, creando quattro domini funzionali distinti:

```
entity Bridge is
    Port (
        clk             : in  std_logic;
        rst             : in  std_logic;
-------------------- INPUT dal RISC-V -----------------------------
        r_w_adr         : in STD_ULOGIC_VECTOR (31 downto 0);
        strobe          : in STD_LOGIC;
        write_en        : in STD_LOGIC;
        byte_en         : in STD_ULOGIC_VECTOR (3 downto 0);
        data_in_write   : in STD_ULOGIC_VECTOR (31 downto 0);
------------------- INPUT dalle Periferiche------------------------
        data_in_read    : in STD_LOGIC_VECTOR (7 downto 0);
-------------------- OUTPUT verso logica 6502 --------------------
        rd_o            : out STD_LOGIC;
        wr_o            : out STD_LOGIC;
        adr_o           : out STD_LOGIC_VECTOR (15 downto 0);
-------------------- OUTPUT verso RISC-V--------------------------
        data_out_r      : out STD_ULOGIC_VECTOR (31 downto 0);
        ack             : out STD_LOGIC;
        err             : out STD_LOGIC;
-------------------- OUTPUT verso Periferiche --------------------
        data_out_w      : out STD_LOGIC_VECTOR (7 downto 0)
    );
end Bridge;
```

Listing 3.1: Dichiarazione dell'Entity del Bridge e porte di I/O.

- **Input dal RISC-V**: Contiene i segnali di controllo e l'indirizzo generati dalla CPU (in base alle specifiche XBUS analizzate in precedenza), unitamente al bus dati a 32-bit utilizzato durante le operazioni di scrittura.
- **Input dalle Periferiche**: Composto da un singolo bus a 8-bit che trasporta il dato prelevato dalla periferica bersaglio durante una lettura.
- **Output verso logica 6502**: È il set di segnali di uscita che emula il comportamento del vecchio processore, fornendo all'infrastruttura di interconnessione preesistente (*Crossbar*) gli storici segnali di *Read* (`rd_o`), *Write* (`wr_o`) e l'indirizzo troncato a 16-bit.
- **Output di ritorno**: Include le linee dirette al RISC-V (il bus dati in lettura a 32-bit e i segnali di *handshaking* `ack` ed `err`) e il bus a 8-bit verso le periferiche per trasferire il dato in scrittura.

3.3.2 Architettura della FSM e Logica di Controllo

Il comportamento dinamico del Bridge è governato da una Macchina a Stati Finiti (FSM) di tipo sincrono. La FSM si occupa di gestire il disaccoppiamento temporale e la conversione del parallelismo dei dati tra il bus a 32-bit del NEORV32 e il bus *legacy* a 8-bit del sistema ECG preesistente.

I diversi stati logici necessari a coordinare le transazioni sequenziali di lettura e scrittura sono dichiarati all'interno della sezione dichiarativa dell'*architecture*, come mostrato nel seguente frammento di codice:

```
architecture Behavioral of Bridge is

    -- Definizione degli stati della FSM (Finite State Machine)
    type state_type is (IDLE, REQUEST_READ, WAIT_RESET, WAIT_DATA,
                        SECOND_BYTE, THIRD_BYTE, FOURTH_BYTE,
                        SECOND_READ, THIRD_READ, FOURTH_READ);
    signal current_state : state_type := IDLE;

begin

process (clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                current_state <= IDLE;
                rd_o <= '0';
                wr_o <= '0';
                adr_o <= (others => '0');
                data_out_r <= (others => '0');
                ack <= '0';
                data_out_w <= (others => '0');
                err <= '0';
            else
                case current_state is
```

Listing 3.2: Dichiarazione degli stati della FSM e inizializzazione del processo sincrono.

Il blocco descritto implementa un processo sensibile al solo fronte di salita del segnale di clock (`clk`), all'interno del quale è previsto un reset sincrono (`rst`). All'attivazione del reset, la macchina viene forzata nello stato di riposo `IDLE` e tutte le uscite verso il processore e verso le periferiche vengono portate a un livello logico nullo o di riposo, garantendo uno stato iniziale predicibile e privo di commutazioni spurie sulle linee di controllo del bus.

3.3.3 Stato IDLE e Cicli di Scrittura (8-bit e Multi-byte)

In assenza di reset, la FSM analizza le richieste provenienti dal processore valutando lo stato `IDLE`. Questo stato rappresenta il punto di discriminazione principale tra le transazioni di lettura e scrittura e si occupa della validazione dell'indirizzo (*Address Decoding*), come illustrato nel listato 3.3.

```
-------------------------IDLE -------------------------------
                    when IDLE =>
                         if strobe = '1' then
                            if r_w_adr(31 downto 16) = x"F000" then
```

```
                                adr_o <= r_w_adr(15 downto 0);
                                if write_en = '0' then
                                    current_state <= REQUEST_READ;
                                    rd_o <= '1';
                                elsif write_en = '1' then
                                    rd_o <= '0';
                                    wr_o <= '1';
                                    data_out_w(7 downto 0) <= data_in_write(7 downto 0);

                                    if byte_en /= "0001" and byte_en /= "0010" and byte_en
/= "0100" and byte_en /= "1000" then
                                        current_state <= SECOND_BYTE;
                                    else
                                        ack <= '1';
                                        current_state <= WAIT_RESET;
                                    end if;
                                end if;
                            else
                                err <= '1';
                                current_state <= WAIT_RESET;
                            end if;
                        end if;
```

Listing 3.3: Codice VHDL dello stato IDLE per la discriminazione delle transazioni.

La FSM campiona in modo continuo il segnale `strobe` in ingresso dal RISC-V (e non il segnale `cyc`, garantendo l'attivazione solo per un ciclo). Se viene richiesta un'operazione viene eseguito un controllo di validazione critico, noto come *Address Translation & Masking*. L'architettura originale a 8-bit mappava le periferiche su 16 bit. L'architettura RISC-V opera su un *memory space* a 32 bit. Per evitare complessi ricablaggi fisici nell'unità di interconnessione legacy, è stato applicato a livello software un *offset* base pari a `0xF000_0000`. La scelta di tale valore non è stata casuale, ma è strettamente legata all'architettura interna della memoria del processore RISC-V scelto.

Come si evince dalle specifiche tecniche del NEORV32, lo spazio di indirizzamento è rigidamente suddiviso in due regioni principali: i primi 3.75 GB sono dedicati al *Memory Address Space* (accessibile tramite la memoria *cache* della CPU), mentre gli ultimi 256 MB superiori (da `0xF000_0000` a `0xFFFF_FFFF`) costituiscono l'*IO/Peripheral Address Space*, dedicato esclusivamente agli accessi di I/O diretti e non bufferizzati (*uncached*).

Inoltre, il NEORV32 adotta una precisa politica di *routing* interno: qualsiasi tentativo di accesso in lettura o scrittura verso indirizzi non fisicamente popolati da memorie o periferiche interne al chip (definiti in gergo architetturale come *the void*, il vuoto) viene automaticamente reindirizzato verso il *Processor-External Bus Interface* (XBUS).

Per garantire che le transazioni destinate al sistema legacy uscissero effettivamente dai pin del processore e non subissero ritardi dovuti alla memoria *cache*, si è deciso

di mappare le vecchie periferiche all'inizio dello spazio di I/O non bufferizzato. L'hardware esegue quindi un controllo di sicurezza sulla parte alta dell'indirizzo (`r_w_adr(31 downto 16)`):

- Se la maschera coincide, i 16 bit meno significativi vengono inoltrati sul bus `adr_o`
- Se la maschera non coincide, l'operazione viene interpretata come *aliasing* illecito; viene sollevato il flag di errore (`err <= '1'`) e la transazione viene interrotta. in quest'ultimo caso la macchina transita nello stato di riposo forzato `WAIT_RESET`.

A questo punto, la FSM discrimina la natura dell'operazione tramite il segnale `write_en`:

- **Lettura (`write_en = '0'`):** Viene attivata la linea di lettura `rd_o` verso il bus legacy e la FSM si prepara a gestire la latenza di lettura evolvendo nello stato `REQUEST_READ`.
- **Scrittura (`write_en = '1'`):** Viene attivata la linea `wr_o` e il primo byte di dati (`data_in_write(7 downto 0)`) viene esposto su `data_out_w`.

Nel caso della scrittura, la FSM esegue un controllo fondamentale sui segnali di abilitazione dei byte (`byte_en`). Se il firmware del RISC-V richiede un'operazione a singolo byte (corrispondente alle maschere `"0001"`, `"0010"`, `"0100"` o `"1000"`), la transazione si considera conclusa in un solo ciclo: il Bridge asserisce immediatamente il segnale di acknowledge (`ack <= '1'`) e transita verso lo stato di chiusura `WAIT_RESET`. Questa ottimizzazione è resa possibile dal comportamento del core NEORV32, il quale, durante le scritture a singolo byte, replica intrinsecamente il dato su tutte le corsie del proprio bus a 32-bit, permettendo al Bridge di campionare direttamente gli 8 bit inferiori indipendentemente dalla specifica posizione logica del byte.

Qualora la maschera `byte_en` indichi l'esigenza di scrivere una parola multi-byte (a 16 o 32 bit), la FSM inizia una sequenza iterativa evolvendo verso gli stati successivi, descritti nel listato 3.4.

```
-----------------------SECOND BYTE-----------------------------------------
                    when SECOND_BYTE =>
                        adr_o <= std_logic_vector(unsigned(r_w_adr(15 downto 0)) + 1);
                        data_out_w(7 downto 0) <= data_in_write(15 downto 8);
                        if byte_en /= "0011" and byte_en /= "1100" then
                            current_state <= THIRD_BYTE;
                        else
                            ack <= '1';
                            current_state <= WAIT_RESET;
                        end if;

------------------------THIRD BYTE-----------------------------------------
                    when THIRD_BYTE =>
```

```
                        adr_o <= std_logic_vector(unsigned(r_w_adr(15 downto 0)) + 2);
                        data_out_w(7 downto 0) <= data_in_write(23 downto 16);
                        current_state <= FOURTH_BYTE;
-------------------------FOURTH BYTE-----------------------------------------
                    when FOURTH_BYTE =>
                        adr_o <= std_logic_vector(unsigned(r_w_adr(15 downto 0)) + 3);
                        data_out_w(7 downto 0) <= data_in_write(31 downto 24);
                        current_state <= WAIT_RESET;
                        ack <= '1';

------------------------WAIT RESET -----------------------------------------
                    when WAIT_RESET =>
                        current_state <= IDLE;
                        rd_o <= '0';
                        wr_o <= '0';
                        ack <= '0';
                        err <= '0';
```

Listing 3.4: Stati della FSM dedicati alla scrittura sequenziale multi-byte e allo stato di WAIT_RESET.

Durante gli stati `SECOND_BYTE`, `THIRD_BYTE` e `FOURTH_BYTE`, l'indirizzo di uscita `adr_o` viene incrementato progressivamente tramite operazioni di aritmetica sui puntatori (`unsigned`). Ad ogni ciclo di clock, una nuova porzione della parola a 32-bit del RISC-V viene presentata sul bus a 8-bit della periferica. Nello stato `SECOND_BYTE` viene effettuata un'ulteriore verifica su `byte_en`: se l'operazione richiede la scrittura di soli 2 byte (`"0011"` o `"1100"`), il Bridge interrompe la catena sequenziale, attiva l'`ack` e si porta in `WAIT_RESET`. In caso contrario, la macchina prosegue fino al completamento del quarto byte.

Lo stato `WAIT_RESET` funge da barriera di sincronizzazione e riposo forzato: azzera i segnali di controllo (`rd_o`, `wr_o`), disasserisce l'`ack` e riporta la FSM in `IDLE`, garantendo che il processore abbia rimosso la richiesta corrente prima di iniziare una nuova transazione.

3.3.4 Cicli di Lettura e Tecnica di Pipelining degli Indirizzi

La gestione dei cicli di lettura presenta una complessità intrinseca superiore dovuta alla necessità di compensare i tempi di accesso (*access time*) delle periferiche *legacy* e i ritardi di propagazione della logica combinatoria. Per ottimizzare queste latenze, il design introduce un meccanismo di anticipo dell'indirizzo (*address look-ahead*). Tuttavia, tale meccanismo deve tenere conto di un vincolo critico: le operazioni di lettura su alcune periferiche (come i registri FIFO dell'UART) non sono idempotenti. Eseguire letture fittizie causerebbe la perdita irrecuperabile dei dati.

Per questo motivo, l'incremento dell'indirizzo è strettamente condizionato dal segnale `byte_en`, come evidenziato nel codice del Listato 3.5.

```
-----------------------REQUEST_READ--------------------------
when REQUEST_READ =>
```

```
3       current_state <= WAIT_DATA;
4
5       if byte_en = "0001" or byte_en = "0010" or byte_en = "0100" or
        byte_en = "1000" then
6           rd_o <= '0';
7       else
8           adr_o <= std_logic_vector(unsigned(r_w_adr(15 downto 0)) + 1);
9       end if;
10
11  ------------------------WAIT_DATA----------------------------
12  when WAIT_DATA =>
13      if byte_en = "1111" then
14          data_out_r(7 downto 0) <= data_in_read;
15          current_state <= SECOND_READ;
16      elsif byte_en = "0011" or byte_en = "1100" then
17          data_out_r(7 downto 0) <= data_in_read;
18          data_out_r(23 downto 16) <= data_in_read;
19          current_state <= SECOND_READ;
20      else
21          data_out_r(7 downto 0) <= data_in_read;
22          data_out_r(15 downto 8) <= data_in_read;
23          data_out_r(23 downto 16) <= data_in_read;
24          data_out_r(31 downto 24) <= data_in_read;
25          ack <= '1';
26          current_state <= WAIT_RESET;
27      end if;
28
29      if byte_en = "0011" or byte_en = "1100" then
30          rd_o <= '0';
31      elsif byte_en = "1111" then
32          adr_o <= std_logic_vector(unsigned(r_w_adr(15 downto 0)) + 2);
33      else
34          rd_o <= '0';
35          current_state <= WAIT_RESET;
36      end if;
37
38  ------------------------SECOND_READ--------------------------
39  when SECOND_READ =>
40      adr_o <= std_logic_vector(unsigned(r_w_adr(15 downto 0)) + 3);
41      data_out_r(15 downto 8) <= data_in_read;
42
43      if byte_en /= "0011" and byte_en /= "1100" then
44          current_state <= THIRD_READ;
45      else
46          data_out_r(31 downto 24) <= data_in_read;
47          ack <= '1';
48          current_state <= WAIT_RESET;
49      end if;
50
51  ------------------------THIRD_READ---------------------------
52  when THIRD_READ =>
```

```
53      data_out_r(23 downto 16) <= data_in_read;
54      current_state <= FOURTH_READ;
55      rd_o <= '0';
56
57  ------------------------FOURTH_READ--------------------------
58  when FOURTH_READ =>
59      data_out_r(31 downto 24) <= data_in_read;
60      current_state <= WAIT_RESET;
61      ack <= '1';
62  end case;
```

Listing 3.5: Dettaglio degli stati logici dedicati alla lettura, al *mirroring* dei dati e alla prevenzione delle letture distruttive.

Come si osserva dall'evoluzione temporale degli stati, l'indirizzo inviato alle periferiche (`adr_o`) viene aggiornato con due cicli di *clock* di anticipo rispetto all'effettivo campionamento del dato, implementando una struttura di *pipelining* condizionata:

1. Nello stato `IDLE` viene presentato l'indirizzo base `r_w_adr`.
2. Nello stato `REQUEST_READ`, la macchina verifica la natura della richiesta. Se la CPU ha richiesto la lettura di un singolo byte (es. `byte_en = "0001"`), il processo si arresta abbassando il segnale di lettura per evitare accessi distruttivi non necessari. Se invece è richiesta una lettura multipla (es. a 32-bit), il Bridge incrementa proattivamente l'indirizzo a `r_w_adr + 1` mentre la periferica sta ancora decodificando l'indirizzo base.
3. Nello stato `WAIT_DATA`, il Bridge effettua il primo campionamento stabile di `data_in_read` (relativo all'indirizzo richiesto in `IDLE`). Contestualmente, viene eseguito un ulteriore controllo sul `byte_en`: se la transazione prevede l'accesso a tutti e 4 i byte (`1111`), l'indirizzo viene spinto a `r_w_adr + 2` per alimentare la *pipeline*, altrimenti l'operazione viene indirizzata verso il completamento di una lettura a 16 bit.
4. Le fasi successive (`SECOND_READ`, `THIRD_READ` e `FOURTH_READ`) si occupano di raccogliere i byte rimanenti, assemblandoli nel bus d'uscita a 32 bit (`data_out_r`) e gestendo la de-asserzione finale del segnale di lettura.

Questo disallineamento temporale è dovuto al fatto che la memoria è sincrona. Aggiornando l'indirizzo, quindi, esso verrà campionato solo al ciclo successivo ed infine i dati saranno stabili con un ulteriore clock di ritardo.

Nello stato `WAIT_DATA`, la FSM decodifica il segnale `byte_en` per determinare il formato della lettura ed applica una strategia di *data mirroring* (o replica del dato) per ottimizzare la compatibilità software:

- **Lettura a singolo byte:** Se il processore richiede un solo byte, il dato ad 8 bit letto dalla periferica viene duplicato contemporaneamente su tutte e

quattro le corsie del bus del RISC-V (`data_out_r(7 downto 0)`, `(15 downto 8)`, `(23 downto 16)` e `(31 downto 24)`). Questa replica rende ininfluente l'allineamento software lato firmware, permettendo alla CPU di leggere il dato corretto indipendentemente dalla maschera di selezione. La transazione termina immediatamente attivando l'`ack`.

- **Lettura a due byte (Half-Word):** Il Bridge campiona il primo byte e lo specchia sulla prima e sulla terza corsia del bus. Successivamente, nello stato `SECOND_READ`, campiona il secondo byte memorizzandolo nella seconda e nella quarta corsia, completando così l'operazione a 16 bit prima di passare a `WAIT_RESET`.
- **Lettura a quattro byte (Word):** La FSM esegue una lettura sequenziale pura senza specchiature, collezionando un byte alla volta in modo strettamente sequenziale attraverso gli stati `WAIT_DATA`, `SECOND_READ`, `THIRD_READ` e `FOURTH_READ`, ricostruendo l'intera parola a 32-bit all'interno del registro `data_out_r`.

Capitolo 4

Validazione Hardware e Risultati di Simulazione

Al fine di verificare la correttezza funzionale del *Bridge* progettato e di garantirne la perfetta integrazione tra il core NEORV32 e l'architettura periferica *legacy* a 8-bit, è stata eseguita una simulazione all'interno dell'ambiente Xilinx Vivado.

Per stimolare la Macchina a Stati Finiti (FSM) in tutte le sue condizioni operative — ovvero transazioni a singolo byte, a mezza parola (16-bit) e a parola intera (32-bit), sia in modalità di lettura che di scrittura — è stato sviluppato un firmware di test minimale in linguaggio C. Il codice descritto nel listato 4.1 è stato integrato direttamente all'interno della struttura logica dell'FPGA come memoria ROM inizializzata a tempo di sintesi.

```c
#include <stdint.h>

#define NEORV32_GPIO_BASE    (0xFFFC0000U)

static volatile uint8_t    * const fb_page    = (void *) 0xF0000080; // Framebuffer row selector
static volatile uint8_t    * const fb_data    = (void *) 0xF0004000; // Framebuffer row data (8-bit)
static volatile uint16_t   * const fb_data_16 = (void *) 0xF0004000; // Framebuffer row data (16-bit)
static volatile uint32_t   * const fb_data_32 = (void *) 0xF0004000; // Framebuffer row data (32-bit)

int main ()
{
    // --- SEZIONE 1: SCRITTURE  ---
    *fb_page = 0x12;
    fb_data[0x31] = 0x56;
    *(volatile uint16_t *) 0xF0004012 = 0xABCD;
    *(volatile uint32_t *) 0xF0004024 = 0xFEDC4B1A;
    *fb_page = 0x78;
    fb_data[0x93] = 0xBC;
    *fb_page = 0x12;

    // --- SEZIONE 2: LETTURE  ---
    uint8_t x  = fb_data[0x13];
```

```
23      uint16_t y = fb_data_16[0x13];
24      uint32_t z = fb_data_32[0x9];
25
26      // --- SEZIONE 3: VERIFICA FINALE ---
27      fb_data[0x13] = x;
28
29      return 0;
30  }
```

Listing 4.1: Firmware in linguaggio C utilizzato come testbench software per la simulazione del Bridge.

4.1 Analisi della Sequenza degli Eventi Software

Il firmware mappa i puntatori di test all'interno dello spazio di I/O non bufferizzato (`0xF000_xxxx`) concordato in fase di progetto. L'esecuzione del codice all'interno del microprocessore genera una sequenza di eventi sul bus XBUS che il Bridge traduce nei seguenti passaggi hardware:

4.1.1 Fase 1: Validazione dei Cicli di Scrittura

- **Scrittura a 8-bit (`*fb_page = 0x12` e `fb_data[0x31] = 0x56`):** Il processore richiede l'accesso rispettivamente agli indirizzi fisici `0xF000_0080` e `0xF000_4031`. In entrambi i casi, la maschera `byte_en` corrisponde a un singolo byte attivo. La FSM del Bridge intercetta la richiesta nello stato `IDLE`, attiva la linea `wr_o`, inoltra il dato sul bus a 8-bit e, rilevando l'accesso singolo, asserisce immediatamente il segnale `ack` terminando il ciclo tramite lo stato `WAIT_RESET` in due soli colpi di clock.
- **Scrittura a 16-bit (`0xF000_4012`):** Il firmware esegue un *casting* a puntatore a 16-bit scrivendo il valore `0xABCD`. In questo istante, la FSM rileva che la maschera `byte_en` coinvolge più corsie. Il Bridge scrive il primo byte (`0xCD`) all'indirizzo base, per poi transitare nello stato `SECOND_BYTE`, dove incrementa l'indirizzo a `0xF000_4013`, espone il secondo byte (`0xAB`) e solo a questo punto solleva il segnale di `ack`.
- **Scrittura a 32-bit (`0xF000_4024`):** Viene tentata la scrittura della Word completa `0xFEDC4B1A`. Questo comando forza la FSM ad attraversare l'intera catena di stati sequenziali (`IDLE` → `SECOND_BYTE` → `THIRD_BYTE` → `FOURTH_BYTE`), incrementando l'indirizzo `adr_o` da `0x4024` fino a `0x4027` e parzializzando il dato a 32-bit in quattro transazioni successive da 8-bit ciascuna sul bus della periferica.
- **Scrittura a 8-bit su indirizzo dispari (`fb_data[0x93] = 0xBC`):** Dopo il secondo cambio di pagina (`*fb_page = 0x78`), il firmware esegue una scrittura all'indirizzo `0xF000_4093`.

4.1.2 Fase 2: Validazione dei Cicli di Lettura e Aritmetica dei Puntatori

La seconda parte del codice mette in evidenza il comportamento del Bridge durante le letture sequenziali e illustra come l'aritmetica dei puntatori del compilatore influenzi gli indirizzi generati:

- **Lettura a 8-bit (`fb_data[0x13]`):** La CPU richiede il byte all'indirizzo `0xF000_4013`. La FSM passa da `IDLE` a `REQUEST_READ` impostando la linea `rd_o`. Allo stato `WAIT_DATA`, il Bridge acquisisce il byte singolo e attiva il meccanismo di *data mirroring*, replicando il valore letto su tutte e quattro le corsie del bus verso il RISC-V, chiudendo subito la transazione con `ack <= '1'`.
- **Lettura a 16-bit (`fb_data_16[0x13]`):** Essendo `fb_data_16` un puntatore a 16-bit (`uint16_t`), l'indice `0x13` (corrispondente a 19 in decimale) subisce la scalatura automatica del compilatore C, traducendosi in un offset reale di $19 \times 2 = 38$ byte (ovvero `0x26` in esadecimale). L'indirizzo generato sul bus sarà quindi `0xF000_4026`. La FSM esegue due letture consecutive sfruttando gli stati `WAIT_DATA` e `SECOND_READ` per ricostruire la mezza parola.
- **Lettura a 32-bit (`fb_data_32[0x9]`):** Similmente, il puntatore a 32-bit scala l'indice `0x9` di un fattore 4 ($9 \times 4 = 36$ byte, pari a `0x24` esadecimale), puntando all'indirizzo `0xF000_4024`. Per soddisfare la richiesta del processore, la FSM si sposta sequenzialmente fino a `FOURTH_READ`, campionando quattro byte consecutivi dal bus *legacy* prima di rilasciare il segnale di acknowledge.

4.2 Analisi delle Forme d'Onda in Simulazione

Per validare sperimentalmente le considerazioni teoriche, vengono di seguito analizzati i diagrammi temporali ottenuti dalla simulazione comportamentale in Vivado. L'attenzione è focalizzata sulla transazione più complessa gestita dal *Bridge*: il ciclo di scrittura e successiva lettura a 32-bit, legato all'istruzione C:

```
*(volatile uint32_t *) 0xF0004024 = 0xFEDC4B1A;
```

4.2.1 Analisi Temporale del Ciclo di Scrittura a 32-bit

La figura 4.1 mostra l'andamento dei segnali durante l'esecuzione della scrittura della Word `0xFEDC4B1A` a partire dall'indirizzo base `0x4024`.

Dall'analisi del cronogramma si possono identificare chiaramente le seguenti fasi hardware:

- **Inizio della transazione:** Il processore asserisce il segnale `strobe <= '1'` per indicare l'inizio di un'operazione sul bus. Contemporaneamente, la linea `write_en` si porta a `'1'`, notificando al *Bridge* che si tratta di un ciclo di

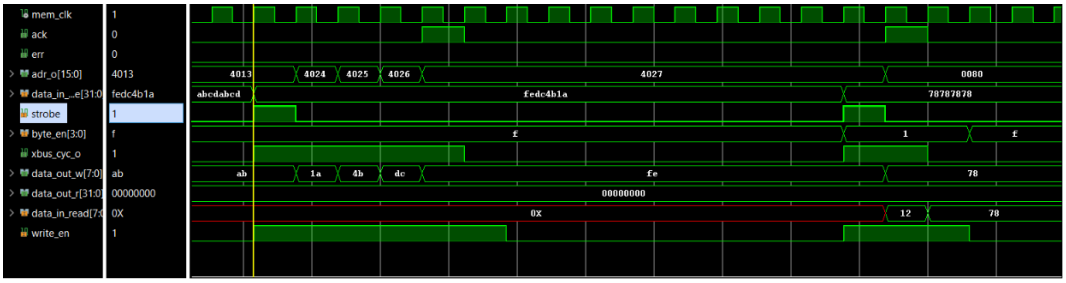

Figura 4.1: Forme d'onda di Vivado relative alla scrittura a 32-bit sul bus della periferica.

scrittura. La maschera dei byte abilitati `byte_en` assume il valore esadecimale `f` (ovvero la stringa binaria `"1111"`), confermando la richiesta di un trasferimento a 32-bit (Word intera).

- **Mantenimento del ciclo:** Il segnale di ciclo del bus (`xbus_cyc_o`) si alza e rimane fisso a `'1'` per tutta la durata dell'operazione multi-byte.
- **Incremento sequenziale dell'indirizzo:** Come previsto dal design della FSM, l'indirizzo sul bus periferico `adr_o[15:0]` parte dal valore base `4024` e si incrementa automaticamente di un'unità a ogni ciclo di clock successivo (`4024` $\rightarrow$ `4025` $\rightarrow$ `4026` $\rightarrow$ `4027`).
- **Scomposizione del dato (Byte Serialization):** Sul bus dati a 8-bit in uscita dal *Bridge* (`data_out_w[7:0]`), la Word a 32-bit viene parzializzata e presentata un singolo byte alla volta, seguendo la codifica *Little-Endian* nativa del RISC-V. Vengono quindi trasmessi in sequenza i valori: `1a` (all'indirizzo `4024`), `4b` (al `4025`), `dc` (al `4026`) e infine `fe` (al `4027`).
- **Chiusura della transazione:** Una volta completato lo sblocco dell'ultimo byte (`fe`), il *Bridge* asserisce il segnale di acknowledge (`ack <= '1'`). Al rilevamento dell'attivazione di `ack`, il master abbassa la linea `xbus_cyc_o`, ponendo fine all'intera transazione hardware in modo corretto.

4.2.2 Analisi Temporale del Ciclo di Lettura e Verifica Coerenza Dati

Per verificare che la transazione precedente sia andata a buon fine e che i dati siano stati memorizzati correttamente nella periferica, il firmware esegue un ciclo di lettura a 32-bit dallo stesso indirizzo base (`0x4024`), visualizzato nella figura 4.2.

Il comportamento rispecchia in modo duale la scrittura, ma introduce un fenomeno di troncamento hardware di estremo interesse progettuale:

- **Dinamica dei segnali di controllo:** Lo `strobe` si attiva nuovamente all'inizio del ciclo, ma in questo caso il segnale `write_en` rimane fisso a `'0'`, configurando l'operazione come una lettura. Anche in questo caso l'indirizzo `adr_o[15:0]` incrementa sequenzialmente da `4024` a `4027` per ricostruire i 4 byte della parola.

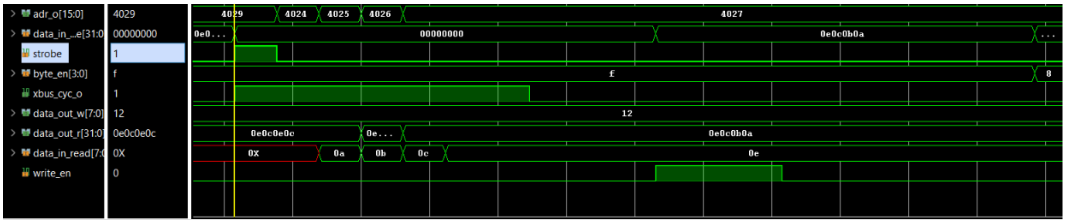

Figura 4.2: Forme d'onda di Vivado relative alla lettura a 32-bit per la verifica della coerenza dei dati.

- **Lettura e Troncamento sul Framebuffer:** Sul bus di ingresso dal lato periferica (`data_in_read[7:0]`), si osserva la risposta dei singoli byte letti. Tuttavia, come definito dalle specifiche del sistema, la memoria del framebuffer mappa ogni pixel su un singolo byte di cui però solo i 4 bit inferiori sono fisicamente implementati per pilotare i segnali colore (`0000IRGB`), mentre il *nibble* superiore è strutturalmente nullo. Di conseguenza, i dati precedentemente scritti subiscono un troncamento hardware dei 4 bit più significativi durante il campionamento:
  - Il byte `0x1A` scritto in `4024` viene riletto come `0x0A`.
  - Il byte `0x4B` scritto in `4025` viene riletto come `0x0B`.
  - Il byte `0xDC` scritto in `4026` viene riletto come `0x0C`.
  - Il byte `0xFE` scritto in `4027` viene riletto come `0x0E`.
- **Ricostruzione sul bus Master:** Il *Bridge* colleziona correttamente questi contributi sequenziali a 8-bit e, allo scadere dei quattro cicli di lettura, ricompatta la parola sul bus a 32-bit diretto al processore (`data_out_r[31:0]`), presentando il valore finale `0x0E0C0B0A`.

Il perfetto allineamento tra i valori teorici attesi (corrispondenti alla maschera di bit `0x0F0F0F0F` applicata al dato originale) e i vettori esadecimali effettivamente registrati sul bus in simulazione attesta la piena validità funzionale della FSM del *Bridge* e la sua tolleranza ai vincoli fisici delle periferiche connesse.

Capitolo 5

Integrazione dell'Interfaccia di Debug JTAG On-Chip

Al fine di verificare a fondo il corretto interfacciamento tra il processore NEORV32, il modulo Bridge sviluppato e le periferiche del sistema legacy, si è resa necessaria l'implementazione di un'interfaccia di *debugging* hardware direttamente su FPGA. Il NEORV32 supporta il *debug* on-chip tramite il protocollo standard JTAG, demandando la gestione delle transazioni a un modulo dedicato chiamato DTM (*Debug Transport Module*). Questo modulo ha il compito di tradurre i segnali seriali del bus JTAG in transazioni parallele compatibili con la DMI (*Debug Module Interface*) interna del processore.

5.1 Limitazioni fisiche e utilizzo delle primitive BSCANE2

Nelle moderne schede di sviluppo basate su FPGA Xilinx (come l'architettura Serie 7 impiegata in questo progetto), i pin fisici associati all'interfaccia JTAG sono instradati direttamente a livello di silicio verso il chip programmatore integrato sulla board. Questo vincolo hardware impedisce il collegamento diretto dei segnali JTAG del processore *soft-core* ai pin esterni della scheda.

Per aggirare tale limitazione e ottenere l'accesso al bus JTAG interno, è necessario ricorrere all'istanziazione di specifiche primitive hardware messe a disposizione dalla libreria Xilinx, denominate `BSCANE2`. Tali primitive permettono di intercettare il traffico JTAG proveniente dall'interfaccia di programmazione e reindirizzarlo verso la logica interna dell'FPGA. Nello specifico del *debugging* RISC-V, si rende necessario l'utilizzo di due primitive `BSCANE2` distinte, configurate su istruzioni USER differenti: una dedicata all'accesso e al controllo del registro di stato (*Control and Status*), e l'altra dedicata al trasferimento effettivo dei dati di *debug*. L'integrazione di queste primitive richiede inevitabilmente una profonda modifica architetturale del modulo DTM originale fornito dal progetto NEORV32.

5.2 Analisi del Debug Transport Module (DTM) originale

Prima di procedere con le modifiche per l'integrazione delle primitive Xilinx, è fondamentale analizzare l'architettura logica del modulo `neorv32_debug_dtm` nella sua versione originale.

L'entità nativa si presenta come un componente puramente RTL, indipendente dalla piattaforma FPGA, che accetta in ingresso i quattro segnali classici del bus JTAG (`TCK`, `TDI`, `TDO`, `TMS`) e restituisce i segnali `dmi_req_o` e `dmi_rsp_i`. L'architettura interna è divisa in tre stadi funzionali principali:

- **Sincronizzazione dei segnali**:

  Poiché l'interfaccia JTAG è pilotata da un segnale di clock esterno dedicato (`TCK`), asincrono rispetto al clock principale del processore, l'architettura prevede un primo stadio di sincronizzazione. I segnali in ingresso (`TCK`, `TMS`, `TDI`) vengono campionati mitigando il rischio di metastabilità ed estraendo in modo sincrono i fronti di salita e discesa del clock JTAG per orchestrare la logica successiva.

  Il segnale JTAG in ingresso (`jtag_tck_i`), quindi, non viene instradato sulla rete di *clock* dell'FPGA, ma viene trattato alla stregua di un normale segnale dati asincrono. Per poter rilevare i fronti del TCK, il design originale impiega una catena di sincronizzazione a tre stadi, come visibile nel seguente estratto di codice:

```
1  -- JTAG Input Synchronizer --
2  tap_synchronizer: process(rstn_i, clk_i)
3  begin
4    if (rstn_i = '0') then
5      tck_ff <= (others => '0');
6      -- [Omissis: reset di tdi_ff e tms_ff]
7    elsif rising_edge(clk_i) then
8      tck_ff <= tck_ff(1 downto 0) & jtag_tck_i;
9      -- [Omissis: shift di tdi_ff e tms_ff]
10   end if;
11 end process tap_synchronizer;
12
13 -- JTAG clock edges --
14 tck_rise <= '1' when (tck_ff(2 downto 1) = "01") else '0';
15 tck_fall <= '1' when (tck_ff(2 downto 1) = "10") else '0';
```

Listing 5.1: Circuito di sincronizzazione del segnale TCK nel DTM originale.

  Il vettore `tck_ff` funge da *shift register* a 3 bit: il primo stadio (*flip-flop*) campiona il segnale asincrono, il secondo mitiga eventuali stati di metastabilità sincronizzandolo col dominio `clk_i`, e il terzo permette al blocco logico combinatorio di rilevarne le transizioni (`tck_rise` e `tck_fall`).

  Affinché questa topologia di campionamento funzioni correttamente, il periodo del *clock* di sistema deve essere sufficientemente breve da garantire l'intercetta-

mento di entrambi i fronti (alto e basso) del segnale TCK. Da questo principio deriva una nota regola empirica del progetto digitale: la frequenza del JTAG deve essere rigorosamente inferiore a un quinto della frequenza della CPU ($f_{TCK} < f_{CPU}/5$). Nella realtà fisica, a causa della latenza di instradamento (*routing*), del *jitter* intrinseco del TCK e della relazione di fase casuale tra i due domini, questo rapporto teorico spesso non garantisce la totale immunità da errori di campionamento, costringendo il progettista a ridurre ulteriormente la velocità del *debugger*.

Questa dipendenza gerarchica introduce un problema insormontabile quando si implementano logiche di gestione energetica dinamica (*Low Power*). Si consideri uno scenario operativo in cui la CPU opera a una frequenza nominale di 32.5 MHz; in tali condizioni, l'interfaccia JTAG può lavorare stabilmente a 5 MHz. Tuttavia, qualora il processore decidesse di abbattere dinamicamente la propria frequenza (es. scendendo a 1 MHz per limitare i consumi), il limite teorico di campionamento per il TCK crollerebbe a circa 200 kHz.

Se il *debugger* software host (come OpenOCD) non fosse informato in tempo reale di questo declassamento e continuasse a trasmettere a 5 MHz, il DTM non sarebbe più fisicamente in grado di campionare i fronti del TCK. Il risultato diretto di questo disallineamento è la perdita di pacchetti JTAG, la corruzione dei registri di stato e l'inevitabile disconnessione irreversibile della sessione di *debug*.

- **TAP Controller FSM**: Il controllo del flusso è affidato a un TAP (Test Access Port) Controller, realizzato tramite una Macchina a Stati Finiti (FSM) a 16 stati. Il TAP gestisce la transizione tra le fasi di configurazione dei comandi (ramo IR) e quelle di trasferimento dati (ramo DR). I comandi in ingresso vengono serializzati all'interno dell'Instruction Register (`ireg`, a 5 bit). L'architettura ottimizza l'implementazione dello stato di `Update-IR` (l'aggiornamento del comando) gestendolo in modo passivo: non essendo presente uno *shadow register* dedicato, il semplice arresto dello scorrimento alla fine dello stato di `IR_SHIFT` assicura che il dato in `ireg` rimanga stabile (*latched*). Il comando così trattenuto agisce da selettore per un multiplexer interno, stabilendo quale registro dati instradare (es. `IDCODE` per l'identificazione, `DTMCS` per il controllo del modulo, o `DMI` per l'accesso al processore).

- **Il Data Register: Ottimizzazione MSB-Aligned e Trasmissione Seriale**: Per minimizzare l'utilizzo di area logica (silicio/LUT), i molteplici registri dati previsti dallo standard JTAG non sono implementati come entità separate. Il DTM istanzia un unico registro a scorrimento da 41 bit (`dreg`). Durante lo stato `DR_CAPTURE`, il sistema acquisisce lo stato interno coerentemente con l'istruzione attiva, posizionando il carico utile (*payload*) sempre in allineamento

al Most Significant Bit (*MSB-aligned*). I bit non necessari per le istruzioni più corte (come l'`IDCODE` a 32 bit) vengono posti a zero.

Nello stato `DR_SHIFT`, i bit traslano verso destra ad ogni fronte di salita del clock. Coerentemente con lo standard IEEE 1149.1, il dato in uscita sul pin `TDO` transisce esclusivamente sul fronte di discesa. Questo è ottenuto sfruttando un multiplexer combinatorio sul pin d'uscita: invece di utilizzare contatori complessi, il sistema calcola staticamente l'indice del Logical Least Significant Bit (LSB) in base all'istruzione corrente e vi connette fisicamente l'uscita, intercettando il flusso seriale man mano che i bit scorrono all'interno del registro.

- **L'Interfaccia DMI e la Transazione (Handshake)**: L'interazione diretta con il processore RISC-V si concretizza nello stato di `Update-DR`, qualora l'istruzione attiva sia indirizzata al DMI (`ireg` = "10001"). Un impulso dedicato attiva il `dmi_controller`, un processo operante interamente nel dominio di clock del sistema. In questa fase, il *payload* di 41 bit prelevato dal `dreg` viene deserializzato per estrarre parallelamente l'indirizzo a 7 bit, il dato a 32 bit e il codice operativo a 2 bit.

  Per garantire la coerenza dei dati e prevenire condizioni di *race*, la transazione è regolata da un rigido protocollo di *handshake*. Il controller DMI inoltra la richiesta parallela al Debug Module (DM) del processore e asserisce immediatamente un segnale di interfaccia occupata (`busy`). Qualora il controller JTAG tenti di inviare un nuovo comando mentre la risorsa è `busy`, il sistema scarta l'istruzione e solleva un'eccezione hardware persistente (*sticky error*), che richiederà un reset esplicito da parte del *debugger*.

  Quando il processore termina l'operazione richiesta, risponde asserendo un segnale di *acknowledge* (`ack`). A questo punto, il DTM campiona l'eventuale dato in lettura fornito dalla CPU, de-asserisce il segnale di `busy` e si rende disponibile per il successivo ciclo di cattura e trasmissione dati verso l'host.

5.3 Riprogettazione del DTM

Per superare i limiti architetturali e i vincoli legati alla temporizzazione precedentemente descritti, il modulo DTM originale è stato profondamente modificato operando su due fronti principali: l'astrazione dell'accesso fisico al bus JTAG tramite le primitive hardware della libreria Xilinx e la totale separazione dei domini di *clock*.

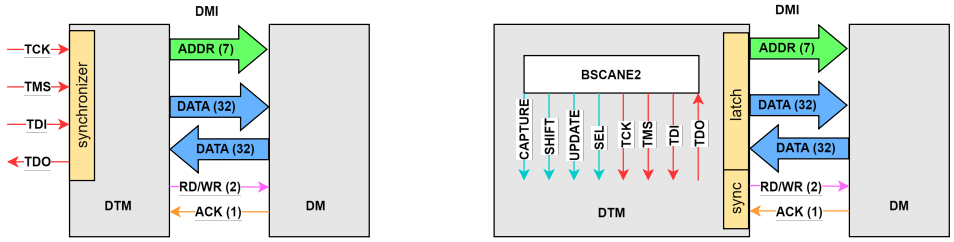

Figura 5.1: Confronto architetturale: a sinistra il DTM originale, a destra il DTM modificato con primitive BSCANE2, *latch* e logica di sincronizzazione.

5.3.1 Sostituzione della TAP FSM con la primitiva BSCANE2

Come analizzato in precedenza, il modulo originale implementava al suo interno un intero controller TAP (Test Access Port) sotto forma di Macchina a Stati Finiti in VHDL, il quale doveva decodificare manualmente i segnali seriali. Nella soluzione proposta, questa logica software viene completamente rimossa e sostituita dall'istanziazione diretta delle primitive `BSCANE2`, fornite nativamente dalla libreria `UNISIM` di Xilinx.

Queste primitive fungono da "sonde" (hard-macro) capaci di intercettare il traffico di *debug* all'interno dell'infrastruttura di *routing* fisica dell'FPGA (la quale è cablata on-silicon al chip programmatore FTDI della *board*). Invece di dover decodificare lo stato della comunicazione analizzando i fronti di `TCK` e i valori di `TMS`, la primitiva `BSCANE2` decodifica il protocollo a monte e restituisce direttamente al DTM le informazioni di stato sotto forma di segnali logici puliti.

```
BSCANE2_inst_1 : BSCANE2
generic map (
    JTAG_CHAIN => 1
)
port map (
    CAPTURE => bscane_capture,
    SHIFT   => bscane_shift,
    UPDATE  => bscane_update,
    TCK     => bscane_tck,
    TDI     => bscane_tdi,
    SEL     => bscane_sel_dmi,
    TDO     => tdo_dmi
);
```

Listing 5.2: Istanziazione della primitiva BSCANE2 per la catena DMI.

Come si evince dal Listato 5.2, il DTM non deve più calcolare in quale stato della comunicazione si trova il debugger: i segnali `CAPTURE`, `SHIFT` e `UPDATE` vengono asseriti direttamente dalla primitiva quando la catena JTAG selezionata (`SEL`) entra nella rispettiva fase. Di conseguenza, tutti i blocchi logici `when` (precedentemente legati allo stato calcolato dalla FSM) sono stati snelliti e sostituiti da semplici costrutti `if` basati su questi segnali già decodificati.

5.3.2 Separazione dei domini di Clock

La seconda e più significativa modifica riguarda la temporizzazione. A differenza della versione originale interamente guidata dalla frequenza della CPU, la nuova implementazione introduce formalmente due domini di *clock* distinti e disaccoppiati:

- **Dominio JTAG**: Sincronizzato in modo nativo sul *clock* seriale in ingresso `bscane_tck`. In questo dominio opera tutta la logica di accesso e scorrimento dei registri JTAG (`SHIFT` e `UPDATE`).
- **Dominio CPU**: Sincronizzato sul *clock* di sistema `clk_i`, nel quale opera esclusivamente l'interfaccia verso il *Debug Module* interno del RISC-V.

Questa separazione elimina la necessità di campionare i fronti del TCK con un *clock* ad altissima frequenza, risolvendo definitivamente la vulnerabilità negli stati di *low-power* della CPU. Tuttavia, la presenza di due domini asincroni richiede l'implementazione di un meccanismo sicuro (*Clock Domain Crossing*) per lo scambio di dati.

5.3.3 Protocollo di Handshake a quattro fasi (*Four-Phase Handshake*)

La comunicazione tra il dominio JTAG (lento e asincrono) e il dominio CPU (veloce) viene gestita tramite un robusto protocollo di *four-phase handshake*. A differenza dell'interfaccia DMI standard del NEORV32, che predilige impulsi temporanei di richiesta e conferma (`request/ack`), questo protocollo implementa due segnali *di livello* che mantengono il proprio stato logico fintanto che la transazione non viene riconosciuta dall'altro dominio:

- `busy`: generato nel dominio JTAG per segnalare una transazione in corso.
- `rsp.ack`: generato nel dominio CPU per segnalare l'avvenuta elaborazione.

L'utilizzo di segnali di livello prolungati previene in modo assoluto la perdita di informazioni (*pulse skipping*) durante l'attraversamento del confine asincrono.

Il protocollo si articola in quattro fasi sequenziali:

1. **Richiesta**: Il dominio JTAG genera una nuova richiesta ponendo il livello logico `busy = 1`.
2. **Elaborazione e Conferma**: Il dominio CPU rileva e aggancia la richiesta; al termine dell'operazione, segnala il completamento ponendo `rsp.ack = 1`.
3. **Rilascio della Richiesta**: Il dominio JTAG rileva la conferma dal lato CPU e, in risposta, abbassa il segnale di occupazione (`busy = 0`).
4. **Ripristino**: Il dominio CPU intercetta la discesa del `busy` e ripristina la propria linea di acknowledge (`rsp.ack = 0`), riportando il sistema allo stato iniziale di riposo per la transazione successiva.

5.3.4 Analisi del trasferimento DMI: Dal JTAG alla CPU

Dal punto di vista dell'implementazione RTL, quando il debugger remoto completa una transazione ed entra nella fase `UPDATE`, il dominio JTAG memorizza i campi operativi (`addr`, `data`, `op`) e asserisce il segnale `busy`.

```
dmi.addr <= dreg(40 downto 34);
dmi.data <= dreg(33 downto 2);
dmi.op   <= dreg(1 downto 0);

busy <= or dreg(1 downto 0);
```

Listing 5.3: Salvataggio dei dati DMI e asserzione di livello del busy.

Il segnale `busy` non costituisce un impulso ma un livello logico che rimane attivo per tutta la durata della transazione. Tale caratteristica consente al dominio CPU di rilevarne correttamente la presenza indipendentemente dal rapporto tra le due frequenze di clock.

Il trasferimento nel dominio CPU avviene mediante una classica catena di flip-flop sincronizzatori.

```
syncstage_busy <= busy & syncstage_busy(syncstage_busy'left downto 1);
```

Listing 5.4: Catena di sincronizzazione nel dominio CPU.

Una volta sincronizzato e privo di metastabilità, il dominio della CPU ha la necessità di ritradurre questo livello logico in un impulso della durata esatta di un colpo di *clock* Questo avviene tramite un classico rilevatore di fronte di salita (*edge detector*). L'impulso così ottenuto viene infine utilizzato dal processo `dmi_controller` per generare la richiesta DMI verso il Debug Module.

```
synchronized_busy_rise <= '1' when syncstage_busy(1 downto 0) = "10"
    else '0';

if synchronized_busy_rise then
    dmi_req_o <= dmi;
else
    dmi_req_o.op <= dmi_req_nop_c;
end if;
```

Listing 5.5: Rilevamento del fronte e ricostruzione dell'impulso di richiesta DMI.

Si osserva quindi come il segnale di livello `busy` venga trasformato in un impulso solamente dopo aver attraversato in sicurezza il confine tra i due domini di clock.

5.3.5 Analisi del trasferimento DMI: Dalla CPU al JTAG

In modo del tutto speculare, quando il *Debug Module* ha terminato l'esecuzione dell'istruzione, emette un impulso `dmi_rsp_i.ack`. Affinché questo non vada perso, il dominio CPU lo intrappola immediatamente nel registro locale `rsp`, convertendolo di fatto in un segnale di livello prolungato.

```
elsif dmi_rsp_i.ack then
    rsp <= dmi_rsp_i;
end if;
```

Listing 5.6: Cattura dell'impulso di acknowledge e conversione in segnale di livello.

Il segnale `rsp.ack` viene quindi inviato indietro verso il dominio JTAG, attraversando un'analoga catena di sincronizzazione `syncstage_ack`. Ottenuta la conferma di ricezione (`synchronized_ack`), il lato JTAG conclude la transazione (Fase 3) abbassando il proprio flag:

```
if synchronized_ack then
    busy <= '0';
end if;
```

Listing 5.7: Fase 3: Rilascio del segnale busy da parte del dominio JTAG.

Infine il dominio CPU rileva il fronte di discesa di `busy` e provvede ad azzerare il segnale `rsp.ack`, completando il protocollo di handshake.

```
if synchronized_busy_fall then
    rsp.ack <= '0';
end if;
```

In questo modo il protocollo termina riportando entrambi i domini nello stato di riposo, rendendo possibile l'avvio della successiva operazione di debug senza alcuna dipendenza dal rapporto tra la frequenza del clock JTAG e quella della CPU.

5.4 Validazione funzionale tramite Debugger (GDB)

Al fine di validare le profonde modifiche architetturali apportate al modulo DTM e verificare la corretta integrazione del Bridge, si è proceduto con una sessione di collaudo *on-target*. L'ambiente di test è stato configurato collegando il debugger standard GDB (GNU Debugger) al processore NEORV32 tramite il server OpenOCD, sfruttando la nuova interfaccia JTAG basata sulle primitive `BSCANE2`.

La schermata del debugger, operante in modalità TUI (*Text User Interface*), è suddivisa in tre sezioni principali che permettono un'analisi a diversi livelli di astrazione:

- **Pannello superiore**: Mostra il codice sorgente C in esecuzione, evidenziando le operazioni di scrittura e lettura puntuale in memoria.
- **Pannello centrale**: Mostra il codice disassemblato (linguaggio macchina RISC-V), evidenziando l'esatta istruzione *assembly* attualmente puntata dal Program Counter (nel caso specifico, un'istruzione di caricamento in memoria `lw`).
- **Pannello inferiore**: Contiene la console di comando interattiva di GDB, tramite la quale vengono inviate le richieste di ispezione.

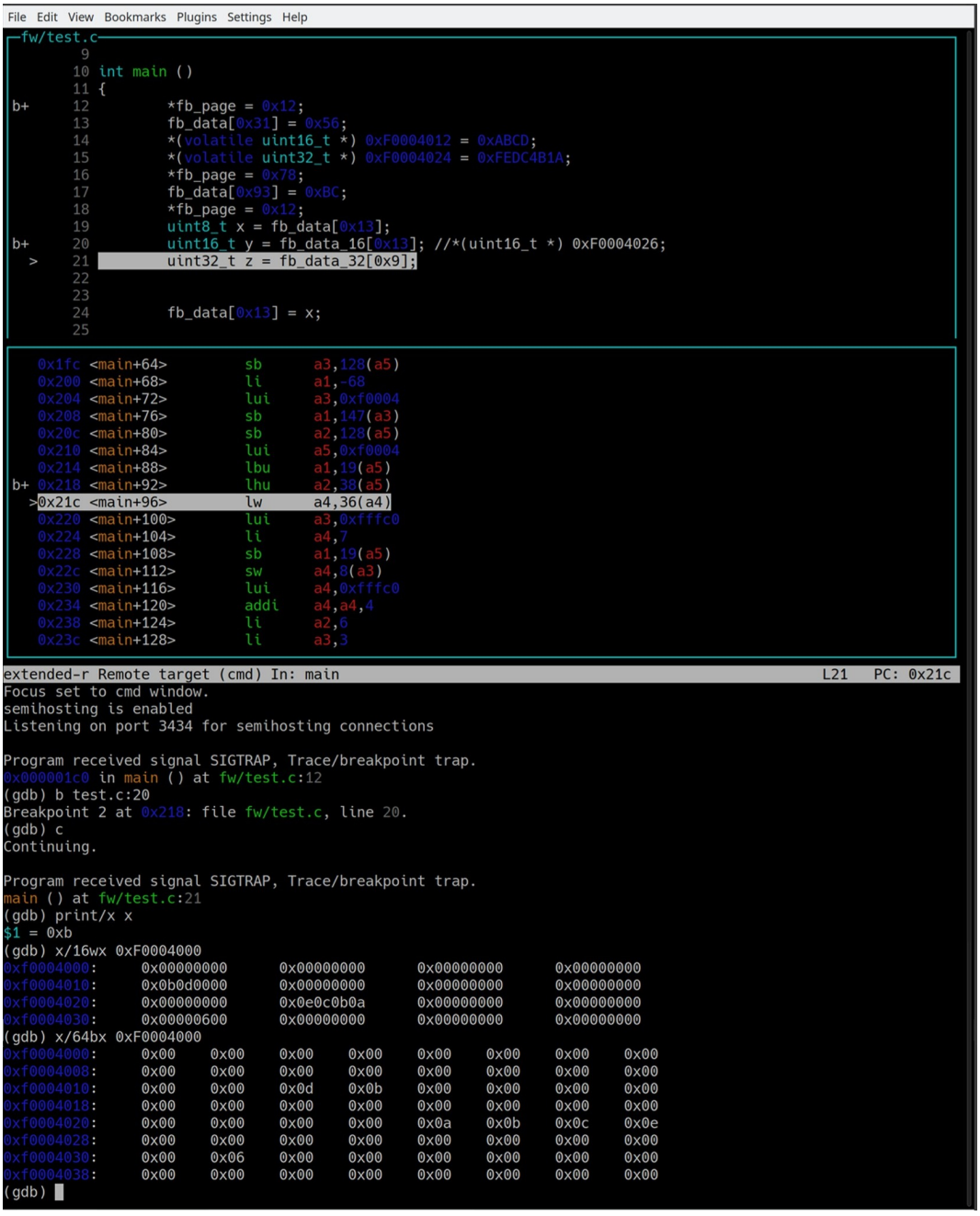

Figura 5.2: Interfaccia GDB durante la sessione di test: ispezione del codice C, disassemblato assembly e comandi di verifica della memoria tramite il Bridge.

5.4.1 Test ad alto livello: Breakpoint e ispezione delle variabili

La prima fase del test ha dimostrato il corretto funzionamento del meccanismo di *halting* e ispezione dei registri. Dalla riga di comando è stata inviata la richiesta di inserimento di un *breakpoint* hardware alla riga 20 del firmware (`b test.c:20`). Come si evince dal log, l'inserimento va a buon fine e, impartendo il comando di

continuazione (`c`), il processore riprende l'esecuzione per poi arrestarsi esattamente al *breakpoint* stabilito, sollevando il segnale `SIGTRAP`.

Immediatamente dopo, è stato richiesto al debugger di stampare il valore della variabile locale `x` in formato esadecimale (`print/x x`). Il sistema ha restituito il valore `$1 = 0xb` (corrispondente al valore decimale 11). Questo risultato conferma che la riga precedente (`uint8_t x = fb_data[0x13];`) è stata eseguita correttamente e che il DTM è in grado di interrogare i registri interni della CPU senza incorrere nei problemi di desincronizzazione analizzati nei capitoli precedenti.

5.4.2 Test a basso livello: Ispezione della memoria e validazione del Bridge

La seconda fase della simulazione ha avuto un duplice scopo: testare l'accesso alla memoria da parte del debugger e validare il funzionamento del Bridge VHDL, dimostrando la sua capacità di gestire accessi di larghezza variabile.

Tramite GDB, sono state effettuate due letture dirette nell'area di memoria mappata per le periferiche esterne (partendo dall'indirizzo base `0xF000_4000`), zona precedentemente popolata dalle istruzioni del firmware C:

1. **Lettura a 32-bit (Word)**: Tramite il comando `x/16wx 0xF0004000`, è stato richiesto al sistema di estrarre 16 *word* (blocchi da 32 bit). Il risultato a schermo mostra l'aggregazione dei dati a 4 byte per volta, restituendo correttamente i pattern inseriti dal firmware.

2. **Lettura a 8-bit (Byte)**: Tramite il comando `x/64bx 0xF0004000`, è stata eseguita la lettura della medesima porzione di memoria, ma richiedendo 64 singoli byte. In questo caso, il log mostra la scomposizione esatta e progressiva dei dati (es. `0x0a`, `0x0b`, `0x0c`, `0x0e`).

L'esito positivo di queste due operazioni di lettura dimostra inequivocabilmente che:

- Il DTM modificato riesce a orchestrare trasferimenti complessi sul bus *uncached* del RISC-V senza perdita di dati.

- Il Bridge intercetta correttamente le richieste sull'indirizzo con *offset* `0xF000`, gestendo in modo trasparente i segnali di *Byte Enable* (`byte_en`) per assecondare sia le richieste a singola *word* a 32-bit che quelle frammentate a 8-bit, coerentemente con le capacità del bus *legacy*.

Capitolo 6

Validazione del Sistema Integrato: Esecuzione del Firmware ECG

L'ultimo capitolo di questo lavoro di tesi è dedicato al test finale dell'intero sistema integrato. L'obiettivo ultimo della migrazione architetturale era permettere l'esecuzione del firmware applicativo originale — originariamente progettato per l'architettura a 8 bit del processore MOS 6502 — sul nuovo processore RISC-V a 32 bit, interfacciandosi con successo alle periferiche preesistenti.

L'output visivo viene gestito da un modulo dedicato che, tramite un convertitore digitale-analogico (DAC) esterno, trasmette il segnale video a un monitor VGA.

6.1 Analisi della logica di acquisizione e tracciamento

Il funzionamento del firmware si basa su un'architettura *event-driven*, dove l'acquisizione dei dati è gestita tramite interruzioni. Di seguito viene riportata la Routine di Servizio dell'Interruzione (ISR) associata all'ADC e il ciclo principale (*main loop*) del programma:

```
// Interrupt Service Routine (ISR)
{
    *led ^= 0x20;          // Toggle LED di debug
    adc_value = *adc;      // Lettura del convertitore
    adc_data_ready = true; // Set del flag di dato pronto
}

// Ciclo Main
while (1) {
    // Attesa bloccante (idle)
    while (!uart->control.rx_ready && !adc_data_ready) ;

    if (adc_data_ready) {
        adc_data_ready = false;
        process_sample(adc_value);
    }
    // ... gestione UART
}
```

Listing 6.1: Routine di interruzione e ciclo principale del firmware.

Come si evince dal Listato 6.1, l'ISR ha il compito di campionare il valore dal convertitore, far lampeggiare un LED diagnostico per confermare visivamente l'avvenuta interruzione e asserire il flag `adc_data_ready`. Nel ciclo `main`, il programma rimane in una fase di attesa (fino all'arrivo di un comando seriale o di un nuovo campione). Quando il flag dell'ADC viene alzato, il sistema richiama la funzione `process_sample(adc_value)`, il vero nucleo elaborativo del sistema.

```
static void process_sample (uint16_t value)
{
    // Trasmissione telemetrica via UART
    serial_send_hex(value >> 8);
    serial_send_hex(value & 0xFF);
    serial_send("\r\n");

    // Elaborazione e normalizzazione
    int16_t  v = value - 0x4000;
    uint8_t  y = 150 - (v >> 6);
    uint16_t x = signal_idx << 2;  // 4 pixels per timestep

    // Disegno del nuovo segmento
    current_color = line_color;
    plotLine(x, signal_buf[signal_idx], x + 4, y);
    signal_buf[++signal_idx] = y;

    // Cancellazione della coda (effetto spazzata)
    uint8_t k = signal_idx - 200;
    uint8_t k1= k + 1;
    x = k << 2;
    current_color = back_color;
    plotLine(x, signal_buf[k], x + 4, signal_buf[k1]);
}
```

Listing 6.2: Funzione di elaborazione del campione e plottaggio video.

La funzione `process_sample` (Listato 6.2) esegue diverse operazioni critiche ad ogni campione ricevuto. Inizialmente, invia il dato grezzo a 16 bit sulla porta seriale frammentandolo in due invii da 8 bit, utili per un eventuale *datalogging* su PC.

Successivamente, il dato deve essere scalato per poter essere rappresentato sullo schermo VGA. Sapendo che l'ADC produce valori distribuiti attorno al valore `0x4000` (che rappresenta lo zero analogico), il firmware esegue la sottrazione `v = value - 0x4000` per rimuovere la componente continua e centrare l'oscillazione attorno allo zero logico.

Vengono poi calcolate le coordinate per il *rendering*:

- **Coordinata verticale (`y`)**: Il display utilizzato ha un'altezza di 256 pixel. Si è scelto il pixel 150 come asse centrale del tracciato. Il valore del campione `v` viene scalato dividendolo per 64 (operazione di *shift* a destra `v >> 6`) e sottratto a 150. La sottrazione è necessaria poiché, negli standard video, l'origine degli assi ($y = 0$) si trova nell'angolo in alto a sinistra dello schermo.

- **Coordinata orizzontale (x)**: I singoli campioni vengono spaziati orizzontalmente di 4 pixel l'uno dall'altro (`signal_idx « 2`). Questi punti non appaiono isolati, ma vengono uniti da segmenti (interpolati), formando visivamente un'unica linea continua che allarga la scala orizzontale del tracciato per renderlo agevolmente leggibile.

Il programma imposta quindi il colore del tratto (giallo/verde, tipico dei monitor medicali) e traccia un segmento interpolato tra il campione precedente e quello attuale tramite la funzione `plotLine`. Per simulare il tipico effetto "a spazzata" dei monitor ECG hardware senza dover riempire e cancellare l'intero *framebuffer* video, si utilizza una tecnica di *tail erasing*: il firmware calcola un indice arretrato di 200 campioni (`k = signal_idx - 200` e individuando l'estremo successivo del segmento `k+1`) e richiama nuovamente `plotLine` impostando il colore in nero (`back_color`), cancellando selettivamente solo la coda vecchia del segnale.

6.2 Interfaccia di Comando e Output Video

L'interazione con l'utente è demandata alla decodifica dei comandi ricevuti tramite UART, come evidenziato nel seguente blocco di codice:

```
if (uart->control.rx_ready) {
    uint8_t x = uart->data;
    uart->data = x; // Echo del carattere
    if (x == '\r') { // add LF
        while (uart->control.tx_busy) ;
        uart->data = '\n';
    }
    if (x == '0') load_image(0x00000);
    if (x == '1') load_image(0x20000);
    if (x == '2') load_image(0x40000);
    if (x == ' ') clear_screen();
    if (x == 'H') draw_grid();
    if (x == 'P') show_palette();
    if (x == 'E') read_edid();
    if (x == 'L') *led ^= 0x30;
    if (x == 'l') *led  = 0x00;
    // ...
}
```

Listing 6.3: Gestione dei comandi ricevuti da terminale seriale UART.

Inviando specifici caratteri al terminale seriale, è possibile scatenare diverse routine hardware. Inviando il comando **'0'**, il processore RISC-V accede alla memoria video per caricare il monoscopio di test (Figura 6.1), utile per validare il corretto funzionamento del DAC e la correttezza della *palette* colori.

Infine, inviando il comando **'E'**, il sistema e inizia il tracciamento in tempo reale dell'elettrocardiogramma sullo schermo. Come illustrato nella Figura 6.2, per testare

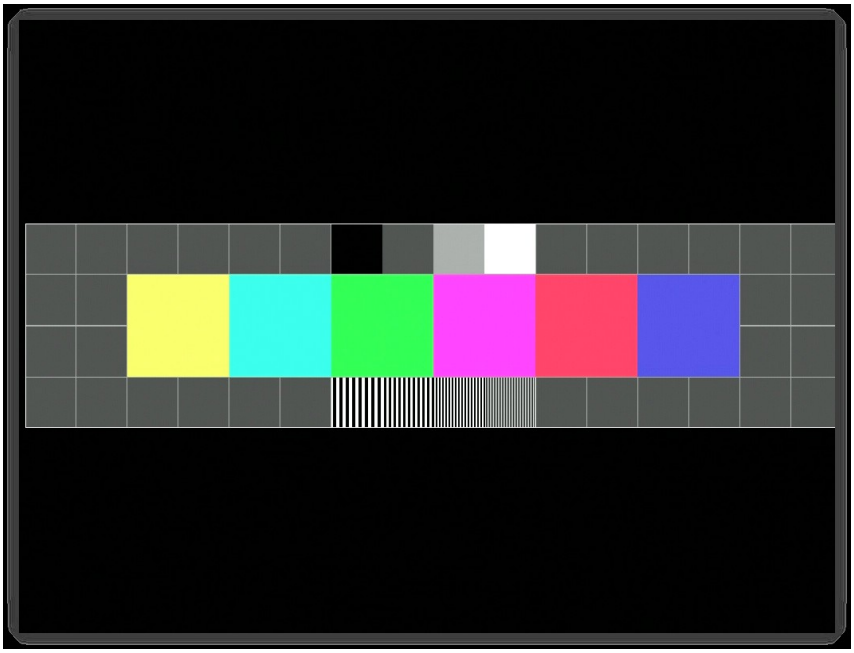

Figura 6.1: Output VGA a seguito del comando '0': Monoscopio di test per la validazione della generazione del segnale video e della palette cromatica.

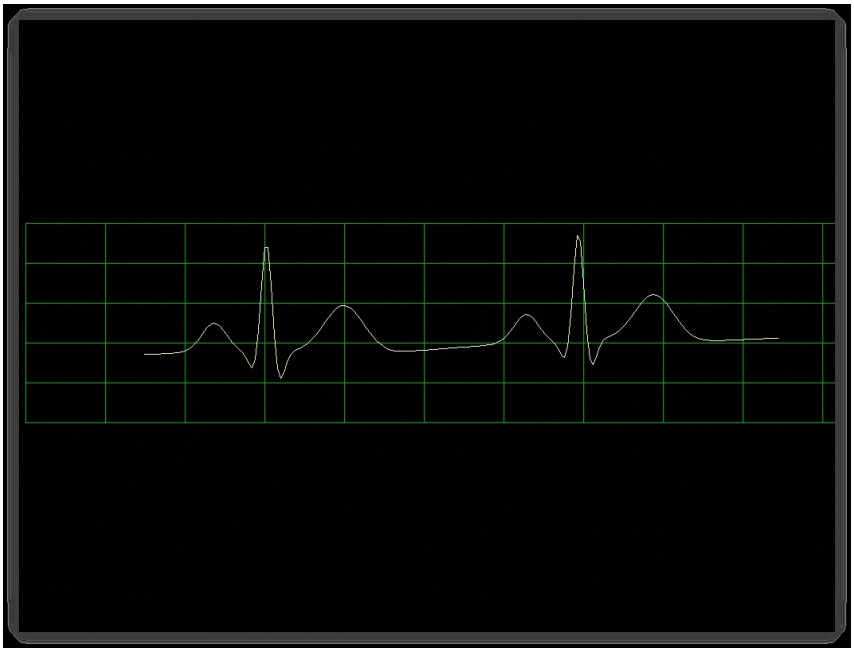

Figura 6.2: Esecuzione finale dell'ECG sul monitor VGA. Il tracciato viene generato interpolando in tempo reale un dataset di dati medicali.

le potenzialità di plottaggio è stato utilizzato un *dataset* di valori pre-acquisiti tramite un convertitore Sigma-Delta (*Sigma-Delta ADC*).

Il tracciato risulta fluido e correttamente interpolato, dimostrando che il processore NEORV32, il Bridge VHDL e l'infrastruttura legacy comunicano in perfetta sinergia, soddisfacendo pienamente le specifiche temporali e funzionali del progetto originale.

6.3 Analisi comparativa delle prestazioni: 6502 vs NEORV32

A conclusione della fase di validazione, si è ritenuto opportuno effettuare un'analisi comparativa delle prestazioni tra il sistema originale basato sul processore MOS 6502 e la nuova architettura basata su NEORV32 interfacciato tramite Bridge.

L'obiettivo di questo test è quantificare l'impatto prestazionale del nuovo processore su operazioni ad alta intensità di memoria. A tal fine, è stata analizzata la fase di inizializzazione del tracciato video, che prevede due operazioni computazionalmente pesanti: la pulizia del *framebuffer* (`clear_screen()`) e il disegno della griglia di base dell'elettrocardiogramma (`draw_grid()`).

Per misurare con precisione il tempo di esecuzione di queste funzioni senza introdurre *overhead* software sono stati utilizzati i bit del registro di controllo dei LED come *marker* temporali, facendoli commutare (operazione di XOR) immediatamente prima e dopo le chiamate a funzione.

```
int main ()
{
    spi->data = 0;     // Dummy write per sincronizzazione STARTUPE2
    busy_wait(10000); // Attesa assestamento linee UART
    *led = 0x10;       // Segnale di sistema pronto

    *led ^= 0x30;      // MARKER 1: Inizio operazioni (LED -> 01)
    clear_screen();
    *led ^= 0x30;      // MARKER 2: Fine clear_screen
    draw_grid();
    *led ^= 0x30;      // MARKER 3: Fine draw_grid (LED -> 10)

    // ... prosecuzione esecuzione
}
```

Listing 6.4: Frammento di codice C utilizzato per il *profiling* temporale.

Monitorando in fase di simulazione il segnale di uscita `led_user`, è stato possibile misurare l'esatto intervallo di tempo intercorso tra l'istante in cui il LED assume il valore `01` (MARKER 1) e l'istante in cui transita al valore `10` (MARKER 3).

Nel sistema *legacy* basato su **MOS 6502** (Figura 6.3), l'esecuzione inizia approssimativamente al millisecondo 0.317 e termina al millisecondo 283.815. L'intervallo temporale totale richiesto per completare le operazioni sulla memoria video ammonta quindi a circa **283.5 ms**.

Eseguendo il medesimo test sulla nuova architettura ibrida basata su **NEORV32** (Figura 6.4), il marker iniziale si attiva a 0.089 ms, mentre il completamento della

routine avviene a 120.90 ms. L'intervallo di esecuzione effettivo risulta essere di circa **120.8 ms**.

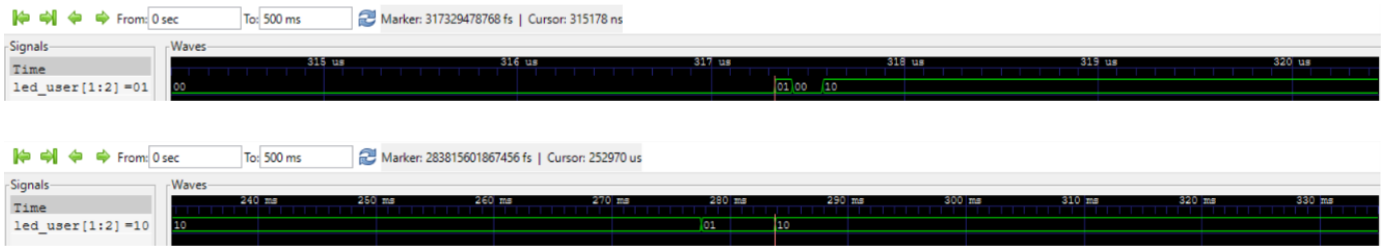

Figura 6.3: Simulazione del sistema legacy (6502): i cursori delimitano l'intervallo di esecuzione tra l'inizio di `clear_screen()` e la fine di `draw_grid()`.

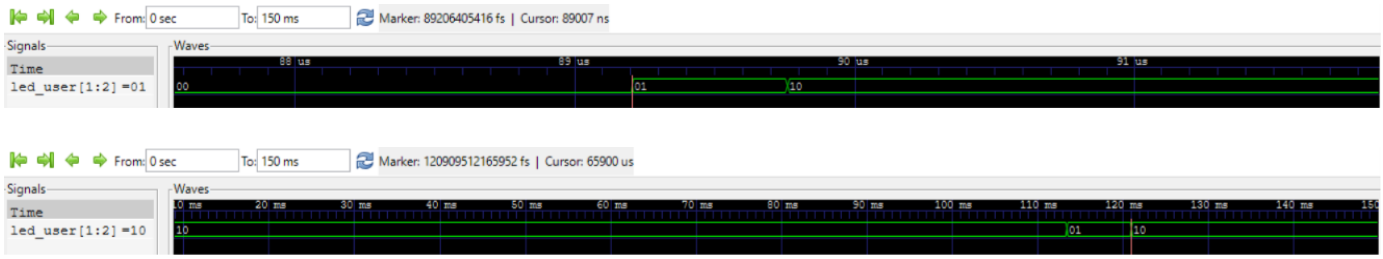

Figura 6.4: Simulazione del sistema ibrido (NEORV32 + Bridge): si osserva una netta riduzione del tempo impiegato per le medesime operazioni in memoria.

6.3.1 Considerazioni finali

I risultati della simulazione certificano che il nuovo sistema è in grado di eseguire le operazioni di gestione del *framebuffer* in meno della metà del tempo rispetto al processore *legacy* (circa **2.3 volte più veloce**).

Questo risultato è di particolare rilevanza se contestualizzato rispetto all'architettura sviluppata in questo lavoro: benché l'inserimento del Bridge VHDL introduca inevitabilmente cicli di latenza (stati di attesa della Macchina a Stati) dovuti all'*handshaking* del protocollo XBUS e alla traduzione degli indirizzi, la superiore efficienza dell'architettura a 32 bit del NEORV32 compensa ampiamente tale *overhead*. Il sistema non solo acquisisce le moderne capacità di *debugging* JTAG e la versatilità dell'ecosistema RISC-V, ma ottiene anche un drastico miglioramento delle prestazioni globali.

Capitolo 7

Conclusioni e sviluppi futuri

Il lavoro di tesi presentato ha documentato l'intero percorso di migrazione architetturale di un sistema *embedded* basato su microprocessore MOS 6502 verso una moderna piattaforma basata su NEORV32 (RISC-V a 32 bit). La sfida principale ha risieduto nell'integrazione di un *core* a 32 bit all'interno di un ecosistema vincolato da un bus a 8 bit per i dati e 16 bit per gli indirizzi, imponendo la progettazione di un *Bridge* VHDL dedicato che gestisse la discrepanza tra le diverse temporizzazioni e le logiche di *handshaking*.

L'attività si è sviluppata attraverso un rigoroso processo di validazione:

- **Sviluppo del Bridge**: è stata progettata una logica sequenziale in grado di tradurre le transazioni del bus RISC-V in operazioni compatibili con il sistema *legacy*, implementando tecniche di *pipelining* e di *address look-ahead* per minimizzare le latenze di accesso.
- **Integrazione JTAG/DTM**: per garantire la piena osservabilità del sistema, è stato riprogettato il *Debug Transport Module*. L'impiego delle primitive `BSCANE2` ha permesso di superare le limitazioni fisiche della *board*, mentre il nuovo meccanismo di sincronizzazione a quattro fasi ha disaccoppiato i domini di *clock*, consentendo operazioni di *debug* stabili e indipendenti dalla frequenza operativa del processore.
- **Validazione finale**: l'esecuzione del firmware ECG originale su hardware NEORV32 ha confermato la correttezza del porting. L'analisi comparativa delle prestazioni ha inoltre evidenziato un incremento velocistico superiore a 2.3 volte rispetto al sistema originale, dimostrando che i benefici computazionali dell'architettura a 32 bit superano ampiamente le latenze introdotte dal *Bridge*.

Sviluppi futuri

Il lavoro svolto ha dimostrato la piena fattibilità dell'approccio proposto, validando l'integrazione tra l'architettura RISC-V e l'infrastruttura *legacy*. In un'ottica di rifinitura incrementale, il progetto consente ulteriori ottimizzazioni mirate, senza la necessità di stravolgere l'architettura consolidata:

- **Sincronizzazione delle periferiche**: un primo intervento potrebbe riguardare la rimozione delle poche periferiche asincrone ancora presenti nel sistema, integrandole nativamente nel dominio di *clock* del processore per migliorare la stabilità complessiva del *timing*.
- **Ottimizzazione del datapath**: il *firmware* attualmente esegue operazioni di scrittura e lettura sulla memoria video a 8 bit. Data la natura a 32 bit del core NEORV32 e la capacità di *burst* del Bridge progettato, si potrebbe ottimizzare il protocollo di comunicazione verso il *framebuffer*, implementando scritture e letture native a 32 bit per ridurre drasticamente il numero di cicli necessari all'aggiornamento grafico.

Tali interventi, pur mantenendo inalterata la struttura del sistema, permetterebbero di saturare le potenzialità del nuovo *core* e di ridurre ulteriormente l'impronta temporale delle routine di gestione video.

Ringraziamenti

Desidero esprimere la mia più sincera gratitudine a tutte le persone che, con il loro sostegno, la loro presenza e il loro incoraggiamento, hanno contribuito a rendere possibile il raggiungimento di questo importante traguardo.

Innanzitutto, desidero rivolgere un ringraziamento speciale al Professor Giorgio Biagetti, che con la sua encomiabile dedizione all'insegnamento ha reso unico il percorso intrapreso insieme. La mia crescita, sia personale sia professionale, è stata possibile grazie alle sue profonde e versatili competenze, ma soprattutto grazie alla disponibilità, all'umiltà, alla pazienza e al costante supporto che mi ha offerto durante tutto il percorso: dal corso da lui tenuto fino allo svolgimento del tirocinio e della tesi.

Un ringraziamento speciale va poi alla mia famiglia, e in particolare ai miei genitori, che con il loro amore, i sacrifici e il sostegno incondizionato mi hanno permesso di vivere con serenità le sfide del percorso universitario.

Infine, desidero ringraziare di cuore Eli, Thomas, Pincia, Nicola, Ferri e Giovi, amici con i quali ho condiviso questi anni universitari. Oltre ad aver reso questo percorso ricco di momenti di amicizia, sono stati per me un prezioso punto di riferimento. Grazie alla loro esperienza e ai loro consigli, sempre generosamente condivisi, ho potuto affrontare con maggiore consapevolezza molte delle sfide incontrate lungo il cammino. Il loro aiuto e la loro vicinanza sono stati fondamentali, sia dal punto di vista accademico sia da quello umano.

Sempre con me, nonno

Ancona, Luglio 2026

Lorenzo Fabiani

Bibliografia

[1] S. Nolting, "The neorv32 processor." https://github.com/stnolting/neorv32, 2024.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2020.

[3] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Tech. Rep. UCB/EECS-2014-146, EECS Department, University of California, Berkeley, 2014.

[4] RISC-V International, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, 2022. https://lists.riscv.org/g/tech-unprivileged/attachment/535/0/unpriv-isa-asciidoc.pdf.

[5] Advanced Micro Devices, Inc. (AMD) / Xilinx, *Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide (UG953) - BSCANE2*, 2024. https://docs.amd.com/r/en-US/ug953-vivado-7series-libraries/BSCANE2.

[6] S. Nolting, *The NEORV32 RISC-V Processor - Datasheet*, 2024. https://stnolting.github.io/neorv32/.

[7] S. Nolting, *The NEORV32 RISC-V Processor - User Guide*, 2024. https://stnolting.github.io/neorv32/ug/.

[8] S. Nolting and NEORV32 Community, "Neorv32 github discussions: JTAG/OCD implementation (#28)." https://github.com/stnolting/neorv32/discussions/28, 2021.

[9] P. W. e OpenCores Community, *6502 IP Core Specification, Version 0.7*. OpenCores, 2011. https://opencores.org/websvn/filedetails?repname=cpu6502_true_cycle&path=%2Fcpu6502_true_cycle%2Ftrunk%2Fdoc%2F6502+IP+Core+Specification_V0_7.pdf.

al migliore