UNIVERSITÀ DEGLI STUDI DI PISA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

ANNO ACCADEMICO 2003-2004

TESI DI LAUREA:

Progetto e Realizzazione di un Sistema Operativo Real-Time per Sistemi Embedded con riferimento al microcontrollore Motorola PowerPC

CANDIDATO:

Antonio Fiaschi .....

RELATORI:

Prof. Paolo Ancilotti .....

Prof. Luigi Rizzo .....

Dott. Ing. Giuseppe Lipari .....

ai miei genitori e a Marina,

pazienti sostenitori del mio lavoro

Il primo ringraziamento in assoluto va ai miei genitori, Guido e Maria Rosa, perchè senza di loro la realizzazione di questo mio sogno non sarebbe stata possibile.

Il secondo invece va a Marina che in questi anni ha saputo aiutarmi e spronarmi per far sì che io riuscissi a terminare questa esperienza.

Un pensiero va a tutti i miei parenti: anche loro hanno sempre attivamente contribuito a spronarmi nello studio. Insieme a loro vorrei anche ringraziare i genitori di Marina, Bruno ed Armida, e tutta la sua famiglia in particolare Andrea, Cinzia e Bruno .

Ed ora la parte difficile, ci sono decine di persone che devo e voglio ringraziare e tutte meriterebbero il primo posto.

Un particolare ringraziamento va a Paolo, e a tutta Evidence, che mi ha dato la possibilità di cimentarmi in questo lavoro. Poi ci sono Luca, Igor e tutto il ReTiS Lab: grazie ai loro consigli ed al loro aiuto sono riuscito ad affrontare tutti i nuovi problemi che durante lo svolgimento di questa tesi mi si sono presentati.

Ora vorrei ringraziare tutti i miei più cari amici, in rigoroso ordine alfabetico: Andrea, Andrea, Chiara, Daniele, Francesca, Michela, Nicola e Simone. Grazie al loro aiuto, magari inconsapevole, sono riuscito ad affrontare ed a superare i momenti difficili e le delusioni della mia vita di studente.

Un particolare ringraziamento va anche a tutti i miei compagni di studio di questi anni, con cui ho condiviso tensioni e gioie della mia carriera universitaria. In particolare il ringraziamento va a Fabio, Michele, Riccardo, Sergio ed Umberto, con cui ho condiviso la maggior parte delle ore i studio e gli esami più impegnativi.

Poi vorrei anche ringraziare tutti gli amici della “Biblioteca Civica” di Massa: Daniele, Francesca, Laura, Laura, Matteo, Michele, Simona, Stefania e Veronica. Grazie per la vostra amicizia e per i vostri consigli.

E per ultimi vorrei ringraziare tutti quelli che mi hanno sempre detto:“**Ma non hai ancora finito!!??**”. Inconsapevolmente mi spingevano a dare sempre di più nello studio e quindi hanno contribuito al mio sforzo per arrivare sino a qui.

Indice

1 Introduzione **13**
1.1 Scopo del Lavoro 13
1.2 Contenuto del Documento 14

2 I Sistemi embedded **17**
2.1 Caratteristiche 18
2.2 Standard e tecniche di progettazione 20
2.2.1 OSEK / VDX 22
2.2.2 Platform-Based Design 24
2.3 Considerazioni e prospettive 26

3 Microcontrollore Motorola MPC5xx **29**
3.1 Caratteristiche generali 29
3.1.1 Architettura PowerPC 30
3.1.2 I livelli dell'architettura 31
3.2 RCPU 32
3.2.1 Registri 32
3.2.1.1 Registri UISA 33
3.2.1.2 Registri VEA 37
3.2.1.3 Regitsri OEA 37
3.2.2 Modalità di indirizzamento e Set di istruzioni 41
3.2.2.1 Modalità di indirizzamento 42
3.2.2.2 Istruzioni di sincronizzazione 42
3.2.2.3 Istruzioni UISA 43
3.2.2.4 Istruzioni VEA 46

3.2.2.5 Istruzioni OEA 46
3.2.3 Eccezioni 47
3.2.3.1 Classi di Eccezioni 48
3.2.3.2 Sincronizzazioni 50
3.2.3.3 Elaborazione delle Eccezioni 51
3.2.3.4 Switch tra Processi 52
3.2.4 Memory Management Unit 53
3.2.4.1 Meccanismoo di Traduzione 53
3.2.4.2 Real Addrressing Mode 54
3.2.4.3 Block Address Translation 55
3.2.4.4 Modello di Memoria Segmentata 58
3.3 Convenzioni di programmazione 60
3.3.1 Tipi di Dati ed Allineamento 60
3.3.2 Convenzione sull'utilizzo dei Regsitri 61
3.3.3 Convenzioni sullo Stack 62
3.4 Processore di riferimento 63
3.4.1 Caratteristiche 65
3.4.2 Memoria 67
3.4.3 Dispositivi ed Interfacce 67
3.4.3.1 Interruzioni 70
3.4.3.2 Dispositivi Interni – Timers 70

4 ERIKA 73
4.1 Hardware Abstraction Layer (HAL) 74
4.1.1 Tipi di dati e costanti 75
4.1.1.1 Tipi di dati 75
4.1.1.2 Costanti 76
4.1.2 Strutture dati 76
4.1.3 Primitive esportate verso lo strato di kernel 77
4.1.3.1 Manipolazione dei contesti 77
4.1.3.2 Definizione delle primitive 78
4.1.3.3 Gestione delle interruzioni 79
4.1.3.4 Utilità 79

4.2 Kernel Layer 80
4.2.1 Tipi di dati e costanti 80
4.2.1.1 Tipi di dati 80
4.2.1.2 Costanti 81
4.2.2 Strutture dati 81
4.2.3 Primitive utilizzabili dall'applicazione 82
4.2.3.1 Gestione dei thread 82
4.2.3.2 Gestione dei thread nei driver 83
4.2.3.3 Gestione delle risorse condivise 84
4.2.3.4 Installazione dei driver 84
4.2.3.5 Utilità 84
4.3 Stati di un thread 85
4.4 Struttura di un thread 86
4.5 Il thread dummy() 86
4.6 Macro per la configurazione del kernel 87

5 Porting di ERIKA su MPC566 89
5.1 Hardware Abstraction Layer (HAL) 89
5.1.1 HAL Monostack 91
5.1.2 HAL Multistack 92
5.1.3 Primitive comuni a Monostack e Multistack 92
5.1.4 Strutture Dati 93
5.2 Driver per Dispositivi Interni 95
5.2.1 Periodic Interrupt Timer 95
5.2.2 Real-Time Clock 96

6 Conclusioni 99

Elenco delle tabelle

3.1 Impostazione bit di CR0 34
3.2 Impostazione bit di CR1 35
3.3 Impostazione bit di CRn 35
3.4 Definizione dei Bit di XER 36
3.5 Impostazioni dei bit di MSR 39
3.6 Classificazione PowerPC delle Eccezioni 49
3.7 Dati scalari PowerPC 60
3.8 Utilizzo dei Registri 61
3.9 Stack Frame EABI 63

4.1 Macro per la configurazione del kernel 88

Elenco delle figure

3.1 Condition Register (CR) 33
3.2 Registro XER 36
3.3 Machine Status Register 38
3.4 Modalità di Traduzione 54
3.5 Formato di una coppia di registri BAT 56
3.6 Registri BAT – Descrizione 56
3.7 Organizzazione Array BAT 57
3.9 Registro di Segmento 58
3.8 Page Address Translation 59
3.10 Diagramma a Blocchi 64
3.11 Memory Map 68
3.12 Modulo USIU 69
3.13 Real-Time Clock 71
3.14 Periodic Interrupt Timer 71

4.1 Thread 86
4.2 dummy() 87

5.1 Contesto breve 90
5.2 Contesto completo 90
5.3 Strutture Dati HAL Multistack 94

Capitolo 1

Introduzione

1.1 Scopo del Lavoro

Lo scopo di questa tesi è quello di realizzare un sistema operativo real-time per sistemi embedded, affrontando pertanto le problematiche che caratterizzano lo sviluppo di questi sistemi, la cui importanza è sottolineata dalla loro continua diffusione.

La realizzazione di un intero sistema embedded è un problema piuttosto complesso che interessa diversi settori ingegneristici: l'elettronica, per quanto concerne la parte hardware del sistema, e l'informatica, per la parte software; a seconda del tipo di sistema che viene realizzato possono essere coinvolti anche i settori della meccanica, delle telecomunicazioni, ecc...

Nel corso di questa tesi ci si occuperà dello sviluppo della parte software del sistema, ovvero della progettazione e realizzazione di un sistema operativo real-time sul quale, successivamente, sarà possibile scrivere le applicazioni che si vogliono realizzare: tutto ciò verrà fatto dopo aver illustrato le caratteristiche dello hardware utilizzato.

Naturalmente quella che viene presentata è solo una delle varie modalità con cui vengono progettati sistemi embedded: in particolare, non è sempre vero che la parte hardware è assegnata a priori, anzi con la crescente complessità in termini di funzionalità che questi sistemi richiedono, negli ultimi anni si stanno diffondendo tecniche di progetto cosiddette di *Hw/Sw co-design*, in cui i vari settori ingegner-

istici di cui si parlava sopra sono chiamati ad una collaborazione più stretta di quanto invece avveniva in passato.

Il sistema embedded che qui viene alla fine realizzato utilizza come architettura hardware il microcontrollore **Motorola MPC566**: dopo aver descritto le caratteristiche di questo microprocessore, verrà presentato il sistema operativo **E.R.I.K.A.** che sarà successivamente implementato, analizzandone anche le prestazioni e i requirement. Il sistema sarà poi esteso con delle funzionalità che consentano poi all'applicazione di utilizzare i vari dispositivi hardware integrati nel microntrollore.

Quello che ci interessa mettere in evidenza con questa tesi, al di là del particolare sistema qui considerato, sono tutte quelle problematiche che riguardano il progetto della parte software di un sistema embedded generico.

1.2 Contenuto del Documento

Questa tesi si articola nei seguenti capitoli:

- nel **Capitolo 2** vengono introdotti i sistemi embedded, con le loro caratteristiche e le modalità di progetto; in particolare vedremo come queste stiano cambiando rispetto al passato, come conseguenza della crescente diffusione di questi sistemi, accompagnata dalla necessità di immettere sul mercato nuovi prodotti in tempi molto brevi ed a costi bassi.

- Il **Capitolo 3** presenta la famiglia di microcontrollori Motorola MPC5xx, con una descrizione del particolare microcontrollore che è stato utilizzata nello sviluppo di questa tesi, cioè il microcontrollore MPC566.

- Nel **Capitolo 4** vengono illustrate le caratteristiche del nucleo del sistema operativo che è stato sviluppato, con particolare riferimento alla API.

- Nel **Capitolo 5** vengono descritti i concetti principali che sono stati utilizzati per implementare il nucleo del sistema operativo e le funzionalità che sono state aggiunte a quest'ultimo per permettere alle applicazioni di utilizzare in modo semplice ed efficiente i vari dispositivi hardware presenti sul microcontrollore.

- In conclusione nel **Capitolo 6** vengono riportate le conclusioni sul lavoro svolto.

Nota

In questo documento non sono presenti nè il codice sorgente dell'applicazione nè altri riferimenti alle reali implementazioni delle funzioni. Questo perchè ERIKA Enterprise è coperta da copyright da parte di Evidence S.r.l.

Capitolo 2

I Sistemi embedded

Non esiste una definizione rigorosa ed universalmente riconosciuta di sistema embedded, tuttavia viene incluso in questa categoria ogni apparato elettronico che svolge una specifica funzione all'interno (da qui il termine embedded, "immerso") di un sistema più vasto che interagisca con l'ambiente circostante.

Negli ultimi anni questi sistemi si stanno sempre più diffondendo in ogni ambito applicativo, alcuni dei quali sono:

- ▷ i prodotti di consumo (video camere, telefoni cellulari, lettori CD, ecc...);
- ▷ gli autoveicoli: sistemi di sicurezza attiva (controllo della frenata, della trazione, della tenuta di strada), di sicurezza passiva (ABS, airbags), gestione elettronica del motore, ecc...
- ▷ i mezzi di trasporto su rotaia, l'aviazione (sia civile che militare), le navi, ecc...
- ▷ le industrie, per quanto riguarda ad esempio il controllo dei processi.

Ai suddetti classici ambiti in cui si trovano sistemi embedded, se ne aggiungono altri che si stanno diffondendo recentemente:

- ▷ nuove applicazioni multimediali (ad esempio server per video conferenze, ecc...);

- prodotti di uso comune (PDA[1], game boxes interattivi, ecc...).

Dobbiamo osservare che la diffusione di questi sistemi è in continua evoluzione, al punto che il numero di microcontrollori venduti ogni anno per la realizzazione di sistemi embedded è attualmente dell' ordine di qualche miliardo di unità mentre il numero di processori venduti per i sistemi general purpose (tra cui i Personal Computer) è intorno alle centinaia di milioni di unità. In termini monetari, i due volumi di affari al momento sono pressochè uguali, dato che mentre i processori utilizzati ad esempio nei PC sono dispositivi molto sofisticati e dunque costosi, nei sistemi embedded vengono utilizzati molto spesso dei semplici microcontrollori ad 8, 16 ed al più, in alcuni casi, a 32 bit.

2.1 Caratteristiche

Ogni sistema embedded presenta un insieme di caratteristiche che impongono dei vincoli che devono dunque essere rispettati in fase di progettazione del sistema stesso. Elenchiamo di seguito queste caratteristiche:

- **Applicazione specifica**: il sistema è progettato per svolgere una precisa funzione nota a priori, a differenza di quanto accade per i sistemi general purpose. Questo consente varie ottimizzazioni: si potrebbe pensare ad esempio di realizzare praticamente tutto il sistema ad hardware a favore di una elevata efficienza; tuttavia questa soluzione non viene quasi mai adottata, perchè oltre agli elevati costi, non consente di avere la flessibilità che serve per apportare aggiornamenti, differenziare il prodotto, ecc...

- **Reattività e real-time**: il sistema compie le proprie elaborazioni in seguito a certi segnali che arrivano attraverso i sensori dal mondo esterno, operando su quest'ultimo mediante degli attuatori. Questo è il concetto di reattività che molto spesso si affianca anche al fatto che il sistema presenti dei requisiti real-time, ovvero, che il risultato dipenda non solo dal valore di ritorno ma anche dal tempo in cui i valori sono prodotti. A livello di design questo si traduce spesso nella presenza di *deadline* che il sistema deve rispettare

[1] Personal Digital Assistants

nel compiere certe operazioni. Uno dei problemi che il progettista si trova ad affrontare è di poter determinare il caso peggiore[2] per poter verificare se i vincoli temporali imposti vengono rispettati.

- **Sistema distribuito**: è molto frequente la situazione in cui sono presenti più microcontrollori che comunicano tra loro atraverso opportuni *links*. Questo avviene essenzialmente per ragioni economiche: risulta infatti meno dispendioso utilizzare ad esempio 4 microcontrollori ad 8 bit interconnessi, piuttosto che un unico microcontrollore a 32 bit. Ci sono tuttavia altre ragioni per adottare questa soluzione, come dover gestire processi critici su processori distinti oppure fare fronte alla situazione in cui il sistema controllato dal sistema embedded sia fisicamente distribuito.

- **Architettura eterogenea**: molto spesso questi sistemi sono costituiti da processori diversi (ad esempio può essere presente un DSP ed un microcontrollore) e questo garantisce una maggiore flessibilità di progettazione e la possibilità di rispettare in modo più semplice i vincoli che l'applicazione impone.

- **Condizioni operative proibitive**: può capitare che questi sistemi si trovino ad operare in condizioni particolari come di intenso calore oppure spesso ci possono essere anche situazioni in cui il sistema deve essere protetto da vibrazioni, luce, acqua, ecc... Tutto ciò naturalmente introduce dei vincoli che devono essere tenuti in considerazione durante il progetto.

- **Affidabilità**: ci sono applicazioni in cui deve essere garantito il corretto funzionamento del sistema anche in seguito al verificarsi di guasti: possiamo pensare a tutti quei sistemi di controllo presenti ad esempio su un autoveicolo, oppure nei processi industriali, ecc... Uno degli obiettivi del progettista è quello di *costruire un sistema affidabile utilizzando componenti inaffidabili*, con una minima ridondanza (soprattutto in termini di hardware) ed a basso costo.

[2] La difficoltà di questa operazione è tanto maggiore quanto più la complessità del sistema aumenta

- **Peso e dimensioni opportune, basso consumo energetico, emissioni ridotte**: sono vincoli che, in misura diversa e dipendente dalla particolare applicazione, caratterizzano la maggior parte dei sistemi.

2.2 Standard e tecniche di progettazione

In questo contesto vogliamo solo dare un'idea delle metodologie di progetto dei sistemi embedded, tenendo conto che si tratta di un'area di ricerca che recentemente sta interessando molto le aziende produttrici: ciò è dovuto essenzialmente al fatto che la crescente complessità di questi sistemi e la necessità di avere dei *time-to-market* sempre minori, rendono le attuali tecniche di progettazione inadeguate.

L'aspetto fondamentale nella progettazione di un sistema embedded è l'implementazione di uno specifico insieme di funzionalità, rispettando i vincoli imposti dalle caratteristiche che il sistema stesso deve avere: la scelta di una particolare architettura determina se una certa funzione verrà realizzata utilizzando componenti hardware oppure via software. In particolare, negli ultimi anni le caratteristiche che questi sistemi presentano sono diventate tali per cui i progettisti inevitabilmente sono dovuti ricorrere a tecniche implementative che presentassero una certa flessibilità, in modo tale da poter modificare il prodotto in tempi rapidi.

Se consideriamo che i cicli di produzione dello hardware sono più costosi e richiedono più tempo per essere portati a termine, si capisce l'attuale tendenza a realizzare questi sistemi mediante implementazioni basate sul software piuttosto che sullo hardware. Questo comunque introduce dei problemi che erano trascurabili fino a quando l'implementazione era prevalentemente hardware (in particolare ci riferiamo al fatto che il software embedded presenta caratteristiche diverse dal software con il quale vengono realizzate applicazioni per PC), ovvero:

- la **correttezza** è nella maggior parte dei casi un requisito fondamentale e critico in termini di sicurezza: ci sono applicazioni (ad esempio quelle di controllo) in cui un eventuale bug potrebbe portare danni anche catastrofici;
- l'**occupazione di memoria** deve essere ridotta al minimo: ricordiamo che tipicamente questi sistemi utilizzano semplici microcontrollori con limitate

quantità di memoria. Questo problema in pratica non c'è nelle applicazioni per PC dove la quantità di memoria presente è così estesa da non costituire quasi mai un limite per lo sviluppo dell'applicazione;

- il **rispetto dei vincoli temporali** è fondamentale perchè il sistema svolga correttamente le funzionalità per le quali è stato progettato;
- gli **strumenti per il debugging** sono di qualità spesso scadente: in alcuni casi, si devono fare dei test in modo empirico e non esaustivo, non essendo possibile utilizzare strumenti che invece sono disponibili quando si scrivono applicazioni non embedded.

Questi sono alcuni dei problemi che il progettista embedded deve tenere in considerazione e che evidenziano uno stato attuale di crisi per quanto riguarda la progettazione di questo tipo di software.

Dobbiamo anche osservare come l'adozione di tecniche di implementazione flessibili abbiamo spinto le industrie produttrici di circuiti integrati ad introdurre sul mercato dei chip in grado di essere utilizzati in varie tipologie di progetti, nell'ottica di ammortizzare i costi di sviluppo su un grande quantitativo di unità.

Dall'altro lato, molte aziende che producono sistemi embedded impiegano risorse nella ricerca di metodologie di progetto che prendano in considerazione tutti questi aspetti.

Un caso emblematico è quello delle industrie automobilistiche che nel corso degli ultimi anni si sono trovate di fronte alla necessità di utilizzare questi sistemi non più per i prodotti di nicchia bensì per i veicoli prodotti su larga scala. Infatti oggi, tali sistemi sono praticamente presenti su tutte le auto, dalla gestione elettronica del motore all'ABS, dal controllo della stabilità agli airbags e ad altri apparati che verranno via via introdotti per aumentare le funzionalità del prodotto.

Attualmente questi sistemi incidono per circa il 15-30% sul costo totale di un veicolo: ciò spiega perchè le aziende automobilistiche hanno investito nella ricerca di soluzioni implementative che possano rispondere alle nuove esigenze di mercato in questo ambito, riducendo quanto più possibile i costi: su queste premesse è stato creato lo standard *OSEK/VDX*.

Sono in corso anche degli studi, sia a livello accademico che industriale (molto spesso i due ambiti si trovano a collaborare) che hanno l'obiettivo di introdurre

delle metodologie di progetto innovative che interessano ogni aspetto della progettazione. Tra i vari studi, ne citiamo uno effettuato dalla Gigascale Silicon Research Center (GSRC) e che viene indicato con *Platform-Based Design* [ASV01, KK00].

Nel seguito introduciamo brevemente sia lo standard OSEK, sia lo studio della GSRC, con l'obiettivo di dare un' idea, seppur vaga, di quelle che sono le tendenze attuali e le direzioni in cui la ricerca si sta muovendo per trovare soluzioni efficienti al problema della progettazione dei sistemi embedded.

2.2.1 OSEK / VDX

Il termine OSEK è un acronimo tedesco che sta per "Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug", ovvero in inglese "open systems and the corresponding interfaces for automotive electronics" ; invece, VDX sta per " Vehicle Distributed eXecutive".

Si tratta di un progetto proposto inizialmente dall'industria automobilistica tedesca ed al quale ha successivamente aderito la maggior parte delle aziende automobilistiche che si trovavano di fronte sostanzialmente a due problemi:

- **costi ricorrenti molto elevati** nello sviluppo e nella gestione di aspetti non legati alla funzionalità delle unità di controllo;
- **incompatibilità tra le varie unità di controllo** realizzate da produttori diversi, a causa delle differenti interfacce e protocolli utilizzati da questi ultimi.

Con l'introduzione di OSEK si è cercato di risolvere tali problemi cercando di supportare la *portabilità* e la *riusabilità* del software relativo ad una certa applicazione. Per ottenere questi obiettivi tale standard prevede di:

- specificare le interfacce relative al sistema operativo real-time, alla gestione della rete ed alla comunicazione, in modo quanto più possibile astratto ed indipendente dall'applicazione;
- specificare un' interfaccia utente che sia indipendente dallo hardware e dalla rete;

- ▷ avere un' architettura progettata in modo efficiente, ovvero tale da avere una funzionalità configurabile e scalabile, che renda possibile un perfetto adattamento dell'architettura stessa all'applicazione che viene considerata;
- ▷ verificare la funzionalità e l'implementazione di prototipi.

Tutto ciò secondo gli sviluppatori di OSEK dovrebbe portare ad una serie di vantaggi, tra i quali abbiamo:

- ▷ una riduzione dei costi e del tempo di sviluppo di un sistema;
- ▷ una migliore qualità del software delle unità di controllo prodotte dalle varie aziende;
- ▷ delle unità di controllo che presentino un'interfaccia con caratteristiche standard, pur avendo architetture tra loro differenti;
- ▷ la possibilità di utilizzare sequenzialmente le varie unità distribuite nel veicolo, senza richiedere l'aggiunta di altri componenti hardware, migliorando le prestazioni globali del sistema;
- ▷ nessun vincolo implementativo, dato che nelle spefiche dello standard non viene fatto riferimento ad aspetti riguardanti l'implementazione.

Lo standard OSEK definisce i requisiti riguardanti:

- ▷ il sistema operativo;
- ▷ i dati scambiati all'interno di un' unità di controllo e tra unità diverse;
- ▷ la strategia di gestione della rete che interconnette le varie unità.

Osservazione:

> il sistema operativo ERIKA a cui facciamo riferimento nel corso di questa tesi e che viene utilizzato per realizzare il nostro sistema embedded, utilizza lo standard OSEK/VDX .

Naturalmente essendo molte le soluzioni allo studio non c'è uniformità di giudizio su di esse. Ad esempio, verso lo stesso OSEK vengono sollevate alcune critiche, tra cui:

- ▷ il fatto che le specifiche riguardanti i driver delle periferiche di I/O non siano definite, consente un riutilizzo del software solo a livello di codice sorgente, mentre se fossero aggiunte allo standard la definizione dei sottosistemi di I/O e l'utilizzo di un certo set di istruzioni, sarebbe possibile aumentare la percentuale di codice portabile che viene scritta;

- ▷ il fatto che venga complementamente specificato l'algoritmo di scheduling sembra un vincolo eccessivo dato che si otterrebbe un riutilizzo del software anche con una parziale specifica dell'algoritmo stesso.[3]

2.2.2 Platform-Based Design

Partendo dal presupposto che l'implementazione dei sistemi embedded sarà sempre più basata sul software, diventa fondamentale, al fine di una riduzione dei costi, il concetto di riutilizzo del software stesso. Per ottenere questo è necessario che l'architettura hardware sul quale il sistema viene implementato sia la stessa (almeno entro un certo grado di parametrizzazione) per una certa famiglia di applicazioni in cui si ha la possibilità di sfruttare software già scritto.

Una architettura che consente il riutilizzo del software e che generalmente per questi sistemi è costituita da un nucleo programmabile, delle periferiche di I/O e dalla memoria, viene detta *piattaforma hardware*.

Per raggiungere l'obiettivo prefissato, il concetto di piattaforma hardware deve essere ampliato mediante uno strato di software costituito essenzialmente dalle seguenti parti:

- ▷ un nucleo real-time che astrae il nucleo programmabile e la memoria;

- ▷ i driver che dunque astraggono le periferiche di I/O.

[3] D'altra parte la specifica attuale di fatto impone la struttura dati da utilizzare.

Tale strato di software viene detto *piattaforma software* ed è quello che si interfaccia con l'applicazione attraverso la API (Application Program Interface) detta anche Programmers Model.

Osserviamo che con i metodi classici, il progetto può essere portato avanti seguendo uno dei seguenti approcci:

- **Top-down**: si parte dalle specifiche dell'applicazione e questo individua un' insieme di architetture adatte per implementare l'applicazione stessa (e tra queste viene scelta quella che minimizza i costi): a questo punto il progettista deve scegliere la piattaforma hardware. Tale scelta si basa su due parametri: la frequenza di clock della CPU e la quantità di memoria presente.

- **Bottom-up**: viene fissata un'architettura in grado di implementare un certo insieme di applicazioni di solito definite in modo abbastanza vago; questo approccio è utilizzato dalle aziende produttrici di circuiti integrati, con lo scopo di aumentare il volume di piattaforme hardware (di uno stesso tipo) prodotte.

  Nota:

  Può capitare che la piattaforma scelta sia sovraddimensionata rispetto alle attuali necessità dell'applicazione e che dunque alcune delle potenzialità offerte dalla piattaforma hardware stessa non vengano utilizzate. Questo fatto che apparentemente potrebbe sembrare negativo, in realtà fornisce la possibilità di creare applicazioni simili riutilizzando lo stesso hardware oppure di effettuare gli aggiornamenti di quelle esistenti.

Invece, nel caso della metodologia "platform-based design" viene introdotto l'approccio "meet-in-the-middle" che si basa sul concetto di *piattaforma di sistema*: questa può essere pensata come un unico strato costituito dalla API e dallo strato sottostante che comprende la piattaforma hardware. Questo metodo di progettazione si basa sulla ricerca di un punto di compromesso: da un lato vogliamo avere una API quanto più possibile astratta, ma questo comporta avere un insieme

di piattaforme più ampio e dunque aumenta la difficoltà nel trovare la soluzione ottima.

Quindi utilizzando questa metodologia il punto di partenza del progettista è quello di determinare un insieme di vincoli che determinino la piattaforma hardware per quella particolare applicazione.

Non ci addentriamo ulteriormente nell'argomento, ma facciamo notare che su queste ed altri basi teoriche sono stati realizzati vari ambienti di sviluppo per la progettazione e lo sviluppo di sistemi embedded, tra i quali abbiamo:

- POLIS
- Ptolemy
- COSY
- METROPOLIS

2.3 Considerazioni e prospettive

In passato (ed in alcuni casi anche attualmente!), la produzione di sistemi embedded avveniva in modo quasi artistico, con il progettista che tipicamente aveva a disposizione un certo componente hardware sul quale doveva implementare un sistema operativo ed utilizzando le funzionalità da questo offerte, scrivere l'applicazione per la quale il sistema era stato progettato. Nel caso in cui venisse utilizzata una nuova architettura hardware il progettista provvedeva a riscrivere quelle parti del sistema operativo e del codice necessarie per fare il porting dell'applicazione su tale architettura.

C'è poi da tener conto di un aspetto importante, ovvero degli strumenti per lo sviluppo ed il debugging di cui il progettista embedded poteva usufruire: nella maggior parte dei casi la qualità di questi non è neppure paragonabile a quella degli equivalenti strumenti destinati alla realizzazione di software standard (non embedded).

Questo modo di progettare, che fin quando il numero di sistemi da produrre è molto limitato può avere l'aspetto positivo di sfruttare le capacità del progettista,

per ottimizzare il sistema trovando delle soluzioni ad hoc per il caso specifico, presenta vari aspetti negativi:

- ▷ bug presenti nel software: non vengono di solito effettuati test esaustivi a causa della carenza di strumenti e talvolta anche dello hardware che molto spesso non fornisce il supporto adeguato. Ciò comporta il rischio che il sistema venga utilizzato con una certa percentuale di presenza di guasti;
- ▷ non riutilizzabilità del codice: il progettista pensa alla specifica funzione che il sistema deve realizzare, senza preoccuparsi di astrarre il problema in modo da poter riutilizzare parte del codice per applicazioni della stessa natura;
- ▷ tempi di sviluppo e costi elevati: se in passato erano accettabili dei time-to-market di qualche mese, oggi si richiede al massimo qualche settimana.

In un futuro prossimo questi sistemi acquisteranno un ruolo sempre più determinante nel volume di affari delle aziende coinvolte in questo settore che, come già sta avvenendo, investiranno risorse per la ricerca di metodologie di progetto che consentano un drastico abbattimento dei costi ed una riduzione dei tempi di immissione dei prodotti sul mercato, offrendo una qualità (auspicabilmente) superiore.

Come abbiamo accennato sopra, la tendenza è quella di andare verso metodologie di progetto basate su modelli formali e standardizzati: questa permetterà l'introduzione di nuovi e più sofisticati tools che assisteranno il progettista; in particolare con ogni probabilità si tenderà a creare degli strumenti di sintesi automatica in modo tale che l'intervento umano sia essenzialmente confinato ad un livello di astrazione superiore rispetto a quanto era chiamato a fare nel passato, occupandosi di fornire le specifiche (mediante formalismi matematici, ad esempio) ed al più di scegliere l'architettura hardware sulla quale realizzare l'applicazione. Da questo punto in poi il codice verrà generato automaticamente e sarà tanto più ottimizzato quanto più i modelli del sistema saranno in grado di descrivere correttamente le caratteristiche del sistema stesso.

Osserviamo infine che proprio l'automatizzazione nella produzione di software embedded è un settore ancora poco sviluppato ma che proprio per questo,

data la sua importanza, con molta probabilità sarà molto remunerativo in un futuro prossimo.

Alla luce di quanto abbiamo finora detto, durante la fase di progettazione del sistema operativo che viene realizzato nel corso di questa tesi, si sono tenuti in considerazione in modo particolare i seguenti aspetti:

- ▷ la minima occupazione di memoria: si è cercato specialmente di ridurre al minimo la quantità di RAM necessaria (RAM *footprint*) dato che questa è una risorsa ancora più critica della memoria ROM sui microcontrollori tipicamente utilizzati nei sistemi embedded;
- ▷ la portabilità verso architetture differenti: in questo modo si riducono i tempi di sviluppo qualora il sistema dovesse essere utilizzato su un processore diverso. Sono inoltre stati pensati degli strumenti ausiliari per aumentare la portabilità, come ad esempio **RT-Druid** [http://www.evidence.eu.com/];
- ▷ la riutilizzabilità del software: si è cercato di fornire una API il più possibile indipendente dal particolare processore sottostante, in modo tale che un'applicazione scritta per una certa architettura restasse praticamente invariata al variare di quest'ultima;
- ▷ la modularità del kernel: in questo modo il sistema operativo si adatta alle caratteristiche dell'applicazione, in modo tale da avere nel file immagine solo la parte del nucleo strettamente necessaria per gli scopi dell'applicazione;
- ▷ garanzia del rispetto dei vincoli temporali: il nucleo è stato pensato affinchè possa garantire il corretto scheduling dei task, siano essi periodici che aperiodici o sporadici;
- ▷ offrire buone prestazioni: i tempi di risposta del sistema devono essere i più brevi possibile.

Osserviamo che alcune degli obiettivi che ci poniamo sono tra loro contrastanti: da un lato abbiamo la necessità di una certa astrazione al fine di ottenere portabilità, riusabilità e modularità, mentre dall'altro una implementazione molto meno astratta favorirebbe le prestazioni. Si tratta pertanto di trovare il giusto compromesso.

Capitolo 3

Microcontrollore Motorola MPC5xx

La famiglia di microcontrollori Motorola MPC5xx fa parte della famiglia architetturale PowerPC implementata da Motorola.

In questo capitolo verranno descritte le caratteristiche generali del microcontrollore, poi quelle essenziali della famiglia PowerPC ed infine daremo uno sguardo alla specifica implementazione utilizzata nello sviluppo di questo progetto.

3.1 Caratteristiche generali

La famiglia MPC5xx è una famiglia di implementazione dei microcontrollori RCPU[1] di Motorola. RCPU è un'implementazione a 32-bit dell'architettura PowerPC, single issue. Il processore fornisce uno spazio di indirizzamento a 32-bit, interi mappati su 8, 16 o 32-bit e floating-point mappati su 32 o 64-bit.

La RCPU integra quattro unità di esecuzione: una Integer Unit (**IU**), una Floating-point Unit (**FPU**), una Load/Store Unit (**LSU**) ed una Brach Processing Unit (**BPU**). Il processore può inviare (*issue*) una istruzione sequenziale ad una di queste unità per ogni ciclo di clock. In aggiunta il processore tenta di valutare anticipatamente le condizioni di salto in modo da eseguire salti simultaneamente all'esecuzione delle istruzioni sequenziali. Per incrementare le prestazioni l'esecuzione viene eseguita out-of-order, ma il processore la fa apparire come sequenziale.

[1] Risc Central Processing Unit

I microcontrollori basati sulla RCPU (**MCU**) includono nel chip una memoria flash di 4-Kbytes ed una I-cache set associative a 2 vie.

Per ulteriori informazioni consultare [RCP99].

3.1.1 Architettura PowerPC

L'architettura PowerPC, sviluppata unitamente da Motorola, IBM e Apple Computer, è basta sull'architettura POWER implementata dalla famiglia di computer RS/6000.

L'architettura PowerPC definisce le seguenti carattestiche generali:

- ▷ 32 registri generali per operazioni sugli interi (**GPR**s) e 32 registri per le operazioni sui floating-point (**FPR**s).
- ▷ Istruzioni per le operazioni tra la memoria ed i registri generali, sia quelle per operandi interi sia quelle per operandi in virgola mobile.
- ▷ Istruzioni codificate su numero di bit uniformi per semplificare il pipelining e l'esecuzione parallela del meccanismo del dispatch delle istruzioni.
- ▷ Un utilizzo non distruttivo dei registri nell'esecuzione delle operazioni aritmetiche.
- ▷ Un modello *preciso* per le eccezioni.
- ▷ Il supporto per le operazioni in virgola mobile che include lo standard IEEE-754.
- ▷ La possibilità di eseguire operazioni floating-point sia in singola che doppia precisione.
- ▷ Instruzioni di livello utente per salvare, flushare ed invalidare dati nella cache interna.
- ▷ Definizione di un modello della memoria che consenta un accesso debolmente ordinato (questo consente alle operazioni sul bus di essere riordinate dinamicamente).

- Supporto per cache istruzioni e cache dati, architettura Harvard, e cache unificate.
- Supporto sia per big- che per little-endian mode.

I processori che aderiscono allo standard PowerPC funzionano a due livelli di privilegio:

- **Supervisor Mode** –Utilizzato esclusivamente dal sistema operativo.
- **User Mode** –Utilizzato sia dal sistema operativo che dalla applicazioni software.

Questi due livelli creano una netta distinzione tra tutte le funzionalità e le risorse che sono messe a disposizione dal processore.

3.1.2 I livelli dell'architettura

L'architettura è definita in tre livelli che corrispondono a tre ambienti di programmazione. Questa stratificazione dell'architettura fornisce una maggiore flessibilità al sistema, consentendo un maggior grado di compatibilità software tra un'ampia gamma di implementazioni.

I tre livelli sono definiti in questo modo:

- **User Instruction Set Architecture (UISA)** –UISA definisce il livello di architettura a cui il software user-level deve essere conforme. UISA definisce così il set base di istruzioni per l'utente, i registri a cui può accedere, le convenzioni per la memorizzazione degli operandi in virgola mobile ed il modello delle eccezioni visto dall'utente. Le risorse definite a questo livello possono essere accedute in modalità *user*.
- **Virtual Environment Architecture (VEA)** –A questo livello sono definite delle funzioni addizionali che esulano dai requirement software del livello utente. VEA descrive il modello della memoria in un ambiente in cui molti dispositivi vi possono accedere e definisce i servizi *time base* dalla prospettiva dell'utente. Le risorse definite a questo livello possono essere accedute in modalità *user*.

- ▷ **Operating Environment Architecture (OEA)** –Definisce le risorse di livello supervisore tipicamente richieste dal sistema operativo. OEA definisce il modello di gestione della memoria, i registri supervisor level, i requisiti di sincronizzazione ed il modello delle eccezioni. Le risorse definite a questo livello possono essere accedute esclusivamente in modalità *supervisor*.

Le implementazioni che sono conformi a OEA lo sono anche a UISA e VEA.

Tutte le architetture PowerPC sono conformi a UISA, offrendo così compatibilità tra tutte le applicazioni. Esistono invece varie versioni di OEA e VEA dipendenti dalle specifiche implementazioni dello standard PowerPC.

3.2 RCPU

Si tratta di un processore RISC che aderisce allo standard PowerPC, per cui analizzeremo le sue caratteristiche seguendo i livelli architetturali definiti dallo standard. Per maggiori dettagli vedere [RCP99].

3.2.1 Registri

L'architettura definisce operazioni registro-registro per le istruzioni computazionali. Gli operandi sorgenti di queste istruzioni sono acceduti dai registri architetturali o forniti come operandi immediati integrati nel codice dell'istruzione.

Il formato a tre registri delle istruzioni consente di specificare un destinatario diverso dai due operandi sorgenti[2]. Le istruzioni di load/store sono le uniche che fanno interagire i registri e la memoria.

Tutti i registri del processore vengono catalogati in insiemi in base al privilegio richiesto per accedervi. Questa divisione di privilegi consente al sistema operativo di avere un controllo completo sull'ambiente dell'applicazione, e consente inoltre una maggiore protezione della macchina.

[2] Utilizzo non distruttivo dei registri

3.2.1.1 Registri UISA

I registri di questo insieme possono essere acceduti sia da istruzioni *user-level* che da quelle *supervisor-level*.

I registri generali e quelli floating-point vengono acceduti come operandi delle istruzioni. L'accesso ai registri può avvenire in modo esplicito, cioè inserendo il registro nel codice dell'istruzione, o implicito, cioè come parte dell'esecuzione di un'istruzione.

Fanno parte di questa categoria i seguenti registri:

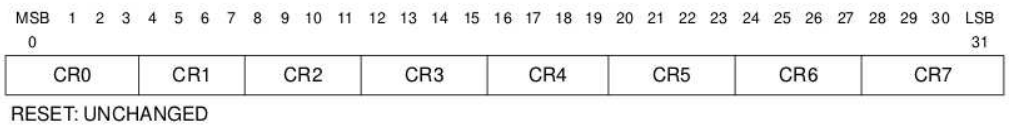
General-purpose Registers (GPRs) I dati interi vengono manipolati nei 32-GPR del processore. Questi vengono acceduti come operandi sorgenti e destinatari delle istruzioni. Alcuni di questi registri hanno un significato particolare come vedremo in 3.3.

Floating-point Registers (FPRs) L'architettura fornisce 32 registri floating-point a 64-bit. Questi registri vengono acceduti come sorgenti e destinatari delle istruzioni floating-point e supportano sia la modalità a singola che a doppia precisione.

Condition Register (CR) I 32-bit di questo registro riflettono il risultato di alcune operazioni e forniscono un meccanismo per implementare test e salti.

I bit di CR vengono raggruppati in 8 campi da 4-bit, CR0–CR7, come mostrato in tabella 3.1.

Figura 3.1: Condition Register (CR)



I bit di CR possono essere impostati utilizzando le seguenti istruzioni:

- ▷ Specifici campi di CR possono essere impostati dai GPR utilizzando **`mtcrf`**[3].
- ▷ Il contetnuto di un campo di CR può essere copiato in un altro tramite **`mcrf`**[4]. Tutti gli altri campi rimangono inalterati.
- ▷ Il contenuto di XER[0–3] può essere copiato in un campo di CR utilizzando **`mcrxr`**[5].
- ▷ Uno specifico campo di FPSCR viene copiato in uno specifico campo di CR utilizzando **`mcrfs`**[6].
- ▷ Le istruzioni logiche eseguono operazioni logiche su specifici campi di CR.
- ▷ CR0 può essere il risultato implicito di una istruzione su interi.
- ▷ CR1 può essere il risultato implicito di una istruzione su floating-point.
- ▷ Uno specifico campo di CR può indicare il risultato di una istruzione di confronto sia su interi che su floating-point.

Definizione di CR0 Per tutte le istruzioni intere, quando CR è impostato per riflettere il risultato dell'operazione, i primi tre bit di CR0 sono impostati con un confronto algebrico del risultato con zero; il quarto bit è copiato da XER[SO]3.2.1.1. I bit di CR0 vengono interpretati come in tabella 3.1:

Tabella 3.1: Impostazione bit di CR0

Bit	**Descrizione**
0	Negativo (LT)
1	Positivo (GT)
2	Zero (EQ)
3	Summary Overflow (SO)

[3]Move To Condition Register Field
[4]Move Condition Register Field
[5]Move to Condition Register XER
[6]Move to CR Floating-point Status

Definizione di CR1 Per tutte le istruzioni floating-point, quando CR è impostato per riflettere il risultato dell'operazione, CR1 viene copiato da FPSCR[0–3] ed indica lo stato delle eccezioni floating-point. I bit di CR1 vengono interpretati come in tabella 3.2:

Tabella 3.2: Impostazione bit di CR1

Bit	**Descrizione**
0	Floating-point Exception (FX)
1	Floating-point Enabled Exception (FEX)
2	Floating-point Invalid Exception (VX)
3	Floating-point Overflow Exception (OX)

Definizione di CR*n* –istruzione di confronto Per le istruzioni di confronto, quando uno specifico campo di CR è impostato per riflettere il risultato dell'operazione, i bit di quel campo vengono interpretati come in tabella 3.3:

Tabella 3.3: Impostazione bit di CRn

Bit	**Descrizione**
0	Minore (`r`A < `r`B) o Minore Floating-point (`f`A < `f`B)
1	Maggiore o Maggiore Floatig-point
2	Uguale o Uguale Floating-point
3	Summary Overflow o Floating-point non ordinato

Floating-point Status and Control Register (FPSCR) É un registro a 32-bit i quali indicano le situazioni seguenti:

- ▷ Registrano le eccezioni generate dalle operazioni floting-point.

- ▷ Registrano il tipo del risultato prodotto da un'operazione floating-point.
- ▷ Controllano la modalità di arrotondamento utilizzato dalle operazioni floating-point.
- ▷ Abilitano o disabilitano la segnalazione delle eccezioni floating-point.

I bit 0–23 sono bit di stato e sono, quindi, di sola lettura mentre i bit 24–31 sono bit di controllo, che possono essere manipolati via software.

XER register (XER) La definizione dei bit di questo registro è basato sull'operazione di un'istruzione considerata completa, non sui suoi risultati intermedi.

In questo registro vengono registrati i risultati delle operazioni su interi che possono produrre *overflow* e *carry*.

Figura 3.2: Registro XER

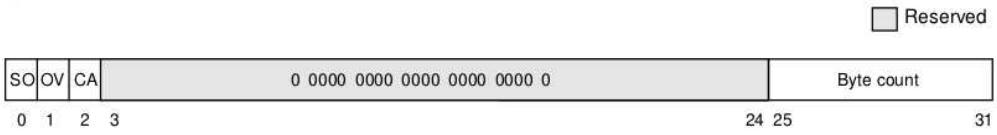


Tabella 3.4: Definizione dei Bit di XER

Nome	**Descrizione**
SO	Summary Overflow: questo bit viene settato quando l'esecuzione di un'istruzione set l'overflow bit di questo registro.
OV	Overflow: viene settato per segnalare un overflow durante l'esecuzione di un'istruzione.
CA	Carry: settato durante l'esecuzione di un'istruzione algebrica se si verifica un carry.
Byte Count	Specifica il numero di byte trasferiti da una load multipla o una store multipla.

Link Register (LR) É un registroo a 32-bit che fornisce l'indirizzo di un salto per le istruzioni **`bclr`***x*[7], e nel caso di branch con aggiornamento del link, può essere utilizzato come *ancora*, cioè per mantenere l'indirizzo dell'istruzione successiva a quella di salto.

LR può inoltre essere acceduto tramite l'utilizzo delle istruzioni **`mtspr`**[8] e **`mfspr`**[9].

Count Register (CTR) É un registro a 32-bit che può mantenere un contatore di loop che viene decrementato durante l'esecuzione delle istruzioni di salto le quali hanno un codice appropriato nel campo BO dell'istruzione. Se il valore di CTR è 0 prima del decremento sarà 0xFFFF_FFFF dopo l'operazione.

CTR può inoltre essere utilizzato come indirizzo destinatario di un salto tramite l'utilizzo dell'istruzione **`bcctr`***x*[10].

3.2.1.2 Registri VEA

VEA definisce registri in aggiunta a quelli definiti da UISA, accessibili da livello user, introducendo il servizio time base (TB), una struttura a 64-bit che consiste in due registri a 32-bit –*time base upper* (TBU) e *time base lower* (TBL). É da notare che questi registri, a livello user, possono solo essere letti.

3.2.1.3 Regitsri OEA

OEA completa la trattazione dei registri. Fanno parte di questa categoria, oltre ai registri degli altri livelli architetturali, tutti gli *special-purpose register* utilizzati per la configurazione e la gestione della macchina. Per questo motivo i registri vengono così catalogati:

- ▷ Registri di configurazione:
  - ⇝ Machine Status Register (MSR)

[7] Branch Conditional to Link Register
[8] Move To Special Purpose Register
[9] Move From Special Purpose Register
[10] Brach Conditional to Count Register

  - ⇝ Processor Version Register (PVR)

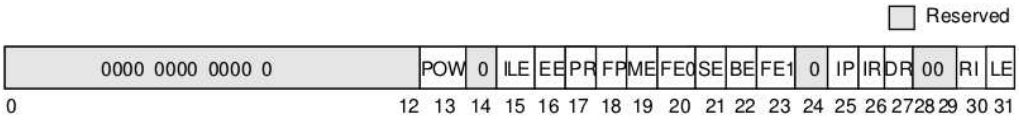
- ▷ Registri di gestione della memoria.
  - ⇝ Block Address Translation (BAT) registers
  - ⇝ SDR1
  - ⇝ Segment Registers (SR)

- ▷ Registri per la gestione delle eccezioni.
  - ⇝ Data Address Register (DAR)
  - ⇝ Machine Status Save/Restore Register 0 (SRR0)
  - ⇝ Machine Status Save/Restore Register 0 (SRR0)

- ▷ Registri vari:
  - ⇝ Time Base (TB)
  - ⇝ Decrementer Register (DEC)

Machine Status Register (MSR) É un registro a 32-bit che definisce lo stato del processore. Quando viene sollevata un'eccezione i bit di MSR vengono modificati come definito dalla particolare eccezione. Il registro può inoltre essere modificato dall'istruzione **`mtmsr`**[11] e può essere letto tramite la **`mfmsr`**[12].

Figura 3.3: Machine Status Register



I bit di maggior interesse, ed il loro significato, sono riassunti in tabella 3.5. Per una visione completa dei bit di MSR vedere [RCP99].

[11] Move To MSR
[12] Move From MSR

Tabella 3.5: Impostazioni dei bit di MSR

Bit	**Nome**	**Descrizione**
16	EE	External Interrupt Enabled: ▷ se settato questo bit indica che il processore è abilitato a riconoscere interruzioni esterne ed eccezioni di decremento; ▷ se posto uguale indica che le eccezioni vengono mascherate, e quindi il loro riconoscimento è ritardato.
17	PR	Privilege Level: indica il livello di privilegio dell'istruzione in esecuzione: ▷ se uguale a 0 indica che il procesore può eseguire sia istruzioni user- che supervisor-level; ▷ se uguale ad 1 indica che il processore può eseguire solo istruzioni user-level.
25	IP	Exception Prefix: indica l'indirizzo base a cui è allocato l'exception vector: ▷ uguale a 0 indica che l'exception vector è allocato a partire dall'indirizzo 0x0000_0000; ▷ uguale ad 1 indica che l'exception vector è allocato a partire dall'indirizzo 0xFFF0_0000.
30	RI	Recoverable Exception: indica se è possibile o no recuperare il sistema dopo l'esecuzione di un'eccezione.

Processor Version Register (PVR) É un registro a 32-bit di sola lettura che contiene un valore che identifica la specifica *versione* ed il livello di *revisione* del processore.

BAT Registers Questi registri mantengono le informazioni necessarie per la traduzione degli indirizzi di otto blocchi di memoria.

I registri BAT vengono manipolati dal sistema operativo e sono implementati come otto coppie di registri special-purpose. Ogni blocco è definito da una coppia di registri, registro *upper* e *lower*, che forniscono l'indirizzo iniziale e la dimensione del blocco – vedere figura 3.5 a pagina 56.

Il livello OEA definisce otto registri per la traduzione degli indirizzi delle istruzioni (**IBAT**), che consistono in quattro coppie di registri, ed otto per quella dei dati (**DBAT**), anche questi in quattro coppie.

Tramite l'utilizzo di questi registri sono dimensionabili pagine di memoria da 128-Kbytes fino a 256-Mbytes.

Per una definizione completa dei valori possibili dei registri BAT fare riferimento a [PEM01].

SDR1 É un registro a 32-bit che viene utilizzato per implementare il meccanismo di *paginazione* nel microcontrollore. É suddiviso in due campi: il campo **HTABORG** contiene i 16 bit più significativi dell'indirizzo fisico di pagina ed il campo **HTABMASK** contiene una maschera che determina quanti bit del campo HTABORG intervengono nella generazione dell'indirizzo.

Segment Registers Questi registri contengono i descrittori di segmento. OEA definisce sei registri di segmento di 32-bit ciascuno.

Data Address Register (DAR) Questo registro viene utilizzato per salvare l'*effective address* generato da un accesso in memoria quando questa operazione solleva un'eccezione.

Machine Status Save/Restore Register 0 (SRR0) Il registro SRR0 viene utilizzato per salvare lo stato del processore quando viene sollevata un'eccezione e per ripristinarlo nel momento in cui viene eseguita la `rfi`[13].

Quando viene accettata l'eccezione SRR0 è impostato per puntare o all'istruzione che ha generato l'eccezione o a quella sequenzialmente successiva.

Machine Status Save/Restore Register 1 (SRR1) Anche questo registro, in coppia con SRR0, viene utilizzato per salvare lo stato del processore quando viene accettata un'eccezione e per il suo ripristino all'esecuzione di una `rfi`. SRR0 viene però utilizzato per salvare il registro MSR e per ripristinarlo al ritorno dall'handler.

Decrementer Register (DEC) É un contatore a 32-bit che fornisce un meccanismo per causare un'eccezione di decremento dopo un ritardo programmabile.

Il registro DEC causa un'eccezione quando il suo valore passa per zero.

3.2.2 Modalità di indirizzamento e Set di istruzioni

Questo paragrafo descrive il set di istruzioni e le modalità di indirizzamento definite dai tre livelli dell'architettura PowerPC. Queste istruzioni vengono suddivise per categorie funzionali:

- ▷ Istruzioni che operano su interi, che includono le istruzioni aritmetiche e logiche.
- ▷ Istruzioni che operano su floating-point, che includono tutte le istruzioni aritmetiche come quelle che influenzano il registro FPSCR.
- ▷ Istruzioni di load/store, sia per interi che per floating-point.
- ▷ Istruzioni per il controllo del flusso di programma, che includono salti, istruzioni logiche sul condition register, istruzioni di trap e tutte quelle che influenzano il flusso di esecuzione.

[13] Return From Interrupt

- Istruzioni per il controllo del processore, che vengono utilizzate per la sincronizzazione degli accessi in memoria e la gestione della cache, il TLB e i segment register.
- Istruzioni di sincronizzazione della memoria, che controllano l'ordine con cui vengono completate le operazioni in memoria in funzione di eventi asincroni.
- Istruzioni per il controllo della memoria (che includono quelle per la gestione della cache), istruzioni per la gestione dei segment register, e gestione del TLB.

3.2.2.1 Modalità di indirizzamento

Un programma accede in memoria utilizzando l'*indirizzo effettivo* (EA), mappato su 32-bit, calcolato dal processore quando viene eseguita un'istruzione di load, store, branch oppure quando viene eseguito il fetch dell'istruzione sequenzialmente successiva a quella in esecuzione.

Le istruzioni di load/store hanno tre modalità per la generazione di EA:

- registro indiretto con indice immediato.
- registro indiretto con indice.
- registro indiretto.

Le istruzioni di salto hanno le seguenti modalità di indirizzamento:

- indirizzamento immediato.
- link register indiretto.
- count register indiretto.

3.2.2.2 Istruzioni di sincronizzazione

Le istruzioni di sincronizzazione si dividono in due categorie in base alle garanzie assicurate successivamente alla loro esecuzione:

Sincronizzazione del contesto fanno parte di questa categoria le istruzioni **`sc`**[14], **`rfi`**[15] e **`isync`**[16]. Queste istruzioni eseguono una sincronizzazione del contesto consentendo a quelle precedenti nel flusso del programma di essere completate prima di eseguire un cambio di contesto. L'esecuzione di una queste istruzioni garantisce:

- ▷ nel caso della **`sc`** che non ci siano eccezioni a più alta priorità pendenti.
- ▷ tutte le precedenti istruzioni sono completate fino ad un punto in cui non possono più provocare eccezioni.
- ▷ le istruzioni precedenti completano la loro esecuzione nel contesto in cui sono state inviate alle unità di esecuzione.
- ▷ le istruzioni che le seguono sono eseguite nel contesto che viene stabilito.

Sincronizzazione dell'esecuzione un'istruzione è di sincronizzazione dell'esecuzione se soddisfa le condizioni dei primi due punti del paragrafo precedente. Fanno parte di questa categoria le istruzioni **`sync`**[17] e **`mtmsr`**[18].

3.2.2.3 Istruzioni UISA

In questo paragrafo vengono presentate le istruzioni di livello UISA. Per ogni gruppo vengono riportati solo alcuni tra tutti i codici operativi definiti. Per una lista completa delle istruzioni consultare [PEM01].

Istruzioni Intere Fanno parte di questa categoria tutte le istruzioni che lavorano con operandi di tipo intero:

- ▷ istruzioni aritmetiche
- ▷ istruzioni di confronto

[14] System Call
[15] Return From Interrupt
[16] Instruction Synchronize
[17] Synchronize
[18] Move To Machine Status Register

- ▷ istruzioni logiche
- ▷ istruzioni di rotazione e traslazione

Le istruzioni intere utilizzano il contenuto dei registri GPRs come operandi sorgenti e mettono poi il risultato della loro elaborazione in un registro GPR. Inoltre, come risultato della loro elaborazione, possono modificare i registri XER e CR se questo è previsto nel loro codice operativo.

Istruzioni Floating-point In questo insieme di istruzioni sono incluse:

- ▷ istruzioni aritmetiche
- ▷ istruzioni di moltiplicazione-addizione[19]
- ▷ istruzioni di arrotondamento e conversione
- ▷ istruzioni di confronto
- ▷ istruzioni per la manipolazione del registro di stato e controllo
- ▷ istruzioni di move

É da notare che il bit FP[20] di MSR deve essere settato per consentire l'esecuzione di questa categoria di istruzioni. Se questa condizione non è verificata nel momento in cui il processore tenta di eseguire una di queste istruzioni viene sollevata un'eccezione "Floating-Point Unavailable Exception".

Istruzioni di Load/Store Le istruzioni di load/store sono inviate alle unità di esecuzione in ordine col flusso del programma ma gli accessi in memoria possono comunque avvenire out-of-order. Vengono infatti fornite delle istruzioni di sincronizzazione per forzare l'accesso strettamente in ordine.

Di questa categoria fanno parte:

- ▷ istruzioni di load su interi

[19] Vengono definite *multiply-add*
[20] Floating-Point enabled

- ▷ istruzioni di store su interi
- ▷ istruzioni di load e di store con byte-riservati
- ▷ istruzioni di load e di store multiple
- ▷ istruzioni di load per floating-point
- ▷ istruzioni di store per floating-point
- ▷ istruzioni di sincronizzazione della memoria

Istruzioni di salto e Controllo del flusso Alcune istruzioni di salto possono modificare il flusso di esecuzione delle istruzioni condizionatamente al valore del registro CR. Quando il processore incontra un'istruzione di questo tipo controlla nella pipeline delle istruzioni se ce n'è qualcuna che possa modificare il valore del registro CR: se non ce ne sono, il salto viene risolto immediatamente, altrimenti o si utilizza un campo del codice operativo dell'istruzione o una *dynamic prediction*, con un meccanismo di esecuzione speculativa. Viene in seguito monitorata l'esecuzione di quella particolare istruzione per determinare se la predizione era corretta o sbagliata: nel primo caso il salto viene considerato completato ed il fetch continua, altrimenti viene svuotata la coda di fetch e si ricomincia il caricamento nel percorso corretto.

Da notare che l'indirizzo di salto deve essere allineato alla parola.

Questo insieme di istruzioni può essere così suddiviso:

- ▷ salti relativi
- ▷ salti condizionali ad indirizzi relativi
- ▷ salti ad indirizzi assoluti
- ▷ salti condizionali ad indirizzi assoluti
- ▷ salti condizionali al link register
- ▷ salti condizionali al count register

Ognuna di queste tipologie di istruzioni può essere eseguita o meno con aggiornamento del link register.

Fa inoltre parte di questa categoria di istruzioni l'istruzione **`sc`** che modifica il normale flusso di esecuzione.

Istruzioni per il Controllo del Processore Di questo insieme fanno parte tutte le istruzioni che vengono utilizzate per leggere e scrivere il condition register (CR), il machine status register (MSR) ed i registri speciali (SPR), ovviamente con le restrizioni del livello UISA.

3.2.2.4 Istruzioni VEA

Questo livello dell'architettura descrive il modello di memoria visto dai processi software includendo il modello della cache e le istruzioni per il suo controllo.

Istruzioni per il Controllo del Processore A questo livello viene definita l'istruzione **`mftb`**[21] di livello utente che viene utilizzata per leggere il valore dei registri time base definiti a livello architetturale VEA.

Istruzioni di Sincronizzazione della Memoria Sono definite due istruzioni, entrambe di livello user:

- **`eieio`**[22] – É una funzione di ordinamento per gli effetti di load e store;
- **`isync`**[23] – L'esecuzione di essa assicura che tutte le istruzioni precedenti nel flusso delle istruzioni siano completate prima che la **`isync`** stessa sia completata.

3.2.2.5 Istruzioni OEA

Questo livello architetturale include la struttura per la gestione della memoria, i registri speciali di livello supervisore (quelli cioè non visibili a livello utente) ed il modello di gestione delle eccezioni.

[21] Move From Time Base
[22] Enforce In-Order Execution of I/O
[23] Instruction SYNChronize

Istruzioni di System Linkage Fanno parte di questa categoria le istruzioni **`sc`** e **`rfi`**. L'istruzione **`sc`** è di livello UISA, e quindi utilizzabile dal livello user. La sua esecuzione permette ad un processo utente di chiamare il sistema operativo per svolgere un servizio causando un'eccezione. La sua gestione è quindi del tutto analoga a quella di una qualsiasi eccezione: viene salvato negli appositi registri lo stato del processore e viene fetchata la prima istruzione dall'exception vector[24] corretto per quel tipo di eccezione.

L'esecuzione della **`rfi`** non fa altro che ripristinare lo stato della macchina e riprendere l'esecuzione dal program counter corretto.

Istruzioni per il Controllo del Processore Di questo insieme fanno parte tutte quelle istruzioni che consentono di leggere e di scrivere sui registri speciali di livello OEA. Per questa funzione vengono utilizzate due istruzioni: **`mtspr`**[25] e **`mfspr`**[26]. É da notare che per i registri maggiormente utilizzati vengono usati dei codici mnemonici semplificati (vedere appendice di [PEM01]).

Istruzioni per il Controllo della Memoria Di questa categoria fanno parte tutte le istruzioni che consentono di leggere e di scrivere sui registri speciali di configurazione della memoria e della cache ed anche le istruzioni che consento di invalidare alcune entry o tutta la cache.

Queste istruzioni vengono ulteriormente suddivise in tre categorie:

- ▷ gestione delle cache
- ▷ manipolazione dei registri di segmento
- ▷ gestione del translation lookaside buffer (TLB)

3.2.3 Eccezioni

La porzione OEA dell'architettura PowerPC definisce il meccanismo con cui il processore implementa le eccezioni.

[24] L'exception vector è un indirizzo di memoria
[25] Move To SPR
[26] Move From SPR

Il meccanismo delle eccezioni consente al processore di passare in modalità supervisore, aumentando il livello di privilegio, come risultato di un segnale esterno, di un errore o del verificarsi di una condizione anomala durante l'esecuzione di un'istruzione.

Quando viene sollevata un'eccezione lo stato della macchina viene salvato in appositi registri ed il processore inizia la sua esecuzione da un indirizzo predeterminato per ogni tipo di eccezione –*exception vector*. L'elaborazione di un'eccezione viene eseguita in modalità supervisore.

Ogni tipo di eccezione ha un suo *offset* predeterminato che sommato all'indirizzo base dell'exception vector fornisce l'indirizzo della prima istruzione da eseguire.

L'architettura PowerPC richiede che le eccezioni vengano gestiste in ordine col programma, quindi, nonostante particolari implementazioni possano riconoscere le condizioni di eccezione out-of-order, queste devono essere gestite strettamente in ordine, rispettando il flusso delle istruzioni. Quando un'eccezione causata da un'istruzione viene riconosciuta, tutte le istruzioni non eseguite che sono precedenti a questa nel flusso originale devono essere completate prima che l'eccezione venga gestita.

L'elaborazione di un'eccezione passa attraverso tre fasi:

1. **Riconoscimento**: avviene quando la condizione di eccezione è identificata dal processore.

2. **Accettazione**: un'eccezione si dice accettata quando il controllo dell'istruzione in esecuzione passa al suo handler, cioè viene salvato il contesto e l'istruzione che si trova all'appropriato offset dell'exception vector viene fetchata.

3. **Gestione**: l'eccezione viene gestita dal suo handler agganciato dall'appropriato exception vector.

3.2.3.1 Classi di Eccezioni

Come specificato dall'architettura PowerPC, tutte le eccezioni possono essere classificate come *precise* o *imprecise* o come *sincrone* ed *asincrone*. Le eccezioni

asincrone sono causate da eventi esterni al processore, quelle sincrone dalle istruzioni. Le eccezioni possono quindi essere raggruppate come in tabella 3.6.

Tabella 3.6: Classificazione PowerPC delle Eccezioni

Tipo	**Eccezione**
Asincrone/ Non-mascherabili	Machine Check System Reset
Asincrone/ Mascherabili	Interruzioni Esterne Eccezione di Decremento
Sincrone/ Precise	Eccezioni causate da istruzioni, escluse quelle floating-point imprecise
Sincrone/ Imprecise	Floating-point imprecise

Quando viene riconosciuta un'eccezione l'invio delle istruzioni alle unità di esecuzione viene bloccato e viene eseguita la seguente sincronizzazione:

1. Il meccanismo delle eccezioni attende che tutte le istruzioni precedenti nel flusso originale di esecuzione siano completate, almeno fino ad un punto in cui non possono più provocare eccezioni.

2. Il processore assicura che tutte le istruzioni precedenti vengono completate nel contesto in cui hanno iniziato la loro esecuzione.

3. Il meccanismo delle eccezioni implementato in hardware ed in software è responsabile del salvataggio e ripristino dello stato del processore.

Questo tipo di sincronizzazione è conforme allo standard di *sincronizzazione del contesto* (3.2.3.2).

Eccezioni Sincrone/Precise Quando l'esecuzione di un'istruzione causa un'eccezione *precisa* queste condizioni sono verificate al punto di eccezione:

- ▷ In dipendenza dal tipo di eccezione, SRR0 indirizza o l'istruzione che ha causato l'eccezione o quella immediatamente successiva.

- ▷ Tutte le istruzioni che precedono quella che ha causato l'eccezione sono completate prima che l'eccezione sia processata.
- ▷ L'istruzione che ha sollevato l'eccezione può non aver iniziato la sua esecuzione, può essere parzialmente completata o aver terminato la sua esecuzione, in dipendenza dal tipo di eccezione.
- ▷ Nessuna istruzione successiva a quella che ha causato l'eccezione ha iniziato la sua esecuzione.

Eccezioni Asincrone Ci sono quattro tipi di eccezioni asincrone: *system reset* e *machine check* –non mascherabili e ad alta priorità, e gli *interrupt esterni* e l'*eccezione di decremento* –mascherabili ed a bassa priorità.

Eccezioni Imprecise L'architettura PowerPC definisce una sola eccezione imprecisa: *imprecise floating-point enabled exception*.

3.2.3.2 Sincronizzazioni

Le sincronizzazioni descritte in questa sezione si riferiscono alle attività all'interno del processore che esegue la sincronizzazione.

Sincronizzazione del Contesto Un'istruzione o un evento è di *sincronizzazione del contesto* se soddisfa *tutti* i requisiti elencati:

- ▷ L'operazione causa il blocco dell'invio delle istruzioni dalla fetch unit alle unità di esecuzione –dispatch.
- ▷ L'operazione non viene iniziata o, nel caso della **`isync`**[27] non è completata, finchè tutte le istruzioni in esecuzione non raggiungono il punto in cui hanno riportato tutte le possibile eccezioni che possono causare.
- ▷ Le istruzioni che precedono l'operazione sono completate nel contesto in cui hanno iniziato.

[27] Instruction Synchronize

- ▷ L'operazione, sia essa un'istruzione che ha causato un'eccezione o sia un'eccezione, non è iniziata finchè esistono eccezioni che hanno priorità maggiore dell'eccezione associata con l'operazione di context synchronizing.

Execution Sinchronization Un'istruzione è di *sincronizzazione dell'esecuzione* se soddisfa solo le prime due condizioni descritte precedentemente.

Quindi tutte le istruzioni context synchronizing sono anche execution sinchronizing e, non è vero il contrario.

Le istruzioni **`sync`**[28] e **`mtmsr`**[29] sono esempi di questa categoria.

3.2.3.3 Elaborazione delle Eccezioni

Quando un'eccezione viene *accettata*, il processore utilizza i registri SRR0–SRR1 per salvare il *program counter* ed il contenuto del registro MSR in modo da poter ripristinare il normale flusso di programma una volta ritornati dall'handler.

L'indirizzo salvato in SRR0 viene utilizzato per calcolare il punto di recupero dell'elaborazione quando l'handler finisce ed il controllo ritorna al processo interrotto.

Il contenuto di SRR1 viene invece utilizzato per ripristinare lo stato della macchina al ritorno dall'handler.

Questo due operazioni vengono svolte dal μ-codice di esecuzione della **`rfi`**.

Per una visione completa sulla gestione delle interruzioni fare riferimento a [JDM01].

Steps dell'Elaborazione di un'Eccezione Dopo aver accettato l'eccezione il processore compie le seguenti azioni:

- ▷ Il registro SRR0 viene caricato con l'indirizzo di un'istruzione che dipende dal tipo di eccezione.

- ▷ I bit 1–4 e 10–15 di SRR1 vengono caricati con i valori specifici per l'eccezione.

[28] Synchronize
[29] Move To Machine Status Register

- I bit 16–23, 25–27 e 30–31 del registro SRR1 vengono caricati con i corrispondenti bit del registro MSR.
- Il registro MSR viene impostato con un valore che dipende dal tipo di eccezione. Per una descrizione completa di questi valori consultare [PEM01].

A questo punto viene fetchata la prima istruzione dell'exception vector relativo a quel tipo di eccezione ed il controllo passa all'handler.

Quando l'hadler termina viene eseguita una **`rfi`** che assicura le seguenti condizioni:

- Tutte le istruzioni sono completate ad un punto dove non possono causare eccezioni.
- Le istruzioni precedenti vengono completate nel contesto in cui erano state inviate alle unità di esecuzione.
- L'istruzione **`rfi`** copia il contenuto di SRR1 nel registro MSR.
- Tutte le istruzioni che la seguono vengono eseguite nel contesto da essa stabilito.

Per una definizione completa delle eccezioni vedere [PEM01].

É compito dell'handler fare una copia dei registri SRR0–SRR1 in modo da poter ripristinare il loro contenuto anche in seguito al verificarsi di un'eccezione annidata con la prima. Fatto questo salvataggio è sempre compito dell'handler segnalare, attraverso un'apposita manipolazione del bit di RI di MSR, che l'eccezione è adesso recuperabile e di riabilitare o no, in funzione dell'implementazione, le eccezioni.

3.2.3.4 Switch tra Processi

Il sistema operativo deve eseguire le seguenti operazioni:

- l'istruzione **`sync`** che ordina gli effetti dell'esecuzione delle istruzioni.
- l'istruzione **`isync`** che attende il completamento di tutte le istruzioni precedenti e svuota la coda di fetch.

- l'istruzione **`stwcx.`** che elimina tutte le mutue esclusioni in memoria, assicurando che una **`lwarx`** nel vecchio processo venga accoppiata con una **`stwcx.`** nel nuovo.

Il sistema operativo deve inoltre manipolare opportunamente il bit RI di MSR.

3.2.4 Memory Management Unit

Questo paragrafo presenta il modello di gestione della memoria, che definisce come vengono tradotti gli indirizzi e come la memoria viene protetta.

Il processore genera indirizzi di memoria in due casi: per il fetch delle istruzioni e per l'accesso ai dati richiesto dall'esecuzione di una load o di una store.

Generalmente il meccanismo di traduzione è definito in termini di *descrittori di segmento* e di *tabella delle pagine* utilizzate per mappare l'indirizzo *effettivo* in quello *fisico*. Le informazioni sui segmenti traducono l'indirizzo effettivo in indirizzo *virtuale* e le informazioni contenute nella tabella delle pagine traducono l'indirizzo virtuale in quello fisico. Inoltre viene utilizzato un TLB[30] che mantiene informazioni sulle traduzioni più recenti.

Parallelamente a questo meccanismo ne viene utilizzato uno per la traduzione di interi blocchi di indirizzamento, *block address translation* (BAT) appunto.

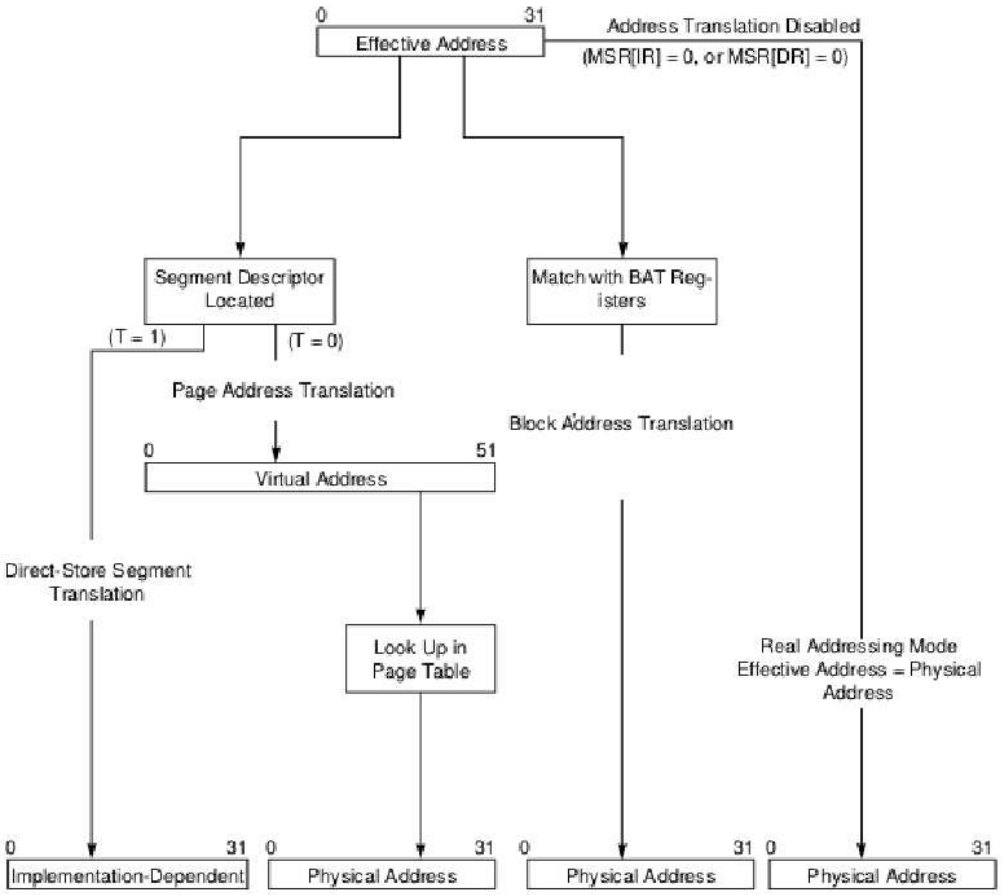
In questo modo la MMU, unitamente al meccanismo di gestione delle eccezioni, fornisce il supporto necessario al sistema operativo per implementare la paginazione della memoria virtuale e per rafforzare la protezione della memoria.

3.2.4.1 Meccanismoo di Traduzione

Il microcontrollore supporta i seguenti tipi di traduzione degli indirizzi:

- traduzione dell'indirizzo di pagina.
- traduzione dell'indirizzo di un blocco.
- modalità *indirizzo reale* –quando il meccanismo di traduzione è disabilitato l'indirizzo fisico corrisponde all'indirizzo reale.

Figura 3.4: Modalità di Traduzione



La figura 3.4 mostra il meccanismo di traduzione della MMU.

Vediamo ora in modo più dettagliato le varie modalità di traduzione.

3.2.4.2 Real Addrressing Mode

É la modalità di traduzione più semplice perchè la traduzione non avviene affatto. Questa situazione si verifica quando la traduzione degli indirizzi generati dal processore è disabilitata, cioè quando il bit IR[31] o il bit DR[32] del registro MSR è

[30] Translation Lookaside Buffer
[31] Instruction Address Translation
[32] Data Address Translation

posto uguale a zero. In queste condizioni l'indirizzo effettivo viene trattato come indirizzo fisico e passato direttamente al sottosistema di gestione della memoria.

Un indirizzo di memoria che arriva con modalità real address abilitata, cioè con traduzione degli indirizzi disabilitata, bypassa tutti i meccanismi di protezione presenti nella MMU.

Questa modalità di funzionamento è molto utilizzata in quanto ogni volta che viene accettata un'eccezione il processore resetta i bit IR e DR del registro MSR. In questo modo, almeno per le prime istruzioni dell'handler, il processore opera in real addressing mode. Se la traduzione degli indirizzi è richiesta dall'handler l'abilitazione deve essere esplicitamente eseguita via software.

3.2.4.3 Block Address Translation

Questo meccanismo consente di mappare un certo range di indirizzi effettivi in aree contigue di memoria fisica.

Il meccanismo di traduzione è implementato come un array BAT[33] controllato via software in cui vengono mantenute le informazioni per la traduzione di otto blocchi di memoria. Questo array è gestito ed aggiornato dal sistema operativo, o comunque da un software di livello superuser, ed implementato come un insieme di 16 special-purpose register (SPRs). Ogni blocco è definito da una coppia di questi registri chiamati upper- e lower- BAT che contengo l'indirizzo effettivo e quello fisico di ogni blocco. Essendo dei registri speciali i registri BAT possono essere letti e scritti utilizzando le istruzioni **`mtspr`** e **`mfspr`**.

Ogni coppia di registri BAT definisce l'indirizzo iniziale di un blocco dello spazio di indirizzamento effettivo, la dimensione del blocco e l'indirizzo iniziale del blocco nello spazio di indirizzamento fisico. Se un indirizzo effettivo è interno allo spazio definito dalla coppia di registri BAT allora il suo indirizzo fisico è definito dall'indirizzo fisico di partenza del blocco più i bit di ordine più basso dell'effective address.

Le dimensioni possibili di un blocco sono ristrette ad un numero finito, da 128-Kbytes (2^{17}bytes) a 256-Mbytes (2^{28}bytes). L'indirizzo di partenza di un blocco è sempre multiplo della sua dimensione.

[33] Block Address Translation

In figura 3.5 è mostrata una coppia di registri BAT ed in tabella 3.6 la descrizione dei loro bit.

Figura 3.5: Formato di una coppia di registri BAT

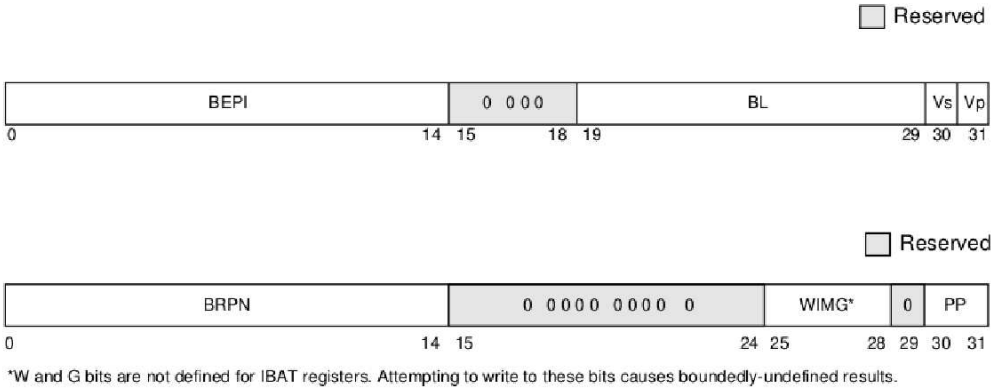


Figura 3.6: Registri BAT – Descrizione

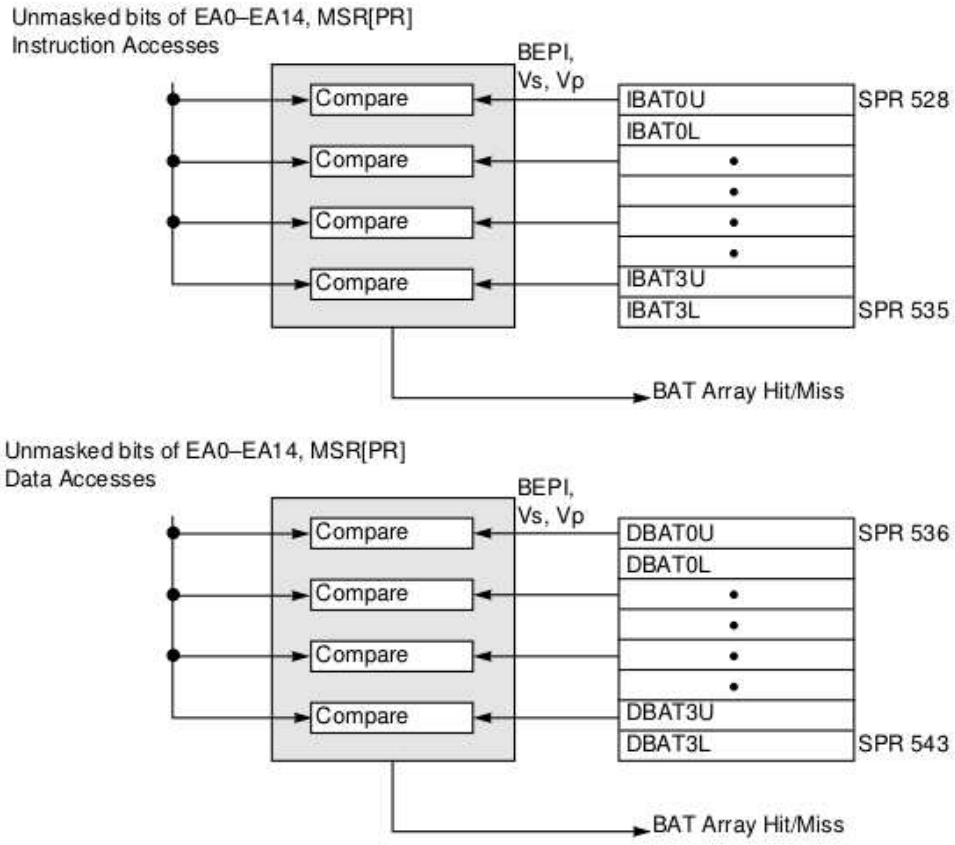
Upper/Lower BAT	Bits	Name	Description
Lower BAT Register	0–14	BRPN	This field is used in conjunction with the BL field to generate high-order bits of the physical address of the block.
	15–24	—	Reserved
	25–28	WIMG	Memory/cache access mode bits W Write-through I Caching-inhibited MMemory coherence GGuarded Attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results. For detailed information about the WIMG bits, see Section 5.3.1, "Memory/Cache Access Attributes."
	29	—	Reserved
	30–31	PP	Protection bits for block. This field determines the protection for the block as described in Section 7.5.4, "Block Memory Protection."

Nota:

Gli entry dell'array BAT sono completamente ignorati dalle operazioni di *invalidate* del TBL.

La figura 3.7 nella pagina successiva mostra l'organizzazione dell'array BAT.

Figura 3.7: Organizzazione Array BAT



L'array BAT è full-associative così ogni indirizzo può risiedere in uno qualsiasi dei suoi entry.

3.2.4.4 Modello di Memoria Segmentata

La memoria in OEA è divisa in segmenti di dimensione 256-Mbytes. Questo modello di memoria segmentata fornisce un modo per mappare pagine di 4-Kbytes di indirizzamento effettivo in pagine di 4-Kbytes di indirizzamento fisico (*page address translation*).

La traduzione dell'indirizzo viene eseguita in due passi, come mostrato in figura 3.8 a fronte:

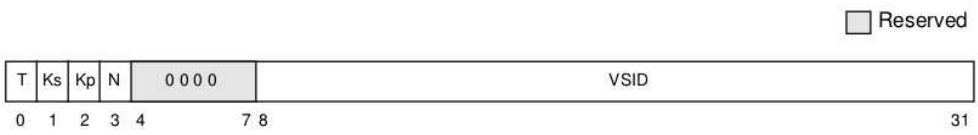
1. viene prima tradotto l'indirizzo effettivo in indirizzo virtuale –questo viene fatto utilizzando i descrittori di segmento

2. in seguito l'indirizzo virtuale viene tradotto in indirizzo fisico –in questa fase vengono utilizzate le informazioni della tabelle delle pagine

I descrittori di segmento vengono programmati per fornire l'ID virtuale per un segmento dal sistema operativo, che deve inoltre creare la tabella delle pagine in memoria che fornisce la corrispondenza indirizzo virtuale-fisico.

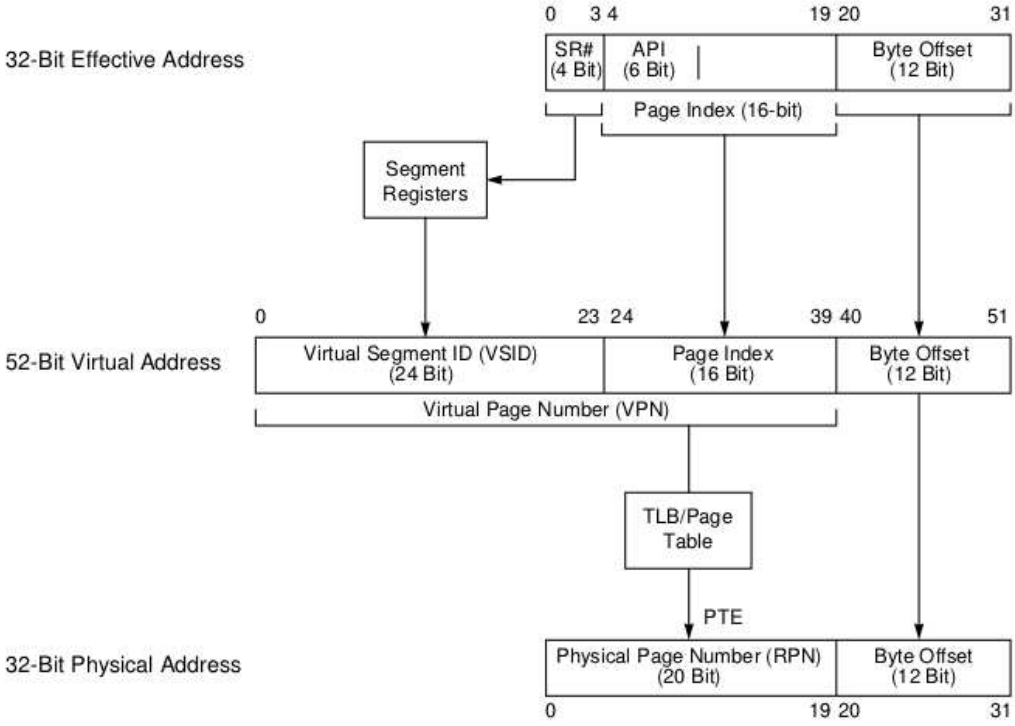
I descrittori di segmento sono lunghi 32-bit e risiedono in uno dei 16 registri di segmento. In figura 3.9 viene mostrata la composizione di uno di questi registri.

Figura 3.9: Registro di Segmento



Per una trattazione completa del meccanismo di paginazione dell'architettura MPC500 riferirsi a [PEM01].

Figura 3.8: Page Address Translation



3.3 Convenzioni di programmazione

Questa sezione fa riferimento al documento [EAB98]. Visto che l'architettura PowerPC è trasversale tra vari produttori, sono state stabilite delle convenzioni per la programmazione con lo scopo di migliorare la compatibilità tra le varie implementazioni. L'insieme di queste convenzioni è conosciuto come Embedded Application Binary Inetrface –EABI appunto.

EABI descrive quindi le convenzioni per l'utilizzo dei registri, per il passaggio dei parametri, per l'organizzazione dello stack, per le piccole aree dati, per i file oggetto e per il formato dei files eseguibili.

3.3.1 Tipi di Dati ed Allineamento

Lo standard PowerPC definisce come dimensioni per i tipi di dati scalari quelle riportate in tabella 3.7:

Tabella 3.7: Dati scalari PowerPC

Tipo	**Dimensione (in byte)**
Byte	1
Mezza Parola	2
Parola (Word)	4
Parola Doppia	8

Tutti i dati di tipi predefiniti sono allineati in memoria, e quindi anche nello stack, ad un indirizzo che è multiplo della loro dimensione. Le strutture, o le unioni, sono allineate ad un indirizzo multiplo del loro membro più grande. La dimensione di una struttura è quindi multipla del suo allineamento.

Compilatori ed assemblatori che rispondono allo standard EABI creano dati allocati in modo rispondente all'allineamento richiesto dalle specifiche.

3.3.2 Convenzione sull'utilizzo dei Regsitri

Come già detto l'architettura PowerPC definisce 32 registri generali (GPRs) e 32 registri floating-point (FPRs). Lo standard EABI classifica questi registri come *volatili*, *non volatili* e *dedicati*. I registri non volatili devono preservare il loro contenuto al ritorno da una funzione chiamata, quindi le funzioni che li modificano devono salvarli e poi ripristinarli prima di ritornare al chiamante. I registri volatili non hanno questo requisito.

Tre dei registri classificati non volatili sono dedicati per utilizzi specifici: r1, r2 ed r13.

Tutti i registri PowerPC ed il loro utilizzo sono elencati in tabella 3.8.

Tabella 3.8: Utilizzo dei Registri

Registro	**Tipo**	**Utilizzato per:**
R0	Volatile	Specifico del Linguaggio
R1	Dedicato	Stack Pointer (SP)
R2	Dedicato	Small Data Area di sola lettura
R3 ÷ R4	Volatile	Passaggio dei Parametri / Valori di Ritorno
R5 ÷ R10	Volatile	Passaggio dei Parametri
R11 ÷ R12	Volatile	
R13	Dedicato	Small Data Area di lettura/scrittura
R14 ÷ R31	Non Volatile	
F0	Volatile	Specifico del Linguaggio
F1	Volatile	Passaggio dei Parametri / Valori di Ritorno
F2 ÷ F8	Volatile	Passaggio dei Parametri
F9 ÷ F13	Volatile	
F14 ÷ F31	Non Volatile	

3.3.3 Convenzioni sullo Stack

L'architettura PowerPC non ha le istruzioni di **push**/**pop** per implementare lo stack –lo stack non è implementato a microprogramma.

Le convenzioni EABI per la creazione e l'utilizzo dello stack sono definite per supportare il passaggio dei parametri, la conservazione del valore dei registri non volatili, le variabili locali ed il debugging.

Ogni funzione che può chiamarne un'altra o che modifica i registri non volatili deve creare uno stack frame in memoria. Lo stack pointer, cioè il registro r1 per convenzione, punta alla word di indirizzo più basso dello stack frame della funzione in esecuzione. Ogni funzione chiamata crea il suo stack sopra[34] il frame del suo chiamante. Il frame viene creato nel prologo della funzione e distrutto nel suo epilogo.

Lo stack frame è sempre allineato alla doppia parola (8 byte) utilizzando, se necessario, anche bytes di padding. Uno stack frame completo è riassunto in tabella 3.9 nella pagina successiva.

Tutti i frame hanno un header che consiste in due word – la word Back Chain e la word di salvataggio di LR.

Il campo Back Chain contiene l'indirizzo della word Bach Chain del frame precedente –quello cioè del chiamante. In questo modo si instaura una lista di collegamenti tra i vari frame. La word Bach Chain è sempre allocata nella word di indirizzo più piccolo del frame.

La word predisposta al salvataggio del LR serve per mantenere un collegamento al chiamante. Il registro LR deve essere salvato nel prologo della funzione chiamata prima che venga modificato –in questo modo LR contiene l'indirizzo di ritorno al chiamante.

Nella zona di passaggio di parametri vengono salvati i parametri della funzione nel caso in cui i registri R3 $\div$ R10 non siano sufficenti.

[34] Cioè per indirizzi descrescenti.

Tabella 3.9: Stack Frame EABI

Back Chain Word
salvataggio LR
Padding (opzionale, dimensione variabile tra 1÷7 bytes)
Parametri della Funzione (opzionale, dimensione variabile)
Variabili Locali (opzionale, dimensione variabile)
salvataggio CR (opzionale)
Area di salvataggio GPRs (opzionale, dimensione variabile)
Area di salvataggio FPRs (opzionale, dimensione variabile)

3.4 Processore di riferimento

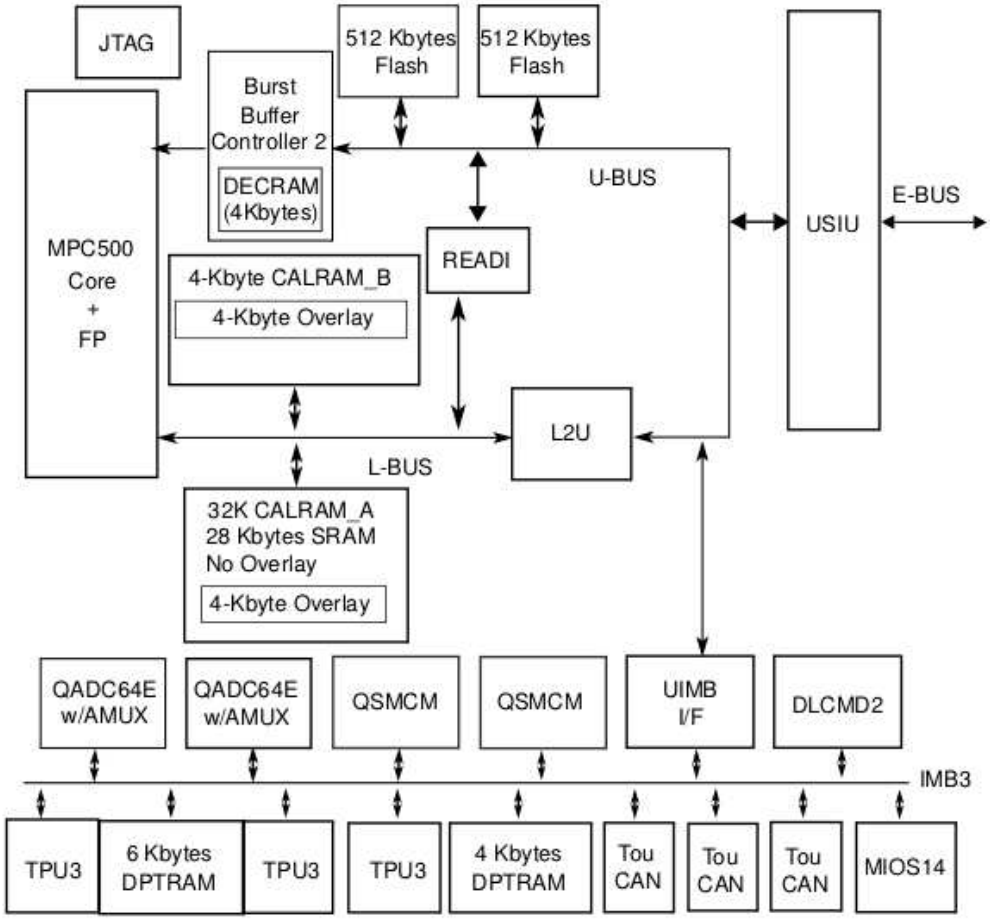
In questo paragrafo verranno descritte le caratteristiche della particolare implementazione utilizzata come piattaforma hardware nello sviluppo di questa tesi: il microcontrollore MPC566. Verranno inoltre introdotti i moduli di cui dispone e l'organizzazione logica della sua memoria.

Daremo poi uno sguardo al controller interrupt migliorato[35] di cui la serie MPC56x dispone.

Uno schema a blocchi del microcontrollore MPC566 è visibile in figura 3.10 nella pagina seguente.

[35] Enhanced Interrupt Controller

Figura 3.10: Diagramma a Blocchi



3.4.1 Caratteristiche

Le caratteristiche principali dei microntrollori MPC565/MPC566[36] sono:

- ▷ un core MPC500, aderente allo standard PowerPC
- ▷ 36-Kbytes di RAM statica, suddivisa in due moduli CALRAM –che da ora chiameremo CALRAM_A e CALRAM_B
- ▷ 1-Mbytes di memoria FLASH (UC3F)
- ▷ una unità USIU[37], un controller della memoria flessibile ed un enhanced interrupt controller
- ▷ tre Time Processor Unit (TPU3)
- ▷ un canale modulato temporizzato per il sistema di I/O (MIOS14)
- ▷ tre moduli TouCAN (TOUCAN_A, TOUCAN_B, TOUCAN_C)
- ▷ due convertitori analogico digitali ad accodamento (QADC64E_A, QADC64E_B) con MUiltipleXor analogico (AMUX) per un totale di 40 canali analogici
- ▷ una porta di debug NEXUS (di classe 3)[38]
- ▷ JTAG e Background Debug Mode (BDM)

Per una lista completa delle caratteristiche del microntrollore fare riferimento a [MRM03].

In particolare per il microntrollore MPC566:

- ▷ può operare a 40-MHz o a 56-MHz, selezionabile via software
- ▷ interfacciabile con bus esterni alimentati a 2.6 V $\pm$ 0.1 V
- ▷ logica interna alimentata a 2.6 V $\pm$ 0.1 V

[36] I due microcontrollori appartengono alla famiglia Motorola MPC500 RISC Microntroller
[37] Unified System Integration Unit
[38] IEEE-ISTO 5001-1999

- ▷ interfacce a 5.0 V $\pm$ 0.25 V
- ▷ CPU high-performance
- ▷ core high-performance
  - ⇝ PowerPC single issue integer core
  - ⇝ modelle preciso per le eccezioni
  - ⇝ floating-point
  - ⇝ supporto al debug estensivo
  - ⇝ supporto alla compressione del codice
- ▷ interfacce (USIU, BBC, L2U)
  - ⇝ Periodic Interrupt Timer (PIT), bus monitor, clocks, decrementer e time base
  - ⇝ Enhanced Interrupt Controller che fornisce il supporto a più interrupt vector separati per più di 8 interrupt esterni e 40 interni
  - ⇝ accesso alla porta di test IEEE 1149.1 JTAG
  - ⇝ il bus supporta master multipli
  - ⇝ unità di protezione della memoria flessibili nel BBC (IMPU) e L2U (DMPU)
  - ⇝ possibilità di rilocare la tabella degli exception vector in varie zone di memoria:
    - ⇝ 0x0000 0000 ÷ 0x0000 1FFF –locazione classica per MPC500
    - ⇝ 0x0001 0000 ÷ 0x0001 1FFF –seconda pagina della flash interna
    - ⇝ nel secondo modulo di flash interna
    - ⇝ nella SRAM interna
    - ⇝ 0xFFFF 01000 –nello spazio di memoria esterno, anche questo è una zona tipicamente utilizzata nello sviluppo

3.4.2 Memoria

La memoria interna è organizzata in un singolo blocco da 4-Mbytes. Questo blocco può essere allocato in una zona tra le otto differenti locazioni possibili.

Lo spazio di memoria interna è suddiviso nelle seguenti sezioni:

- memoria Flash (1-Mbyte) – memoria U-bus
- memoria RAM statica (36-Kbytes CALRAM) – memoria L-bus
- registri di controllo e modulo IMB3 (64-Kbytes) partizionata in:
  - registri di controllo per USIU e flash
  - interfaccia UIMB e moduli IMB3
  - registri di controllo CALRAM e READI (spazio del registro di controllo L-bus)

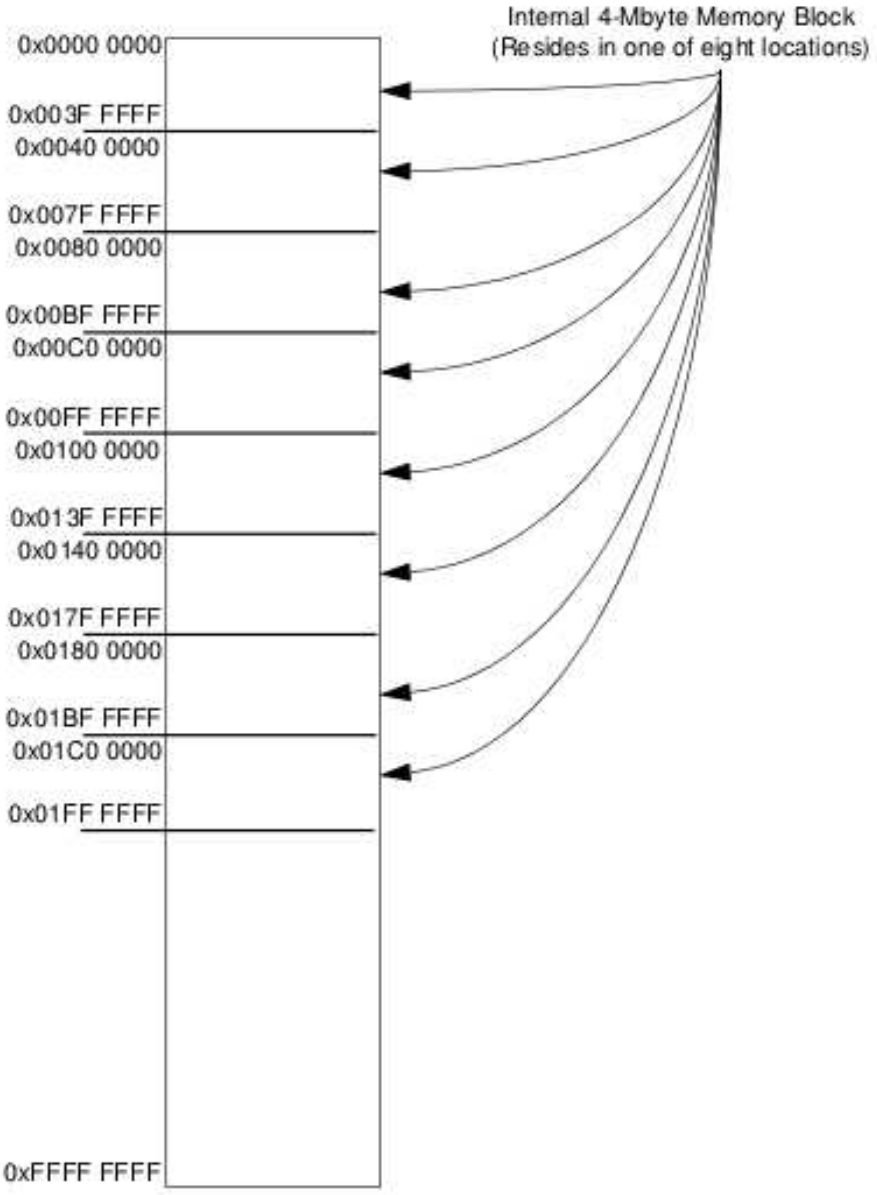
Il blocco di memoria interna può risiedere in uno degli otto possibili blocchi da 4-Mbyte che partono dall'indirizzo 0x0000_0000 come mostrato in figura 3.11 nella pagina successiva.

In questo spazio sono presenti anche i registri dell'unità USIU programmabili per configurare la memoria interna in una delle otto locazioni. Questa rilocabilità della memoria interna consente lo sviluppo di sistemi a chip multipli.

3.4.3 Dispositivi ed Interfacce

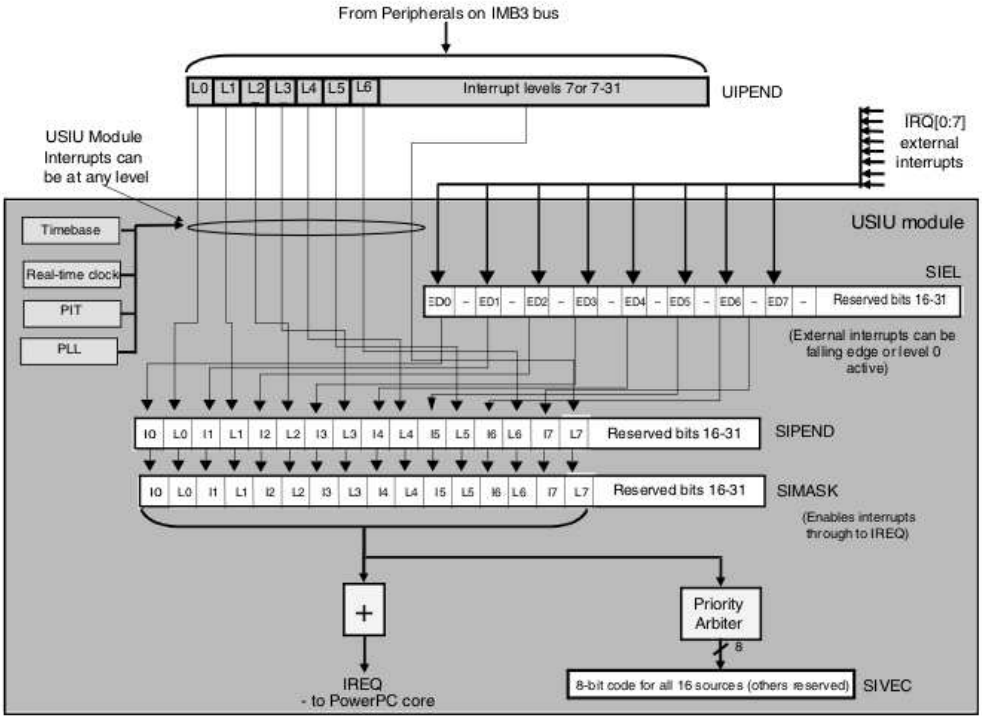
Come già accennato il microcontrollore MPC566 (come tutti gli altri microcontrollori della famiglia a cui appartiene) è dotato di un modulo di interfacciamento con i dispositivi interni e le interfacce verso l'esterno. Questo modulo, rappresentato in figura 3.12, è il modulo USIU.

Figura 3.11: Memory Map



▷

Figura 3.12: Modulo USIU



Come si vede questo modulo svolge la funzione di unificare le richieste di interruzione che provengono dai piedini esterni, dalle interfacce esterne e dai dispositivi interni.

Nella letteratura che tratta della gestione delle interruzioni nei microcontrollori Motorola, come viene spiegato in [JDM01], si fa una netta distinzione tra *eccezioni* ed *interruzioni*: le eccezioni sono eventi che possono modificare il normale flusso di esecuzione e lo stato della macchina mentre le interruzioni sono un particolare tipo di eccezioni. Il reset, il passaggio per zero del contenuto del decrementer register ed una system call vengono classificati come eccezioni, mentre le interruzioni vengono causate dai piedini esterni o dalle periferiche interne.

3.4.3.1 Interruzioni

Una *sorgente di interruzione* è una periferica che può iniziare un interrupt. Per la nostra architettura queste sono:

- ▷ i piedini di input[39] – IRQ[0÷ 7] in figura 3.12;
- ▷ i timer interni – time base (TBL), il timer ad interrupt periodici (PIT) o il real-time clock (RTC);
- ▷ i moduli di interfacciamento alle periferiche esterne o dell'intermodule bus (IMB3) – TPU3, QADC, ...

Tutte queste sorgenti di interruzione vengono mappate sullo stesso exception vector (0x500) che è quello degli **interrupt esterni**. Nasce quindi il problema di riconoscere, all'interno dell'handler, quale sia la sorgente di interruzione da servire. Per questo motivo vengono assegnati dei *livelli di interrupt* a tutti i moduli interni (timers e moduli di interfacciamento) in fase di configurazione –ogni modulo dispone di un registro di configurazione in cui scrivere questo valore. Vengono quindi mappati nel registro delle eccezioni pendenti del modulo USIU (registro SIPEND) otto livelli di interrupt ed otto IRQ –provenienti dai piedini esterni.

3.4.3.2 Dispositivi Interni – Timers

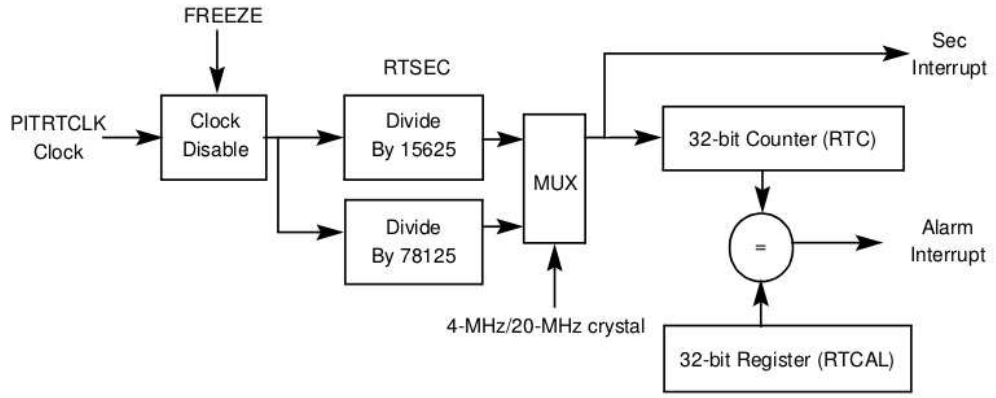
Come già detto, nel microcontrollore sono integrati vari timer. In questo paragrafo verranno analizzati i due che sono stati utilizzati nello sviluppo di ERIKA. Per una visione completa dei dispositivi dell'MPC566 vedere [MRM03].

Real-Time Clock – RTC Questo timer può essere programmato per generare interrupt mascherabili quando il valore del suo contatore interno raggiunge il valore che è stato scritto in fase di configurazione nel suo alarm register. Inoltre può anche essere abilitato ad inviare richieste di interrupt ogni secondo.

Il RTC è fornito di un contatore a 32-bit.

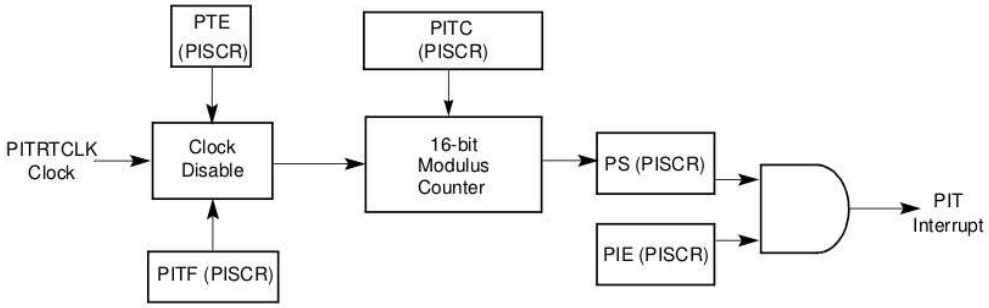
[39] IRQ input pins

Figura 3.13: Real-Time Clock



Period Interrupt Timer – PIT In questo caso si ha un contatore a 16-bit. In fase di configurazione viene scritto un valore in un registro: questo valore sarà caricato all'avvio in un registro interno e verra decrementato ogni tik del clock. La richiesta di interruzione, anche in questo caso mascherabile, verrà inviata quando il valore del registro interno arriva a zero, prima di essere nuovamente caricato con il valore contenuto nel registro di configurazione.

Figura 3.14: Periodic Interrupt Timer



Capitolo 4

ERIKA

Si tratta di un sistema operativo per sistemi embedded, originariamente nato per supportare le architetture dei microcontrollori utilizzati in campo automobilistico ed è dunque destinato a tutti quei sistemi in cui la predicibilità degli eventi e la minima occupazione di memoria risultano essere i requisiti fondamentali.

E.R.I.K.A. (Embedded Real tIme Kernel Architecture) implementa il modello a memoria comune e presenta una architettura costituita da due strati distinti:

- Kernel Layer.
- Hardware Abstraction Layer (HAL).

Una tale architettura permette di effettuare il porting del sistema operativo su piattaforme hardware differenti andando a modificare soltanto la parte HAL: attualmente sono implementate le HAL relative ai processori Siemens ST10/C167 [ST00] ed ARM7TDMI [arm95].

Tra i vari algoritmi di scheduling [But95] quelli che sono implementati attualmente sono:

- Fixed Priority (FP) [LL73].
- Earliest Deadline first (EDF) [Hor74].

In entrambi i casi viene inoltre utilizzato il protocollo SRP (Stack Resource Policy) con soglie di preemption [Bak91, SW99]: ciò garantisce che un processo che

va in esecuzione possa subire preemption ma non possa mai bloccarsi, in quanto nel caso in cui dovesse utilizzare una risorsa condivisa già in uso, tale processo non viene assolutamente schedulato e dunque non va in esecuzione. L'utilizzo delle soglie di preemption consente poi un risparmio nella quantità di stack utilizzato, tanto maggiore quanto più è elevato il numero di processi appartenenti allo stesso livello di preemption: il concetto è che processi appartenenti allo stesso livello non possono trovarsi contemporaneamente in esecuzione e dunque si può preventivamente conoscere quale è il massimo numero di frames che può essere presente sullo stack, ovvero tanti quanti sono i gruppi di non preemption.

Osserviamo che esistono degli algoritmi per la scelta ottimale dei gruppi di non preemption [GLDN01].

Come note generali osserviamo che il codice del sistema operativo è "immerso" nel codice dell'applicazione e questo consente delle ottimizzazioni sia in termini di chiamata delle primitive che di cambio del contesto. È inoltre importante osservare che ERIKA è altamente configurabile e modularizzabile in modo tale che l'applicazione possa sfruttare tutte e sole le parti del sistema operativo che sono necessarie.

Osserviamo infine che viene adottato il modello a memoria comune [AB87] per quanto riguarda la gestione della memoria.

Per ulteriori dettagli si rimanda al sito ufficiale di ERIKA: **http://erika.sssup.it** ed a quello di Evidence: **http://www.evidence.eu.com**.

4.1 Hardware Abstraction Layer (HAL)

È la parte del sistema operativo che dipende dallo hardware che viene utilizzato: lo scopo principale di questo strato è quello di esportare il concetto di *thread*[1], oltre a gestire i cambi di contesto e le richieste di interruzione. Sono possibili due configurazioni:

- **Monostack**: tutti i task ed i driver utilizzano un unico stack. Questa soluzione può essere adottata solo nei casi in cui l'algoritmo di scheduling

[1] Visibile ed utilizzabile dallo strato kernel

garantisca che un task che ha subito preemption non possa tornare in esecuzioni prima che siano terminati tutti i task che hanno fatto preemption su di esso. In particolare osserviamo che non è possibile utilizzare semafori con questo tipo di stack.

- ▷ **Multistack:** ogni task può avere il proprio stack privato oppure può condividerlo con un gruppo di task. Questo tipo di stack presenta una maggiore occupazione di memoria ed anche un overhead dovuto alle operazioni necessarie per il cambio di stack: per contro, presenta il vantaggio di poter essere utilizzato quando sono presenti primitive bloccanti, come ad esempio le semaforiche e consente di arricchire il kernel con algoritimi di scheduling non fattibili su monostack, come ad esempio i PIP (Priority Inheritance Protocols) [SRL90]. Osserviamo che alcuni processori consentono di avere la memoria segmentata ed in questi casi il multistack, a differenza del monostack, consente di sfruttare questa funzionalità.

In ogni caso, come modello di esecuzione dei thread viene utilizzato lo *one-shot*: in pratica il corpo della funzione rappresenta un'istanza del thread; quando un thread viene attivato si effettua una chiamata della funzione e quando questa termina l'istanza di quel thread è terminata: ciò questo comporta la rimozione dallo stack del frame ad essa relativo.

Nel seguito vengono presentati i tipi di dati, le costanti e le primitive esportate dallo HAL, mentre per quanto riguarda le strutture dati queste risultano dipendenti dall'architettura e dunque non vengono qui menzionate.

4.1.1 Tipi di dati e costanti

Alcuni tipi possono essere dipendenti dall'applicazione e dunque devono essere definiti nei file opportuni: per questo si rimanda la loro trattazione al capitolo 5.

4.1.1.1 Tipi di dati

- ▷ ERIKA_TID: identificatore di thread.
- ▷ ERIKA_TIME: valore temporale.

- ERIKA_ADDR: indirizzo di memoria.
- ERIKA_WORD: parola di memoria.
- ERIKA_BIT: singolo bit.
- ERIKA_INT8: intero ad 8 bit con segno.
- ERIKA_UINT8: intero ad 8 bit senza segno.
- ERIKA_INT16: intero a 16 bit con segno.
- ERIKA_UINT16: intero a 16 bit senza segno.
- ERIKA_INT32: intero a 32 bit con segno.
- ERIKA_UINT32: intero a 32 bit senza segno.
- ERIKA_UREG: intero a 32 bit senza segno.
- ERIKA_SREG: intero a 32 bit con segno.

4.1.1.2 Costanti

- ERIKA_NULL: indirizzo di memoria non valido.

4.1.2 Strutture dati

Ogni architettura utilizza le proprie strutture dati; in ogni caso l'unica che al momento è comune a tutte le implementazioni è la seguente:

- ERIKA_ADDR ERIKA_x_thread_body[]: vettore che contiene l'indirizzo del corpo di ogni thread. x è il nome dell'architettura (es:. st10, arm7, h8, mpc5, ecc...)

4.1.3 Primitive esportate verso lo strato di kernel

4.1.3.1 Manipolazione dei contesti

▷ void **ERIKA_hal_endcycle_ready**(ERIKA_TID thread) ;

Descrizione Il contesto del task attualmente in esecuzione (che dunque ha invocato questa primitiva) viene distrutto e viene mandato in esecuzione il task individuato da *thread*. Nel caso multistack può provocare un cambio di stack.

▷ void **ERIKA_hal_endcycle_stacked**(ERIKA_TID thread);

Descrizione Il contesto del task che ha invocato la primitiva viene distrutto e viene rimandato in esecuzione il task individuato da *thread* che aveva precedentemente subito preemption. Il parametro *thread* viene ignorato nel caso monostack (il task che torna in esecuzione è quello che era stato interrotto dal task che ora sta terminando), mentre nel multistack serve per risalire allo stack associato al task che viene riportato in esecuzione.

▷ void **ERIKA_hal_ready2stacked**(ERIKA_TID thread);

Descrizione Manda in esecuzione il task individuato da *thread*. Il thread correntemente in esecuzione subisce preemption ed il suo contesto viene inserito nello stack. Questa primitiva può provocare un cambio di stack nel caso multistack.

▷ void **ERIKA_hal_IRQ_ready**(ERIKA_TID thread);

Descrizione Il task che è stato interrotto dal driver subisce preemption da parte del task *thread* che viene dunque mandato in esecuzione dopo che è stato salvato il contesto del task preempted. Può provocare un cambio di contesto nel multistack.

▷ void **ERIKA_hal_IRQ_stacked**(ERIKA_TID thread);

Descrizione Viene riattivato il task individuato da *thread* che era stato interrotto dal driver. Nel caso in cui sia possibile avere interruzioni annidate, si ritorna ad eseguire il driver che era stato interrotto.

▷ void **ERIKA_hal_stkchange**(ERIKA_TID thread);

Descrizione Prevista solo nel caso multistack. Permette la commutazione di contesto al task *thread* che si trova su uno stack diverso da quello corrente.

4.1.3.2 Definizione delle primitive

▷ void **ERIKA_hal_begin_primitive**(void);

Descrizione Ogni primitiva deve iniziare eseguendo questo codice. Su alcune architetture (come h8 ed st10) viene usata per disabilitare le interruzioni e rendere dunque atomica la primitiva.

▷ void **ERIKA_hal_end_primitive**(void);

Descrizione Ogni primitiva deve terminare eseguendo questo codice.

▷ void **ERIKA_hal_IRQ_begin_primitive**(void);

Descrizione Ha lo stesso scopo della *ERIKA_hal_begin_primitive()*, ma deve essere eseguita come prima istruzione di una primitiva che viene invocata da un driver anzichè da un thread.

▷ void **ERIKA_hal_IRQ_end_primitive**(void);

Descrizione Deve essere eseguita come ultima istruzione di una primitiva invocata all'interno di un driver.

4.1.3.3 Gestione delle interruzioni

▷ void **ERIKA_hal_enableIRQ**(void);

Descrizione Abilita le interruzioni.

▷ void **ERIKA_hal_disableIRQ**(void);

Descrizione Disabilita le interruzioni. Eventuali richieste di interruzioni restano pendenti.

▷ void **ERIKA_hal_IRQ_enableIRQ**(void);

Descrizione Abilita le interruzioni. Viene invocata all'interno di un driver.

▷ void **ERIKA_hal_IRQ_disableIRQ**(void);

Descrizione Disabilita le interruzioni. Viene invocata all'interno di un driver.

4.1.3.4 Utilità

▷ ERIKA_TIME **ERIKA_hal_gettime**(void);

Descrizione

ritorna il valore dell'orologio di sistema, utilizzato come riferimento temporale.

▷ void **ERIKA_hal_reboot**(void);

Descrizione Riavvia il sistema.

▷ void **ERIKA_hal_panic**(void);

Descrizione Tipicamente provoca il riavvio del sistema, eventualmente segnalando la causa del malfunzionamento verificatosi.

- void **ERIKA_hal_idle**(void);

  Descrizione Se lo hardware lo prevede, questa primitiva porta il processore in uno stato di risparmio energetico durante il quale non vengono eseguite istruzioni (a seconda dei casi possono anche essere sospese alcune funzionalità del chip): si esce da tale stato in seguito all'arrivo di una interruzione interna o esterna a seconda dei casi.

4.2 Kernel Layer

Da un punto di vista logico questo strato sta sopra lo HAL e fornisce una serie di funzioni raggruppabili in tre categorie:

- gestione delle code
- scheduling
- interfacciamento con l'applicazione

Soltanto le costanti, i tipi, i dati e le funzioni appartenenti a quest'ultima categoria sono utilizzabili dall'applicazione; a differenza che nello HAL, qui elenchiamo le strutture dati utilizzate dal kernel essendo queste indipendenti dall'architettura hardware sottostante.

4.2.1 Tipi di dati e costanti

4.2.1.1 Tipi di dati

La maggior parte di questi tipi dipende dall'applicazione e dunque devono essere definiti nei file opportuni: si veda a tal proposito il paragrafo.

- ERIKA_TYPENACT: numero di attivazioni pendenti di un thread.
- ERIKA_TYPEPRIO: priorità di un thread.
- ERIKA_TYPESTATUS: stato in cui si trova un thread (ready, stacked)[2].

[2]Come sarà precisato più avanti, con il protocollo SRP lo stato bloccato non esiste.

- ERIKA_MUTEX: semaforo di mutua esclusione.
- ERIKA_TID: identificatore di un thread.
- ERIKA_TYPEIRQ: numero associato ad un certo IRQ.
- ERIKA_TYPERELDLINE: numero di tick del timer tramite il quale viene indicata la deadline relativa di un thread.
- ERIKA_TYPEABSDLINE: valore del timer tramite il quale viene indicata la deadline assoluta di un thread.

Osserviamo che i tipi di dati riguardanti le deadline relative ed assolute sono presenti solo se l'algoritmo di schedulazione utilizzato è EDF.

4.2.1.2 Costanti

- ERIKA_NIL: TID non valido.

4.2.2 Strutture dati

- ERIKA_TYPEPRIO ERIKA_th_ready_prio[THREAD_MAX]: priorità del thread.
- ERIKA_TYPEPRIO ERIKA_th_dispatch_prio[THREAD_MAX]: priorità del thread quando va in esecuzione (serve per implementare il protocollo SRPT con i gruppi di non preemption).
- ERIKA_TYPEPRIO ERIKA_mutex_ceiling[MUTEX_MAX]: ceiling associati ad ogni semaforo.
- ERIKA_TYPEPRIO ERIKA_mutex_old_ceiling[MUTEX_MAX]: vecchi ceiling dei semafori. È utilizzata internamente al kernel.
- ERIKA_TYPEPRIO ERIKA_sys_ceiling: ceiling di sistema.
- ERIKA_TYPESTATUS ERIKA_th_status[THREAD_MAX]: stato dei thread.

- ERIKA_TYPENACT ERIKA_th_nact[THREAD_MAX]: numero di attivazioni pendenti dei thread.
- ERIKA_TID ERIKA_th_next[THREAD_MAX]: prossimo task da eseguire rispetto a quello in esecuzione.
- ERIKA_TID ERIKA_rqfirst: primo thread in coda ready.
- ERIKA_TID ERIKA_stkfirst: primo thread in coda stacked.
- ERIKA_TYPERELDLINE ERIKA_th_reldline[THREAD_MAX]: deadline relative dei thread (espresse in numero di tick).
- ERIKA_TYPEABSDLINE ERIKA_th_absdline[THREAD_MAX]: deadline assolute dei thread (espresse in numero di tick).

  Osservazione:

  Per ragioni di efficienza, è stata fatta la scelta progettuale di rendere visibili all'applicazione le strutture dati interne al kernel, imponendo all'applicazione stessa di inizializzarle utilizzando il costrutto ' {...}' del C.

  Se da un lato questa scelta richiede da parte di chi scrive l'applicazione un minimo di conoscenza in più del nucleo, dall'altro lato offre sicuramente dei vantaggi:

  - è così possibile memorizzare in ROM quelle strutture dati che non cambiano durante la vita dell'applicazione (è bene tener presente che la RAM è una risorsa più critica della ROM);
  - il kernel si adatta per l'applicazione specifica, senza che debbano essere fornite ulteriori primitive, e quindi risulta più snello.

4.2.3 Primitive utilizzabili dall'applicazione

4.2.3.1 Gestione dei thread

- void **ERIKA_thread_make_ready**(ERIKA_TID thread, ERIKA_TYPENACT num_attivazioni);

Descrizione Il task individuato da *thread* dovrà essere eseguito per *num_attivazioni* volte; esso viene inserito in coda ready, oppure, se già presente, viene incrementato il numero di attivazioni pendenti di una quantità pari a *num_attivazioni*. Questa primitiva non invoca lo scheduler e dunque può essere convenientemente utilizzata quando si vogliono attivare contemporaneamete[3] più thread: una volta che questi saranno inseriti in coda pronti si potrà invocare lo scheduler.

▷ void **ERIKA_thread_activate**(ERIKA_TID thread);

Descrizione Se il task individuato da *thread* ha priorità più grande di tutti i task presenti in coda ready o stacked allora viene mandato in esecuzione (dunque inserito in coda stacked); in caso contrario viene inserito in coda ready oppure, se già presente, viene incrementato il numero di attivazioni pendenti di quel task, continuando l'esecuzione di chi ha invocato questa primitiva. Come si vede a differenza di *thread_make_ready()*, questa primitiva prevede l'invocazione dello scheduler.[4]

▷ void **ERIKA_sys_scheduler**(void);

Descrizione Viene effettuato il check di preemption: se in coda pronti è presente un thread a priorità maggiore di tutti quelli presenti in coda stacked allora viene mandato in esecuzione (estraendolo dunque dalla coda ready ed inserendolo in coda stacked), altrimenti si prosegure l'esecuzione del thread che ha invocato questa primitiva.

4.2.3.2 Gestione dei thread nei driver

Una di queste due primitive viene invocata sempre alla fine di un driver:

[3] L' implementazione attuale prevede che questa primitiva chiami la funzione *ERIKA_sys_gettime()* e venga calcolata la deadline assoluta. Dunque non c'è una vera attivazione contemporanea dei vari task.

[4] Non viene invocata la primitiva ERIKA_sys_scheduler(), bensì contiene il codice di quest'ultima.

▷ void **ERIKA_IRQ_make_ready(**ERIKA_TID thread, ERIKA_TYPENACT num_attivazioni**);**

Descrizione Esegue le stesse operazioni della *ERIKA_thread_make_ready(...)* solo che viene invocata all'interno di un driver, anzichè di un thread.

4.2.3.3 Gestione delle risorse condivise

▷ void **ERIKA_mutex_lock**(ERIKA_MUTEX mutex_id);

Descrizione Viene bloccata la risorsa a cui è associata la variabile di mutua esclusione *muted_id.* Osserviamo che utilizzando il protocollo SRP, questa primitiva non provoca mai cambio di contesto: il thread che la invoca non si blocca mai e prosegue la sua esecuzione.

▷ void **ERIKA_mutex_unlock**(ERIKA_MUTEX mutex_id);

Descrizione Viene liberata la risorsa a cui è associata la variabile di mutua esclusione *muted_id.* Viene effettuato il check di preemption e dunque potrebbe andare in esecuzione qualche thread che attendeva il rilascio (osserviamo che tale thread si trova eventualmente in coda pronti, dato che l'utilizzo di SRP impedisce la presenza di processi bloccati).

4.2.3.4 Installazione dei driver

▷ void **ERIKA_set_handler**(ERIKA_TYPEIRQ int_id, ERIKA_ADDR func);

Descrizione In seguito ad una richiesta di interruzione di tipo *int_id* viene eseguito il driver. *func*. In alcune architetture, come ad esempio su H8,

4.2.3.5 Utilità

▷ **ERIKA_TIME ERIKA_sys_gettime**(void);

Descrizione Invoca la corrispondente funzione dello HAL.

▷ void **ERIKA_sys_reboot**(void);

Descrizione Invoca la corrispondente funzione dello HAL.

▷ void **ERIKA_sys_panic**(void);

Descrizione Invoca la corrispondente funzione dello HAL.

▷ void **ERIKA_sys_idle**(void);

Descrizione Invoca la corrispondente funzione dello HAL.

4.3 Stati di un thread

Ad ogni istante un thread può trovarsi in uno dei tre seguenti stati:

▷ **ready**: il processo è stato attivato, ma non è ancora andato in esecuzione;

▷ **stacked**: il processo è in esecuzione, oppure è stato interrotto ed è in attesa di terminare la sua esecuzione.

▷ **idle**: il processo ha terminato la propria esecuzione ma non è stato ancora riattivato.

Osserviamo che con entrambi gli algoritmi di scheduling utilizzati (RM ed EDF), viene utilizzato il protocollo SRP (Stack Resource Policy[5]) e dunque non si prevede uno stato 'bloccato' in quanto tale protocollo ci garantisce che se un thread va in esecuzione questo può subire preemption ma non si bloccherà mai.

Il descrittore di ciascun thread viene pertanto inserito in coda ready, oppure in coda stacked a seconda dello stato in cui si trova. Quando un thread non è nè in coda ready, nè in coda stacked significa che è nello stato idle.

[5] In realtà viene utilizzato il protocollo SRPT, ovvero SRP con soglia di preemption.

4.4 Struttura di un thread

Tutti i thread che costituiscono l'applicazione devono risiedere in RAM. Ogni thread per poter essere eseguito deve anzitutto essere attivato e dunque inserito in coda pronti. Da questo momento in poi sarà lo scheduler a mandarlo in esecuzione quando possibile.

Anche se viene utilizzato il modello *one-shot*, i thread non possono chiamare esplicitamente la primitiva *ERIKA_thread_end_istance()*: infatti questa primitiva non viene esportata dal kernel verso lo strato applicativo. La primitiva viene comunque chiamata dal kernel quando un thread ha eseguito l'ultima istruzione del suo corpo.

La primitiva rimuove dallo stack il frame relativo al thread terminato che viene portato nello stato idle. È dunque evidente che per ottenere un thread periodico, si dovrà utilizzare un timer che provveda ad attivare periodicamente il thread stesso.

Figura 4.1: Thread

void **thread1** (void)

```
{
    /* corpo del thread */
     ..............
    /* fine dell'istanza corrente del thread1 */
}
```

4.5 Il thread dummy()

Ogni applicazione deve definire un thread avente la seguente struttura:

Questo è un thread particolare, in quanto:

- non ha un TID;

Figura 4.2: dummy()

void dummy(void)

```
{
   /* inizializzazione di sistema */
   ............
   /* attivazione dei thread */
   ............
   /* invocazione dello scheduler */
   ............
   /* loop infinito */
   for (;;)
   {
      /* azioni da effettuare in background oppure sys_idle() */
   }
}
```

- è il primo thread ad andare in esecuzione;

- è il trhead a priorità più bassa;

4.6 Macro per la configurazione del kernel

Riportiamo nella tabella 4.1 le macro che l'applicazione deve definire per configurare opportunamente il kernel, stabilendo pertanto l'architettura per la quale l'applicazione stessa è stata creata, il tipo di stack (mono o multi), l'algoritmo di scheduling, ecc...

Tabella 4.1: Macro per la configurazione del kernel

Macro	**Descrizione**
__ST10__	L'applicazione è scritta per girare su una scheda ST10.
__ARM7ADS__, __ARM7GNU__	L'applicazione è scritta per girare su una cpu ARM7 su scheda Evaluator 7T.
__H8__	L'applicazione è scritta per girare sul microcontrollore Hitachi H8/300.
__MPC5XX__	L'applicazione è scritta per girare su un microcontrollore Motorola PowerPC
__MONO__	HAL monostack.
__MULTI__	HAL multistack.
__SEGM__ (solo st10)	HAL multistack segmentato.
__FP__	Algoritmo di scheduling FP.
__SRPT__	Algoritmo di scheduling EDF.
__TIME_SUPPORT__	Vengono rese disponibili primitive apposite per leggere l'orologio di sistema (riferimento temporale).
__SEM__	L' applicazione usa i semafori.
__DEBUG__	Quando viene compilata l' applicazione vengono generati file aggiuntivi utili per il debug.

Ovviamente le macro indicate nella tabella sopra, sono comuni a tutte le architetture, mentre ciascuna di queste ultime poi metterà a disposizione altre macro per l'utilizzzo di funzionalità specifiche offerte dall'implementazione di ERIKA su quell'architettura stessa.

Capitolo 5

Porting di ERIKA su MPC566

In questo capitolo vengono presentati i concetti e le tecniche utilizzate per implementare ERIKA sul dispositivo MPC566, estendendo il kernel con le funzionalità necessarie per consentire alle applicazioni di utilizzare i dispositivi hardware presenti sul chip del microcontrollore.

La parte kernel layer di ERIKA resta invariata come avevamo accennato precedentemente, quindi verrà affrontata l'implementazione dello HAL, sia nel caso *monostack* che in quello *multistack*.

Si ricorda inoltre che per compilare il kernel è stato utilizzato il compilatore GNU/GCC 3.3.3 [Sta] aderente allo standard EABI (3.3).

5.1 Hardware Abstraction Layer (HAL)

Come stabilito dalle convenzioni sull'utilizzo dei registri (tabella 3.8 a pagina 61) i registri volatili vengono considerati *sporchi* al ritorno da una funzione chiamata. É quindi del tutto inutile considerare questi registri parte del contesto del thread se non nel caso in cui questo viene interrotto da un driver: ciò è ovvio considerato che la richiesta di interruzione è un evento asincrono e dunque non può essere fatta nessuna assunzione in merito.

Bisogna inoltre pensare che quando un thread chiama una primitiva[1] questa

[1] Si ricorda che la call di una primitiva di sistema è fatta con il meccanismo di chiamata a funzione

potrebbe ritornare nel thread che l'ha chiamata come in un altro thread, quello a priorità più alta che magari è stato attivato con una `ERIKA_thread_activate(...)`. Visto che il nuovo thread viene chiamato con la stessa modalità di una chiamata a funzione sarà lui ad effettuare il salvataggio dei registri non-volatili che utilizzerà, quindi, alla primitiva, non resta che salvare i registri non-volatili da lei esclusivamente utilizzati –consentendo un risparmio di tempo e di spazio. In questo caso si è comunque scelto di salvare tutto il contesto breve: grazie all'istruzione **`stmw`** siamo in grado di salvare tutti i registri non-volatili in un solo colpo.

Per questi motivi sono stati definiti vari tipi di contesto per il thread – figura 5.1 e figura 5.2.

Figura 5.1: Contesto breve

LR	CR	R14	$\cdots$	R31

Figura 5.2: Contesto completo

LR	CR	R0	R2	$\cdots$	R31

Del contesto breve fanno parte i soli registri non volatili, mentre di quello completo fanno parte tutti i registri del microcontrollore.

Sono stati omessi dal contesto completo i registri SRR0 e SRR1 in cui vengono salvati il program counter ed il contenuto del registro MSR quando viene accettata un'eccezione.

> **Nota:**
>
> *Il registro R1 è lo stack pointer e quindi non fa parte del contesto del thread.*

Per quanto riguarda la gestione delle interruzioni si hanno due possibilità: la prima è eseguire l'handler ad interruzioni disabilitate, scelta obbligata per l'implementazione MONOSTACK, e la seconda di consentire l'annidamento delle interruzioni. Queste due possibilità vengono espresse tramite l'opzione di compilazione **__ALLOW_NESTED_IRQ__** che richiede comunque che sia espressa l'opzione **__MULTI__**.

***Nota*:**

in questo capitolo non sono presenti i sorgenti del codice implementato perchè E.R.I.K.A. è coperta da copyright da parte di Evidence S.r.l.

5.1.1 HAL Monostack

In questa situazione abbiamo un unico stack la cui cima è indicata dal registro R1: quindi tutti i frame creati dai vari thread e gli handler utilizzano questo stack. Di seguito vengono riportate le primitive esportate dallo HAL verso lo strato Kernel, specificando le eventuali scelte fatte per il particolare microcontrollore:

- Primitiva **ERIKA_hal_ready2stacked()** – viene mandato in esecuzione il thread specificato che creerà il suo frame sopra a quello del thread che ha subito preemption. Per questo thread occorre salvare il contesto breve.

- Primitiva **ERIKA_hal_endcycle_stacked()** – vengono ripristinati i registri non volatili, contesto breve, del thread che aveva subito preemption e in seguito viene rimosso dallo stack il frame relativo al thread che sta per terminare.

- Primitiva **ERIKA_hal_endcycle_ready()** – il thread che sta per andare in esecuzione non ha bisogno di nessun ripristino, infatti essendo READY non aveva nessun frame allocato nello stack. Lo stack frame relativo al thread che sta per terminare viene rimosso dallo stack.

- Primitiva **ERIKA_hal_IRQ_stacked()** – svolge la stessa funzione della primitva ERIKA_hal_endcycle_stacked() se non che sta per terminare un driver e non un thread.

- Primitiva **ERIKA_hal_IRQ_ready()** – svolge la stessa funzione della primitiva ERIKA_hal_endcycle_ready() se non che sta per terminare un driver e non un thread.

5.1.2 HAL Multistack

Ogni thread, compreso il *dummy*, può avere un proprio stack oppure condividerlo con altri thread. Per quanto riguarda gli handler delle interruzioni abbiamo due scelte: se specificata l'opzione __IRQ_STACK_NEEDED__ allora hanno un loro stack su cui creano i loro frame, altrimenti creano il frame sullo stack del thread interrotto.

Rispetto al caso monostack dobbiamo avere un modo per sapere che il thread aveva subito preemption, nel qual caso dobbiamo ripristinare il suo contesto e ritornare al suo stack, oppure era ready, cioè non aveva nessun frame allocato e dobbiamo solo saltare al suo stack.

Le primitive illustrate nel caso monostack mantengono la loro funzione anche nel caso multistack, con la sola accortezza di cambiare stack quando viene eseguito un cambio di contesto. Abbiamo inoltre una primitiva in più, non presente nel caso monostack:

- ▷ Primitiva **ERIKA_hal_stkchange()** – viene salvato il contesto breve del thread che l'ha invocata, in seguito si procede al cambio di stack. Il nuovo thread che va in esecuzione era un thread STACKED, quindi ha il suo frame allocato nel nuovo stack puntato da R1.

5.1.3 Primitive comuni a Monostack e Multistack

In questo paragrafo vengono elencate solo le primitive che, nell'implementazione specifica, richiedono delle considerazioni particolari rispetto alla loro descrizione in 4.1.3.

- ▷ Primitiva **ERIKA_hal_begin_primitive()** – questa primitiva viene utilizzata per disabilitare gli interrupt. Nell'implementazione MPC si traduce in una sola istruzione.

- ▷ Primitiva **ERIKA_hal_end_primitive()** – questa primitiva, dualmente alla precedente, serve a riabiliatere le interruzioni. Anche in questo caso si traduce in una sola istruzione in linguaggio assembler.

- Primitiva **ERIKA_hal_IRQ_begin_primitve()** – svolge la funzione di disabilitare le richieste di interruzione ma all'interno di un driver. Come detto precedentemente non sempre sono consentite richieste annidate, e per questo nell'implementazione monostack non fa nulla.

- Primitiva **ERIKA_hal_IRQ_end_primitve()** – svolge la funzione di riabilitare le richieste di interruzione ma all'interno di un driver. Come per la primitiva predente non esegue nulla nel caso monostack.

- Primitiva **ERIKA_hal_gettime()** – questa funzione viene utilizzata per leggere il valore del *real-time clock* presente sul chip del microcontrollore. Deve essere esplicitamente selezionata con l'opzione di compilazione __TIME_SUPPORT__.

5.1.4 Strutture Dati

Come era stato accennato nel paragrafo 4.1.2 abbiamo utilizzato un vettore contenente l'indirizzo dei corpi dei thread, sia nel caso monostack che in quello multistack:

- **ERIKA_ADDR ERIKA_mpc5_thread_body[THREAD_MAX]**;

Per quello che riguarda l'implementazione monostack non state utilizzate altre strutture dati mentre per quella multistack sono state utilizzate strutture dati descritte in seguito.

Strutture Dati usate per il Multistack

- **struct ERIKA_TOS ERIKA_mpc5_system_tos[STACK_MAX+1]** – in questo vettore vengono memorizzati gli stack pointer di ogni stack

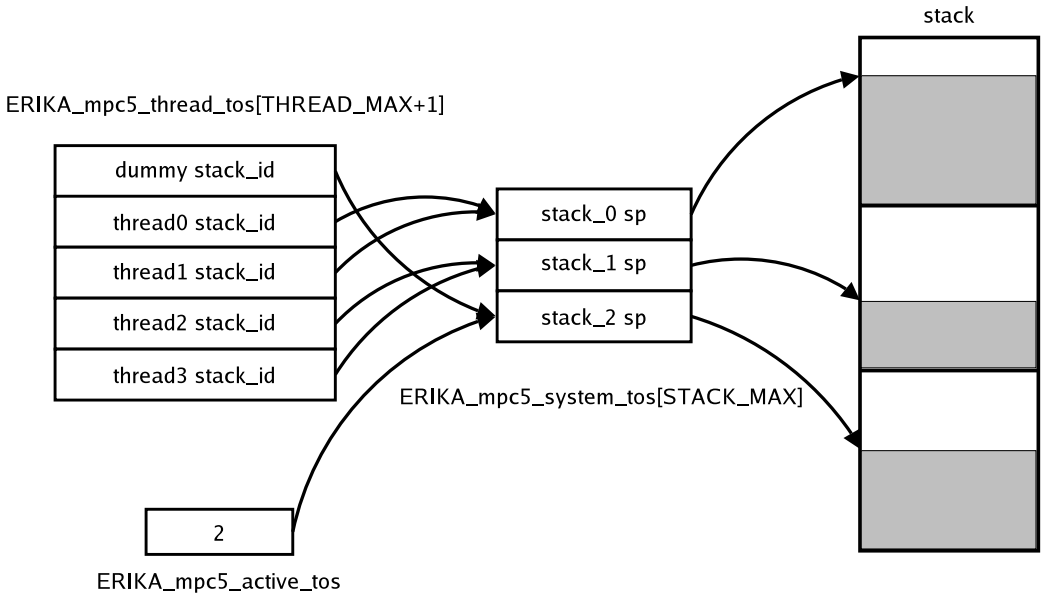
- **ERIKA_UREG ERIKA_mpc5_thread_tos[THREAD_MAX+1]** – ad ogni thread, compreso il dummy, viene associato l'identificatore dello stack che gli è stato assegnato.

- **ERIKA_UREG ERIKA_mpc5_active_tos** – contiene l'identificatore dello stack relativo al thread attualmente in esecuzione

Per meglio comprendere queste strutture dati riferirsi alla figura 5.3:

- ▷ nel sistema sono presenti 4 thread – THREAD_MAX = 4
- ▷ vengono utilizzati 3 stack – STACK_MAX = 3
- ▷ nel caso specifico il thread attivo, *thread3*, utilizza lo stack con stack_id = 2

Figura 5.3: Strutture Dati HAL Multistack



Per quanto riguarda la gestione delle interruzioni vengono utilizzate le seguenti strutture:

- ▷ **ERIKA_ADDR ERIKA_mpc5_irq_table[$\cdots$]** – ad ogni entry di questo vettore corrisponde un'IRQ (entry con indice pari) o un livello di interrupt (entry con indice dispari). Il valore ad un certo indice è l'indirizzo della funzione handler, se è specificato, o ERIKA_NULL, se l'handler non è specificato.

5.2 Driver per Dispositivi Interni

Sono stati realizzate varie funzioni per la configurazione dei dispositivi interni del microcontrollore, in particolare per il PIT ed il Real-Time Clock. Di seguito verranno esposte i prototipi di queste funzioni con una breve descrizione.

> **Nota:**
>
> L'utilizzo di una delle due variabili di compilazione previste per utilizzare il PIT o il Real-Time Clock impone l'utilizzo dell'opzione __TIMER_USED__.

5.2.1 Periodic Interrupt Timer

Tramite l'opzione di compilazione **__PIT_USED__** vengono rese visibili le seguenti funzioni:

- **void ERIKA_pit_start(void)**

  Descrizione Questa funzione fa partire il countdown del timer.

- **void ERIKA_pit_stop(void)**

  Descrizione Questa funzione fa terminare il countdown del timer.

- **ERIKA_UREG ERIKA_pit_get(void)**

  Descrizione Tramite questa funzione è possibile leggere il valore del registro che viene decrementato.

- **void ERIKA_pit_set(ERIKA_UREG count_down)**

  Descrizione Viene utilizzata per impostare il valore iniziale del registro di countdown.

- **void ERIKA_pit_enable(void)**

  Descrizione Serve per abilitare il PIT ad inviare la richiesta di interruzione.

▷ **void ERIKA_pit_disable(void)**

Descrizione Viene utilizzata per disabilitare il PIT all'invio della richiesta di interruzione.

▷ **void ERIKA_pit_set_IRQlevel(ERIKA_TYPEIRQ level)**

Descrizione Viene utilizzata per scrivere nel registro di configurazione del PIT il livello di interrupt che gli è stato assegnato.

▷ **void ERIKA_pit_freeze(void)**

Descrizione Serve per sospendere il countdown del PIT.

▷ **void ERIKA_pit_resume(void)**

Descrizione Serve per riprendere il countdown del PIT.

5.2.2 Real-Time Clock

Tramite l'opzione di compilazione **__RT_CLOCK_USED__** vengono rese visibili le seguenti funzioni:

▷ **void ERIKA_rt_clock_start(void)**

Descrizione Questa funzione fa partire il contatore del timer.

▷ **void ERIKA_rt_clock_stop(void)**

Descrizione Questa funzione fa terminare l'incremento del registro contatore del timer.

▷ **ERIKA_UREG ERIKA_rt_clock_get(void)**

Descrizione Tramite questa funzione è possibile leggere il valore del registro contatore.

▷ **void ERIKA_rt_clock_alarm_set(ERIKA_UREG alarm)**

Descrizione Viene utilizzata per impostare il valore del registro di alarm con cui il valore del contatore viene confrontato.

- ▷ **void ERIKA_rt_clock_alarm_enable(void)**

  Descrizione Serve per abilitare il real-time clock ad inviare la richiesta di interruzione quando viene raggiunto il valore contenuto nel registro alarm.

- ▷ **void ERIKA_rt_clock_alarm_disable(void)**

  Descrizione Viene utilizzata per disabilitare il real-time clock all'invio della richiesta di interruzione.

- ▷ **void ERIKA_rt_clock_sec_enable(void)**

  Descrizione Serve per abilitare il real-time clock ad inviare la richiesta di interruzione ogni secondo.

- ▷ **void ERIKA_rt_clock_sec_disable(void)**

  Descrizione Viene utilizzata per disabilitare il real-time clock all'invio della richiesta di interruzione ogni secondo.

- ▷ **void ERIKA_rt_clock_set_IRQlevel(ERIKA_TYPEIRQ level)**

  Descrizione Viene utilizzata per scrivere nel registro di configurazione del real-time clock il livello di interrupt che gli è stato assegnato.

- ▷ **void ERIKA_rt_clock_freeze(void)**

  Descrizione Serve per sospendere l'incremento del registro contatore.

- ▷ **void ERIKA_rt_clock_resume(void)**

  Descrizione Serve per riprendere l'incremento del registro contatore.

Capitolo 6

Conclusioni

In questa tesi sono stati richiamati i concetti generali relativi ai cosiddetti sistemi embedded, mettendo in evidenza quali sono le caratteristiche che essi presentano. Questo ha consentito di introdurre il problema di adottare delle opportune tecniche di progetto sia a livello software che hardware.

In particolare in questo ambito ci siamo occupati della realizzazione della parte software di un sistema operativo real-time embedded. É stato pertanto progettato e successivamente implementato un sistema operativo che presenta le seguenti caratteristiche:

- ▷ minimizzazione dell'occupazione di memoria: è stata ridotta al minimo soprattutto la quantità di RAM necessaria (RAM *footprint*), essendo questa una risorsa ancora più critica della memoria ROM sui microcontrollori tipicamente utilizzati nei sistemi embedded;

- ▷ la portabilità verso architetture differenti: in questo modo si riducono i tempi di sviluppo qualora il sistema dovesse essere utilizzato su un processore diverso. Questo obiettivo è stato ottimamente raggiunto suddividendo il kernel in due strati (HAL e kernel layer), in modo tale che il porting da una architettura ad un' altra necessiti soltanto di modificare lo HAL ;

- ▷ la riutilizzabilità del software: abbiamo realizzato una API il più possibile indipendente dal particolare processore sottostante, in modo tale che

sia possibile condividere buona parte del software scritto per applicazioni diverse;

- ▷ la modularità del kernel: in questo modo il sistema operativo si adatta alle caratteristiche dell' applicazione; ciò è stato ottenuto defininendo una macro per ognuna delle funzionalità offerte dal kernel, in modo tale che l'applicazione si limiti a dichiarare (defininendo la macro opportuna) quali funzionalità intende usare;

- ▷ garanzia del rispetto dei vincoli temporali: il nucleo è stato pensato affinchè possa garantire il corretto scheduling dei task, siano essi periodici che aperiodici o sporadici.

Come conclusione del lavoro sono stati affrontati vari test per garantire la correttezza delle componenti critiche del sistema. Sono state testati sia lo HAL monostack che quello multistack: in entrambi i casi sono stati riportati dei buoni risultati sia come occupazione di memoria, vincolo critico nei sistemi embedded, sia come tempi di risposta del sistema.

Questi risultati ottenuti hanno dimostrato la bontà e la correttezza del sistema operativo che è stato realizzato nel corso di questa opera.

Bibliografia

[AB87] Paolo Ancilotti and Maurelio Boari. *Principi e Tecniche di Programmazione Concorrente*. UTET Libreria, 1987.

[arm95] *The ARM CPU Data Sheet*, August 1995. Sigla documento ARM DDI 0029E, prelevabile in formato elettronico allo URL http://www.arm.com.

[ASV01] Grant Martin A. Sangiovanni-Vincentelli. Platform-based design and software design methodology for embedded systems. *IEEE Design and test of Computers*, December 2001.

[Bak91] T. P. Baker. Stacked-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3:67–99, 1991.

[But95] Giorgio C. Buttazzo. *Sistemi in Tempo Reale*. Pitagora Editrice, Bologna, 1995.

[EAB98] IBM Microelectronics. *PowerPC Embedded Processors Application Note*, version 1.0 edition, September 1998. Developing PowerPC EABI Compliant Programs.

[GLDN01] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. May 2001.

[Hor74] W. Horn. Some simple scheduling algorithms. In *Naval Research Logistics Quarterly*, volume 21, pages 177–185, 1974.

[JDM01] Josef Fuchs John Dunlop and Steve Mihalik. *MPC555 Interrupts*. Motorola Semiconductor, July 2001. Application Note Revision 0.

[KK00] A. Richard Newton Jan M. Rabaey A. Sangiovanni-Vincentelli Kurt Keutzer, Sharad Malik. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on computer aided design of integratedd circuits and systems*, December 2000.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[MRM03] Motorola. *MPC565/MPC566 Reference Manual*, September 2003. Revision 1.

[PEM01] Motorola. *Programming Environment Manual For 32-Bit Implementation of the PowerPC Architecture*, December 2001. Revision 2.

[RCP99] Motorola. *RCPU Risc Central Processing Unit Reference Manual*, February 1999.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[ST00] STMICRO. *Family Programming Manual*, jan 2000. Prelevabile in formato elettronico allo URL http://www.st.com.

[Sta] Richard M. Stallman. *Using the GNU Compiler Collection*. GNU GCC 3.3.3, 30 December 2002.

[SW99] M. Saksena and Y. Wang. Scheduling fixed-priority tasks with preemption threshold. In *International Conference on Real-Time Computing Systems and Applications*, December 1999.