POLITECNICO DI TORINO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea

Progetto e Realizzazione del Satellite PICPOT: Scheda di Acquisizione Immagini

Relatori:
ing. Claudio Passerone
prof. Leonardo M. Reyneri

Candidato:
Mauro Caule

Settembre 2005

Indice

Sommario

Sommario

Il progetto PICPOT, PICosatellite del POlitecnico di Torino, è nato nel Gennaio 2004 con lo scopo di costruire nell'arco di un anno un picosatellite universitario, di massa minore di 1 kg **??**, con finalità educative e di ricerca; PICPOT è inserito in un progetto più ampio, costituito da una costellazione di satelliti universitari italiani, attualmente in via di definizione. Il progetto del satellite si è basato su diversi requisiti, quali forma cubica con lato di 13 cm, massa inferiore ad 2 kg, potenza non superiore a 1.5 W, almeno 90 giorni di vita, orbita LEO (Low Earth Orbit) e compatibile con il lanciatore POD (Pico Orbital Deployer).

PICPOT ha come obiettivi quelli di verificare il funzionamento di componenti COTS (Components Off The Shelf) nello spazio, trasmettere dati telemetrici (temperature, tensioni, correnti, etc...) alla stazione di Terra e scattare delle fotografie dallo spazio. La sua struttura esterna definitiva è un cubo di sei facce quadrate ed ortogonali tra loro, ricoperto di cinque pannelli solari, dotato esternamente di due antenne (2.4 GHz, 435 MHz), tre fotocamere, due kill-switch ed un connettore di test.

La parte elettronica del satellite è formata da 6 schede che sovraintendono alle varie funzioni previste; nel seguito viene data una breve descrizione di ciascuna scheda:

- *PowerSupply*: mantiene cariche le batterie ricaribili del satellite e controlla lo stato elettrico e termico dei pannelli, dei caricabatteria e delle batterie stesse.
- *PowerSwitch*: alimenta le altre schede generando le tensioni necessarie a partire dalle tensioni delle batterie.
- *Tx/Rx*: trasmette e riceve i segnali tra il satellite PICPOT e la stazione di Terra, alle frequenze di 435 MHz e 2.4 GHz nella banda dedicata alle comunicazioni satellitari amatoriali.
- *ProcA* e *ProcB*: elaborano i dati ricevuti da Terra e trasmettono le fotografie e la telemetria del satellite, alle due diverse frequenze. Le due schede, ognuna dotata di un microprocessore, hanno funzioni simili duplicate per ridondanza.
- *Payload*: acquisisce le immagini dalle tre fotocamere, le comprime in formato JPEG e le trasmette su richiesta ad uno dei due microprocessori di bordo.

Il lavoro svolto in questa tesi riguarda in particolare il progetto hardware e software, la realizzazione e il collaudo della scheda Payload. Nella sua versione definitiva essa adotta un Digital Signal Processor (DSP) a 16 bit prodotto da Analog Devices, chiamato Blackfin, che può raggiungere la massima frequenza di clock di 400 MHz. Il sottosistema di memorizzazione include

una memoria SDRAM da 8 Mbyte, una memoria SRAM da 2 Mbyte e una memoria FLASH non volatile da 2Mbyte. Inoltre per l'acquisizione delle foto, il processore è interfacciato, mediante una porta parallela a 8 bit dedicata, ad un decoder video, il quale converte in digitale il segnale analogico PAL proveniente dalle videocamere e lo codifica in un formato standard chiamato ITU-R BT.656-4. Il collegamento con i due processori di controllo delle schede ProcA e ProcB avviene tramite porte seriali, rispettivamente asincrona (UART) e sincrona (SPI). Per la programmazione e il debug è presente un'interfaccia JTAG.

Il progetto della scheda è iniziato da un'attenta fase di analisi delle specifiche richieste, riportate sul documento di descrizione del sistema, cercando di capire quali tipi di componenti fossero necessari, in relazione soprattutto ai vincoli ambientali previsti e all'interazione dei componenti con raggi cosmici o ioni pesanti presenti alla quota del satellite. Per esempio, una delle decisioni prese in questa fase riguarda la selezione della memoria che contiene il programma da eseguire sul DSP, in cui uno dei requisiti fondamentali è la robustezza rispetto a errori e/o modifiche accidentali su singoli bit, per garantire il buon funzionamento del sistema. Dopo attente ricerche, è stata quindi selezionata una memoria FLASH a gate NOR.

Il passo successivo è stato quello di cercare sul mercato dei componenti commerciali in grado di rispondere alle esigenze di progetto principali, nell'ottica del sistema in cui dovrà funzionare e delle condizioni ambientali interne; l'aspetto del consumo di potenza assume un ruolo di fondamentale importanza, sia perché l'intero sistema è alimentato esclusivamente da batterie con limitato contenuto energetico, sia a causa della ridotta capacità di dissipazione della potenza sotto forma di calore data l'assenza di atmosfera a 600 km dalla superficie terrestre. Altri requisiti ricavati dalle specifiche per il processore riguardano la presenza di una porta video compatibile con quella implementata sul decoder adottato, un DMA controller per l'acquisizione dell'immagine e una struttura ottimizzata per le operazioni più complesse e lunghe che avrebbe dovuto eseguire, cioè la compressine JPEG. Per questo ultimo motivo ci si è indirizzati verso l'adozione di un DSP. La scelta dei componenti è anche condizionata dalla loro disponibilità e reperebilità presso i distributori utilizzati dal progetto PICPOT, quali DigiKey, RS Components e Farnell.

Definita l'architettura di sistema e scelti i componenti da adottare, la fase successiva è consistita nel progetto schematico e nel disegno del circuito stampato (PCB) della scheda, utilizzando il pacchetto software CAE (Computer Aided Design) di Mentor Graphics. Ogni componente inserito nel database è caratterizzato da due viste distinte: un simbolo logico e una cella fisica. La prima viene utilizzata a livello schematico, mentre la seconda, che definisce le dimensioni del package e la sua occupazione di area sulla scheda (footprint) viene usata per creare il PCB.

Dopo aver ordinato i componenti, coordinandoli con quelli utilizzati dalle altre schede, e aver mandato a fabbricare il PCB attraverso i files gerber, si è passati alla fase di montaggio e saldatura dei componenti sulla scheda. Questa operazione è stata molto delicata data la compattezza dei package adottati, tutti di tipo SMD, con distanza minima tra i piedini in certi casi pari a 0.5mm. La scheda montata è stata quindi collaudata mediante piccoli programmi di test appositamente sviluppati.

Lo sviluppo del software si è avvalso inizialmente di una scheda di valutazione dimostrativa prodotta da Analog Devices, ma ha poi avuto un forte impulso una volta resasi disponibile la prima scheda prototipo, in quanto alcuni dei componenti periferici utilizzati erano diversi. Allo scopo di ottenere un sistema perfettamente funzionante, nella prima stesura del software ci si è concentrati soprattutto sulla corretta inizializzazione dei registri di configurazione del core e delle diverse periferiche del processore, quali PLL, DMA, interrupt, UART e controller per memorie sincrone e asincrone.

Non essendo possibile la comunicazione con le schede ProcA e ProcB, perché non ancora disponibili, si è creato un canale di comunicazione tra la scheda e il PC, alternativo alla porta JTAG. Si è infatti implementato un driver per la porta UART per la comunicazione serial RS-232 e per il trasferimento bidirezionale PC – Payload di file o stream di dati utilizzando il protocollo Xmodem. Ciò ha permesso in parte di emulare la presenza delle altre schede del sistema satellite.

Si è testata quindi tutta la catena d'acquisizione video. Le immagini acquisite dalle videocamere vengono inizialmente immagazzinate in memoria nel loro formato originale, costitutio da due field interlacciati aventi un campione di luminanza per ogni pixel e due campioni di crominanza per ogni quadretto 2x2 di pixel (formato YCrCb 4:2:0). Il software esegue perciò le seguenti funzioni: 1) deinterlacciamento; 2) up-sampling della risoluzione della crominanza; 3) compressione in formato JPEG. Per quest'ultimo passo si è sfruttata una libreria liberamente disponibile di funzioni per la manipolazione e la trasformazione di immagini JPEG, opportunamente sfrondata da quanto non necessario e ottimizzata per l'esecuzione sul DSP.

Tutte le operazioni sopra elencate vengono eseguite, per ragioni di prestazioni, in memoria RAM. Il risultato dell'elaborazione, cioè l'immagine JPEG, deve però essere memorizzata in modo permanente in attesa della sua successiva trasmissione. Si è quindi sviluppato un driver per la memoria FLASH adottata, partendo da un codice sorgente fornito dalla ST Microelectronics. Anche questo codice è stato modificato e adattato alle esigenze della scheda Payload, per poi poter salvare sulla memoria FLASH sia le immagini JPEG che il programma da eseguire.

Il progetto della parte hardware è concluso e completamente testato mediante la simulazione delle diverse fasi di funzionamento che richiedevano l'utilizzo di tutti i componenti presenti sulla scheda,

mentre per quanto riguarda la parte software, è stata sviluppata la parte utile ad acquisire, comprimere e trasmettere via UART le foto JPEG, ma comunque deve essere completata la parte di coordinamento con le altre schede, soprattutto per quanto riguarda l'analisi dei telecomandi e l'interfaccia SPI.

Capitolo 1

Introduzione

1.1 Il progetto PICPOT

Il progetto PICPOT, PICosatellite del POlitecnico di Torino, è nato nel Gennaio 2004 con l'obiettivo di costruire nell'arco di un anno un picosatellite universitario, con una massa minore di 1 kg, a scopo educativo e di ricerca.
Il punto di partenza del progetto è stato il concetto di Cubesat.

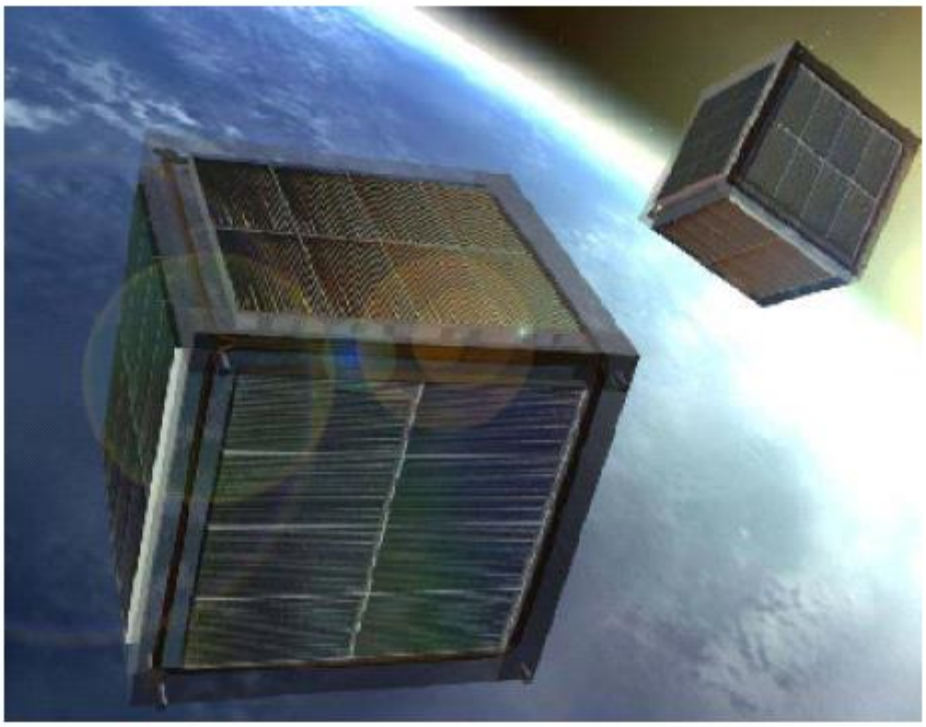


Figura 1.1 Prototipo di Cubesat ideato dalla CalPoly e dalla Stanford University

Cubesat significa satellite di forma cubica ed è nel contempo una filosofia progettuale.
E' uno standard per picosatelliti sviluppato nel 2001 dal Professore Robert Twiggs, docente alla Stanford University, USA, in collaborazione con la Space Systems Development Laboratory (SSDL) della Stanford University e la California Polytechnic State University, USA, per permettere

alle Università che intendono partecipare a questa sperimentazione di realizzare il proprio satellite e di mandarlo nello spazio a costi contenuti.

Un Cubesat è una scatola di forma cubica di 10 cm di lato e con una massa di 1 kg la cui struttura `e definita in funzione dell'adattamento al lanciatore POD (Picosatellie Orbital Deployer).

L'idea alla base di Cubesat è dunque quella di permettere agli studenti dei vari Dipartimenti di Ingegneria di cooperare per la realizzazione di un progetto globale che richiede nel contempo ottime conoscenze in campo ingegneristico e capacità sviluppate di risolvere di problemi di natura interdisciplinare.

L'idea di progettare un satellite è stata adottata da numerose Università italiane e straniere, fra le quali citiamo l'Universitat Wurzburg,(D), la Norwegian University of Science and Technology (NO), l'Aalborg University (DK) e l'Universit`a della Sapienza(IT).

Successivamente il Politecnico di Torino si `e impegnato a partecipare alla progettazione ideando PICPOT e collocandolo in un progetto più ampio, quello di una costellazione di satelliti universitari italiani, attualmente in via di definizione.

La figura 1.2 `e una fotografia della parte esterna del satellite, progettata e costruita dagli studenti del Dipartimento di Ingegneria Areonautica e Spaziale del Politecnico di Torino.

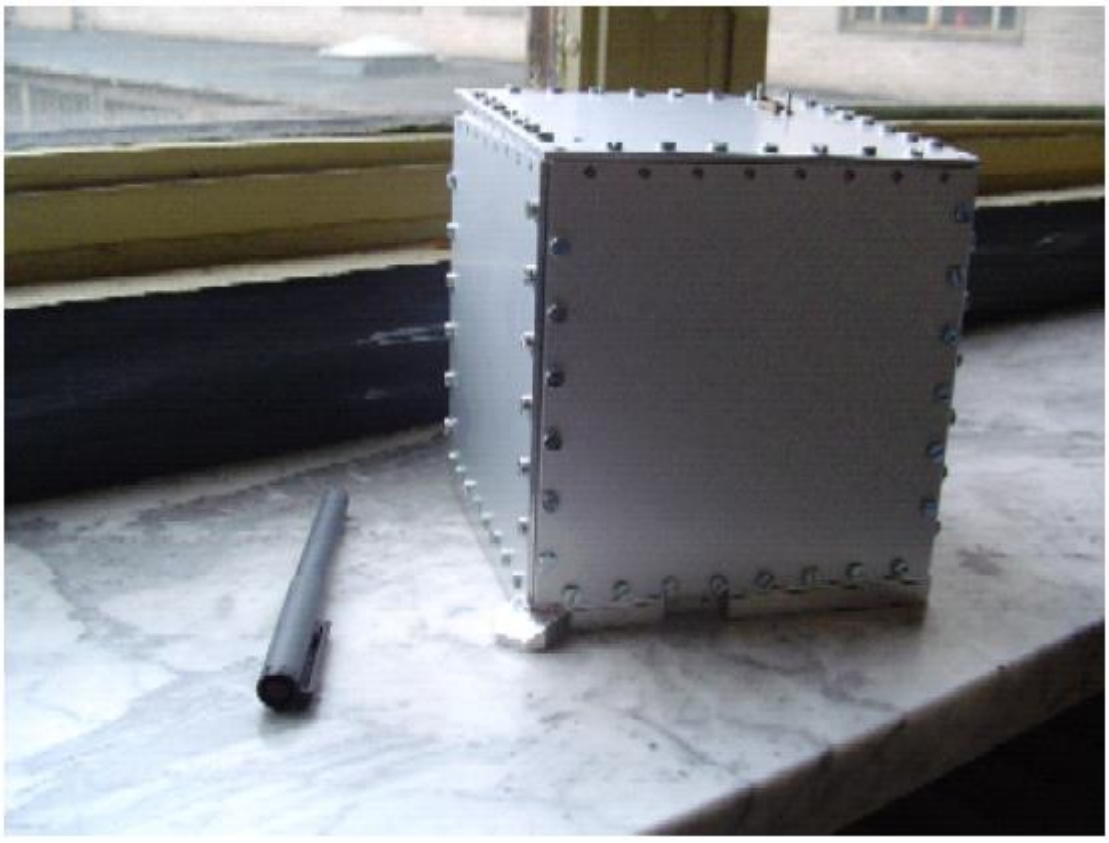


Figura 1.2 PICPOT-PICosatellite POlitecnico di Torino

Il progetto del satellite si è basato sui seguenti requisiti:

- forma cubica con lato di 13 cm
- massa inferiore ad 2 kg
- potenza non superiore a 1.5 W
- almeno 90 giorni di vita
- orbita LEO
- compatibilie con il lanciatore POD

Inizialmente PICPOT aveva come obiettivi quelli di verificare il funzionamento di componenti COTS (Components Off The Shelf) nello spazio, trasmettere dei dati alla stazione di Terra, scattare una o più fotografie e valutare il funzionamento del GPS in orbita LEO.
Per motivi economici e di tempo, non `e stato possibile dotare il satellite del GPS ma questo verrà realizzato in una delle prossime versioni.
Nella figura 1.4 è rappresentata la struttura esterna definitiva di PICPOT: è un cubo dotato di sei facce quadrate ed ortogonali tra loro, di circa 13 cm di lato, ricoperto di pannelli solari (P1, P2, P3, P4, P5) e dotato esternamente di due antenne (2.4 GHz, 435 MHz), tre fotocamere(T1, T2, T3), due kill-switch (K1, K2) ed un connettore di test.

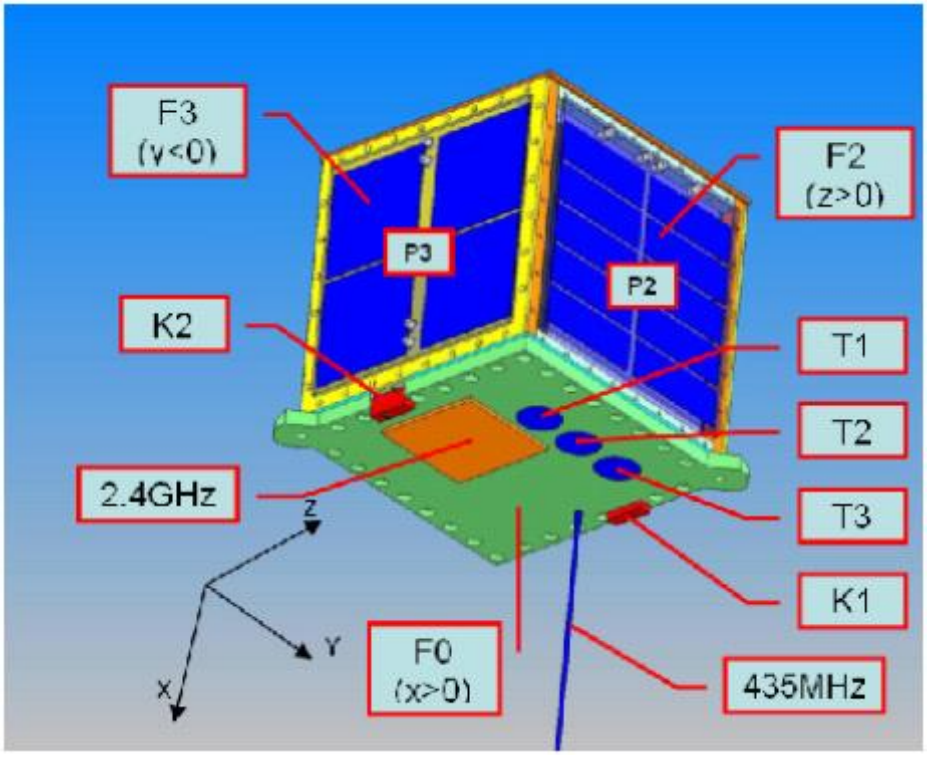


Figura 1.3 PICPOT-vista esterna

La parola chiave per capire l'idea alla base della struttura del satellite è ridondanza:
si è cercato infatti, dove possibile, di duplicare ogni sottosistema per evitare che il guasto di una singola parte compromettesse il funzionamento dell'intero satellite.
Le celle solari, ad esempio, sono di due tipi differenti, come si vede nella figura 1.3, proprio per motivi di ridondanza.
La comunicazione, inoltre, avviene su due canali indipendenti e mutualmente esclusivi:
quella assosciata alla frequenza di 435 MHz è gestita esclusivamente dal processore ProcA mentre quella associata alla frequenza di 2.4 GHz è gestitita esclusivamente dal ProcB. I due processori sono indipendenti e non comunicano tra loro.
Le antenne sono un'antenna patch per la frequenza intorno ai 2.4 GHz progettata apposta per il satellite ed un'antenna a dipolo per la frequenza intorno ai 435 MHz, di tipo commerciale.
I kill-switch sono montati su richiesta del lanciatore e garantiscono l'isolamento elettrico del satellite al momento del lancio.
L'oggetto di questa tesi è il progetto della scheda Payload che ha il compito di gestire le tre telecamere e, in particolare, di scattare delle fotografie in seguito ad una richiesta da parte della stazione di Terra. L'uso delle fotocamere non è funzionale ad un rilevamento di tipo scientifico, ma ha invece due obiettivi, uno di carattere didattico-educativo e l'altro di verifica del funzionamento di componenti COTS nello spazio, come il progetto dell'intero satellite richiede.
La parte elettronica del satellite è formata da una scheda PowerSwitch, da una scheda PowerSupply, da una scheda Tx/Rx e dai tre processori di bordo, ProcA, ProcB e Payload.
La scheda PowerSupply ha il compito di mantenere cariche le batterie ricaricabili del satellite e di monitorare lo stato elettrico e termico dei pannelli, dei caricabatteria e delle batterie stesse.
La funzione principale della scheda PowerSwitch è quella di alimentare le altre schede generando le tensioni necessarie a partire dalle tensioni delle batterie.
La scheda Tx/Rx ha la funzione di trasmissione e ricezione di segnali tra il satellite PICPOT e la stazione di Terra ideata a tale proposito, alle frequenze di 435 MHz e 2.4 GHz nella banda dedicata alle comunicazioni satellitari amatoriali.
I due processori, ProcA e ProcB, progettati ciascuno da un diverso studente, svolgono funzioni simili alle due diverse frequenze, ma sono stati duplicati per motivi di ridondanza, come spiegato in precedenza.
La figura 1.4 è una fotografia della parte interna di PICPOT che evidenzia la disposizione delle schede elettroniche e delle tre fotocamere.

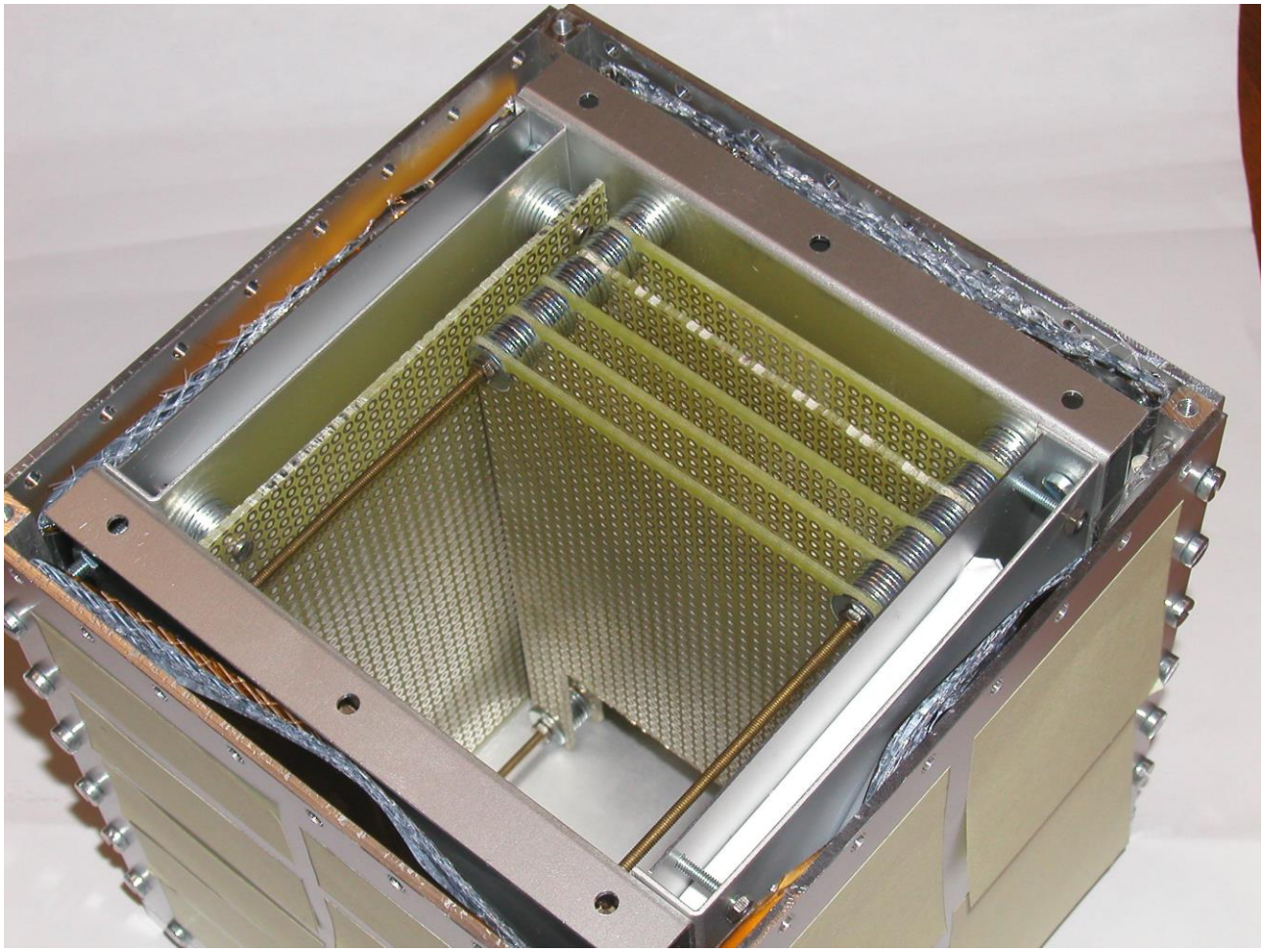


Figura 1.4 PICPOT-vista interna

1.2 Vincoli ambiente spaziale

Uno degli aspetti più importanti tenuti in considerazione nel progetto di tutto il sistema PICPOT è stato l'ambiente dove esso sarebbe dovuto funzionare: lo spazio.

Lo spazio libero presenta alcuni importanti aspetti legati al funzionamento dei circuiti integrati, quali i raggi cosmici. I Raggi Cosmici sono delle particelle prodotte dalle reazioni termonucleari che possono avvenire in alcuni fenomeni celesti, come l'esplosione di supernovae, i fenomeni di collassi stellari o direttamente all'interno delle stelle stesse. Essi sono particelle subatomiche e fotoni ad alta energia, che bombardano costantemente la Terra da ogni direzione. Le energie di queste particelle ricoprono un vasto intervallo fino ad arrivare oltre 1020eV. Tali raggi sono quindi, particelle provenienti dallo spazio, che, interagendo con gli atomi presenti nell'atmosfera, producono un alto numero di particelle che arrivano fino a terra.

Per sistemi orbitanti attorno alla terra, come i satelliti per le telecomunicazioni, la prima sorgente di radiazione sono gli elettroni ed i protoni intrappolati dalla magnetosfera terrestre e sono di interesse

rilevante ad altitudini comprese fra 1000Km e 32000Km. La loro estensione è in realtà maggiore, ma il livello del flusso di particelle diminuisce rapidamente al di fuori di questo range di altitudine.
Le distribuzioni in altitudine per protoni ed elettroni, sono significativamente diverse fra loro, ma entrambe sono accomunate dal fatto che sia per gli elettroni che per i protoni, le particelle più energetiche si trovano ad altitudini maggiori.
I dispositivi elettronici impiegati nella fabbricazione di satelliti ed apparati spaziali di volo, devono normalmente rispondere ai requisiti normativi definiti ed emessi dalle agenzie spaziali NASA ed ESA. Per quanto non previsto in questi requisiti, ci si riferisce di regola alle specifiche MIL applicabili al settore avionico-spaziale. Questo perché i dispositivi elettronici al silicio sono particolarmente sensibili alle radiazioni cosmiche e possono cessare di funzionare se queste superano particolari livelli energetici, anche in dipendenza dalla tecnologia di fabbricazione utilizzata. Anche questi aspetti di resistenza alle radiazioni cosmiche sono regolamentati dalle normative sopra citate; queste prevedono per le differenti applicazioni l'uso di componenti definiti nel linguaggio tecnico "rad-hard" o "radtollerant".
I componenti rad-hard sono normalmente in grado di funzionare anche se esposti a radiazioni di livello fino a 100krad, ed in alcuni casi fino a 300krad; per quelli rad-tollerant i livelli si riducono rispettivamente a 10krad e 30krad. Dispositivi certificati resistenti alle radiazioni, per come sopra esposto, in accordo con la normativa hanno un costo fino a 1000 volte quello dell'equivalente dispositivo qualificato per uso industriale. La reperibilità sul mercato di materiali con queste caratteristiche richiede tempi di consegna che vanno tipicamente da 6 a 24 mesi.
Quanto esposto è in fortissimo contrasto con lo sviluppo del mercato spaziale, che richiede già oggi la fabbricazione di apparati e satelliti in soli 6 mesi dall'emissione di un ordine (mercato spaziale commerciale) ed a costi sempre più bassi.
Il progetto PICPOT che si basa sull'utilizzo di componenti commerciali COTS (Components Off The Shelf) rappresenta una sfida alle considerazioni precedenti. Infatti, sebbene il satellite orbiterà in un'orbita bassa, circa 600 Km, sicuramente potrà risentire di queste radiazioni, per questo numerose scelte di progetto sono state fatte in funzione di questo.
Tra le principali conseguenze dell'interazione delle radiazioni cosmiche con i dispositivi a semiconduttore vi sono gli effetti di danneggiamento ad evento singolo (SEE). Questo tipo di danneggiamento riguarda l'azione di una singola particella che attraversa il substrato dei circuiti.
Tali effetti devono la loro importanza al notevole aumento della miniaturizzazione dei dispositivi elettronici.

I single event effect sono di diverso tipo e possono essere distinti, in effetti transitori ed effetti permanenti. In alcuni casi gli effetti permanenti possono essere così disastrosi da causare la rottura totale del dispositivo.

Per i diversi effetti ad evento singolo si può fare la seguente classificazione:

- Effetto transitorio: SEU ovvero single event burnout.
- Effetto permanente: SEL, single event latchup; SEB, single event burnout e SEGR, single event gate rupture ed altri effetti ad evento singolo.

1.2.1 Single event upset

Quando si verifica un single event upset, quello che accade è che ioni pesanti depositano una tale quantità di energia su un elemento bistabile, da causarne il cambiamento di stato logico.Tali effetti sono facilmente osservabili in memorie non protette. La figura 1.5, mostra un esempio di SEU su un dispositivo elettronico.

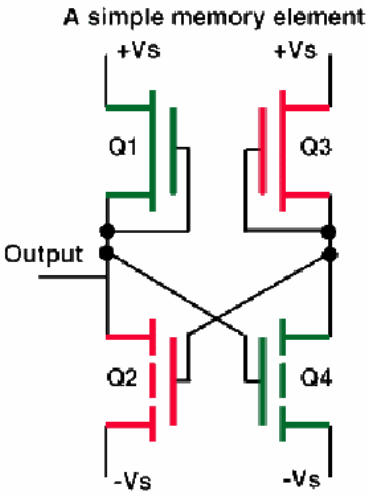


Figura 1.5: SEU su un dispositivo a un bit

Il circuito è disegnato in modo da avere due stati stabili rappresentati reciprocamente con “0” ed “1”. Per ogni stato due transistor vengono commutati on ed off. Un SEU avviene quando una particella energetica che attraversa il dispositivo fa cambiare stato al transistor.

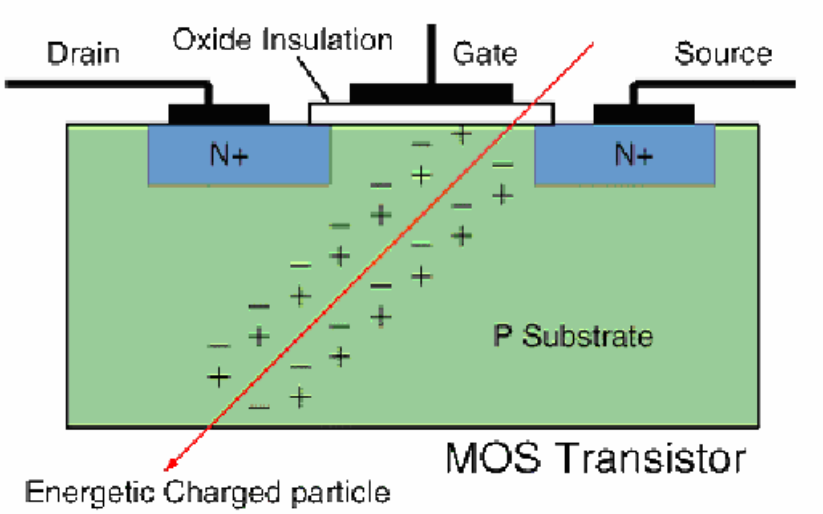


Figura 1.6: generazione di spike di corrente a causa di effetti SEU nei MOSFET

La figura 1.6 mostra, invece, come la particella energetica che penetra nel dispositivo genera un segnale elettrico spurio. La particella produce, infatti, cariche durante il suo percorso sotto forma di elettroni e lacune, che si addensano al source ed al drain e generano un impulso di corrente.

1.2.2 Single event latchup

Un single event latchup è un percorso anomalo di corrente, prodotto fra strutture n-p-n o p-n-p, che sparisce soltanto nel momento in cui viene rimossa l'alimentazione dal dispositivo. Tale problema affligge la maggior parte dei dispositivi microelettronici.

Si ha un latchup quando una particella carica che transita nel dispositivo, induce una corrente tale da aumentare la corrente operativa oltre le specifiche del dispositivo e notevolmente superiore a quella incontrata nelle normali operazioni.

Questo effetto è potenzialmente distruttivo perché la corrente aumenta oltre le specifiche per cui il dispositivo è realizzato, quindi se il dispositivo non viene protetto o limitato in corrente rischia di subire un danneggiamento irreparabile o comunque riduce notevolmente le sue prestazioni peggiorando notevolmente le sue specifiche di funzionamento.

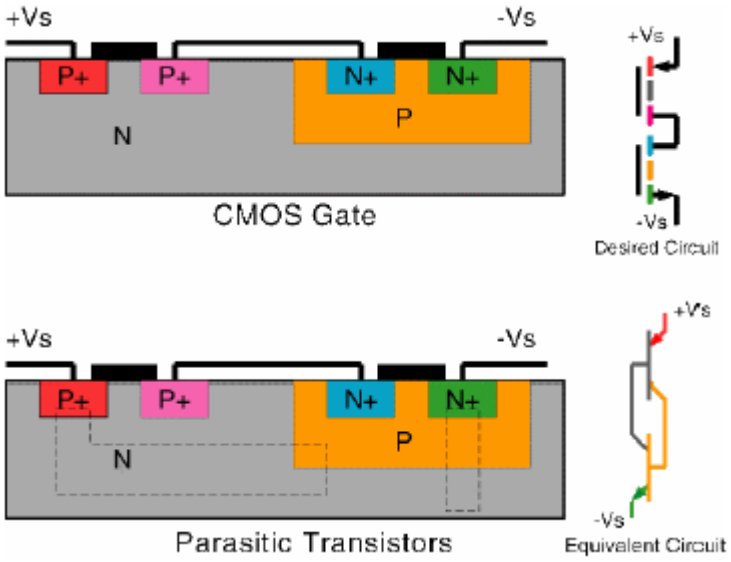


Figura 1.7: schema di un latchup CMOS

Un latchup nei circuiti digitali può avvenire anche quando un impulso spurio di corrente, come quelli prodotti dai raggi cosmici, attiva il transistor parassita presente nella tecnologia CMOS, che nelle normali condizioni operative è fuori uso. Il risultato che si ottiene è che il dispositivo permane nello stato di On. Si parla di transistor parassiti, perché il bulk dei CMOS contiene due transistor bipolari parassiti che formano una struttura a quattro layer.

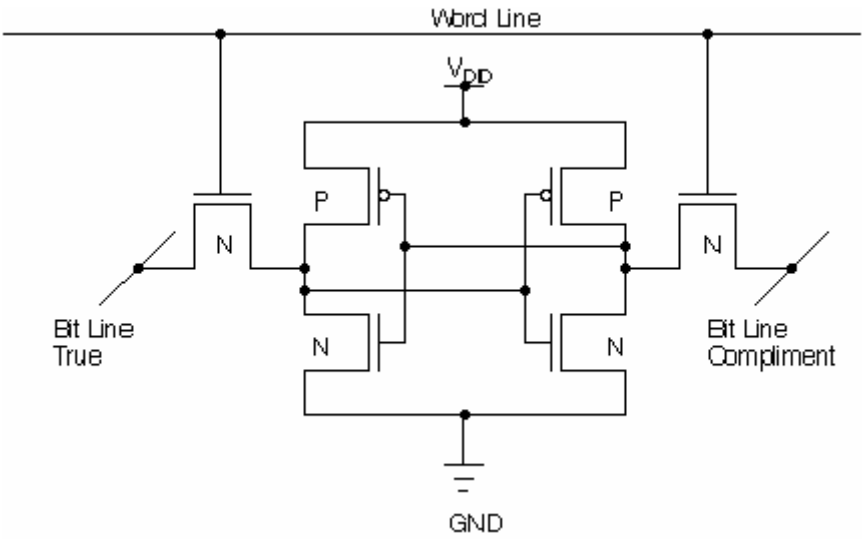


Figura 1.8: schema di una cella CMOS

Le sorgenti di latchup nei CMOS non sono contenute nelle normali operazioni di tali dispositivi ma, segnali transitori posti ai terminali di ingresso o di uscita li possono inavvertitamente triggerare portandoli nella condizione di On.

Quando i transistor parassiti sono attivi, pilotano correnti molto grandi che possono provocare danni anche permanenti. Una volta che il dispositivo è andato in latchup la struttura a quattro layer è

portata in conduzione continua; da questa condizione si esce soltanto se viene rimossa l'alimentazione in un tempo dell'ordine di millisecondi.

Durante il latchup le correnti possono essere molto elevate dell'ordine delle centinaia di milliampere o maggiori, tali correnti circolando nel dispositivo ne aumentano la temperatura. Tali aumenti di temperatura non solo possono danneggiare il circuito, ma possono far si che il latchup si estenda ad altre regioni che prima non ne erano interessate.

Proprio per le sue potenzialità distruttive, il latchup si pone come un grave problema per i sistemi spaziali.

Capitolo 2

Progetto Hardware

2.1 Specifiche scheda Payload

La scheda Payload è una delle schede processore presente su PICPOT, la quale si occupa di gestire le tre videocamere presenti al fine di ottenere delle foto della terra durante il passaggio nella zona in vista del satellite.
Il progetto della scheda Payload è iniziato dalle seguenti specifiche:

- Acquisire un'intero fotogramma da una tra le tre diverse videocamere presenti sul satellite di lunghezza focale diversa e con uscita analogica in formato PAL.
- Codificarlo in formato digitale.
- Campionarlo e comprimerlo in formato JPEG.
- Trasmetterlo ad uno dei due processori presenti sul satellite che gestiscono la trasmissione a terra dell'immagine.
- Permettere il caricamento e l'esecuzione di un programma esterno proveniente da terra.
- Permettere l'acquisizione della temperatura della scheda Payload da parte delle schede ProcA e ProcB

La scheda sarà totalmente comandata mediante diversi telecomandi codificati dai due processori alternativamente, con due canali di comunicazione distinti e dalla scheda PowerSwitch che fornisce le alimentazioni necessarie al funzionamento dei componenti della scheda.
Al fine di soddisfare tale specifiche si è partiti con un progetto di massima della scheda, ipotizzando un processore centrale affiancato da una memoria su cui memorizzare l'immagine da comprimere e quella compressa; inoltre il processore dovrà avere almeno due porte di comunicazione hardware standard distinte, compatibili con quelle presenti sui due processori con cui la scheda dovrà comunicare. Infine essendo tre le videocamere, il processore dovrà avere a disposizione un certo numero di pin general purpose, utilizzabili come segnali di scelta della videocamere e magari alcuni segnali di controllo. Infine dato che le videocamere forniscono un segnale analogico in uscita bisognerà riportarlo in un formato digitale tramite un video decoder e poi acquisire i dati col processore.

Il seguente schema a blocchi descrive lo schema di principio della scheda Payload al fine di soddisfare tali specifiche:

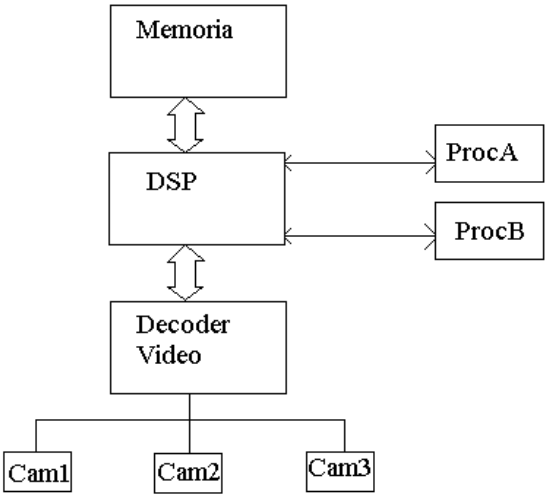


Figura: 2.1 Schema a blocchi specifiche scheda Payload

2.2 Scelta dei componenti

2.2.1 Criteri di scelta dei componenti

Il progetto della scheda Payload si è basato dapprima su alcune considerazioni sull'ambiente in cui avrebbe dovuto funzionare, ossia all'interno di un satellite di dimensioni ridotte quindi di un sistema alimentato da batterie caricate mediante pannelli solari; il consumo energetico della scheda è il fattore principale che è stato tenuto in considerazione nel progetto. Legato ad esso vi è infatti un forte interesse a non far surriscaldare troppo i componenti durante il loro funzionamento, dato che l'unica modalità di dissipazione del calore a 600 Km di altitudine è l'irraggiamento e la conduzione dei piani di alimentazione, quindi di bassa efficienza.

Altro fattore importante per alcune scelte di progetto è stato la possibile interazione dei componenti di tecnologia CMOS, con particelle pesanti presenti nello spazio esterno alla nostra atmosfera, creando la possibilità di effetti di latchup in circuiti CMOS. Tale effetto può condizionare sia i dati memorizzati nelle memorie, sia le correnti erogate da regolatori di tensione, sia i valori di alcuni MOS integrati utilizzati come switch per alimentazioni. Da questo punto di vista inoltre, è stato fondamentale un accurato calcolo dei consumi di corrente massimi da parte della scheda nelle varie fasi di funzionamento, al fine di poter permettere ai progettisti della scheda PowerSwitch di realizzare dei circuiti che staccano l'alimentazione alla scheda in presenza di picchi di assorbimento di corrente, denominati circuiti anti-latchup, adeguati al normale assorbimento della scheda.

Un'ulteriore caratteristica di sistema che ha caratterizzato la scelta dei componenti è che la scheda Payload viene attivata dalla scheda PowerSwitch, la quale fornisce l'alimentazione al sistema dato il comando che le proviene da uno dei due processori che gestiscono tutto il sistema.
Quando la scheda payload deve svolgere qualche operazione essa viene accesa interamente, ma essa non usa mai tutti i circuiti integrati presenti su di essa contemporaneamente, quindi è importante che essi presentino una modalità di Power Save con un bassissimo consumo di potenza, affinché l'integrale nel tempo dell'energia assorbita dal componente quando non è utilizzato sia molto basso, dato che magari la scheda può rimanere accesa quasi un minuto ininterrottamente (da specifiche di sistema). Nella scheda quindi sono state studiate delle soluzioni affinché ogni componente non utilizzato venisse portato in Power Save Mode.
Infine anche il lato economico ha avuto un certo peso nelle scelte effettuate, favorendo la scelta di prodotti disponibili sotto forma di samples gratuiti in quantità adeguate, in modo da permettere l'utilizzo di soli samples per la prototipazione e le schede definitive.

2.2.2 Componenti Video

I componenti video presenti sulla scheda Payload sono le videocamere e il decoder video.
Le videocamere da utilizzare sono state scelte svolgendo dapprima su delle considerazioni sulla tipologia di sensore da utilizzare. Esistono in commercio due tipi di sensori per videocamere:

- Sensori CMOS (Copper Metal Oxide Semiconductor)
- Sensori CCD (Charge Coupled Devices)

I sensori di tipo CCD rispetto ai sensori CMOS sono molto più sensibili, ossia forniscono alla videocamera la capacità di fornire immagini di buona qualità anche a basse condizioni di luce. Questo inoltre è anche associato ad un miglior rapporto segnale/rumore rispetto ai sensori CMOS, i quali a causa delle correnti di leakage presentano una maggiore interferenza di base, che diventa subito importante a bassi valori di illuminazione. Inoltre la risposta di un CCD è indipendente dall'intensità luminosa incidente; infatti, i sensori CCD sono dei rivelatori perfettamente lineari (la linearità è di solito migliore dello 0.01%). In pratica ciò significa che il numero di elettroni generati in un pixel è direttamente proporzionale alla quantità di luce incidente. Tutto questo comporta quindi una maggiore dinamica del sensore CCD.
I vantaggi tipici del sensore CMOS è il prezzo contenuto dato che sono prodotti nelle stesse fonderie da produttori di circuiti integrati, il cui numero prodotto fa quindi scendere notevolmente il costo; inoltre il sensore CMOS presenta un basso assorbimento di potenza rispetto a sensori CCD.

Questi due parametri sono due dei criteri fondamentali delle nostre scelte di progetto, ma il parametro di costo è ininfluente dato che le videocamere sono state cortesemente fornite dalla ITEM s.r.l. ed inoltre rese disponibili in package sotto vuoto per essere messe all'interno del satellite; inoltre sono state anche fornite tre videocamere campione per il debug e i test in laboratorio della scheda Payload.
Infine il parametro di consumo di potenza è relativo in quanto il nostro scopo non è registrare uno stream video di lunga durata, che quindi significherebbe un notevole consumo di energia ma consiste nell'acquisizione di un solo fotogramma (dalla durata relativamente breve di circa 50ms), che quindi sebbene il consumo di potenza istantanea di una videocamera basata su sensore CCD sia importante, l'energia consumata è comunque molto piccola dato che essa viene accesa e poi spenta solo per l'acquisizione della foto.
Quindi si è deciso di utilizzare tre videocamere basate su sensore CCD, di lunghezza focale diversa e quindi angoli di ripresa diversi, denominate MTV–54C5P. Inoltre queste videocamere sono intrinsecamente analogiche e quindi il segnale uscente è in formato analogico televisivo PAL.
Date le caratteristiche delle videocamere adottate, il segnale analogico PAL uscente bisognava che fosse convertito in digitale e codificato in un formato standard, possibilmente implementato in un processore per l'elaborazione numerica dei segnali presente in commercio.
Infatti in commercio sono presenti componenti integrati denominati "Decoder Video", i quali convertono in digitale uno o più canali analogici PAL o NTSC in ingresso e li codificano in un formato standard per il segnale televisivo digitale chiamato ITU-R BT.601-4. Il segnale televisivo digitale così codificato viene poi reso disponibile su 8 bit secondo un formato standard d'interfaccia di segnali video chiamato ITU-R BT.656-4.
Il decoder video è stato selezionato in base ai problemi principali a cui è sottoposto il sistema, presentati in precedenza; inoltre è stato preso in considerazione il numero di canali d'ingresso video analogico disponibili, in quanto la scheda deve gestire tre videocamere. I decoder video standard in commercio presi in considerazione sono stati di due tipi:

- con 6 canali d'ingresso analogico
- con 1 canale d'ingresso analogico

Il primo tipo aveva il pregio di permette l'ingresso dei tre canali in tre ingressi distinti e di poterli selezionare internamente, a fronte dell'altra soluzione che obbligava ad avere le videocamere in parallelo sul segnale video. Quest'ultimo però non risultava un problema perché il datasheet delle video camere garantiva alta impedenza sul segnale video in mancanza di alimentazione; inoltre, il segnale analogico PAL uscente dalle videocamere è di 1V picco-picco (da datasheet) mentre quello

accettato in ingresso dal decoder video del secondo tipo è 0,75V picco-picco. Il segnale dovrà essere partizionato obbligatoriamente, quindi la presenza di altre 2 videocamere in parallelo saranno solo un impedenza in parallelo a quella di partizione, non comportando grandi problemi dato che anche misure effettuate in laboratorio hanno dato un'alta impedenza d'ingresso a dispositivo spento. Inoltre l'alimentazione per le tre videocamere sarebbe stata comunque da selezionare, perché dato che tutte e tre le video camere sarebbero state alimentate dalla stessa linea a 12V proveniente dalla PowerSwitch, solo una doveva essere alimentata per l'acquisizione della foto.

Dato che il decoder video a un canale analogico in ingresso sembrava la miglior soluzione, anche a fronti di consumi ridotti e package più compatti, il decoder video adottato è denominato TVP5150A prodotto dalla Texas Instruments.

Dato che questo componente viene solo utilizzato nella fase di acquisizione dell'immagine dalle videocamere (dalla durata relativamente breve di circa 50ms), ma non nella fase di compressione JPEG e nella fase di trasmissione dell'immagine definiva ai processori, sebbene esso rimanga acceso, quindi deve consumare il minimo di energia possibile. Infatti un'altra caratteristica fondamentale di questo componente è il bassissimo consumo di corrente in modalità Power Save, con un valore massimo stimato di 1μA. Nelle fasi in cui esso non viene utilizzato dal processore verrà messo in modalità di risparmio energetico tramite una porta logica OR sui tre segnali pilota delle alimentazioni delle videocamere (vedi seguito).

Infine questo decoder fornisce dopo il power-up un'interrupt quando avviene la sincronizzazione col segnale PAL, così che il processore dopo averlo ricevuto può iniziare a scaricare il fotogramma senza attendere ulteriore tempo e consumare energia invano (anche perché la potenza assorbita dalla scheda in questa fase risulta elevata data la presenza una videocamera accesa).

Al fine di poter selezionare l'alimentazione in modo distinto per le tre videocamere, la soluzione in linea di principio era quella di avere un mosfet in configurazione di pass transistor. Il problema di utilizzare un solo transistor è che il pilotaggio rimane in configurazione *high-side*, ossia il potenziale di riferimento del segnale pilota e quello del transistor sono diversi, uno il round e per l'altro il 12V. Quindi dato che i pin di I/O di un processore vanno da 0V a 3.3V, non era possibile pilotare un solo transistor. Quindi si è pensato di adottare una configurazione con un n-mos che pilota un p-mos come pass transistor, e pilotare con un pin del processore l'n-mos in configurazione *low-side*. Il sistema funziona se si pone una resistenza di pull-up tra il gate e il source del pass transistor che lo tiene chiuso se l'n-mos è aperto, quindi se il segnale pilota è a "0".

Tale circuito così realizzato lo si è trovato sotto forma di componente integrato in package estremamente piccolo prodotto dalla Fairchild, denominato FDC6324L. La stessa azienda ne

fornisce campioni per la prova del componente in quantità piuttosto cospicue, così tutte le schede realizzate monteranno samples della casa produttrice stessa.

2.2.3 Processore

Come si nota dallo schema a blocchi in figura 2.1, il processore sarà il cuore di tutte le funzioni svolte dalla scheda Payload. Quindi dovrà integrare varie funzionalità sia di tipo standard, quali almeno due porte di comunicazione hardware standard e almeno un controller per memorie non volatili, in quanto dovrà essere memorizzato il programma per il funzionamento della scheda, sia più specifiche per applicazioni video, come una porta per l'acquisizione di stream video nel formato del decoder video selezionato, l' ITU-R BT.656-4. Inoltre come per ogni componente, il processore doveva essere comunque contenuto in dimensioni e nei consumi ma doveva essere adeguato alla funzionalità di calcolo principale per cui sarebbe stato usato: la compressione JPEG. Il compressore JPEG come si vedrà in seguito, ha alla base un calcolo di una trasformata coseno discreta, la quale prevede come operazione fondamentale la seguente:

$$X[k] = \alpha[k] \sum_{n=0}^{N-1} x[n] \cos\left(\frac{\pi (2n+1) k}{2N}\right) \qquad k = 0, 1, \ldots, N-1$$

Quindi moltiplicazioni e addizioni diventano le operazioni più frequenti e quindi quelle che devono essere ottimizzate per rendere più veloce possibile il calcolo della trasforamata coseno e la compressione di un intera immagine.

Dato che nei controllori o processori general-purpose, l'operazine di moltiplicazione richiede un elevato costo computazionale, risulterebbe più adatto un processore progettato per l'elaborazione numerica, chiamato DSP (Digital Signal Processor), dove la sua unità di calcolo ALU (Arithmetic Logic Unit) contiene una unità funzionale chiamata MAC (Multiply Accumulator), predisposta ad eseguire moltiplicazioni e addizioni in contemporanea, abbassando notevolmente il costo computazionale di una trasformata coseno discreta e quindi della compressione JPEG di un'intera immagine.

Dopo una attenta rassegna dei vari processori per sistemi embedded per applicazioni video presenti sul mercato, il processore adottato è denominato Blackfin e prodotto da Analog Devices. Questo recentissimo prodotto è stato scelto appunto perché soddisfava le esigenze della scheda Payload appena esposte.

Infatti il Blackfin utilizzato, nome in codice BF532, presenta una porta video ITU-R BT.656-4 a 8 bit, due controller per memorie esterne, quindi risultava possibile collegare una memoria non volatile su cui poter salvare il programma. Inoltre queste due funzionalità erano potenziate dalla presenza di un DMA (Direct Memory Access) controller, che permetteva il salvataggio dei dati video acquisiti dalla porta ITU-R BT.656-4 direttamente in real-time durante l'acquisizione su una memoria esterna mediante i controller interni.
Inoltre il Blackfin presenta due porte standard quali SPI (Serial Peripheral Interface) e UART (Universal Asynchronous Receiver Transmitter) oltre a due porte seriali generiche veloci bidirezionali che quindi garantivano la possibilità di avere due porte hardware distinte per avere due canali di comunicazione separati con i due processori sempre per l'obiettivo di avere ridondanza del sistema, nel caso uno dei due andasse perso, non influenzerebbe la comunicazione con l'altro.
Infine il Blackfin risulta essere un Digital Signal Processor (DSP), quindi adatto come spiegato in precedenza alle funzioni di compressione JPEG a cui è anche destinato mediante le sue potenti unità di calcolo che verranno illustrate in seguito.

2.2.4 Memorie

Dalle specifiche del sistema Payload, si può notare come il ruolo delle memorie sia importante per le diverse funzioni che devono essere adempite. Inoltre esse rivelano come a seconda delle diverse fasi, possa essere più funzionale ed efficiente un certo tipo di memoria rispetto ad un altro.
La scelta di adottare il DSP Blackfin di Analog Devices, ha permesso un'ampia libertà di scelta sulle memorie, presentando due controller distinti, uno per memorie sincrone, quali SDRAM tipo PC100 e PC133, e uno per memorie asincrone, quali SRAM e FLASH.
La funzione principale delle memorie che devono essere adottate è quella di contenere il codice sorgente del programma utilizzato dal DSP in maniera sicura rispetto ai problemi legati all'ambiente spaziale, spiegati nel capitolo 1. Si è cercato quindi una memoria adatta a svolgere questa funzione, quindi non volatile, riprogrammabile on board e a basso voltaggio, e compatibile con la tipologia di memorie supportate dal processore.
Si è quindi scelto di utilizzare per svolgere tali funzioni una memoria di tipo FLASH.
Le memorie FLASH sono basate su due architetture:

- a porte NOR
- a porte NAND

A seconda dell'architettura utilizzata, queste memorie presentano diverse caratteristiche e differenti comportamenti in ambienti spaziali. Qui di seguito vengono riportate le principali caratteristiche che differenziano le due tipologie:

FLASH Characteristics	
NOR	**NAND**
Low Density	Higher Density
Higher Cost/Bit	Lower Cost/Bit
Faster Random Access	Faster Sequential Access
Not Scalable	Scalable
Supplier Differences	Single Standard

Figura 2.2: caratteristiche memorie FLASH

Le memorie OR sono le memorie FLASH più antiche e quelle più utilizzate per memorizzare sistemi operativi o sorgenti di BIOS. Questo data anche la loro bassa densità rispetto alle memorie FLASH a gate NAND che grazie alla loro densità riescono a raggiungere dimensioni enormi. Questo però è il loro principale svantaggio rispetto al tipo a gate NOR di fronte al problema delle radiazioni cosmiche perchè le FLASH NAND usano tipicamente 8 o 16 celle impacchettate in serie con una linea bit in comune. Questo rende più densa la memoria ma causa una maggiore sensibilità al danneggio a causa delle radiazioni, in quanto il leakage indotto dalle cariche ionizzate incidenti il dispositivo si sommano insieme.

In fase di progetto si cercato quindi di capire quale tipologia di memoria potesse soddisfare le specifiche di progetto e contemporaneamente essere più robusta rispetto ai vincoli ambientali dello spazio, spiegati nel capitolo 1.

Si è utilizzato quindi una ricerca svolta dal California Institute of Technology, sugli effetti delle radiazioni cosmiche sui due tipi di memorie FLASH; infatti utilizzando una memoria FLASH a gate NOR prodotta da Intel e una memoria a gate NAND prodotta da Samsung hanno ottenuto i seguenti risultati:

- irradiando i dispositivi accesi alla tensione di alimentazione (in questo caso 5V) a temperature ambiente, si è riscontrato una maggiore robustezza da parte delle memorie di tipo a gate NOR , come è possibile notare dal grafico seguente:

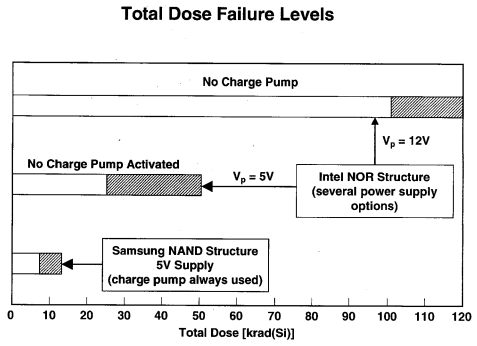


Figura 2.3: risultati dopo il test d'irradiazione sul funzionamento delle memorie FLASH

- eseguendo lo stesso test e mappando l'andamento della corrente di alimentazione, si vede come la memoria FLASH a gate NAND sia molto più sensibile.

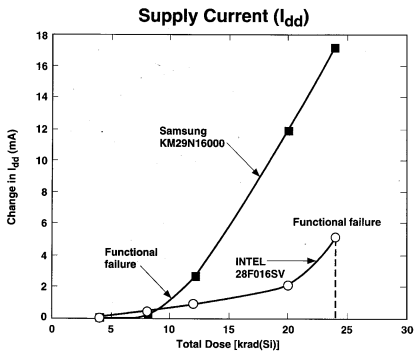


Figura 2.4: andamento della I_{dd} delle FLASH dopo il test d'irradiazione

Le conclusioni della ricerca svolta sono che l'architettura di entrambe le tipologie delle memorie FLASH sono intrinsecamente robuste rispetto i single event upset (SEU); inoltre però, la tipologia a gate NOR risulta essere più robusta alle radiazioni ionizzate presenti nello spazio libero, sia in termini di energia incidente suportata, sia in termini di aumento della corrente di alimentazione. L'unico parametro che può variare in egual misura su entrambe le tipologie a causa dell'interazione con particelle ionizzate è il tempo di cancellazione e scrittura dei blocchi, ma comunque ininfluente per la scelta tra le due tipologie di memoria.

Si è quindi appreso tramite questa documentazione che le memorie FLASH a gate NOR risultavano essere più adatte alle specifiche della scheda Payload, non necessitando inoltre di memorie di elevate dimensioni, quali sono solitamente quelle di tipo NAND. Infatti si era calcolato in fase di progetto che la memoria FLASH dovesse quindi contenere il binario del software utilizzato dal

DSP, che doveva essere di al massimo 200 Kbyte, e al massimo 5 immagini compresse JPEG di dimensioni massime di 100 Kbyte ciascuna, per motivi di tempistiche nella trasmissione a terra dell'immagine da parte delle schede ProcA e ProcB. Inoltre si era previsto anche una possibile copia di backup del software da utilizzare nel caso di corruzione della copia principale e la possibilità di contenere una specie di look up table per la codifica dell'immagine compressa, non ancora ben definita in fase di progetto. Si necessitava quindi di almeno 1Mbyte di spazio, ma pe eesere sicuri si è cercato una memoria di dimensioni maggiori.

Si è quindi andati a cercare tra le memorie presenti in commercio e si è scelto di adottare la memoria FLASH NOR della STMicroelectronics denominata M29W160EB, da 2 Mbyte.

Oltre a soddisfare le nostre specifiche ed essere disponibile presso i fornitori utilizzati, questa memoria è stata adottata in quanto la STMicroelectronics rendeva disponibile un driver completo, scritto in linguaggio C e accompagnato da una descrizione dettagliata contenuta in un application note. Questi sorgenti sono stati utilizzati come base per sviluppare le funzioni necessarie nel nostro software per poter leggere e scrivere sulla memoria Flash utilizzata.

Inoltre la presenza di una memoria sul sistema Payload era richiesta dalla necessità di memorizzare l'immagine acquisita dalla porta video ITU-R BT.656-4. Il DSP Blackfin da la possibilità come spiegato in precedenza, di memorizzare l'intero fotogramma su una memoria esterna in real-time, quindi dato che la frequenza del segnale video del formato standard in uscita è di 27 MHz (si veda il sottocapitolo 2.3.2.1), il DSP acquisisce i dati via DMA controller con delle restrizioni nelle tempistiche della memoria destinazione. Questo quindi rende inutilizzabile la memoria FLASH per tale scopo perché intrinsecamente lenta, come verrà spiegato in seguito, quindi si è dovuto pensare di adottare un altro tipo di memoria.

Di fondamentale importanza è la considerazione che subito dopo l'acquisizione del fotogramma, lo si sarebbe compresso in un immagine JPEG e salvata in una memoria sicura (quale poteva essere la memoria di tipo FLASH, in quanto intrinsecamente insensibili al single event upset, SEU), tutto nello stesso ciclo di funzionamento, dove la scheda Payload sarebbe rimasta accesa per tutto questo periodo quindi anche una memoria volatile sarebbe andata bene.

Si aveva quindi la possibilità di scegliere tra memorie di diverso tipo della FLASH e anche volatili, tipo memorie sincrone SDRAM o memorie asincrone SRAM, entrambe supportate dal DSP adottato e utilizzabili come memoria destinazione esterna di un trasferimento di dati mediante il controller DMA, quindi adatte per l'acquisizione del fotogramma dalla porta video.

Seguendo uno dei principi base del progetto PICPOT, ossia di avere un sistema ridondante ove è possibile di un livello, ossia tollerante ad un guasto o malfunzionamento, si è deciso di adottare entrambe le tipologie di memorie per provare quale delle due tipologie fosse più sensibile

all'interazione con i single event effect, spiegati nel capitolo 1, ed essere coperti nel caso di malfunzionamento di una delle due, dato che entrambe dovranno principalmente svolgere lo stesso compito.

La memoria doveva essere di dimensioni sufficienti almeno da contenere l'intero fotogramma acquisito, di circa 830Kbyte, quindi la memoria doveva essere almeno da 1Mbyte.

Allora per essere sicuri di avere lo spazio necessario per l'acquisizione e il processamento dell'immagine e per utilizzare anche i restanti due banchi di indirizzamento interno del DSP per memorie asincrone, da 1Mbyte ciascuno, si è deciso di adottare come memoria SRAM, una memoria da 2Mbyte prodotta dalla Toshiba, denominata TC55VBM416AFTN55, prodotta in un package da 48 pin TSOP. La scelta della memoria SRAM di questo produttore è stata forzata dalla scarsa possibilità di scelta di questo tipo di memorie nei cataloghi dei più grandi distributori di componenti elettronici. Infatti il progetto prevedeva l'utilizzo di una memoria simile prodotta da Samsung, ma non reperibile in tempi brevi e mediante canali diretti col Politecnico di Torino; quindi si è utilizzato la memoria della Toshiba, in quanto disponibile presso il distributore Digi-Key e con consumi e tempistiche praticamente identici alla memoria prevista in precedenza.

La memoria SDRAM è molto più diffusa e utilizzata nei sistemi embedded, quindi anche più facilmente reperibile presso i distributori. Purtroppo però, solitamente le dimensioni di queste memorie sono molto maggiori di quelle da noi richieste ed inoltre il controller del DSP per memorie esterne sincrone supportava un indirizzamento da 16Mbyte a 128Mbyte. Il problema fondamentale è che una memoria sincrona deve eseguire il refresh di tutte le righe ogni qualche decina di millisecondi e quindi, maggiori sono le dimensioni della memoria, maggiore è il numero di righe, maggiore è la corrente assorbita e quindi maggiori consumi energetici. Quindi bisognava cercare di adottare una memoria di ridotte dimensioni e adatta alle nostre necessità di avere almeno 2Mbyte di indirizzamento del DSP contigui, per poter indirizzare il fotogramma acquisito e altri possibili stream di dati necessari per il processamento dell'immagine.

Si è deciso quindi di utilizzare una memoria di dimensioni di 8Mbyte, la quale appunto garantiva di avere 4 locazioni di memoria contigue da 2Mbyte ciascuna, come verrà spiegato meglio in seguito; essa è prodotta da Micron, denominata MT48LC4M16A2TG, disponibile in un package standard di tipo TSOP a 54 pin.

2.3 Funzionamento componenti principali

2.3.1 DSP Blackfin BF-532

Il DSP Blackfin BF-532 di Analog Devices è un processore ad alte performances e bassi consumi, grazie alla possibilità di arrivare ad una frequenza di clock di 400 MHz e varie modalità di funzionamento per una dinamica gestione della potenza assorbita. Infatti la possibilità di variare sia la tensione del core che la sua frequenza, permette di ottimizzare il consumo di potenza per ogni specifico task da eseguire.

Il processore è alimentato esternamente a 3,3V, ma tale tensione è utilizzata per alimentare tutte le periferiche, mentre l'alimentazione del core è ottenuta mediante un regolatore di tensione interno che comanda un regolatore di tensione switching di tipo buck, montato a componenti discreti esternamente sulla scheda Payload. Il regolatore da una tesnisone d'ingresso da 2,25V a 3,6V, fornisce da 0,8V a 1,2V al core del processore. Lo schema utilizzato è il seguente:

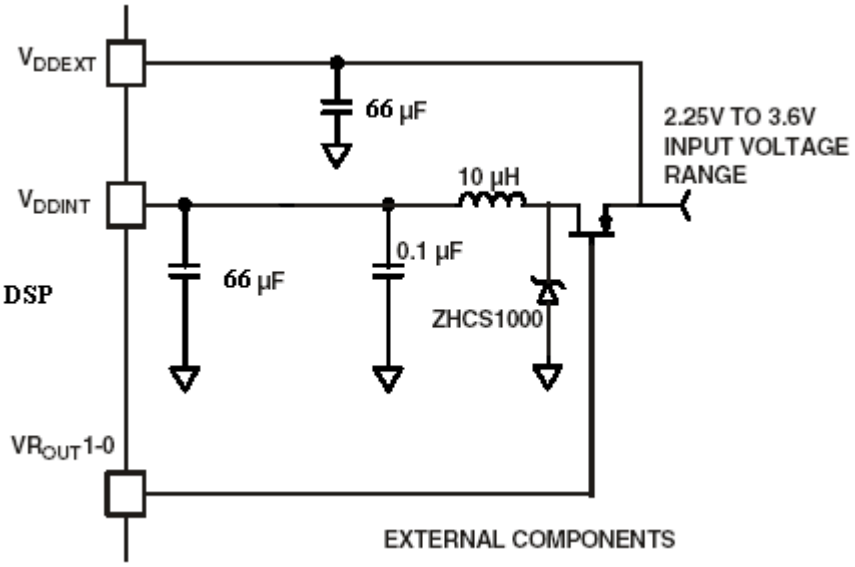


Figura 2.5: Circuito del regolatore di tensione

Il transistor disponibile in forma di circuito integrato su 3 pin in package SOT-23, è prodotto dall'International Rectifier e presenta una tensione di soglia V_{GS} massima di –0,95V, che quindi lo rende pilotabile dal comando a 3,3V dato dai due pin.

Inoltre è stato utilizzato un diodo shottcky ZHCS1000 da 1A, disponibile anch'esso in un package a 3 pin SOT-23 e un induttore da 10µH, anch'esso da 1A e disponibile in package smd chiamato SLF 7mm x 7mm x 3,2mm, prodotto da TDK.

Due pin (cortocircuitati) del processore, chiamati VROUT0 e VROUT1, forniscono il segnale a onda quadra ad una frequenza di circa 300 kHz al gate del P-MOS discreto per ottenere la tensione destinata al core del processore.

Il core del Blackfin BF-532 ha una struttura interna costituita da sei unità funzionali principali per l'esecuzione di algoritmi sia di processamento numerico che di controllo generico dei segnali provenienti dall'esterno; il DSP presenta:

- due unità MAC (Multiply Accmulate) da 16 bit
- due unità aritmetico/logiche (ALU) da 40 bit
- uno shifter di tipo Barrell
- un set di quattro ALU Video

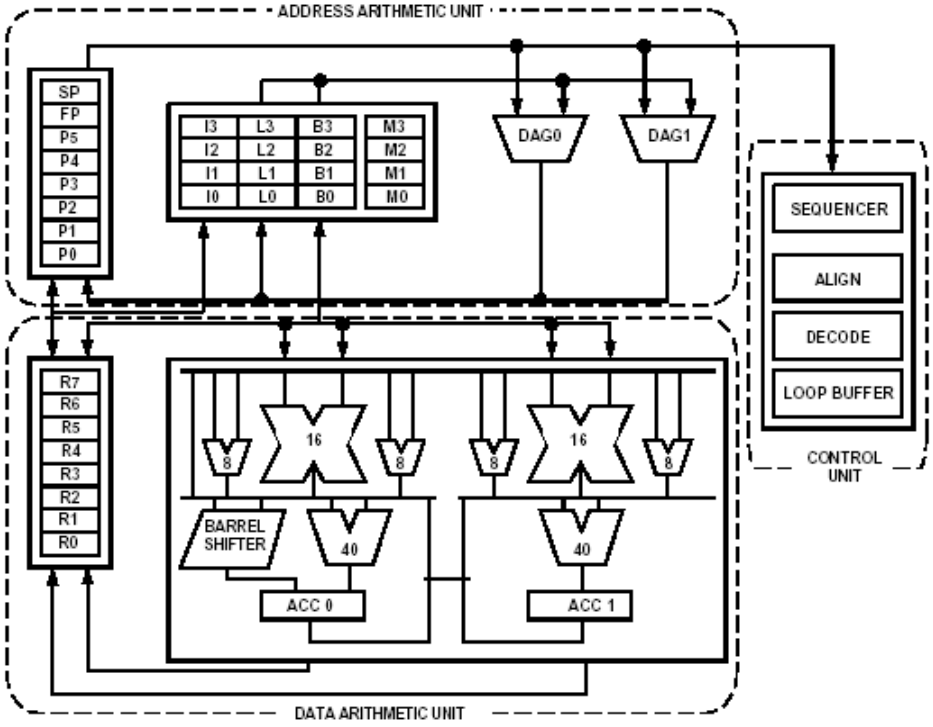


Figura 2.6: Architettura del core del processore

Le due unità MAC vengono utilizzate per operazioni di moltiplicazione o moltiplicazione e accumulo con addizione o sottrazione. Infatti i moltiplicatori operano su dati a virgola fissa a 16 bit e producono quindi risultati a 32 bit i quali possono essere sommati o sottratti dal registro accumulatore a 40 bit.

Il software che poi verrà eseguito per la compressione dell'immagine in formato JPEG utilizzerà pesantemente tali unità funzionali, rendendo la compressione molto più veloce rispetto ad un processore general purpose.

Le due unità ALU, sono ovviamente utilizzate dalle operazioni seguenti, presenti nel codice poi sviluppato, per compiere le varie funzione richieste dalle specifiche:

- somme e sottrazioni di registri in virgola fissa
- accumulo e sottrazione di risultati dei moltiplicatori
- funzioni logiche AND, OR, NOT, XOR
- funzioni: ABS, MAX, MIN

Il barrell shifter è ovviamente utilizzato per le operazioni di shift aritmetico e logico dei bit su registri da 16, 32 e 40 bit, rotazioni e vari test sui bit.

Infine, il set di quattro ALU vengono utilizzate per il processamento dei segnali video con alta efficienza; ogni ALU video prevede come input da una a quattro coppie di registri a 8 bit e come output da uno a quattro registri da 8 bit; tra le funzioni implementate vi sono somma e sottrazioni, medie, packing o unpacking dei dati.

La funzione utilizzata da quelle offerte delle ALU Video dal software della scheda Payload è quella di packing dei dati. Infatti i dati video acquisiti nel formato ITU-R BT.656-4 sono a 8 bit e sono destinati a essere salvati su una memoria esterna SRAM o SDRAM. Entrambe prevedono un parallelismo dei dati, data bus, di 16 bit. Quindi questa funzione permette di accedere alla memoria esterna alla metà della frequenza con cui sono acquisiti i dati, sfruttando pienamente ogni cella di memoria e rendendone meno stringenti le tempistiche. Infatti così il DMA controller può acquisisce due dati a 8 bit dalla porta video a 27 MHz (che corrisponde a un periodo di 37 ns), li “impacca”, ossia li mette insieme in unico dato a 16 bit mediante un’ALU Video e poi li va salvare nella memoria destinazione, ad una frequenza di 13,5Mhz (ossia 74 ns).

Infine il DMA verrà settato in modalità di acquisizione di un singolo fotogramma, chiamato DMA STOP Mode, e qundi alla fine di questo il controller si fermerà generando un interrupt; infatti il processore prevede anche una modalità di acquisizione continua, utile per i video.

L’architettura della memoria interna del processore Blackfin è costituita da tre blocchi, i quali consentono un accesso ad elevata velocità dal core del processore; i tre blocchi sono i seguenti:

- L1 memoria codice, che consiste in una SRAM e/o una 4-way set associative cache. La velocità di accesso di questa memoria è la stessa del core. La memoria è costituita da due blocchi, una solo come SRAM da 32 Kbyte e una utilizzabile sia come SRAM e come cache da 16 Kbyte.
- L1 memoria dati da 32 Kbyte, che consiste in una SRAM e o un 2-way set associative cache. Anche questa memoria è utilizzata alla piena velocità del core.
- L1 scratchpad RAM di 4 Kbyte, la quale permette la stesa velocità di accesso delle altre due ma non può essere utilizzata come memoria cache ma solo com SRAM dati.

Nel nostro sistema il software si prevede che abbia dimensioni massime di circa 200 Kbyte, quindi i 48 Kbyte di memoria codice interna non sono sufficienti (senza utilizzarla come cache), quindi a seconda della modalità di boot predefinità, il codice verrà mantenuto su una memoria esterna tipo SDRAM o FLASH, come verrà spiegato in seguito. La seguente tabella riporta la mappa di memoria intera del processore e spiega in modo migliore la sua struttura, fornendo anche gli indirizzi utilizzati per accedere alle varie aree di memoria:

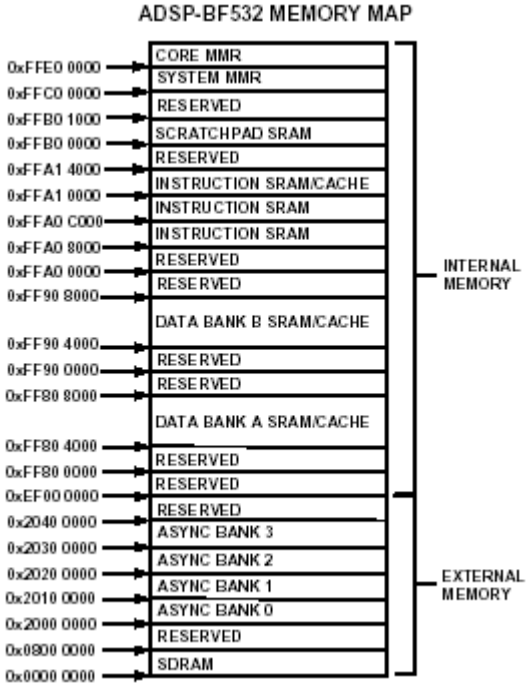


Figura 2.7: Mappa di memoria Blackfin BF-532

La memoria esterna è gestita tramite l'External Bus Interface Unit (EBIU). Esso è suddiviso in due controller, uno per memorie sincrone SDRAM fino al tipo PC133 e uno per memorie asincrone quali FLASH, SRAM, EEPROM, ROM, etc. Quest'ultimo viene chiamato Asynchronous Memory Interface e permette l'accesso di quattro banchi di memoria esterna, denominati ASYNC BANK come si può vedere dalla figura 2.7, ciascuno di dimensioni di 1Mbyte. Quindi in totale può gestire 4Mbyte di memoria asincrona esterna. Come spiegato in precedenza, il sistema Payload prevede l'utilizzo di una memoria FLASH di 2Mbyte per contenere il programma da eseguire all'avvio e le cinque possibili foto scattate dalle videocamere, da trasmettere poi a terra, e l'utilizzo di una memoria SRAM come memoria temporanea per il fotogramma acquisito dalla porta video, in alternativa all'utilizzo della memoria SDRAM. Quindi tutti e 4Mbyte gestibili dall'interfaccia per memorie asincrone è utilizzato. Qui di seguito viene riportata la tabella d'indirizzamento dei quattro banchi, dove i segnali AMS sono gli Asynchronous Memory Select di ogni banco utilizzato e quindi i chip select per ciascuna memoria gestita dal banco:

Memory Bank Select	Address Start	Address End
$\overline{AMS[3]}$	2030 0000	203F FFFF
$\overline{AMS[2]}$	2020 0000	202F FFFF
$\overline{AMS[1]}$	2010 0000	201F FFFF
$\overline{AMS[0]}$	2000 0000	200F FFFF

Figura 2.8: Range indirizzamento Banchi di memoria asincrona

Il controller SDRAM permette di gestire memorie di questo tipo dalle dimensioni di 16 Mbyte a 128 Mbyte massimo, alla velocità massima delle memorie PC133, cioè 133 MHz di clock. Il controller permette inoltre di usare al massimo quattro banchi di memoria esterna distinti, quindi prevede anche una codifica del banco da utilizzare, fatta sui bit d'indirizzo 18 e 19, mentre i primi 15 bit sono disponibili per essere utilizzati come address bus. Inoltre è previsto la possibilità di mascherare gli 8 bit più significativi o quelli meno significativi, tramite due segnali, chiamati SDQM[0] e SDQM[1], mentre se sono posti entrambi a '0', viene letta l'intera word sui 16 bit del data bus.

Le periferiche sono connesse al core medianti molti bus ad alta banda, come si può vedere dalla figura seguente:

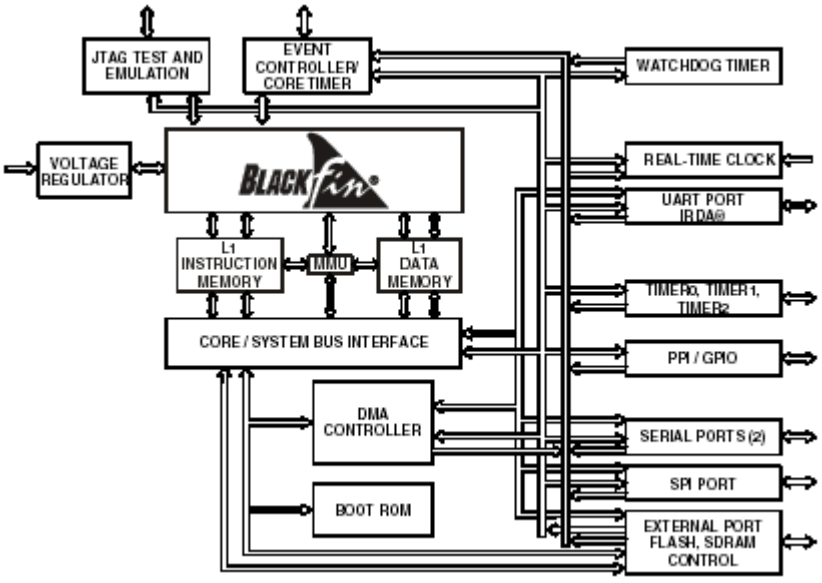


Figura 2.9: schema a blocchi processore

Oltre all'External Bus Interface Unit, il sistema di periferiche include diverse interfacce e unià funzionali:

- Parallel Port Interface (PPI)

- Serial Ports (SPORTs)
- Serial Perpheral Interface (SPI)
- Universal Asynchronous Receiver Transmitter(UART)
- General-purpose timers
- Real time clock timer
- Watchdog timer
- General pur pose I/O

Nel sistema Payload, i diversi tipi di timers non sono utilizzati. Inoltre anche le porte seriali generiche, denominate SPORTS, presenti in due unità, non sono direttamente utilizzate; infatti, dato che i canali di comunicazione con i due processori che gestiscono il sistema PICPOT, ProcA e ProcB, saranno rispettivamente le due porte seriali standard UART e SPI, come richiesto da specifiche, allora le SPORTs sono state collegate in parallelo, una al canale UART e una al SPI, in modo tale che se una delle due porte non avesse funzionato, le sarebbe stato implementato un driver SPI-like o UART-like per poter comunicare con i due processori.
La porta denominata PPI è la porta parallela a 8 bit più un segnale di clock compatibile col formato video digitale ITU-R BT.656-4, che verrà collegata al decoder video per l'acquisizione del fotogramma dalle videocamere.
Infine i general-purpose I/O è costituito da 16 pin, i quali però sono multiplexati con alcuni segnali della porta PPI, dei timer e dei slave select SPI. Nel sistema Payload, di questi 16 pin se ne sono utilizzati 4 per la PPI (PPI4, PPI5, PPI6, PPI7) e uno per il segnale di slave select input della porta SPI, chiamato SPISS (SPI Slave Select). Quindi i restanti 11 pin sono stati utilizzati come pin di controllo sulla scheda Payload.
Questi pin possono essere configurati come sensibili al fronte o al livello logico, la polarità del fronte o del livello che li rende attivi e la loro direzione ossia, se viene utilizzato come pin d'ingresso (valore di default) o come pin d'uscita. Qui di seguito viene riportata la tabella riassuntiva di come sono stati impiegati questi general purpose pin:

DISPOSITIVO		PIN	NOME	FUNZIONE
		33	PF11	SDA
DECODER PAL		34	PF10	SCL
		35	PF9	HSYNC
		36	PF8	RESET
		38	PF6	AVID

		49	PF2	VSYNC
		50	PF1	FID/GCLO
		37	PF7	INTREQ/GPCL/VBLK
SWITCH TELECAM		46	PF5	TELECAM1
		47	PF4	TELECAM2
		48	PF3	TELECAM3

Figura 2.10: descrizione connessioni general purpose I/O pin

La maggior parte sono stati utilizzati per l'interfaccia col decoder video, mentre i restanti tre sono stati impiegati per la gestione dei load switch, spiegati in precedenza per l'alimentazione delle videocamere.

Come è stato accennato all'inizio di qusto paragrafo, la frequenza del core e delle periferiche è ottenuta tramite un quarzo esterno e poi agganciata ad un PLL frazionario, che quindi utilizza un divisore programmabile per ottenere una frequenza massima di 400 MHz. Inoltre i segnali di clock, ossia quelli uscenti dal VCO, destinati alle periferiche (chiamato SCLK) e al core (chiamato CCLK) sono distinti, e sui due sono presenti due divisori distinti, in modo da avere la possibilità di avere le periferiche a bassa frequenza ma il core ad una più elevata; questo comporta un notevole risparmio di potenza, dato che le periferiche sono alimentate a 3,3V. Lo schema è il seguente:

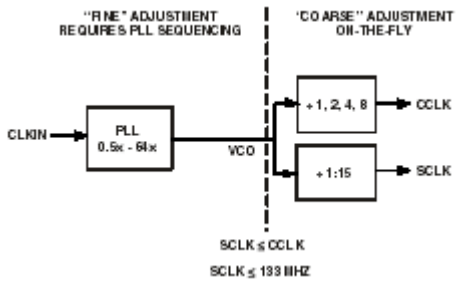


Figura 2.11: catena di gestione del clock

Il quarzo utilizzato per il processore BF-532 è da 11.0592 MHz, prodotto dalla FOX Electronics in package smd denominato HC49SD. Il divisore del PLL è stato settato a 36x, in modo tale da avere un VCO a 398.1312 MHz. Così si è poi settato i due divisori a 4, così da avere le due frequenze sulle periferiche e sul core uguali a 99.5328 MHz. Questo permette di utilizzare la memoria SDRAM adottata in modalità PC100 e di non consumare eccessivamente col core.

Interessante particolarità del processore Blackfin è la modalità di boot; esso prevede tre modalità di boot diverse, due delle quali caricano internamente il programma da una memoria esterna nella memoria codice interna L1, dopo aver ricevuto il comando di reset esterno. Un terzo modo prevede, sempre dopo il reset, l'esecuzione del programma direttamente dalla memoria esterna, saltando la

sequenza di boot. La modalità di boot viene impostata esternamente mediante due pin, chiamati BMODE0 e BMODE1, del processore; un piccolo boot kernel presente all'interno della Boot ROM interna del processore, campiona dopo il reset il valore dei due pin e agisce secondo la seguente tabella:

Boot Source	BMODE[1:0]	Execution Start Address
Bypass boot ROM; execute from 16-bit wide external memory (Async Bank 0)	00	0x2000 0000
Use boot ROM to boot from 8-bit or 16-bit flash	01	0xEF00 0000
Reserved	10	0xEF00 0000
Use boot ROM to configure and load boot code from SPI serial EEPROM (8-, 16-, or 24-bit address range)	11	0xEF00 0000

Figura 2.12: Modalità di boot

Per ogni modalità di boot, un header di 10 byte è dapprima letto dalla memoria esterna; esso specifica il numero di byte trasferiti e l'indirizzo della memoria destinazione; appena tutti byte sono stati caricati, l'esecuzione inizia dall'indirizzo iniziale della memoria codice L1, 0xFFA08000, come si può vedere in figura 2.7.

Nel sistema Payload le modalità di boot utilizzate sono le prime due, codificate con 00 e 01. Infatti il programma è contenuto nella memoria esterna FLASH a 16-bit. In queste due modalità, il processore esegue il boot dall'indirizzo iniziale del banco 0 delle memorie asincrone, ASYNC Bank 0 all'indirizzo 0x2000000. Ecco che quindi la memoria FLASH sarà utilizzata mediante i primi due banchi, banco 0 e banco 1, dell'Asynchronous Memory Interface, mentre qundi i restanti due banchi saranno utilizzati per indirizzare i dati sulla memoria SRAM esterna.

La differenza delle due modalità di boot è che quella codificata 00 viene eseguita direttamente sulla memoria FLASH esterna, mentre nell'altra modalità, codificata 01, il processore punta alla Boot Rom la quale scarica tutto il programma in L1, riempiendola, e la restante parte in SDRAM esterna, dato che il programma sarà di circa 200 Kbyte e la L1 solo di 48 Kbyte in totale. Come verrà realizzato ciò, sarà spiegato in seguito nel capitolo dedicato al Software.

Le modalità di boot che vengono quindi impiegate, prevedono sempre uno dei due bit a zero, BMODE1, il quale è stato quindi collegato a massa.

L'altro pin, BMODE0, può assumere sia '1' che '0' a seconda della modalità di boot selezionata. La selezione avverrà da parte del processore che ha richiesto il funzionamento della scheda Payload, dando quindi la possibilità di pilotare tale pin da parte del ProcA o del ProcB. Tale possibilità è

stata garantita collegando i due segnali digitali provenienti dai processori ad una porta logica AND, collegata in uscita al pin di boot del processore. Il gate è quello sotto forma integrata utilizzato dalle memorie due memorie asincrone, come verrà spiegato in seguito, è prodotto dalla Texas Instruments denominato SN74AGC1G08. Dato che questa soluzione no è stata pienamente testata in fase di prototipazione, sui due segnali provenienti da i processori ProcA e ProcB, è stata posta una resistenza serie da 0Ω, per garantire la possibilità di scollegare il segnale in caso di funzionamento errato, e la possibilità di collegare una resistenza di pull-up da $10k\Omega$ e pull-down da 0Ω. Inoltre queste ultime due sono state previste anche all'uscita del gate AND.

L'assorbimento di corrente da parte del gate AND integrato è di circa 10 μA, comportando un consumo di potenza di circa 33 μW.

Il processore Blackfin esegue il boot dopo aver ricevuto un segnale di reset da un pin esterno di durata minima di 11 cicli di clock esterno; la scheda Payload adotta un quarzo da 11.0592 MHz, corrispondente a 90,42 ns, quindi la durata del segnale di reset deve essere di almeno 994,64 ns, ossia 1 μs.

Nel sistema Payload il boot deve avvenire ad ogni accensione della scheda e quindi si adottato un circuito integrato di power-on reset, esterno al Blackfin, il quale per almeno 140 ms dall'accensione del sistema, tiene a livello logico basso il pin di reset del processore. Lo schema a blocchi del componente integrato è il seguente:

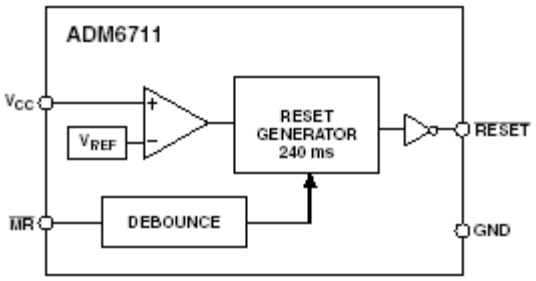


Figura 2.13: Schema a blocchi ADM6711

Come si vede dalla figura precedente, l'ADM6711 prodotto da Analog Devices e disponibile in package di dimensioni molto compatte SC70, prevede anche la possibilità di collegare un pulsante esterno, che mediante un circuito anti-rimbalzo genererebbe un reset asincrono; questo non è stato utilizzato sulla scheda definitiva ma solo sul prototipo, dato che quando la scheda sarà installata all'interno del satellite nessuno potrà comandare il pulsante.

Dallo schema a blocchi si capisce che il componente attende 140 ms da quando l'alimentazione ha sorpassato una certa soglia; per il sistema Payload la soglia adottata è di 2,95V, garantiti costanti fino a circa 85°C, con un buon margine di sicurezza sui 3,3V forniti dalla scheda PowerSwitch.

L'assorbimento in termini di corrente da parte della circuiteria interna è di circa 12 μA, comportando un consumo di potenza di circa 40 μW.

Il consumo di corrente e di potenza del processore Blackfin risulta piuttosto complesso da calcolare; infatti, esso sarà composto dal consumo del core, sommato al consumo di tutti i componenti esterni attivi, ossia in commutazione:

$$P_{TOTAL} = P_{EXT} + (I_{DD} \text{ x } V_{DDINT})$$

l'assorbimento dei componenti esterni infatti, dipende dal numero di pin coinvolti (0), dalla frequenza di commutazione (dei pin coinvolti, quindi la metà del clock utilizzato nei protocolli sincroni quali quello SDRAM, ITU-R BT.656-4 e SPI, in quanto transizioni successive '0'-'1' e '1'-'0' su linee dati o indirizzi avvengono ogni due cicli di clock), il cui valore non può superare quello del clock del processore per le periferiche (SCLK), dalla tensione di funzionamento, tipicamente 3,3V e dalla capacità d'ingresso (C), delle porte CMOS dei componenti integrati:

$$P_{EXT} = O \text{ x } C \text{ x } V^2_{DD} \text{ x } f$$

In seguito verranno presentati il funzionamento dei vari componenti presenti sulla scheda Payload e la loro interfaccia con il processore; per ognuno quindi verrà calcolato il loro consumo interno e dell'interfaccia, in modo da poi riassumere i consumi totali del sistema nelle varie fasi di funzionamento nel sottocapitolo CONSUMI ENRGETICI, sapendo che l'assorbimento di corrente del core del processore funzionando a 100 MHz circ è di 60 mA alla tensione di 1,2V nominali.

2.3.2 Decoder Video TVP5150AP

Il TVP5150AP è un decoder video PAL ultra low power. Disponibile in package molto compatto da 32 pin TQFP, esso converte il segnale PAL ricevuto dalle videocamere nel formato 8 bit ITU-R BT.656-4, in diverse modalità.

L'architettura ottimizata del decoder permette un grande risparmio di potenza assorbita. Infatti il decoder, prevede una modalità di power down, utilizzata dal sistema Payload, quando non vengono utilizzate le videocamere; questo avviene tenendo a livello logico basso il pin chiamato PDN, attivo basso. Inoltre questo componente prevede due tensioni distinte per il core e per le periferiche; infatti queste ultime sono alimentate a 3,3V, mentre il core verrà alimentato a 1,8V; tale tensione sarà ottenuta dalla linea di alimentazione principale a 3,3V mediante un regolatore di tensione low voltage drop out (LDO) esterno, prodotto da Texas Instruments denominato TPS76918, disponibile in package estremamente compatto SOT-23 a 5 pin. Sebbene dalle specifiche del decoder, il consumo di corrente del core dovrebbe essere di circa 55 mA , il regolatore che è stato utilizzato è

in grado di fornire 100 mA, cosi da non essere molto stressato e quindi non scaldare troppo, in quanto come spiegato in precedenza, il fattore di assenza di atmosfera comporta una notevole difficoltà dei componenti nel dissipare il calore generato.

Il decoder utilizza un quarzo esterno ad una frequenza standard tra i componenti integrati per segnali video, 14.31818MHz prodotto da FOX Electronics e disponibile nel package smd HC49SD.

Il decoder contiene diversi registri interni di configurazione sia di caratteristiche video tipo luminosità, contrasto, saturazione, sia di funzionamento delle interfacce. Tali registri di configurazione sono accessibili mediante una porta hardware standard I^2C; per poter configurare quindi il decoder è stato implementato sul processore BF-532 il protocollo I^2C via software, il quale per funzionare utilizza due pin general purpose, si veda figura 2.10, il cui funzionamento verrà spiegato in seguito.

La seguente tabella riporta le varie modalità di funzionamento del decoder:

PDN	RESETB	CONFIGURATION
0	0	Reserved (unknown state)
0	1	Powers down the decoder
1	0	Resets the decoder
1	1	Normal operation

Figura 2.14: Modalità di Reset e Power Down

Dato che il decoder video viene solo utilizzato nella fase di acquisizione del fotogramma dalle videocamere, si è cercato una soluzione che mantenesse sempre in stand-by il dispositivo durante le altre fasi e fosse attivo in contemporanea a quando una delle videocamere fosse stata accesa. Quindi si è posta una porta OR a 3 ingressi discreta, prodotta da Texas Instruments, chiamata SN74LVC1G332, disponibile in un package ultra compatto SC70, low power in quanto la corrente massima assorbita è di 10 μA e quindi 33 μW in potenza. Tale dispositivo prende in ingresso i tre segnali (attivi a livello logico alto) che comandano i tre load switch che danno l'alimentazione di 12V ad una delle tre videocamere da cui deve essere acquisito il fotogramma, e l'uscita va a comandare il pin PDN (Power Down) del decoder (attivo basso).

L'interfaccia principale tra il processore e il decoder video è la seguente:

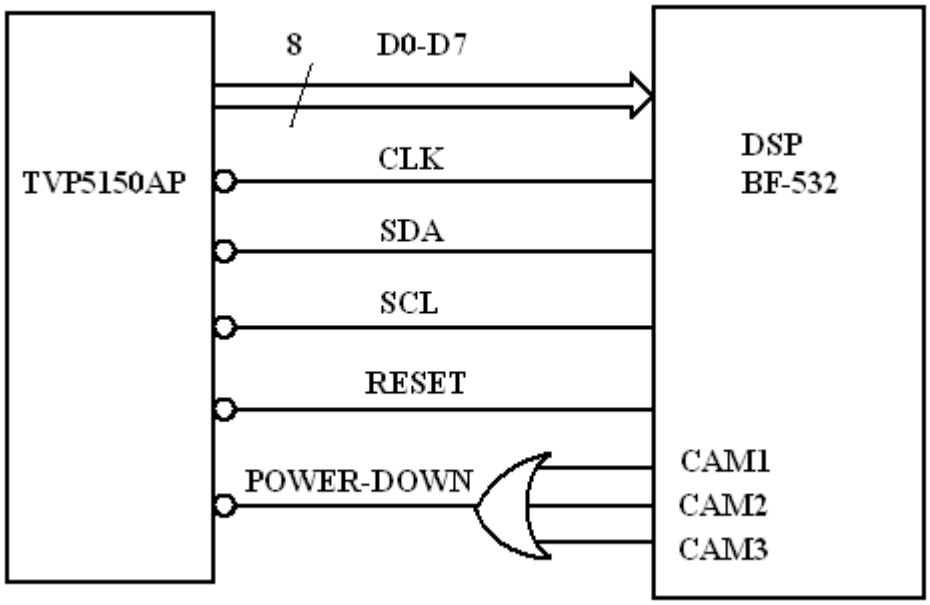


Figura 2.15: Interfaccia PPI, I^2C e controllo tra decoder video e Blackfin

Quindi appena il processore entra nella fase di acquisizione, accende una delle tre videocamere e, contemporaneamente accende il decoder. Inoltre il processore porterà il pin collegato al reset del decoder video al livello '0', così da far passare il componente dalla configurazione di power down mode a quella di reset, come si può vedere dalla figura 2.13. Infine quindi il processore dopo qualche millisecondo, riempito da altre istruzioni indipendenti come si vedrà nel capitolo successivo del progetto Software, porterà il pin di reset a livello alto e porterà il decoder in modalità di normale operatività.

Mediante i registri di configurazione, dopo la routine di accensione descritta in precedenza, vengono settati i due parametri di funzionamento:

- Abilitazione di un'interrupt hardware generato dal decoder su un pin chiamato INTREQ al momento in cui il PLL d'ingresso risulta agganciato al segnale PAL analogico d'ingresso, proveniente da una delle videocamere. Tale pin sarà poi collegato ad un general purpose pin del processore, si veda la figura 2.10, il quale abiliterà immediatamente l'inizio della procedura di acquisizione dell'immagine.
- Abilitazione dei segnali di uscita della porta video, dato che di default si trovano in alta impedenza.

Il decoder video come detto in precedenza permette diverse modalità di acquisizione del formato video. Infatti il segnale video digitale standard, come verrà spiegato in seguito, prevede alcuni segnali di sincronizzazione, relativi alle coordinate orizzontale e verticale del fotogramma acquisito. Tale decoder fornisce uno stream di dati video con incluse queste sincronizzazioni che inoltre

fornisce anche su alcuni pin esterni; essi, come si può vedere dalla figura 2.10, sono stati collegati ad alcuni pin general purpose del processore, per poterne usufruire in caso di malfunzionamenti, in quanto il processore sarà impostato per acquisire solo il segnale video con tutte le sincronizzazioni.
Infine, per un corretto funzionamento del componente, è stato messo a massa il secondo canale video, in quanto non utilizzato, mediante un condensatore da 100 nF.
Il consumo di corrente del decoder è diverso nelle due modalità di funzionamento:

- Normal mode:

Alimentazione	Corrente	Potenza
3.3V I/O	4.8 mA	15.84 mW
1.8V core	55.1 mA	99.18 mW

Nel funzionamento normale quindi l'assorbimento normale dovrebbe essere di 115 mW. Dal datasheet del componente si deve tenere come valore conservativo di potenza massima assorbita 150 mW.
Inoltre a tale valore bisogna aggiungere i consumi dati dalla trasmissione dei dati al processore, che quindi dovranno essere considerati per il tempo effettivo di acquisizione, come verrà spiegato in seguito; presentando una capacità di 8 pF, ad una frequenza di 27 MHz e alla tensione di 3,3V su 8 pin di dati (nell'ipotesi conservativa che ogni bit commuti per ogni dato) più il segnale di clock, la potenza assorbita dall'interfaccia è la seguente:

$$P = 9 * (27MHz)/2 * (3,3V)^2 * 8pF = 10,6\ mW \approx 11\ mW$$

E' stato considerato trascurabile l'assorbimento dovuta alla programmazione in I^2C dato il numero ridotto di pin coinvolti (2), la bassa frequenza di commutazione (massima 400 kHz) e il ridotto utilizzo (scrittura di soli due registri); tale assorbimento può essere inglobato nell'arrotondamento precedente.

- Standby mode: da datasheet 1mW.

2.3.2.1 ITU-R BT.656-4

Lo standard ITU-R BT.656-4, definisce i vari parametri per la connessione e la trasmissione unidirezionale di segnali video tra una singola sorgente e una singola destinazione. Nella scheda

Payload la sorgente del segnale video in suddetto formato digitale sarà il decoder TVP5150AP mentre il destinatario sarà il DSP Blackfin BF532.
I segnali dei dati sono trasmessi in formato binario codificato, in parallelo su 8 bit; i segnali sono di due tipi diversi:

- Segnali video
- Segnali di sincronizzazione

I segnali video sono secondo lo standard ITU-R BT.601-4, o 4:2:2. I parametri di tale standard su segnale analogico PAL sono:

- 625 linee a 50 field/sec interlacciati, di cui 576 linee attive, le restanti sono di sincronizzazione
- Codifica su 3 segnali, uno di luminanza e due di crominanza (detti anche colori differenza), rispettivamente Y, C_B e C_R, ricavati segnali analogici primari E_R, E_G e E_B (RGB).
- Campioni attivi di luminanza Y e di crominanza per ogni linea attiva sono rispettivamente 720 e 360
- Frequenza di campionamento del segnale di luminanza e di crominanza è rispettivamente 13,5 MHz e 6,75 MHz.
- Quantizzazione uniforme PCM a 8 bit
- La corrispondenza tra il segnale video e la quantizzazione dei segnali prevede una scala da 0 a 255 (8 bit), dove il segnale di crominanze prevede 220 livelli di quantizzazione con il nero che corrisponde a 16 e il bianco a 235, mentre il segnale di crominanza, o differenza, vede 235 livelli centrati su 128.

Il formato di un immagine digitale proveniente dal formato PAL avrà la risoluzione quindi di 576 (linee attive) x 720 (campioni per linea); il segnale prevede per ogni linea 720 campioni di luminanza Y, 360 campioni di crominanza C_B e 360 campioni di crominanza C_R, ognuno di dimensioni da 1 byte e quindi arrivando ad avere 1440 byte x 576 linee, ossia 829440 byte. Infatti come si può vedere dalla seguente figura, per ogni due campioni di luminanza Y c'è un campione di crominanza C_B e C_R, quindi alternativamente viene trasmesso un campione di luminanza Y dopo un campione C_B e un altro campione di luminanza Y dopo lo stesso campione C_R, del tipo C_B, Y, C_R, Y, C_B, Y, C_R, etc in 27Mword/sec.

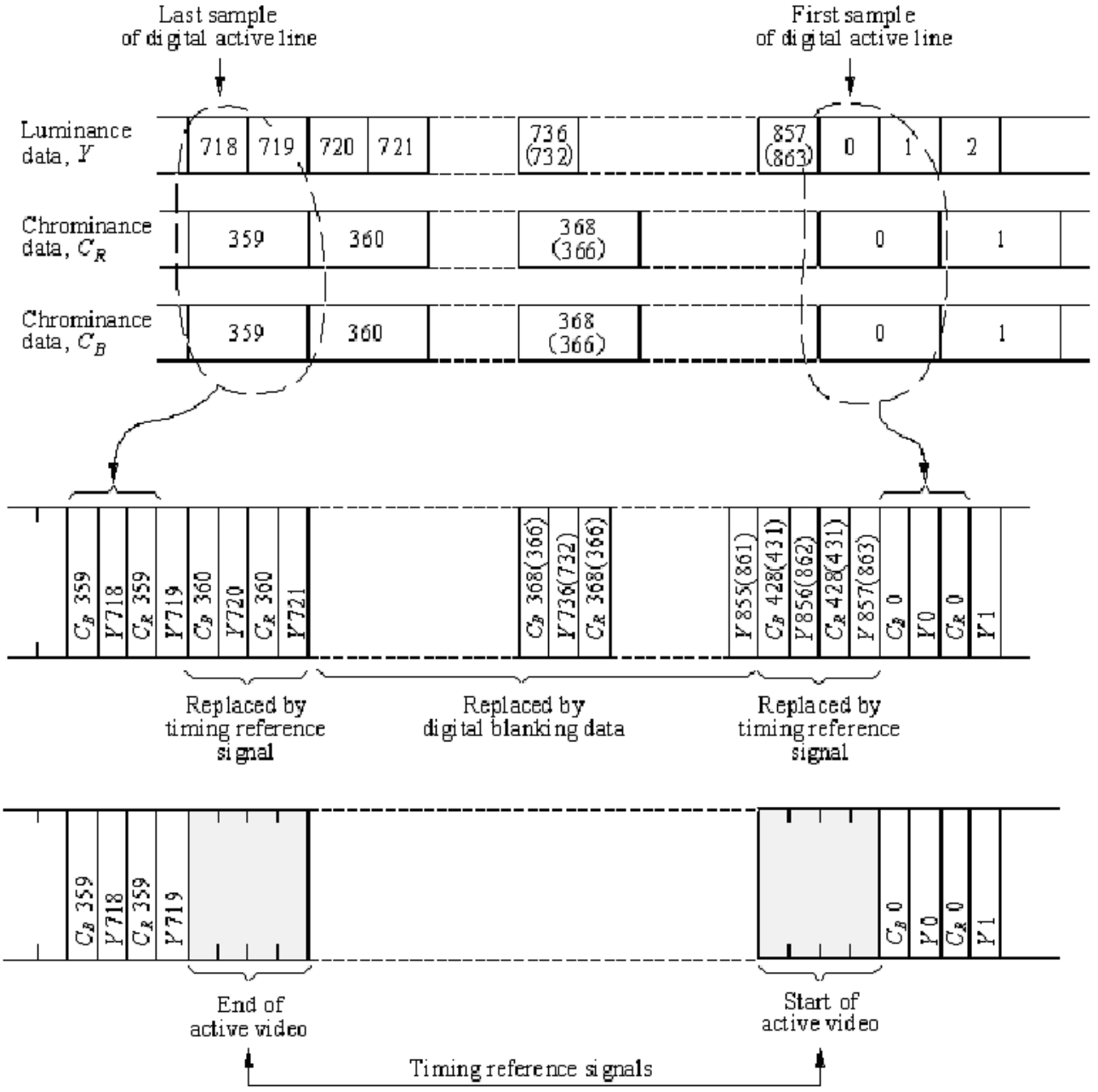


Figura 2.16: Stream di dati di interfaccia

Come si vede dalla figura ci sono due segnali di sincronizzazione denominati SAV, start of active video, e EAV, end of active video. Essi sono composti da 4 byte, i primi tre sono un preambolo fisso, mentre il quarto indica il frame (1 o 2, cioè pari o dispari), se sono dati attivi o blanking e se è un SAV o un EAV, oltre a 4 bit di protezione dagli errori, che fannop corrispondere un codice per ogni possibile combinazione dei 3 bit, quindi 8 possibli parole di codice.

Questa codifica permette una sincronizzazione del segnale video, interna allo stram di dati validi, denominata sincronizzazione embedded. Il decoder TVP5150AP, prevede appunto due modalità di funzionamento distinte, una a sincronizzazione embedded e l'altra a sincronizzazione hardware,

dove tramite la porta parallela 8 bit vengono trasmessi solo gli 829440 byte validi e la loro sincronizzazione viene fornita tramite i pin HSYNC, VSYNC, AVID, FID.
Il DSP Blackfin, prevede la possibilità di acquisire e memorizzare tutti i dati ricevuti oppure di memorizzare solo i dati attivi oppure di memorizzare solo quelli di sincronizzazione. Si vedrà in seguito infatti, che sarà adottata la modalità di acquisizione denominata Active Video Only, la quale ignora tutti i dati tra l'EAV e lo SAV, andando a memorizzare sulla memoria destinazione 829440 byte validi dell'immagine.
Sebbene in questa modalità tutti i dati di sincronizzazione siano scartati essi vengono comunque eseguiti dalla porta sorgente per poter implementare correttamente la trasmissione; i dati trasmessi saranno quindi 864 di luminanza (di cui 720 attivi), 864 dalle crominanze (432 per ognuna di cui 360 attivi) per 625 linee (di cui 576 attive); quindi la quantità totale di dati trasmessi è di 1728 byte x 625 linee che corrisponde a 1080 Kbyte. Alla velocità di 27Mbyte/s (a cui corrisponde un periodo di 37 ns), l'acquisizione di un'intero fotogramma, avrà la durata di 40ms.

2.3.2.2 Programmazione via I^2C

I registri che devono venire settati per avere la configurazione di funzionamento descritta in precedenza sono tre:

- Il Miscellaneus Control Register che è di 8 bit e presenta la seguente struttura:

7	6	5	4	3	2	1	0
VBKO	GPCL pin	GPCL I/O mode select	Lock status (HVLK)	YCbCr output enable (TVPOE)	HSYNC, VSYNC/PALI, AVID, FID/GLCO output enable	Vertical blanking on/off	Clock output enable

Figura 2.17:Miscellaneus Control Register

Questo registro, accessibile all'indirizzo 0x3, è di default pari a 0x01, cioè solo il clock della porta video abilitato, mentre per abilitare le 8 uscite YCbCr, il bit 3 (il quarto), deve essere a uno, quindi il registro assumerà il valore di 0x09.
Come si può vedere, nel caso si volessero utilizzare i segnali di sincronizzazione esterni collegati al processore, bisognerebbe abilitare il bit 2, andando a programmare un valore di 0x0D.

- L'Interrupt Enable Register A a 8 bit la cui struttura è la seguente:

7	6	5	4	3	2	1	0
Reserved	Lock interrupt enable	Cycle complete interrupt enable	Bus error interrupt enable	Reserved	FIFO threshold interrupt enable	Line interrupt enable	Data interrupt enable

Figura 2.18:Interrupt Enable Register

Questo registro, accessibile all'indirizzo 0xC1, è nullo di default, mentre bisogna abilitare il Lock Interrupt sul bit 6, andando a scrivere a quell'indirizzo 0x40. Questo abilità l'interrupt generato quando il PLL è agganciato al segnale PAL d'ingresso.

- L'Interrupt Configuration Register a 8 bit la cui struttura è la seguente:

7	6	5	4	3	2	1	0
Reserved					YCbCr enable (VDPOE)	Interrupt A	Interrupt polarity A

Figura 2.19:Interrupt Configuration Register

Questo registro, accessibile all'indirizzo 0xC2, è pari a 0x04 di default, mentre bisogna abilitare il l'interrupt di tipo A sul bit 1 e settare a '1' anche la polarità del segnale sul bit 0, andando a scrivere a quell'indirizzo 0x07. Questo abilità l'interrupt generato di tipo A e setta il segnale attivo a livello logico alto.

Al fine di poter configurare il decoder come visto in precedenza, il processore DSP deve utilizzare due pin, uno di sincronizzazione chiamato SCL e uno per i dati chiamato SDA, dove implementare il protocollo seriale I^2C.

Questo protocollo, ideato per usi industriali con molti dispositivi collegati, infatti prevede anche qui la possibilità di utilizzare due decoder collegati allo stesso DSP, infatti, il segnale corrispondente al bit più significativo della porta video, chiamato YOUT7, assume la funzione di slave select temporaneamente appena dopo l'accensione e il raggiungimento della modalità di normale operatività; qui, il decoder campiona il suo valore, il quale può essere o '0' o '1' e se lo memorizza e poi lo confronta con il bit 3 della parola mandata dopo lo start bit all'inizio di una nuova scrittura.

Di seguito viene riportata la sequenza di scrittura su un registro:

Step 1	**0**
I^2C Start (master)	S

Step 2	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
I^2C General address (master)	1	0	1	1	1	0	X	0

Step 3	**9**
I^2C Acknowledge (slave)	A

Step 4	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
I^2C Write register address (master)	addr	addr	addr	addr	addr	addr	addr	addr

Se il decoder rivela di essere quello richiesto, manda un acknowledge; il processore quindi ricevuto questo può trasmettere l'indirizzo e poi il valore del registro da configurare.

Step 5	9
I2C Acknowledge (slave)	A

Step 6	7	6	5	4	3	2	1	0
I2C Write data (master)	Data	Data	Data	Data	Data	Data	Data	Data

Step 7†	9
I2C Acknowledge (slave)	A

Step 8	0
I2C Stop (master)	P

Dopo ogni 8 bit ricevuti, il decoder trasmette un'acknowledge ed inoltre la trasmissione si concluderà con la trasmissione di uno stop bit.

2.3.3 Videocamere

Le videocamere MTV–54C5P sono composte da dei sensori CCD da 0.25 pollici (misura della diagonale del sensore), da un numero totale di pixel di 542(orizz) x 586(vert). Essendo l'uscita un segnale PAL, è composto da 625 linee a 50 field/sec.

Il sensore CCD compie la stessa funzione del film posizionato di fronte all'otturatore nelle macchine fotografiche tradizionali, la cui superficie viene esposta alla luce.

La superficie del sensore è formata da pixel identici, disposti con assoluta regolarità lungo le colonne e le righe di una matrice rettangolare. Al momento dell'esposizione alla luce, i fotoni, ossia i quanti di energia radiante proveniente dall'oggetto inquadrato inizieranno a cadere sulla superficie del sensore; così, ciascun pixel ne raccoglierà una quantità proporzionale alla durata dell'esposizione e all'intensità del flusso luminoso incidente. Il CCD trasformerà parte dell'energia E associata ai fotono in elettroni, ossia cariche elettriche che vengono immagazzinate immediatamente nel substrato adiacente alla matrice di pixel.

Dato che una percentuale ben più alta di fotoni (dal 20% al 60% rispetto al 2-3% delle macchine fotografiche tradizionali) sarà catturata e trasformata in elettroni, tali videocamere saranno dotate di un'ottima sensibilità (efficienza quantica). Al termine del processo di esposizione, le cariche accumulatesi negli elementi fotosensibili sono istantaneamente trasferite nei registri verticali per poi essere trasferite riga per riga, nel registro orizzontale di lettura del segnale di uscita del CCD. Mediante tali shift register, il segnale viene poi converito nel formato PAL standard, a due frame interfacciati.

Questa videocamera presenta una sensibilità di 0,5 lux, dovuta all'apertura di diaframma di F=1,2 e un rapporto segnale rumore 60 dB tipici con l'Automatic Gain Control (AGC) spento.
Inoltre la videocamera è fornita di shutter elettronico, consentendo una notevole escursione dell'illuminazione della scena ripresa dalla videocamera, dato che va a ridurre il tempo di sensibilizzazione del sensore da 1/50 di secondo a circa 1/120000 di secondo prima del trasferimento, permettendo di ottenere immagini anche più nitide.
Le videocamere presentano come ingesso la tensione di alimentazione 12V e due segnali di massa, uno per l'alimentazione le l'altro per il segnale video, che però devono essere allo stesso potenziale per funzionare correttamente, oltre al segnale video come uscita.
Esse sono alimentate a 12V e prevedono un consumo (misurato in laboratorio) di 65mA.
I tre segnali video delle videocamere saranno fisicamente saldati insieme e messi quindi in parallelo, grazie all'alta impedenza presentata dalla porta video della videocamera in assenza di alimentazione (da misure in laboratori è di circa 400 kΩ), mentre l'impedenza di uscita a dispositivo acceso è 75 Ω, quindi le videocamere spente in parallelo non dovrebbero dare molto fastidio.
Inoltre dato che le videocamere da datasheet presentano un'uscita analogica di 1V picco-picco e il decoder video adottato accetta in ingresso un segnale analogico di 0,75Vpp, si sarebbe comunque dovuto mettere una partizione del segnale sorgente, oltre ad un condensatore in serie da 100 nF per togliere la componente continua con cui esce il segnale video dalle videocamere. Infatti da progetto si era adottato una partizione resistiva utilizzando due resistenze, una da 39 Ω a massa e una resistenza da 12 Ω in serie, in modo da ottenere una partizione di circa 0,75. Misure però eseguite in laboratorio hanno portato a constatare che l'ampiezza del segnale delle video camere era circa 1,3V picco-picco e quindi nella scheda definitiva si è posto una partizione da 0,5, ponendo due resistenze uguali da 39 Ω, come suggerito dal datasheet del decoder video. La connessione quindi del segnale video PAL e l'ingresso analogico del decoder video è il seguente:

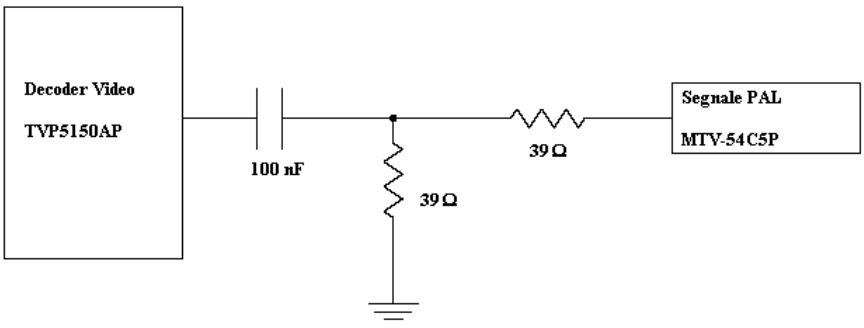


Figura 2.20: connessione segnale PAL

La tensione di alimentazione di 12V raggiunge le tre videocamere mediante tre connessioni distinte. Infatti il processore accende una sola delle telecamere, quella che gli è stata ordinata tramite un telecomando da terra, per acquisire un fotogramma. Le altre telecamere rimarranno spente.
Quindi mediante 3 pin general pur pose, il processore può decidere quando fornire la tensione di alimentazione alle tre videocamere mediante il componente precedentemente presentato FDC6324L. Mediante due resistenze esterne al componente, da 27KΩ e da 10KΩ, il circuito ottenuto è il seguente::

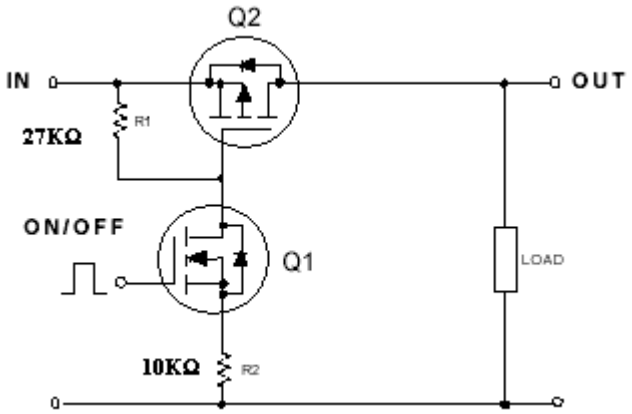


Figura 2.21: circuito Loadswitch con componenti esterni

Mediante tali resistenze esterne si è cercato di posizionare la soglia di commutazione dell'n-mos a circa 1,5V, sui 3,3V possibili dati dal pin del processore, in modo da rendere il più possibile immune da spike e rumore la commutazione del p-mos. Per ottenere ciò, si è utilizzato il modello Pspice reso disponibile dalla Fairchild Semiconductors e, mediante OrCad si è potuto simulare così l'accensione e lo spegnimento di una videocamera con il circuito in diverse configurazioni, andando a scegliere quello presentato nella figura precedente; il suo comportamento è rappresentato nella figura 2.22.
La potenza dissipata da questo componente durante il funzionamento è molto ridotta e risulta essere la corrente che scorre tra il drain e il source del p-mos, cioè 65mA richiesti dalle videocamere, in relazione alla loro resistenza:

$$P_D = (I_D)^2 \text{ x } R_{DSON} = (65\text{mA})^2 \text{ x } 0{,}3\Omega = 1{,}2 \text{ mW}$$

Tale potenza quindi risulta trascurabile rispetto a quella dissipata dalle videocamere per i calcoli energetici del sistema.
Infatti la corrente assorbita dalle videocamere durante il loro funzionamento è di 65mA, misurata anche in laboratorio, che quindi con un'alimentazione di 12V comporta un'assorbimento di potenza di circa 780 mW.

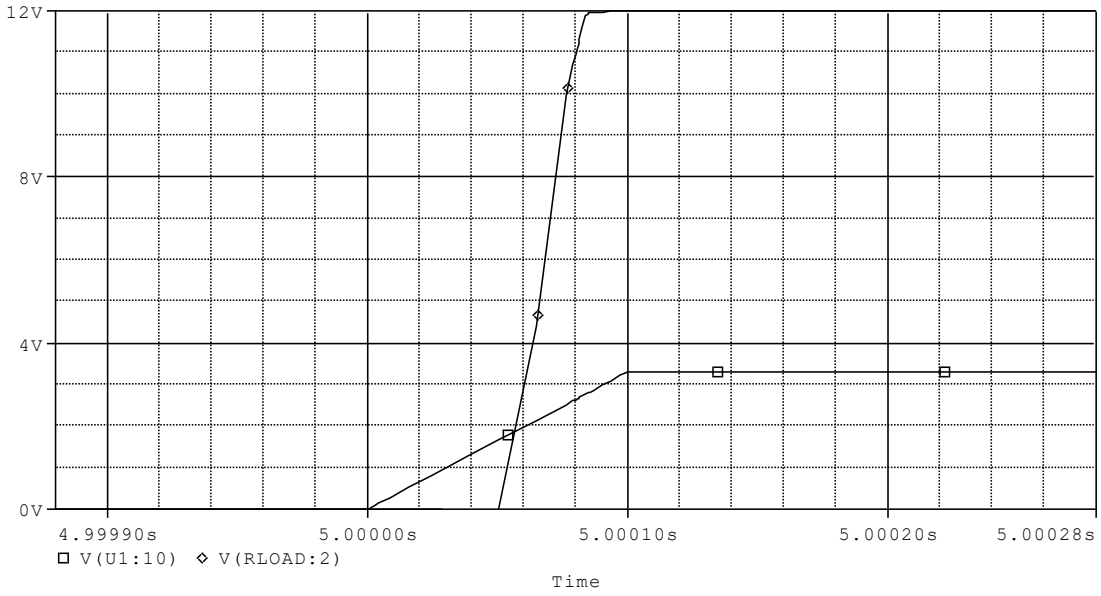


Figura 2.22: Grafico di accensione del LoadSwitch

2.3.4 Memoria FLASH

La memoria flash adottata M29W160EB da 2Mbyte, è una memoria alimentata a 3,3V senza la necessità di una tensione maggiore per la cancellazione, con un data bus a 16 bit e un address bus da 20 bit.

La memoria è suddivisa internamente in 35 blocchi di cui 31 blocchi da 64Kbyte e un blocco di boot. Tale blocco da 64 Kbyte posto all'indirizzo 0x0 (di qui la lettera B del nome della memoria che sta per 'Bottom' Boot Block, per distinguerla dall'altro tipo 'Top' Boot Block che vede tale blocco all'indirizzo 0xFFFFF), è suddiviso da un primo blocco da 16Kbyte, poi du da 8Kbyte e infine uno da 32Kbyte. Sarà quindi da tale blocco di boot che partirà il binario del codice sorgente del software che verrà eseguito dal DSP. La struttura è rappresentata nella figura 2.23.

Essendo una memoria asincrona l'interfaccia con il DSP, prevede anche l'utilizzo di segnali di controllo quali write enable, chip enable e output enable. Mediante tali segnali di controllo vengono distinte fasi di scrittura o di lettura e accensione e spegnimento della memoria. Inoltre presenta un pin, denominato BYTE, che la memoria campiona per saper se viene utilizzato un parallelismo 16 bit o 8 bit; nel sistema Payload tale pin è stato posto alla tensione di alimentazione mediante una resistenza di pull-up da 4,7 KΩ, definendo un bus dati da 16 bit. Inoltre connesso al pin del DSP chiamato ARDY, vi è un pin denominato READY open drain (quindi normalmente posto alla tensione di alimentazione mediante una resistenza di pull-up da 4,6 KΩ) comandato dalla memoria,

il quale viene tenuto basso dalla memoria finchè non ha completato un ciclo di funzionamento, tipo scrittura o cancellazione. Questo può essere ignorato o utilizzato dal Blackfin.

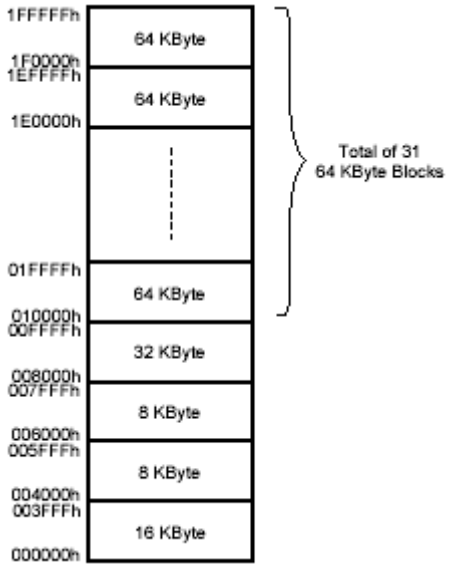


Figura 2.23: Mappa di memoria della M29W160EB

Infine è presente un pin di RESET attivo basso il quale è posto alla tensione di alimentazione mediante una resistenza di pull-up da 4,7 KΩ, quindi la memoria FLASH non viene mai resettata.
La memoria ha un bus indirizzi da 20 bit, così da poter gestire 1Mword da 16 bit, ossia 2 Mbyte, mentre il processore ha solo 19 segnali d'indirizzo. Si è quindi cercato una soluzione per poter utilizzare lo stesso tali memorie, partendo dalla considerazione che i segnali di chip select AMS sono attivi bassi e quando sono inattivi sono a '1' logico.
Si è quindi adottato una porta logica AND integrata in un componente prodotto dalla Texas Instruments denominato SN74AGC1G08, disponibile in un package molto compatto a 5 pin SC70, con un ritardo massimo di 10 ns; in ingresso di tale gate AND si sono posti i due chip select uscenti dal DSP AMS[0] e AMS[1], connettendo poi l'uscita all'ingresso chip enable della memoria FLASH. In questo modo solo quando entrambi i banchi non sono utilizzati, la memoria è spenta cioè chip select a livello logico '1'. Altrimenti il chip select della memoria è attivo, in modo trasparente dal banco di memoria utilizzato.
Inoltre si è collegato l'AMS[0] al pin corrispondente al pin 20 del bus indirizzi della memoria FLASH, così da fungere da ultimo bit per gli indirizzi; infatti con l'AMS[0] attivo, ci si riferisce al primo banco di memoria dell'Asynchronous Memoriy Controller, e quindi agli indirizzi 0x00000-0x7FFFF (1Mbyte) della memoria, e quando disattivo ossia a livello logico '1', ci si riferisce agli indirizzi 0x80000-0xFFFFF utilizzando il secondo banco di memoria.

Si è utilizzato l'AMS[0] come bit di indirizzo, in quanto il DSP al boot punta all'indirizzo 0 gestito da tale banco di memoria, il quale come spiegato in precedenza risulta essere il suo blocco di boot, dove inizia il codice sorgente del programma da eseguire sul DSP. Quindi tale DSP garantiva coerenza nella fase di boot sugli indirizzamenti.

L'interfaccia completa tra il processore e la memoria FLASH è riportata nella figura seguente:

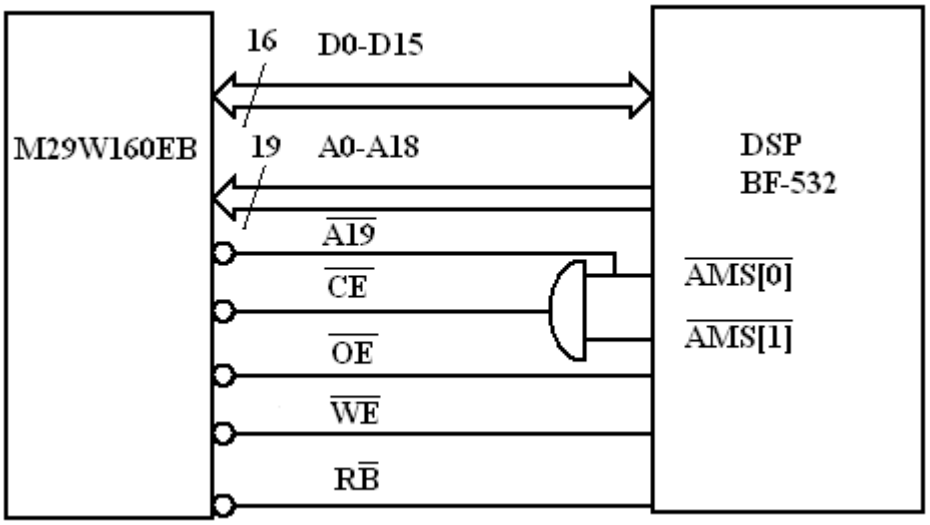


Figura 2.24: Interfaccia BF-532 e Memoria FLASH

Il funzionamento della memoria FLASH è diverso da una memoria ad accesso casuale. Infatti la sua tecnologia, permette la scrittura di soli '0', quindi prima delle scrittura di un dato, la locazione di memoria deve essere cancellata, ossia i suoi bit devono essere tutti posti a '1' e poi può essere scritta. Inoltre però per scrivere su una locazione di memoria non può venire cancellata solo essa, ma deve essere cancellato l'intero blocco a cui appartiene tale locazione.

Infatti per l'utilizzo della memoria sono previsti diversi tipi di comandi d'interfaccia, attivati presso la memoria mediante la trasmissione di codici in sequenze corrette in locazioni determinate, secondo la tabella rappresentata in figura 2.25.

In tale tabella, X significa don't care, mentre PA program address, PD program data e BA qualsiasi indirizzo. Tramite tali sequenze di codici, vengono attivati i vari comandi della memoria e poi quindi possono essere letti, cancellati o scritti dei blocchi di memoria.

In tali condizioni, la scrittura di una parola da 16 bit richiede circa 10μs, dato che il tempo di accesso della memoria, ossia l'operazione di scrittura o lettura di una singola parola sul bus asincrono, è di 70 ns. Per questi motivi la memoria FLASH risulta essere inuttilizzablie per l'acquisizione dell'immagine dalla porta video.

I vari comandi utilizzati dal processore per svolgere le varie attività di funzionamento saranno descritti in seguito nella sezione software.

Command	Length	Bus Write Operations											
		1st		2nd		3rd		4th		5th		6th	
		Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read/Reset	1	X	F0										
	3	555	AA	2AA	55	X	F0						
Auto Select	3	555	AA	2AA	55	555	90						
Program	4	555	AA	2AA	55	555	A0	PA	PD				
Unlock Bypass	3	555	AA	2AA	55	555	20						
Unlock Bypass Program	2	X	A0	PA	PD								
Unlock Bypass Reset	2	X	90	X	00								
Chip Erase	6	555	AA	2AA	55	555	80	555	AA	2AA	55	555	10
Block Erase	6+	555	AA	2AA	55	555	80	555	AA	2AA	55	BA	30
Erase Suspend	1	X	B0										
Erase Resume	1	X	30										
Read CFI Query	1	55	98										

Figura 2.25: Comandi memoria FLASH M29W160EB

I consumi di corrente e di potenza sono differenti nelle varie fasi di utilizzo, come riassume la seguente tabella:

Fase di utilizzo	Corrente	Potenza
Lettura	10 mA	33 mW
Scrittura/Cancellazione	20 mA	66 mW
Standby	100 μA	330 μW

Inoltre bisogna calcolare quanto dissipato in potenza dall'interfaccia tra il processore e la memoria FLASH; per essere conservativi, essa prevede un tempo di accesso di 70 ns e quindi utilizzeremo la frequenza corrispondente di 14,3 MHz.. Infine sappiamo che il bus indirizzi da 20 segnali è sempre visto come scrittura, quindi vede in ingresso una capacità di 6 pF, mentre il bus dati da 16 segnali vede la stessa capacità nelle fasi di lettura mentre nelle fasi di scrittura vede una capacità di 12 pF di uscita.

Ecco quindi il consumo delle due fasi:

- Lettura: $P = 36 * 14{,}3MHz * 6pF * (3{,}3V)^2 = 33{,}6\ mW \approx 34\ mW$
- Scrittura:

$P = (20 * 14{,}3MHz * 6pF * (3{,}3V)^2)+(16 * 14{,}3MHz * 12pF * (3{,}3V)^2) = 48{,}58\ mW \approx 49\ mW$

2.3.5 Memoria SRAM

La memoria SRAM TC55VBM416AFTN55,è appunto una memoria volatile ad accesso casuale di tipo statico, alimentata anch'essa a 3,3V.

L'interfaccia con il processore prevede un parallelismo per i dati di 16 bit e un address bus da 20 bit. Ovviamente per gestire 2 Mbyte, si è adottata la stessa tecnica utilizzata con la memoria FLASH, andando a collegare ad una porta AND (dello stesso tipo di quella presentata precedentemente) i due chip select AMS[2] e AMS[3], e l'uscita di essa al chip select della memoria.

In questo caso però non aveva importanza quale segnale andasse a pilotare il bit più significativo del bus indirizzi, così si è collegato l'AMS[3] comportando la mappatura del banco 4 sul primo Mbyte di memoria SRAM, nell'intervallo 0x00000-0x7FFFF, e la mappatura del banco 3, pilotato dall'AMS[2], sul secondo Mbyte di memoria esterna, nel range 0x80000-0xFFFFF.

L'interfaccia completa tra il processore e la memoria SRAM risulta essere la seguente:

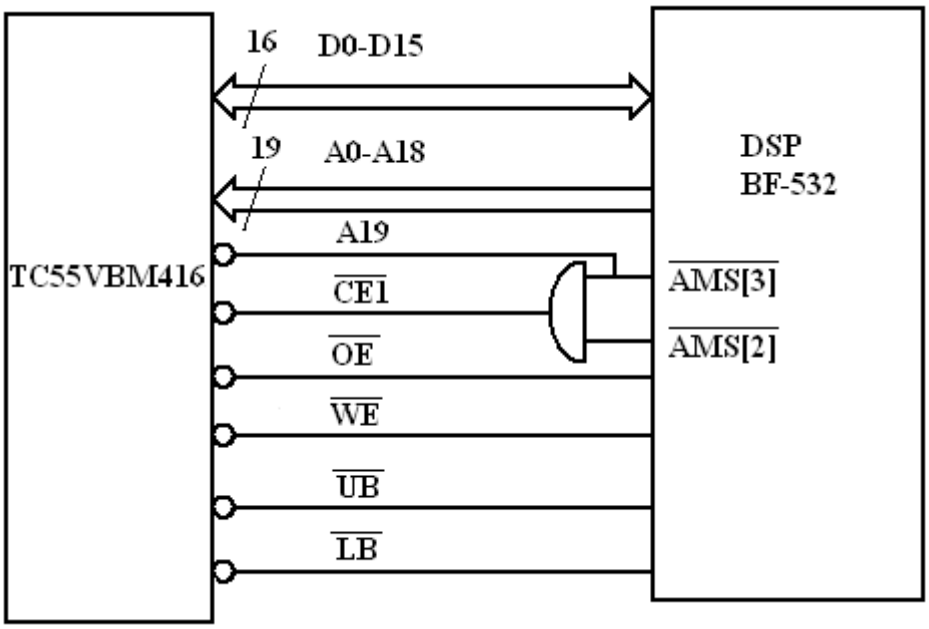


Figura 2.26: Interfaccia BF-532 e Memoria SRAM

I segnali UB e LB sono pilotati da due pin del controller asincrono del processore per definire il parallelismo utilizzato sui dati e, nel caso di 8 bit, se considerare il byte più o meno significativo. Nel nostro caso ovviamente sono entrambe attive, ossia a '0', dato il parallelismo a 16 bit.

Inoltre si nota come il chip enable sia chiamato CE1, in quanto c'è un altro pin chip enable, chiamato CE2. Il chip enable 2, attivo alto, deve rimanere sempre alto in tutte le diverse modalità di funzionamento, tranne che nello standby, in cui deve rimanere basso, ma la nostra memoria FLASH

non deve essere in standby fisso in quanto contiene il codice programma che deve essere sempre a disposizione del processore e per tale motivo è stato messo a livello alto mediante una resistenza di pull-up da 4,7 KΩ.
La memoria SRAM prevede 4 diverse modalità di funzionamento:

- Lettura
- Scrittura
- Uscite in alta impedenza
- Standby

Nel sistema Payload, solo le prime tre saranno impiegate, evitando la modalità standby vero per le motivazioni spiegate in precedenza, ma essendo una memoria asincrona, quando non viene utilizzato il bus e il chip select 1 viene disattivato, ossia portato a livello logico alto, la memoria entra in standby, pronta comunque a un rapido utilizzo.
La memoria va in fase di lettura con il write enable a livello logico alto, mentre va in modalità di scrittura nella stessa configurazione ma con il write enable a '0'. Infine tenendo la stessa configurazione ma mantenendo l'output enable alto, quindi disattivo, si porta la memoria con le uscite in alta impedenza.
Il tempo di accesso della memoria, ossia l'operazione di scrittura o lettura di una singola parola sul bus asincrono, è di 55 ns, andando a soddisfare le tempistiche per l'aquisizione del fotogramma dalla porta video tramite il DMA controller.
Sebbene entrambe siano due memorie asincrone, date le modalità di funzionamento differenti la memoria SRAM consuma la stessa corrente sia in scrittura che lettura, a differenza della memoria FLASH, e ovviamente consuma molto meno corrente in fase di standby:

Fase di utilizzo	Corrente	Potenza
Lettura/Scrittura	35 mA	115.5 mW
Standby	15 μA	49.5 μW

Inoltre bisogna calcolare quanto dissipato in potenza dall'interfaccia tra il processore e la memoria SRAM; per essere conservativi, essa prevede un tempo di accesso di 55 ns e quindi utilizzeremo la frequenza corrispondente di 18,2 MHz.. Infine sappiamo dal datasheet del componente che le capacità in ingresso e in uscita sono uguali e valgono 10 pF. Quindi la potenza assorbita in questa fase dall'interfaccia col processore è la seguente:

$$P = 36 * 18{,}2MHz * 10pF * (3{,}3V)^2 = 71{,}4\ mW \approx 72\ mW$$

2.3.6 Memoria SDRAM

La memoria SDRAM MT48LC8M8A2TG-75, è una memoria sincrona PC100 compliant; per questo prevede una frequenza di clock di 100MHz, quello appunto settato nel clock del processore per le periferiche (SCLK).
Questa memoria di 8Mbyte, con parallelismo del data bus di 16 bit, divisa in 4 banchi da 2 Mbyte, ossia 1Mword. Per la selezione dei banchi di memoria, il processore utilizza due bit del bus indirizzi non utilizzati per l'indirizzamento vero nelle diverse possibili fasi di funzionamento, precisamente il bit 18 e il bit 19. Inoltre ogni banco è suddiviso in 4096 righe e 256 colonne; infatti, il processore per poter accedere ad una qualsiasi locazione di memoria, utilizzerà 12 bit (2^{12} = 4096) del bus indirizzi per mandare il valore della riga da che vuole utilizzare e, nel ciclo successivo vengono utilizzati i primi 8 bit (2^8 = 256) per mandare il valore della colonna puntata. In particolare però, l'undicesimo bit (chiamato A10 perché la numerazione inizia da A0) degli indirizzi non è comandato dall'undicesimo bit dell'address bus, come tutti gli altri segnali d'indirizzo ma è comandato da un pin distinto chiamato SA10 del controller del processore. Questo segnale infatti normalmente funge da indirizzo ma viene anche utilizzato dalla memoria durante il comando di PRECHARGE" per capire se tutti i banchi sono stati prevaricati (SA10 a livello logico alto) o solo quello indicato (SA10 a '0') tramite i due bit appositi.
Inoltre l'interfaccia prevede due segnali DQM, uno chiamato DQML che si riferisce ai primi 8 bit di dato mentre il secondo chiamato DQMH che si riferisce al byte più significativo del bus dati. Essi se campionati a livello logico alto durante una fase di lettura, il bus dati viene messo in alta impedenza, impedendo l'uscita del dato e causando 2 cicli di clock di ritardo, mentre se ciò avviene in fase di scrittura il dato presente sul bus viene ignorato e non memorizzato.
Infine l'interfaccia presenta segnali di controllo di scrittura e di accesso alle righe (SRAS) e alle colonne (SCAS) attivi bassi, che quando attive indicano che l'indirizzo presente sul bus è riferito alla riga o alla colonna. Inoltre è presente un write enable (SWE) per definire la modalità di accesso, se in lettura o scrittura.
Inoltre è presente il segnale di clock per la sincronizzazione dei segnali e il chip select, il quale quando campionato alto rende la memoria insensibile a qualsiasi commutazione su qualsiasi suo pin.

Infine troviamo un segnale fondamentale denominato clock enable (SCKE), il quale non solo abilita o disabilita il funzionamento del segnale di sincronizzazione principale, ma in questo modo porta la memoria in modalità di power down.

L'interfaccia completa del controller sincrono del processore e la memoria SDRAM appena spiegata è la seguente:

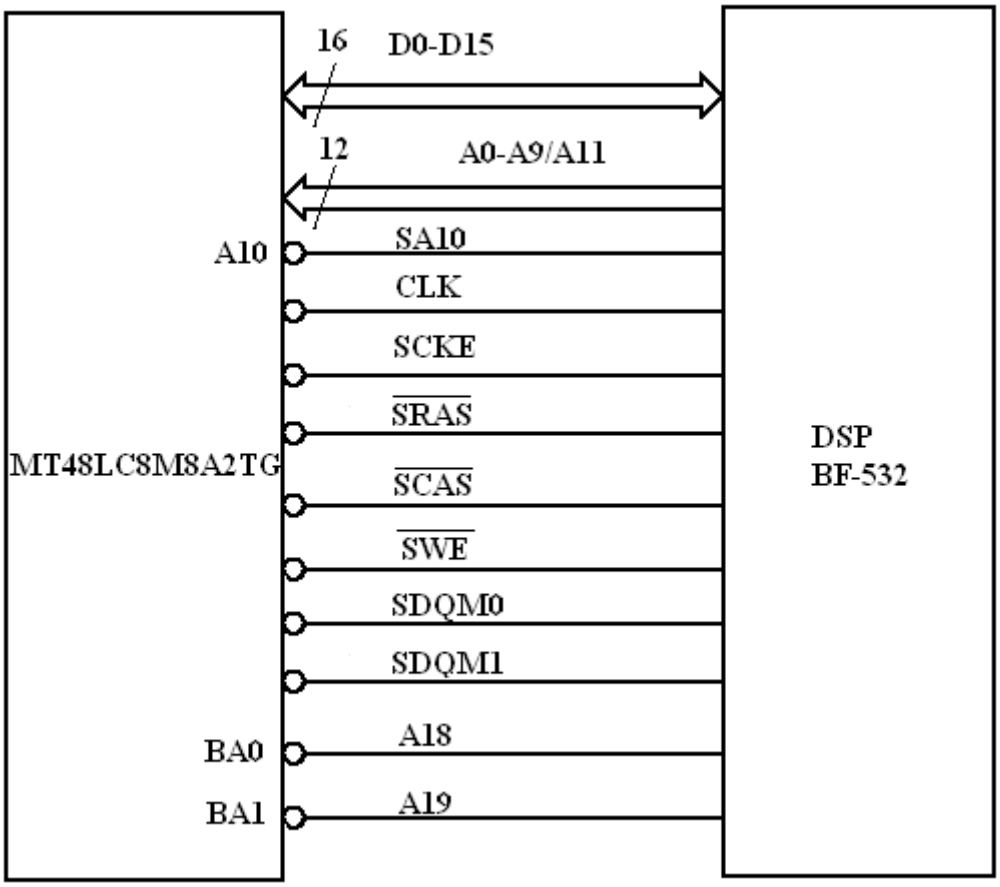


Figura 2.27: Interfaccia tra processore e memoria SDRAM

La memoria prevede all'accensione una fase di inizializzazione, dove vengono impostati i parametri principali, quali il CAS latency, che fornisce il ritardo in termini di colpi di clock dal comando, per esempio di lettura, e la disponibilità del dato sul bus e la modalità di accesso, singola o a burst; infatti la memoria prevede la possibilità di accedere a burst di 1,2,4,8 o l'intera colonna (256), in modo sequenziale o interfacciato, ossia che ad ogni comando di lettura o scrittura vengono letti o scritti 2,4,8 o 256 (nel nostro caso) locazioni di memoria, mandando solo l'indirizzo iniziale del burst. Il controller per memorie SDRAM del processore supporta solo accessi a burst pari a 1; in fase di inizializzazione del mode regoster verrà settato il burst length a 1 e il burst type come sequenziale, sebbene essendo a lungo 1 word questo settaggio è ininfluente. Inoltre la memoria adottata, prvede due cicli di clock come CAS latency. Le varie tempistiche sono state definite via software come verrà illustrato in seguito.

Particolarità fondamentale della memoria dinamica è la necessità di un continuo 'refresh' di ogni colonna ogni 15,625μs ossia di una riga ogni 64ms. Questo avviene automaticamente mediante un contatore interno alla memoria che provvede al continuo refresh delle colonne di memoria. Questo causa un assorbimento di 3 mA con il controller sincrono del processore attivo, quindi con la linea di clock funzionante, altrimenti assorbe 1 mA se il sistema è acceso ma la SDRAM e il controller sul Blackfin non sono utilizzati.
Quindi per il calcolo della corrente assorbita e della potenza dissipata dal componente, si devono distinguere le due modalità principali, quella operativa e quella di standby:

Fase di utilizzo	Corrente	Potenza
Lettura/Scrittura	115 mA	379.5 mW
Standby Active Mode	45 mA	148.5 mW
Standby	2 mA	6.6 mW

In realtà quella di standby può essere di due tipi, ossia quella a clock spento riportata nella tabella precedente e quella di stato di temporaneo inutilizzo tra per esempio accessi consecutivi, causati per esempio da elaborazione dei dati acquisiti da parte del processore. Noi supponiamo di trascurare tale fase e di considerare la memoria sempre operativa, in lettura o scrittura, nelle fasi in cui accediamo direttamente alla memoria; tale strategia comporta calcoli conservativi, infatti nei calcoli dei consumi di energia considereremo operativa la memoria per tutto il tempo in cui viene utilizzata, trascurando quindi i vari tempi morti, che sebbene sicuramente di dimensioni ridotte ma sicuramente presenti. Ma comunque come si vedrà in seguito, potrebbe servire il consumo in questa fase perché non tutti gli accessi alla memoria SDRAM saranno noti al nostro programma, come verrà spiegato nella sezione software.
Come è stato accennato in precedenza, la memoria adottata è di 8Mbyte, in modo da poter avere 4 locazioni di memoria da 2 Mbyte ciascuno contigue, dove poter mettere temporaneamente 4 fotogrammi prima della compressione JPEG; questo può essere utile nel caso si voglia scattare 4 foto una subito dopo l'altra, quindi nello stesso ciclo di funzionamento, magari per fotografare qualcosa di particolare.
Si parla di locazioni da 2Mbyte contigue perché il controller della memoria sincrona supporta un indirizzamento per memorie da 16Mbyte a 128 Mbyte, quindi prevede l'utilizzo di 13 linee di indirizzo per le righe e 2 per l'accesso ai banchi interni della memoria. Si poteva anche adottare una memoria da 16 Mbyte ma il consumo sarebbe notevolmente aumentato, in quanto le celle su cui passa la linea di clock o su cui eseguire il refresh sono il doppio. Con la nostra memoria si va ad

utilizzare entrambi i segnali dei banchi di memoria ma non viene utilizzata una linea di indirizzo per le righe. Qui di seguito viene illustrato il particolare indirizzamento adottato per la memoria SDRAM:

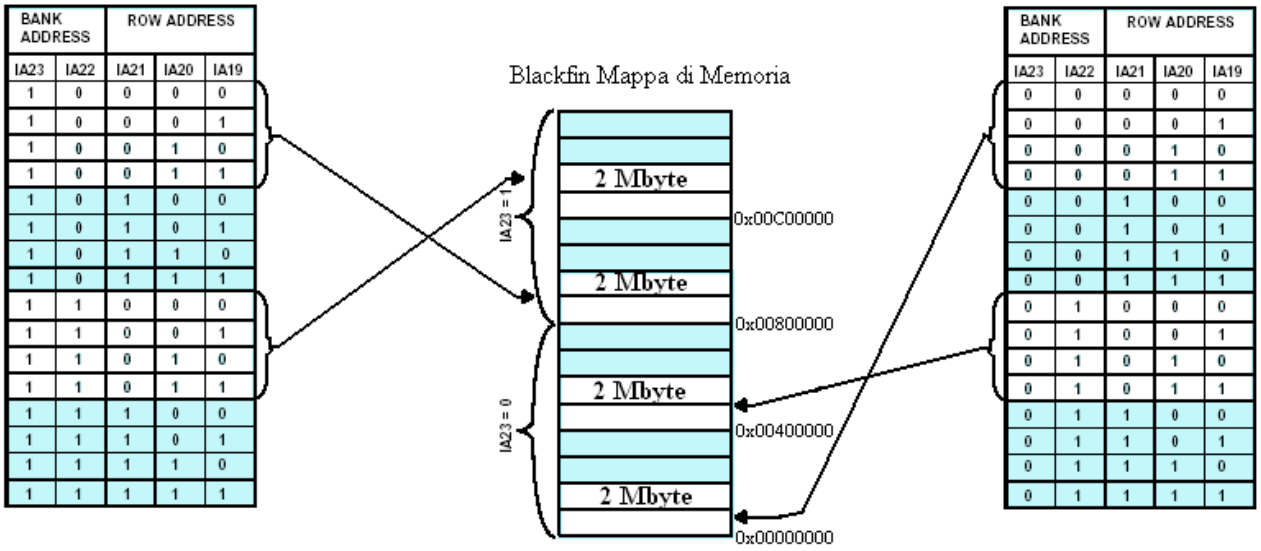


Figura 2.28: Indirizzamento Blackfin per la memoria SDRAM

Infine la potenza assorbita dal processore Blackfin nell'interfaccia con la memoria SDRAM risulta molto complicato da calcolare, dato il complesso funzionamento della memoria stessa. L'Analog Devices fornisce un datasheet per questo e da esso si sa che la potenza assorbita a questa frequenza è di circa 180 mW, in ottica molto conservativa.

2.4 Interfacce

2.4.1 UART

Come descritto in precedenza il processore Blackfin è dotato di periferiche di diverso tipo e il sistema Payload necessita di interfacciarsi con due processori, residenti su due schede diverse all'interno del satellite; allora per la comunicazione con il processore chiamato ProcA si è deciso di utilizzare la porta Universal Asynchronous Receiver Transmitter(UART). Essa è una porta seriale di comunicazione seriale asincrona full-duplex, infatti prevede la presenza di soli due segnali, uno per la ricezione denominato RX e uno per la trasmissione chiamato TX.

Il lato trasmissione prevede la presenza di un registro che funge da buffer del dato da trasmettere per lo shifter register utilizzato in modalità P.I.S.O. (Parallel In Serial Out) che effettua realmente la trasmissione. Quindi quando il processore vede il buffer vuoto può riempirlo col dato successivo,

anche se il dato precedente non è ancora stato completamente trasmesso, in modo da poter mantenere una piccola coda di trasmissione. Quando il buffer è vuoto può generare un interrupt o settare un flag in un registro, in modo che il processore possa riempirlo subito dopo il suo svuotamento.

Il lato ricezione è molto simile, in cui l'obiettivo è invece quello di mantenere lo shift register sempre libero e pronto a ricevere. Infatti quest'ultimo è qui utilizzato in configurazione S.I.P.O. cioè Serial In Parallel Out, appena ricevuto interamente un dato, lo passa ad un registro di buffer, rendendolo disponibile al processore e svuotandosi subito in modo da essere pronto a ricevere subito un altro dato. Anche qui, appena al buffer è passato un dato ricevuto, può venir generato un interrupt oppure può venire settato un flag in un registro, in modo da poter svolgere una ricezione continua dei dati.

Essendo una trasmissione asincrona, quindi senza una linea esterna di clock, la sincronizzazione è a livello di bit; infatti ogni parola trasmessa dovrà essere preceduta da uno start bit, che segnala l'inizio della trasmissione di un dato e seguita da uno stop bit che ne conclude la trasmissione. Quesit due sono ottenuti mantenendo sempre la linea a livello logico alto quando non utilizzata e quindi il primo livello logico basso viene letto come uno start bit e la transizione da '0' a '1' dopo l'intera parola viene letto come uno stop bit.

Infatti il processore deve essere settato nei diversi parametri della parola di dato trasmessa:

- Bit di dato variabili da 5 a 8
- Assenza o presenza di bit di parità pari o dispari
- 1, 1,5 o 2 stop bit
- Baude rate = SCLK/(16 × Divisore), dove l'SCLK è il clock delle periferiche del processore (settato a circa 100MHz) e il divisore va da 1 a 65536, mediante il quale si ottiene la velocità di trasmissione voluta.

Qui di seguito viene riportato un'esempio della trasmissione UART:

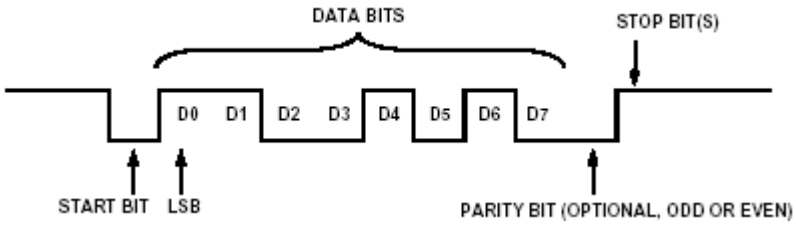


Figura 2.29: Stream di una trasmissione UART

Nel sistema Payload per la comunicazione con il ProcA si è adottato una parola da 8 bit, senza bit di parità e con uno stop bit. Inoltre la velocità di trasmissioneè stata impostata a 9600 bps andando a

impostare il divisore a 10368, sapendo che la frequenza del clock sugli I/O del processore SCLK è di 99,5328 MHz.

Inoltre sono stati collegati in parallelo alla linea TX e RX, il segnale di ricezione DR1PRI e trasmissione DT1PRI rispettivamente della porta seriale generica SPORT1 presente sul processore. Mediante questi due segnali e altri due, uno di sincronizzazione (TSCLK1 e FSCLK1 in parallelo pechè sono gli stessi nelle due fasi) e uno di select (TFS1 e RFS1 in parallelo perche identici nelle due fasi), si è previsto un canale di comunicazione di riserva su cui sarebbe potuto essere implementato un protocollo SPI, utilizzabile in caso di erroneo funzionamento della porta UART. Infatti anche il ProcA da lato suo, ha collegato questi segnali alla sua porta SPI.

La dissipazione di potenza dovuta a tale porta di comunicazione è molto ridotta data la presenza di un solo segnale e della frequenza bassa di utilizzo. Comunque dato che la durata della trasmissione di un'intera immagine al ProcA potrebbe durare molto tempo, può essere poi interessante sotto l'aspetto di energia consumata. Per questo motivo viene calcolata solo quella dissipata in fase di trasmissione e viene trascurata quella in fase di ricezione, dato che serve solo per la ricezione di uno o pochi byte di telecomando.

Sapendo inoltre che la capacità d'ingresso del processore chiamato ProcA è di 8pF, la potenza dissipata è la seguente:

$$P = 1 * 8\ pF * (3{,}3V)^2 * 9600 \approx 1\mu W$$

Il protocollo seriale asincrono implementato sulla porta UART è anche quello utilizzato dal protocollo seriale RS-232 presente generalmente sui PC. L'unica differenza sta nell'ampiezza dei livelli logici, solitamente lo '0' è +6V o +12V e l'1' è –6V o –12V.

Per quindi poter comunicare con il PC durante il debug del software e poter simulare la trasmissione col ProcA si è dovuto adottare un driver 232 prodotto da Analog Devices chiamato ADM3202, il quale crea i due livelli raddoppianto internamente la tensione di alimentazione mediante dei condensatori esterni in modo da creare un charge pump. Questo componente nel prototipo del sistema era stato montato direttamente sulla scheda mentre sulla scheda definitiva è stato poi montato su una scheda creata apposta per il debug del sistema.

2.4.2 SPI

Per la comunicazione con l'altro processore presente all'interno del satellite, chiamato ProcB, si è cercato di utilizzare una porta hardware distinta e di diverso tipo da quella impiegata nella comunicazione con il ProcA, sempre nell'ottica della ridondanza del sistema.

Si è quindi deciso di adottare la porta di comunicazione Serial Peripheral Interface (SPI) per comunicare con il ProcB; l'SPI è un protocollo seriale sincrono, che quindi prevede la presenza di un segnale di clock, chiamato SCK, studiato per la comunicazione su bus industriali ricchi di dispositivi con arbitraggio di tipo geografico; questo significa che la comunicazione prevede la presenza di un master e di molti slave, ognuno dei quali viene interpellato solo quando diventa attivo il suo chip select, qui denominato SPISS, SPI device Select Signal. Per questo motivo il processore BF-532 presenta sette pin di uscita per poter selezionare sette dispositivi slave se utilizzato in modalità master e un solo pin SPISS se impiegato come slave della comunicazione.
Essendo la scheda Payload una scheda che riceve telecomandi ed esegue solo quello contenuto in essi, il suo comportamento all'interno del sistema PICPOT è intrinsecamente slave rispetto ai due processori ProcA e ProcB. Quindi il ruolo del DSP nella comunicazione SPI sarà di slave, quindi sarà connesso all'MSP430, ossia il processore montato sulla scheda ProcB, mediante il segnale SPISS, quello di sincronizzazione SCK e i due segnali di dato che in tale configurazione saranno per la ricezione quello denominato MOSI (Master Out Slave In) e per la trasmissione quello chiamato MISO (Master In Slave Out).
Qui di seguito è rappresentato lo schema dell'interfaccia SPI del processore con il ProcB:

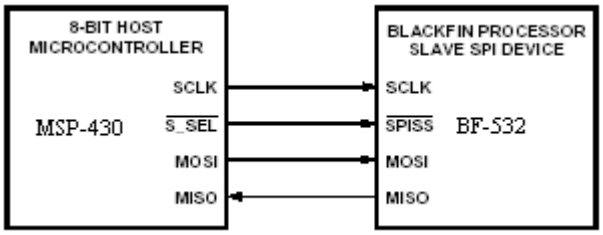


Figura 2.30: Interfaccia ProcB - sistema Payload

La struttura interna della porta è molto simile all'UART; infatti anche qui vi è un solo shift register utilizzato in modalità P.I.S.O o S.I.P.O., rispettivamente nelle fase di trasmissione e ricezione, con due buffer distinti per le due modalità di comunicazione. Inoltre qui è presente una coda FIFO profonda 4 words, per garantire sempre dei dati pronti ad essere trasmessi o il buffer di ricezione vuoto per essere pronto a ricevere nuovi dati.
La struttura della porta SPI è riportata nella figura 2.31.
In modalità di slave la comunicazione ha inizio appena diventa attivo, ossia va a livello logico basso, il segnale di select SPISS. Da qui il segnale di clock viene pilotato dal master e a seconda delle polarità impostato, sul livello alto o basso del segnale SCK, viene campionato il valore del dato in ricezione. Quando il buffer è pieno esso genera un interrupt al processore per avvertirlo del

nuovo dato ricevuto. La parola può essere di 8 o 16 bit, con il primo bit l'LSB o l'MSB e altri parametri possono essere impostati nel registro di controllo.

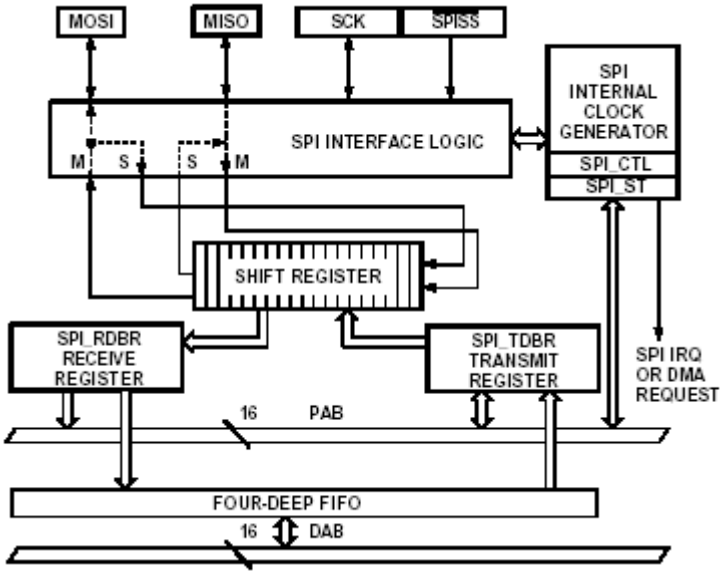


Figura 2.31: Struttura a blocchi porta SPI

La comunicazione col ProcB dovrebbe avvenre su 8 bits, con il Most Significant Bit come primo, il clock SCK attivo sul livello alto, con l'inizio della sua commutazione con l'inizio del primo bit, purtroppo però non è ancora stato collaudata.
Inoltre la frequenza del segnale di clock è ottenuta nel modo seguente:

$$SPI\ Clock\ Rate = \frac{f_{SCLK}}{2 \times SPI\_Baud}$$

dove l'SPI_Baud è il registro a 16 bit che funge da divisore da 1 a 65536 al clock SCLK degli I/O del processore.
La dissipazione di potenza dovuta a tale porta di comunicazione è comunque anch'essa piuttosto ridotta data la presenza di soli due segnali utilizzati contemporaneamente e la frequenza bassa di utilizzo. Comunque dato che la durata della trasmissione di un'intera immagine al ProcB potrebbe durare molto tempo, può essere poi interessante sotto l'aspetto di energia consumata. Per questo motivo viene calcolata solo quella dissipata in fase di trasmissione e viene trascurata anche qui quella in fase di ricezione, dato che serve solo per la ricezione di uno o pochi byte di telecomando.
Sapendo inoltre che la capacità d'ingresso del processore chiamato ProcB è di 8pF, e che il segnale di dati commuta al massimo con una frequenza pari alla metà di quella di clock, la potenza dissipata è la seguente:

$$P = (1 * 8\ pF * (3{,}3V)^2 * 19200) + (1 * 8\ pF * (3{,}3V)^2 * 9600) \approx 2\mu W$$

2.4.3 JTAG/IEEE 1149.1

Il processore BF-532 è completamente compatibile col IEEE 1149.1 standard, anche conosciuto come lo standard del Join Test Action Group (JTAG).
Il JTAG standard definisce la ciruiteria presente sul processore per eseguire la programmazione e il test del componente saldato su una scheda. Esso definisce la logica di test che è inclusa sul processore per implementare:

- Test sulle interconnessioni tra i circuit integrati collegati al processore sulla scheda
- Test interni del componente integrato
- Osservazione e modificazone dell'attività circuitale durante le normali operazioni svolte dal componente

La logica di test consiste in uno registro chiamato Boundary-Scan. L'accesso ad esso avviene tramite una Test Access Port (TAP). La struttura di tale logica è la seguente:

- Un TAP composto da 5 pin, si veda la tabella seguente
- Un TAP controller che controlla tutte le sequenze di eventi attraverso i registri di test
- Un Instruction Register (IR) che interpreta le istruzioni codificate su 5 bit per selezionare la modalità di test e l'operazione desiderata
- Numerosi registri di dati definiti dallo standard JTAG

Pin Name	Input/Output	Description
TDI	Input	Test Data Input
TMS	Input	Test Mode Select
TCK	Input	Test Clock
$\overline{\text{TRST}}$	Input	Test Reset
TDO	Output	Test Data Out

Il controller TAP è macchina a stati sincrona a 16 stati controllata dai pin TCK e TMS. Le transizioni tra i vari stati nel diagramma avvengono sui fronti di salita di TCK e sono definiti dallo stato del pin TMS.
Al fine di utilizzare la porta JTAG del processore Blackfin anche per la programmazione e il debug del software, oltre che per le funzionalità illustrate in precedenza, si è utilizzato un emulatore JTAG prodotto da Analog Devices, chiamato SUMMIT ICE JTAG Emulator Interface, il quale mediante una circuiteria particolare e una scheda PCI collegata ald un PC, permetteva appunto di

programmare, svolgere debug, monitorare il contenuto della memoria interna ed esterna e lo stato di tutti i registri sul processore, saldato sulla scheda Payload. Si è quindi installato sul prototipo della scheda e per la versione finale sulla scheda chiamata J0 Expansion creata per il test, un connettore d'interfaccia tra l'emulatore JTAG e il processore. Tale interfaccia è composta dai 5 pin illustrati in precedenza più un pin chiamato EMU, il quale è un flag generato dal processore quando riconosce la presenza dell'emulatore collegato. Inoltre il connettore dell'emulatore è di 14 pin, di cui uno non connesso, e dei restanti 3 sono segnali di massa, 1 fornisce l'alimentazione 3,3V alla logica dell'emulatore, prelevandola direttamente dalla scheda e gli ultimi 3 pin, nel nostro caso non utilizzati quindi connessi a massa, servono se si utilizza un processore senza Boundary-Scan controller, il quale può essere svolto dall'emulatore JTAG mediante questi 3 segnali.
Ecco che quindi l'interfaccia completa del processore con l'emulatore JTAG è la seguente:

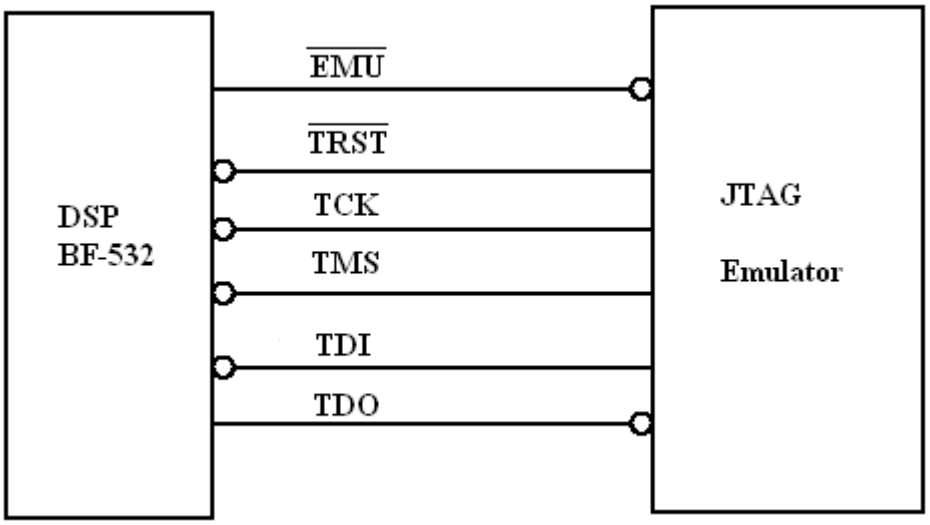


Figura 2.32: Interfaccia porta JTAG

2.4.4 Alimentazioni

Come spiegato in precedenza, i segnali di alimentazione per il funzionamento dell'intera scheda sono forniti dalla scheda PowerSwitch, presente all'interno di PICPOT. Le tensioni richieste dai componenti adottati sulla scheda Payload, sono:

- 3,3V necessari per tutte le memorie, per le periferiche del DSP e del decoder video, per tutti i restanti componenti integrati digitali utilizzati sulla scheda
- 12V necessari per il funzionamento delle videocamere adottate
- 1,8V necessari per alimentare il core del decoder video

Si deve notare che non è stata inclusa la tensione necessaria per il funzionamento del core del DSP, la quale infatti è ottenuta sulla scheda stessa tramite un regolatore buck realizzato mediante componenti discreti e comandato direttamente dal processore stesso, quindi non richiede una tensione esterna.
Essendo il segnale da 1,8V limitato solo al core del decoder video, quindi un ridotto assorbimento di corrente, di poco superiore a 50 mA, e facilmente ottenibile dalla linea a 3,3V, tale tensione è stata appunto ottenuta mediante un regolatore di tensione low voltage drop out (LDO) montato direttamente sulla scheda Payload e quindi non incluso nell'interfaccia con la scheda PowerSwitch.
Le tensioni quindi direttamente fornite da quest'ultima sono quella da 12V e da 3,3V che, insieme ai segnali di massa, compongono l'interfaccia di alimentazione della scheda Payload.
Inoltre saranno presenti segnali di massa di due tipi:

- digitale, per tutti i componenti digitali utilizzati all'interno del sistema
- analogica, per il circuito di condizionamento del sensore di temperatura, in quanto è quella utilizzata come riferimento dalle schede che ne andranno a leggere il valore.

Infine, come verrà spiegato in seguito, sono previste due linee distinte di alimentazioni per il circuito di condizionamento del sensore di temperatura, provenienti rispettivamente una dal ProcA e una dal ProcB, sono alimentazioni non fisse e gestite completamente dalle due schede, in modo indipendente dal sistema Payload e dal suo funzionamente. Per questo motivo non sono considerate delle componenti dell'interfaccia di alimentazione della scheda, ma ciascuna verrà inglobata nell'interfaccia col rispettivo processore.

2.4.5 Connettore J0

Il connettore J0 è l'unico connettore presente sulla scheda Payload, il quale essendo di tipo backplane permette il passaggio dei segnali da una scheda all'altra, oltre alla possibilità di collegarsi ad essi.
La scheda Payload utilizza tale connettore per collegare i diversi segnali d'interfaccia spiegati in precedenza:

- 2 segnali UART per la comunicazione col ProcA e 2 segnali supplementari per una possibile implementazione del protocollo SPI su una porta seriale generica (SPORT)
- 1 segnale digitale per la selezione della modalità di boot da parte del ProcA

- 1 segnale di alimentazione del circuito di condizionamento del sensore di temperatura proveniente dal ProcA
- 1 segnale analogico di uscita del sensore di temperatura per il ProcA
- 4 segnali SPI per la comunicazione col ProcB
- 1 segnale digitale per la selezione della modalità di boot da parte del ProcB
- 1 segnale di alimentazione del circuito di condizionamento del sensore di temperatura proveniente dal ProcB
- 1 segnale analogico di uscita del sensore di temperatura per il ProcB
- 2 tensioni di alimentazioni, 12V e 3,3V
- 3 segnali di alimentazione distinti per le 3 videocamere
- massa analogica per il sensore di temperatura
- massa digitale per il resto del sistema Payload
- i 6 segnali JTAG, i 4 standard TDI, TDO, TCK, TMS oltre ai 2 utilizzati dall'emulatore di Analog Devices, l'EMU e il TRST
- segnale video analogico PAL delle videocamere (valido per tutte e tre dato che sono in parallelo)

Questo è l'insieme delle interfacce della scheda Payload verso il resto del sistema PICPOT, raggruppabili in 7 segnali verso la scheda ProcA, 7 segnali verso la scheda ProcB, 5 segnali verso le video camere (3 di alimentazione, la massa 3 il segnale video PAL), 4 segnali dalla PowerSwitch (le due alimentazioni e le due masse) e 6 segnali verso il connettore di Test.

Il connettore di Test è appunto il connettore raggiungibile dall'esterno a satellite chiuso, per la programmazione dei vari processori e lo sniffing delle comunicazione tra le varie schede presenti al suo interno durante il suo funzionamento. La scheda Payload fornisce al connettore di test il segnale analogico PAL e la sua rispettiva massa (ossia quella digitale usata come riferimento), per poter visualizzare esternamente cosa punta il satellite, oltre ai 4 segnali standard JTAG per la programmazione esterna.

Infatti si è previsto di collegare tutti i processori presenti all'interno del satellite in un'unica catena JTAG. Quindi i segnali dei processori TDI e TDO sono gli unici segnali collegati a un pin non passante del connettore backplane J0; infatti, il segnale TDI sarà collegato al pin TDI del socket del J0 mentre il TDO al TDI del plug del connettore. Invece il pin TDO sul connettore J0 sarà passante e sarà quello a cui collegherà il suo segnale TDO l'ultimo processore della catena per chiuderla e ritornare al connettore di Test, mentre le schede in mezzo, quale per esempio la Payload, non sarà collegata ad esso.

2.5 Sensore di Temperatura

Le misure di temperatura sono utilizzate per conoscere le condizioni ambientali cui è sottoposto il satellite durante i periodi di tempo in cui è illuminato dal sole rispetto a quelli in cui è coperto e per conoscere il surriscaldamento relativo alle dissipazioni di energia per effetto Joule.
In fase di progetto si è ritenuto opportuno come range di temperatura che doveva essere tollerato dal sensore, tra 0°C e 100°C. Nella scelta dei sensori di temperatura se ne sono analizzati diversi tipi; infatti i trasduttori a filo metallico, lineari e con ampie regioni di funzionamento, hanno però due svantaggi:

- una bassa resistività del materiale utilizzato che causa una bassa resistenza del sensore e, quindi, un'elevata dissipazione di potenza;
- un ingombro notevole, dato dall'avvolgimento del filo metallico, effettuato molte volte appunto per innalzare il valore resistivo del sensore stesso.

A causa di questi inconvenienti si è stati costretti ad utilizzare dei termistori di tipo NTC. Questi hanno pregi ed inconvenienti scambiati rispetto ai sensori a filo, infatti sono poco lineari, poco precisi e con range limitato, ma presentano un'elevata resistenza, in concomitanza ad un occupazione ridotta di spazio.
Gli unici termistori capaci di soddisfare le specifiche sono stati gli NTC TR0603J252K della RTI Electronics, le cui caratteristiche sono:

- spessore pari a 0,4mm;
- resistenza variabile da 100 a 42,5k;
- range di funzionamento compreso tra -55_C e +160_C;
- tolleranza costruttiva del sensore del 10%.

La scelta di un valore di resistenza così basso è da imputare alla curva della caratteristica del sensore: infatti è l'NTC con valore resistivo maggiore che presenta una curva caratteristica con minore escursione. E' noto, come la caratteristica degli NTC sia altamente non lineare, rappresentata in figura 2.33.
Per questo motivo è necessario effettuare una prima linearizzazione via hardware per permettere di realizzare una seconda, più accurata linearizzazione software della caratteristica. Per questo, si è utilizzata la non linearità introdotta dal ponte di Wheatstone, per ridurre la dinamica del termistore: si è realizzato un circuito come quello riportato nella seguente.

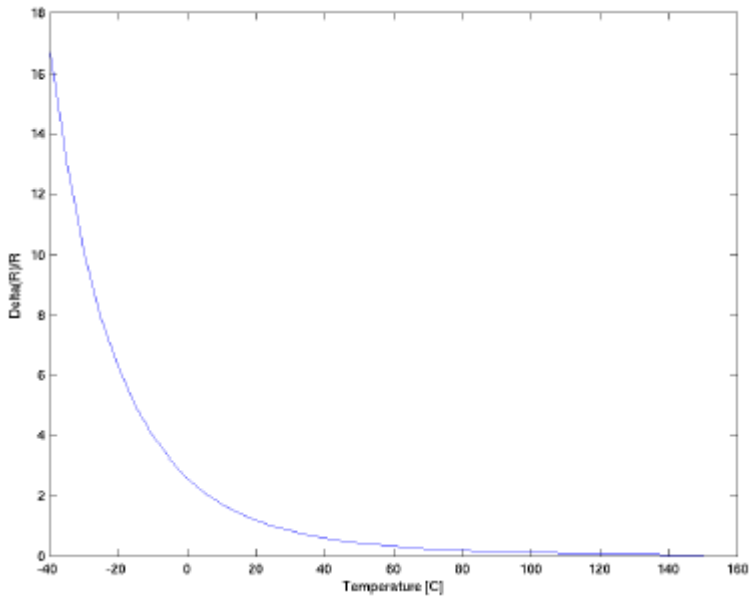


Figura 2.33: Caratteristica dell'NTC, normalizzata al valore di resistenza a 25°C

Oltre questo motivo l'introduzione del ponte ha permesso di ridurre il consumo di potenza, in quanto la resistenza inserita in serie all'NTC riduce notevolmente la corrente che lo attraversa. Per evitare in uscita delle tensioni negative, non gestibili nè dall'operazionale, nè dal convertitore A/D, è importante regolare il ramo di riferimento in modo da fornire una tensione pari alla minima generata dal ramo di misura.

L'amplificatore operazionale utilizzato è prodotto dalla Maxim, denominato MAX4092 e disponibile in package SOIC8. La tensione di riferimento Vr è ottenuta da un riferimento di tensione di 1,2V integrato in un SOT23, mentre i restanti sono componenti discreti.

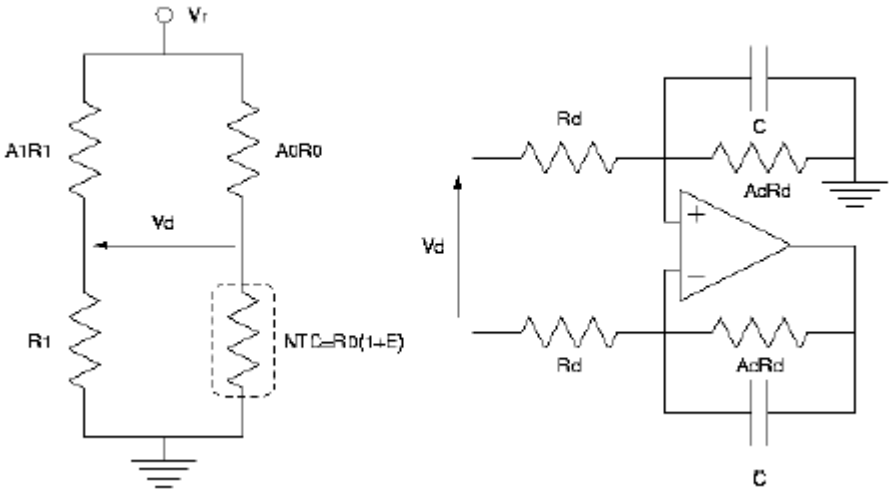


Figura 2.34: Schema del circuito di condizionamento del sensore di temperatura

L'alimentazione dell'amplificatore operazionale avviene da due segnali provenienti, uno dal ProcA e uno dal ProcB. Dovendo andare sullo stesso pin, si è posto un diodo su ciascun segnale, così da avere un'or logico sulla linea di alimentazione e non cortocircuitare i pin della scheda ProcA con quelli della scheda ProcB, nel caso in cui siano entrambi accesi.
Il consumo di corrente da parte del sensore di temperatura interessa i due processori ProcA e ProcB, non il nostro sistema che fornisce solo il riferimento di tensione da 1,25V, che alimenta la scala di resistori e NTC, che nel caso peggiore assorbono circa 50 μA, da 1,25V / (47 kΩ /2), visto che nel caso limite sono due resistenze da 47 kΩ in parallelo.

2.6 Consumi Energetici

I calcoli dei consumi di corrente, potenza ed energia sono di vitale importanza per un sistema alimentato a batterie, che deve funzionare in ambiente spaziale (quindi con una ridotta capacità di dissipazione del calore) e che deve essere alimentata da un alimentatore specifico. Infatti il consumo massimo e minimo di corrente servono al progettista degli alimentatori (in questoi caso la scheda PowerSwitch) per dimensionare i componenti degli stessi; la dissipazione di potenza serve per il calcolo dell'andamento del calore all'interno del satellite e il consumo energetico nelle diverse fasi di funzionamento serve a chi gestisce le batterie per sapere se l'energia immagazzinata in esse è sufficiente per più cicli di funzionamento.
Per tale scopo si sono suddivisi i calcoli in base ai diversi cicli di funzionamento, descritti poi meglio nella sezione dedicata al progetto software. Quindi questi dati saranno maggiormente comprensibili dopo aver compreso come vengono organizzati i dati, soprattutto nelle memorie esterne, le cui interfacce col Blackfin comportano un importante consumo di potenza, anche se però difficile da stimare con certezza in quanto molto diverso tra loro. Per questo per i calcoli sull'energia consumata, dove è presente anche il fattore tempo, si sono fatte delle stime del tempo di rispettivo utilizzo delle memorie.
I calcoli sono riportati nella tabella riportata nella pagina successiva.

Fase	Componenti attivi	Corrente assorbita	Potenza media	Energia
Scatto Foto	Fotocamera Decoder video DSP SDRAM SRAM(opzionale)	65 mA @ 12V 55 mA @ 1,8V 268 mA@ 3,3V	1,8mW	2,8J(Nota 1)
Compressione blocco JPEG	SDRAM SRAM DSP	265 mA @ 3,3V	750 mW	375mJ(Nota 3)
Salvataggio blocco JPEG	SDRAM SRAM DSP FLASH	285 mA @ 3,3V	820 mW(Nota 2)	164mJ(Nota 7)
Rx/Tx Telecomando UART	DSP SDRAM	230 mA @ 3,3V	630 mW	630μJ(Nota 6)
Tx Foto UART	DSP FLASH SDRAM(Nota 4)	240 mA @ 3,3V	700 mW	38,5J(Nota 5)

NOTE:

1. Si è considerato per i circa 2s d'accensione della videocamera solo l'assorbimento della stessa più quello del DSP in attesa, mentre per 50 ms (tempo d'acquisizione stimato) l'assorbimento dovuto a tutti i componenti.
2. Si è considerato l'interfaccia alle memoria come se fosse sempre quella alla SDRAM, in quanto comporta il maggiore consumo di potenza, comportando quindi un valore molto conservativo.
3. Si è considerato un tempo di compressione per ogni blocco di 0,5s.
4. Si è considerato sulla potenza dissipata dal sistema nell'interfaccia tra DSP e le memorie, un derating di 0,5 utilizzando la memoria SDRAM e di 0,5 utilizzando la memoria FLASH, dato che per metà tempo legge dati e per metà tempo il processore eseguirà il codice contenuto probabilmente in SDRAM.
5. Si è considerato un tempo massimo di trasmissione di 55s.
6. Si è considerato un tempo massimo di 1ms.
7. Si è considerato un derating di 0,5 sulla potenza dissipata dal sistema scrivendo o leggendo in SDRAM e quella scrivendo e leggendo sulla FLASH. Inoltre si e` considerato il tempo tipico richiesto da una scrittura di una word da 16 bit sulla FLASH di 15 μs (dai dal datasheet), e considerando una dimensione massima di 12 Kbyte per ogni blocco (100 Kbyte / 9), comporta un tempo impiegato per il salvataggio di un blocco pari a 200ms

Capitolo 3

Realizzazione

La realizzazione del PCB della scheda e la saldatura dei componenti sono risultate le fasi più lunghe e delicate dell'intero lavoro.
La realizzazione del PCB è avvenuta mediante il pacchetto software CAE (Computer Aided Design) di Mentor Graphics, disponibile presso il dipartimento di elettronica del Politecnico di Torino.
Per arrivare a realizzare uno stampato mediante tale ambiente di sviluppo, si devono seguire diversi step adottando diversi sottoprogrammi, ciascun alquanto complesso e non sempre user friendly.
Infatti il processo si suddivide in tre grandi operazioni:

1. creazione di una libreria dei componenti utilizzati sulla scheda mediante Library Manager di Mentor Graphics
2. disegno dello schema elettrico, mediante il programma Design Capture di Mentor Graphics
3. posizionamento dei componenti sullo stampato, routing delle piste e verifica dei constraints mediante Express PCB di Mentor Graphics.

3.1 Libreria

Il primo step, prevede la creazione della libreria dei componenti utilizzati; tale libreria sarà composta di tre parti:

- la cella logica rappresentante il componente discreto o integrato, che fornisce informazioni sul tipo (alimentazione, massa, analogico, digitale, ingresso, uscita, bidirezionale), il nome e il numero dei pin del componente. Queste celle, chiamate SYMBOLS, sono impiegati per disegnare lo schema elettrico e le loro informazioni sono utilizzate per eseguire controlli sulla logica delle interconnessioni. Tali celle sono creabili e modificabili mediante il Symbol Editor.
- la cella fisica, cioè le dimensioni del package, di ogni componente integrato o discreto. Tali celle, chiamate CELL, sono quelle poi impiegate nel posizionamento

dei footprint dei componenti sul PCB. Solitamente i componenti integrati utilizzano package standard e quindi tali celle si trovano in librerie standard già esistenti. Comunque mediante il Cell Editor, si può creare o modificare le dimensioni dei footprint.

- Il componente completo, sia della parte logica che di quella fisica, chiamato PARTS. Questo passo è quello più importante nelle operazioni di libreria, in quanto esegue il collegamento tra la cella logica e la cella fisica, collegando i pin disegnati sulla cella per lo schema elettrico con quelli che poi andranno sul PCB e orinandoli per tipo e numero. Il programma permette di controllare il risultato di tale collegamento, perché se si sbaglia in questo step, il PCB non riporterà i collegamenti dello schema elettrico e tutto sarà sbagliato e molto difficile da vedere, se non con un'analisi del PCB, molto arduo in scheda leggermente complesse.

La parte è poi la cella a cui si fa riferimento nello schema elettrico, dove viene utilizzata la sua parte logica, e nel PCB dove viene impiegata la sua parte fisica.
Alla parte viene associato un numero (part number), un nome (part name)e un'etichetta univoca (part label). Nella libreria della scheda Payload si è inserito come part number per ogni componente il suo codice d'ordine del distributore da cui è stato acquistato, preceduto dalle sue iniziali, come DK per DigiKey, RS per RS Components, FA per Farnell. Un part number tipico è DK296-8743-1-ND. Se invece il componente è un stato ottenuto come sample, si è scritto la sigla del componente preceduta da SM. Invece come part name è stato messa la sigla del componente, se esso integrato, altrimenti il valore del componente discreto preceduto da R nel caso di una resistenza, C se condensatore e L se induttore. Infine come etichetta univoca è stata messa, nel caso di componenti integrati la loro sigla per intero, mentre bel caso di componenti discreti è stata utilizzata una stringa che raccoglie tutte le informazioni più rilevanti per ogni componente. Infatti a seconda della tipologia del componente le stringe utilizzate sono le seguenti:

- Resistenze: R_valore(Ω)_dimensioni(mils)_potenza(mW)_tolleranza. Così una resistenza da 15kΩ, in un package 06x03 mils, da 100mW all'1% diventa: R_15k_0603_100_1
- Condensatori: C_valore(F)_dimensioni(mils)_dielettrico_tensioneMax(V). Cosi un condensatore da 22 pF, in un package 06x03 mils, con un dielettrico C0G e un tensione massima supportata di 50V diventa: C_22p_0603_C0G_50V
- Induttori: L_valore(H)_dimensioni(mm)_correnteMax(A)_tolleranza. Cosi un induttore da 10μH, in un package 7mm x 7 mm, con una corrente massima supportata di 1,4A e una tolleranza del 20% diventa: L_10μ_7x7_1A4_20

Inoltre ad ogni parte può essere associata la lettera da utilizzare per il Reference Designator dei componenti; nella scheda Payload sono state utilizzate:

- R per i resistori
- C per i condensatori
- L per gli induttori
- D per i diodi
- Q per i transistor discreti (lunico presente è il p-mos per il regolatore buck per il core del processore)
- X per i quarzi
- J per il connettore
- U per i componenti integrati

Infine la libreria deve essere anche completa di tutti i fori utilizzati nella scheda, sia quelli strutturali che quelli funzionali come via e hole per connettori non superficiali. Tutte le informazioni sulle dimensioni e forme dei buchi impiegati sono creabili e modificabili mediante il PadStack Editor.
A questo punto si hanno tutti i componenti disponibili per la creazione dello schema elettrico e del PCB.

3.2 Schema elettrico

Per disegnare lo schema elettrico, si è utilizzato il software Design Capture, che si deve riferire alla libreria della scheda. A questo punto si utilizzano le celle logiche disegnate prima nella libreria e si collegano tra di loro mediante il comando Wire oppure per collegamenti di segnali multipli si può utilizzare il comando Bus. Inoltre sono presenti simboli generici, come il simbolo per le alimentazioni Vdd (utilizzato per la line a 1,8V del decoder video), ValPay33 (per la linea di alimentazione a 3,3V), ValPay12 (per il segnale di alimentazione a 12V) e le masse digitali e analogiche, Gnd e GndAnal. Questi simboli valgono in tutti i documenti dello schema elettrico.
All'interno di una stessa pagina si sono anche utilizzati simboli di collegamenti chiamati 'intra-page', ossia che collegano due punti all'interno della stessa pagina e collegati a tali simboli, nominando allo stesso modo la net collegata ed esso nei due o più punti collegati insieme.
Inoltre, data la possibilità di creare lo schema elettrico diviso su diverse pagine, per sistemi più complessi esistono simboli per la connessione dei punti a loro collegati su diverse pagine, oltre ai

simboli generici nominati in precedenza, chiamati ‘inter-page’. Questo avviene sempre nominando allo stesso modo la net a loro collegata.

Lo schema elettrico della scheda Payload è diviso su 4 pagine.

La prima pagina contiene i due simboli del connettore J0 da 140 pin, sia quello utilizzato per il lato superiore della scheda (socket), sia quello utilizzato sul lato inferiore (plug). Essendo i segnali passanti all’interno del connettore, ad ogni pin dei due simboli vi sono simboli di connessione ‘intra-page’ con il nome della net corrispondente, tranne come spiegato nel caso del segnale TDI del JTAG. Inoltre, ai segnali utilizzati dalla scheda Payload di tale connettore sono collegati simboli di connessione ‘inter-page’ con il nome della net corrispondente, che permettono ai segnali disegnati sulle altre pagine chiamati nello stesso modo e collegati anch’essi a simboli ‘inter-page’, di collegarsi al connettore J0. Questi simboli inoltre indicano se il segnale è uscente o entrante alla net a cui sono collegati, per un controllo della logica delle interconnessione, per facilitare di trovare eventuali errori presenti sullo schema, nel caso per esempio di due net collegate a due simboli di output o di input, invece di essere uno di input e uno di output.

La seconda pagina dello schema elettrico contiene lo schema principale della scheda Payload. Infatti questa pagina include tutti i componenti principali, quali il processore, le tre memorie, il decoder video, i tre loadswitch, il generatore del power on reset, il regolatore LDO per la tensione di 1,8V per il core del decoder video, i due quarzi con i loro condensatori di carico, i tre gate AND e un gate OR, i componenti discreti per il regolatore buck per il core del processore ,i componenti discreti quali resistenze di pull-up o pull down e condensatori di uscita e d’ingresso per i due regolatori di tensione e tutte le loro interconnessioni. Anche qui tutti i segnali delle interfacce del sistema, quali UART, SPI, JTAG, Test, Video sono collegati a simboli di connessioni ‘inter-page’ per il collegamento con i segnali del connettore J0.

La terza pagina contiene tutti i condensatori di bypass per tutti i componenti integrati digitali presenti sulla scheda:

- Processore Blacfin BF-532: la linea di alimentazione da 3,3V prevede 14 pin d’ingresso sparsi su tutto il componente e quindi ci sono 14 condensatori di bypass, di diverso tipo per avere una maggiore copertura in frequenza. Sono stati impiegati 3 condensatori da 10 μF, 4 condensatori da 10 nF e i restanti 7 da 100 nF.
- Processore Blacfin BF-532: la linea di alimentazione per il core prevede 8 pin d’ingresso sparsi su tutto il componente e quindi ci sono 8 condensatori di bypass, di cui 4 da 10 nF e 4 da 100 nF.

- Decoder video: la linea di alimentazione per il core è fornita mediante 3 pin e quindi sono collegati 3 condensatori da 100 nF mentre l'alimentazione per gli I/O è data mediante un solo pin e quindi vi è un solo condensatore di bypass da 100 nF tra l'alimentazione e massa.
- Memoria SDRAM: possiede 7 pin per l'ingresso della tensione di alimentazione quindi sono stati inseriti 7 condensatori da 10 nF
- Sono stati posti un condensatore da 100 nF di bypass sull'unico pin d'alimentazione della memoria SRAM, FLASH e generatore di power on reset, mentre sull'unico pin d'alimentazione dei gate AND, OR sono stati inseriti un condensatore da 10 nF per ciascuno.

Tali condensatori di bypass sono collegati ai simboli generici ValPay33 e Gnd, oltre a quelli del core del decoder collegati al Vdd, quindi i diversi condensatori fino al livello di schema elettrico sono solo logicamente collegati ai diversi componenti appena descritti mentre fisicamente possono collegarsi a qualsiasi punto di massa o di alimentazione a 3,3V, non quindi ai pin dei componenti, ad eccezione sempre di quelli per il core del decoder che essendo gli unici ad essere collegati a Vdd tranne l'uscita dell'LDO e i tre pin del decoder, che quindi li forza ad essere collegati a loro.
L'ultima pagina contenuta nello schema elettrico della scheda Payload è il sensore di temperatura, con i suoi ingressi e la sua uscita analogica, portati sempre attraverso simboli di connessioni inter-page dal connettore J0. Si deve notare che qui è l'unico punto dove cine impiegata la massa analogica GndAnal.
Dopo aver disegnato tutte 4 le pagine, mediante il comando Verify il software consente di riscontrare la presenza di errori nelle connessioni e sulla loro consistenza logica: l'elenco di tali errori è visibile nella finestra di output, nella sezione relativa alla verifica.
Il passo successivo alla verifica `e la compilazione del progetto attraverso il CDBC (common database compiler). Durante questa fase, vengono create le netlists, contenenti i dati relativi alla connettività strutturale e gerarchica del design. I dati da essa generata vengono memorizzati in un file il cui nome viene specificato dall'utente (attraverso la finestra di dialogo CDB Compiler), mentre l'esito, insieme agli eventuali errori, può essere visualizzati nella finestra di output sotto la sezione Command. Gli errori riscontrati durante la compilazione sono dovuti ad un'incorretta costruzione degli schematici in esame.
L'ultima operazione da eseguire per poi passare alla realizzazione del PCB è quella di impacchettamento (packaging) del progetto viene effettuata al fine di mappare i simboli logici nelle rispettive celle fisiche. Infatti, il packager rileva tutte le informazioni relative ai simboli e alle reti del progetto dal file generato durante la compilazione, localizza le informazioni sulle corrispondenti

celle fisiche nel PDB (parts database) e le assegna ai simboli riportati nello schematico. I Reference Designator e i numeri dei pins vengono assegnati al simbolo secondo quanto specificato nella corrispondente parte all'interno del PDB.
Ora si è pronti per la realizzazione fisica della scheda.

3.3 PCB

La realizzazione del PCB è stata eseguita mediante il software Express PCB. Dopo averlo collegato alla libreria della scheda Payload e al suo schema elettrico, si può iniziare seguendo i seguenti passi:

1. disegnare il contorno della scheda
2. posizionare i componenti
3. eseguire il routine delle piste
4. verifica di tutte le interconnessioni e dei parametri di configurazione

3.3.1Vincoli strutturali

Il circuito stampato della scheda ProcA deve rispettare dei vincoli di dimensioni a causa dello spazio a disposizione dentro al satellite e di come sono state posizionate le scheda al suo interno. Nella figura seguente sono riportate le dimensioni fisiche dello stampato:

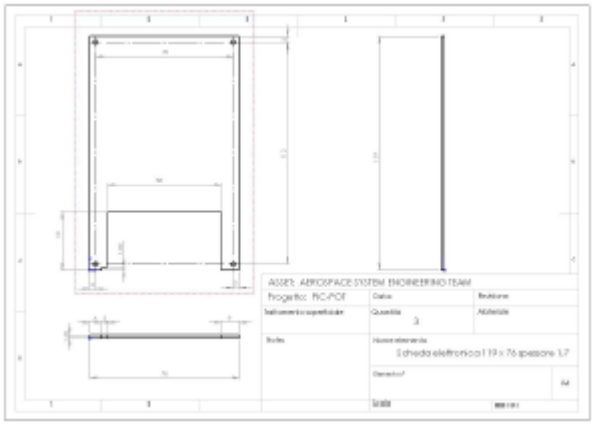


Figura 3.1. Layout della scheda Payload

Infatti si può notare la disposizione dei fori per le viti di fissaggio ed il posizionamento della resega per la collocazione del motore e della tre videocamere.

3.3.2 Posizionamento dei componenti

Lo stampato si sviluppa su 4 strati, due di segnale, lo strato 4 sul lato inferiore della scheda su cui sono stati posizionati tutti i condensatori di bypass e lo strato 1 sul lato superiore della scheda su cui sono stati posizionati tutti i restanti componenti, un piano di massa, lo strato 2, e un piano d'alimentazione per la tensione 3,3V, chamato strato 3.

Il primo componente che è stato posizionato è il connettore J0. Esso ha una posizione fissa, prevista a livello di sistema PICPOT, in quanto deve essere fissa in tutte le schede posizionate una dietro l'altra, incastrate mediante tale connettore e tenute fisse nel satellite mediante quattro guide metalliche. Il connettore è stato posizionato in una posizione molto defilata, vicina al bordo destro dello strato superiore (1) della scheda, ma non troppo dato che bisognava lasciare la possibilità a tutti i segnali di uscire dal lato esterno del connettore ed andare verso gli altri componenti. Questo non era un problema della scheda Payload, la quale utilizza di più segnali mappati sul lato interno del connettore, ma di altre scheda come la PowerSwitch.

Il secondo componente posizionato è il processore, che essendo quello che si interfaccia direttamente al connettore e a quasi tutti gli altri componenti integrati, risultava il più delicato da inserire successivamente.

Dopo è stato inserito il decoder video, che sebbene sia di dimensioni notevolmente ridotte, si voleva che fosse inserito con gli 8 pin del segnale video digitale di fronte alla porta PPI del processore, in modo da facilitare il routing e ridurre la lunghezza delle interconnessioni.

Numerosi tentativi sono stati necessari per posizionare le tre memorie in modo da favorire un routing dei segnali ragionevole. Così i tre integrati piuttosto ingombranti sono stati posizionati di fronte ai due lati del processore dove si trovano il bus indirizzi e quello dati e tutti i segnali del controller SDC e AMC.

Infine sono stati posizionati i gate AND vicino alle rispettive memorie e ai pin di boot, il componente del reset di fronte al rispettivo pin del Blackfin e i restanti componenti discreti, quali resistenze e condensatori vari e il sensore di temperatura NTC, il più vicino possibile al processore, in modo da sentire maggiormente la sua temperatura.

Per ultimi sono stati posizionati i condensatori di bypass nello strato 4, più vicino possibile ai pin d'alimentazione dei componenti integrati, in modo tale da ridurre l'induttanza dei collegamenti e i relativi picchi di corrente possibili.

3.3.3 Routing

Lo step più importante è quello del routing. Infatti è fondamentale impostare correttamente tutte i parametri di route, al fine di non avere piste troppo vicine o troppo sottili o via di dimensioni sbagliate, non supportate poi dal costruttore del PCB.
La larghezza tipica utilizzata per le piste della scheda Payload è di 0,2mm, ma nei casi limite la larghezza minima utilizzata è di 0,15mm. Quest'ultima dimensione è anche quella che è stata fissata per le distanze minime tra i via e i pad dei componenti, tra due pad, tra i pad e le piste, tra le piste e i via e tra due via.
Dapprima sono state tracciate a mano e fissate, ossia rese non modificabili dal software durante il routing, tutte le piste che collegavano i condensatori di bypass sul lato 4 ai pin d'alimentazione dei componenti integrati. Questo perché nello schema elettrico tutti i condensatori erano solo collegati al piano a 3,3V e a quello di massa, non ai pin dei componenti direttamente, ma essendo anche questi ultimi collegati all'alimentazione, posizionandoli a mano si è raggiunto lo scopo di averli connessi ai componenti in modo molto ravvicinato.
Come secondo step, si è settata una gerarchia delle connessioni da tracciare per il router. Infatti si è preferito far tracciare subito le piste del segnale video a 27 MHz, le quali risultavano essere potenzialmente le più sensibili ai ritardi, se ci fossero stati grandi differenze di lunghezze tra le 9 piste.
Infine sono state fatte tracciare tutte le restanti connessioni e realizzare i due piani presenti nella scheda, quello di massa e quello d'alimentazione, in modo da avere tutti i via corretti tra tutti i piani.
Per verificare se tutte le piste tracciate durante il routine sono corrette e rispettano tutte le distanze minime inserite nei parametri di configurazione, si esegue il controllo DRC (Design Rule Check). I controlli che esegue sono due:

- controlli di *Prossimità*: verifica che siano state rispettate tutte le clearances specificate dapprima nei parametri di routine
- controlli di *Connettività*: verifica che le tracce ed i piani connettano effettivamente i pin facenti parte della stessa rete definita nello schema elettrico e che non vi siano hanger, cioè tracce flottanti.

Infine gli ultimi tre step da eseguire per avere il PCB pronto per essere spedito al produttore sono:

- generazione del file per le macchine perforatrici che effettueranno i fori (Drill) sul PCB, chiamato NC Drill

- generazione del Silkscreen, mediante il Silkscreen Generator, ossia del lato superiore dello stampato ove vi saranno tutte le scritte e i Reference designator dei componenti. Infatti prima di fare ciò, si deve controllare che essi non siano sopra qualche via sullo stampato che li rendereberro illeggibili. Se avviene ciò, si passa in draw mode e si spostano manualmente
- generazione del file Gerber utilizzato dalle macchine della ditta produttrice di PCB per tracciare fisicamente il PCB, mediante il Gerber Output del Gerber Tool, il quale permette anche di eseguire una verifica finale del PCB.

3.4 Saldatura

Tutta la scheda Payload è stata saldata manualmente mediante un classico saldatore da 50W, a temperatura variabile. La saldatura è stata effettuata per i componenti discreti mediante lo stagno veduto sotto forma di filo, di diametro molto fine, mentre per i componenti integrati, soprattutto per quelli con passo molto piccolo, è stata impiegata una pasta di stagno e lussante; questa pasta veniva letteralmente spalmata sui pad da saldare disegnati sul PCB mediante una scatolina o una pinzetta, poi veniva posizionato sopra di essi il componente e, appoggiando la punta del saldatore tra il pad e il pin, la metallizzazione si scalda e la pasta diventa stagno liquido che si avvolge al pin sulla metallizzazione. Mettendo una quantità modesta di pasta, una volta fissato il componente su qualche punto, si può passare il saldatore tra i pin, i quali si scaldano e rendono la pasta stagno liquido il quale, grazie al flussante presente al suo interno, è attirato verso il pin, ossia la fonte di calore e non si sperde creando cortocircuiti tra due pin, che è facile che si creino se la quantità di pasta spalmata sul pad è troppa.
Il primo componente che è stato saldato è il processore, le memorie e il decoder video che sono quelli a passo più fine, poi tutti i restanti componenti sul lato superiore, tranne il connettore J0.
Dopo è stato saldato tutto lo strato inferiore, con i condensatori di bypass.
Infine è stato saldato il connettore, il quale è stato lasciato per ultimo data la sua altezza ripetto agli altri componenti e essendo composto da due parti, una sul lato superiore e uno sul lato inferiore.

3.5 Scheda espansione J0

Dopo aver realizzato la scheda Payload, si poneva il problema di come accedere ai segnali d'interfaccia della scheda, da che tutti erano connessi al connettore J0, estremamente scomodi da raggiungere magari con dei fili dato il passo veramente minimo tra un pin e l'altro.
Si è quindi pensato di realizzare un'altra scheda, che fungesse da espansione su 4 connettori BERG dei segnali portati dal connettore J0; tali connettori sono quelli rettangolari a due file di pin da 13 ciascuna, utilizzati per i cavi flat di segnale dei PC o schede embedded.
Si sono quindi smistati i vari segnali, cercando di raggruppare i segnali per le diverse schede e i segnali del connettore di test. Inoltre su ogni connettori sono stati messi i 5 segnali d'alimentazione diversi (12V motore, 12V videocamere, 3,3V Payload, 3,3 ProcA e 3,3V ProcB) e le due masse (una digitale e una analogica), così da averle sempre a disposizione sui fili provenienti da qualsiasi connettore. Infatti la comodità dei cavi flat è la possibilità si staccare uno o più fili sui 26 per portarli all'alimentatore o ad altri connettori possibili.
La scheda è stata adattata ovviamente alle esigenze della scheda Payload. Infatti rispetto alla scheda prototipo, la scheda definitiva risulta priva del driver RS-232, per poter utilizzare la porta UART per debug, inutile però al sistema PICPOT montato. Si è quindi posizionato tale driver su questa scheda con i suoi condensatori esterni, e il segnale RX e TX sono stati portati ad un connettore dei 4, accessibile quindi mediante il cavo che lo collega al PC. Inoltre dato che questo driver 232 presentato in precedenza, prodotto da Analog Devices chiamato ADM3202 è composto da due buffer in ricezione e due in trasmissione. Allora dato che l'UART è utilizzata per connettere la scheda Payload al ProcA, il segnale RX per una è il TX per l'altra e quindi si sono collegati i due segnali provenienti dal J0 anche ai due buffer opposti e mandati ad un altro connettore, così che anche il ProcA potesse utilizzare l'interfaccia seriale RS-232 per debug.
Come spiegato in precedenza, il connettore J0 porta i 4 segnali JTAG standard per una potenziale programmazione a satellite chiuso, ma l'emulatore dell'Analog Devices da noi utilizzato per la programmazione e il debug del sistema ne prevede altri due. Per tale motivo, è stato messo sulla scheda espansione J0 il connettore del cavo dell'emulatore di 14 pin, con i quattro segnali standar provenienti dal J0 e i due segnali in più provenienti da un connettore piccolo a due pin. Uno stesso connettore così è stato posizionato sulla scheda Payload in modo da poter accedere a tali segnali e collegare le due schede mediante due fili. In questo modo ci si garantiva la possibilità di programmare la scheda Payload, nel caso non si fosse riusciti mediante la catena JTAG del connettore di test, data la mancanza dei due segnali usati dall'emulatore per il Blackfin.

Infine è stato ricavato un connettore BERG a 10 pin per le videocamere, dove sono stati riportati il segnale video e le tre alimentazioni distinte delle videocamere, mentre i restanti sono segnali di massa, che ricordiamo sono richiesti 2 per ciascuna videocamera. In tal modo si è potuto utilizzare un connettore solo per le videocamere, in modo da poter saldare insieme il segnale video senza problemi.

La scheda espansione J0 è stata quindi realizzata nel modo descritto in questo capitolo e mediante gli stessi software. Qui però troviamo connettori di tipo through hole e quindi la saldatura risulta più semplice, perché una volta posizionati i connettori e girata la scheda, si tratta solo di saldare alle piazzole i piedini che spuntano dai buchi mediante lo stagno a filo. Qui si preferisce un saldatore di minor precisione ma maggiore potenza per scaldare velocemente il piedino e fondere lo stagno alla base nel minor tempo possibile.

Capitolo 4

Progetto Software

4.1 Diagramma a stati

Il progetto della parte software del sistema Payload, inizia da un'attenta analisi delle specifiche, per avere un'idea precisa delle funzioni che la scheda deve adempire all'interno del sistema PICPOT. Infatti come spiegato nel capitolo 2, la scheda Payload s'interfaccia alla scheda ProcA e alla scheda ProcB, dalla quale riceve dei telecomandi di diverso tipo, i quali comportano diverse possibilità di funzionamento della scheda, chiamati anche 'casi d'uso' nella sintassi UML. Essi sono i seguenti:

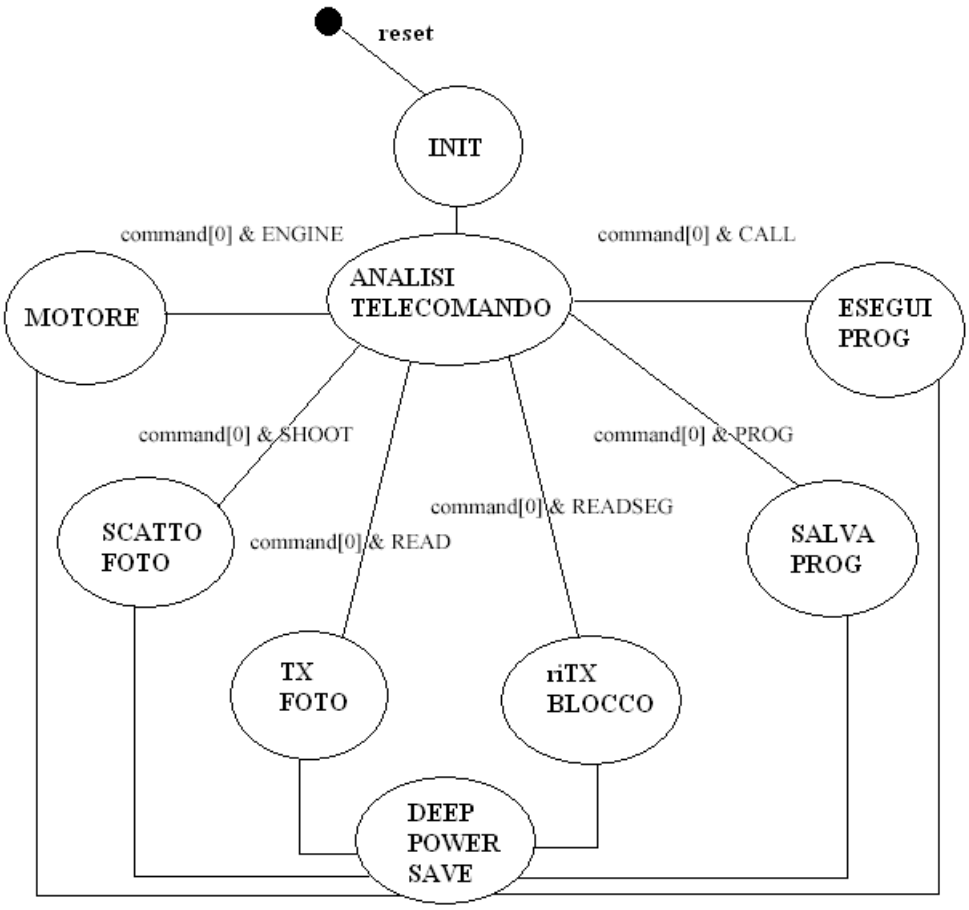


Figura 4.1: Diagramma a stati scheda Payload

Il software è stato quindi progettato sulla tipologia a macchina a stati, che prevede stati comuni a tutte i casi d'uso, quali quello d'inizializzazione e quello di analisi del telecomando. A seconda poi

del telecomando ricevuto, le modalità di funzionamento possibili sono sei, dove la scheda svolge uno dei comandi richiesti da uno dei due processori, ricordando che l'accensione e il campionamento del sensore di temperatura è gestito dai due processori indipendentemente dal sistema Payload.

4.2 Descrizione generale

Le operazioni che deve eseguire la scheda sono le seguenti:

- ricevere un telecomando da parte del ProcA e del ProcB
- acquisire un fotogramma da una delle videocamere, dividerla in 9 blocchi, comprimerli in formato JPEG e salvarli
- trasmettere tutti e 9 i blocchi corrispondenti ad una certa foto alle schede ProcA e ProcB
- ritrasmettere un certo blocco relativo ad una delle 5 possibili immagini salvate in memoria
- salvare un programma utente mandato dalla stazione di terra
- eseguire il programma utente salvato
- permettere l'utilizzo del motore per il controllo del momento d'inerzia da parte del ProcB

E' facile ritrovare tali operazione nel diagramma a stati in figura 3.1.

Il flusso di esecuzione del programma principale inizia una volta avvenuta l'accensione della scheda, dove il processore campionerà i boot mode pin e eseguirà il boot.

L'unica modalità di boot finora testata è la modalità 01, ossia quella che parte leggendo il binario del programma dall'indirizzo iniziale della memoria FLASH e lo trasferisce sulla sua memoria interna veloce per poterlo eseguire.

Seguirà poi l'esecuzione del codice dall'indirizzo iniziale, specificato nei primi byte acquisiti dalla memoria FLASH, dove vengono svolte le diverse inizializzazioni:

- sulle frequenze di funzionamento del processore
- sulle tempistiche del controller per memorie asincrone
- sulle tempistiche del controller per memorie sincrone
- sulle due porte di comunicazione UART e SPI.

Infine viene impostata la Interrupt Vector Table, in modo da rendere sensibile il sistema ai seguenti interrupt:

- interrupt generato dal DMA controller al termine dell'acquisizione di un fotogramma
- interrupt generato dal controller UART quando il buffer di ricezione è pieno
- interrupt generato dal controller UART quando il buffer di trasmissione è vuoto

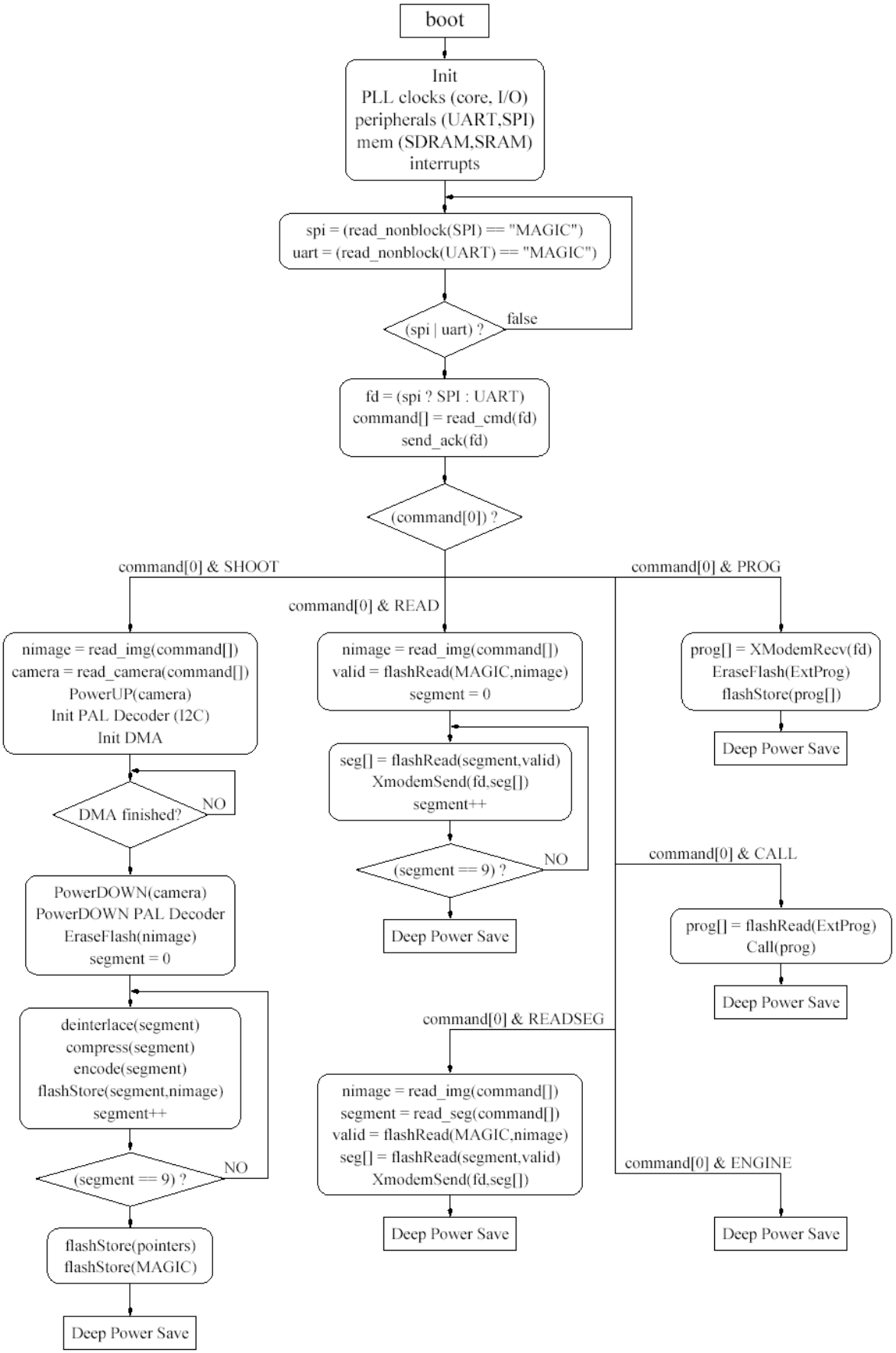


Figura 4.2: Diagramma di flusso

Il funzionamento della porta UART e SPI avviene mediante interrupt, in questo modo il sistema è più efficiente nelle tempistiche perché non deve rimanere in polling sui registri dei rispettivi flag.
Il processore, passata questa fase d'inizializzazione, si sveglia alla ricezione di un interrupt causato dall'arrivo di un telecomando ad una delle due porte. Come si vede dalla figura 3.2, il processore lo legge, memorizzandolo su una variabile qui chiamata *command[]*, che sarà quindi la variabile di stato del programma così organizzato; infine il processore trasmette un acknowledge al processore che ha trasmesso il telecomando. A seconda del valore della variabile di stato, il processore eseguirà task diversi, che verranno descritti in seguito.
Il seguente diagramma di flusso descrive in modo chiaro come avvengono sia la fase d'inizializzazione e di analisi del telecomando, sia i vari task richiesti dai diversi stati del programma; al termine di ogni stato è previsto che il processore vada nella modalità di profondo risparmio energetico, qui chiamato Deep Power Save, ma utilizzando la terminologia del processore viene detto HIBERNATE, dove il core del processore e la sua alimentazione sono fermati e sono solo alimentate le periferiche statiche.

4.2.1 Scatto Foto

Lo stato principale dei tre è denominato SCATTA FOTO.
All'interno di questo stato vi sono numerosi task diversi, come si può vedere dalla figura 3.2. Infatti dall'analisi del telecomando, il processore capisce da quale videocamera deve acquisire un fotogramma e con che numero identificarlo, dato che sarà possibile acquisire e salvare al massimo 5 foto e l'identificazione sarà necessari per la richiesta di trasmissione. Inoltre verrà anche impostata la memoria di destinazione dell'immagine, in corrispondenza del processore che ha richiesto l'operazione, in modo da conoscere la memoria utilizzata e poterne valutare il differente comportamento in ambiente spaziale, valutando l'eventuale presenza di errori nell'immagine.
Successivamente all'analisi del telecomando, verrà quindi accesa la videocamera mediante uno dei tre pin general purpose PF3, PF4, PF5 che pilotano i loadswitch sulle loro linee di alimentazione; in questo modo, mediante la porta OR esterna, il decoder video uscirà dallo stato di standby e entrerà nella fase dire reset, da qui sarà tolto portando a livello logico alto il pin general purpose del processore, chiamato PF8.
Appena il decoder video si troverà nel suo stato di normale operatività, avverrà la programmazione dei tre registri spiegati in precedenza, mediante il protocollo I^2C implementato sui due pin PF10 e PF11. Quindi andrà in attesa dell'interrupt causato dall'aggancio del PLL del decoder video al

segnale PAL delle videocamere, dato mediante il pin PF7. Questo segnale indica che la catena hardware video è pronta per l'acquisizione del fotogramma.

A tale scopo deve essere impostato il DMA controller, con i parametri visti nel capitolo precedente, qui di seguito descritti in modo più dettagliato:

- Data Packing, ossia acquisizione a 8 bit ma salvataggio ogni 2 byte messi insieme su una parola di 16 bit, parallelismo delle memorie esterne
- Stop Mode, ossia acquisizione di un singolo fotogramma e generazione di un interrupt al termine dell'operazione
- DMA a 2 dimensioni, ossia 2 contatori distinti, uno per le righe incrementato al termine di quello per le colonne, ideale per questo tipo di applicazioni
- Indirizzo iniziale della memoria destinazione del fotogramma

Infine dopo aver abilitato il DMA controller, anche la porta video PPI verrà settata con i seguenti parametri:

- Parallelismo dei dati a 8 bit
- Active video mode
- Input mode
- Ricezione di entrambi i frame (o field pari e dispari)
- Numero di righe attive per frame: 576

Si abilita poi la porta PPI e inizia l'acquisizione dello stream di dati del fotogramma. Come detto in precedenza, circa 40 ms dopo viene generato un'interrupt dal DMA che segnale il termine dell'acquisizione, dopo il quale viene spenta la videocamera e quindi anche il decoder è riportato in standby. Inoltre viene cancellata la locazione della FLASH utilizzata per memorizzare l'immagine appena acquisita; per ogni immagine sono stati riservati 3 blocchi da 64 Kbyte, ossia 192 Kbyte, mappati in modo statico a seconda del numero identificativo dell'immagine mandato mediante il telecomando da 0 a 4.

A questo punto l'immagine viene suddivisa su 9 blocchi e su ogni blocco sarà eseguita l'operazione di de-interlacciamento, compressione e salvataggio sulla memoria FLASH. La prima operazione esegue la fusione tra le righe pari e le righe dispari dell'immagine; essa consiste nel prendere una riga dal frame pari e una riga dal frame dispari e metterle una dopo l'altra, cioè la prima di quello pari rimane prima e la prima di quello dispari diventa la seconda, poi la seconda del frame pari diventa la terza e la seconda di quello dispari diventa la quarta e così via. Al termine di tale operazione otterremo blocchi da 240 x 192 pixel.

Tali blocchi poi vengono compressi con una qualità fissa del 75% e poi vengono salvati sulla memoria FLASH in modo contiguo.

Completate queste operazioni su tutti i 9 blocchi, all'inizio del blocco della memoria FLASH riservato per suddetta immagine, si pone una mappa degli indirizzi di ciascun suo blocco, in modo da poter leggere dalla memoria anche solo uno qualsiasi dei 9 blocchi e non per forza l'intera immagine.

4.2.2 Trasmissione Foto

Una volta ricevuto il telecomando della trasmissione immagine, viene abilitata la porta UART con i parametri spiegati nel capitolo precedente e poi si va nella locazione di memoria della FLASH corrispondente all'immagine richiesta e si porta sulla memoria interna la mappa dei suoi 9 blocchi.
Quindi viene scandita la mappa dei blocchi, partendo dal primo vengono letti i dati dalla memoria FLASH e passati alla porta UART per la trasmissione contemporanea. La trasmissione avviene con un driver spiegato in un capitolo successivo e i dati vengono trasmessi mediante un protocollo seriale standard chiamato XMODEM.

4.2.3 Ritrasmissione Blocco

L'immagine è stata suddivisa in blocchi per evitare la ritrasmissione di un'intera immagine a terra nel caso in cui una parte dell'immagine fosse stata piena di pixel sbagliati. In tal caso la stazione di terra potrebbe quindi richiede la ritrasmissione di un singolo blocco, o diversi blocchi ricevuti con errori.
In questo caso viene sfruttata la mappa dei blocchi salvata all'inizio delle locazioni di memoria in FLASH, in modo da poter acquisire dalla memoria solo i dati riferiti a tale o tali blocchi. La trasmissione avverrà nello stesso modo in cui avviene quella di tutta l'immagine.

4.2.4 Salvataggio Programma Utente

Lo stato di salvataggio di un nuovo programma esterno, come si può vedere dalla figura 3.2, è identificato da un telecomando specifico seguito poi dallo stream di byte che costituisce il nuovo programma da salvare. Il processore prenderà questi dati e li salverà in una locazione fissa e nota

della memoria FLASH esterna, partendo da un indirizzo fisso, come verrà spiegato in seguito con la mappa di memoria della FLASH.

4.2.5 Esecuzione Programma Utente

Lo stato invece di esecuzione del programma salvato viene raggiunto da un differente telecomando; questo genera i seguenti step:

- allocazione dinamica di un'area di memoria interna delle massime dimensioni disponibili, assegnandola ad un puntatore a funzione
- passaggio dei dati del programma da eseguire dalla memoria FLASH esterna a tale locazione di memoria

chiamata di tale funzione per l'esecuzione del programma

4.2.6 Motore

Lo stato più semplice da descrivere è lo stato denominato ENGINE. L'esistenza di tale stato nel programma Payload è causato da scelte di sistema avvenute dopo il progetto della scheda PowerSwitch. All'interno del satellite è presente un motore elettrico brushless alimentato a 12V comandato dal ProcB per la correzione del momento d'inerzia del satellite. La scheda PowerSwitch prevede l'esistenza di un unico alimentatore per i 12V e per i 3,3V destinati alla scheda Payload. Quindi quando la scheda ProcB richiede di alimentare il motore elettrico per eseguire delle manovre, la scheda PowerSwitch attiva tale alimentatore, il quale provoca oltre all'accensione del motore anche quella dell'intera scheda Payload. In questi casi il ProcB dovrà comunicare mediante il telecomando specifico, via SPI al nostro sistema, che l'accensione è dovuta all'utilizzo del motore. Sapendo questo, il processore va in modalità HIBERNAT, dove sono alimentate solo le periferiche statiche, che serviranno a mantenere i loadswitch sulle linee a 12V per le videocamere spenti., permettendo il pilotaggio del motore da parte del ProcB ma riducendo al minimo i consumi da parte della scheda Payload forzatamente accesa.

4.3 Flusso di sviluppo del software

La parte software della scheda Payload è stata sviluppata in linguaggio C mediante il software disponibile presso l'Analog Devices chiamato Visual DSP++; questo ambiente permette di programmare il processore direttamente sulla scheda mediante la porta JTAG e l'emulatore collegato al PC.
Inoltre il programma permette la visualizzazione di tutti i registri del processore, di tutte le aree di memoria sia interna che esterna e diverse funzioni tipiche di debug, quali finestre di watch, inserimento di breakpoint e esecuzione step by step. Infine include la possibilità di visualizzare il file di gestione della memoria utilizzato dal linker, chiamato linker description file (.LDF) e il disassembly del codice.
Il flusso di sviluppo del software può essere diviso in tre fasi:

- Compiling e Assembling
- Linking
- Loading e Splitting

Le diverse fasi sono rappresentate nella figura 4.3.

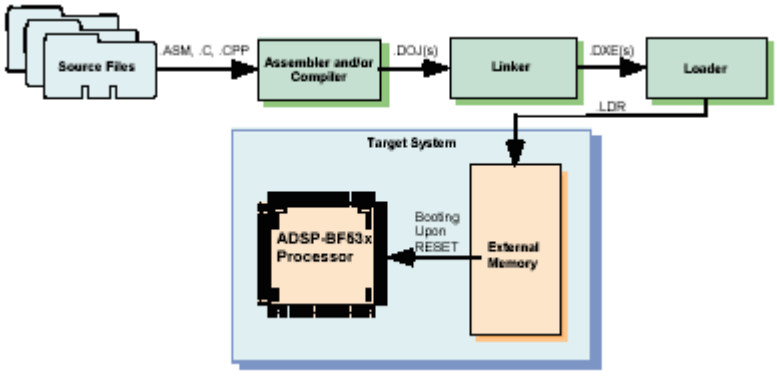


Figura 4.3: Flusso del programma in un sistema stand-alone BF-532

4.3.1 Compiling e Assembling

In questa fase, i file sorgente assembler (.ASM) e C (.C) (passando al formato assembler), portano alla creazione di file oggetto (.DOJ). Tale file è composto da parti chiamate input sections. Ogni input section contiene un particolare tipo di codice sorgente compilato o assemblato, ossia vengono raggruppate le constanti, il codice programma, le variabili. Alcune di tali input sections possono contenere informazioni per il debug del codice sorgente.

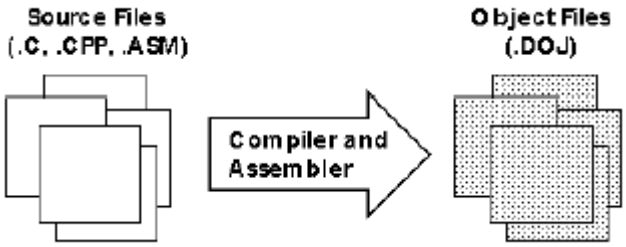


Figura 4.4 : Fase di Compiling

Nella scheda Payload si è sviluppato, come detto in precedenza, tutto il codice sorgente in linguaggio C, tranne il codice di inizializzazione della memoria SRAM, il quale è stato scritto in assembler, data la sua semplicità e l'efficienza richiesta dalla sua posizione che verrà spiegata in seguito. Il compiler e l'assembler agiscono in modo a noi trasparente quindi non è possibile vedere come vengono organizzati i dati e il codice in ogni singolo programma.

4.3.2 Linking

Mediante quanto scritto nel Linker Description File (.LDF), un linker da linea di comando e i la configurazione impostata sul VisulaDSP++, viene letto il file oggetto e prodotto il file eseguibile (.DXE). Il Linker mappa ogni input section (mediante una corrispondente output section nell'eseguibile) in un segmento di memoria, che consiste in un range contiguo di indirizzi di memoria sul processore utilizzato.

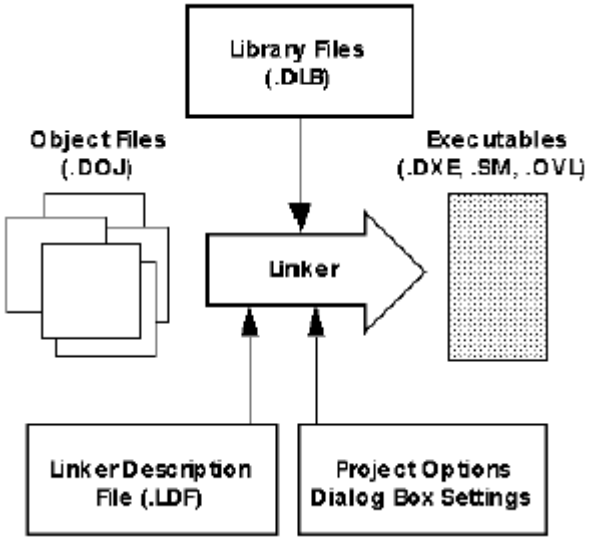


Figura 4.5: Fase di Linking

Ogni input section nel file (.LDF) richiede un nome unico e le principali sono:

- **program** che si riferisce alla sezione dove risiede il codice programma, solitamente mappato sulla memoria codice L1 ma nel nostro caso viene suddivisa tra questa e la memoria SDRAM esterna
- **data1** che si riferisce alla sezione dove risiede i dati globali del programma, come variabili
- **constdata** che si riferisce alla sezione dove risiedono tutte le costanti o le stringhe

4.3.2 Loading e Splitting

Il file eseguibile è processato dall'applicazione elfloader, la quale produce il file (.LDR) chiamato loader file. Questo file sarà quello che permette il boot dalla memoria FLASH, chamato boot-loadable file. Nella figura seguente viene visualizzato come dall'eseguibile si ottiene questo file mediante il loader, che attraverso l'emulatore ICE verrà portato sulla memoria interna del processore per le prove di debug. Nel caso della scheda Payload, l'eseguibile in formato loader era di 338 Kbyte, quindi doveva essere necessariamente messo in una memoria esterna, come la FLASH, per poi essere eseguito mediante un comando di reset. Il software Visual DSP++ mette a disposizione un pacchetto chiamato Flash Programmer che, dandogli il driver per la memoria FLASH utilizzata e il file loader da scriverci dentro, esso convertiva il file loader in binario e lo scriveva sulla memoria esterna utilizzando i registri JTAG.

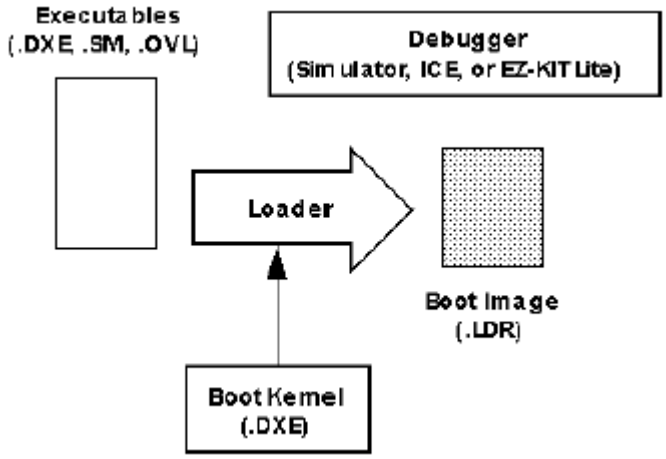


Figura 4.6: Fase di Loading

Il file di loader consiste in un insieme di blocchi precededuti da un'intestazione. Sono queste intestazioni quelle lette e processate dal kernel contenuto nella PROM durante il boot.

L'Analog Devices mette a disposizione un software chiamato Loader Viewer, il quale permette di evidenziare i diversi blocchi di dati del file loader e ne facilita la comprensione, dato che il file è in formato binario:

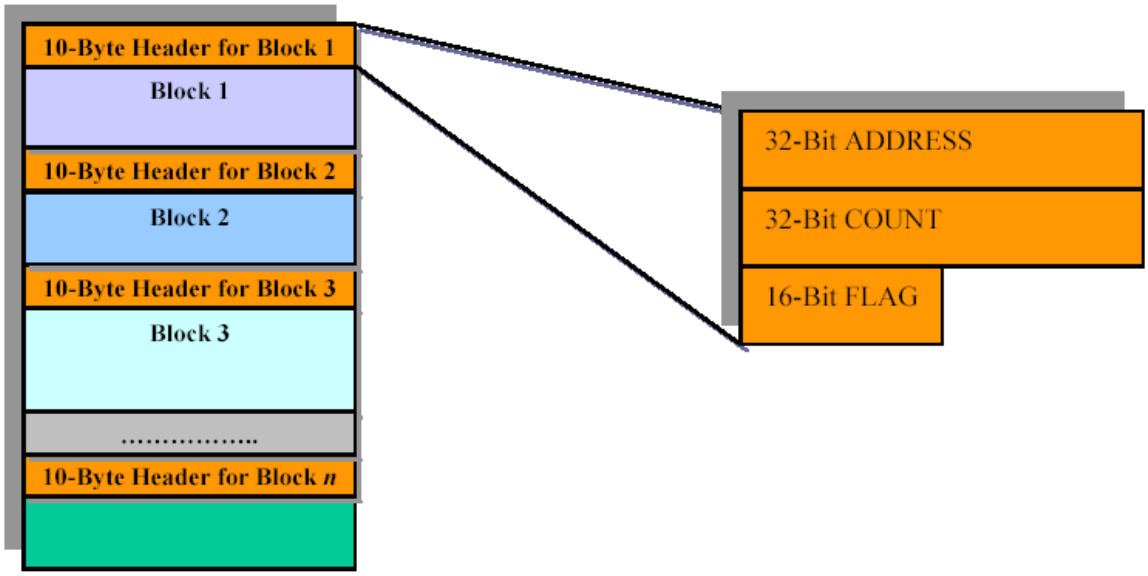


Figura 4.7: Struttura del file Loader(.ldr)

Ogni blocco ha un'intestazione composta da:

- **ADDRESS (4 bytes)** – l'indirizzo destinazione, in cui il blocco sarà scritto sulla memoria
- **COUNT (4 bytes)** – il numero di byte del blocco
- **FLAG (2 BYTES)** – specifica il tipo di blocco e i comandi di controllo

L'ultima sezione dell'header, è quella che definisce se il blocco è un blocco d'inizializzazione, ossia un blocco che dovrà essere eseguito prima del codice dell'applicazione, oppure se è un suo blocco di dati. Inoltre fornisce le informazioni di quale modalità di boot è stata settata, l'indirizzo d'inizio dal comando di reset, detto reset vector, se il blocco di dati è l'ultimo da scaricare, se è da ignorare o considerare e se il blocco è un blocco con dati tutti nulli, utilizzato per inizializzare dei buffer, poi utilizzati dal codice C.

4.4 Mappa di memoria

Il linker, associa a delle sezioni di memoria un range d'indirizzi, che sarà quello utilizzato per accedervi, oltre a definirne il tipo, cioè se RAM ossia leggibile e scrivibile o ROM, cioè solo leggibile. Inoltre associa le input section contenute nel file oggetto ad alcune di queste sezioni di

memoria; tutto questo viene svolto mediante i parametri forniti mediante il Linker Description File (.LDF), il quale fornisce quindi tutta la mappatura della memoria di tutto il sistema che s'interfaccia con il processore BF-532.

La prima parte del file contiene la mappatura della memoria, interna ed esterna. Le memorie interne hanno indirizzi fissi e le memorie esterne asincrone sono state mappate sui banchi di memoria rispettivi, descritti nel capitolo sul progetto hardware, il cui codice è il seguente:

```
MEM_ASYNC3     {  /* Async Bank 3 - 1MB */
      TYPE(RAM) WIDTH(8)
      START(0x20300000) END(0x203FFFFF)
}
MEM_ASYNC2     {  /* Async Bank 2 - 1MB */
      TYPE(RAM) WIDTH(8)
      START(0x20200000) END(0x202FFFFF)
}
MEM_ASYNC1     {  /* Async Bank 1 - 1MB */
      TYPE(RAM) WIDTH(8)
      START(0x20100000) END(0x201FFFFF)
}
MEM_ASYNC0     {  /* Async Bank 0 - 1MB */
      TYPE(RAM) WIDTH(8)
      START(0x20000000) END(0x200FFFFF)
}
```

La memoria sincrona invece è stata divisa in due sezioni:

- la prima, di dimensioni di 256 Kbyte, contiene la parte di codice dell'applicazione che non sta nella memoria interna
- la seconda, ossia la restante porzione di memoria fino a 16Mbyte, è disponibile per l'allocazione dinamica e quindi viene destinata alla sezione chiamata heap

Il codice che mappa la memoria SDRAM è il seguente:

```
MEM_SDRAM0_HEAP{
      TYPE(RAM) WIDTH(8)
      START(0x00100000) END(0x00FFFFFF)
}
MEM_SDRAM0
      TYPE(RAM) WIDTH(8)
      START(0x00000004) END(0x000FFFFF)
}
```

Bisogna ricordare che qui il massimo indirizzamento della memoria specificato al linker è di 16 Mbyte, ma dalla sezione di progetto hardware sappiamo che solo 8 Mbyte possono essere utilizzati, raggruppati in 4 blocchi da 2 Mbyte ciascuno. Questa mappatura "logica" della memoria non è

definibile all'interno del linker, ma deve essere gestita dall'applicazione, utilizzando gli indirizzi definiti nella figura 2.28, come verrà spiegato nel sottocapitolo successivo.

La seconda parte del file contiene l'associazione delle diverse input sections alle sezioni di memoria precedentemente specificate. Alla memoria interna L1 viene associato il codice programma e la configurazione della cache, nel nostro sistema non utilizzata, mediante il seguente codice:

```
program_ram
{
INPUT_SECTION_ALIGN(4)
INPUT_SECTIONS( $OBJECTS(L1_code) $LIBRARIES(L1_code))
INPUT_SECTIONS( $OBJECTS(cplb_code) $LIBRARIES(cplb_code))
INPUT_SECTIONS( $OBJECTS(cplb) $LIBRARIES(cplb))
INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
} >MEM_L1_CODE
```

La sezione *L1_code* permette l'associazione del codice alla memoria L1 e la voce *program* lo associa, mentre *cplb* sta per Cache Protection Lookaside Buffer che sono tabelle che mantengono la configurazione della memoria utilizzata come cache, da noi non utilizzata, ma deve essere obbligatoriamente associata in qualche locazione di memoria. Come si vede dal codice, questa sezione chiamata a piacere *program_ram*, viene associata alla sezione *MEM_L1_CODE*, mappata precedentemente in questo modo:

```
MEM_L1_CODE {
      TYPE(RAM) WIDTH(8)
      START(0xFFA08000) END(0xFFA0FFFF)
}
```

Come già accennato in precedenza, l'indirizzo di start è quello d'inizio della SRAM codice interna da 32 Kbyte. Inoltre è utilizzata per il codice dell'applicazione anche la parte di SRAM interna utilizzabile anche come cache da 16 Kbyte, chiamata *MEM_L1_CODE_CACHE*, mappata dal seguente codice e associata alle stesse input sections:

```
MEM_L1_CODE_CACHE {
      TYPE(RAM) WIDTH(8)
      START(0xFFA10000) END(0xFFA13FFF)
}
```

Alla parte di memoria esterna SDRAM destinata al codice, chiamata *MEM_SDRAM0*, vengono associate le input sections principali, descritte nel paragrafo del Linking, precedute dalla sections *sdram0*, la quale da la possibilità al sistema di utilizzare questa memoria nel caso non sia sufficiente la memoria interna:

```
sdram
{
INPUT_SECTION_ALIGN(4)
```

```
INPUT_SECTIONS($OBJECTS(sdram0) $LIBRARIES(sdram0))
INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
} >MEM_SDRAM0
```

4.5 Organizzazione della memoria

Nel sottocapitolo precedente, è stato quindi analizzato come sono state partizionate e impiegate la memoria interna ed esterna dal processore, nello spazio degli indirizzi.
Ora verrà descritto però come sono state partizionate logicamente la memoria SDRAM e la memoria FLASH.
Quest'ultima infatti è di fondamentale importanza, in quanto è destinata a contenere:

- il codice dell'applicazione, il cui file binario ha dimensioni di 140 Kbyte, quindi saranno dedicati 256 Kbyte
- 5 immagini JPEG, a cui verranno dedicati 192 Kbyte ciascuna, dato che deve essere multiplo di 64 Kbyte (ossia la dimensione di un blocco)

Come già accennato nello spiegare la struttura della memoria, i dati del programma risiedono nel primo blocco, chiamato blocco di boot, dato che che è quello a cui il processore punta all'accensione.
I restanti blocchi della memoria potranno essere impiegati per contenere una versione di backup del codice programma e una Lookup Table per il codice convoluzionale che potrà essere applicato sui dati dei blocchi JPEG. Queste due soluzioni, in fase di progetto della scheda Payload, non erano ancora state decise in modo definitivo a livello di sistema PICPOT quindi non sono state attuate.
La memoria SDRAM invece, come accennato nel sottocapitolo precedente, verrà partizionata in due sezioni distinte. Come detto i blocchi di memoria contigui da 2 Mbyte, sono stati previsti per permettere l'acquisizione di foto consecutive dalle videocamere. Anche questa opzione non è stata sviluppata perché non ancora definita precisamente a livello di sistema.

#	Size (KBytes)	Address Range (x8)	Address Range (x16)	
34	64	1F0000h-1FFFFFh	F8000h-FFFFFh	
33	64	1E0000h-1EFFFFh	F0000h-F7FFFh	
32	64	1D0000h-1DFFFFh	E8000h-EFFFFh	
31	64	1C0000h-1CFFFFh	E0000h-E7FFFh	
30	64	1B0000h-1BFFFFh	D8000h-DFFFFh	
29	64	1A0000h-1AFFFFh	D0000h-D7FFFh	
28	64	190000h-19FFFFh	C8000h-CFFFFh	
27	64	180000h-18FFFFh	C0000h-C7FFFh	FOTO JPEG #5
26	64	170000h-17FFFFh	B8000h-BFFFFh	
25	64	160000h-16FFFFh	B0000h-B7FFFh	
24	64	150000h-15FFFFh	A8000h-AFFFFh	
23	64	140000h-14FFFFh	A0000h-A7FFFh	FOTO JPEG #4
22	64	130000h-13FFFFh	98000h-9FFFFh	
21	64	120000h-12FFFFh	90000h-97FFFh	
20	64	110000h-11FFFFh	88000h-8FFFFh	
19	64	100000h-10FFFFh	80000h-87FFFh	FOTO JPEG #3
18	64	0F0000h-0FFFFFh	78000h-7FFFFh	
17	64	0E0000h-0EFFFFh	70000h-77FFFh	
16	64	0D0000h-0DFFFFh	68000h-6FFFFh	
15	64	0C0000h-0CFFFFh	60000h-67FFFh	FOTO JPEG #2
14	64	0B0000h-0BFFFFh	58000h-5FFFFh	
13	64	0A0000h-0AFFFFh	50000h-57FFFh	
12	64	090000h-09FFFFh	48000h-4FFFFh	
11	64	080000h-08FFFFh	40000h-47FFFh	FOTO JPEG #1
10	64	070000h-07FFFFh	38000h-3FFFFh	
9	64	060000h-06FFFFh	30000h-37FFFh	
8	64	050000h-05FFFFh	28000h-2FFFFh	
7	64	040000h-04FFFFh	20000h-27FFFh	
6	64	030000h-03FFFFh	18000h-1FFFFh	CODICE PROGRAMMA
5	64	020000h-02FFFFh	10000h-17FFFh	
4	64	010000h-01FFFFh	08000h-0FFFFh	
3	32	008000h-00FFFFh	04000h-07FFFh	
2	8	006000h-007FFFh	03000h-03FFFh	
1	8	004000h-005FFFh	02000h-02FFFh	
0	16	000000h-003FFFh	00000h-01FFFh	

Figura 4.8: Mappa di memoria FLASH di tipo Bottom

4.6 Processo di boot

La condizione iniziale affinché si possa procedere con il boot dell'applicazione è che il codice stesso risieda nella memoria FLASH esterna.
Questo obiettivo è stato raggiunto attraverso i seguenti step:

- conversione in binario del formato .LDR mediante un programma scritto in C ed eseguito sul PC dove risiede l'ambiente di sviluppo. La conversione consiste nel passaggio dal formato Intel Hex a binario. A seconda della versione del processore, il file binario sarà diverso. Infatti i DSP della versione 0.2 supportano solo il boot da memorie esterne con parallelismo 8 bit e quindi, dato che il sistema Payload adotta una memoria con bus dati da 16 bit, ogni byte viene inserito un byte di zeri, '0', così che i byte utilizzati effettivamente siano quelli corretti. Mentre nelle versioni 0.3, il processore supporta il boot a 16 bit, quindi il file loader deve essere convertito normalmente e le istruzioni saranno lette a 16 bit sul bus dati
- scrittura del file binario sulla memoria FLASH. Si è quindi realizzato un programma C pilota, il quale viene caricato ed eseguito sulla memoria interna SRAM mediante l'emulato ICE e l'ambiente Visual DSP++. Tale programma eseguiva due operazioni nella seguente sequenza:
    - acquisizione del file binario dal PC mediante la porta UART, mediante il protocollo Xmodem, implementato in linguaggio C come verrà illustrato in seguito, e trasmesso dal PC mediante un terminale per la porta seriale RS – 232. Il programma richiede le dimensioni del file e ne alloca dinamicamente una variabile, mediante una *malloc*, in memoria SDRAM, come descritto nel file .ldf di questo programma, dove la memoria sincrona è usata come heap, in modo da poter trasferire applicazioni anche di dimensioni importanti. Infine richiede anche l'indirizzo di partenza di dove allocare i dati
    - sapendo le dimensioni e l'indirizzo di partenza del file da scrivere in memoria, il programma prepara la FLASH per la scrittura, cioè calcola i blocchi da utilizzare e li cancella
    - infine, partendo dall'indirizzo iniziale inserito, scrive il programma in memoria mediante il driver per la FLASH adottata, scritto per l'utilizzo della memoria stessa. Questo codice è stato scritto partendo da sorgenti C resi disponibili dalla ST Microelectronics, ma ricchi di funzioni superflue al nostro utilizzo. Si veda il sottocapitolo corrispondente per maggiori informazioni.

Questa è stata la procedura impiegata in fase di debug. Inoltre il programma su FLASH aveva anch'esso la possibilità di eseguire tali operazioni, scegliendo questa possibilità mediante un menù iniziale. Così il programma stesso permetteva di scrivere versioni modificate sulla FLASH e poi eseguire il boot da tale programma, dato che l'altro veniva perso in quanto caricato ormai sulla RAM interna ed esterna volatile.

Con il binario dell'applicazione in memoria FLASH, si può quindi procedere con la descrizione vera e propria del procedimento seguito dal boot del sistema all'accensione.

Partendo dalla considerazione che il file binario del software è di dimensioni superiori alla memoria volatile interna disponibile per le istruzioni, di 48 Kbyte, si deve quindi utilizzare una memoria veloce che contenga una parte del codice. Per questo motivo, i pin di boot devono venir settati in configurazione 01, ossia in modalità di boot attraverso la boot PROM interna.

Infatti in questa modalità, all'accensione il codice programma viene processato dal kernel di boot dopo aver campionato i pin di boot, il quale riconosce un blocco di inizializzazione, quello della SDRAM, e i blocchi di codice destinati alla suddetta memoria esterna e alla memoria SRAM interna L1, come visualizzato nella figura 4.9; come detto precedentemente, il blocco di tipo d'inizializzazione verrà eseguito prima dell'acquisizione dell'applicazione; per fare ciò, nel file loader tale blocco vede come indirizzo destinazione l'indirizzo iniziale della memoria codice L1 0XFFA08000, come si può vedere dalla figura 4.10; inoltre viene evidenziato il blocco di dati contenuto nel file, e la sua intestazione, visualizzati mediante il programma Loader Viewer:

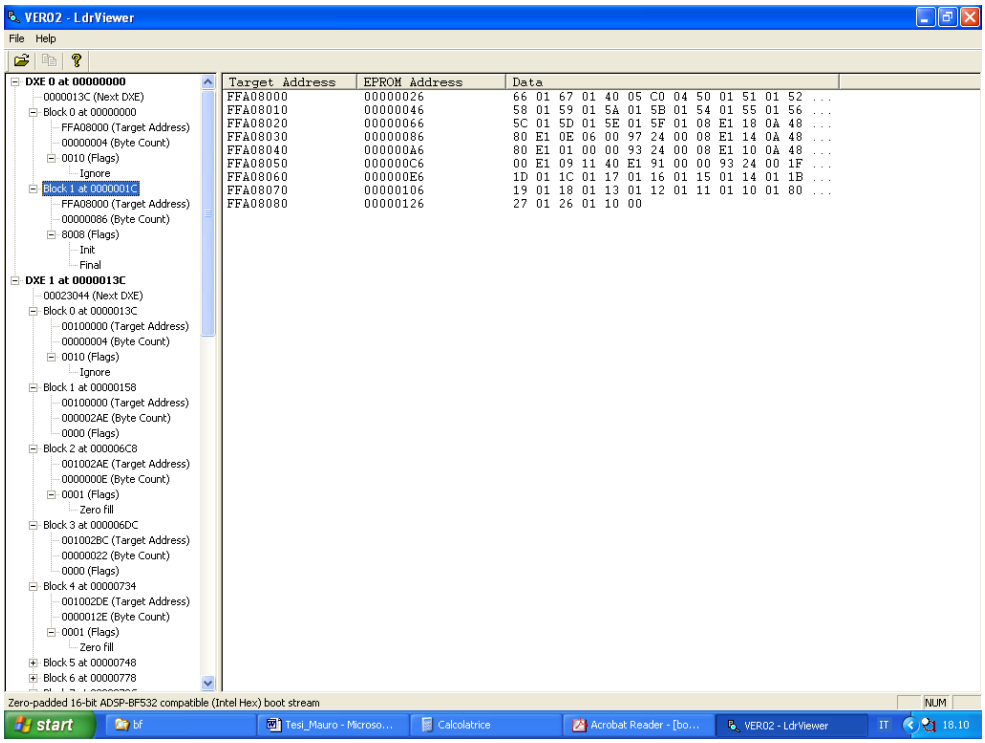


Figura 4.9: Dati inizializzazione SDRAM nel file loader

Questo blocco è quello che setta tutti i parametri di configurazione della memoria SDRAM per averla disponibile prima di scaricare tutto il programma dalla FLASH. In questo modo, il processore dopo aver eseguito il blocco d'inizializzazione, inizia a acquisire i dati programma dalla FLASH, così 48 Kbyte della memoria interna verranno riempiti con le parti descritte precedentemente nella sezione di mappa della memoria, e i restanti saranno messi sulla SDRAM esterna.

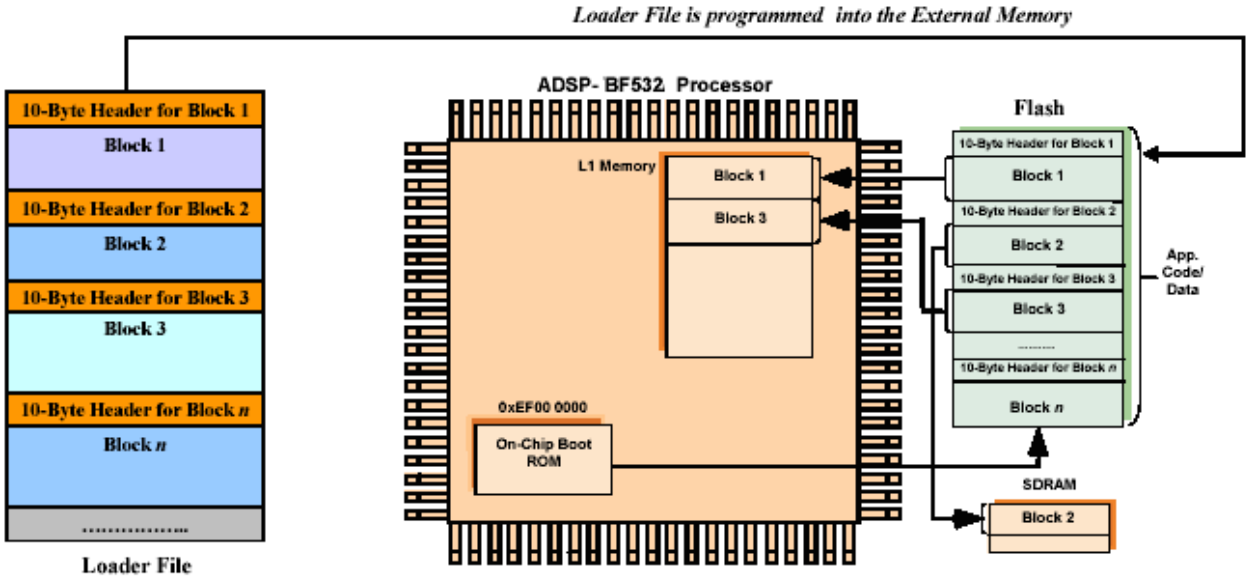


Figura 4.10: Processo di boot del BF-532

Ora il processore inizia ad eseguire i blocchi di codice dell'applicazione.

Con questa configurazione, la memoria FLASH non verrà più utilizzata per il codice dell'applicazione e quindi andrà in standby automatico, mentre la memoria SDRAM esterna dovrà rimanere sempre accesa.

4.7 Descrizione codice

In questa sezione verranno descritte parti delle principali funzioni implementate in C e le principali configurazioni del processore e delle sue periferiche per permettere un corretto utilizzo delle memorie e delle diverse interfacce.

4.7.1 Programmazione e Configurazione DSP e periferici

Il processore attraversa una prima fase dopo quella boot, in cui vengono settati tutti i registri per un suo corretto funzionamento.

Al fine di riferirsi a tutti i registri interni, l'ambiente di sviluppo Visual DSP ++ fornisce due file di libreria .h, dei quali uno contiene la dichiarazione dei nomi dei registri come *DEFINE* associati ai loro indirizzi interni. Nel file successivo sono dichiarate altre *DEFINE* che dichiarano un puntatore ai nomi dei registri, chiamati con gli stessi nomi preceduti dalla lettera *p*. Quindi nel codice dell'applicazione, si è dovuto semplicemente far riferimento a questi puntatori, utilizzando la loro sintassi, per la loro modifica.

Il principale registro da configurare è quello che riguarda il PLL che si aggancia al quarzo esterno, il quale mediante un divisore nell'anello, ottiene poi due linee di clock distinte, una per il core (CCLK) e una per i periferici (SCLK), mediante due ulteriori divisori distinti.

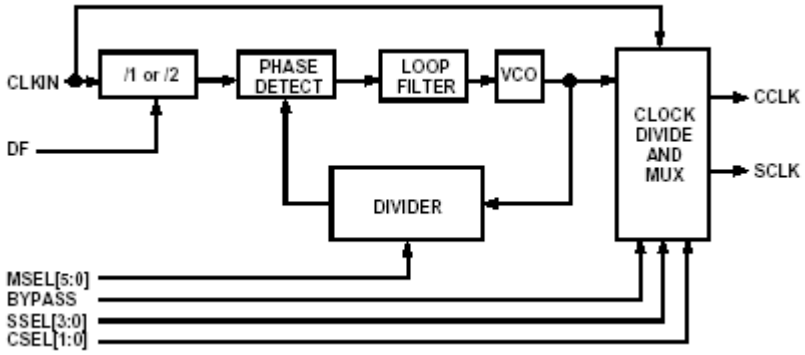
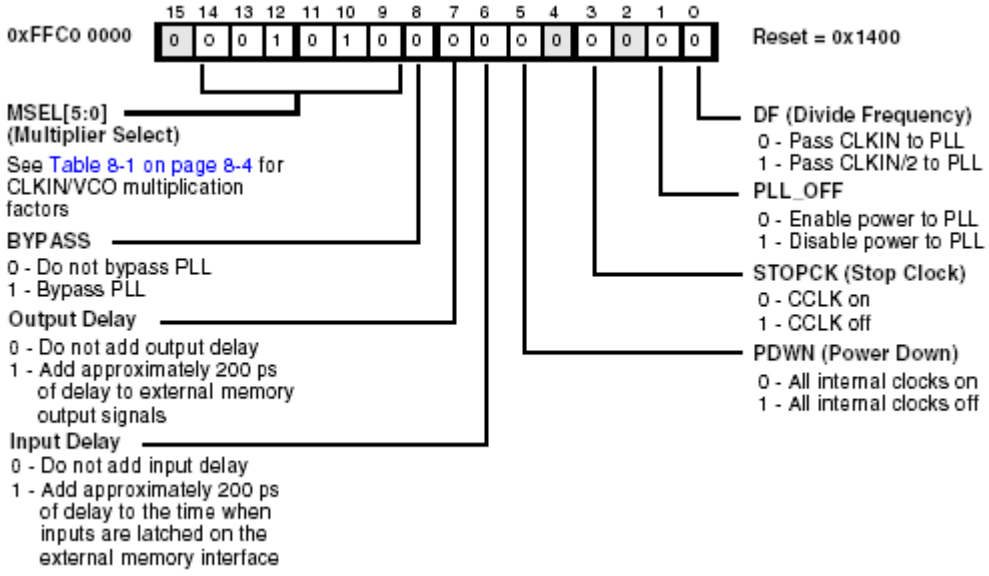


Figura 4.11: Diagramma a blocchi del PLL

Essendo il clock esterno a 11.0592 MHZ, si è settato il divisore d'anello su 36 mediante il registro PLL_CTL, prtando il VCO ad una frequenza di lavoro di 398 MHz. Impostando poi i divisori dei due clock interni entrambi a 4, si è ottenuta una frequenza, nel core e sulle periferiche di circa 99.5 MHz. Questo divisore è ottenuto mediante il registro PLL_DIV. La struttura dei registri è la seguente:



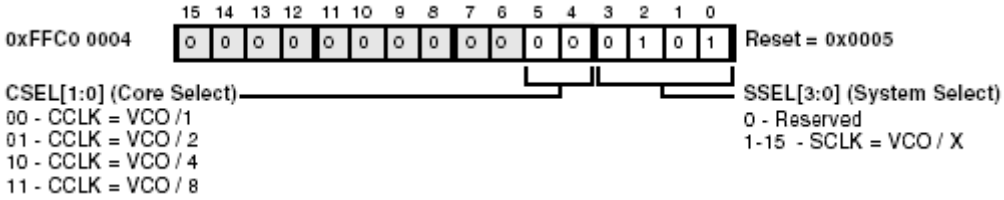


Figura 4.12: I registri PLL_CTL (sup) e PLL_DIV

L'utilizzo di tali registri è un'po particolare. Infatti, i due divisori devono essere settati prima di abilitare l'alimentazione e settare il divisore d'anello mediante il PLL_CTL. Per questo il codice scritto per questo scopo ha questa sequenza:

```
*pPLL_DIV = 0x0024;
*pSIC_IWR = 0x1;
*pPLL_CTL = 0x4800
ssync();
idle();
```

Come si può vedere, viene settato un'ulteriore registro chiamato SIC_IWR, che sta per System Interrupt Wakeup Enable Register. Il primo bit settato configura il PLL Wakeup, coiè il risveglio del sistema dal suo stato di idle alla ricezione di un'interrupt, generato quando il PLL risulta agganciato. Infatti mediante le istruzioni *ssync* e *idle*, si svuota il core istantaneamente dai processi in corso e si lascia il processore in uno stato di nullafacenza da cui si risveglia grazie a questo registro settato.

Senza settare il rispettivo Wakeup flag e passare dallo stato di idle, sono stati settati in questo modo i registri per le diverse periferiche.

I registri delle memorie per esempio, sono fondamentalmente configurazioni di tempistiche dei segnali di controllo, quali write enable, output enable, etc. Per esempio il controller di memorie asincrono, si distingue in due registri da 32 bit, uno che regola il banco 0 e il banco 1, 16 bit per ciascuno, chiamato EBIU_AMBCTL0 a cui è collegata la memoria FLASH e l'altro che regola i restanti due banchi, a cui è collegata la SRAM chiamato EBIU_AMBCTL1. Inoltre un registro di controllo chiamato EBIU_AMGCTL permette l'abitazione dei banchi di memoria. Il codice è il seguente:

```
*pEBIU_AMBCTL1    = 0x33243324;
*pEBIU_AMGCTL     = 0x000F;
```

Il registro della memoria FLASH è lasciato con le impostazioni di default, ossia con le tempistiche più rilassate possibili, in quanto si vuole garantire una comunicazione con questa memoria priv di errori. La struttura dei due registri è la seguente:

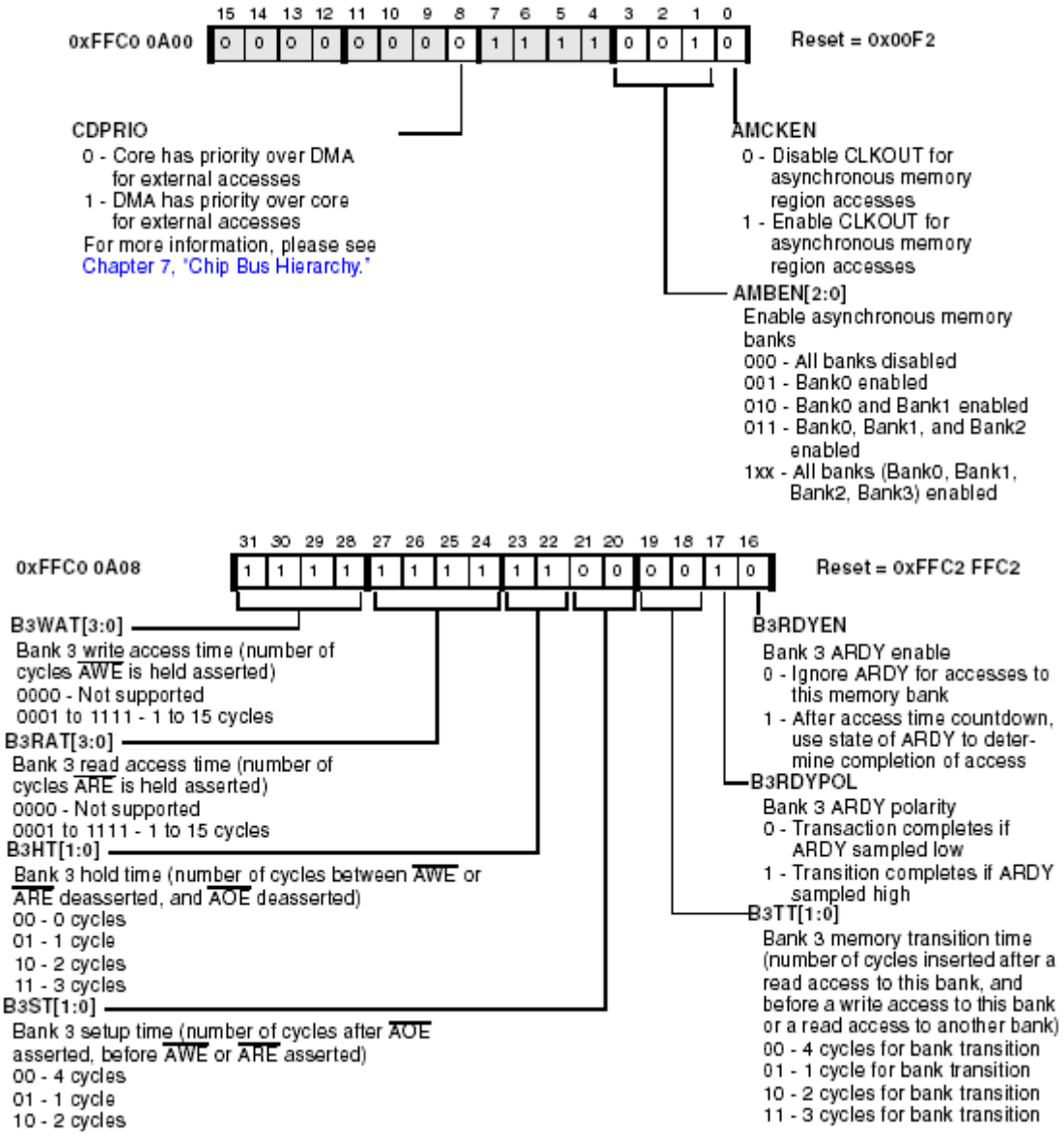


Figura 4.13: I registri EBIU_AMGCTL (sup) e EBIU_AMBCTL1 (lsw)

Come si può notare dai registri sono stati abilitati tutti i banchi in questa situazione e sono stati settate le stesse tempistiche, sia per il banco 2 che per il banco 3, dato che la memoria puntata è la stessa, ossia l'SRAM.

Dall'altro registro invece si può notare come la configurazione della memoria preveda 3 cicli di clock per poter dare il segnale di write enable, e 3 per quello di read, disabilitando l'attesa del wait asincrono. Inoltre sono stati settati 0 cicli di clock di hold ma 2 cicli di setup ed un ciclo di clock di transizione, ossia il tempo atteso tra una scrittura/lettura ed una successiva. La somma di questi ritardi fornisce il tempo impiegato per leggere o scrivere una parola da 16 bit in memoria. Affinché la memoria potesse fungere da destinazione del fotogramma acquisito dalle videocamere, come visto in precedenza, il tempo di scrittura doveva essere inferiore a 74 ns e contale configurazione è 60 ns, dato che a 99.5 MHz il ciclo di clock è circa 10 ns e i cicli d'attesa sono circa 6, sia in lettura che in scrittura, vicino al limite minimo della memoria che è di 55 ns.

Allo stesso modo sono state impostate le configurazioni della porta PPI e UART, mediante i rispettivi registri

4.7.2 Interrupt e API per UART

Un interrupt è un evento asincrono che cambia il normale flusso d'istruzioni all'interno del processore, mentre una exception è un'evento generato dal software quindi sincrono con il flusso di programma. Entrambi sono gestiti da un sistema di eventi a priorità, dove ognuno di questi può essere associato ad una routine diversa che viene attivata con una priorità fissa.
Il processore impiega un meccanismo di controllo per gli eventi a due livelli. Il System Interrupt Controller (SIC) agisce insieme al Core Event Controller (CEC) per controllare e gestire le priorità tra i diversi interrupt. Il SIC provvede alla mappatura tra le diverse possibili sorgenti periferiche di interrupt e gli input a priorità fissa general-purpose del core. Questa mappatura è programmabile mediante un registro chiamato SIC_IMASK, il quale può mascherare una sorgente d'interrupt al SIC, rendendolo insensibile a l'evento generato dalla periferica associata.
Il CEC supporta nove interrupts general-purpose (IVG7 - IVG15), che diventano sette (IVG7 – IVG13), dato che gli ultimi due sono riservati per le interrupt software handlers.

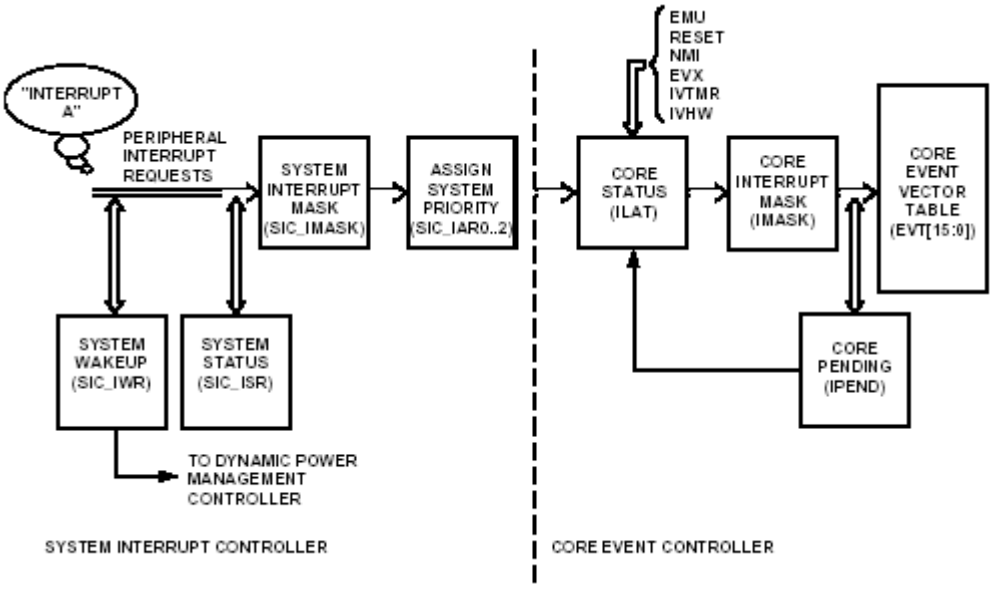


Figura 4.14: Schema a blocchi dell'Interrupt Processing

Ognuno di questi è associato ad un numero identificativo, utilizzato nel registro chiamato SIC_IAR (Interrupt Assignment Register) in cui si associa la periferica all'interrupt general-purpose nel core. Più di una periferica è mappata sullo stesso interrupt del core, e le sorgenti sono messi on OR, senza priorità alcuna.

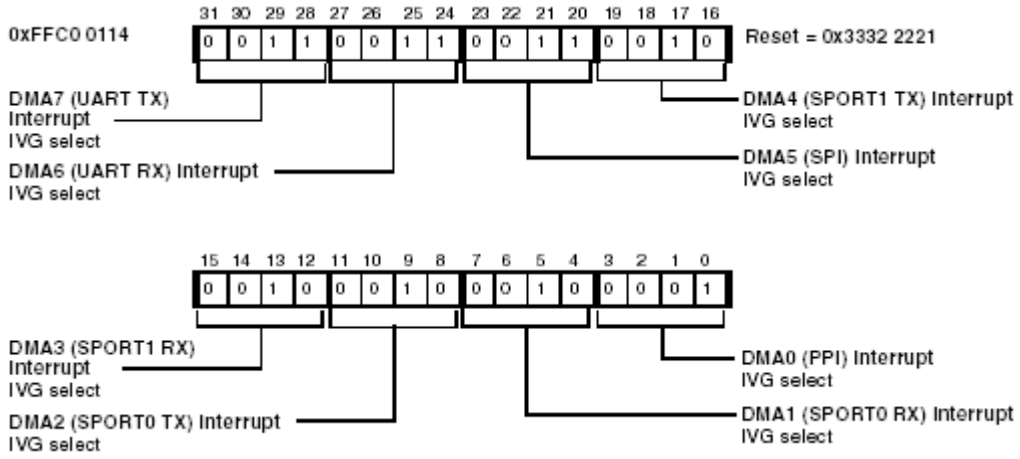


Figura 4.15: System Interrupt Assignment Register 1

Nel programma per il sistema Payload, gli interrupt utilizzati sono stati i seguenti 3:

- DMA 0 generato dal DMA controller in acquisizione della porta PPI in modalità *STOP*, al termine di un fotogramma dalle righe indicate nel registro del controller. Questo interrupt è mappato all'interrupt general-purpose IVG8, il cui core interrupt ID è 1.
- UART RX generato dal riempimento del buffer in ricezione, assegnato all'IVG10, il cui ID è 3.
- UART TX generato dallo svuotamento del buffer in trasmissione, assegnato all'IVG11, il cui ID è 4.

In realtà, i due interrupt sono mappati sullo stesso general-purpose interrupt, cioè l'IVG 10 il cui ID è 3, come si vede dalla figura 4.13. Per poter però riconoscere senza alcun controllo aggiuntivo se l'interrupt è generato dalla trasmissione o dalla ricezione, in modo da rendere la porta UART modulabile con altre funzioni, si è mappato l'interrupt in trasmissione su un altro interrupt input del core non impiegato in altri modi. Il codice che implementa tali eventi è il seguente, che va a scrivere sul registro il valore 0x43000001:

```
*pSIC_IAR1 = (*pSIC_IAR1 & 0x00fffff0) | 0x43000001;
register_handler(ik_ivg8, DMA0_PPI_ISR);
register_handler(ik_ivg10, UART_RX_ISR);
register_handler(ik_ivg11, UART_TX_ISR);
```

La funzione register_handler associata le diverse Interrupt Service Routine (ISR) all'interrupt associato alla periferica.

Come spiegato prima si è poi mascherato tutti gli altri interrupt, lasciando il SIC sensibile solo a questi tre ponendo a '1' i bit associati nel registro SIC_IMASK.

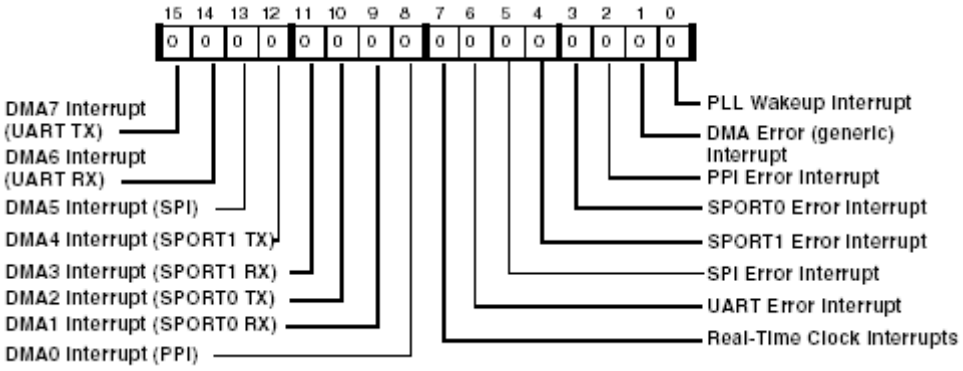


Figura 4.16: System Interrupt Mask Register (parte bassa)

Il codice quindi che maschera tutti gli interrupt possibili tranne questi tre è il seguente:

```
*pSIC_IMASK=0x0000C100;
```

Alla ricezione dell'evento, il processore esegue la funzione interrupt handler associata all'interrupt con maggiore priorità e setta il bit associato in un registro chiamato IPEND, il quale mantiene traccia di tutti gli interrupt del sistema al momento caricati nel processore e diventa zero al termine dell'esecuzione della routine.

Dal lato della periferica, ciascuna possiede un registro degli interrupt generati. All'esecuzione della funzione interrupt handler, tale registro va ripristinato al valore che aveva prima della generazione dell'interrupt, ossia va azzerato il bit che identifica l'interrupt pending, affinché al termine dell'esecuzione della funzione la periferica possa generare un altro interrupt associato allo stesso evento, altrimenti la porta ritiene di averlo già generato e tutto il sistema rimane fermo.

```
temp = *pUART_IIR;
```

Infatti nel caso della porta UART, tale registro si chiama UART_IIR (Interrupt Identification Register) e anche solo l'operazione di lettura di tale registro porta a zero il bit di pending interrupt, più veloce che la scrittura richiesta nel caso dell'interrupt del DMA controller, dove va riportato il registro DMA_IRQ_STATUS al valore iniziale 0x1, che lo riporta al normale funzionamento e potenziale generatore di successivi interrupt.

```
*pDMA0_IRQ_STATUS = 0x1;
```

Gli interrupt per la porta UART sono utilizzati da un'insieme di funzioni per la trasmissione e ricezione di un singolo byte o un certo numero di byte da suddetta periferica. Nella fase di ricezione l'interrupt handler si limita, oltre a riabilitare la generazione degli interrupt, a salvare il dato ricevuto come elemento di un vettore e incrementarne l'indice che lo scandisce, decrementando il conto del numero massimo di byte ricevibili. Il codice è il seguente:

```
uart_rxbuffer[intr_rxubp++] = *pUART_RBR;
intr_rxubp &= (UART_RXBUFFER_SIZE - 1);
```

Le funzioni riportate nella seguente tabella, utilizzano il dato salvato nel vettore e l'indice per svolgere le operazioni di ricezioni in diverse modalità:

Nome Funzione	Argomenti	Scopo	Return
read_uart_byte	-	Ricezione singolo byte	Byte ricevuto
read_uart_blocking	- unsigned char *buffer - int count	Ricezione numero di byte passato e non ritorna fino al loro raggiungimento, attendendoli	Numero di byte ricevuto
read_uart_nonblocking	- unsigned char *buffer - int count	Ricezione numero di byte passato, count, ma ritorna appena non vi siano più dati disponibili senza aver raggiunto il valore passato.	Numero di byte ricevuto

La fase di trasmissione invece impiega le funzioni riportate nella tabella che segue:

Nome Funzione	Argomenti	Scopo	Return
write_uart _ blocking	- unsigned char *buffer - int count	Trasmissione del numero di byte passato	Numero di byte trasmessi
write_uart _ byte	- unsigned char data	Trasmissione singolo byte	-
write_uart_sync	-	Inserisce “nop” in attesa dello svuotamento del buffer per trasmissioni consecutive	-

La funzione write_uart_byte, chiama la funzione write_uart_blocking passandole il dato da inviare e 1 come numero di byte da trasmettere. Quindi la funzione principale è quest’ultima; essa impiega due vettori da 150 elementi di tipo *unsigned char*, che riempie con dei dati da trasmettere e mentre l’interrupt handler di trasmissione riempie il buffer d’uscita dell’UART prelevando da loro i dati ad ogni interrupt, questa funzione riempie il vettore svuotato con dei dati ancora da trasmettere. In questo modo, uno dei due vettori è sempre pieno con 150 byte da trasmettere a disposizione per l’interrupt handler, garantendo un flusso continuo di dati in trasmissione.
In questo stesso modo verrà implementato il driver per la comunicazione mediante la porta SPI, al momento non ancora implementato.

4.7.3 Compressione JPEG e de-Intelacciamento

Dopo l'acquisizione del fotogramma dalle videocamere, l'immagine risulta essere nel formato YCrCb 4:2:2 e interfacciata su due frame, uno chiamato frame pari e l'altro chiamato frame dispari.
Al fine di ottenere 9 blocchi dell'immagine distinti in formato JPEG, gli step seguiti per ogni blocco all'interno di due cicli *for* annidati che scandiscono 9 blocchi con due contatori chiamati X e Y che vanno fino a 3, sono i seguenti:

1. de-Interlacciamento e riscrittura dell'immagine in formato YCbCr 4:4:4.
2. compressione

La prima operazione viene svolta mediante una singola funzione chiamata DeInterleave, a cui vengono passati gli indici X e Y dei due cicli annidati, per sapere che porzione dell'immagine ancora intera e interfacciata considerare. Inoltre vengono passati come argomenti anche due puntatori a due locazioni di memoria precedentemente allocati dinamicamente, uno che punta all'immagine appena acquisita chiamato ACQ_IMAGE_POINTER e l'altro, chiamato IMAGE_BUFFER, che punta all'area di memoria utilizzata come buffer utilizzato poi dal compressore JPEG.
L'operazione di de-interlacciamento consiste nella fusione tra le righe pari e le righe dispari; questo avvine prendendo una riga dal frame pari e una riga dal frame dispari e mettendole una dopo l'altra, cioè la prima di quello pari rimane prima e la prima di quello dispari diventa la seconda, poi la seconda del frame pari diventa la terza e la seconda di quello dispari diventa la quarta e così via.
Questo scopo è raggiunto utilizzando due puntatori *unsigned char*, uno utilizzato per scandire il frame pari e uno che punta al frame dispari, contenuto in un'area di memoria superiore di metà della dimensione dell'immagine (cioè 829440/2 = 414720 che in esadecimale è 0x65400), così implementati:

```
punt_1 = acq_img_pointer + (righe_b * x);
punt_2 = acq_img_pointer + 0x65400 + (righe_b * x);
```

La costante righe_b è 480, che è il numero di pixel da considerare sulle righe per ogni blocco, coiè 1/3 del numero di byte totali per ogni riga ossia 1440, come spiegato in precedenza.
Inoltre viene utilizzato una variabile come flag ciamata K, incrementata di '1' al termine della scrittura di una nuova riga; andando a testare il suo bit meno significativo, si capisce se al ciclo successivo si deve copiare una riga dal frame pari o dal frame dispari nel buffer finale. Quindi vi sarà un ciclo *for* esterno che conta fino a 192, contenuto in una costante chiamata COLONNE_B, cioè 1/3 delle 576 righe dell'immagine acquisita. Al suo interno verrà testata K e verrà copiata la riga pari o dispari nel buffer finale mediante un ciclo *for* . Al termine della scrittura di ogni riga, il

puntatore impiegato deve andare a puntare alla riga successiva del rispettivo frame, all'interno di ogni blocco, ossia 960 byte successivi (distanza di 2 blocchi, 480 *2) . Il codice è il seguente:

```
for(j=0;j<colonne_b;j++)
{
      if(!(k&0x1)) {
            for(l=0;l<righe_b;l++)  {
            CAMBIO FORMATO E COPIA RIGA NEL BUFFER FINALE
            }
      }
      else
      {
            for(l=0;l<righe_b;l++){
            CAMBIO FORMATO E COPIA RIGA NEL BUFFER FINALE
            }
      }
      if(!(k&0x1)){
            punt_2 += 0x3c0;
      }
      else
      {
            punt_1 += 0x3c0;
      }
      k = (k + 1)&0x1;
}
```

Dopo aver selezionato la porzione di riga dell'immagine acquisita da copiare nel buffer destinato poi al compressore, chiamato IMAGE_BUFFER[], questa viene riscritta in formato YCbCr 4:4:4, necessario per la compressione JPEG, nella sezione riportata nel codice precedente **CAMBIO FORMATO E COPIA RIGA NEL BUFFER FINALE .**

L'immagine acquisita vede 720 valori di luminanza (Y) e 360 valori di crominanaza (Cr) e 360 per l'altro segnale differenza (Cb) per ogni riga. Quindi i campioni per ogni pixel sulla riga hanno il seguente formato:

$$Cb_0Y_0Cr_0Y_1\ Cb_1Y_2Cr_1Y_3\ Cb_2Y_4Cr_2Y_5\ldots\ldots\ldots$$

Il formato invece che si cerca di ottenere possiede per ogni pixel le informazioni sia di luminanza che dei due segnali differenza, come il seguente:

$$Y_0Cr_0Cb_0Y_1Cr_1Cb_1Y_2Cr_2Cb_2\ldots\ldots\ldots$$

Si è quindi dichiarato un vettore di 4 elementi di tipo *unsigned char* chiamato CB0Y0CR0Y1[], il quale ogni 4 cicli viene fatto puntare dove punta il puntatore del frame, e quindi i suoi elementi

diventano i 4 byte successivi a tale locazione, che come dice il nome saranno due di crominanza e due di luminanza.

Per ottenere il formato 4:4:4, si attuano quindi gli scambi tra i byte contenuti in questo vettore, come si vede dal codice seguente:

```
for(l=0;l<righe_b;l++) {
      cb0y0cr0y1[i++] = *(punt_2++);
      if (!(i &= 0x3)) {
            image_buffer[index++] = cb0y0cr0y1[1];    // Y0
            image_buffer[index++] = cb0y0cr0y1[2];    // Cr
            image_buffer[index++] = cb0y0cr0y1[0];    // Cb
            image_buffer[index++] = cb0y0cr0y1[3];    // Y1
            image_buffer[index++] = cb0y0cr0y1[2];    // Cr
            image_buffer[index++] = cb0y0cr0y1[0];    // Cb
      }
}
```

La seconda operazione che deve eseguire il blocco è la compressione JPEG, la quale è stata ottenuta utilizzando una libreria già funzionante, modificata leggermente per contenere solo le funzioni utilizzate dal sistema Payload.

La funzione principale acquisisce l'immagine da comprimere passata come argomento e restituisce il buffer che contiene il blocco compresso in JPEG, che verrà poi salvato in memoria FLASH da una funzione esterna mediante il driver implementato, spiegato nel prossimo sottocapitolo. Al suo interno vengono distinti i diversi passi attraverso cui viene compressa l'immagine:

```
// Step 1: allocate and initialize JPEG compression object
cinfo.err = jpeg_std_error(&jerr);
jpeg_create_compress(&cinfo);

// Step 2: specify data destination (eg, a file)
jpeg_mem_dest(&cinfo, final_buffer, FINAL_BUF_SIZE);

// Step 3: set parameters for compression
cinfo.image_width = image_width;
cinfo.image_height = image_height;
cinfo.input_components = 3;
cinfo.in_color_space = JCS_YCbCr;
jpeg_set_defaults(&cinfo);
jpeg_set_quality(&cinfo, quality, TRUE);

// Step 4: Start compressor
jpeg_start_compress(&cinfo, TRUE);
```

In questi primi 4 passi, viene creata un *structure* chiamata CINFO, la quale conterrà tutte le informazioni sulle caratteristiche di compressione, viene inizializzata l'area di memoria destinazione dell'immagine e vengono allocate dinamicamente delle locazioni riservate all'elaborazione dei suoi dati che richiedono circa 30 Kbyte di memoria libera. Infine vengono inserite le caratteristiche della compressione voluta, come le dimensioni dell'immagine, il numero di componenti per pixel (YCbCr) e il livello di compressione e chiamato l'inizio della compressione, dove vengono eseguiti i settagli delle funzioni da utilizzare nella compressione partendo dai parametri inseriti.

```
// Step 5: while (scan lines remain to be written)
row_stride = image_width * 3; // JSAMPLEs per row in image_buffer
while (cinfo.next_scanline < cinfo.image_height) {
      row_pointer[0] = & image_buffer[(cinfo.next_scanline * row_stride)];
      (void)jpeg_write_scanlines(&cinfo, row_pointer, 1);
}
```

Questa sezione invece è quella in cui avviene veramente la compressione JPEG. Qui vengono svolte tutte le diverse operazioni base definite dallo standard.

Infatti con la chiamata di funzione incluse nella libreria l'immagine viene divisa in blocchi di 8 pixel x 8 pixel, a cui iene applicata la trasformata coseno (ovviamente discreta).

Qui di seguito viene riportata una riga esempio di come è stato scritto il codice in modo da utilizzare un'unità MAC del DSP per il calcolo base, moltiplicazione e somma del risultato, della trasformata:

```
tmp12 = MULTIPLY(z10, - FIX_2_613125930) + z5;
```

Dopo inoltre l'operazione di quantizzazione si ottiene la matrice dei valori pesati delle frequenze presenti sul blocco, la quale viene letta a zig-zag per avere un'ordine che rende efficiente il successivo passo di codifica, diversa per la componente DC e quella AC.

```
// Step 6: Finish compression
jpeg_finish_compress(&cinfo);
segment_length = jpeg_mem_dest_getcount(&cinfo);

// Step 7: release JPEG compression object
jpeg_destroy_compress(&cinfo);
```

Infine in questi ultimi due passi, l'immagine JPEG viene completata con l'inserimento dell'intestazione JFIF (JPEG File Interchange Format), la quale come dice il nome permette di rendere compatibile questo formato a diverse applicazioni e piattaforme. Infatti in questa parte vengono inserite le tabelle di codifica e quantizzazione, necessarie alla decodifica in decompressione, ossia per visualizzare l'immagine. Inoltre nell'ultimo step viene liberata tutta la memoria per la struttura dati allocata inizialmente.

4.7.4 Driver FLASH M29W160EB

La memoria FLASH utilizzata oltre ad un datasheet molto chiaro, forniva un set di funzioni per il suo utilizzo accompagnate da un documento di descrizione piuttosto chiaro, facilitando l'uso del componente. Essendo il driver completo piuttosto corposo, verranno qui spiegate le principali funzioni implementate e utilizzate.

La funzione principale è quella di lettura. Semplicemente viene passato l'indirizzo (relativo all'interno della memoria non assoluto del processore) da cui leggere, chiamato ULOFFSET e il numero di word, ossia 2 byte, che si vogliono leggere dalla memoria, chiamato LCOUNT, ritornando un puntatore ad essi, dapprima allocato dinamicamente nella funzione chiamante, chiamato PNDATA:

```
for (i = 0; i < lCount; i += 2, ulOffset += 2) {
      ReadFlash(ulOffset, &data, FALSE);
      pnData[i] = (unsigned char) (data & 0x00FF);
      pnData[i + 1] = (unsigned char)((data & 0xFF00) >> 8);
}
```

Come si vede dal codice viene quindi chiamata la vera funzione di lettura, scritta con istruzioni assembler in C mediante il comando *asm*, la cui operazione è quella di tradurre l'indirizzo in modo che il processore lo metta sul bus del controller asincrono (si veda la figura 2.8) e metterlo nel registro chiamato p2. L'operazione fondamentale è quella successiva, dove viene scritto sul bus p2 e viene letto il valore del data bus, assegnandolo al buffer temporaneo nValue.

```
asm ("p2.l = 0x0000;");
asm ("p2.h = 0x2000;");
asm ("r3 = %0;": : "d" (ulOffset));
asm ("r2 = p2;");
asm ("r2 = r2 + r3;");
asm ("p2 = r2;");
asm ("%0 = w[p2] (Z);": "=d" (nValue) : );
```

Questo saà poi ritornato a PNDATA, il cui salvataggio prevede la distinzione dei due byte, quello che risiede negli 8 bit più significativi e qullo che risiede in quelli meno significativi andranno ad occupare due elementi contigui della memoria allocata per questo vettore.

Affinché comunque la memoria si trovi in modalità di lettura sia dopo l'accensione che dopo cicli di scrittura o cancellazione, deve essere dapprima mandato un comando chiamato di Read/Reset, che significa mandare la parola 0xF0 all'indirizzo 0x0AAA. Questa operazione nel codice è svolta attraverso una funzione chiamata ResetFlash la cui unica riga appunto è la seguente:

```
WriteFlash(0x0AAA, 0xf0, TRUE);
```

La funzione WriteFlash esegue qunto descritto e verrà spiegato in dettaglio in seguito.

L'operazione invece di scrittura di dati in memoria FLASH, avviene in due fasi principali:

- cancellazione dei blocchi
- scrittura dei dati nei blocchi

La prima fase richiede la conoscenza della mappa di memoria della FLASH, ossia dell'associazione tra i blocchi e gli indirizzi. Per questi motivi, prima di qualsiasi utilizzo della memoria, vi è una fase d'inizializzazione dove viene creata una struttura dati chiamata SECTORINFO[], in cui si trova per ogni valore dell'indice due indirizzi, quello iniziale e quello finale del blocco relativo all'indice.
Avendo queste informazioni, viene chiamata una funzione EraseBlock a cui viene passato il numero del blocco da cancellare:

```
WriteFlash(0x0555, 0xaa, TRUE);
WriteFlash(0x02AA, 0x55, TRUE);
WriteFlash(0x0555, 0x80, TRUE);
WriteFlash(0x0555, 0xaa, TRUE);
WriteFlash(0x02AA, 0x55, TRUE);
WriteFlash(ulSectorOff, 0x30, FALSE);
```

Il comando di cancellazione di un blocco, consiste nel mandare 5 byte fissi ad indirizzi fissi (il dato 0xAA all'indirizzo 0x0555, il dato 0x55 alla locazione 0x02AA e così via), più il sesto che deve contenere l'indirizzo iniziale o interno al blocco da cancellare. Tutto mediante la solita funzione per la scrittura di una singola parola denominata WriteFlash.
Ora la memoria è pronta per essere scritta all'interno del blocco cancellato, quindi l'applicazione per scrivere dei dati chiama un funzione, detta WriteData, passandole come argomenti un puntatore ai dati e l'indirizzo iniziale della memoria su cui si vuole scrivere, che mette insieme due byte di dati e richiama la funzione WriteFlash un numero di volte pari al numero di parole da scrivere:

```
for (i = 0; (i < lCount) && (ErrorCode == NO_ERR); i += 2, ulOffset += 2) {
data = (((unsigned short) pnData[i + 1]) << 8) | ((unsigned short) pnData[i]);
WriteFlash(ulOffset, data, FALSE);
}
```

Quindi la funzione WriteFlash, trasmette l'indirizzo posto nel valore assoluto del processore (come nella fase di lettura) mediante i registri r2 e p2, il quale viene quindi scritto sul bus indirizzi, insieme al dato passato per valore alla variabile NVALUE:

```
asm ("p2.l = 0x0000;");
asm ("p2.h = 0x2000;");
asm ("r3 = %0;": : "d" (ulOffset));
asm ("r2 = p2;");
asm ("r2 = r2 + r3;");
asm ("p2 = r2;");
asm ("SSYNC;");
```

```
asm ("w[p2] = %0;": : "d" (nValue));
```

Le funzioni di scrittura WriteData e di cancellazione EraseBlock, utilizzano una funzione chiamata PollToggleBit, per capire se l'operazione richiesta si è conclusa con successo. In questa funzione, la memoria ritorna in modalità di lettura e si testano i valori del bit 6 (Toggle Bit) e 5 (Error Bit) del bus dati, per dire se si sono verificati errori o non si è ancora conclusa l'operazione. La proceduraè la seguente:

se dopo aver mandato il comando di scrittura e cancellazione il segnale del bus dati numero 6 non commuta, allora si è tutto concluso con successo mentre se commuta si testa il bit 5 di errore, il quale se è a '1' significa che probabilmente c'è stato un'errore quindi si ritesta il bit 6 che se continua a commutare allora l'operazione è andata a buon fine altrimenti se è ancora fisso si frema e segnala un errore; nel caso il bit 5 sia a '0' inizia tutto il ciclo, per un tempo limite di 1 μs, dopo il quale viene mandato l'errore di time out.

Qui di seguito viene riportato il diagramma di flusso di questa funzione, più immediato e leggibile del codice scritto in assembler su C mediante sempre la funzione *asm*:

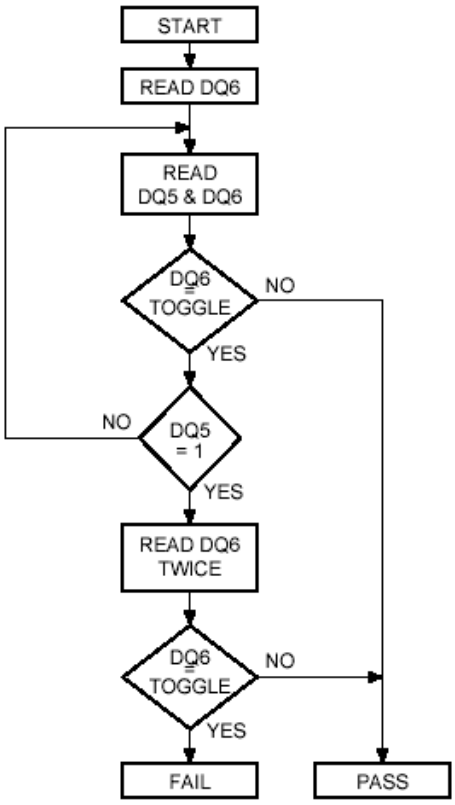


Figura 4.17: Diagramma di flusso funzione PollToggleBit

4.7.5 Protocollo Xmodem

Per la trasmissione di uno stream di dati superiore ai due byte previsti per il telecomando, come la trasmissione dell'intera immagine compressa e anche solo un blocco, mediamente di dimensione di

qualche Kbyte e la trasmissione del programma utente, si è utilizzato un protocollo seriale standard al fine di avere un handshake sulla comunicazione e un minimo controllo di errore sui dati.
Il protocollo implementato è chiamato Xmodem ed era uno standard nelle comunicazioni interne delle telescriventi di venti anni fa. Tale protocollo è stato scelto principalmente per due motivi:

- semplicità d'implementazione
- molto utilizzato nei software per windows che fungono da terminali RS-232, per l'acquisizione e la trasmissione diversi da file di testo

Infatti mediante tale protocollo si è potuto trasferire o acquisire dal processore qualsiasi stream di dato in forma binaria, per eseguire debug o per caricare programmi in formato binario, che salvati sulla flash sarebbero potuti essere utilizzati per il boot del sistema.
Prima di vedere il codice C delle funzioni che lo implementano, qui di seguito verrà descritto brevemente la struttura del protocollo.
Il funzionamento del protocollo si basa sull'utilizzo delle seguenti stringhe costanti:

- <soh> 01H, start of header
- <eot> 04H, end of transmission
- <ack> 06H, acknowledge
- <nak> 15H, negative acknowledge

La trasmissione dei dati prevede una fase d'handshake iniziale, limitato alla trasmissione di un <nak>da parte del ricevitore.
Alla ricezione di tale valore, il trasmettitore della comunicazione inizia la trasmissione del primo frame di dati. Il frame di dati è composto da:

- <soh>, che delimita l'inizio dell'intestazione
- un byte che identifica il numero di frame
- un byte complemento a uno del numero di frame
- 128 byte di dati del file da trasmettere
- un byte di checksum, ossia la somma senza riporto di tutti i 128 byte di dati

<SOH> 1 byte	<BlockNo> 1 byte	<~BlockNo> 1 byte	<Data> 128 bytes	<Checksum> 1 byte

Al termine di ogni frame, il ricevitore trasmette un <ack> se il checksum ricevuto coincide con quello da lui calcolato in ricezione, altrimenti trasmette un <nak> e il trasmettitore ritrasmette il blocco.
Il trasmettitore nel caso di esito positivo della trasmissione, trasmette il secondo frame, incrementando il numero identificativo e quindi anche quello complimentato, fino a completare tutti i dati da trasmettere e quindi va a trasmettere un <eot>, che segnale la fine della trasmissione, il quale se ricevuto correttamente viene corrisposto da un <ack> trasmesso dal ricevitore, che conclude la trasmissione. Il flusso appena spiegato a parole è riprtato qui di segutio:

Transmitter		**Receiver**	
	<-	<NAK>	Handshake
<SOH><1><254><Data 128 bytes><CHK>	->		Primo frame di dati
	<-	<ACK>	acknowledgment
<SOH><2><253><Data 128 bytes><CHK>	->		Secondo frame di dati
	<-	<ACK>	
<EOT>	->		end of transmission
	<-	<ACK>	

Lo standard prevede 128 byte di dati ma essendo utilizzato solo tra i processori della scheda Payload, ProcA e ProcB all'interno del sistema PICPOT, si può aumentare o diminuire il numero di byte di dati trasmessi per ogni frame.
Infine si presenta un piccolo problema se il numero di byte da trasmettere non sono multipli di 128, in quanto l'ultimo frame non è completo. In tal caso per le comunicazioni interne di PICPOT, i byte restanti per i dati nell'ultimo frame è riempito da un stringa definita, che nelle immagini JPEG viene ignorata, dato che l'header contiene la lunghezza dell'immagine e nel programma utente non verrebbero considerati.
Il codice che lo implementa presenta un'insieme di *define* che contengono le 4 costanti necessarie al funzionamento del protocollo, chiamate UART_ACK, UART_NACK, UART_SOH, UART_EOT oltre al padding definito come UART_PADDING.
Le variabili utilizzate saranno un vettore *unsigned char* da 132 elementi chiamato BLOCK[], cioè un vettore da 132 byte che verrà riempito dei 128 byte di dati, 1 di soh, 2 per i numeri di blocco e in

fondo 1 di crc. Inoltre servirà una variabile temporanea sempre da 8 bit chiamata CRC per mantenere il crc relativo ai dati, oltre ad un contatore chiamato NBLOCK per il conto di quanti blocchi di dati sono richiesti per trasmettere i byte indicati e altri due contatori per i cicli, tutti dichiarati come tipo *integer*.

La fase di trasmissione via Xmodem, prevede due funzioni:

- alla prima le vengono passati due argomenti che saranno l'indirizzo iniziale e finale di memoria dei dati da trasmettere chiamata Xmodem_addrs, la quale chiama poi quella di trasmissione, fornendole i byte di differenza tra i due indirizzi e mandando l'indirizzo di partenza:

```
int Xmodem_addrs(unsigned char * addr1, unsigned char * addr2)
{
      int count;
      count = addr2 - addr1;
      return(Xmodem_count(count, addr1));
}
```

  Questa funzione sarà quella utilizzata dal codice eseguito alla ricezione del telecomando di trasmissione di uno o tutti i blocchi dell'immagine, che utilizzerà la mappa dei segmenti dell'immagine salvata all'inizio del blocco in memoria FLASH; per questo motivo era più semplice chiamare la funzione per l'Xmodem dando immediatamente solo gli indirizzi di inizio e fine ottenuti dalla mappa precedentemente salvata in memoria con l'immagine

- la funzione che implementa veramente il protocollo è chiamata Xmodem_count, che consiste in due cicli *for* annidati, quello esterno è eseguito un numero di volte pari ai blocchi di dati necessari, calcolati inizialmente nella variabile NBLOCK, e inserisce nei primi tre elementi del vettore BLOCK[], l'soh, il numero di blocco e quello complementato, previsti dal protocollo. Il ciclo interno invece viene eseguito 128 volte, e inserisce 128 byte nel vettore temporaneo BLOCK[], dalla posizione 3 in poi, e calcola il valore del crc, sempre controllando che il numero di dati sia 128, perché se minore inserisce i dati validi e poi completa i 128 byte col pattern, smettendo di calcolare il crc. Il codi ce della funzione è il seguente:

```
while (read_uart_byte() != UART_NACK) {}
nblock = (count) / 128 + 2;
for (j = 1; j <= nblock; j++) {
      crc = 0;
      block[0] = UART_SOH;
      block[1] = j & 0xff;
      block[2] = 0xff - block[1];
      for (x = 0; x < 128; x++) {
```

```
                if (j == nblock && ((j - 1) * 128 + x) > count) {
                        block[x + 3] = UART_PADDING;
                } else {
                        block[x + 3] = *(addr++);
                }
                        crc += block[x + 3];
        }
        block[x + 3] = crc;
        do {
                write_uart_blocking(block, 132);
        } while (read_uart_byte() == UART_NACK);
}
do {
        write_uart_byte(UART_EOT);
} while (read_uart_byte() == UART_NACK);
```

Infine viene trasmesso il vettore tramite la funzione del driver UART e il byte eot di fine trasmissione.

La fase di ricezione di dati mediante il protocollo Xmodem è gestita invece mediante una funzione singola, alla quale viene passato il numero di dati che verranno inviati, associati ad una variabile *integer* chiamata MAXCOUNT, e l'indirizzo di partenza della memoria dove verranno poi scritti, associato ad un puntatore di tipo *unsigned char* chiamato ADDRS.

La funzione, trasmette il comando nack e rimane in attesa del riempimento del buffer di ricezione con il primo byte inviato dal trasmettitore. Quindi inizia un ciclo che termina solo alla ricezione del comando eot, riceve i vari blocchi, controllando l'soh, andando a calcolare infine con i dati ricevuti il crc è confrontarlo con quello ricevuto. Nel caso di uguaglianza tra i due, vengono salvati in memoria i dati ricevuti tramite il comando *memcpy*, controllando che il pacchetto non sia l'ultimo, dove in tal caso vengono salvati solo i dati utili se non in numero multiplo di 128; dopo viene trasmesso poi un ack e viene incrementato il contatore interno di blocchi N. Nel caso di crc diversi, viene semplicemente trasmesso un nack e non aggiornato il contatore, ignorando quindi i dati ricevuti.

```
write_uart_byte(UART_NACK);
while (uart_select() == 0) {
        if (uart_select() == 0) {
                write_uart_byte(UART_NACK);
        }
}
while ((data = read_uart_byte()) != UART_EOT) {
        if (data != UART_SOH) {
        return -1;
```

```
        }
        block[0] = data;
        crc = 0;
        read_uart_blocking(block + 1, 131);
        for (i = 0; i < 128; i++) {
        crc += block[i + 3];
        }
        if (crc != block[131]) {
              write_uart_byte(UART_NACK);
        } else {
              if (maxcount >= n + 128) {
                    memcpy(addr + n, block + 3, 128);
              } else if (maxcount > n) {
                    memcpy(addr + n, block + 3, maxcount - n);
                    } else {
                    /* Do nothing here */
              }
              n += 128;
              write_uart_byte(UART_ACK);
        }
}
```

Capitolo 5

Conclusioni

Il lavoro svolto per questa tesi è consistito nel progetto hardware e software descritto nelle pagine precedenti, in concomitanza ad un continuo aggiornamento e coordinamento con gli sviluppatori delle altre schede del satellite PICPOT.

Il progetto e la realizzazione della parte hardware è completo ed è stato pienamente testato in tutti i suoi componenti, mediante diverse fasi di collaudo, sia a livello ohmico che mediante il test mirato dei componenti attraverso programmi scritti in linguaggio C.

Il progetto software non è ancora completamente concluso, in quanto oltre alla mancanza di un driver per la porta SPI e alla mancanza di una funzione di analisi dei telecomandi, è necessario anche un'ottimizzazione finale della memoria utilizzata nelle diverse fasi di funzionamento, per avere un miglioramento dei consumi energetici.

Il lavoro svolto mi ha permesso di acquisire una certa esperienza e capacità di lettura dei datasheet dei componenti elettronici, soprattutto digitali, e di acquisire una certa padronanza della lingua inglese; inoltre mi ha dato la possibilità di acquisire una certa capacità nella realizzazione di schede elettroniche, sia nella fase di sviluppo del PCB mediante il software CAE di Mentor Graphics, sia nella fase di saldatura di componenti sia SMD che PTH. Infine la possibilità di utilizzare un DSP nuovo e complesso come il Blackfin mi ha permesso di capire profondamente in cosa consta lo sviluppo di software per l'hardware e di apprendere l'utilizzo di un ambiente di sviluppo avanzato come il Visual DSP++ 4.0.

Appendice A

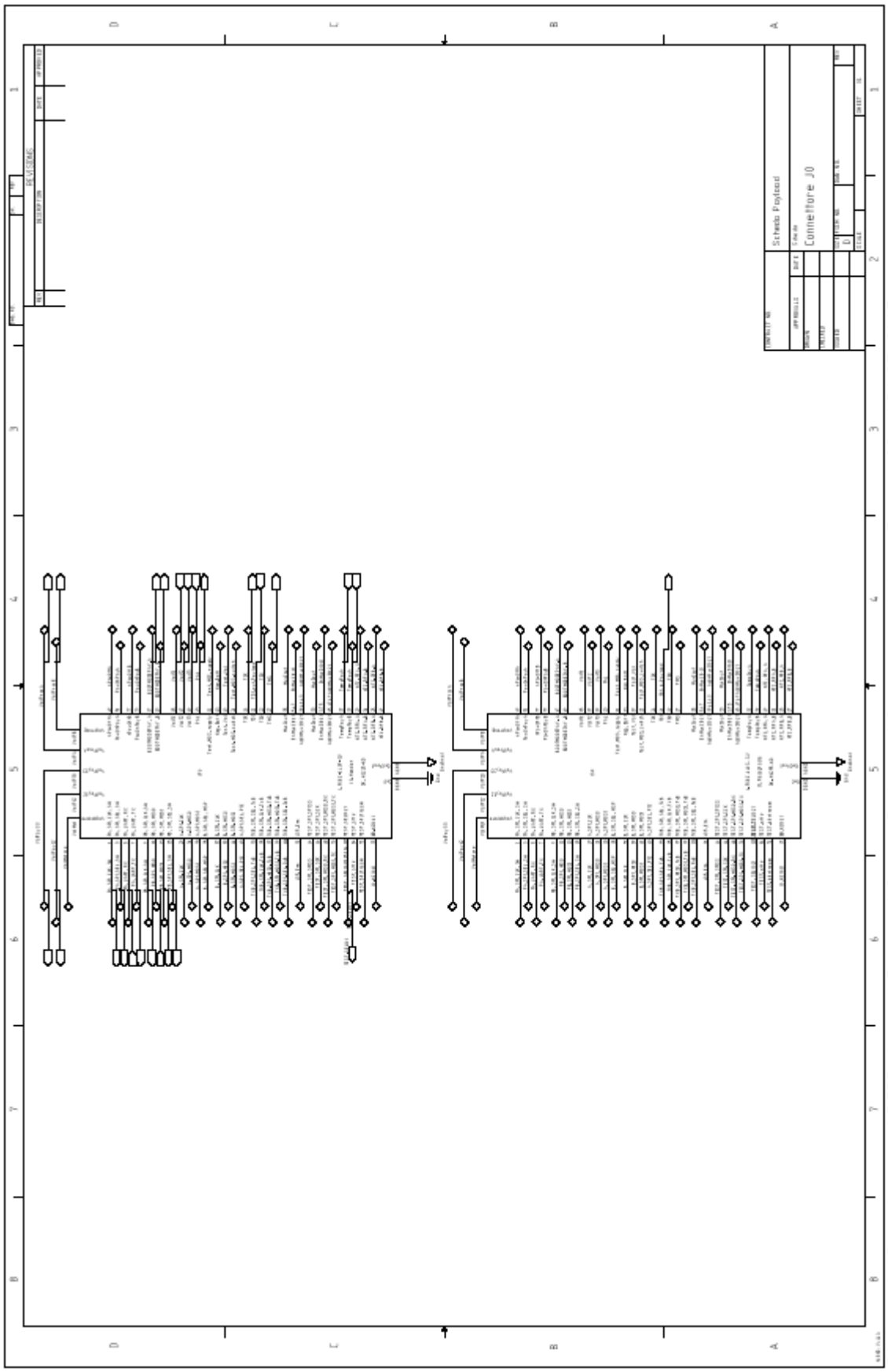
Schemi Elettrici

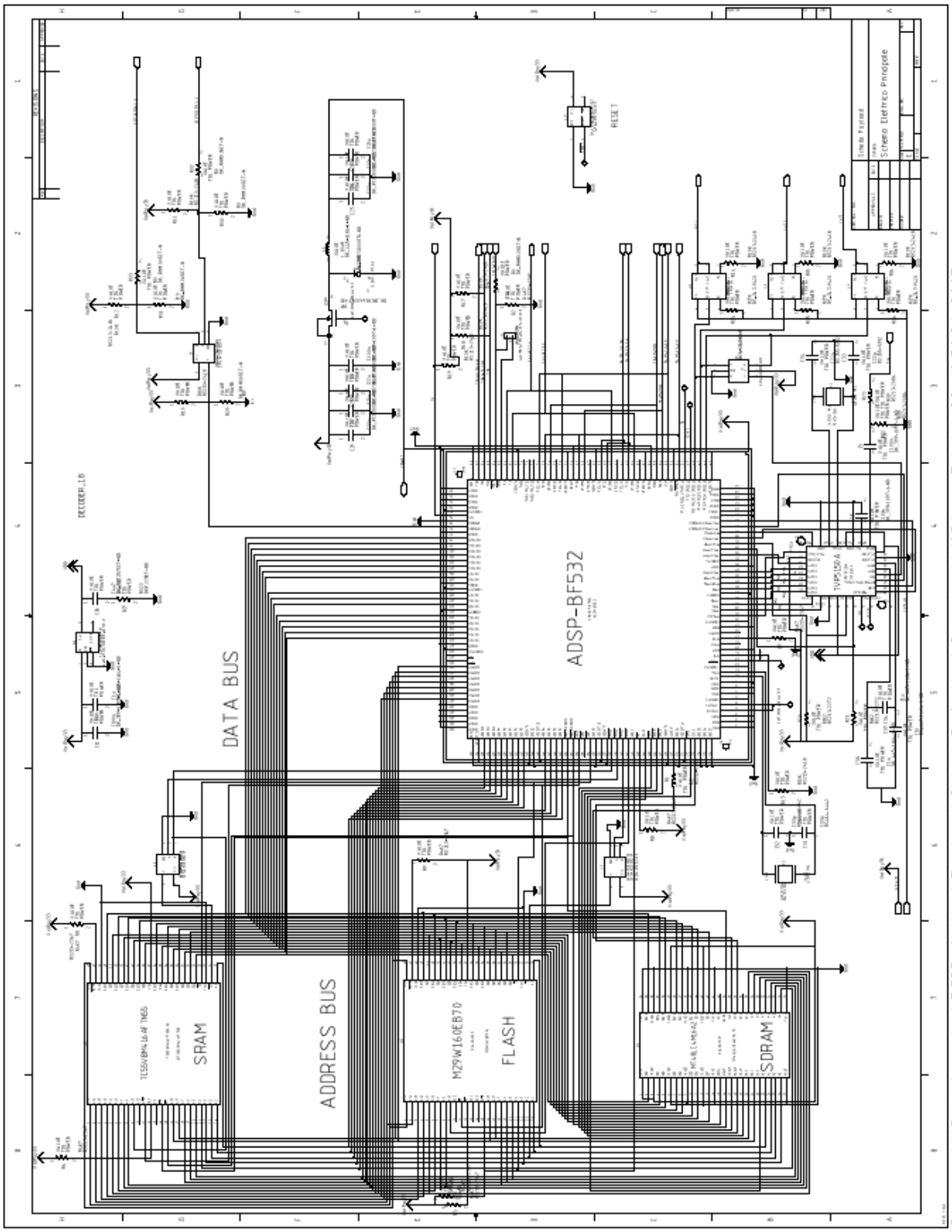
In questa sezione sono riportati gli schemi elettrici, divisi nelle seguenti 4 pagine:

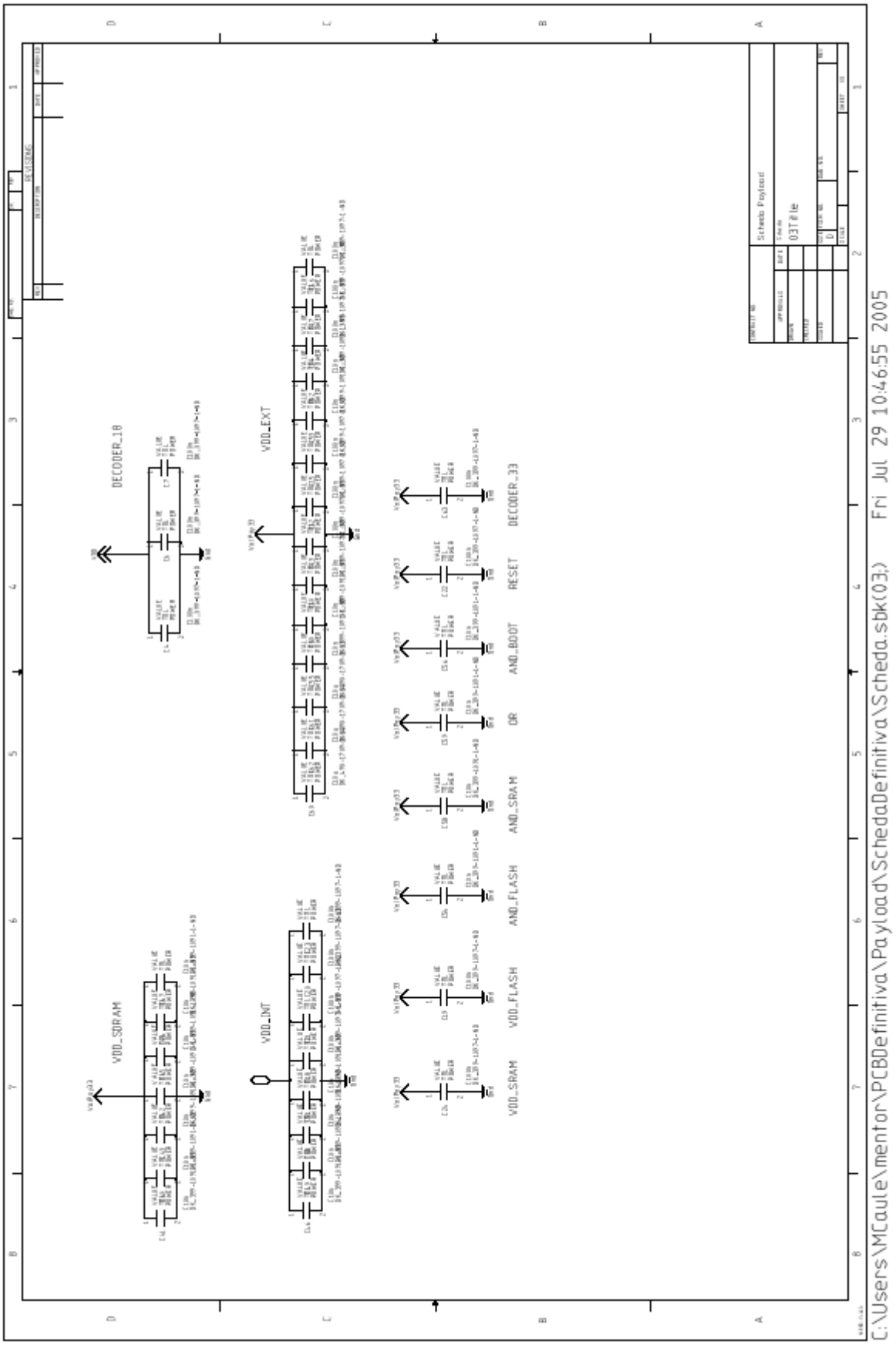
1. Connettore J0
2. Layer 1 – Schema Elettrico Principale
3. Layer 4 – Condensatori di Bypass
4. Sensore di Temperatura

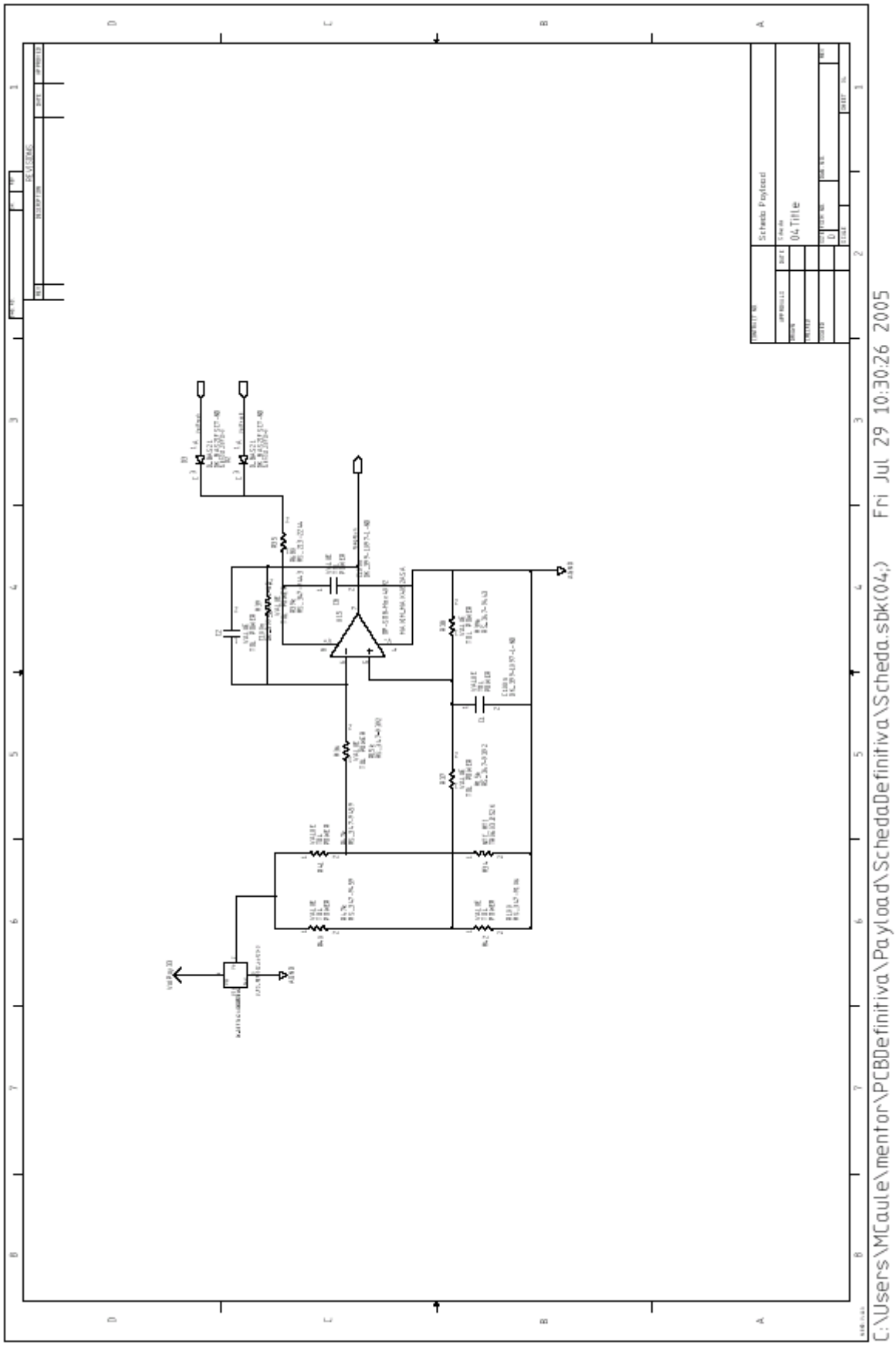
Inoltre sono riportati gli schemi elettrici della scheda Espansione J0, divisi nelle seguenti 2 pagine:

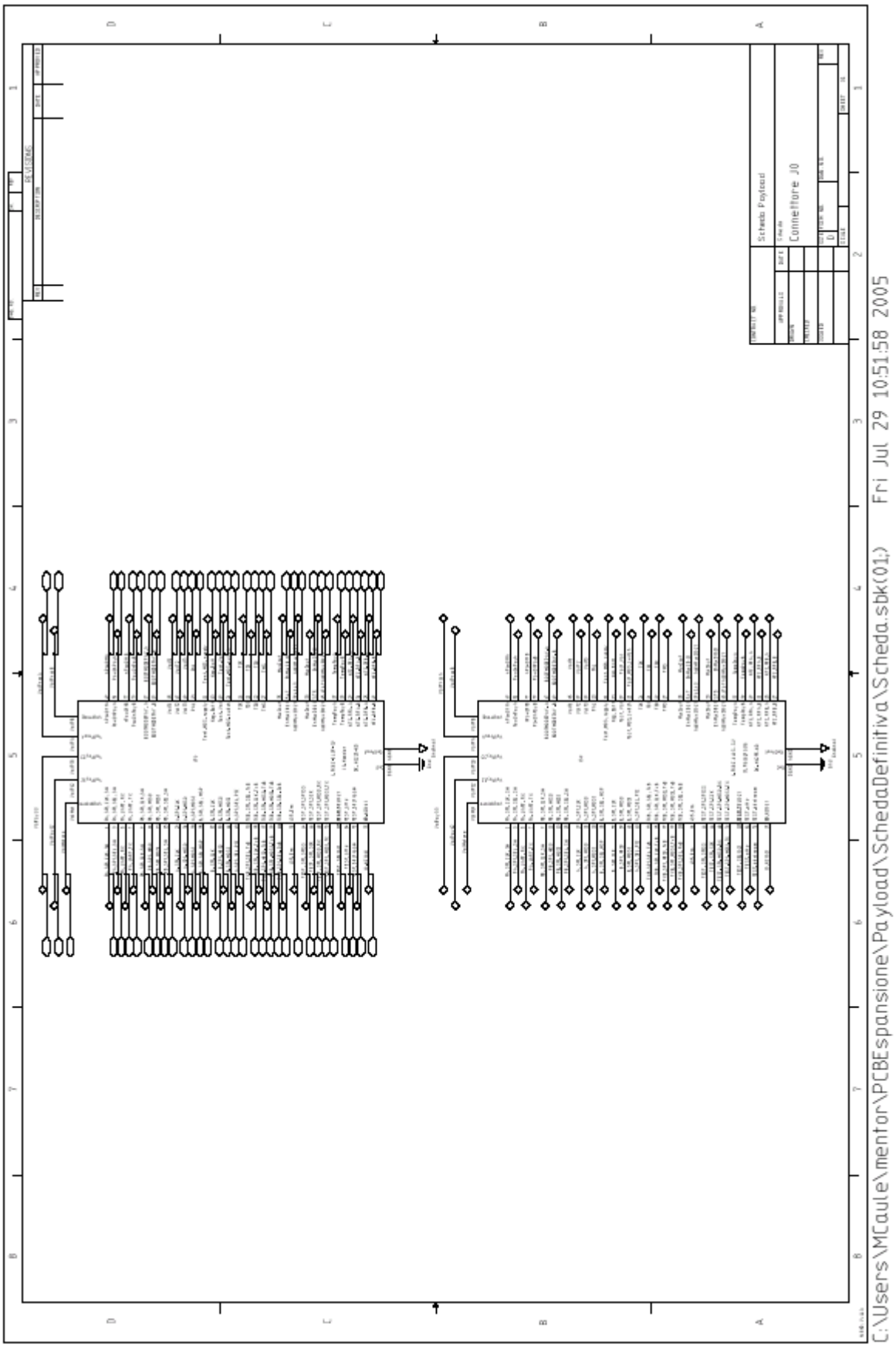
1. Connettore J0
2. Layer 1 – Schema Elettrico Principale

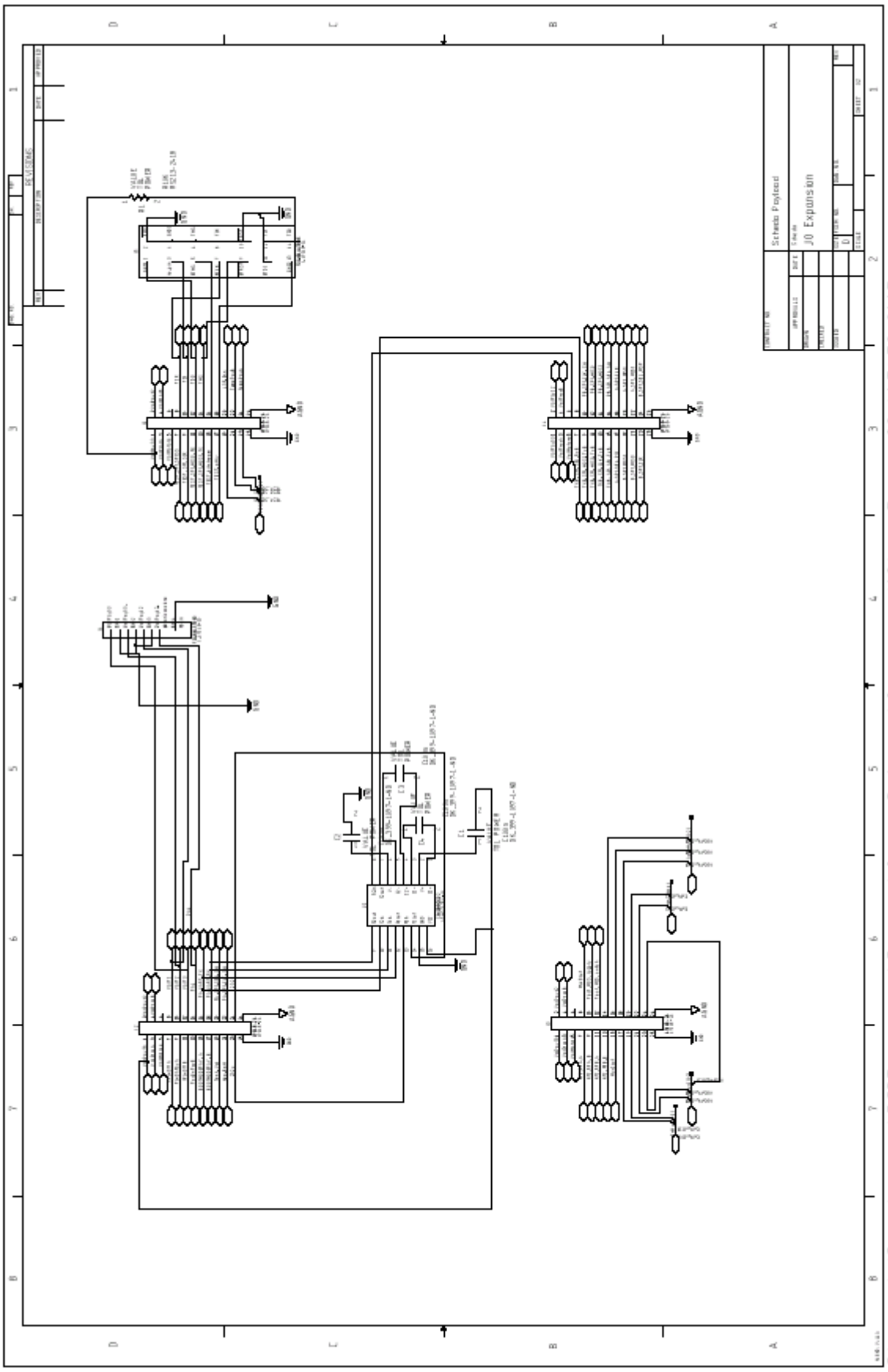

Scheda Payload
Connettore J0
C:\Users\MCaule\mentor\PCBDefinitiva\Payload\SchedaDefinitiva\Scheda.sbk(01;) Fri Jul 29 10:30:25 2005


ADSP-BF532
DATA BUS
ADDRESS BUS
SRAM
FLASH
SDRAM
RESET
C:\Users\MCaule\mentor\PCBDefinitiva\Payload\SchedaDefinitiva\Scheda.sbk(02,) Fri Jul 29 10:30:26 2005


VDD_SDRAM
VDD_INT
DECODER_18
VDD_EXT
VDD_SRAM
VDD_FLASH
AND_FLASH
AND_SRAM
OR
AND_BOOT
RESET
DECODER_33
Scheda Payload
037File
C:\Users\MCaule\mentor\PCBDefinitiva\Payload\SchedaDefinitiva\Scheda.sbk(03;) Fri Jul 29 10:46:55 2005


Scheda Payload
04.Title
C:\Users\MCaule\mentor\PCBDefinitiva\Payload\SchedaDefinitiva\Scheda.sbk(04;) Fri Jul 29 10:30:26 2005


Scheda Payload
Connettore J0
C:\Users\MCaule\mentor\PCBEspansione\Payload\SchedaDefinitiva\Scheda.sbk(01;) Fri Jul 29 10:51:58 2005

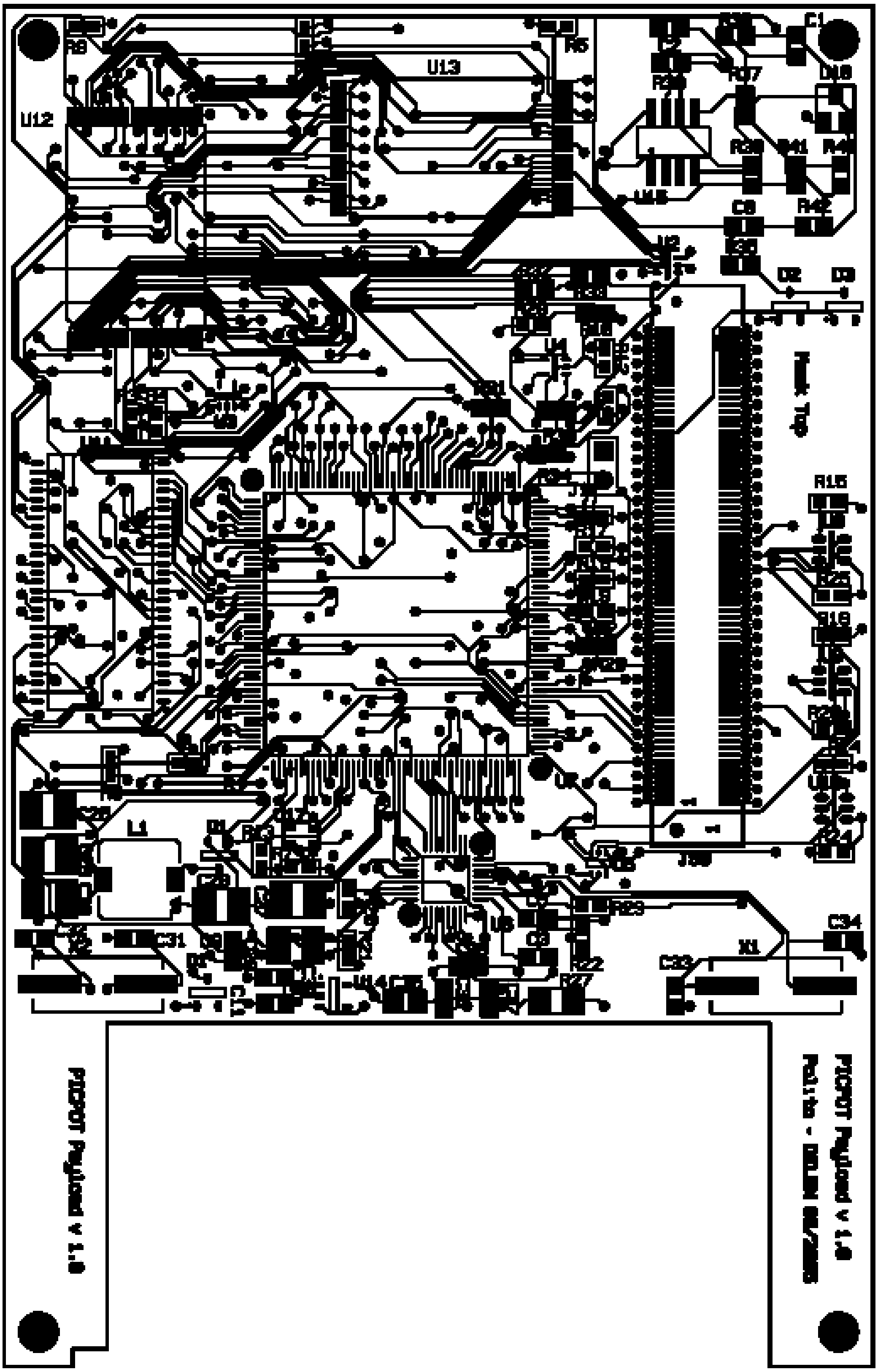

Scheda Payload
J0 Expansion
C:\Users\MCaule\mentor\PCBEspansione\Payload\SchedaDefinitiva\Scheda.sbk(02;) Fri Jul 29 10:51:58 2005

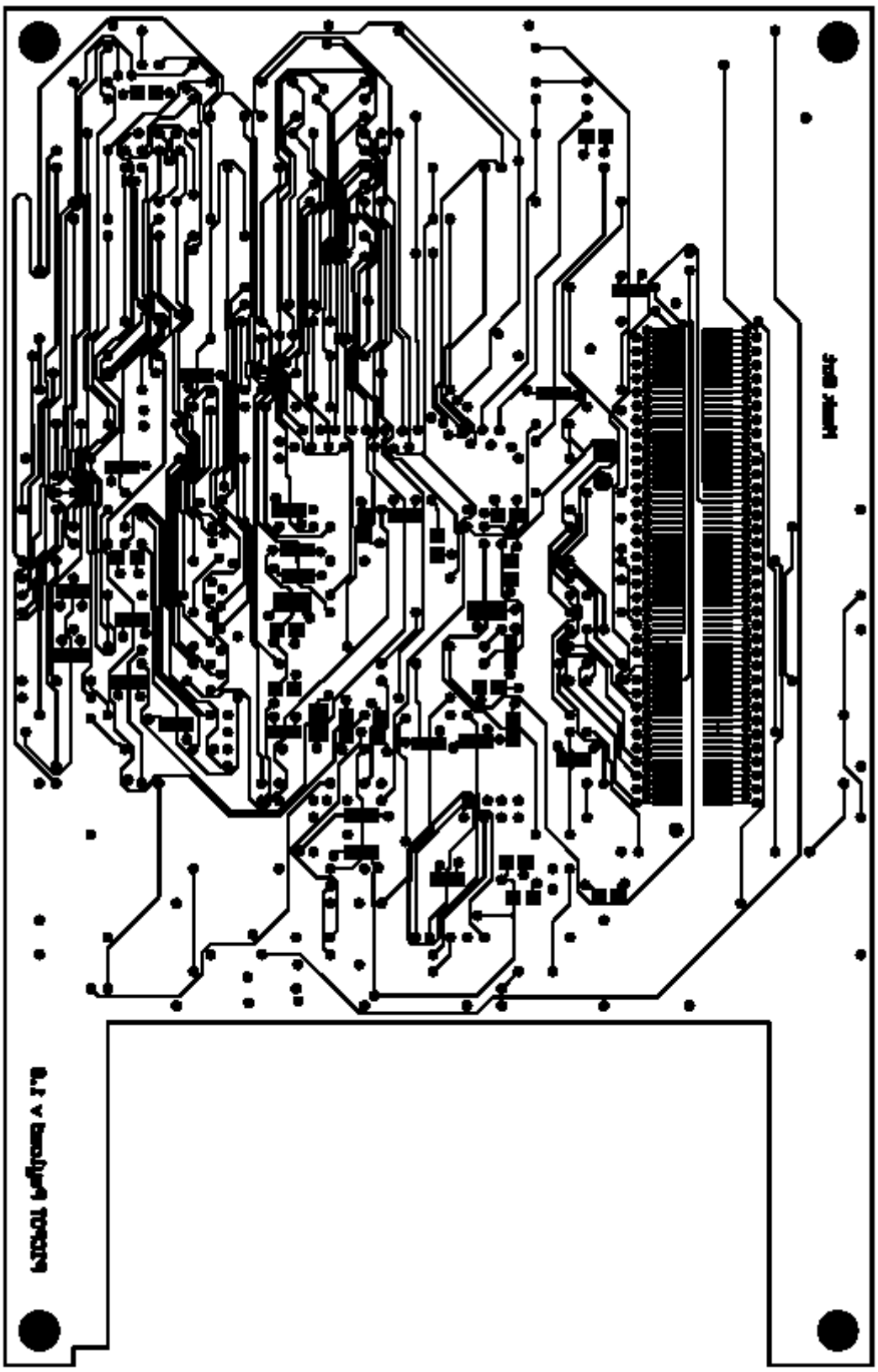
Appendice B

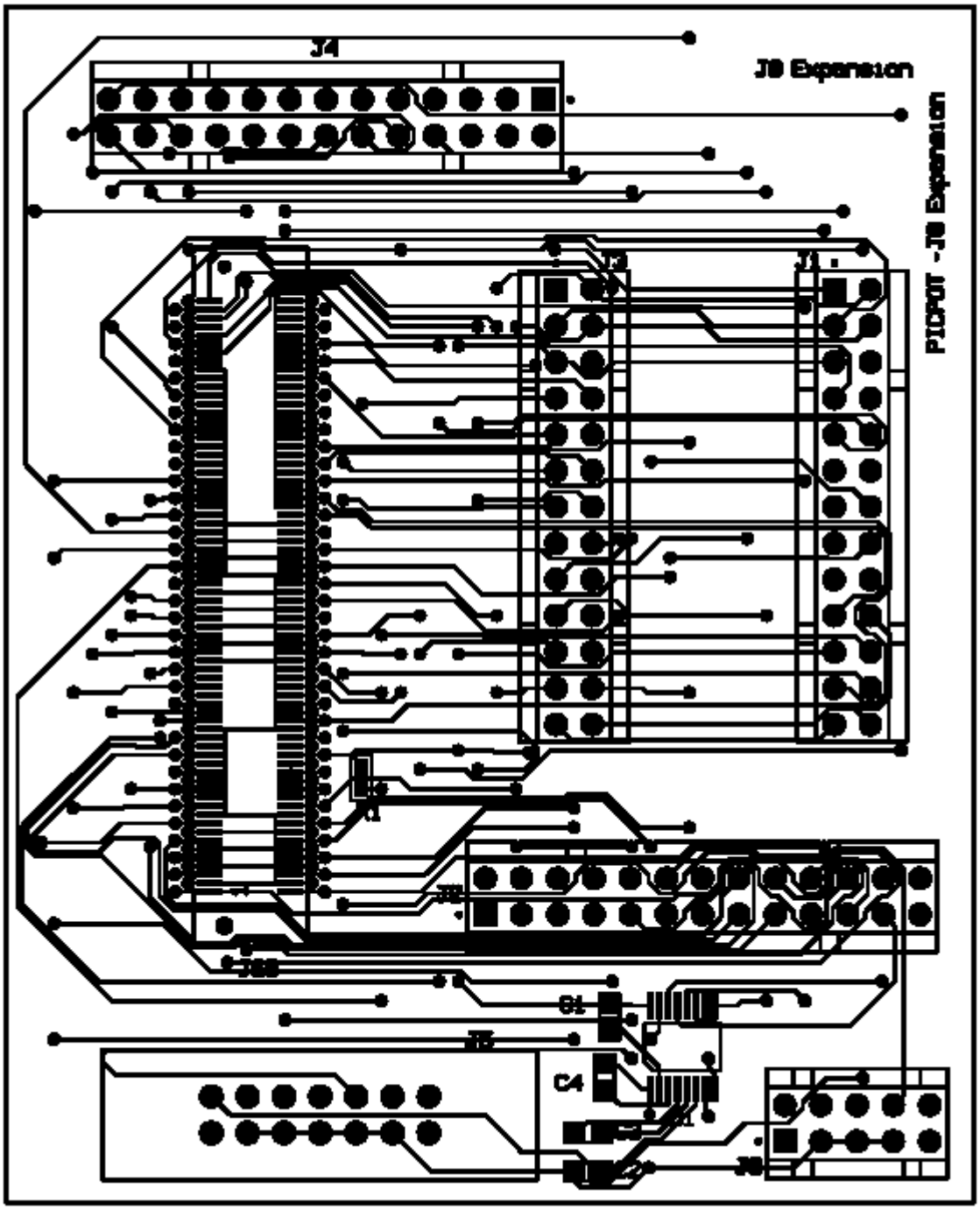
PCB

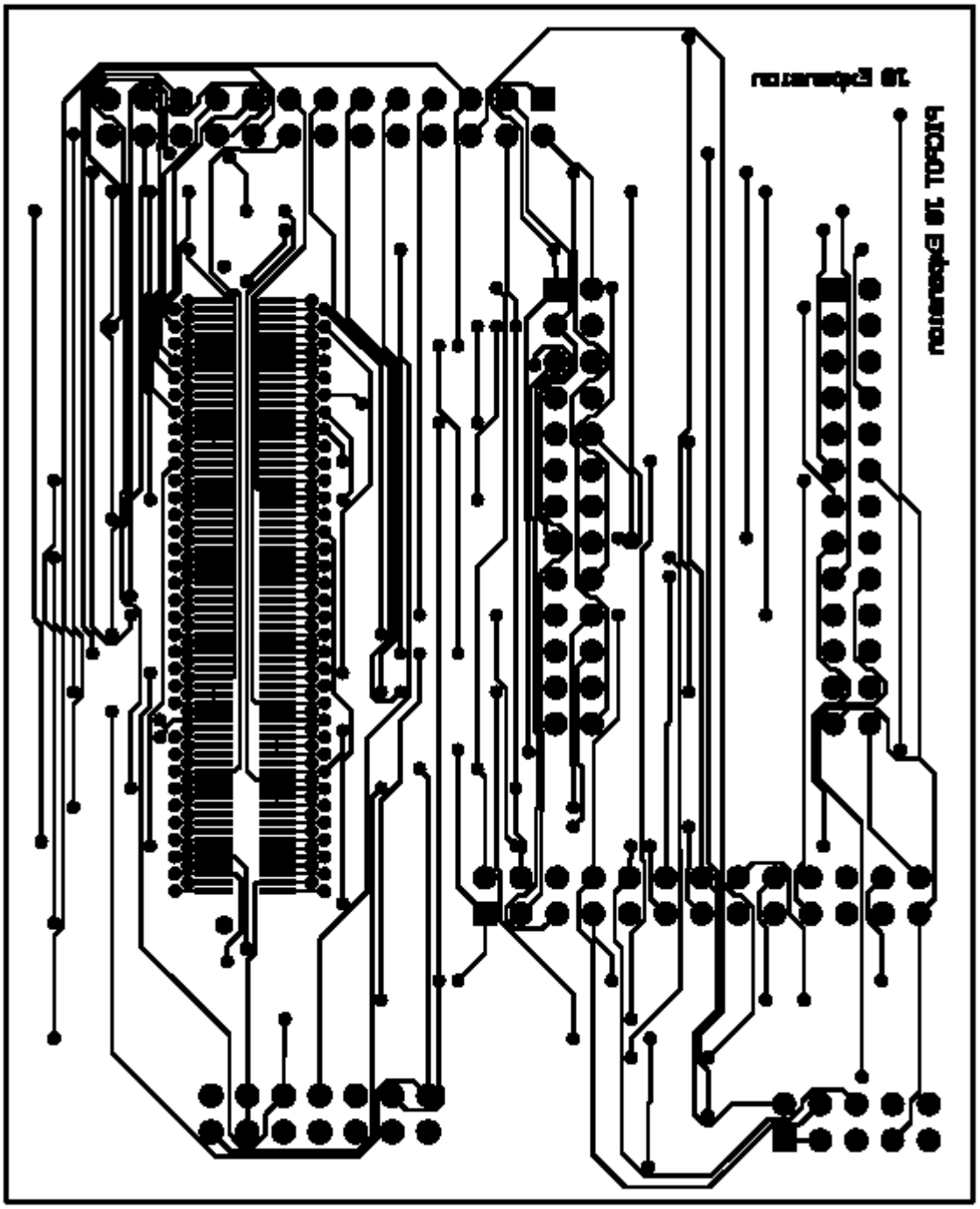
In questa sezione sono riportati tutti i PCB realizzati delle due schede, Payload e Espansione J0, nel seguente ordine:

1. Scheda Payload – Layer 1
2. Scheda Payload – Layer 4
3. Scheda Espansione J0 – Layer 1
4. Scheda Espansione J0 – Layer 4






J4
J8 Expansion
PICPOT - J8 Expansion
J1
J2
J3
C1
C4
J5
J6



Appendice C

Codice Sorgente

In questa sezione è riportato tutto il codice scritto in linguaggio C e assembler per l'applicazione richiesta dalle specifiche del sistema Payload.

Il codice che segue è strutturato nelle seguenti sezioni:

1. Codice del Linker Description File dell'applicazione
2. Codice d'inizializzazione memoria SDRAM in assembler
3. Codice d'inizializzazione dei registri del core e dei periferici del DSP
4. Codice d'inizializzazione interrupt e API per UART
5. Codice fase acquisizione e chiamata alla compressione JPEG (senza libreria completa)
6. Codice protocollo Xmodem
7. Codice protocollo I^2C
8. Driver Flash

```
/*****************************************************************/
** LDF for ADSP-BF532.
**
** There are a number of configuration options that can be specified
** either by compiler flags, or by linker flags directly. The options are:
**
** USE_PROFILER0
**      Enabled by -p. Link in profiling library, and write results to
**      both stdout and mon.out.
** USE_PROFILER1
**      Enabled by -p1. Only write profiling data to mon.out.
** USE_PROFILER2
**      Enabled by -p2. Only write profiling data to stdout.
** USE_PROFILER
**      Equivalent to USE_PROFILER0.
** USE_FILEIO
**    Always defined; enables the File I/O Support, which is necessary
**    for printf() to produce any output.
** __WORKAROUNDS_ENABLED
**    Defined by compiler when -workaround is used to direct LDF to
**    link with libraries that have been built with work-arounds
**    enabled.
**
*/

ARCHITECTURE(ADSP-BF532)

#ifndef __NO_STD_LIB
SEARCH_DIR( $ADI_DSP/Blackfin/lib )
#endif

/* Moving to primIO means that we must always include the FileIO support,
** so that printf() will work.
*/

#ifndef USE_FILEIO    /* { */
#define USE_FILEIO 1
#endif  /* } */

#ifdef USE_PROFILER   /* { */
#define USE_PROFILER0
#endif  /* } */

#ifdef USE_PROFILER0  /* { */
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define PROFFLAG prfflg0_532y.doj
#else
#define PROFFLAG prfflg0_532.doj
#endif  /* } */
// The profiler needs File I/O to write its results.
#define USE_FILEIO 1
#ifndef USE_PROFILER  /* { */
```

```
#define USE_PROFILER
#endif  /* } */
#endif  /* } */

#ifdef USE_PROFILER1   /* { */
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define PROFFLAG prfflg1_532y.doj
#else
#define PROFFLAG prfflg1_532.doj
#endif  /* } */
#define USE_FILEIO 1
#ifndef USE_PROFILER   /* { */
#define USE_PROFILER
#endif  /* } */
#endif  /* } */

#ifdef USE_PROFILER2   /* { */
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define PROFFLAG prfflg2_532y.doj
#else
#define PROFFLAG prfflg2_532.doj
#endif  /* } */
#define USE_FILEIO 1
#ifndef USE_PROFILER   /* { */
#define USE_PROFILER
#endif  /* } */
#endif  /* } */

#ifdef __WORKAROUNDS_ENABLED  /* { */
#define OMEGA idle532y.doj
#else
#define OMEGA idle532.doj
#endif  /* } */

#define MEMINIT __initsbsz532.doj,

#ifdef __WORKAROUNDS_ENABLED  /* { */
#define LIBSMALL libsmall532y.dlb,
#else
#define LIBSMALL libsmall532.dlb,
#endif  /* } */

#ifdef M3_RESERVED    /* { */
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define LIBM3 libm3res532y.dlb
#define LIBDSP libdspm3res532y.dlb
#define SFTFLT libsftflt532y.dlb
#else
#define LIBM3 libm3res532.dlb
#define LIBDSP libdspm3res532.dlb
#define SFTFLT libsftflt532.dlb
#endif  /* } */
```

```
#else
#ifdef __WORKAROUNDS_ENABLED /* { */
#define LIBM3 libm3free532y.dlb
#define LIBDSP libdsp532y.dlb
#define SFTFLT libsftflt532y.dlb
#else
#define LIBM3 libm3free532.dlb
#define LIBDSP libdsp532.dlb
#define SFTFLT libsftflt532.dlb
#endif /* } */
#endif /* } */

#ifdef IEEEFP /* { */
#define FPLIBS SFTFLT, LIBDSP
#else
#define FPLIBS LIBDSP, SFTFLT
#endif /* } */


#ifdef __WORKAROUNDS_ENABLED /* { */
#ifdef __ADI_LIBEH__
#define LIBS LIBSMALL MEMINIT libc532y.dlb, LIBM3, libevent532y.dlb, libx532y.dlb, libio532y.dlb,
libcpp532yx.dlb, libcpprt532yx.dlb, FPLIBS, libetsi532.dlb, OMEGA
#else
#define LIBS LIBSMALL MEMINIT libc532y.dlb, LIBM3, libevent532y.dlb, libx532y.dlb, libio532y.dlb,
libcpp532y.dlb, libcpprt532y.dlb, FPLIBS, libetsi532.dlb, OMEGA
#endif
#else
#ifdef __ADI_LIBEH__
#define LIBS LIBSMALL MEMINIT libc532.dlb, LIBM3, libevent532.dlb, libx532.dlb, libio532.dlb,
libcpp532x.dlb, libcpprt532x.dlb, FPLIBS, libetsi532.dlb, OMEGA
#else
#define LIBS LIBSMALL MEMINIT libc532.dlb, LIBM3, libevent532.dlb, libx532.dlb, libio532.dlb,
libcpp532.dlb, libcpprt532.dlb, FPLIBS, libetsi532.dlb, OMEGA
#endif
#endif /* } */
#if defined(USE_FILEIO) || defined(USE_PROFGUIDE)
#ifdef __WORKAROUNDS_ENABLED /* { */
$LIBRARIES = LIBS, librt_fileio532y.dlb;
#else
$LIBRARIES = LIBS, librt_fileio532.dlb;
#endif /* } */
#else
#ifdef __WORKAROUNDS_ENABLED /* { */
$LIBRARIES = LIBS, librt532y.dlb;
#else
$LIBRARIES = LIBS, librt532.dlb;
#endif /* } */
#endif /* } */

// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
```

```
#ifdef USE_PROFILER   /* { */
#ifdef USE_FILEIO     /* { */
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define CRT crtsfpc532y.doj, libprofile532y.dlb, PROFFLAG
#else
#define CRT crtsfpc532.doj, libprofile532.dlb, PROFFLAG
#endif  /* } */
#else
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define CRT crtscp532y.doj, libprofile532y.dlb, PROFFLAG
#else
#define CRT crtscp532.doj, libprofile532.dlb, PROFFLAG
#endif  /* } */
#endif   /* USE_FILEIO */      /* } */
#else
#ifdef USE_FILEIO     /* { */
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define CRT  crtsfc532y.doj
#else
#define CRT  crtsfc532.doj
#endif  /* } */
#else
#ifdef __WORKAROUNDS_ENABLED  /* { */
#define CRT  crtsc532y.doj
#else
#define CRT  crtsc532.doj
#endif  /* } */
#endif   /* USE_FILEIO */      /* } */
#endif   /* USE_PROFILER */    /* } */

#ifdef __WORKAROUNDS_ENABLED  /* { */
#define ENDCRT , crtn532y.doj
#else
#define ENDCRT , crtn532.doj
#endif  /* } */

$OBJECTS = CRT, $COMMAND_LINE_OBJECTS ,cplbtab532.doj ENDCRT;

MEMORY
{
#if 0
MEM_CORE_MMRS  {      /* Core memory-mapped registers - 2MB */
        TYPE(RAM) WIDTH(8)
        START(0xFFE00000) END(0xFFFFFFFF)
}
#endif
MEM_SYS_MMRS  {       /* System memory-mapped registers - 2MB */
        TYPE(RAM) WIDTH(8)
        START(0xFFC00000) END(0xFFDFFFFF)
}
MEM_L1_SCRATCH  {
        TYPE(RAM) WIDTH(8)
```

```
        START(0xFFB00000) END(0xFFB00FFF)
}
/* Instruction SRAM, 48K, some usable as cache */
MEM_L1_CODE_CACHE {    /* L1 Instruction SRAM/Cache - 16K */
        TYPE(RAM) WIDTH(8)
        START(0xFFA10000) END(0xFFA13FFF)
}
MEM_L1_CODE {  /* L1 Instruction SRAM - 32K */
        TYPE(RAM) WIDTH(8)
        START(0xFFA08000) END(0xFFA0FFFF)
}
/* Instruction ROM, 32K */
MEM_L1_CODE_ROM {
        TYPE(ROM) WIDTH(8)
        START(0xFFA00000) END(0xFFA07FFF)
}
/* L1 Data B bank SRAM/Cache - 16K */
/* Used for normal data - 8K of 16K */
MEM_L1_DATA_B  {
        TYPE(RAM) WIDTH(8)
        START(0xFF906000) END(0xFF907FFF)
}
/* Stack - 8K of 16K */
MEM_STACK       {
        TYPE(RAM) WIDTH(8)
        START(0xFF904000) END(0xFF905FFF)
}
/* L1 Data A bank SRAM/Cache - 16K */
/* Used for cache, if enabled. */
#ifdef USE_CACHE /* { */
MEM_L1_DATA_A_CACHE   {       /* L1 Data A SRAM/Cache - 16K */
        TYPE(RAM) WIDTH(8)
        START(0xFF804000) END(0xFF807FFF)
}
#else
#ifdef IDDE_ARGS /* { */
MEM_ARGV        {
#define ARGV_START 0xFF807E00
        TYPE(RAM) WIDTH(8)
        START(0xFF807E00) END(0xFF807FFF)
}
MEM_L1_DATA_A_HEAP        {
        TYPE(RAM) WIDTH(8)
        START(0xFF805000) END(0xFF807DFF)
}
#else
MEM_L1_DATA_A_HEAP        {
        TYPE(RAM) WIDTH(8)
        START(0xFF805000) END(0xFF807FFF)
}
#endif /* IDDE_ARGS } */
MEM_L1_DATA_A        {
```

```
        TYPE(RAM) WIDTH(8)
        START(0xFF804000) END(0xFF804FFF)
}
#endif  /* USE_CACHE } */
MEM_ASYNC3      {       /* Async Bank 3 - 1MB */
        TYPE(RAM) WIDTH(8)
        START(0x20300000) END(0x203FFFFF)
}
MEM_ASYNC2      {       /* Async Bank 2 - 1MB */
        TYPE(RAM) WIDTH(8)
        START(0x20200000) END(0x202FFFFF)
}
MEM_ASYNC1      {       /* Async Bank 1 - 1MB */
        TYPE(RAM) WIDTH(8)
        START(0x20100000) END(0x201FFFFF)
}
MEM_ASYNC0      {       /* Async Bank 0 - 1MB */
        TYPE(RAM) WIDTH(8)
        START(0x20000000) END(0x200FFFFF)
}
/* Claim some of SDRAM Bank 0 for heap */
/* since it needs a separate section */


MEM_SDRAM0_HEAP     {  /* ext heap */
        TYPE(RAM) WIDTH(8)
        START(0x00130000) END(0x007FFFFF)
}
MEM_SDRAM0      {       /* Program SDRAM - 256K */
        TYPE(RAM) WIDTH(8)
        START(0x00000004) END(0x000FFFFF)
}
}

PROCESSOR P0
{
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

        /* Following address must match start of MEM_L1_CODE */
        RESOLVE(start,0xffa08000)
#ifdef IDDE_ARGS
        RESOLVE(___argv_string, ARGV_START)
#endif
        KEEP(start,_main)

    SECTIONS
    {
        program_ram
        {
            INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(L1_code) $LIBRARIES(L1_code))
            INPUT_SECTIONS( $OBJECTS(cplb_code) $LIBRARIES(cplb_code))
```

```
            INPUT_SECTIONS( $OBJECTS(cplb) $LIBRARIES(cplb))
            INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
        } >MEM_L1_CODE

        program_rom
        {
            INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(program_rom) $LIBRARIES(program_rom))
            INPUT_SECTIONS( $OBJECTS(L1_code_rom) $LIBRARIES(L1_code_rom))
        } >MEM_L1_CODE_ROM

        l1_code
        {
#ifdef USE_CACHE /* { */
               ___l1_code_cache = 1;
#else
               ___l1_code_cache = 0;
            INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(L1_code) $LIBRARIES(L1_code))
            INPUT_SECTIONS( $OBJECTS(cplb_code) $LIBRARIES(cplb_code))
            INPUT_SECTIONS( $OBJECTS(cplb) $LIBRARIES(cplb))
            INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
#endif /* USE_CACHE } */
        } >MEM_L1_CODE_CACHE

#ifdef USE_CACHE /* { */
        bsz ZERO_INIT
        {
               INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(bsz) $LIBRARIES(bsz))
        } >MEM_SDRAM0

        bsz_init
        {
               INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(bsz_init) $LIBRARIES(bsz_init))
        } >MEM_SDRAM0
        .meminit {} >MEM_SDRAM0
#else
        bsz ZERO_INIT
        {
               INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(bsz) $LIBRARIES(bsz))
        } >MEM_SDRAM0

        bsz_init
        {
               INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(bsz_init) $LIBRARIES(bsz_init))
        } >MEM_SDRAM0
        .meminit {} >MEM_SDRAM0
#endif /* USE_CACHE */ /* } */
```

```
        data
        {
            INPUT_SECTION_ALIGN(4)
             INPUT_SECTIONS( $OBJECTS(L1_data_b) $LIBRARIES(L1_data_b))
             INPUT_SECTIONS( $OBJECTS(cplb_data) $LIBRARIES(cplb_data))
             INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
            INPUT_SECTIONS($OBJECTS(voldata) $LIBRARIES(voldata))
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(ctorl) $LIBRARIES(ctorl) )
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(.gdt) $LIBRARIES(.gdt) )
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(.gdtl) $LIBRARIES(.gdtl) )
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(.edt) $LIBRARIES(.edt) )
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(.cht) $LIBRARIES(.cht) )
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(.frt) $LIBRARIES(.frt) )
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS( $OBJECTS(.frtl) $LIBRARIES(.frtl) )
        } >MEM_L1_DATA_B
        constdata
        {
            INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
        } >MEM_L1_DATA_B

#ifdef USE_CACHE /* { */
        l1_data_a
        {
            INPUT_SECTION_ALIGN(4)
                ___l1_data_a_cache = 1;
        } >MEM_L1_DATA_A_CACHE
#else
        l1_data_a
        {
            INPUT_SECTION_ALIGN(4)
                ___l1_data_a_cache = 0;
            INPUT_SECTIONS($OBJECTS(l1_data_a) $LIBRARIES(l1_data_a))
        } >MEM_L1_DATA_A
#endif /* USE_CACHE } */

        stack
        {
             ldf_stack_space = .;
             ldf_stack_end = ldf_stack_space + MEMORY_SIZEOF(MEM_STACK);
        } >MEM_STACK
```

```
#ifdef USE_CACHE /* { */
        heap
        {
            // Allocate a heap for the application
            ldf_heap_space = .;
            ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MEM_L1_DATA_A_HEAP) - 1;
            ldf_heap_length = ldf_heap_end - ldf_heap_space;
        } >MEM_L1_DATA_A_HEAP
#else
        heap
        {
            // Allocate a heap for the application
            ldf_heap_space = .;
            ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MEM_SDRAM0_HEAP) - 1;
            ldf_heap_length = ldf_heap_end - ldf_heap_space;
        } >MEM_SDRAM0_HEAP
#endif /* USE_CACHE */ /* } */

        sdram
        {

           INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS($OBJECTS(sdram0) $LIBRARIES(sdram0))
            INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
            INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
           INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
        } >MEM_SDRAM0
    }
}
/*****************************************************************/



/*****************************************************************/
/*  This file contains 3 sections:                               */
/*     1) A Pre-Init Section - this section saves off all the    */
/*             registers of the DSP.                             */
/*     2) A Init Code Section - this section is the customer     */
/*             initialization code which can be modified by the customer. */
/*             As an example, an SDRAM initialization code is supplied.   */
/*     3) A Post-Init Section - this section restores all the    */
/*             register from the stack.                          */
/*     Customers should not modify the Pre-Init and Post-Init Sections.*/
/*     The Init Code Section can be modified for application use.        */
/*****************************************************************/

#include <defBF532.h>

.section program;

/*******Pre-Init Section*********************************************/
        [--SP] = ASTAT;                     //Save Regs onto stack
```

```
        [--SP] = RETS;
        [--SP] = (r7:0);
        [--SP] = (p5:0);
        [--SP] = I0;
        [--SP] = I1;
        [--SP] = I2;
        [--SP] = I3;
        [--SP] = B0;
        [--SP] = B1;
        [--SP] = B2;
        [--SP] = B3;
        [--SP] = M0;
        [--SP] = M1;
        [--SP] = M2;
        [--SP] = M3;
        [--SP] = L0;
        [--SP] = L1;
        [--SP] = L2;
        [--SP] = L3;
/*********************************************************************/
/*******Init Code Section*********************************************/
/*******SDRAM Setup************/
Setup_SDRAM:
        P0.L = EBIU_SDRRC & 0xFFFF;
        P0.H = (EBIU_SDRRC >> 16) & 0xFFFF;         //SDRAM Refresh Rate Control Register
        R0 = 0x060E(Z);
        W[P0] = R0;
        SSYNC;

        P0.L = EBIU_SDBCTL & 0xFFFF;
        P0.H = (EBIU_SDBCTL >> 16) & 0xFFFF; //SDRAM Memory Bank Control Register
        R0 = 0x0001(Z);
        [P0] = R0;
        SSYNC;

        P0.L = EBIU_SDGCTL & 0xFFFF;
        P0.H = (EBIU_SDGCTL >> 16) & 0xFFFF; //SDRAM Memory Global Control Register
        R0.L = 0x1109;
        R0.H = 0x0091;
        [P0] = R0;
        SSYNC;
/*****************************/
/*********************************************************************/
/*******Post-Init Section*********************************************/
        L3 = [SP++];
        L2 = [SP++];
        L1 = [SP++];
        L0 = [SP++];
        M3 = [SP++];
        M2 = [SP++];
        M1 = [SP++];
        M0 = [SP++];
```

```
        B3 = [SP++];
        B2 = [SP++];
        B1 = [SP++];
        B0 = [SP++];
        I3 = [SP++];
        I2 = [SP++];
        I1 = [SP++];
        I0 = [SP++];
        (p5:0) = [SP++];              //Restore Regs from Stack
        (r7:0) = [SP++];
        RETS = [SP++];
        ASTAT = [SP++];
/*****************************************************************/
END:    RTS;
/*****************************************************************/



/*****************************************************************/
//PLL Setup
void Init_PLL(void)
{
        sysreg_write(reg_SYSCFG, 0x32);             //Initialize System Configuration Register
        *pPLL_DIV = 0x0024;    //PLL Divide register - CCLK = VCO/4 - SCLK = VCO/4 -> 99,5328 MHz
        *pSIC_IWR = 0x1;       //Interrupt Wake-up Register set first bit for PLL wake-up
        *pPLL_CTL = 0x4800;    //turn on PLL and set VCO frequency at 36x  100100
        ssync();
        idle();
}//end Init_PLL
/*****************************************************************/


/*****************************************************************/
void Init_DMA(unsigned char *acq_img_pointer)
{
        //Target address of the DMA
        (*(unsigned int * *)pDMA0_START_ADDR) =     (unsigned int *)acq_img_pointer;
        //number of transfers will be executed
        *pDMA0_X_COUNT = righe;
        //number of transfers will be executed
        *pDMA0_Y_COUNT = colonne;
        //The modifier is set to 2 because of the 16bit transfers
        *pDMA0_X_MODIFY = 0x2;
        *pDMA0_Y_MODIFY = 0x2;
        //PPI Peripheral is used
        *pDMA0_PERIPHERAL_MAP = 0x0;

        //DMA Config: Enable DMA | Memory write DMA | Discard DMA FIFO before start | enable
        assertation of interrupt | NDSIZE for stop mode | Enable STOP DMA
        *pDMA0_CONFIG = DMAEN | DI_EN | WNR | WDSIZE_16 | RESTART | DMA2D;
}//end Init_DMA
/*****************************************************************/
```

```
/*********************************************************************/
void Init_PPI(void)
{
        //The PPI is set to receive 576 lines for each frame of data
        *pPPI_FRAME=RIGHE_PAL;
        //PPI enabled, input mode, active video only, receive field 1&2,
        //packing enabled, DMA32 enabled, skipping disabled, 8bit data bus, nothing inverted
        *pPPI_CONTROL = PORT_EN | FLD_SEL | PACK_EN | DLEN_8;
}//end Init_PPI
/*********************************************************************/


/*********************************************************************/
//Setup of the async interface
void Init_EBIU(void)
{
        //|Write access time = 3 cycles, read access time = 3 cycles, no ARDY
        //|Hold time = 0 cycles, setup time = 2 cycles, transition time = 1 cycles
        *pEBIU_AMBCTL1 = 0x33243324;
        //|Enable all memory banks
        *pEBIU_AMGCTL  = 0x000F;
}//end Init_EBIU
/*********************************************************************/



/*********************************************************************/
void Init_ProgrammableFlags(void)
{

        /*Decoder Video */

        //RESET
        *pFIO_EDGE &= ~RESET_DV;      // Enable Level Sensitivity
        *pFIO_DIR |=  RESET_DV;              // Set RESET as output
        *pFIO_FLAG_D &= ~RESET_DV;  // Set Reset as LOW

        //SDATA
        *pFIO_POLAR &= ~SDATA;        // Enable Active High level
        *pFIO_EDGE &= ~SDATA;         // Enable Level Sensitivity
        *pFIO_INEN |=  SDATA;         // Enable SDATA Input Buffer
        *pFIO_DIR |=  SDATA;          // Set SDATA as output

        //SCLK
        *pFIO_EDGE &= ~SCLK;          // Enable Level Sensitivity
        *pFIO_DIR |=  SCLK;                   // Set SCLK as output

        //SCLKI
        *pFIO_POLAR &= ~SCLKI;        // Enable Active High level
```

```
        *pFIO_EDGE &= ~SCLKI;         // Enable Level Sensitivity
        *pFIO_INEN |=  SCLKI;         // Enable SCLKI Input Buffer
        *pFIO_DIR &=  ~SCLKI;         // Set SCLKI as input

        //I2CSEL
        *pFIO_EDGE &= ~SEL;                   // Enable Level Sensitivity
        *pFIO_DIR |=  SEL;                    // Set SEL as output
        *pFIO_FLAG_D &=  ~SEL;        // Set SEL low

        /* GPIO */

        //PF3 Camera 1
        *pFIO_EDGE &= ~PF3;                   // Enable Level Sensitivity
        *pFIO_DIR |=  PF3;                    // Set PF3 as output
        *pFIO_FLAG_D &=  ~PF3;        // Set PF3 low

        //PF4 Camera 2
        *pFIO_EDGE &= ~PF4;                   // Enable Level Sensitivity
        *pFIO_DIR |=  PF4;                    // Set PF4 as output
        *pFIO_FLAG_D &=  ~PF4;        // Set PF4 low

        //PF5 Camera 3
        *pFIO_EDGE &= ~PF5;                   // Enable Level Sensitivity
        *pFIO_DIR |=  PF5;                    // Set PF5 as output
        *pFIO_FLAG_D &=  ~PF5;        // Set PF5 low

        //PF7
        *pFIO_EDGE &= ~PF7;                   // Enable Level Sensitivity
        *pFIO_INEN |=  PF7;           // Enable PF7 Input Buffer
        *pFIO_DIR &=  ~PF7;                   // Set PF7 as input
}
/*************************************************************************/



/*************************************************************************/
//UART Setup
void Init_UART(void)
{
               //Enable UART clock
               *pUART_GCTL = 0x1;
                //8 bit word length, 1 bit stop, no parity, enable access to divisor
               *pUART_LCR = 0x0083;
               //64 divisor to have 115200 baud rate with CLKIN of 11,0592MHz,VCO 36x and SCLK=VCO/4
               *pUART_DLL = 0x0036;
               *pUART_DLH = 0x0000;
               //8 bit word length, 1 bit stop, no parity
               *pUART_LCR = 0x0003;
               //Enable Receive Buffer Full Interrupt
               *pUART_IER = ERBFI | ETBEI;
}
/*************************************************************************/
```

```
/*****************************************************************/
#include <ccblkfn.h>
#include <sys/exception.h>
#include <string.h>
#include "payload.h"

/* UART related macros */

static unsigned char uart_rxbuffer[UART_RXBUFFER_SIZE];
static unsigned char uart_txbuffer[2][UART_TXBUFFER_SIZE];
static volatile int txbuf_ready[2] = {0, 0};
static volatile int intr_txbuf_select = 0;
static volatile int write_txbuf_select = 0;
static volatile int txbuf_writing = FALSE;
static volatile int intr_rxubp = 0;
static volatile int intr_txubp = 0;
static volatile int read_rxubp = 0;

extern int flag;

EX_INTERRUPT_HANDLER(DMA0_PPI_ISR)   // <--|declaration in  exception.h -->
{                                    //    |declaration with _pragma(interrupt) the ISR Startaddress
        //disable the interrupt request
        *pDMA0_IRQ_STATUS = 0x1;
        flag=0;
}//end DMA0_PPI_ISR

EX_INTERRUPT_HANDLER(UART_RX_ISR)    // <--|declaration in  exception.h -->
{                                    //    |declaration with _pragma(interrupt) the ISR Startaddress
        //Interrupt Identification Register
        //Status Receive data ready
        *pUART_IIR = 0x5;
        uart_rxbuffer[intr_rxubp++] = *pUART_RBR;
        intr_rxubp &= (UART_RXBUFFER_SIZE - 1);
}//end UART_RX_ISR

/* Interrupt service routine for UART writing */
EX_INTERRUPT_HANDLER(UART_TX_ISR)
{
        unsigned char temp;

        /* Clear all pending interrupts (legacy support) */
        temp = *pUART_IIR;
        if (txbuf_ready[intr_txbuf_select]) {
                txbuf_writing = TRUE;
                *pUART_THR = uart_txbuffer[intr_txbuf_select][intr_txubp++];
                txbuf_ready[intr_txbuf_select]--;
                if (txbuf_ready[intr_txbuf_select] == 0) {
                        intr_txbuf_select = 1 - intr_txbuf_select;
```

```
                        intr_txubp = 0;
                }
        } else {
                txbuf_writing = FALSE;
        }
}

void Init_Interrupts(void)
{
        // configure interrupt
        *pSIC_IAR0 = *pSIC_IAR0 & 0xffffffff | 0x00000000;
        *pSIC_IAR1 = (*pSIC_IAR1 & 0x00fffff0) | 0x43000001;       // map DMA0 PPI Interrupt -> IVG8
                                                                   // map UART RX Interrupt -> IVG10
                                                                   // map UART TX Interrupt -> IVG11
        *pSIC_IAR2 = *pSIC_IAR2 & 0xffffffff | 0x00000000;
        register_handler(ik_ivg8, DMA0_PPI_ISR);     // assign DMA0 PPI ISR to interrupt vector 8
        register_handler(ik_ivg10, UART_RX_ISR);     // assign UART RX to interrupt vector 10
        register_handler(ik_ivg11, UART_TX_ISR);     // assign UART TX to interrupt vector 11
        *pSIC_IMASK=0x0000C100; // DMA0 PPI interrupt 1 (bit 8)- UART RX (bit 14) E UART TX (bit 15)
        - enabled
}//end Init_Interrupts

/* Blocking read: the function does not return until all requested
 * data is stored in the buffer */
int read_uart_blocking(unsigned char * buffer, int count)
{
        int i;
        i = 0;

        while (count > 0) {
                if (read_rxubp != intr_rxubp) {
                        buffer[i++] = uart_rxbuffer[read_rxubp++];
                        read_rxubp &= (UART_RXBUFFER_SIZE - 1);
                        count--;
                }
        }
    return i;
}

/* Blocking read of one byte */
unsigned char read_uart_byte()
{
        unsigned char data;

        /* Wait for data */
        while (read_rxubp == intr_rxubp) {}
        /* Read data */
        data = uart_rxbuffer[read_rxubp++];
        read_rxubp &= (UART_RXBUFFER_SIZE - 1);
        return data;
}
```

```
/* Non blocking read: the function returns as soon as thre is no more
 * data available, or when count bytes are received */
int read_uart_nonblocking(unsigned char * buffer, int count)
{
        int i;
        i = 0;

        while (count > 0 && read_rxubp != intr_rxubp) {
                buffer[i++] = uart_rxbuffer[read_rxubp++];
                read_rxubp &= (UART_RXBUFFER_SIZE - 1);
                count--;
        }
        return i;
}

/* Returns the nunber of available data */
int uart_select()
{
        int diff;

        diff = intr_rxubp - read_rxubp;
        return ((diff >= 0 ? diff : diff + UART_RXBUFFER_SIZE) & (UART_RXBUFFER_SIZE - 1));
}

/* Blocking write: sends count bytes from buffer. Returns number of byte
   sent */
int write_uart_blocking(unsigned char * buffer, int count)
{
        unsigned save_mask;
        int n, i = 0;
        unsigned char data, t;

        while (count > 0) {
                /* Wait for a buffer to become ready */
                while (txbuf_ready[write_txbuf_select]) {}
                /* Copy buffer to be transferred */
                n = (count > UART_TXBUFFER_SIZE ? UART_TXBUFFER_SIZE : count);
                memcpy(uart_txbuffer[write_txbuf_select], buffer + i, (size_t) n);
                count -= n;
                i += n;
                save_mask = cli();
                txbuf_ready[write_txbuf_select] = n;
                /* Start transfer if not already started */
                if (!txbuf_writing && intr_txbuf_select == write_txbuf_select && intr_txubp == 0) {
                        if ((*pUART_LSR & 0x0020) != 0) {
                                data = uart_txbuffer[intr_txbuf_select][intr_txubp++];
                                txbuf_ready[intr_txbuf_select]--;
                                if (txbuf_ready[intr_txbuf_select] == 0) {
                                        intr_txbuf_select = 1 - intr_txbuf_select;
                                        intr_txubp = 0;
                                }
                                sti(save_mask);
```

```
                                t = *pUART_IIR;
                                *pUART_THR = data;
                        } else {
                                sti(save_mask);
                        }
                } else {
                        sti(save_mask);
                }
                write_txbuf_select = 1 - write_txbuf_select;
        }
        return i;
}

/* Blocking write of a single byte */
void write_uart_byte(unsigned char data)
{
        (void) write_uart_blocking(&data, 1);
        return;
}

/* Wait the end of a write transmission */
void write_uart_sync(void)
{
        while (txbuf_writing) asm("nop;");
        return;
}
/*********************************************************************/



/*********************************************************************/
#include <stdlib.h>
#include "jpeg/jpeglib.h"
#include "payload.h"

//prototypes
int DeInterleave(int,int,JSAMPLE *,unsigned char *);       //de-interlacciamento
unsigned char *jpeg_main(int,int,JOCTET *,JSAMPLE *);      //jpeg compressor
void save_map_flash(int,int,unsigned char*);               //to save image in flash and keep its map
void write_map_flash();                                    //to save the flash map into itself

//global variables
extern unsigned char *seg_table[10],*flash_addr;
int flag=1;

#define ONESECOND 2000000

void Acq_Image(char nimage,char camera_number, char memory_destination)
{
        unsigned char internal_flag;
        char uart_msg[128];
        int x=0,y=0,i,block;                //counters
```

```
JSAMPLE * image_buffer;             /* Points to large array of R,G,B-order data */
JOCTET * final_buffer;               /* Points to the final destination buffer */
unsigned char *acq_img_pointer,*temp_byte_read;
int image_start_blocks[5] = {8,12,16,20,24};
int image_end_blocks[5] = {11,15,19,23,27};

Init_SDRAM();
Init_EBIU();                        //Async Memory Interface Setup
Init_ProgrammableFlags();

//Before calling Init_Decoder to exit from PowerDown Mode the Decoder
switch (camera_number){
       case 1:  *pFIO_FLAG_D |= PF3;
       break;
       case 2:  *pFIO_FLAG_D |= PF4;
       break;
       case 3:  *pFIO_FLAG_D |= PF5;
       break;
}
Init_Decoder();
switch (memory_destination)          //select the image destination memory
{
       case 1:  acq_img_pointer = (unsigned char *)0x20200000;       //SRAM
       break;
       case 2:  acq_img_pointer = (unsigned char *)0x00040008;       //SDRAM
       break;
}
image_buffer = (JSAMPLE *) malloc((size_t) SEGMENT_BUF_SIZE);
final_buffer = (JOCTET *) malloc((size_t) FINAL_BUF_SIZE);
//waiting for PAL interrupt
while((*pFIO_FLAG_D & PF7)==0) {}
sprintf(uart_msg, "\n\rAcq_img_pointer address: %x\n\r",acq_img_pointer);
write_uart_blocking((unsigned char *) uart_msg, strlen(uart_msg));
write_uart_sync();
Init_DMA(acq_img_pointer);    //DMA Setup "PPI->DMA->SDRAM" It's configured in Stop Mode
Init_PPI();                   //PPI Setup

internal_flag = 1;
do
{
       if((flag==0)||(*pDMA0_CURR_ADDR == (volatile void *)0x20400002)){
              sprintf(uart_msg, "\n\rImage Acquired!!\n\r");
              write_uart_blocking((unsigned char *) uart_msg, strlen(uart_msg));
              write_uart_sync();
              internal_flag=0;
       }
}while(internal_flag==1);
internal_flag = 1;

/* Turn off the camera*/
switch (camera_number){
       case 1:  *pFIO_FLAG_D &= ~PF3;
```

```
                break;
                case 2:  *pFIO_FLAG_D &= ~PF4;
                break;
                case 3:  *pFIO_FLAG_D &= ~PF5;
                break;
        }

        /* Setup flash */
        write_error(SetupForFlash());
        ResetFlash();
        sprintf(uart_msg,   "Erasing   blocks   %02d   -   %02d\n\r",   image_start_blocks[nimage],
        image_end_blocks[nimage]);
        write_uart_blocking((unsigned char *) uart_msg, strlen(uart_msg));
        write_uart_sync();
        for (block = image_start_blocks[nimage]; block <= image_end_blocks[nimage]; block++) {
                sprintf(uart_msg, "%02d . ", block);
                write_uart_blocking((unsigned char *) uart_msg, strlen(uart_msg));
                write_uart_sync();
                write_error(EraseBlock(block));
                for (i = 0; i < ONESECOND * 10; i++) asm("nop;");
        }
        seg_table[0] = ((unsigned char *) FLASH_BASE_ADDRESS_IMAGE_SECTION) +
                    (10 * sizeof(unsigned char *)) +
                    (nimage * BLOCK_SPACE_MEM);
        flash_addr = seg_table[0];
        for (x = 0; x < 3; x++)
        {
                for (y = 0; y < 3; y++)
                {
                        DeInterleave(x,y,image_buffer,(unsigned char *)acq_img_pointer);
                        ResetFlash();
                        for (i = 0; i < ONESECOND * 10; i++) asm("nop;");
                        temp_byte_read=jpeg_main(x,y,final_buffer,image_buffer);
                        save_map_flash(x,y,temp_byte_read);
                }//end for y
        }//end for x
        write_map_flash();
}//end
/*********************************************************************/



/*********************************************************************/
int DeInterleave(int x,int y,JSAMPLE *image_buffer,unsigned char * acq_img_pointer)
{
        static unsigned char *punt_1;        //pointer SDRAM field 1
        static unsigned char *punt_2;        //pointer SDRAM field 2
        int i,l,j;                           //counters
        int index=0;                         //counter image buffer
        char k = 0x1;                        //flag that keep a odd or even value
        unsigned char cb0y0cr0y1[4];         //temporary vector for luminance and chrominance
```

```
if(y==0)                            //refresh pointer after 3 segments
{
        punt_1 = acq_img_pointer + (righe_b * x);
        punt_2 = acq_img_pointer + 0x65400 + (righe_b * x);
}
for(j=0;j<colonne_b;j++)            //for all the rows (odd and even)
{
        if(!(k&0x1))                //if k is odd or even
        {
                i=0;
                for(l=0;l<righe_b;l++)                      //for all the byte in a row
                {
                        cb0y0cr0y1[i++] = *(punt_2++);      //save old configuration
                        if (!(i &= 0x3)) //set new configuration with the third chrominance
                        {
                                image_buffer[index++] = cb0y0cr0y1[1];// Y0
                                image_buffer[index++] = cb0y0cr0y1[2];// Cr
                                image_buffer[index++] = cb0y0cr0y1[0];// Cb
                                image_buffer[index++] = cb0y0cr0y1[3];// Y1
                                image_buffer[index++] = cb0y0cr0y1[2];// Cr
                                image_buffer[index++] = cb0y0cr0y1[0];// Cb
                        }//end if
                }//end for
        }
        else
        {
                i=0;
                for(l=0;l<righe_b;l++)
                {
                        cb0y0cr0y1[i++] = *(punt_1++);
                        if (!(i &= 0x3))
                        {
                                image_buffer[index++] = cb0y0cr0y1[1];// Y0
                                image_buffer[index++] = cb0y0cr0y1[2];// Cr
                                image_buffer[index++] = cb0y0cr0y1[0];// Cb
                                image_buffer[index++] = cb0y0cr0y1[3];// Y1
                                image_buffer[index++] = cb0y0cr0y1[2];// Cr
                                image_buffer[index++] = cb0y0cr0y1[0];// Cb
                        }//end if
                }//end for
        }//end if
        if(!(k&0x1))          //if k is odd or even
        {
                punt_2 += 0x3c0;            //go to the next line
        }
        else
        {
                punt_1 += 0x3c0;            //go to the next line
        }//end if
        k = (k + 1)&0x1;            //sum 1 and mask only the first bit
}//end 2nd for
if(y==2)
```

```
	{
		punt_1 = (unsigned char *)acq_img_pointer;
		punt_2 = (unsigned char *)acq_img_pointer + 0x65400;
	}
		return 1;
}//end



/*****************************************************************/
#include <stdio.h>
#undef FILEOUT
#undef MATHIMAGE
#undef MYUSAGE
#define COMP
#include <cdefBF533.h>	//BF533 Register Pointer Definition
#include <stdlib.h>
#include <time.h>
#ifdef MYUSAGE
#include <sys\time.h>
#include <resource.h>
#include <unistd.h>
#include <stdio.h>
#endif
#ifdef MATHIMAGE
#include <math.h>
#endif
#include "jpeglib.h"
#include "payload.h"

//global variables
int segment_length;   //counter of # of bytes of final image

unsigned char *jpeg_main(int xx,int yy,JOCTET * final_buffer,JSAMPLE * image_buffer)
{
	int quality = 75;
	struct jpeg_compress_struct cinfo;
	struct jpeg_error_mgr jerr;
	JSAMPROW row_pointer[1];	/* pointer to JSAMPLE row[s] */
	int row_stride;		/* physical row width in image buffer */
	int x, y, i;
	double val;
	unsigned char * flash_addr;
	int image_height = 192;	/* Number of rows in a image segment */
	int image_width = 240;	/* Number of columns in a image segment */
	clock_t c1,c2;

	#ifdef FILEOUT
	FILE * fout;
	#endif

	#ifdef MYUSAGE
```

```
struct rusage myusage1;
struct rusage myusage2;
struct rusage myusage3;
struct rusage myusage4;
struct rusage myusage5;
float val1, val2;
#endif

#ifdef MYUSAGE
getrusage(RUSAGE_SELF, &myusage1);
#endif

c1 = clock();
// Step 1: allocate and initialize JPEG compression object
cinfo.err = jpeg_std_error(&jerr);
// Now we can initialize the JPEG compression object.
jpeg_create_compress(&cinfo);
// Step 2: specify data destination (eg, a file)
jpeg_mem_dest(&cinfo, final_buffer, FINAL_BUF_SIZE);
// Step 3: set parameters for compression
cinfo.image_width = image_width;     // image width and height, in pixels
cinfo.image_height = image_height;
cinfo.input_components = 3;          // # of color components per pixel
cinfo.in_color_space = JCS_YCbCr;    // colorspace of input image
jpeg_set_defaults(&cinfo);
jpeg_set_quality(&cinfo, quality, TRUE /* limit to baseline-JPEG values*/ );
// Step 4: Start compressor
jpeg_start_compress(&cinfo, TRUE);
// Step 5: while (scan lines remain to be written)
row_stride = image_width * 3; // JSAMPLEs per row in image_buffer

#ifdef MYUSAGE
getrusage(RUSAGE_SELF, &myusage3);
#endif

while (cinfo.next_scanline < cinfo.image_height) {
       row_pointer[0] = & image_buffer[(cinfo.next_scanline * row_stride)];
       (void)jpeg_write_scanlines(&cinfo, row_pointer, 1);
}
#ifdef MYUSAGE
  getrusage(RUSAGE_SELF, &myusage4);
#endif

// Step 6: Finish compression
jpeg_finish_compress(&cinfo);
segment_length = jpeg_mem_dest_getcount(&cinfo);
// Step 7: release JPEG compression object
jpeg_destroy_compress(&cinfo);

c2 = clock();
#ifdef FILEOUT
```

```
        if ((fout = fopen("C:\\Users\\MCaule\\JPEG\\files-modificati\\jpeg-6b\\pluto2.jpg", "wb")) ==
        NULL) {
                fprintf(stderr, "Unable to open output file\n");
                exit(1);
        }
        for (i = 0; i < segment_length; i++)
        {
                fputc(final_buffer[i], fout);
        }
        fclose(fout);
        #endif
        //      printf("\n c1 = %ld\n",c1);
        //      printf("\n c2 = %ld\n",c2);
return final_buffer;
}
/*********************************************************************/



/*********************************************************************/
#define BLOCKSIZE 128
#define TRUE 1

extern unsigned char *seg_table[10],*flash_addr;           //vector pointer
extern int segment_length;                  //# of bytes current segment

void save_map_flash(int subx,int suby,unsigned char *final_buffer)
{
        char uart_msg[128];
        unsigned long count,ndata;
        int i,tries;
        ERROR_CODE ercode;

        for (count = 0; count < segment_length; count += BLOCKSIZE) {
                ndata = (segment_length - count > BLOCKSIZE ? BLOCKSIZE : segment_length - count);
                tries = 0;
                do {
                        ercode = WriteData((unsigned long)flash_addr+count, (long) ndata, final_buffer
                + count, TRUE);
                tries++;
                if (!(tries % 50) && ercode != NO_ERR) {
                        sprintf(uart_msg, "Still trying (count=%d tries=%d)\n\r", count, tries);
                        write_uart_blocking((unsigned char *) uart_msg, strlen(uart_msg));
                        write_uart_sync();
                        write_error(ercode);
                }
                } while (ercode != NO_ERR);
        }//end for
        flash_addr +=  segment_length;
        seg_table[subx*3+suby+1] = flash_addr;
}//end
```

```
void write_map_flash()
{
	int i;
	unsigned long ndata;
	ERROR_CODE ercode;

	/* Write segment table in flash */
	flash_addr = seg_table[0] - (10 * sizeof(unsigned char *));
	ndata = 10 * sizeof(unsigned char *);
	do {
		ercode = WriteData((unsigned long)flash_addr, (long) ndata, (unsigned char
	*)seg_table, TRUE);
	} while (ercode != NO_ERR);

/*	for (i = 0; i < 10; i++) {
		*((unsigned char * *) flash_addr + i) = seg_table[i];
	}
}//end
/*********************************************************************/



/*********************************************************************/
void Init_UART(void);
void Init_DMA_UART(void);
int Xmodem_addrs(unsigned char * addr1, unsigned char * addr2)

void Tx(int nimage)
{
	char x;
	unsigned char **flash_table;		//pointer to flash table saved on flash
	unsigned char *seg1_table[10];		//image pointers flash table on internal memory

	Init_UART();		//UART Setup
	flash_table = (unsigned char * *)((unsigned char *)FLASH_BASE_ADDR + (nimage *
BLOCK_SPACE_MEM));
	for (x = 0; x < 10; x++)
	{
		seg1_table[x] = *(flash_table++);
	}
	for (x = 0; x < 10; x++)
	{
		Xmodem_main(seg1_table[x],seg1_table[x+1]);
	}
}//end main
/*********************************************************************/



/*********************************************************************/
/* Xmodem related macros */
#define UART_NACK 0x15
```

```
#define UART_ACK 0x06
#define UART_SOH 0x01
#define UART_EOT 0x04
#define UART_PADDING 0x4d
/* UART related macros */
#define UART_RXBUFFER_SIZE 16 /* Must be a power of 2 */
#define UART_TXBUFFER_SIZE 150
#define TRUE 1
#define FALSE 0
#define ONESECOND 2000000

/* Function prototypes */
int Xmodem_count(int count, unsigned char * addr);
int Xmodem_addrs(unsigned char * addr1, unsigned char * addr2);
int Xmodem_receive(int maxcount, unsigned char * addr);
extern int write_uart_blocking(unsigned char * buffer, int count);
extern int read_uart_blocking(unsigned char * buffer, int count);
extern unsigned char read_uart_byte(void);
extern void write_uart_byte(unsigned char data);

int Xmodem_count(int count, unsigned char * addr)
{
        unsigned char crc, block[132];
        int j, x, nblock;

        /* Wait NACK */
        while (read_uart_byte() != UART_NACK) {}
        nblock = (count) / 128 + 2;
        for (j = 1; j <= nblock; j++) {
                crc = 0;
                /* Store SOH */
                block[0] = UART_SOH;
                /* Store blocknumber and ~blocknumber */
                block[1] = j & 0xff;
                block[2] = 0xff - block[1];
                /* Store block data */
                for (x = 0; x < 128; x++) {
                        if (j == nblock && ((j - 1) * 128 + x) > count) {
                                block[x + 3] = UART_PADDING;
                        } else {
                                block[x + 3] = *(addr++);
                        }
                crc += block[x + 3];
                }
        /* Store crc */
        block[x + 3] = crc;
        /* Send packet */
        do {
                write_uart_blocking(block, 132);
        } while (read_uart_byte() == UART_NACK);
        }
        /* Send EOT */
```

```
        do {
                write_uart_byte(UART_EOT);
        } while (read_uart_byte() == UART_NACK);
        return 0;
}

int Xmodem_addrs(unsigned char * addr1, unsigned char * addr2)
{
        int count;
        count = addr2 - addr1;
        return(Xmodem_count(count, addr1));
}

/*
 * Stores at most maxcount bytes into buffer pointed by addr
 * using xmodem protocol.
 * Returns the number of actual bytes received (might be less,
 * equal or greater than maxcount).
 */
int Xmodem_receive(int maxcount, unsigned char * addr)
{
        unsigned char data, crc, block[132];
        int n, i;
        n = 0;
        i = 0;

        write_uart_byte(UART_NACK);
        while (uart_select() == 0) {
                for (i = 0; (i < ONESECOND * 5) && (uart_select() == 0); i++) {
                        asm("nop;");
                }
                if (uart_select() == 0) {
                        write_uart_byte(UART_NACK);
                }
        }
        while ((data = read_uart_byte()) != UART_EOT) {
                if (data != UART_SOH) {
                /* Unexpected data, bail out */
                        return -1;
                }
                block[0] = data;
                crc = 0;
                read_uart_blocking(block + 1, 131);
                for (i = 0; i < 128; i++) {
                        crc += block[i + 3];
                }
                if (crc != block[131]) {
                        write_uart_byte(UART_NACK);
                } else {
                        if (maxcount >= n + 128) {
                                memcpy(addr + n, block + 3, 128);
                        } else if (maxcount > n) {
```

```
                                memcpy(addr + n, block + 3, maxcount - n);
                                } else {
                                        /* Do nothing here */
                                }
                        n += 128;
                        write_uart_byte(UART_ACK);
                        }
                }
                return n;
}
/*********************************************************************/



/*********************************************************************/
//Functions Prototypes
bool I2C_Write(unsigned char, unsigned char);
bool I2C_Read(unsigned char address); //"value" by reference bring the read value
extern void delay(int nsec);
extern int write_uart_blocking(unsigned char * buffer, int count);
extern void write_uart_byte(unsigned char data);
extern void write_uart_sync(void);

unsigned char value;

void Init_Decoder(void)
{
        char uart_msg[128];
        unsigned char value_reg;
        int i;

        I2C_Init();                                                     // Initialize I2C port
        value_reg = MISC_CNTL_REG_DEF | P_YCBCR;
        I2C_Write(MISC_CNTL_REG_ADDR,value_reg);    // num. of bytes to write, and address
        delay(4000);
        I2C_Init();
        I2C_Write(0xc2,0x07); // num. of bytes to write, and address
        delay(4000);
        I2C_Init();
        I2C_Write(0xc1,0x40); // num. of bytes to write, and address
}

//-------------------------------------------------------------------------
// I2C Peripheral Function Prototypes
//-------------------------------------------------------------------------
void I2C_Init(void);                //  Initialize I2C port
void Start(void);                   //  Sends I2C Start Trasfer
void Stop(void);                    //  Sends I2C Stop Trasfer
bool Write(unsigned char);          //  Writes data over the I2C bus
bool Read(bool);                    // Reads data from the I2C bus
void SetSCLK(bool);                 //  Set SCLK to <state>
void SetSDATA(bool);                //  Set SDATA to <state>
```

```
void SetSDIR(bool);                  //  Set DIR to <state>
bool GetSDATA();                     //  Get SDATA state
void DeSetSEL(void);                 //  Set PF12 like PPI7 as input
void delay(int nsec);                //  Delay software
void SetRESET(bool state);
//---------------------------------------------------------------------------
// I2C Peripheral Variables
//---------------------------------------------------------------------------
unsigned char rd_cnt;
unsigned char i2c_rd_cnt;
extern unsigned char value;
//---------------------------------------------------------------------------
// Procedure:  I2C_Write
// Inputs:            num_bytes, address
// Outputs:           bool
// Description:       Writes a byte to the given address and return status.
//---------------------------------------------------------------------------
bool I2C_Write(unsigned char address, unsigned char i2c_rd_wr_buffer)
{
       Start();                      // Send start signal
       if (!Write(IdentAddr))        // Send identifier I2C address
       {
              Stop();                // Send I2C Stop Transfer
              return false;
       }
       if (!Write(address))          // Send address to device
       {
              Stop();                // Send I2C Stop Transfer
              return false;
       }
       if (!Write(i2c_rd_wr_buffer)) // Send byte to device
       {
              Stop();                // Send I2C Stop Transfer
              return false;
       }
       Stop();                       // Send I2C Stop Transfer
       return true;
}
//---------------------------------------------------------------------------
// Procedure:  I2C_Read
// Inputs:            num_bytes, address
// Outputs:           bool
// Description:       Reads a byte from the given address and return status.
//---------------------------------------------------------------------------
bool I2C_Read(unsigned char address)
{
       Start();                      // Send start signal
       if (!Write(IdentAddr))        // Send identifer I2C address
       {
              Stop();                // Send I2C Stop Transfer
              return false;
       }
```

```
	if (!Write(address))		// Send address to device
	{
		Stop();			// Send I2C Stop Transfer
		return false;
	}
	Start();			// Send I2C Start Transer
	if (!Write(IdentAddr+1))	// Send identifer I2C address
	{
		Stop();			// Send I2C Stop Transfer
		return false;
	}
	if (!Read(true))		// Read byte
	{
		Stop();			// Send I2C Stop Transfer
		return false;
	}
	Stop();				// Send I2C Stop Transfer
	return true;
}
//-------------------------------------------------------------------------------
// I2C Functions - Master
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//	Routine:	I2C_Init
//	Inputs:		none
//	Outputs:	none
//	Purpose:	Initialize I2C for the ADV7179/83
//-------------------------------------------------------------------------------
void I2C_Init(void)
{
	SetRESET(NORM_MODE);		//Set to 1 the Reset pin
	SetSCLK(1);			// Set SCLK high
	SetSDATA(1);			// Set SDATA high
	DeSetSEL();			// free YOUT7
}
//-------------------------------------------------------------------------------
//	Routine:	Start
//	Inputs:		none
//	Outputs:	none
//	Purpose:	Sends I2C Start Transfer - "S"
//-------------------------------------------------------------------------------
void Start(void)
{
	SetSCLK(1);			// Set SCLK high
	SetSDATA(0);			// Set SDATA output/low
	SetSCLK(0);			// Set SCLK low
}
//-------------------------------------------------------------------------------
//	Routine:	Stop
//	Inputs:		none
//	Outputs:	none
//	Purpose:	Sends I2C Stop Transfer - "P"
```

```
//---------------------------------------------------------------------------
void Stop(void)
{
	SetSDIR(1);			// Set to write mode
	SetSDATA(0);			// Set SDATA low
	SetSCLK(1);			// Set SCLK high
	delay(200);
	SetSDATA(1);			// Set SDATA high
}
//---------------------------------------------------------------------------
//	Routine:	Write
//	Inputs:		data_out
//	Outputs:	bool
//	Purpose:	Writes data over the I2C bus and return status.
//---------------------------------------------------------------------------
bool Write(unsigned char data_out)
{
	unsigned char index;
	// An I2C output byte is bits 7-0 (MSB to LSB). Shift one bit at a time to
	// the SDATA output, and then clock the data to the I2C Slave device.
	// Send 8 bits out the port
	for(index = 0; index < 8; index++)
	{
		// Output the data bit to the device
		SetSDATA(((data_out & 0x80) ? 1 : 0));
		data_out <<= 1;		// Shift the byte by one bit
		SetSCLK(1);		// Set SCLK high
		SetSCLK(0);		// Set SCLK low
	}
	SetSDIR(0);			// Set SDATA input/high
	SetSCLK(1);			// Set SCLK high
	if (!GetSDATA())
	{
		SetSCLK(0);		// Set SCLK low
		SetSDIR(1);		// Set SDATA output
		return true;		// ACK from slave
	} else
	{
		SetSCLK(0);		// Set SCLK low
		return false;		// NACK from slave
	}
}
//---------------------------------------------------------------------------
//	Routine:	Read
//	Inputs:		*data_in, send_ack (if true send the ACK signal else send NACK)
//	Outputs:	bool
//	Purpose:	Reads data from the I2C bus and return it in data_in.
//			Returns status.
//---------------------------------------------------------------------------
bool Read(bool send_ack)
{
	unsigned char index;
```

```
//      data_in = 0x00;
        SetSCLK(0);                     // Set SCLK low
        SetSDIR(0);                     // Set SDATA input/high

        // Get 8 bits from the device
        for(index = 0; index < 8; index++)
        {
                value <<= 1;            // Shift the data right 1 bit
                SetSCLK(1);             // Set SCLK high
                value |= GetSDATA();    // Read the data bit
                SetSCLK(0);             // Set SCLK low
        }
        if(send_ack)
        {
                SetSDIR(1);             // Set SDATA output
                SetSDATA(0);            // Set data pin low to ACK the read
        }
        else
        {
                SetSDIR(1);             // Set SDATA output
                SetSDATA(1);            // Set data pin high to NACK the read
        }
        SetSCLK(1);                     // Set SCLK high
        SetSCLK(0);                     // Set SCLK low
        SetSDIR(0);                     // Set SDATA input/high
        return true;
}
//-------------------------------------------------------------------------------
//      Routine:        SetSDIR
//      Inputs:         state
//      Outputs:        none
//      Purpose:        Set the I2C port SDATA pin to <state>.
//-------------------------------------------------------------------------------
void SetSDIR(bool state)
{
        unsigned int i;
        if (state)
        {
                *pFIO_DIR |=  SDATA;  // Set SDATA as output.
        } else
        {
                *pFIO_DIR &=  ~SDATA; // Set SDATA as input/high.
        }
        delay(1000);
}
//-------------------------------------------------------------------------------
//      Routine:        SetSDATA
//      Inputs:         state
//      Outputs:        none
//      Purpose:        Set the I2C port SDATA pin to <state>.
//-------------------------------------------------------------------------------
void SetSDATA(bool state)
```

```
{
	unsigned int i;
	if (state)
	{
		*pFIO_FLAG_D |=  SDATA;		// Set SDATA high.
	} else
	{
		*pFIO_FLAG_D &= ~SDATA;		// Set SDATA low.
	}
	delay(1000);
}
//-----------------------------------------------------------------------------
//	Routine:	SetSCLK
//	Inputs:		state
//	Outputs:	none
//	Purpose:	Set the I2C port SCLK pin to <state>.
//-----------------------------------------------------------------------------
void SetSCLK(bool state)
{
	unsigned int i;
	if (state)
	{
		*pFIO_FLAG_D |= SCLK;		// Set SCLK high.
	} else
	{
		*pFIO_FLAG_D &= ~SCLK;		// Set SCLK low.
	}
	delay(1000);
}
//-----------------------------------------------------------------------------
//	Routine:	GetSDATA
//	Inputs:		none
//	Outputs:	bool
//	Purpose:	Get the I2C port SDATA pin state.
//-----------------------------------------------------------------------------
bool GetSDATA()
{
	return ((*pFIO_FLAG_D & SDATA) ? 1 : 0);
}
//-----------------------------------------------------------------------------
//	Routine:	DeSetSEL
//	Inputs:		none
//	Outputs:	none
//	Purpose:	Set PF12 like PPI7 as input
//-----------------------------------------------------------------------------
void DeSetSEL(void)
{
	*pFIO_DIR &=  ~SEL;			// Set SEL as input
}
//-----------------------------------------------------------------------------
//	Routine:	SetRESET
//	Inputs:		bool
```

```
//      Outputs:        none
//      Purpose:        Set the RESET  pin to <state>.
//----------------------------------------------------------------------------
void SetRESET(bool state)
{
        if (state)
        {
                *pFIO_FLAG_D |=  RESET_DV;    // Set SEL high.
        } else
        {
                *pFIO_FLAG_D &= ~RESET_DV;    // Set SEL low.
        }
        delay(80000);
}
//----------------------------------------------------------------------------
//      Routine:        delay
//      Inputs:         nsec
//      Outputs:        none
//      Purpose:        delay software
//----------------------------------------------------------------------------
void delay(int nsec)
{
        int i = (int)((nsec)/10);
        for(;i>0;i--) { asm("nop;"); }
}
/*********************************************************************/



/*********************************************************************/
#include "flash_bf532.h"
/************************************************************
 * ERROR_CODE SetupForFlash()
 * Perform necessary setup for the processor to talk to the
 * flash such as external memory interface registers, etc.
 *
 ************************************************************/
ERROR_CODE SetupForFlash(void)
{
        int i;

        for (i = 0; i < NUM_SECTORS; i++) {
        GetSectorStartEnd(&SectorInfo[i].ulStartOff, &SectorInfo[i].ulEndOff, i);
        }

        return NO_ERR;
}
////////////////////////////////////////////////////////////
// ERROR_CODE WriteData()
// Write a buffer to flash.
// Inputs:     unsigned long ulStart - offset in flash to start the writes at
//                     long lCount - number of elements to write, in this case bytes
```

```
//                    long lStride - number of locations to skip between writes
//                    int *pnData - pointer to data buffer
////////////////////////////////////////////////////////////
ERROR_CODE WriteData(unsigned long ulStart, long lCount, unsigned char *pnData, int verify)
{
        long i = 0;                          // loop counter
        unsigned long ulOffset = ulStart;    // current offset to write
        ERROR_CODE ErrorCode = NO_ERR;       // tells whether there was an error trying to write
        unsigned short nCompare = 0;         // value that we use to verify flash
        unsigned short data;

        /* write the buffer */
        for (i = 0; (i < lCount) && (ErrorCode == NO_ERR); i += 2, ulOffset += 2) {
                data = (((unsigned short) pnData[i + 1]) << 8) | ((unsigned short) pnData[i]);
                UnlockFlash(ulOffset);
                WriteFlash(ulOffset, data, FALSE);
                if ((ErrorCode = PollToggleBit(ulOffset)) != NO_ERR) {
                        continue;
                }
                if (verify) {
                        ReadFlash(ulOffset, &nCompare, FALSE);
                        if (nCompare != data) {
                                return VERIFY_WRITE;
                        }
                }
        }
        return ErrorCode;
}
////////////////////////////////////////////////////////////
// ERROR_CODE ReadData()
// Read a buffer from flash.
// Inputs:     unsigned long ulStart - offset in flash to start the reads at
//                    int nCount - number of elements to read, in this case bytes
//                    int nStride - number of locations to skip between reads
//                    int *pnData - pointer to data buffer to fill
////////////////////////////////////////////////////////////
ERROR_CODE ReadData(unsigned long ulStart, long lCount, unsigned char *pnData )
{
        long i = 0;                          // loop counter
        unsigned long ulOffset = ulStart;    // current offset to write
        unsigned short data;

        /* read the buffer  */
        for (i = 0; i < lCount; i += 2, ulOffset += 2) {
                /* Read flash */
                ReadFlash(ulOffset, &data, FALSE);
                pnData[i] = (unsigned char) (data & 0x00FF);
                pnData[i + 1] = (unsigned char)((data & 0xFF00) >> 8);
        }
        return NO_ERR;
}
////////////////////////////////////////////////////////////
```

```
// ERROR_CODE WriteFlash()
// Write a value to an offset in flash.
// Inputs:     unsigned long ulOffset - offset to write to
//                     int nValue - value to write
////////////////////////////////////////////////////////////
ERROR_CODE WriteFlash(unsigned long ulOffset, unsigned short nValue, int bCmd)
{
        /* get the offset from the start of flash and set the P register */
        asm ("p2.l = 0x0000;");
        asm ("p2.h = 0x2000;");
        asm ("r3 = %0;": : "d" (ulOffset));
        if (bCmd) asm ("r3 = r3 << 1;");
        asm ("r2 = p2;");
        asm ("r2 = r2 + r3;");
        asm ("p2 = r2;");
        /* write the value to the flash */
        asm ("SSYNC;");
        asm ("w[p2] = %0;": : "d" (nValue));
        asm ("SSYNC;");

        return NO_ERR;
}
////////////////////////////////////////////////////////////
// ERROR_CODE ReadFlash()
// Read a value from an offset in flash.
// Inputs:     unsigned long ulOffset - offset to read from
//                     int pnValue - pointer to store value read from flash
////////////////////////////////////////////////////////////
ERROR_CODE ReadFlash(unsigned long ulOffset, unsigned short *pnValue, int bCmd)
{
        /* temp holder */
        int nValue = 0x0;

        /* get the offset from the start of flash and put it in a P register */
        asm ("p2.l = 0x0000;");
        asm ("p2.h = 0x2000;");
        asm ("r3 = %0;": : "d" (ulOffset));
        if (bCmd) asm ("r3 = r3 << 1;");
        asm ("r2 = p2;");
        asm ("r2 = r2 + r3;");
        asm ("p2 = r2;");
        /* now place it into memory */
        asm ("SSYNC;");
        asm ("%0 = w[p2] (Z);": "=d" (nValue) : );
        asm ("SSYNC;");
        /* put the value at the location passed in */
        *pnValue = nValue;

        return NO_ERR;
}
////////////////////////////////////////////////////////////
// ERROR_CODE PollToggleBit()
```

```
// Polls the toggle bit in the flash to see when the operation
// is complete.
// Inputs:     unsigned long ulOffset - offset in flash
////////////////////////////////////////////////////////////
ERROR_CODE PollToggleBit(unsigned long ulOffset)
{
        ERROR_CODE ErrorCode = NO_ERR;      // flag to indicate error

        /* read from the flash twice and check the toggle bit */
        asm ("POLL_TOGGLE_BIT:");
        asm ("p2.l = 0x0000;");
        asm ("p2.h = 0x2000;");
        asm ("r2 = r0;");                   // ulOffset is passed into PollTogglebit in R0
        asm ("r1 = p2;");
        asm ("r1 = r1 + r2;");
        asm ("p2 = r1;");
        asm ("r1 = w[p2] (Z);");
        asm ("SSYNC;");
        asm ("r2 = w[p2] (Z);");
        asm ("SSYNC;");
        // set bits in r0 where data toggled
        asm ("r1 = r1 ^ r2;");
        // see if we are still toggling, if not we are done
        asm ("cc = bittst(r1, 6);");
        asm ("if !cc jump DONE_TOGGLE_BIT;");
        // see if there was an error
        asm ("cc = bittst(r2, 5);");
        asm ("if !cc jump POLL_TOGGLE_BIT;");
        // if we get here we detected the error bit set
        asm ("r1 = w[p2] (Z);");
        asm ("SSYNC;");
        asm ("r2 = w[p2] (Z);");
        asm ("SSYNC;");
        // set bits in r0 where data toggled
        asm("r1 = r1 ^ r2;");
        // see if we are still toggling, if not we are done
        asm ("cc = bittst(r1, 6);");
        asm ("if !cc jump DONE_TOGGLE_BIT;");
        // else we have failed, restore page, set error flag, and reset
        ErrorCode = POLL_TIMEOUT;
        reset = 0x1;
        ResetFlash();
        // we are done toggling
        asm ("DONE_TOGGLE_BIT:");

        // we can return
        return ErrorCode;
}
////////////////////////////////////////////////////////////
// ERROR_CODE ResetFlash()
// Sends a "reset" command to the flash.
////////////////////////////////////////////////////////////
```

```
ERROR_CODE ResetFlash(void)
{
        /* send the reset command to the flash */
        WriteFlash(0x0AAA, 0xf0, TRUE);

        /* reset should be complete */
        return NO_ERR;
}
////////////////////////////////////////////////////////////
// ERROR_CODE EraseFlash()
// Sends an "erase all" command to the flash.
////////////////////////////////////////////////////////////
ERROR_CODE EraseFlash(void)
{
        ERROR_CODE ErrorCode = NO_ERR;      // tells us if there was an error erasing flash

        // erase contents of Flash
        WriteFlash(0x0555, 0xaa, TRUE);
        WriteFlash(0x02AA, 0x55, TRUE);
        WriteFlash(0x0555, 0x80, TRUE);
        WriteFlash(0x0555, 0xaa, TRUE);
        WriteFlash(0x02AA, 0x55, TRUE);
        WriteFlash(0x0555, 0x10, TRUE);
        // poll until the command has completed
        ErrorCode = PollToggleBit(0x0000);

        // erase should be complete
        return ErrorCode;
}
////////////////////////////////////////////////////////////
// ERROR_CODE EraseBlock()
// Sends an "erase block" command to the flash.
////////////////////////////////////////////////////////////
ERROR_CODE EraseBlock(int nBlock)
{
        unsigned long ulSectorOff = 0x0;
        ERROR_CODE ErrorCode = NO_ERR;      // tells us if there was an error erasing flash

        /* if the block is invalid just return */
        if ((nBlock < 0) || (nBlock > NUM_SECTORS))
        return INVALID_BLOCK;
        /* figure out the offset of the block in flash */
        ulSectorOff = SectorInfo[nBlock].ulStartOff;
        // send the erase block command to the flash
        WriteFlash(0x0555, 0xaa, TRUE);
        WriteFlash(0x02AA, 0x55, TRUE);
        WriteFlash(0x0555, 0x80, TRUE);
        WriteFlash(0x0555, 0xaa, TRUE);
        WriteFlash(0x02AA, 0x55, TRUE);
        // the last write has to be at an address in the block
        WriteFlash(ulSectorOff, 0x30, FALSE);
        // poll until the command has completed
```

```
        ErrorCode = PollToggleBit(ulSectorOff);
        // block erase should be complete

        return ErrorCode;
}
////////////////////////////////////////////////////////////
// ERROR_CODE UnlockFlash()
// Sends an "unlock" command to the flash to allow the flash
// to be programmed.
////////////////////////////////////////////////////////////
ERROR_CODE UnlockFlash(unsigned long ulOffset)
{
        unsigned long ulOffsetAddr = ulOffset;
        ulOffsetAddr &= 0xFFFF0000;

        // send the unlock command to the flash
        // ORed with lOffsetAddr so we know what block we are in
        WriteFlash(0x0555, 0xaa, TRUE);
        WriteFlash(0x02AA, 0x55, TRUE);
        WriteFlash(0x0555, 0xa0, TRUE);

        return NO_ERR;
}
////////////////////////////////////////////////////////////
// ERROR_CODE GetSectorNumber()
// Gets a sector number based on the offset.
////////////////////////////////////////////////////////////
ERROR_CODE GetSectorNumber(unsigned long ulOffset, int *pnSector)
{
        int nSector = 0;

        nSector = (ulOffset & 0xFFFF0000) >> 16;
        if (nSector == 0) {
                switch ((ulOffset & 0x0000E000) >> 13) {
                        case 0:
                        case 1:
                                nSector = 0;
                                break;
                        case 2:
                                nSector = 1;
                                break;
                        case 3:
                                nSector = 2;
                                break;
                        case 4:
                        case 5:
                        case 6:
                        case 7:
                                nSector = 3;
                                break;
                        default:
                                nSector = 0;
```

```
		}
	} else nSector += 3;
	/* if it is a valid sector, set it */
	if ((nSector >= 0) && (nSector < NUM_SECTORS))
		*pnSector = nSector;
	else
		return INVALID_SECTOR;

	return NO_ERR;
}
////////////////////////////////////////////////////////////
// ERROR_CODE GetSectorStartEnd()
// Gets a sector number based on the offset.
// Inputs:	unsigned long *lStartOff - pointer to the start offset
//			unsigned long *lEndOff - pointer to the end offset
//			int nSector - sector number
////////////////////////////////////////////////////////////
ERROR_CODE GetSectorStartEnd(unsigned long *lStartOff, unsigned long *lEndOff, int nSector)
{
	if (nSector >= 4) {
		*lStartOff = ((nSector - 3) * SECTORSIZE1);
		*lEndOff = ((*lStartOff) + SECTORSIZE1) - 1;
	} else if (nSector == 3) {
		*lStartOff = 0x008000;
		*lEndOff = (unsigned long) ((*lStartOff) + SECTORSIZE2) - 1;
	} else if (nSector == 2) {
		*lStartOff = 0x006000;
		*lEndOff = (unsigned long) ((*lStartOff) + SECTORSIZE4) - 1;
	} else if (nSector == 1) {
		*lStartOff = 0x004000;
		*lEndOff = (unsigned long) ((*lStartOff) + SECTORSIZE4) - 1;
	} else if( nSector == 0 ) {
		*lStartOff = 0x0;
		*lEndOff = (unsigned long) ((*lStartOff) + SECTORSIZE3) - 1;
	} else
	return INVALID_SECTOR;

	return NO_ERR;
}

ERROR_CODE GetCodes(int *AFP_DevCode, int *AFP_ManCode)
{
	unsigned short devcode, mancode;

	// send the auto select command to the flash
	WriteFlash( 0x0555, 0xaa, true );
	WriteFlash( 0x02AA, 0x55, true );
	WriteFlash( 0x0555, 0x90, true );

	// now we can read the codes
	ReadFlash( 0x0200, &devcode, true );
	*AFP_DevCode = devcode & 0x00FF;
```

```
        ReadFlash( 0x0201, &mancode, true );
        *AFP_ManCode = mancode & 0xFFFF;

        // we need to issue another command to get the part out
        // of auto select mode so issue a reset which just puts
        // the device back in read mode
        ResetFlash();

        // ok
        return NO_ERR;
}

ERROR_CODE CheckProtection(int *protstatus, int nblock)
{
        unsigned short prot;

        WriteFlash(0x00555, 0x00AA, true);  /* 1st Cycle */
        WriteFlash(0x002AA, 0x0055, true);  /* 2nd Cycle */
        WriteFlash(0x00555, 0x0090, true);  /* 3rd Cycle */

        /* Step 3: Read Protection Status */
        ReadFlash(SectorInfo[nblock].ulStartOff + 0x02, &prot, true);
        *protstatus = prot;
        ResetFlash();

        // ok
        return NO_ERR;
}
/*****************************************************************/
/*************************************************************************************************/
```

Bibliografia

[1] **EE-257**: A Boot Compression/Decompression Algorithm for Blackfin Processors
http://www.analog.com/UploadedFiles/Application_Notes/48548288698368EE257v01.pdf

[2] **EE-240**: ADSP-BF533 Blackfin Booting Process
http://www.analog.com/UploadedFiles/Application_Notes/304621268EE240v03.pdf

[3] **EE-239:** Running Programs from Flash on ADSP-BF533 Blackfin Processors
http://www.analog.com/UploadedFiles/Application_Notes/44848287486669EE239v01.pdf

[4] **EE-237**: Guide to Blackfin Processor LDF Files
http://www.analog.com/UploadedFiles/Application_Notes/3446741932285465795EE237v01.pdf

[5] **EE-229:** Estimating Power for ADSP-BF533 Blackfin Processors
http://www.analog.com/UploadedFiles/Application_Notes/125876083EE229v01.pdf

[6] **EE-210:** SDRAM Selection and Configuration Guidelines for ADI Processors
http://www.analog.com/UploadedFiles/Application_Notes/53047287221168EE210v02.pdf

[7] **EE-68:** Analog Devices JTAG Emulation Technical Reference
http://www.analog.com/UploadedFiles/Application_Notes/4280679728866EE068v09.pdf

[8] ADSP-BF531/ADSP-BF532/ADSP-BF533: Blackfin® Embedded Processor
http://www.analog.com/UploadedFiles/Data_Sheets/744511807ADSP_BF535_a.pdf

[9] ADSP-BF533 Blackfin Processor Hardware Reference
http://www.analog.com/processors/epManualsDisplay/0,2795,,00.html?SectionWeblawId=207&ContentID=69184&Language=English

[10] VisualDSP++4.0 Assembler and Preprocessor Manual
http://www.analog.com/processors/epManualsDisplay/0,2795,,00.html?SectionWeblawId=207&ContentID=60885&Language=English

[11] VisualDSP++ 4.0 Loader Manual
http://www.analog.com/processors/epManualsDisplay/0,2795,,00.html?SectionWeblawId=207&ContentID=60854&Language=English

[12] VisualDSP++4.0 Linker and Utilities Manual
http://www.analog.com/processors/epManualsDisplay/0,2795,,00.html?SectionWeblawId=207&ContentID=60894&Language=English

[13] ADSP-BF533 EZ-KIT Lite Evaluation System Manual
http://www.analog.com/processors/epManualsDisplay/0,2795,,00.html?SectionWeblawId=207&ContentID=32890&Language=English

[14] ADSP-BF533 STAMP v1.2
http://blackfin.uclinux.org/frs/download.php/420/stamp_schematics_v1.2.pdf

[15] TVP5150 Low-Power Video Decoder
http://focus.ti.com/lit/ds/symlink/tvp5150.pdf

[16] M29W160EB FLASH Memory Data Sheet
http://www.st.com/stonline/products/literature/ds/9195.pdf

[17] TC55VBM416AFTN55 SRAM Memory Data Sheet
http://www.toshiba.com/taec/components/Datasheet/tc55vbm416a.pdf

[18] MT48LC4M16A2 SDRAM Memory Data Sheet
http://fp.cse.wustl.edu/cse362/Datasheets/sdram.pdf

[19] ADM3202 Low Power RS/232 Line Drivers/Receivers Data Sheet
http://www.analog.com/UploadedFiles/Data_Sheets/79657292ADM3202_22_1385_b.pdf

[20] ADM6711 Microprocessor Supervisory Data Sheet
http://www.analog.com/UploadedFiles/Data_Sheets/526737ADM6711_13_0.pdf

[21] SN74LVC1G332 - Single 3-Input Positive-OR Gate Data Sheet
http://focus.ti.com/lit/ds/symlink/sn74lvc1g332.pdf

[22] SN74AHC1G08 - Single 2-Input Positive-AND Gate Data Sheet
http://focus.ti.com/lit/ds/symlink/sn74ahc1g08-q1.pdf

[23] TPS76918 - Ultra Low-Power Low-Dropout (LDO) Line Regulators Data Sheet
http://focus.ti.com/lit/ds/symlink/tps76918-q1.pdf

[24] FDC6324L Integrated Load Switch Data Sheet
http://www.fairchildsemi.com/ds/FD/FDC6324L.pdf

[25] ITU-R BT.656-4 Reccomendation
http://www-inst.eecs.berkeley.edu/~cs150/Documents/ITU656.doc

[26] ITU-R BT.601-4 Reccomendation
http://www-inst.eecs.berkeley.edu/~cs150/Documents/ITU601.PDF

[27] Brian W.Kernighan, Dennis M.Ritchie. *Linguaggio C*. Gruppo Editoriale Jackson, Milano, 1989