

# **Arhitectura memoriei kitului RCM 3365. Modelul de memorie al microprocesorului Rabbit 3000**

## **1. Scopul lucrării**

Scopul acestei lucrări de laborator este familiarizarea studenților cu arhitectura memoriei sistemului de dezvoltare RCM 3365, precum și cu modelul de memorie al microprocesorului Rabbit 3000. Se vor studia unitățile de interfațare și mapare a memoriei, va fi prezentat un model de mapare a memoriei și se va discuta despre spațiul de instrucțiuni și date combinat sau separat.

## **2. Memoria kitului de dezvoltare RCM 3365**

Sistemele RCM 3365 prezintă memorie de tip SRAM, Flash EPROM și Flash NAND. Există un circuit de memorie SRAM cu capacitatea de 512KB folosit pentru execuția programelor (U11), precum și un circuit de memorie SRAM tot de 512 KB folosit ca memorie de date și care poate fi asigurat pe baterie (U10). De asemenea, pe sistemele RCM 3365 mai întâlnim și un circuit de memorie Flash cu capacitatea de 512 KB (U9), memorie folosită pentru stocarea programelor. Suplimentar, modulele RCM 3365 oferă și memorie de stocare de masă implementată sub forma unui circuit de memorie Flash NAND de 16 MB ce se găsește pe placă (U15), precum și sub forma cardurilor portabile de memorie xD-Picture, ce pot fi atașate extern (conectorul pentru cardul de memorie este J6). Mărimea maximă a cardurilor de memorie NAND Flash este de 128 MB. La sfârșitul lucrării de laborator se vor găsi schemele electrice ale modulului RCM 3365 (figurile 7 – 11).

Memoria NAND Flash și cardul xD-Picture sunt potrivite pentru aplicații de stocare în masă, dar neadevrate, în general, execuției directe a programului. Memoriile NAND Flash diferă de memoriile NOR Flash paralel (tipul de memorie Flash folosit pentru stocarea de cod pe circuitele Rabbit și modulele RabbitCore aflate în producție în prezent) prin două aspecte. Mai întâi, memoria NAND Flash are nevoie de cod corector de erori (ECC) pentru fiabilitate. Deși producătorii de memorii NAND Flash garantează că blocul 0 va fi fără erori, cei mai mulți producători garantează că un circuit de memorie NAND Flash nou va fi livrat cu un procent relativ mic de erori și că acesta nu va depăși un număr maxim sau un procent de erori pe timpul duratei sale de viață de până la 100000 de scrieri. În al doilea rând, metodele standard de adresare a memoriei NAND Flash multiplexează comenzi, date și adrese pe aceiași pini de I/O, timp în care cer ca anumite linii de control să fie păstrate stabile pe durata accesului la memoria NAND Flash. Apelurile de funcții software furnizate de către Z-World pentru NAND Flash se asigură de integritatea datelor și a atributelor de fiabilitate.

### 3. Interfațarea și maparea memoriei microprocesorului Rabbit 3000

Chiar dacă schema de mapare a memoriei la microprocesorul Rabbit este complexă, utilizatorul nu trebuie să se îngrijoreze, deoarece de detalii se ocupă sistemul de dezvoltare Dynamic C.

Cu excepția câtorva instrucțiuni speciale, instrucțiunile Rabbit adresează direct un spațiu de memorie de date de 64 KB. Acest lucru semnifică faptul că în cadrul instrucțiunilor, câmpul de adresă are 16 biți, și că regiștrii ce pot fi folosiți ca pointeri la memorie (regiștrii index IX și IY, PC și SP) au tot 16 biți.

Deoarece instrucțiunile Rabbit folosesc adresarea pe 16 biți, ele sunt mai scurte și se execută mai repede decât dacă am fi avut adrese pe 32 de biți de exemplu. Astfel, codul executabil este foarte compact.

Unitatea de mapare a microprocesorului Rabbit este similară, dar mult mai puternică decât cea a procesorului Z180. Figura 1 prezintă relațiile între componentele implicate în adresarea memoriei.

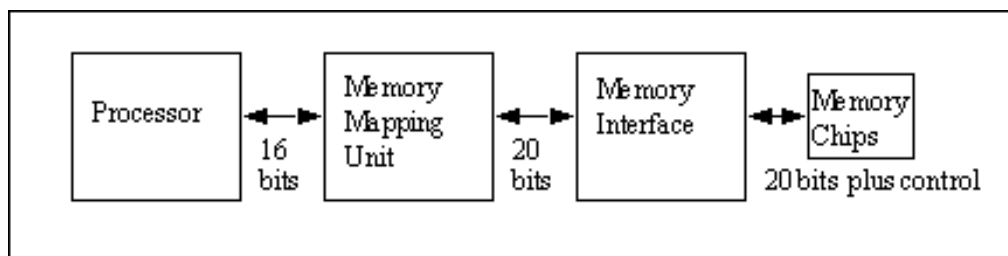


Figura 1. Componentele adresării memoriei

Unitatea de mapare a memoriei primește la intrare o adresă pe 16 biți și scoate la ieșire o adresă pe 20 de biți. Procesorul vede (cu excepția instrucțiunilor LDP) un spațiu de adrese de 16 biți, adică vede 65536 de octeți adresabili în mod distinct care pot fi manipulați de către instrucțiunile sale. Trei regiștri de segment sunt utilizați pentru a mapa acest spațiu de 16 biți într-un spațiu de 1 megaoctet. Spațiul de 16 biți este divizat în 4 zone separate. Fiecare zonă, exceptând prima (zona rădăcină), are un registru de segment care este adăugat adresei pe 16 biți din cadrul zonei pentru a crea o adresă pe 20 biți. Registrul de segment are 8 biți și acești 8 biți sunt adăugați la cei mai semnificativi 4 biți ai adresei pe 16 biți, formând adresa pe 20 biți. Astfel, fiecare zonă din memoria pe 16 biți devine o fereastră către un segment de memorie din spațiul de adresă pe 20 biți. Dimensiunea relativă a celor 4 segmente din spațiul de 16 biți este controlată de către registrul SEGSIZE. Acesta este un registru pe 8 biți care conține 2 regiștri de 4 biți, care controlează granița dintre primul și al 2-lea segment și granița dintre al 2-lea și al 3-lea segment. Locația celor 2 granițe mobile de segment este determinată de o valoare pe 4 biți care specifică cei mai semnificativi 4 biți ai adresei unde se află granița. Aceste relații sunt ilustrate în figura 2.

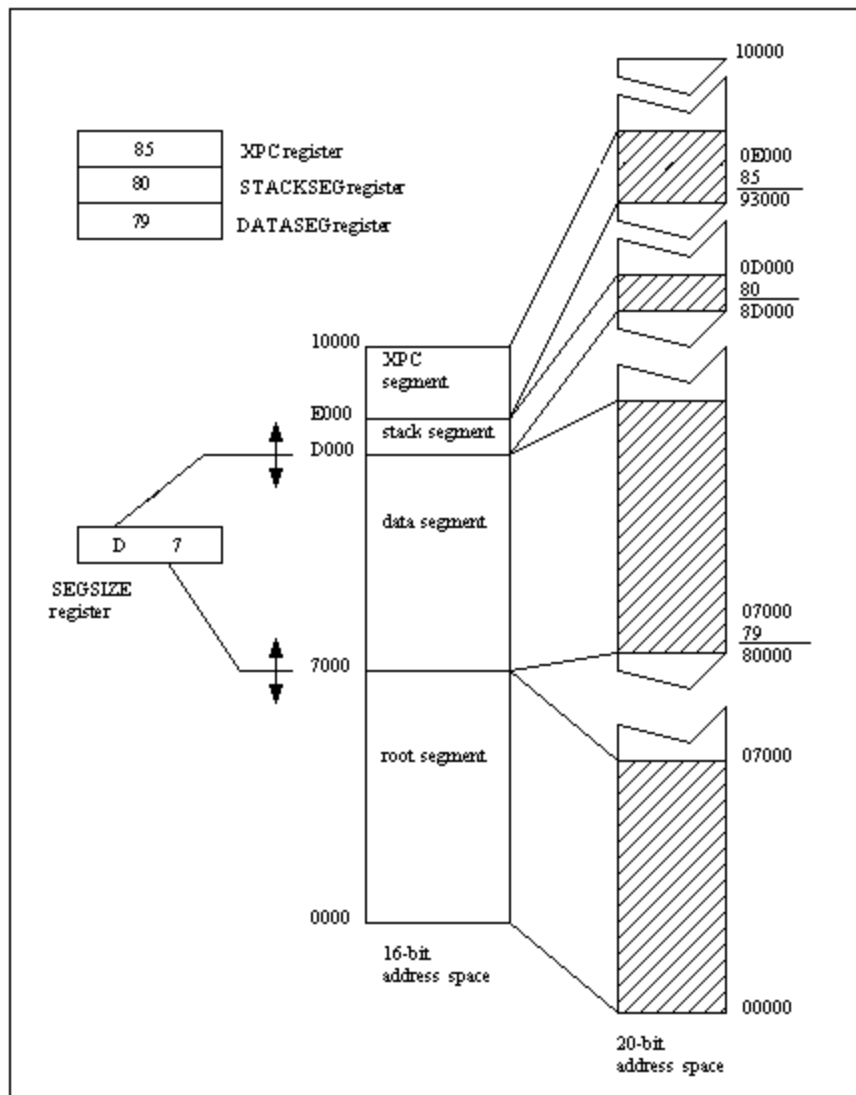


Figura 2. Exemplu de mapare a memoriei

Numele date segmentelor din figură evocă utilizarea comună pentru fiecare segment. Segmentul rădăcină este mapat către baza memorie flash și conține codul de pornire și orice alt cod posibil a fi stocat acolo. Utilizarea segmentului de date variază, depinzând de strategia de setare a memoriei. Poate fi o extensie a segmentului rădăcină sau poate conține date. Segmentul de stivă are în general lungimea de 4 KB și conține stiva sistemului. Segmentul XPC este utișizat în mod normal pentru a executa codul care nu este memorat nici în segmentul rădăcină, nici în cel de date. Instrucțiuni speciale suportă cod executabil care este vizibil în segmentul XPC.

Unitatea de interfață cu memoria primește adresele pe 20 de biți generate de unitatea de mapare a memoriei. Unitatea de interfață cu memoria modifică condiționat liniile de adresă A16, A18 și A19. Celelalte linii ale adresei pe 20 de biți sunt transmise necondiționat. Unitatea de interfață cu memoria furnizează semnale de control pentru cipurile de memorie externe. Aceste semnale de interfață sunt semnale de selecție a circuitului (chip select – /CS0, /CS1, /CS2), semnale de activare a ieșirii (output enable – /OE0, /OE1) și semnale de

permitere a scrierii (write enable –  $\overline{WE0}$ ,  $\overline{WE1}$ ). Aceste semnale corespund liniilor normale de control de la circuitele de memorie statică (chip select sau  $\overline{CS}$ , output enable sau  $\overline{OE}$  și write enable sau  $\overline{WE}$ ). Pentru a genera aceste semnale de control pentru memorie, spațiul de memorie de 20 biți este divizat în cadrante de 256 KB fiecare. Un registru pentru controlul bank-urilor pentru fiecare quadrant determină care dintre semnalele de chip select și care pereche de semnale de output enable și de write enable (dacă există) este activ atunci când are loc scrierea sau citirea din memorie din acel cvadrant. De exemplu, dacă o memorie flash de 512K x 8 urmează a fi accesată în primii 512 KB ai spațiului de adrese de 20 de biți, atunci  $\overline{CS0}$ ,  $\overline{WE0}$  și  $\overline{OE0}$  ar putea fi active în ambele cadrante.

Figura 3 prezintă o unitate de interfață cu memoria.

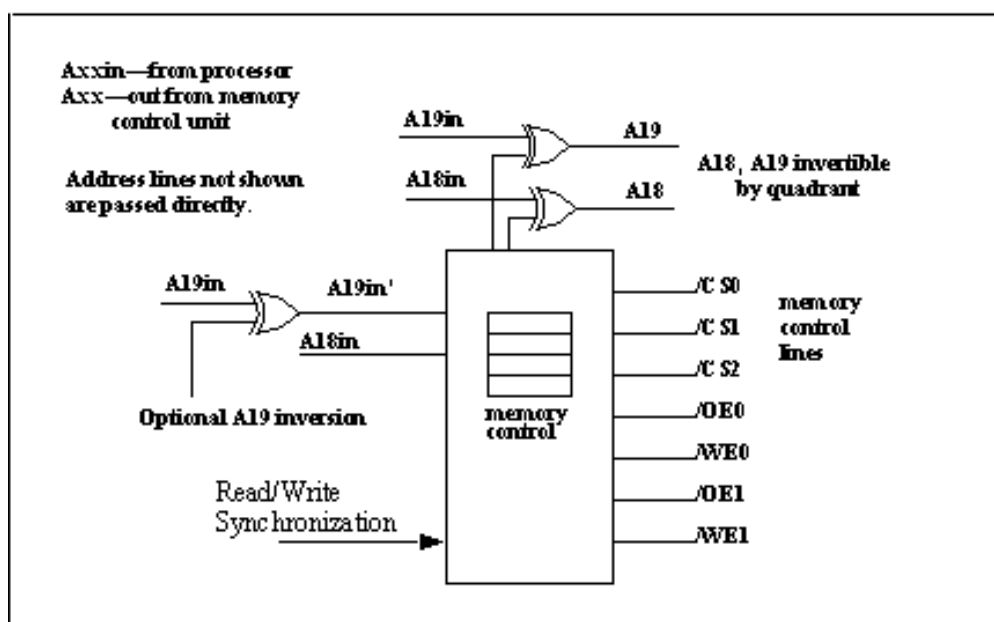


Figura 3. Unitatea de interfațare cu memoria

### 3.1. Spațiul extins de cod

Un element crucial al schemei de mapare a memoriei la microprocesorul Rabbit îl reprezintă capacitatea de a executa programe care conțin până la 1MB de cod într-o manieră eficientă. Această abilitate este absentă la un procesor pur pe 16 biți și este slab suportată de către microprocesorul Z180 prin intermediul unității sale de mapare a memoriei. La procesoarele paginate, cum ar fi 8086, această abilitate este furnizată prin paginarea spațiului de memorie de cod astfel încât codul să fie stocat în mai multe pagini separate. La 8086 mărimea paginii este de 64K, astfel încât tot codul dintr-o pagină dată este accesibil utilizând o adresare pe 16 biți pentru salturi, apeluri (call) și reveniri. Când este utilizată paginarea, un registru separat (CS la 8086) este utilizat pentru a determina unde se află pagina activă în spațiul total de memorie. Instrucțiuni speciale fac posibile salturi, apelurile și revenirile de la o pagină la alta. Aceste instrucțiuni speciale sunt denumite salturi lungi (long jumps), apeluri lungi (long calls) și reveniri lungi (long returns), pentru a le deosebi de aceleași operații care lucrează doar cu variabile pe 16 biți.

Microprocesorul Rabbit utilizează de asemenea o schemă de paginare pentru a extinde spațiul de cod dincolo de limita adresării pe 16 biți. Schema de paginare a microprocesorului Rabbit utilizează conceptul de pagină mobilă, care are lungimea de 8K. Acesta este segmentul XPC. Registrul XPC pe 8 biți servește ca și registru de pagină pentru a specifica partea de memorie către care pointează fereastra. Când se execută un program în segmentul XPC, sunt utilizate salturi, apeluri și reveniri normale pe 16 biți pentru majoritatea salturilor din cadrul ferestrei. Salturi, apeluri și reveniri normale pe 16 biți pot fi de asemenea utilizate pentru a accesa cod în celelalte 3 segmente din spațiul de adresă de 16 biți. Dacă este necesar un transfer de control către cod din afara ferestrei, atunci se utilizează salturi lungi, apeluri lungi sau reveniri lungi. Aceste instrucțiuni modifică atât numărătorul de program (program counter – PC), cât și registrul XPC, determinând fereastra XPC să poarte către o altă zonă de memorie, unde este localizată ținta saltului lung, a apelului lung sau a revenirii lungi. Segmentul XPC are întotdeauna lungimea de 8K. Granularitatea cu care segmentul XPC poate fi poziționat în memorie este de 4K. Deoarece fereastra poate fi deplasată cu o jumătate din lungimea sa, este posibilă compilarea continuă, fără spații neutilizate în memorie.

În timp ce compilatorul generează cod rezident în fereastra XPC, fereastra alunecă în jos cu 4K atunci când codul depășește adresa 0xF000. Acest lucru este realizat printr-un salt lung care repositionează fereastra cu 4K mai jos, fapt ilustrat în figura 4. Compilatorul nu are o graniță bine definită la sfârșitul unei pagini deoarece fereastra nu rămâne fără spațiu când codul depășește 0xF000 decât dacă 4K sau mai mult de cod este adăugat înainte ca fereastra să alunece în jos. Tot codul compilat pentru fereastra XPC are o adresă pe 24 de biți constând în XPC-ul pe 8 biți și adresa pe 16 biți. Salturi scurte și apelurile scurte pot fi utilizate, cu condiția ca instrucțiunile sursă și destinație să aibă aceeași adresă XPC. În general asta înseamnă că fiecare instrucțiune aparține unei ferestre care are lungimea de aproximativ 4K și are o adresă de 16 biți între 0xE000+n și 0xF000+m, unde n și m sunt de ordinul a câtorva zeci de octeți, dar pot ajunge până la 4096 octeți în lungime. Deoarece fereastra e limitată la o lungime de cel mult 8K, compilatorul nu poate compila o singură expresie care necesită mai mult de 8K de spațiu de cod. Aceasta nu este o considerație practică, deoarece expresiile mai lungi de câteva sute de octeți nu se întâlnesc în programele practice.

Codul programului se poate afla în segmentul rădăcină sau în segmentul XPC. Codul program poate de asemenea să se găsească în segmentul de date. Codul poate fi executat și în segmentul de stivă, dar acest lucru este în general restricționat doar pentru situații speciale. Codul din rădăcină, însemnând oricare din segmente în afară de segmentul XPC, poate să apeleze alt cod din rădăcină utilizând salturi și apeluri scurte. Codul din segmentul XPC poate de asemenea să apeleze cod din segmentul rădăcină folosind salturi sau apeluri scurte. Totuși, atunci când se apelează cod din segmentul XPC, trebuie utilizat un apel lung. Funcțiile localizate în rădăcină au un avantaj din punct de vedere al eficienței, deoarece un apel lung sau o revenire lungă necesită 32 de cicli de ceas pentru a se executa, pe când un apel scurt sau o revenire scurtă necesită numai 20 de cicli de ceas. Diferența e mică, dar semnificativă pentru subrutine scurte.

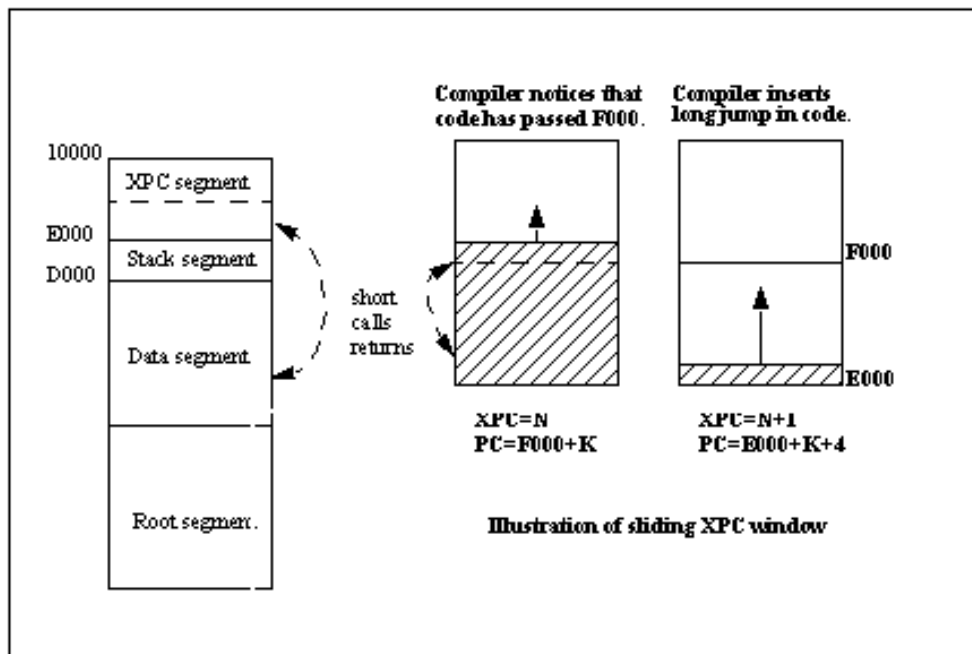


Figura 4. Utilizarea segmentului XPC

### 3.2. Spații separate pentru Instrucțiuni și Date – Extinderea memoriei de date

În modelul normal de memorie, spațiul de date trebuie să împartă un spațiu de 64K cu codul rădăcină, cu stiva și cu fereastra XPC. De obicei, acest lucru lasă un potențial spațiu de date de 40K sau mai puțin. XPC necesită 8K, stiva 4K și majoritatea sistemelor necesită cel puțin 12K de cod rădăcină. Această cantitate de spațiu de date este suficient pentru multe aplicații embedded.

O abordare pentru a obține mai mult spațiu de date este să se plaseze date în RAM sau în memoria flash care nu sunt mapate în spațiul de memorie de 64K și apoi să se acceseze aceste date utilizând apeluri de funcții sau, în limbaj de asamblare, instrucțiuni LDP care pot accesa memoria utilizând o adresă pe 20 de biți. Acest lucru extinde cu mult spațiul de date, dar instrucțiunile necesare accesării datelor sunt mai puțin eficiente decât instrucțiunile care accesează spațiul de 64K utilizând adrese pe 16 biți.

Microprocesorul Rabbit 3000 suportă spații de Instrucțiuni și Date separate. Când spațiile de instrucțiuni și date separate sunt activate, acest lucru se aplică numai pentru adresele din segmentul rădăcină sau din segmentul de date. Spații de instrucțiuni și date separate înseamnă că execuția unei instrucțiuni face distincție între aducerea unei instrucțiuni din memorie și aducerea sau stocarea de date în memorie. Când sunt activate, spațiile de instrucțiuni și date separate fac să fie disponibil segmentul combinat rădăcină și date, de obicei 52 KB pentru codul rădăcină în spațiul de instrucțiuni. În spațiul de date, segmentul codului rădăcină ce aparține spațiului de date este utilizat pentru constantele mapate în memoria flash, în timp ce segmentul de date ce aparține spațiului de date este folosit pentru datele variabile mapate în memoria RAM. Spațiul separat de instrucțiuni și date

mărește cantitate de cod rădăcină și date rădăcină, deoarece nu mai este necesar să împartă aceeași memorie, chiar dacă împart aceleași adrese.

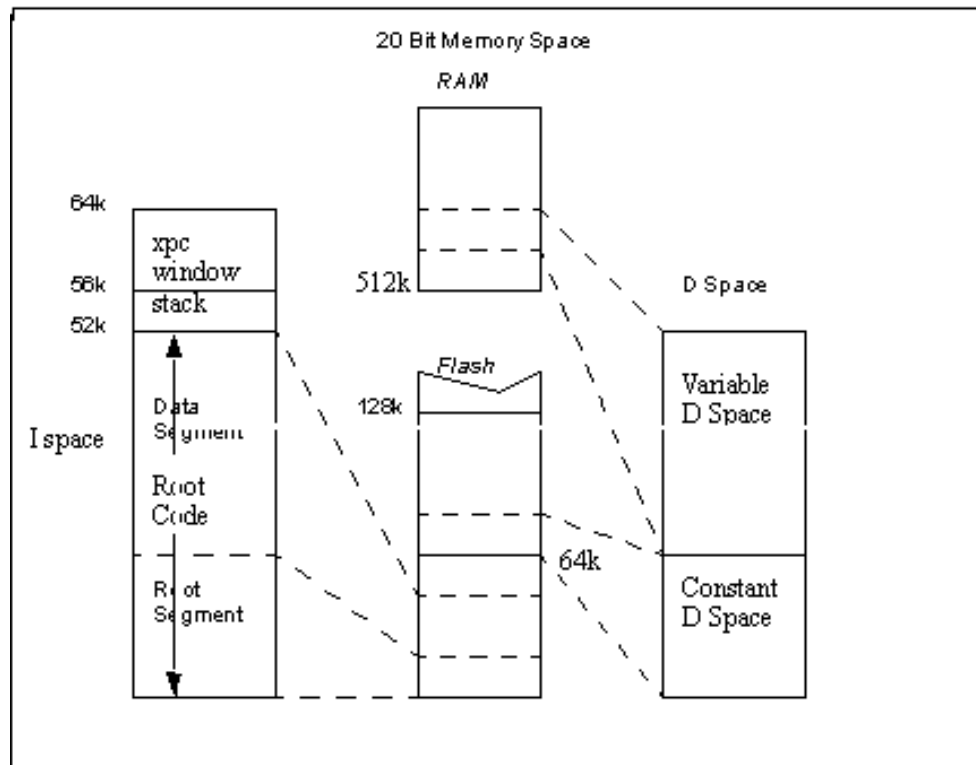


Figura 5. Spații separate pentru Instrucțiuni și Date

În mod normal, spațiile separate de instrucțiuni și date sunt implementate ca în figura 5. În spațiul de instrucțiuni, segmentul rădăcină și cel de date sunt combinate într-un singur segment de cod rădăcină. În spațiul de date, segmentele sunt mapate separat către flash sau RAM pentru a furniza spațiu de stocare pentru datele constante și variabile. Metoda hardware pentru a obține adrese separate pe 20 biți pentru spațiul de date este de a inversa fie A16, fie A19 pentru accesul la date. Această inversare poate fi specificată separat pentru segmentul rădăcină și pentru cel de date. În mod normal, A16 este inversat pentru accesul la datele din segmentul rădăcină. Acest lucru face ca accesul la datele din segmentul rădăcină să fie mutat mai sus cu 64K, către o zonă a memoriei flash începând la adresa pe 20 de biți 64K, care este rezervată pentru date constante. A19 este inversat de obicei pentru accesul la datele din segmentul de date, cauzând accesul la datele din segmentul de date să fie mutat la o adresă mai sus cu 512K în spațiul de 20 biți, o adresă mapată de obicei în memoria RAM. Segmentul de stivă și XPC nu au spații separate de instrucțiuni și date și accesul la memorie către aceste segmente nu fac deosebirea între spațiile de instrucțiuni și date.

Avantajul de a avea mai mult spațiu pentru codul rădăcină este următorul: codul rădăcină se execută mai repede deoarece, pentru apelul lui, sunt folosite apeluri scurte utilizând adrese pe 16 biți, în comparație cu apelurile lungi care au o adresă pe 20 de biți pentru cod extins. Datele localizate în rădăcină pot fi accesate mult mai comod față de cele aflate în spațiul de memorie de 20 de biți datorită numărului limitat de instrucțiuni valabile

pentru accesul datelor din spațiul de 20 de biți și datorită complexității mult mai mari în manipularea adreselor pe 20 biți la un procesor care are regiștri pe 8 și 16 biți.

### 3.3. Utilizarea segmentului de stivă pentru stocarea datelor

O altă abordare pentru extinderea memoriei de date este utilizarea segmentului de stivă pentru a accesa datele, plasând stiva în segmentul de date pentru a elibera segmentul de stivă. Această abordare funcționează bine pentru un sistem software care utilizează grupări de date care sunt independente și sunt accesate câte o înregistrare pe rând față de accesarea aleatoare între toate grupările. Un exemplu ar fi structurile software asociate cu o conexiune cu protocol de comunicare TCP/IP, unde același cod accesează structurile de date asociate cu fiecare conexiune după un model determinat de traficul pe fiecare conexiune.

Avantajul acestei abordări este că tehnicile normale C de acces la date, cum ar fi pointerii pe 16 biți, pot fi utilizate. Registrul segmentului de stivă trebuie modificat pentru a aduce structura de date la vedere în segmentul de stivă înainte ca operațiile să se execute pe o anumită structură de date particulară. Deoarece stiva trebuie mutată în zona de date, este important ca numărul de stive necesare să fie menținut la un minim atunci când se utilizează segmentul de stivă pentru a vizualiza date. Bineînțeles, taskurile care nu au nevoie să vadă structurile de date pot avea stiva lor în segmentul de stivă. O altă posibilitate este de a avea o structură de date și o stivă localizate împreună în segmentul de stivă, și să se utilizeze un alt segment de stivă pentru alte taskuri, fiecare task având propria zonă de date și stivă.

Aceste abordări sunt prezentate în figura 6 de mai jos.

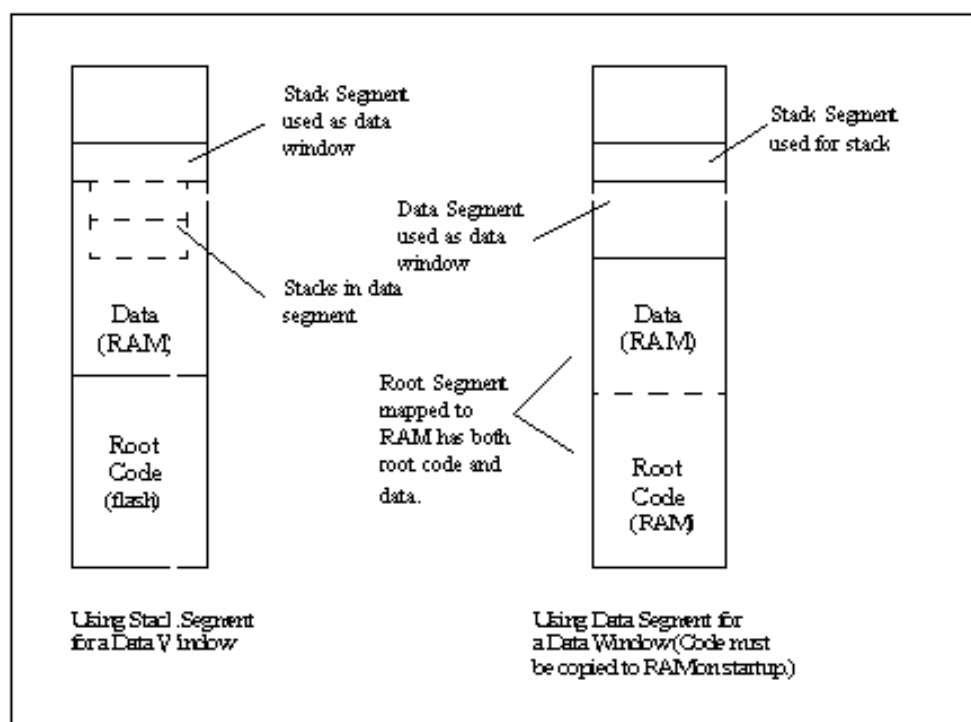


Figura 6. Scheme pentru fereastra memoriei de date



O a treia abordare este să se plaseze data și codul rădăcină în RAM în segmentul rădăcină, eliberând segmentul de date pentru a fi o fereastră pentru memoria extinsă. Pentru aceasta este necesar să se copieze codul rădăcină în RAM în momentul pornirii. Copierea codului rădăcină în RAM nu este neapărat atât de incomodă, deoarece cantitatea de RAM necesară poate fi foarte mică, spre exemplu 12K.

Segmentul XPC din vârful memoriei poate fi de asemenea utilizat ca și segment de date de către programe care sunt compilate în memoria rădăcină. Acest lucru este util pentru programele mici care au nevoie să acceseze multe date.

### 3.4. Considerații practice privind memoria

Cea mai simplă configurație Rabbit folosește un circuit de memorie flash interfațat utilizând /CS0 și un circuit de memorie RAM interfațat utilizând /CS1. Sistemele tipice bazate pe microprocesoare Rabbit utilizează 256K de memorie flash și 128K de memorie RAM, dar pot fi utilizate și memorii mai mici sau mai mari.

Deși Rabbit poate suporta până la 1 MB de cod, se anticipează că majoritatea aplicațiilor vor utiliza mai puțin de 250KB de cod, echivalent cu aproximativ 10 000 - 20 000 de instrucțiuni C. Acest lucru reflectă natura compactă a codului Rabbit și mărimea tipică a aplicațiilor embedded.

Variabilele C accesibile direct sunt limitate la aproximativ 44KB de memorie, împărțit între datele stocate în flash și în RAM. Acest lucru va fi mai mult decât adecvat pentru majoritatea aplicațiilor embedded. Unele aplicații s-ar putea să necesite tablouri mari de date sau tabele care vor necesita memorie de date adițională. În acest scop Dynamic C suportă un tip de memorie de date extinsă care permite utilizarea de memorie de date adițională, extinsă chiar și dincolo de 1MB.

Cerințele pentru memoria de stivă depind de tipul aplicației și, în special, dacă este utilizat multitasking preemptiv. Dacă este utilizat multitasking preemptiv, atunci fiecare task necesită stivă proprie. Deoarece stiva are propriul segment în spațiul de adrese de 16 biți, este simplu de utilizat memoria RAM disponibilă pentru a suporta un număr mare de stive. Când are loc un schimb preemptiv de context, registrul STACKSEG poate fi modificat pentru a mapa segmentul de stivă către porțiunea de memorie RAM care conține stiva asociată noului task care urmează a fi rulat. În mod normal, segmentul de stivă este de 4K, ceea ce este de obicei suficient de mare pentru a furniza spațiu pentru mai multe (de obicei 4) stive. Este posibil să se mărească segmentul de stivă dacă stive mai mari de 4K sunt necesare. Dacă este necesară doar o stivă, atunci este posibil să se elimine complet segmentul de stivă și să se plaseze stiva în segmentul de date. Această opțiune este atractivă pentru sisteme cu doar 32K de memorie RAM care nu au nevoie de mai multe stive.

## 4. Echipamente și dispozitive folosite

Pentru buna desfășurare a lucrării de laborator se vor folosi următoarele dispozitive și resurse software:

- Modul RCM3365	10
- Placă de bază pentru modulul RCM3365	10
- Sursă de alimentare 12V	10
- Cablu serial pentru programarea RCM3365	10
- Compilator Dynamic C	10

## 5. Teme

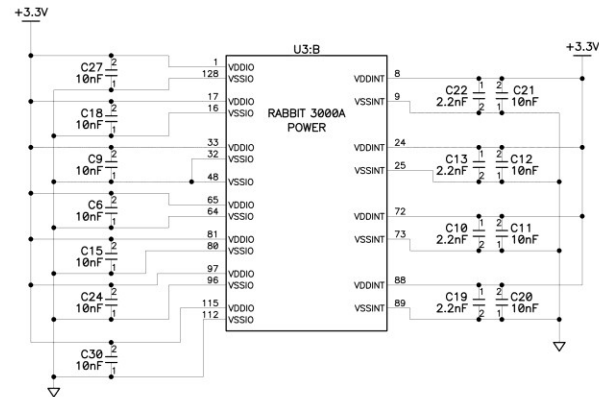
1. Ce spațiu de memorie putem adresa cu un registru de 20 de biți?
2. În câte zone este împărțit spațiul de adrese pe 16 biți, adresabil direct de către microprocesor? Care sunt acestea?
3. Pentru maparea spațiului de adrese pe 16 biți în spațiul de adrese pe 20 biți sunt folosiți 3 regiștri de segment. Care sunt aceștia?
4. Cum se realizează translatarea unei adrese pe 16 biți într-o adresă pe 20 de biți, folosind regiștrii de segment?
5. Arătați în binar cum s-a realizat translatarea adresei 0x7000 (vezi figura 2) la adresa 0x80000. Ce registru de segment s-a folosit?
6. Care este rolul registrului SEGSIZE?
7. Care este rolul registrului DATASEG?
8. Care este rolul registrului STACKSEG?
9. Care este rolul registrului XPC?
10. Identificați circuitele de memorie de pe modulul RCM 3365. Care sunt caracteristicile acestor circuite de memorie?

# 4 STUFFING CHART

PART	RCM3365	RCM3365 32MB	RCM3375
U15	16MB	32MB	NOT INSTALLED
E-NET	ASIX	ASIX	ASIX

PART	SMSC	ASIX
U8	LAN91C113I	NOT INSTALLED
U13	NOT INSTALLED	AX88796L
R1	NOT INSTALLED	470 ohm
R14	NOT INSTALLED	470 ohm
R15	NOT INSTALLED	470 ohm
R26	NOT INSTALLED	470 ohm
R27	NOT INSTALLED	470 ohm
R30	24.9 ohm	NOT INSTALLED
R31	24.9 ohm	NOT INSTALLED
R32	10K	NOT INSTALLED
R33	NOT INSTALLED	2M
R34	0 ohm	470 ohm
R43	NOT INSTALLED	470 ohm
R44	0 ohm	NOT INSTALLED
R47	10K	NOT INSTALLED
R48	10K	NOT INSTALLED
R49	10K	NOT INSTALLED
R51	10K	NOT INSTALLED
R52	10K	NOT INSTALLED
R55	24.9 ohm	NOT INSTALLED
R56	24.9 ohm	NOT INSTALLED
R57	NOT INSTALLED	49.9 ohm
R62	470 ohm	NOT INSTALLED
R63	NOT INSTALLED	49.9 ohm
R75	470 ohm	NOT INSTALLED
R79	NOT INSTALLED	24.9K
R81	NOT INSTALLED	2.49K
R82	NOT INSTALLED	20K
C36	NOT INSTALLED	10uF @10V
C42	NOT INSTALLED	10uF @10V
C58	100nF	NOT INSTALLED
C62	10nF	2.2nF
C63	10nF	NOT INSTALLED
C69	NOT INSTALLED	2.2nF
C70	NOT INSTALLED	10nF
C71	NOT INSTALLED	10nF
C72	NOT INSTALLED	10nF
C73	NOT INSTALLED	100nF
C74	NOT INSTALLED	10nF

PART	SMSC	ASIX
C77	NOT INSTALLED	10nF
C78	NOT INSTALLED	10nF
C79	NOT INSTALLED	100nF
C80	NOT INSTALLED	100nF
C81	NOT INSTALLED	10nF
C82	NOT INSTALLED	10uF @ 10V
C85	NOT INSTALLED	100nF
L1	NOT INSTALLED	10 @ 100MHz
L3	NOT INSTALLED	10 @ 100MHz
DS4	NOT INSTALLED	Green LED



NOTES: UNLESS OTHERWISE SPECIFIED:

1. ALL RESISTOR VALUES ARE IN OHMS, 1/16W, 5%
2. ALL CAPACITORS ARE 16VDC OR HIGHER.
3. THE ORIGIN SOURCE OF A VOLTAGE IS REPRESENTED BY (  $V_{CC}$  ), AND ALL REFERENCES TO THAT VOLTAGE ARE REPRESENTED BY (  $V_{CC}$  ).

4. COMPONENT IN DASHED BOX AND COMPONENT VALUES SHOWN WITH AN ASTERISK (\*) FOLLOWING THE VALUE MAY NOT BE INSTALLED, REFER TO THE STUFFING CHART.

COPYRIGHT 2007, RABBIT SEMICONDUCTOR, INC.

APPEND THE FOLLOWING DOCUMENTS WHEN CHANGING THIS DOCUMENT:		DRAWING CONTENT:		TITLE  RCM3365/3375 RABBITCORE		<div>RABBIT SEMICONDUCTOR</div> <a href="http://www.rabbit.com">www.rabbit.com</a>						
		DRAWN BY: (INITIAL RELEASE)	6/23/2005									
		REVISED BY: XT	05/30/07									
		APPROVALS: INITIAL RELEASE										
		PROJECT ENGINEER:		SIZE  B		DWG NO.  090-0214						
		ENGINEERING MANAGER:										
		SIGNATURES		DATE		SCALE	NONE	RELEASE DATE	SHEET	1	OF	5

