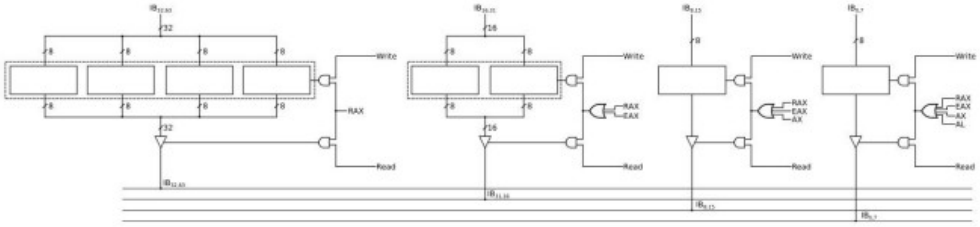
Lezione 19 Z64 5

giovedì 9 novembre 2023 15:00

Riassumendo l'indirizzamento nel processore : per l'indirizzo il processore esegue e soddisfa l'equazione :

Disp(Rbase (64 bit), Rindice (64 bit), scala (tipi primitivi 8,16,32,64) , spiazzamento 32 bit)

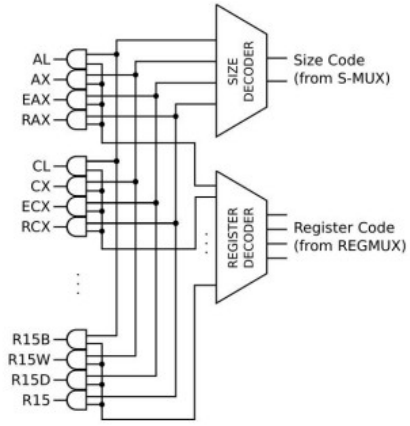
Vediamo ora come funzionano i registri virtuali (scrivo e leggo solo i ff che mi servono) senza usarli tutti :



In dettaglio :

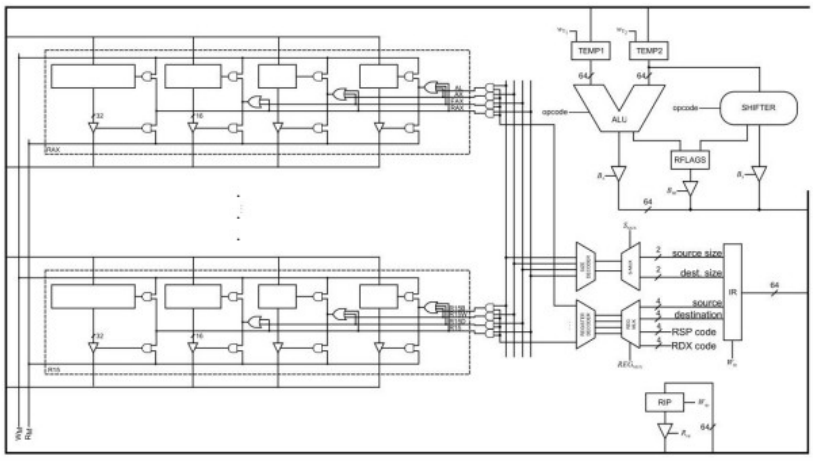
1. Si parte dai più significativi a sx verso i meno significativo a dx.
2. Si propaga un segnale W/R se e solo serve a tutti (OR a tutti i registri) .
3. I buffer-tri-state servono a fare da interruttore , ed in base alla porzione che devo usare del registro, e con una AND decide se scrivere o no .
4. Quindi porte AND e OR abilitano la lettura/scrittura nei vari blocchi (blocchi da 8 bit-> 1 byte) : specificano la porzione virtuale di quello fisico .
5. Cosi aggiungo 4 segnali in più in input alla rete sequenziale .
6. Per ovviare a ciò , vado a vedere il campo MODE : data la codifica usando un Decoder , genero in uscita una sola uscita che serve per generare uno dei quattro segnali di controllo W/R , andando cosi a scrivere nella parte corretta del registro virtualizzato (8,16,32,64) sia della sorgente , sia della destinazione:

i.



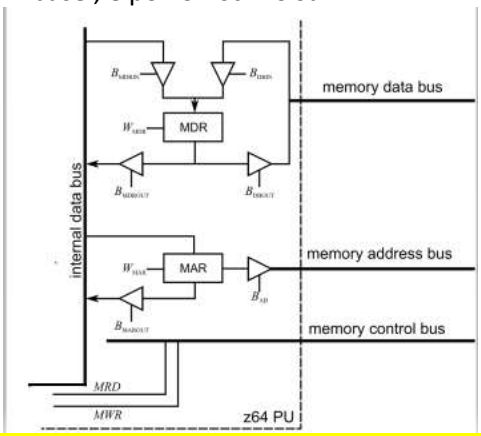
ii. Gli input di questi decoder li prendo da due Mux : prendono la codifica di R/M e SS , mentre per la destinazione R/M e DS , arrivando al seguente circuito :

1)

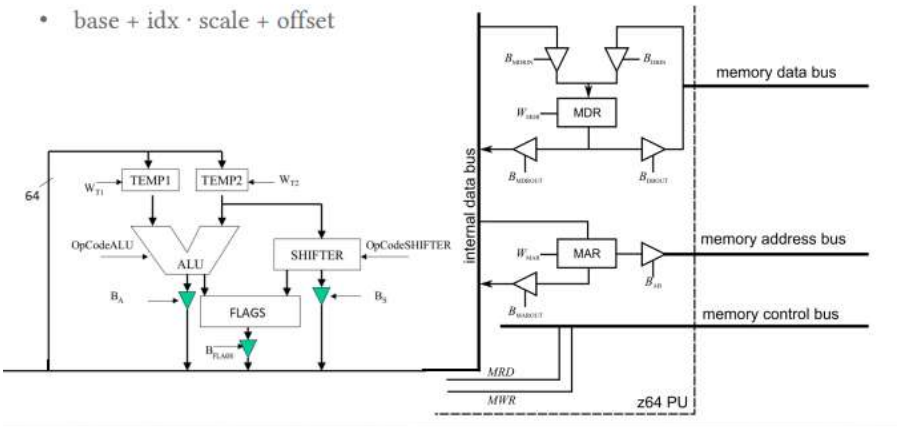


2) <u>Cosi dall'Alu a posto di 6 bit di controllo ,ne usiamo 1 (sia per sorgente e destinazione) : ma in realtà non basta :</u> Ingressi aggiuntivi da IR . Può capitare che IR non sappia tutte le info, in quanto alcune istruzione (costanti) siano già memorizzati in registri appositi . Come le generiamo le costanti ? Uso il registro FLAGS , il quale al suo interno c'è un valore 1 (posizione 1), il quale può essere collegato in vari modi per ottenere la costante.

Tornando alla modalità di indirizzamento : prima la moltiplicazione e poi 2 addizioni : si potrebbe usare gli adder per fare queste operazioni, ma aumenta la complessità e la circuiteria quindi optiamo per lo Shifter : il risultato della moltiplicazione viene salvato dove? In un registro general purpose? Assolutamente no : (sovrascrivo dati con altri dati temporanei) , **quindi uso un registro tampone** (invisibile al programmatore) , ma facendo cosi aggiungo 64 FF .. Nel nostro caso si usa il registro MAR , che viene usato come registro accumulatore : indice * scala , poi lo passo ad alu -> indice*scala + base , e poi lo riscrivo su MAR :



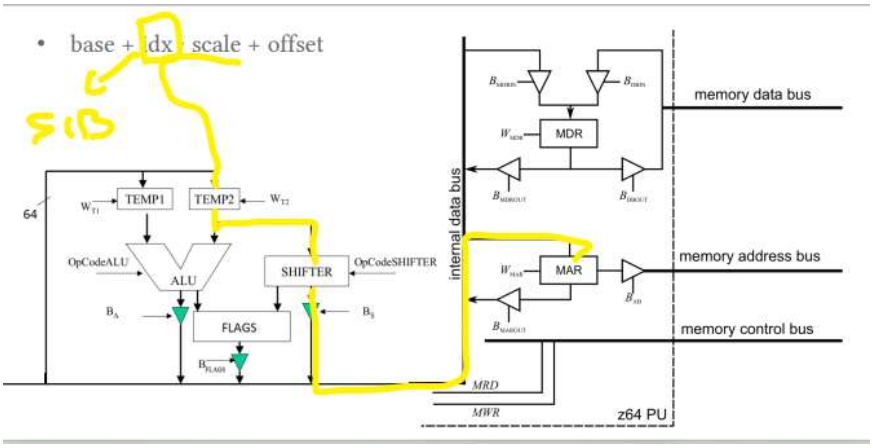
Possiamo usare questo circuito ? Assolutamente no , perché il flusso dati su data bus interno è uni-direzionale . Quindi devo aggiungere un collegamento diretto tra MAR e il resto della circuiteria , arrivando a questo circuito :



1. Il contenuto del Registo index (sotto porzione di SIB) viene fatto passare sul data bus interno ,

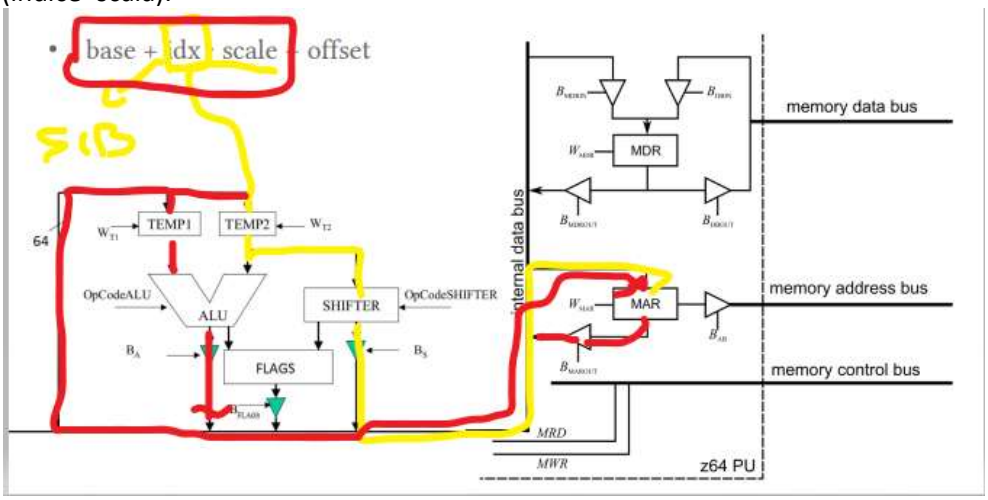
fino ad arrivare a Temp2. Poi lo mando allo shifter il quale ha come op-code la quantità da shiftare (1,2,4,8) , abilito uscita dello shifter , viaggia su internal data bus, arriva a MAR , ed abilito scrittura : su MAR ho indice*scala:

a.



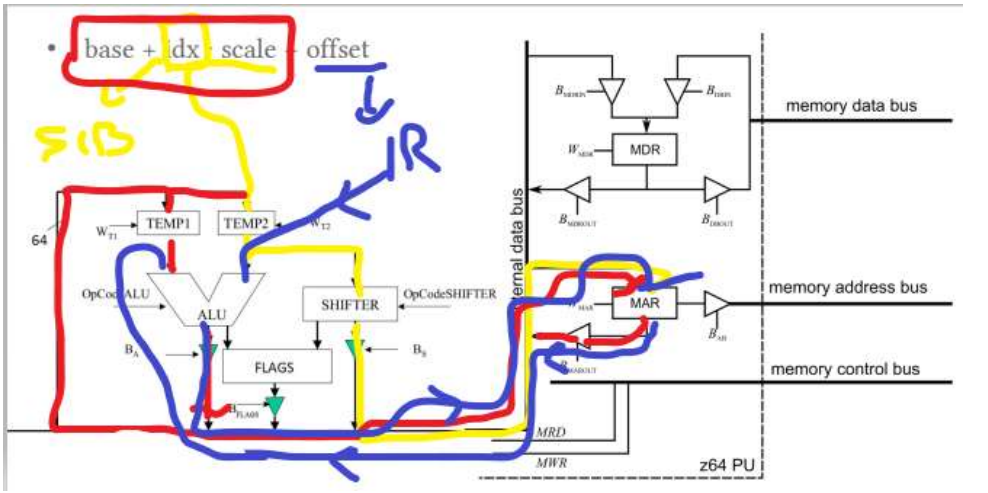
2. Dopo aver scritto su MAR, abilito la scrittura, quindi buffer-tri-state , dati viaggiano sul bus interno dati ed arrivano fino a Temp1 (indice*scala) , mentre su Temp2 c'è l'indirizzo base , poi vado nell'alu che viene settata per fare la somma , abilito buffer-tri-state e lo faccio viaggiare sul bus interno dati , lo scrivo su MAR , abilitando il segnale di scrittura, avendo cosi base+(indice*scala):

a.



3. Vediamo ora per lo spiazzamento : 4 byte meno significativi della codifica istruzione (32 bit meno significativi) da IR : me li salvo in un registro temporaneo , poi abilito la lettura da MAR , lo salvo in altro registro temporaneo , configuro alu per fare somma , abilito buffer-tri-state (dati viaggiano su bus dati interno) , abilito scrittura su MAR , avendo cosi base + (indice*scala) + offset :

a.



Questo processo è possibile visto che il processore è multi -ciclo : ogni istruzione in un colpo di clock differente : ogni micro-istruzione viene eseguita ogni colpo di clock : scrivi index dentro temp2 , moltiplica usando shifter e scrivi dentro MAR, terza copia MAR dentro un registro temporaneo .

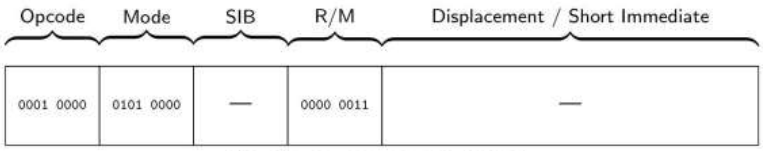
Andiamo ora a vedere alcune istruzioni assembly :

1. Movb %al,%bl
   a. Spostamento dati tra registri virtuali . Istruzione solo sugli 8 bit meno significativi
2. Movw %ax,(%rdi)
   a. %rdi è operando in memoria . Quindi muovo in %ax l'indirizzo base in cui vado a fare scrittura di due byte .
3. Movl (%rsi), %eax
   a. Dest reg virtuale a 32 bit, src a 64 bit . Quindi vado ad indirizzo puntato da %rsi, prendo 4 byte (long-word), li porto nel processore e poi li scrivo nella virtualizzazione a 32 di RAX (nel nostro caso virtualizzato a 32 -> %eax)
4. Movq (%rsi,%rcx,8), %rax
   a. %rsi è la base , %rcx indice , 8 scala
   b. Copia in %rax il contenuto degli 8 byte della memoria il cui indirizzo iniziale è calcolato come RSI+RCX*8
5. Subl %eax,%edx
   a. Sottrazione tra due registri a 32 bit .
   b. Faccio b-a
   c. Lo scrivo nel registro destinazione.
   d. Usa il complemento a 2 per fare la somma
6. Add $d,%al
   a. Da notare che i"$" sta a simboleggiare una costante , altrimenti prendo il byte in memoria alla cella "d"

Andiamo a vedere ora in dettaglio come funzionano le funzioni (con tutti i campi). **Ogni bit in hex rappresenta la codifica dei 4 bit di ogni segnale** :

1. Movw %ax,%bx

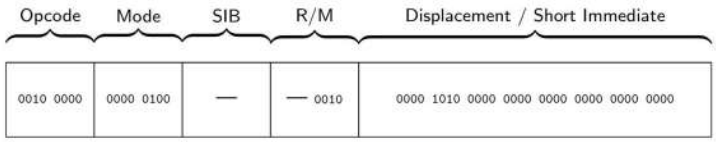
a.



10 50 00 03 00 00 00 00

   b. **Dove la codifica in esadecimale rappresenta i vari segnali di controllo espressi nell'istruzione**
   c. In questo tipo di indirizzamento nonsi ha nessun accesso in memoria, nessuna costante coinvolta (displacement tutto 0) e gli operandi sono a 2 byte (double word).
2. Addb $0xa,%al
   a. Utilizza una costante numerica come operando
   b. Non utilizza spiazzamento
   c. Se prefisso "0x" si parla di costanti esadecimali

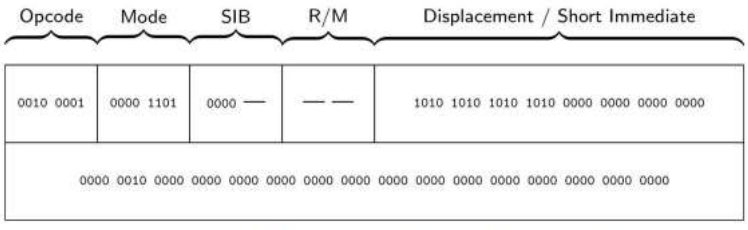
i.



20 04 00 02 0A 00 00 00

3. Subb $0x2,0xAAAA
   a. Costante numerica come operando
   b. Presente uno spiazzamento

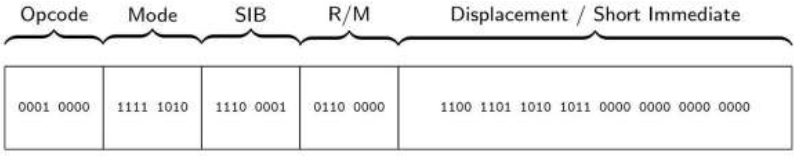
c.



21 0D 00 00 AA AA 00 00 02 00 00 00 00 00 00 00

d. AAAA è indirizzo in memoria espresso in esadecimale, quindi va nello spiazzamento.

4. Movq 0xabcd(%rsi,%rcx,4),%rax
   a. Full addressing (base(%rsi), indice, scala e spiazzamento)
   b.



10 F8 E1 60 CD AB 00 00

5. **Attenzione all'ordine dei bit nello spiazzamento : è invertito (da primo secondo) a (Secondo primo) : dovuto alla codifica che usiamo : little endian:**

…	…	…	…	**AB**	**CD**	…	…	…	…

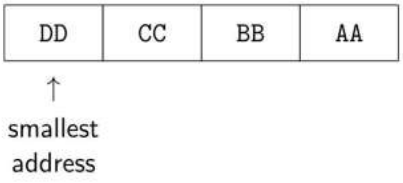
   a. Se lavoro con le long word (16 bit/ 2 byte) ok , ma se volessi lavorare con singole word(8 bit/1 byte) ?? Non possiamo lavorarci , ed in questo caso usiamo il **troncamento** : forzo da 16 a 8 , ma perdo il byte più significativo . Quindi il processore prende solo cella contenente "CD" e per trovare cella che contiene "AB" deve andare in avanti di n byte. Questo procedimento si ripete in base alla grandezza del dato:

…	…	…	…	**CD**	**AB**	…	…	…	…

In generale : si parla di memory endianness : **ovvero di come le architetture codificano i dati :** si parla di **Big Endian e di Little Endian** : **nel primo caso le codifiche dei dati avvengono da più significativo a quello meno significativo (vedendo lo stesso indirizzo di memoria) , mentre nella secondo si parte dal bit meno significativo fino a quello più significativo** . I processori intel usando questa codifica . Vediamo un esempio :

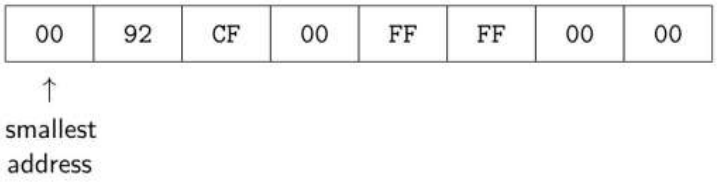
Il valore `0xAABBCCDD` è posto nel layout di memoria come segue:

a.



Prendiamo ad esempio `0x00cf9200` e `0x0000ffff`

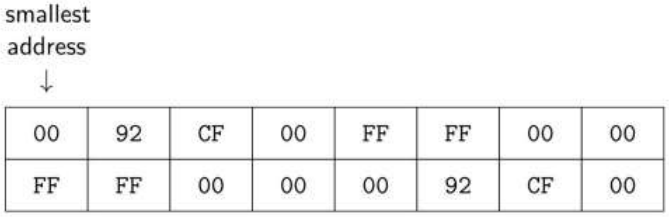
b.



   i. Stesso ordine : vengono scambiate solo i byte , ma non l'ordine

Prendiamo ad esempio `0x00cf9200`, `0x0000ffff` e `0x00cf92000000ffff`

c.



   d. **Riassumendo il tutto : questa codifica ed in generale qualunque codifica (istruzioni , dati ecc ecc) è sconosciuta al processore , quindi il programmatore deve fornirla : La codifica delle istruzioni dipende dal contesto .**