Abbiamo immaginato una ripartizione della memoria in tre parti: una dedicata al codice, una ai dati statici e una chiamata Stack che cresce dal basso verso l’alto (da un indice dello SP grande a un indice piccolo) . Ci siamo posti nella configurazione PDP-11, in cui la Cpu ha 6 registri di uso generale, uno SP e un PC.

L’indirizzo nello Stack Pointer indica la cima dello Stack nel momento attuale. Dopo aver salvato i valori dei registri nello Stack si possono modificare (attraverso l’esecuzione di un secondo programma) e per ripristinarli basta una serie di pop che li riempiano in ordine contrario. Nel PDP-11 l’autodecremento è implementato modificando prima il contenuto del registro e poi di usare il valore già decrementato per trovare una cella della RAM: R <- R-2; RAM[R].

L’autoincremento funziona al contrario: RAM[R]; R <- R+2; (nella funzione POP, R è SP).

La cosa importante è che l’autodecremento e l’autoincremento facciano le cose in ordine opposto, per poter costruire funzioni POP e PUSH non confusionarie.

Una istruzione importante del PDP-11 è la CHIAMATA DI PROCEDURA. Essa viene codificata con l’espansione a 7 bit del codice operativo:

0000100 – ci sono poi 3 bit per il registro **R** del primo operando – e un secondo operando codificato su 3+3 bit.

Questa istruzione è chiamata JSR (Jump To Sub-Routine)

Questa funzione accede in automatico al registro 6 (lo Stack Pointer) ed effettua una PUSH all’interno dello Stack:

SP <- SP-2;

RAM[SP] <- **R**;

Es: 0000100011 – XXXX:

SP <- SP-2;

RAM[SP] <- R2;

Adesso è possibile modificare il contenuto di R senza ripercussioni. Quindi il contenuto del PC viene inserito in R:

R <- PC; //nel nostro esempio R2 <- PC;

Adesso ho liberato il Program Counter, quindi vi posso inserire l’indirizzo per avviare la procedura:

PC <- Indirizzamento immediato. //tramite il registro nel secondo operando.

Per riprendere l’esecuzione del main, è necessaria la funzione RTS (return from subroutine):

000000000011 (il codice operativo è esteso a 13 bit) i 3 bit rimanenti individuano un registro R.

PC <- R; //ovviamente R deve essere lo stesso che è stato indicato prima

R <- RAM[SP];   
SP <- SP + 2;

I codici operativi JSR e RTS in realtà combinano soltanto operazioni che sarebbero comunque realizzabili con due o più istruzioni già definite nel PDP-11, tuttavia avere delle funzioni particolari per questi specifici scopi torna molto utile in fatto di efficienza.

Tipicamente le prime istruzioni di una Subroutine saranno istruzioni di tipo PUSH, perché deve salvare i valori dei registri nello Stack prima di modificarli (per poterli utilizzare) e analogamente prima dell’RTS verranno chiamante tante POP quante erano state le PUSH (in ordine inverso -> LIFO). Un’ottimizzazione che può quindi venire implementata (e che in genere viene implementata in automatico dal compilatore) è appunto la limitazione al massimo del numero di registri da salvare, modificare e poi ripristinare.

Per esempio:

int c = 5;

main() {

int i = 5;

…

i = funz(7);

…

}

Int funz(int p) {

Int a, b;

…

Return a;

}

Il nostro compilatore quindi parte guardando le dichiarazioni di variabili:

c verrà allocata nell’area dedicata ai dati statici (poiché è dichiarata fuori dal main).

Le variabili i,a e b vengono invece allocate all’interno dello Stack (poiché sono locali) -> all’assegnazione di 5 a i corrisponderà un’istruzione di tipo PUSH. Quando viene eseguita la funzione viene allocata memoria nello Stack per le variabili a e b (-> PUSH) e al termine della funzione avvengono i rispettivi POP.

Quindi la memoria è occupata solo finché le variabili restano in vita, dopo viene deallocata.

Quando il programma viene avviato alle varie funzioni vengono assegnati degli indirizzi per allocare le proprie variabili (secondo un algoritmo presente nel gestore di memoria). Per esempio:

main: 152

funz: 740

Nell’esempio il programma partirà all’indirizzo 152, dove ci si aspetta di trovare una PUSH del valore 5. Quando però avviene la chiamata di funzione abbiamo alcuni problemi: innanzitutto bisogna comunicare il valore 7 come parametro della nostra funzione, in secondo luogo dobbiamo prendere il risultato della funzione e assegnarlo alla variabile i. Per quanto riguarda questa comunicazione, ci sono due alternative principali, una più veloce (perché fa uso dei registri interni del processore) e un’altra meno veloce ma più necessaria perché fa uso della struttura dello Stack.

La prima alternativa usa un registro per scrivervi il valore passato come parametro (quindi è possibile e anzi facile passare un valore come parametro per permettere a una funzione di raggiungerlo). Il problema in questo caso è che sia la funzione chiamante che la funzione chiamata devono conoscere quale registro ospita il parametro, si segue quindi una convenzione che in genere viene deciso dal compilatore. Un tremendo limite di questa funzione è che, avendo un numero limitato di registri ad uso generale (6) se si provano a passare troppi parametri sorgono diversi ovvi problemi.

La seconda alternativa prevede di effettuare una PUSH dei parametri in cima allo Stack. Anche qui l’unica difficoltà è avere una convenzione per capire dove si trova esattamente il parametro: è indispensabile compilare main e funzione con lo stesso compilatore, essendo il compilatore a decidere tale convenzione.

Ci sono quindi casi fortunati in cui si può evitare l’uso dello Stack, ma ci sono anche molti altri casi in cui è necessario ricorrere allo Stack e che quindi richiedono un tempo di esecuzione maggiore.

Supponiamo che venga utilizzato il modo di passaggio dei parametri tramite registro: la costante 7 è stata inserita in R0 e il programma chiamato ovviamente non dovrà salvare nello Stack R0. Una ottimizzazione però sarebbe utilizzare lo stesso registro per comunicare il valore di ritorno al main.

Quando chiamiamo la funzione, quindi, è necessario determinare quanti registri utilizzare per lo svolgimento delle istruzioni: nel nostro esempio si può decidere di usare due registri (R1 e R2) e lasciare stare tutti gli altri, all’avvio della funzione quindi abbiamo: il registro 0 occupato dal parametro, il PUSH dei registri 1 e 2 nello Stack e poi lo svolgimento del processo. Al termine si effettuerà il POP delle due celle precedentemente allocate nello STACK e si reinseriscono i valori nei registri R1 e R2, mentre in R0 si troverà il valore che la funzione deve ritornare.

Come fare però per accedere alle variabili locali che sono state allocate nello Stack? Se noi avessimo soltanto le istruzioni di PUSH e di POP non potremmo limitarci a leggere ed eventualmente sovrascrivere valori nello Stack (o meglio dovremmo fare qualcosa di poco efficiente come fare un POP, modificare il valore e rifare un PUSH, senza contare che ciò funzionerebbe solo per l’ultima cella di memoria). Ecco che torna utile il modo di indirizzamento indicizzato. Infatti, si legge il contenuto di una cella della RAM come RAM[SP + Index]. È quindi sufficiente conoscere la distanza di una cella di memoria dalla cima dello Stack per potervi accedere.

In sostanza, alla chiamata di una funzione si fanno i PUSH necessari alle variabili dichiarate (nel nostro esempio 2, prima quello di a e poi quello di b). A quel punto SP punterà alla cella di b. Per poter leggere/sovrascrivere b bisogna quindi accedere a RAM[SP+0], mentre per poter leggere/sovrascrivere a bisogna accedere a RAM[SP+2] (si ricorda che SP autodecrementa, quindi l’indice è >= 0). Quando si modifica il registro SP diventa complicato ritrovare le distanze delle varie celle da esso (specie se si programma direttamente in codice macchina lol), quindi in genere si hanno DUE registri che puntano allo Stack: lo Stack Pointer e il Frame Pointer (anche detto Base Pointer), essi puntano agli estremi dell’area dello Stack che è stata allocata. Lo Stack Pointer può cambiare il proprio valore in seguito a funzioni di tipo PUSH e POP, mentre il Base Pointer no: è quindi più facile utilizzare il Base/Frame Pointer per indicizzare lo Stack piuttosto che lo Stack Pointer.

Per permettere la scrittura di un codice umanamente comprensibile e debuggabile si è quindi deciso di applicare l’indirizzamento indicizzato attraverso il FP.

La modifica del FP avviene a seguito delle istruzioni JSR e RTS (quindi all’avvio e alla fine di una funzione), mentre la modifica dello SP avviene a seguito di istruzioni del tipo PUSH e POP.

In Sostanza: per far partire il main si inizializza il Frame Pointer all’ultima cella di memoria e lo Stack Pointer alla cella 0 -> lo Stack è vuoto perché il FP è oltre lo SP, alla prima allocazione di memoria per la dichiarazione della prima variabile lo SP viene decrementato, salendo sopra il FP e adesso lo Stack ha una cella allocata. Per ogni PUSH successivo lo SP continua a salire e a permettere di allocare memoria. Per accedere alle variabili attraverso l’indirizzamento indicizzato, però, esso si effettua prendendo come riferimento il Base Pointer/Frame Pointer e sottraendovi l’indice necessario (RAM[BP - 0] … RAM[BP - 2] … RAM[BP - 6] …).