LA SAPIENZA · UNIVERSITÀ DEGLI STUDI DI ROMA

FACOLTÀ DI INGEGNERIA INFORMATICA, GESTIONALE E STATISTICA SEDE DI ROMA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA ED AUTOMATICA

TESI DI LAUREA TRIENNALE

PROGETTO DI UN ASSEMBLATORE PER IL PROCESSORE Z64

RELATORE: PROF. BRUNO CICIANI

CORRELATORE: ALESSANDRO PELLEGRINI

PRESENTATA DA:

DANIELE MORIGGI

ANNO ACCADEMICO 2015/2016

Sommario

1. Abstract

Il progetto prevede lo studio del processore z64 sottoinsieme dell'ISA Intel x86_64. Saranno discusse all'interno di tale trattato, l'architettura, il design e il set di istruzioni dello z64. Si partirà dalla struttura interna del processore, in cui si analizzeranno le interconnessioni tra i principali elementi che compongono lo z64 (ALU, SHIFTER, banco dei registri); successivamente si analizzerà in particolare il formato istruzione a 64 bit che la CPU è in grado di gestire e infine si passerà alla definizione formale di tutte le microoperazioni di controllo necessarie per l'esecuzione di ogni singola istruzione supportata dallo z64.

Il progetto terminerà con la programmazione Java di un assemblatore, componente fondamentale di un programma per simulare il comportamento di tale processore. Partendo dall'output del parser, che analizza il programma assembly scritto dall'utente, tale assemblatore avrà il compito di trasformare le istruzioni scritte in linguaggio di più alto livello in un linguaggio in formato macchina definendo in particolare le microoperazioni generate dal programma assembly.

2. Architettura del processore Z64

1.1. Introduzione

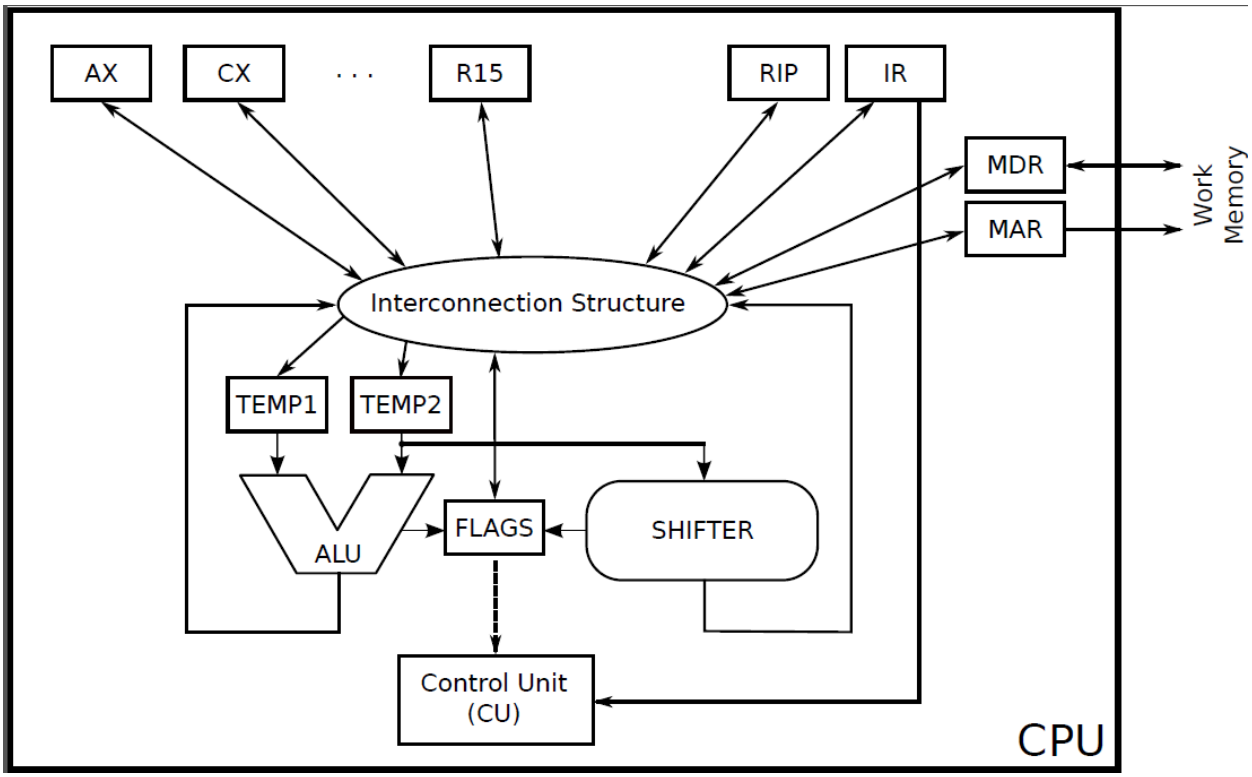
L'architettura del processore z64 è sostanzialmente divisa in due blocchi separati che cooperano tra di loro. Il primo blocco chiamato SCO (o Sottosistema di Controllo) ha il compito di supervisionare e gestire l'esecuzione di ogni funzione supportata dalla CPU, il secondo blocco è lo SCA (o Sottosistema di Calcolo), che gestito dallo SCO provvede ad eseguire le operazioni elementari.

Lo SCO è realizzato mediante una rete sequenziale e tutti i segnali di controllo che deve generare, sono dettati dal programma assembly scritto dall'utente.

Lo SCA, invece, è costituito dalle parti elementari del processore, ad esempio l'ALU, lo SHIFTER e i registri interni.

La relazione di complessità che lega i due blocchi è inversamente proporzionale, infatti, al crescere della complessità implementativa dello SCO, minore sarà quella dello SCA e viceversa.

In Figura1 è possibile osservare uno schema a blocchi relativa all'organizzazione della CPU, in cui le frecce continue rappresentano il collegamento diretto tra i singoli componenti, mentre la freccia tratteggiata indica che la Control Unit può gestire il funzionamento dei vari componenti in funzione del valore dei dati contenuti nelle variabili di stato, operando tramite segnali di controllo.



1.2.Registri interni

Per quanto riguarda i registri interni del processore, questi saranno a 64 bit e saranno accessibili al programmatore sia interamente, sfruttando tutti i 64 bit, sia in taglie inferiori: la metà (32 bit meno importanti), un quarto (16 bit meno importanti), un ottavo (8 bit meno importanti).

Questa implementazione dei registri, permette al programmatore di scrivere software in cui si usano dati a 8, 16, 32 e 64 bi.

Lo Z64 dispone di un insieme registri visibili al programmatore tra cui i *general purpose registers* e gli *special registers* che contengono i flag di stato (status flag), visibili dal programmatore attraverso specifiche istruzioni interpretate dalla CPU.

In aggiunta ci sono registri che invece non sono visibili al programmatore, ma che sono utilizzabili solo dallo SCO che ne modifica e legge il contenuto per supportare la corretta esecuzione delle istruzioni. Questi sono soprattutto usati per interpretare istruzioni macchina, per tenere traccia di risultati parziali di operazioni svolte per esempio dall'ALU.

- **Registri Visibili**

Lo Z64 è equipaggiato di 16 registri general purpose a 64 bit (tabella 1) ognuno dei quali ha un codice mnemonico per accedere allo stesso per scrivere o leggere dati. La regola per accedere ai diversi tagli di registri, è cambiare, per lo stresso registro, il nome di accesso. Ad esempio, come mostrato in figura1), per accedere ai 32 bit meno significativi del registro *rax*, basta utilizzare il nome *eax*; lo stesso vale per i 16 bit meno significativi, in cui basta utilizzare il nome *ax* e infine *al* è il nome per accedere agli 8 bit meno significativi.

I registri ai quali è possibile accedere su tutte le taglie sono: %rax, %rbx, %rcx, %rdx e da %r8 a %r15, mentre i registri %rbp, %rsi, %rdi, %rsp e %rip possono essere acceduti solo fino alla base word esclusa, cioè i registri %bl, %sl, %dl, %sl, %il non esistono.

TABELLA 1

Mnemonic	Codifica	Uso Comune
rax	0000	Registro di accumulazione
rcx	0001	Counter Register
rdx	0010	Data Register
rbx	0011	Base Register
rsp	0100	Stack Pointer
rbp	0101	Base Pointer
rsi	0110	Source Register
rdi	0111	Destination Register
r8	1000	General-Purpose Register
r9	1001	General-Purpose Register
r10	1010	General-Purpose Register
r11	1011	General-Purpose Register
r12	1100	General-Purpose Register
r13	1101	General-Purpose Register
r14	1110	General-Purpose Register
r15	1111	General-Purpose Register

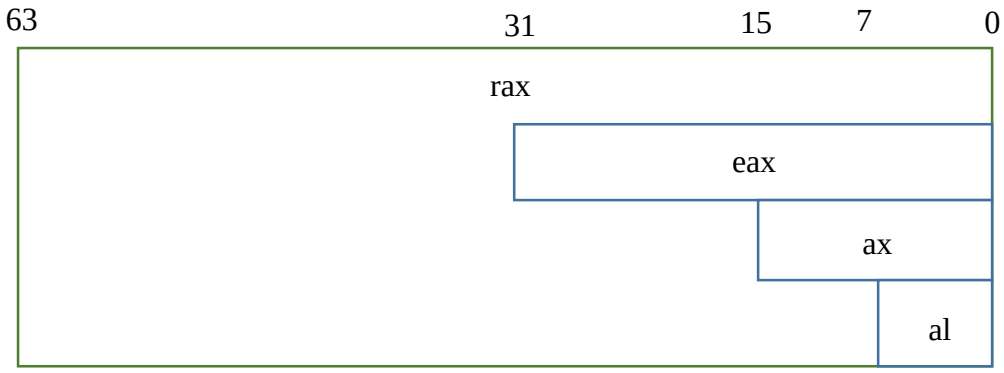


FIGURA 1

- **Registri nascosti al programmatore**

Nella CPU Z64, ci sono due registri addizionali, che non sono direttamente accessibili dal programmatore.

Il primo è l'*instruction register*, a 64 bit, tiene traccia dell'indirizzo di memoria della prossima istruzione da eseguire, permettendo alla CPU di farne il *fetch* facilmente dalla memoria di lavoro e mantenere traccia dell'evoluzione del programma. È chiamato *%rip* e viene utilizzato sempre in forma integrale.

All'utente non è concesso modificarlo direttamente, ma ad ogni esecuzione di un'istruzione viene modificato indirettamente. In particolar modo, normali istruzioni incrementano il valore di *%rip* della grandezza dell'istruzione stessa.

È possibile, attraverso due istruzioni, forzare il contenuto di tale registro. Queste istruzioni sono *call* e *jump*. Lo scopo di queste due operazioni è controllare il flusso del programma, chiamando subroutine o saltando a differenti porzioni di codice.

Il secondo registro nascosto al programmatore è il *%flags register*, questo registra attributi speciali di ogni esecuzioni di istruzione. L'utente può leggere il contenuto di tale registri, ma non può modificarne il contenuto a suo piacimento, non può quindi salvare dati in tale registro. È comunque possibile alterare il valore di tali bit tramite istruzioni specifiche. Gli *status bit* sono modificati dal risultato di ogni operazione aritmetica, per tenere traccia dell'ultima operazione eseguita. I *control flag* invece possono essere settati/resettati a piacimento dall'utente tramite i rispettivi comandi *setX/clrX*.

Il *%flags register* è costituito da 16 bit

Gli aggiornamenti del control flag sono così determinati:

- Il *Carry flag (CF)* è settato, vale 1, se l'addizione di due numeri causa un riporto oltre il bit più significativo. Questo bit è utile quando si utilizzano operandi non segnati, per vedere se l'operazione ha causato overflow. In alcune architetture è presente il *borrow flag (BF)*, lo Z64 non ce l'ha, ma il CF può essere utilizzato al suo posto, infatti quando CF vale 0, il valore dell'evenutale borrow flag è 1 (non c'è stato bisogno del prestito).

  Esempio:

  0001 - 0001 = 0001 + 1111 = 0000

  Quest'operazione imposta il CF a 1 e il BF vale 0.

- Il *Parity flag (PF)* indica se il numero di bit settati nel byte più importante del risultato è pari o disparo. Il PF dello Z64 è un registro a parità pari, infatti è settato a 1 se il numero di 1 trovati nel byte preso in considerazione è pari. Il valore del PF è calcolato tramite somma XOR tra i bit, restituendo 0 per parità pari e 1 per parità dispari. Successivamente questo risultato è poi negato e messo dentro il PF. Questo flag è particolarmente utile, ad esempio, per controllare errori di trasmissioni o per calcolare CRC.
- Lo *Zero flag (ZF)* vale 1 se tutti i bit dell'ultimo risultato calcolato sono zeri. È calcolato facento l'OR di tutti i bit del risultato e poi negando il risultato. Questo flag è di solito

utilizzato per verificare se il risultato di un'operazione è esattamente zero, può essere utilizzato anche per verificare se due operandi contengono lo stesso valore.

- Il *Sign flag (SF)* mantiene il segno degli operandi. Ovviamente non ha senso parlare di questo flag se gli operandi sono rappresentati come interi non segnati, mentre in caso di aritmetica segnata il flag contiene il bit più rappresentativo del risultato. Ad ogni operazione logica ed aritmetica il flag viene modificato, perciò sta all'utente capire quando ha veramente senso controllare il suo valore.
- L'*Overflow flag (OF)* ci dice se nell'ultima operazione c'è stato un overflow. È importante notare che l'OF non deve essere confuso con il CF, poiché entrambi rappresentano lo stesso risultato ma le condizioni iniziali sono diverse, ossia se gli operandi sono segnati oppure no. L'OF ha senso solo se gli operandi sono in aritmetica segnata. Se, ad esempio, i due operandi sono positivi ed il risultato è negativo, è ovvio che si sia verificato overflow, che è correttamente catturato dall'OF.

Altri registri nascosti sono TEMP1 e TEMP2. Questi hanno l'utilità di mantenere in memoria gli operandi dell'operazione che deve essere svolta dall'ALU o dallo SHIFTER. La presenza di questi registri è obbligatoria per questo tipo di architettura. Infatti l'ALU si suppone abile a calcolare la somma tra due operandi, leggendo i loro valori nello stesso istante. Le operazioni di trasferimento dati sono coordinate dallo SCO, attraverso l'utilizzo di una struttura di interconnessione che potrebbe permettere di trasferire dati da più di un registro alla volta. Istruzioni di rotazione e spostamento di bit sono coordinati dallo SCO e sono assegnate allo SHIFTER, mentre le operazioni logico/aritmetiche sono assegnate all'ALU.

1.3.La Struttura di Interconnessione dello Z64

Nel processore z64 si è adottata un'interconnessione a BUS, in modo da risparmiare componenti hardware sacrificando la possibilità di poter trasferire più dati simultaneamente. Infatti solo una coppia alla volta, sorgente-destinazione, può essere abilitata alla lettura-scrittura di un dato. Com'è possibile notare nella Figura2 se volessimo copiare il contenuto del registro %rax nel registro

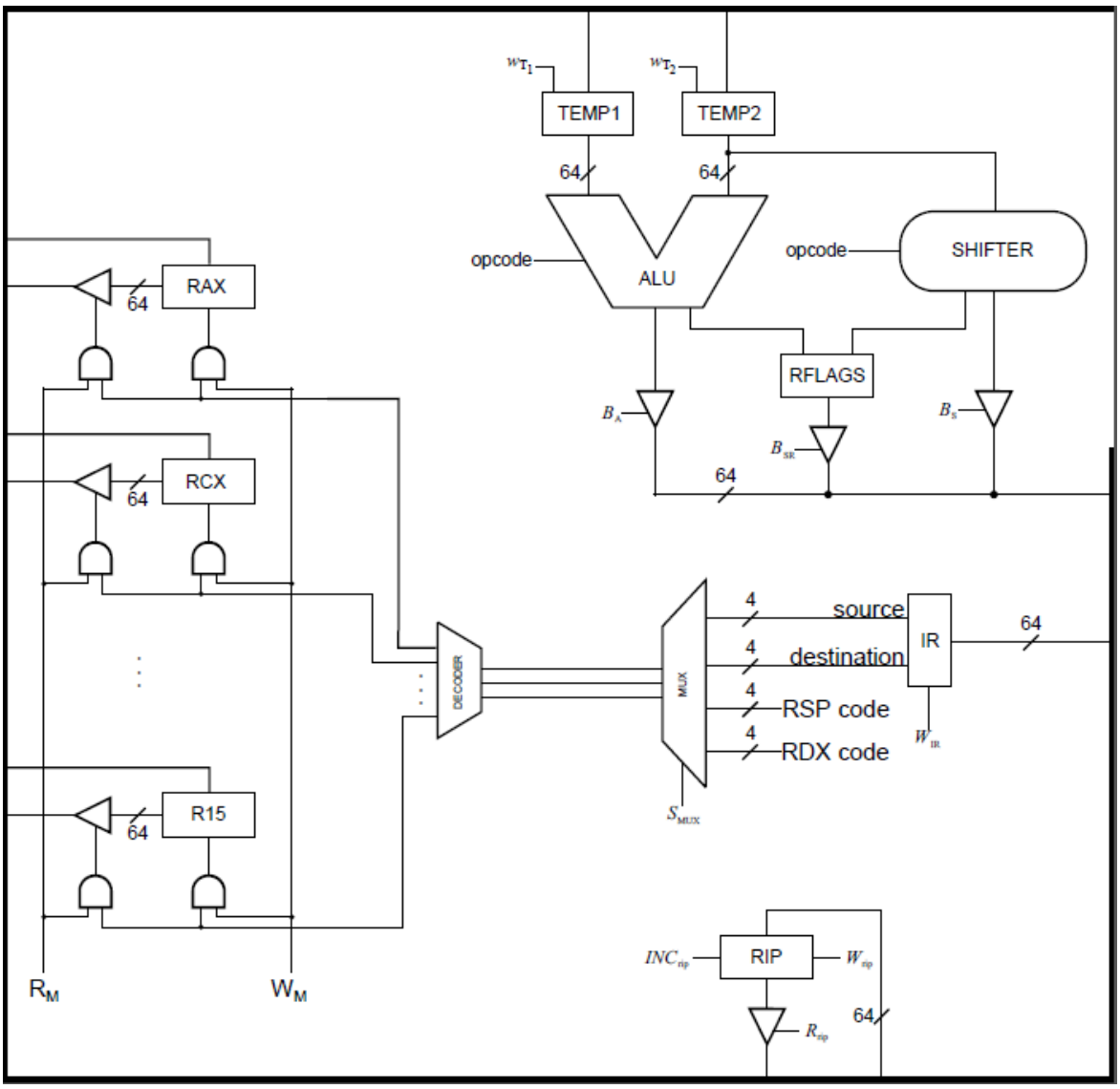


FIGURA 2

tampone TEMP2, i segnali generati dallo SCO saranno quelli di abilitazione alla lettura dal registro %rax, tramite il proprio buffer three-state, e l'abilitazione del segnale di scrittura w_{t2} del registro TEMP2.

In nero è evidenziato il BUS dati, esistono però anche il BUS relativo alla memoria e il BUS di input/output.

Dato che i registri visibili al programmatore sono i 16 (%rbx,…,%rsp, %r8-%r15) più %rip, %ir, TEMP1 e TEMP2 e dato che è possibile trasferire dati solo su questi, le complessità dello SCA e dello SCO posso essere sensibilmente semplificate.

I registri sono quindi organizzati in banchi, similmente ad una piccola memoria, e per ogni singolo registro c'è una linea di input e una linea di output controllate accuratamente dai propri buffer three-state che ne abilitano lettura e scrittura. Per fare ciò lo SCO si serve di DECODER per identificare quale registro si deve utilizzare e segnali di abilitazione di scrittura W e di lettura R per comandarne l'utilizzo.

Facendo tutto questo si sono risparmiate numerose componenti hardware, si sono utilizzate il minor numero di linee e segnali di controllo, minimizzando quindi anche lo spazio utilizzato.

1.4.L'estensione dello SCA per gestire dati di dimensione variabile

L'instruction set dello Z64 consente di manipolare dati a 8, 16, 32 e 64 bit, è perciò necessario che il banco dei registri permetta di accedere a dimensioni diverse dei registri.

Per selezionare uno dei 64 registri, è possibile usare due decoder, uno che utilizza 4 input (per selezionare uno dei registri), l'altro usa 2 input (per selezionare uno dei 4 formati). Gli input per questi 2 decoder vengono direttamente dal *%ir*, passando attraverso due multiplexer, uno che utilizza 5 input e l'altro che ne utilizza 4.

Consideriamo ad esempio il *%rax*, questo è suddiviso in 4 parti:

- La prima parte (%al) è composta da 8 flip-flop ed è connesso ai bit $IB_{0,7}$ dell'*Internal Bus IB;*
- La seconda parte (%ax) è composta da 16 flip-flop. I primi 8 sono quelli costituiti da %al e rappresentano i *least-significant bit.* Gli 8 bit rimanenti sono connessi ai bit $IB_{8,15}$ dell'IB;
- La terza parte (%eax) è composta da 32 flip-flop, in cui come nel caso precedente, i primi 16 sono composti dall'intero registro %ax e rappresentano i least-significant- bit, mentre i restanti 16 sono connessi all'IB per mezzo dei bit $IB_{16,31}$;
- Analogamente la quarta parte (%rax) rappresenta l'intero registro a 64 bit ed è composto dalla terza parte, %eax, che corrisponde ai primi 32 least-significant bit, mentre gli altri 32 sono i most-significant bit e sonoconnessi all'IB tramite i bit $IB_{32,63}$.

La connessione tra flip-flop e Internal Bus è usata sia per scritture che per letture. Quando si scrive sul bus (ossia quando si legge dai registri), l'uscita dei flip-flop è comandata dai buffer three-state, ancora una volta per prevenire fenomeni di cortocircuito. Invece quando leggiamo dal bus (ossia

quando scriviamo nel registro) l'interconnessione è diretta. Questo perché il segnale generato dallo SCO abiliterà solo il registro su cui dovremo andare a scrivere.

C'è da notare che il registro %al, a 8 bit, viene utilizzato per tutte le dimensioni di operazioni, indipendentemente dalla grandezza dei dati trattati, mentre le altre parti vengono coinvolte solo se i dati sono word, longword o quadword.

2. Il set di istruzioni dello Z64

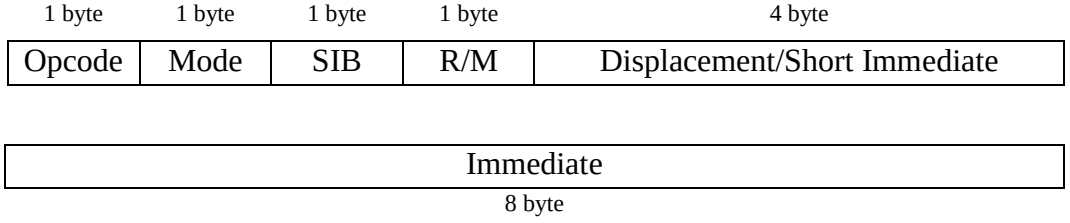
Le istruzioni usate per lo Z64 possono essere raggruppate in 8 differenti classi, dipendenti dalle azioni che esse devono effettuare. Ogni istruzione è codificata da una stringa di bit, permettendo allo Z64 di eseguirle. Queste stringhe sono generabili facilmente, tramite codici mnemonici che il programmatore deve ricordare. Sarà l'assemblatore che tradurrà quei codici mnemonici in sequenze di bit appropriate per eseguire quella specifica istruzione.

Per ogni classe di istruzione saranno specificati il tipo di operandi e quali Status Flag saranno coinvolti nell'operazione

La dimensione delle istruzioni dello z64 è variabile, infatti, dipende dall'operazione e dagli operandi coinvolti, intendendo che un'istruzione potrà essere a 64 o 128 bit.

Come è possibile vedere nella figura seguente, ci sono 6 campi che compongono il formato dell'istruzione.

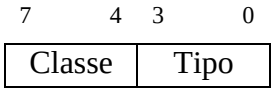
- Opcode: dice alla CPU qual è il microcodice per l'esecuzione dell'istruzione;
- Mode: informazioni addizionali riguardo al modo operativo specifico;
- SIB: da informazioni riguardo la scala, l'indice e la base dell'operando in memoria;
- R/M: utilizzato quando le istruzioni necessitano operandi espliciti;
- Displacement: blocco da 32 bit per tenere traccia di un indirizzo di memoria, usato in operazioni di spostamento dati per puntare variabili specifiche in memoria
- Immediate: composto da 64 bit consente di memorizzare oltre all'istruzione, anche un dato a 64 bit che verrà utilizzato durante l'esecuzione



1. Opcode

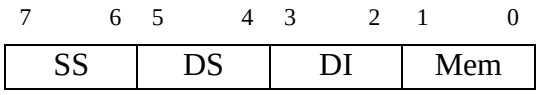
   Questo blocco è formato da due parti, una è la Classe, che identifica la classe dell'istruzione, lo Z64 ha 8 classi e per questo il bit più importante è sempre settato a 0. Avere questo bit libero implica che le classi possono essere ampliate fino a 16; l'altra parte è il Tipo, che

permette di selezionare con precisione l'esatta istruzione di quella classe che dovrà essere eseguita dal processore.



2. Mode

Questo blocco è formato da 4 parti. Le prime due sono costituite da SS e DS che contengono rispettivamente le dimensioni dell'operando sorgente e dell'operando destinatario specificate nell'istruzione mnemonica, talvolta possono anche essere identici. DI invece comunica al processore se c'è spiazzamento, se il dato è immediato o entrambe le soluzioni. Mem infine dichiara se gli operandi devono essere interpretati come operandi in memoria o registri. Nel caso in cui uno dei due operandi sia un operando di memoria, allora la CPU dovrà interpretare i parametri del blocco SIB per l'indirizzamento corretto. La tabella successiva mostra la composizione del blocco Mode:



La tabella successiva mostra i valori assunti nei possibili casi:

Campo	**Valore**	**Spiegazone**
SS	00 01 10 11	La sorgente è un byte La sorgente è una word La sorgente è una longword La sorgente è una quadword
DS	00 01 10 11	La destinazione è un byte La destinazione è una word La destinazione è una longword La destinazione è una quadword
DI	00 01 10 11	Displacement assente, immediate assente Immediate Displacement Displacement e immediate
Mem	00 01 10 11	Entrambi sono registri La Sorgente è un registro, la destinazione è in memoria La sorgente è in memoria, destinazione è un registro Condizione impossibile (runtime error)

3. SIB

   Questo campo specifica informazioni addizionali sugli operandi ed è suddiviso in quattro parti B_p, I_p, *Scale* e *Index Register*. B_p e I_p indicano se l'istruzione sta usando una base e/o un *index register* rispettivamente. *Scale* tiene in considerazione il valore della scala, i cui valori possibili sono 1 (codificato con 00), 2 (codificato con 01), 4 (codificato con 10) e 8 (codificato con 11). Se $I_p == 1$ il campo *index* mantiene la rappresentazione binaria dell'index register.

7	6	5 4	3 0
B_p	I_p	Scale	Index register

4. R/M

   Il byte R/M ha spazio per due codici di registro, *Source (Base) register* e *Destination (Base) Register*. L'interpretazione di entrambi i registri dipende dal valore Mem del blocco *Mode* e dal valore di B_p del blocco *SIB*. Nello specifico, questi due registri possono essere registri *general purpose* oppure possono essere lo stesso registro utilizzato come *base register*. Questo dipende se uno dei due operandi deve essere interpretato come un operando memoria oppure no (come definito nel campo Mem), oppure se l'indirizzamento in memoria usa *base register* oppure no (come definito dal bit B_p del campo SIB).

7 4	3 0
Source (Base) register	Destination (Base) Register

Nelle seguenti tabelle saranno utilizzate lettere maiuscole e minuscole per identificare gli operandi utilizzati nelle istruzioni. Le lettere maiuscole saranno:

- B – Base register
- D – Destination register
- I – Index register
- O – Displacement (Offset)
- S – Source register
- T – Scale

- X – Operand size
- Y – Additional operand size (se necessario)

Le lettere minuscule compariranno direttamente nell'istruzione:

- b – base register
- d – destination register
- i – index register
- k – constant unsigned value fino a $2^{32} - 1$
- m – memory location
- o – displacement (offset)
- p – I/O port rappresentato come un intero senza segno fino a $2^{16} - 1$
- s – source register
- t – scale
- x – operand size
- y – additional operand size (se necessario)
- \- – don't care bit
- ✓ – c'è un dato immediato subito dopo l'istruzione
- × – non c'è un dato immediato subito dopo l'istruzione

2.1. Classe 0: Istruzioni di controllo hardware

Type	Mnemonic	Operands	O	S	Z	P	C	Description
1	hlt	-	-	-	-	-	-	Mette la cpu in low-power mode, fino alla prossima interruzione
2	nop	-	-	-	-	-	-	No operation
3	int	-	-	-	-	-	-	Chiamata al gestore interruzioni

La Codifica binaria sarà:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
hlt	0000 0001	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×
nop	0000 0010	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×

2.2. Classe 1: Istruzioni di spostamento

Type	Mnemonic	Operands	O	S	Z	P	C	Description
0	mov	B,E	-	-	-	-	-	Copia B in E
1	movsX	E,G	-	-	-	-	-	Copia E in G e estende il segno
2	movzX	E,G	-	-	-	-	-	Copia E in G e estende con zeri
3	lea	E,G	-	-	-	-	-	Calcola l'indirizzamento e mette in G
4	push	E	-	-	-	-	-	Copia E in cima allo stack
5	pop	E	-	-	-	-	-	Copia la cima dello stack in E
6	pushf	-	-	-	-	-	-	Copia %flags in cima allo stack
7	popf	-	-	-	-	-	-	Copia la cima dello stack in %flags
8	movs	-	-	-	-	-	-	Sposta dalla memoria alla memoria
9	stos	-	-	-	-	-	-	Imposta la regione di memoria ad un valore

La codifica binaria della mov:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
movX S,D	0001 0000	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
movX S,(B)	0001 0000	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
movX S,(B, I, T)	0001 0000	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
movX S,O(B,I,T)	0001 0000	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
movX S,(,I,T)	0001 0000	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
movX S, O(,I,T)	0001 0000	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
movX S,O	0001 0000	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
movX I,D	0001 0000	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
movX I, (B)	0001 0000	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
movX I,(B, I, T)	0001 0000	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
movX I,O(B,I,T)	0001 0000	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
movX I,(,I,T)	0001 0000	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
movX I, O(,I,T)	0001 0000	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
movX I,O	0001 0000	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
movX (B),D	0001 0000	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
movX (B, I, T),D	0001 0000	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
movX O(B,I,T),D	0001 0000	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
movX (,I,T),D	0001 0000	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
movX O(,I,T),D	0001 0000	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
movX O,D	0000 0010	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

La codifica binaria della movsXY:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
movsXY S,D	0001 0001	xxyy 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
movsXY (B),D	0001 0001	xxyy 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
movsXY (B, I, T),D	0001 0001	xxyy 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
movsXY O(B,I,T),D	0001 0001	xxyy 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
movsXY (,I,T),D	0001 0001	xxyy 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
movsXY O(,I,T),D	0001 0001	xxyy 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

movsX O,D	0000 0011	xxyy 1010	00-- ----	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×

La codifica binaria della movzXY:

Instruction	Encoding					
	Opcode	Mode	SIB	R/M	Displace	Imm
movzXY S,D	0001 0001	xxyy 0000	00-- ----	ssss dddd	---- ---- ---- ---- ---- ---- ---- ----	×
movzXY (B),D	0001 0001	xxyy 0010	10-- ----	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
movzXY (B, I, T),D	0001 0001	xxyy 0010	11tt iiii	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
movzXY O(B,I,T),D	0001 0001	xxyy 1010	11tt iiii	bbbb dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
movzXY (,I,T),D	0001 0001	xxyy 0010	01tt iiii	---- dddd	---- ---- ---- ---- ---- ---- ---- ----	×
movzXY O(,I,T),D	0001 0001	xxyy 1010	01tt iiii	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
movzX O,D	0001 0011	xxyy 1010	00-- ----	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×

La codifica binaria della leaX:

Instruction	Encoding					
	Opcode	Mode	SIB	R/M	Displace	Imm
leaX (B),D	0001 0001	--xx 0010	10-- ----	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
leaX (B, I, T),D	0001 0001	--xx 0010	11tt iiii	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
leaX O(B,I,T),D	0001 0001	--xx 1010	11tt iiii	bbbb dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
leaX (,I,T),D	0001 0001	--xx 0010	01tt iiii	---- dddd	---- ---- ---- ---- ---- ---- ---- ----	×
leaX O(,I,T),D	0001 0001	--xx 1010	01tt iiii	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
leaX O,D	0001 0011	--xx 1010	00-- ----	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×

La codifica binaria della pushX:

Instruction	Encoding					
	Opcode	Mode	SIB	R/M	Displace	Imm
pushX S	0001 0100	xxxx 0000	00-- ----	ssss dddd	---- ---- ---- ---- ---- ---- ---- ----	×
pushX (B),D	0001 0001	xxxx 0010	10-- ----	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
pushX (B, I, T),D	0001 0001	xxxx 0010	11tt iiii	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
pushX O(B,I,T),D	0001 0001	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
pushX (,I,T),D	0001 0001	xxxx 0010	01tt iiii	---- dddd	---- ---- ---- ---- ---- ---- ---- ----	×
pushX O(,I,T),D	0001 0001	xxxx 1010	01tt iiii	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
pushX O,D	0001 0011	xxxx 1010	00-- ----	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×

La codifica binaria della popX:

Instruction	Encoding					
	Opcode	Mode	SIB	R/M	Displace	Imm
popX S	0001 0101	xxxx 0000	00-- ----	ssss dddd	---- ---- ---- ---- ---- ---- ---- ----	×
popX (B),D	0001 0101	xxxx 0010	10-- ----	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
popX (B, I, T),D	0001 0101	xxxx 0010	11tt iiii	bbbb dddd	---- ---- ---- ---- ---- ---- ---- ----	×
popX O(B,I,T),D	0001 0101	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
popX (,I,T),D	0001 0101	xxxx 0010	01tt iiii	---- dddd	---- ---- ---- ---- ---- ---- ---- ----	×
popX O(,I,T),D	0001 0101	xxxx 1010	01tt iiii	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×
popX O,D	0001 0101	xxxx 1010	00-- ----	---- dddd	oooo oooo oooo oooo oooo oooo oooo oooo	×

La codifica binaria della pushf:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
pushfw	0001 0110	xx01 ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×
pushfl	0001 0110	xx10 ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×
pushfq	0001 0110	xx11 ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×

La codifica binaria della popf:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
popfw	0001 0111	xx01 ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×
popfl	0001 0111	xx10 ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×
popfq	0001 0111	xx11 ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×

La codifica binaria della movsX:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
movsX	0001 1000	xx-- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×

La codifica binaria della stosX:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
stosX	0001 1001	xx-- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	×

2.3. Classe 2: Istruzioni Logico-Aritmetiche

Type	Mnemonic	Operands	O	S	Z	P	C	Description
0	add	B,E	⇳	⇳	⇳	⇳	⇳	Mette in E E+B
1	sub	B,E	⇳	⇳	⇳	⇳	⇳	Mette in E E-B
2	adc	B,E	⇳	⇳	⇳	⇳	⇳	Mette in E E+B+CF
3	sbb	B,E	⇳	⇳	⇳	⇳	⇳	Mette in E E-(B + neg(CF))
4	cmp	B,E	⇳	⇳	⇳	⇳	⇳	E-B il risultato è scartato
5	test	B,E	⇳	⇳	⇳	⇳	⇳	Confronta bit-bit B ed E il risultato è scartato
6	neg	E	⇳	⇳	⇳	⇳	⇳	Rimpiazza E con il suo complemnto a 2
7	and	B,E	O	⇳	⇳	⇳	O	Mette in E B&&E
8	or	B,E	O	⇳	⇳	⇳	O	Mette in E B||E
9	xor	B,E	O	⇳	⇳	⇳	O	Mette in E BxorE
10	not	E	O	⇳	⇳	⇳	O	Rimpiazza E con tutti i bit negati

Codifica binaria addX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
addX S,D	0010 0000	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
addX S,(B)	0010 0000	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
addX S,(B, I, T)	0010 0000	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
addX S,O(B,I,T)	0010 0000	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
addX S,(,I,T)	0010 0000	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
addX S, O(,I,T)	0010 0000	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
addX S,O	0010 0000	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
addX I,D	0010 0000	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
addX I, (B)	0010 0000	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
addX I,(B, I, T)	0010 0000	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
addX I,O(B,I,T)	0010 0000	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
addX I,(,I,T)	0010 0000	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
addX I, O(,I,T)	0010 0000	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
addX I,O	0010 0000	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
addX (B),D	0010 0000	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
addX (B, I, T),D	0010 0000	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
addX O(B,I,T),D	0010 0000	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
addX (,I,T),D	0010 0000	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
addX O(,I,T),D	0010 0000	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
addX O,D	0010 0010	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria subX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
subX S,D	0010 0001	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
subX S,(B)	0010 0001	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
subX S,(B, I, T)	0010 0001	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
subX S,O(B,I,T)	0010 0001	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
subX S,(,I,T)	0010 0001	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
subX S, O(,I,T)	0010 0001	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
subX S,O	0010 0001	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
subX I,D	0010 0001	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
subX I, (B)	0010 0001	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
subX I,(B, I, T)	0010 0001	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
subX I,O(B,I,T)	0010 0001	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
subX I,(,I,T)	0010 0001	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
subX I, O(,I,T)	0010 0001	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
subX I,O	0010 0001	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
subX (B),D	0010 0001	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
subX (B, I, T),D	0010 0001	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
subX O(B,I,T),D	0010 0001	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
subX (,I,T),D	0010 0001	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
subX O(,I,T),D	0010 0001	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
subX O,D	0010 0011	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria adcX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
adcX S,D	0010 0010	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
adcX S,(B)	0010 0010	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
adcX S,(B, I, T)	0010 0010	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
adcX S,O(B,I,T)	0010 0010	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
adcX S,(,I,T)	0010 0010	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
adcX S, O(,I,T)	0010 0010	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
adcX S,O	0010 0010	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
adcX I,D	0010 0010	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
adcX I, (B)	0010 0010	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
adcX I,(B, I, T)	0010 0010	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
adcX I,O(B,I,T)	0010 0010	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
adcX I,(,I,T)	0010 0010	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
adcX I, O(,I,T)	0010 0010	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
adcX I,O	0010 0010	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
adcX (B),D	0010 0010	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
adcX (B, I, T),D	0010 0010	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
adcX O(B,I,T),D	0010 0010	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
adcX (,I,T),D	0010 0010	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
adcX O(,I,T),D	0010 0010	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
adcX O,D	0010 0010	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria sbbX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
sbbX S,D	0010 0011	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
sbbX S,(B)	0010 0011	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
sbbX S,(B, I, T)	0010 0011	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
sbbX S,O(B,I,T)	0010 0011	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
sbbX S,(,I,T)	0010 0011	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
sbbX S, O(,I,T)	0010 0011	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
sbbX S,O	0010 0011	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
sbbX I,D	0010 0011	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
sbbX I, (B)	0010 0011	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
sbbX I,(B, I, T)	0010 0011	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
sbbX I,O(B,I,T)	0010 0011	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
sbbX I,(,I,T)	0010 0011	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
sbbX I, O(,I,T)	0010 0011	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
sbbX I,O	0010 0011	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
sbbX (B),D	0010 0011	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
sbbX (B, I, T),D	0010 0001	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
sbbX O(B,I,T),D	0010 0011	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
sbbX (,I,T),D	0010 0011	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
sbbX O(,I,T),D	0010 0011	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
sbbX O,D	0010 0011	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria cmpX:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
cmpX S,D	0010 0100	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
cmpX S,(B)	0010 0100	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	×
cmpX S,(B, I, T)	0010 0100	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	×
cmpX S,O(B,I,T)	0010 0100	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
cmpX S,(,I,T)	0010 0100	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	---- ----	×
cmpX S, O(,I,T)	0010 0100	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
cmpX S,O	0010 0100	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
cmpX I,D	0010 0100	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	---- ----	✓
cmpX I, (B)	0010 0100	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	✓
cmpX I,(B, I, T)	0010 0100	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	✓
cmpX I,O(B,I,T)	0010 0100	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
cmpX I,(,I,T)	0010 0100	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	✓
cmpX I, O(,I,T)	0010 0100	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
cmpX I,O	0010 0100	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
cmpX (B),D	0010 0100	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
cmpX (B, I, T),D	0010 0100	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
cmpX O(B,I,T),D	0010 0100	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
cmpX (,I,T),D	0010 0100	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
cmpX O(,I,T),D	0010 0100	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
cmpX O,D	0010 0100	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria testX:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
testX S,D	0010 0101	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
testX S,(B)	0010 0101	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	×
testX S,(B, I, T)	0010 0101	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	×
testX S,O(B,I,T)	0010 0101	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
testX S,(,I,T)	0010 0101	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	---- ----	×
testX S, O(,I,T)	0010 0101	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
testX S,O	0010 0101	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
testX I,D	0010 0101	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	---- ----	✓
testX I, (B)	0010 0101	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	✓
testX I,(B, I, T)	0010 0101	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	---- ----	✓
testX I,O(B,I,T)	0010 0101	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
testX I,(,I,T)	0010 0101	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	✓
testX I, O(,I,T)	0010 0101	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
testX I,O	0010 0101	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
testX (B),D	0010 0101	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
testX (B, I, T),D	0010 0101	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
testX O(B,I,T),D	0010 0101	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
testX (,I,T),D	0010 0101	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	---- ----	×
testX O(,I,T),D	0010 0101	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
testX O,D	0010 0101	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria negX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
negX S	0010 0110	xxxx 0000	00-- ----	ssss ----	---- ----	---- ----	---- ----	---- ----	×
negX (B)	0010 0110	xxxx 0010	10-- ----	bbbb ----	---- ----	---- ----	---- ----	---- ----	×
negX (B, I, T)	0010 0110	xxxx 0010	11tt iiii	bbbb ----	---- ----	---- ----	---- ----	---- ----	×
negX O(B,I,T)	0010 0110	xxxx 1010	11tt iiii	bbbb ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
negX (,I,T)	0010 0110	xxxx 0010	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	×
negX O(,I,T)	0010 0110	xxxx 1010	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
negX O	0010 0110	xxxx 1010	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria andX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
andX S,D	0010 0111	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
andX S,(B)	0010 0111	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
andX S,(B, I, T)	0010 0111	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
andX S,O(B,I,T)	0010 0111	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
andX S,(,I,T)	0010 0111	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
andX S, O(,I,T)	0010 0111	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
andX S,O	0010 0111	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
andX I,D	0010 0111	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
andX I, (B)	0010 0111	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
andX I,(B, I, T)	0010 0111	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
andX I,O(B,I,T)	0010 0111	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
andX I,(,I,T)	0010 0111	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
andX I, O(,I,T)	0010 0111	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
andX I,O	0010 0111	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
andX (B),D	0010 0111	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
andX (B, I, T),D	0010 0111	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
andX O(B,I,T),D	0010 0111	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
andX (,I,T),D	0010 0111	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
andX O(,I,T),D	0010 0111	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
andX O,D	0010 0111	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria orX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
orX S,D	0010 1000	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
orX S,(B)	0010 1000	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
orX S,(B, I, T)	0010 1000	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
orX S,O(B,I,T)	0010 1000	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
orX S,(,I,T)	0010 1000	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
orX S, O(,I,T)	0010 1000	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
orX S,O	0010 1000	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
orX I,D	0010 1000	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
orX I, (B)	0010 1000	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
orX I,(B, I, T)	0010 1000	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
orX I,O(B,I,T)	0010 1000	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
orX I,(,I,T)	0010 1000	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
orX I, O(,I,T)	0010 1000	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
orX I,O	0010 1000	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
orX (B),D	0010 1000	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
orX (B, I, T),D	0010 1000	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
orX O(B,I,T),D	0010 1000	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
orX (,I,T),D	0010 1000	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
orX O(,I,T),D	0010 1000	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
orX O,D	0010 1000	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria xorX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
xorX S,D	0010 1001	xxxx 0000	00-- ----	ssss dddd	---- ----	---- ----	---- ----	---- ----	×
xorX S,(B)	0010 1001	xxxx 0001	01-- ----	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
xorX S,(B, I, T)	0010 1001	xxxx 0001	11tt iiii	ssss bbbb	---- ----	---- ----	---- ----	---- ----	×
xorX S,O(B,I,T)	0010 1001	xxxx 1001	11tt iiii	ssss bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
xorX S,(,I,T)	0010 1001	xxxx 0001	01tt iiii	ssss ----	---- ----	---- ----	---- ----	---- ----	×
xorX S, O(,I,T)	0010 1001	xxxx 1001	01tt iiii	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
xorX S,O	0010 1001	xxxx 1001	00-- ----	ssss ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
xorX I,D	0010 1001	xxxx 0100	00-- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	✓
xorX I, (B)	0010 1001	xxxx 0101	10-- ----	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
xorX I,(B, I, T)	0010 1001	xxxx 0101	11tt iiii	---- bbbb	---- ----	---- ----	---- ----	---- ----	✓
xorX I,O(B,I,T)	0010 1001	xxxx 1101	11tt iiii	---- bbbb	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
xorX I,(,I,T)	0010 1001	xxxx 0101	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	✓
xorX I, O(,I,T)	0010 1001	xxxx 1101	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
xorX I,O	0010 1001	xxxx 1101	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	✓
xorX (B),D	0010 1001	xxxx 0010	10-- ----	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
xorX (B, I, T),D	0010 1001	xxxx 0010	11tt iiii	bbbb dddd	---- ----	---- ----	---- ----	---- ----	×
xorX O(B,I,T),D	0010 1001	xxxx 1010	11tt iiii	bbbb dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
xorX (,I,T),D	0010 1001	xxxx 0010	01tt iiii	---- dddd	---- ----	---- ----	---- ----	---- ----	×
xorX O(,I,T),D	0010 1001	xxxx 1010	01tt iiii	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
xorX O,D	0010 1001	xxxx 1010	00-- ----	---- dddd	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

Codifica binaria notX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
notX S	0010 1010	xxxx 0000	00-- ----	ssss ----	---- ----	---- ----	---- ----	---- ----	×
notX (B)	0010 1010	xxxx 0010	10-- ----	bbbb ----	---- ----	---- ----	---- ----	---- ----	×
notX (B, I, T)	0010 1010	xxxx 0010	11tt iiii	bbbb ----	---- ----	---- ----	---- ----	---- ----	×
notX O(B,I,T)	0010 1010	xxxx 1010	11tt iiii	bbbb ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
notX (,I,T)	0010 1010	xxxx 0010	01tt iiii	---- ----	---- ----	---- ----	---- ----	---- ----	×
notX O(,I,T)	0010 1010	xxxx 1010	01tt iiii	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×
notX O	0010 1010	xxxx 1010	00-- ----	---- ----	oooo oooo	oooo oooo	oooo oooo	oooo oooo	×

2.4. Classe 3: Istruzioni di Rotate and Shift

Type	Mnemonic	Operands	O	S	Z	P	C	Description
0	sal	K,G	⇳	⇳	⇳	⇳	⇳	Moltiplica G per 2 K volte
1	sal	G	⇳	⇳	⇳	⇳	⇳	Moltiplica G per 2 %rcx volte
2	shl	K,G	⇳	⇳	⇳	⇳	⇳	Moltiplica G per 2 K volte
3	shl	G	⇳	⇳	⇳	⇳	⇳	Moltiplica G per 2 %rcx volte
4	sar	K,G	⇳	⇳	⇳	⇳	⇳	Divide G per 2 K volte
5	sar	G	⇳	⇳	⇳	⇳	⇳	Divide G per 2 %rcx volte
6	shr	K,G	⇳	⇳	⇳	⇳	⇳	Divide G per 2 K volte
7	shr	G	⇳	⇳	⇳	⇳	⇳	Divide G per 2 %rcx volte
8	rcl	K,G	⇳	-	-	-	⇳	Ruota a sinistra G, K volte
9	rcl	G	⇳	-	-	-	⇳	Ruota a sinistra G, %rcx volte
10	rcr	K,G	⇳	-	-	-	⇳	Ruota a destra G, K volte
11	rcr	G	⇳	-	-	-	⇳	Ruota a destra G, %rcx volte
12	rol	K,G	⇳	-	-	-	⇳	Ruota a sinistra G, K volte
13	rol	G	⇳	-	-	-	⇳	Ruota a sinistra G, %rcx volte
14	ror	K,G	⇳	-	-	-	⇳	Ruota a destra G, K volte
15	ror	G	⇳	-	-	-	⇳	Ruota a destra G, %rcx volte

Codifica binaria salX/shlX/sarX/shrX:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
salX K,D	0011 0000	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk	×
salX D	0011 0001	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	×
shlX K,D	0011 0010	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk	×
shlX D	0011 0011	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	×
sarX K,D	0011 0010	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk	×
sarX D	0011 0011	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	×
shrX K,D	0011 0100	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk	×
shrX D	0011 0101	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----	×

Codifica binaria rclX/rcrX/rolX/rorX:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
rclX K,D	0011 0110	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk		×
rclX D	0011 0111	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----		×
rcrX K,D	0011 1000	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk		×
rcrX D	0011 1001	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----		×
rolX K,D	0011 1010	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk		×
rolX D	0011 1011	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----		×
rorX K,D	0011 1100	xxxx ----	---- ----	---- dddd	0000 0000	0000 0000	0000 0000	0kkk kkkk		×
rorX D	0011 1101	xxxx ----	---- ----	---- dddd	---- ----	---- ----	---- ----	---- ----		×

2.5. Classe 4: Istruzioni per la manipolazione dei Flag bit

Type	Mnemonic	Operands	O	S	Z	P	C	Description
0	clc	-	-	-	-	-	0	Azzera il carry-flag
1	clp	-	-	-	-	0	-	Azzera il parity-flag
2	clz	-	-	-	0	-	-	Azzera lo zero-flag
3	cls	-	-	0	-	-	-	Azzera il sign-flag
4	cli	-	-	-	-	-	-	
5	cld	-	-	-	-	-	-	
6	clo	-	0	-	-	-	-	Azzera l'overflow flag
7	stc	-	-	-	-	-	1	Setta il carry-flag
8	stp	-	-	-	-	1	-	Setta il parity-flag
9	stz	-	-	-	1	-	-	Setta lo zero-flag
10	sts	-	-	1	-	-	-	Setta il sign-flag
11	sti	-	-	-	-	-	-	
12	std	-	-	-	-	-	-	
13	sto	-	1	-	-	-	-	Setta l'overflow-flag

Codifica binaria:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
clc	0100 0000	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
clp	0100 0001	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
clz	0100 0010	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
cls	0100 0011	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
cli	0100 0100	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
cld	0100 0101	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
clo	0100 0110	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
stc	0100 0111	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
stp	0100 1000	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
stz	0100 1001	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
sts	0100 1010	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
sti	0100 1011	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
std	0100 1100	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
sto	0100 1101	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×

2.6. Classe 5: Istruzioni per il controllo del flusso del programma

Type	Mnemonic	Operands	O	S	Z	P	C	Description
0	jmp	M	-	-	-	-	-	Effettua un salto relativo
1	jmp	*G	-	-	-	-	-	Effettua un salto assoluto
2	call	M	-	-	-	-	-	Effettua una chiamata a subroutine relativa
3	call	*G	-	-	-	-	-	Effettua una chiamata a subroutine assoluta
4	ret	-	-	-	-	-	-	Ritorna da una subroutine
5	iret	-	⇳	⇳	⇳	⇳	⇳	Ritorna da un’interruzione

Codifica binaria:

Instruction	Encoding									
	Opcode	Mode	SIB	R/M	Displace					Imm
jmp M	0101 0000	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm		×
jmp *R	0101 0001	---- ----	---- ----	---- rrrr	---- ----	---- ----	---- ----	---- ----		×
call M	0101 0010	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm		×
call *R	0101 0011	---- ----	---- ----	---- rrrr	---- ----	---- ----	---- ----	---- ----		×
ret	0101 0100	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×
iret	0101 0101	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----	---- ----		×

2.7. Classe 6: Istruzioni condizionali per il flusso di controllo

Type	Mnemonic	Operands	O	S	Z	P	C	Description
0	jc	M	-	-	-	-	-	Salta a M se c è settato
1	jp	M	-	-	-	-	-	Salta a M se p è settato
2	jz	M	-	-	-	-	-	Salta a M se z è settato
3	js	M	-	-	-	-	-	Salta a M se s è settato
4	jo	M	-	-	-	-	-	Salta a M se o è settato
5	jnc	M	-	-	-	-	-	Salta a M se c non è settato
6	jnp	M	-	-	-	-	-	Salta a M se p non è settato
7	jnz	M	-	-	-	-	-	Salta a M se z non è settato
8	jns	M	-	-	-	-	-	Salta a M se s non èsettato
9	jno	M	-	-	-	-	-	Salta a M se o non è settato

Codifica binaria:

Instruction	Encoding								
	Opcode	Mode	SIB	R/M	Displace				Imm
jc M	0110 0000	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jp M	0110 0001	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jz M	0110 0010	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
js M	0110 0011	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jo M	0110 0100	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jnc M	0110 0101	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jnp M	0110 0110	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jnz M	0110 0111	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jns M	0110 1000	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jno M	0110 1001	---- ----	---- ----	---- ----	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×

2.8. Classe 7: Istruzioni Input/Output

Type	Mnemonic	Operands	O	S	Z	P	C	Description
0	in		-	-	-	-	-	Copia dati dalla porta I/O all'accumulatore
1	out		-	-	-	-	-	Copia dati dall'accumulatore all'I/O
2	ins		-	-	-	-	-	Copia dati dalla porta I/O alla memoria
3	outs		-	-	-	-	-	Copia dati dalla memoria alla porta I/O
4	start	P	-	-	-	-	-	Resetta lo STATUS della periferica e avvia il device
5	clear	P	-	-	-	-	-	Resetta lo STATUS sul device P
6	jr	P,M	-	-	-	-	-	Salta ad M se P è ready
7	jnr	P,M	-	-	-	-	-	Salta ad M se P non è ready
8	wait		-	-	-	-	-	Blocca l'esecuzione finchè il f/f WAIT non è resettato

Codifica binaria:

Instruction	Encoding							
	Opcode	Mode	I/O Port	Displace				Imm
inX	0111 0000	xx-- ----	---- ---- ---- ----	---- ----	---- ----	---- ----	---- ----	×
outX	0111 0001	xx-- ----	---- ---- ---- ----	---- ----	---- ----	---- ----	---- ----	×
insX	0111 0010	xx-- ----	---- ---- ---- ----	---- ----	---- ----	---- ----	---- ----	×
outX	0111 0011	xx-- ----	---- ---- ---- ----	---- ----	---- ----	---- ----	---- ----	×
start P	0111 0100	xx-- ----	pppp pppp pppp pppp	---- ----	---- ----	---- ----	---- ----	×
clear P	0111 0101	xx-- ----	pppp pppp pppp pppp	---- ----	---- ----	---- ----	---- ----	×
jr P,M	0111 0110	xx-- ----	pppp pppp pppp pppp	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
jnr P,M	0111 0111	xx-- ----	pppp pppp pppp pppp	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	×
wait	0111 1000	xx-- ----	---- ---- ---- ----	---- ----	---- ----	---- ----	---- ----	×

3. Cenni di programmazione Assembly

La programmazione assembly è un tema molto delicato, in quanto richiede che lo sviluppatore di software ragioni come una macchina, affinchè possa produrre codice che sia efficiente in termini di prestazione e uso della memoria.

Quando si programma in assembly, le istruzioni sono in rapporto 1:1 con le parole binarie che la CPU è capace di interpretare, quindi un'attenta scrittura di *routine* in assembly può essere molto più veloce della stessa routine scritta in un linguaggio di alto livello.

Ogni linguaggio assembly ha la sua sintassi, il codice assembly dello Z64 usa la sintassi AT&T, che utilizza istruzioni nella forma:

mnemonic source, destination

in cui mnemonic corrisponde all'istruzione assembly, mentre source e destination possono essere registri, locazioni di memoria, dati immediati, a seconda del significato dell'istruzione.

Un programma assembly consiste in un insieme di istruzioni che operano sui dati. Queste istruzioni sono tradotte in codice binario da uno specifico programma chiamato *assembler.* Affinchè l'assembler sia in grado di riconoscere che tipo di dati stiamo trattando, ovvero riconoscere la dimensione di tali dati, è necessario l'utilizzo di un *suffisso* alla fine dell'istruzione mnemonica. Per esempio prendendo l'istruzione add, se vogliamo aggiungere il contenuto di %eax al contenuto di %ebx esplicitando il fatto che ci interessano solo 32 bit, la sintassi sarà la seguente:

addl %eax, %ebx

mentre se vogliamo realizzare una somma su 16 bit tra i registri %ax e %bx scriveremo:

addw %ax, %bx

nella tabella seguente sono presenti i suffissi per le 4 dimensioni, Byte, Word, Long-Word, Quad-word.

Suffisso	Tipo di Operando	Quantità di bit
b	Byte	8 bit
w	Word	16 bit
l	Long-Word	32 bit
q	Quad-Word	64 bit

Nel caso in cui il suffisso non dovesse essere specificato, la grandezza è stabilita dall'assembler che la imposta alla dimensione del registro di destinazione, così le seguenti istruzioni sono equivalenti:

addl %eax, %ebx

add %eax, %ebx

è interessante notare che alcune operazioni potrebbero richiedere un suffisso addizionale, per permettere all'assembler di riempire correttamente i campi dell'istruzione. Questo è il caso in cui si vuole utilizzare l'istruzione dell'estensione del segno, in cui il processore è incaricato di trasformare dati da un formato ad un altro. L'istruzione *movsX* ci dice come la conversione deve essere fatta, così vengono utilizzati due suffissi, uno per la dimensione dell'operando sorgente, uno per la dimensione dell'operando destinazione. Per esempio se vogliamo spostare un valore a 16 bit dal registro %ax al registro %ebx, dovremo scrivere:

movswl %ax, %ebx

l'istruzione movsX può essere utilizzata per eseguire estensione di segno *in-place* (sul posto), semplicemente utilizzando lo stesso registro come sorgente e come destinazione, ma usando i due differenti nomi che identificano 2 dimensioni diverse:

movswl %ax, %eax

certamente la destinazione dovrà sempre essere più grande della sorgente, pertanto sono possibili varie combinazioni tra dimensioni diverse. L'istruzione mov, invece, nel caso in cui abbia la sorgente di dimensione più piccola della destinazione, provvederà ad estendere con tutti 0 il suo valore.

La tabella illustra le possibili combinazioni di estensione del segno.

Istruzione	**Tipo di conversione**
movsbw %al, %ax	Estensione del segno da byte a word
movsbl %al, %eax	Estensione del segno da byte a longword
movsbq %al, %rax	Estensione del segno da byte a quad-word
movswl %ax, %eax	Estensione del segno da word a longword
movswq %ax, %rax	Estensione del segno da word a quad-word
movslq %eax,%rax	Estensione del segno da longword a quad-word

Nel linguaggio assembly dello Z64 caratteri speciali hanno specifici significati. Ogni volta che si vuole accedere ad un registro, questo viene specificato anteponendo al nome del registro il simbolo "%", altrimenti viene segnalato un errore di sintassi. Per commentare il codice è invece necessario inserire il simbolo "#", che indica l'inizio del commento; non sono ammesse righe multiple di commenti. Ogni numero che rappresenta una costante dovrà essere preceduto dal simbolo "$".

È possibile rappresentare numeri in base decimale o in base esadecimale, la distinzione è implementata anteponendo al numero esadecimale il prefisso “0x”. Tutti i numeri che non hanno il simbolo “$” come prefisso, sono interpretati come indirizzi di memoria.

4. Microoperazioni z64

Quando si definisce qualsiasi istruzione assembly presentata nel capitolo 2, è necessario, affinchè il processore possa riuscire a portare a termine l’operazione, istanziare una serie di microoperazioni che scandiscano in maniera sistematica tutti i segnali di controllo generati dallo SCO per pilotare i componenti dello SCA. Sono definite 3 fasi che lo SCO definisce periodicamente:

- Fetch: fase identica che si ripete per ogni istruzione in cui si preleva dal registro %rip l’indirizzo di memoria in cui è contenuta l’istruzione da eseguire, si incrementa il valore di %rip in modo che assuma il valore della cella di memoria che punta alla prossima istruzione, infine si sposta l’istruzione da eseguire nell’%ir;
- Decode: serve per identificare quale micro-programma del sottosistema di controllo (SCO) attivare;
- Execute: serve ad implementare la “semantica” dell’istruzione. Può richiedere uno o più cicli macchina (interazioni con la memoria o i dispositivi di I/O);

Nella programmazione Java le differenti microoperazioni vengono definite nel momento in cui vengono creati gli oggetti che identificano le istruzioni. a seconda della classe e del tipo di istruzione, le microoperazioni saranno diverse.

Qui di seguito saranno definite tutte le microoperazioni necessare a svolgere tutte le istruzioni dell’instruction set del processore z64.

La sintassi utilizzata di seguito è abbastanza semplice:

- *Source* → *Destination* rappresenta lo spostamento di informazione dalla sorgente alla destinazione;
- *(registro)* indica l’accesso in memoria alla cella numero “registro”;
- *IR[0:31]* indica i bit 0-31 dell’istruzione in esecuzione

4.1. Classe 0

- Halt

  - - - FASE DI FETCH - - -
  RIP → EMAR
  (EMAR) → EMDR ; RIP+8 → RIP
  EMDR → IR
  - - - FINE FETCH - - -

- Nop

  - - - FASE DI FETCH - - -
  RIP → EMAR
  (EMAR) → EMDR ; RIP+8 → RIP
  EMDR → IR

- - - FINE FETCH - - -

La fase di FETCH è uguale per tutte le operazioni, quindi non sarà più trascritta.

Ho cercato di rendere sistematica la creazione della lista delle microoperazioni necessarie all'esecuzione di ogni singola istruzione tramite costrutti if-else. I passi necessari per interpretare il codice mnemonico sono simili per la maggior parte delle istruzioni, è però necessario interpretare con attenzione quali operazioni saranno svolte sui dati dalla specifica istruzione.

4.2. Istruzioni di Classe 1

Per la maggior parte delle istruzioni occorre valutare se il source o il destination sono operandi memorizzati nella memoria. Stabilito ciò il flusso del programma segue in maniera diversa a seconda dell'istruzione dichiarata nel programma assembly. Ogni microoperazione ha un preciso significato e deve essere messa nel posto esatto, in modo da non alterare la semantica del programma. I componenti utilizzati in seguito sono i registri, in cui vengono salvati dati e su cui vengono effettuate operazioni aritmetico-logiche dallo SHIFTER e dall'ALU.
È necessario valutare dapprima se uno dei due operandi è memorizzato in memoria, successivamente qual è l'indirizzo esatto della cella di memoria in cui andare a leggere o scrivere il dato, infine è possibile operare sui dati in modo da eseguire correttamente l'istruzione.

4.2.1. MOV

La mov sposta semplicemente il dato dal source al destination.

In questo caso abbiamo due rami if che stabiliscono se uno dei due operandi è inserito nella memoria, dopodichè si eseguono spostamenti da/verso la memoria.
Lo pseudocodice è il seguente:

```
if source is memory
        if source has Index register
                I → TEMP2
                SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
        if source has Base register
                B → TEMP1
                ALU_OUT[ADD] → TEMP2
         If source has Displacement
                IR[0:31] → TEMP1
                ALU_OUT[ADD] → TEMP2
        SHIFTER_OUT[SX, 0] → EMAR
        (EMAR) → EMDR
        EMDR → D

if else destination is memory
```

if destination has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if destination has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If destination has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Immediate
RIP → EMAR
(EMAR) → TEMP1; RIP+8 → RIP
If source has register
S → TEMP1
SHIFTER_OUT[SX, 0] → EMAR
TEMP1 → (EMAR)

4.2.2. LEA

L'operazione lea calcola l'indirizzo specificato in source e lo mette in destination. Rispetto alla mov solo il source può essere memorizzato in memoria, pertanto è necessario un solo ramo if.

if source is memory
if source has index
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if source has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
if source has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
SHIFTER_OUT[SX, 0] → EMAR
(EMAR) → EMDR
EMDR → D

4.2.3. PUSH

Nel caso della push alcune operazioni vanno fatte a priori, ad esempio prima di calcolare il valore dell'indirizzamento in memoria è necessario decrementare il valore del registro %rsp (lo stack pointer) della dimensione dell'operando. Lo pseudocodice è:

If source is memory
RSP → TEMP1
ALU_OUT(SUB,X) → RSP
if source has index
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if source has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2

```
        if source has Displacement
            IR[0:31] → TEMP1
            ALU_OUT[ADD] → TEMP2
        TEMP2 → EMAR
        (EMAR) → EMDR
        RSP → EMAR
        EMDR → (EMAR)
    If source is register
        RSP → TEMP1
        ALU_OUT(SUB,X) → RSP
        S → EMDR
        RSP → EMAR
        EMDR → (EMAR)
```

4.2.4. POP

L'istruzione di pop prende il valore in cima alla pila dello stack pointer e lo copia in destination. Finita quest'operazione il valore dello stack pointer viene incrementato. La dimensione dell'operando stabilisce quanto lo stack pointer debba essere incrementato. Le microoperazioni necessarie saranno definite attraverso lo pseudocodice seguente:

```
If source is memory
        RSP → EMAR
        (MAR) → EMDR
        if source has index
            I → TEMP2
            SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
        if source has Base register
            B → TEMP1
            ALU_OUT[ADD] → TEMP2
        if source has Displacement
            IR[0:31] → TEMP1
            ALU_OUT[ADD] → TEMP2
        SHIFTER_OUT[SX, 0] → EMAR
        EMDR → (EMAR)
        RSP → TEMP1
        ALU_OUT[ADD, X]
    If source is register
        RSP → EMAR
        (MAR) → EMDR
        EMDR → S
        RSP → TEMP1
        ALU_OUT[ADD,X] → RSP
```

4.2.5. PUSHF

Questa è un'istruzione è simile alla push ma più semplice, richiede infatti una quantità di microoperazioni costanti e non necessita di rami if. L'unica cosa che può cambiare è la dimensione del suffisso che può avere dimensioni word, long e quadword (pushw, pushl, pushq). Le microoperazioni sono le seguenti:

RSP → TEMP1
ALU_OUT[SUB, X] → RSP //X è la dimensione del suffisso
FLAGS → EMDR
RSP → EMAR
EMDR → (EMAR)

4.2.6. POPF

Analogamente all'istruzione pushf, questa è la versione più semplice della rispettiva pop. La dimensione dei dati spostati è definita dal suffisso della popf (popfw, popfl, popfq). Queste sono le microoperazioni:

RSP → EMAR
(EMAR) → EMDR
EMDR → FLAGS
RSP → TEMP1
ALU_OUT[ADD, X] → RSP //X è la dimensione del suffisso

4.2.7. MOVS

RSI → EMAR
(EMAR) → TEMP2
RDI → EMAR
SHIFTER_OUT[SX, 0] → (EMAR)

4.2.8. STOS

RDI → EMAR
RAX → (EMAR)

4.3. Classe 2

Le istruzioni di classe due possono essere a due o a un operando. In entrambi i casi è necessario riempire i registri tampone, eseguire l'operazione sui dati, decidere dove e se salvare il contenuto.

Mentre nelle operazioni di classe 1 non venivano fatti gli update sui registri di stato, nelle istruzioni di classe 2 tale aggiornamento sarà abilitato quando verranno effettuate le operazioni sui dati.

4.3.1. ADD

Per quanto riguarda l'istruzione di add il flusso è questo: si decide se il source o il destination sono memorizzati in memoria, si calcola l'indirizzamento in memoria, si effettua l'addizione tra gli operandi e infine vengono salvati i dati in memoria. Lo pseudocodice relativo alla add sarà:

```
If source is memory
    if source has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    if source has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
     If source has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP2
    D → TEMP1
    ALU_OUT[ADD] → D; SR_UPDATE = 1

If destination is memory
    If destination has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    If destination has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
    If destination has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP2
    If source has Immediate
        RIP → EMAR
        (EMAR) → TEMP1; RIP+8 → RIP
    If source is register
        S → TEMP1
    ALU_OUT[ADD] → TEMP1; SR_UPDATE = 1
    TEMP2 → EMAR
    TEMP1 → (EMAR)
```

4.3.2. SUB

L'istruzione di è in sostanza molto simile alla add, con la variazione del calcolo finale in cui l'alu effettua la sub tra i due registri tampone. È così definito lo pseudocodice:

```
If source is memory
    if source has Index register
```

I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if source has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
D → TEMP1
ALU_OUT[SUB] → D; SR_UPDATE = 1

If destination is memory
If destination has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
If destination has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If destination has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP1
If source has Immediate
RIP → EMAR
(EMAR) → TEMP2; RIP+8 → RIP
If source is register
S → TEMP2
ALU_OUT[SUB] → TEMP2; SR_UPDATE = 1
TEMP1 → EMAR
TEMP2 → (EMAR)

4.3.3. ADC

Questa istruzione identica all'operazione di add, si distungue da questa per via dell'operazione sul bit di carry, che viene inserito come terzo addendo della add. Il risultato finale sarà:

$$destination = source + destination + CF$$

Lo pseudocodice:

If source is memory
if source has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if source has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Displacement
IR[0:31] → TEMP1

ALU_OUT[ADD] → TEMP2
D → TEMP1
ALU_OUT[ADC] → D; SR_UPDATE = 1

If destination is memory
If destination has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
If destination has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If destination has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Immediate
RIP → EMAR
(EMAR) → TEMP1; RIP+8 → RIP
If source is register
S → TEMP1
ALU_OUT[ADC] → TEMP1; SR_UPDATE = 1
TEMP2 → EMAR
TEMP1 → (EMAR)

4.3.4. SBB

Questa particolare istruzione realizza la sottrazione:

$$destination = destination - (source + (CF)^c)$$

In genere utilizzata quando occorre operare sottrazioni su multibyte o multiword. Per esempio se si volesse eseguire una sottrazione su due dati a 128 bit, occorrerebbe dividere i due operandi in due parti e successivamente utilizzare l'istruzione sbb sulla prima e sulla seconda parte dei due dati. Le microoperazioni saranno definite dallo pseudocodice seguente:

If source is memory
if source has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if source has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
D → TEMP1
ALU_OUT[SBB] → D; SR_UPDATE = 1

If destination is memory
    If destination has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    If destination has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
    If destination has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP1
    If source has Immediate
        RIP → EMAR
        (EMAR) → TEMP2; RIP+8 → RIP
    If source is register
        S → TEMP2
    ALU_OUT[SBB] → TEMP2; SR_UPDATE = 1
    TEMP1 → EMAR
    TEMP2 → (EMAR)

4.3.5. CMP

L'istruzione di compare è la prima, tra quelle finora analizzate, che si disinteressa completamente del risultato dell'ultima operazione. Gli unici valori di interesse in questo caso sono gli status bit, attraverso i quali è possibile analizzare se il source è minore, maggiore o uguale rispetto al destination. Lo pseudocodice sarà:

If source is memory
    if source has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    if source has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
     If source has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP2
    D → TEMP1
    ALU_OUT[SUB] → TEMP3; SR_UPDATE = 1

If destination is memory
    If destination has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    If destination has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
    If destination has Displacement

IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP1
If source has Immediate
RIP → EMAR
(EMAR) → TEMP2; RIP+8 → RIP
If source is register
S → TEMP2
ALU_OUT[SUB] → TEMP3; SR_UPDATE = 1

4.3.6. TEST

L'istruzione di test esegue l'and logico tra il source e il destination, operando sui bit di stato e scartando il risultato. È utilizzata ad esempio per calcolare il valore di specifici bit. Questo è lo pseudocodice:

If source is memory
if source has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if source has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
D → TEMP1
ALU_OUT[AND] → TEMP3; SR_UPDATE = 1

If destination is memory
If destination has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
If destination has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If destination has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Immediate
RIP → EMAR
(EMAR) → TEMP1; RIP+8 → RIP
If source is register
S → TEMP1
ALU_OUT[AND] → TEMP3; SR_UPDATE = 1

4.3.7. NEG

Questa operazione unaria, lavora sul source rimpiazzando tale valore con il suo complemento a 2. Un ramo if scompare per via dell'operando unico. Lo pseudocodice è il seguente:

```
If source is memory
    if source has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    if source has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP1
     If source has Displacement
        IR[0:31] → TEMP2
        ALU_OUT[ADD] → TEMP1
    TEMP2 → EMAR
    ALU_OUT[NEG]→ (EMAR); SR_UPDATE = 1
```

4.3.8. AND

L'istruzione di and è analoga all'istruzione di test, ma il risultato viene salvato in destination. Lo pseudocodice sarà:

```
If source is memory
    if source has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    if source has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
     If source has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP2
    D → TEMP1
    ALU_OUT[AND] → D; SR_UPDATE = 1

If destination is memory
    If destination has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    If destination has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
    If destination has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP2
    If source has Immediate
        RIP → EMAR
        (EMAR) → TEMP1; RIP+8 → RIP
```

If source is register
S → TEMP1
TEMP2 → EMAR
(EMAR) → EMDR
EMDR → TEMP2
ALU_OUT[AND] → TEMP1; SR_UPDATE = 1
TEMP1 → (EMAR)

4.3.9. OR

Esegue l'or logico tra source e destination, provvedendo a salvare il risultato nella cella di memoria esatta. Lo pseudocodice per generare le microoperazioni necessarie è:

If source is memory
if source has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
if source has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
D → TEMP1
ALU_OUT[OR] → D; SR_UPDATE = 1

If destination is memory
If destination has Index register
I → TEMP2
SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
If destination has Base register
B → TEMP1
ALU_OUT[ADD] → TEMP2
If destination has Displacement
IR[0:31] → TEMP1
ALU_OUT[ADD] → TEMP2
If source has Immediate
RIP → EMAR
(EMAR) → TEMP1; RIP+8 → RIP
If source is register
S → TEMP1
TEMP2 → EMAR
(EMAR) → EMDR
EMDR → TEMP2
ALU_OUT[OR] → TEMP1; SR_UPDATE = 1
TEMP1 → (EMAR)

4.3.10. XOR

Esegue l'or esclusivo tra source e destination, salvando il risultato in destination. Le microoperazioni saranno definite dal seguente pseudocodice:

```
If source is memory
    if source has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    if source has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
     If source has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP2
    D → TEMP1
    ALU_OUT[XOR] → D; SR_UPDATE = 1

If destination is memory
    If destination has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    If destination has Base register
        B → TEMP1
        ALU_OUT[ADD] → TEMP2
    If destination has Displacement
        IR[0:31] → TEMP1
        ALU_OUT[ADD] → TEMP2
    If source has Immediate
        RIP → EMAR
        (EMAR) → TEMP1; RIP+8 → RIP
    If source is register
        S → TEMP1
    TEMP2 → EMAR
    (EMAR) → EMDR
    EMDR → TEMP2
    ALU_OUT[XOR] → TEMP1; SR_UPDATE = 1
    TEMP1 → (EMAR)
```

4.3.11. NOT

Anche questa operazione è di tipo unario e inverte tutti i bit dell'unico operando, salvando infine il risultato nell'indiriz'o di memoria corretto. Lo pseudocodice per generare le necessarie microoperazioni sarà il seguente:

```
If source is memory
    if source has Index register
        I → TEMP2
        SHIFTER_OUT[SX,T] → TEMP2; SR_UPDATE = 0
    if source has Base register
```

B → TEMP1
ALU_OUT[ADD] → TEMP1
If source has Displacement
IR[0:31] → TEMP2
ALU_OUT[ADD] → TEMP1
TEMP2 → EMAR
ALU_OUT[NOT]→ (EMAR); SR_UPDATE = 1

4.4. Classe 3

4.4.1. sal/sar/shl/shr

Queste sono operazioni di shift, il parametro K se presente decide di quanti posti deve essere shiftato il dato, se non è presente allora lo shift sarà effettuato di (rcx) volte, cioè tante volte quanto il dato contenuto in rcx. Lo pseudocodice, per generare le microoperazioni, relativo a queste istruzioni è:

if K è definito
D → TEMP2
SHIFTER_OUT[SAL/SAR/SHL/SHR, K] → D
if K non è definito
D → TEMP2
SHIFTER_OUT[SAL/SAR/SHL/SHR, RCX] → D

4.4.2. rcl/rcr/rol/ror

Queste invece sono operazioni di rotazione, il parametro K se presente decide di quanti posti deve essere ruotato il dato, se non è presente allora la rotazione sarà effettuata di (rcx) volte, cioè tante volte quanto il dato contenuto in rcx. Lo pseudocodice, per generare le microoperazioni, relativo a queste istruzioni è:

if K è definito
D → TEMP2
SHIFTER_OUT[RCL/RCR/ROL/ROR, K] → D
if K non è definite
D → TEMP2
SHIFTER_OUT[RCL/RCR/ROL/ROR, RCX] → D

4.5. Classe 4

Questa classe di istruzioni serve ad impostare in maniera arbitraria i bit di stato. Possono essere impostati a 0 tramite l'istruzione di clear, o impostati a 1 tramite l'istruzione di set. Le microoperazioni sono le seguenti:

4.5.1. clc/clp/clz/cls/cli/cld/clo

Queste istruzioni sono ad impostare i bit al valore 0

FLAGS[CF/PF/ZF/SF/IF/DF/OF] = 0

4.5.2. stc/stp/stz/sts/sti/std/sto

Queste istruzioni servono ad impostare i bit al valore 1.

FLAGS[CF/PF/ZF/SF/IF/DF/OF] = 1

4.6. Classe 5

Le istruzioni di questa classe servono ad alterare il flusso del programma attraverso salti forzati, chiamate di subroutine, ritorno da subroutine o ritorno da interruzioni.

4.6.1. jmp

Questa istruzione esegue un salto, o verso l'inidirizzo specificato, o attraverso l'indirizzo specificato nel registro. Lo pseudocodice per generare le microoperazioni è il seguente:

if indirizzo è un valore M
    IR[0:31] → EMAR
    (EMAR) → EMDR
    EMDR → RIP
if indirizzo è nel registro R
    R → EMAR
    (EMAR) → EMDR
    EMDR → RIP

4.6.2. call

Anche in questo caso si esegue un salto verso un indirizzo di memoria specificato nell'istruzione o in un registro, ma si salva il contesto nel momento in cui la call è effettuata, mettendo nello stack queste informazioni. Questo consentirà al processore di poter tornare all'esecuzione del normale flusso del programma una volta terminata la subroutine. Lo pseudocodice sarà:

if indirizzo è un valore M
    RSP → TEMP1
    ALU_OUT(SUB,8) → RSP
    RIP → EMDR
    RSP → EMAR
    EMDR → (EMAR)
    M → RIP
If indirizzo è nel registro R
    RSP → TEMP1
    ALU_OUT(SUB,8) → RSP
    RIP → EMDR
    RSP → EMAR
    EMDR → (EMAR)
    R → EMAR
    (EMAR) → EMDR
    EMDR → RIP

4.6.3. ret

La ret segnala il ritorno da una subroutine. Le microoperazioni sono standard:

RSP → EMAR
(MAR) → EMDR
MDR → RIP
RSP → TEMP1
ALU_OUT[ADD,8] → RSP

4.6.4. iret

La iret serve a tornare da un'interruzione, anche qui le microoperazioni sono standard:

RSP → EMAR
(EMAR) → EMDR
EMDR → FLAGS
RSP → TEMP1
ALU_OUT[ADD, 8] → RSP
RSP → EMAR
(EMAR) → EMDR
EMDR → RIP
RSP → TEMP1
ALU_OUT[ADD, 8] → RSP

4.7. Classe 6

Le istruzioni di questa classe servono ad eseguire salti condizionati. Se verificata la condizione, il salto verà effettuato.

4.7.1. jc/jp/jz/js/jo M

Se il bit di stato preso in considerazione è impostato a 1 allora verrà eseguito il salto, le microoperazioni saranno:

if FLAGS[CF/PF/ZF/SF/OF] == 1 then
IR[0:31] → EMAR
(EMAR) → EMDR
EMDR → RIP

4.7.2. jnc/jnp/jnz/jns/jno M

Se il bit di stato preso in considerazione è impostato a 0 allora verrà eseguito il salto, le microoperazioni saranno:

if FLAGS[CF/PF/ZF/SF/OF] == 0 then
IR[0:31] → EMAR
(EMAR) → EMDR
EMDR → RIP