Labs

1.1 Lab 1 / Polling and Interrupts

L'obiettivo principale del laboratorio è stato comprendere l'integrazione tra hardware e software attraverso l'architettura AXI (Advanced eXtensible Interface). Abbiamo analizzato l'AXI Interconnect, che funge da vera e propria spina dorsale del Sistem-on-Chip. A differenza dei cablaggi tradizionali, l'interconnessione avviene a livello di logica programmabile nel silicio, permettendo comunicazioni ad alta velocità e bassa latenza tra il processore e le periferiche. Un concetto chiave del laboratorio è quello del Memory-Mapped I/O. In questo schema, ogni periferica (GPIO) viene vista dalla CPU come un semplice indirizzo di memoria. Nello specifico, abbiamo utilizzato:

- `AXI GPIO 0` : Gestisce i LED.
- `AXI GPIO 1` : Gestisce gli interruttori.
- `AXI GPIO 2` : Gestisce i pulsanti.

Quando la CPU deve leggere lo stato di un pulsante o accendere un LED, esegue operazioni di lettura o scrittura su specifici indirizzi di memoria mappati dall'Interconnect. Questo sistema permette di gestire l'hardware esterno con la stessa semplicità con cui si accede alla RAM.

Infine, è essenziale distinguere tra le due metodologie di gestione dell'I/O: Polling e Interrupts.

- *Polling*: In questa modalità, la CPU interroga ciclicamente la periferica per verificare se un evento è accaduto (ad esempio, se un pulsante è stato premuto). Sebbene semplice da implementare, questo approccio non è efficiente: la CPU consuma il 100% delle risorse in un ciclo di attesa, rimanendo bloccata e non potendo svolgere altri compiti.
- *Interrupts*: Questo metodo permette una gestione asincrona degli eventi. La CPU può dedicarsi ad altre elaborazioni finché non riceve un segnale d'interruzione. In questo contesto, l'AXI Interrupt Controller svolge un ruolo fondamentale: concentra i segnali provenienti dalle diverse periferiche e notifica il processore solo quando necessario. Quando l'Interrupt viene attivato, la CPU sospende temporaneamente il programma principale, salva il proprio stato e salta a una funzione specifica chiamata ISR (Interrupt Service Routine) per gestire l'evento. Una volta terminata la gestione (ad esempio, dopo aver aggiornato lo stato di un LED in base a un pulsante), la CPU torna esattamente dove si era interrotta.

Task 1. Il primo task illustra il concetto di polling. In questo breve script, un concetto importante è la dichiarazione dei puntatori come `volatile int *` : questa keyword informa il compilatore che il contenuto di questi indirizzi di memoria può variare indipendentemente dal software (a causa di modifiche dal lato hardware). Senza `volatile` , il compilatore applicherebbe delle ottimizzazioni assumendo che il valore non cambi mai, leggendo il dato una sola volta e ignorando gli aggiornamenti successivi. Sebbene sia funzionale, il polling risulta estremamente inefficiente: la CPU è impegnata al 100% in un ciclo di attesa, consumando risorse per campionare continuamente il registro `DATA_1` anche quando non avvengono variazioni.

Task 2. In questo secondo task, il sistema viene configurato per reagire agli eventi hardware (pressione di pulsanti o cambio stato degli switch) tramite Interrupt, eliminando la necessità per la CPU di controllare ciclicamente le periferiche. Il codice introduce una catena di abilitazione tipica dei sistemi MicroBlaze:

(1) *Livello Periferico (GPIO)*: Vengono abilitati gli interrupt locali per i due canali GPIO (`GIER` e `GGIER`).

(2) *Livello Controller (INTC)*: Vengono configurati i registri `IER` (Interrupt Enable Register) e `MER` (Master Enable Register) per permettere ai segnali delle periferiche di raggiungere la CPU.

(3) *Livello Processore*: La funzione `microblaze_enable_interrupts()` attiva globalmente la ricezione delle interruzioni nel core.

A differenza del polling, il `main()` termina con un ciclo `while(1)` vuoto. In una situazione reale, qui la CPU potrebbe eseguire altri calcoli o entrare in modalità Low Power.

1.2 Lab 2 / The PS

L'obiettivo principale del laboratorio è stato lo sviluppo di un microserver seriale capace di gestire un flusso I/O. Il cuore del laboratorio consiste nel mettere in comunicazione il microcontrollore con un host esterno tramite il protocollo UART (Universal Asynchronous Receiver-Transmitter), una periferica che trasforma i dati paralleli del bus interno in un treno di bit seriali. Il funzionamento si articola in tre fasi sequenziali, gestite in modalità polling:

(1) *Ricezione*: Il sistema resta in un ciclo di attesa finché la flag di ricezione della UART non segnala l'arrivo di un nuovo byte. Essendo una gestione blocking, la CPU è interamente dedicata al monitoraggio della periferica, garantendo la lettura immediata del dato non appena disponibile nel registro di ricezione.

(2) *Processing*: Una volta acquisito il dato, il codice esce dal loop di attesa per eseguire le trasformazioni richieste, elaborando i dati grezzi prima della loro trasmissione.

(3) *Trasmissione*: I dati processati vengono inviati nuovamente sulla stessa linea UART. Anche in questo caso, utilizziamo un approccio blocking: il programma attende che il registro di trasmissione sia vuoto prima di caricare il nuovo byte, assicurando che nessun dato venga sovrascritto o perso durante l'invio.

Task 1. L'obiettivo di questo primo task è stato l'implementazione di un filtro di negazione su un'immagine in formato PPM, gestendo il flusso di dati in real-time. Il processo si divide in due fasi logiche distinte:

- *Trasmissione dell'header*: Poiché la struttura e la dimensione dei metadati sono note, il microserver si limita a ricevere e ritrasmettere i primi n byte dell'header. Questa fase avviene in modo trasparente: il microcontrollore funge da ponte, garantendo che i parametri dell'immagine rimangano intatti per il file di destinazione.
- *Elaborazione Pixel-by-Pixel*: Terminata la fase dell'header, il sistema inizia il ciclo di processing. Viene prelevato il singolo byte del pixel non appena disponibile nel registro di ricezione. Viene applicata l'operazione di negazione calcolando $255 - \text{pixel}_{\text{in}}$. Il valore risultante viene caricato nel registro di trasmissione e inviato immediatamente.

Task 2. Il second task ha richiesto un salto di complessità notevole: l'implementazione dell'Histogram Linear Stretching. A differenza del filtro negativo, questo algoritmo richiede la conoscenza dei valori estremi (minimo e massimo) dell'intera immagine, rendendo impossibile l'elaborazione "al volo" e rendendo necessari buffering e parsing dell'header.

(1) *Parsing*: Poiché il microserver deve ora supportare immagini di dimensioni arbitrarie, non è più possibile basarsi su costanti predefinite. Ho implementato un parser per l'header PPM che opera sui metadati in formato ASCII. Il parser analizza lo stream di byte cercando i line breaks che separano le tre sezioni dell'header (Magic Number, Dimensioni, Intensità Massima). È stato necessario convertire le stringhe ASCII in interi per poter calcolare l'area totale dell'immagine e procedere all'allocazione dinamica della memoria nella RAM tramite la struttura `PPMImage` .

(2) *Algoritmo di Stretching*: Una volta memorizzata l'intera matrice dei pixel, il processing avviene in tre passaggi. Una prima scansione completa dei dati per individuare il valore minimo ($I_{\min}$) e massimo ($I_{\max}$) effettivamente presenti nell'immagine. Definizione della costante di espansione per mappare l'intervallo sull'intero range. La scala è definita come $S = \frac{255.0}{I_{\max} - I_{\min}}$. Una seconda scansione in cui ogni pixel viene normalizzato secondo la formula:

$$\text{pixel}_{\text{out}} = (\text{pixel}_{\text{in}} - I_{\min}) \cdot S \tag{1}$$

Task 3. Il terzo task ha riguardato l'implementazione dell'Histogram Equalization, un algoritmo avanzato che, a differenza del semplice Linear Stretching, opera una trasformazione non lineare per uniformare la distribuzione delle intensità luminose. Mentre lo stretching espande l'intervallo, l'equalizzazione appiattisce l'istogramma, potenziando il contrasto locale nelle aree con frequenze di pixel più elevate.

- Sfruttando il parser sviluppato nel Task 2, il sistema esegue una scansione dei dati per costruire l'istogramma delle frequenze. Da questo viene derivata la CDF (Cumulative Distribution Function), che rappresenta la probabilità cumulata di ogni livello d'intensità.
- Per massimizzare l'efficienza, è stata usata una `unsigned char map[256]` . Invece di rieseguire il calcolo della normalizzazione per ogni singolo pixel dell'immagine, il valore normalizzato viene calcolato una sola volta per ognuno dei 256 livelli e memorizzato nella mappa. Il processing dell'immagine si riduce quindi a un semplice look-up a costo costante velocizzando drasticamente l'esecuzione.
- La trasformazione segue la formula:

$$\text{normalized} = \frac{\text{cdf}[i] - \text{cdf}_{min}}{\text{total\_pixels} - \text{cdf}_{min}} \tag{2}$$

  Un dettaglio implementativo interessante è l'aggiunta dell'offset di `+ 0.5f` prima del casting a `unsigned char` . Questa tecnica di arrotondamento all'intero più vicino, anziché il semplice troncamento, preserva una maggiore fedeltà e riduce gli artefatti visivi nel risultato finale.

1.3 Lab 3 / AD-DA Conversion and Audio Processing

L'obiettivo di questo laboratorio è realizzare un sistema di elaborazione audio in tempo reale: acquisizione tramite codec (ADC), filtraggio digitale (FIR), e riproduzione (DAC). Il sistema sfrutta una gerarchia di bus per bilanciare velocità e controllo: l'AXI Bus è l'infrastruttura ad alta velocità interna al chip che collega la CPU alle periferiche senza fili fisici, agendo come dorsale per i dati. I2C è un protocollo per inviare comandi di configurazione al codec, mentre I2S è un protocollo dedicato esclusivamente al trasporto dei campioni audio ad alta fedeltà. Il filtro FIR (Finite Impulse Response) elabora l'audio calcolando una media ponderata di una finestra di campioni: i taps sono il numero di campioni passati mantenuti in memoria, mentre i coefficienti sono i pesi numerici che determinano la risposta del filtro.

Funzionamento di base. L'applicazione è capace di reagire agli input fisici della scheda Zybo per modificare il comportamento in tempo reale.

(1) *Inizializzazione e Interfaccia Utente*: Il sistema monitora lo stato degli switch per definire la modalità operativa. Ho implementato una logica di controllo basata su bitmasking per mappare gli switch ai filtri desiderati. Il feedback visivo è garantito dall'accensione dei LED corrispondenti, assicurando che lo stato interno del software sia sempre sincronizzato con l'interfaccia hardware. Un primo check fondamentale riguarda la frequenza di campionamento, impostata a 48 kHz, standard necessario per garantire la fedeltà del segnale audio trattato.

(2) *Test Run*: Prima di processare lo stream audio reale, il sistema esegue una test run deterministica. Viene somministrato un vettore d'input noto alla funzione `FIR_Filter` e il risultato viene confrontato con un vettore di output atteso. Questo passaggio serve a confermare la correttezza matematica dell'algoritmo e della gestione del buffer circolare. L'assenza di errori in questa fase garantisce che eventuali artefatti nell'audio finale siano imputabili esclusivamente ai parametri del filtro e non a bug logici o problemi di memoria.

(3) *Processing Loop Infinito*: Una volta validato, il sistema entra nel core operativo. All'interno del loop, il codice interroga lo stato degli switch. A seconda dell'input, la funzione `FIR_Filter` riceve il puntatore all'array di coefficienti specifico. Questo permette di cambiare il comportamento del filtro al volo senza interrompere il flusso audio. L'elaborazione avviene campione per campione. Grazie alle ottimizzazione fatte sulla funzione, il tempo di calcolo è ampiamente inferiore al periodo di campionamento.

Ottimizzazione del filtro. Entrambe le funzioni implementano lo stesso filtro FIR su un buffer circolare, quindi dal punto di vista matematico producono lo stesso risultato. Nella versione originale, il riavvolgimento dell'indice del buffer è gestito con un'operazione di modulo all'interno del ciclo di convoluzione, quindi viene eseguito un modulo per ogni tap e per ogni campione. A una frequenza di campionamento di 48 kHz, questo significa eseguire un numero molto elevato di operazioni relativamente costose. La versione ottimizzata riscrive il ciclo in modo da attraversare il buffer circolare in due segmenti lineari: prima dall'indice corrente fino a zero, poi dalla fine del buffer fino all'indice corrente. In questo modo si elimina il modulo dal ciclo interno e rimane un solo modulo per aggiornare l'indice del buffer. L'algoritmo resta identico, ma il costo per campione si riduce trasformando la gestione del wraparound da un costo `O(N)` a un costo `O(1)` , migliorando le prestazioni in tempo reale.

- `inline` : Suggerisce al compilatore di sostituire la chiamata alla funzione direttamente nel codice chiamante, eliminando l'overhead del function call, che sarebbe molto pesante se ripetuto per ogni campione.
- `static` : Limita la visibilità della funzione al singolo file, permettendo al compilatore di ottimizzare ulteriormente il codice sapendo che non verrà richiamata dall'esterno.
- `const` : Definire i coefficienti come costanti garantisce che i pesi del filtro non vengano modificati accidentalmente e permette al compilatore di gestire meglio i registri.

1.4 Lab 5 / DNN on MNIST

L'obiettivo del laboratorio è la creazione di un microserver per il riconoscimento di cifre scritte a mano (dataset MNIST), integrando un modello di Deep Neural Network direttamente sulla scheda. Il laboratorio è stato diviso in due fasi:

(1) *Test Statico*: Inizialmente, l'inferenza è stata testata su un set d'immagini pre-caricate in memoria (`test_images.h`) per validare la correttezza della rete.

(2) *Integrazione UART*: Successivamente, il sistema è stato trasformato in un server reale. Sfruttando la logica di ricezione del Lab 2, il microserver riceve file `.bin` via UART, esegue l'inferenza sull'immagine ricevuta e restituisce il risultato della classificazione (0–9) all'host.

Ottimizzazioni. L'analisi dei file tramite hex editor ha rivelato che, sebbene le immagini siano codificate in formato Q8.8 (16 bit), il byte più significativo è sistematicamente nullo. Questo indica che il range dei valori è interamente contenuto negli otto bit meno significativi, rendendo il formato equivalente a un Q0.8. Per ottimizzare il sistema, ho fatto le seguenti modifiche:

- *Riduzione del Payload*: Le immagini vengono ora archiviate e trasmesse nel formato compatto Q0.8. Questo dimezza istantaneamente la quantità di dati inviati via UART. Questa ottimizzazione comporta anche un abbattimento del 50% dei tempi di trasmissione e una riduzione della metà dello spazio di archiviazione necessario, senza alcuna perdita d'informazione.
- *Consistenza dell'Infrastruttura*: Al momento della ricezione, il microserver effettua un semplice casting a 16 bit per riportare i dati al formato Q8.8 originale. In questo modo, garantiamo la piena compatibilità con l'architettura della rete neurale preesistente.

La seconda ottimizzazione riguarda la gestione dei parametri della rete. Anche in questo caso, l'analisi tramite hex editor ha evidenziato un'opportunità di compressione significativa. Il byte più significativo dei parametri salvati in Q8.8 presenta esclusivamente i valori `0xFF` o `0x00` . Questo pattern indica che la parte intera non trasporta magnitudo, ma funge solo da estensione del segno per valori compresi nell'intervallo $[-1, 1]$. Data questa caratteristica, i parametri possono essere efficacemente rappresentati nel formato Q1.7. Questa transizione permette di mantenere la precisione necessaria per l'inferenza della DNN eliminando la ridondanza informativa.

Avanzato. L'ultima evoluzione del sistema prevede il passaggio integrale a una precisione a otto bit per pesi, bias e attivazioni. Questa scelta è strategica: non solo dimezza l'occupazione di memoria e i tempi UART, ma abilita l'uso efficiente delle istruzioni SIMD del processore ARM, permettendo di processare più campioni in un singolo ciclo di clock.

Dopo aver normalizzato pesi e bias nell'intervallo $[-1, 1]$, ho valutato due opzioni:

- *Q1.7*: Sebbene naturale per il complemento a due, riserva un bit per la parte intera che, dopo la normalizzazione, rimarrebbe inutilizzato.
- *Q0.7*: Ho scelto questo formato perché concentra l'intera risoluzione disponibile sulla parte frazionaria. In Q0.7, l'intero range degli otto bit è dedicato alla precisione del valore decimale, mappando perfettamente la distribuzione statistica dei nostri parametri.

Per garantire la congruenza dei dati all'interno della rete, ho implementato due funzioni che uniformano i diversi formati sorgente verso il target Q0.7:

- `read_weight_Q88_to_Q07()` : Riceve il dato a 16 bit, applica un arrotondamento e scala il valore per passare da Q8.8 a Q0.7.
- `read_pixel_to_Q07()` : Converte i pixel (originariamente in Q0.8, range $[0, 255]$) nel range $[0, 127]$ del formato Q0.7, garantendo che l'input della rete sia coerente con i pesi.

La funzione d'inferenza è stata modificata per operare interamente con aritmetica a virgola fissa in Q0.7. Come nella versione originale, uso un accumulatore a 64 bit per evitare l'overflow durante le somme dei prodotti. Il bias viene scalato (`« qf`) per allinearsi alla precisione del prodotto input × weight. Al termine del calcolo, il risultato viene riportato a otto bit tramite shift e processato da una funzione `saturate()` . Questo previene errori di wrapping tipici del complemento a due quando si supera il valore massimo rappresentabile.