UNIUNEA EUROPEANĂ

GUVERNUL ROMÂNIEI

Instrumente Structurale 2007-2013

Platformă de e-learning și curriculă e-content pentru învățământul superior tehnic

Proiectarea Logică

10. Proiectarea unitatilor de comanda microprogramate

Proiectarea unitatilor de comanda microprogramate comparativ cu metodele clasice

Introducere

In acest ultim capitol vom examina diferite moduri de implementare a sectiunii de control a unui processor. In sistemele reale, unitatea de control este deseori cea mai complexa parte.

Vom studia patru moduri de organizare a controlerelor. Primul este bazat pe *masini de stare finite clasice*, folosind structura Moore sau Mealy. Aceasta abordare este uneori in mod eronat numita control "logic aleator", pentru a sublinia diferenta fata de alte metode mai structurate bazate pe memorii ROM si alte forme de logica programabila. Metoda clasica este singura abordare folosita in capitolul 11.

A doua metoda este numita *time state*. Descompune o masina de stare finita clasica in mai multe masini de stare finite mai simple. Este o strategie pentru partitionarea unei masini de stare finita care se potriveste perfect structurii de controler de procesor.

A treia metoda foloseste contoare cu salt, introduse in capitolul 10. Aceasta abordare foloseste in mod extensiv componente de nivel MSI precum numaratoare, multiplexoare si decodificatoare pentru implementarea controlerului.

Ultima metoda, *microprogramarea,* foloseste memorii ROM pentru a codifica starile urmatoare si semnalele de control in forma de biti stocati in memorie. Vom examina trei metode alternative de microprogramare: microprogramare orizontala, verticala si *branch sequencer.* Aceasta ultima metoda codifica variantele multiple de stari urmatoare in memorii ROM. Microprogramarea pe orizontala asigneaza cate o memorie ROM fiecarei iesiri a controlerului. Microprogramarea pe verticala codifica atent iesirile controlerului pentru a reduce dimensiunea cuvintelor din memorie. Orice abordare practica a microprogramarii combina aceste trei variante.

Cuprins

12.1.Logica aleatoare

In acest subcapitol vom examina structuri clasice de control bazate pe metode standard de implementare a masinilor Moore si Mealy. Aceasta organizare a controlerului este uneori numita *logica aleatoare(random logoc)* , pentru ca starea urmatoare si iesirea sunt formulate folosind porti logice discrete. Se poate folosi la fel de usor logica programabila, precum PAL/PLA, EPDL, FPGA sau ROM pentru a implementa aceste functii.

In acest subcapitol vom examina doua metode alternative de implementare a controlerului pentru setul de instructiuni si traseul datelor introduse in ultimul capitol. Deoarece am studiat implementarea pentru automatul Mealy in capitolul 11, vom aborda automatul Moore in urmatorul subcapitol.

O masina Mealy e deseori cea mai economica cale de a implementa masina de control al starilor, dar iesirile asincrone introduc probleme de sincronizare. Ne vom uita la diferentele dintre masinile Mealy sincrone si asincrone si la relatia de sincronizare intre asertiunile semnalelor si efectul lor asupra traseului datelor.

12.1.1.Automatul Moore

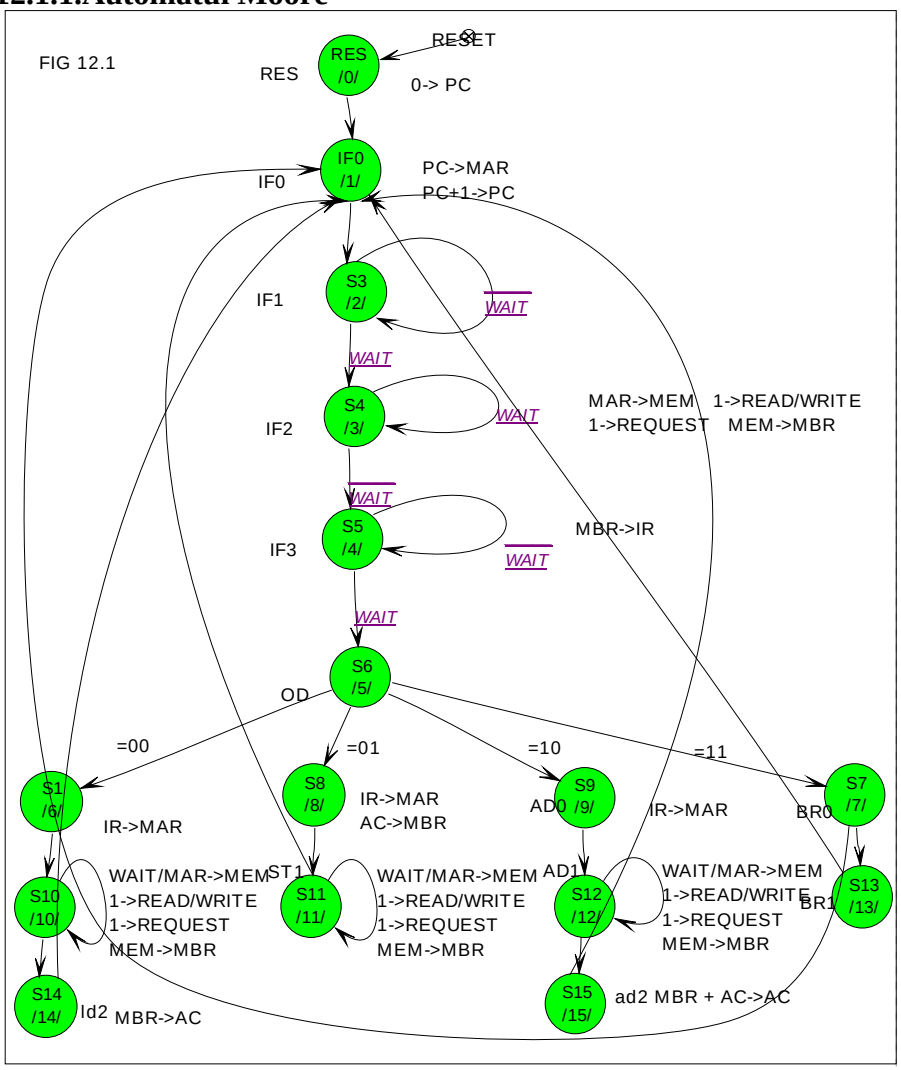


Figura 12.1 reda diagrama completa a starilor, inclusive operatiile de transfer intre registri, pentru o implementare a masinii Moore din sectiunea 11.3 .Necesita mai multe stari decat diagrama Mealy echivalenta dar diferentele sunt minore. In particular, avem nevoie de o stare suplimentara in secventa de reset/fetch si de o alta in secventa negata a ramurii.

Atribuirea operatiilor de transfer dintre registrele starilor nu este fortata. Doar o singura combinatie de operatii de transfer de registre in aceeasi stare poaate fi intrigant. De aceea

cererea de citire din memorie este folosita in acelasi timp ca si trecerea prin latch a magistralei de date in MBR(vedeti starile IF2, LD1 si AD1). Are acest lucru drept rezultat trecerea prin latch a unor date invalide?
Nu.

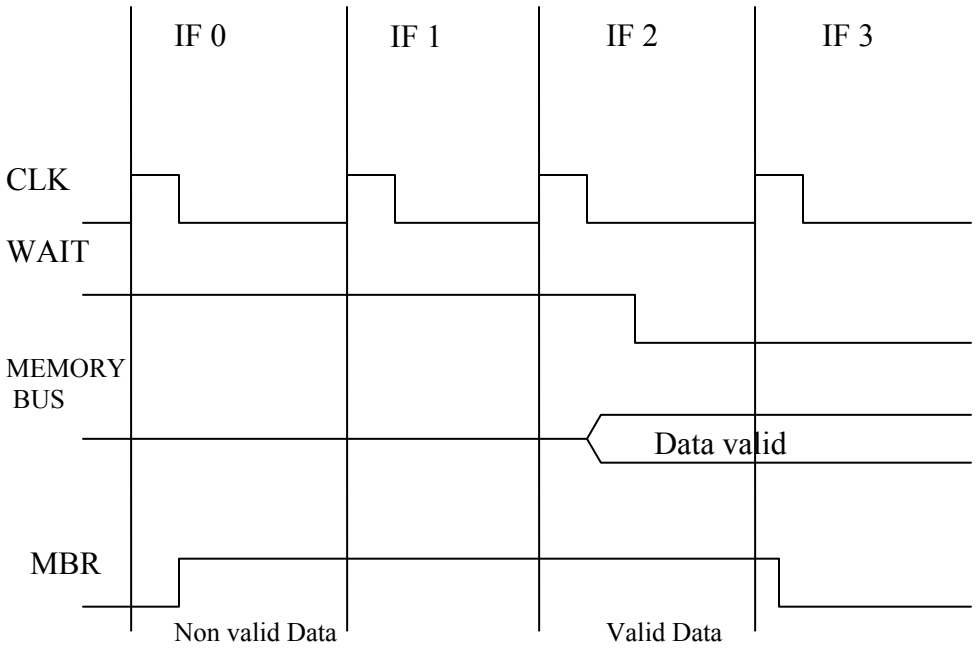
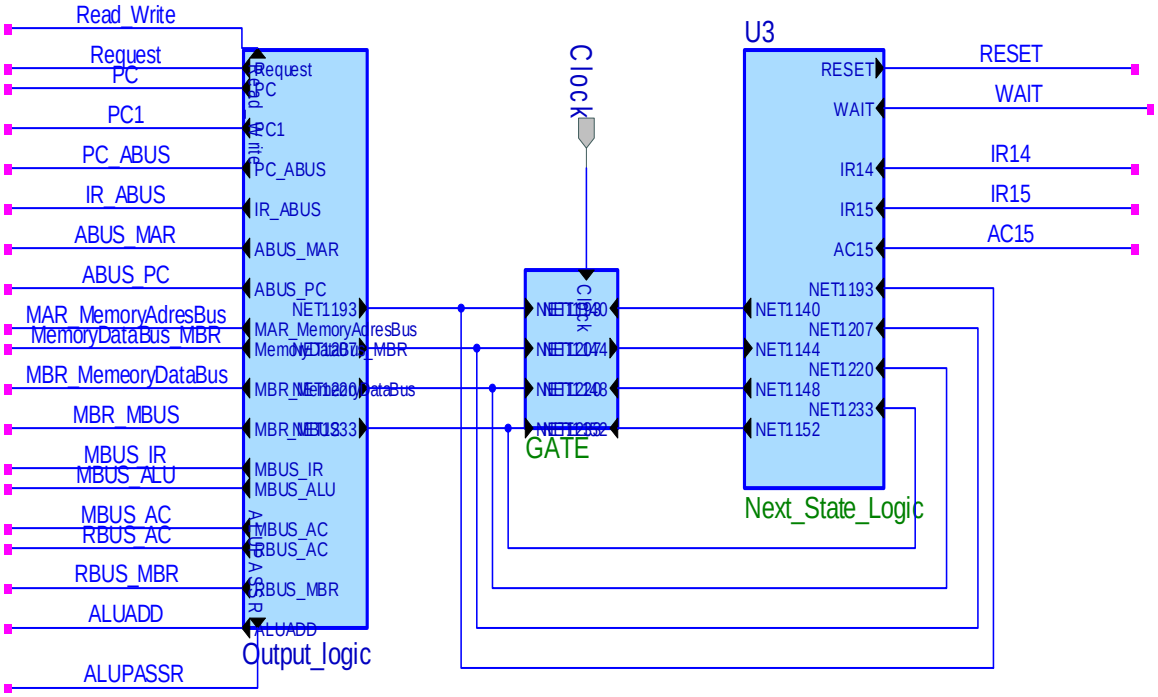


Figura 12.2. arata evenimentele detaliate in timp pentru secventa de stari IF1, IF2 si IF3.De fiecare data cand buclam intr-o astfel de stare, MBR capteaza valoarea curenta a magistralei de memorie. Primele dati cand buclam intr-o stare, data primita de MBR este invalida. Totusi, semnalul Wait ramane neschimbat pana cand din memorie se pun date valide pe magistrala. Cand Wait se schimba, valoarea trecuta prin latch in MBR la urmatoarea perioada de ceas este valida. Aceasta este aceeasi tanzitie de stare care face ca masina de stare sa treaca in starea sa urmatoare(IF3, LD2 sau AD2).

Diagrama bloc a masinii Moore.



Are nevoie de 16 stari.Vom codifica aceste stari condensate intr-un registru de stare pe 4 biti. Logica starii urmatoare are 9 intrari(4 biti pentru starea curenta, Reset, doi biti IR, un bit AC) si patru iesiri (pentru starea urmatoare). Deoarece semnalele de control pentru date sunt decodificate din aceasta stare, blocul logicii are 4 intrari si 18 iesiri.

Variante de implementare. ROM contra PAL/PLA.Putem implementa blocul de control al logicii starii urmatoare fie cu ROM fie cu PAL/PLA. Folosind memorii ROM putem implementa logica starii urmatoare cu o memorie 512*4 biti si logica de iesire cu o memorie 16*18 biti. Deoarece memoriile stand-alone ROM au cuvinte multiplu de 2, o vom implement pe ultima fie cu 4 fie cu 8 biti/cuvant.

Incepem implementarea cotrolerului obtinand tabela simbolica a starilor urmatoare. Aceasta este prezentata in figura 12.4.. Merita facute cateva observatii. In primul rand putem folosi extensiv simbolurile *don't care* printre liniile de adresa/intrare. Observati ca un semnal de intrare dat este examinat in foarte putine stari. De exemplu, bitii IR sunt examinati in starea OD si semnul bitului AD este testat doar in starea BR0.

In al doilea rand, numarul operatiilor de transfer cu registre incluse in orice stare data este destul de mic. In figura 12.4. doar 4 operatii sunt incluse in oricare stare. Cateva iesiri, precum acelea asociate referintelor de memorie sunt mereu incluse impreuna. Vom folosi acest lucru in unele strategii de implementare a controlerelor mai tarziu in acest capitol.

Desigur, o implementare bazata pe ROM nu poate beneficia de avantajul *don't care*. Trebuiesc programata toate cele 512 cuvinte ale memoriei asociate logicii starii

urmatoare, o treaba destul de migaloasa. Insa, un avantaj al folosirii ROM este ca nu trebuie sa te preocupi de o atribuire atenta a semnalelor.

Daca se foloseste PAL/PLA, atunci este esentiala o buna atribuire a starilor pentru a reduce complexitatea logicii starii urmatoare. De exemplu atribuirea simplista sugerata in figura 12.4 presupune 21 de productii.

Reset	Wait	IR<15>	IR<14>	AC<15>	Current State	Nest State	RT operations
1	X	X	X	X	X	RES(0000)	0->PC
0	X	X	X	X	IF1(0001)	IF1(0001)	PC->MAR PC+1->PC
0	0	X	X	X	IF1(0010)	IF1(0010)	
0	1	X	X	X	IF1(0010)	IF2(0011)	
0	1	X	X	X	IF1(0011)	IF2(0011)	MAR->MemRead
0	0	X	X	X	IF1(0011)	IF3(0100)	
0	0	X	X	X	IF1(0100)	IF(0100)	MBR->IR
0	1	X	X	X	IF1(0100)	OD(0101)	
0	X	0	0	X	IF1(0101)	LD=(0110)	
0	X	0	1	X	IF1(0101)	STO(1001)	
0	X	1	0	X	IF1(0101)	AD0(1011)	
0	X	1	1	X	IF1(0101)	BR0(1110)	
0	X	X	X	X	IF1(0110)	LD1(0111)	
0	1	X	X	X	IF1(0011)	LD2(1000)	
0	0	X	X	X	IF1(0011)	LD1(0111)	
0	X	X	X	X	IF1(1000)	IF0(0001)	MBR->AC
0	X	X	X	X	IF1(10001	ST1(1010)	IR->MAR
0	1	X	X	X	IF1(1010)	ST1(1010)	
0	0	X	X	X	IF1(1010)	IF0(0001)	
0	X	X	X	X	IF1(1011)	AD1(1100)	IR->MAR
0	1	X	X	X	IF1(1100)	AD1(1100)	MAR->MemRead
0	0	X	X	X	IF1(1100)	AD21101()	
0	X	X	X	X	IF1(1101)	IP0(0001)	
0	0	X	X	0	IF1(1110	IP0(0001)	
0	1	X	X	1	IF1(1110)	BR1(1111)	
0	X	X	X	X	IF1(1111)	IF0(0001)	

Aceasta este acceptabila fata de 512 productii in cazul ROM(un termen pentru fiecare cuvant).

```
.i 9
.o 4
.ilb reset wait ir15 ir14 ac15 q3 q2 q1 q0
.ob p3 p2 p1 p0
.p 26
1----- ---- 0000
0---- 0001 0001
00--- 0010 0010
01--- 0010 0011
01--- 0011 0011
00    0011 0100
00--- 0100 0100
01--- 0100 0101
0-00- 0101 0110
0-01- 0101 1001
0-10- 0101 1011
0-11- 0101 1110
U---- U110 U111
01--- 0111 0111
00--- 0111 1000
0---- 1000 0001
0---- 1001 1010
01--- 1010 1010
00--- 1010 0001
0---- 1011 1100
U1--- 1100 1100
00--- 1100 1101
0---- 1101 0001
U---0 1110 UUU1
0   1 1110 1111
0---- 1111 0001
.e
```

(a) *Espresso* input

```
.i 9
.o 4
.ilb reset wait ir15 ir14 ac15 q3 q2 q1 q0
.ob p3 p2 p1 p0
.p 21
0-00-0101 0110
0-01-0101 1001
0-11-0101 1110
0-10-0101 1011
01---1010 1010
00---0111 1000
00----011 0100
0----1000 0001
0---1110 1110
01---011- 0100
0----0001 0001
01---01-0 0001
U----1001 1010
0----1011 1100
00---1--0 0001
U----1100 1100
0     0  10 0010
0-----110 0001
U----11-1 UUU1
0     01 0 0100
01---0-1- 0011
.e
```

(b) *Espresso* output

Figure 12.5 *Espresso* input and output for Mano processor control

Figura 12.5 arata intrarile si iesirile *espresso* pentru aceasta atribuire de stari particulare. Logica starii urmatoare este mult mai complexa. Fiecare bit al starii urmatoare cere intre 7 si 9 productii pentru implementare. Aceasta insemna ca ar trebui folosite componente PAL cu plaje de intrare ale portilor OR mari (fan-in), precum P22V10. Pentru o implementare bazata pe PLA, tot ce este necesar este o placa PLA ce furnizeaza 21 de termeni de productii unici.

O atribuire de stare de tip *nova* poate avea rezultate imbunatatite. Are nevoie de 18 termeni:

```
state IF0: 0000                    state ST0: 0101

state IF1: 1011                    state ST1: 0110

state IF2: 1111                    state AD0: 0111

state IF3: 1101                    state AD1: 1000

state  OD:  0001                   state AD2: 1001

state LD0: 0010                    state BR0: 1010
state LD1: 0011                    state BR1: 1100
state LD2: 0100                    state RES: 1110
```

12.1.2. Masina sincrona Mealy

Organizarea unei masini Mealy sincrone nu este foarte diferita de cea a masinii More tocmai descrisa. Chiea este unirea functiilor booleene de iesire si stare urmatoare intr-un singur bloc logic. Pentru masina Mealy, logica are 9 intrari si 22 iesiri(4 iesiri de stare si 18 intrari de control al mocrooperatiilor).

Functiile combinate de iesire si stare urmatoare au cateva aplicatii interesannte pentru o implementare bazata pe ROM. Masina Moore cerea doar 2336 biti ROM pentru implementarea ei (512*4 + 16*8), in timp ce masina Mealy are nevoie de 12.264 biti (512 * 22). Acest calcul arata cateva discrepante privind eficienta implementarii cu ROM. Multi dintre acesti biti sunt simboluri *don't care.* Desigur, memoriile ROM sunt foarte dense si o memorie mai mare este cu putin mai scumpa. Vom vedea alte metode de folosire ale memoriilor ROM in discutia despre microprogramare.

Masina Mealy sincrona versus asincrona
O masina Mealy conventionala este asincrona. Canalele de intrare duc la schimbari ale iesirilor, independent de ceas. Acest lucru poate cauza dezordine cand semnalele de iesire reprezinta controlul imediat al datelor. Trebuie sa putem sa integram semnalele de control intr-un mediu propice, intr-o maniera sincrona.

Pana la un anumit punct putem minimize riscul semnalelor de control asincron selectand componente din cadrul traseului datelor cu controale sincrone. Aceste intrari nu trebuie sa se stabilizeze decat cu un timp setat inainte de schimbarea frontului ceasului de control.

Insa exista inca o problema deoarece semnalele de control isi fac simtit efectul imediat. Cel mai sigur remediu este sa sa tranformam masina Mealy intr-una sincrona. Intr-o astfel de masina iesirile se schimba doar cand starile se schimba si raman stabile pe parcursul starii. Putem realize acest lucru punand registri intre semnalele de intrare, logica combinationala care calculeaza iesirile si semnalele de iesire. Sa examinam abordarile in cazul contruirii unei masini Mealy sincrone in cele ce urmeaza.

Sicronizarea unei masini Mealy

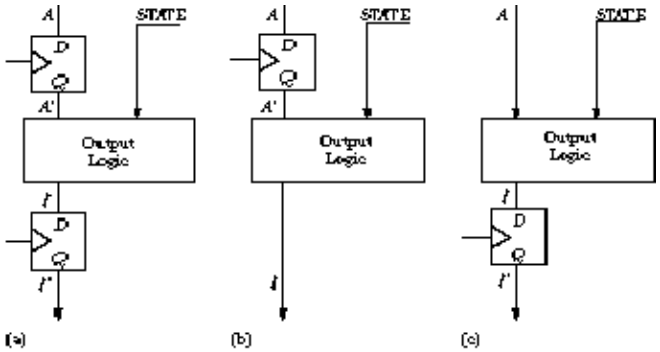
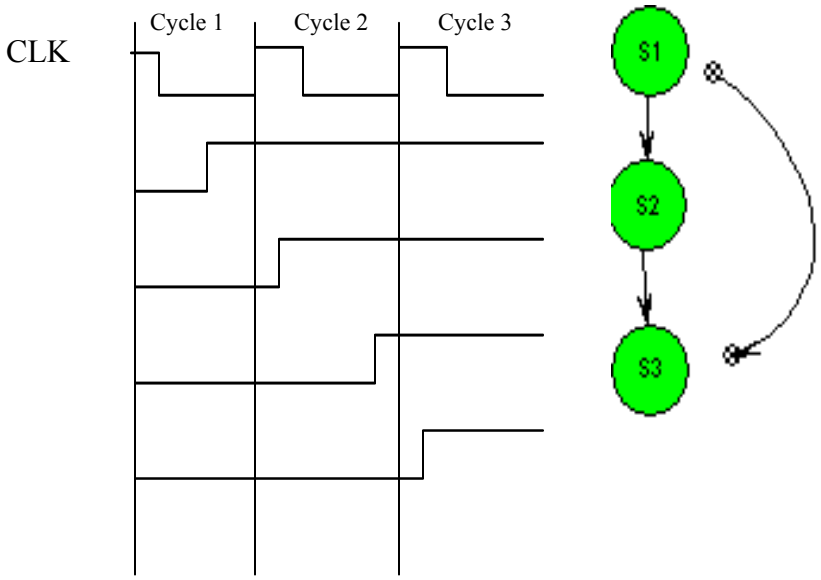


Figure 12.6 Three approaches to building a Mealy machine with synchronous outputs.

Figura 12.6 arata cele 3 moduri posibile de realizare a unei masini Mealy cu iesiri sincronizate: folosind dispozitive edge-trigger la intrari si iesiri (a), doar la intrari(b), doar

la iesiri(c). Fiecare afecteaza temporizarea semnalelor de control in moduri usor diferite. In figura presupunem ca iesirile ar trebui exprimate doar cand intrarea A este exprimata.

Sa incepem cu cazul (a), care sincronizeaza atat intrarea cat si iesirea. Presupunand ca A este valid in ciclul 0, iesirea sincronizata nu va fi valida pana in ciclul 2. Aceasta intarzie calcularea cu 2 cicluri. Deci, daca A este validat in starea S0, iesirea nu este valida decat din starea S2.



Trebuie sa realizati ca punand registre de sincronizare atat la intrari cat si la iesiri este teribil. Putem obtine sincronizarea dorita prin dispozitive flip-flop de o parte sau alta a logicii de iesire. Sa consideram cazul (b):

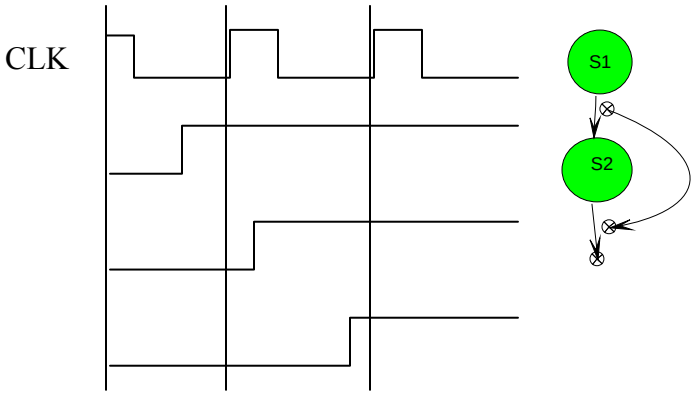
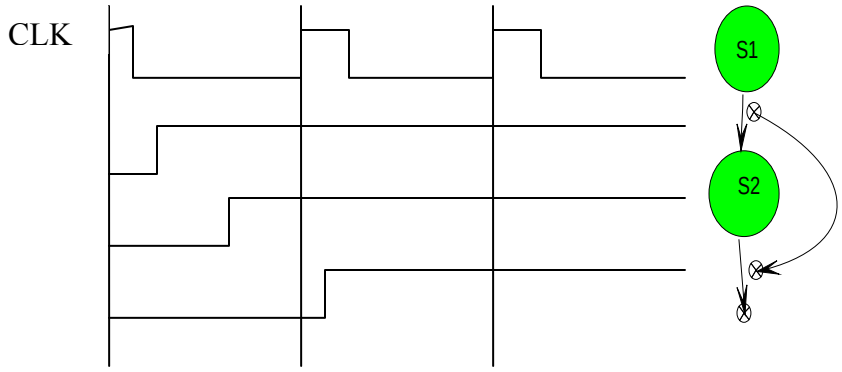


Figura 12.8 arata efectele. Daca A este validat in ciclul 0, iesirea este validata in ciclul urmator.

Cazul (c) plaseaza logica de sincronizare doar la iesire.



Semnalul sincronizat de iesire isi face efectul in starea ulterioara celei in care A este validat prima oara.

Sincronizarea diagramei de stare a masinii CPU simple Mealy
Pentru a concretize aceste idei, sa examinam implementarea Mealy a masinii de stare a controlului procesorului discutat in sectiunea 11.3. Cazul (b) care punea registri la intrare este cel optim pentru a sincroniza aceasta masina. Din cele 5 intrari, IR<15:14> si AC<15> sunt deja sincronizate deoarece ele sunt registrele de date ce au ca ceas acelasi semnal ca si masina de control a starilor. Efectul intarziat al semnalelor de control nu se aplica aici;nu punem un registru aditional pe traseul dintre IR si AC si control.
Reset si Wait nu conteaza. Deoarece aceste semnale vin din exteriorul procesorului, este indicat sa le trecem prin dispozitivele de sicronizare flip-flop. Asta insemna ca semnalele externe Reset si Wait sunt decalate cu o stare de ceas inainte sa poata influenta masina de stare.

Intarzierea resetului cu o stare nu are efect major; masina de stare se va reserta oricum. Insa intarzierea cu o stare a semnalului Wait afecteaza performanta. Procesorul bucleaza in mod normal intr-o stare pana apare o schimbare a semnalului Wait. Asta inseamna ca masina sta in bucla inca un ciclu. Chiar si o memorie ce raspunde unei cerei imediat are nevoie de o stare de ceas a procesorului inainte ca acesta sa recunoasca operatia ca fiind acceptata.

Daca proiectam sistemul de memories sa fie sincronizat cu procesorul, putem evita pierderea de performanta. Deoarece controlerul memoriei sistemului este conectat la acelasi semnal de ceas ca si procesorul, semnalul Wait nu mai trebuie sincronizat.

12.2.Time state(Divide and Conquer)

Abordarea clasica pentru implementarea masinilor finite de stare se invarte in jurul unei implementari monolitice care este uneori greu de schimbat. O abordare alternativa, bazata pe o strategie divide&conquer, imparte aceasta masina in mai multe submasini care comunica intre ele.

12.2.1.Impartirea masinii de stare

O impartire obisnuita este sa se imparta controlerul in trei masini:de timp, de instructiuni si de conditii. Prima determina faza curenta a interpretarii instructiunii. Include fetch-ul instructiunilor, decodificarea si executia lor.

A doua identifica instructiunea curenta care este executata, precum load, store, add sau branch. Se ocupa de intregul proces de decodificare a instructiunii.

A treia reprezinta starea conditiei curente a caii de date. In exemplul nostru cu procesorul, singura conditie interesanta este bitul cel mai semnificativ al AC. Alte conditii posibile se refera la ultima operatie ALU: rezultat egal cu zero, overflow, underflow.

Impartirea in aceste trei feluri de masini este avantajoasa deoarece de obicei sunt diferente minore intre modurile de control pentru diferite instructiuni. Daca aceste secvente sunt parametrizate adecvat, putem evita o secventa unica pentru fiecare instructiune. Daca instructiunea poate fi usor decodata pornind de la IR si conditiile privitor calea de date, putem reduce numarul starilor in masina finita generica.

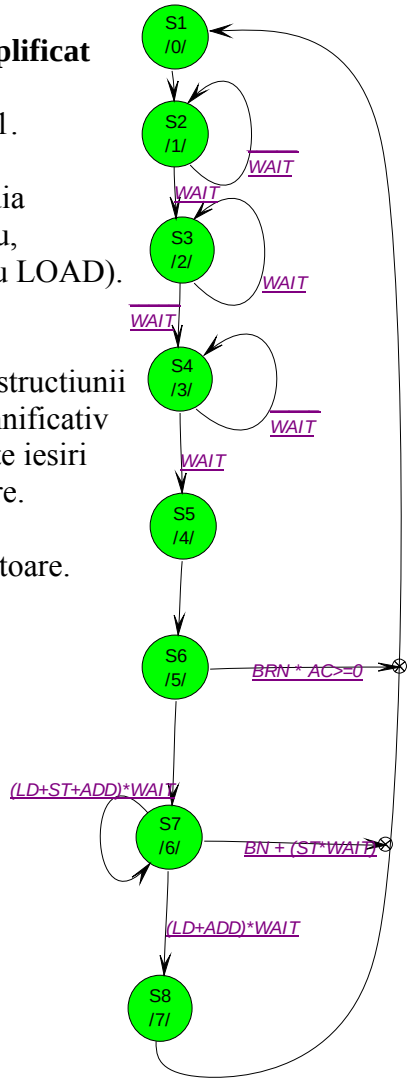
12.2.2.Masini time state pentru procesorul exemplificat

Incepem cu diagrama de stare Moore din figura 12.1.
Pentru a obtine masina finita de stare time state, trebuie sa cautam calea cea mai defavorabila prin dia grama clasica. In cazul procesorului luat ca exemplu, calea cea mai defavorabila este de 8 stari (ADD sau LOAD).

Fiind date aceasta secventa de 8 stari, ideea este sa parametrizam secventa de baza prin iesirile starii instructiunii (LD, ST, ADD, BRN) si starii conditiilor(bitul semnificativ al AC: AC 0, AC < 0) masinii finite de stare. Aceste iesiri sunt asociate cu tranzitiile din cadrul masinii de stare.

In conditii potrivite, masina va trece in starea urmatoare.

Diagrama de stare parametrizata a masinii este:



Starea instructiunii si a conditiei masinii sunt date in figura. Lasam Reset-ul in seama altei masini care nu apare aici. Fiecare instructiune, indiferent de tipul ei, cicleaza prin primele 5 stari, IF0 pana la IF3 si OD. Acestea sunt atribuite starilor T0->T4.

Dupa aceasta, instructiunile urmeaza cai de executie diferite. Din fericire, o mare parte din trasee sunt comune. Starile LD0-LD2, ST0, ST1, AD0-AD2 si BR0, BR1 au fost inglobate in starile T5, T6, si T7. Singura iesire a masinii este starea curenta, T0->T7.

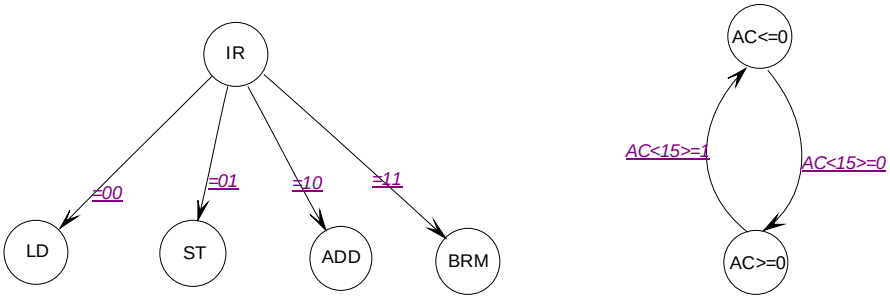


Figura 12.10 arata cum iesirile starii instructiune si conditie influenteaza secventa starii urmatoare in cazul masinii de stare *time state.* De exemplu in starea T5, daca instructiunile sunt BRN si AC0, masina se intoarce in starea T0. Altfel avanseaza in starea T6. Aceasta intoarcere brusca la inceputul diagramei se cheama tranzitie de *scurt circuit.*Desi am putea sa fortam toate instructiunile sa treaca prin T7 inainte sa se intoarca, ar fi afectata eficienta deoarece fiecare instructiune ar necesita tot atatea cicluri cat cea mai lunga instructiune.

Ca un exemplu suplimentar, sa consideram starea T6. Daca instructiunile sunt LD, ST, sau ADD si Wait este valid, masina ramane in T6. Daca instructiunile sunt BRN sau ST cu Wait invalid, masina se intoarce inT0. Altfel avanseaza inT7.

Generarea Microoperatiilor. Nu este dificil sa se genereze mocrooperatiile pentru iesirile functie de timp, conditii si instructiuni. Operatiile de transfer cu registri sunt descrise mai jos:

0 --> PC: Reset
PC + 1 --> PC: T0
PC --> MAR: T0
MAR --> Memory Address Bus: T2 + T6 (LD + ST + ADD)
Memory Data Bus --> MBR: T2 + T6 (LD + ADD)
MBR --> Memory Data Bus: T6 ST
MBR --> IR: T4
MBR --> AC: T7 LD
AC --> MBR: T5 ST
AC + MBR --> AC: T7 ADD
IR<13:0> --> MAR: T5 (LD + ST + ADD)
IR<13:0> --> PC: T6 BRN

1 --> Read/Write: T2 + T6 (LD + ADD)
0 --> Read/Write: T6 ST
1 --> Request: T2 + T6 (LD + ST + ADD)

Starea conditiei nu este folosita explicit pentru generarea unei operatii de transfer cu registrii. Desigur, influenteaza tranzitia catre starea urmatoare.

Discutii. In general, abordarea de tipul *time state* poate reduce numarul starilor si sa simplifice logica starii urmatoare, singurul cost suplimentar fiind introducerea de elemente flip-flop. Tehnica scurtcircuitului face logica de tranzitie catre starea urmatoare sa fie complexa dar da voie instructiunilor de contorizare de scurtcircuite sa se termine mai devreme Acest aspect duce la executii mai rapide ale programelor.

12.3.Contoare de salt-*jump counters*

In sectiunea 10.2 am descris metodele de implementare ale masinii finite de stare folosind un contor pe post de registru de stare. Aceasta metoda porta denumirea de *jump counter*. Abordarea foloseste componente MSI precum numaratoare sincrone, multiplexoare si decodificatoare pentru implementarea masinii. In acest subcapitol vom aprofunda aceasta descriere , aratand cum numaratoarele de salt pot fi folosite la implementarea UC al unei CPU.

Numaratoarele de salt se impart in doua categorii: *pure* si *hibride.* Un numerator pur permite doar una din urmatoarele stari: starea curenta(numaratorul asteapta), urmatoarea stare din secventa(numaratorul numara), starea 0(numaratorul se reseteaza) sau o singura stare de salt(numaratorul incarca).In cazul numaratoarelor pure, starea de salt este strict o functie a starii curente. Un numarator hibrid suporta aceleasi tipuri de tranzitii dar permite starii de salt sa fie o functie a intrarii cat si a starii curente.

12.3.1.Numaratoare de salt pure

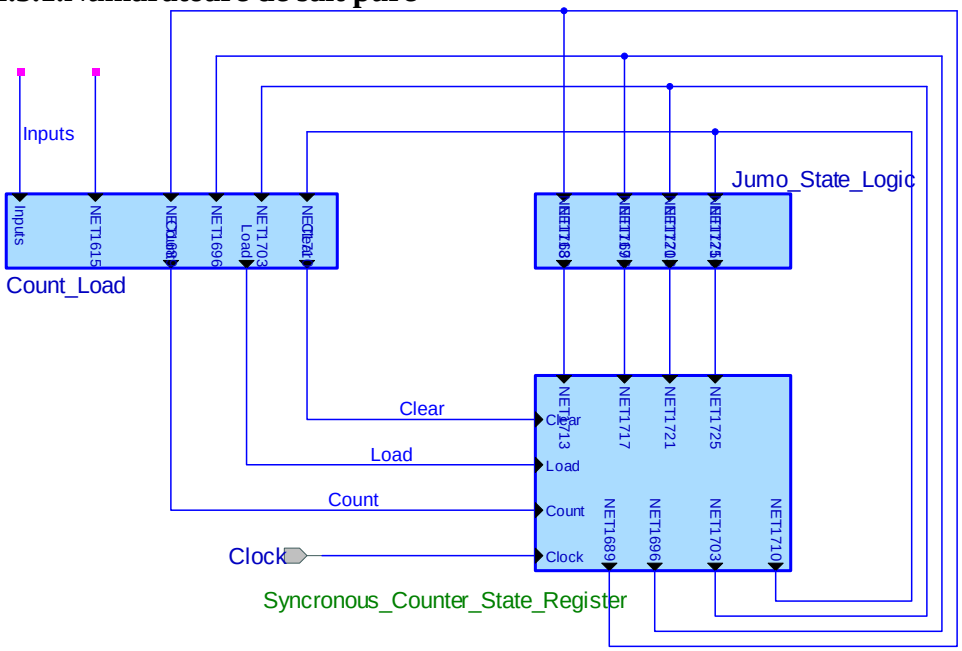
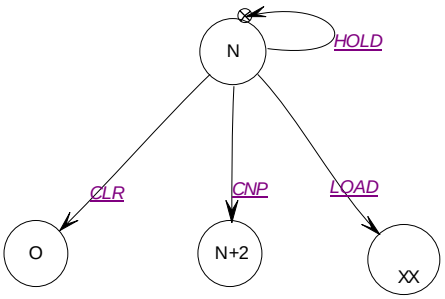


Figura 12.12. este diagrama bloc a unui astfel de numarator. Starea de salt este o functie a starii curente, in timp ce intrarile clear, inpur si count in registrul de stare depind doar de starea curenta si de intrarile curente. Presupunem ca semnalul clear este prioritar fata de load, care este prioriar fata de count. Blocurile logice din figura pot fi implementate cu porti logice discrete, PAL/PLA sau ROM. Se folosesc frecvent memorii ROM pentru logica de salt.

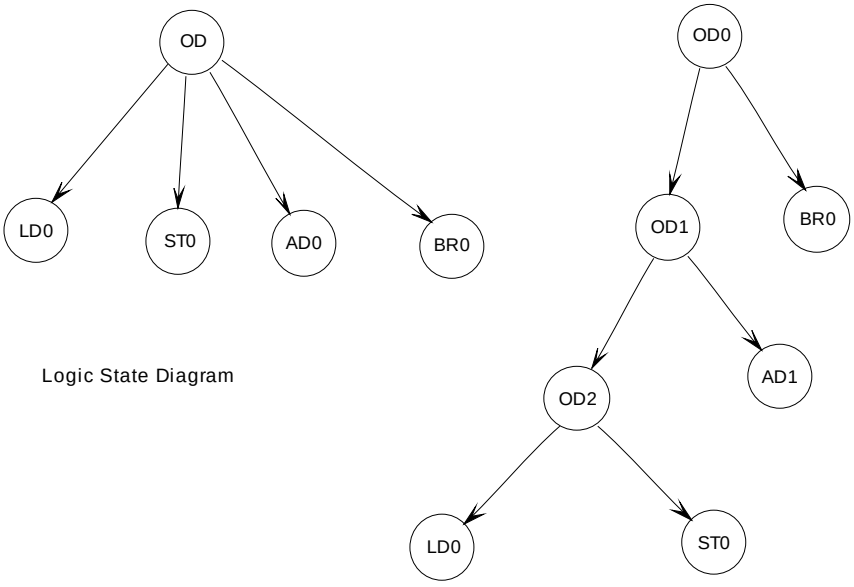
Secventa starilor este prezentata in varianta restictiva in fig. 12.13.
Pentru a maximiza avantajul folosirii registrului de stare, vom atribui starile in ordinea numararii. Cea mai recventa tinta a unei tranzitii ar trebui sa fie atribuita starii 0.



Acest lucru se dovedeste insa prea restrictiv pentru stari ce necesita o ramificare multinivel mai generala, precum starea operatiei de decodare din diagrama de stare a unui procesor. Pentru stari ramificate multinivel, o abordare tip numarator de salt pur trebuie sa introduca un numar suplimentar de stari.

Figura 12.14 (a) arata fragmentul diagramei de stare a operatiei de decodare din figurile 11.23 si 12.1. Pentru implementarea drept numarator de salt pur, vom avea nevoie de

diagrama din fig 12.14 (b).Trebuie sa introducem 2 noi stari decodificatoare si sa crestem numarul total de stari necesare excutiei pentru load, store sau add. Nu este de mirare ca s-a trecut la inventarea numaratorului hibrid.



Logic State Diagram

State Diagrama and asignment implied by pure jump counter

12.3.2.Numaratorul de salt hibrid

Rezolva problema ramurilor multinivel. Transforma starea de jump intr-o functie apartinand intrarilor dar si starii curente. Detalierea este aratata in figura 12.15. Starea de jump, logica pentru load, store si numarare sunt toate functii depinzand de intrari si starea curenta.

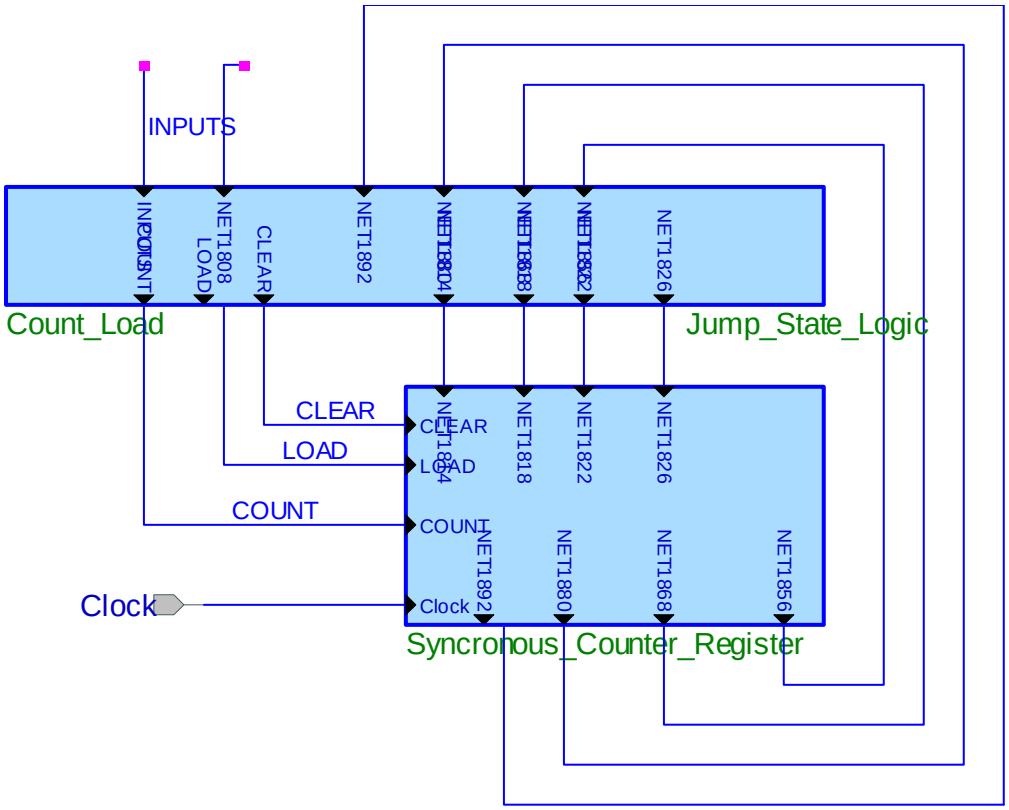
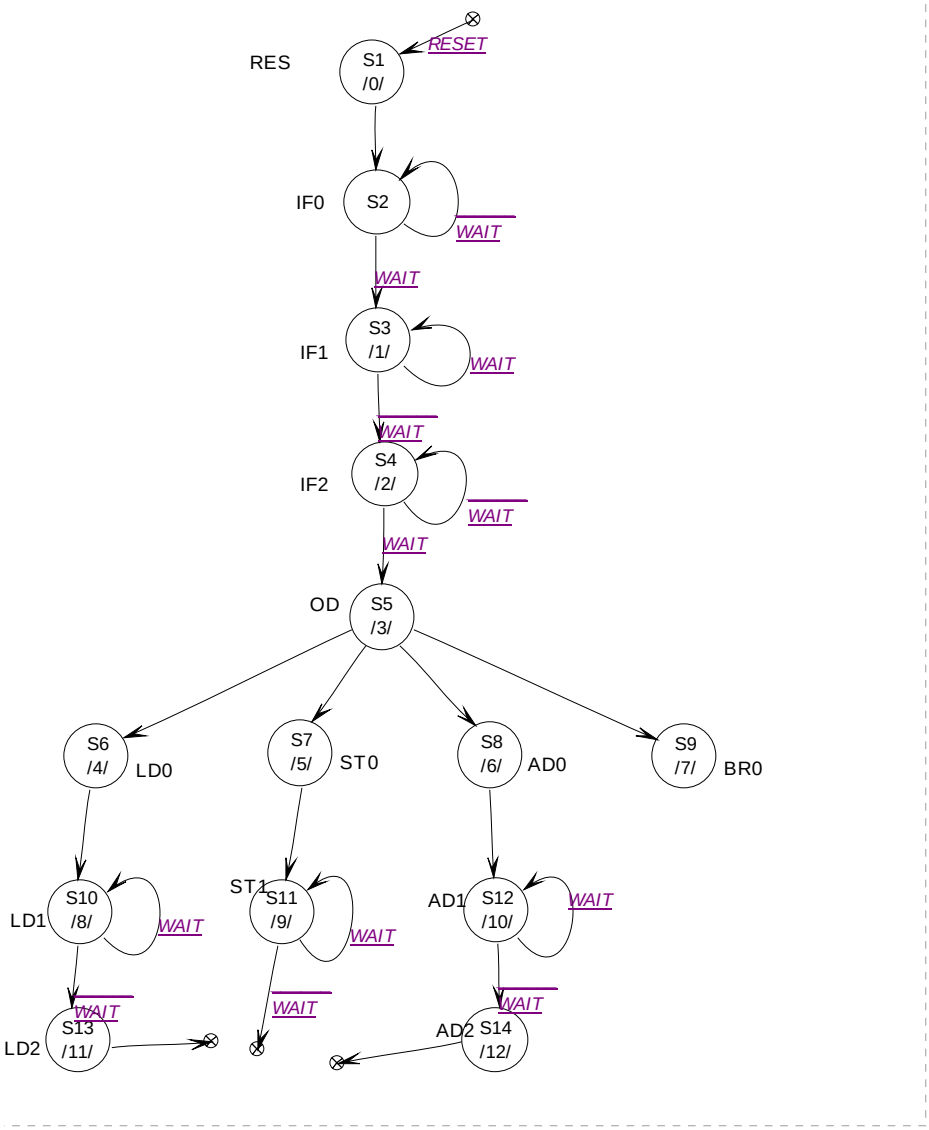


Diagrama de stare Mealy pentru CPU simpla.
Vom incerca sa implementam diagrama de stari –Mealy din fig 11.23 cu numaratoare de salt hibride. Prima problema este cum sa atribuim codificarile diagramei.Majoritatea tranzitiilor de stare avanseaza sau stau in starea curenta, depinzand de valoarea semnalului Wait. Deci putem folosi o atribuire de stare naturala depth-firsth. Vedeti figura 12.16.

Apoi vom gasi o stare sa o atribuim valorii 0. Deoarece starea RES este cea mai frecventa destinatie a unei tranzitii, este cel mai potrivit candidat. Incepand cu 0, atribuim stari in secventa pe masura ce avansam in jos prin diagrama de stari.

Ultima preocupare este sa identificam ramificatiile de stare, *branch states,* ale caror tranzitii urmatoare nu pot fi descries simplu intermeni de hold(ramai instarea curenta), count(avanseaza in urmatoarea stare din secventa) sau reset(du-te in starea 0). Singura strare de acest gen este OD, operation decode. Din fericire, putem descrie tranzitia catre starea urmatoare drept o functie a starii curente si a bitilor codului operatiei IR. De fapt, deoarece aceasta este singura ramura multinivel in intreaga diagrama, logica starii de salt este o functie ce depinde in intregime de bitii codului operatiei IR.

Atribuirea completa a starilor e este prezentata in figura 12.16. Presupunem ca starile sunt notate de la S0 la S13. Pentru tranzitii in starea RES (S0), semnalul CLR al contorului de stare ar trebuie validat in starile S7, S9(daca Wait negat este valid), S12 si S13.



Semnalul numaratorului LD este validat iar pentru starile cu ramificare multinivel. Pentru aceasta dagrama de stare, ar trebui validat in starile S4, starea OD. Logica starii de jump determinata de bitii codului operatiei IR genereaza ca noua stare sa fie incarcata in numaratorul de stare.

Cand sa validam semnalul CNT este o chestiune mai complicata. Ecuatiile booleene pentru count /hold(don’t hold) sunt:

CNT = (S0 + S5 + S10) + Wait*($\overline{S2}$+ S6 + S9 + S11)

HOLD = $\overline{Wait}$ *(S2 + S1) + Wait*(S2 + S6 + S9 + S11)

Ecuatia pentru HOLD este mai simpla decat cea pentru CNT. Se poate economisi efort de implementare a logicii prin obtinerea semnalului CNT din semnalul HOLD
Logica starii de salt pentru sta rea S4 (OD) este simpla. Poate fi implementata fie printr-o memorie 4 cuvinte*4 biti ale carei intrari de adresa sunt cele doua coduri de biti de operatie IR.

Implementare schematica pentru numaratorul cu salt.

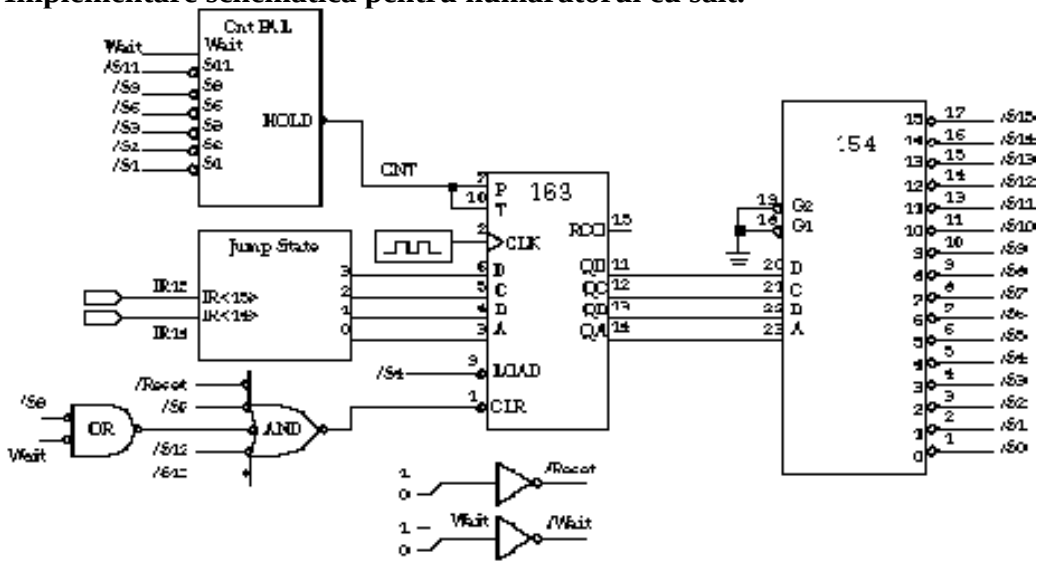


Figure 12.17 Schematic description of jump-counter implementation.

Componentele majore sunt (1)numarator sincron 74163 folosit ca registru de stare, (2)decodificator 4-16 74154 ce genereaza semnale pentru identificarea starii curente, (3)logica starii de salt implementata cu o memorie ROM 4*4, indexata de codurile operatiei celor 2 IR, (4) PAL ce implementeaza logica booleana pentru intrarea CNT a numaratorului si (5) logica discreta ce implementeaza semnalul de intrare CLR al numaratorului de stare. Deoarece LD este deja validat doar intr-o stare, S4, iesirea decodorului pentru acea stare conduce direct intrarea load.

Sa ne uitam in amanunt la logica pentru controler bazat pe numarator de salt hibrid. Pentru fiecare model de cod de operatie, bitii potriviti ai starii urmatoare sunt stocati in ROM.

Address	Contents (Symbolic State)
00	**0101 (LD0)**
01	**1000 (ST0)**
10	**1010 (AD0)**
11	**1101 (BR0)**

Restul proiectarii numaratorului este lesne de dedus, dar apare o singura problema. Polaritatea negativa a semnalelor de control LD si CLR ale numaratorului 74163, ca si iesirile low active ale decodificatorului 74154. In logica pozitiva, semnalul CLR poate fi exprimat

$$CLR=Reset+S7+S12+S13+(S0+\overline{Wait})$$

Deoarece intrarea CLR este activa low, functia ar trebui rescrisa folosind legile De Morgan:

$$CLR=\overline{Reset}+\overline{S7}+\overline{S12}+\overline{S13}+(\overline{S0}+Wait)$$

Din fericire, decodificatorul furnizaza axact aceste semnale active low.O poarta AND cu 5 intrari si una OR cu 2 intrari implementeaza semnalul.

Uneori este mai convenabil sa implementam HOLD (DON'T CNT) decat CNT. Complementul lui Hold este CNT, primul putand fi implementat folosind PAL cu polaritate de iesire programabila si alegand iesirea sa fie logic negat. Functia PAL devine:

$$HOLD=S1*\overline{Wait}+S2*Wait+S3*\overline{Wait}+S6*Wait+S9*Wait+S12*Wait$$

Din moment ce decodificatorul furnizaza starea curenta in logic negat, este usor de descris functia HOLD folosind versiunile active low ale intrarilor starii. PAL este programat astfel:

$$HOLD=\overline{S1}*\overline{Wait}+\overline{S2}*Wait+\overline{S3}*\overline{Wait}+\overline{S6}*Wait+S9*Wait+S12*Wait$$

Implementare alternativa bazata pe MSI a numaratorului cu salt

Pana acum am implementat semnalele CLR, LD si CNT cu porti discrete sau PAL-uri. Putem folosi o alta metoda, bazata pe multiplexoare, pentru a a calcula aceste semnale.

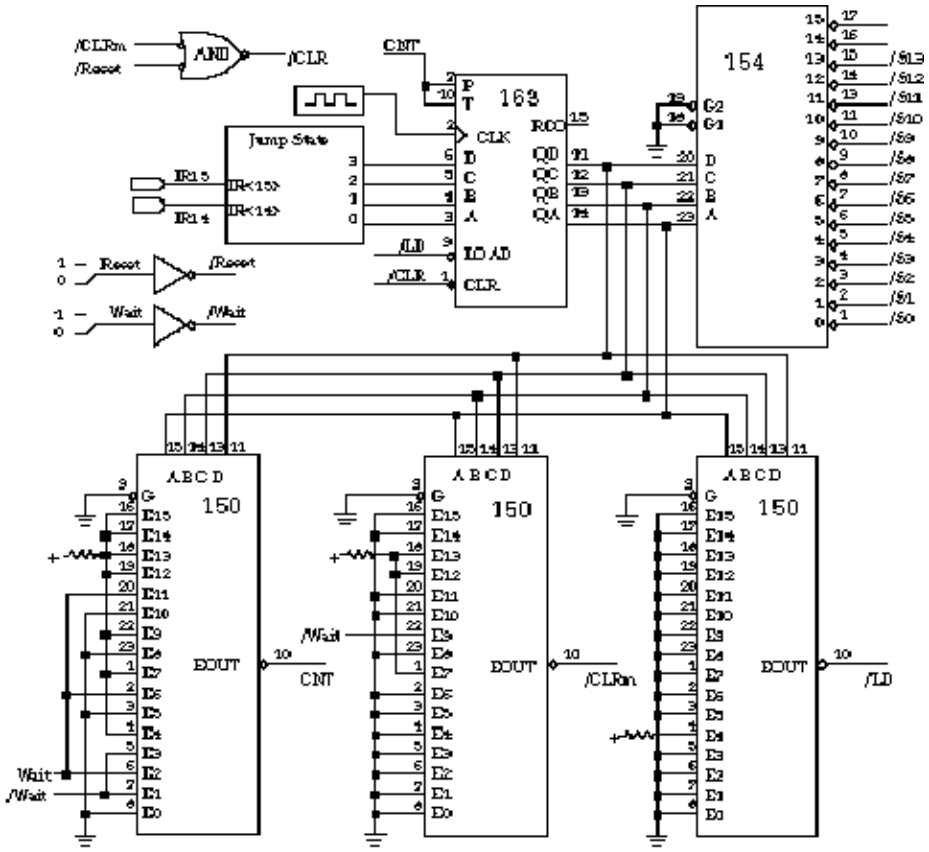


Figure 12.18 Schematic description of another jump-counter implementation

Figura 12.18 arata cum putem face acest lucru pentru diagrama de stari a unei CPU simple. Calculam CLR, LD si CNT folosind starea curenta pentru a selecta exact o singura intrare a multiplexorului. De exemplu, semnalul LD ar trebui validat in starea S4 (OD) si nu in alta. Cand liniile selectate ale LD din multiplexor ajung in 0100 (starea 4), intrarea E4 este directata catre iesirea EOUT. E4 este pe valoare ridicata, in timp ce toate celelalte intrari in multiplexor au valori low. Iesirea multiplexorului este validata activ low in acest caz. Este invalida in toate celelalte cazuri.

Iesirile active low ale multiplexorului fac ca aceasta implementare sa fie dificila. Deoarece intrarea LD a numartorului este de asemenea activa low, mentinem polaritatea corecta. O intrare valida logic de la multiplexor genereaza o iesire negativa valida, care determina contorul sa incarce(load).

Semnalul CLR de la multiplexor, CLRm, trebuie sa fie trecut printr-o operatie SAU cu Reset pentru a forma semnalul CLR al numaratorului.

CLR=CLRm+Reset

Insa deoarece semnalul clear al numaratorului este activ low, trebuie sa aplicam legile DeMorgan. Acum devine o functie NOR:

$\overline{CLR}=\overline{CLRm+Reset}$ $\overline{CLR}=\overline{CLRm} * \overline{Reset}$

Folosim a doua implementare in figura 12.18 (AND este la fel cu OR cu intrari si iesiri complementate). Iesirea activa low a multiplexorului CLR este acum polarizata corect. Combinat cu semnalul global de Reset, poate aduce masina in starea S0.

Sa ne ocupam de intrarile in multiplexorul CLR. Inca o data, sunt active high: valideaza high o intrare a multiplexorului daca semnalul CLR va avea efect in starea asociata. Asadar, E7, E12 si E13 sunt legate high, deoarece acestea corespund starilor S7, S12 si S13. Ne vom intoarce mereu in S0 din aceste stari. Pe de alta parte, tranzitiile din s) in S0 au loc doar cand Wait nu mai este valid. Asadar, Intrarea la E9 este semnalul $\overline{Wait}$, care este conditia de intoarcere in S0.

Controlul CNT este cel mai complex, deoarece iesirea multiplexorului este activa low dar semnalul de control al numaratorului este activ high. Solutia este "impingerea bulei" pana la intrari. In acest caz, validati o intrare a multiplexorului low daca vreti ca o numarare (count) sa aiba loc. Deoarece aceasta are loc neconditionat in starile S0, S5, S8 si S10, apoi E0, E5, E8 si E10 sunt toate grupate low.

Pentru starile in care numararea este optionala, intrarile multiplexorului sunt grupate in complementul conditiei. Deci, S1 si S3 avanseaza cand Wait este valid, E1 si E3 sunt grupate la $\overline{Wait}$.S2, S6 si S11 avanseaza cand $\overline{Wait}$ este valid, deci E2, E6 si E11 sunt grupate cu Wait.

Generarea microoperatilor

Micropoperatiile pot fi generate din starile si intrarile decodate. Expresiile logice sunt descries mai jos. Pentru a asigura o operatie de sincronizare potrivita, intrarile Reset si Wait ar trebui sincronizate inainte de a fi folosite ca mai jos:

0 --> PC: Reset
PC + 1 --> PC: *S*0
PC --> MAR: *S*0
MAR --> Memory Address Bus:
Wait (*S*1 + *S*2 + *S*5 + *S*6 + *S*8 + *S*9 + *S*11 + *S*12)
Memory Data Bus --> MBR:
$\overline{Wait}$(*S*2 + *S*6 + *S*11)
MBR --> Memory Data Bus:
Wait (*S*8 + *S*9)
MBR --> IR: Wait *S*3
MBR --> AC: Wait *S*7

AC --> MBR: IR15 IR14 *S*4
AC + MBR --> AC: Wait *S*12
IR<13:0> --> MAR:
($\overline{IR15}$ $\overline{IR14}$+ $\overline{IR15}$IR14 + IR15 $\overline{IR14}$) *S*4
IR<13:0> --> PC: AC15 *S*13
1 --> Read/$\overline{Write}$:

Wait (*S*1 + *S*2 + *S*5 + *S*6 + *S*11 + *S*12)
0 --> Read/$\overline{\text{Write}}$:
Wait (*S*8 + *S*9)
1 --> Request:
Wait (*S*1 + *S*2 + *S*5 + *S*6 + *S*8 + *S*9 + *S*11 + *S*12)

Acestea sunt implementate cel mai usor printr-o forma de programare logica.

12.4.Branch sequencers

Intr-o masina de stare clasica, finita, starea urmatoare este calculata explicit drept functie de iesire a masinii. O strategie directa este de a plasa bitii reprezantand starea urmatoare intr-o memorie ROM adresata de starea curenta si intrari. Problema cu aceasta abordare este ca dimensiunea memoriei se dubleaza pentru fiecare intrare aditionala.

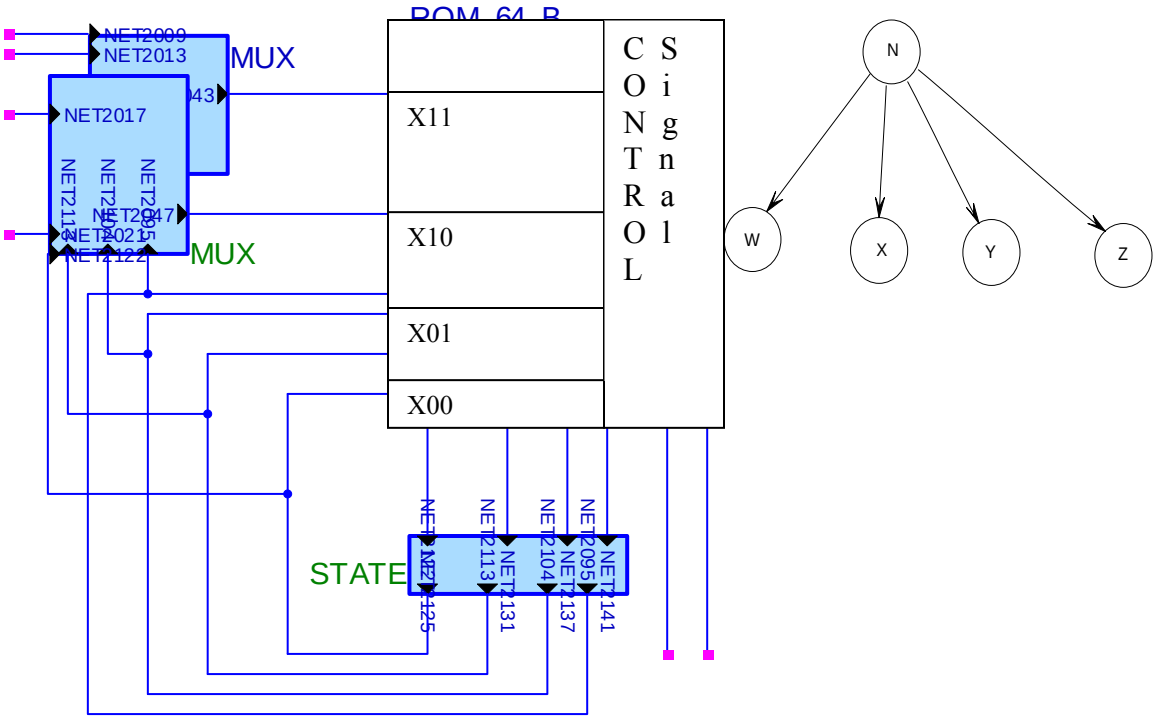
Numaratorul cu salt reprezinta un compromis intre dimensiunea ROM si circuite externe. Doar starile de salt sunt incluse in ROM. Intr-un contror de salt pur, memoria ROM este adresata doar de starea curenta. Chiar si abordarea hibrida selecteaza in mod tipic o mica parte din intrari pentru a forma o parte din adresa ROM. Alte iesiri ale masinii sunt constituite prin logica ce combina starea decodata si intrarile procesorului.

Urmatoarea arhitectura se numeste branch sequencer. Se afla intre cele doua extreme: implementarea clasica cu memorii ROM a masini si numaratoare de satre. Starile urmatoare sunt stocate in ROM, dar fiecare stare este obligata sa aiba doar un numar limitat de stari urmatoare, intotdeauna putere a lui 2.

Deoarece este rar cazul in care o masina trebuie sa vada toate intrarile pentru a determina starea urmatoare in orice stare in care se afla, doar o mica parte a intrarilor trebuie examinata. Ideea este sa folosim o logica externa care sa selecteze acele intrari.

12.4.1.Organizarea branch sequencer

Figura 12.19 arata structura ramificata pe 4 cai a unui secventiator ce implementeaza un controler Mealy. Bitii semnificativi de adresa ai ROM sunt formati din starea curenta., cei putin semnificativi din valorile a doua intrari a si b. Deci fiecare stare are 4 posibile stari urmatoare. Daca o stare data are doar o stare urmatoare, aceeasi valoare este pusa in toate cele 4 cuvinte ale ROM.



Bitii de adresa a si b deriva din iesirile multiplexorului. Starea selecteaza dintre intrari setul particular de doua intrari ce detremina starea urmatoare a masinii.

Ca un exemplu sa luam masina din figura 11.23. Intrarile sunt Wait, bitul semnificativ al AC si cele doua coduri de operatie ale IR.

Generalizare

Este posibil sa construim un secventiator cu 2N cai folosind N intrari pentru a forma o parte din adresa ROM. Desigur exista un compromis intre gradul de ramificare, dimensiunea ROM si numarul de stari.

Implementarea CPU simpla cu branch sequencer

	ROM ADDRESS			ROM CONTENTS	
	Reset	Current State	α β	Next State	Register Transfer Operations
RES	0	0000	X X	0001 (IF0)	PC → MAR, PC + 1 → PC
IF0	0	0001	0 0	0001 (IF0)	
	0	0001	1 1	0010 (IF1)	MAR → Mem, Read, Request
IF1	0	0010	0 0	0011 (IF2)	MAR → Mem, Read, Request
	0	0010	1 1	0010 (IF1)	Mem → MBR
IF2	0	0011	0 0	0011 (IF2)	
	0	0011	1 1	0100 (ID)	MBR → IR
ID	0	0100	0 0	0101 (LD0)	IR → MAR
	0	0100	0 1	1000 (ST0)	IR → MAR, AC → MBR
	0	0100	1 0	1001 (AD0)	IR → MAR
	0	0100	1 1	1101 (ER0)	IR → MAR
LD0	0	0101	X X	0110 (LD1)	MAR → Mem, Read, Request
LD1	0	0110	0 0	0111 (LD2)	Mem → MBR
	0	0110	1 1	0110 (LD1)	MAR → Mem, Read, Request
LD2	0	0111	X X	0000 (RES)	MBR → AC
ST0	0	1000	X X	1001 (ST1)	MAR → Mem, Write, Request; MBR → Mem
ST1	0	1001	0 0	0000 (RES)	
	0	1001	1 1	1001 (ST1)	MAR → Mem, Write, Request; MBR → Mem
AD0	0	1010	X X	1011 (AD1)	MAR → Mem, Read, Request
AD1	0	1011	0 0	1100 (AD2)	
	0	1011	1 1	1011 (AD1)	MAR → Mem, Read, Request
AD2	0	1100	X X	0000 (RES)	MBR + AC → AC
ER0	0	1101	0 0	0000 (RES)	
	0	1101	1 1	0000 (RES)	IR → PC
	1	XXXX	X X	0000 (RES)	PC → 0

Figure 12.20 Branch sequencer ROM encoding

Figura 12.20 arata programarea ROM pentru controlerul procesorului Mealy.Adresa este formata din semnalul Reset , starea curenta- 4 biti, si intrarile conditionale a si b. Includerea Resetului dubleaza marimea ROM dar acesta este cel mai usor mod de a implementa functia de reset.

IR<15> si IR<14> sunt conenctate la intrarile selectate de starea 4(OD). Wait este cuplat cu ambele multiplexoare la intrarile 1,2,3,6,9 si 11.

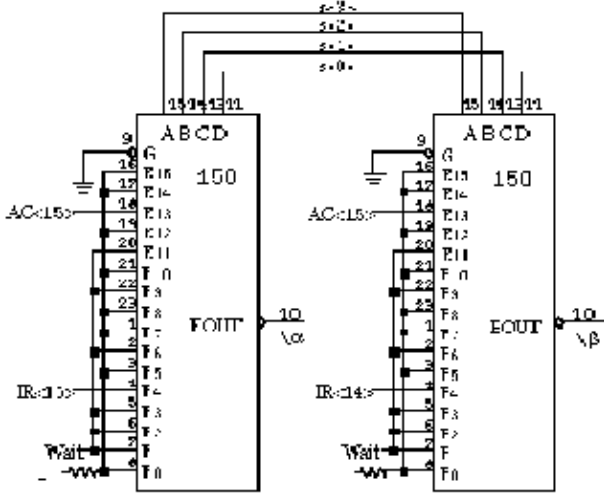


Figure 12.21 Multiplexer setup for example branch sequencer.

Destul de putine stari folosesc abilitatea controlerului de a suporta stari cu ramificari pe 4 niveluri. Apoi sunt mai putine intrari decat stari: 4 intrari si 16 stari.Deoarece masina de stare se uita doar la trei seturi diferite de intrari pare destul de anevoios sa foloseasca multiplexoare 16-1 pe semnalele a si b.

Branch sequencer-organizare orizontala

Folosind cateva multiplexoare in plus putem inlocui memoria ROM din fig 12.19 cu o memorie de o alta arhitectura.

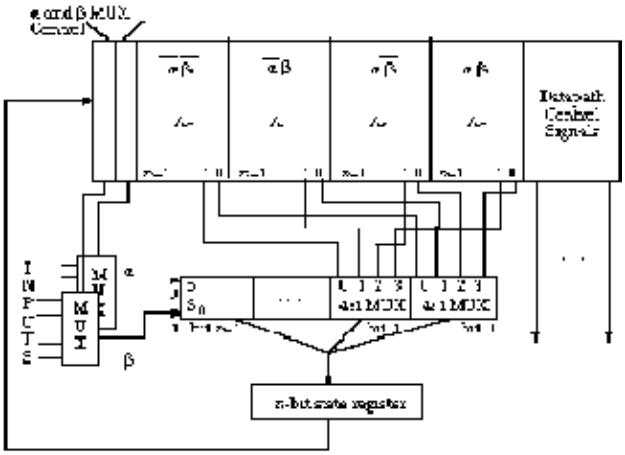


Figure 12.22 Horizontal next-state organization

Exista doua diferente. Intai MUX sunt controlate de semnale de iesire codate, de la ROM, si nu direct de catre stare.

Apoi, sunt asezate 4 posibile stari urmatoare in pozitie orizontala.Daaca masina are un numar mare de stari dar putine intrari, bitii de control ai MUX inglobati in ROM dau voie sa se foloseasca MUX cu mai putine intrari pentru controlul logicii starii urmatoare.

Sa examinam exemplul cu CPU simpla. Folosind aceasta abordare putem inlocui MUXurile verticale 16-1 cu 2-1. MUXul a are IR <15> si AC<15> intrari, MUXul b are intrarile Wait si IR<14>. Fiecare MUXeste controlat de un singut bit dintr-un cuvant ROM. Cand masina este in starea OD bitii de control ai lui a si b sunt setati sa selecteze IR<14> si IR<15>.

Cand masina este in starea ei de executie pentru instructiunile BRN, bitii de control ai MUXurilor sunt setati sa selecteze AC <15> din MUXul a. Bitul MUXului b este *don't care*.

Schema orizontala a starilor urmatoare este avantajoasa in controlerele mari si complexe cu multe ramifictii. Marirea lungimea cuvantului ROM necesita de obicei mai putini biti decat cresterea numarului de cuvinte.

12.5.Microprogramare

Ne-am ocupat pana acum de metode alternative pentru organizarea logicii starii urmatoare.. Acum putem discuta varinte de organizare a semnalele de iesire ce controleaza calea datelor. De obicei semnalele de control sunt implementate folosind logica discreta, chiar daca implementarea foloseste PAL/PLA. *Microprogramarea,* pe de alta parte, reprezinta o metoda pentru implementarea controlului procesorului in care semnalele de iesire sunt stocate in ROM.

Cele doua variante de microprogramare sunt verticala si orizontala. *Microprogramarea orizontala* foloseste o iesire ROM pentru fiecare punct de control al caii de date. *Microprogramarea verticala* este bazata pe observatia ca doar o parte a acestor semnale

este vreodata avtivata in o stare data. Deci iesirile ce control pot fi stocate in ROM intr-o forma codata, reducand efectiv lungimea cuvantului ROM pe baza introducerii unei logici externe de decodare.

Codificarea semnalelor de control poate limita operatiile de pe calea de date ce pot avea loc in paralel. Daca acesta este situiatia, vom avea nevoie de mai multe cuvinte ROM multiple pentru a coda aceeasi operatie care poate fi facuta intr.un singur cuvant ROM.

Arta realizarii unei unitati de control microprogramate consta in a gasi balanta potrivita intre paralelismul abordarii orizontale si economia codarii verticale a ROM.

12.5.1.Microprogramarea orizontala

Figura 12.22. ofera principalele elemente ale unui astfel de controler. Un format de cuvant extrem de orizontal ar avea 1 bit pentru fiecare microoperatie.

Procesorul considerat ca exemplu accepta 14 operatii cu transfer pe registre. Le vom descompune in 22 de microoperatii discrete.

PC --> ABUS
IR --> ABUS
MBR --> ABUS
RBUS --> AC
AC --> ALU *A*
MBUS --> ALU B
ALU ADD
ALU PASS B
MAR --> Address Bus
MBR --> Data Bus
ABUS --> IR
ABUS --> MAR
Data Bus --> MBR
RBUS --> MBR
MBR --> MBUS
0 --> PC
PC + 1 --> PC
ABUS --> PC
Read/$\overline{\text{Write}}$
Request
AC --> RBUS
ALU Result --> RBUS

Un cuvant ROM foarte lung pentru un secventiator cu 4 ramuri ar avea a si b biti ai multiplexorului, patru 4bit stari urmatoare, si 22 de biti de microoperatii. Asta inseamna un cuvant ROM de 40 biti lungime.

1

A MUX	b MUX	A0	A1	A2	A3	PC->ABUS	IR->ABUS	MBR->ABUS	RBUS->AC	AC->ALU a

MBUS->ALU b	ALU ADD	MAR->Adress Buss	MBR->Data Buss	ABUS->IR	ABUS->MAR	RBUS->MBR

40

MBR->MBUS	0->PC	PC+1->PC	ABUS->PC	Read/Write	Request	Ac->rbus	Alu Result->RBUS

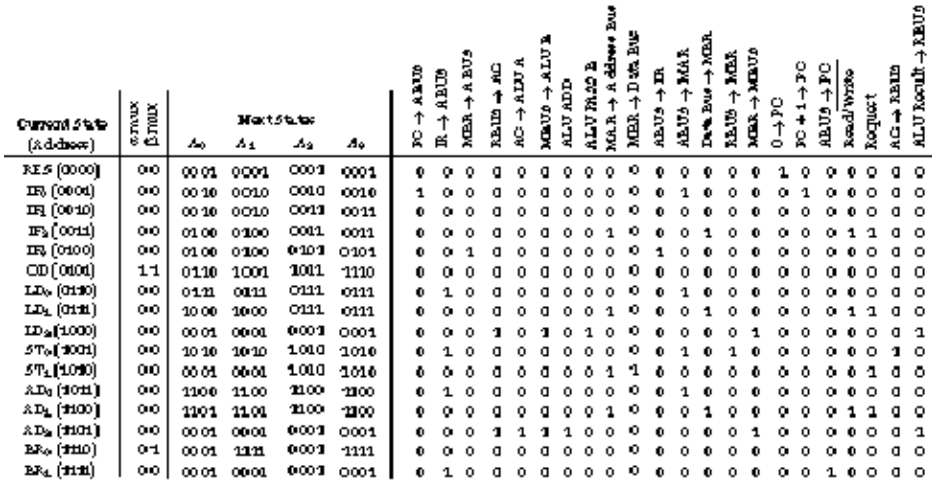


Figure 12.24 ROM content for the Moore processor controller.

Figura 12.24 descrie continutul ROM pentru controlerul Moore din figura 12.1. MUXul a are intrarile Sel0=Wait, Sel1=IR<15> si intrarile pentru b sunt Sel0=AC15, Sel1=IR<14>.Presupunem ca registrul starilor urmatoare se reseteaza direct.

Reducerea dimensiunii cuvantului ROM prin codificare

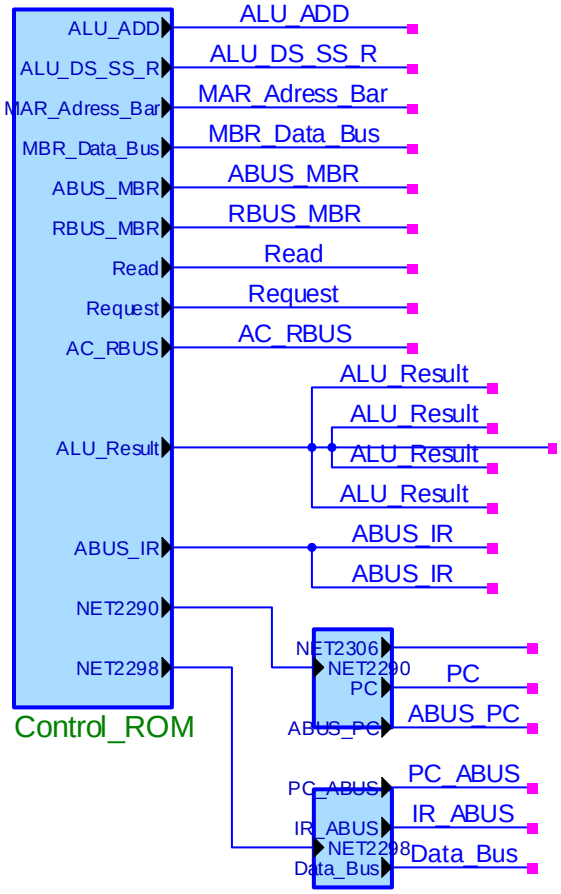
Abordarea orizontala ofera cea mai mare flexibilitate, furnizand acces la toate punctele de control ale caii de date in acelsi timp. Dezavantajul este dimensiunea cuvantului, care poate depasi cateva sute de biti in controlere complexe.

Un mod bun de a reduce dimensiunea ROM este codificarea iesirii. Acest lucru nu duce neaparat la o pierdere inerenta a paralelismului. In concluzie, anumite combinati de control nu au sens d.p.d.v. logic(de exemplu 0->PC si PC +1 ->PC sunt logic exclusive). Mai mult, continutul ROM din figura 12.24 este inconsistent. In orice strare data, doar cateva semnale de control sunt valide.

De exemplu, cele 3 microoperatii `0 --> PC, PC + 1 --> PC, si ABUS -->PC` nu sunt validate in aceeasi stare. Cu un decodificator 2-4 extern, un bit ROM poate fi salvat prin codificare:

00	No PC control
01	**0 --> PC**
10	**PC + 1 --> PC**
11	**ABUS --> PC**

Putem salva biti ROM aditionali daca gasim semnale care nu au legatura, fiind nevalidate simultan. Acestea sunt bune pentru codare. De exemplu putem combina `PC --> ABUS, IR --> ABUS, si Data Bus --> MBR` codificandu-le in 2 biti.



Pe masura ce in ROM sunt bagate mai multe semnale sub forma codata, ne mutam de la un format orizontal extrem la unul care este si mai vertical.

12.5.2.Microprogramare verticala

Microprogramarea verticala foloseste extensiv codarea ROM pentru a reduce lungimea cuvantului de control. Pentru a realiza acest lucru, folosim multiple formate microcuvant. De exemplu, multe stari nu necesita o ramura conditionala; avanseaza in urmatoarea stare secvential. Decat sa aiba in fiecare microcuvant o stare urmatoare si o lista de microoperatii, putem scurta cuvantul ROM prin separarea acestor 2 functii in formate de microcuvinte individuale: unul pentru salturi de ramuri conditionale si altul pentru operatiile/microoperatiile registrelor de transfer.

Scurtarea dimensiunii cuvantului ROM are insa neajunsuri. E posibil sa avem nevoie de mai multe ROM intr-o secventa pentru a putea realiza aceleasi operatii ca in cazul unui microcuvant orizontal. Combinatia intre niveluri suplimentare de decodare, accese multiple ROM pentru executarea secventei de operatii de control si sacrificarea

potentialului paralelism al abordarii verticale duce la o implementare inceata. Ciclul masina creste ca si numarul de cicli necesari pentru a executa o instructiune.

In ciuda aceste ineficiente, proiectantii prefera microcod vertical deoarece seamana cu programarea in limbaj de asamblare. Deci compromisul intre mocrocod vertical si orizontal e o problema de implementare facila contra performanta.

Formatul microcodului vertical pentru CPU simpla
Vom realiza un microcod simplu pentru procesor. Vom introduce doar doua formate: branch jump si transfer/operatii pe registri.

Intr-un microcuvant branch jump, includem un camp pentru a selecta semnalul de testat (Wait, AC<15>, IR<15>, IR<14>) si valoarea pentru test(0 sau 1). Daca semnalul se potriveste cu valoarea specificata, restul microcuvantului contine adresa urmatoare din ROM care trebuie adusa (fetch).

Microcuvantul pentru transfer/operatie pe registru contine 3 campuri: registru sursa, registru destinatie, camp pentru operatie care instruieste unitatile functonale precum ALU ce sa faca.

Surse:

PC --> ABUS
IR --> ABUS
MBR --> MBUS
AC --> ALU A
MAR --> Mem Address Bus
MBR --> Mem Data Bus
MBR --> MBUS
AC --> RBUS
ALU Result --> RBUS

Destinatii:

RBUS --> AC
MBUS --> ALU B
MBUS --> IR
ABUS --> MAR
Mem Data Bus --> MBR
RBUS --> MBR
ABUS --> PC

Operatii:

ALU ADD
ALU PASS B

0 --> PC
PC + 1 --> PC
Read (Read, Request)
Write (Write, Request)

Putem codifica cele 9 surse intr-un camp de 4 biti. Avem mai multe surse decat destinatii. Putem presupune ca AC este legat la intrarea A a ALU, cum MBUS este legat la intrarea B. MBR este singura sursa pe MBUS, deci putem elimina microoperatiile MBR->MBUS. Deci avem 7 surse si 6 destinatii, codate cu 3 biti.

La scrieri de memorie, in timpul unei stocari, MAR trebuie sa conduca liniile de adresa si MBR liniile de date. Dar aceste operatii se exclud reciproc.

Putem muta operatiile MBR-> Mem Data Bus de la sursa la destinatie, gandind memoria ca o destinatie in loc sa consideram MBR o sursa. Codarea celor 2 formate poate fi facuta cu 10 biti.

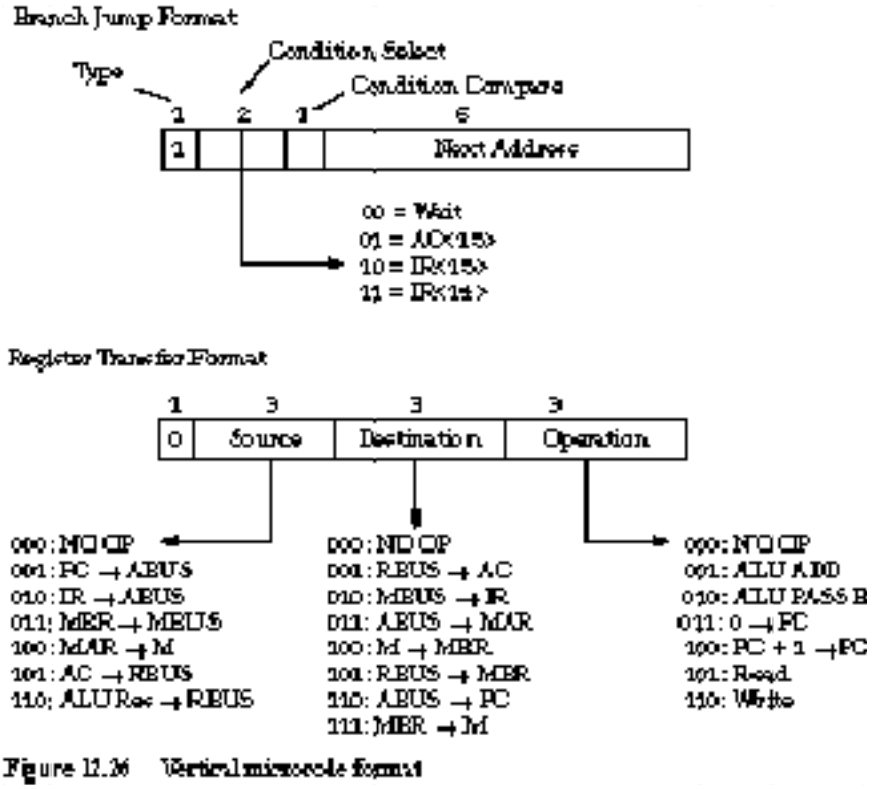


Figure 12.26 Vertical microcode format

Continutul ROM pentru controlerul Moore din figura 12.27.

ROM ADDRESS	SYMBOLIC CONTENTS		BINARY CONTENTS
000000	RES	RT PC → MAR, PC + 1 → PC	0 001 011 100
000001	IF0	RT MAR → M, Read	0 100 000 101
000010		BJ Wait=0, IF0	1 000 000 001
000011	IF1	RT MAR → M, M → MBR, Read	0 100 100 101
000100		BJ Wait=1, IF1	1 001 000 011
000101	IF2	RT MBR → IR	0 011 010 000
000110		BJ Wait=0, IF2	1 000 000 101
000111		RT IR → MAR	0 010 011 000
001000	OD	BJ IR<15>=1, OD1	1 101 010 101
001001		BJ IR<14>=1, ST0	1 111 010 000
001010	LD0	RT MAR → M, Read	0 100 000 101
001011	LD1	RT MAR → M, M → MBR, Read	0 100 100 101
001100		BJ Wait=1, LD1	1 001 001 011
001101	LD2	RT MBR → AC	0 110 001 010
001110		BJ Wait=0, RES	1 000 000 000
001111		BJ Wait=1, RES	1 001 000 000
010000	ST0	RT AC → MBR	0 101 101 000
010001		RT MAR → M, MBR → M, Write	0 100 111 110
010010	ST1	RT MAR → M, MBR → M, Write	0 100 111 110
010011		BJ Wait=0, RES	1 000 000 000
010100		BJ Wait=1, ST1	1 001 010 010
010101	OD1	BJ IR<14>=1, ER0	1 111 011 101
010110	AD0	RT MAR → M, Read	0 100 000 101
010111	AD1	RT MAR → M, M → MBR, Read	0 100 100 101
011000		BJ Wait=1, AD1	1 001 010 111
011001	AD2	RT AC + MBR → AC	0 110 001 001
011010		BJ Wait=0, RES	1 000 000 000
011011		BJ Wait=1, RES	1 000 000 000
011100	ER0	BJ AC <15>=0, RES	1 010 000 000
011101		RT IR → PC	0 010 110 000
011110		BJ AC<15>=1, RES	1 011 000 000

Figure 12.27 Vertical microprogramming ROM contents

Ne vom ocupa de Reset in exterior. Formatul simbolic ar trebuie sa fie usor de intuit si are o asemanare mare cu programarea in asamblare. Cele doua formate alternative sunt notate BJ-branch jump si RT- register transfer. Intai se scrie conditia, urmata de adresa.

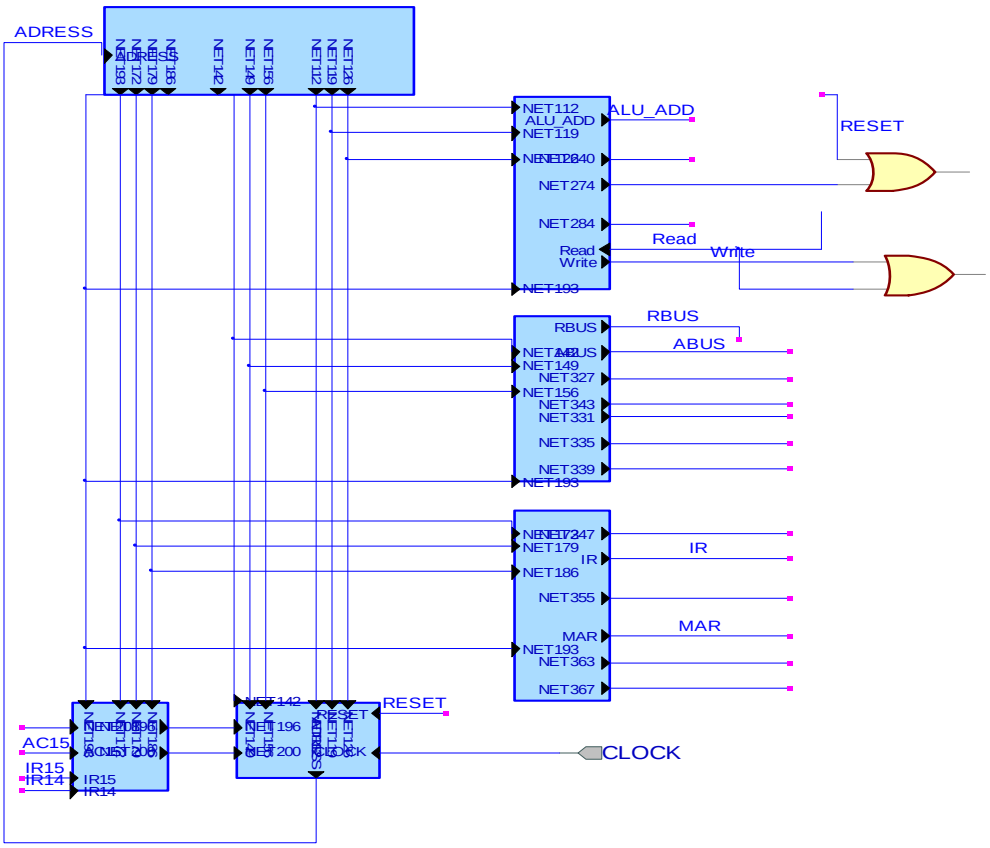
BJ Wait = 0, IF0

Testeaza daca Wait este sau nu asignat. Daca da, atunci urmatoarea instructiune efectueaza fetch.
Formatul RT este scris ca SRC->DST urmat de orice operatii care sunt efectuate in paralel cu transferul din registri.

RT PC --> MAR, PC + 1 --> PC

Este o operatie de transfer care mapeaza pe microoperatiile PC->ABUS, ABUS->MAR si PC+1->PC.

Detalii ale implementarii controlerului prin microcod vertical.



Implementam registrul de stare urmatoare drept un numarator microprogramat, cu CLR, CNT si LD. Un bloc logic conditional determina daca sa valideze CNT sau LD in functie de tipul microinstructiunii si conditia care este testata. Decodoarele externe mapeaza operatiile de transfer cu registru codate cu microoperatiile acceptate de calea de date.

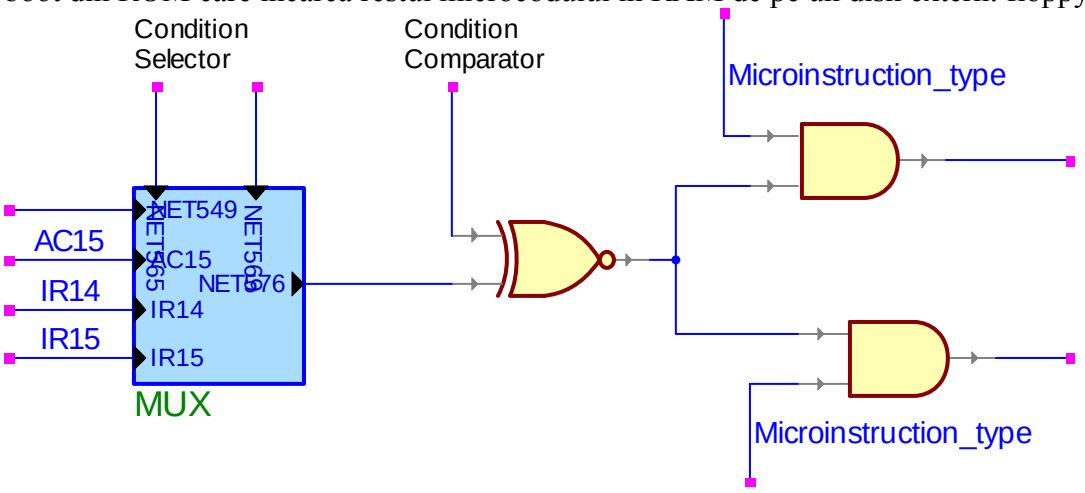
Blocul logicii conditionale e descris in fig 12.29. Bitii selectorului conditional ai microinstructiunii selecteaza unul din patru semnale posibile de testat. Conditia selectata este comparata cu bitul specificat. Semnalul de incarcare(load) al microprocesorului este validat daca microinstructiunea curenta este de tip BJ –tip 1, si daca bitii de conditie si comparare sunt identici. Valoarea de incarcat vine de la cei 6 biti putin semnificativ ai microinstructiunii. Semnalul count este validat daca instructiunea este de tip RT –tip 0 sau bitii de conditie si comparare sunt diferiti.

12.5.3.Extinderea controlului

Acesta nu este fix in cazul ROM. Unele computere mapeaza o parte a adreselor de stocare in RAM, aceeasi memorie pe care programatorii o folosec pentru instructiuni si date. Aceasta are flexibilitatea ca in limbaj de asmblare sa fie scris propriul cod, extinzand setul de baza al masinii cu instuctiuni specifice.

Desigur, majoritatea programatorilor nu sunt destul de priceputi pentru acest lucru, desi multe masini cu seturi complexe de instructiuni au implementata aceasta facilitate de a permite scrierea propriilor instructiuni. Motivul este simplu. Din moment ce

microprogramul pentru o masina de stare complexa este complicat, nu este nemaintalnit ca el sa fie plin de bug-uri. Avand la indemana facilitatea de a scrie instructiuni permite revizuirea si updatarea ei pe moment. Dupa pornire, masina executa un microprogram de boot din ROM care incarca restul microcodului in RAM de pe un disk extern: floppy, etc.



Concluzii finale

In acest capitol am prezentat metode pentru organizarea unitatii de control a unui procesor simplu. Am inceput cu masina Moore si Mealy. Desi potrivite pentru masini de stare simple, finite, aceste implementari monolitice nu sunt suficiente in structurarea masinilor de stare complexe.

Apoi am studiat numaratoarele de salt, o metoda ce foloseste componente MSI TTL precum numaratoare, multiplexoare si decodificatoare pentru a implementa starea urmatoare si functiile de iesire ale masinii. Aceasta abordare conduce la o implementare simpla care este usor de modificat si reparat. Insa, este limitata de un numar mic de stari.

Urmatorul tip de controler este branch sequencer. Aceasta este o metoda pentru aranjarea logicii starii urmatoare ce scrie potentiale stari urmatoare intr-o memorie ROM. Logica selectorului foloseste intrarile masinii pentru a alege.

La sfarsit, am examinat microproramarea, incluzand cele doua variante: verticala si orizontala. Microprogramare evita implementarea logicii discrete ce genereaza starea urmatoare si functiile de iesire prin implementarea acestora sub forma unei tabele de 0 si 1 stocata in memorie. Microprogramarea orizontala aloca un singur bit pentru fiecare semnal de control, iar varianta verticala incearca sa reduca dimensiunea ROM prin codare extensiva. O implementare tipica de microcod vertical accepta diverse formate de microcuvinte in cadrul ROM.

Toate aceste abordari sunt potrivite pentru implementari cu dispozitive logice programabile sau ROM. Proiectarea bazata pe ROM este cea mai simpla, dar se poate ajunge la dimensiuni considerabile. In cea mai simpla implementarea, fiecare semnal de

intrare poate deveni parte a adresei ROM. Programarea verticala reduce dimensiunea memoriei ROM de control prin separarea alegerii starii urmatoare de operatiile de transfer pe registri si prin codificare extensiva ulterioara.

Merita sa comparam controlerele cu componente logice discrete cu cele realizate prin mcroproiramare. Controlul „cablat" tinde sa fie mai rapid, dar mai greu de modificat. Formeaza baza masinilor cu set de instructiuni redus.

Controlerele microprogramat, pe de alta parte, beneficiaza de densitatea memoriei prin implementarea de functii complexe sub forma de succesiuni de 1 si 0 stocate in ROM. Arhitectura unor astfel de controlerele este mai generala decat cea a celor „cablate", facand posibila schimbarea setului de instructiuni fara schimbarea traseelor de date. Totusi, aceasta flexibilitate are drept urmare cresterea timpilor ciclilor procesorului, iar dimesiunea ROM are tendinta sa creasca. Comparat cu abordarea „cablata" microprogramarea furnizeaza o modalitate rezonabila de a structura implementarea si proiectarea controlerelor complexe. Este modalitatea adoptata in cazul mesinii VAX. Digital Equipment Corporation, Intel 80x86 sau Motorola 68x00.