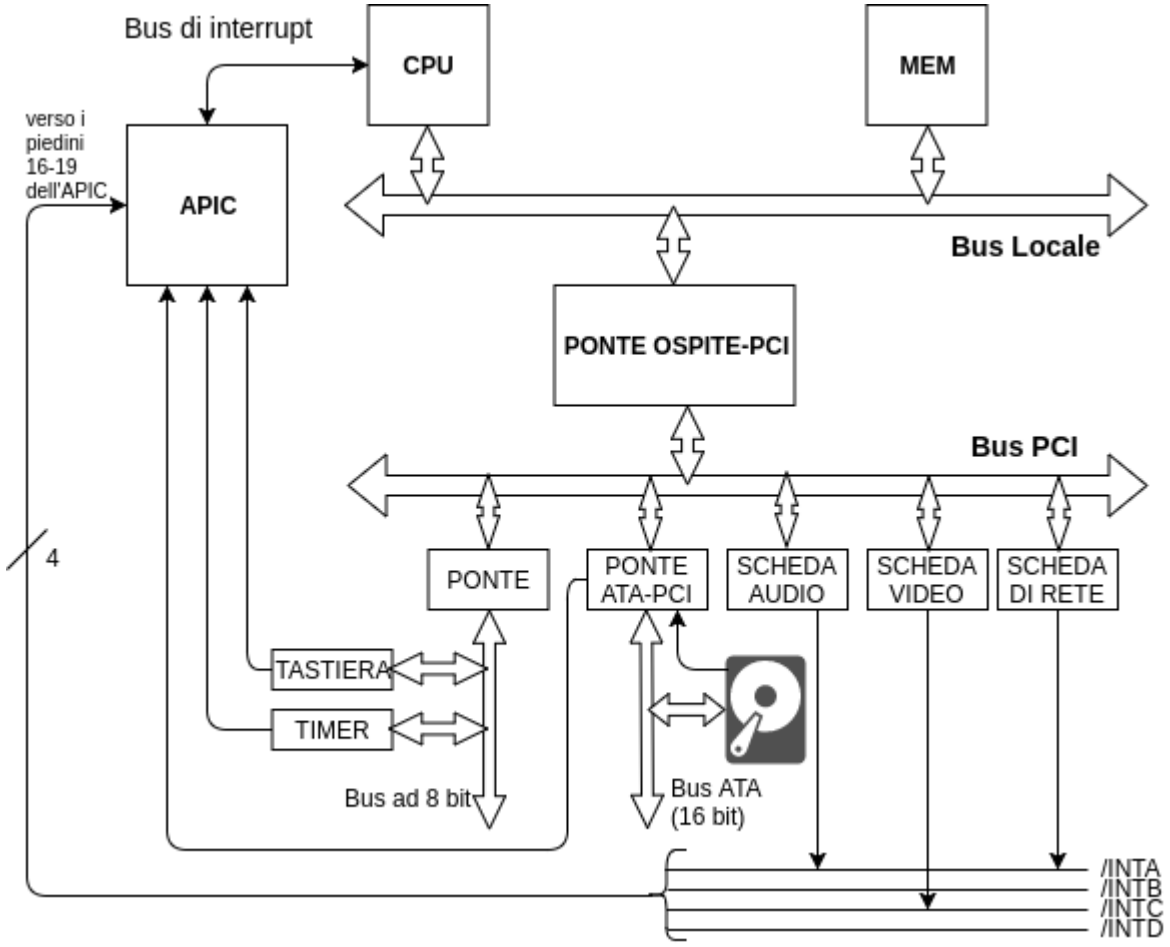
SCHEMI CONCETTUALI DI CALCOLATORI ELETTRONICI (programma a 64 bit)



Autore : Luigi Gjoni

Prefazione

Questa raccolta di schemi è stata ottenuta dopo aver seguito le lezioni di calcolatori elettronici a 64 bit tenute dai professori Giuseppe Lettieri e Graziano Frosini.
Questo lavoro da me svolto può aiutare in un ripasso veloce della materia “CALCOLATORI ELETTRONICI” in vista della sessione d’esame. Il mio consiglio è di studiare prima il materiale fornito dai professori e poi venire quì per poter riassumere i concetti in maniera veloce, in modo da avere una visione d’insieme quanto più ampia possibile. Si ricordi inoltre che nessun materiale scritto da studenti potrà mai compararsi con le lezioni frontali del professore che sarà sempre disponibile a rispondere ad eventuali domande da nostra parte.
Auguro a tutti buon studio ed un gran “In bocca al lupo” !

Luigi Gjoni

Ringraziamenti

Ringrazio di cuore chi ha arricchito il mio arsenale di nozioni per questa materia e ringrazio anche mi ha aiutato a correggere/rivedere/ampliare questi schemi : il prof. Giuseppe Lettieri che ha avuto tanta pazienza nel rivedere tutto il materiale in tempi davvero stretti ed i miei colleghi Marilisa L., Paolo T., giomba, Davide C., Fabio L., Giovanni M., Silva, Matilde M., Mattia C. e Matteo B. che mi hanno spronato a fare di meglio e mi hanno dato un aiuto fondamentale per rivedere questi schemi. Se non fosse per VOI questi schemi non sarebbero mai usciti così ricchi di informazioni.

Ovviamente se tu dovessi notare qualche svista me lo puoi notificare su telegram (@xhigjo). Come gesto di gratitudine, posso includerti nei ringraziamenti :)

CALCOLATORI ELETTRONICI

HW
-Prelievo/Esecuzione istruzioni
- Programmatore(sistema) non ha alcun controllo su ciò che viene fatto in HW

SW
-Una cosa è fatta in sw se esiste un programma che fa quella cosa
-Qualcosa non fattibile con un'istruzione elementare

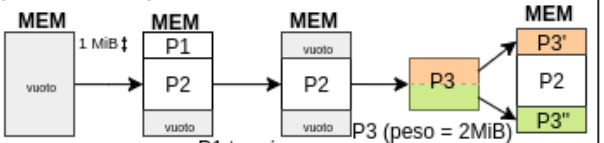
HW vs SW

PROTEZIONE
"Mai fidarsi dell'utente" cit. prof. Lettieri.
Il sistema deve offrire un metodo sicuro tramite il quale l'utente possa fare le sue cose senza causare danni al NUCLEO.

ARGOMENTI PRINCIPALI

INTERRUZIONE
Segnale inviato alla CPU, emesso via HW o SW, che indica il bisogno di attenzione per il soddisfacimento di una data richiesta di servizio.

MEMORIA VIRTUALE
Meccanismo che mappa la memoria virtuale di un processo in memoria fisica. Risolve 3 problemi:
1) RAM non basta mai.
2) Processi P1 e P2 scrivono entrambi all'indirizzo x. Si corromperanno i dati a vicenda?
3) "Buchi" nello spazio di indirizzamento.

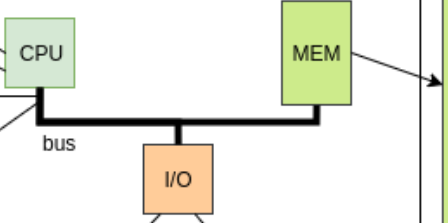


P1 termina
P3 (peso = 2MiB) deve essere eseguito. Abbiamo 2MiB liberi NON CONTIGUI.

COSA FA LA CPU
1) Prelievo istruzione puntata da RIP.
2) Decodifica
3) Esegue
4) Vede se ci sono richieste di interruzioni (e, nel caso, le gestisce)
5) Aggiorna RIP e ritorna al punto 1

NOTE CPU
- CPU non smette mai di eseguire istruzioni a meno che non abbia eseguito HLT
- CPU non è in grado di eseguire istruzioni al di fuori del suo set predefinito
- CPU vede solo il suo stato corrente. Non può ricordare stati passati nè prevedere (con certezza) stati futuri.

SCHEMA GENERALE DI UN CALCOLATORE



PROPRIETÀ MEM
- Sistema organizzato in celle.
- Ogni cella ha un indirizzo che la identifica univocamente.
- Ogni cella contiene una sequenza di bit.
- Il contenuto di una cella può assumere vari significati a seconda dell'uso che se ne fa.
- Sa solo eseguire operazioni di R/W.
- È consentita 1 operazione/volta.
- R : indirizzoPrimaCella + n° Byte da leggere.
- W : indirizzoPrimaCella + n° Byte da scrivere + nuovo contenuto della/e cella/e.
- È passiva, cioè il suo stato non cambia (a meno che la MEM non sia coinvolta in un'operazione di W).
- Ogni op. richiede un tempo pressochè costante.

COSA FA IL BUS
- Connette fra loro i dispositivi
- Comunicazione tra i dispositivi solo tramite R/W

COMUNICAZIONE NEL BUS
L'indirizzo ci deve far capire se stiamo operando con la MEM o l'I/O

2 MODI

Riservo indirizzi diversi per MEM ed I/O

Definisco spazi di indirizzamento separati

Per ogni operazione R/W specifico con un bit se opero su MEM o I/O

REGISTRI
Operazione di R/W su di un registro **modifica lo stato** della periferica

INTERFACCIA
Utile per la comunicazione con la CPU e periferica

REGISTRI GENERALI

63 0

RAX
...
RDX
RDI
RSI
RBP
RSP
R8
...
R15

64	32	16	8 (sup)	8 (inf)
RAX R8	EAX R8D	AX R8W	AH -	AL R8B

REGISTRI DI STATO

63 0

RIP	Contiene puntatore alla prossima istuzione
RFLAGS	Contiene lo stato corrente della CPU

CPU

FORMATO ISTRUZIONI

$OPCODE_{suff}$ firstOp, secondOp

dove **suff** ϵ {**b**, **w**, **l**. **q** }
1, 2, 4, 8 Byte

MODELLO GRANDE

- *.text* e *.data* **memorizzabili ovunque**.
- istruz. *operative* devono usare **indirizzam. canonico con registro base** e l'istruz. **MOVABSQ $x** (dove x è un operando immediato a 64 bit) per caricare il registro base
- istruzioni *di salto non condizionato* usano **l'indirizzam. indiretto**

MODELLO PICCOLO

- .text e .data **devono** essere memorizzate nei primi 2 GiB di memoria
- istruz. *operative* possono usare **indirizzam. canonico senza registro base.**
- istruz. *di salto (condizionato e non)* possono usare tutte **l'indirizzam. relativo**

MODELLO PIC (Position Independent Code)

- .text e .data memorizzate in una zona di 2GiB di memoria, posizionata ovunque
- istruz. operative devono usare **l'indirizzam. relativo** oppure **l'indirizzam. canonico con reg. base.** Per *caricare il reg. base* devono usare **LEAQ con indirizzam. relativo**.
- istruzioni di salto (condizionato e non) possono usare tutte l'indirizzam. relativo.

MODELLI MEMORIA

MODALITÀ DI INDIRIZZAMENTO

- **REGISTRO**
  addl %EBX, %EAX
- **IMMEDIATO**
  addl $2, %EAX
  costante su 32 bit
- **MEMORIA**
  Operando contenuto in MEM.
  L'istruzione deve specificarne l'indirizzo

MEM

SPIAZZAMENTO (MEMORIA)

displacement (base, index, scale factor)

costante 32 bit | registri CPU (64 bit) | ϵ {1,2,4,8}

dove l'indirizzo viene calcolato come

*address = (base + index * scaleFactor) + displacement*

RELATIVO A RIP (MEMORIA)

displacement (%RIP)
costante 32 bit
address = %RIP + displacement

SPAZIO DI INDIRIZZAMENTO

FISICO

63 51 0

VIRTUALE

63 47 0

SUDDIVISIONE MEMORIA

0000 0000 0000 0000
0000 7FFF FFFF FFFF
FFFF 8000 0000 0000
FFFF FFFF FFFF FFFF
indirizzi in forma canonica

ARCHITETTURA AMD 64

COMANDI LINUX UTILI

- **objdump -d :** mostra le info dei file oggetto.
- **gdb** : GNU debugger.
  1) b nomeEtichetta / rigaFile : blocca l'esecuzione del programma in quel punto specifico del programma.
  2) *info registers registro* : mostra il contenuto di tale registro

NB Il formato '*comando -opzione*' indica che l'opzione (ovviamente) può essere omessa. Per avere più informazioni su come si usano determinati comandi digitare dal terminale "*man comandoDaStudiare*" per leggere le pagine del manuale in merito. Se invece si ha idea di cosa si vuol fare ma non si sa se c'è un comando che aiuta a farlo, digitare "*apropos parolaChiave*". Ad esempio voglio vedere se c'è un comando che mi unisce più file pdf. Digiterei "*apropos pdf*" e leggerei nome del comando (che è spesso autoesplicativo) e relativa descrizione per essere sicuro di aver scelto il comando giusto.

I/O

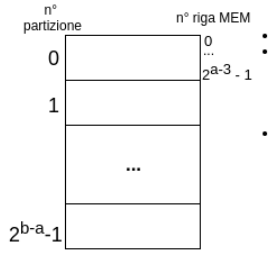
SPAZIO I/O

-È ampio
2^{16} *Byte = 64 KiB*

STATO DEL CALCOLATORE

STATO REG CPU + STATO MEM + STATO PERIF. I/O

SCOMPOSIZIONE INDIRIZZI



- **b** : 2^b = n° Byte MEM
- **a** : lo scegliamo noi, in modo da poter dimensionare a piacere le singole "partizioni" di memoria principale di dimensione pari a 2^a Byte
- Usiamo una maschera per ogni partizione di memoria principale. In tal modo, quando una partizione viene scelta per un'operazione di R/W, le altre partiz. vengono ignorate.

ORDINE DEI BYTE

Supponiamo di dover memorizzare il valore 0x0A0B0C0D

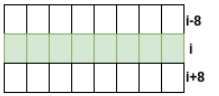
BIG ENDIAN		LITTLE ENDIAN (usata da noi)	
0A	X	X	0D
0B	X+1	X+1	0C
0C	X+2	X+2	0B
0D	X+3	X+3	0A

ALLINEAMENTO

file.cpp
...
long a; // 8 Byte
...

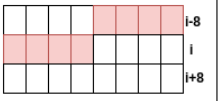
Domanda : come memorizzo 'a' ? Ho due modi.

Memorizzato su una linea



Per leggere 'a' devo fare una sola lettura da 8 Byte => l'allineamento naturale mi va bene.

Memorizzato su più linee



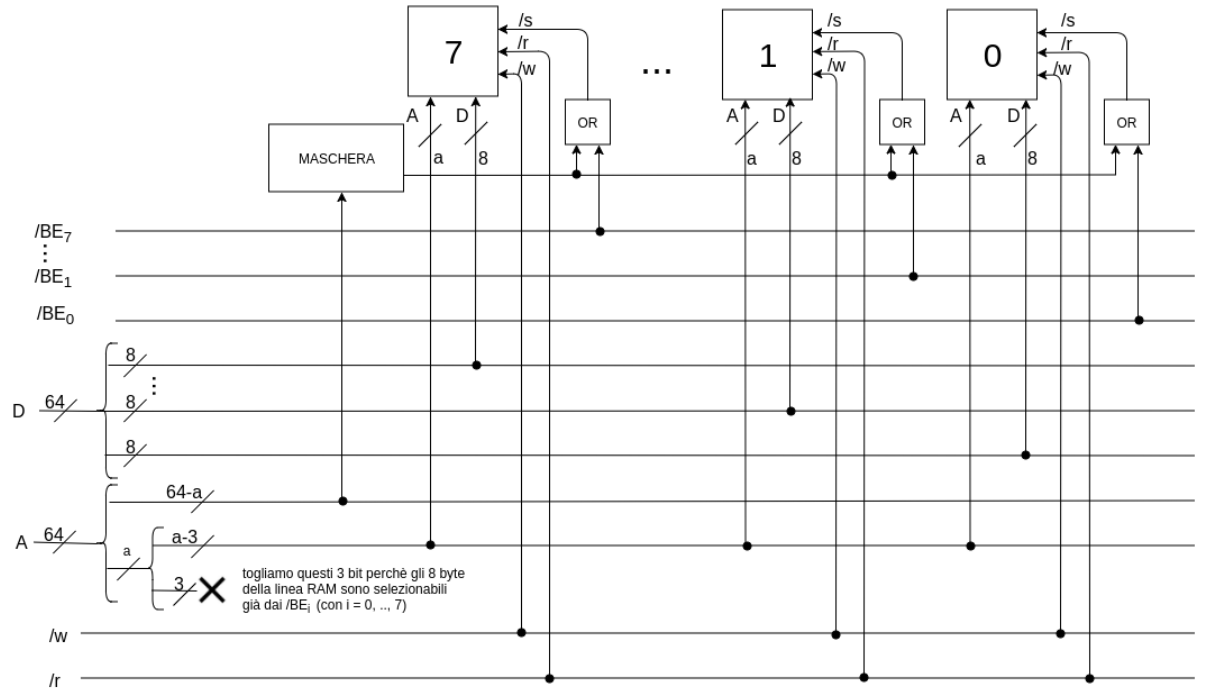
Per leggere 'a' devo compiere due letture da 4 byte ed in più dovrei ordinare i gruppi da 4 Byte letti

MEMORIA PRINCIPALE

- Celle da 1 Byte
- Ogni cella ha un indirizzo € [0, 2^{64} - 1]
- Sono ammesse solo operaz. di R/W
- È organizzata in linee da 8 Byte

numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	indirizzo LSB
0									0
1									8
...									...
2^{61}-1									2^{64}-8

PARTIZIONAMENTO MEMORIA PRINCIPALE



A cosa servono quegli 'a' fili?
'a' stabilisce la dimensione (2^a Byte) della partizione di MEM

WRITE BACK
Per ogni operazione di W, la cache viene aggiornata ma non la MEM. Quest'ultima verrà aggiornata SOLO quando la linea di cache modificata verrà sacrificata per far spazio ad altro

WRITE THROUGH
Per ogni operazione di W, CACHE e MEM sono entrambe aggiornate

BIT UTILI
V: linea di cache è significativa <=> V=1.
D : linea di cache è stata modificata <=> D=1 (usato nella politica Write Back per vedere se è necessario riscrivere il dato in RAM)

POLITICHE AGGIORNAMENTO MEM

CONTROLLORE CACHE

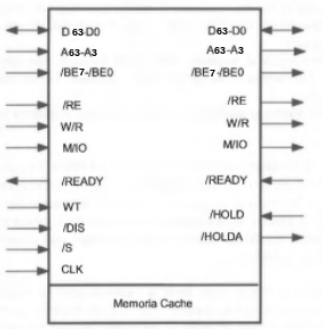
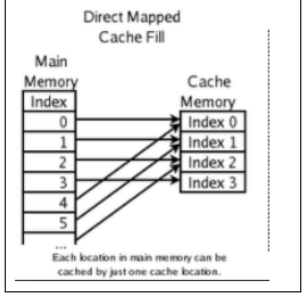


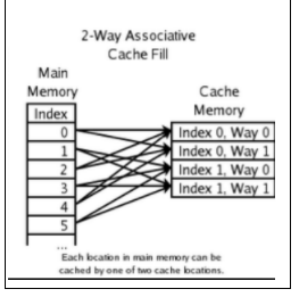
Figura 2.15 Schema di un controllore cache

ASSOCIATIVITÀ

MAPPATURA DIRETTA
1 Modulo



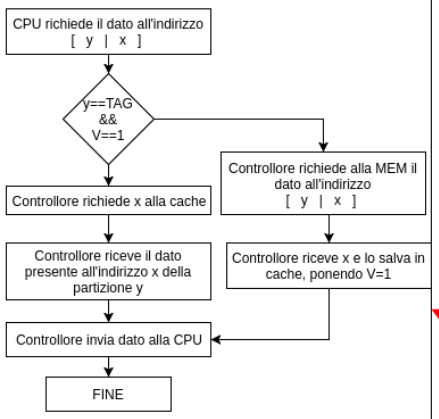
PARZ. ASSOCIATIVA
k moduli con k €]1,2^{b-a}[



TOT. ASSOCIATIVA
2^{b-a} moduli

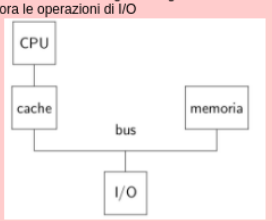
Come quella parzialmente associativa, con la differenza che il dato può risiedere ovunque in cache.

HIT / MISS

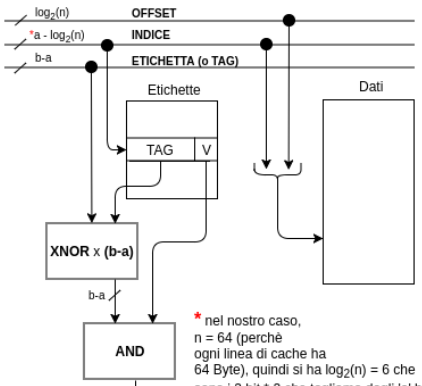


MEMORIA CACHE

- Memoria piccola, veloce e costosa.
- serve per accelerare gli scambi di informazioni tra CPU e MEM
- Ha un controllore che gestisce gli scambi con l'esterno
- Ignora le operazioni di I/O



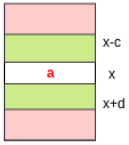
CIRCUITO HIT / MISS



* nel nostro caso, n = 64 (perchè ogni linea di cache ha 64 Byte), quindi si ha $\log_2(n) = 6$ che sono i 3 bit * 2 che togliamo dagli **'a'** bit che servirebbero per indirizzare ogni Byte della partizione di RAM.

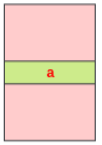
PRINCIPI DI LOCALITÀ

SPAZIALE



Il dato all'indirizzo x è richiesto. Allora, tutti i dati € [x-c, x+d], con c non necessariamente uguale a d, potrebbero essere richiesti con una data probabilità

TEMPORALE

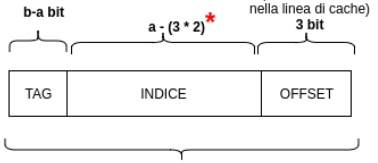


Il dato all'indirizzo x è stato richiesto. Allora il dato all'**indirizzo x** potrebbe essere richiesto in seguito

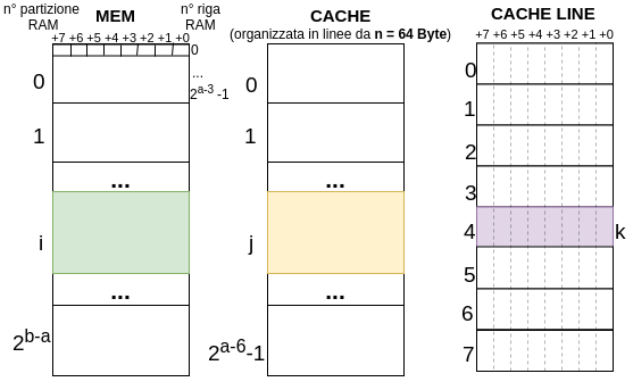
LEGENDA
- indirizzi richiedibili in base al principio di località esaminato
- indirizzo richiesto l'ultima volta
- indirizzi ignorati dal principio di località esaminato

SCOMPOSIZIONE INDIRIZZIO

- Cache con capacità pari a 2^a **Byte**.
- Linea di cache è di **n Byte, n € {16, 32, 64}.** Per noi, **n = 64**
- Suddividiamo la RAM in partizioni della dimensione della cache, cioè **partizioni di RAM da 2^a Byte**.
- Suddividiamo l'indirizzo come segue



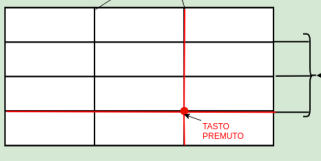
Supponiamo di avere un indirizzo (61 bit più significativi) scomposto come segue



* togliamo 3 bit * 2 perchè
- primi 3 bit meno significativi : sono i 3 bit che ci danno il byte nella linea di RAM (che sono già specificati dai $/BE_i$, i = 0,..,7).
- secondi 3 bit meno significativi : sono i 3 bit di offset della cache, ovvero la linea di RAM selezionata all'interno della linea di cache (selezionata coi bit di **OFFSET**).

In poche parole, gli **'a-6' bit servirebbero per indirizzare linee di cache**. Prendendola in senso generale, una linea cache da n byte (con n ovviamente potenza di 2) dovrebbe usare $\log_2(n)$ bit per indirizzare i singoli byte della linea di cache. Questi primi **$\log_2(n)$ bit vanno tolti dagli 'a' bit per avere l'indirizzo della linea di cache**. Nel nostro caso particolare, la linea contiene 64 Byte quindi 64 Byte = 2^6 Byte quindi dobbiamo togliere proprio 6 bit da 'a' per ricavare l'indirizzo della linea di cache.

TASTIERA



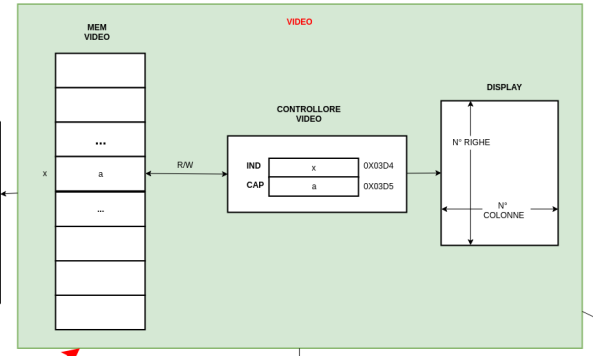
- Possiamo immaginare la tastiera come una matrice di tasti. La matrice dell'esempio ha 3x2 tasti.
- Ogni volta che si **preme** un tasto, la tastiera invia un **make code.** Se si tiene premuto il tasto, la tastiera continua ad inviare make code con una certa frequenza.
- Ogni volta che si **rilascia** un tasto premuto, la tastiera invia un **break code**.
- BreakCode = MakeCode | 0x80 (in altre parole il break code non è altro che il make code con il bit più significativo posto ad 1).

// esempio tastiera-1 : tastcode.cpp

```
#include <libce.h>

const ioaddr iSTR = 0x64;
const ioaddr iRBR = 0x60;

natb get code()
{
    natb c;
    do
        inputb(iSTR, c);
    while (!(c & 0x01));
    inputb(iRBR, c);
    return c;
}

int main()
{
    natb c;
    for (;;) {
        c = get_code();
        if (c == 0x01)
            break;  // make code di ESC
        for (int i = 0; i < 8; i++) {
            if (c & 0x80)   // bit piu' significativo di c
                char write('1');
            else
                char write('0');
            c <<= 1;
        }
        char write('\n');
    }
}
```

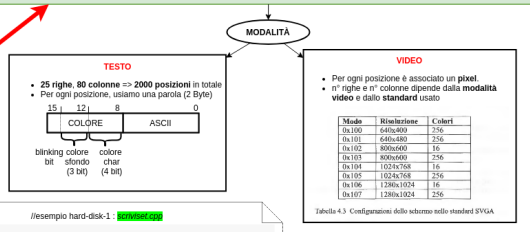
// video-testo-1 : videoM1.cpp

```
#include <libce.h>

natw* video = reinterpret_cast<natw*>(0x000B8000);
// array di 2000 word;

int main()
{   for(int i=0; i<2000; i++)
        video[i] = 0x4B00 | 'a';
        // sfondo rosso (0100), simbolo azzurro chiaro (1011)

    char c;
    for (;;)
    {   c = char_read();
        if (c == 0x1B) break;
        // carattere ASCII esc
    }
    return 0;
}
```



// video-testo-2 : videobis.cpp

```
// file videobid.cpp
#include "libce.h"

const ioaddr IND = 0x03D4;
const ioaddr DAT = 0x03D5;
const natl COLS = 80;
const natl ROWS = 25;
const natl VIDEO_SIZE = COLS*ROWS;
const natb CUR_HIGH = 0x0e;
const natb CUR_LOW = 0x0f;


natw attr;              // attributo colore
natb x, y;              // coordinate x e y inizialmente a 0
natw* video = reinterpret_cast<natw*>(0x000B8000);
    // array di 80*25 = 2000 posizioni, ciascuna di una parola

void cursore()
{   natw pos = COLS * y + x;
    outputb(CUR_HIGH, IND);
    outputb(pos >> 0x8, DAT);
    outputb(CUR_LOW, IND);
    outputb(pos, DAT);
}

void clear_screen(natb col)
{   attr = static_cast<natw>(col) << 8;
    for (int i = 0; i < VIDEO_SIZE; i++)
        video[i] = attr | ' ';
    x = 0; y = 0; cursore();
}

void scroll()
{   for (int i = 0; i < VIDEO_SIZE - COLS; i++)
        video[i] = video[i + COLS];
    for (int i = 0; i < COLS; i++)
        video[VIDEO_SIZE - COLS + i] = attr | ' ';
    y--;
}


void char_write(natb c)
{   switch (c)
    {   case 8:
            break;
        case '\n': case '\r':
            x = 0;              // azione associate a \r
            y++; if (y >= ROWS) scroll();       // azione associate a \n
            break;
        default:
            video[y * COLS + x] = attr | c;
            x++;
            if (x >= COLS) {   x = 0; y++;   }
            if (y >= ROWS) scroll();
            break;
    }
    cursore();
}

int main()
{   natb c;
    clear_screen(0x48);
    for (;;)
    {   c = char_read();
        if (c == 0x1B) break;               // carattere ASCII esc
        char_write(c);
    }
}
```

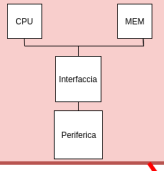


// esempio tastiera-2 : tastcar.cpp

```
#include "libce.h"

const natl CODICI = 29;
bool shift = false;
natb tab[CODICI] =   // tasti lettere (26), spazio, enter, esc
    { 0x39, 0x1E, 0x30, 0x20, 0x12, 0x21, 0x22, 0x23, 0x17, 0x24, 0x25, 0x26,
      0x32, 0x31, 0x18, 0x19, 0x10, 0x13, 0x1F, 0x14, 0x16, 0x2F, 0x11, 0x2D,
      0x15, 0x2C, 0x1C, 0x01 };
natb tabmin[CODICI] =
    { ' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
      'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
      'z', '\n', 0x1B };
natb tabmai[CODICI] =
    { ' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
      'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
      'Z', '\n', 0x1B };

char conv(natb c)
{
    natb cc;
    natl pos = 0;

    while (pos < CODICI && tab[pos] != c)
        pos++;
    if (pos == CODICI)
        return 0;
    if (shift)
        cc = tabmai[pos];
    else
        cc = tabmin[pos];
    return cc;
}

char char_read()
{
    natb c;
    char a;

    do {
        c = get_code();
        if (c == 0x2A) // left shift make code
            shift = true;
        else if (c == 0xAA) // left shift break code
            shift = false;
    } while (c >= 0x80 || c == 0x2A); // make code;
    a = conv(c);    // conv() puo' restituire 0
    return a;   // 0 se tasto non riconosciuto
}

int main()
{
    char c;

    for (;;) {
        c = char_read(); // puo' restituire 0
        if (c == 0x1B) // carattere ASCII esc
            break;
        char_write(c);  // non effettua azioni se c vale 0
    }
    return 0;
}
```

PERIFERICHE I/O

I dispositivi connessi ai bus locale, sono connessi per mezzo di interfacce in modo da comunicare con la CPU



//esempio hard-disk-1 : scriviset.cpp

```
#include <libce.h>

const ioaddr iBR = 0x01F0;
const ioaddr iCNL = 0x01F4;
const ioaddr iCNH = 0x01F5;
const ioaddr iSNR = 0x01F3;
const ioaddr iHND = 0x01F6;
const ioaddr iSCR = 0x01F2;
const ioaddr iERR = 0x01F1;
const ioaddr iCMD = 0x01F7;
const ioaddr iSTS = 0x01F7;
const ioaddr iDCR = 0x03F6;

void scrivisett(natl lba, natb quanti, natw vetti[])
{
    natb lba_0 = lba,
         lba_1 = lba >> 8,
         lba_2 = lba >> 16,
         lba_3 = lba >> 24;
    natb s;
    do
        inputb(iSTS, s);
    while (s & 0x80);

    outputb(lba_0, iSNR);                   // indirizzo del settore e selezione drive
    outputb(lba_1, iCNL);
    outputb(lba_2, iCNH);
    natb hnd = (lba_3 & 0x0F) | 0xE0;
    outputb(hnd, iHND);
    outputb(quanti, iSCR);                  // numero di settori
    outputb(0x0A, iDCR);                    // disabilitazione richieste di interruzione
    outputb(0x20, iCMD);                    // comando di scrittura
    for (int i = 0; i < quanti; i++) {
        do
            inputb(iSTS, s);
        while (!(s & 0x08) != 0x08);
        for (int j = 0; j < 512/2; j++)
            outputw(vetti[i*512/2 + j], iBR);
    }
}

natb buff[8*512];
int main()
{
    natl lba = 1;
    natb quanti = 2;                        // massimo 8
    for (int i = 0; i < quanti*512; i++)
        buff[i] = 'c';
    scrivisett(lba, quanti, reinterpret_cast<natw*>(buff));
    str_write("OK\n");
    pause();
    // ...
    return 0;
}
```

//timer : timeb.cpp

```
#include <libce.h>

const ioaddr iSPR = 0x61;
const ioaddr iCWR = 0x43;
const ioaddr iCTR2_LSB = 0x42;
const ioaddr iCTR2_MSB = 0x42;

void ini_cont()
{   natw N = 1190;
    // costante per 1000 Hz

    outputb(0x86, iCWR);
    // contatore 2, modo 3
    outputb(N, iCTR2_LSB);
    outputb(N >> 8,  iCTR2_MSB);
}
int main()
{   ini_cont();
    outputb(3, iSPR);
    pause();
    outputb(0, iSPR);
    return 0;
}
```

INTERFACCIA DI CONTEGGIO 8254

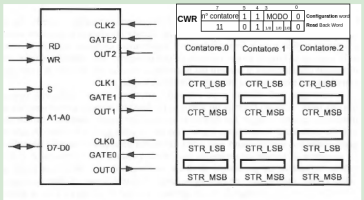


Figura 5.1 Schema dell'interfaccia 8254

- **CWR**(*Control Word Register*, 8 bit) : permette di comandare il contatore in due modi (N.B. Nello schema dell'interfaccia, sono state inserite le due configurazioni per un ripasso più veloce)

CONFIGURAZIONE
1) Selezionare un contatore
2) Stabilire il **modo** di funzionamento del contatore selezionato

LETTURA
Fa memorizzare, in uno o più registri di stato, il valore dei registri di conteggio

bit i-esimo (i = 1, 2, 3) di CWD: se vale 1, il contatore 'i-1'-esimo andrà a memorizzare nel proprio registro STR il valore attuale del registro interno di conteggio.

- **CTR** (*Constant Time Register*, 16 bit) : costante di tempo di un timer.
- **STR** (*STatus Register*, 16 bit) : serve per conoscere il valore attuale del registro di conteggio.

HDD

LIMITAZIONI HDD
- Lentezza

- Non esiste alcun collegamento diretto tra CPU e HDD
- Unità di trasferimento dell'HDD è il **settore** = 512 Byte.
- L'HDD è composto da vari **piatti**. In ciascuno di questi, possono essere depositate informazioni sulle 2 superfici del piatto => per leggere/modificare le informazioni contenute sui vari piatti dell'HDD, dobbiamo usare 2 testine/piatto.
- Per leggere / scrivere un settore, tutte le testine (actuator arms) si spostano sulla traccia corrispondente.
- **Come si indirizza un settore?** Servono **n° testina** (per identificare la superficie coinvolta), **traccia (o n° cilindro)** sulla superficie, e **n° settore** sulla traccia.
- L'indirizzo di un settore lo identifichiamo come **LBA**(Logical Block Address, 28 bit)

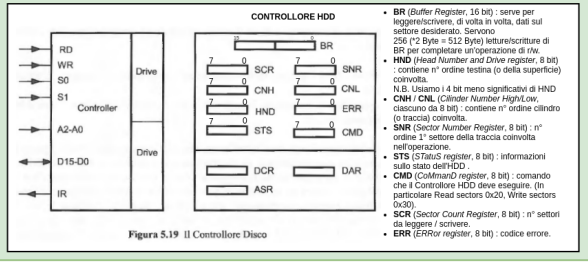
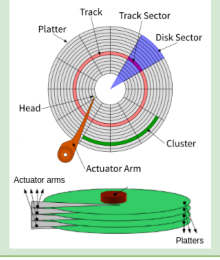


Figura 5.19 Il Controllore Disco

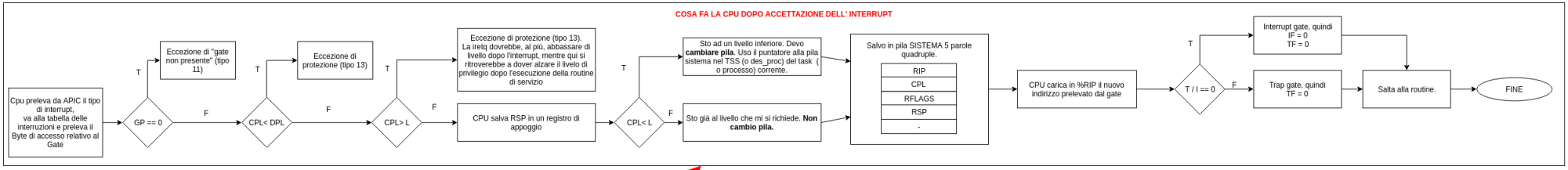
- **BR** (*Buffer Register*, 16 bit) : serve per leggere/scrivere, di volta in volta, dati sul settore desiderato. Servono 256 (*2 Byte = 512 Byte) letture/scritture di BR per completare un'operazione di r/w.
- **HND** (*Head Number and Drive register*, 8 bit) : contiene n° ordine testina (o della superficie) coinvolta.
  N.B. Usiamo i 4 bit meno significativi di HND
- **CNH / CNL** (*Cilinder Number High/Low*, ciascuno da 8 bit) : contiene n° ordine cilindro (o traccia) coinvolta.
- **SNR** (*Sector Number Register*, 8 bit) : n° ordine 1° settore della traccia coinvolta nell'operazione.
- **STS** (*STatuS register*, 8 bit) : informazioni sullo stato dell'HDD.
- **CMD** (*CoMmanD register*, 8 bit) : comando che il Controllore HDD deve eseguire. (In particolare Read sectors 0x20, Write sectors 0x30).
- **SCR** (*Sector Count Register*, 8 bit) : n° settori da leggere / scrivere.
- **ERR** (*ERRor register*, 8 bit) : codice errore.



//hard-disk-2 : leggiset.cpp

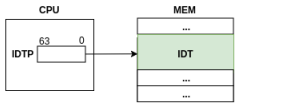
```
#include <libce.h>

const ioaddr iBR = 0x01F0;
const ioaddr iCNL = 0x01F4;
const ioaddr iCNH = 0x01F5;
const ioaddr iSNR = 0x01F3;
const ioaddr iHND = 0x01F6;
const ioaddr iSCR = 0x01F2;
const ioaddr iERR = 0x01F1;
const ioaddr iCMD = 0x01F7;
const ioaddr iSTS = 0x01F7;
const ioaddr iDCR = 0x03F6;

void leggisett(natl lba, natb quanti, natw vetti[])
{
    natb lba_0 = lba,
         lba_1 = lba >> 8,
         lba_2 = lba >> 16,
         lba_3 = lba >> 24;
    natb s;
    do
        inputb(iSTS, s);
    while (s & 0x80);

    outputb(lba_0, iSNR);           // indirizzo del settore e selezione drive
    outputb(lba_1, iCNL);
    outputb(lba_2, iCNH);
    natb hnd = (lba_3 & 0x0F) | 0xE0;
    outputb(hnd, iHND);
    outputb(quanti, iSCR);          // numero di settori
    outputb(0x0A, iDCR);            // disabilitazione richieste di interruzione
    outputb(0x20, iCMD);            // comando di lettura

    for (int i = 0; i < quanti; i++) {
        do
            inputb(iSTS, s);
        while ((s & 0x80) != 0x08);
        for (int j = 0; j < 512/2; j++)
            inputw(iBR, vetti[i*512/2 + j]);
    }
}

natb buff[8*512];
int main()
{
    natl lba = 1;
    natb quanti = 2;                // massimo 8
    leggisett(lba, quanti, reinterpret_cast<natw*>(buff));
    for (int i = 0; i < quanti*512; i++)
        char_write(buff[i]);
    char_write('\n');
    pause();
    // ...
    return 0;
}
```

COSA FA LA CPU DOPO ACCETTAZIONE DELL' INTERRUPT



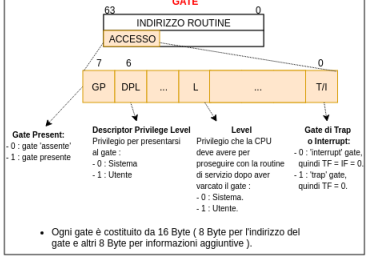
INTERRUZIONE
Segnale inviato alla CPU, emesso via HW o SW, che indica il bisogno di attenzione per il soddisfacimento di una data richiesta di servizio.

IDT (Interrupt Descriptor Table)



- Contiene 256 gate (uno per ogni tipo di interrupt).

GATE



- Ogni gate è costituito da 16 Byte (8 Byte per l'indirizzo del gate e altri 8 Byte per informazioni aggiuntive).

ROUTINE DI SERVIZIO

- Inizia dall'indirizzo corrispondente al nuovo valore caricato in %RIP
- Terminazione routine :
  1) Ritorno al programma interrotto.
  2) Il ritorno avviene con l'esecuzione di IRETQ che preleva 5 parole quadruple (RIP, RFLAGS, CPL, RSP e quegli 8 Byte a noi oscuri che vengono messi per motivi di compatibilità)

TIPOLOGIA ROUTINE

DRIVER
Ha il compito di trasferire effettivamente i byte e sbloccare il processo quando l'operazione si è conclusa.

PRIMITIVA
Prodotta dall'esecuzione un'istruzione 'int $x' per eseguire operazioni che l'utente non può/ non dovrebbe fare, altrimenti potrebbe alterare strutture del sistema, corrompere file di sistema e chi più ne ha, più ne metta.

La primitiva ha lo scopo di avviare l'operazione di I/O e bloccare il processo, garantendo anche la mutua esclusione

TIPOLOGIE INTERRUZIONI

SW
- *Prodotto da* un'istruzione del tipo *int $tipo*
- **Sincrona** al programma in esecuzione
- **NON sospende** l'esecuzione di un'istruzione
- **tipo** interruzione : specificato dall'**operando immediato** (32 bit) dell'istruzione 'int $..'
- **Indirizzo di ritorno** : **istruzione successiva** all'ultima istruzione eseguita prima dell'accettazione dell'interrupt.
- **Routine di servizio** : invoca routine di sistema (PRIMITIVE) con un meccanismo alternativo alla call.

ECCEZIONE
- *Prodotta da* circuiterie interne alla CPU al verificarsi di una condizione anomala che impedisce il completamento dell'esecuzione dell'istruzione in corso.
- **Sincrona** al programma in esecuzione
- **PUÒ sospendere** l'esecuzione di un'istruzione quando questa causa un'anomalia.
- **Tipo** interruzione : **implicito**.
- **Indirizzo di ritorno** : **dipende**. Vedere le tipologie di eccezione.
- **Routine di servizio** : sostituendosi al programma in esecuzione, essa impedisce che il programma continui ad operare in modo scorretto.

ESTERNA
- *Prodotto da* :
  - **/INTR** : interruzioni mascherabili.
  - **/NMI** : interruzioni non mascherabili.
- **Asincrona** al programma in esecuzione.
- **NON sospende** l'esecuzione di un'istruzione.
- ***tipo*** interruzione :
  - /INTR : ciò che l'APIC invia alla CPU.
  - /NMI : tipo implicito '2'.
- ***Indirizzo di ritorno*** : **istruzione successiva** all'ultima istruzione eseguita prima dell'accettazione dell'interrupt.
- ***Routine di servizio*** : invoca routine di sistema (PRIMITIVE) con un meccanismo alternativo alla call.

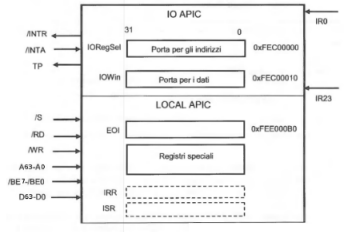
TIPOLOGIE ECCEZIONI

ABORT
- Errore irrecuperabile => terminazione forzata dell'esecuzione dell'istruzione.
- ***Indirizzo di ritorno*** : **non** è **significativo**, perchè l'istruzione eseguita per ultima viene terminata in maniera forzata.

FAULT
- Errore recuperabile.
- Viene sollevato durante l'esecuzione dell'istruzione.
- **Indirizzo di ritorno** : indirizzo dell'**ultima istruzione eseguita (cioè quella che ha generato il fault)**.
  Esempio : nella traduzione di un indirizzo da virtuale a fisico, ho un page fault. Parte la routine che recupera la pagina dallo swap e la carica in RAM e poi si ricomincia da dove ci eravamo interrotti, riseguendo proprio l'istruzione che aveva generato il fault.

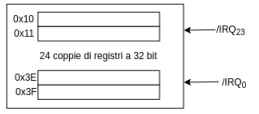
TRAP
- **Sincrona** col programma in esecuzione.
- Solitamente genera un passaggio alla modalità SISTEMA, in cui il sistema operativo esegue alcune azioni prima di restituire il controllo al processo di origine
- **Indirizzo di ritorno** : **istruzione successiva** all'ultima istruzione eseguita prima dell'accettazione dell'interrupt.

SINGLE STEP TRAP
- Consente di eseguire il programma in single step. (una domanda in proposito che bisogna farsi è come funziona il meccanismo di esecuzione di una singola istruzione da parte di gdb?).
- La routine gira con TF = 0 per non essere la routine stessa eseguita in single step.



CARATTERISTICHE APIC

- 3 registri accessibili : **IoRegSel**, **IOWin**, **EOI** (End Of Interrupt).
- 64 registri speciali che stabiliscono alcune modalità di funzionamento, di cui vi sono 24 coppie di registri (una per ogni IR_i , i = 0,...,23) che costituiscono la **TABELLA INTERNA** dell'APIC.

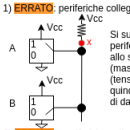


- Livello di priorità è dato dai 4 bit più significativi del tipo di interrupt (quindi avremmo 2^4 = 16 livelli di priorità)
- **IRR** (Interrupt Request Register 256 bit) : setta il bit i-esimo (0 <= i <= 255) corrispondente al tipo i
- **ISR** (In Service Register, 256 bit) : indica i tipi di interruzione attualmente serviti.
- Invia seralmente su TP il tipo della interrupt alla CPU.
- Le periferiche possono condividere linee di interruzione.
- Cosa fa l'APIC?
  1) Ogni volta che riceve una richiesta di interrupt tramite uno dei piedini IR_0-IR_{23}, pone ad 1 il bit di IRR corrispondente al tipo di interrupt. Il controllore però può ricevere degli interrupt in due modi:
  1.1) int. riconosciuta sul fronte di salita di un segnale a forma di impulso.
  1.2) int. riconsciuta a livello, rappresentata da un segnale che risulta attivo nel momento in cui il controllore riceve l'EOI.
  2) APIC ripete ciclicamente i seguenti passi :
  2.1) attende che il bit più significativo di IRR di valore 1 abbia una priorità > bit più significativo di ISR.
  2.2) invia /INTR alla CPU.
  2.3) attende la risposta su /INTA e, quando la riceve :
  2.3.1) considera il bit più significativo di IRR più a sinistra. Sia j.
  2.3.2) pone a 0 il bit j-esimo di IRR e pone ad 1 il bit j-esimo di ISR
  2.3.3) invia j alla CPU in maniera seriale tramite TP.
  2.3.4) disattiva /INTR.
  3) L'APIC, ogni volta che riceva (via SW) l'EOI, azzera il bit (posto precedentemente a 1) più significativo di ISR.

COME FANNO LE PERIFERICHE A CONDIVIDERE LINEE DI INTERRUPT SENZA DARSI NOIA A VICENDA???

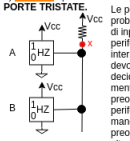
La domanda chiave da farsi, di caso in caso, è "quanto vale x"?

1) ERRATO: periferiche collegate **DIRETTAMENTE** allo stesso filo.



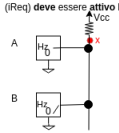
Si supponga che due periferiche siano collegate allo stesso filo. Una invia 0 (massa) e l'altra invia 1 (tensione Vcc). Sarebbe un cc quindi rischieremmo di danneggiare le periferiche.

2) ERRATO : periferiche collegate allo stesso filo tramite **PORTE TRISTATE.**



Le porte tristate rimanderebbero il problema di condivisione della linea di input perchè ogni volta che una periferica deve inviare un segnale di interrupt, tutte le periferiche si devono coordinare tra loro per decidere chi deve usare la linea, mentre le periferiche non dovrebbero preoccuparsi di questo. Una periferica, quando ha bisogno, deve mandare il segnale di interrupt senza preoccuparsi di cosa vogliano fare le altre periferiche

3) CORRETTO : periferiche collegate direttamente allo stesso filo tramite **OPEN COLLECTOR**. Il segnale o non passa o vale 0. La richiesta di interruzione è ottenuta tramite il cosiddetto OR DI COLLETTORE => il segnale di interruzione (iReq) **deve** essere **attivo basso**



A	B	/iReq
0	0	0
0	-	0
-	0	0
-	-	1

LEGENDA (per gli schemi relativi alla condivisione di linee di interrupt da parte di più periferiche)

Collegamento a terra (altresì chiamato "a massa")

Vcc
Tensione Vcc di alimentazione

PRIMITIVA I/O

Deve:
1) Avviare l'operazione di I/O.
2) Bloccare il processo, garantendo la mutua esclusione.

//utente.cpp
```
extern "C" void read_n(natl id, natb* buf, natl quanti)
```

#utente.s
```
.global read_n
read_n:
  int $IO_TIPO_RN
  ret
```

#sistema.s
```
.extern c_read_n
a_read_n:
  cavallo_di_troia %rsi
  cavallo_di_troia2 %rsi %rdx
  call c_read_n
  iretq
```

GL : UTENTE
L : SISTEMA
T/I : indifferente

//sistema.cpp
```
extern "C" void c_read_n(natl id, char* buf, natl quanti){
  //seleziono descrittore di I/O. Sia esso puntato da 'd'
  struct des_io *d = &array_des_io[id];

 //aspetto che la risorsa(descrittore) sia liberata
sem_wait(d->mutex);
  //...preparo i campi del descrittore
  d->buf = buf;
  d-> quanti = quanti;
  //...abilito le interruzioni
  outputb(1, d->iCTL);

  /*aspettiamo che l'operazione sia conclusa*/
  sem_wait(d->sync);

  sem_signal(d->mutex);   /*liberiamo la risorsa
                            precedentemente occupata*/
}
```

I/O NEL NUCLEO

Un processo che inizia un'operazione di I/O non può proseguire finchè l'operazione non è terminata => dobbiamo bloccare un processo che inizia un'operazione di I/O e risvegliarlo (riportarlo in coda 'pronti') una volta che l'operazione di I/O è conclusa.

MODULO I/O

I problemi del driver sono risolvibili introducendo un modulo separato dai moduli UTENTE e SISTEMA : il MODULO I/O.
Esso ha il compito di fornire tutte le primitive l'accesso alle periferiche, gestendo le richieste di I/O tramite veri e propri processi detti **PROCESSI ESTERNI**

DRIVER

Deve:
1) Trasferire effettivamente i Byte.
2) Sbloccare il processo quando l'operazione si è conclusa.

#sistema.s
```
.extern c_driver
a_driver_i:
  call salva_stato
  movq $i, %rdi
  call c_driver
  call APIC_SEND_EOI
  call carica_stato
  iretq
```

GL : SISTEMA
L : SISTEMA
T/I : INTERRUPT

//sistema.cpp
```
extern "C" void c_driver(natl id){
  des_io *d = &array_des_io[id];
  char c;

  d->quanti--;  //aggiorno quanti
  if( !d->quanti ) { //ho finito i trasf.
    outputb(0, d->iCTL);
    c_sem_signal(d->sync);
    /*uso la c_primitiva e non la primitiva per intero
    perchè sennò avrei due salva_stato una di seguito
    all'altra senza avere almeno una carica_stato nel
    mezzo */
  }

  inputb(d->iRBR, c); //prendo il contenuto del buffer
  *d->buf = c;
  d->buf++;
}
```

PROBLEMI :
1) **Driver deve essere eseguito con interruzioni disabilitate, dato che manipola code di processi**. Inoltre si supponga il caso di read_n : io devo disabilitare le interruzioni in quanto, se devo leggere l'ultimo byte restituisco una risposta alla richiesta di interruzione della periferica, quest'ultima, se le interruzioni fossero abilitate, potrebbe inviarmi una nuova interruzione rimandando in esecuzione il driver e sovrascrivendo zone casuali di memoria
2) **Il driver non si può bloccare dato che non è un processo**.
3) **Buffer deve essere allocato in zone di memoria utente/condivisa**.
La scrittura/lettura di un buffer è portata avanti mentre è in esecuzione un processo differente da quello che aveva allocato il buffer. Supponiamo di avere due processi P1 e P2.
P2 legge/scrive e P1 ha allocato il buffer. Questa cosa è un problema se P1 ha allocato il buffer nella sua memoria utente/privata (dato che gli indizzi virtuali delle sezioni private hanno significati diversi per ogni processo). Da un lato P1 non riceverebbe i suoi dati e, dall'altro, P2 si vedrebbe sovrascrivere parti casuali della SUA memoria.

HANDLER

1) Deve mettere in esecuzione il processo esterno relativo ad una data interruzione, prendendo il proc_elem da una tabella (modulo SISTEMA) **a_p** con 24 entrate (tanti quanti sono i piedini di interrupt dell'APIC).
2) In fase di inizializzazione della IDT, il modulo sistema deve programmare l'APIC e riempire le entrate della IDT, in modo che ogni richiesta salti al giusto handler.

#sistema.s

```
handler_i:
  call salva_stato
  call inspronti  /*mette in testa alla coda
pronti il processo finora in esecuzione*/
  movq $i, %rcx
  movq a_p(,%rcx,8), %rax
  movq %rax, esecuzione  /*esecuzione
punta al proc_elem del processo esterno*/
  call carica_stato
  iretq
```

PRIMITIVA SISTEMA + DRIVER

- Driver userà le tabelle di traduzione del processo che ha interrotto => se vogliamo che il driver acceda al **buffer** tramite le suddette tabelle, esso **si deve trovare nella zona UTENTE/CONDIVISA**.
- **Buffer deve essere non rimpiazzabile** altrimenti il codice del modulo sistema potrebbe generare un page fault e, se il codice della parte sistema dovesse mai sollevare un'eccezione, sarebbe da considerare come un grave errore.

HANDLER + PROCESSO ESTERNO

- Buffer deve essere accessibile sia al processo utente, sia al processo esterno => **buffer in zona UTENTE/CONDIVISA**
- Il **processo esterno**, in quanto processo, **può generare page fault senza problemi** => **non è necessario che buffer sia non rimpiazzabile**.
- Se si vogliono gestire i trasferimenti dallo swap tramite le interruzioni, tali interruzioni devono avere priorità massima, altrimenti l'APIC non le farebbe passare mentre è in esecuzione un processo esterno.

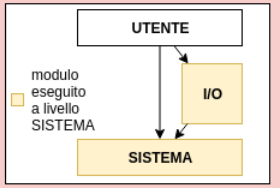
2 CASI

MEMORIA VIRTUALE

I byte sono trasferiti mentre c'è in esecuzione, in genere, un processo diverso da quello che aveva richiesto il trasferimento.
Il processo utente che ha iniziato il trasferimento ha passato alla primitiva l'indirizzo di un suo buffer. Tale buffer sta nella memoria virtuale (vedere schemi sulla MEMORIA VIRTUALE) del processo.

MODULO I/O

Fornisce le primitive di I/O per l'accesso alle periferiche, gestendo le operazioni di I/O tramite processi detti **PROCESSI ESTERNI**



NB : codice modulo I/O gira a interruzioni abilitate

INTERAZIONE CON ALTRI MECCANISMI

BUS MASTERING

I trasferimenti eseguiti dalle periferiche in grado di operare in Bus Mastering non attraversano la MMU (vedere gli schemi sulla MEMORIA VIRTUALE).

- L'indirizzo VIRTUALE del buffer passato alla primitiva non può essere usato dalla periferica perchè va prima tradotto in FISICO. Per tale scopo, si usa la funzione

  //io.cpp
  addr trasforma(addr ind_virt);

- Usando l'indirizzo fisico, la periferica accederà direttamente alla memoria fisica. Ne consegue che

Non è **necessario** che il **buffer si trovi nella parte UTENTE/CONDIVISA**

Il buffer **non deve essere rimpiazzato** per tutta la durata del trasferimento

- Se il buffer attraversa più pagine, non sarà possibile completare tutto il trasferimento in un'unica operazione dato che i frame (o PAGINE FISICHE) contenenti pezzi del buffer potrebbero non essere contigui in memoria.

PROCESSO ESTERNO

- Lo scopo principale del processo, come quello del driver, è di leggere (scrivere) dall'interfaccia (nell'interfaccia) e copiare i dati nel (dal) buffer dell'utente.
- Si assume che nel sistema vi siano più periferiche simili, ciascuna gestita da un diverso processo esterno il cui codice può essere scritto una volta per tutte => scriviamo il codice del processo esterno una volta.
- Dopo la creazione, il processo esterno non termina più (quindi il contatore globale 'processi' non viene incrementato ogni volta che si risveglia un processo esterno).
- Dopo aver risposto ad una richiesta di interruzione, il processo esterno si sospende in attesa della prossima richiesta.

//io.cpp

```
extern "C" void estern(natl i){
  des_io *d = &array_des_io[i];
  for(;;){
    /*
     Qui il processo esterno (1) risponde alla richiesta di interruzione e
     (2) procede a leggere (scrivere) Byte dall'interfaccia (nell'interfaccia) e
     copiarli nel (dal) buffer utente.
    */

    wfi();
  }
}
```

WFI (Wait For Interrupt)

#io.s

```
.global wfi
wfi:
  int $TIPO_WFI
  ret
```

#sistema.s

```
a_wfi :
  call salva_stato
  call APIC_SEND_EOI
  call schedulatore
  call carica_stato
  iretq
```

istruzioni che mettono in esecuzione un altro processo (cioè sospendiamo il processo esterno).
NB **Non sappiamo quale processo verrà messo in esecuzione** dopo l'esecuzione di a_wfi perchè **il processo esterno gira a interruzioni abilitate**, quindi vari processi (anche a priorità più alta del processo in testa alla coda 'pronti') potrebbero essere finiti in coda 'pronti'

ACTIVATE_PE

- Analoga alla activate_p()
- Deve creare un processo esterno
- Contrariamente alla activate_p() **non aumenta** la variabile globale **'processi'** perchè i processi esterni non terminano se non alla chiusura, e quindi impedirebbero al processo 'dummy' di andare in esecuzione.

CODICE ACTIVATE_PE()

#sistema.s

```
carica_gate   TIPO_APE   a_activate_pe   LIV_SISTEMA
/*..*/
  .extern c_activate_pe
a_activate_pe:
  cavallo_di_troia %rdi
  call c_activate_pe
  iretq
```

//sistema.cpp

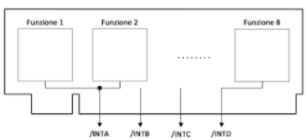
```
extern "C" natl c_activate_pe(void f(int), int a, natl prio, natl liv, natb type){
proc_elem*p;   // proc_elem per il nuovo processo

if (prio < MIN_PRIORITY) {
  flog(LOG_WARN, "priorita' non valida: %d", prio);
  abort_p();
}

p = crea_processo(f, a, prio, liv, true);
if (p == 0)
goto error1;

if (!aggiungi_pe(p, type) )
goto error2;

flog(LOG_INFO, "estern=%d entry=%p(%d) prio=%d liv=%d type=%d",p->id, f, a, prio, liv, type);

return p->id;

error2:distruggi_processo(p);
error1:
return 0xFFFFFFFF;
}
```

DISPOSITIVO

- Ad ogni linea di bus possiamo collegare **al più 32 dispositivi**.
- Ogni dispositivo sul bus è identificato da un **numero** che dipende dallo **slot in cui è fisicamente montato il dispositivo.**
- Possiede **n piedini con cui inviare interruzioni** ($1 \leq n \leq 4$)

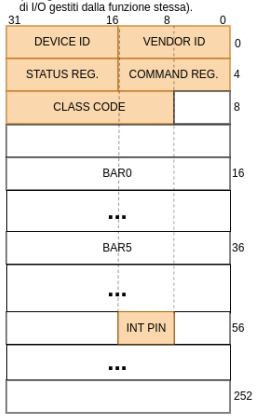
FUNZIONE

- Per ogni dispositivo abbiamo, **al più, 8 funzioni**.
- Il numero di funzione è definito dal particolare costruttore del dispositivo.
- Deve implementare, nello spazio di configurazione, alcuni registri.
- Per ogni funzione viene assegnato al più uno dei 4 piedini di interrupt che vanno a finire nell'APIC (per intenderci, parlo dei piedini /A, /B, /C, /D. Questi vanno a finire nei piedini i-esimi dell'APIC, con i = 16,..,19).
- Più funzioni possono condividere uno stesso piedino di interrupt



SPAZIO DI CONFIGURAZIONE PRIVATO DI UNA FUNZIONE

- Specifico di ogni funzione
- 256 locazioni da 1 Byte
- Registri lunghi da 1 a 4 Byte
- Scopi dei registri :
  1) identificazione funzione;
  2) programmare la configurazione dellla funzione (assegnando indirizzi ai blocchi di memoria ed ai blocchi di I/O gestiti dalla funzione stessa).

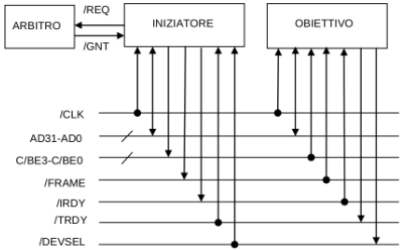


- **DEVICE ID**(16 bit, R) : identificatore del dispositivo.
- **VENDOR ID**(16 bit, R) : identificatore del venditore.
- **STATUS REG**(16 bit, R) : segnala gli errori rilevati durante le transazioni.
- **COMMAND REG**(16 bit, R/W) :
  - bit 0 (o bit IO) : se settato, abilita le transazioni nello spazio di I/O.
  - bit 1 (o bit MEM) : se settato, abilita le transazioni nello spazio di MEM.
  - bit 2 (o bit BM) : se settato, abilita le funzionalità di bus mastering.
- **CLASS CODE**(24 bit, R) : specifica il tipo di operazione svolta (secondo la codifica PCI).
- **INT PIN** (8 bit, R) : specifica il piedino tramite il quale la funzione invia richieste di interrupt.

LINEE PRINCIPALI DEL BUS PCI

- **AD$_{31\text{-}0}$** (Address / Data)
- **C/BE$_{3\text{-}0}$** (Command / Byte enable)
- **/FRAME** : usato da un iniziatore per cominciare una transazione
- **/IRDY** (Initiator ready) : attivato dall'iniziatore per segnalare (al target) che è pronto a ultimare una fase dati.
- **/TRDY** (Target ready) : come per /IRDY ma in questo caso è il target che segnala all'iniziatore di essere pronto a ultimare una fase dati.
- **/DEVSEL** (Device select) : attivato dal target nella fase di indirizzamento per rispondere alla richiesta di transazione dell'iniziatore.
- **IDSEL** (Initialization Device SELect) : usato nelle transazioni di configurazione.

NB Dato che le linee AD e C/BE hanno duplice funzionalità, ho sottolineato la loro specifica funzione passo dopo passo. Esempio : A**D** significa che uso le linee Address/Data come Data.



SPAZIO DI CONFIGURAZIONE GLOBALE

- Costituito dagli spazi di configurazione privati delle funzioni.
- Ogni indirizzo è composto da :
  1) bus;
  2) dispositivo (nel bus);
  3) funzione (del dispositivo);
  4) indirizzo del registro nello spazio di configurazione privato.

CONFIGURAZIONE

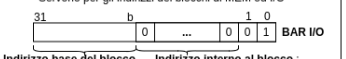
SPAZI DI INDIRIZZAMENTO

MEM o I/O

- 2^{32} locazioni da 1 Byte

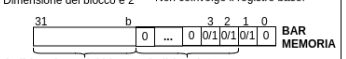
BAR(Base Address Register)

Servono per gli indirizzi dei blocchi di MEM od I/O



Indirizzo base del blocco (lungo al più 30 bit). Dimensione del blocco è 2^b

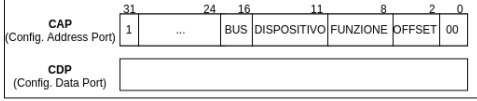
Indirizzo interno al blocco : anche 0 bit (1 parola lunga). Non coinvolge il registro base.



Indirizzo base del blocco (lungo al più 28 bit). Dimensione blocco è 2^b

Indirizzo interno al blocco : almeno 2 bit (4 parole lunghe). Non coinvolge registro base.

PONTE OSPITE-PCI



TRANSAZIONE

- Operazione svolta sul bus PCI
- 2 tipi : R/W
- Transazione comune :
  1) CPU comanda ciclo di R/W (entità indirizzata non si trova sul bus locale ma sul bus PCI)
  2) Ponte OSPITE-PCI trasforma il ciclo di R/W in una transazione.
- Prevede 3 entità :

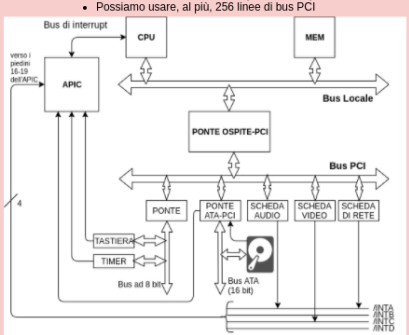
ARBITRO
- Dispositivo che coordina l'accesso al bus da parte dei dispositivi.
- Usa i piedi **/REQ** (REQuest) e **/GNT**(GraNT) per comunicare con l'INIZIATORE.

INIZIATORE
Dispositivo che inizia la transazione

OBIETTIVO
Dispositivo indirizzato dalla transazione

BUS PCI

- Standard di comunicazione tra periferiche.
- Si trova nelle versioni a 32 bit (usata da noi) e 64 bit
- Possiamo usare, al più, 256 linee di bus PCI



FASI DI UNA TRANSAZIONE

TIPI DI TRANSAZIONE

FASE 0 : ARBITRAGGIO

1) Iniziatore attiva /REQ.
2) Arbitro risponde attivando /GNT.
3) Iniziatore attende che un'eventuale ultima transazione in corso si concluda (aspetta che /FRAME ritorni ad 1).
4) Iniziatore attiva /FRAME.
5) Iniziatore disattiva /REQ.
6) Arbitro risponde disattivando /GNT.

NB Se ti dovesse mai sorgere il dubbio di "Cosa potrebbe avvenire tra il punto 4 ed il punto 5? Avviene tutto direttamente oppure si interrompe l'arbitraggio e si prosegue con le altre fasi per poi ritornare all'arbitraggio e disattivare /REQ e /GNT?" pensa al fatto che per tutte le fasi (indirizzamento e le prime N-1 fasi dati), ad eccezione dell'ultima fase dati, avviene tutto tenendo /FRAME attivo e l'iniziatore nel punto 3 attende che un'eventuale ultima transazione sia conclusa. Fose è un consiglio superfluo ma credo che il dubbio possa sempre presentarsi.

FASE 1 : INDIRIZZAMENTO

1) Iniziatore attiva /FRAME e pilota A**D** e **C**/BE.
2) Il dispositivo che riconosce l'indirizzo e lo spazio, risponde attivando /DEVSEL e diventa l'obiettivo della transazione.
3) Se la linea /DEVSEL non si attiva entro 6 cicli di clock (non della CPU ma del bus PCI!), la fase di indirizzamento termina restituendo il valore di errore 0xFFFFFFFF, altrimenti si procede con la/le fase/i dati.

NB : se non hai capito perchè ho sottolineato/messo in grassetto i vari elementi nel testo, ti consiglio di leggere il riquadro **LINEE PRINCIPALI DEL BUS PCI.**

FASE 2 : DATI

Possiamo avere una o più fasi dati. Se abbiamo a che fare con blocchi di I/O, in genere basta una sola fase mentre coi blocchi di MEM abbiamo mediamente più di una fase dati.

Supponiamo di avere N fasi dati.
Per tutte le fasi i-esime (i = 1, ... , N-1) cioè, per tutte le fasi diverse dall'ultima,
1) l'iniziatore tiene attivo /FRAME e pilota C/**BE**.
2) Nel caso di una transazione di WRITE (READ), l'iniziatore (obiettivo) pilota A**D**, attiva /IRDY (/TRDY) e attende che l'obiettivo (iniziatore) abbia attivato /TRDY (/IRDY).
3) Quando sono entrambi attivi, la fase dati termina, quindi disattivano /IRDY e /TRDY.

Si continua così fino all' N-esima fase dati, cioè l'ultima.
Questa è simile in tutto e per tutto alle precedenti N-1 fasi dati, se non per il fatto che l'iniziatore non tiene attivo /FRAME.

MEM od I/O

1) Iniziatore emette indirizzo su **A**D.
2) Inizatore emette il tipo di operazione (R/W) e spazio (MEM od I/O) codificati su **C**/BE.
3) nel caso di una WRITE (READ), l'iniziatore (obiettivo) pilota A**D** per trasferire al più 4 Byte di dati per volta.
4) In ogni caso, l'iniziatore pilota C/**BE** per abilitare il trasferimento dei singoli Byte.

CONFIGURAZIONE

1) Dispositivo PCI possiede piedino di ingresso IDSEL (Initialization Device SELect) e risponde ad una transazione di configurazione **se e solo se** IDSEL è attivo e AD$_{10}$ = 00.
2) Se il dispositivo risponde alla transazione di configurazione,
   - 2.1) Il dispositivo legge AD10-8 (n° funzione).
   - 2.2) Il dispositivo legge AD7-2 (offset).
   - 2.3) Se la funzione esiste, il dispositivo risponde attivando /DEVSEL.

   → **FASE INDIRIZZAM.**

   - 2.4) Si comporta come in una normale transazione di R/W.

   → **FASE DATI**

COLLOQUIO CPU - PONTE OSPITE-PCI

1) /HOLD e /HOLDA (/HOLD Accept) sono disattivati.
2) Ponte riceve una richiesta di transazione per Bus Mastering, quindi attiva /HOLD.
3) La CPU, dopo aver completato l'eventuale ciclo di bus (locale), si "disconnette" dal bus locale e attiva /HOLDA.
4) Il ponte si impossessa del bus locale e, dopo l'utilizzo, disattiva /HOLD.
5) CPU disattiva /HOLDA.

DMA (Direct Memory Access)

Non è opportuno occupare la CPU con l'esecuzione di istruzioni per il trasferimento dati tra MEM e una qualsiasi interfaccia di un dispositivo collegato al calcolatore, ergo col DMA, i trasferimenti tra interfacce varie e MEM avviene in autonomia, senza l'intervento della CPU

OPERAZIONI IN BUS MASTERING

- Può coinvolgere uno o più buffer di MEM.
- Usa buffer di MEM nei primi 4 GiB.
- Richiede la preparazione in MEM di una tabella di descrittori di buffer.

DESCRITTORE DI BUFFER

INDIRIZZO BUFFER		
EOT		n° Byte

End Of Table : indica se questo descrittore è relativo all'ultimo pezzo di buffer.

Numero di Byte della porzione di buffer da usare **NB** NON È la dimensione del buffer

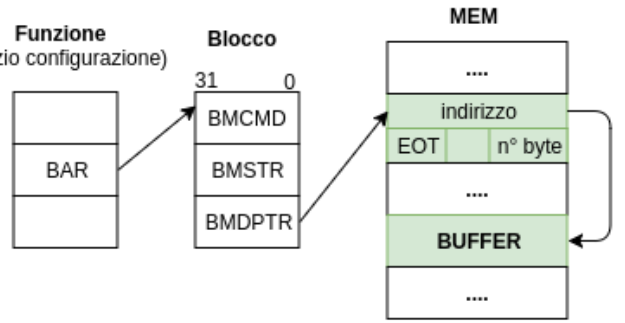
Perchè abbiamo bisogno di questi descrittori di trasferimento? Spesso la periferica deve ordinare più trasferimenti perchè non ha pronti tutti i dati o perchè l'invio dei dati è pacchettizzato (si pensi ad esempio all'appello 2018-09-19).

NB Il buffer DEVE essere allineato naturalmente

FUNZIONI CON CAPACITÀ DI BUS MASTERING

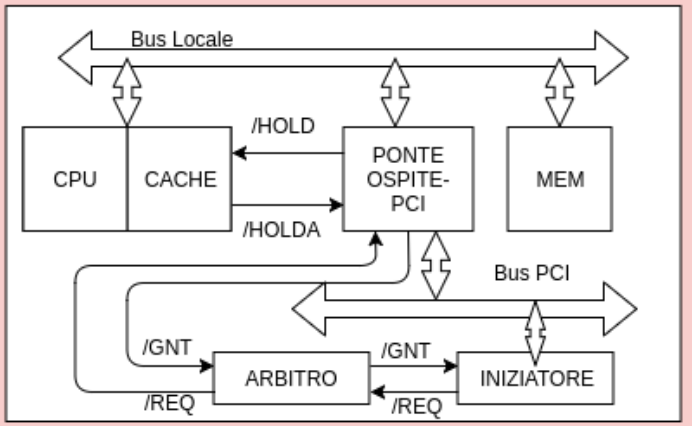
Implementano 3 registri :

- **BMCMD**(Bus Mastering CMD Reg, 32 bit) : specifica il tipo di trasferimento (R/W).
- **BMSTR**(Bus Mastering STatus Reg, 32 bit) : stato dell'ultimo trasferimento.
- **BMDPTR**(Bus Mastering Descriptor Table Pointer Reg, 32 bit) : punta alla tabella dei descrittori di buffer.



BUS MASTERING

- È una forma di DMA
- La logica fondante è la seguente:
  - INIZIATORE = dispositivo collegato al bus PCI
  - OBIETTIVO = PONTE OSPITE-PCI
  - ARBITRO = dispositivo collegato al bus PCI
- La CPU deve garantire **l'accesso in mutua esclusione al bus locale**



PROBLEMI CON LA MEMORIA CACHE

LETTURA DALLA MEM

Se la **cache segue la politica WRITE BACK** e il dispositivo vuole un dato che la CPU ha modificato solo in cache, leggendo il corrispettivo dato dalla MEM, il dispositivo leggerebbe un dato obsoleto.

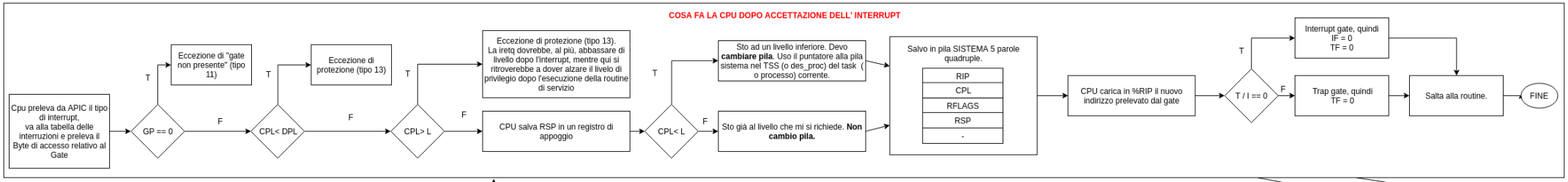
SOLUZIONE : ogni volta che il controllore CACHE attiva /HOLDA, esamina anche l'indirizzo e il tipo di operazione (in tal caso, READ) sul bus locale. Se l'indirizzo è relativo ad un dato già presente in CACHE, il controllore CACHE comanda la lettura direttamente in CACHE, così che la periferica possa leggere il dato aggiornato.

SCRITTURA SULLA MEM

Quando avviene un'operazione di scrittura verso la MEM, le eventuali informazioni che sono presenti anche in CACHE, se non vengono aggiornate, non sono più valide => se un dispositivo scrive in MEM e la CPU continua ad usare la CACHE, la CPU userebbe un dato obsoleto.

SOLUZIONE 1 : ogni volta che il controllore CACHE attiva /HOLDA, esamina anche l'indirizzo e il tipo di operazione sul bus locale. Se l'indirizzo è relativo ad un dato già presente in CACHE, il controllore CACHE comanda la scrittura direttamente in CACHE prelevando il dato dal bus dati ($D_{63\text{-}0}$ per intenderci)
SOLUZIONE 2 : se si verifica la stessa condizione, invece di modificare in CACHE, si invalida la linea di CACHE contenente il dato modificato in MEM.

COSA FA LA CPU DOPO ACCETTAZIONE DELL' INTERRUPT



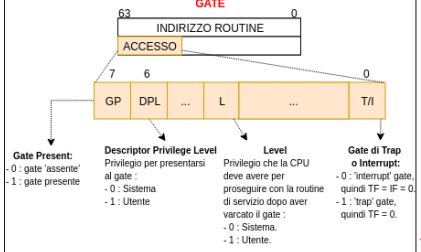
CAMBIO DI LIVELLO DI PRIVILEGIO

Viene fatto con un'istruzione del tipo
'INT $TIPO_INT'.
Le routine che vanno in esecuzione in questo modo, prendono il nome di **primitive di sistema**. (guardare lo schema "**PRIMITIVA**" per ulteriori chiarimenti).

Perchè usare "INT $ TIPO_INT" invece di una "CALL NOME_FUNZIONE " particolare che innalza il livello di privilegio della CPU?

1) UTENTE potrebbe saltare in mezzo al codice delle primitive (ad esempio per poter saltare alcuni controlli)
2) Se gli indirizzi sono specificati in maniera simbolica, i programmi UTENTE andrebbero collegati col CODICE SISTEMA per poter risolvere i simboli.

GATE



- Ogni gate è costituito da 16 Byte (8 Byte per l'indirizzo del gate e altri 8 Byte per informazioni aggiuntive).

PROTEZIONE

PROBLEMI :
1) La CPU esegue. nel corso del tempo, vari JOBS (o PROCESSI). Ciascun JOB è il risultato dell'esecuzione di un programma UTENTE. Noi **non possiamo aspettarci che un UTENTE voglia cedere la CPU ai JOB di altri UTENTI**. Ciascuno vorrebbe tenere la CPU tutta per sé. Mentre è in esecuzione un certo programma, la CPU obbedisce a quel programma e dunque obbedisce al volere degli UTENTI.
Dobbiamo quindi sviluppare un meccanismo che permetta alla CPU di non credere ciecamente alle istruzioni del programma corrente.
2) I PROGRAMMATORI SISTEMA possono scrivere le routine necessarie per svolgere le operazioni "delicate" che manipolano strutture dati sensibili, ma nulla vieta agli UTENTI di usare altre vie. **Dobbiamo impedire loro determinati comportamenti, come usare istruzioni privilegiate oppure leggere / modificare determinate sezioni di MEM.**
3) Supponendo di usare i CONTESTI, dobbiamo fare in modo che gli UTENTI non possano modificare le routine di SISTEMA. Come? Ci basta **suddividere la MEM in due zone** : una parte **UTENTE** e una **SISTEMA**. **La seconda deve essere acceduta con CPU a livello SISTEMA**. Ogni tentativo di accesso alla MEM SISTEMA da parte di JOB di livello UTENTE deve sollevare un'eccezione di protezione.
4) Il programma dell'UTENTE 1 potrebbe modificare il programma dell'UTENTE 2. Da quello che abbiamo detto al punto 3, i programmi UTENTE possono accedere alla MEM UTENTE, si... Ma anche qui dobbiamo porre dei paletti : **quando è attivo il contesto del JOB 1, questo non deve poter accedere al contesto privato del JOB 2 o di altri JOB**.

SOLUZIONE : abbiamo bisogno di :
1) un modo per definire dei **contesti** privilegiati e non privilegiati.
2) Un modo per stabilire quale è il **contesto corrente**.
3) **Regole** che stabiliscano **cosa si può/non si può fare** in ogni contesto.
4) Un modo per fare un **cambio di contesto**.

CAMBIO PILA

1) CPU deve garantire di poter scrivere le 5 quad-words senza sovrascrivere altre cose in memoria => usiamo la PILA SISTEMA.
2) Queste info vanno salvate in M1, in modo che l'UTENTE non le possa alterare/corrompere

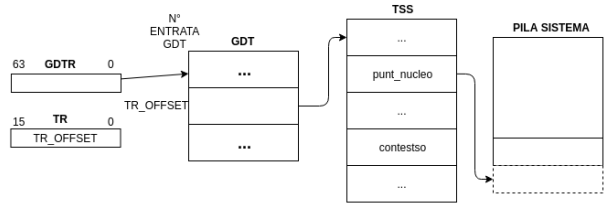
AZIONI IRETQ

1) Se il livello di privilegio della CPU è inferiore al contenuto di CPL salvato nella PILA SISTEMA scatta un'eccezione di protezione.
2) Ripristina i valori di RIP, RSP, CPL ed RFLAGS.

TSS (Task State Segment)

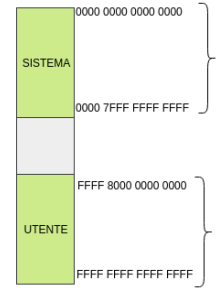
Descrittore (del processo) che contiene il contesto del processo e vari altri dati utili per gestire i vari processi.

- Al 4° if statement di "COSA FA LA CPU DOPO L'ACCETTAZIONE DELL'INTERRUPT" noi cambiamo pila. Il TSS contiene un campo di tipo *addr*, chiamato **punt_nucleo** che punta **SEMPRE** alla **base della pila sistema**.
- Per ogni processo abbiamo un solo TSS.
- Tabella **GDT** (**Global Descriptor Table**) contiene tutti i descrittori TSS (indirizzo del TSS, dimensione in byte).
- Nuovi registri CPU :
  1) **TR** (**Task Register**) che memorizza l'id del processo attualmente in esecuzione. Per essere precisi, per recuperare il TSS "giusto" l'indirizzo sarebbe GDTR + TR*8 dato che ogni elemento della GDT pesa ed è allineato ad 8 byte.
  Si noti che, nonostante il campo 'id' della struct proc_elem sia di 4 Byte, 'id' viene troncato ai primi 2 Byte meno significativi e conseguentemente usato. (per capire cosa intendo, si legga il riquadro GDT).
  2) **GDTR** (**Global Descriptor Table Register**) : punta alla GDT.
- Nuove istruzioni assembly :
  1) LTR aggiorna TR ad ogni cambio di processo in esecuzione.
  2) LGDT carica la tabella di descrittori di task (o processi) in MEM.



LIVELLI DI PRIVILEGIO

- CPU si trova sempre in uno dei due livelli (di privilegio) SISTEMA / UTENTE
- Per vedere il livello di privilegio attuale della CPU dobbiamo consultare il **CPL** (Current Privilege Level).
- La MEM è separata in due parti :
  1) **M1** (o MEM **SISTEMA**).
  2) **M2** (o MEM **UTENTE**).
- *Cambi di livello* di privilegio:
  1) **INNALZAMENTO (da UTENTE a SISTEMA)** : effettuato grazie all'attraversamento del gate della IDT.
  2) **ABBASSAMENTO (da SISTEMA a UTENTE)** : effettuato tramite l'esecuzione della 'iretq'.
- Alcune istruzioni privilegiate : IN, OUT, LIDT, LGDT, modifica di CR3, modifica di RFLAGS (ad es. modifica di IOPL ->I/O Privilege Level : specifica il livello di privilegio minimo che la CPU deve avere per poter abilitare/disabilitare le interruzioni ed eseguire le istruzioni IN e OUT), ecc..
- Cose che un programma UTENTE non può (e **non deve**) fare:
  - Eseguire istruzioni privilegiate.
  - Tentare di accedere (R/W) ad M1.
  - Tentare di accedere (R/W) alla memoria di un altro processo UTENTE..
- Se un programma UTENTE prova a fare qualcosa di illecito solleviamo un'eccezione di protezione.
- Salviamo tutte le strutture dati sensibili del kernel nella zona M1 della MEM.



GDT (approfondimento)

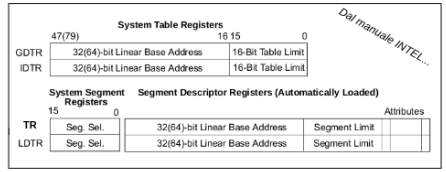
Premetto che questo materiale non è spiegato nelle dispense ma sarebbe pur sempre utile per chi ha cominciato a spulciare il nucleo e magari vuole chiarire qualche dubbio sulla GDT.

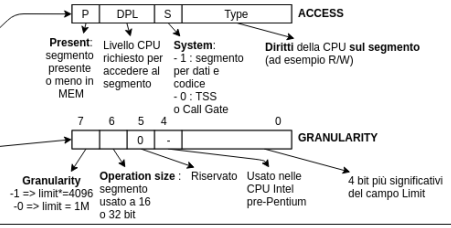
La GDT è una struttura dati usata per definire aree di MEM nell'esecuzione di un programma, come per esempio l'indirizzo base, la dimensione e i privilegi di accesso ed eseguibilità e la scrittura. Queste aree di MEM sono chiamate **segmenti** nella terminologia Intel.

Ogni 8 byte nella GDT rappresentano un descrittore tra :
- TSS (Task State Segment)
- LDT (Local Descriptor Table, non trattato nel nostro corso)
- Call Gate (non trattato nel nostro corso).

Nel nostro nucleo noi supponiamo di avere 1024 TSS. (file costanti.h, #define NUM_TSS 1024).
Di seguito è proposta la struttura dettagliata dei descrittori inclusi nella GDT.

Dimensione	Nome	Descrizione
word	LimitLow	I primi 16 bit del Limit
word	BaseLow	I primi 16 bit della Base
byte	BaseMiddle	I bit 17 - 24 della Base
byte	Access	
byte	Granularity	
byte	BaseHigh	I bit 25 - 32 della Base





PROGRAMMA vs PROCESSO

- Uno stesso programma può essere associato a più processi (ma non è vero il viceversa).
- In generale, non è esclusivamente il programma a decidere attraverso quali stati il processo dovrà passare, ma contribuirà anche l'input stesso.

CAMBIO DI CONTESTO

- Il cambio di contesto può avvenire solo quando il processo si porta a livello SISTEMA, quindi solo quando si verifica una interrupt.
- Per non alterare lo stato del processo che è stato interrotto, dobbiamo prima salvarlo per poi ricaricarlo in un secondo momento, prima di passare alla sua esecuzione. Facciamo tutto usando la sequenza di istruzioni

#file.s
CALL **salva_stato**
...
CALL **carica_stato**
IRETQ

PROCESSO DUMMY

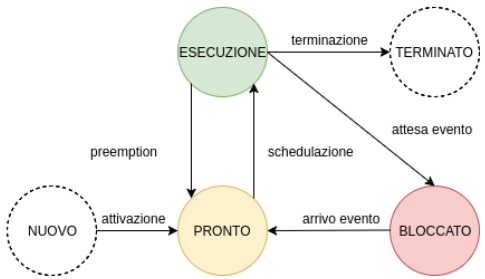
- Per evitare di gestire in modo speciale il caso in cui tutti i processi sono bloccati, è sufficiente che il sistema crei un processo a priorità minima, detto DUMMY.
- Può eseguire un ciclo infinito.
- Quando DUMMY scopre che tutti gli altri processi sono terminati, esso esegue lo shutdown del sistema.

PRIMITIVA ACTIVATE_P

#sistema.s
1) Controllo che il puntatore a funzione (in %rdi) sia un parametro che stia in una zona lecita di MEM.
2) Chiamo c_activate_p
//sistema.cpp
3) Controllo che la priorità *prio* sia ammissibile, cioè non minore di MIN_PRIORITY (priorità del processo dummy) e non maggiore del processo che ha attivato il processo invocando activate_p. Se la condizione è violata, abortisco il processo CORRENTE.
4) Controllo che il livello *liv* sia ammissibile cioè che abbia un valore pari o a LIV_UTENTE o a LIV_SISTEMA. Il livello del nuovo processo, inoltre, deve essere minore o uguale al livello di privilegio corrent della CPU, cioè mentre gira il processo chiamante.
5) Creiamo il nuovo processo.
6) Se la creazione del processo è fallita, restituisco un valore di errore 0xFFFFFFFF, altrimenti inserisco in lista pronti il processo appena creato, aumento la variabile globale processi e restituisco il suo id.
#sistema.s
7) Esecuzione iretq.

NB: Non farò tutti i passaggi con chiamata lato UTENTE, gate di interrupt ecc. Vado direttamente ad analizzare il vero e proprio contenuto della primitiva che sarebbero la a_primitiva e la c_primitiva. Se vuoi farti un'idea del giro che si fa invocando una primitiva, ti consiglio di dare un'occhiata alla dispensa intitolata "Le primitive" (pg. 2, 5, 6) del prof. Lettieri.

STATO DI UN PROCESSO



PROCESSO

- È un programma in esecuzione su dei dati in ingresso.
- L'esecuzione è la sequenza di stati che il sistema CPU + MEM attraversa eseguendo il programma sui dati di ingresso, dall'inizio alla fine.
  NB. Ricordati l'esempio della pizza del prof. Lettieri : la ricetta (programma/ sequenza istruzioni) è la stessa, ma la pizza sfornata (risultato) può cambiare dipendentemente da ciò che ordina il cliente (input).

PRIMITIVA TERMINATE_P

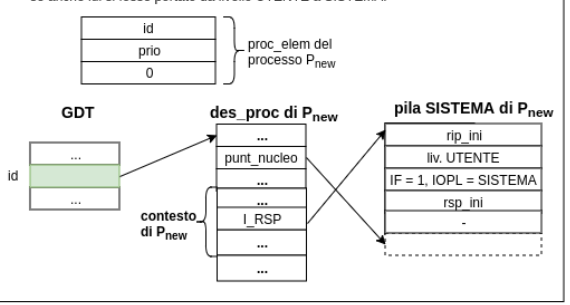
Il processo che ha invocato la terminate_p è quelo che è in esecuzione, quindi :
#sistema.s
1) Metto il valore *terminate_stack_end* in RSP.
2) Chiamo c_terminate_p
//sistema.cpp
3) rilasciamo tutte le strutture dati private associate al processo puntato da un proc_elem p (che punta ad *esecuzione*).
4) decremento la variabile globale *processi*.
5) dealloco *p*.
6) Chiamo la funzione schedulatore che farà puntare *esecuzione* al proc_elem in testa alla coda *pronti*.
#sistema.s
7) Si carica lo stato del processo puntato da esecuzione.
8) Esecuzione iretq.

DI COSA HA BISOGNO UN PROCESSO

- Descrittore di processo -> **des_proc**.
- Pila SISTEMA.
- MEM processo
  1) .text
  2) .data
  3) pila UTENTE
- CONTESTO del processo

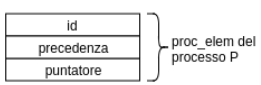
ATTIVAZIONE DI UN PROCESSO

- Un processo può attivarne un altro invocando una opportuna primitiva.
- Cosa serve allocare ed inizializzare?
  1) des_proc
  2) proc_elem
  3) pila SISTEMA
- Il processo appena creato deve essere portato in coda 'pronti' e si deve comportare come se anche lui si fosse portato da livello UTENTE a SISTEMA.



proc_elem

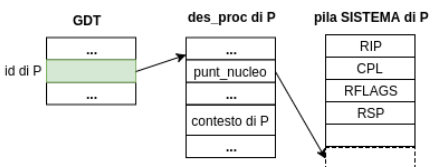
Serve per gestire le code di processi.



- id = id di P (cioè l'indice del descrittore del TSS nella GDT).
- precedenza = priorità di P.
- puntatore (ad un altro proc_elem). Serve per le code di processi

des_proc

Contiene il TSS di un processo P più altre informazioni (CR3, CPL e contesto[N_REG])



c_sem_wait

1)counter--
2) se counter < 0, inserisco il processo nella coda di processi (bloccati) del semaforo.
3) chiamo la funzione '*schedulatore*'.

c_sem_signal

1)counter++
2) se counter ≤ 0, rimuovo il primo processo dalla coda del semaforo e lo metto in coda '*pronti*'.
3) chiamo la funzione '*schedulatore*'.

MUTUA ESCLUSIONE

PROBLEMA : 2 processi modificano la stessa struttura dati. "Che cosa può mai andare storto?" cit. ennesimo studente malcapitato di ingegneria informatica.
Si può verificare :
1) inconsistenza struttura dati.
2) struttura dati non aggiornata come dovrebbe essere se le azioni fossero state eseguite sequenzialmente.

SOLUZIONE : abbiamo N coppie di processi (P_i, i = 1, ... , N) ed azioni (A_i, i = 1, ... , N).

Procedimento:
1) Scatola contiene 1 gettone.
2) Chiunque voglia compiere un'azione, deve prima prendere il gettone e rimetterlo nella scatola quando ha finito di fare quello che doveva fare.

Pseudocodice:

```
    natl mutex = sem_ini(1);
Pi : sem_wait(mutex);
    << Ai >>
    sem_signal(mutex);
```

FUNZIONI SEMAFORI

SINCRONIZZAZIONE

PROBLEMA : un processo PRODUTTORE deve tirar fuori un risultato. Il PRODUTTORE deve restare fermo e aspettare che il CONSUMATORE abbia prelevato il dato. Però anche il CONSUMATORE deve aspettare che il PRODUTTORE abbia tirato fuori il suo risultato e abbia conseguentemente aggiornato il buffer.
Si può verificare :
1) PRODUTTORE potrebbe decidere di aggiornare il contenuto del buffer prima che il CONSUMATORE abbia effettivamente prelevato il dato.
2) CONSUMATORE potrebbe prelevare prima che il PRODUTTORE abbia emesso un risultato sensato quindi rischierebbe di prelevare un dato privo di senso.

SOLUZIONE : 2 processi P_1 (PRODUTTORE) e P_2 (CONSUMATORE), ciascuno con la sua corrispettiva azione A_1 e A_2
Procedimento:
1) Scatola inizialmente vuota.
2) Dopo aver eseguito A_1, P_1 lascia un gettone nella scatola.
3) Prima di eseguire A_2, P_2 deve prendere un gettone dalla scatola.

In altre parole, se P_1 arriva per primo alla scatola, lascia il gettone e così P_2 potrà proseguire. Se P_2 arriva per primo alla scatola, la trova vuota e deve aspettare che P_1 metta il gettone per proseguire.

Pseudocodice:

Comportamento P1

```
global natl sync = sem_ini(0);
P1 : << A1 >>
     sem_signal(sync);
```

Comportamento P2

```
global natl sync = sem_ini(0);
P2 : sem_wait(sync);
     << A2 >>
```

SEMAFORO

Struttura dati che seve per risolvere problemi di concorrenza tra processi relativamente alle strutture dati (per non lasciarle in uno stato inconsistente, per tenerle debitamente aggiornate dopo una serie di modifiche, ecc.)

PRIMITIVA sem_ini

#sistema.s

```
carica_gate   TIPO_SI   a_sem_ini   LIV_UTENTE
...
   .extern c_sem_ini
a_sem_ini:
   call c_sem_ini
   iretq
```

//sistema.cpp

```
extern "C" natl c_sem_ini(int val)
{
natl i = alloca_sem();

if (i != 0xFFFFFFFF)
array_dess[i].counter = val;

return i;
}
```

des_sem

proc_elem dei processi bloccati in attesa che il semaforo sia liberato dal processo che lo sta attualmente occupando

CONSIDERAZIONI GENERALI SULLA SCRITTURA DELLE PRIMITIVE

- Le primitive girano ad interruzioni disabilitate, quindi le eccezioni si verificano se e solo se vi è un errore di programmazione lato SISTEMA.
- Aggiornare la variabile 'esecuzione' non implica il fatto di cambiare "istantaneamente" il processo in esecuzione. Aggiornare 'esecuzione' significa cambiare il proc_elem a cui punta 'esecuzione'. Una volta terminata la primitiva che si stava eseguendo, si ritornerà alla 'a_primitiva' (in **sistema.s**) che farà partire la carica_stato e la iretq. Solo allora si starà eseguendo il nuovo processo.

CONTROLLO PARAMETRI

Usiamo le funzioni '**cavallo_di_troia indBuffer**' e '**cavallo_di_troia2 indBuffer numByte**' per verificare che l'UTENTE non ci stia passando l'indirizzo di un buffer che, o parte da un indirizzo SISTEMA o da un indirizzo UTENTE di una zona di memoria UTENTE/PRIVATA appartenente ad un altro processo utente (primo controllo) oppure parte da un indirizzo ammissibile e poi la dimensione del buffer sfora la pagina dove sta, andando a finire in qualche parte di memoria in cui, da livello utente, non si può (e non si dovrebbe!) accedere.

PRIMITIVA

Funzione offerta dal kernel ai vari programmi UTENTE affinchè possano completare determinate azioni (in maniera controllata) che altrimenti non potrebbero fare.
Si noti che :

- i programmi UTENTE sono eseguiti con CPU a livello UTENTE.
- le strutture dati critiche per il funzionamento del sistema operativo, devono essere inaccessibili dal livello UTENTE, quindi le poniamo in M1.
- Il programmatore SISTEMA scrive primitive per conto dell'UTENTE, assicurandosi di manipolare correttamente le strutture dati del sistema.
- Il programmatore SISTEMA permette di invocare le sue primitive tramite gate della IDT (che innalzano il livello della CPU portandolo a livello SISTEMA).

COME SCRIVERE UNA NUOVA PRIMITIVA

Ad esempio vogliamo aggiungere una primitiva getid() che restituisce l'id del processo che l'ha invocata.

1) **ASSEGNARE IL TIPO DI INTERRUPT ALLA PRIMITIVA**
//costanti.h
```
#define TIPO_GETID 0x59
```

2) **CARICARE GATE IDT**
#sistema.s
```
carica_gate   TIPO_GETID   a_getid   LIV_UTENTE
```

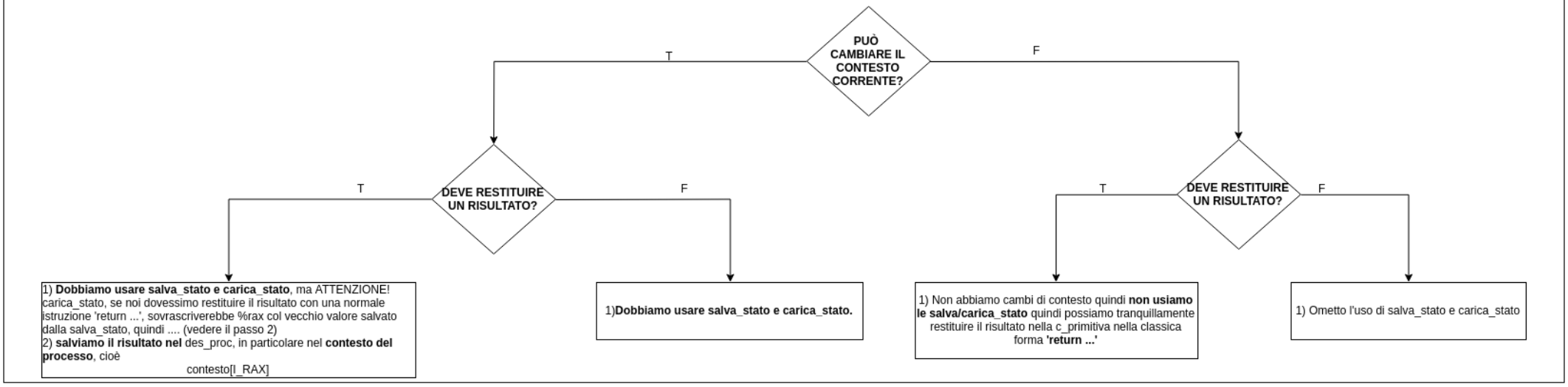
3) **SCRIVERE LA 'a_primitiva'**
#sistema.s
```
   .extern c_getid
a_getid:
   call c_getid
   iretq
```

4) **SCRIVERE LA 'c_primitiva'**
//sistema.cpp
```
extern "C" natl getid(){
   return esecuzione->id;
}
```

5) **DICHIARARE LA PRIMITIVA IN MODO CHE L'UTENTE POSSA USARLA**
//sys.h
```
extern "C" natl getid();
```

6) **SCRIVERE PROGRAMMA ASSEMBLY DI INTERFACCIA PER L'UTENTE**
#utente.s
```
.global getid
getid :
   int $TIPO_GETID
   ret
```

COME VA IMPOSTATA LA PRIMITIVA ?(RIGUARDO I DUBBI SULLE SALVA / CARICA_STATO E TIPO RESTITUITO DALLA PRIMITIVA)

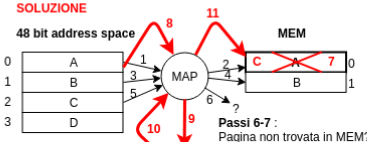


PROBLEMA 1 : LA RAM NON BASTA MAI

48 bit address space — **MEM**

SOLUZIONE

48 bit address space — **MEM**

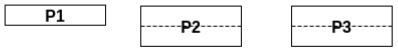


Passo 10 : Recupero dall'HDD, (sempre durante la **routine di PF**) la pagina che manca in MEM dallo swap, per poi mapparla in MEM

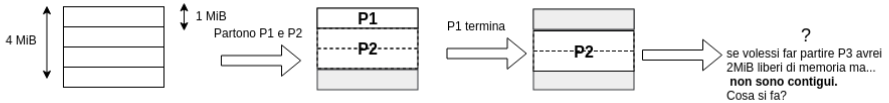
Passi 6-7 : Pagina non trovata in MEM? Parte la **ROUTINE DI PAGE FAULT**. Scelgo una **pagina vittima** dalla MEM e la metto nell'HDD (usando l'algoritmo LFU)

PROBLEMA 2 : FRAMMENTAZIONE MEMORIA

In questo esempio abbiamo una MEM da 4MiB.
Supporremo (banalmente) che codice e dati dei seguenti processi rientrino rispettivamente in :
P1 -> 1 MiB
P2 -> 2 MiB
P3 -> 2 MiB



MEM

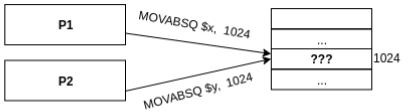


SOLUZIONE

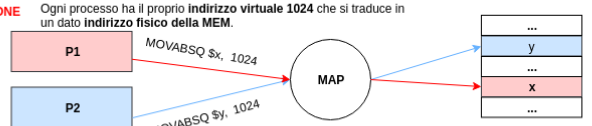
Mappiamo la memoria relativa a P3 spezzandola in due parti distinte da 1 MiB

PROBLEMA 3 : DUE PROGRAMMI SCRIVONO ALLO STESSO "INDIRIZZO"

Che succede se abbiamo due programmi che modificano entrambi qualcosa all'indirizzo 1024? Si sovrascrivono?
Supponete un caso (banalmente) estremo : un processo (chiamiamolo P1) memorizza il punteggio di un gioco. Un altro processo (P2) serve per un'applicazione che deve gestire i nostri dati bancari.
La soluzione consiste nel separare le MEM VIRTUALI di P1 e P2. Ogni processo ha la SUA MEM VIRTUALE.
All'**indirizzo 1024 di P1** memorizziamo il risultato del gioco, mentre all'**indirizzo 1024 di P2** memorizziamo il nostro conto in banca.



SOLUZIONE Ogni processo ha il proprio **indirizzo virtuale 1024** che si traduce in un dato **indirizzo fisico della MEM**.



NB Questo ci indica che la MEMORIA VIRTUALE può essere **usata per la multiprogrammazione**, dato che con essa possiamo far girare più processi tenendo i loro spazi di indirizzamento separati l'uno dall'altro.

PAGINAZIONE

Suddivisione dello spazio di indirizzamento e della MEM fisica in porzioni di memoria dette, rispettivamente **pagina** e **frame** (o **pagina fisica**), ciascuna grande

$2^{12}B = 4$ KiB

MEMORIA VIRTUALE

Meccanismo (**trasparente** al programmatore UTENTE) che attua una **separazione tra spazio di indirizzamento di un programma e la MEM fisica** installata sul calcolatore.

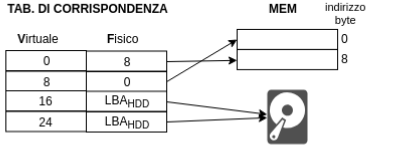
L'idea fondante è la seguente : usiamo la **memoria di swap** sull'hard disk (denominato d'ora in poi HDD) per ovviare al problema della mancanza di MEM principale. Salveremo dei dati in MEM e sull'HDD in modo da poter **mappare** i dati/codice del programma passo passo.

Per comprendere i motivi del perchè vada usato un simile meccanismo, bisogna mettersi al posto dei programmatori del secolo XX : i **computer erano costosi**, avevano **poca memoria** (e quella poca **memoria era costosa**..), dovevano essere **usati quanto più efficientemente** e **dovevano far girare i task anche se la MEM fisica installata sul calcolatore non era sufficiente** per tenere tutti i task in MEM.

TABELLA DI CORRISPONDENZA

Struttura dati che permette di memorizzare le informazioni di mapping delle pagine in MEM fisica.

TAB. DI CORRISPONDENZA — **MEM** — indirizzo byte



LBA (Logical Block Address) = [HND,CNH,CNL,SNR]

TLB (Translation Lookaside Buffer)

- Cache specifica della MMU che ricorda le traduzioni usate più di recente.
- Senza usare il TLB, per ogni accesso in MEM, ho altri 4 accessi in MEM (dato che devo consultare i 4 livelli delle tabelle di traduzione. Guardare lo schema "**TABELLE MULTILIVELLO**").
- Usando il TLB avrei un solo accesso molto veloce che mi traduce direttamente l'indirizzo in fisico.
- La TLB memorizza SOLO descrittori di livello 1.

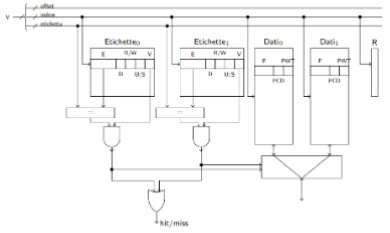


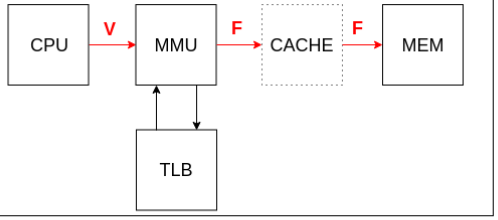
Figura 5: Un esempio di TLB a 2 vie.

ISTRUZIONI TLB

- **MOVQ %RAX, %CR3** invalida tutto il TLB.
  Si noti che invalidare il TLB non è qualcosa che va fatta a cuor leggero, dato che quello che si scrive in CR3 è l'indirizzo (fisico) della tabella di livello 4 del processo attualmente in esecuzione. Se si volesse svuotare il TLB senza cambiare processo, sarebbe utile prima salvare il contenuto di CR3 in RAX e poi mettere il contenuto di RAX in CR3, cioè
  MOVQ %CR3, %RAX
  MOVQ %RAX, %CR3
- **INVLPG 0xY** invalida la singola traduzione relativa ad Y (con 0xY indirizzo virtuale)

MMU (Memory Management Unit)

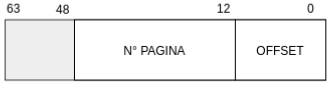
Unità che effettua la traduzione da indirizzi VIRTUALI (**V**) a FISICI(**F**)



INDIRIZZO VIRTUALE (V)

Indirizzo che sta **'a monte' della MMU.**

RICEVE



INDIRIZZO FISICO (F)

Indirizzo **'a valle' della MMU.**

INVIA

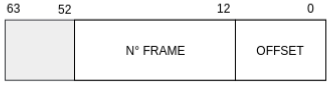


TABELLE MULTILIVELLO

- La MEMORIA VIRTUALE di ogni processo è di 2^{48} Byte.
- Ogni pagina è grande 2^{12} Byte

MEMORIA VIRTUALE di ogni processo contiene $\frac{2^{48}}{2^{12}} = 2^{36}$ **pagine**

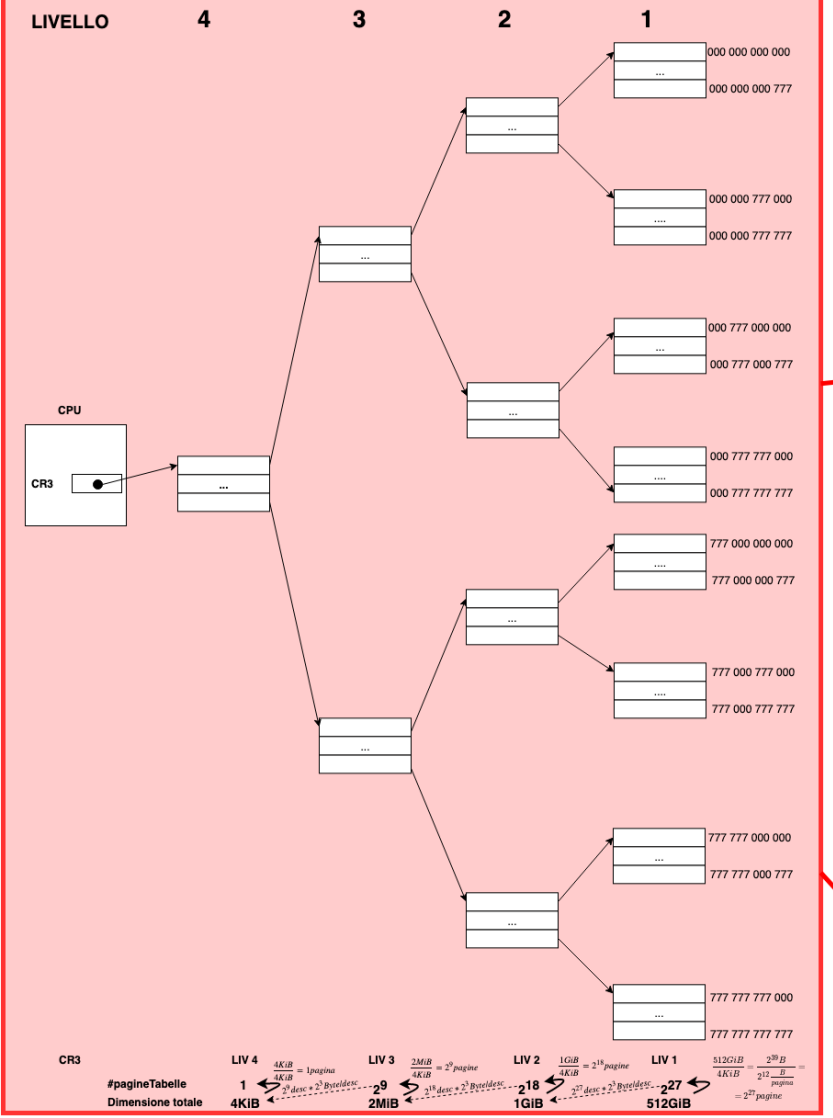
- Per ogni entrata della tabella dobbiamo memorizzare almeno 7 bit di utilità (livello 1 : *D, A, PCD, PWT, CPL, R/W, P* o livelli 2,3 e 4 : *PS, A, U/S, R/W, P*) e 40 bit per il N° FRAME. Abbiamo quindi 47 bit < 6 Byte. Dato che vogliamo, per comodità e gestibilità, che ogni entrata della tabella di traduzione sia una potenza di 2, la minima potenza di 2 che è maggiore di 6 Byte è 8 Byte. Usiamo quindi **8 Byte per ogni descrittore di pagina virtuale**. Quindi, ogni singolo processo potrebbe avere bisogno di

  2^{36} pagine * 2^3 Byte/pagina = 2^{39} Byte = **512GiB**
- ATTENZIONE : Spezziamo la tabella di traduzione in pagine in modo da poter caricare solo ciò che ci serve e non tutta la tabella di traduzione perchè sarebbe impossibile con le MEM fisicamente installate sui nostri computer. Dobbiamo partire da TUTTA la tabella di livello 1. Questa pesa 512GiB in totale ed andremo a spezzarla in 512GiB / 4KiB/pag = 2^{27} pagine => 2^{27} descrittori di tabella di livello 2 => 2^{27} descrittori$_{liv2}$ * 8 byte/descrittore = 1 GiB per il livello 2.

  Dividiamo questo 1GiB in pagine da 4KiB quindi otteniamo 2^{18} pagine => 2^{18} descrittori$_{liv3}$ => 2^{18} descrittori$_{liv3}$ * 8 Byte/descrittore = 2MiB.

  Dividiamo questi 2MiB in pagine da 4KiB quindi otteniamo 2^9pagine => 2^9 descrittori$_{liv4}$=> 2^9descrittori$_{liv4}$* 8 Byte/descrittore = 4KiB.

  Questa suddivisione implementazione ci costerebbe 4KiB + 2MiB + 1GiB + 512 GiB ~ 513GiB/processo
- Dato il punto precedente, usiamo la notazione in base 8 per gli indirizzi, dato che un numero in base 8 mi rappresenta 3 cifre in base 2, quindi mi bastano 3 cifre in base 8 (3*3 bit = 9 bit) per selezionare una singola pagina di una tabella.



BIT DI UTILITÀ NEL DESCRITTORE DI PAGINA VIRTUALE

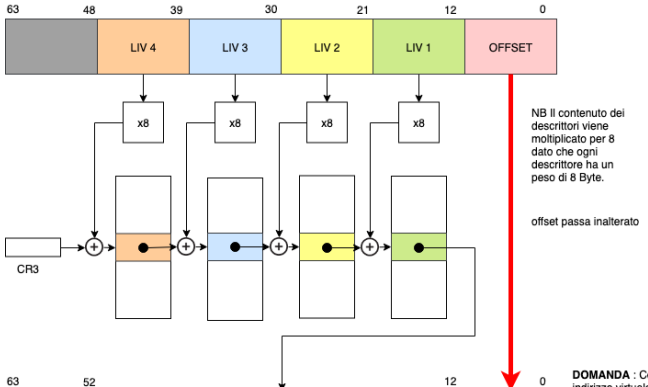
	D	A	PCD	PWT	CPL	R/W	P
DESCRITTORE PAGINA (LIV. 1)							

DESCRITTORE TABELLA (LIV. 2, 3, 4)	PS		A			U/S	R/W	P

A cosa servono i vari bit?
1) **D**(irty) : serve per vedere se una pagina è stata modificata (se scelgo una pagina vittima e questa non è stata modificata, è inutile che io proceda con lo swap-out (passaggio da MEM ad HDD).
2) **A**(ccess) : bit di accesso. Mi indica se "ultimamente" la pagina è stata consultata.
3) **PCD** (Page Cache Disable) : se vale 1, indica che la cache deve essere disabilitata per ogni accesso (READ / WRITE) solo su questa pagina.
4) **PWT** (Page Write Through) : se vale 1, specifica che la cache dovrà usare politica Write Through per tutte le WRITE effettuate su questa pagina.
5) **CPL** (Current Privilege Level) - **U/S** (User / System) : dice se la pagina appartiene a livello SISTEMA od UTENTE.
6) **R/W** (Read / Write) : vieta(0) o permette(1) le operazioni di scrittura su questa pagina virtuale.
7) **P**(resent) : indica se la pagina è presente(1) o meno(0) in MEM.
8) **PS** (Page Size) : usata per alterare le dimensioni della pagina (vedasi il riquadro "TRADUZIONE DA V->F"). Si noti che è concesso mettere ad 1 il bit PS **solo su un descrittore di tabella incontrato lungo il percorso di traduzione**.

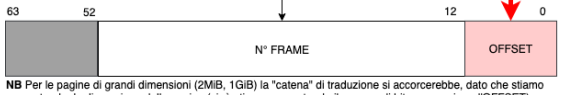
Cosa fa la MMU mentre stiamo traducendo l'indirizzo virtuale?
Durante la traduzione, la MMU fa anche altre cose oltre a smistare tra le tabelle. In particolare :
1) **aggiorna il bit A** per gli Accessi. (serve alla **routine di PF** per effettuare delle statistiche che serviranno per decidere quale pagina scegliere come vittima).
2) **controlla tutti i bit R/W** lungo il cammino : una WRITE è permessa SE E SOLO SE tutti e 4 i bit R/W impongono W.
3) **controlla tutti i bit U/S** lungo il cammino : una pagina è di livello UTENTE SE E SOLO SE tutti i bit U/S impongono UTENTE.
4) **in caso di scrittura, pone ad 1 il bit D** nella tabella di livello 1 (cioè nel **descrittore di pagina**).

TRADUZIONE V->F



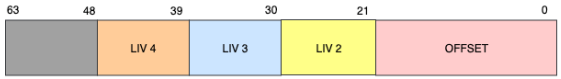
NB Il contenuto dei descrittori viene moltiplicato per 8 dato che ogni descrittore ha un peso di 8 Byte.

offset passa inalterato



NB Per le pagine di grandi dimensioni (2MiB, 1GiB) la "catena" di traduzione si accorcerebbe, dato che stiamo aumentando la dimensione della pagina (cioè stiamo aumentando il numero di bit necessari per l'OFFSET).

DOMANDA : Come mai si parte da un indirizzo virtuale di 48 bit e si giunge ad un indirizzo fisico di 52 bit?
Nel descrittore di pagina, dopo la catena di traduzioni da liv 4 a liv 1, preleviamo il campo F di 40 bit. A questo ci concateniamo i 12 bit di OFFSET => 40 bit + 12 bit = 52 bit

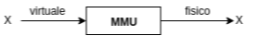


Possiamo ridimensionare le pagine in questa maniera : solo nei descrittori di tabella (pagine di livello 4, 3 e 2) abbiamo un bit **PS** (**Page Size**) che indica se vogliamo fermare la traduzione al livello cui appartiene il descrittore di tabella.

	Liv 4	LIV 3	LIV 2	LIV 1	OFFSET	DIM_PAGINA
Bit PS	0	0	0	-	[0,11]	4KiB
	0	0	1	-	[0,20]	2MiB
	0	1	0	-	[0,29]	1GiB

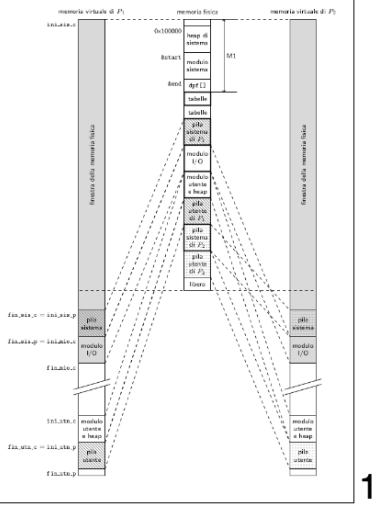
FINESTRA DI MEMORIA (FM)

- Porzione dello spazio di indirizzamento di un processo, tramite la quale vediamo la MEM fisica.
- Le tabelle vanno memorizzate in MEM fisica (assieme alle pagine dei processi, al codice ed alle strutture dati del SISTEMA).
- Concettualmente, tutte le tabelle farebbero parte di M1, però devono essere allocate dinamicamente, quindi le salviamo in M2.
- Il **SISTEMA** dovrà quindi accedere a tutti i frame di M2 (per consultare le tabelle) e deve poter accedere (per sua stessa natura di "SISTEMA") anche ad M1, quindi **deve poter accedere a tutta la MEM fisica**.
- D'altro canto, dobbiamo impedire ai processi UTENTE di accedere ad M1, se non per le parti che contengono le loro pagine, quindi la MMU si dovrebbe "disattivare" ogni volta che la CPU passa a livello SISTEMA.
- La MMU però è sempre attiva quindi dobbiamo fare in modo che la traduzione sia una traduzione identità, cioè gli indirizzi virtuali che passiamo alla MMU sono tradotti in se stessi.



- Così facendo permettiamo alle routine di SISTEMA di usare indirizzi fisici proprio come se la MMU non ci fosse, indipendentemente da quale processo sia attualmente in esecuzione. È come se avessimo creato, nella parte alta della MEM VIRTUALE di ogni processo, una "finestra" che ci permette di vedere la MEM fisica così come è effettivamente.
- La finestra FM è accessibile solo da livello SISTEMA.
- La finestra FM viene creata prima di attivare la MEM VIRTUALE.
- All'avvio, la CPU parte con la MEM. VIRTUALE disattivata ed esegue la routine di inizializzazione che può allocare ed inizializzare tutte le tabelle necessarie alla definizione della finestra FM. In seguito la MEM. VIRTUALE può essere attivata. Dato che, dopo l'attivazione della MEM. VIRTUALE stiamo ancora a livello SISTEMA, la routine di inizializzazione può continuare ad usare gli indirizzi fisici come se la MEM. VIRTUALE non fosse attivata.
  Quando si passa a livello UTENTE, queste traduzioni diventano inaccessibili e ridiventano accessibili ogni volta che si torna a livello SISTEMA.
- Senza la FM sarebbe molto complicato per la routine PF compiere le seguenti azioni:
  1) **percorrere l'albero di traduzione di un processo.** Tutti gli indirizzi delle tabelle (es. il contenuto di CR3, il contenuto del campo F dei singoli descrittori di tabella / pagina) sono fisici. Per poter accedere alle tabelle, la routine PF deve disporre degli indirizzi "virtuali" che vengano tradotti in quegli indirizzi fisici. La FM risolve questo problema perchè gli indirizzi virtuali sono effettivamente gli indirizzi fisici da usare.
  2) **eseguire i trasferimenti MEM <-> SWAP durante i caricamenti e scaricamenti**. Se tali trasferimenti non sono effettuati in Bus Mastering, la routine PF deve poter leggere/scrivere nel frame F. Per farlo, la routine PF ha bisogno di un qualche indirizzo virtuale che venga tradotto negli indirizzi fisici del frame.

ESEMPIO DI MEM. VIRTUALE CON DUE PROCESSI
(fonte : Implementazione della memoria virtuale, pg 2, Auth: G. Lettieri)



DESCRITTORE DI PAGINA VIRTUALE

63–52	12–7	6	5	4	3	2	1	0	
	F	...	D	A	PCD	PWT	CPL	R/W	P	DESCRITTORE PAGINA (LIV. 1)
	F	... PS		A			U/S	R/W	P	DESCRITTORE TABELLA (LIV. 2, 3, 4)

Considerazioni sui campi del descrittore di pagina virtuale :

- **P**(resent) :
  - marca le **pagine virtuali** alle quali un dato processo **non deve accedere**.
  - segnala la presenza / assenza della pagina dalla MEM fisica.
- **D**(irty) :
  - segnala se la pagina è stata modificata.
  - Perchè segnalare se la pagina è stata modificata quando possiamo fare swap-out per ogni volta che è necessario? Operare con l'HDD è TROPPO COSTOSO. Infatti, quando siamo costretti a chiamare la routine di PF, perdiamo molto tempo (dal punto di vista della CPU) prima di riprendere effettivamente l'esecuzione da dove era stata interrotta.
  - D è **presente solo per le pagine di livello 1.** Perchè? Le pagine di livello 4, 3 e 2 mi indicano solo la struttura della tabella di traduzione, quindi non verranno modificate. Al più le pagine di livello 4 resteranno sempre in MEM fisica mentre quelle di livello 3 e 2 verranno scambiate tra MEM fisica e SWAP dipendentemente dagli accessi nel corso del tempo. Le uniche pagine che verranno effettivamente modificate saranno le pagine dei processi, quindi i descrittori di livello 1 mi segnaleranno se la pagina che indirizzano è stata modificata.
- **A**(ccess) :
  - serve alla routine di PF per aggiornare il campo "contatore" (vedere "DESCRITTORE DI PAGINA FISICA") del descrittore di FRAME relativo al FRAME contenente la pagina virtuale.
  - L'aggiornamento di "contatore" non tiene esattamente conto di quanti accessi sono stati fatti nel corso del tempo. È solo un campionamento : ogni tot di tempo, statisticamente, vediamo i bit A, aggiorniamo il campo "contatore" al corrispondente descrittore di FRAME se A=1, e li azzeriamo.

DESCRITTORE DI PAGINA FISICA

È una struttura dati usata solo dalla routine di PF. Serve a:

LIVELLO	RESIDENTE	V	B	CONTATORE	ID
ϵ **[-1,4]** dove : **a) -1** -> frame libero **b) 0** -> pagina **c) sia i = 1,..,4 ->** frame contiene una pagina relativa ad una tabella di livello i, i ϵ [1,..4].	**rimpiazzabile o meno**	**n° pagina / prossimo frame libero**	**n° blocco**	**statistiche di accesso**	**identificatore di processo**

A cosa servono questi descrittori?

- Vedere dove si trovano le pagine nell'area di swap.
- Sapere quali frame sono liberi e quali no.
- Avere un conteggio accurato per gli accessi alle singole pagine, in modo da sapere quale pagina deve essere scelta come vittima.

ROUTINE DI PAGE FAULT (detta anche ROUTINE DI PF)

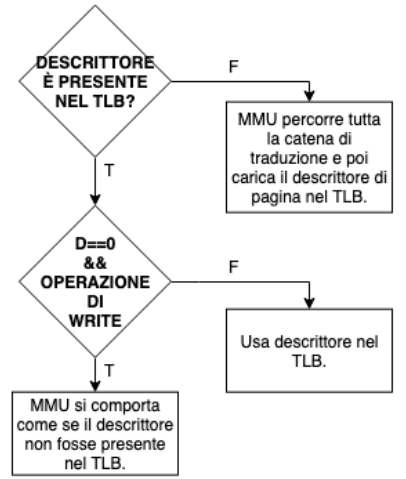
1) CPU inizia un nuovo accesso in MEM.
2) MMU intercetta l'operazione e comincia la traduzione.
3) Supponiamo che si verifichi un'eccezione di PF lungo il tragitto per via di un descrittore di pagina/tabella con bit P = 0;
4) Sia V l'indirizzo (VIRTUALE) che ha generato il fault.
5) V viene memorizzato in CR2.
6) Parte la routine di PF.
7) La routine di PF ripercorre il tragitto che la MMU ha seguito quando si è verificata l'eccezione di PF, dato che conosciamo V ma non sappiamo in quale punto ci siamo interrotti nella traduzione. La routine si ferma quando legge il primo descrittore di tabella/pagina con P = 0;
8) Per individuare la posizione della pagina V nello swap, legge il campo F del descrittore di pagina/tabella di V (inutilizzabile per memorizzare un numero di frame, dato che non era presente in MEM fisica) che è stato usato per memorizzare la posizione del blocco B nello swap.
9) Cerca un frame libero.
  9.1) se c'è già un frame libero (cioè pagine_libere != 0) ho già finito, altrimenti ....
  Se non c'è un frame libero :
  9.2) Routine PF seleziona una pagina vittima controllando tutti i frame.
    ATTENZIONE : vi sono **determinate pagine** che **non devono essere scelte come vittime**.
    Vediamo quali:
    a) non possiamo svuotare un frame F che ha, nel suo descrittore di frame, il campo **residente = 1** (traduzione CALCOLATORI -> LINGUA ITALIANA : non possiamo eliminare pagine che devono restare sempre in MEM fisica. Un esempio? Buffer che serve ad un driver, che non si può permettere di lanciare page fault in quanto (il driver) fa parte del codice SISTEMA).
    b) Non possiamo svuotare un frame F contenente una **porzione di tabella che la routine ha attraversato in precedenza per giungere al descrittore che ha causato il fault.**
    c) Non dobbiamo scegliere una porzione di **tabella di livello i** (con i ϵ **[1,4]**) che **descrive ancora almeno una pagina virtuale presente in MEM fisica** (cioè, non dobbiamo eliminare porzioni di tabella i-esima se abbiamo almeno un descrittore che in essa ha P=1).
    Fa partire la routine delle statistiche stat() che controlla i bit A dei descrittori di pagina/tabella e aggiorna i relativi contatori nei descrittori di frame (relativi ai frame in cui sono posti), aggiorna i bit A(ponendoli a 0) e va alla ricerca del descrittore di frame con campo contatore minimo tra tutti. A parità di contatori minimi, si sceglie come vittima quella con livello minore. Sia la V' la pagina vittima corrispondente al frame F.
  9.3) SE D=1 => Ricopia V' nello swap leggendo il campo B del descrittore di frame F, altrimenti elimina la pagina dalla MEM fisica e basta.
  9.4) Nella tabella di corrispondenza, poniamo P = 0 nel descrittore di pagina relativa a V' e, dato che la pagina non sta più in MEM fisica, uso il campo F del descrittore di tabella/pagina per memorizzare il blocco B' dal quale reperire la pagina che stiamo portando dalla MEM fisica alla memoria di swap.
  9.5) Il frame F così liberato, può essere usato per memorizzare V.
10) La routine carica V da B (letto dal campo F del descrittore di tabella/pagina) nel frame F ed aggiorna il campo B del descrittore di frame ponendolo uguale a F del descrittore di tabella/pagina. (leggere la giustificazione di quest'operazione al punto 8)
11) Nella tabella di corrispondenza, poniamo P = 1 nel descrittore di pagina relativo a V e poniamo F = numero di frame F scelto al passo 9. Viene modificato anche il descrittore di frame di F ponendo il campo B = blocco dove stava.
12) Termina ritornando all'indirizzo salvato in pila al sollevamento del fault.

PROBLEMI TRA ROUTINE PF E TLB

1) RIMPIAZZAMENTO PAGINA
La traduzione nel TLB non rispecchierebbe più la realtà => invalido SOLO la traduzione relativa alla pagina rimpiazzata.

2) MMU NON AGGIORNA I BIT 'A' FINO A QUANDO IL DESCRITTORE RESTA NEL TLB
TLB falsa le statistiche di accesso, quindi si usa la routine di PF in modo che si invalidi tutto il TLB.

3) CPU esegue una READ su una pagina => memorizzo traduzione di quella pagina nel TLB. Se la CPU dovesse mai dover effettuare una WRITE, sempre su quella pagina e la MMU trova "comodamente" la traduzione già nel TLB. Se quella pagina, dopo essere stata modificata senza aggiornare il bit D, dovesse essere scelta per far posto ad un'altra pagina che serve in quel momento alla CPU, essa non verrebbe aggiornata nell'HDD perchè la MMU vede che il bit D = 0.

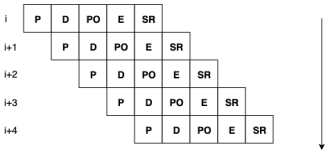


PAGINAZIONE SU DOMANDA
Usiamo la MEM come una cache di pagine. Invece di caricare interi processi dallo swap, carichiamo solo le pagine che i processi richiedono

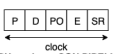
PIPELINE

Ogni e-istruzione deve attraversare 5 fasi :

Tali fasi sono eseguite da parti distinte della CPU => mentre l'e-istruzione i-esima si trova nella fase di, per esempio, Decodifica (**D**), il circuito relativo al prelievo delle istruzioni (**P**) sarebbe usabile, ragion per cui la usiamo per prelevare l'istruzione 'i+1'-esima. Nel caso ottimo, possiamo avere il seguente andamento temporale.

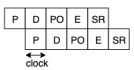


Ciascuna e-istruzione, quindi, attraversa successivamente tutti gli stadi del pipeline e, ad ogni istante, ogni stadio si occupa di una e-istruzione diversa. In una CPU semplice la sequenza di fetch-decode-execute veniva fatta da un'unica (grande) rete combinatoria RC. Se volessimo passare alla soluzione delle CPU moderne, occorrerebbe aggiungere dei registri a valle di ciascuna (piccola) RC che si occupa di una sola fase. Perchè si ha quindi l'aumento di prestazioni con la soluzione più moderna? Bisogna parlare di velocità del clock della CPU.
Da cosa dipende il clock? Esso dipende dal più lungo percorso tra un registro e l'altro. Mettiamo che il ritardo delle RC sia T per la RC della CPU "vecchio stile" e T' (<T) per le singole reti combinatorie P, D, PO, E ed SR. Allora, si avrà :
- CPU SENZA PIPELINE :



$T_{regToReg\_w/o\_PIPELINE} = T_{prop} + T + T_{setup}$

- CPU moderna CON PIPELINE :



con RCi = {P, D, PO, E, SR}

$T_{regToReg\_w\_PIPELINE} = T_{prop} + T' + T_{setup} < T_{regToReg\_w/o\_PIPELINE}$

Col metodo della pipeline, ogni singola istruzione ci impiegherà di più ad attraversare tutti gli stadi rispetto al caso della CPU "vecchio stile" senza pipeline con un'unica grande RC, ma il vantaggio sta nel fatto che dopo la prima fase dell'e-istruzione i-esima, la i+1-esima può partire, se va tutto bene, dal ciclo di clock successivo. Però, come per ogni cosa, abbiamo dei problemi :
1) nel set di istruzioni INTEL, la lunghezza delle istruzioni è molto variabile quindi bisogna decodificarla per capire quanto è lunga => prelievo e decodifica vanno fatte insieme.... A meno che noi non usiamo le **e-istruzioni** o **istruzioni elementari** che hanno un formato uniforme (guardare riquadro **"RISC vs CISC"**).
2) Prelievo degli operandi e scrittura dei risultato possono essere entrambe operazioni operazioni in memoria? Possono essere svolte insieme? No. Ci sono solo due operazioni (**load** e **store**) che operano sulla MEM mentre
tutte le altre operano sui registri.
3) Il clock riusciamo a dividerlo per 5 **solo nel caso ottimo**.

RISC vs CISC

- Come detto nel punto 1 del riquadro "PIPELINE", per i processori INTEL, non sarebbe possibile suddividere le fasi di prelievo e decodifica poichè le istruzioni hanno lunghezza variabile e quindi non possiamo sapere quanto è grande l'istruzione finchè non l'abbiamo letta.
- Ogni istruzione originaria, cioè di tipo complesso o appartenente al set di istruzioni CISC (**Complex Instruction Set Computer**), viene tradotta in una o più e-istruzioni, cioè istruzioni elementari o appartenente al set di istruzioni RISC (**Reduced Instruction Set Computer**).
Cosa fanno i processori INTEL? Via HW le istruzioni di tipo CISC passano per un modulo di traduzione che le trasforma in una sequenza di istruzioni elementari o e-istruzioni.

- Esempio :
ISTRUZIONE CISC:
MOVQ 1000(%RBX, %RCX, 8), %RAX

E-ISTRUZIONE/I RISC:
MOV $0, %R0
ADD %R0, %RCX, %R0 # metto il contenuto di RCX in R0
SHL %R0, %R0, $3 # moltiplico per 8 il contenuto di R0 dato che devo avere, per ora R0*8
# (si ricordi che 8 = 2^3)
ADD %R0, %RBX, %R0 # è come se avessi ottenuto l'indirizzo (%RBX, %RCX, 8)
LD 1000(%R0) , %RAX # mettiamo 1000(%RBX, %RCX, 8)
 # R0 è solo un registro d'appoggio NON visibile al programmatore

- Dopo aver fatto questo esempio però uno può pensare "Ma con tutte queste benedette e-istruzioni che sono uscite,
dove sta il guadagno effettivo?". Ebbene, la gran parte delle istruzioni CISC della INTEL, si traducono in 1-2 e-istruzioni
quindi, data la struttura della pipeline, riusciamo a guadagnarci lo stesso. (esempio numerico?)

ALEE

Situazioni in cui non è possibile iniziare una nuova e-istruzione ad ogni ciclo di clock. Supponiamo, ad esempio, che alcune e-istruzioni siano già in alcuni stadi della pipeline e che la prossima e-istruzione da iniziare sia la i-esima.

TIPI ALEE

ALEE STRUTTURALI

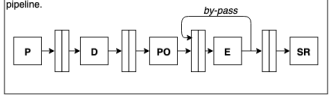
Si presentano quando la e-istruzione i-esima dovrebbe usare una risorsa (ALU, registri, ..) già utilizzata da un'altra e-istruzione già nel pipeline.

ALEE SUI DATI

Si presentano quando la e-istruzione i-esima avrebbe bisogno di un dato non ancora prodotto da una e-istruzione precedente.

ADD **R1**, R2, R3
SUB R4, R5, **R1**

La soluzione è possibile ottenerla introducendo un circuito di by-pass pilotato dal circuito che controlla il flusso della pipeline.
L'istruzione SUB vuole leggere R1, ma più che il registro, le interessa il suo contenuto che sarebbe il risultato del circuito di ESECUZIONE della pipeline.



ALEE SUL CONTROLLO

Si presentano quando una delle e-istruzioni nel pipeline è una e-istruzione di salto condizionale di cui non è stata ancora calcolata la condizione di salto.

JZ R1, avanti
ADD R2, R3, R4
avanti : SUB R5, R6, R7

Dopo il prelievo della e-istruzione JZ, non è immediatamente disponibile l'indirizzo della prossima e-istruzione da prelevare, dato che la e-istruzione JZ dovrà essere decodificata ed eseguita prima che si possa decidere se è necessario prelevare la e-istruzione ADD (condizione non verificata) o la e-istruzione SUB (condizione verificata)

Il prelievo delle e-istruzioni successive dovrà essere sospeso fino a quando la condizione di salto non è stata risolta. Questo tipo di stallo può essere ridotto cercando di prevedere il risultato della condizione di salto. e continuando a prelevare e-istruzioni assumendo che la previsione sia corretta. Ovviamente la previsione può non essere corretta, ergo il pipeline deve essere organizzato in modo tale da disfare i risultati di una previsione errata.
Se fosse questo il caso, allora è sufficiente :
1) marcate come errate tutte le e-istruzioni che occupano gli stadi del pipeline precedenti allo stadio attualmente occupato dalla e-istruzione di salto.
2) prevedere che lo stato della CPU non possa essere modificato da e-istruzioni marcate come errate.
3) ricominciare a prelevare le e-istruzioni dall'indirizzo corretto.

ARCHITETTURA INTERNA CPU

Abbiamo detto che la CPU fa sempre le stesse cose, cioè fetch, decode, execute e, fra un'istruzione e la successiva, controlla se ci sono richieste di interruzione. Per un lungo periodo di tempo, nel secolo XX, le prestazioni delle CPU aumentavano in maniera brute force perchè si aumentava solo la frequenza alla quale esse lavoravano. Quello che oggi fanno i progettisti è rimpicciolire le componenti (transistor) della CPU in modo da poter aumentare il numero di unità dedicate all'esecuzione delle varie istruzioni.
Le moderne CPU attuano degli accorgimenti che servono per eseguire più velocemente le istruzioni che "gli sono date in pasto". Ciò avviene in due modi :
1) l'esecuzione di ogni singola istruzione viene scomposta in più fasi, ciascuna delle quali richiede un'unità distinta della CPU. Quando un'istruzione ha terminato una fase, libera un'unità della CPU che può quindi essere usata per far avanzare un'altra istruzione. In questo modo, la CPU viene vista come una **catena di montaggio** o **pipeline.**
2) L'esecuzione di ogni istruzione viene fatta iniziare appena possibile, anche se in un ordine diverso da quello stabilito dal programma, senza alterare il risultato finale che si avrebbe avuto eseguendo sequenzialmente le istruzioni.

ESECUZIONE SPECULATIVA

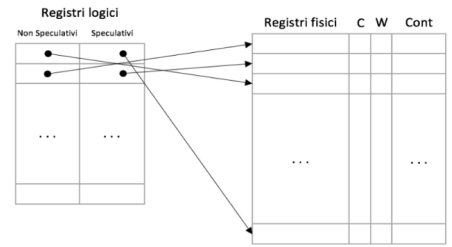
PREMESSA : questa parte è teoricamente completa ma, per una migliore comprensione del funzionamento dell'esecuzione speculativa, rimando ai due esercizi di esempio a pag 221 del libro "ARCHITETTURA DEI CALCOLATORI Vol. 2°.
La tecnica di rinomina dei registri (riquadro "ELIMINAZIONE DIPENDENZE SUI NOMI") affiancata alla tecnica di predizione dei salti, ci permette anche di aggirare le limitazioni dovute alle dipendenza sul controllo.
Possiamo eseguire le e-istruzioni dipendenti da e-istruzioni di salto non ancora risolte purchè i risultati di queste vengano salvati in dei registri temporanei e trasferiti nei registri reali solo dopo aver verificato che quelle e-istruzioni andavano eseguite.
Per attuare questa tecnica occorre introdurre un nuovo stadio detto **ritiro delle istruzioni**. Il completamento di queste e-istruzioni comporta la scrittura dei risultati in dei registri temporanei : tali risultati diverranno effettivi solo se la e-istruzione che li ha prodotti supera lo stadio di ritiro.
Lo stadio introdotto fa uso del **ROB** (**ReOrder Buffer**) che è una **coda di descrittori di e-istruzioni**.
Cosa memorizza il ROB?

- **Tipo** e-istruzione (operativa o controllo?).
- e-istruzione **completata o meno**.
- Per le e-istruzioni di controllo, memorizza l'**esito previsto per il salto**.

Quando un'e-istruzione viene emessa, essa viene messa in fondo alla coda del ROB mentre il ritiro viene fatto dalla testa del ROB.

Per poter rendere effettivi o annullare gli effetti delle e-istruzioni completate, viene estesa la struttura dati che fa corrispondere i registri logici ai registri fisici. In particolare, ogni registro logico possiede una doppia corrispondenza con registri fisici :

- **Non speculativa** : puntatore al registro fisico in cui è contenuto il valore dell'ultima e-istruzione ritirata dal ROB che lo aveva come destinatario.
- **Speculativa** : indica quale registro fisico contiene (o conterrà) il valore corrente.



Cosa avviene quando :

- **un'e-istruzione *operativa* viene *emessa*** (operazioni avvengono solo sui reg. fisici speculativi) :
1) per ogni registro logico sorgente, corrispondenza esistente tra registro logico e registro fisico e incremento di Cont relativo al registro fisico.
2) per il registro logico destinatario, nuova corrispondenza con un registro fisico libero (=> C=1, W=1). Si noti che il campo C del vecchio registro fisico diviene 0 o resta ad 1 a seconda che vi sia una corrispondenza non speculativa con tale registro fisico.
- **un'e-istruzione *operativa* viene *completata***(solo reg. fisici speculativi) :
1) Decremento di Cont per ogni registro fisico sorgente.
2) Azzeramento di W relativo al registro fisico destinatario.
- **un'e-istruzione *operativa* viene *ritirata***: in relazione al registro logico destinatario :
1) C = 0 per il reg. fisico destinatarionon speculativo. (per quello speculativo C resta ad 1).
2) il valore del puntatore speculativo viene ricopiato nel puntatore non speculativo.
- **un'e-istruzione di *controllo***:
1)**se *emessa* o *completata***, ci si comporta come nel caso corrispondente delle e-istruzioni operative relativo al solo registro fisico sorgente.
2) se ***ritirata*** non viene effettuata alcuna azione (previsione di salto corretta) oppure viene svuotato il ROB (previsione di salto errata).

Lo **svuotamento del ROB** comporta le seguenti azioni :

- **annullamento** effetti di una e-istruzione **operativa non completata**:
1) decremento Cont per ogni reg. fisico sorgente.
2) W = 0 per il registro fisico destinatario.
3) C = 0 per il registro fisico destinatario (C potrebbe essere già a 0 prima di questa operazione).
4) per il registro logico destinatario,
speculativo = non_speculativo
- **annullamento** effetti di una e-istruzione **operativa completata**:
1) C = 0 per il registro fisico destinatario (C potrebbe essere già a 0 prima di questa operazione).
2) per il registro logico destinatario,
speculativo = non_speculativo.
- **annullamento** effetti di una e-istruzione **di controllo**:
1) vengono effettuate le azioni previste nel caso delle e-istruzioni operative per il solo registro fisico sorgente.

Il significato dei valori C, W e Cont resta invariato rispetto a quanto visto nel riquadro "SCHEMA INTERNO DELLA CPU".
Dobbiamo fare però alcune precisazioni :

- W = 1 => registro fisico è bloccato.
- C = 1 => registro fisico è selezionato da un puntatore.
- registro fisico è **libero <=> C = 0**, **W = 0**, **Cont = 0**.
- registro fisico è **bloccato <=> W = 1**.
- registro fisico è **sbloccato <=> W = 0**.

Per le ragioni viste, servono più registri fisici che logici. Per essere precisi
numRegFisici = 2*numRegLogici
Quando si emette una e-istruzione, è sufficiente guardare i valori speculativi : se la e-istruzione è arrivata in cima al ROB, si copia il valore speculativo in quello non speculativo altrimenti, se in cima al ROB arriva un'e-istruzione di controllo e si scopre che la predizione era sbagliata, tutti i valori speculativi sono sbagliati e possiamo rimediare alla situazione copiando gli ultimi valori (che erano corretti) nei valori speculativi.

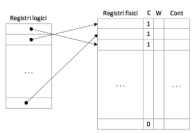
ELIMINAZIONE DIPENDENZE SUI NOMI

La tecnica che usiamo è la cosiddetta tecnica di **rinomina dei registri**.
Prendiamo ad esempio la seguente sequenza di e-istruzioni :

ADD R1, R2, R3 # r2 come src1
ADD R2, R4, R5 # r2 come dest
SUB R6, R2, R7

Che si tradurrebbe in
ADD R1, R2, R3
ADD R20, R4, R5
SUB R6, R20, R7

Apportando la modifica R2 -> R20 non abbiamo alterato la semantica del programma, ma siamo riusciti ad eliminare una dipendenza sui nomi. Tale cosa la fa la CPU in autonomia aggiungendo un componente che fa la rinomina dei registri. Distinguiamo tra **registri logici**(quelli che compaiono nel flusso del programma) e **registri fisici**(quelli su cui si fanno davvero i conti). Quindi oltri ai campi W, Cont e Registro aggiungiamo il campo C che è ad 1 <=> se vi è una corrispondenza tra un registro logico ed il registro fisico F a cui il bit C appartiene.



Un registro fisico F si considera libero se :

- C = 0, nessuna corrispondenza di F con alcun registro logico.
- W = 0, nessuna e-istruzione emessa deve scrivere in F;
- Cont = 0, nessuna e-istruzione emessa deve leggere da F.

Emissione della e-istruzione :

- incremento Cont per i registri fisici corrispondenti ai registri logici *src* coinvolti.
- settaggio dei bit C e W per il registro fisico corrispondente al registro logico *dest*
- resettaggio dei bit C e W per i registri fisici corrispondenti ai registri destinatari della vecchia corrispondenza.

Completamento di una e-istruzione :

- decremento Cont per ogni registro fisico *src*.
- W = 0 per il registro fisico *dest*

ESECUZIONE FUORI ORDINE

Nella tecnica del PIPELINE le e-istruzioni vengono eseguite (1) una per volta e (2) nell'ordine previsto dal programma. Per migliorare le prestazioni nelle situazioni di stallo, è possibile eliminare questi due vincoli, cercando di eseguire più e-istruzioni per volta e anche non consecutive.

Esempio :
MUL R1, R2, R3
SUB R4, R1, R2
ADD R5, R2, R6
ADD R7, R3, R8

La SUB non può andare avanti finchè la MUL non abbia scritto il risultato in R1.
Si noti però che la successiva e-istruzione ADD R5, R2, R6 non ha bisogno dei risultati prodotti dalla MUL o dalla SUB ed inoltre la sua esecuzione necessita di risorse HW (sommatore fra interi) diverse da quelle richieste dalla e-istruzione MUL. Possiamo quindi cominciare ad eseguire la ADD in parallelo con la MUL che (molto probabilmente) richiederà più tempo rispetto alla ADD per la sua esecuzione.Così facendo noi eseguiamo prima la ADD, poi la MUL ed infine la SUB e la ADD. Se la CPU disponesse di più risorse HW (due sommatori) potremmo eseguire le due ADD in fondo e poi eseguire MUL e SUB.

Più in generale, possiamo eseguire una qualunque e-istruzione nel primo momento possibile in cui vi siano risorse libere per poterlo eseguire, tenendo conto delle **dipendenze**

DIPENDENZE SUI DATI

L'e-istruzione *j* dipende sui dati dalla e-istruzione *i* (*i<j*) se *i* produce un risultato usato da *j* oppure usato da una e-istruzione k da cui j dipende per i dati

Esempio:
ADD **R1**, R2, R3
SUB R4, R5,**R1**

DIPENDENZE SUI NOMI

L'e-istruzione *j* dipende per i nomi da una precedente e-istruzione *i* se (1) entrambe scrivono sullo stesso registro o locazione di memoria (istruzioni che scrivono sullo stesso registro) oppure (2) se *j* scrive su un registro letto da *i*. (sequenzialità lettura ed in seguito scrittura)

/*CASO 1 : R1 dest e nuovamente dest*/
ADD **R1**, R2, R3
ADD **R1**, R4, R5

//CASO 2 : R2 sorg e poi dest
ADD R1, **R2**, R3
ADD **R2**, R4, R5

DIPENDENZE SUL CONTROLLO

L'e-istruzione *j* dipende sul controllo da una precedente e-istruzione di salto condizionale *i* se l'e-istruzione *j* può essere eseguita o meno in base al risultato della condizione in *i*.

i \ j	R	W
R	NESSUNA DIPENDENZA	DIP. DATI
W	DIP. NOMI	DIP. NOMI

SCHEMA **DIPENDENZE** DATI-NOMI (con **i < j**)

SCHEMA INTERNO DELLA CPU

Uno schema che permette di eseguire le e-istruzioni fuori ordine, ma tenendo conto delle dipendenze, consiste nell'organizzare i tre stadi di PO, E ed SR di una pipeline nella maniera schematizzata di seguito, nella quale è stato aggiunto il nuovo stadio di EMISSIONE. Diremo che un'operazione è completata quando ha terminato la fase di SR.

Per ogni **Registro**, ci ricordiamo un paio di cose :

- se c'è una e-istruzione di quelle in coda (cioè in stazione di prenotazione) che vorrebbero scriverci (**W**);
- quante e-istruzioni vorrebbero leggere quel registro (**Cont**)

Una e-istruzione può essere emessa se rispetta le dipendenze sui nomi e sul controllo, ma non necessariamente sui dati.

REGOLE DI EMISSIONE
E-ISTRUZIONE : guardiamo dest e vediamo il suo flag W :
se W = 1 => dipendenza sui dati, perchè la e-istruzione vuole leggere il risultato di una e-istruzione che deve ancora essere messa in coda. Questa la emettiamo ugualmente ed andrà a finire in una stazione di prenotazione idonea, in attesa che i suoi operandi siano pronti. Quando questi saranno pronti, verranno prelevati e si procederà con l'esecuzione dell'istruzione.

Quando una e-istruzione del tipo ***op dest, srcI, src2*** viene **emessa** :

- i contatori *Cont* associati a *src1* ed *src2* sono incrementati*;*
- il flag W associato a *dest* viene posto ad 1.

Quando una e-istruzione del tipo ***op dest, srcI, src2*** viene **completata**:

- i Cont associati a *src1* ed *src2* sono decrementati;
- il flag W associato a *dest* viene posto a 0.