Supponiamo di dover eseguire il seguente programma:

```
char buf[0x2000] = { 2, 6, -1, 200, ...,
  15, 3, -32, 1};
int main()
{
  int sum = 0;
  for (int i = 0; i < 0x2000; i++)
    sum += buf[i];
  return sum;
}
```

L'array buf contiene una serie di numeri di cui vogliamo conoscere la somma.

- Supponiamo per semplicità che lo spazio di indirizzamento virtuale vada da 0000 a 7fff (32 KiB).

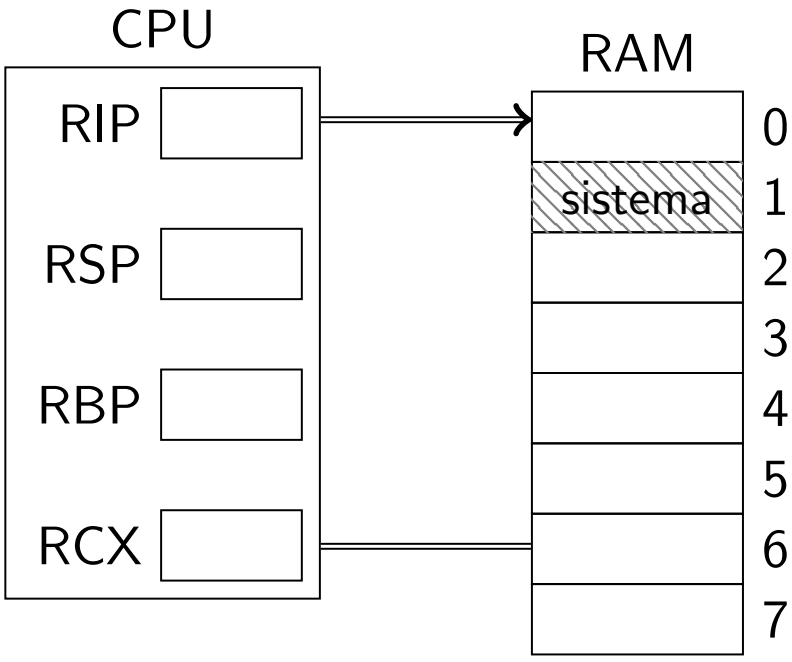
- Supponiamo per semplicità che lo spazio di indirizzamento virtuale vada da 0000 a 7fff (32 KiB).
- Immaginiamo di suddividere lo spazio di indirizzamento in 8 *pagine*, ciascuna grande 4 KiB (1000 in esadecimale);

- Supponiamo per semplicità che lo spazio di indirizzamento virtuale vada da 0000 a 7fff (32 KiB).
- Immaginiamo di suddividere lo spazio di indirizzamento in 8 *pagine*, ciascuna grande 4 KiB (1000 in esadecimale);
- Riserviamo per il sistema le pagine 0 e 1.

- Supponiamo per semplicità che lo spazio di indirizzamento virtuale vada da 0000 a 7fff (32 KiB).
- Immaginiamo di suddividere lo spazio di indirizzamento in 8 *pagine*, ciascuna grande 4 KiB (1000 in esadecimale);
- Riserviamo per il sistema le pagine 0 e 1.
- Il nostro programma contiene il codice nella pagina che va da 2000 a 2fff (pagina 2) e la variabile `buf` nelle due pagine successive (3 e 4). Usiamo l'ultima pagina (7) come pila.

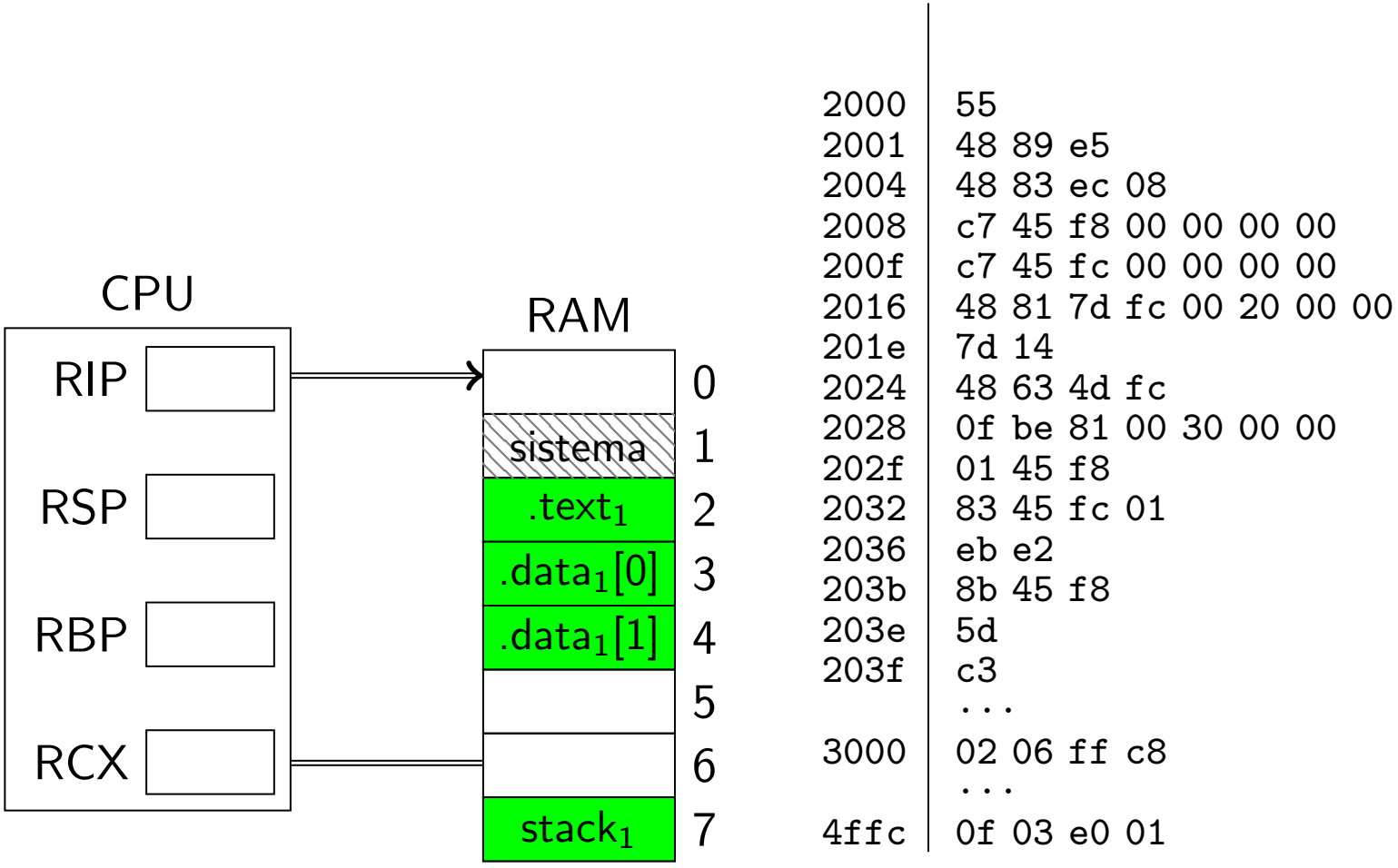
Una possible traduzione in assembly e linguaggio macchina è:

		.text	
		.global main	
2000	main:	pushq %rbp	55
2001		movq %rsp, %rbp	48 89 e5
2004		subq $8, %rsp	48 83 ec 08
2008		movl $0, -8(%rbp)	c7 45 f8 00 00 00 00
200f		movl $0, -4(%rbp)	c7 45 fc 00 00 00 00
2016	for:	cmpq $0x2000, -4(%rbp)	48 81 7d fc 00 20 00 00
201e		jge fine	7d 14
2024		movslq -4(%rbp), %rcx	48 63 4d fc
2028		movsbl buf(%rcx), %eax	0f be 81 00 30 00 00
202f		addl %eax, -8(%rbp)	01 45 f8
2032		addl $1, -4(%rbp)	83 45 fc 01
2036		jmp for	eb e2
203b	fine:	movl -8(%rbp), %eax	8b 45 f8
203e		popq %rbp	5d
203f		ret	c3
		.data	...
3000	buf:	.byte 2, 6, -1, 200	02 06 ff c8
		...	...
4ffc		.byte 15, 3, -32, 1	0f 03 e0 01

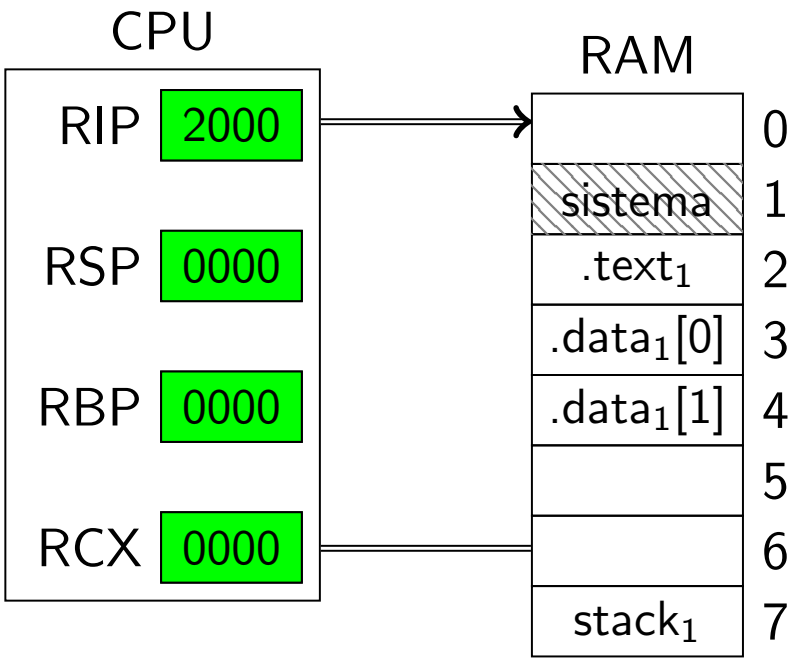



```
2000 | 55
2001 | 48 89 e5
2004 | 48 83 ec 08
2008 | c7 45 f8 00 00 00 00
200f | c7 45 fc 00 00 00 00
2016 | 48 81 7d fc 00 20 00 00
201e | 7d 14
2024 | 48 63 4d fc
2028 | 0f be 81 00 30 00 00
202f | 01 45 f8
2032 | 83 45 fc 01
2036 | eb e2
203b | 8b 45 f8
203e | 5d
203f | c3
     | ...
3000 | 02 06 ff c8
     | ...
4ffc | 0f 03 e0 01
```

Vediamo prima l'esecuzione su un sistema non multiprogrammato. Mostriamo solo alcuni dei registri della CPU. Tutti i numeri saranno mostrati in esadecimale. Supponiamo che tutto lo spazio di memoria sia occupato dalla RAM.

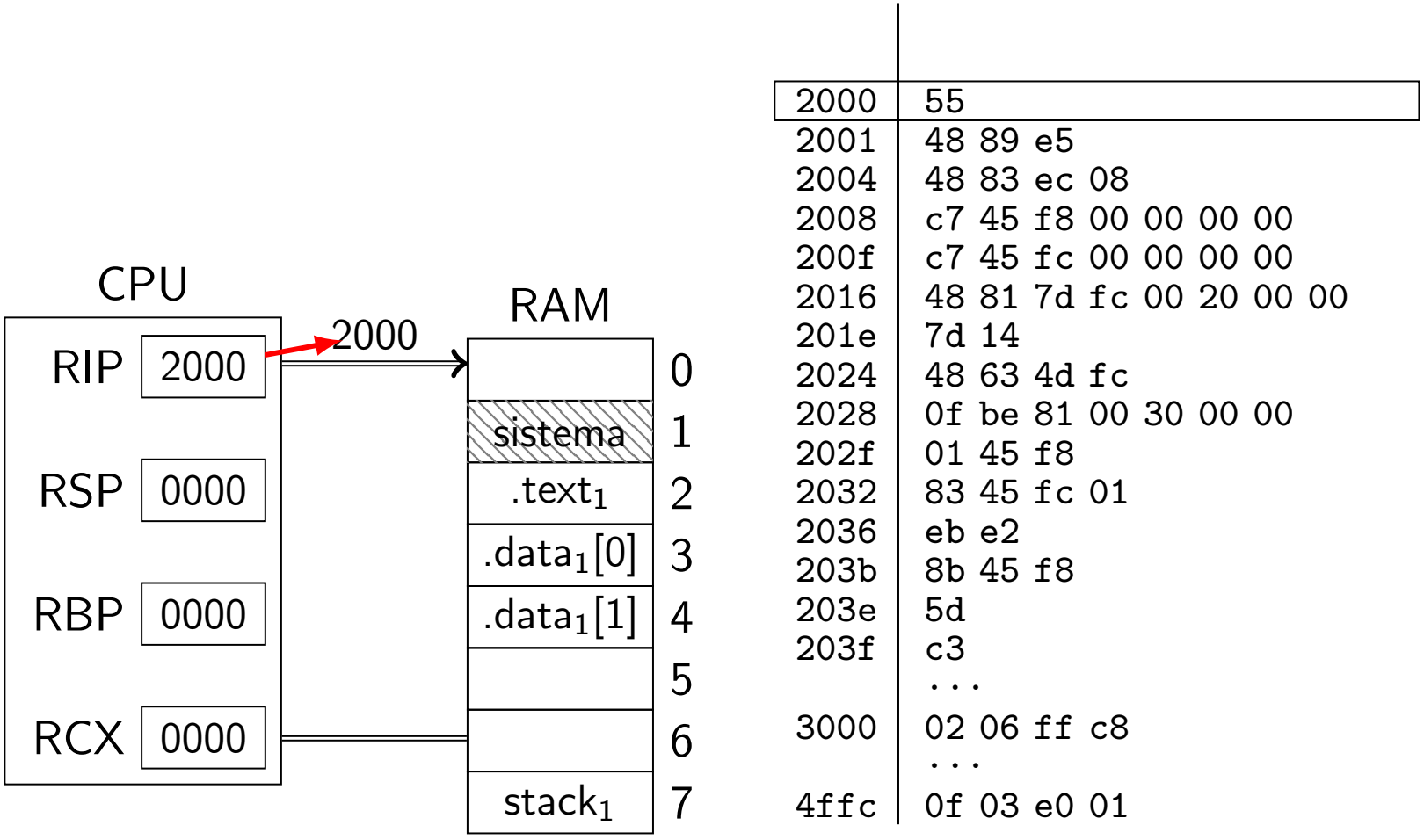


Carichiamo il programma in memoria.

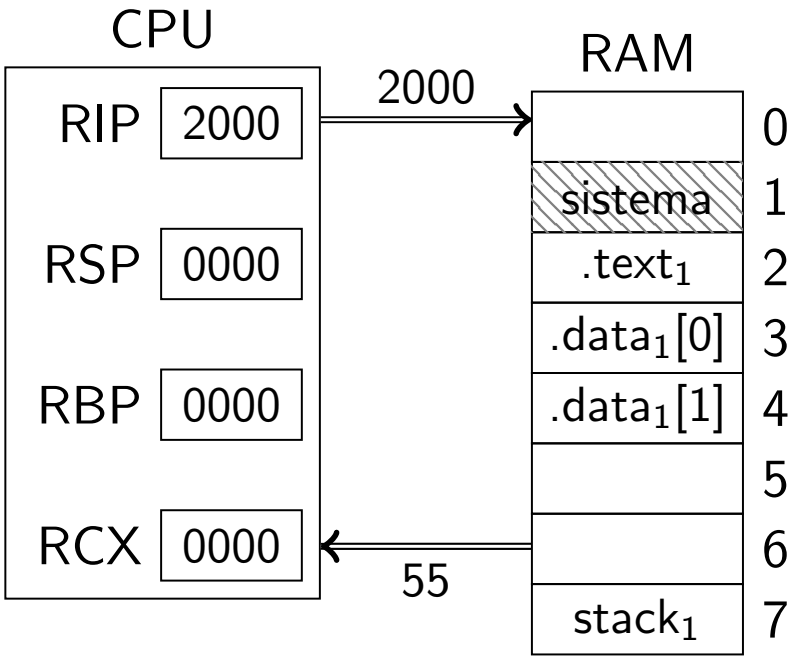



```
2000 | 55
2001 | 48 89 e5
2004 | 48 83 ec 08
2008 | c7 45 f8 00 00 00 00
200f | c7 45 fc 00 00 00 00
2016 | 48 81 7d fc 00 20 00 00
201e | 7d 14
2024 | 48 63 4d fc
2028 | 0f be 81 00 30 00 00
202f | 01 45 f8
2032 | 83 45 fc 01
2036 | eb e2
203b | 8b 45 f8
203e | 5d
203f | c3
     | ...
3000 | 02 06 ff c8
     | ...
4ffc | 0f 03 e0 01
```

Inizializziamo tutti i registri

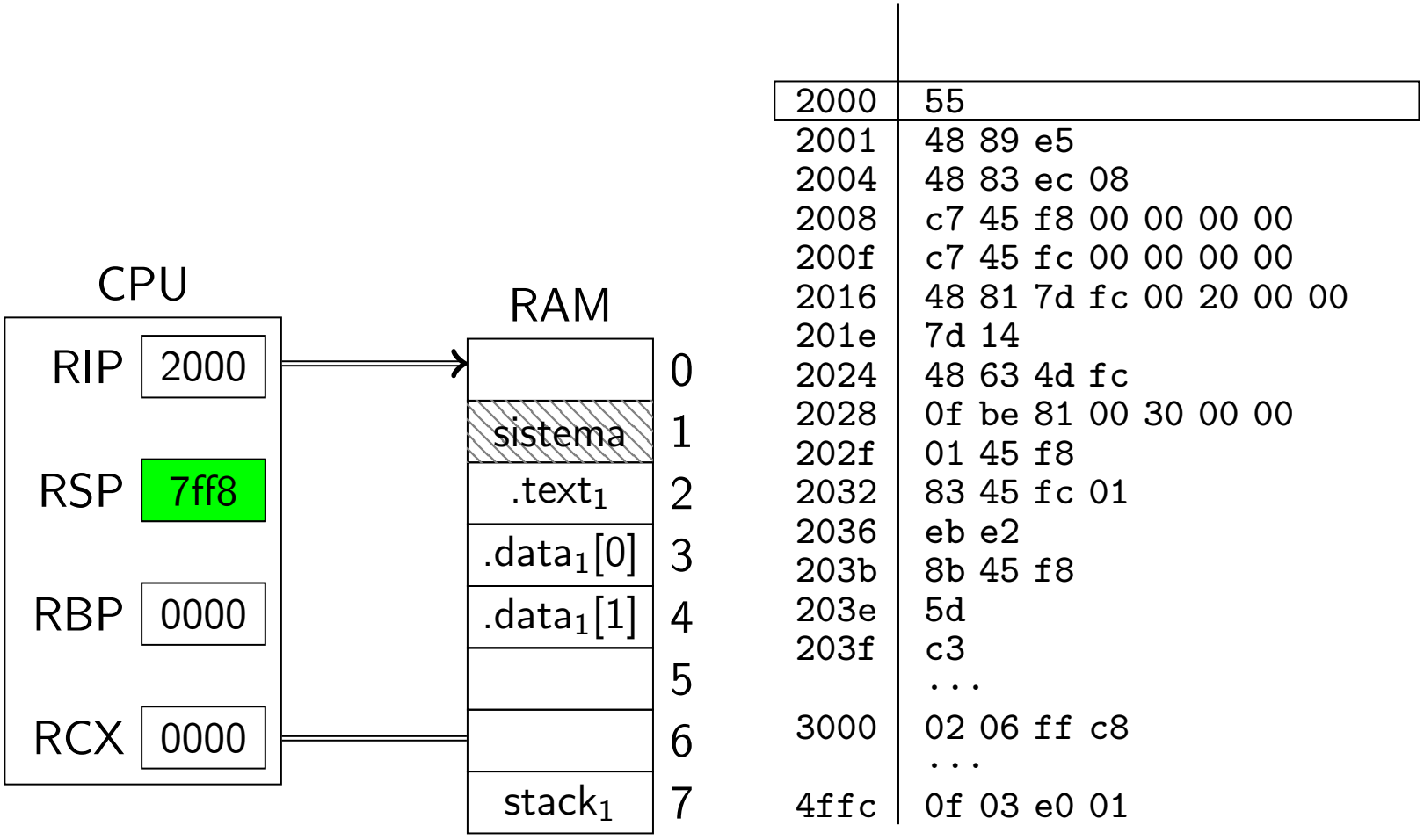


Avviamo il sistema. La CPU tenta di prelevare l'istruzione all'indirizzo contenuto in RIP. Inizia quindi una operazione di lettura in memoria all'indirizzo 2000.

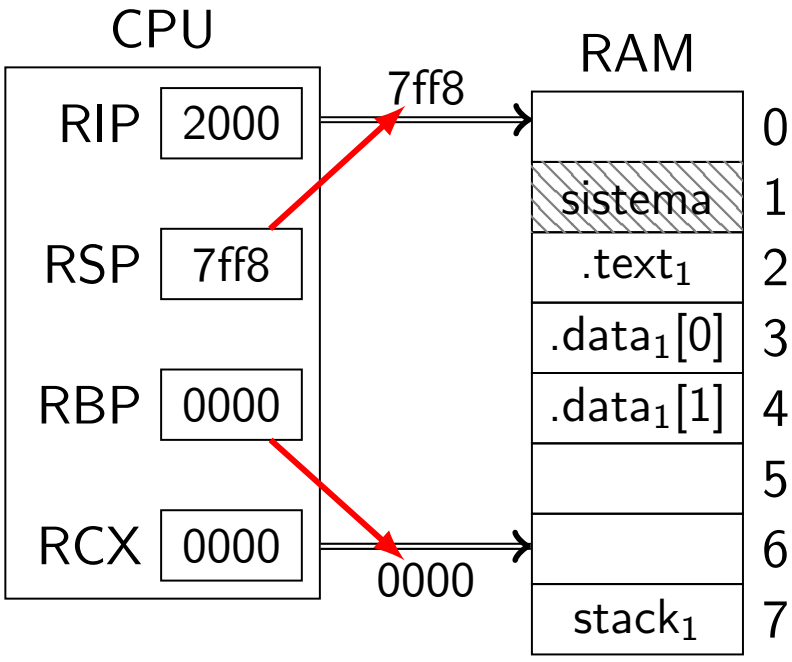


2000	55
2001	48 89 e5
2004	48 83 ec 08
2008	c7 45 f8 00 00 00 00
200f	c7 45 fc 00 00 00 00
2016	48 81 7d fc 00 20 00 00
201e	7d 14
2024	48 63 4d fc
2028	0f be 81 00 30 00 00
202f	01 45 f8
2032	83 45 fc 01
2036	eb e2
203b	8b 45 f8
203e	5d
203f	c3
	...
3000	02 06 ff c8
	...
4ffc	0f 03 e0 01

L'operazione di lettura restituisce l'istruzione `pushq %rbp`.

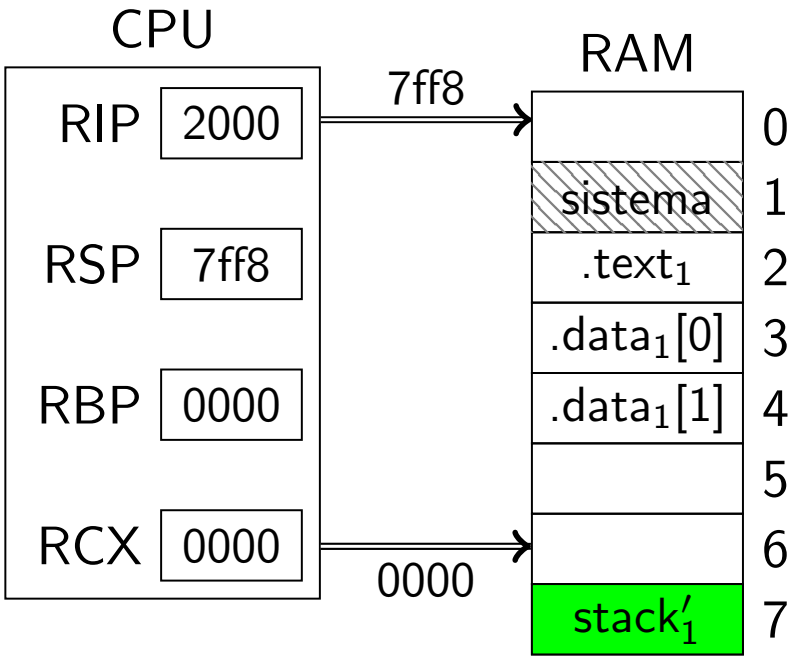


La CPU inizia ad eseguire l'istruzione. Il primo passo è decrementare RSP di 8.



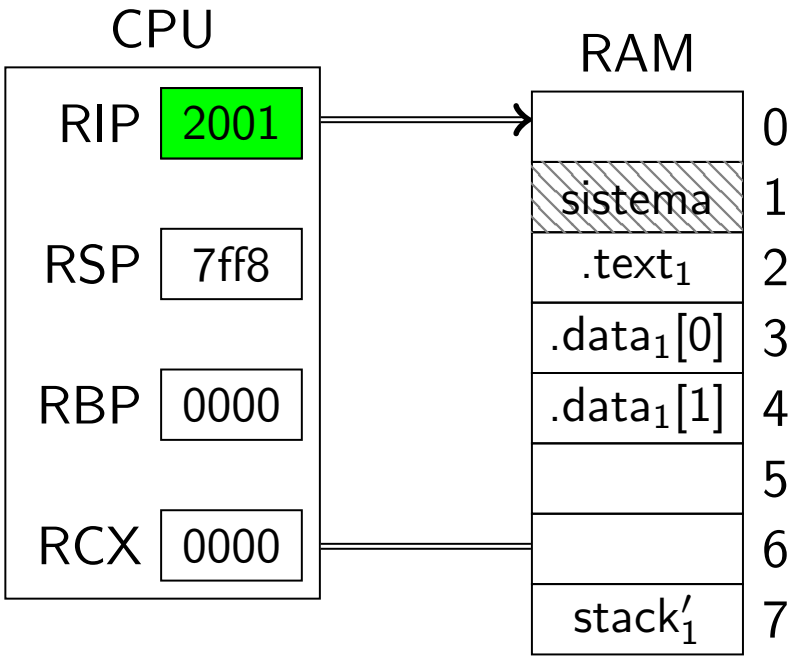

```
2000 | 55
2001 | 48 89 e5
2004 | 48 83 ec 08
2008 | c7 45 f8 00 00 00 00
200f | c7 45 fc 00 00 00 00
2016 | 48 81 7d fc 00 20 00 00
201e | 7d 14
2024 | 48 63 4d fc
2028 | 0f be 81 00 30 00 00
202f | 01 45 f8
2032 | 83 45 fc 01
2036 | eb e2
203b | 8b 45 f8
203e | 5d
203f | c3
     | ...
3000 | 02 06 ff c8
     | ...
4ffc | 0f 03 e0 01
```

Il secondo passo è scrivere il contenuto di RBP all'indirizzo contenuto in RSP. La CPU inizia quindi una operazione di scrittura all'indirizzo 7ff8.



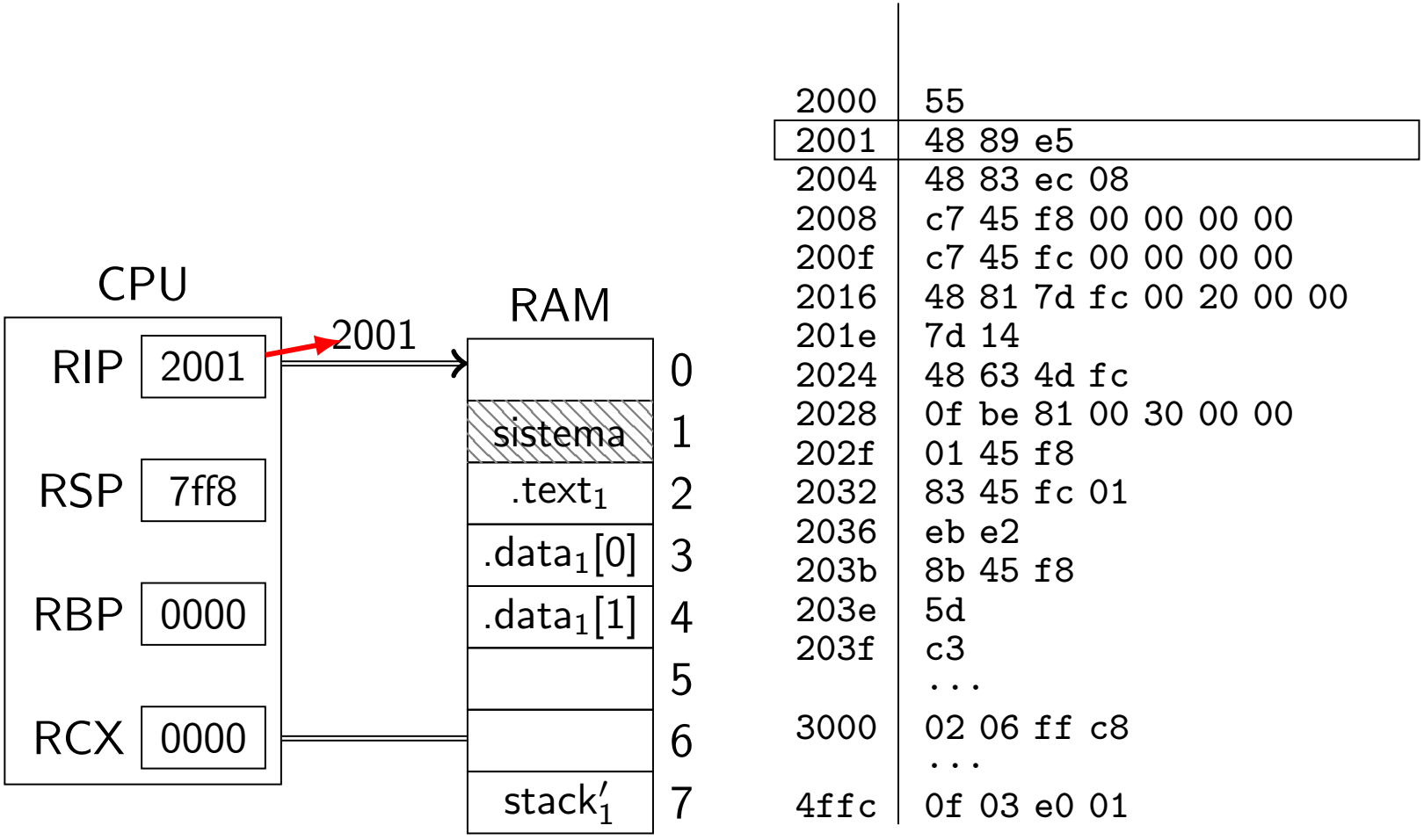

```
2000 | 55
2001 | 48 89 e5
2004 | 48 83 ec 08
2008 | c7 45 f8 00 00 00 00
200f | c7 45 fc 00 00 00 00
2016 | 48 81 7d fc 00 20 00 00
201e | 7d 14
2024 | 48 63 4d fc
2028 | 0f be 81 00 30 00 00
202f | 01 45 f8
2032 | 83 45 fc 01
2036 | eb e2
203b | 8b 45 f8
203e | 5d
203f | c3
     | ...
3000 | 02 06 ff c8
     | ...
4ffc | 0f 03 e0 01
```

Dopo la scrittura il contenuto dello stack sarà cambiato.

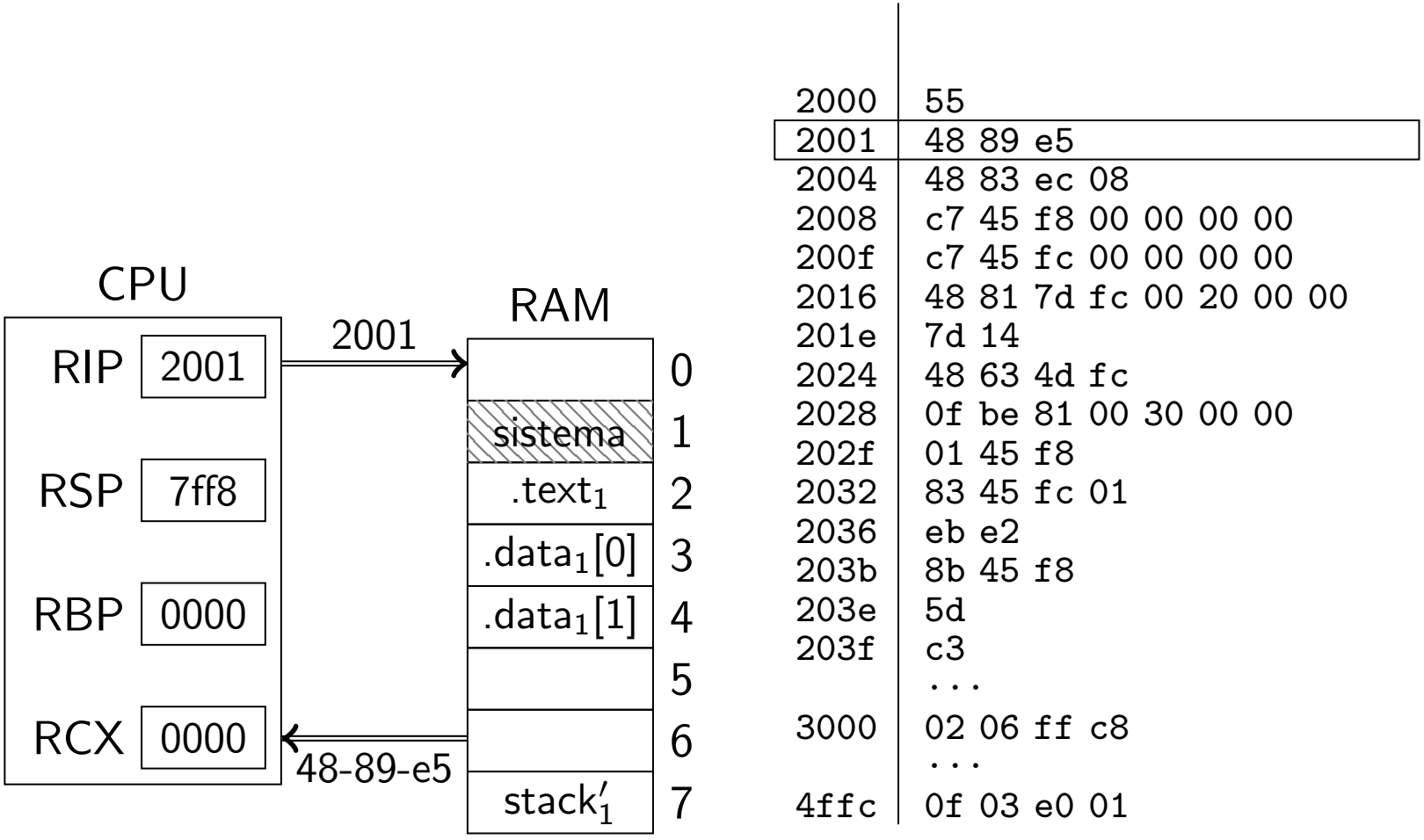



```
2000 | 55
2001 | 48 89 e5
2004 | 48 83 ec 08
2008 | c7 45 f8 00 00 00 00
200f | c7 45 fc 00 00 00 00
2016 | 48 81 7d fc 00 20 00 00
201e | 7d 14
2024 | 48 63 4d fc
2028 | 0f be 81 00 30 00 00
202f | 01 45 f8
2032 | 83 45 fc 01
2036 | eb e2
203b | 8b 45 f8
203e | 5d
203f | c3
     | ...
3000 | 02 06 ff c8
     | ...
4ffc | 0f 03 e0 01
```

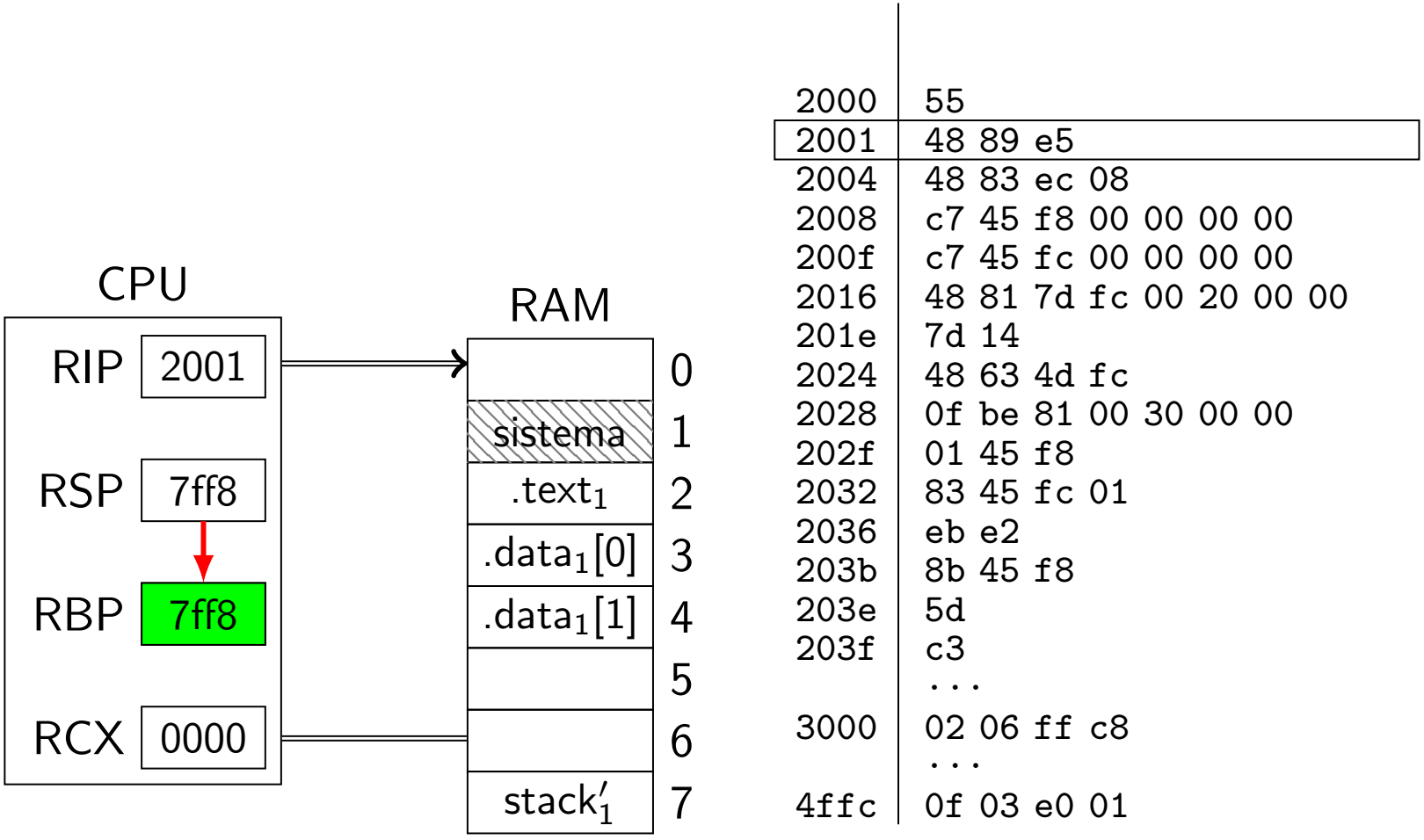
La CPU ha completato l'esecuzione dell'istruzione `pushq %rbp` e può passare alla successiva. Incrementa RIP di 1 (che è la dimensione dell'istruzione appena terminata).



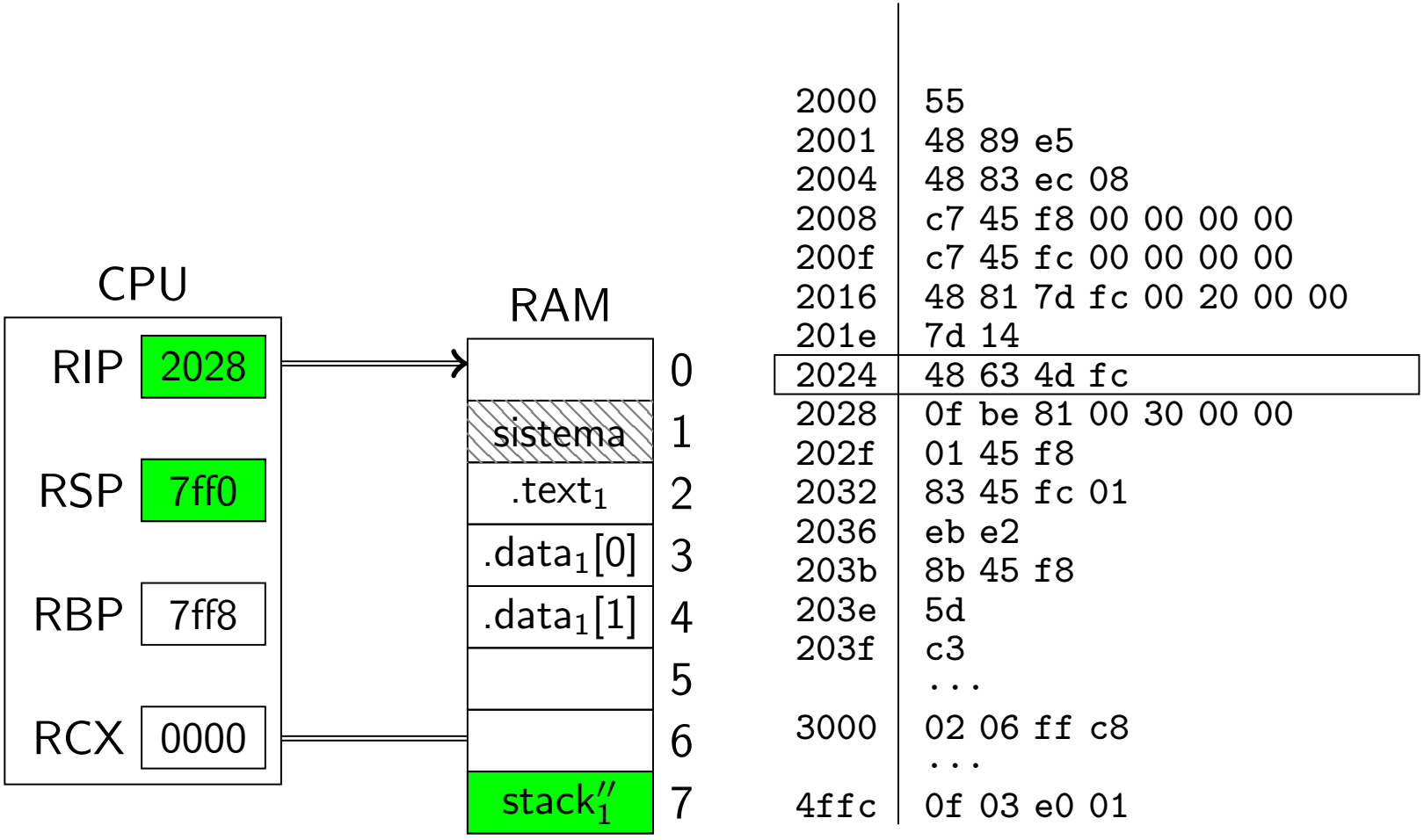
Nuovo ciclo: la CPU tenta di prelevare l'istruzione all'indirizzo contenuto in RIP. Inizia una operazione di lettura in memoria all'indirizzo 2001.



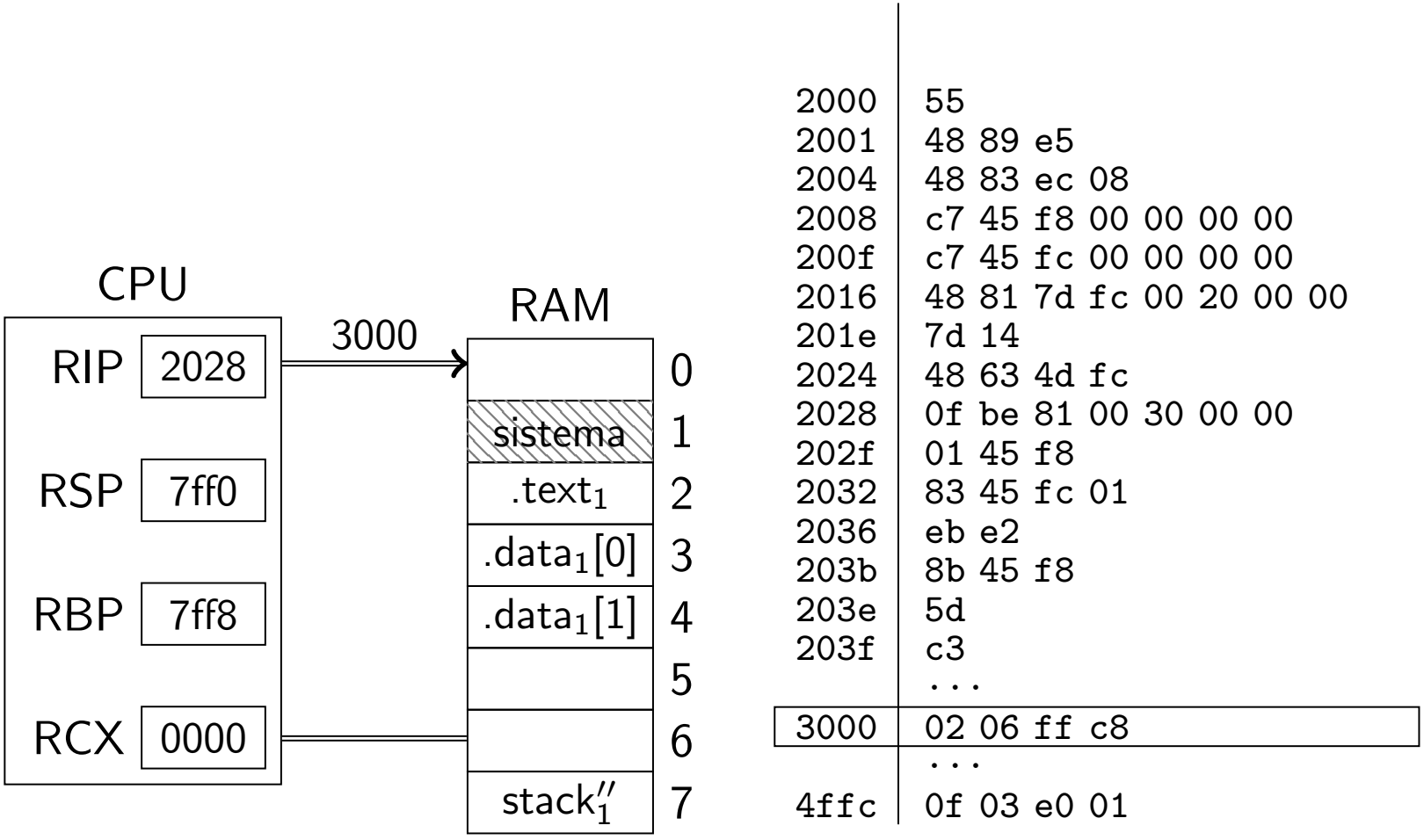
La CPU riceve l'istruzione `movq %rsp, %rbp`.



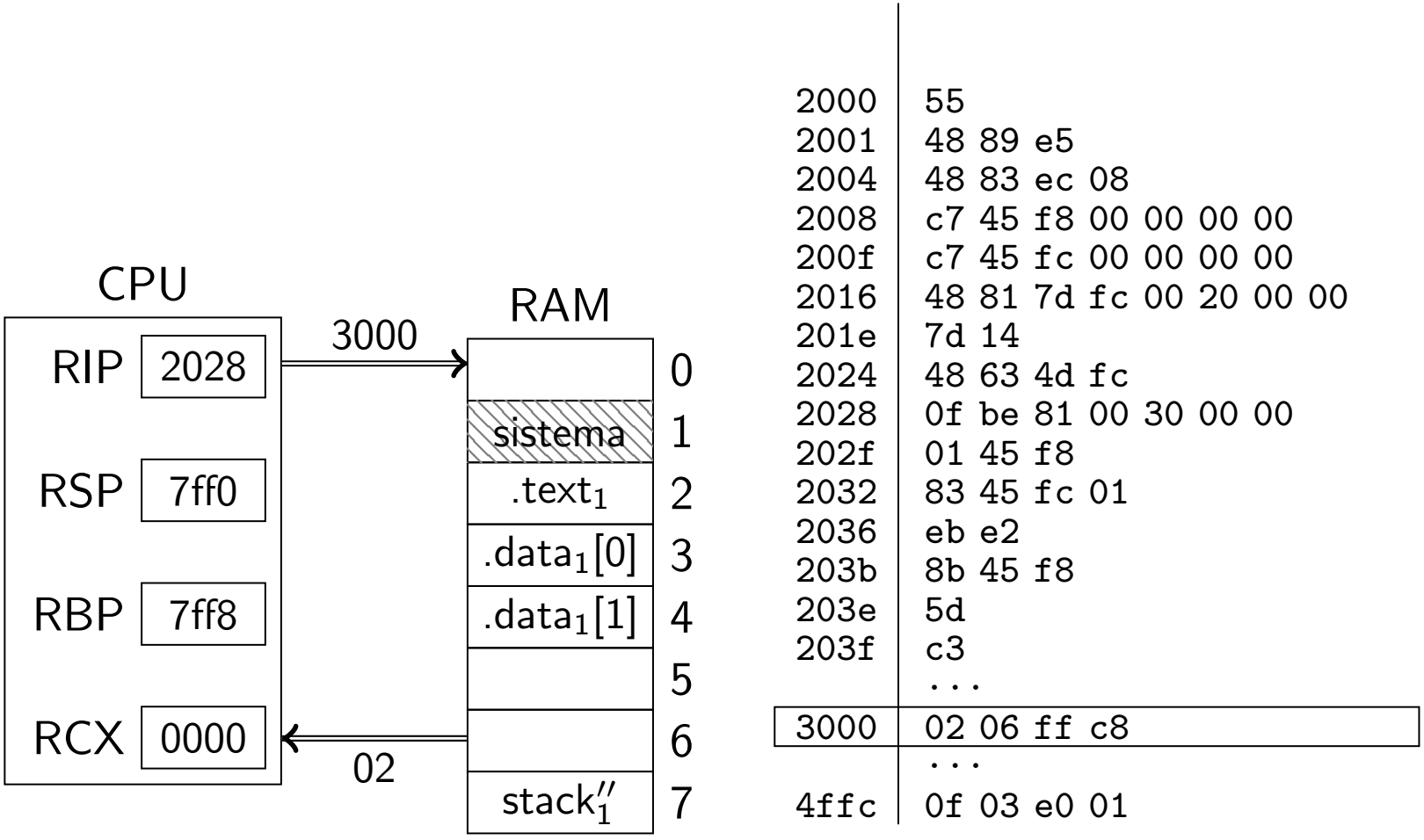
Per eseguirla copia il contenuto di RSP in RBP. L'istruzione non prevede accessi in memoria.



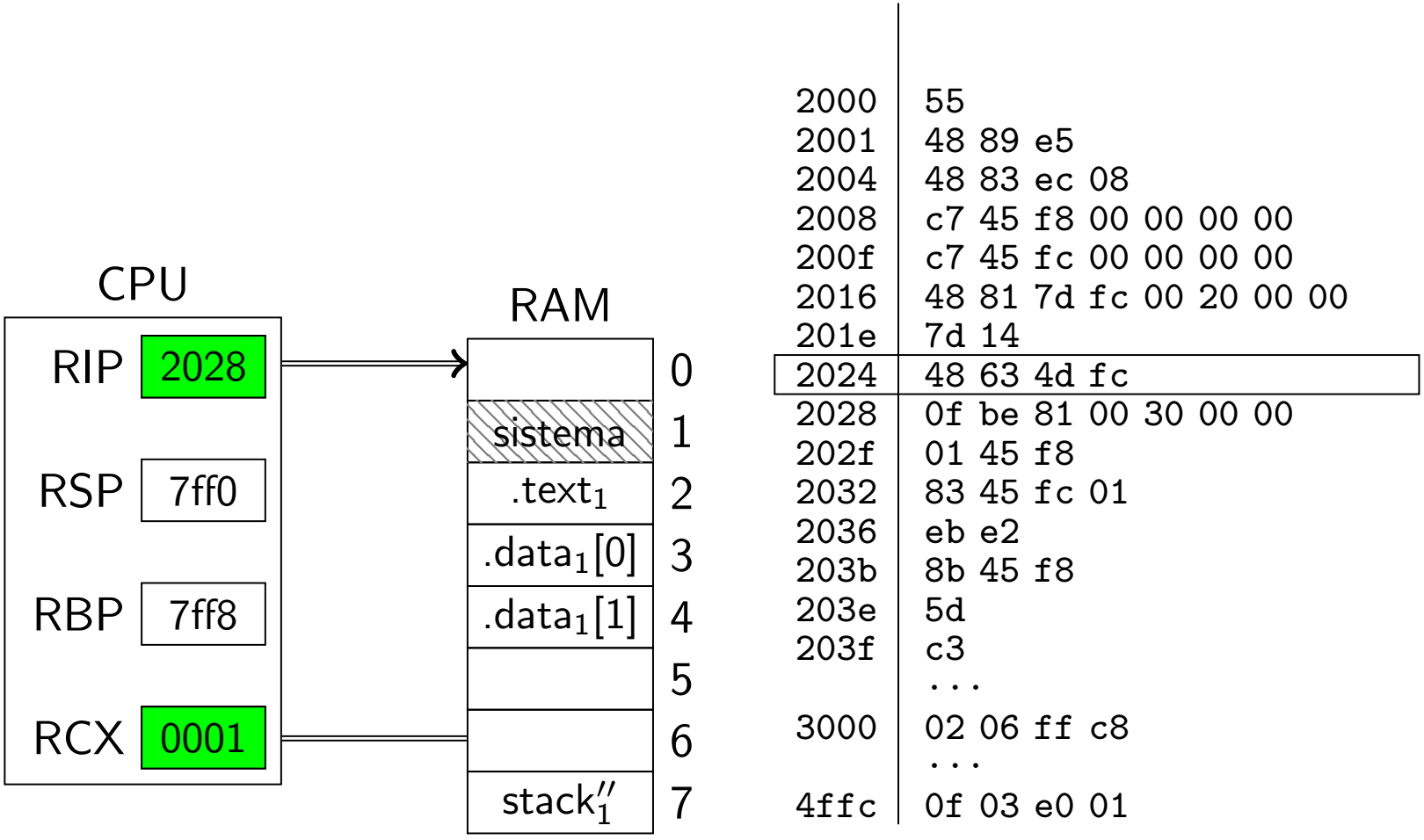
Dopo alcune istruzioni, l'esecuzione del programma arriva all'istruzione "`movsbl buf(%rcx), %rax`" che si trova all'indirizzo 2028. Questa è la prima istruzione che accede alla sezione `.data` del programma.



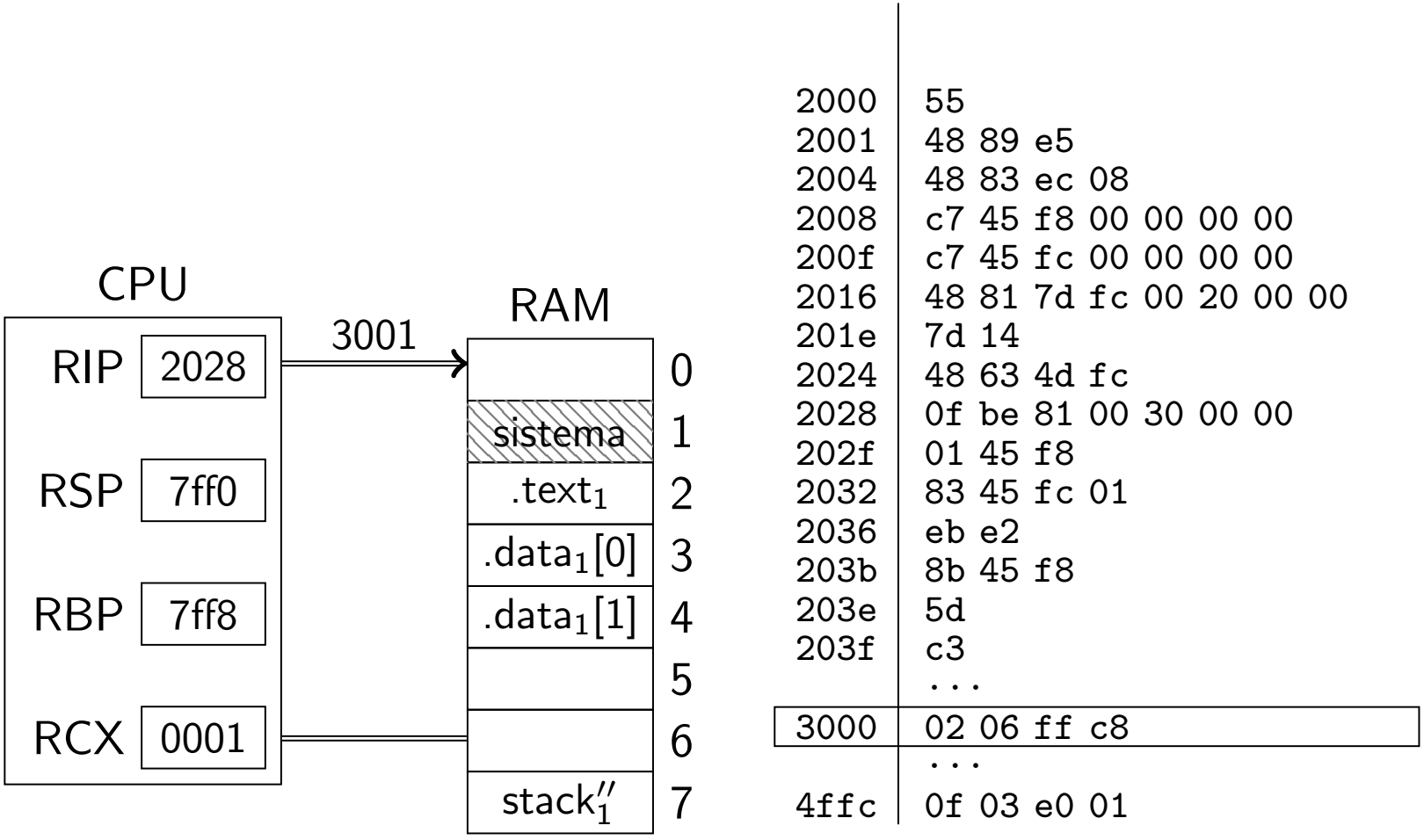
La CPU somma il contenuto di RCX e la costante 3000 (buf nel sorgente), ottenendo l'indirizzo 3000. Inizia dunque una operazione di lettura a questo indirizzo.



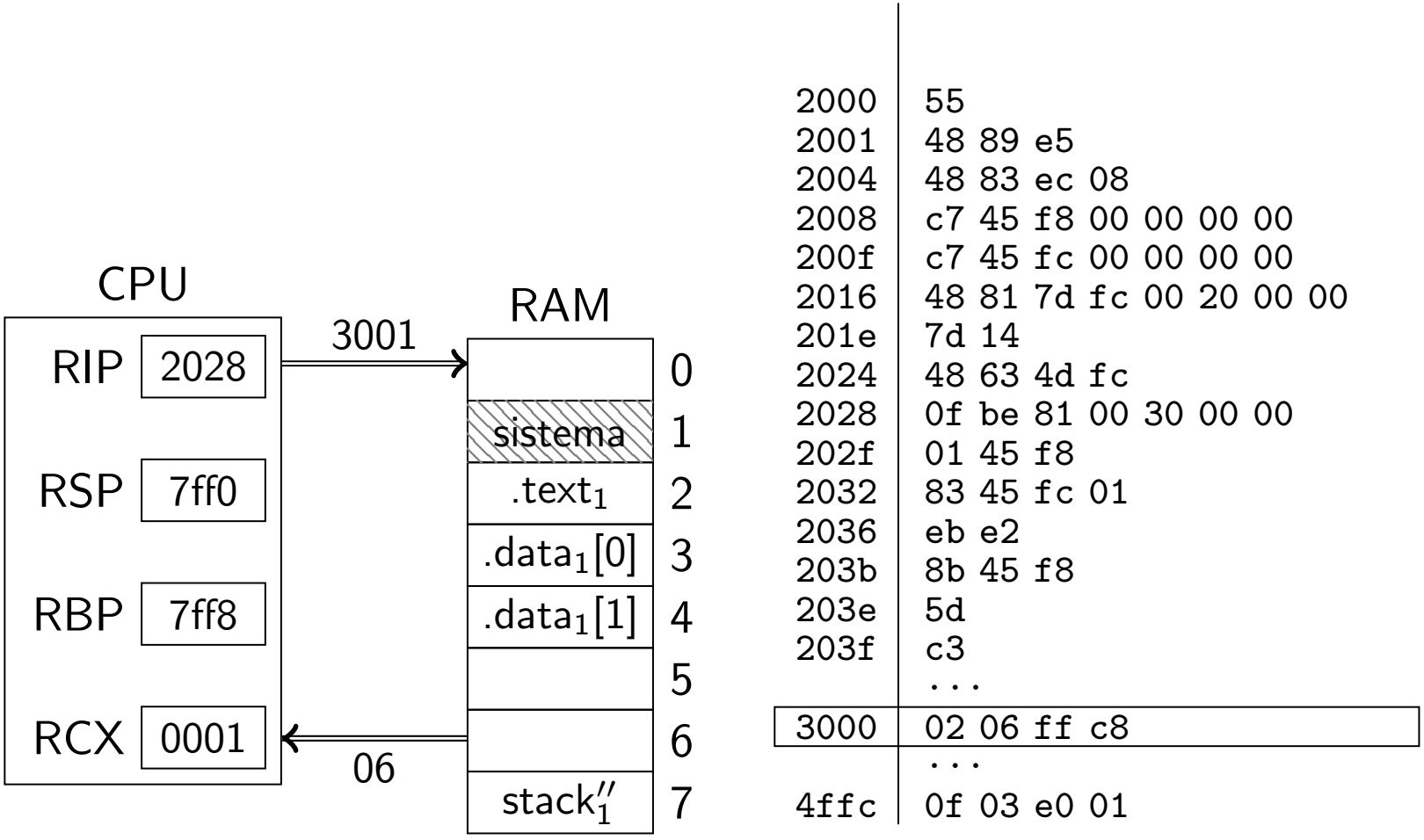
La CPU riceve il primo byte contenuto nel buffer (valore 2) e lo copia nel registro EAX (non mostrato).



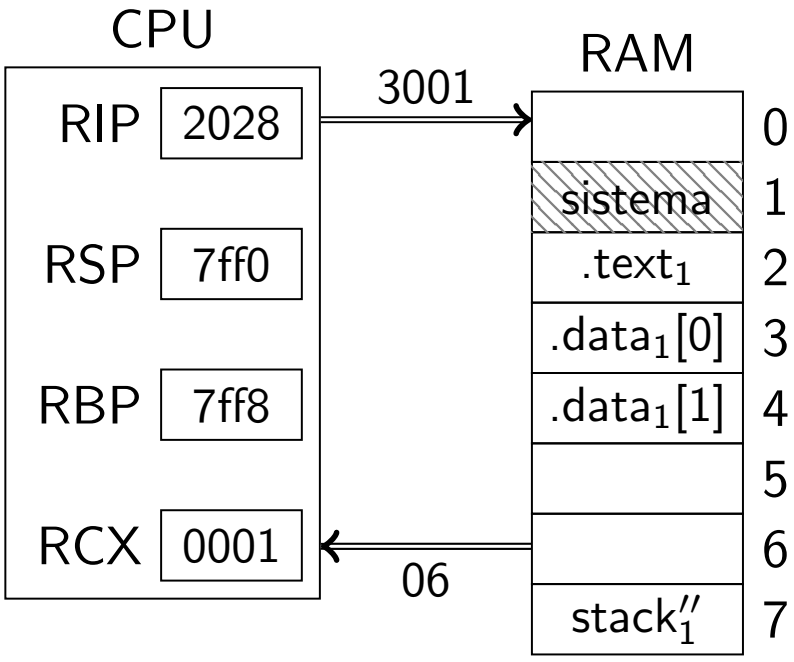
L'esecuzione prosegue sommando EAX in `-8(%rbp)`, incrementando RCX, etc., fino a quando si torna all'indirizzo 2028. La CPU preleva nuovamente l'istruzione "`movsbl buf(%rcx), %rax`".



La CPU somma il contenuto di RCX e la costante 3000 (buf nel sorgente), ottenendo l'indirizzo 3001. Inizia dunque una operazione di lettura a questo indirizzo.

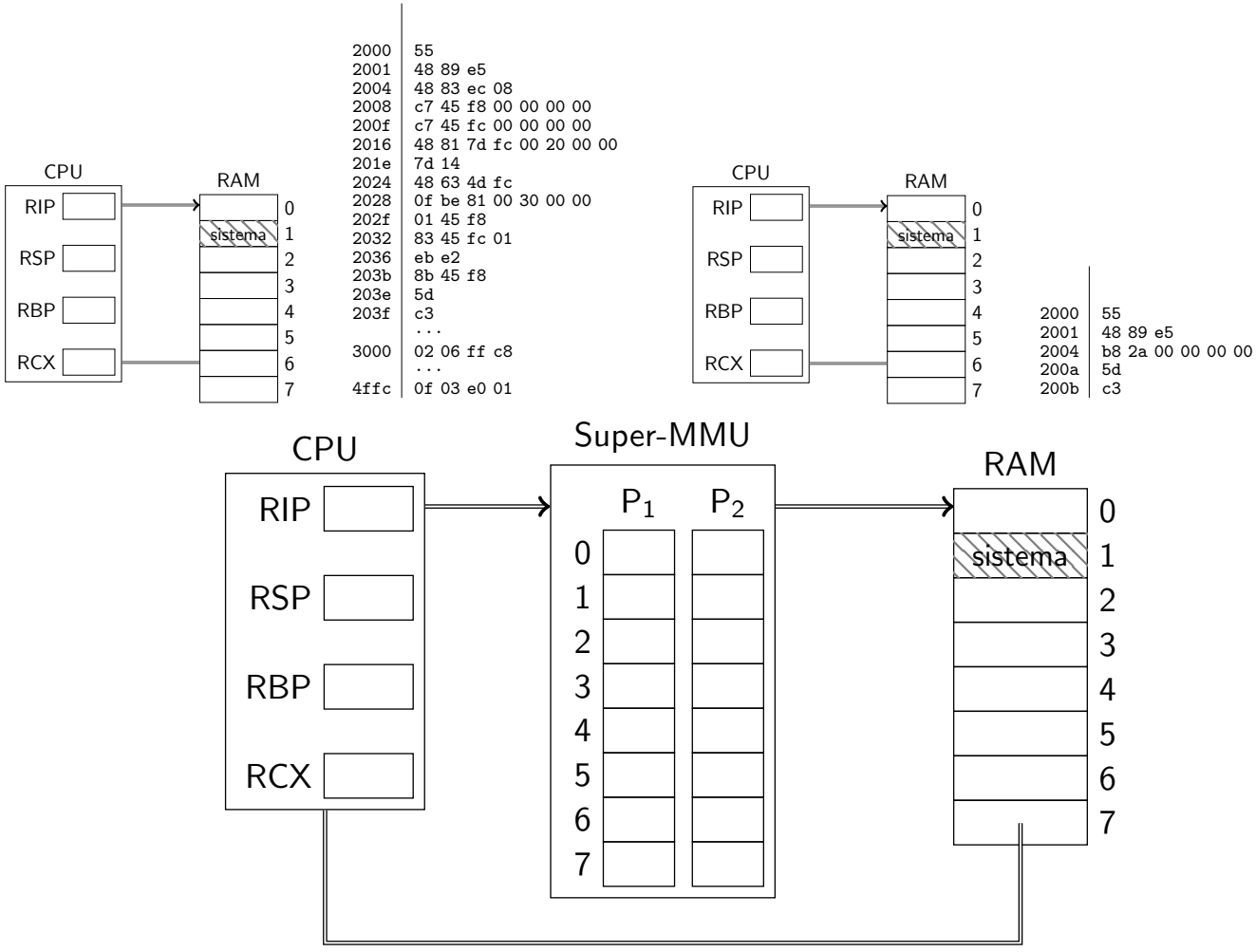


La cpu legge il valore 6 e lo somma al registro EAX (non mostrato).

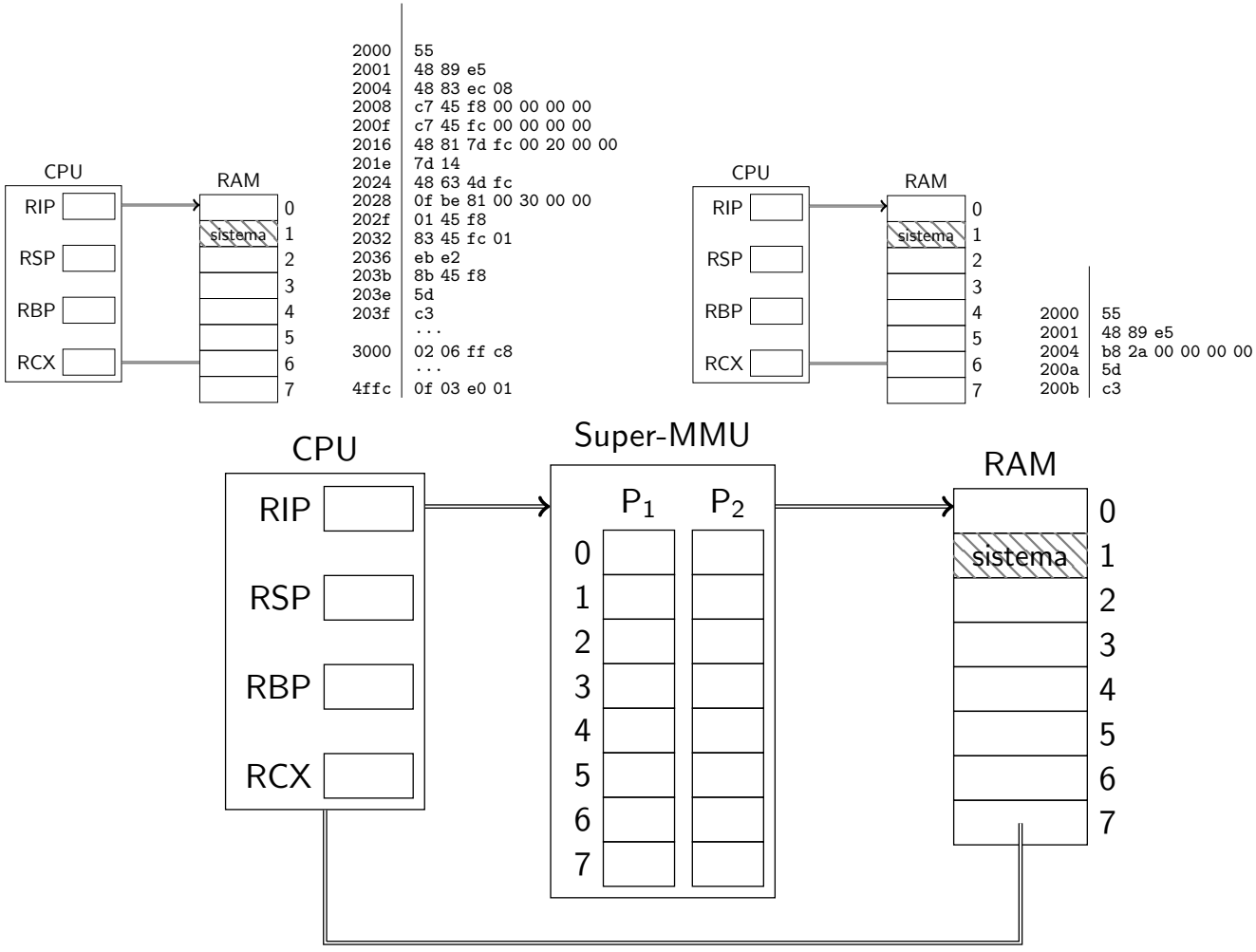



```
2000 | 55
2001 | 48 89 e5
2004 | 48 83 ec 08
2008 | c7 45 f8 00 00 00 00
200f | c7 45 fc 00 00 00 00
2016 | 48 81 7d fc 00 20 00 00
201e | 7d 14
2024 | 48 63 4d fc
2028 | 0f be 81 00 30 00 00
202f | 01 45 f8
2032 | 83 45 fc 01
2036 | eb e2
203b | 8b 45 f8
203e | 5d
203f | c3
     | ...
3000 | 02 06 ff c8
     | ...
4ffc | 0f 03 e0 01
```

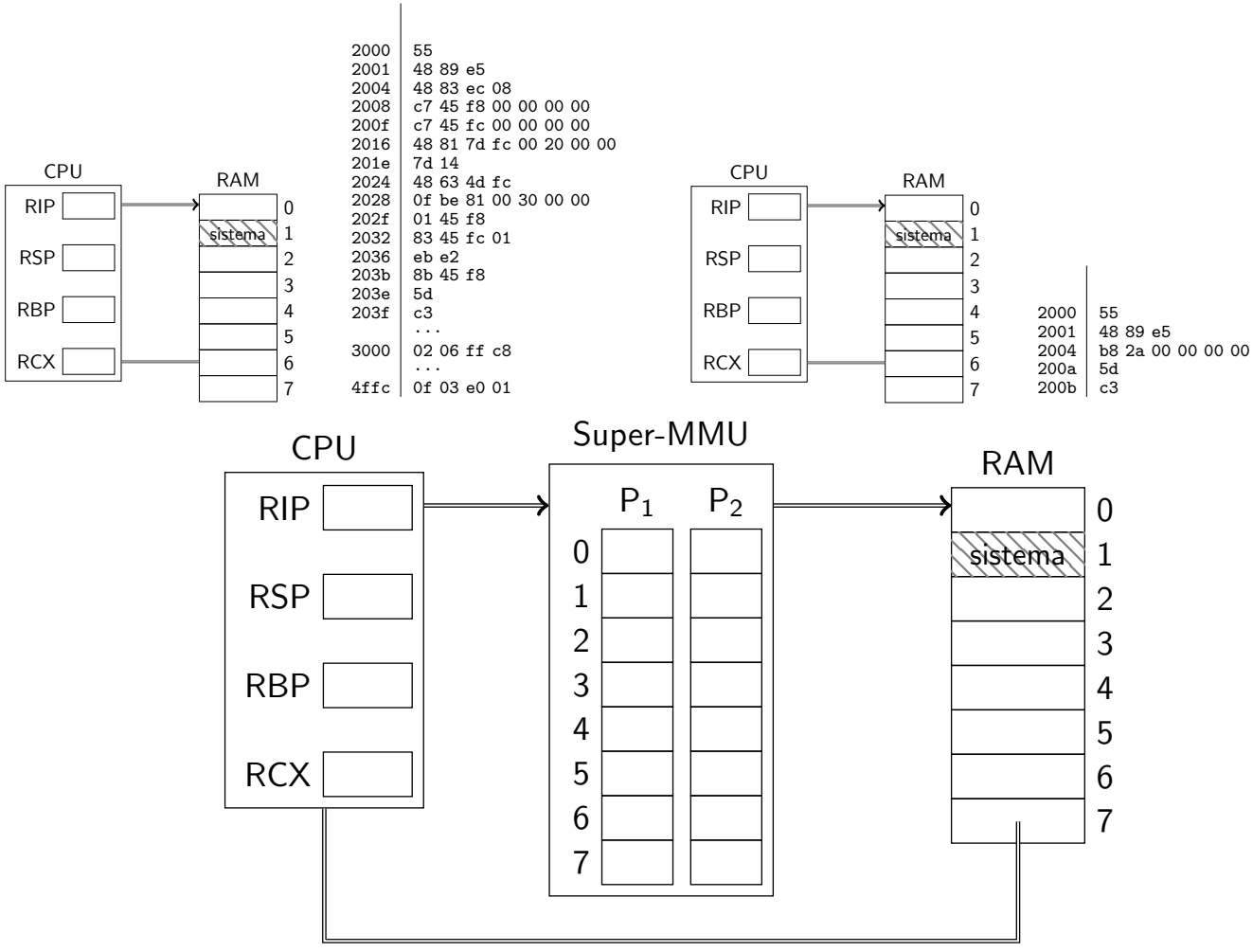
L'esecuzione prosegue in questo modo, sommando tutti i valori di `buf`.



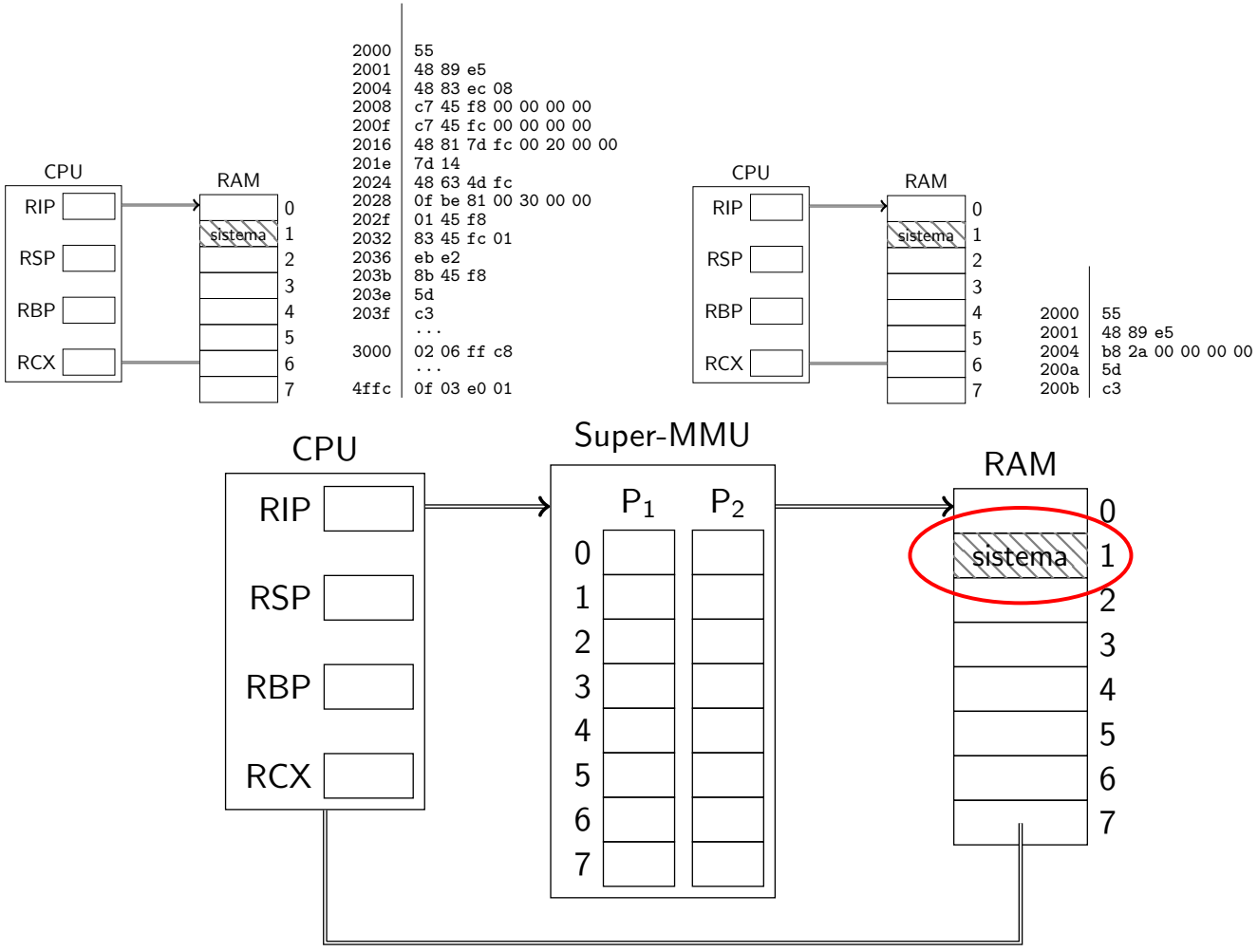
Possiamo immaginare che ciascun processo sia eseguito da un distinto sistema virtuale, con una sua CPU e una sua RAM.



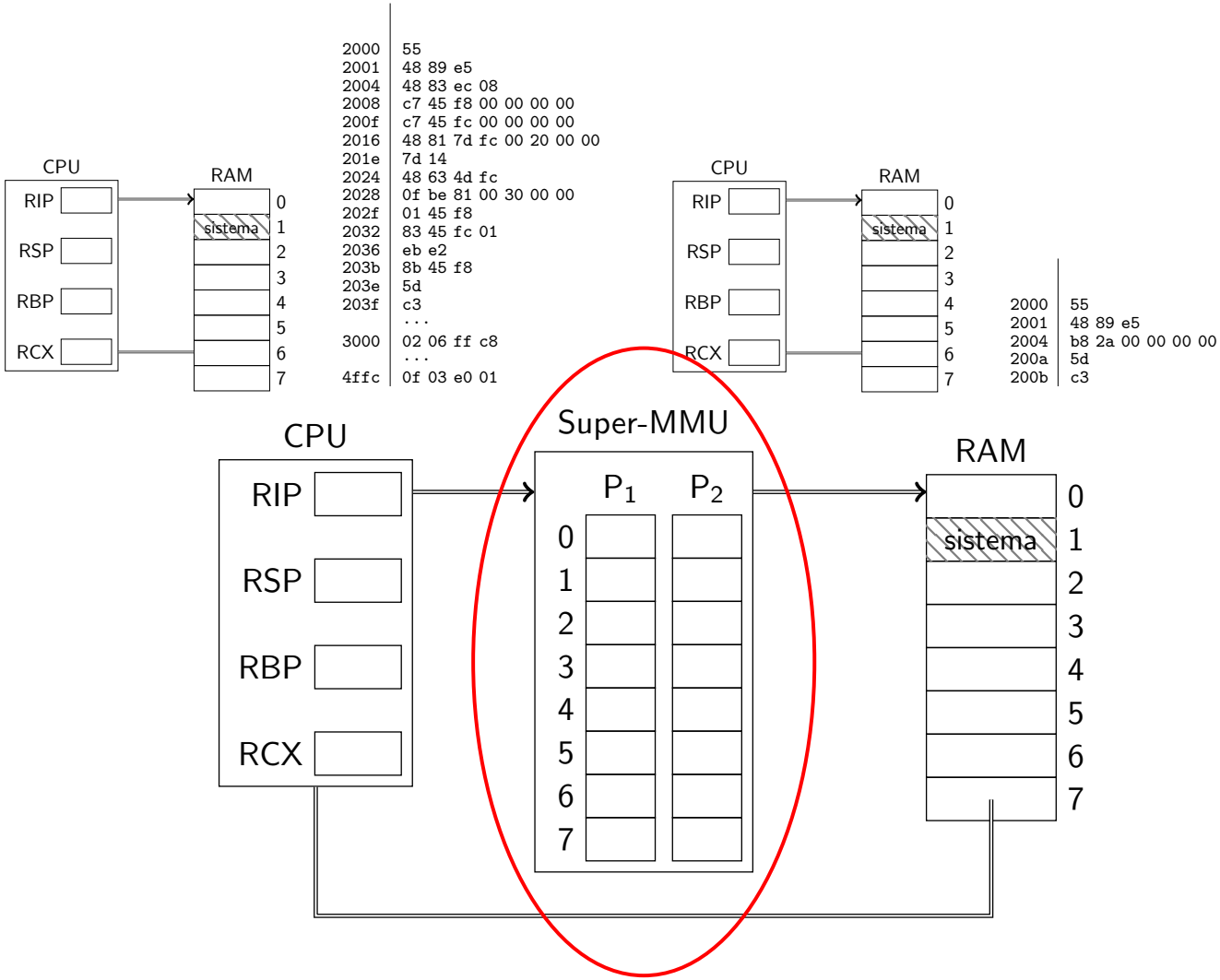
Nella figura, in alto, mostriamo a sinistra lo stato del sistema virtuale che esegue P_1, e a destra quello di un diverso processo P_2.



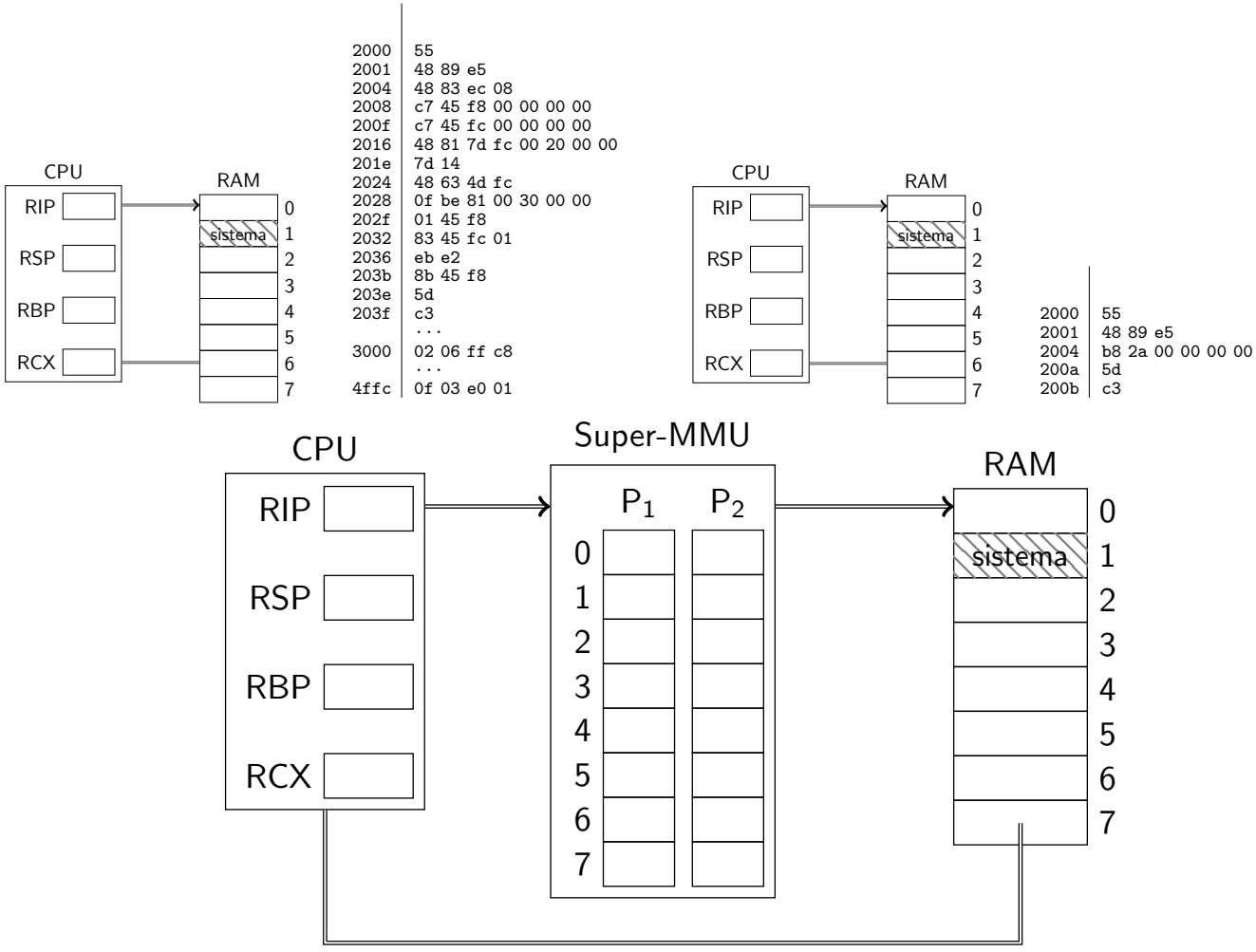
In basso mostriamo invece il sistema fisico, che deve emulare l'evoluzione dei due sistemi virtuali, a divisione di tempo.



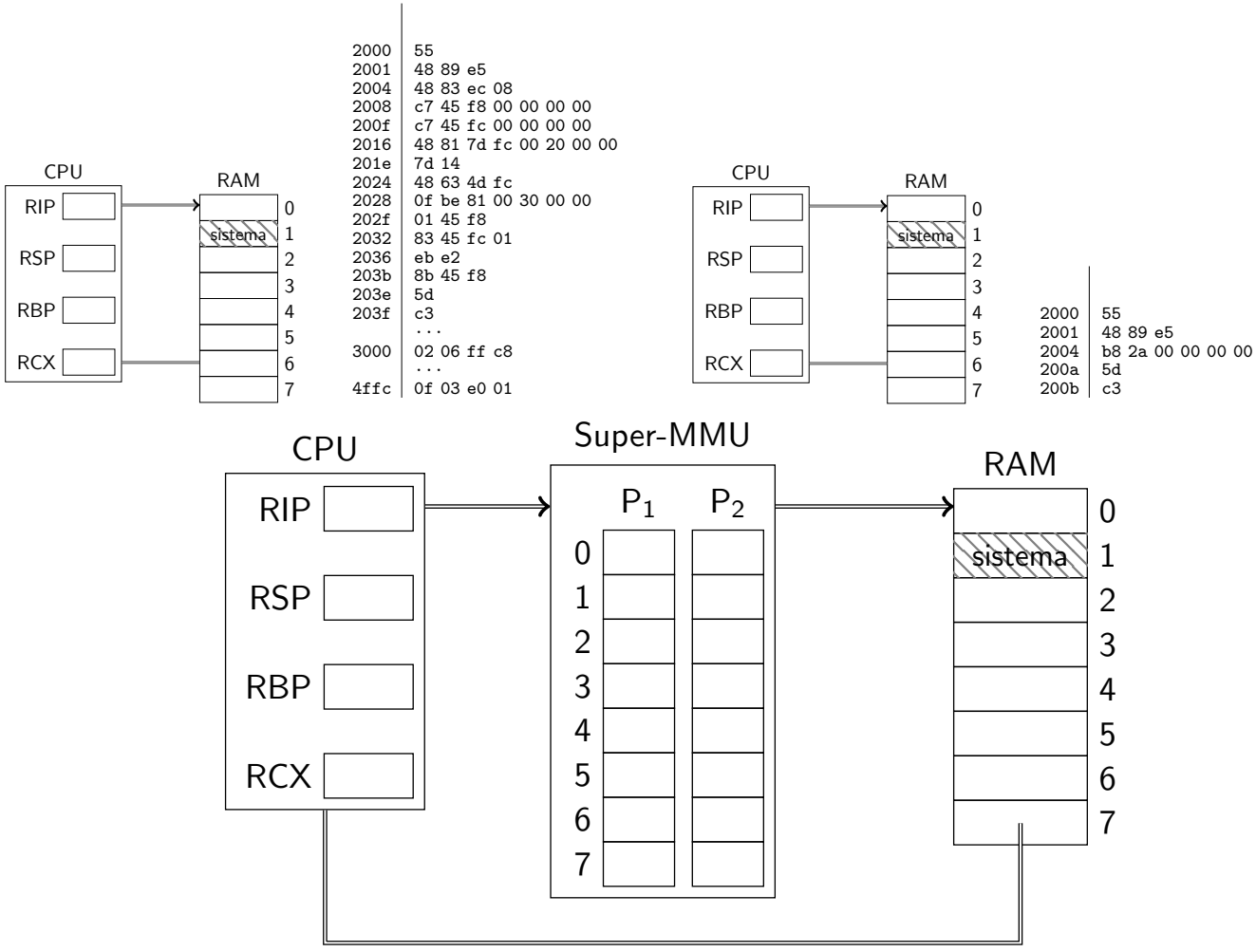
Lo stato delle due CPU virtuali è memorizzato nei descrittori di processo, nella memoria di sistema.



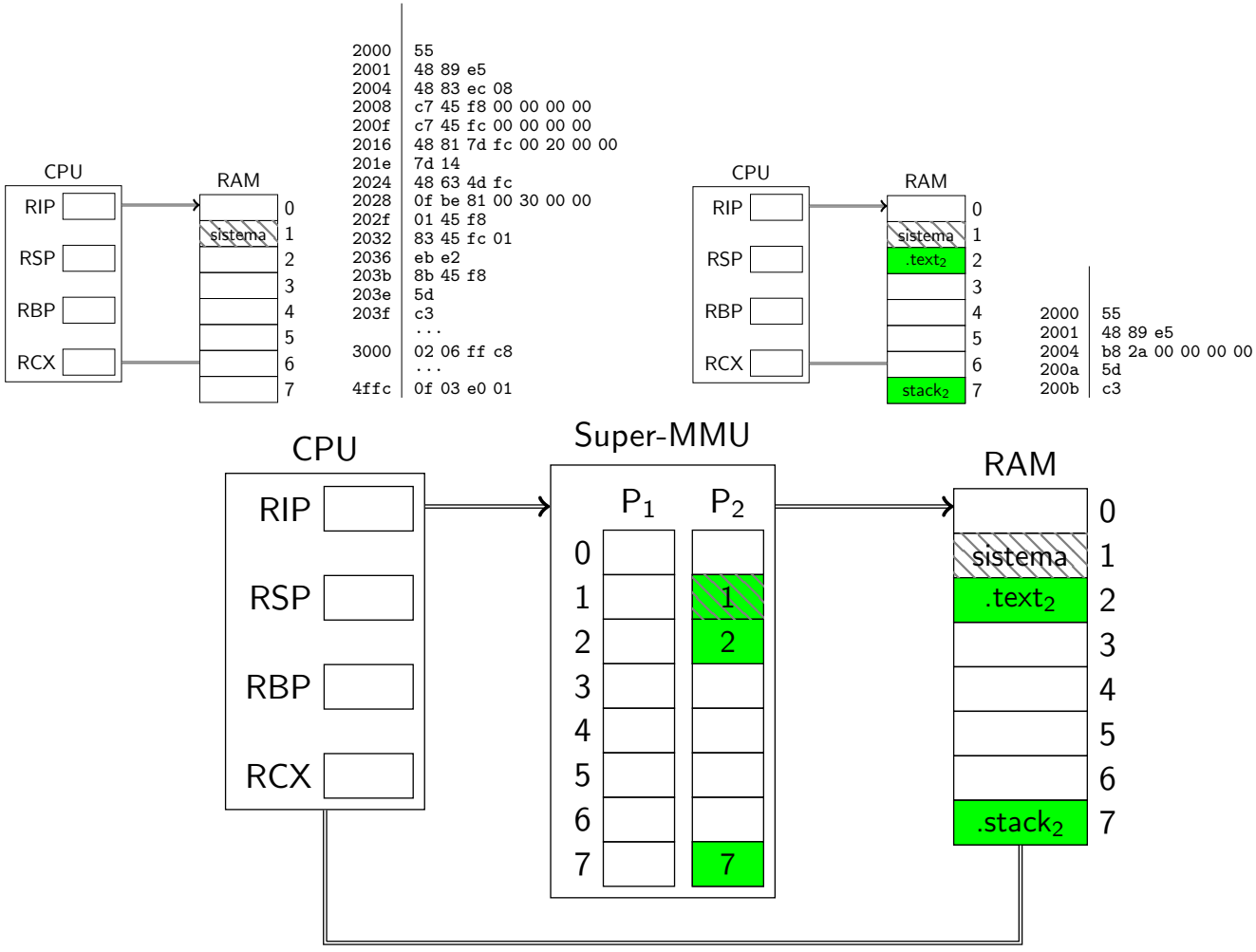
La MMU verrà invece usata per emulare le due memorie virtuali.



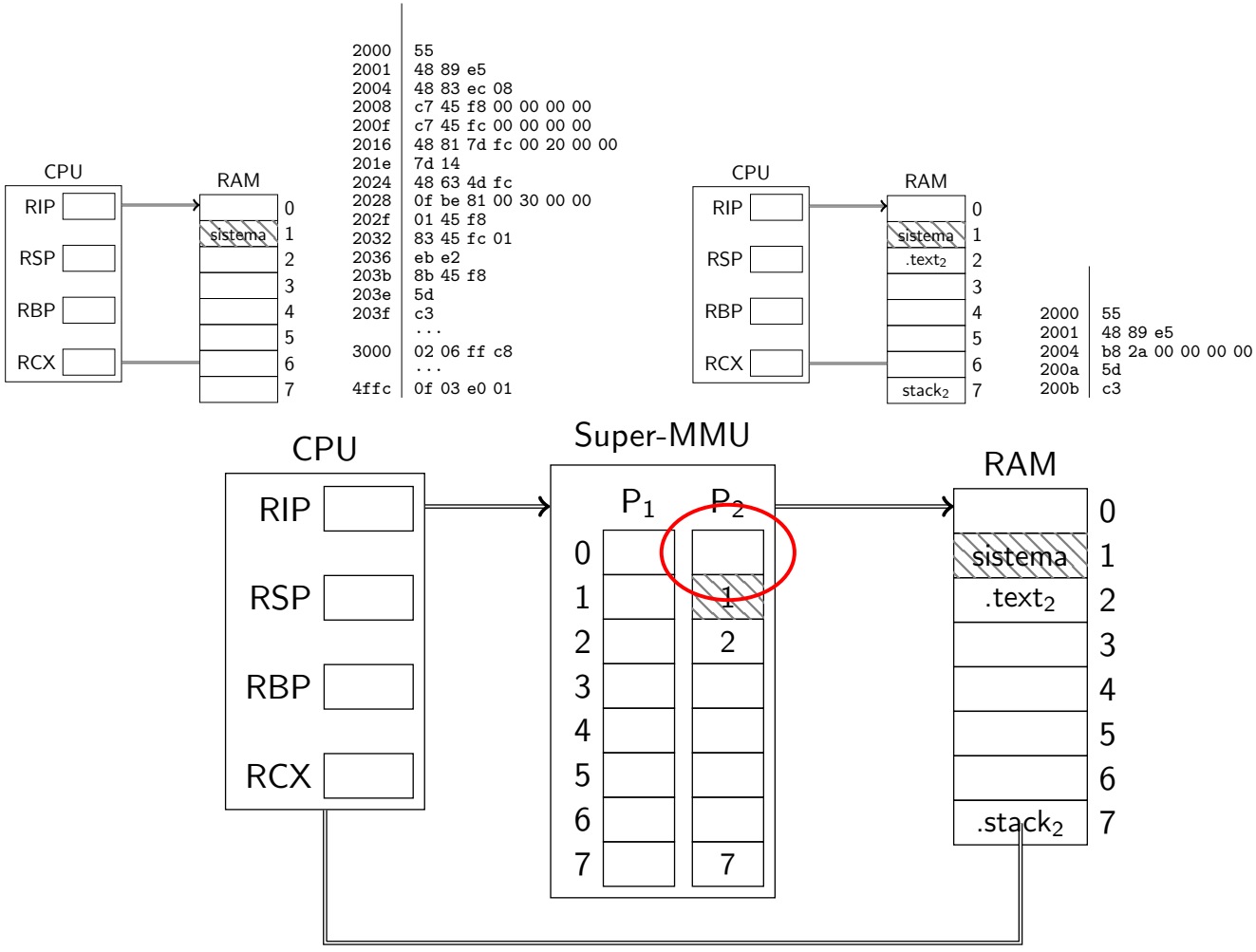
Vogliamo che l'evoluzione del sistema virtuale di P_1 sia identica a quella che abbiamo appena visto.



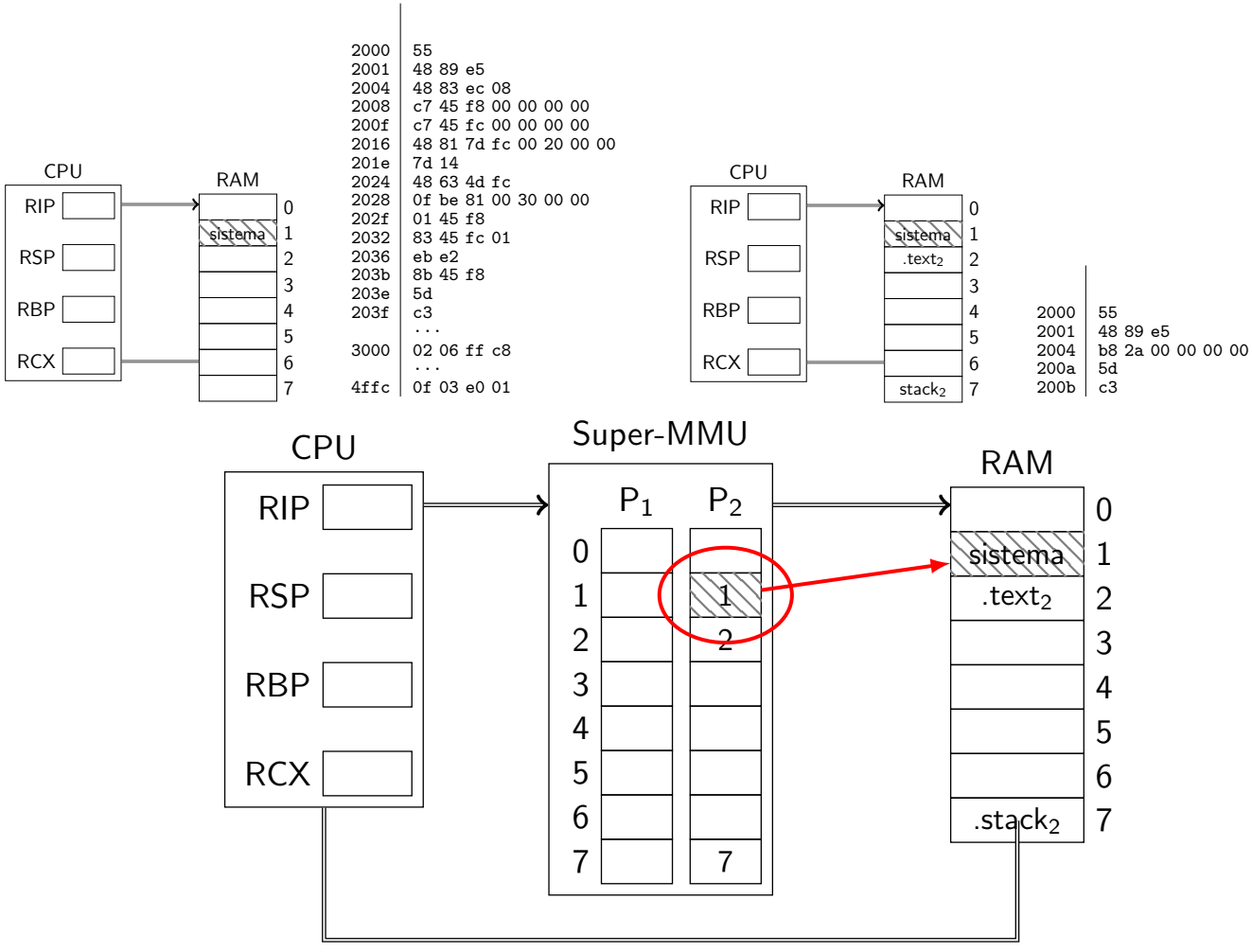
Vogliamo che l'evoluzione del sistema virtuale di P_1 sia identica a quella che abbiamo appena visto.



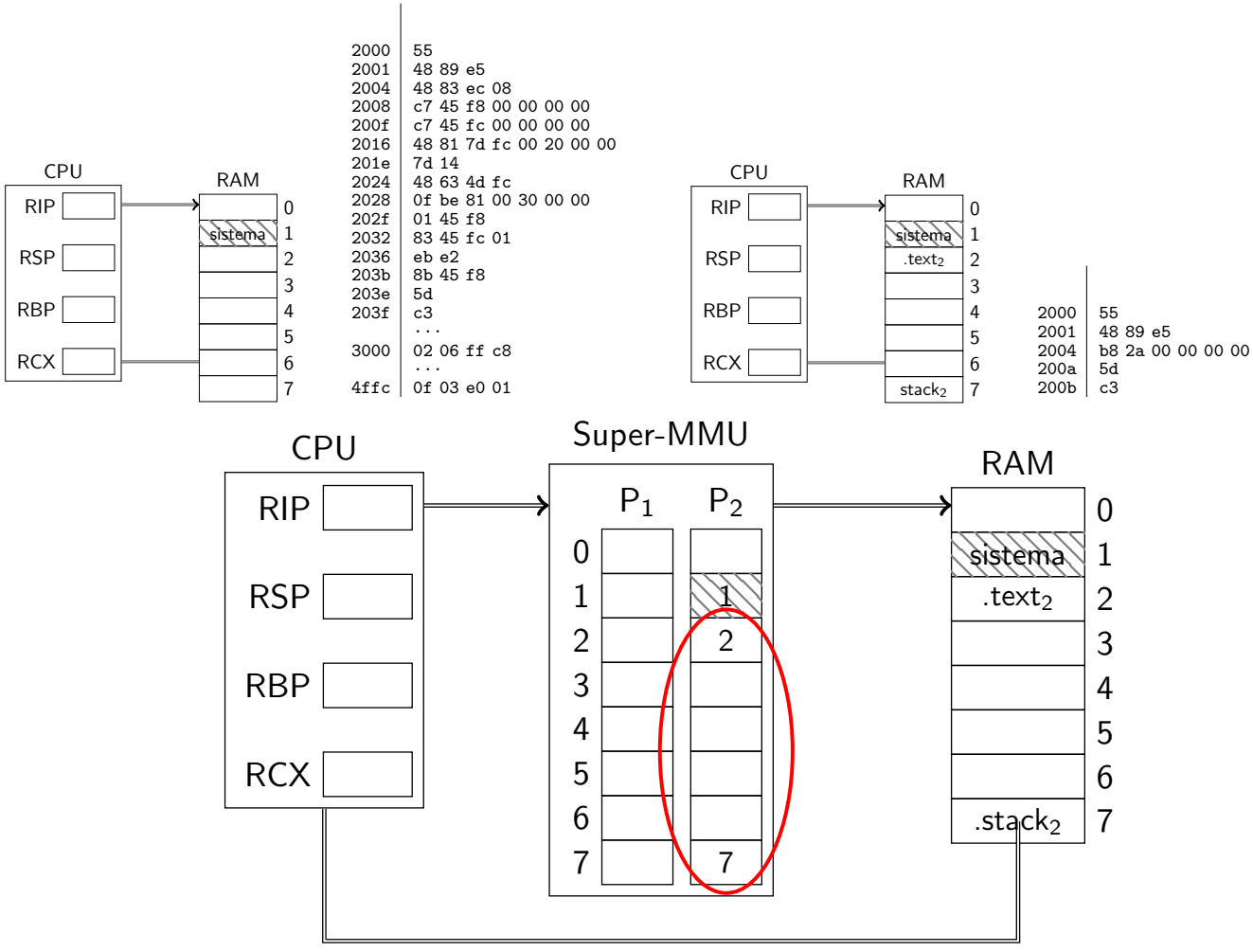
Carichiamo la memoria del processo P_2 e prepariamo la sua tabella di corrispondenza.



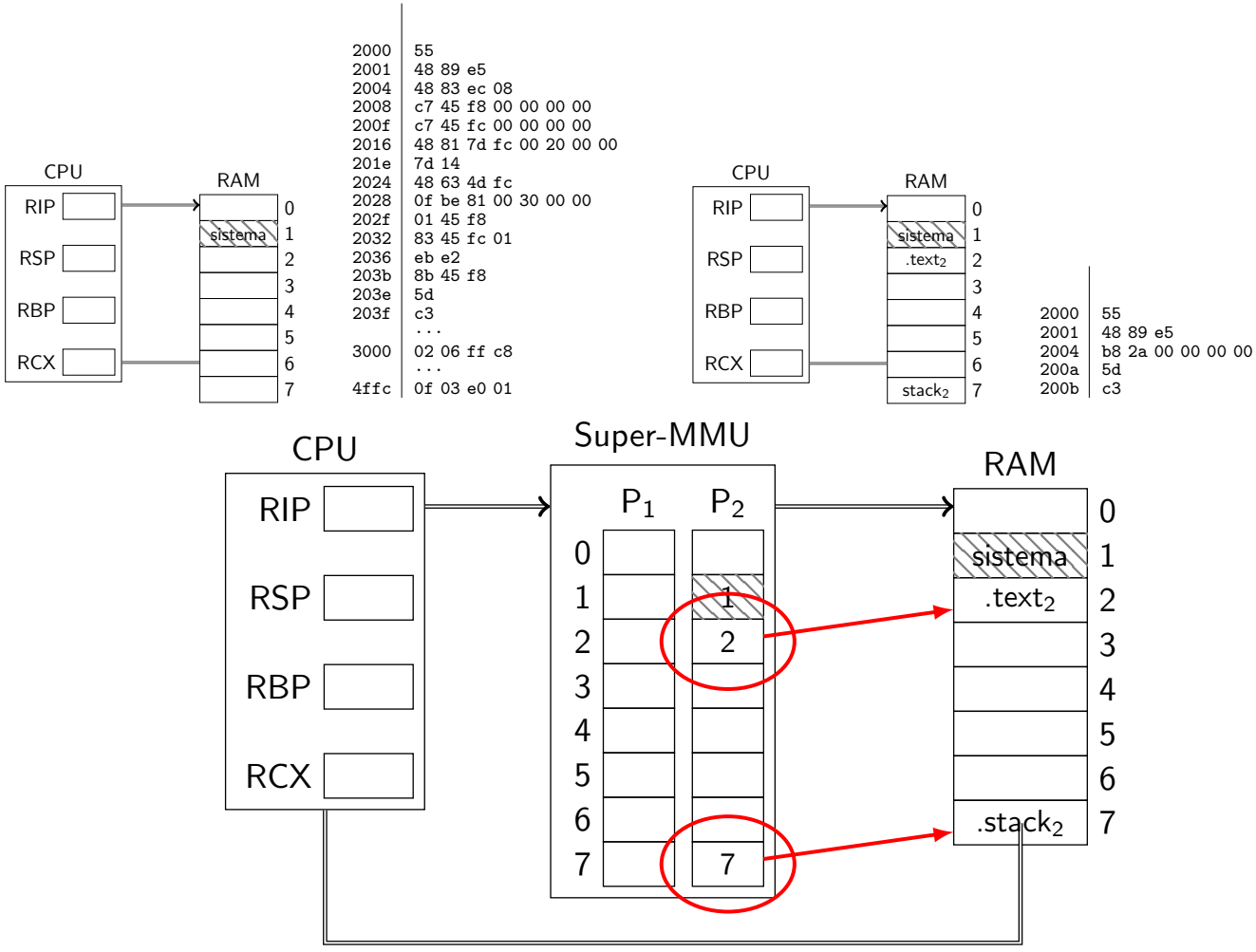
La pagina 0 è lasciata con P=0, per intercettare le derefe-renziazioni di puntatori nulli.



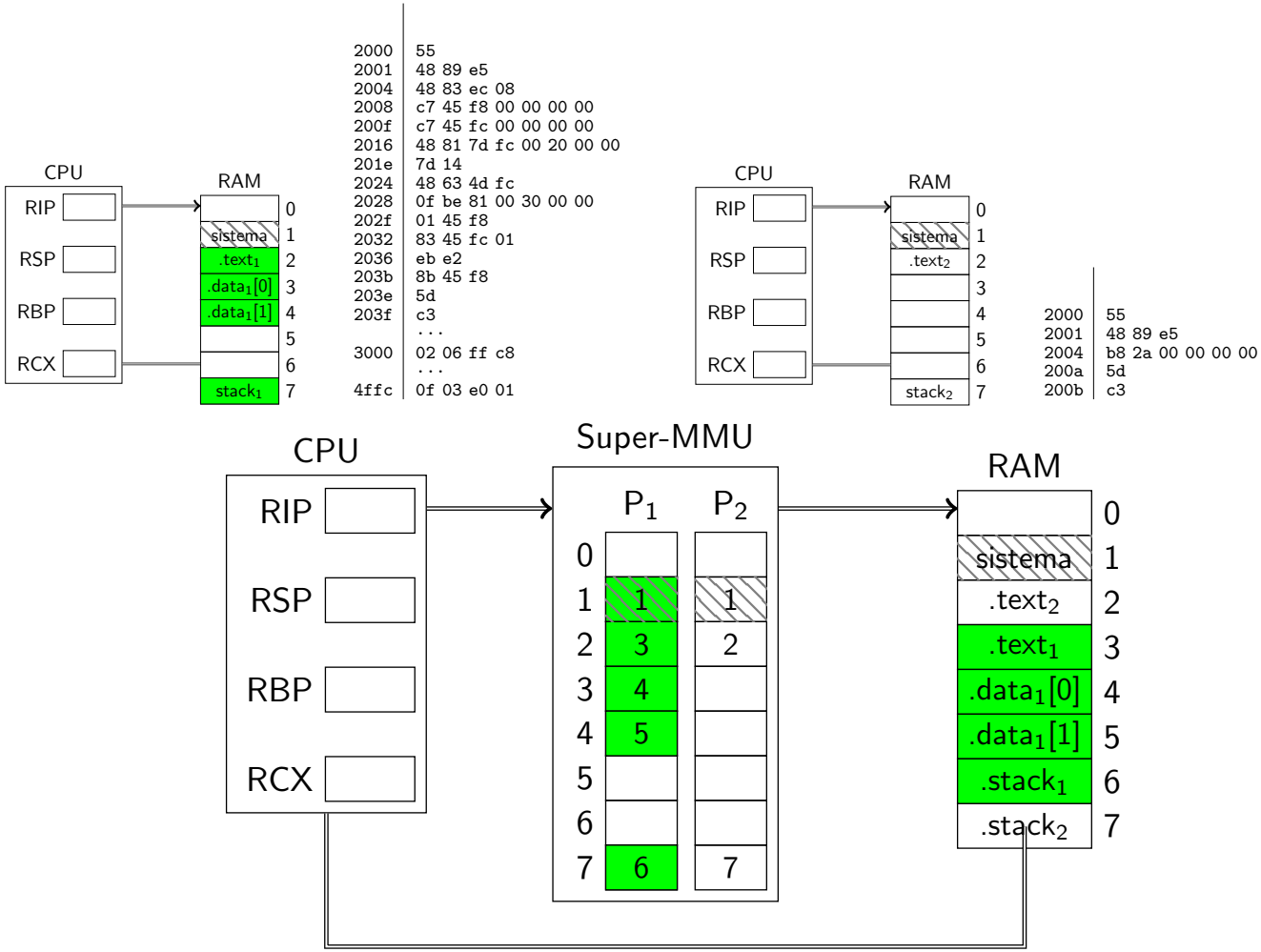
La pagina 1 corrisponde al frame 1, che contiene il sistema.
È marcata come inaccessibile da livello utente.



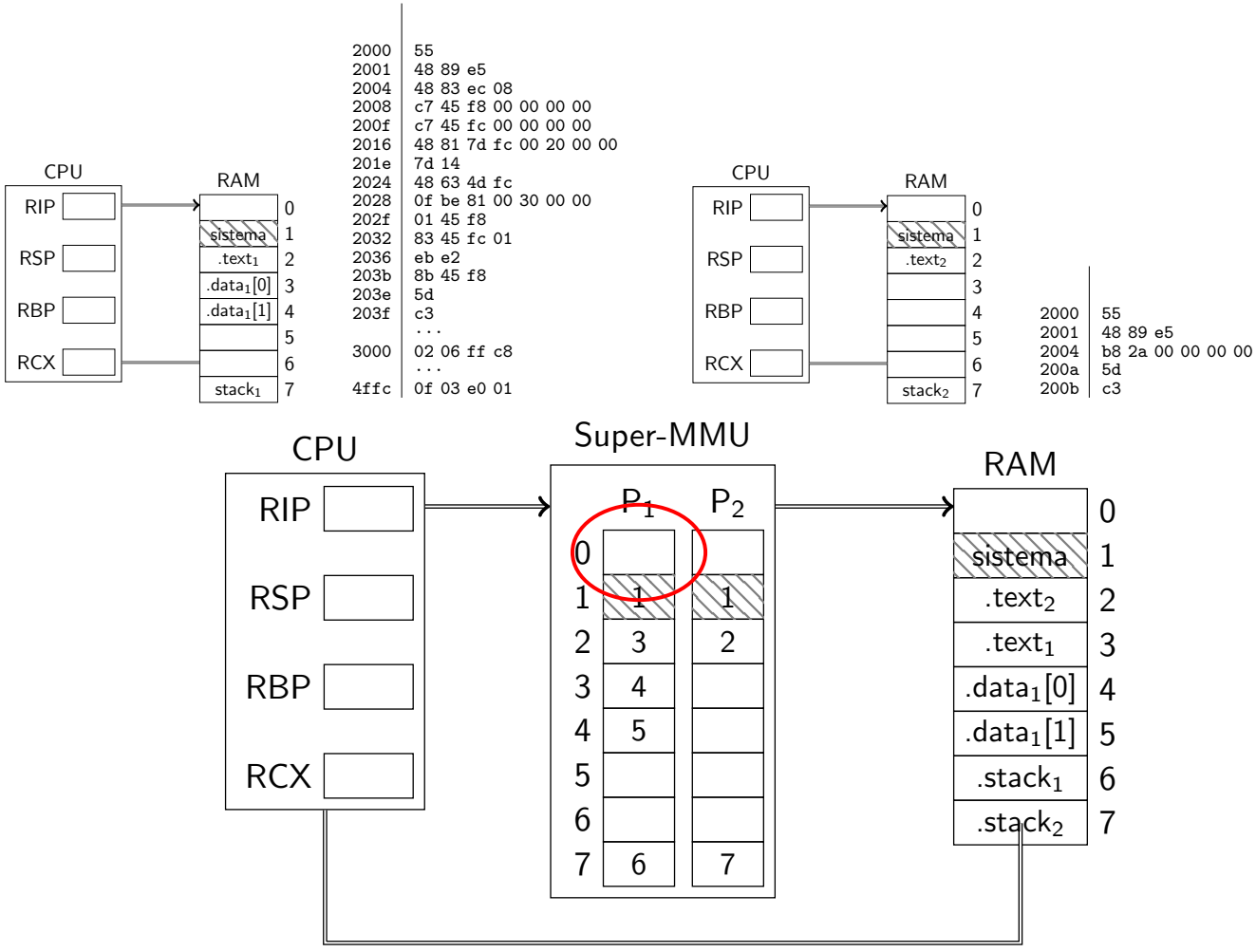
Le pagine da 3 a 6 non sono usate da P_2 (P=0)



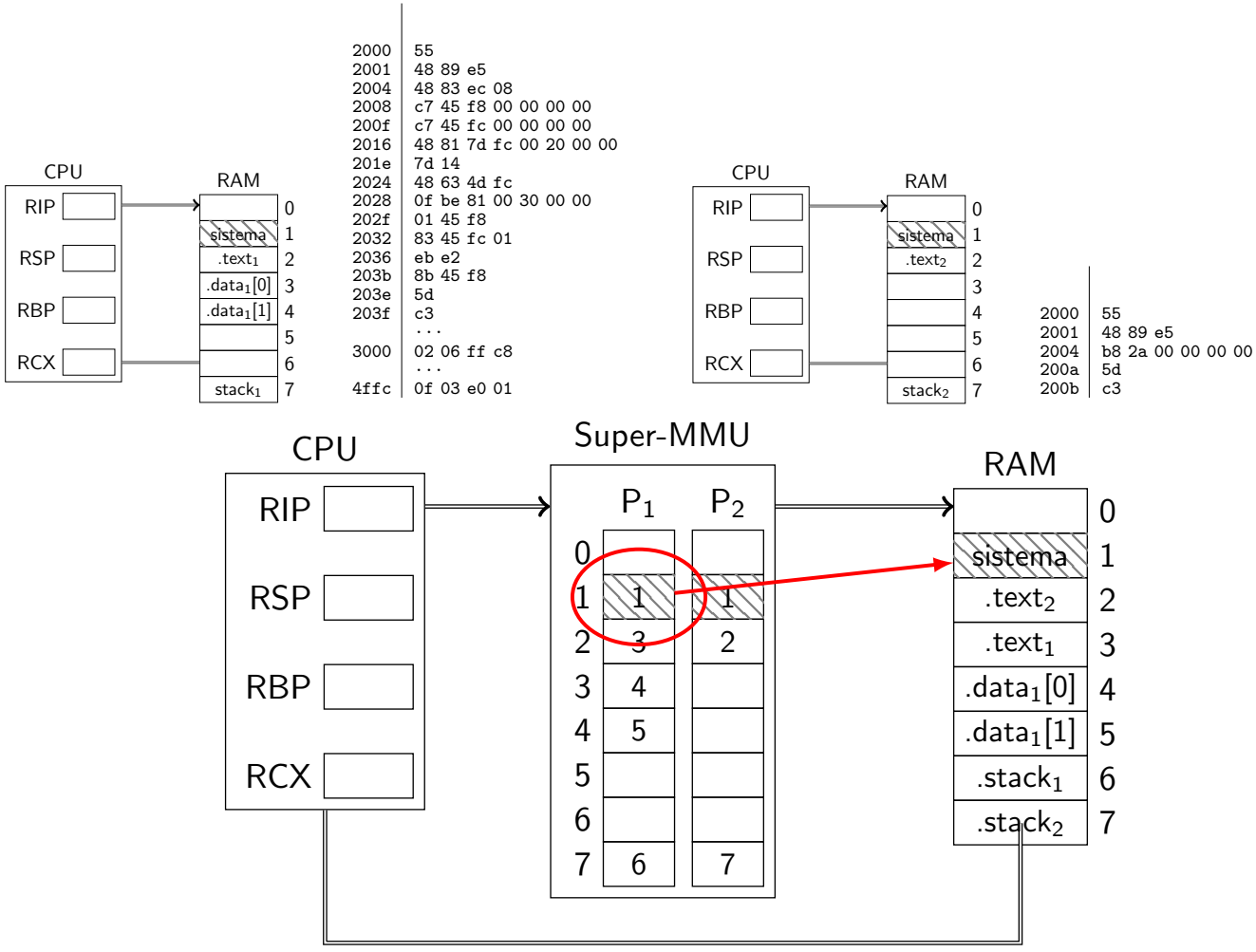
Le altre pagine corrispondono ai frame che le contengono.



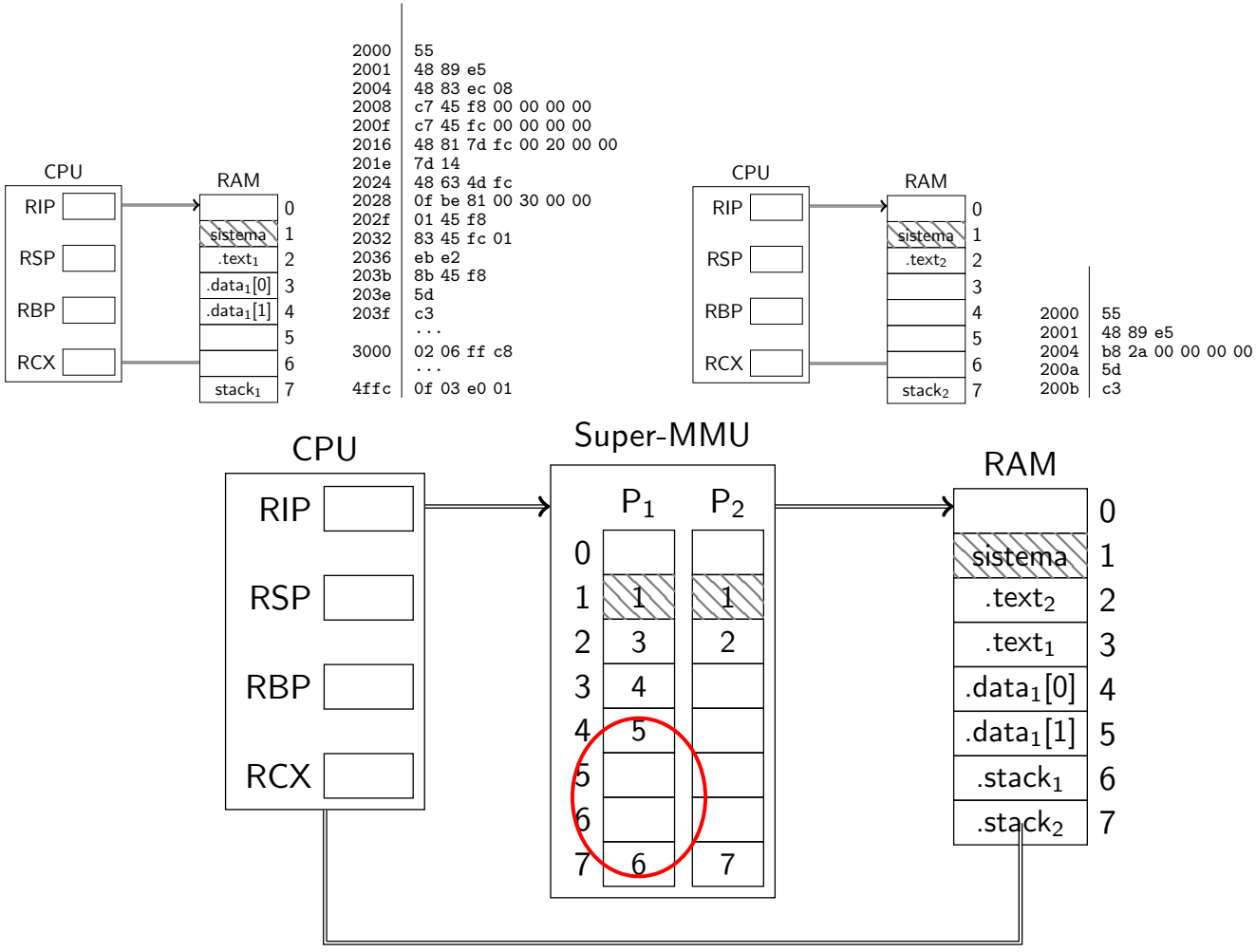
Carichiamo la memoria del processo P_1 nello spazio che rimane e prepariamo anche la sua tabella.



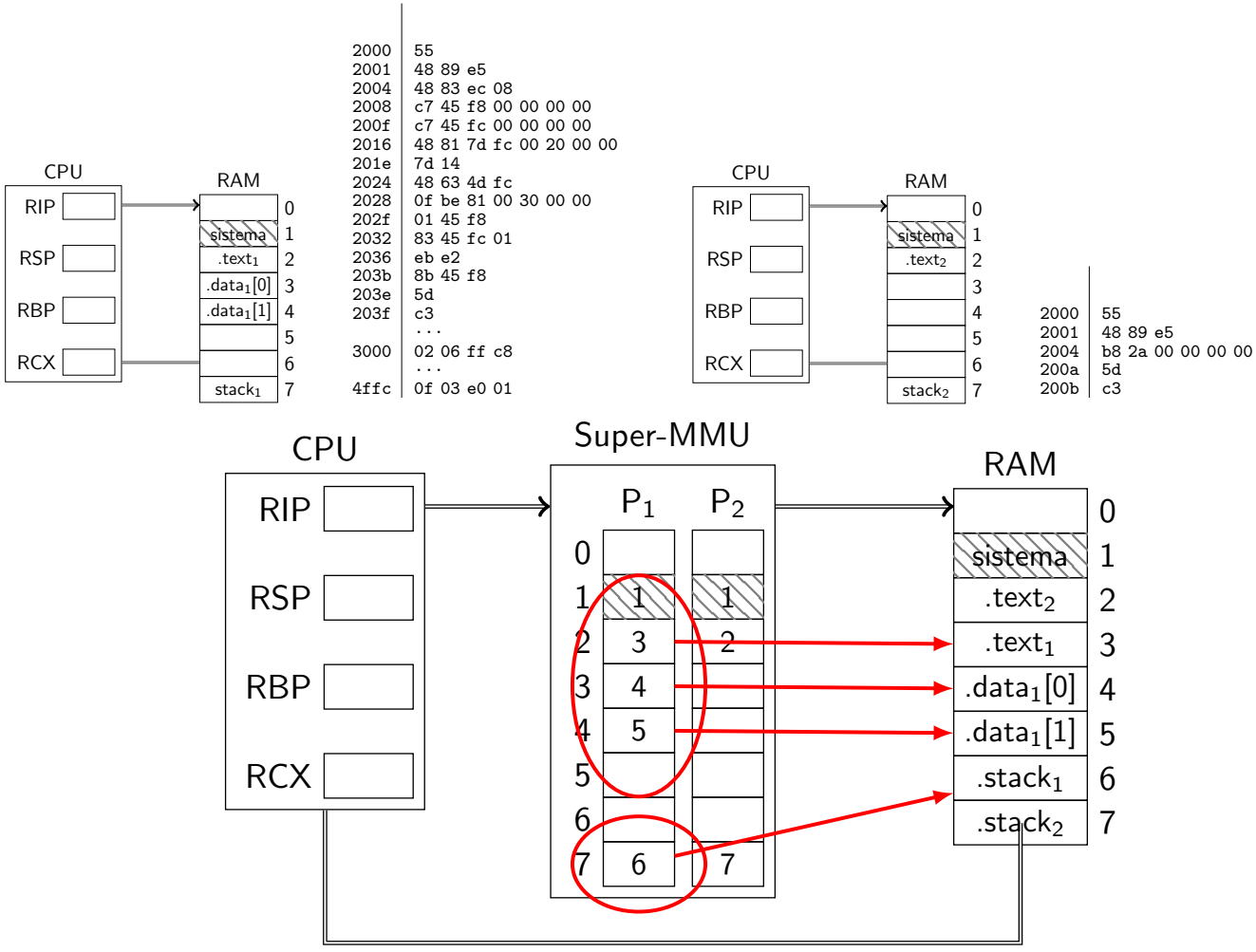
Anche per P_1 La pagina 0 è lasciata con P=0 . . .



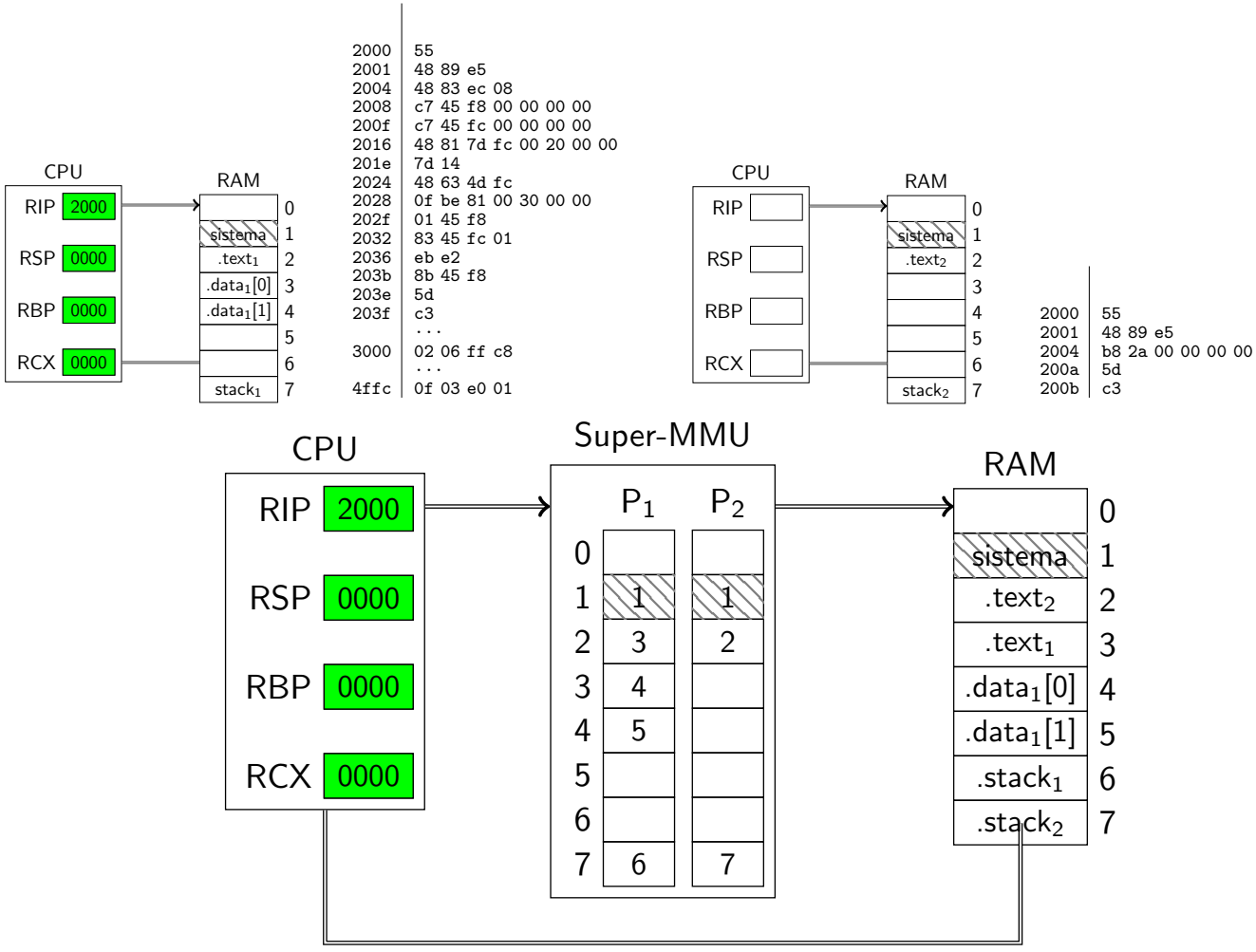
. . . e la pagina 1 corrisponde al frame 1, ma è inaccessibile da livello utente.



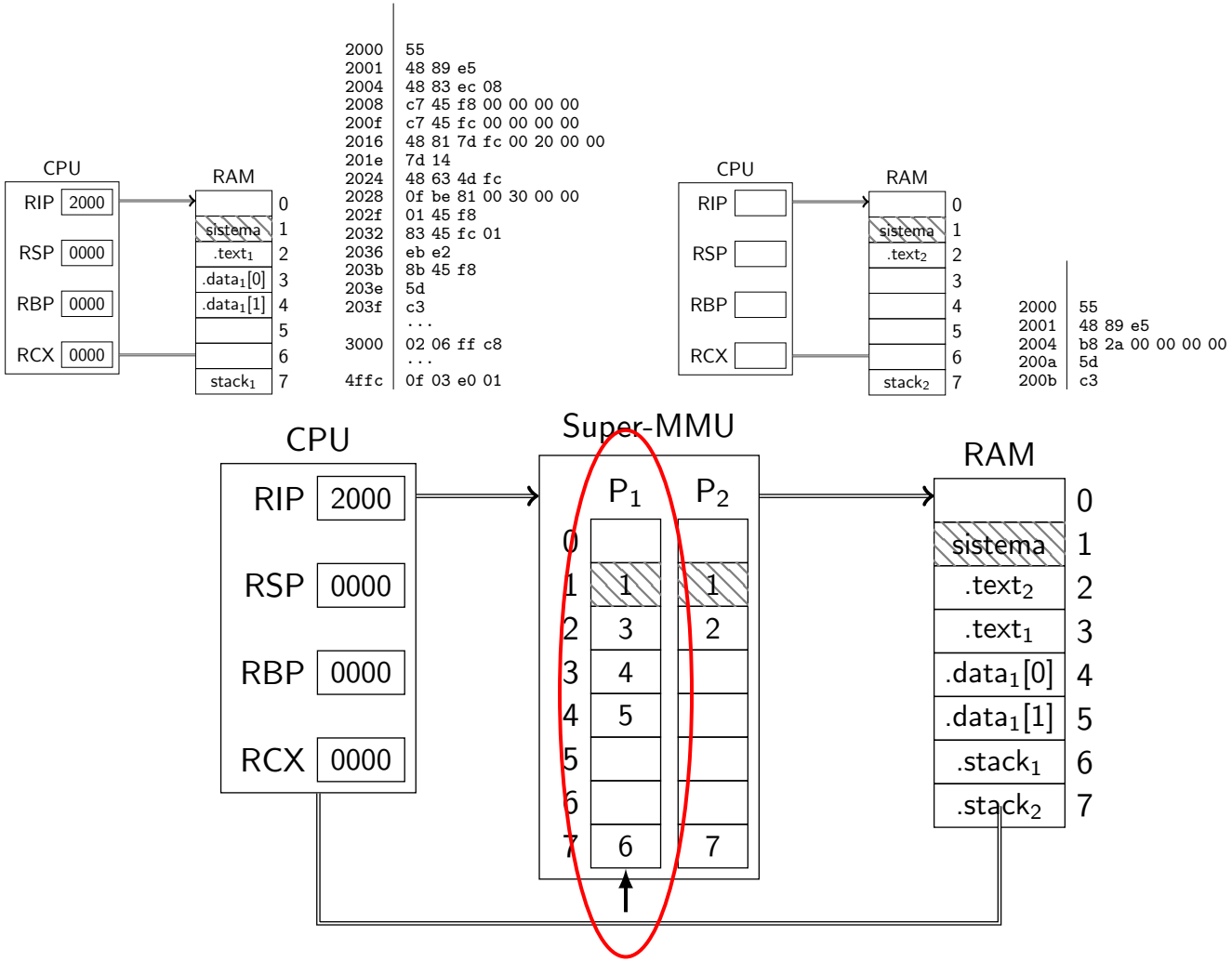
Le pagine 5 e 6 non sono usate (P=0) ...



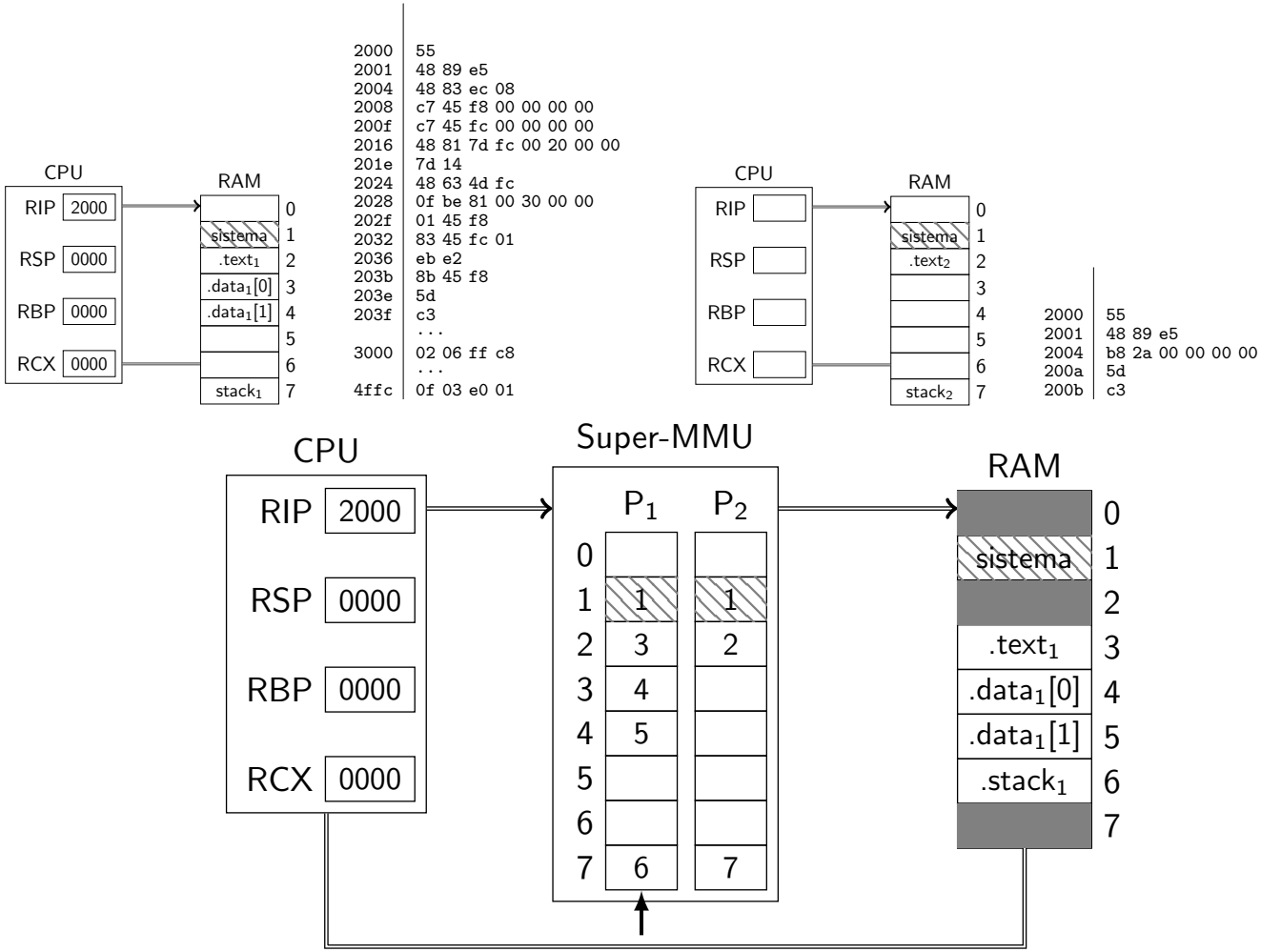
. . . e le altre corrispondono ai frame che le contengono.



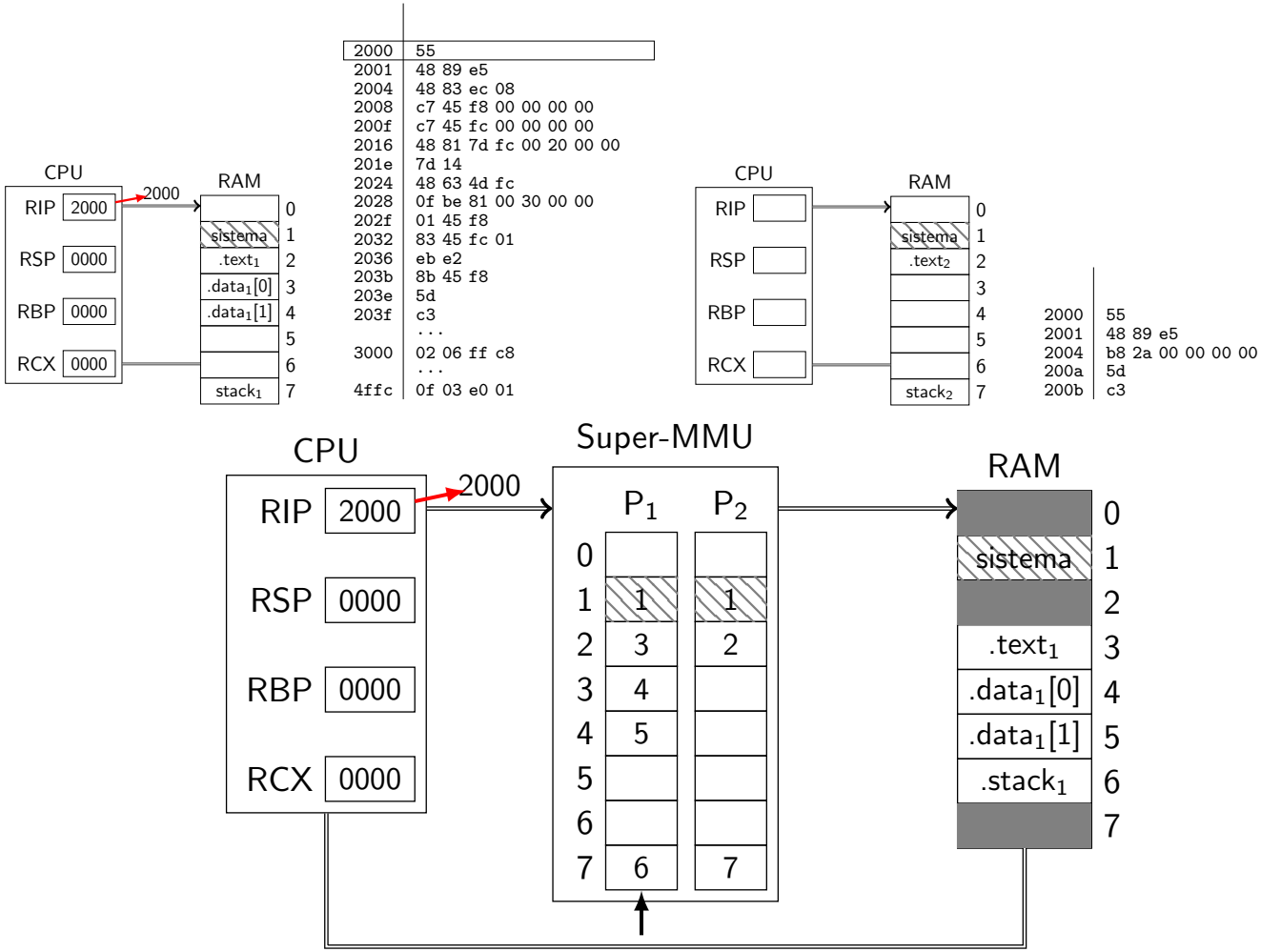
Supponiamo che il sistema metta in esecuzione P_1, caricando lo stato dei registri...



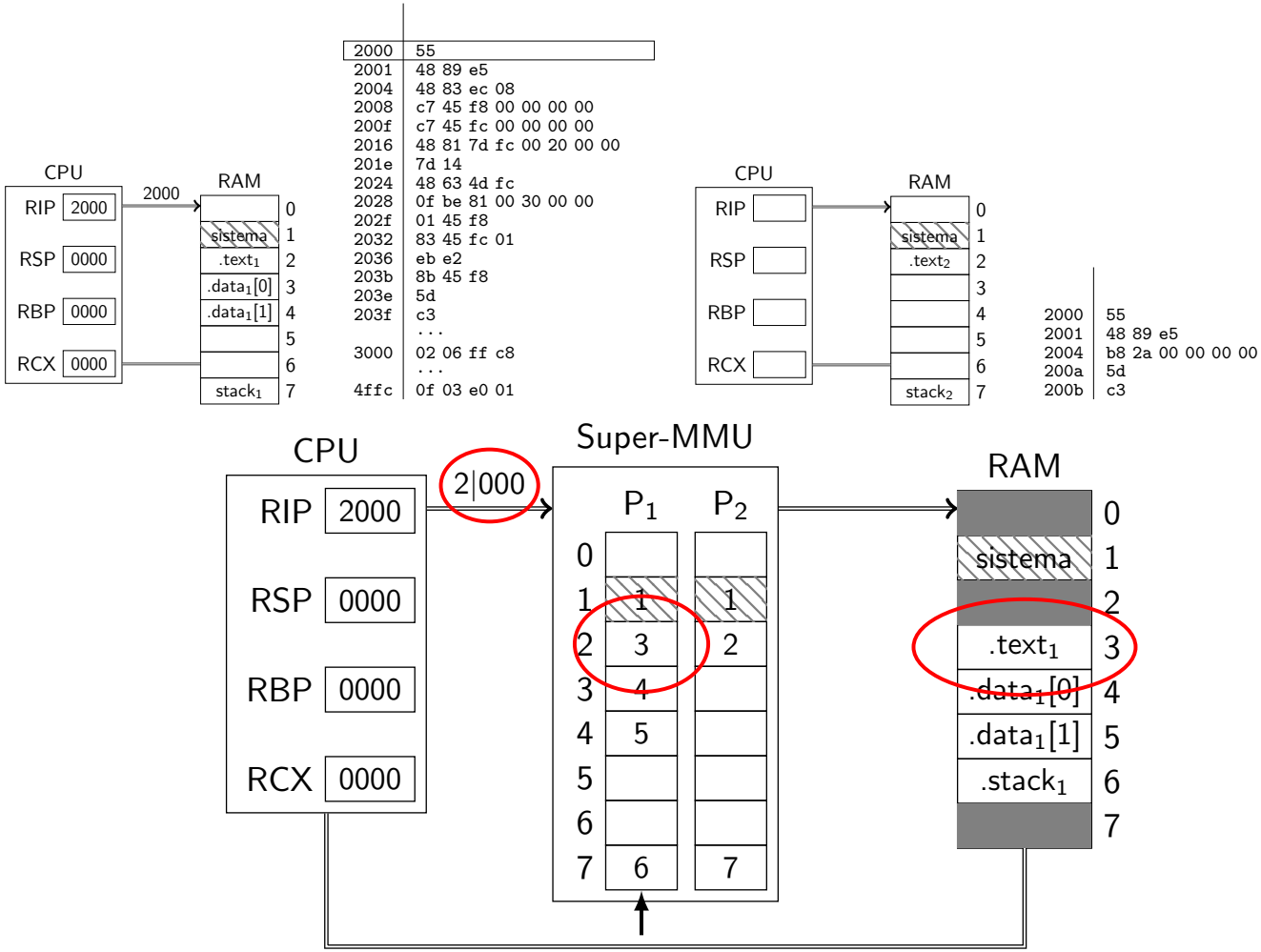
. . . e rendendo attiva la tabella P_1.



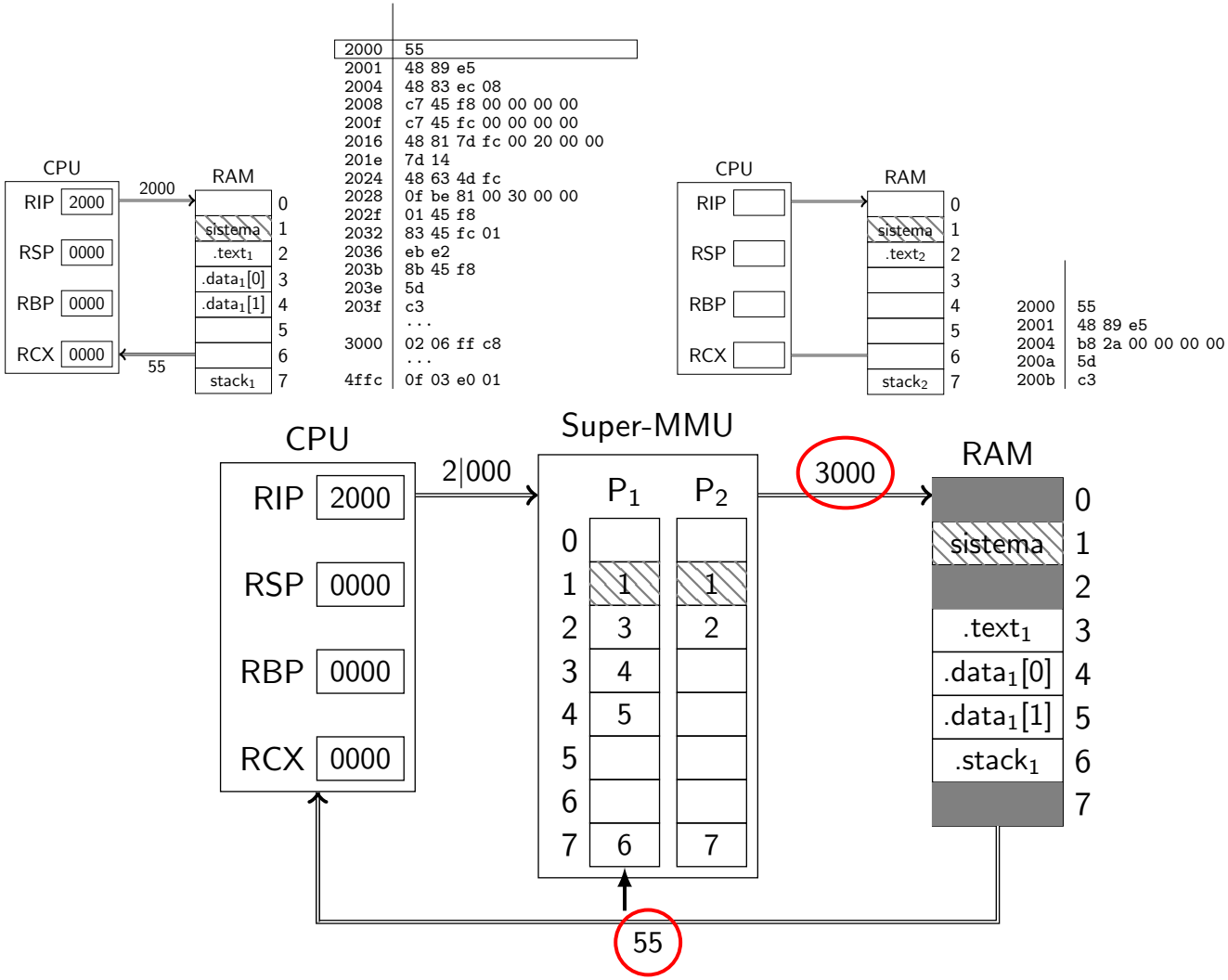
Tutte le pagine che non sono nel codominio di P_1 diventano inaccessibili.



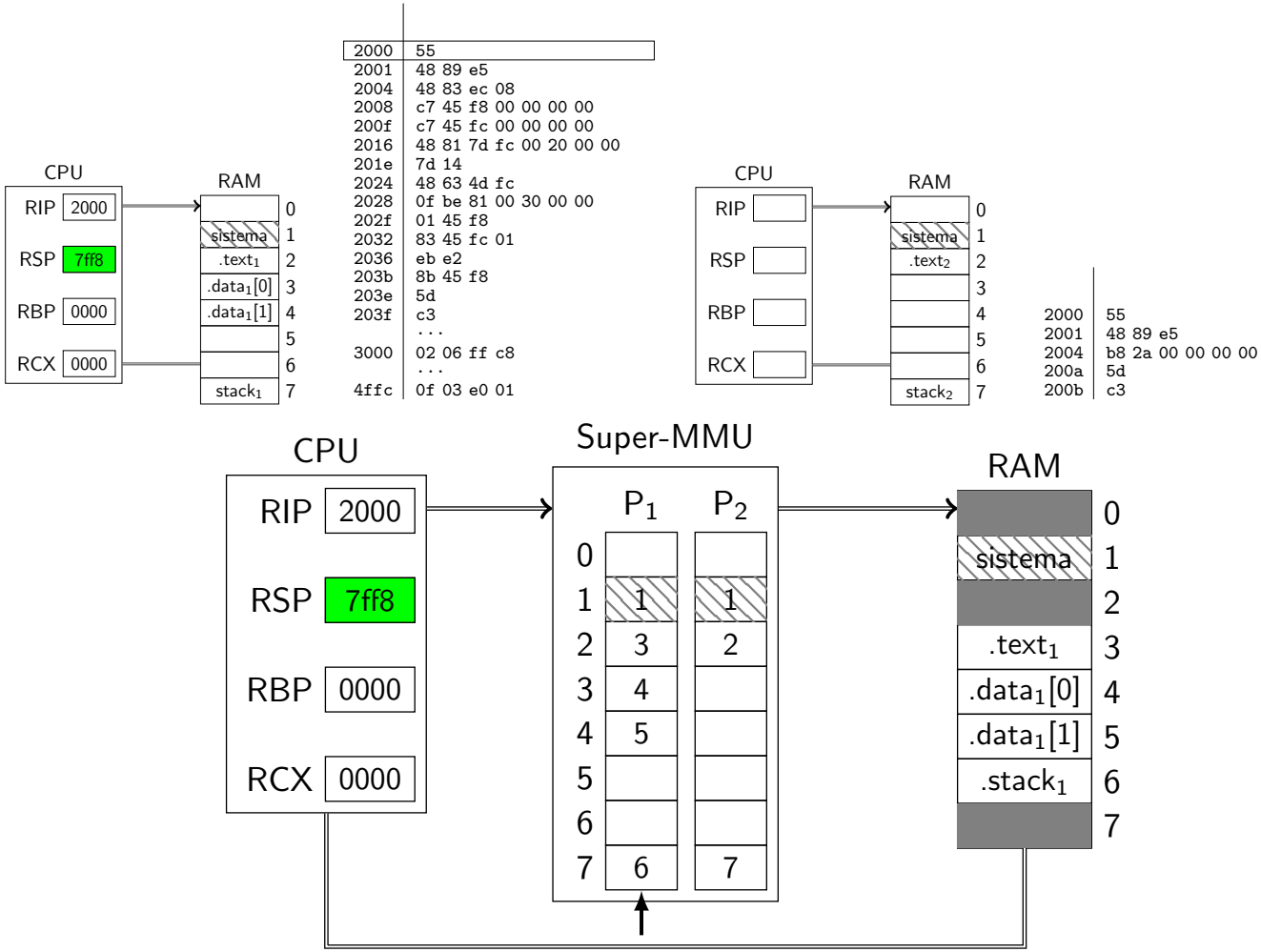
P_1 comincia la sua esecuzione. La CPU fisica esegue una lettura all'indirizzo 2000, esattamente come quella virtuale.



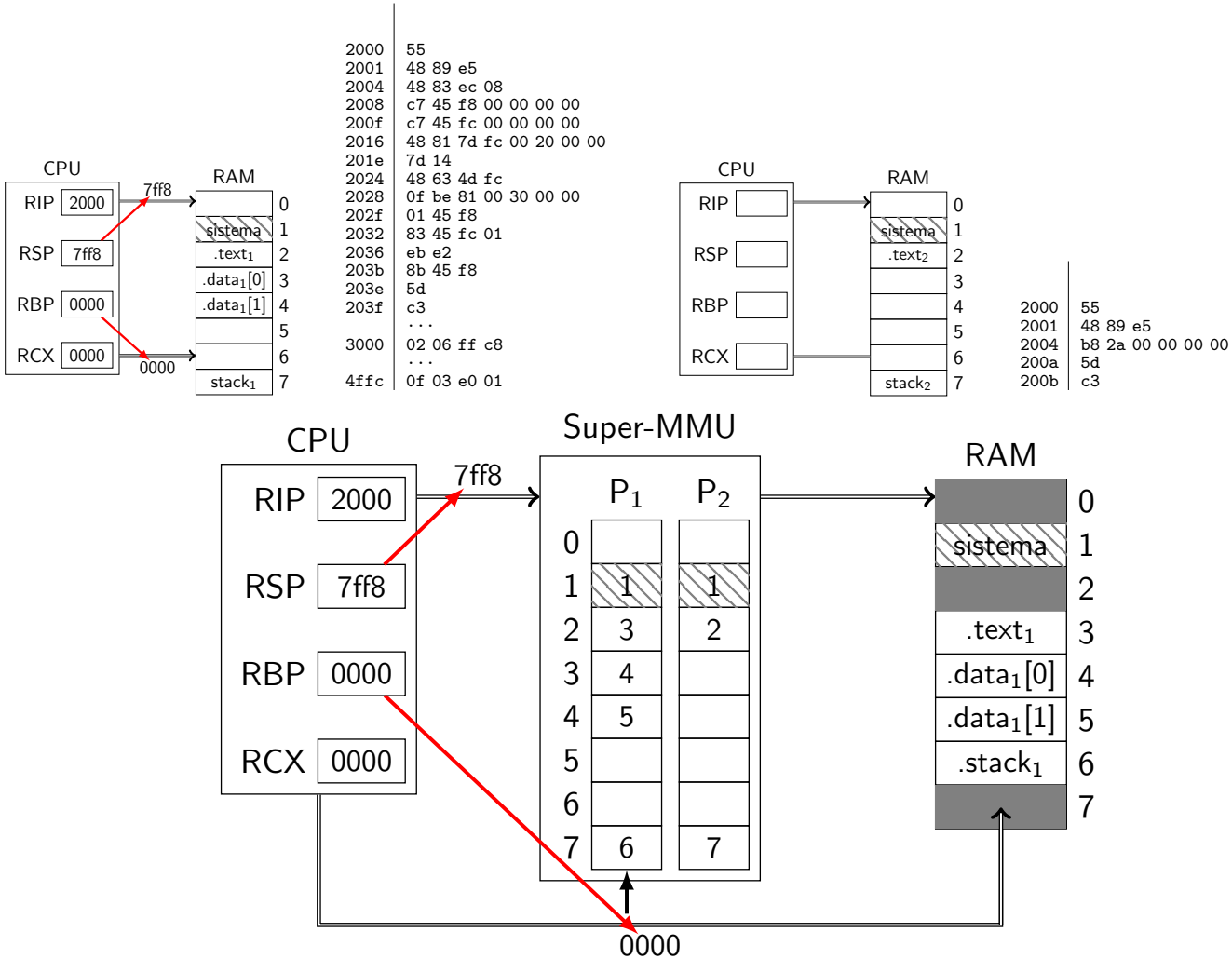
La MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina (2) e offset (000). Consulta quindi l'entrata numero 2 della tabella di corrispondenza e trova il corrispondente numero di frame (3)



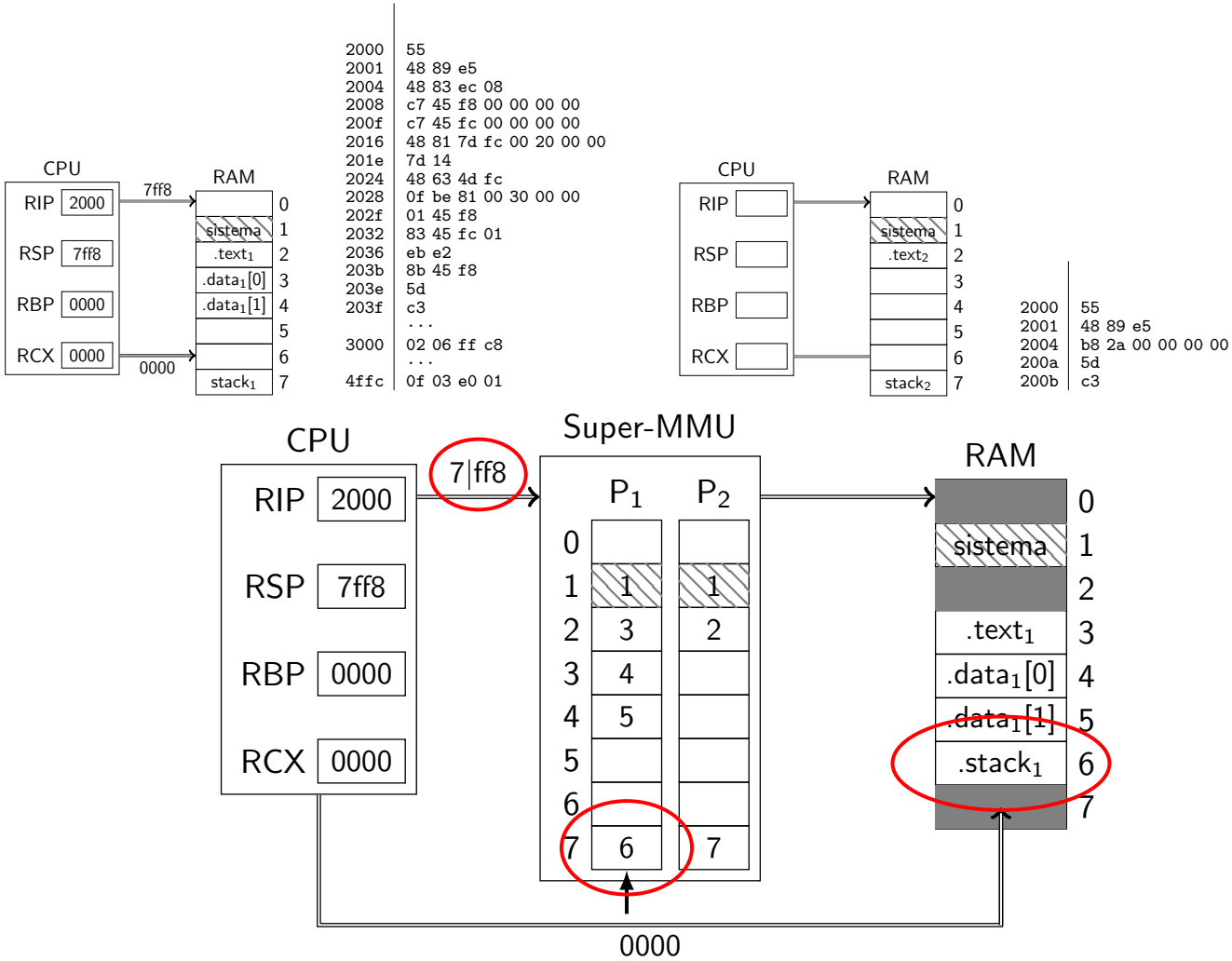
L'accesso viene completato e la CPU fisica riceve `pushq %rbp`, esattamente come la virtuale, e inizia ad eseguirla.



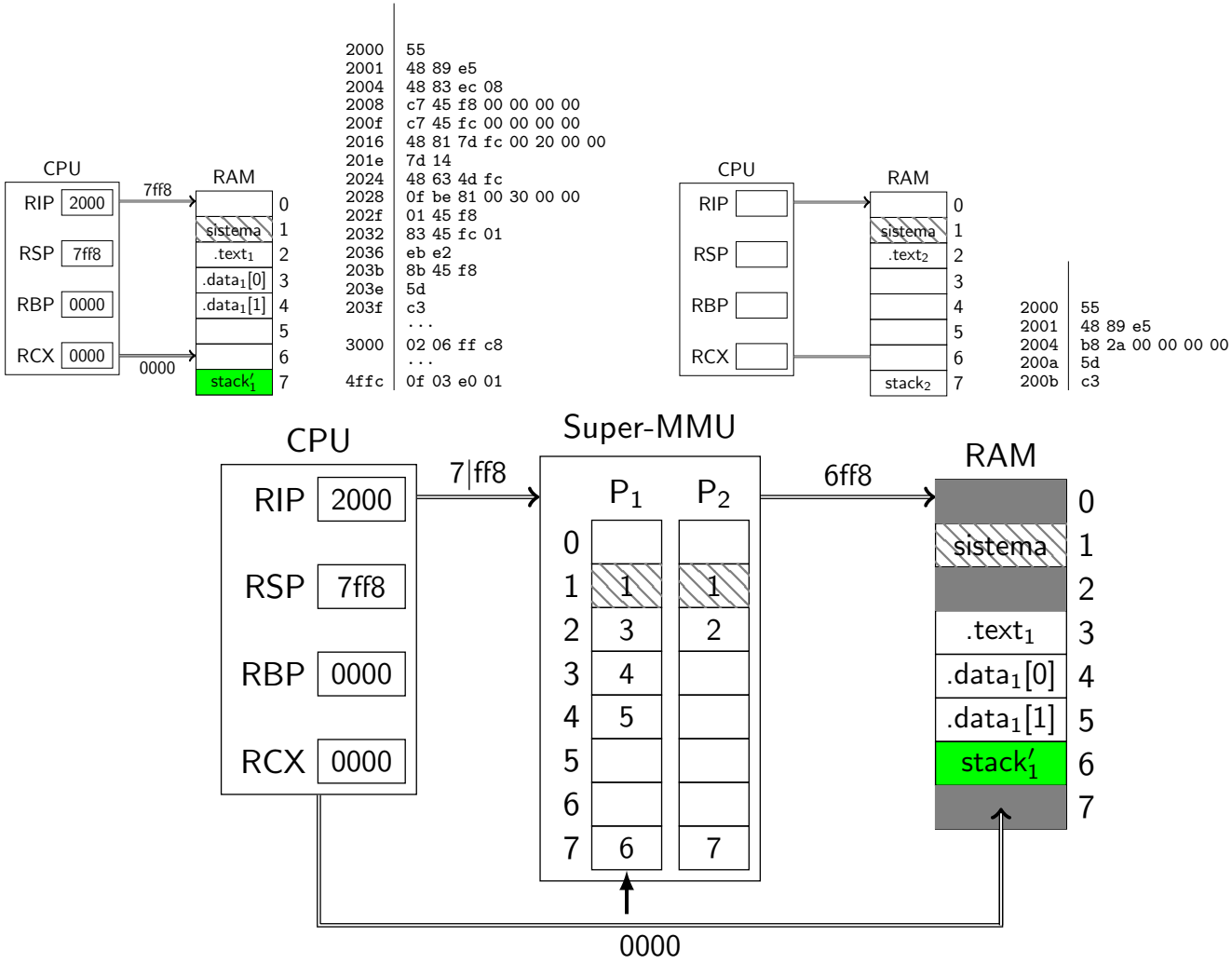
Le due CPU, fisica e virtuale, fanno esattamente le stesse cose. Come primo passo decrementano RSP di 8.



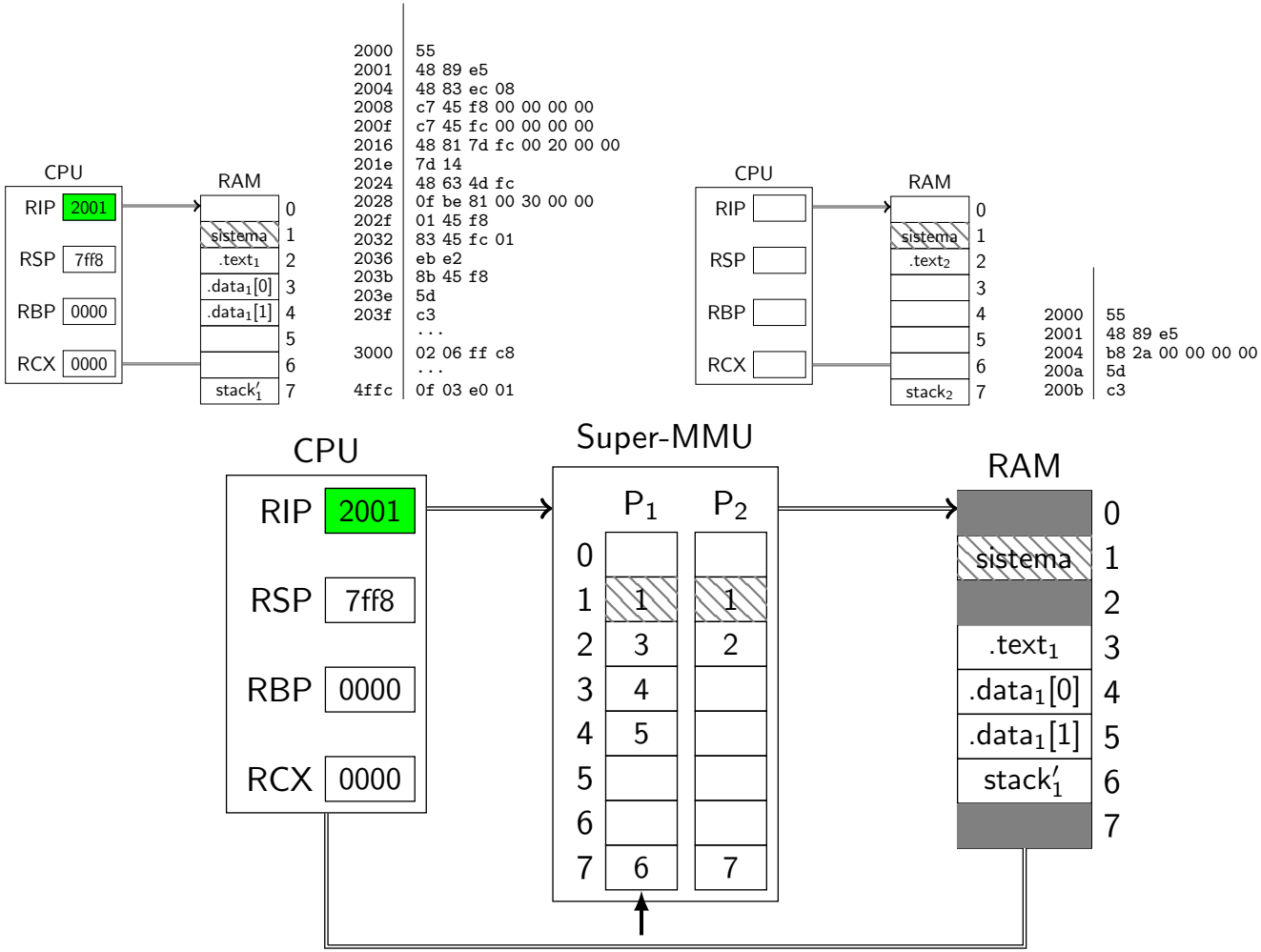
Come secondo passo voglino scrivere il contenuto di RBP all'indirizzo contenuto in RSP. Le due CPU iniziano quindi una operazione di scrittura all'indirizzo 7ff8.



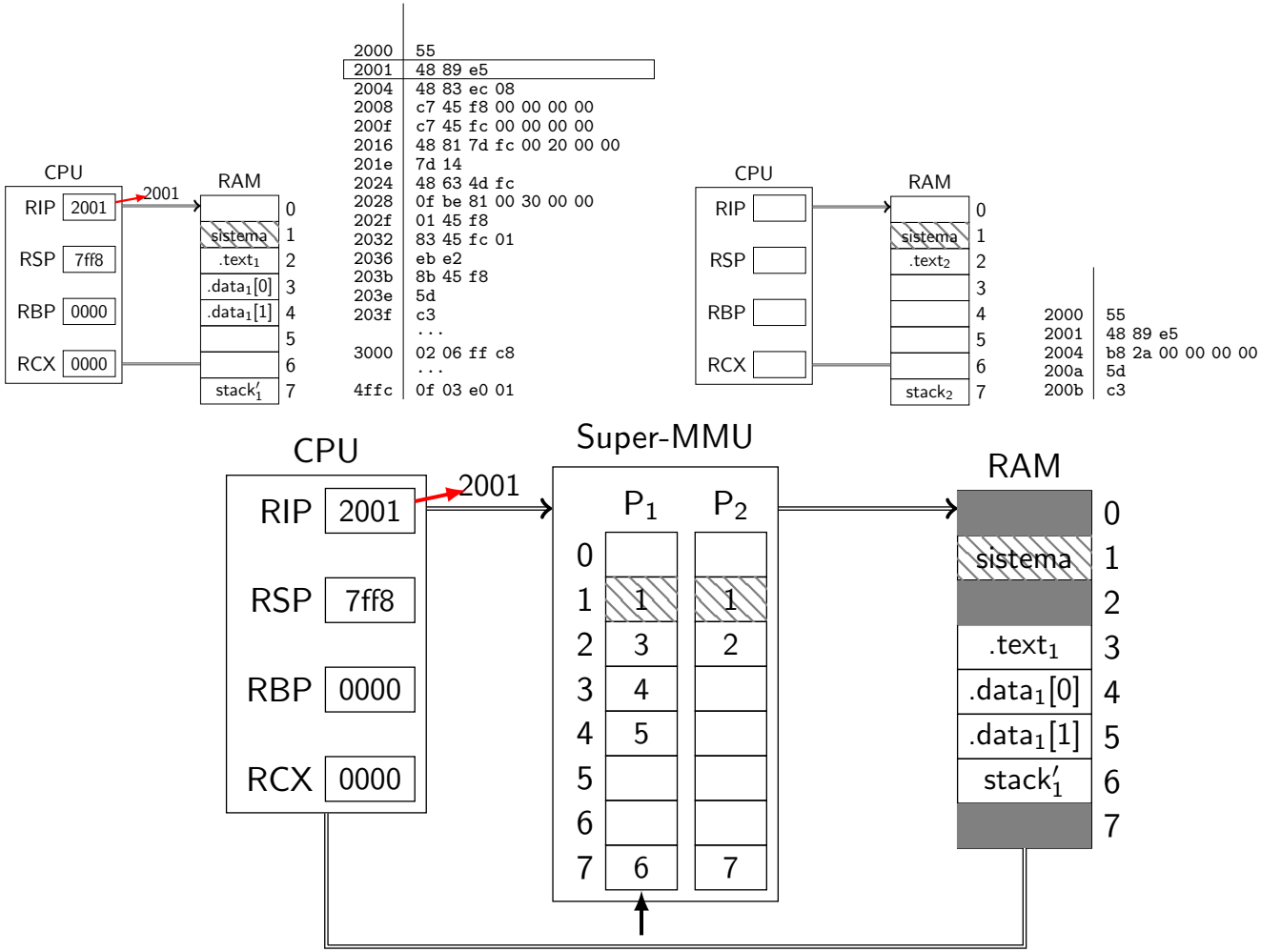
La MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina (7) e offset (ff8). Consulta quindi l'entrata numero 7 della tabella di corrispondenza e trova il numero di frame (6)



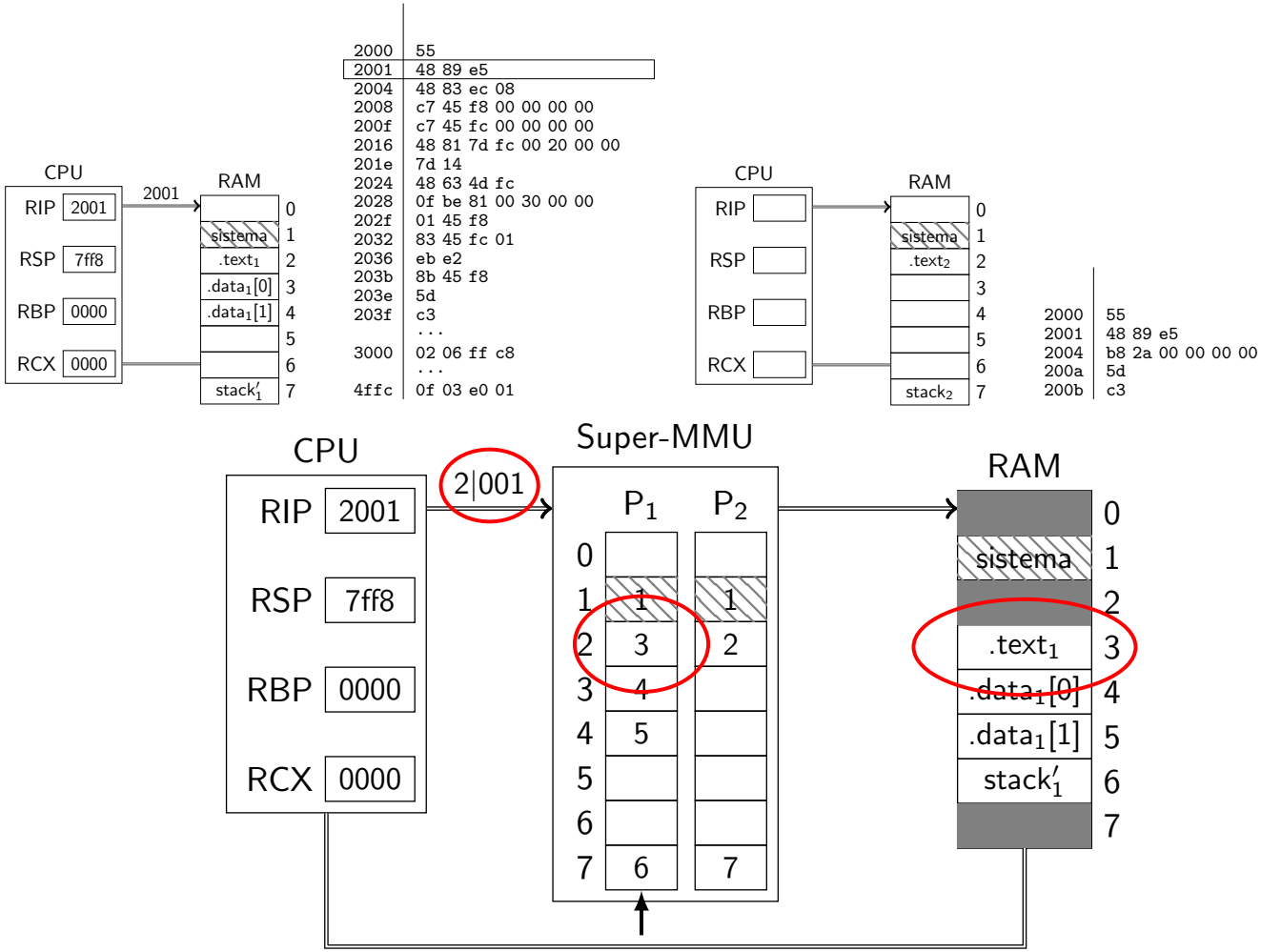
L'operazione di scrittura viene completata, dopo aver trasformato l'indirizzo. Il contenuto del frame 6 del sistema fisico continua ad essere uguale a quello della pagina 7 del sistema virtuale.



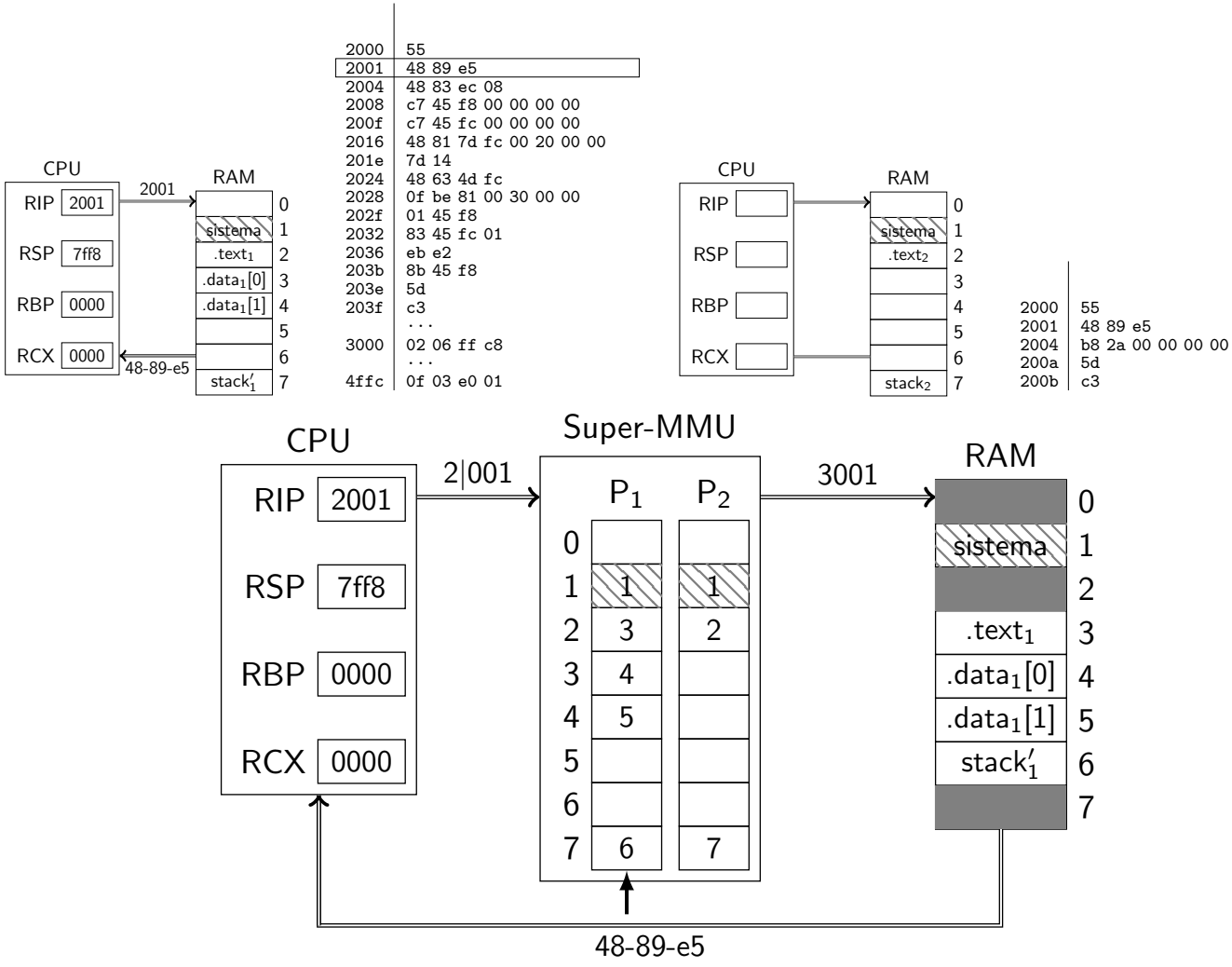
Le due CPU hannoo completato l'esecuzione dell'istruzione `pushq %rbp` e possono passare alla successiva. Incrementano RIP di 1 (che è la dimensione dell'istruzione appena terminata).



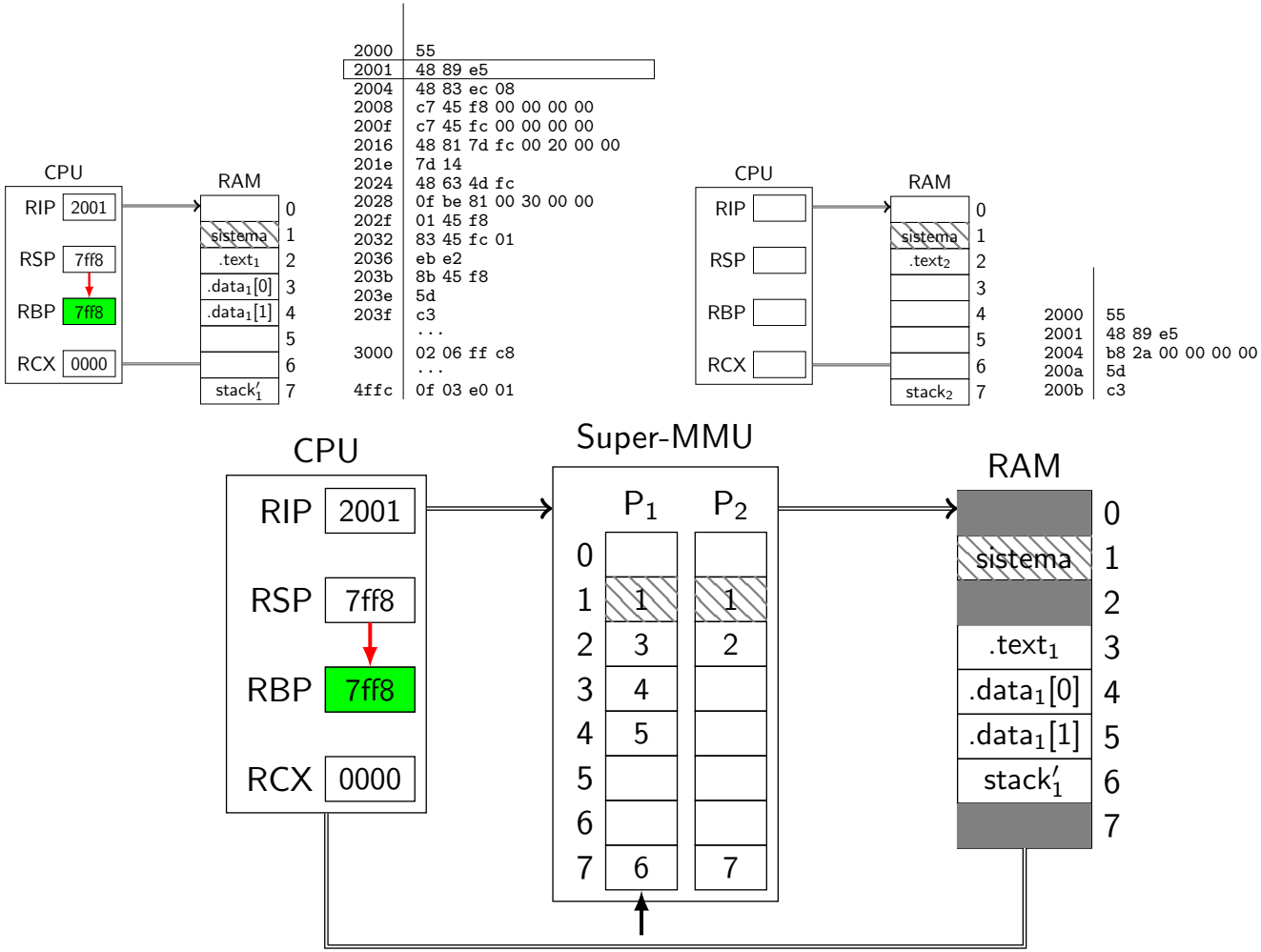
Nuovo ciclo: le CPU tentano di prelevare l'istruzione all'indirizzo contenuto in RIP. Inizia una operazione di lettura in memoria all'indirizzo 2001.



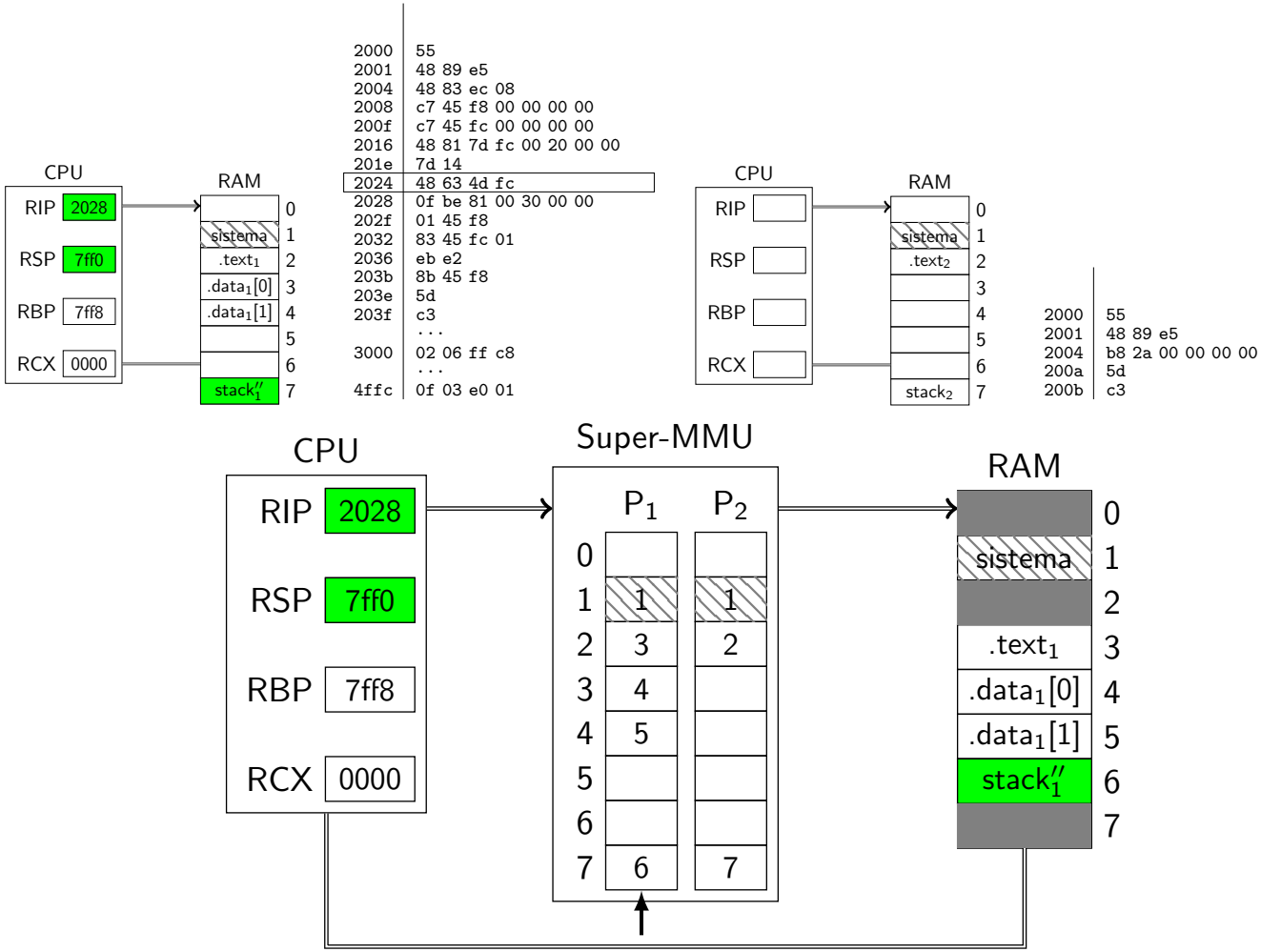
Come al solito, la MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina virtuale (2) e offset (001).



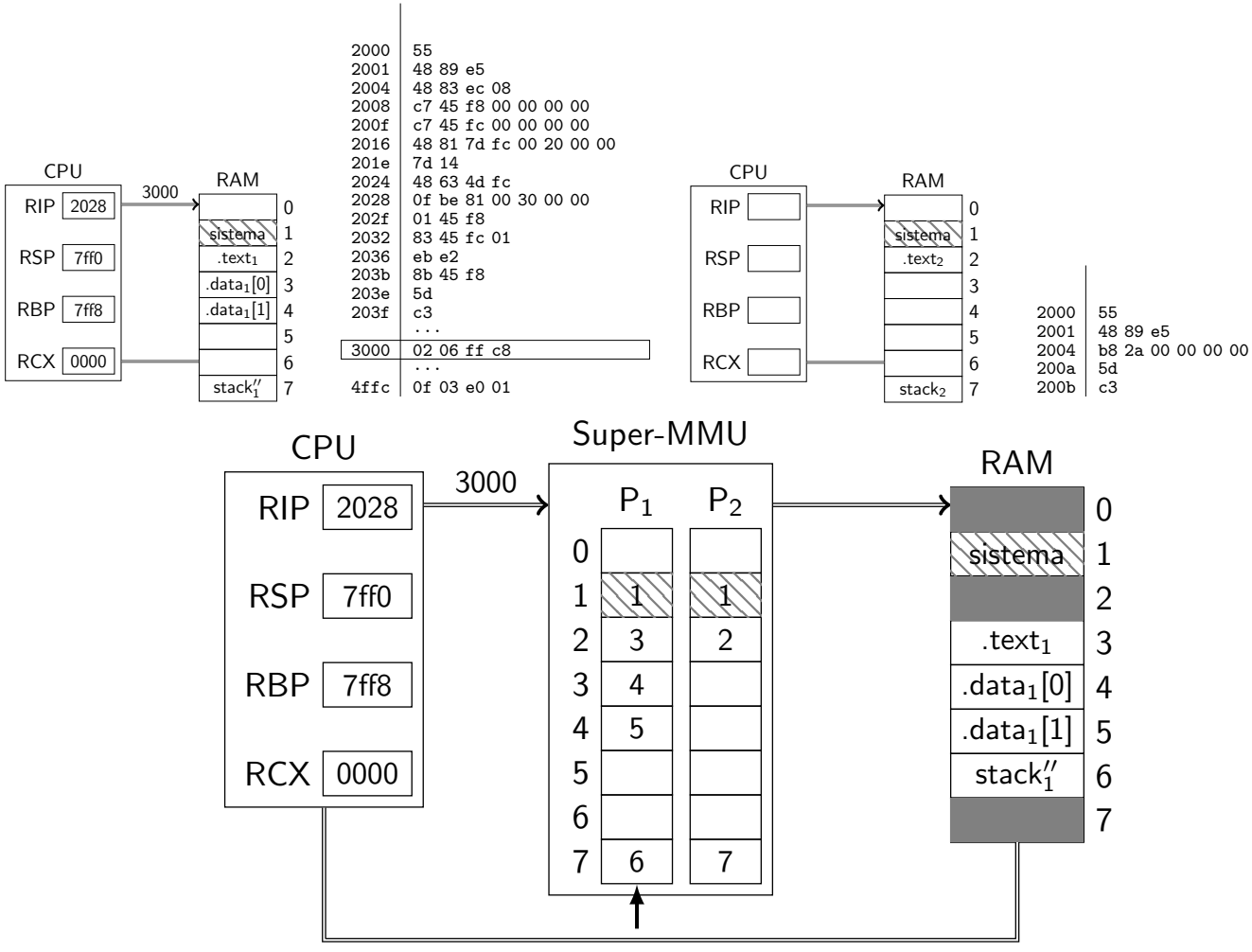
L'accesso viene completato, ovviamente dopo aver trasformato l'indirizzo. Entrambe le CPU ricevono l'istruzione `movq %rsp, %rbp`.



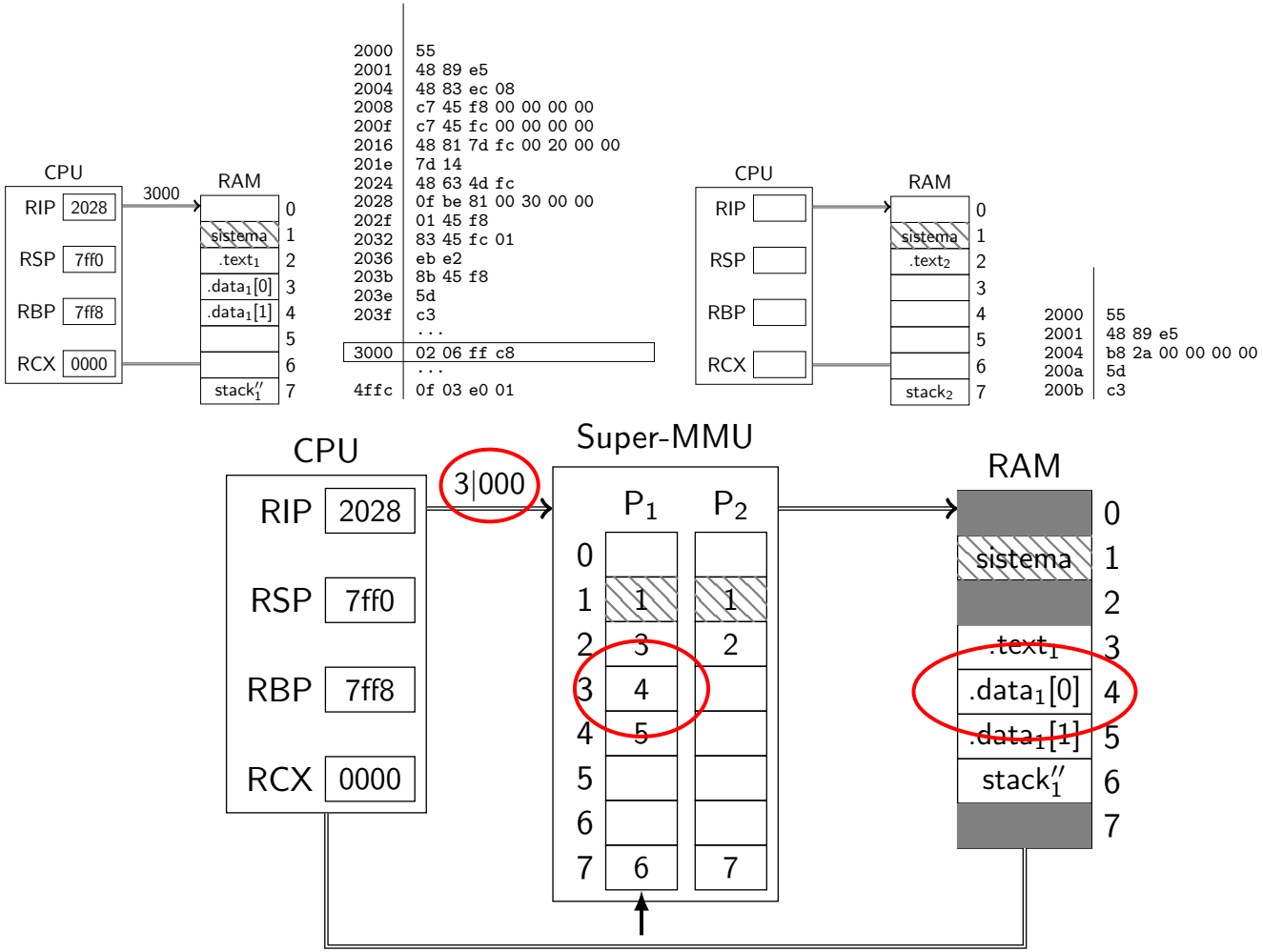
Per eseguirla copiano il contenuto di RSP in RBP. Dovremmo a questo punto convincerci che gli stati fisico e virtuale procedono di pari passo.



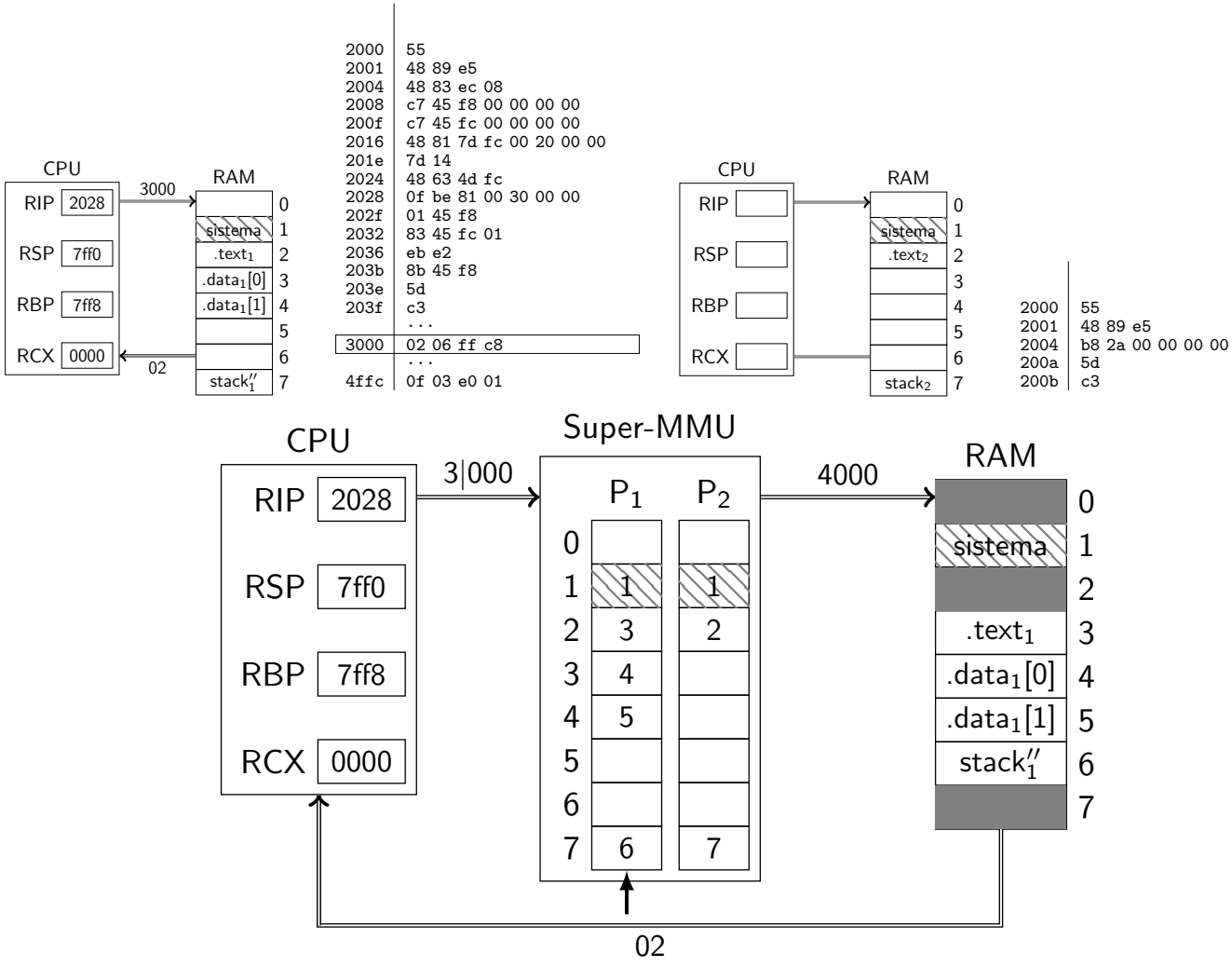
Dopo alcune istruzioni, l'esecuzione del programma arriva all'istruzione `movsbl buf(%rcx), %rax` che si trova all'indirizzo (virtuale) 2028.



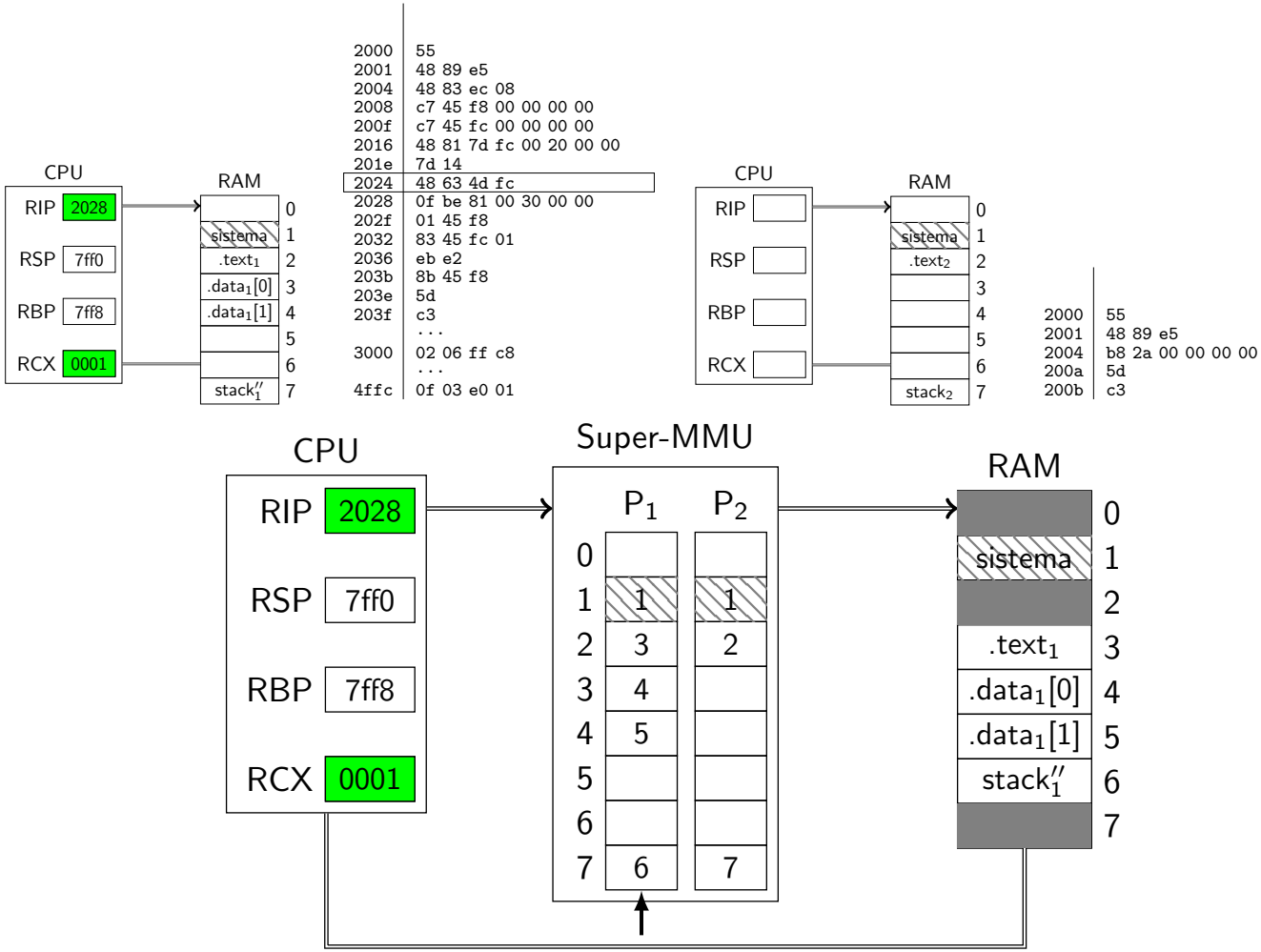
Entrambe le CPU sommano il contenuto di RCX e la costante 3000 (buf nel sorgente), ottenendo l'indirizzo 3000. Iniziano dunque una operazione di lettura a questo indirizzo.



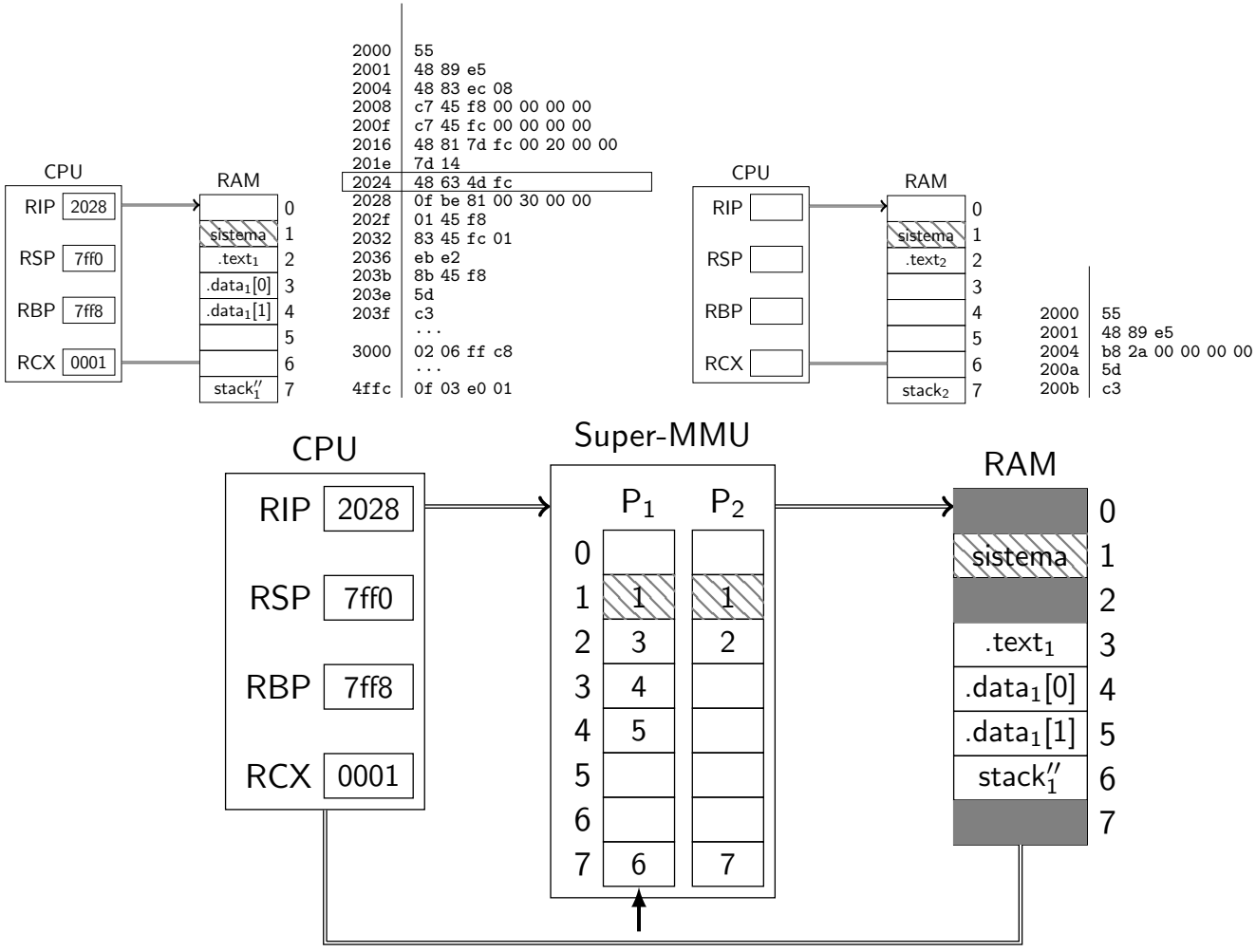
Ancora una volta la MMU scompone l'indirizzo in numero di pagina (3) e offset (000). L'entrata numero 3 della tabella di corrispondenza dice che la pagina 3 si trova nel frame 4.



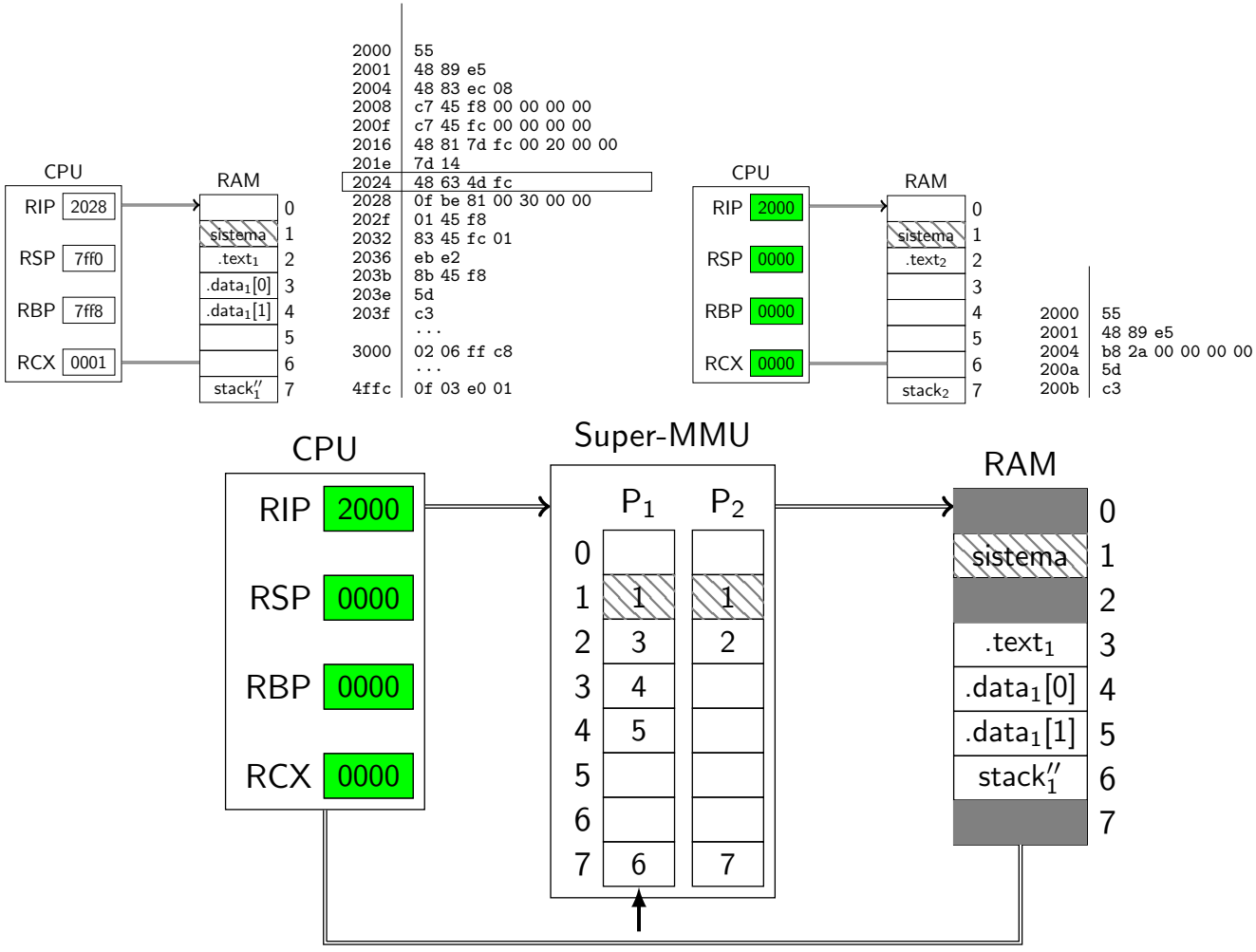
Viene completato l'accesso, sempre dopo aver trasformato l'indirizzo. Ancora una volta entrambe le CPU ricevono lo stesso valore e proseguono di pari passo.



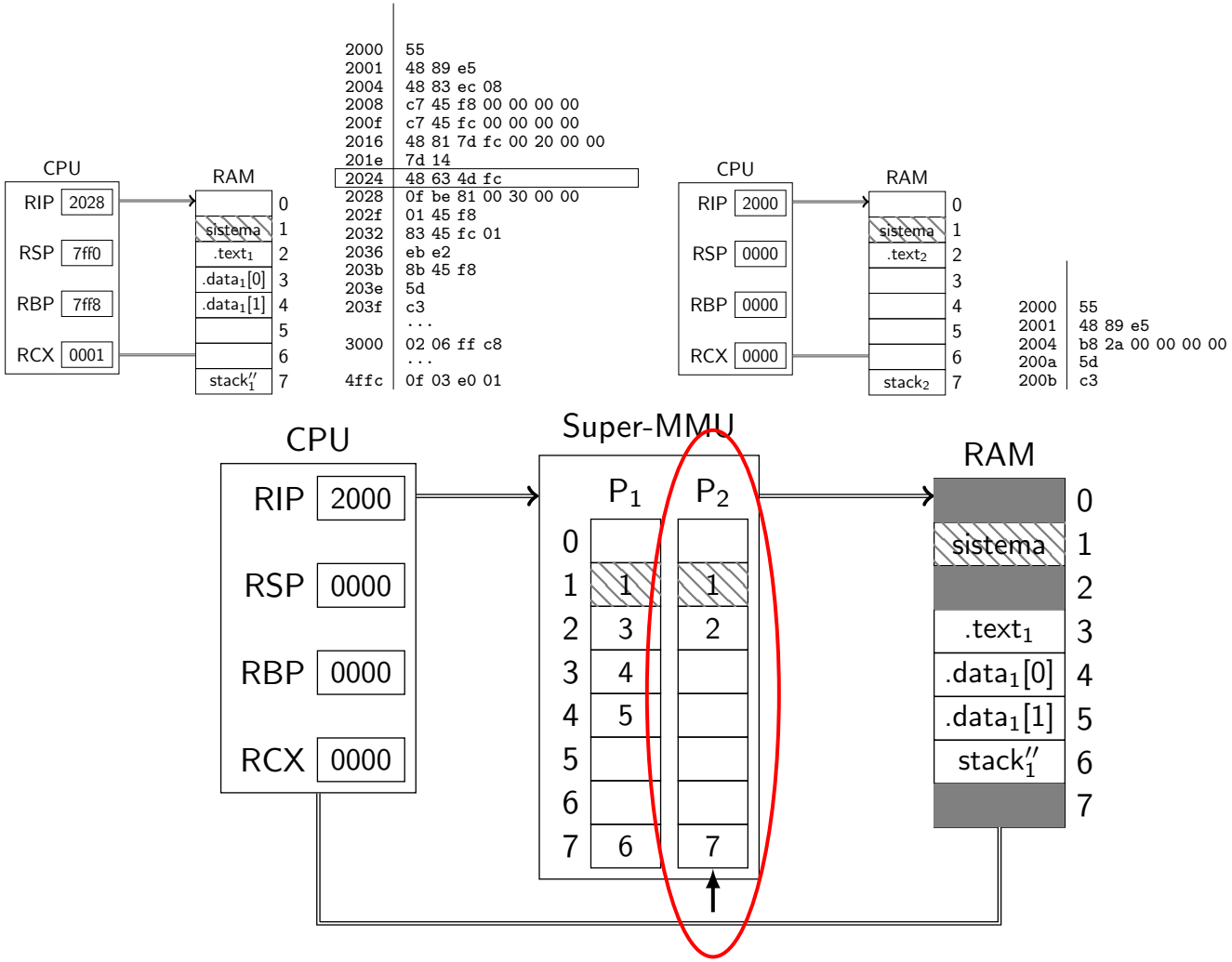
L'esecuzione prosegue aggiungendo al totale il valore precedentemente letto e incrementando `%rcx`, fino a quando si ritorna all'indirizzo 2028.



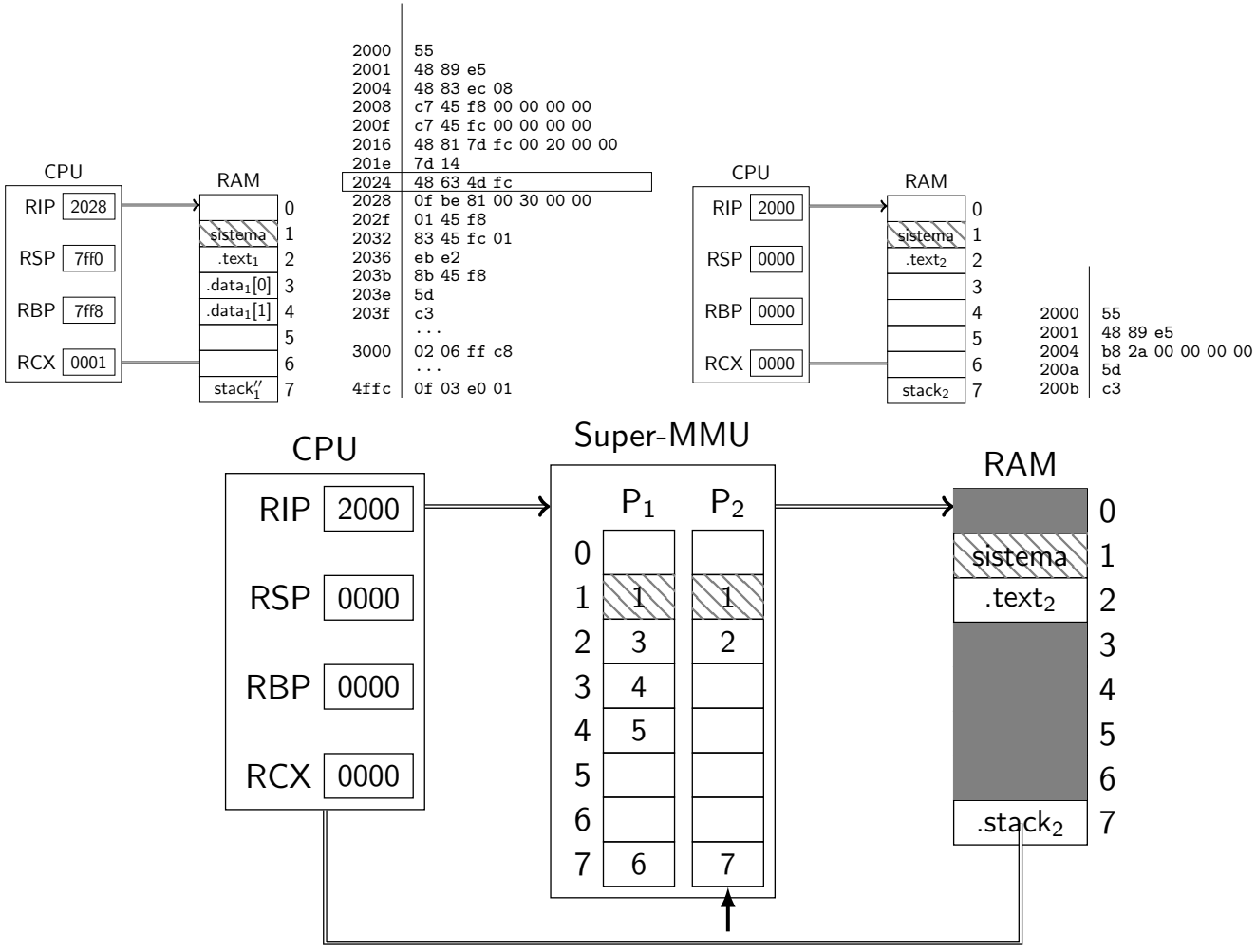
Supponiamo che a questo punto, prima di prelevare la prossima istruzione, il sistema fisico accetti una interruzione con cambio di processo e vada in esecuzione P_2



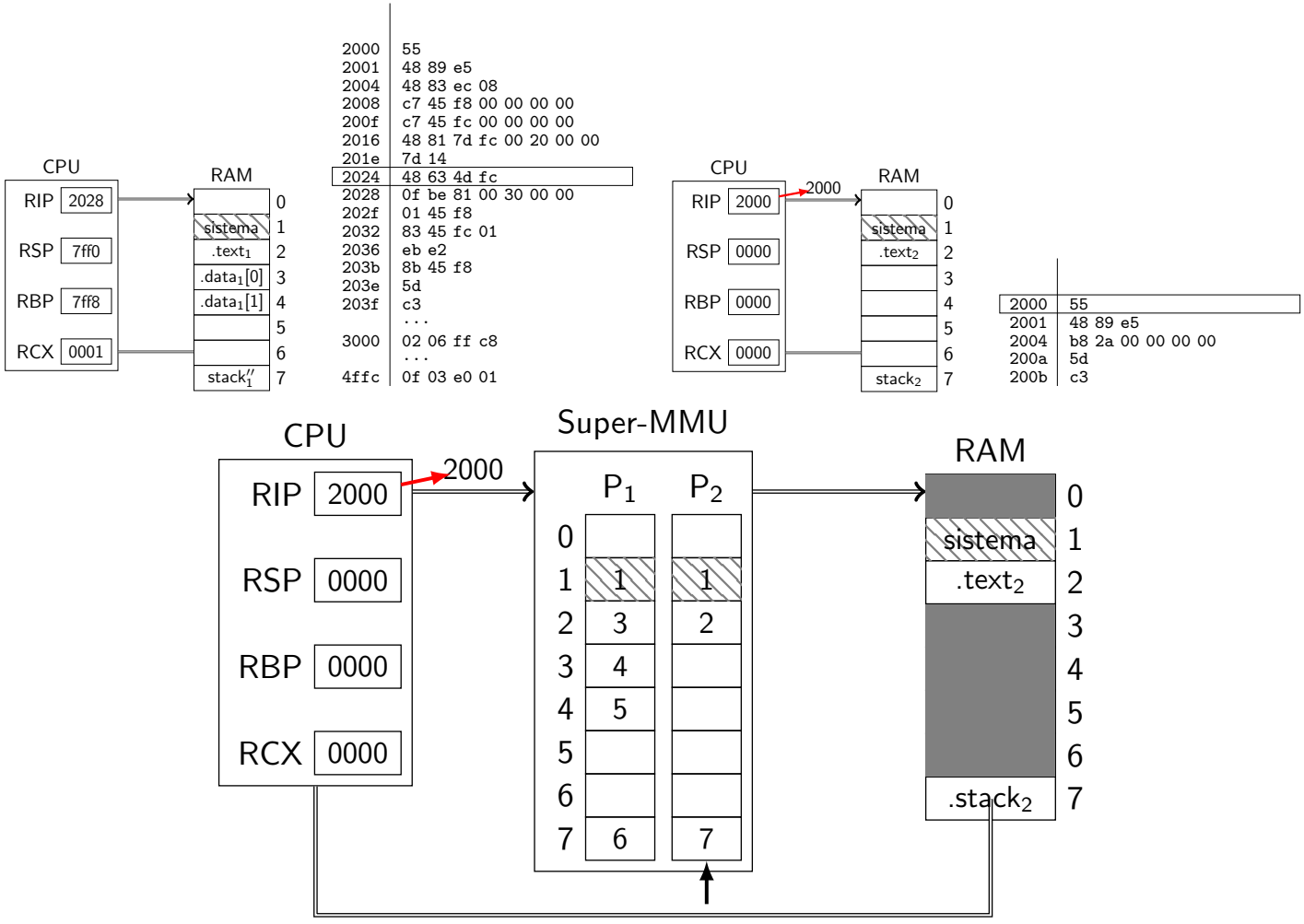
Il sistema carica i registri di P_2...



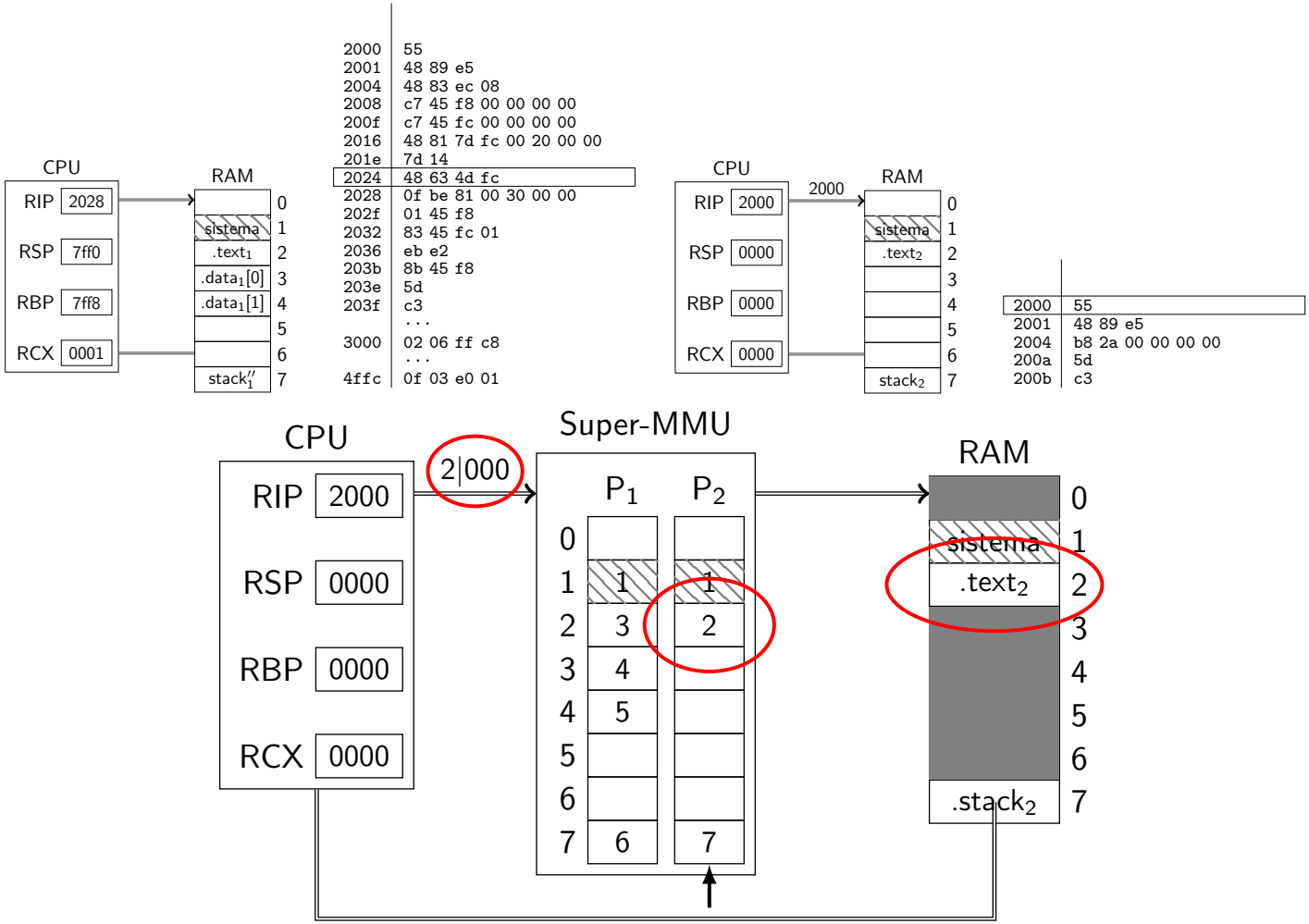
. . . e attiva la tabella P_2



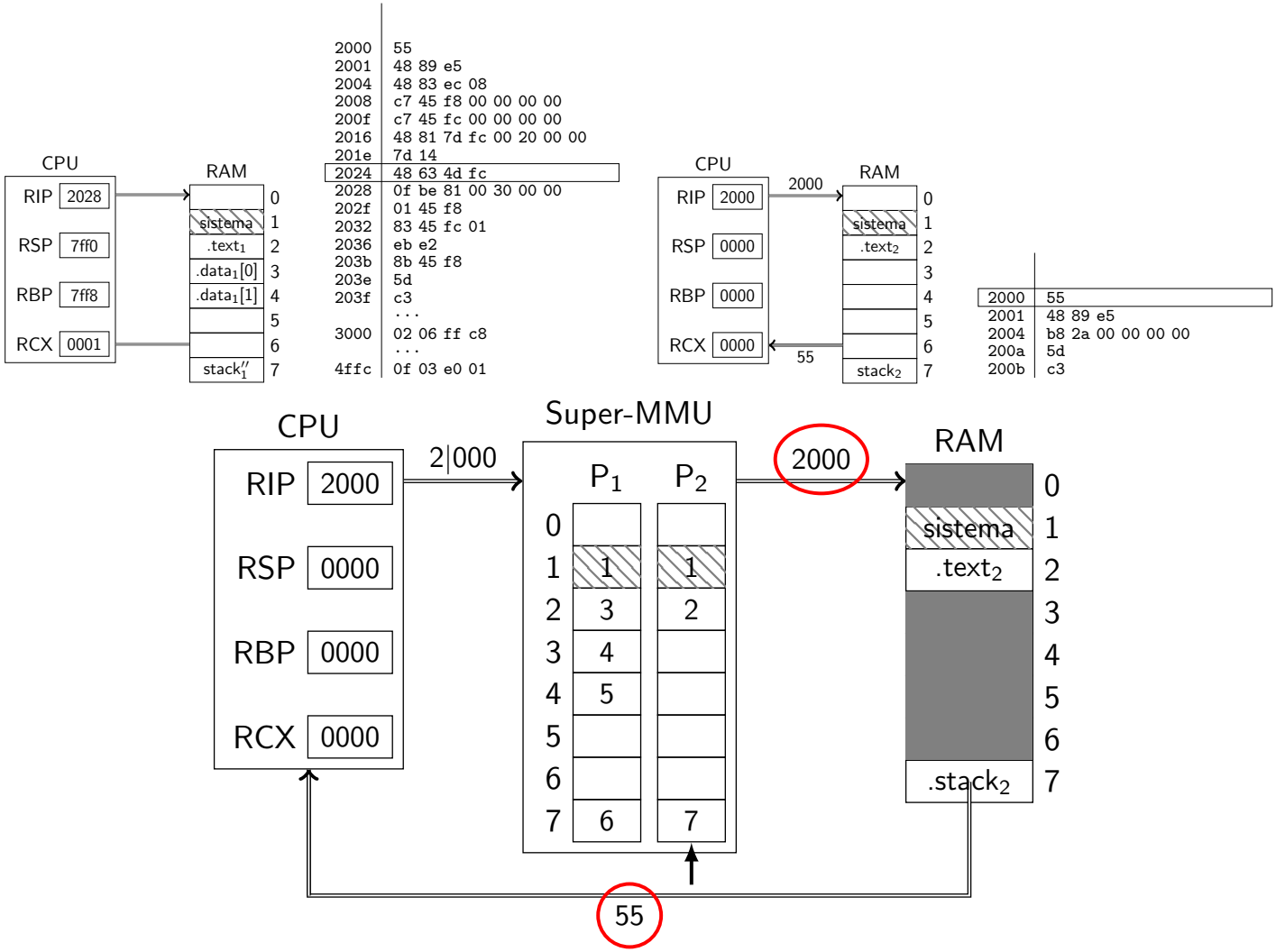
Diventano accessibili le pagine nel codominio di P_2 e non accessibili tutte le altre. Il sistema virtuale di P_1 è "congelato".



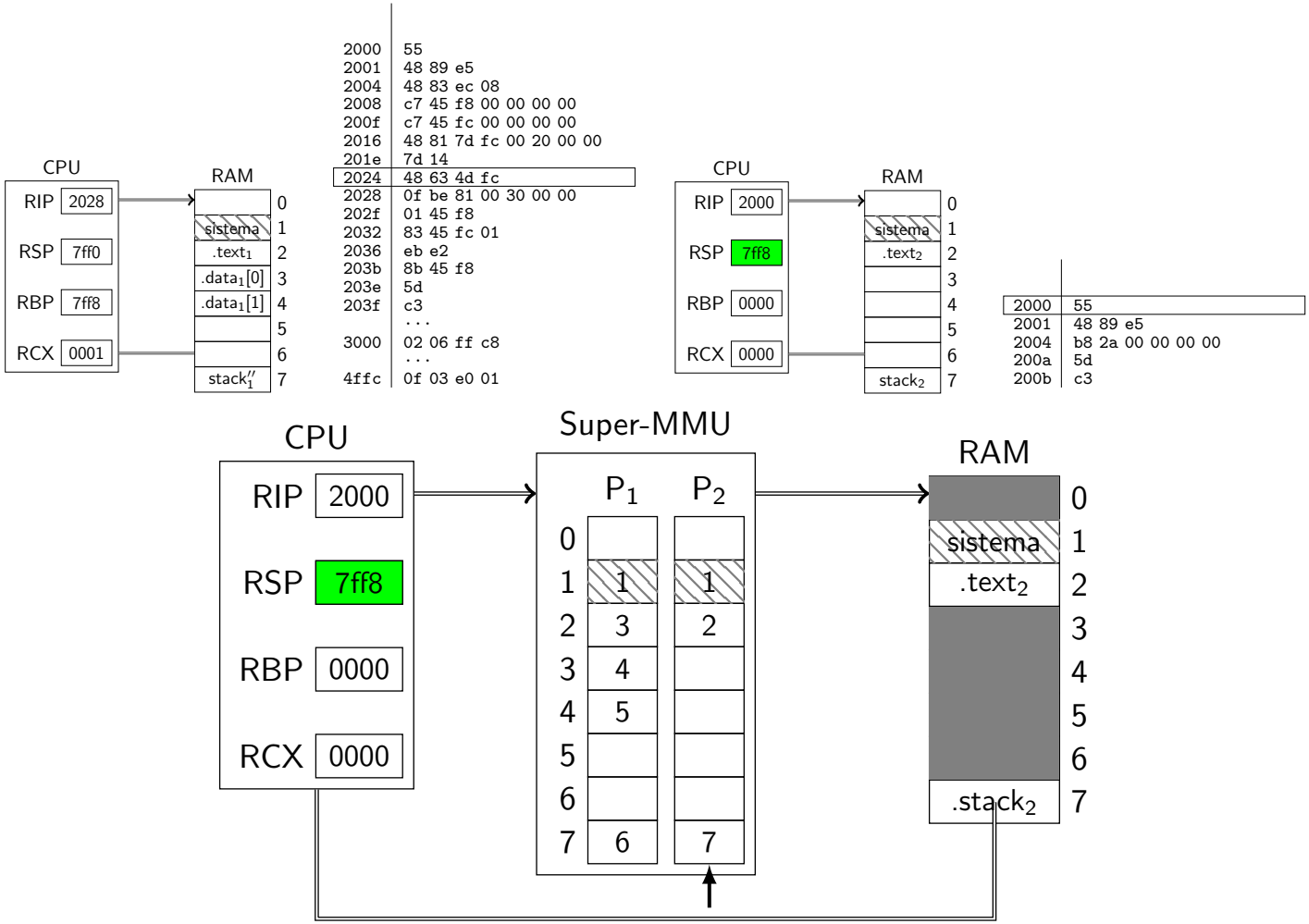
P_2 comincia la sua esecuzione. La CPU fisica e la CPU virtuale di P_2 eseguono una lettura all'indirizzo 2000.



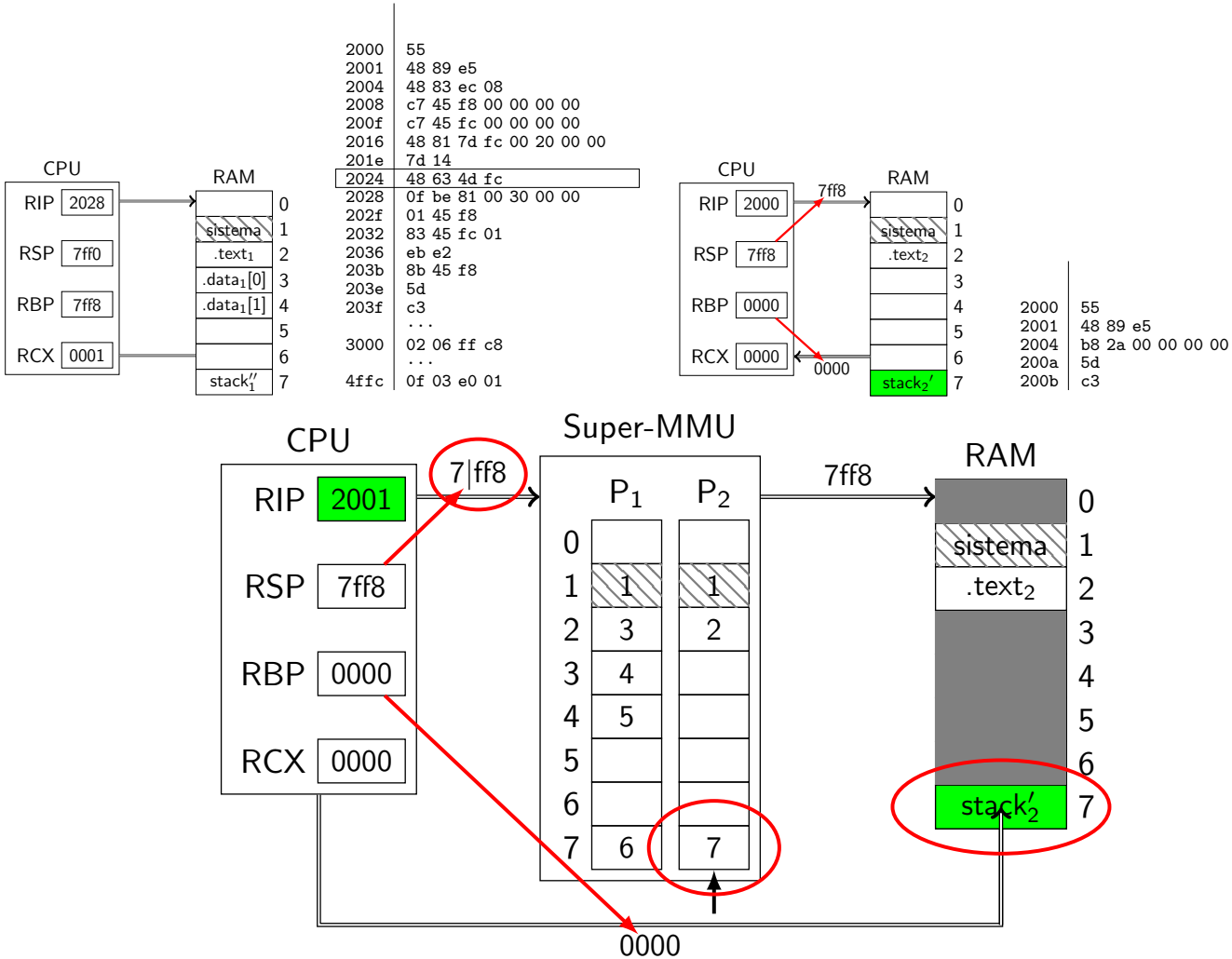
La MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina (2) e offset (000). Consulta quindi l'entrata numero 2 della tabella di corrispondenza e trova il corrispondente numero di frame (2)



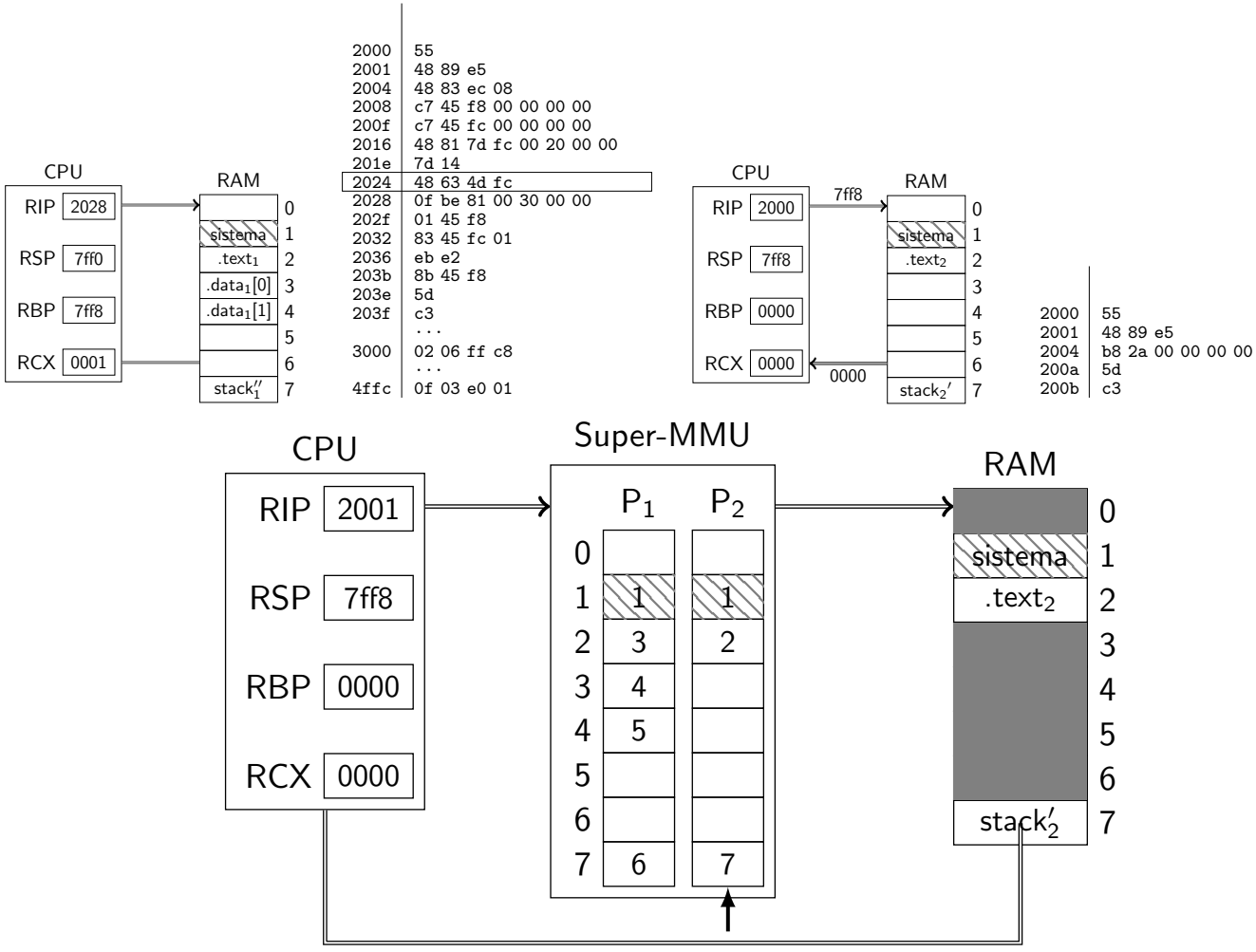
L'accesso viene completato dopo aver tradotto l'indirizzo.
La prima istruzione di P_2 è una `pushq %rbp`.



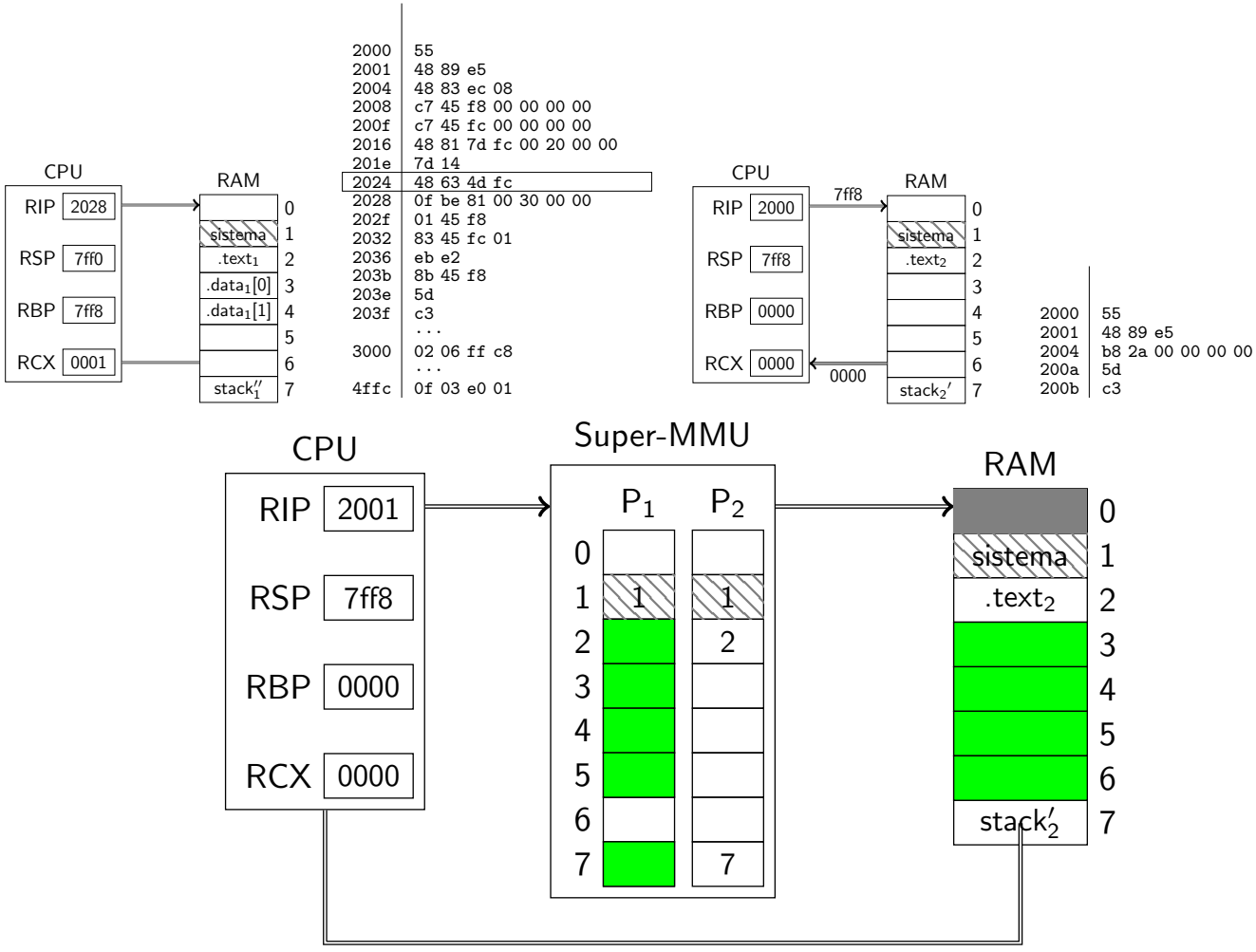
Per eseguirla, entrambe le CPU decrementano RSP di 8 . . .



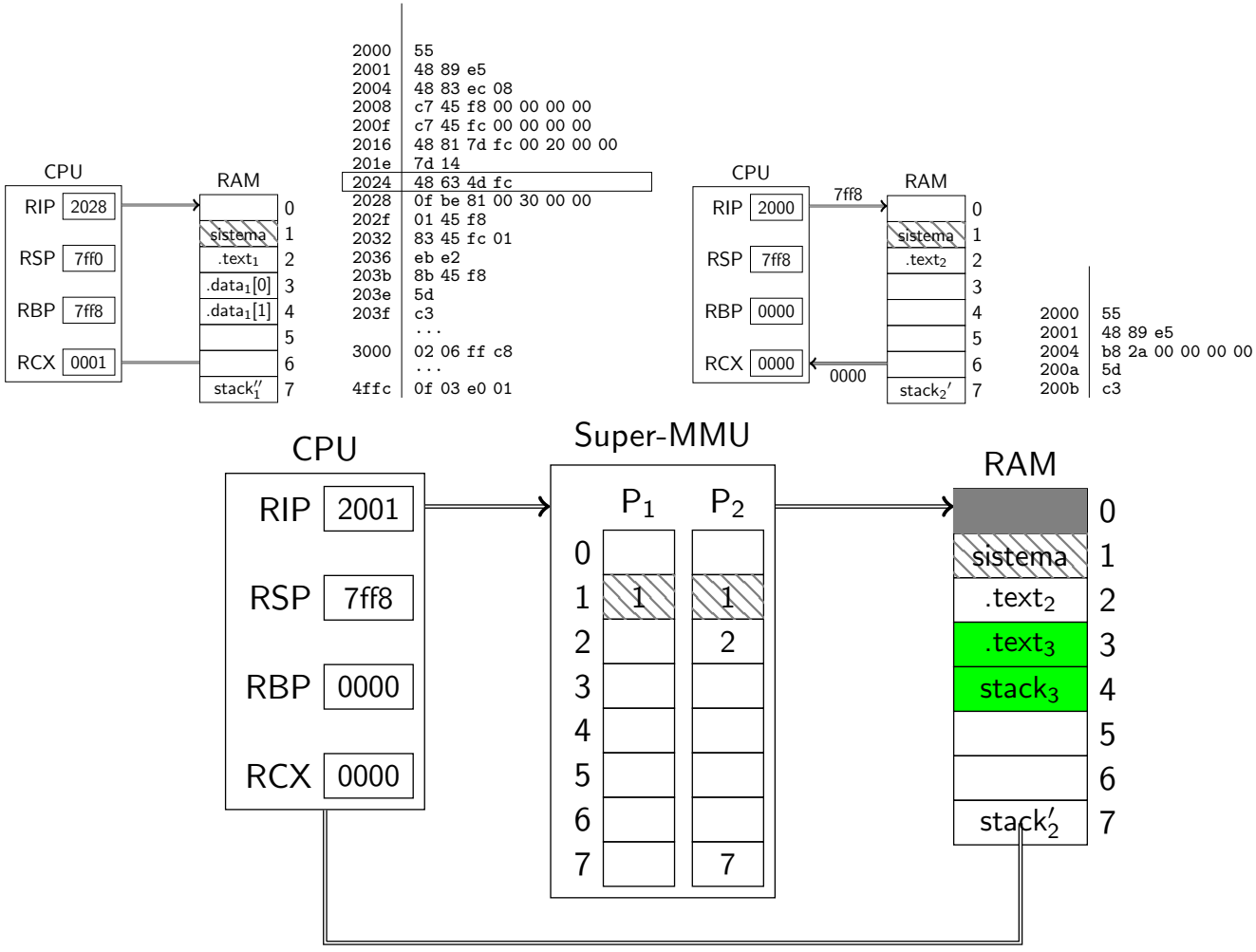
. . . e scrivono il contenuto di RBP all'indirizzo contenuto in RSP. La MMU traduce opportunamente l'indirizzo (in questo caso resta invariato)



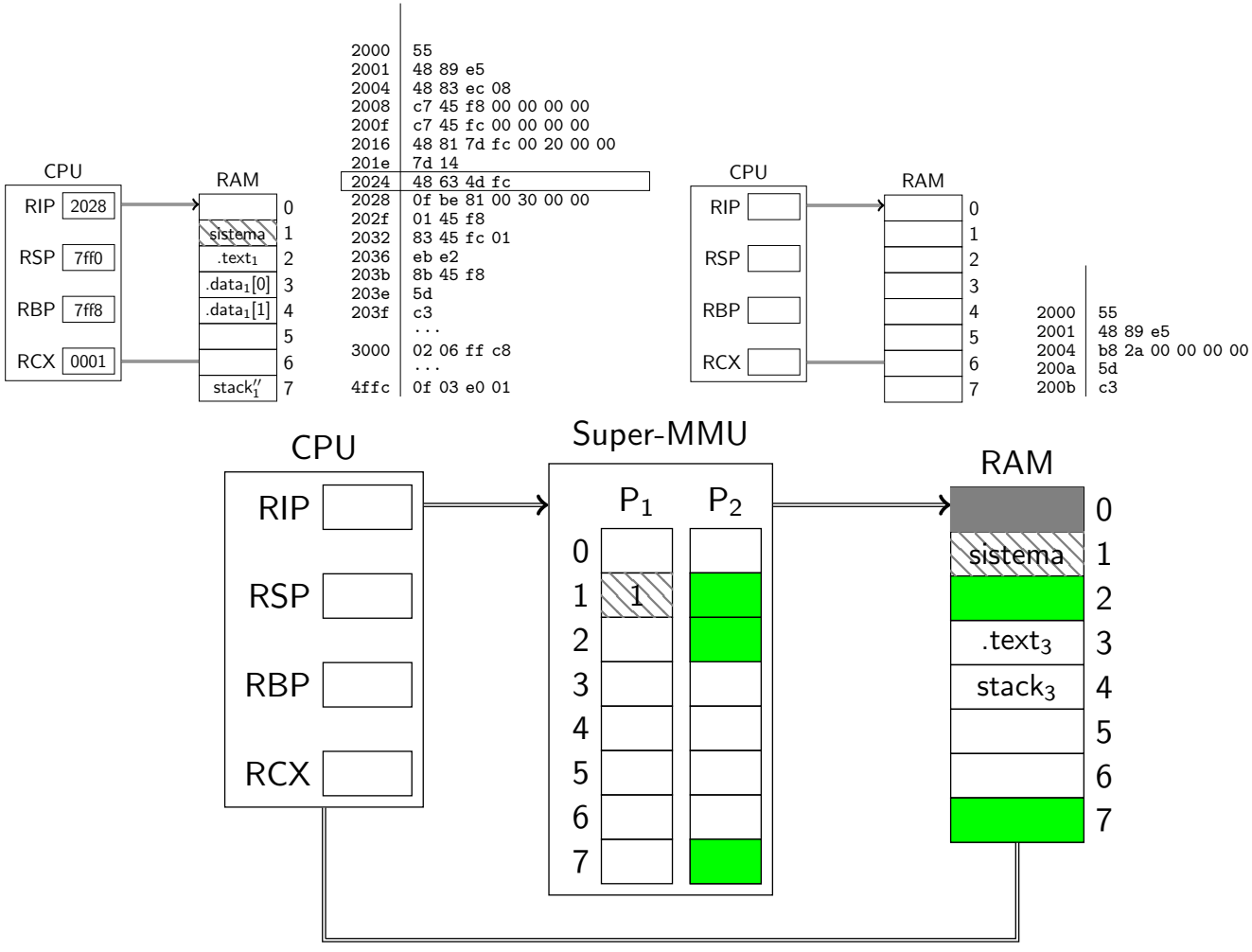
Supponiamo che ora che il sistema fisico venga interrotto nuovamente decida di caricare un altro processo, P_3.



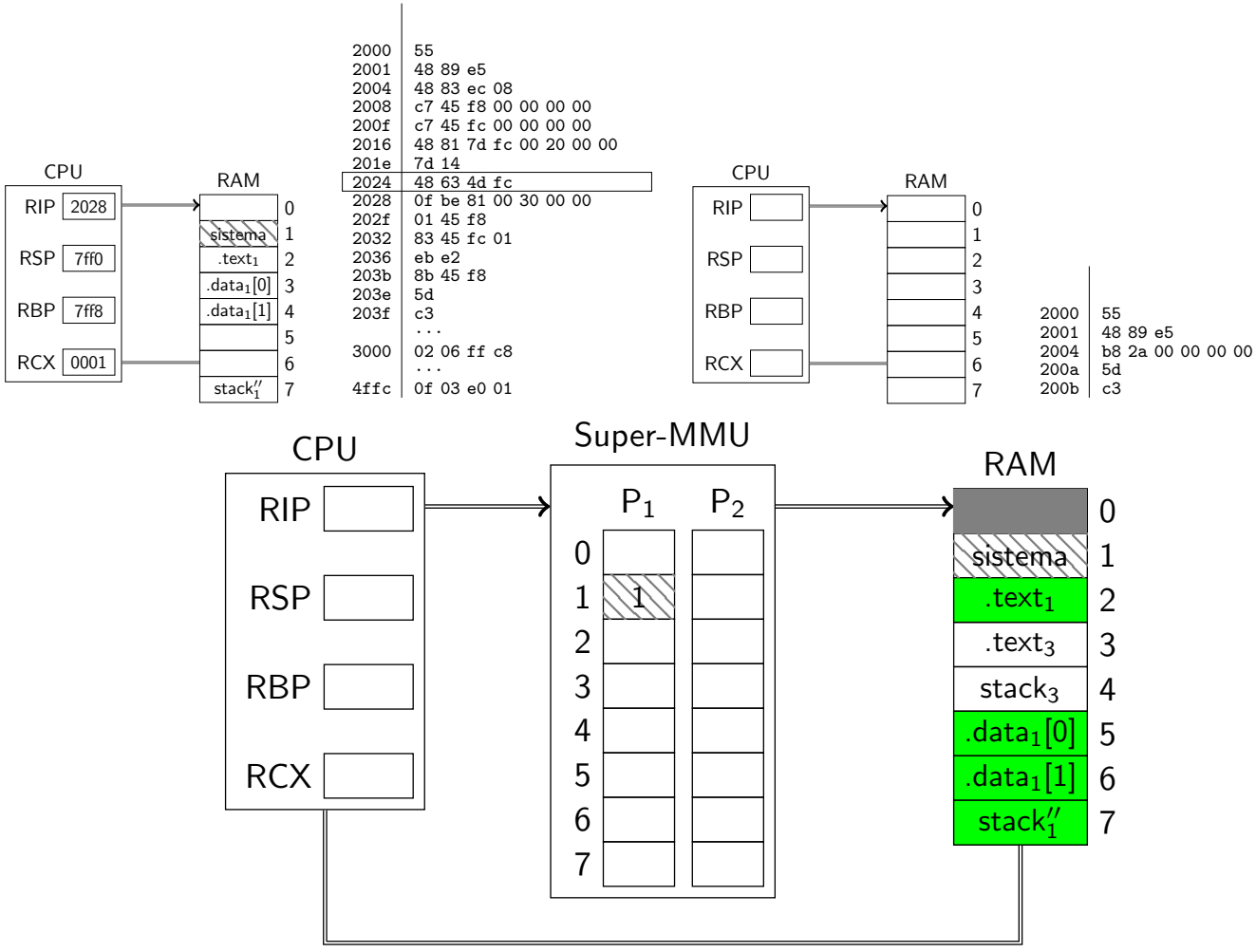
Per fare spazio, il sistema rimuove le pagine di P_1 dopo averle copiate nello swap (operazione di *swap-out*.



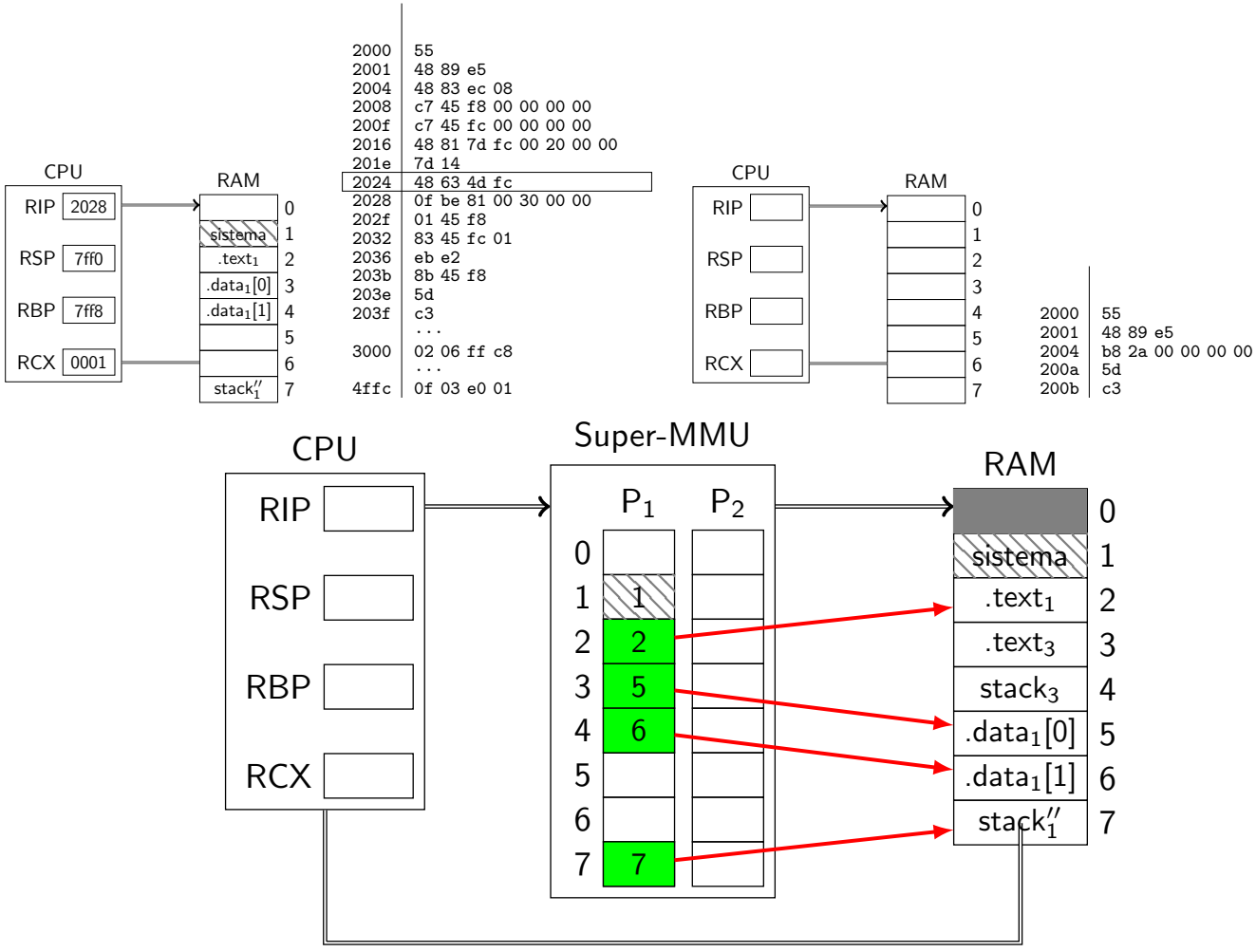
Quindi carica P_3 e inizializza opportunamente la sua tabella di corrispondenza (non mostrata).



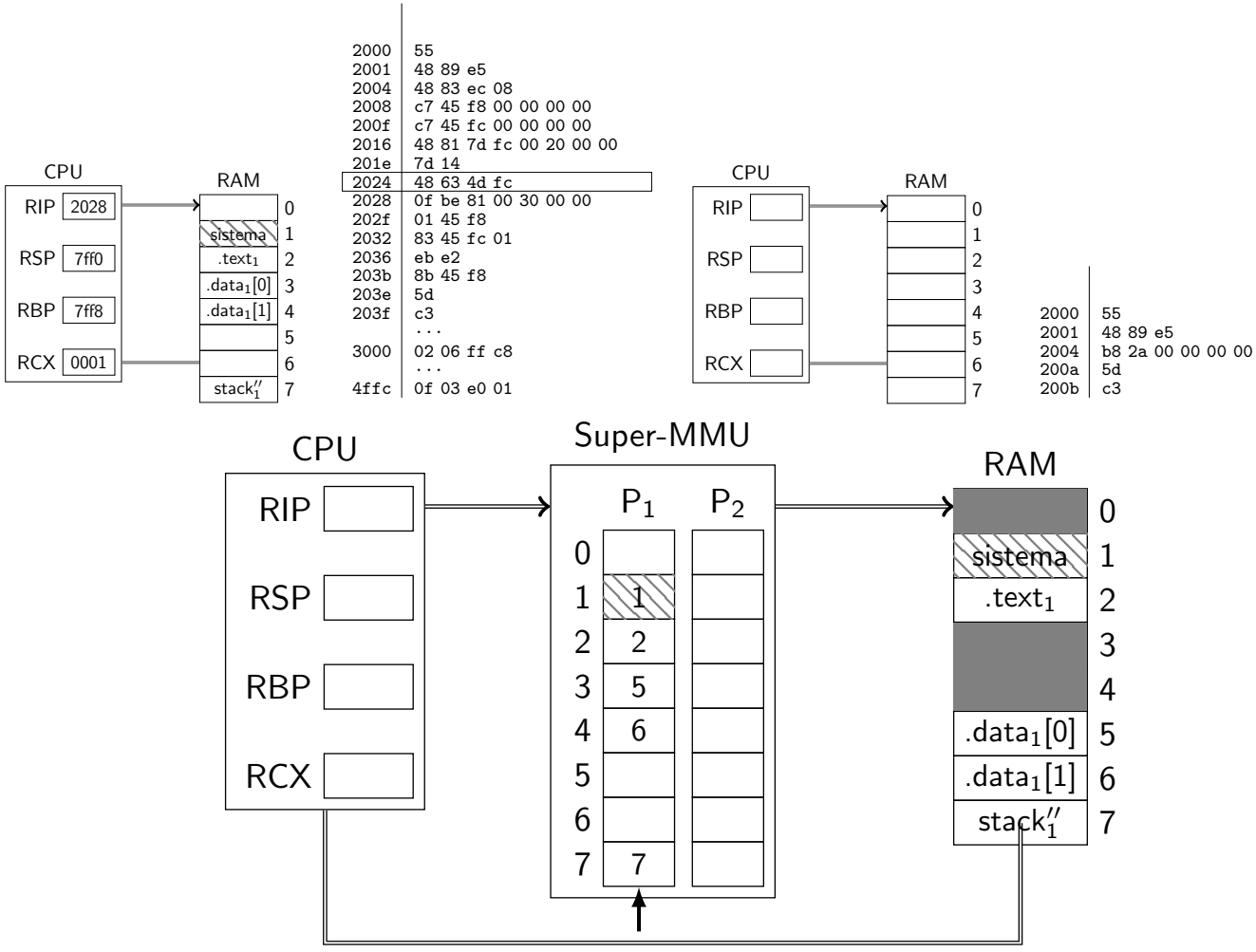
Successivamente, P_2 termina. Il sistema libera tutte le sue pagine.



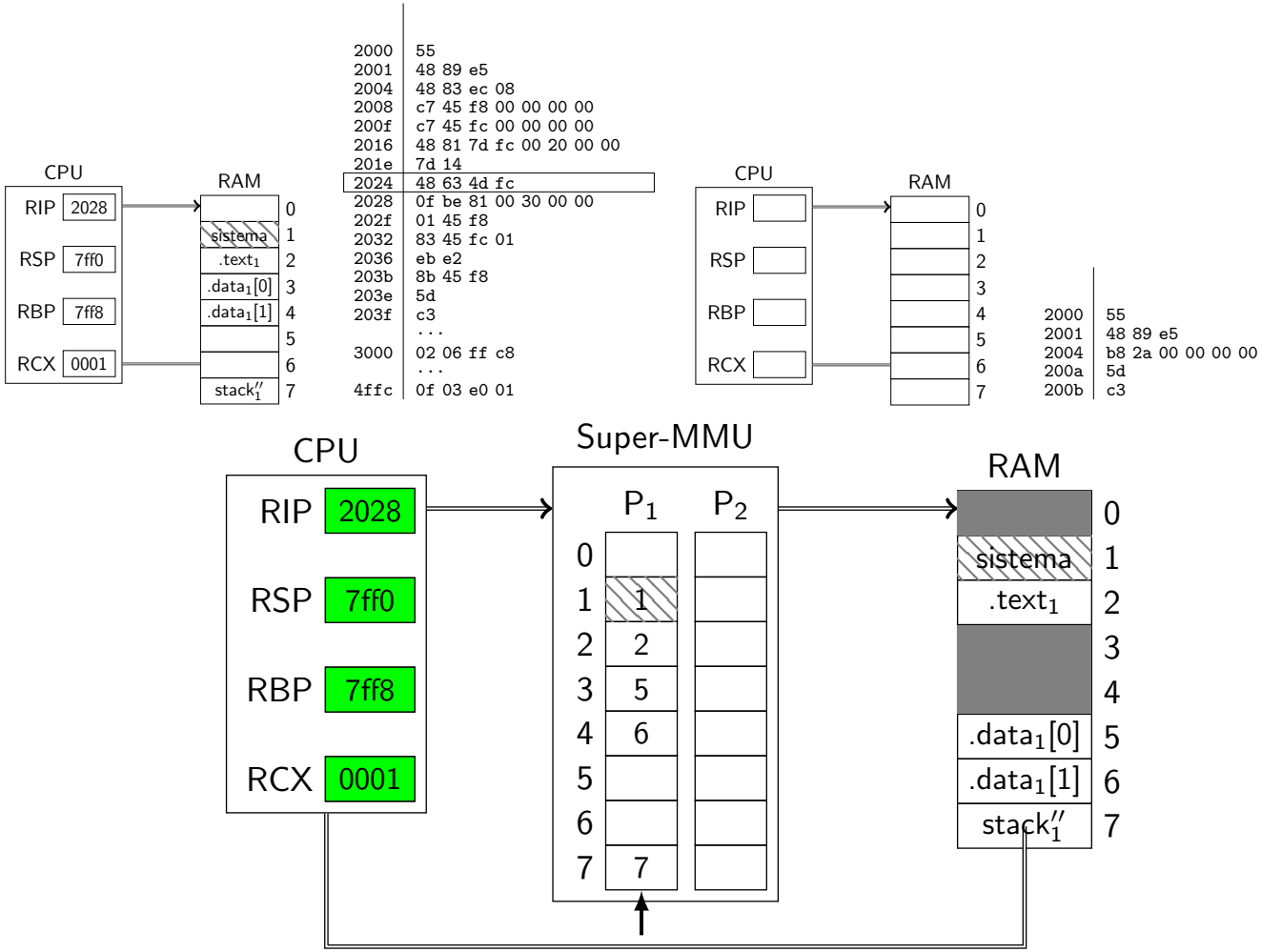
Ancora dopo, il sistema decide di ricaricare P_1 per rimetterlo in esecuzione. Le pagine di P_1 non occupano più i frame che occupavano in precedenza.



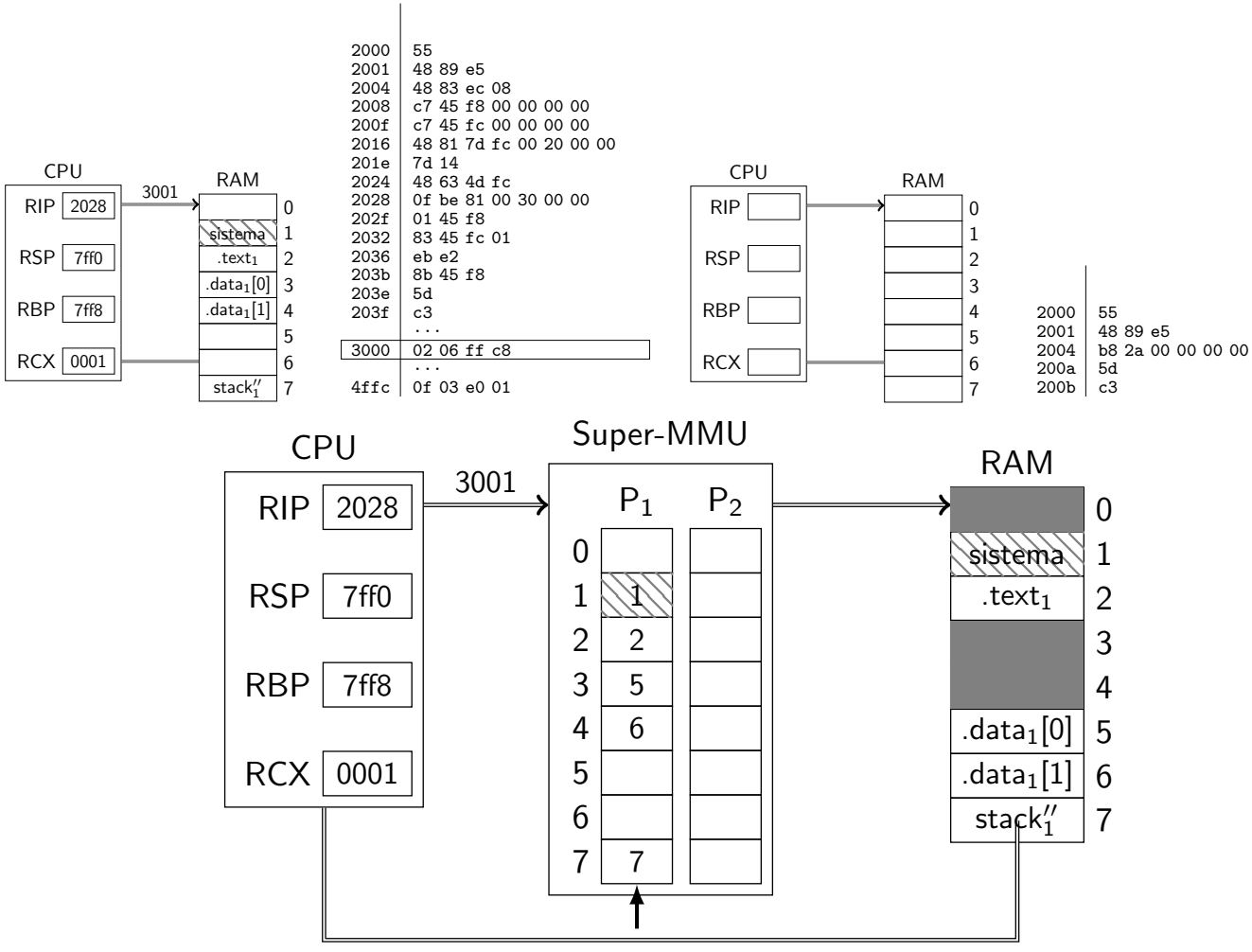
Il sistema inizializza la tabella di corrispondenza di P_1 con i nuovi numeri di frame.



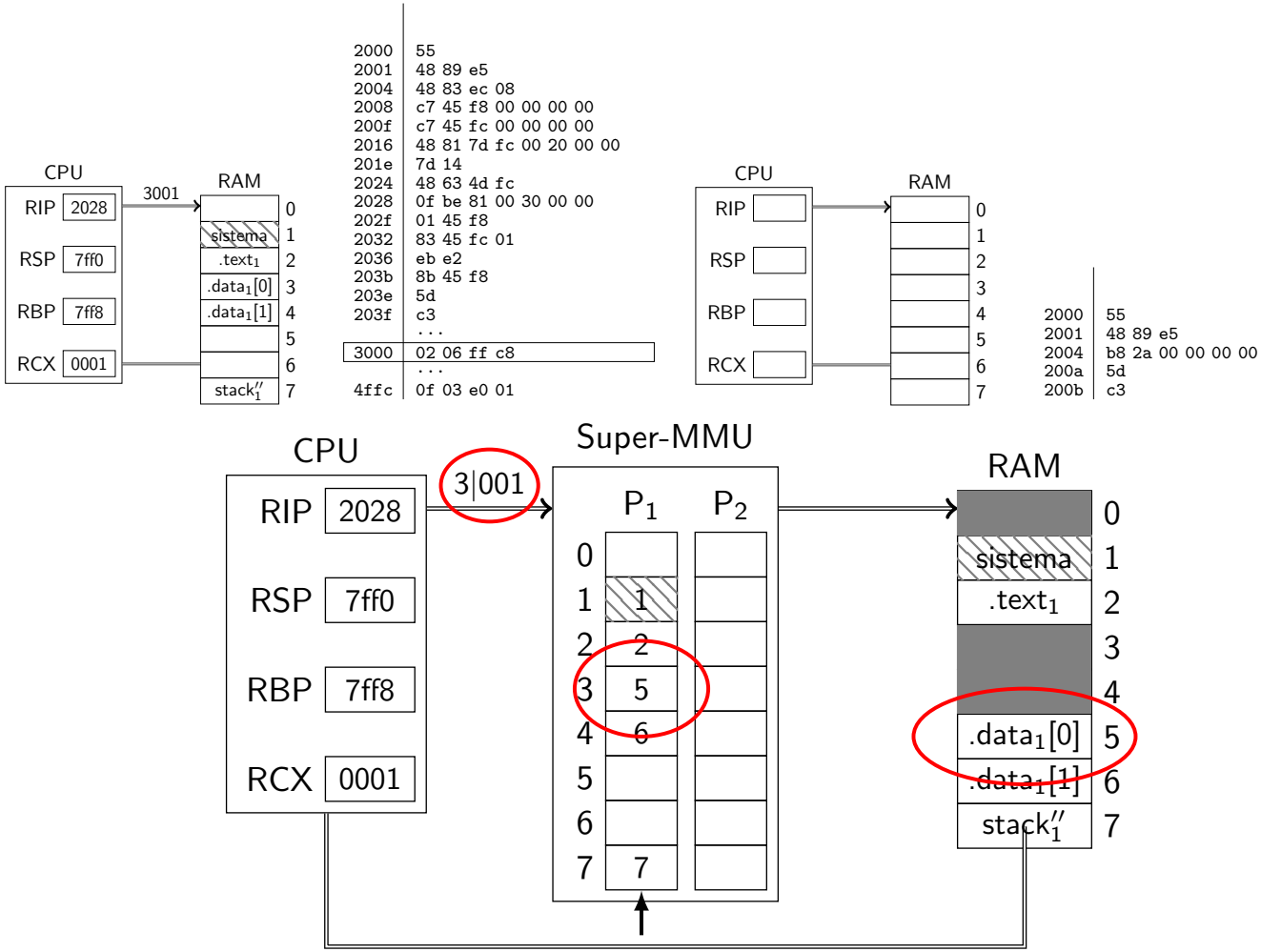
Attiva la tabella di P_1. Le pagine di P_3 diventano inaccessibili.



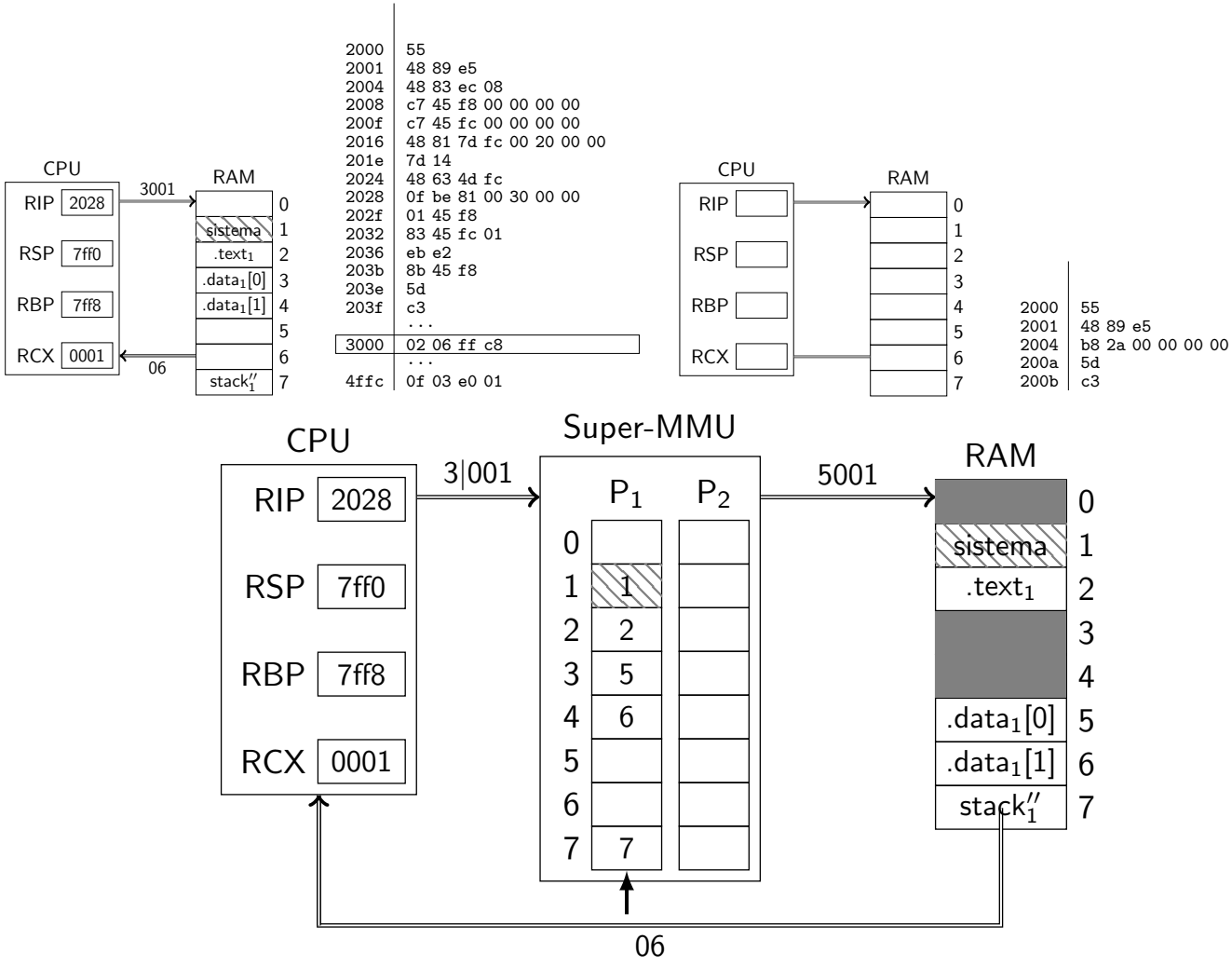
Infine, ricarica i registri con l'ultimo stato salvato di P_1 e gli cede il controllo. Ricordiamo che stava per prelevare `movsbl buf(%rcx), %eax` all'indirizzo 2028.



La CPU fisica e la virtuale (di P_1) prelevano l'istruzione, quindi sommano il contenuto di RCX e la costante 3000, ottenendo l'indirizzo 3001. Iniziano dunque una operazione di lettura a questo indirizzo.



Ancora una volta la MMU scompone l'indirizzo in numero di pagina (3) e offset (001). L'entrata numero 3 della tabella di corrispondenza dice che la pagina 3 si trova nel frame 5.



L'accesso viene completato, sempre dopo aver trasformato l'indirizzo. Entrambe le CPU ricevono lo stesso valore e continuano a procedere di pari passo, nonostante il fatto che le pagine di P_1 siano state spostate.