Abbiamo parlato di interruzioni e trap. Oggi parleremo di altre funzionalità tipiche di un sistema di calcolo, come ad esempio i meccanismi di Memoria Virtuale (Virtual Memory). A livello fisico abbiamo la struttura classica di una macchina di Von Neumann: abbiamo CPU, sistema di Input/Output e RAM tutti connessi tramite bus. La Cpu si comporta come Master e gli altri dispositivi come Slave. Sappiamo che il processore può effettuare operazioni di lettura e di scrittura in qualsiasi cella di memoria RAM, quindi ogni cella di memoria piò, in momenti diversi, essere usata per qualsiasi scopo (non c’è specializzazione). Questo è ottimo per quanto riguarda l’efficienza della RAM (poiché permette di dedicare a vari programmi una quantità variabile di celle di memoria), ma se guardiamo le cose dal punto di vista di chi deve scrivere il programma da mandare in esecuzione (da parte del processore), questa arbitrarietà diventa un po’ pesante da gestire: sarebbe meglio avere una qualche sorta di strutturazione. Abbiamo visto nel caso del PDP11 che viene consigliata la suddivisione della memoria in almeno 3 parti, e di usare le celle con indirizzo maggiore come Stack, le celle con indirizzo minore come contenitore per il codice e le celle con indirizzo basso/intermedio per contenere i dati statici. Quindi può rendere più semplice scrivere codice avere un qualche ordine nella strutturazione della RAM.

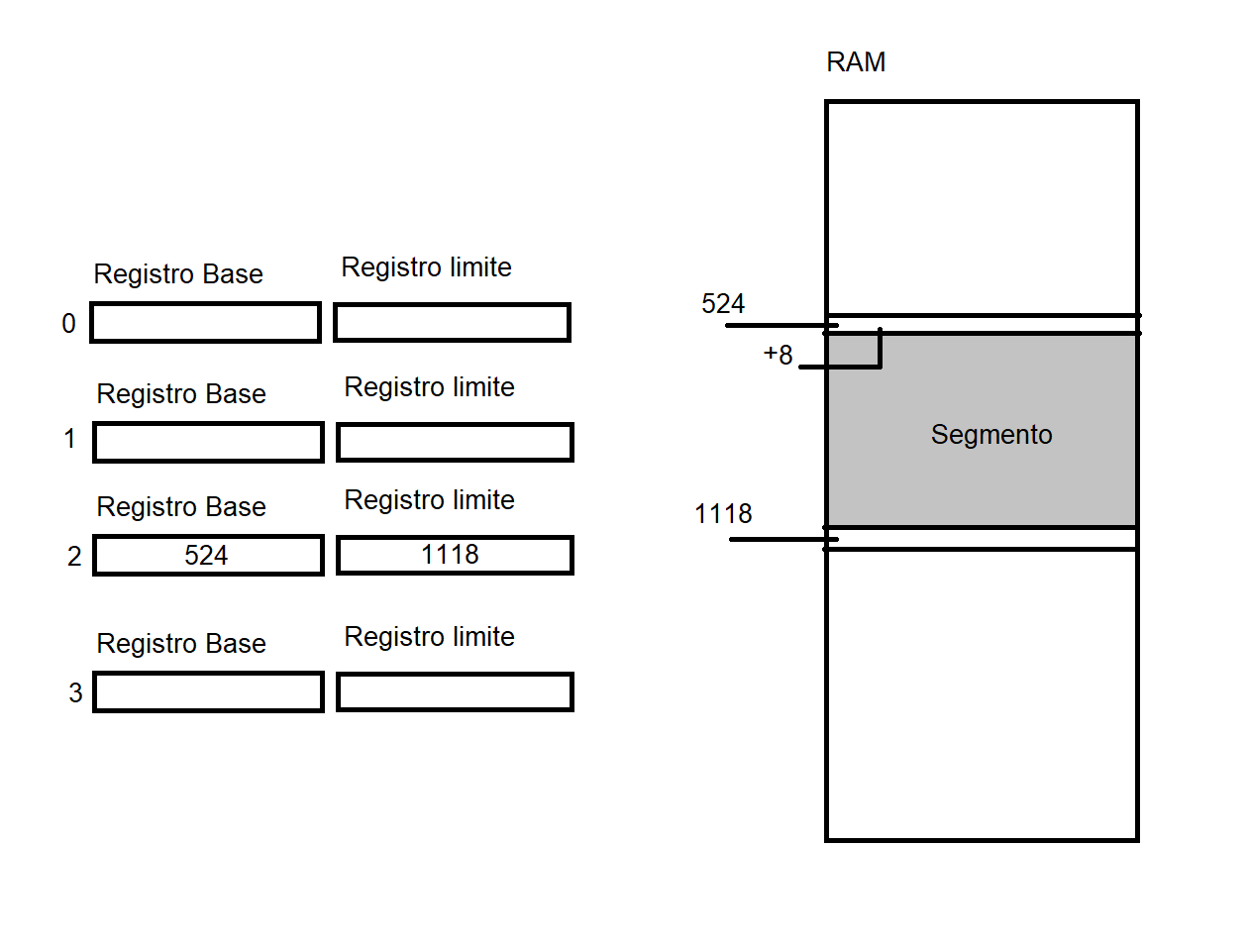
A questo punto entra in gioco il discorso della Memoria Virtuale: essa è un modo per costringere a introdurre una razionalizzazione nell’uso della memoria RAM. Questo avviene tipicamente aggiungendo un dispositivo fisico tra il processore e il bus che realizza la funzionalità della Memoria Virtuale e viene chiamato MMU. Quello che permette di fare è virtualizzare gli indirizzi della memoria RAM: fa si che la realtà che vede il processore sia diversa dalla reale situazione della RAM: il processore vede e invia indirizzi virtuali che vengono tradotti dal MMU in indirizzi fisici (che sono poi quelli veramente utilizzati).

Nel corso del tempo sono state implementate due tecniche di base per la traduzione degli indirizzi: una tecnica è chiamata di Segmentazione (Segmentation), mentre l’altra è chiamata di Impaginazione (Paging).

La Prima è un modo per mettere in ordine la RAM facendo finta che la RAM sia costituita da parti diverse NON comunicanti tra di loro (Quindi il processore non vede una memoria RAM ma ne vede di più, circa dalle 3 alle 8 più o meno). Un Segmento di memoria è quindi una porzione della memoria RAM nella quale vengono inseriti solo un certo tipo di informazioni (e non altri): per esempio, può esistere il segmento Stack (che funziona da Stack), il segmento codice (in cui sono le istruzioni di cui la cpu deve fare il fetch) e il segmento Dati Statici (Static Data). Attraverso la Tecnica di Segmentazione possiamo per esempio forzare il processore a usare quella struttura che abbiamo già visto per il PDP11.

Come avviene tale forzatura? Introducendo dei controlli che verifichino che siano rispettati alcuni vincoli sull’uso di questi segmenti. Per esempio, si potrebbe implementare il segmento del codice in modo che sia soggetto ad accessi in Lettura e Soltanto durante la fase di fetch. Tali controlli sono fatti da dispositivi che lanciano delle interruzioni (del tipo “violazione delle regole”): infatti la Cpu risponde alle interruzioni e il gestore delle interruzioni può gestire la cosa ed eventualmente decidere di interrompere il programma. Questo ci può salvare dal danneggiare la memoria mandando in esecuzione programmi non corretti: se un programma è incompleto o presenta un bug, se non ci fosse il sistema di memoria virtuale allora tale programma potrebbe andare a modificare dati sensibili o provocare danni al sistema.

Se volessimo definire 4 segmenti, un esempio potrebbe essere:



Dove abbiamo un indirizzo che segni l’inizio del segmento e uno che ne segni la fine. Questi due indirizzi sono salvati in due registri (che possiamo chiamare Registro Base e Registro Limite). Se il Processore presenta un registro virtuale contenente il valore 8 tale valore viene sommato al valore contenuto nel Registro Base, per poter ottenere l’indirizzo fisico. Una volta inserito 524 nel registro Base non sarà più possibile generare un numero minore di quello quando si lavora con quel segmento -> Si è privato al processore di accedere a tutti gli indirizzi fisici che vanno da 0 a 523. Un altro controllo che viene fatto prima di effettuare l’accesso è che l’indirizzo virtuale + il valore del Registro Base sia più piccolo del valore contenuto nel Registro limite (nel nostro esempio 524 + 8 < 1118). Se questa condizione non viene rispettata (il risultato di RB + VA è maggiore di RL) viene segnalata una Trap, che come risultato invia sul terminale un errore di Segmentation Fault e viene ucciso/terminato il processo prima che faccia danni. Il controllo è di tipo < o <= a seconda che il valore nel Registro Limite sia considerato come ultima cella disponibile o come confine da non raggiungere (questo è arbitrario). Una variante sarebbe di inserire nel registro limite il valore dell’indirizzo virtuale massimo (quindi nel nostro caso sarebbe 1118 – 524 = 594), in questo caso però bisognerebbe fare il confronto prima dell’operazione di somma.

Si possono avere due tipi di segmentazione: Segmentazione esplicita o implicita.

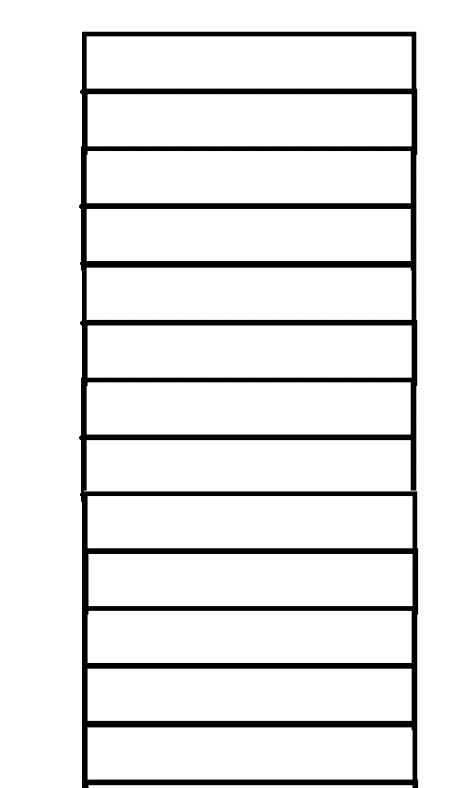
In quella esplicita l’indirizzo virtuale viene diviso in due parti: i primi due (perché in questo caso abbiamo 4 segmenti) bit indicano il numero di segmento (da 0 a 3) e gli altri k-2 bit dell’indirizzo virtuale rappresentano il valore da aggiungere all’indirizzo contenuto nel corrispondente Registro Base. Ciò ci costringe ad utilizzare tanti sommatori quanto il numero di cifre binarie del massimo indirizzo nella nostra RAM e di avere a disposizione 2\*n registri per ogni segmento n che possiamo avere (in questo caso 4); la velocità dovrebbe essere paragonabile a quella di esecuzione di un’istruzione.

L’idea che sta dietro la segmentazione Implicita è quella di non scrivere da nessuna parte il codice del segmento su cui si va ad operare, ma si dà per scontato che si capisca in automatico attraverso il tipo di istruzione che si sta provando ad eseguire: per esempio se si esegue una POP, poiché si fa riferimento allo Stack Pointer si sta chiaramente lavorando sullo Stack, analogamente durante la fase di fetch è ovvio che il processore stia lavorando con una cella della zona della RAM dedicata al codice. Con la Segmentazione Implicita, quindi, permette di risparmiare due bit nell’indirizzo virtuale. Alcuni preferiscono usare la segmentazione esplicita, altri quella implicita: ogni costruttore ha le sue preferenze nella tecnica di segmentazione.

Ci sono alcuni aspetti negativi del sistema di segmentazione: il primo si ha quando si vogliono far girare molti programmi contemporaneamente in uno stesso sistema di calcolo. In questo caso bisognerebbe ripartire la memoria RAM più volte, ciascun programma verrebbe eseguito in maniera indipendente (senza possibilità di interazione), ma a un certo punto la RAM verrebbe occupata totalmente finché uno dei programmi non termina e restituisce la Memoria. Nel momento in cui creo i segmenti, sembrerebbe logico assegnarli in maniera consecutiva (in modo la lasciar liberi blocchi contigui della RAM), ma nel momento in cui questi programmi iniziano a terminare non si riottengono blocchi di memoria libera, ma dei “buchi” che vanno bene solo per programmi che richiedono poca memoria. Se avessimo a disposizione solo la tecnica di segmentazione, in questa situazione ci toccherebbe spostare i segmenti occupati in modo da lasciar liberi blocchi di memoria contigui: tale operazione è facile da implementare (basta modificare i valori del Registro Base e, a seconda della convenzione, del Registro Limite) però dal punto di vista della CPU c’è un grosso lavoro da fare, perché deve copiare i contenuti di tutte le celle di memoria occupate (nei segmenti) in altre celle e questo sicuramente non va bene per grosse fette di memoria. In sostanza, la necessità di Rilocazione è un aspetto che penalizza il sistema di memoria virtuale a segmentazione.

Un altro caso di scarsa efficienza può essere visto nel caso del segmento Stack. Per quanto riguarda lo Stack non si può sapere a priori quanto spazio occuperà, poiché cresce durante l’esecuzione, quindi quale area dedicare a un segmento Stack? Un’opzione potrebbe essere di lasciargli la massima memoria disponibile, ma ciò non permetterebbe di mandare in esecuzione più programmi. Quindi una tecnica potrebbe essere di dargli una dimensione non troppo elevata e fare in modo che quando il processore genererà indirizzi virtuali che stanno fuori dallo Stack subentri una Trap che venga gestita dal sistema operativo. A questo punto il sistema operativo, riconoscendo che si tratta del segmento Stack aggiunge un po’ di memoria al segmento (questo è un caso in cui una trap può essere gestita NON tramite una segnalazione di errore). Però a quel punto bisogna controllare che la memoria libera disponibile sia di indirizzi adiacenti al Segmento Stack, poiché altrimenti è necessaria una Rilocazione.

A questo punto interviene l’altra tecnica che abbiamo citato all’inizio inizio, ossia l’Impaginazione (Paging).



Supponiamo di avere un solo Segmento (l’intera RAM) e di voler trovare un sistema alternativo di traduzione da indirizzi virtuali a indirizzi fisici che permetta di evitare la Rilocazione (e che quindi trovi accettabili la presenza di buchi). Per fare ciò immaginiamo di suddividere la memoria RAM a priori in tanti gruppetti di celle di dimensioni uguali che chiamiamo “Pagine”. Prendiamo quindi 20 bit di indirizzamento e immaginiamo di suddividerli in due parti, una da 14 bit e una da 6. La porzione inferiore ci indica che le pagine sono composte da 2^6 celle, gli altri 14 bit ci indicano il numero della pagina. I bit meno significativi (i 6) gli chiamo Offset, mentre i bit più significativi li chiamo Numero di Pagina.

Se si scrivesse ad esempio 000…011 001101

Si starebbe scrivendo “Offset” = 13 e “Numero di pagina = 3”. Quindi equivarrebbe a livello fisico (se si segue la normale codifica binaria) alla cella n°205 (4° pagina con pagine da 2^6 celle alla 14° cella).

La traduzione da indirizzo virtuale a indirizzo fisico avviene mantenendo uguale la sezione Offset e sostituendo la porzione Numero di Pagine una qualsiasi stringa (arbitraria) di 14 bit. Questo permette di prendere una pagina fisica e spostarne l’indirizzo virtuale da qualsiasi parte. Quindi se le prime 3 pagine sono occupate, la 4° e la 5° sono libere, poi ce ne sono altre 2 occupate, 2 libere, 1 occupata e 3 libere, non importa che siano frammentate, poiché ci basta avere segmenti (si usano sia segmentazione che impaginazione) di dimensioni multiple di una pagina per non dover mai rilocare la memoria (ma piuttosto cambiare la traduzione di un indirizzo da virtuale a fisico).

Per far sì che a una sequenza di 14 bit corrisponda un’altra sequenza arbitraria di 14 bit non si possono usare funzioni (somme/differenze/prodotti ecc.) ma serve necessariamente avere un elenco all’interno dell’MMU che permetta la traduzione dei 14 bit. Serve quindi una tabella che associ il numero di pagina logica (o virtuale) a una pagina fisica (per esempio “alla pagina 0 corrisponde la pagina 27, alla pagina 1 corrisponde la 51, alla pagina 2 corrisponde la 6 ecc.”). Implementare questo non è né facile né difficile, bisogna “solo” trovare il modo di memorizzare da qualche parte la tabella; tale metodo inoltre è più veloce e più lento a seconda di dove la tabella è stata memorizzata. Infatti, tale tabella ha come dimensione minima 2^14 (circa 16.000) celle di 14 bit. C’è inoltre da dire che i 20 bit di indirizzamento che abbiamo supposto sono una cosa ridicola se paragonata ai sistemi di calcolo moderni: quindi se si suddividesse un indirizzamento da 32 bit anche definendo 8 bit di offset avremmo ben 24 bit di numero di pagina, il che implicherebbe che le dimensioni della tabella diventino 2^24 celle = 16.000.000 di celle da 24 bit. In un MMU sarebbe quindi impossibile far stare una memoria statica di quelle dimensioni. Se si facesse una suddivisione diversa (16 bit di Numero di Pagina e 16 bit di Offset) le dimensioni del nostro “dizionario/tabella” si riducono, ma ciò vuol dire che ciascuna delle nostre pagine conterrebbe 2^16 celle di memoria: si tenga conto che quella è la minima quantità di memoria che si potrebbe dedicare (quindi programmi piccoli finirebbero con lo sprecare memoria). Bisogna quindi trovare una soluzione che non esageri da nessuna delle due parti. L’implementazione vera e propria prevede che la tabella di traduzione (Page Table) non sia contenuta nel MMU, ma venga invece inserita nella memoria RAM: ciò porta al vantaggio che, essendo la memoria RAM già grande di suo, non dovrebbe essere un grave problema fare ciò, ma ha lo svantaggio che la RAM non è realizzata con tecnologia statica, ma è una memoria dinamica, il che rallenta incredibilmente la velocità di recupero delle informazioni dalla tabella per poter eseguire le istruzioni. C’è da dire che le prestazioni vengono però migliorate notevolmente attraverso un meccanismo di “Caching” che permette di conservare temporaneamente alcuni dati (resta però da tenere conto di non superare un certo limite per le dimensioni della Tabella).

Quindi una porzione della RAM verrà utilizzata per contenere la Tabella di Impaginazione e solo le celle rimanenti verranno suddivise in pagine.

Riassumendo:

Dato un indirizzo virtuale da 32 bit lo posso suddividere in 2 o 3 insiemi di bit (a seconda che si voglia usare la segmentazione esplicita o implicita). Immaginiamo di voler usare la segmentazione esplicita. I primi 2 bit sono i bit di segmento, poi ci sono 20 bit di Numero di Pagina Virtuale e poi 10 bit di Offset (quindi ciascuna pagina è grande 2^10 celle di memoria).

L’indirizzo fisico avrà come 10 bit meno significativi i bit di Offset e n bit più significativi di pagina fisica. Il fatto di aver usato 20 bit di pagina virtuale non implica che ne dobbiamo avere 20 anche per la pagina fisica (potrebbero essere di più, in generale, o di meno). Se il numero totale di bit di indirizzamento fisico è uguale al numero di bit di indirizzamento logico abbiamo 22 bit dedicati al Numero di Pagina fisico.   
La nostra RAM da 2^32 bit verrà quindi suddivisa in 2^22 pagine da 2^10 celle ciascuna. Non tutte queste pagine saranno però pagine vere e proprie/accessibili, poiché una zona della RAM è dedicata a tenere memorizzate la tabella di traduzione e la tabella dei segmenti. La tabella di segmentazione conterrà in questo caso 4 elementi, ognuno dei quali indicherà quante e quali sono le pagine dedicate a quel determinato segmento (ciascun segmento può quindi contenere un valore compreso tra 0 2 2^22-un po’). Una parte dell’elemento verrà chiamata “dimensione del segmento” che indicherà il numero di pagine che costituiscono quel segmento (essendo esso un multiplo di una pagina) e un’altra equivarrà a un indirizzo che punta alle pagine di quel tipo. Se nell’elemento S0 è indicata come dimensione 15, la tabella corrispondente a quel segmento sarà un vettore che conterrà 15 valori espressi su 22 bit (il numero di bit destinati al numero di pagina fisica).

Riprendendo l’esempio di prima, la CPU prende i 2 bit di Segmento e prende il corrispondente elemento nella tabella dei segmenti, prenderà le dimensioni del segmento (in pagine) e poi userà l’indirizzo contenuto nella seconda parte dei 32 bit iniziali per cercare la corrispondente pagina tra quelle dedicate a quel particolare tipo di segmento (se fosse maggiore si invierebbe una trap, poiché sarebbe un valore out of bounds). Infine, il valore di Offset sarebbe riportato così com’è, e quindi si accederebbe alla cella Offset della pagina presa.