Università della Calabria
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Dipartimento di Informatica Modellistica Elettronica e Sistemistica

Tesi di Laurea

Macchina Virtuale CalcuList

Candidato:
Giuseppe Sottile

Relatore
Prof. Domenico Saccà

Correlatore
Ing. Fabio Fassetti

Anno Accademico
2016/2017

Introduzione

La gioia nell'osservare e nel comprendere
è il dono più bello della natura

(Albert Einstein)

CalcuList è una macchina virtuale a stack che semplifica la complessità di una macchina reale. Non si tratta di un modello fisico reale, ma bensì di un progetto teorico, un po come l'idea di Turing della sua macchina universale, ma con la differenza sostanziale che ne esiste una versione interprete eseguibile su un calcolatore reale. Scopo della tesi è la descrizione architetturale e del set delle istruzioni del modello di macchina virtuale calculist per la gestione delle liste, da un punto di vista *a basso livello* cioè più vicino alla natura intrinseca della macchina, che, rispetto al linguaggio dell'uomo. La trattazione è suddivisa in tre capitoli. Il primo capitolo tratta argomentazioni di carattere generale in cui vengono introdotti i principali modelli di architettura dei calcolatori assieme ad alcune nozioni utili alla comprensione del loro funzionamento e dei capitoli successivi; in particolare nel secondo capitolo viene presentato il modello di architettura della macchina virtuale incluse molte delle caratteristiche e scelte progettuali assieme ad una descrizione sommaria del ciclo di esecuzione dei programmi e del meccanismo di chiamata a funzione tipico delle *architetture a stack*. Nel terzo capitolo infine viene presentato il linguaggio RTL per la descrizione del microcodice e del set delle istruzioni. A supporto di tutta la tesi vi è un'appendice in cui è riportata l'intera architettura ed un glossario delle principali componenti.

◇◇◇

Ringraziamenti

Teoria, pratica, meditazione, consapevolezza, lettura e scrittura unite ad una insaziabile voglia di imparare ed un amore per la scienza, hanno fatto sì che raggiungessi questa meta e concludessi con entusiasmo questa ulteriore avventura nel vaggio della mia vita. Ogni volta è sempre come la prima volta, ogni esame, ogni traguardo, ogni piccolo tassello... sono diventato lo studente di me stesso a seguito del tempo sinora trascorso e dei miei legami intercorsi con le altre persone e con la scienza...

◇ ◇ ◇

Ringrazio il relatore della tesi, il Prof. Domenico Saccà, per la sua disponibilità e per aver fatto si che raggiungessi questa meta, sviluppando un argomento di interesse personale, oltre che scientifico. Ringrazio, inoltre, il correlatore, l'Ing. Fabio Fassetti, per i suoi utili consigli ed i preziosi suggerimenti durante la stesura della tesi. Il $''mio''$ dipartimento, il DIMES, per aver contribuito alla mia crescita intellettuale ed accademica oltre che scientifica! A loro un particolare ringraziamento per la grande professionalità! A mia madre Rossella, a mio padre Alessandro e a mio fratello Manuel per il loro supporto e la pazienza in tutto e per tutto; come non dimenticare le mie bellissime notti insonni in questi anni di studio! Infine ringrazio tutti coloro che ho incontrato finora, e indirettamente quelli che conoscerò, dato che contribuiranno a modificare il mio attuale IO, migliorando ed elevando il mio grado di consapevolezza. Naturalmente ritornerò ad essere lo studente di me stesso perchè ogni fine è sempre un nuovo inizio; così è stato sino a questo momento e così sarà d'ora in avanti, perchè imparare è il mestiere piu bello della vita.

Indice

alla mia famiglia

CAPITOLO 1

Elementi architetturali

La caratteristica fondamentale di un elaboratore elettronico, che lo contraddistingue dalle tradizionali calcolatrici da tavolo, sta nella sua capacità di riuscire ad eseguire sequenze di istruzioni memorizzate in maniera del tutto autonoma. Allo stato attuale della tecnologia i calcolatori sono costruiti attorno ad una unità di elaborazione e controllo (elaboratore). Le istruzioni, come ogni altra informazione, sono codificate in gruppi di bit. L'elaboratore preleva le istruzioni dalla memoria, interpreta i codici di istruzione ed effettua le azioni che questi prevedono. Un insieme organizzato di istruzioni costituisce un programma. In questo capitolo, l'interesse è rivolto essenzialmente ai concetti architetturali di carattere generale, in modo da poter affrontare con estrema semplicità e disinvoltura, nel capitolo successivo, l'architettura ed il funzionamento dell'unità calculist.

1.1 Struttura di un elaboratore

Possiamo descrivere ogni sistema di elaborazione, come costituito da tre componenti o sottosezioni fondamentali: L'unità di elaborazione e controllo, La Memoria ed il Sottosistema I/O di ingresso/uscita. Ogni sezione riveste un ruolo centrale nell'architettura dei sistemi, in quanto svolge un compito essenziale che analizzeremo di seguito:

Unità di Elaborazione e Controllo: rappresenta il nucleo centrale del sistema. È la parte responsabile dell'interpretazione ed esecuzione delle istruzioni, quindi del coordinamento e del controllo dei segnali all'interno del sistema.

Memoria: rappresenta il contenitore delle istruzioni e dei dati. Ogni dato ed ogni istruzione vengono rappresentati sottoforma di bit. In questo modo è impossibile distinguere un dato da un'istruzione all'interno della

memoria[1]; tutto dipende dal modo in cui vengono interpretati i bit. Il fatto che dati ed istruzioni siano di per se indistinguibili permette di costruire programmi che generano dati che, in un secondo tempo, possono essere interpretati come istruzioni.[2]

SOTTOSISTEMA I/O: Tutto ciò cui ruota attorno al nucleo fondamentale del sistema. È costituito dall'insieme dei dispositivi che; da una parte rendono possibile una comunicazione con il mondo esterno, dall'altra estendono le funzionalità del sistema stesso introducendo le interfacce che fanno da ponte tra il sistema e le periferiche (*dischi, tastiere, stampanti ecc)*

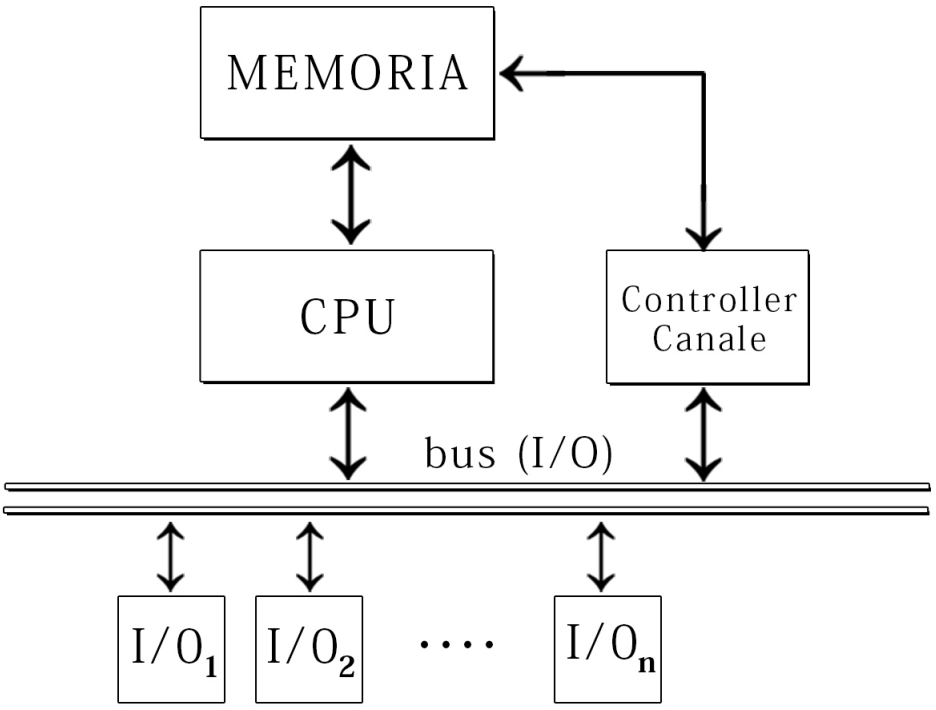


Figura 1.1: Organizzazione di un elaboratore elettronico

[1]Esistono delle architetture in cui vi è una suddivisione della memoria dati e della memoria programma che vedremo in seguito.

[2]Tipico esempio è un compilatore

1.2 Il modello di memoria

Analizziamo ora la parte relativa alla memorizzazione delle informazioni all'interno dei calcolatori, ossia la memoria. Faremo riferimento a due tipi di organizzazione della memoria il classico *modello di Von Neumann* ed il meno noto *modello di Harvard* che come vedremo sarà il modello più vicino a quello della macchina Calculits.

Modello di Von Neumann

John Von Neumann[3] grande matematico e fisico, padre dell'informatica e della teoria dei giochi; tra le sue innumerevoli scoperte ed invenzioni, ebbe un'idea che rivoluzionò l'informatica degli anni '50 e portò a nuove idee e punti di vista, legati principalmente agli aspetti architetturali e di progettazione dei calcolatori. L'idea è quella di **calcolatore a programma memorizzato** in cui sia il programma, sia i dati risiedono in memoria centrale.[4] Nel suo modello, detto appunto: **Modello di Von Neumann**, come già espresso ad inizio paragrafo (*dati ed istruzioni risiedono entrambi in memoria*) e sono indistinguibili; solo l'interpretazione da parte della CPU stabilisce se una data configurazione di bit è da riguardarsi come un dato o come un'istruzione.

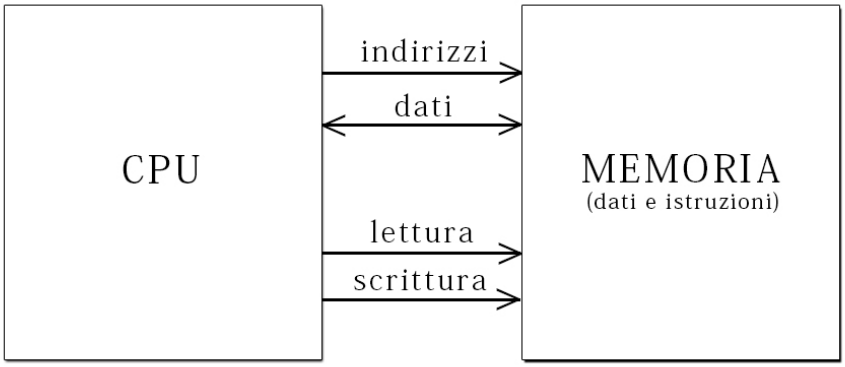


Figura 1.2: Architettura di Von Neumann

[3]John Von Neumann 1945-1955, È stato un matematico e fisico del '900. Pioniere dell'informatica e della teoria dei giochi. A lui è dovuta l'architettura moderna dei calcolatori.

[4]Nei primi calcolatori i programmi venivano caricati esternamente mediante perforatrici, nastri e schede perforate. L'architettura CalcuList prevede un modello di memoria separato detto di Harvard, ma l'idea di programma in memoria fa lo stesso riferimento alla tipologia a programma memorizzato.

Modello di Harvard

Se da un punto di vista architetturale il modello di Von Neumann è la scelta primaria per l'implementazione della gran parte degli elaboratori elettronici, il modello di Harvard[5] riveste un ruolo prettamente specialistico per applicazioni ad hoc[6]. La caratteristica che lo rende peculiare è l'indipendenza dei dati dalle istruzioni. Mentre nel modello classico di Von Neumann la CPU non è in grado di distinguere le istruzioni dai dati, nel modello di Harvard i dati vengono separati dalle istruzioni perchè essi risiedono in memorie differenti, si parla di **Memoria dati** da una parte e **Memoria istruzione** dall'altra. Questa architettura conferisce una proprietà speciale al sistema, infatti è possibile che la CPU prelevi istruzioni e dati parallelamente, cosa non permessa nel modello di Von Neumann. Un'architettura Harvard quindi può eseguire più compiti in parallelo dato che può parallelizzare le operazioni di lettura e scrittura della memoria. Tuttavia, all'aumentare della velocità, si contrappone la presenza di circuiti più complessi all'interno del processore.

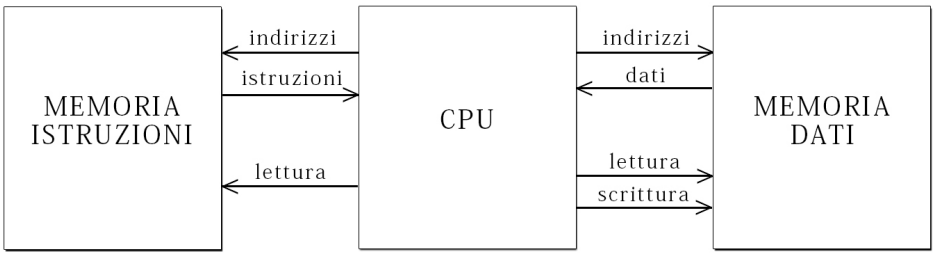


Figura 1.3: Architettura di Harvard

1.3 CPU

Con il termine Cpu, seguendo una definizione classica, si intende *l'unità centrale di elaborazione*, ossia una rete logica che può eseguire un insieme finito di operazioni, sulla base di istruzioni esterne. Come ogni rete, un sistema comunica con l'esterno attraverso una serie di pin o morsetti di ingresso e di uscita, le operazioni vengono eseguite su un insieme di dati provenienti dall'esterno e forniscono risultati verso l'esterno, il sistema acquisisce, come visto, attraverso morsetti di ingreso, anche l'insieme delle istruzioni.

[5]L'architettura Harvard è stata la prima ad essere implementata sul MARK I. Si tratta di un calcolatore elettromeccanico che memorizzava le istruzioni su un nastro perforato mentre i dati venivano memorizzati in un contatore a parte.

[6]Vedi: trattamento audio/video, DSP, elaborazione dei segnali.

Parte operativa e parte di controllo

Ogni unità di elaborazione è suddivisa in due parti fondamentali connesse in stretta simbiosi chiamate **parte operativa** e **parte di controllo**. La parte operativa comprende tutte le componenti hardware del sistema, parliamo di registri, memoria, alu, comparatori, addizionatori ecc... sostenzialmente ci riferiamo all'elettronica del nostro sistema. La parte di controllo invece è il *"cervello elettronico"* del sistema, e si occupa dello smistamento dei segnali e del controllo di tutte le componenti hardware sulla base di un programma che risiede nella memoria centrale;[7] e, che via via, istruzione dopo istruzione viene eseguito in un ciclo fondamentale il cui meccanismo interno verrà analizzato di seguito.

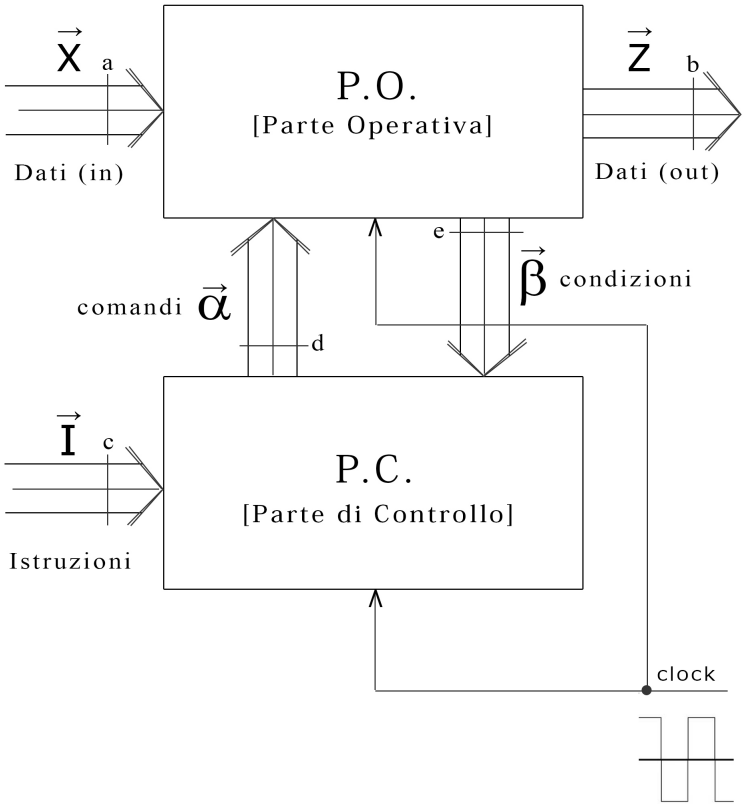


Figura 1.4: Parte operativa e parte di controllo

[7] Osservate come il programma risiede nella parte operativa.

Ciclo istruzione

Dalla figura notiamo che sia la parte operativa PO, sia quella di controllo PC, sono sincronizzate dallo stesso segnale di clock. Esse infatti, a livello macroscopico possono essere viste come due grosse reti sequenziali sincrone la più importante delle quali, l'unità di controllo, può essere descritta dal seguente automa o digramma di stato, detto anche (*ciclo di fetch - execute*).

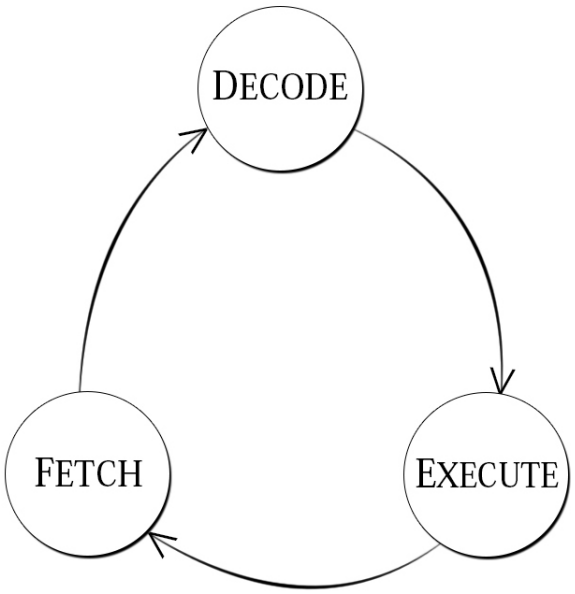


Figura 1.5: Diagramma di stato del funzionamento della parte di controllo

Possiamo analizzare ora in dettaglio ogni fase del diagramma:

FETCH: Ogni programma risiede in memoria nella PARTE DI CONTROLLO a partire da un certo indirizzo. Un registro, PC (*Program Counter*) punta all'istruzione da eseguire. la fase di fetch inizia con il prelievo della prossima istruzione da eseguire dalla memoria in un altro registro detto IR. Ogni sequenza di bit all'interno dell'IR verrà interpretata come istruzione ed eseguita.

DECODE: Una volta trasferita nel registro IR, l'istruzione deve essere decodificata. In questa fase viene decodificato il codice operativo[8] dell'istruzione ed inoltre viene incrementato PC in modo da farlo puntare all'istruzione successiva (per la prossima fase di fetch).

EXECUTE: In questa l'unità di controllo genera i segnali che fanno svolgere le azioni richieste in base al contenuto e di eventuali condizioni di stato della parte operativa.

[8]Il codice operativo identifica ogni istruzione all'interno del sistema ed inoltre definisce i segnali di comando da inviare alla parte operativa per attuare l'istruzione.

Architettura Cablata

Analizziamo brevemente le tipologie di architettura possibili per la parte di controllo di un elaboratore. Una possibile implementazione è nota con il termine di architettura cablata. Sostanzialmente in questo tipo di implementazione la parte di controllo è una vera è propria **rete sequenziale** realizzata mediante un'approccio tradizionale che vede la realizzazione di un automa a stati finiti per il modello concettuale astratto. La rete sequenziale come già mostrato in figura è una rete sequenziale sincrona temporizzata da un clock rappresentato da un'onda quadra. Il cuore del sistema è una rete combinatoria che ha come ingresso complessivo l'insieme formato dal campo operazione del registro IR, dalle condizioni di stato provendienti dalla P.O., mentre le linee di uscita rappresentano i comandi da impartire sia alla memoria che al resto del sistema.

Architettura Microprogrammata

Se da un punto di vista concettuale, l'approccio cablato risulta essere il metodo più naturale per la progettazione della logica di controllo, per l'impiego nei grossi sistemi è inusuale adottare una tecnica cablata e si ricorre spesso ad un tipo di implementazione **microprogrammato**. La microprogrammazione si affermò negli anni sessanta come un approccio orientato alla progettazione di unità di controlto più complesse, in grado di superare tutte le difficolta insite, nella logica cablata (complessità intrinseca della logica, rigidita, alti costi per apportare modifiche alla logica stessa, complessità nella realizzazione dell'automa di stato ecc). Con la microprogrammazione la P.C. viene vista come una sorta di *″micro-calcolatore nel calcolatore″*, in grado di eseguire passi di (micro)programrna. Nella forma piu semplice la parte di controllo microprogrammata e costituita da una memoria (detta memoria di controllo) contente le microistruzioni e da una unità capace di indirizzare, come si trattasse dell'esecuzione di un programma, la memoria di controllo. L'effetto risultante è che alla parte operativa vengono presentate sequenze di microistruzioni che rendono possibile l'attuazione delle istruzioni(macro), vere e proprie del linguaggio *assembly* della macchina.

1.4 Prestazioni

Concludiamo questo capitolo con la valutazione delle prestazioni della Cpu. La valutazione delle prestazioni di un sistema di elaborazione è cosa alquanto complessa da determinare e coinvolge quasi tutti gli aspetti del funzionamento della macchina da una parte, ma anche dell'architettura stessa. Di seguito analizzeremo una formula che rende conto dei molti fattori che la determinano.

La prima relazione è una stima del tempo di esecuzione di un programma: Se indichiamo con N il numero di cicli di clock che esso comporta, con f la frequenza di clock e con T_{CPU} il tempo di esecuzione dell'intero programma, vale la seguente relazione:

$$T_{CPU} = \frac{N}{f}$$

Ossia il tempo di esecuzione di un programma è pari al numero totale di cicli di clock moltiplicato (quanti di essi ne avvengono in un determinato intervallo di tempo).

Si indichi ora con N_{ist} il numero di istruzioni di macchina corrispondente al programma e con $C_{PI} = \frac{N}{N_{ist}}$, il numero medio di cicli di clock a istruzione, si ha (operando una sostituzione) nella prima relazione:

$$T_{CPU} = \frac{N}{f} = \frac{C_{PI} \cdot N_{ist}}{f} = C_{PI} \cdot N_{ist} \cdot \underset{\substack{\uparrow \\ PERIODO}}{T}$$

Formula di valutazione delle prestazioni

Riassumendo quanto espresso, possiamo ricavare una formula in grado di fornire una valutazione pratica delle prestzioni di un sistema di calcolo:

$$C_{PI} = \frac{\sum_i \chi_i \cdot C_{pi}}{\sum_i \chi_i}$$

CAPITOLO 2

Architettura Calculist

In questo capitolo l'attenzione è rivolta principalmente al modello di macchina Calculist abbr.CLVM, sia da un punto di vista funzionale che architetturale. Come ribadito, CalcuList è una macchina virtuale a stack che semplifica la complessità di una macchina reale. Non esiste una versione hardware della stessa ma solo un progetto teorico su carta, un po come l'idea di Turing[1] della sua macchina universale, ma con la differenza sostanziale che ne esiste una versione interprete eseguibile su un calcolatore reale. Inizieremo il capitolo parlando di alcune tipologie di macchine per poi focalizzare l'attenzione sulla scelta per la macchina virtuale clvm.

2.1 tipologie di macchine

Prima di descrivere in dettaglio la struttura della macchina calculist analizziamo brevemente alcune delle tipologie di classi di architettura con cui vengono realizzati gli elaboratori. In generale, un modo per classificare le architetture dei calcolatori consiste nel riferirsi al cosiddetto *modello di esecuzione*, ovvero al modo in cui la CPU accede e manipola gli oggetti dell'elaborazione. Al modello di esecuzione corrisponde la struttura e il formato delle istruzioni, con particolare riferimento al numero di operandi esplicitamente o implicitamente rappresentati nell'istruzione. La classificazione che si traccia qui di seguito deve essere intesa come uno schema generale. Nella pratica reale si trovano macchine che adottano soluzioni architetturali miste o ibride. Fondamentalmente possiamo classificare i modelli di progettazione in tre tipologie standard di architettura per un modello di agente di calcolo inteso come macchina per

[1] Alan Mathison Turing (Londra, 23 giugno 1912 Wilmslow, 7 giugno 1954) è stato un matematico, logico e crittografo britannico, considerato uno dei padri dell'informatica e dell'intelligenza artificiale. Nel suo articolo: ″On computable Numbers, with an application to the Entscheidungsproblem″, descrisse la sua idea di ″macchina di Turing″ come modello primordiale di computer come lo conosciamo oggi.

eseguire istuzioni. Il modello ad **accumulatore**, il modello a **registri** ed il modello a **stack**.

ACCUMULATORE: Nell'architettura ad accumulatore esiste un solo registro generale: *accumulatore* per fare tutte le operazioni. Le istruzioni nominano esplicitamente un solo operando in memoria, mentre l'altro operando è implicitamente preso dal registro accumulatore, dove pure viene depositato il risultato. II vantaggio è la compattezza del codice, contro una non trascurabile rigidità, dovuta al fatto che tutto deve passare per per l'accumulatore[2]. Le Operazioni vengono eseguite tra il registro ed il contenuto di una cella di memoria. L'accumulatore è operando implicito di tutte le istruzioni.[3]

REGISTRO: Questo modello prevede un consistente blocco di registri di uso generale, sostanzialmente equipollenti tra di loro e semplici operazioni di caricamento e memorizzazione. Le operazioni di manipolazione dei dati, usano sempre e solo registri come contenitori degli operandi e come destinatari del risultato dell'operazione. Ad esempio, l'istruzione $ADD\ RA, RB, RC$, calcola la somma del contenuto dei registri RB, RC e deposita il risultato in RA. L'esecuzione di una somma richiede che vengano prima caricati RB ed RC con gli addendi.

STACK: Negli anni sessanta e settanta sono state costruite macchine che modellavano il modello astratto di calcolo basato sullo stack. Operazioni come per esempio un'addizione (ADD), non richiedono che venga esplicitato alcun operando, in quanto i due operandi vengono prelevati implicitamente dalle due posizioni di testa dello stack, mentre il risultato viene depositato sempre sullo stack stesso. Si tratta di architetture che, in termini astratti, modellano al meglio certi formalismi computazionali presenti in alcuni linguaggi di programmazione di alto livello.[4].

---

[2] Nell' architettura 8086 si parla di 8 registri di uso generale, anche se poi tutti quanti hanno una loro specializzazione. Tra di essi c'è un registro, AX, che viene denominato accumulatore, sebbene non goda dell'esclusività di intervenire nelle operazioni aritmetiche. Il termine di accumulatore deriva piuttosto dal fatto che certe istruzioni aritmetiche richiedono per forza AX, mentre altri registri sono specializzati in altre funzioni.

[3] Le seguenti istruzioni fanno riferimento al registro accumulatore come mostrato tra parentesi:

$$ADDx \quad (AC + M[x] \rightarrow AC)$$

$$JZx \quad (\textit{esegue il salto solo se } AC = 0)$$

[4] Vedi ALGOL (anni sessanta)

2.2 Le componenti principali

La scelta architetturale di CALCULIST rispecchia un modello a **stack** con un meccanismo di gestione della memoria di tipo LIFO ossia *Last in First out*, ma di questo ne parleremo in seguito quando presenteremo il modello di memoria.

2.3 Il modello di memoria

Rispetto al classico modello di **Von Neumann** che prevede che la memoria venga adibita sia a programmi che dati, la macchina calculist prevede una netta suddivisione della memoria centrale secondo il cosiddetto **Modello di Harvard**. Sostanzialmente vi sono tre tipi di memorie differenti, ciascuna delle quali viene gestita separatamente come vedremo in seguito da registri ad hoc. Vediamo più in dettaglio brevemente la suddivisione della memoria centrale analizzandone alcuni dettagli implementativi:

1. MEMORIA DATI: L'area di memoria dati, che d'ora in seguito verrà indicata con il nome: DATAMEM, è destinata alla memorizzazione delle variabili e dei valori (non istruzioni) che durante un ciclo di attività della macchina vengono elaborate dalla cpu.

2. MEMORIA CODICE: L'area di memoria destinata al codice CODE, conterrà tutte le istruzioni in linguaggio ASSEMBLY-CLVM. Strutturalmente possiamo vedere la memoria come un array di double in cui ogni cella contiene una singola istruzione assembly.

3. MEMORIA OUTPUT: L'area di memoria OUT è impiegata per la stampa su video dei dati.

La Cpu

: CPU Un'unità di elaborazione è impiegata per la computazione di istruzioni impartite alla macchina.

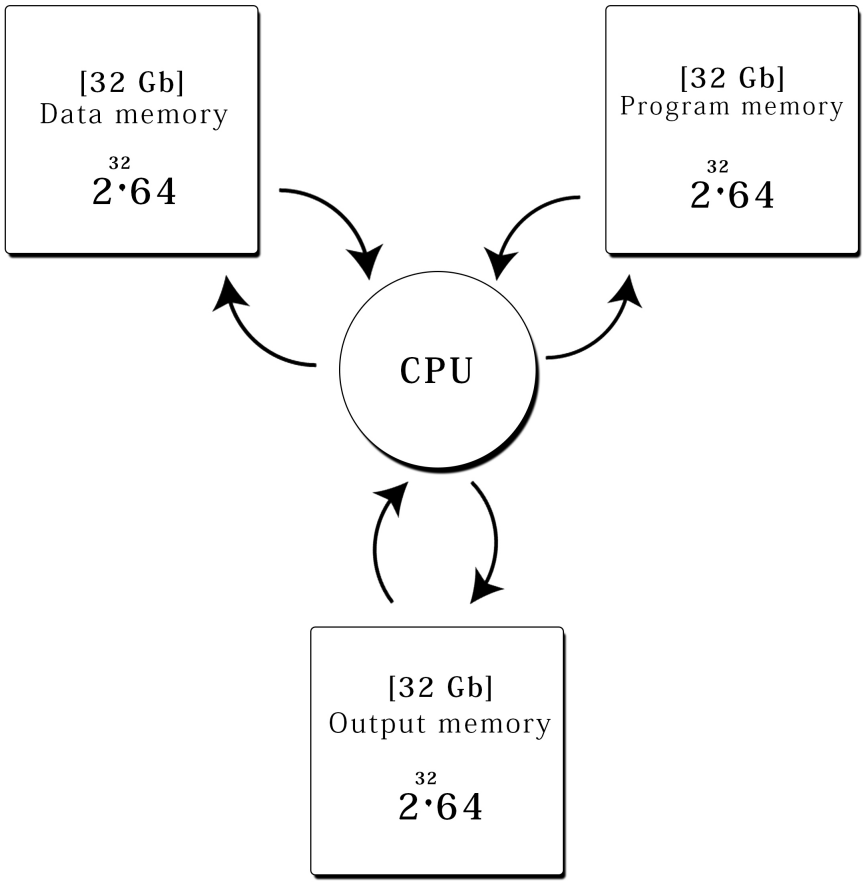


Figura 2.1: Il modello di memoria calculist

2.4 Il modello di programma calculist

Come detto, il nostro modello di macchina prevede una memoria dedicata per le sole istruzioni: la (Program Memory). Rispetto ai modelli reali in cui la memoria viene *rilocata* dinamicamente, il modello in questione è relativamente semplificato in quanto ogni programma inizia sempre dall'indirizzo base della memoria; si parla quindi di programmi assoluti in riferimento all'indirizzamento stesso in memoria. Un programma eseguibile CLVM è costituito da una serie di istruzioni raggruppate in blocchi o sotto-moduli, chiamati anche funzioni.

Vediamo ora cosa accade all'avvio di un programma. Durante la fase di caricamento del programma viene predisposta e allocata sullo stack un'area di

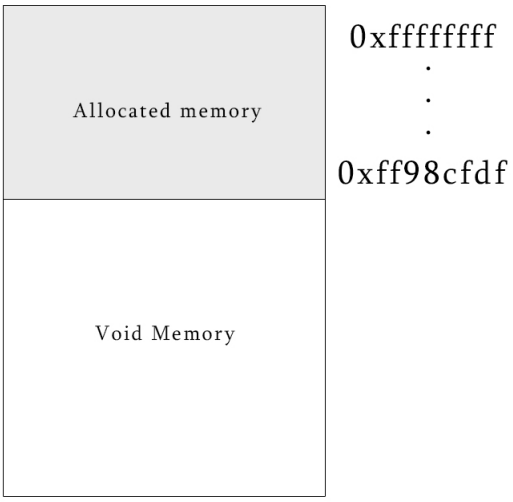


Figura 2.2: Program Memory

memoria adibita alle **variabili globali**, condivisa tra le diverse **unità** o **moduli** del programma stesso (le funzioni). Le variabili globali potranno essere impiegate all'interno dei nostri programmi CLVM per memorizzare come visto, oggetti condivisi, indipendenti dal processo di chiamate nello stack. Per semplicità denoteremo tale area di memoria con il termine **GV**: ossia **Global Variables**.

Lo stack ed i record di attivazione

Ogni funzione (sottoprogramma) ha a disposizione, durante il suo ciclo di attività un'area di memoria riservata nello stack chiamata **record di attivazione**. Ogni record di attivazione conterrà eventuali variabili e parametri locali alla funzione stessa che verranno distrutti non appena il modulo terminerá la sua esecuzione e ritornerà il controllo al chiamante. Possiamo affermare che lo stack durante l'esecuzione di un programma è una struttura che cresce e decresce dinamicamente a seconda dell'esecuzione corrente del programma; in particolare secondo quanto visto in precedenza lo stack cresce per così dire verso il basso.

Supponiamo ora che sia in esecuzione un'unità di programma che per semplicità chiameremo

$$P$$

.

Una volta allocata l'area di memoria per le variabili globali per P, viene allocato un nuovo record di attivazione sulla cima dello stack per consentire l'e-

secuzione del modulo principale, l'**entry point**, (presente in ogni programma) vedi (*fig 2.3.a*).

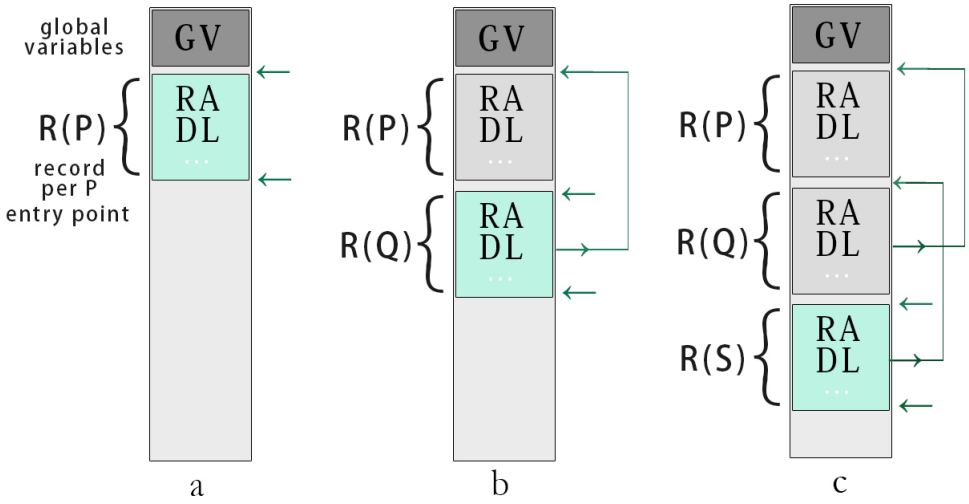


Figura 2.3: Record di attivazione

Durante l'esecuzione del programma nell'area di memoria allocata per P si possono memorizzare valori intermedi di ogni tipo, relativi a risultati intermedi di calcoli, parametri ed altre informazioni di varia natura. Quando l'unità principale invoca un'altra unità ad esempio Q di supporto, ciò che accade nello stack è mostrato in figura (*2.3.b*). Un nuovo record di attivazione viene allocato in cima allo stack ed il record relativo all'unità corrente viene congelato momentaneamente. Il controllo è ora attivo sul record relativo Q, che a sua vola potrebbe invocare altri sottomoduli, generando altri record nello stack (*fig 2.3.c*).

Questo meccanismo di generazione di record nello stack si arresta quando l'ultimo sottoprogramma completa il suo ciclo di esecuzione, ovvero termina l'ultima istruzione, che compone il suo corpo, e quindi ritorna al chiamante. In generale il valore di ritorno può essere sia un dato che vuoto. Supponiamo che venga restituito un qualche valore intermedio in una espressione numerica, richiesto dagli altri sotto-moduli presenti nel programma. Quello che accade é mostrato in figura: Il sottoprogramma che ritorna al chiamante dopo aver terminato il suo compito libera l'area di stack e viene sovrascritto dal risultato (sostanzialmente possiamo per cosí dire che esso si trasforma nel risultato dell'elaborazione).

Ci viene in mente ora una domanda assai semplice: Come può il modulo chiamante sapere dove si trova il risultato appena elaborato? La risposta è triviale: Per poter restituire il controllo al modulo Q, una volta terminata l'esecuzione di S, in S stesso deve esser memorizzato **l'indirizzo di ritorno**

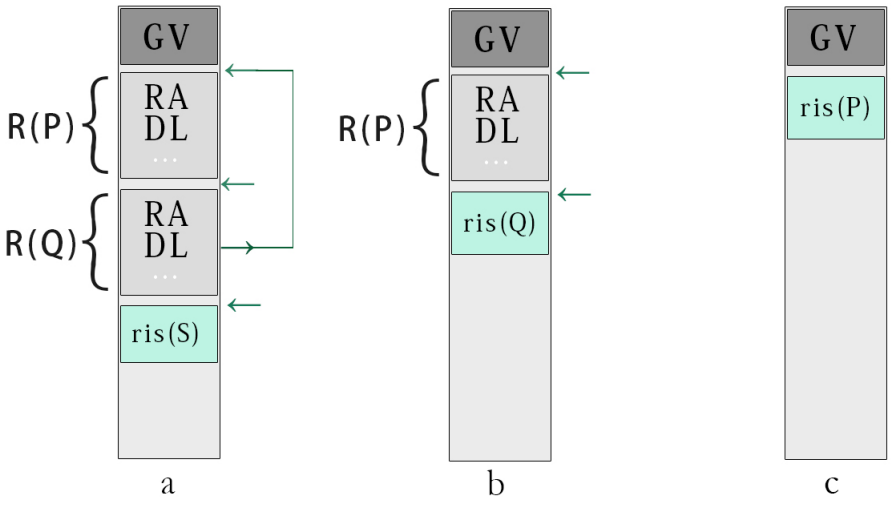


Figura 2.4: Record di attivazione: meccanismo di ritorno

ossia, *l'indirizzo dell'istruzione successiva all'ultima istruzione eseguita da Q prima che questi invocasse S*. In questo modo dopo aver eseguito l'ultima istruzione in S la prossima istruzione ad essere eseguita corrisponde a quella puntata dall'indirizzo di ritorno (RA)[5]

[5]Quello che accade e che il valore del Program Counter viene aggiornato con l'indirizzo di ritorno: PC ← RA;

La struttura dei frame

Ogni frame, come è stato mostrato in figura, include due registri speciali indicati rispettivamente come RA: **Return Address** ed DL: **Dynamic Link**. Si tratta di due registri di supporto indispensabili per la gestione dei frame. Quando viene generato un frame a seguito di una chiamata a funzione in cui vengono passati parametri, questi devono essere memorizzati nel frame stesso, inoltre come visto, per poter ritornare il controllo al chiamante bisogna memorizzare l'indirizzo dell'istruzione successiva alla chiamata (questa informazione viene memorizzata nell'Address Register nel frame stesso. Il link dinamico invece rappresenta l'indirizzo del frame relativo al chiamante e viene memorizzato anch'esso all'interno del frame: Riassumendo in figura è mostrata la struttura di un generico frame:

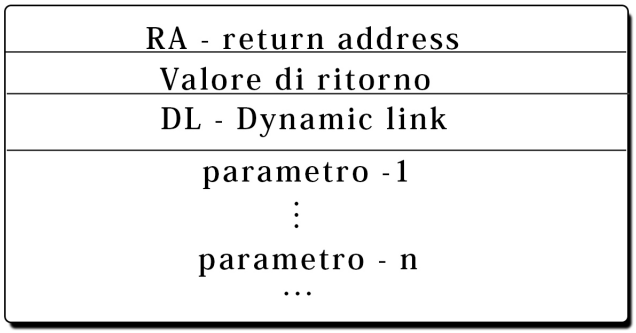


Figura 2.5: Struttura di un frame

Per la gestione dei frame, inoltre bisogna avere a disposizione altri due registri di localizzazione del frame stesso all'interno della memoria. Il registro SP: **Stack Pointer**, il quale punta alla prima posizione disponibile dello stack (la testa) o cima dello stack, ed il registro FP: **Frame Pointer** che punta all'indirizzo base dell'ultimo frame generato[6].

Heap e liste dinamiche

Mentre l'area di stack viene impiegata per la gestione delle chiamate a funzione e tutto il discorso relativo al meccanismo della generazione ed il rilascio dei record di attivazione, nella memoria (programma), vi è un'altra area chiama ta **heap** (il mucchio). La memoria heap è organizzata in **liste**. Una lista è essenzialmente un'estensione del concetto semplice di variabile ed è utilizzata principalmente per allocare dinamicamente i dati generati durante l'esecuzione

[6]L'ultimo frame generato risiede sempre in cima allo stack

del programma. Il modo in cui una lista è gestita è molto semplice. Viene memorizzato un riferimento (puntatore) al primo nodo della lista nello stack[7]. I nodi della lista, assieme al loro contenuto vengono memorizzati invece nell'heap. La convenzione che viene adottata quasi sempre (e quì non si fa eccezione) e che se la lista non contiene nodi[8] il suo puntatore viene inizializzato a **-1**. A differenza dello stack che come abbiamo visto cresce verso il basso, l'heap cresce verso l'alto.[9]

Memorizzazione dei nodi

Un nodo di una lista viene memorizzato mediante un insieme di due celle contigue. Nella prima cella (quella avente indirizzo maggiore) viene memorizato il valore numerico dell'elemento, mentre in alto (quella con indirizzo minore), il puntatore al prossimo nodo (-1 se ci troviamo nell'ultimo nodo).

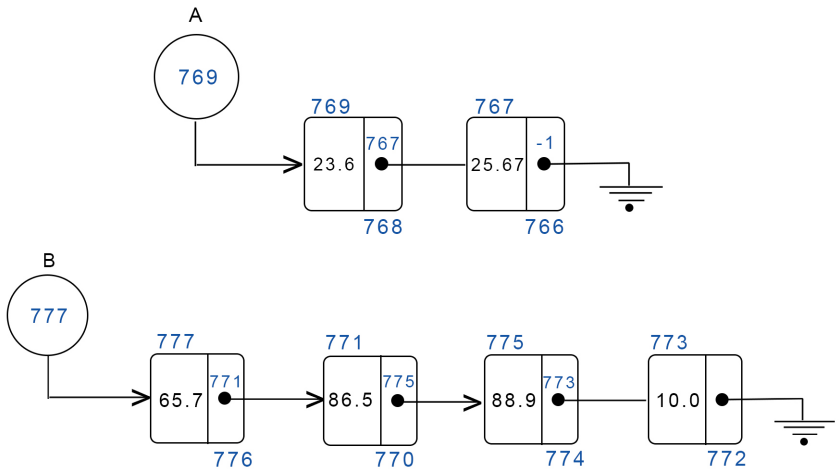


Figura 2.6: Esempi di liste dinamiche

Il registro HP

La macchina calculist prevede un registro dedicato alla gestione dell'heap chiamato **HP** ovvero **Heap Pointer**. Tale registro punta alla prima cella di memoria libera disponibile. In questo modo ogni qualvolta dobbiamo creare un nuovo nodo ed allocare memoria heap sappiamo già dove andare a memorizzare i dati. Nella figura è riportato un grafico dimostrativo della gestione della

[7] Ad esempio nel record relativo ad un sottoprogramma corrente
[8] Una lista che non contiene nodi è una lista vuota
[9] Gli indirizzi di memoria hanno valore minore al crescere dell'heap

memoria in cui vengono messi in risalto lo stack (in alto) e l'heap (in basso); la figura 2.6 mostra, invece le liste in maniera schematica indipendente dall'ordine degli indirizzi in riferimanto all' effettiva memorizzazione.

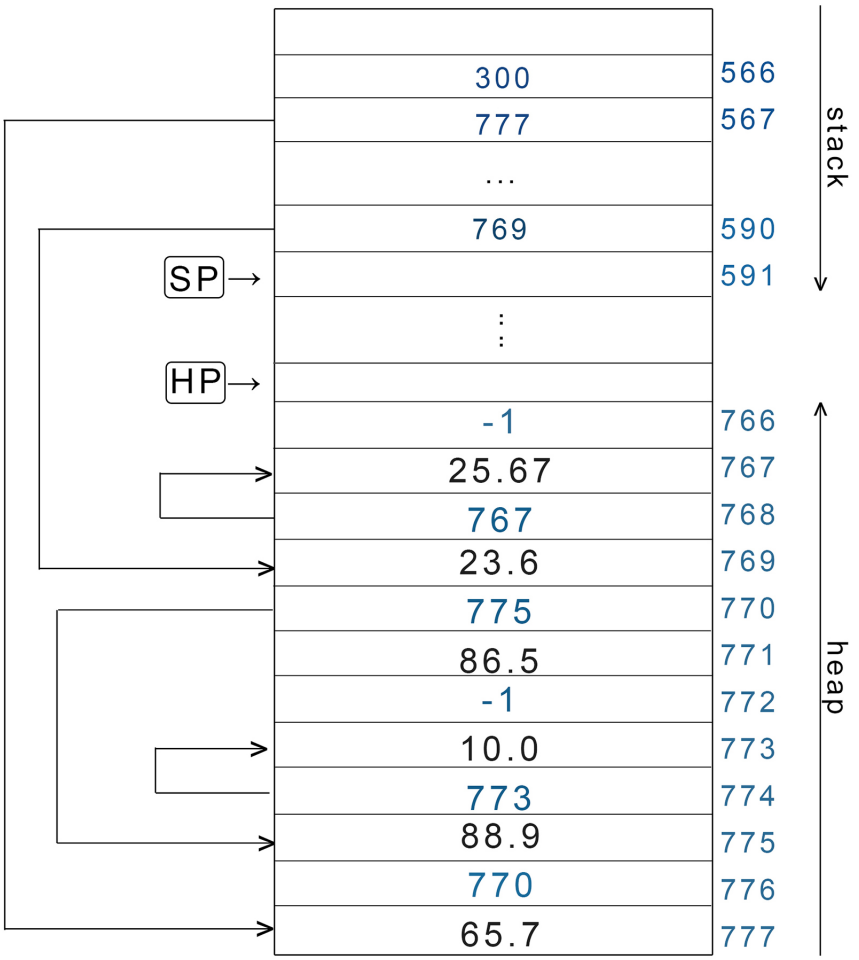


Figura 2.7: Stack ed Heap nel modello di memoria calculist

Osserviamo che i nodi nell'heap non hanno un ordine lineare. Mentre l'ordine dei degli indirizzi è rispettato nell'ambito di un nodo, non è detta che tale ordine venga rispettato nell'ambito dell'intera memoria.[10]

[10]Infatti in ogni nodo l'area ad indirizzo minore contiene il dato numerico, quella ad indirizzo maggiore, il puntatore al nodo successivo e ciò dimostra l'ordine in un nodo. Il nodo 775 tuttavia contiene un puntatore ad un'area casuale non è affatto vero che esso punti all'area di memoria contigua ad esso.

Ulteriori registri

La macchina calculist contiene ulteriori registri, molti dei quali fondamentali per il funzionamento di base della macchina stessa, altri invece a supporto della programmazione. Di seguito vedremo più in dettaglio molte delle caratteristiche di ognuno di essi.

Registro CT: Il registro **Counter** o CT è utilizzato sostanzialmente come contatore di cicli di istruzione. Un contatore è semplicemente un intero che ad ogni ciclo di iterativo di una o più istruzioni, viene incrementato o decrementato di una singola unità. Nel modello calculist (l'azione su CT corrisponde ad un decremento o incremento di 1.

Registro IP: **Instruction Pointer** ovvero, puntatore all'istruzione. Fa parte di ogni macchina a programma memorizzato. Memorizza l'indirizzo della prossima istruzione da eseguire.

Registro IR: Il registro IR, **Instruction Register**, rispetto all'IP che contiene l'indirizzo della prossima istruzione da eseguire, esso contiene l'istruzione stessa corrente in esecuzione.

Registro OP: Nel nostro modello di macchina, abbiamo visto che esiste una memoria dedicata all'output. Associato a questa memoria, vi è un registro **Output Pointer**, il quale punta alla prima locazione di memoria output libera. Quando si desidera mostrare a video o stampare un dato si può fare riferimento al seguente registro.

CAPITOLO 3

Analisi delle istruzioni

Questo capitolo rappresenta il cuore della trattazione riguardo alla macchina CalcuList: ovvero le istruzioni. Sostanzialmente vedremo, sulla base del modello di riferimento della macchina (lo schema a blocchi riportato in appendice), come vengono implementate le singole istruzioni, e cosa avviene a livello di ogni **microistruzione**. Presenteremo dapprima il linguaggio RTL: ossia un linguaggio per la descrizione del trasferimento delle informazioni a livello di registro; (da quì l'acronimo: *Register Transfer Language*) ed in seguito vedremo qual è l'effetto macroscopico di ogni istruzione. Analizzeremo quindi come vengono trasferite le informaizoni all'interno della macchina per implementare un'istruzione assembly (vista come un insieme di micro-istruzioni).

3.1 Il linguaggio RTL

RTL è l'acronimo di Register Transfer Language ossia, **Linguaggio di Trasferimento a livello di Registro**. Si tratta di un modello di linguaggio *descrittivo* non è un linguaggio di programmazione, ma bensì un modo standard e semplice per descrivere analiticamente le istruzioni viste come insieme di micro-passi o letteralmente in gerco: **micro-istruzioni**. Il linguaggio RTL esprime sequenze di operazioni, ciascuna delle quali si svolge in un determinato intervallo di tempo elementare detto τ. Ogni microsequenza è costituita da una o più microperazioni, le quali come vedremo possono impiegare diverse parti del sistema per implementare un'istruzione assembly completa. In linea di massima ogni microsequenza esprime le azioni elementari che la parte operativa deve effettuare a seguito di un comando proveniente dalla parte di controllo.

Definizione di RTL

Presentiamo ora la definizione del linguaggio RTL (in maniera relativamente semplicistica), secondo la notazione di Backus (BNF) [1].

$$\langle \mu\text{Sequenza} \rangle ::= \langle \mu\text{Passo} \rangle | \langle \mu\text{Passo} \rangle ; \langle \mu\text{Sequenza} \rangle$$

$$\langle \mu\text{Passo} \rangle ::= \langle \mu\text{Operazione} \rangle | \langle \text{Etichetta} \rangle : \langle \mu\text{Operazione} \rangle$$
$$| \langle \texttt{frase if} \rangle | \langle \text{Etichetta} \rangle : \langle \texttt{frase if} \rangle$$

$$\langle \mu\text{Operazione} \rangle ::= \langle \text{Assegnamento} \rangle | \emptyset | \langle \text{Assegnamento} \rangle, \langle \texttt{frase go} \rangle |$$
$$\emptyset, \langle \texttt{frase go} \rangle$$

$$\langle \text{Assegnamento} \rangle ::= \langle \text{Trasferimento} \rangle | \langle \text{Trasferimento} \rangle, \langle \text{Assegnamento} \rangle$$

$$\langle \text{Trasferimento} \rangle ::= \langle \text{Dato} \rangle \rightarrow \langle \text{Registro} \rangle$$
$$| \langle \text{Dato} \rangle \langle \text{Operando} \rangle \langle \text{Dato} \rangle \rightarrow \langle \text{Registro} \rangle$$
$$| \langle \text{Funzione} \rangle (\langle \text{Argomento} \rangle) \rightarrow \langle \text{Registro} \rangle$$

$$\langle \text{dato} \rangle ::= \langle \text{Registro} \rangle | \langle \text{Ingresso} \rangle | ' \langle \text{Costante} \rangle '$$

$$\langle \text{Argomento} ::= \langle \text{Dato} \rangle | \langle \text{Dato} \rangle, \langle \text{Argomento} \rangle$$

$$\langle \texttt{frase if} \rangle ::= \texttt{if} \langle \text{Condizione} \rangle \texttt{then} \langle \mu\text{Sequenza} \rangle \texttt{else} \langle \mu\text{Sequenza} \rangle \texttt{fi}$$

$$\langle \text{Condizione} \rangle ::= \langle \text{Dato} \rangle \langle \text{Rel} \rangle \langle \text{Dato} \rangle |$$
$$\langle \text{Funzione} \rangle (\langle \text{Argomento} \rangle) \langle \text{Rel} \rangle \langle \text{Dato} \rangle$$

$$\langle \texttt{frase go} \rangle ::= \texttt{goto} \langle \text{Etichetta} \rangle \quad (3.1)$$

Nella definizione mancano ancora altri metasimboli da definire, legati direttamente al tipo di architettura del sistema. Si tratta di enti di più basso livello come ad esempio ⟨Registro⟩, ⟨Op⟩, ⟨Input⟩ ecc, lo faremo di seguito, brevemente richiamando all'attenzione alcune delle componenti dell'architettura di cui faremo uso nelle prossime sezioni.

[1] La BNF (Backus-Naur Form o Backus Normal Form) è una metasintassi, ovvero un formalismo attraverso cui è possibile descrivere la sintassi di linguaggi formali.

⟨REGISTRO⟩ Indica un qualunque registro della macchina: ad esempio come visto precedentemente:

HR1 FP IR

Per indicare una sottoparte di un registro adotteremo la notazione a pedice mediante parentesi quadre $[\cdot]$: $\text{HR1}_{[h,k]}$ si riferisce all'intervallo di celle che vanno da h fino a k.

⟨INPUT⟩ Indica un qualunque canale di ingresso dati.

⟨OUTPUT⟩ Indica un qualunque canale di uscita dati.

⟨COSTANTE⟩ Indica una qualunque sequenza di caratteri alfanumerici *stringa*, tale da esser contenuta in un registro, o in una parte di esso

⟨FUNZIONE⟩ Indica una funzione binaria svolta da un qualunque modulo combinatorio interconnesso direttamente nel sistema hardware del nostro modello di macchina.

⟨OP⟩ Indica uno dei seguenti operatori aritmetici

$$+, -, *, /, \bigvee, \bigwedge$$

⟨REL⟩ Indica uno dei seguenti operatori relazionali e di uguaglianza

$$=, \neq, <, \leq, >, \geq$$

⟨ETICHETTA⟩ Un numero intero.

$\emptyset$ Rappresenta la micro-operazione *vuota*. In questo caso ci si riferisce alla situazione per cui non avviene alcun trasferimento interno ed inoltre ogni registro e/o cella di memoria rimane congelato al valore corrente.

Temporizzazione

Prima di iniziare la trattazione delle istruzioni, facciamo un paio di osservazioni che ci torneranno utili in seguito: Concentriamoci anzitutto sulla singola micro-operazione. Essa è costituita da uno o più trasferimenti eventualmente seguiti da un'istruzione `goto`. Ogni trasferimento rappresenta uno spostamento delle informazioni da registro in registro: l'operando a sinistra di $\rightarrow$ viene trasferito nell'operando di destra - e questa è l'operazione centrale del linguaggio (che si riferisce ad un vero e proprio switching dei flip-flops costituenti il registro stesso).[2]

Ció di cui bisogna tener conto è che ogni gruppo di trasferimenti deve esser eseguito in un intervallo di tempo elementare τ. In RTL questo fatto è espresso mediante l'operatore di serializzazione *virgola*: (,) - il quale, come detto, separa azioni contemporanee, mentre il *punto e virgola*: (;) - indica il passaggio al tempo immediatamente successivo. Si intuisce che ciò che avviene durante il funzionamento della macchina è un susseguirsi di trasferimenti elementari tra registri, alu, moduli combinatori, memorie ecc il tutto connesso mediante delle interconnesioni tra le compenenti stesse: le *piste* o in gergo, il bus di sistema.

Per quanto riguarda la temporizzazione il funzionamento sincrono dei registri, prevede che tutti i trasferimenti avvengano in un tempo δ in cui appare l'impulso, mentre il calcolo delle parti sinistre di $(\rightarrow)$ (svolto dai moduli combinatori) in un tempo σ che precede δ. In tutto l'operazione di sviluppa in un tempo:

$$\tau = \sigma + \delta$$

[2]Dalla teoria delle reti logiche si è visto come un registro sia costituito da una serie di componenti elementari detti bistabili o flip-flops. Si tratta di circuiti sequenziali in grado di mantenere *congelato* per un tempo indeterminato il proprio stato interno, essi costituiscono i mattoni fondamentali con cui vengono implementate le memorie degli elaboratori elettronici ed hanno sostituito le vecchie memorie a nucleo di ferrite.

3.2 Istruzioni Calculist

Iniziamo ora lo studio e l'analisi delle istruzioni della macchina Calculist. Analizzeremo ogni istruzione sia dal punto di vista RTL, quindi di microistruzioni, sia dal punto di vista dell'azione sui dati.

◇

`HALT`

Ogni programma deve informare il sistema quando finisce il suo computo, ossia qual è l'ultima istruzione da eseguire dopo il quale la macchina deve arrestare il ciclo istruzione. L'istruzione `HALT` rappresenta una situazione in cui la macchina non fa nulla, pur essendo in stato di attività.

$$1 : \emptyset, \texttt{goto 1};$$

`FETCH`

L'istruzione `FETCH` è implicita. Nel senso che non è il programmatore che ne fa uso ma è impiegata all'interno della macchina. Fa parte del ciclo istruzione fondamentale che la macchina deve eseguire per poter automatizzare l'esecuzione dei programmi. Quello ch succede è semplice. Durante la fase di fetch viene prelevata la prossima istruzione e viene trasferita nel registro IR.

$$\text{IR} \leftarrow \textsc{PrgMem}[\text{IP}], \quad \text{IP} \leftarrow \text{IP} + 1;$$

`INIT n`

Questa istruzione inizializza samplicemente il frame iniziale dello stack. Il parametro n rappresenta il numero di celle da inizializzare nel frame stesso. Osservate come la μoperazione: $SP \leftarrow IR_{55-24}$, trasferisce il valore di n espresso all'interno di IR dai 32 bit compresi nell'intervallo $55 - 24$.

$$FP \leftarrow 0, \quad ; SP \leftarrow IR_{55-24}$$

`PUSH val`

Inserisce in cima allo stack il valore val (espresso come double). I registri contatore SP ed IP vengono automaticamente incrementati di 1 per posizionarsi sulla cella successiva, in ambedue le memorie.

$$\text{DataMem[SP]} \leftarrow \text{PrgMem[SP]}, \quad \text{SP} \leftarrow \text{SP} + 1, \quad \text{IP} \leftarrow \text{IP} + 1;$$

`POP`

Molto semplice da un punto di vista implementativo. Sostanzialmente l'istruzione provoca un decremento del registro SP, e l'effetto è quello di eliminare il valore sulla cima dello stack. Il tutto avviene in un unico colpo di clock, in quanto il registro SP è un registro funzione che predispone internamente il decremento di 1.

$$\text{SP} \leftarrow \text{SP-1};$$

`DUPL`

Duplica il valore in cima allo stack. Predispone lo stack aggiornandone la dimensione e riordinando i puntatori. Il tutto viene eseguito in 2 colpi di clock.

$$\begin{gathered}\text{HR1} \leftarrow \text{DataMem[SP-1]};\\ \text{DataMem[SP]} \leftarrow \text{HR1}, \quad \text{SP} \leftarrow \text{SP+1};\end{gathered}$$

`CALL ind`

Si tratta di un'istruzione fondamentale per la gestione delle chiamate a procedure (funzioni). Anzitutto viene salvato nello stack, il valore del registro IP (per poter riprendere l'esecuzione del programma principale dal punto in cui si passa il controllo al metodo); dopodichè viene aggiornato il valore di IP ed il controllo salta all'istruzione di indirizzo *ind* che viene prelevato dalla sottoparte a 32 bit del registro IR_{55-24}.

$$\text{DataMem[SP]} \leftarrow \text{IP}, \quad \text{SP} \leftarrow \text{SP}+1, \text{IR} \leftarrow IR_{55-24}$$

`RESERVE`

Riserva 2 locazioni libere sullo stack a seguito di una chiamata di CALL (le locazioni verranno poi impiegate per memorizzare il valore di ritorno e l'indirizzo di ritorno)[3].

$$\text{SP} \leftarrow \text{SP}+2;.$$

`RETURN`

Analoga all'istruzione HALT. Blocca l'esecuzione del programma principale.

$$\langle \text{HALT} \rangle.$$

[3]Si tratta di RA e DL

`RETURNVAL`

Ritorna il controllo al chiamante. In primo luogo viene memorizzato nel valore di IP l'indirizzo di ritorno (che si trova nella locazione puntata da FP) per ritornare il controllo dal punto in cui si è utilizzata una CALL ind, successivamente si inserisce all'inizio del frame il valore di ritorno (che si trova sulla cima dello stack); infine viene aggiornato il puntatore al frame chiamante.

$$\begin{gathered}\text{IP} \leftarrow \textsc{DataMem}[\text{FP}];\\ \text{HR1} \leftarrow \textsc{DataMem}[\text{SP-1}];\\ \textsc{DataMem}[\text{FP}] \leftarrow \text{HR1}, \quad \text{SP} \leftarrow \text{FP}+1;\\ \text{FP} \leftarrow \textsc{DataMem}[\text{SP}];.\end{gathered}$$

`NEG`

Cambia (inverte) il segno dell'elemento che si trova sulla cima dello stack. Osservate come per poter invertire il segno di HR1, se questi non prevede una funzione incapsulata, bisogna far intervenire la ALU.

$$\begin{gathered}\text{HR1} \leftarrow \textsc{DataMem}[\text{SP-1}];\\ \textsc{DataMem}[\text{SP-1}] \leftarrow \text{-HR1};\end{gathered}$$

`ADD`

Esegue la somma tra gli ultimi due elementi sulla cima dello stack. Naturalmente il risultato va memorizzato al posto dei due operandi. Le μoperazioni effettuano oltre all'operazione, gli aggiustamenti dello stack.

$$\begin{gathered}\text{HR1} \leftarrow \textsc{DataMem}[\text{SP-1}], \text{SP} \leftarrow \textbf{SP-1};\\ \text{HR1} \leftarrow \text{HR1} + \textsc{DataMem}[\text{SP-1}];\\ \textsc{DataMem}[\text{SP-1}] \leftarrow \text{HR1};\end{gathered}$$

`SUB`

Esegue la differenza tra gli ultimi due elementi sulla cima dello stack[4]. Naturalmente il risultato va memorizzato al posto dei due operandi. Le μoperazioni effettuano oltre all'operazione, gli aggiustamenti dello stack.

$$\begin{gathered}\text{HR1} \leftarrow \text{DataMem[SP-2]};\\ \text{HR1} \leftarrow \text{DataMem[SP-1]} - \text{HR1}, \text{SP} \leftarrow \textbf{SP-1};\\ \text{DataMem[SP-1]} \leftarrow \text{HR1};\end{gathered}$$

`SWAP n`

Scambia l'elemento in cima allo stack con l'elemento a distanza n dalla cima. Se $n = 0$, ovviamente non vi è alcuno scambio. Se $n = 1$, allora sono scambiati i due elementi in cima allo stack (classico swap).

$$\begin{gathered}\text{HR1} \leftarrow \text{DataMem[SP-n-1]};\\ \text{HR2} \leftarrow \text{DataMem[SP-1]};\\ \text{DataMem[SP-n-1]} \leftarrow \text{HR2};\\ \text{DataMem[SP-1]} \leftarrow \text{HR1};\end{gathered}$$

`START n`

Costituisce il prologo di un'unità di programma. Sposta nella locazione 0 del frame l'indirizzo di rientro che era in cima allo stack - questa locazione verrà poi resa disponibile. Inoltre inserisce nella locazione 1 del frame il link dinamico (puntatore al frame chiamante).

$$\begin{gathered}\text{HR1} \leftarrow \text{DataMem[SP-1]}, \quad \text{SP} \leftarrow \textbf{SP-1};\\ \text{DataMem[SP-n-2]} \leftarrow \text{HR1};\\ \text{DataMem[SP-n-1]} \leftarrow \text{FP}, \quad \text{FP} \leftarrow \textbf{SP-n-2};\end{gathered}$$

[4]Per essere più precisi, tra il penultimo elemento(minuendo) e l'ultimo (sottraendo)

`PUSHFP n`

Inserisce in cima allo stack l'indirizzo assoluto della locazione n-sima del frame. Primo passo per accedere al valore di un parametro.

$$\text{DataMem[SP]} \leftarrow \text{FP+n}, \quad \text{SP} \leftarrow \mathbf{SP+1};$$

`DEREF`

Effettua il deriferimento, ossia: sostituisce il contenuto della cella con il valore puntato[5].

$$\begin{gathered}\text{HR1} \leftarrow \text{DataMem[SP-1]};\\ \text{HR1} \leftarrow \text{DataMem}[HR1_{0-31}];\\ \text{DataMem[SP-1]} \leftarrow \text{HR1};\end{gathered}$$

`CMP`

Effettua una comparazione tra gli ultimi due valori sulla cima dello stack e memorizza l'esito in un flag della ALU.

$$\begin{gathered}\text{HR1} \leftarrow \text{DataMem[SP-2]};\\ flags \leftarrow \text{DataMem[SP-1]} - \text{HR1}, \quad \text{SP} \leftarrow \text{SP-2};\end{gathered}$$

[5]Simile all'effetto dell'operatore $*$ del C.

Istruzioni di salto

In questa sezione analizziamo il set delle istruzioni di salto. Nella figura è evidenziata la parte dell'architettura addetta al controllo del flusso del programma. Le istruzioni di salto vengono implementate mediante un insieme di flags di stato (flip-flops), i quali incapsulano il risultato di un'operazione di comparazione e determinanto l'andamento del flusso di esecuzione dell'attività principale. Analizzeremo in primo luogo le istruzioni che coinvolgono i flags: AOF, GEZF, LZF, LEZF, GZF, ZF che fanno intervenire il blocco ALU nell'esecuzione, successivamente, in secondo luogo, vedremo il set delle istruzioni associate al registro contatore CT (in figura a sinistra).

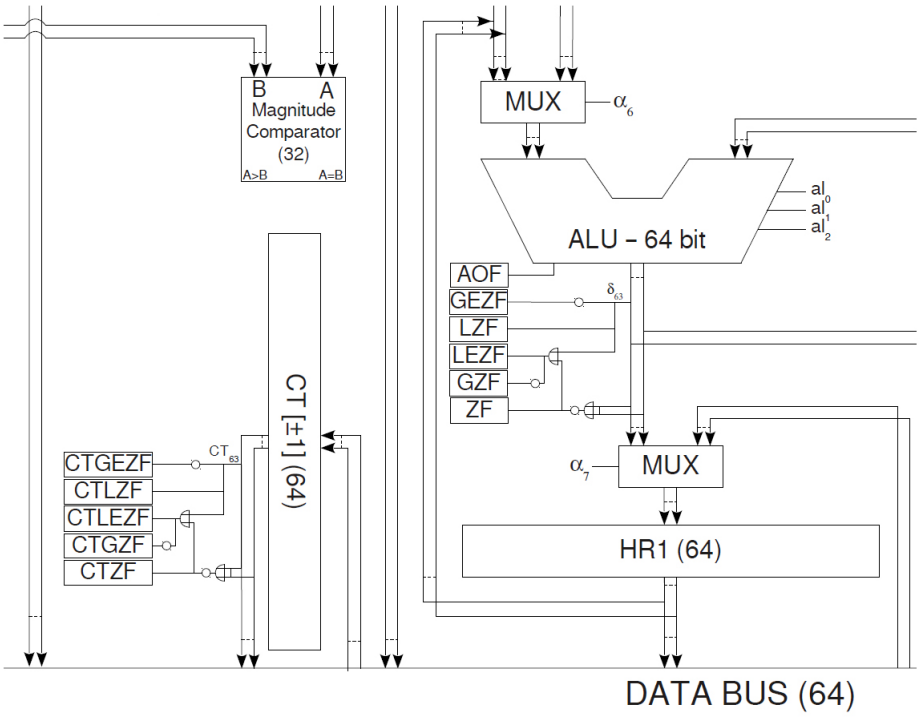


Figura 3.1: Flag di stato

I flags associati al contatore CT, sono: CTGEZF, CTLZF, CTLEZF, CTGZF, CTZF, che analizzeremo di seguito.

I flag di stato

La macchina calculist prevede 11 flag di stato, 5 associati al registro contatore CT, 6 connessi direttamente a valle del modulo ALU. Per quanto riguarda i $flags_{ALU}$, essi vengono implicitamente settati ogni qualvolta viene eseguita l'istruzione di comparazione CMP, parimenti, i $flags_{CT}$ vengono settati ogni qualvolta viene impiegato il registro CT in una microistruzione. Analizziamo ora le istuzioni di salto che impiegano dietro le quinte i flags.

`JUMP ind`

È l'unica istruzione di salto che non fa uso di flags. Essa è nota come *salto incondizionato*, nel senso che essa non dipende al valore corrente di uno qualunque dei flags di stato. L'istruzione è composta da un unico micropasso, in cui viene aggiornato direttamente il valore dell'instruction pointer all'indirizzo `ind`.

$$\text{IP} \leftarrow IR_{55-24};$$

`JUMPZ ind`

Viene controllato il contenuto del flag ZF (*Zero Flag*). Se il valore del flag è 1[6] viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$.

$$\begin{array}{c} \texttt{if ZF} == 1\texttt{then} \\ \text{IP} \leftarrow IR_{55-24}; \\ \texttt{else} \\ \emptyset \\ \texttt{end} \end{array}$$

[6]ciò significa che il flag è stato modificato al valore 'VERO'

`JUMPNZ ind`

Funziona in maniera opposta alla precedente. Viene controllato il contenuto del flag ZF (*Zero Flag*). Se il valore del flag è 0 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$.

.
`if` ZF $==$ 0 `then`
IP $\leftarrow IR_{55-24}$;
`else`
$\emptyset$
`end`

`JUMPLZ ind`

Viene controllato il contenuto del flag LZF (*Lower Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag LZF viene settato ad 1 quando a seguito di una istruzione CMP il primo operando è minore del secondo.

.
`if` LZF $==$ 1 `then`
IP $\leftarrow IR_{55-24}$;
`else`
$\emptyset$
`end`

`JUMPLEZ ind`

Viene controllato il contenuto del flag LEZF (*Lower or Equal Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag $LEZF$ viene settato ad 1 quando a seguito di una istruzione CMP il primo operando è minore o uguale del secondo.

.
`if` LEZF $==$ 1 `then`
IP $\leftarrow IR_{55-24}$;
`else`
$\emptyset$
`end`

`JUMPGZ ind`

Viene controllato il contenuto del flag GZF (*Graded Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag GZF viene settato ad 1 quando a seguito di una istruzione CMP il primo operando è maggiore del secondo.

```
.
if GZF == 1 then
IP ← IR_{55−24};
else
∅
end
```

`JUMPGEZ ind`

Viene controllato il contenuto del flag GEZF (*Graded or Equal Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag $GEZF$ viene settato ad 1 quando a seguito di una istruzione CMP il primo operando è maggiore o uguale del secondo.

```
.
if GEZF == 1 then
IP ← IR_{55−24};
else
∅
end
```

Le seguenti istruzioni fanno riferimento al registro contatore CT.

`JUMPCTZ ind`

Viene controllato il contenuto del flag CTZF (*Counter Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag $CTZF$ viene settato ad 1 quando il valore del registro CT è nullo.

```
.
if CTZF == 1 then
IP ← IR_{55−24};
else
∅
end
```

`JUMPCTGZ ind`

Viene controllato il contenuto del flag CTGZF (*Counter Graded Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag $CTGZF$ viene settato ad 1 quando il valore del registro CT è maggiore di zero.

```
.
if CTGZF == 1 then
IP ← IR_{55−24};
else
∅
end
```

`JUMPCTGEZ ind`

Viene controllato il contenuto del flag CTGEZF (*Counter Graded or Equal Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag $CTGEZF$ viene settato ad 1 quando il valore del registro CT è maggiore o uguale a zero.

.
`if` CTGEZF $==$ 1 `then`
IP $\leftarrow IR_{55-24}$;
`else`
$\emptyset$
`end`

`JUMPCTLZ ind`

Viene controllato il contenuto del flag CTLZF (*Counter Lower Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag $CTLZF$ viene settato ad 1 quando il valore del registro CT è minore di zero.

.
`if` CTLZF $==$ 1 `then`
IP $\leftarrow IR_{55-24}$;
`else`
$\emptyset$
`end`

`JUMPCTLEZ ind`

Viene controllato il contenuto del flag CTLEZF (*Counter Lower or Equal Zero Flag*). Se il valore del flag è 1 viene aggiornato il valore dell'instruction pointer all'indirizzo `ind`, altrimenti non si fa nulla: $\emptyset$. Il flag $CTLEZF$ viene settato ad 1 quando il valore del registro CT è minore o uguale a zero.

.
`if` CTLEZF $==$ 1 `then`
IP $\leftarrow IR_{55-24}$;
`else`
$\emptyset$
`end`

`VALCT`

Il valore contenuto nel registro contatore viene copiato sulla cima dello stack.

$$\text{DataMem}[\text{SP}] \leftarrow \text{CT}; \text{SP} \leftarrow \text{SP}+1$$

`SETCT`

Il valore in cima allo stack viene memorizzato nel registro contatore.

$$\text{CT} \leftarrow \text{DataMem}[\text{SP-1}]; \text{SP} \leftarrow \text{SP-1}$$

`DECRCT`

Il valore del registro contatore viene decrementato di 1.

$$\text{CT} \leftarrow \text{CT-1};$$

`INCRCT`

Il valore del registro contatore viene incrementato di 1.

$$\text{CT} \leftarrow \text{CT}+1;$$

`MODV`

Il valore del registro contatore viene incrementato di 1.

$$\begin{array}{c} \text{HR2} \leftarrow \text{DataMem}[\text{SP-1}], \quad \text{SP} \leftarrow \textbf{SP-1}; \\ \text{HR1} \leftarrow \text{DataMem}[\text{SP-1}], \quad \text{SP} \leftarrow \textbf{SP-1}; \\ \text{DataMem}[\text{HR1}] \leftarrow \text{HR2} \end{array}$$

`PRINT`

Il valore che si trova in cima allo stack, viene copiato nella memoria per essere stampato in uscita.

$$\text{OutMem[OP]} \leftarrow \text{DataMem[SP-1]}, \quad \text{OP} \leftarrow \textbf{OP+1}, \quad \text{SP} \leftarrow \textbf{SP-1};$$

Istruzioni per la gestione delle liste

Presentiamo in questa sezione, un set di istruzioni più applicativo; orientato cioè alla implementazione dei programmi calculist. Rispetto alle istruzioni viste in precedenza, che fanno parte di molte architetture in generale, queste istruzioni sono strettamente connesse, da una parte all'architettura della macchina stessa, dall'altra alla tipologia d'uso nelle applicazioni.

Gestione delle liste

Le liste possono essere gestite secondo due modalità diverse a seconda se:

La lista è APERTA: (è in fase di costruzione, il programmatore deve ancora definire il suo contenuto).

La lista è CHIUSA: (È già costruita).

La gestione delle liste aperte, aviene, mediante 2 locazioni (come già accennato nel capitolo 2) dello stack, che puntano rispettivamente alla testa ed alla coda della lista. L'esempio precedente mostrava un solo puntatore di testa, ma questo è in accordo con la convenzione adottata in fase di uso, in quanto si usa solo il puntatore di testa e non quello di coda.

◇

`SLIST`

(Start List). Crea una lista **aperta** vuota. Alloca due locazioni nello stack entrambe con valore pari a -1. La figura mostra la situazione in memoria prima e dopo l'esecuzione dell'istruzione.

$$\begin{array}{l} \textsc{DataMem}[\mathrm{SP}] \leftarrow -1, \quad \mathrm{SP} \leftarrow \mathbf{SP{+}1}; \\ \textsc{DataMem}[\mathrm{SP}] \leftarrow -1, \quad \mathrm{SP} \leftarrow \mathbf{SP{+}1}; \end{array}$$

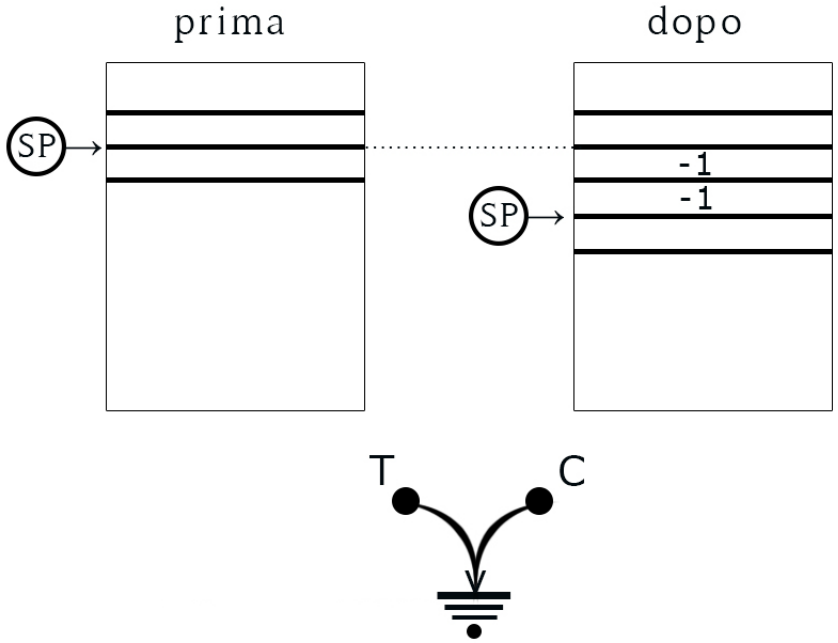


Figura 3.2: contenuto dello stack prima e dopo l'esecuzione di SLIST

`HLIST`

(Head List). Crea la testa della lista. Vengono allocate nell'heap due locazioni contigue. In quella più in basso, viene copiato il valore in cima allo stack, in quella più in alto viene posto il valore -1. infine i due valori di testa e coda vengono fatti puntare all'unico elemento che contiene la lista, che in questo caso è sia testa che coda.

$$\begin{gathered}\textsc{DataMem}[\text{HR1}] \leftarrow \textsc{DataMem}[\text{SP-1}], \quad SP \leftarrow \textbf{SP-1};\\ \textsc{DataMem}[\text{HP}] \leftarrow \text{HR1}, \quad \text{HP} \leftarrow \textbf{HP-1};\\ \textsc{DataMem}[\text{HP}] \leftarrow -1, \quad \text{HP} \leftarrow \textbf{HP-1};\\ \textsc{DataMem}[\text{SP-1}] \leftarrow \text{HP}+2;\\ \textsc{DataMem}[\text{SP-2}] \leftarrow \text{HP}+2;\end{gathered}$$

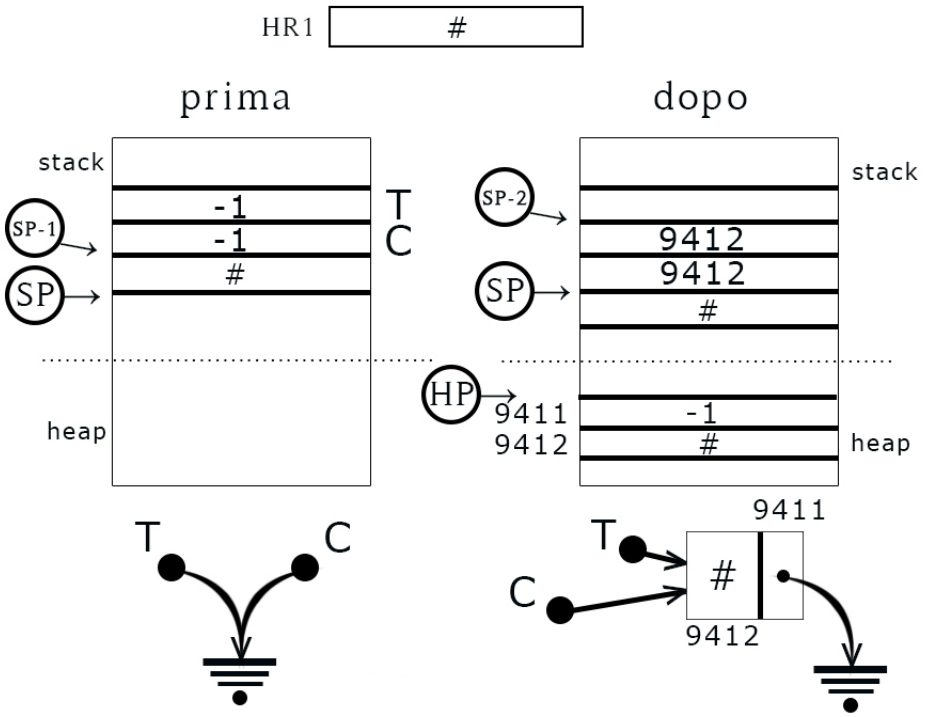
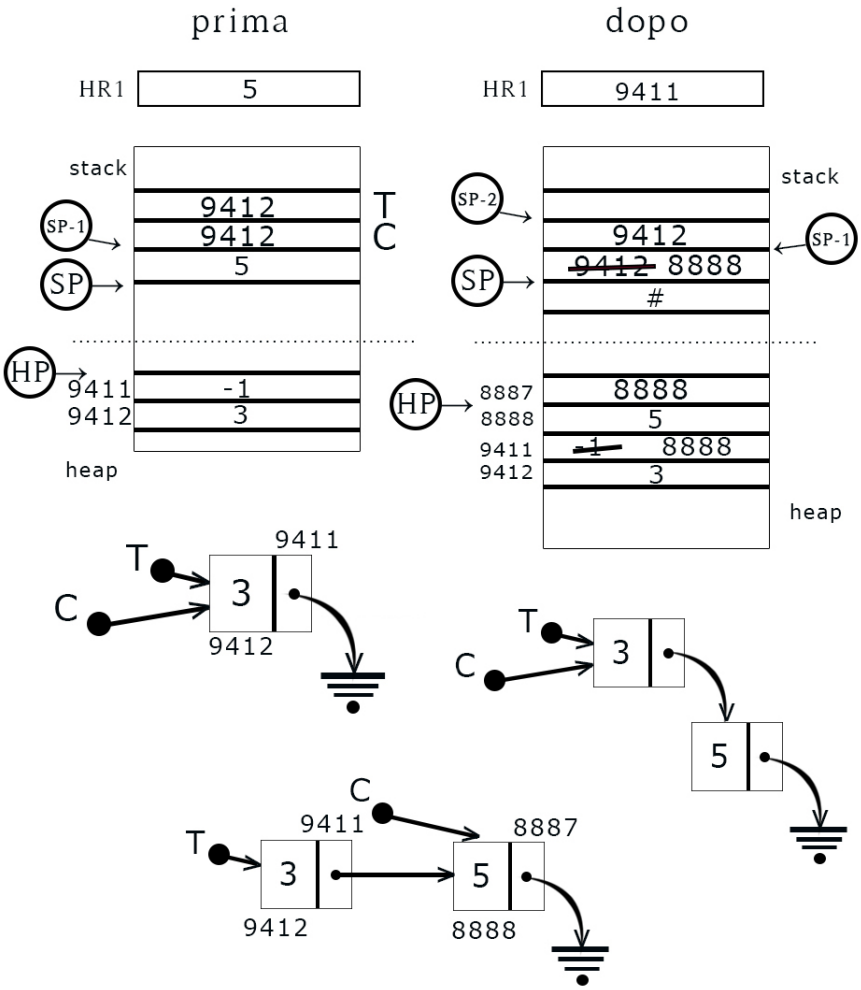


Figura 3.3: contenuto dello stack e dell'heap prima e dopo l'esecuzione di HLIST

`CLIST`

Aggiunge un elemento in coda ad una lista aperta. Vengono allocati due locazioni contigue nel'heap, quella in basso contiene il valore dell'ultimo elemento corrente della lista, copiato dalla locazione in cima allo stack, e quello in alto il puntatore a null: (-1).

$$\text{DataMem[HR1]} \leftarrow \text{DataMem[SP-1]}, \quad SP \leftarrow \textbf{SP-1};$$
$$\text{DataMem[HP]} \leftarrow \text{HR1}, \quad \text{HP} \leftarrow \textbf{HP-1};$$
$$\text{DataMem[HP]} \leftarrow -1, \quad \text{HP} \leftarrow \textbf{HP-1};$$
$$\text{HR1} \leftarrow \text{DM[SP-1]-1};$$
$$\text{DataMem[HR1]} \leftarrow \text{HP+2};$$
$$\text{DataMem[SP-1]} \leftarrow \text{HP+2};$$



`ELIST`

Fa passare una lista dalla modalità aperta a quella chiusa. Viene rimosso dalla cima dello stack il puntatore all'ultimo elemento della lista.[7]

$$SP \leftarrow SP - 1;$$

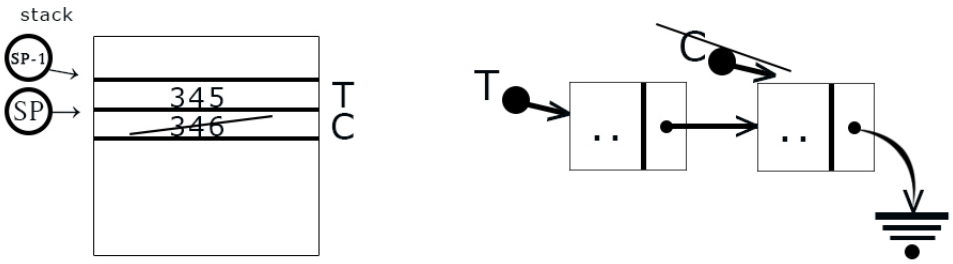


Figura 3.4: esecuzione di ELIST

[7]Le liste chiuse vengono gestite mediante un'unica locazione nello stack che punta alla testa della lista

`SUCCL`

Avanza di un elemento il puntatore all'elemento corrente della lista. L'istruzione mediante un test si accorge quando la lista è terminata. [8]L'indirizzo ad un elemento di una lista viene sostituito con l'indirizzo al successivo di tale elemento.[9].

HR1 ← DM[SP-1]-1;
HR1 ← HR1;
DataMem[SP-1] ← HR1;

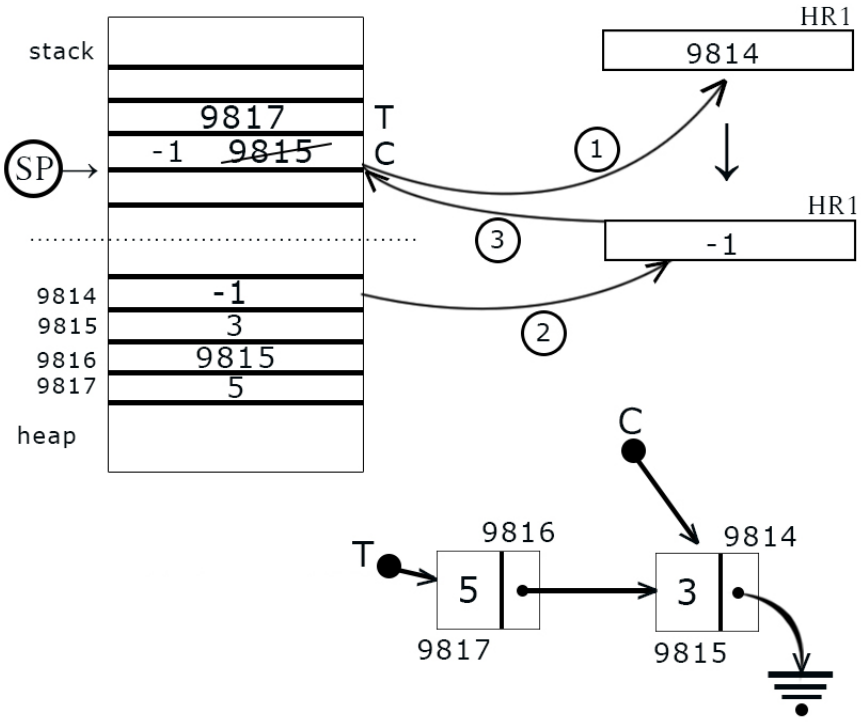


Figura 3.5: contenuto dello stack a seguito dell'esecuzione dell'istruzione SUCCL

[8]Il test hardware da fare per vedere se un puntatore è -1 è un operazione di AND sull'intero registro e si deve ottenere 1

[9]Se applico SUCCL e non c'è alcun elemento nella lista, è un errore del programmatore.

`ALIST`

Prende una lista chiusa e la mette in coda ad una lista aperta, ottenendo una lista aperta. Il puntatore all'ultimo elemento della lista in costruzione, memorizzato nella posizione prima della cima dello stack, viene posto uguale all'indirizzo del primo elemento della lista da appendere in cima allo stack. Quest'ultimo stack viene poi rimosso dallo stack. Il puntatore all'ultimo elemento della lista in costruzione, che è ora in cima allo stack, non punta effettivamente all'ultimo elemento della lista; ma nella prossima istruzione ELIST verr rimosso dallo stack.

$$
\begin{array}{c}
\text{HR2} \leftarrow \text{DM[SP-1]}, \text{SP} \rightarrow \text{SP} - 1; \\
\text{HR1} \leftarrow \text{DM[SP-1]-1}; \\
\text{DataMem[HR1]} \leftarrow \text{HR2}; \\
\text{HR1} \leftarrow \text{HR2}; \\
\text{HR2} \leftarrow \text{DM[SP-1]}; \\
1: \quad \texttt{if } AND(\text{HR1}) = 0 \texttt{ then} \\
\text{HR2} \leftarrow \text{HR1}, \text{HR1} \leftarrow \text{HR1-1} \\
\text{HR1} \leftarrow \text{DM[HR1]}, \texttt{ goto 1}; \\
\texttt{else} \\
\text{DataMem[SP-1]} \leftarrow \text{HR2}; \\
\texttt{end}
\end{array}
$$

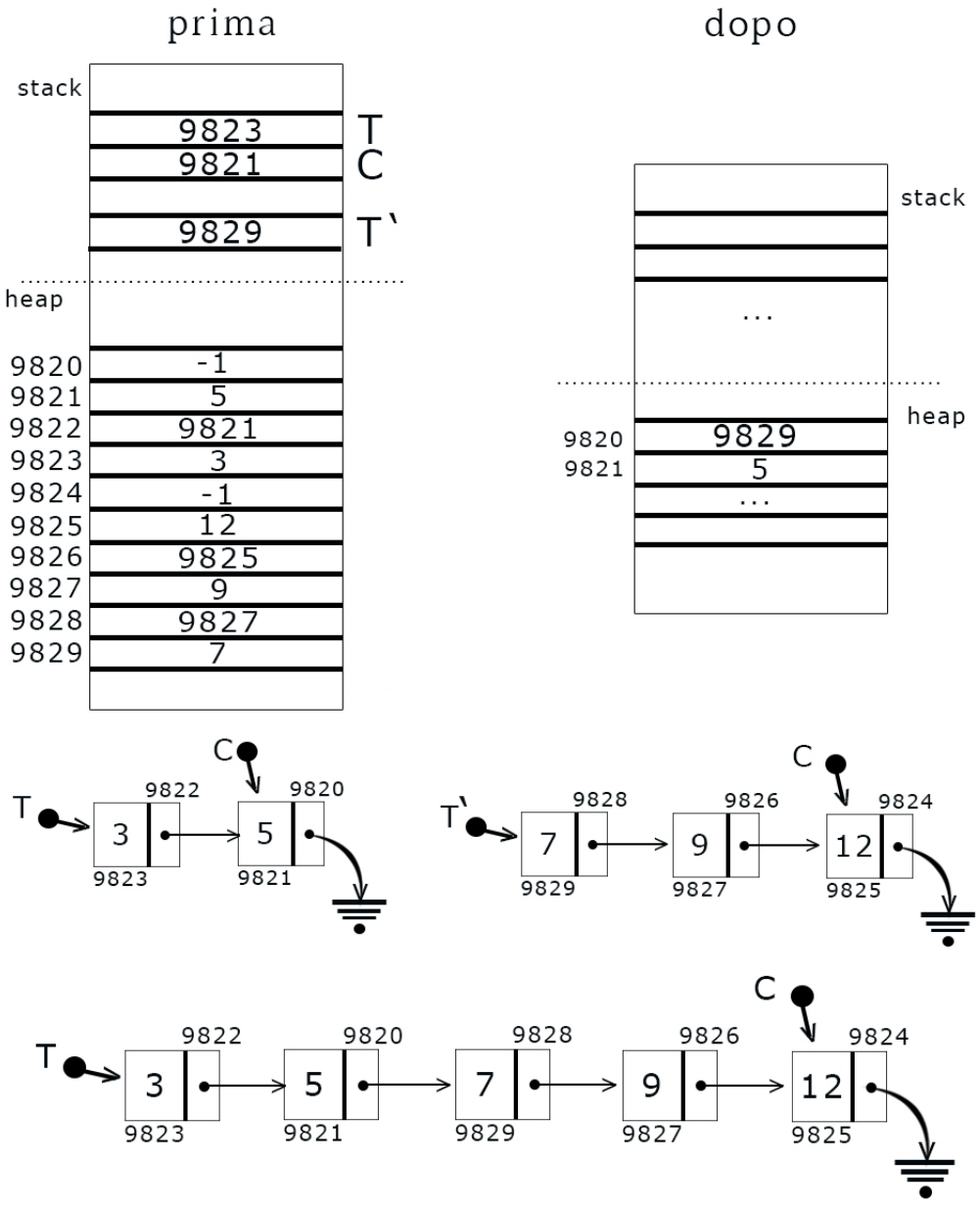


Figura 3.6: contenuto dello stack a seguito dell'esecuzione dell'istruzione ALIST

`AELIST`

Simile ad ALIST (lista aperta con lista chiusa da agganciare), ma il risultato è una lista chiusa.

$$\begin{array}{c} \text{HR2} \leftarrow \text{DM[SP-1]}, \text{SP} \rightarrow \text{SP} - 1; \\ \text{HR1} \leftarrow \text{DM[SP-1]-1}, \text{SP} \rightarrow \text{SP} - 1; \\ \textsc{DataMem}[\text{HR1}] \leftarrow \text{HR2}; \end{array}$$

`PRINTLIST`

Copia i valori contenuti in una lista in memoria di output.

$$\begin{array}{c} \text{HR1} \leftarrow \text{DM[SP-1]}; \\ 1: \quad \texttt{if } AND(\text{HR1}) = 0 \texttt{ then} \\ \textsc{OutMem}[\text{OP}] \leftarrow \textsc{DataMem}[\text{HR1}], \text{HR1} \leftarrow \text{HR1-1}, \text{OP} \leftarrow \text{OP}+1 \\ \text{HR1} \leftarrow \text{DM[HR1]}, \texttt{ goto 1}; \\ \texttt{else} \\ \emptyset; \\ \texttt{end} \end{array}$$

APPENDICE A

Schema a blocchi

In questa appendice viene presentato lo schema dettagliato dell'architettura CalcuList. Partendo dall'alto sono riportate le memorie: data, program e output, insieme ai relativi registri di supporto. La macchina prevede due bus: uno a 64 (per i dati) ed uno a 32 per gli indirizzi dove vengono convogliati i segnali provenienti dalle componenti principali. Il cuore della macchina è costituito da un gruppo di registri, flip-flop e dalla ALU a 64bit; molti dei registri, quali ad esempio FP, HP, CT ecc realizzano come visto le funzioni principali di indirizzamento, meccanismo di ritorno ecc. infine un insieme di MUX-DEMUX è impiegato per il corretto l'instradamento dei segnali.

A.1 Componenti dell'architettura

Lo schema in figura, rappresenta la versione base della macchina. È possibile naturalmente estenderne le funzionalita aggiungendo e/o eliminando alcune delle componenti base. (es registri HR3, Comparatori, mux-demux ecc).

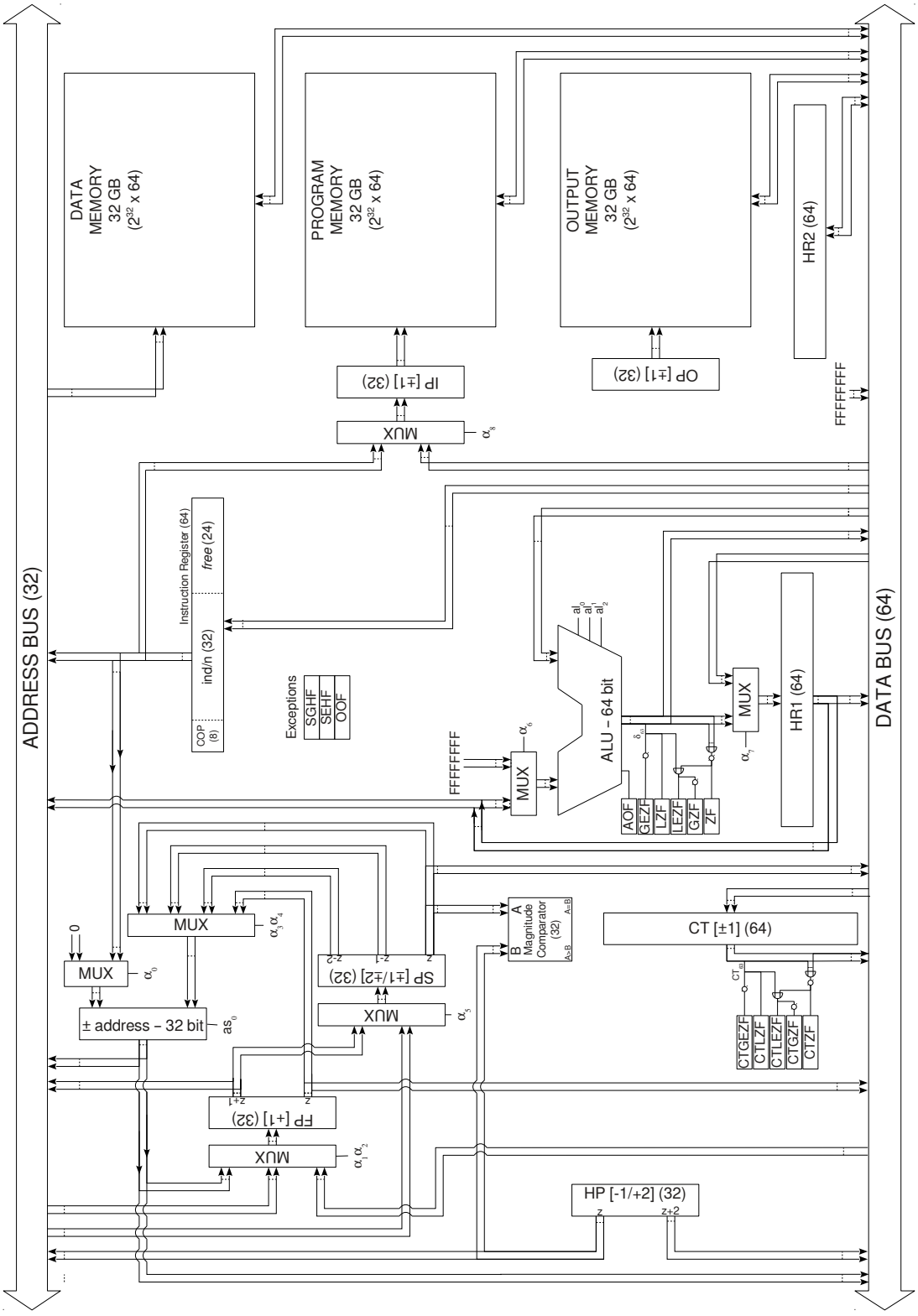


Figura A.1: Schema architetturale CalcuList

A.2 Glossario delle componenti

DATA MEMORY Memoria dati da $32GB = 2^{32} \cdot 64bit$ di capienza. Infatti avendo indirizzi da $32bit$ abbiamo 2^32 combinazioni possibili in cui i bit rappresentano indirizzi. La memoria è utilizzata per memorizzare i dati durante le elaborazioni.

PROGRAM MEMORY Memoria programma da $32GB = 2^{32} \cdot 64bit$ di capienza. Infatti avendo indirizzi da $32bit$ abbiamo 2^32 combinazioni possibili in cui i bit rappresentano indirizzi. La memoria è utilizzata per memorizzare il codice delle istruzioni in ASSEMBLY-CLVM.

> INSTRUCTION POINTER È un registro di tipo contatore ad incremento di 1 unità e punta alla prossima istruzione che dovra essere eseguita. la dimensione rispecchia quella dell'indirizzamento della memoria programma ossia $32bit$.

OUTPUT MEMORY: Memoria dati da $32GB = 2^{32} \cdot 64bit$ di capienza. Infatti avendo indirizzi da $32bit$ abbiamo 2^{32} combinazioni possibili in cui i bit rappresentano indirizzi. La memoria è usata per contenere i dati da stampare in output.

> OUTPUT POINTER: Il registro in questione fa da interfaccia con la memoria di uscita. Esso punta alla prima locazione libera per poter inserire i dati da stampare in output. La dimensione rispecchia quella di un indirizzo a 32bit della memoria stessa il [$\pm$1] indica che il registro è di tipo contatore (ad incremento o decremento di 1.

ADDRESS BUS: Su questo bus viaggiano gli indirizzi per la memoria dati. Quando si fa riferimento alla memoria (ad esempio quando si usano dei nomi di variabili) implicitamente, dietro le quinte, il nome della variabile viene tradotto nel suo indirizzo che punta alla locazione corrispondente nella memoria dati. Ogni indirizzo è a $32bit$.

DATA BUS: Mentre nel bus indirizzi, le informazioni che viaggiano vengono interpretate come indirizzi di memoria dati, i dati stessi viaggiano (dalla memoria verso il resto del sistema e viceversa) in un altro bus(a destra della figura): il bus-dati a $64bit$, perchè come visto il dato contiene più informazioni rispetto ad una istruzione.

ALU: Il modulo Alu (Aritmetic logic unit) corrisponde al nucleo di elaborazione della nostra macchina. La struttura è a $64bit$ in accordo con il formato dei dati. Il sistema di flip-flops a valle, come sarà spiegato di seguito è impiegato per realizzare alcune delle istruzioni di comparazione, somma,

sottrazione ecc sui dati stessi. Inoltre i segnali **alpha** al_0, al_1, al_2, vengono impiegati nel micro-codice per implementare le istruzioni assembly.

Counter CT: Il registro contatore è molto utile nell'implementazione delle istruzioni cicliche che ad esempio scorrono delle liste. Il registro è molto utile in quanto (i flags) a valle del registro sono utili per catturare eventuali situazioni particolari (es: valore maggiore di zero, minore, uguale a zero ecc)

Heap Pointer: Un registro contatore sia ad incremento singolo che doppio, usato per indirizzare l'area heap nella memoria dati

HR1, HR2: Sono dei registri di supporto general pourpose. Possono essere usati per trattenere valori temporanei, durante il ciclo di un'istruzione assembly

IR: Instruction Register: contiene l'istruzione corrente che è in esecuzione.

Bibliografia

Luccio F. Pagli L. ,Reti logiche e calcolatore, Torino, Bollati Boringhieri, 1991

Bucci G., Calcolatori elettronici - Architettura e organizzazione, s.l., McGraw Hill, 2009

Saccà D., The CalcuList Release 4 Tutorial