DATAPATH

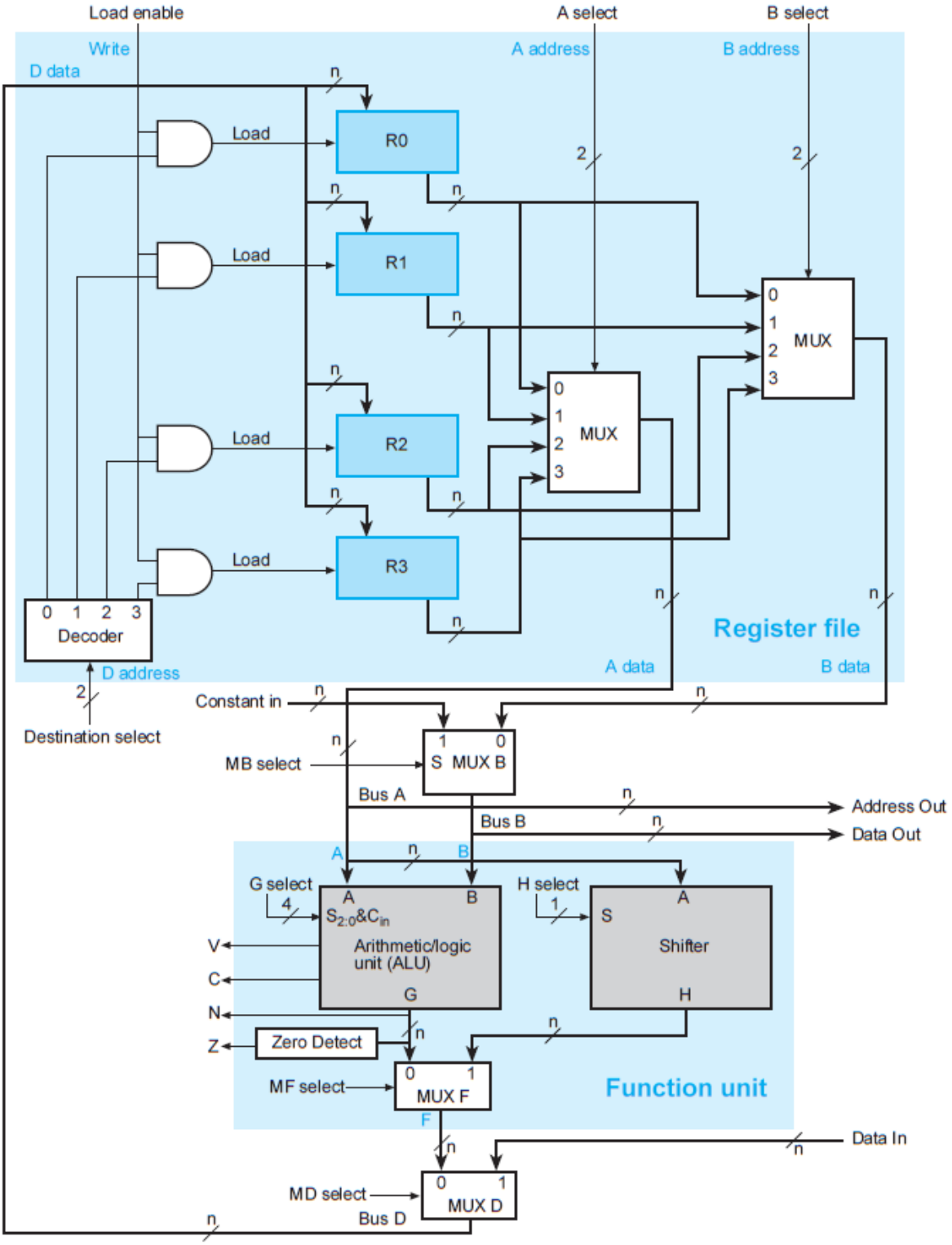
Dr. Cahit Karakuş

2021 - Istanbul

İçindekiler

Giriş

Datapath ve Control unit bir mikroişlemcinin (CPU) iki parçasıdır. Datapath iki temel kısımdan oluşur, Register'lar ve Function Unit. Genel olarak bir Datapath'in block diagramı aşağıdaki şekildeki gibidir.



R1 ← R2 + R3 ifadesi, R2 ve R3 register'larındaki dataları toplayıp R1 register'ına yazma işlemini anlatır. R1 register'ına yükleme Load Enable girişinin aktif olması ile mümkündür. Load Enable girişi 1, Destination Select decoder'ının R1'e giden çıkşının da 1 olması gerekir.

A select'in bağlı bulunduğu Mux'dan R2'yi, B selectin bağlı bulunduğu Mux'dan R3'ü seçilir (A select=10, B select=11 olmalıdır.). Böylece A select'e bağlı olan Mux'dan çıkan R2 register'ındaki data yoluna devam ederek ALU'ya kadar ulaşır.

B select'e bağlı olan Mux'dan çıkan R3 register'ı ise MUXB'ye ulaşır. Bu Mux'da dışarıdan herhangi bir sabitle işlem yapılmayacağıiçin R3'ün yoluna devam etmesi için MB select=0 yapılır.

Şimdi R2 ve R3 register'larındaki dataların ikisi de ALU'ya ulaşmış olur. Bu durumda ne işlem yapılacağı seçilir. ALU, aritmetik ve logic işlemleri gerçekleyebilen bir yapıdır. Burada G select girişinden toplama işlemini yapan kod seçilir.

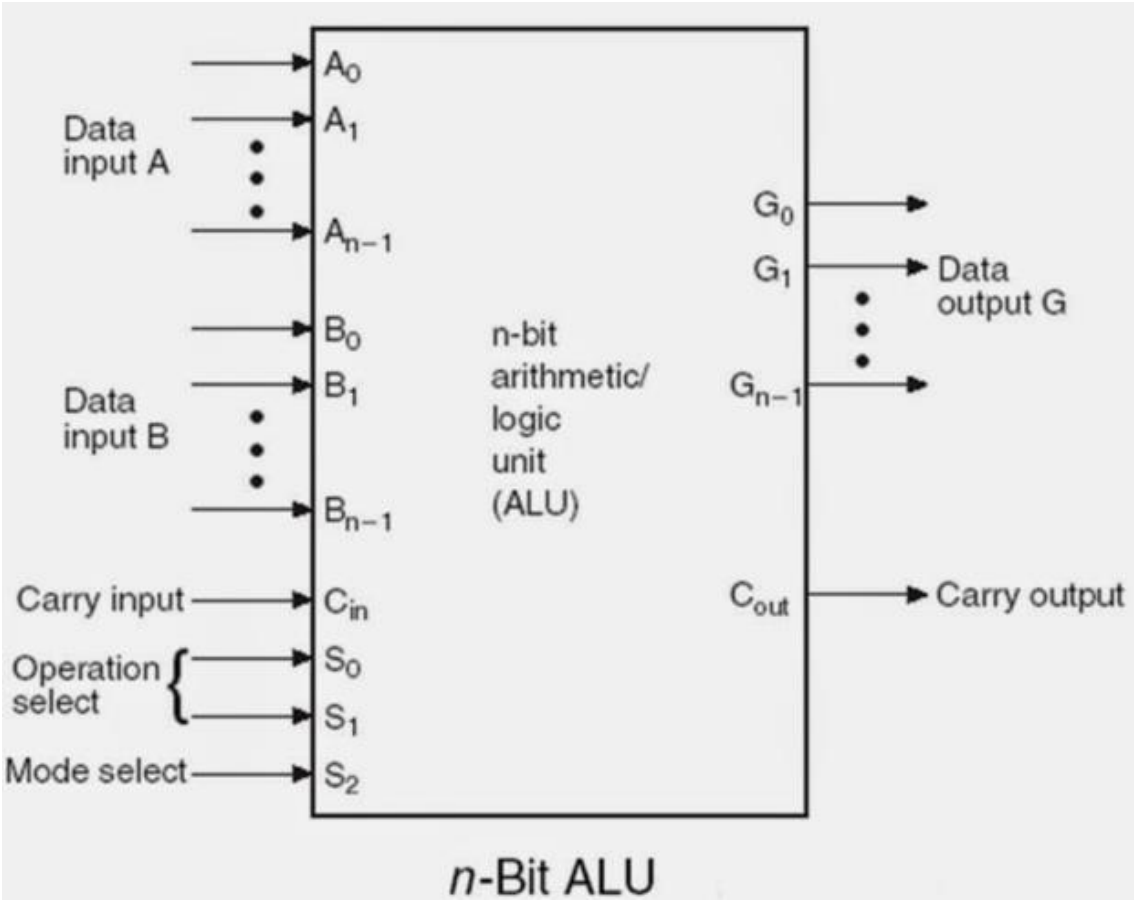
İşlem sonucu ALU'dan çıkış yaptıktan sonra MUXF'ye ulaşır. Burada MUXF'ye shifter'dan gelen bir giriş var. Eğer bir shift(kaydırma) işlemi olsaydı bu Mux'da 1 seçilir ancak ALU'dan gelen bilgi kullanılacağı için MF select =0 olmalıdır.

MUXF'den çıkış yapan bilgi MUXD'ye ulaşır. Bu Mux'un görevi ise dışarıdan bir datanın direk olarak bir register'a yazılması gerektiğinde 1 ucuna bağlı olan data in girişinden bilgiyi aldıktan sonra istenilen register'a yazar. Burada MUXF'den gelen data kullanılacağı için MD select=0 olmalıdır.

En sonunda MUXD'den çıkan data yolu takip ederek Load enable'ı aktif olan R1 register'ına yazılır.

1. Aritmetik Logic Unit(ALU)

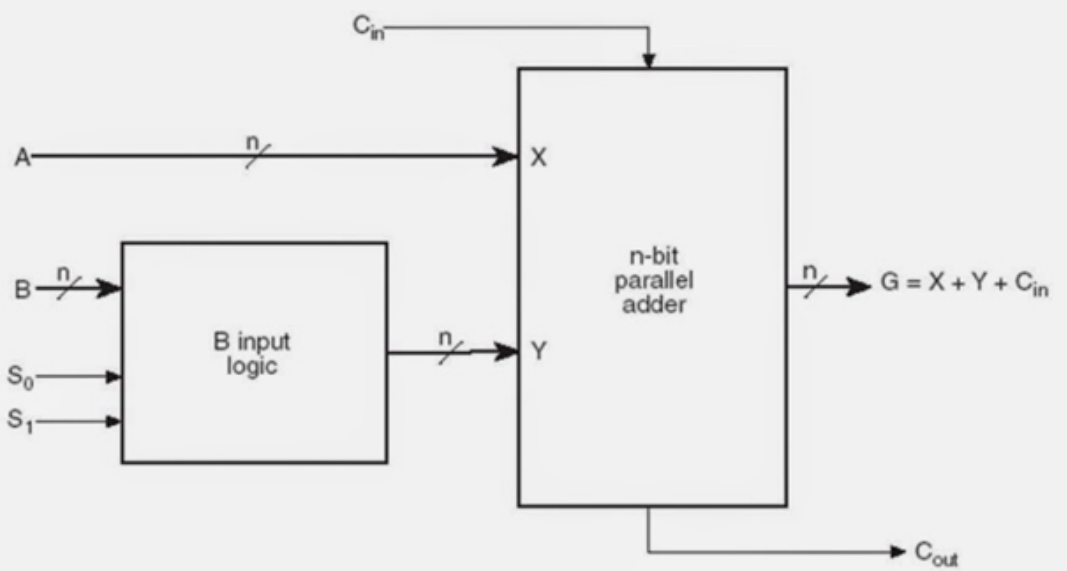
ALU aritmetik ve logic işlemlerin yapıldığı kombinasyonel mantık kapılarından oluşan devredir. ALU, Aritmetik ve Logic kısım olmak üzere iki kısma ayrılır. Herhangi bir clock pulsenin bulunmadığı kısımdır. ALU nun genel olarak yapısı şu şekildedir;



Şekilde görüldüğü üzere A'dan n-bit data, B'den n-bit data ve önceki işlemlerden gelebilecek bir elde değeri için bir Cin girişi var. S2 girişi işlemin aritmetik mi logic mi olacağına karar veren seçimdir. S1 ve S0 girişi ise aritmetik ya da mantıksal işlemin ne olacağını belirler. S1 ve S0 ile birlikte S2 girişi 0 olduğunda 8 adet farklı aritmetik işlem yaparak sonucu G çıkışından verir.

1.1. Aritmetik Devre

Aritmetik devrenin temel parçasına Full Adder devresi denir. Paralel Adder'a giriş yapan datalar kontrol edilerek çeşitli aritmetik işlemleri gerçekleyebilmek mümkündür. Bir aritmetik devrenin block diagramı aşağıda verilmiştir.



Block Diagram of an Arithmetic Circuit

A'dan gelen datalar üzerinde değişiklik yapılmadan doğrudan Paralel Adder'a gider. Şimdi S0 ve S1'in durumlarına göre hangi işlemleri yapabileceğine bakalım.

Select		Input	$G = A + Y + C_{in}$	
S_1	S_0	Y	$C_{in} = 0$	$C_{in} = 1$
0	0	all 0's	$G = A$ (transfer)	$G = A + 1$ (increment)
0	1	B	$G = A + B$ (add)	$G = A + B + 1$
1	0	$\overline{B}$	$G = A + \overline{B}$	$G = A + \overline{B} + 1$ (subtract)
1	1	all 1's	$G = A - 1$ (decrement)	$G = A$ (transfer)

Burada Y'nin hangi S1,S0 değerinde bir seçim olacaktır.

- S1,S0=00 değerinde Y girişi 0 olur ve böylece Cin=0 ise G çıkışı A olacaktır. Yani sadece transfer işlemi yapılmış olur. Yine bu durumda Cin=1 ise bu kez G çıkışı A+1 değerini alacaktır.
- S1,S0=01 değerinde Y çıkışına B datası doğrudan aktarılır. Cin=0 ise G=A+B, Cin=1 ise G=A+B+1 olacaktır.
- S1,S0=10 değerinde Y çıkışına B datası terslenerek aktarılır. Cin=0 ise G=A+B' .Cin=1 ise G=A+B'+1 dolayısıyla bu da G= A-B değerine eşit olacaktır.

- S1,S0=11 değerinde Y çıkışı 1 olur. Buradaki 1'in kaç bit olacağı merak edilebilir. A ve B biti kaç bitse çıkıştaki 1 sayısı da o kadar bittir. A'nın tüm bitlerini 1 ile topladığınızda A daki datanın eğer Cin=0'sa 1 eksiğini aldığınızı farkedeceksiniz. Eğer Cin=1 ise mantıksal olarak A-1+1=A olacaktır.

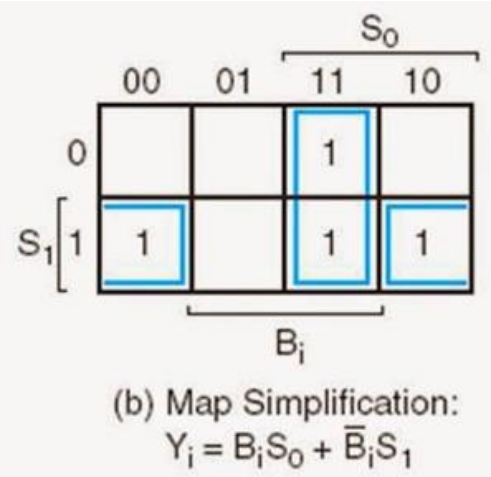
Not: Burada işimize yarayacak olan en kullanışlı sonuçlar;

- G=A (transfer)
- G=A+1 (increment)
- G=A+B (toplama)
- G=A+B'+1=A-B (çıkarma)
- G=A-1 (decrement)

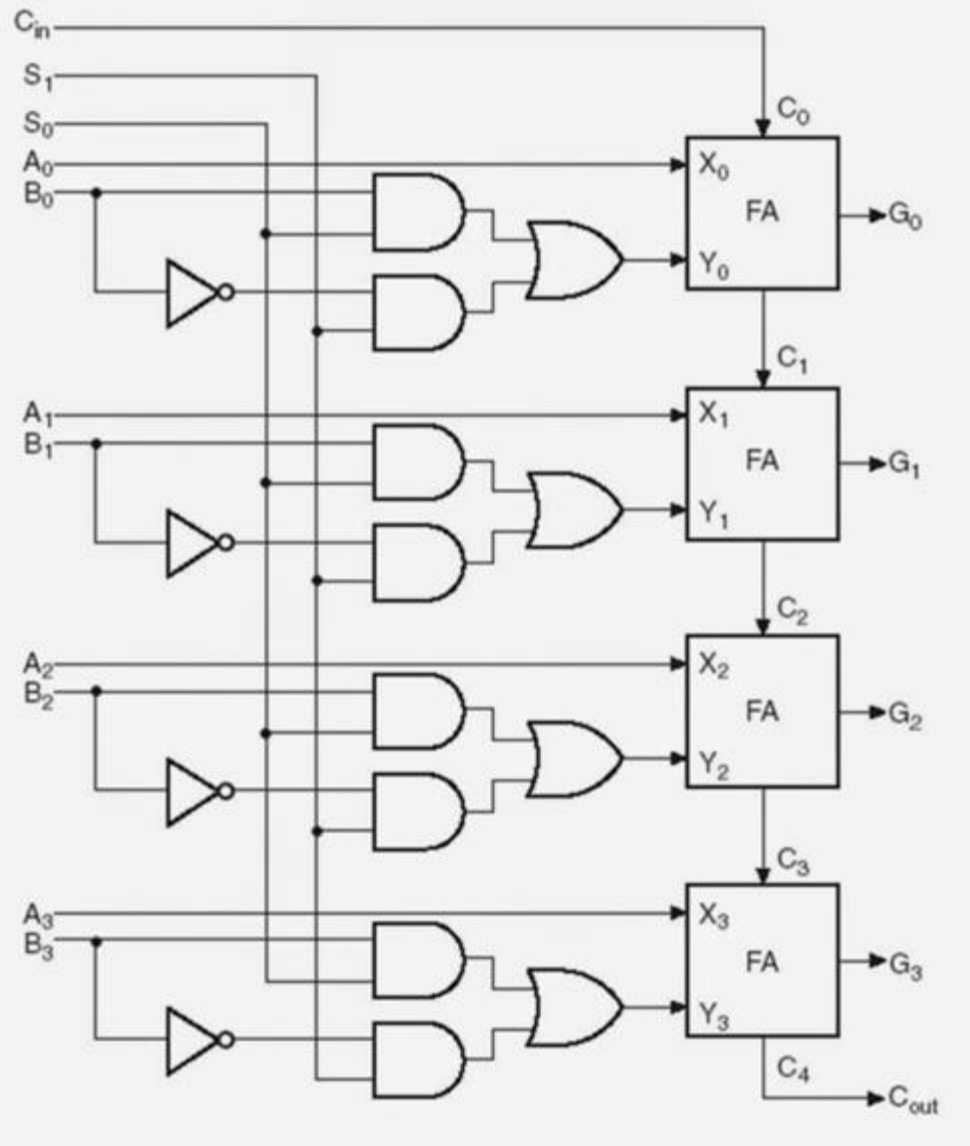
Buradaki ifadeler 4 to 1 MUX kullanarak yapılabilir. Logic kapılarla çözüm elde edilebilir. Doğruluk tablosundaki Bi, B datasının herhangi bir bitidir.

Inputs			Output	
S_1	S_0	B_i	Y_i	
0	0	0	0	$Y_i = 0$
0	0	1	0	
0	1	0	0	$Y_i = B_i$
0	1	1	1	
1	0	0	1	$Y_i = \overline{B}_i$
1	0	1	0	
1	1	0	1	$Y_i = 1$
1	1	1	1	

Bu tablodan Yi çıkışı için mantıkal ifade çıkarmak istersek Karnough diyagramı kullanılır.



Gerekli sadeleştirme yapıldıktan sonra Yi= BiS0+Bi'S1 bulunur. B nin 4 bitlik bir datadan oluştuğunu varsayarsak logic devre şu halde olacaktır.



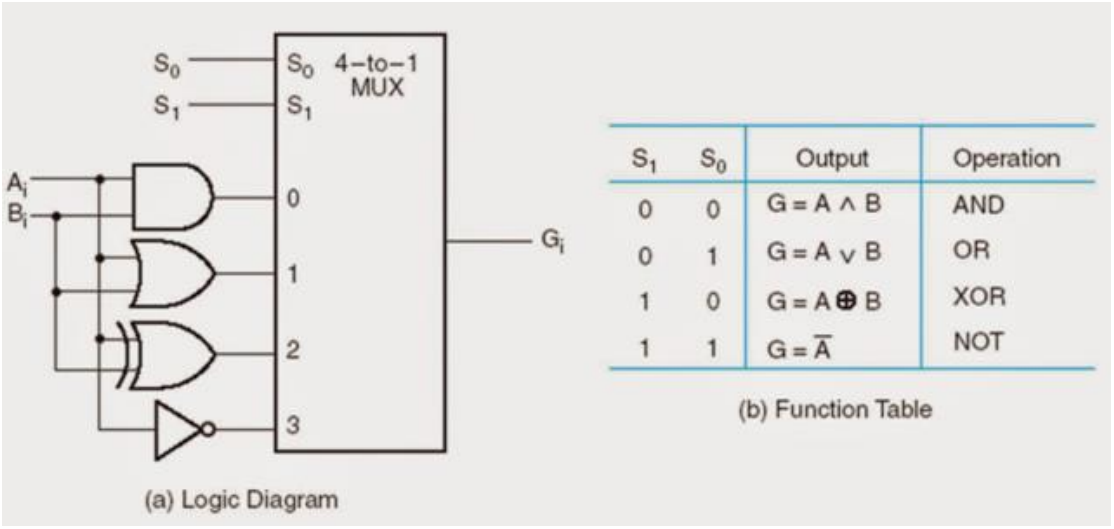
Burada bit sayısı istenilen kadar arttırılabilir. Böylece ALU nun aritmetik kısmını tasarlanmış olunur.

1.2. Logic devre

Bu devrede kullanabilecek bir çok logic ifade olabilir. Ancak çok sık kulanılanlardan devreyi oluşturulması gerekir. Genelde en sık kullanılan logic ifadeler,

- AND
- OR
- XOR
- NOT

Bu logic ifadeleri seçime bağlı şekilde çıkışa verebilecek bir devre tasarlanılması gerekiyor. Bunun için 4 to 1 MUX kullanılır. Aşağıda bu devrenin gerçeklenmiş şekli bulunmaktadır.



(a) Logic Diagram

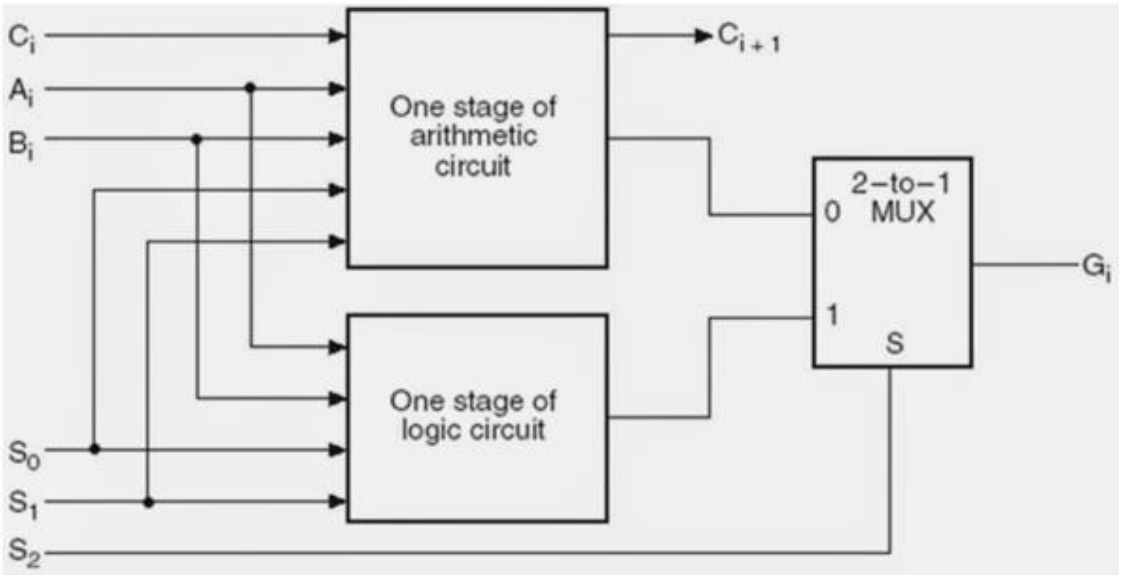
S_1	S_0	Output	Operation
0	0	$G = A \wedge B$	AND
0	1	$G = A \vee B$	OR
1	0	$G = A \oplus B$	XOR
1	1	$G = \overline{A}$	NOT

(b) Function Table

Logic devre de tamamlandığında aritmetik devre ile birleştirip ALU oluşturulur.

1.3. Aritmetik Logic unit

Aritmetik devre ile logic devre birleştirilirken S2 girişi yardımıyla istenildiğinde logic devre, istenildiğinde aritmetik devre seçimi bir mux yardımı ile yapılabilir.



Devrede görüldüğü gibi S2=0 olduğunda aritmetik, S2=1 olduğunda logic işlem aktif olacaktır. Burada dikkat edilmesi gereken önemli bir nokta aritmetik ve logic devrenin S0 ve S1 girişlerinin ortak olarak kullanılmasıdır. S2 seçimine göre yalnız bir devrede aktif olabilceği için bunun hiç bir zararı yoktur. Hatta devreyi basitleştirir. Sonuç olarak ALU' nun içerisinde 8 aritmetik 4 logic işlem yapabilme imkanı bulunmaktadır. Yukarıda verile değerlere göre oluşturulan ALU'nun doğruluk tablosu aşağıdaki gibi olacaktır.

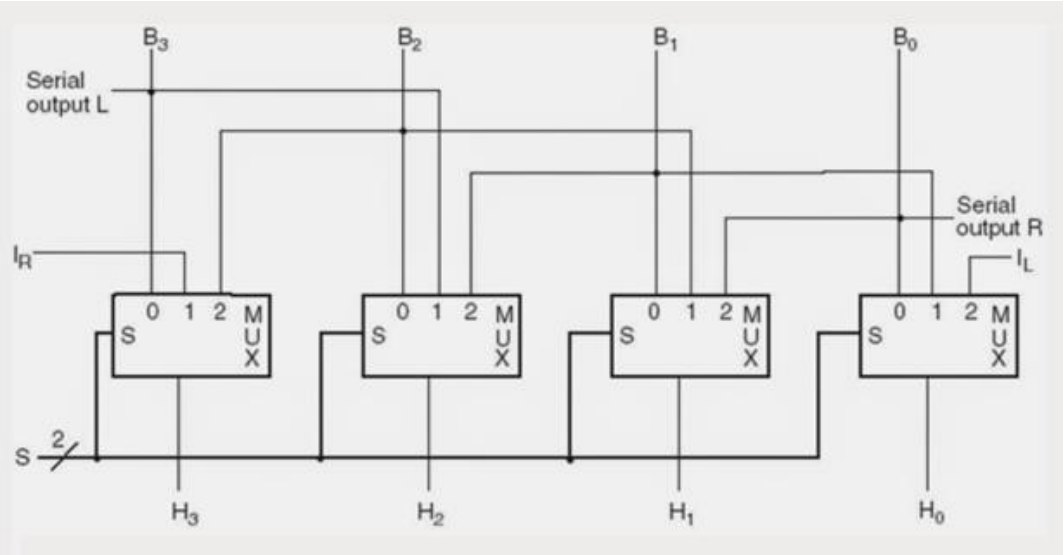
Operation Select					
S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	$G = A$	Transfer A
0	0	0	1	$G = A + 1$	Increment A
0	0	1	0	$G = A + B$	Addition
0	0	1	1	$G = A + \underline{B} + 1$	Add with carry input of 1
0	1	0	0	$G = A + \underline{B}$	A plus 1's complement of B
0	1	0	1	$G = A + \underline{B} + 1$	Subtraction
0	1	1	0	$G = A - 1$	Decrement A
0	1	1	1	$G = A$	Transfer A
1	0	0	X	$G = A \wedge B$	AND
1	0	1	X	$G = A \vee B$	OR
1	1	0	X	$G = \underline{A} \oplus B$	XOR
1	1	1	X	$G = \underline{A}$	NOT (1's complement)

Şimdi de datapath içindeki ALU'nun sağ tarafında bulunan shifter kısmı tasarlanacak.

2. Mantıksal Devreler

2.1. Shifter

Datapath devresinde Shifter'a sadece B select kısmından gelen bilgi giriş yapar. Bu nedenle iki farklı datayı bu datapath içinde kaydırma(shift) işlemine tabi tutulamaz. Şimdi 4 bitlik bir shifter yapısını inceleyelim,



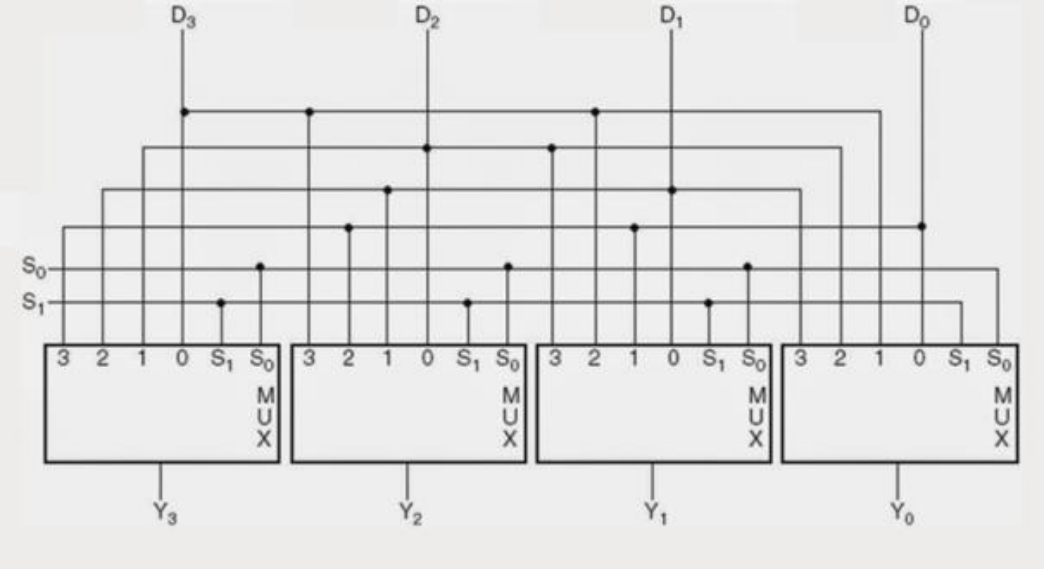
Şekilde görüleceği gibi Shifter, Clock pulse gereği duyulmadan Mux'lar ile kurulmuştur. Bundan dolayı datapath içinde bir shift işlemi yapıp Bus H yoluna(datapath resminde görülen) iletip yükleme yapılacak registera iletmek için bir Clock pulse yeterli olcaktır. şimdi şekildeki 2 bitlik S girişine göre hangi shift işlemlerinin yapılacağını tayin edelim. Burada belirtilmeli ki, hangi 2 bitlik S değerine göre hangi shift işleminin yapılacağı keyfi bir biçimde sıralanır.

- S=00 iken hiç kaydırma yapmadan Bus H yoluna direk aktarılır ki bunun işimize ne kadar yarayacağı tartışma konusu olabilir.
- S=01 iken Right-shift işlemi yapılır. Örneğin B registerındaki data 1010 olsun bunun Bus H ye aktarılmış şekli Ir101 olacaktır. Burada Ir binary bir değerdir eğer Ir=0 ise 0101 ; eğer Ir=1 ise 1101 şeklinde Bus H'ya aktarılacaktır
- S=10 iken Left-shift işlemi yapılır. Yine benzer şekilde bu defa IL değerine bağlı olmak üzere bir bit sola kaydırılarak data Bus H'a aktarılır.

Bu Shifter tipinde bir clock evresinde yalnızca bir bit kaydırma işlemi yapılabilir. Datapath uygulamalarında 1 clock evresinde 1 pozisyon kaydırmadan daha fazlasına ihtiyaç olduğunda Barrel shifter kullanılır.

2.2. Barrel Shifter

4 bitlik Barrel shifter yapısı:



Devremiz yine kombinasyonel bir devre olduğu için istenilen registera datanın kaydırılıp yükleme işlemi yapılması sadece bir clock pulse süresinde gerçekleşecektir.

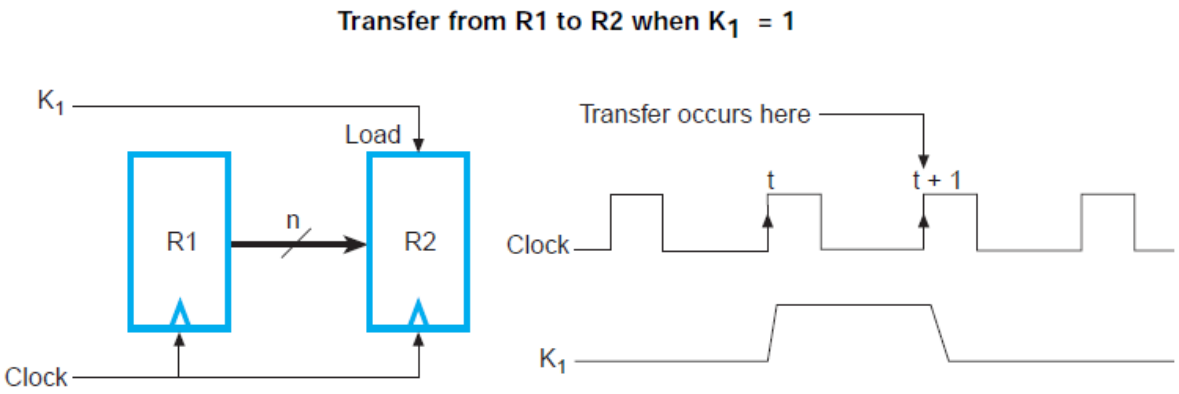
Barrel shifter ın doğruluk tablosu:

Select		Output				
S_1	S_0	Y_3	Y_2	Y_1	Y_0	Operation
0	0	D_3	D_2	D_1	D_0	No rotation
0	1	D_2	D_1	D_0	D_3	Rotate one position
1	0	D_1	D_0	D_3	D_2	Rotate two positions
1	1	D_0	D_3	D_2	D_1	Rotate three positions

S1,S0=10 durumunda ve S1,S0=11 durumunda kaydırma işlemleri tek seferde yapılmaktadır. Önceki Shifter'da bunun yapılması için bir bit kaydırıldıktan sonra tekrar shiftera sokup bir bit daha ve ihtiyaç varsa bir daha bir daha shifter'a sokulması gerekirdi. Ancak Barrel shifter'ın avantajı bu işlemleri tek seferde yapabiliyor olması. Ancak bunun da dezavantajı var. Şekli incelersek aslında tam bir shift işlemi yapılmıyor. Önceki shifter'da kaydırılan datanın boş kalan yerine 0'mı 1 mi kayacağımız IR ve IL girişleri ile belirlenebiliyordu. Barrel shifter'da yapılan işlem ise aslında döndürme(rotation) işlemi örneğin sağa doğru 1 pozisyon döndürme işlemi yaptığımızda en sağda kaybolan bit data kaydırıldıktan sonra en soluna yazılarak döndürme işlemi yapılıyor. Yine iki ve üç pozisyan kaydırma işlemi de buna benzer şekilde çalışıyor. Şekildeki yollar incelenirse daha rahat kavranabilir.

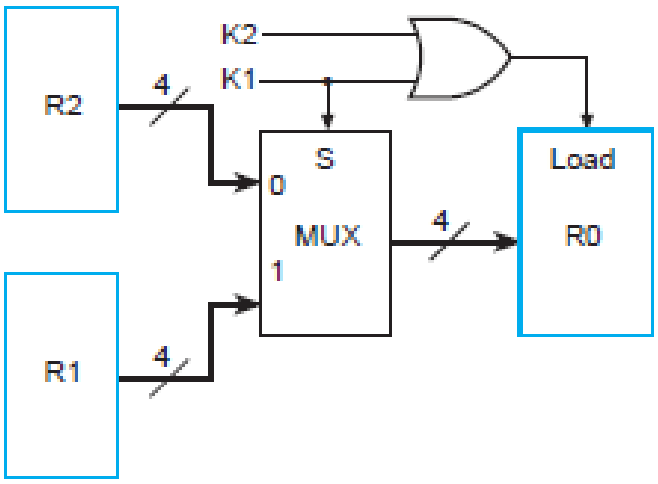
Not: İki shifter tipinde de kaç bit varsa o bit sayısında Mux'a ihtiyaç duyduğumuza dikkat edin.

2.3. R1 Register'ından R2 Register'ına Transfer

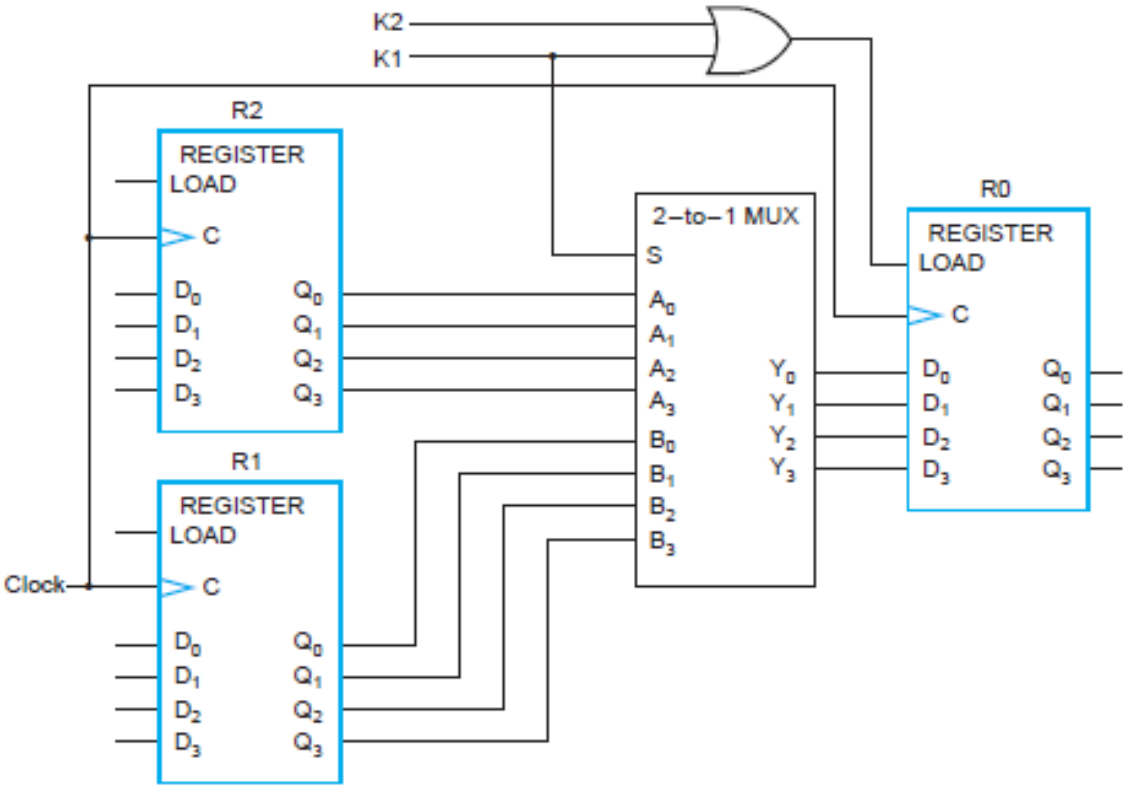


2.4. İki Register Arasında Seçim Yapmak İçin Çoklayıcıların Kullanımı

Blok Diyagram:

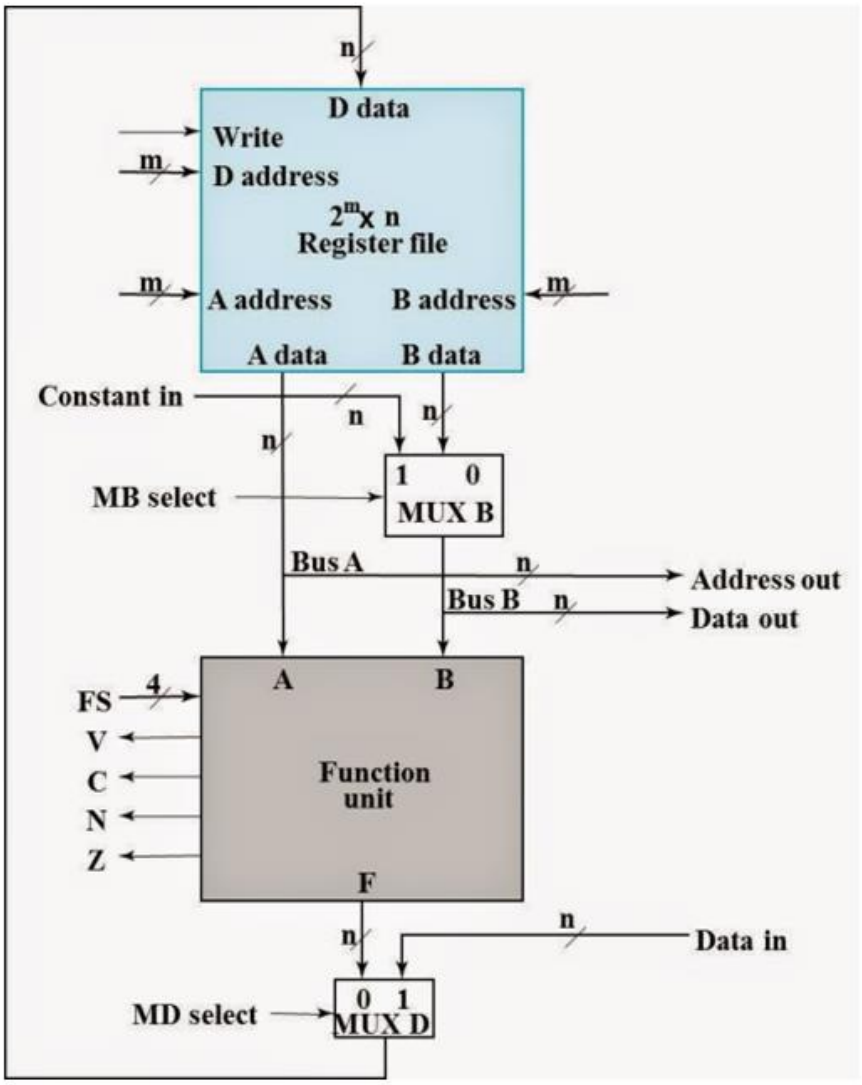


Ayrıntılı mantık devresi:



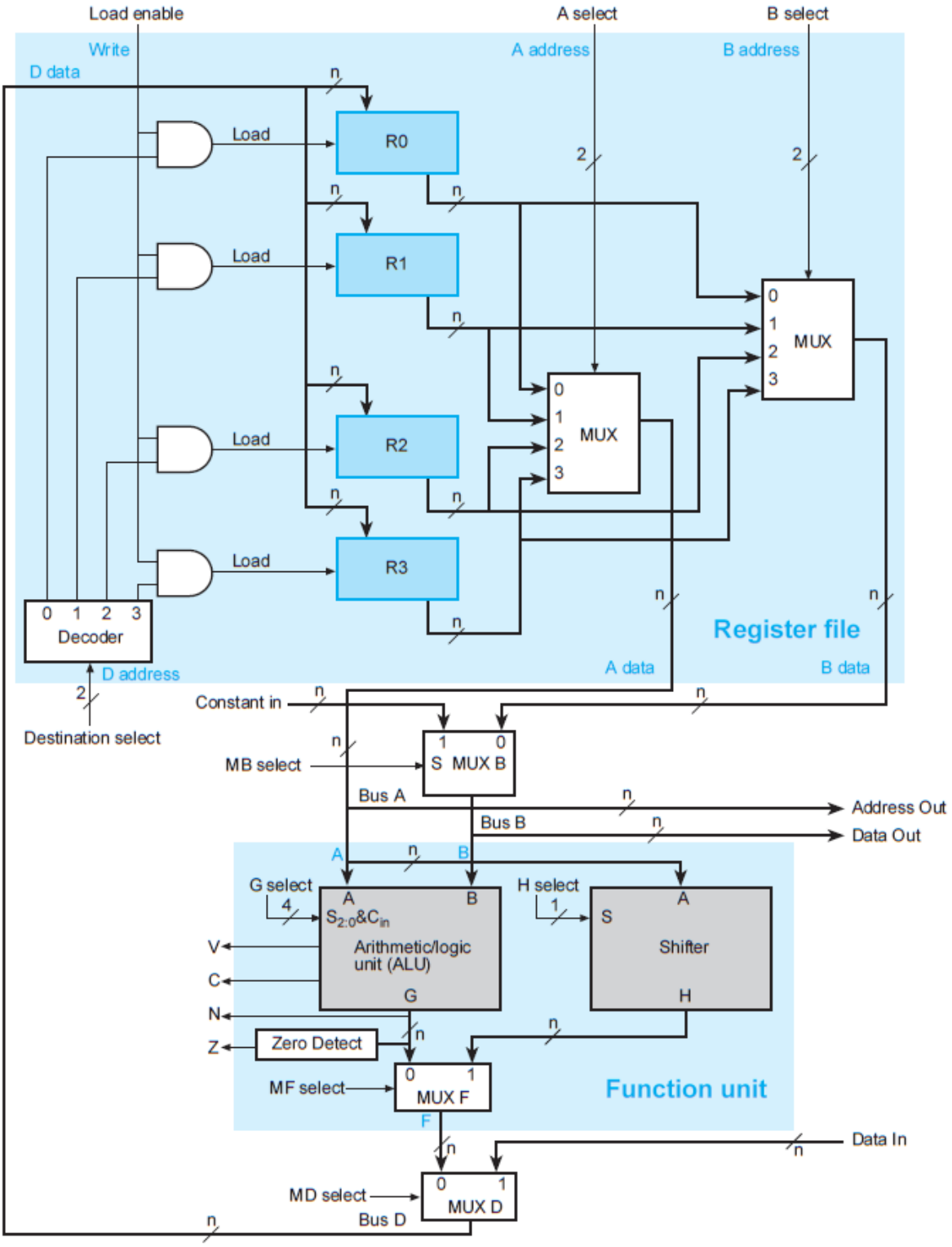
3. Datapath'in İrdelenmesi

4 bitlik register'larla işlem yapılsa da gerçek bilgisayarlarda 32 bit veya daha fazlası kullanılmaktadır. Tabi böylesine büyük yapılar datapath tasarlama da farklı tekniklerin kullanılmasını gerektirecektir. İlk datapath resmindeki register file ve fuction unit'i bir blok halinde gösterirsek,



- V çıkış sinyali bize işlam sonrasında taşma durumu (overflow) varsa 1 değerini alarak taşma durumu olduğunu işaret eder
- C çıkış sinyali Cout'u temsil eder. Yani işlem sonrasında elde kalan değer olarak tanımlanır.
- N çıkış sinyali Function unit'ten çıkan işlem sonucunun negatif olması durumunda 1 olan pozitif olması durumunda 0 değerini veren bir sinyaldir
- Z çıkış sinyali işlem sonucunun 0 (zero) olduğu durumda 1 değerini veren bir çıkış sinyalidir.

Şimdi ilk datapath devresini yeniden irdeleyelim



Yukarıda anlattıklarımızı da göz önüne alarak şöyle bir doğruluk tablosu oluşturabiliriz

FS(3:0)	MF Select	G Select(3:0)	H Select(3:0)	Microoperation
0000	0	0000	XX	$F \leftarrow A$
0001	0	0001	XX	$F \leftarrow A + 1$
0010	0	0010	XX	$F \leftarrow A + B$
0011	0	0011	XX	$F \leftarrow A + B + 1$
0100	0	0100	XX	$F \leftarrow A + \overline{B}$
0101	0	0101	XX	$F \leftarrow A + \overline{B} + 1$
0110	0	0110	XX	$F \leftarrow A - 1$
0111	0	0111	XX	$F \leftarrow A$
1000	0	1X00	XX	$F \leftarrow A \wedge B$
1001	0	1X01	XX	$F \leftarrow A \vee B$
1010	0	1X10	XX	$F \leftarrow A \oplus B$
1011	0	1X11	XX	$F \leftarrow \overline{A}$
1100	1	XXXX	00	$F \leftarrow B$
1101	1	XXXX	01	$F \leftarrow \text{sr } B$
1110	1	XXXX	10	$F \leftarrow \text{sl } B$

Buradaki FS select, MF select, G select, ve H select den oluşmaktadır. FS select için çıkarımlarımız şöyle;

- FS select in en soldaki 2 biti 1 olduğunda MF select=1 olduğu gözüküyor.
- Eğer MF select=0 ise Fs selecti oluşturan kodlar aynen G select deki kodlardır.
- MF select=1 olduğunda FS selectin en soldaki 2 bitini 1 olduğunu belirtmiştik diğer 2 biti ise H select tarafından kullanılır

Burada gerekli boolean işlemleri yapıldığında

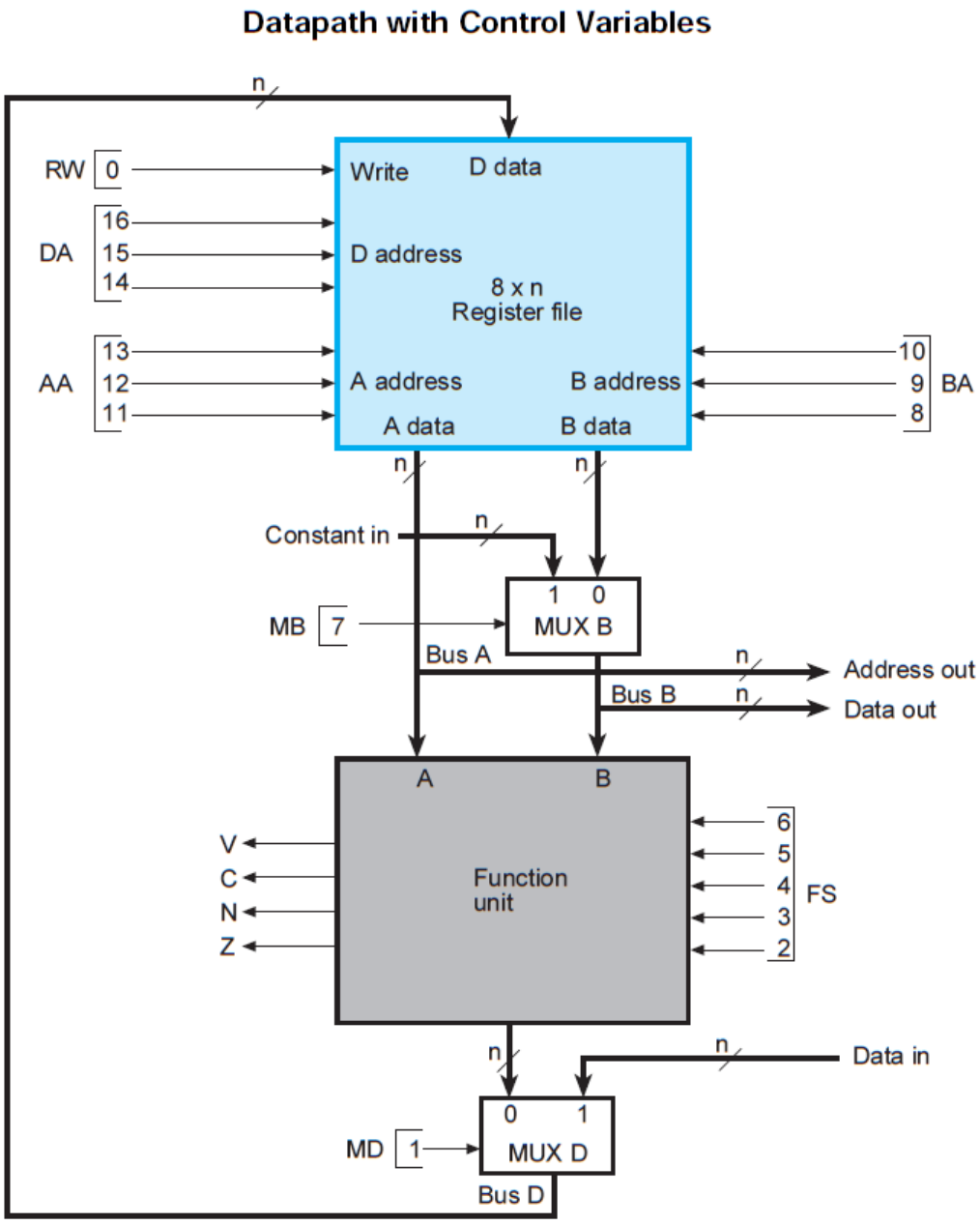
- MF=F3.F2
- G3=F3
- G2=F2
- G1=F1
- G0=F0
- H1=F1
- H0=F0

olduğunu test edebilirsiniz.

Not: Sadece 4 bitlik FS select ile MF, G ve H selectleri kontrol ederek function unit bloğunun ne yapacağına artık karar verebiliriz.

3.1. Control Word

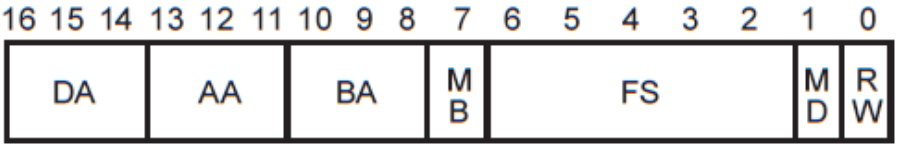
Şimdi datapath'i yeniden ele alalım,



(a) Block Diagram

Yukarıdaki datapath yapısında bulunan register file içerisinde R0'dan R7'ye olmak üzere 8 adet register bulunmaktadır. Bu register'lara 3'e 8 decoderle ulaşılabilir (ilk datapath resminde bu decoderler görülmektedir).

Toplama bakılacak olunursa resimdeki numaralardan belli olacağı üzere 16 adet binary giriş var bunlar yerlerine göre numaralandırılmıştır. Bu girişlerin hepsini bir araya getirerek *Control Word* oluşturulur. *Control Word* aşağıdaki biçimde gösterilir.



(b) Control word

Control word'deki bitlerin numaralarıyla datapath resminde ifade ettikleri kısımların aynı yer olduğunu fakedeceksiniz.

- DA , datanın yazılacağı registerın adresini gireceğimiz decoder'ın kodu
- AA, Bus A'ya konulacak datanın hangi register'dan geleceğini seçecek Mux'un Select bitleri
- BA, Bus B'ya konulacak datanın hangi register'dan geleceğini seçecek Mux'un Select bitleri
- MB, Bus A'dan gelen data ile herhangi bir sabitin mi yoksa B'den gelen data ile mi işleme sokulacağı karar veren Mux'un select biti
- FS, daha önce tabloda boolen ifade ile bulmuş olduğumuz FS Select girişleri
- MD, function unit'ten mi yoksa dışarıdan gelen bir bilginin mi hedefdeki register'a yazılacağını belirleyen Mux'un Select girişi
- RW, ise register dosyalarına yazma veya okuma işlemlerinin hangisinin yapılacağına karar verir.

Şimdi bu control word yardımıyla hangi işlemleri hangi kodlarla yazılacağına bakalım.

Encoding of Control Word for the Datapath

DA, AA, BA		MB		FS		MD		RW	
Function	**Code**	**Function**	**Code**	**Function**	**Code**	**Function**	**Code**	**Function**	**Code**
$R0$	000	Register	0	$F = A$	00000	Function	0	No write	0
$R1$	001	Constant	1	$F = A + 1$	00001	Data In	1	Write	1
$R2$	010			$F = A + B$	00010				
$R3$	011			$F = A + B + 1$	00011				
$R4$	100			$F = A + \overline{B}$	00100				
$R5$	101			$F = A + \overline{B} + 1$	00101				
$R6$	110			$F = A - 1$	00110				
$R7$	111			$F = A$	00111				
				$F = A \wedge B$	01000				
				$F = A \vee B$	01010				
				$F = A \oplus B$	01100				
				$F = \overline{A}$	01110				
				$F = \text{sr } A$	10000				
				$F = \text{sl } A$	10001				

Control Word Example

100 100 001 0 1001 0 1

***R*1** **00100000**
***R*4** **01010100**

$R4 \leftarrow R4 \vee R1 \leftarrow 01110100$ 't'

101 101 001 0 1010 0 1

***R*1** **00100000**
***R*5** **01001100**

$R5 \leftarrow R5 \oplus R1 \leftarrow 01101100$ 'l'

001 001 000 0 1011 0 1

$R1 \leftarrow \overline{R1} \leftarrow 1101111$

001 001 000 0 0001 0 1

$R1 \leftarrow R1+1 \leftarrow 11100000$

110 110 001 0 0101 0 1

***R*6** **01000001**

$R6 \leftarrow R6+\overline{R1}+1 \leftarrow 0110000$ 'a'

111 111 001 0 0101 0 1

***R*7** **01001001**

$R7 \leftarrow R7+\overline{R1}+1 \leftarrow 0110100$ 'i'

011 111 000 0 0000 0 1

$R1 \leftarrow R7 \leftarrow 01101001$ 'i'

Örnek:

R1 ← R2 + R3' + 1 işlemini yapabilmek için gereken *Control Word*'e yazılacak kodları bulun.

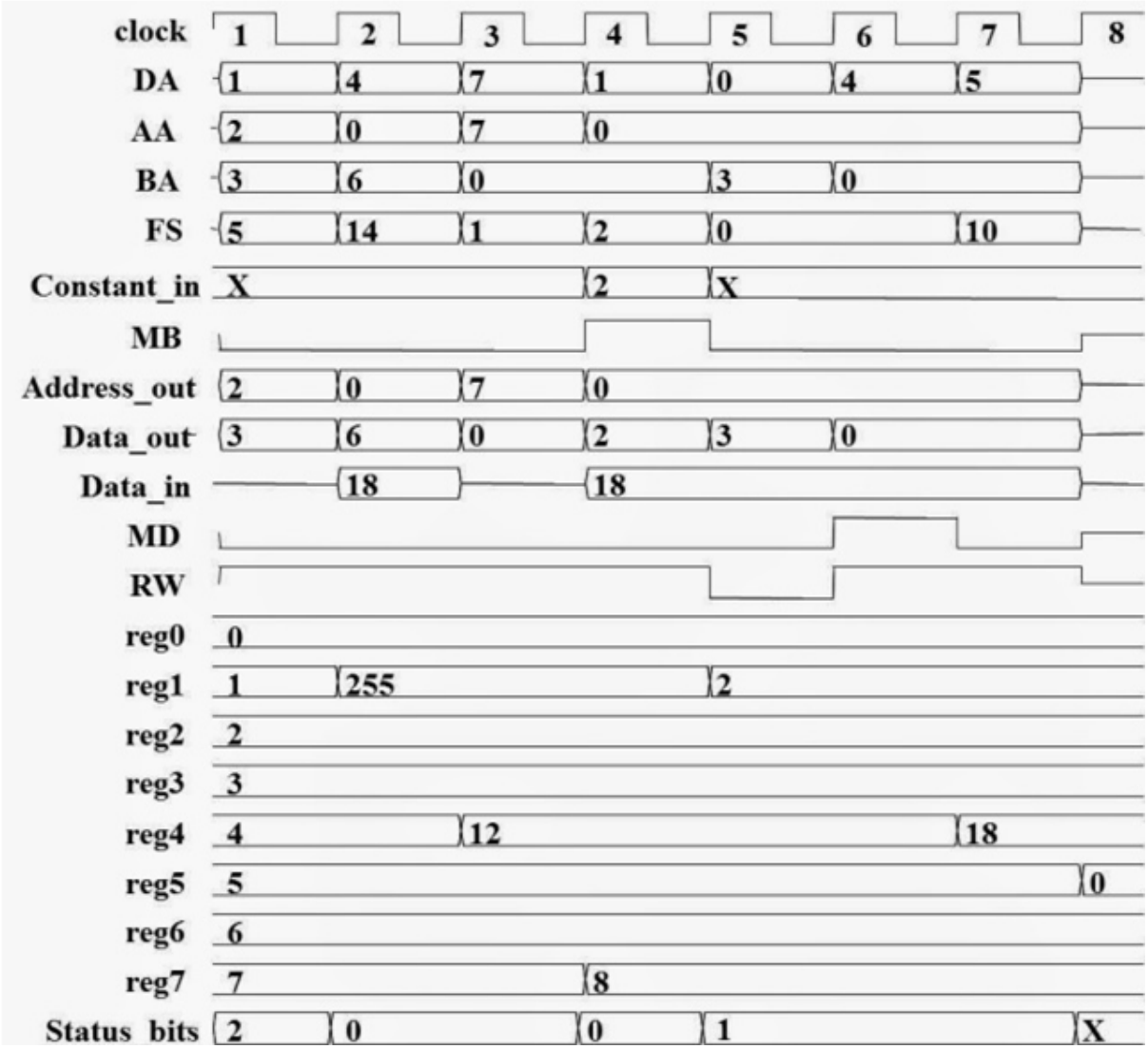
- Yazılacak adres R1 olduğu için Destination Register(hedef Register) R1 olacaktır. Tablodan R1'in koduna bakıldığında, DA=001 olacağı belirlenir.
- Bus A dan gelecek data R2 dir. Yine R2'nin tablodaki koduna bakıldığında, AA=010 olacaktır.
- Bus B den gelecek data da R3 dür ve R3'ün tablodaki koduna bakıldığında, BA=011 elde edilir.
- Daha sonra dışarıdan herhangi bir sabitle işlem olmadığı için MB=0.
- Yapılacak işlem R1 ← R2 + R3' + 1 dir. Function Code'dan bu işlemin hangi kodda sağlandığına bakılır, FS=0101
- Dışarıdan herhangi bir data girişi olmadığı için, MD=0
- Yazma işlemi yapacağımız için, RW=1 olacaktır.

Önce işlemler kod olarak değil de isim olarak verilir ve bu tablonun altına da o isimlerin gerektirdiği kodları benzer bir tabloda verilimiş olur.

Micro-Operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + 1$	Function	Write
$R4 \leftarrow$ sl R6	$R4$	—	$R6$	Register	$F =$ sl B	Function	Write
$R7 \leftarrow R7 + 1$	$R7$	$R7$	—	Register	$F = A + 1$	Function	Write
$R1 \leftarrow R0 + 2$	$R1$	$R0$	—	Constant	$F = A + B$	Function	Write
Data out $\leftarrow R3$	——		$R3$	Register	—	—	No Write
$R4 \leftarrow$ Data in	$R4$	——		—	—	Data in	Write
$R5 \leftarrow 0$	$R5$	$R0$	$R0$	Register	$F = A \oplus B$	Function	Write

Micro-Operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	001	010	011	0	0101	0	1
$R4 \leftarrow$ sl R6	100	XXX	110	0	1110	0	1
$R7 \leftarrow R7 + 1$	111	111	XXX	0	0001	0	1
$R1 \leftarrow R0 + 2$	001	000	XXX	1	0010	0	1
Data out $\leftarrow R3$	XXX	XXX	011	0	XXXX	X	0
$R4 \leftarrow$ Data in	100	XXX	XXX	X	XXXX	1	1
$R5 \leftarrow 0$	101	000	000	0	1010	0	1

3.2. Datapath'in çalışmasının zaman diagramı üzerinde gösterimi



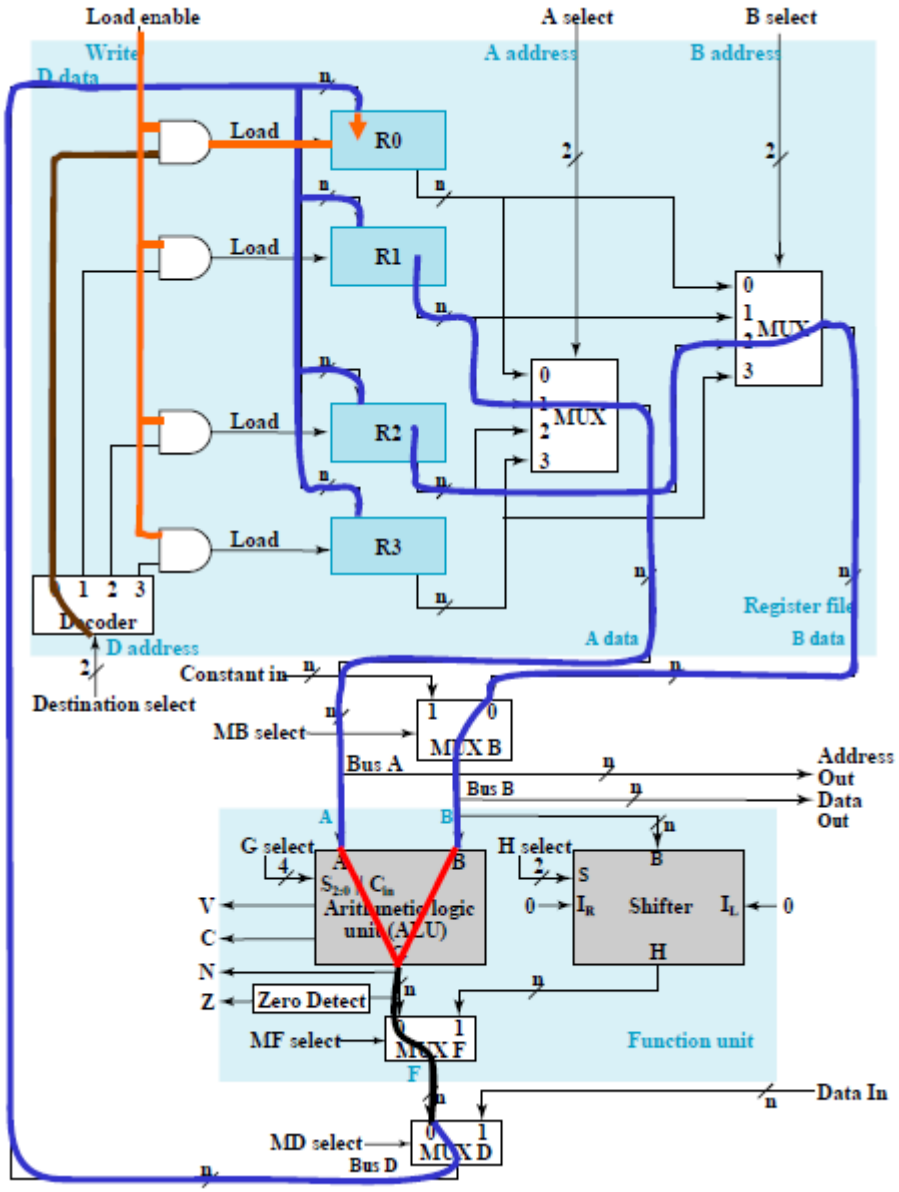
Not: Bu gösterimde değerler binary olarak değilde desimal yazılmıştır

Datapath Örneği:

- Dört paralel register yüklemesi
- İki mux tabanlı register seçici
- Register Hedef kod çözücü
- Harici sabit giriş için Mux B
- Harici adres ve veri çıkışlarına sahip A ve B busları
- Çıkış seçimi için Mux F'li ALU ve Shifter
- Harici veri girişi için Mux D
- V, C, N, Z durum bitlerini oluşturmak için mantık

Mikro İşlem Gerçekleştirme:

R0 ← R1 + R2



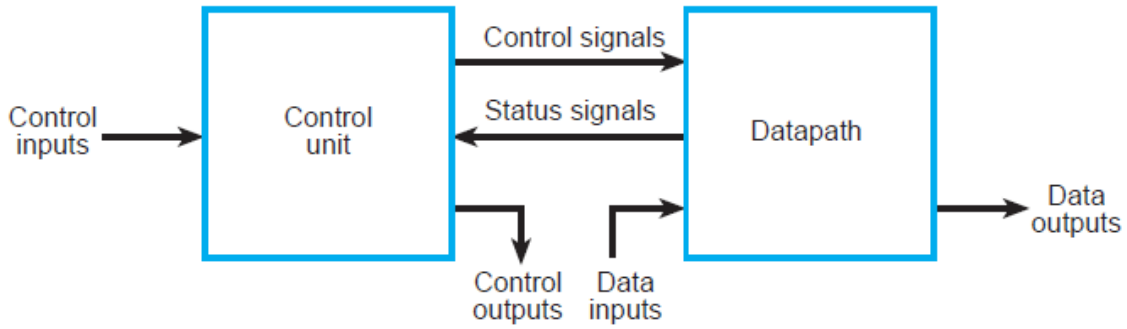
- R1 içeriğini Bus A'ya yerleştirmek için A'ya 01 uygulanır.
- R2'nin içeriğini B verilerine yerleştirmek için B'ye 10 uygulanır ve B verilerini B veriyoluna yerleştirmek için MB'ye 0 uygulanır.
- Eklemeyi gerçekleştirmek için 0010'dan G'ye seçilir, G = Bus A + Bus B
- G değerini BUS D üzerine yerleştirmek için MF seçimine 0 ve MD seçimine 0 uygulanır.
- R0'a Yük girişini etkinleştirmek için Hedefe 00 uygulanır.
- Yük girişini R0'a 1'e zorlamak için Yük Etkinleştir'e 1 uygulanır, böylece R0, saat darbesine yüklenir (gösterilmemiştir)
- Genel mikro işlem 1 saat döngüsü gerektirir.

3.3. Control Unit

Genel olarak bir senkron digital sistemde tüm register'ların zamanlaması bir temel clock generatöründen sağlanır. Bu sistemde clock darbeleri tüm flip-flop'lara ve register'lara uygulanır. Register'ların veya flip-flop'ların durumunu her clock darbesi sırasında değişime uğratmayı engellemek için ise register ve flip-flop'lara bir Enable, Disable ve Load girişleri eklenir. Eğer Enable aktifse Register'dan data okunabilir, Eğer Load aktifse register'a data yüklemesi yapılabilir.

Sistem tasarımlarında, digital sistemlerdeki control unit yapısı iki ayrı türden oluşur, Bunlardan birincisi programlanabilen sistem diğeri ise programlanamayan sistem. Programlanabilir sistemlerde giriş kısımları sıralı bilgilerden oluşur. Programlanamayan sistemler de ise register'da toplanan bilgiyle alakası olmadan tasarlanmış bir devre halinde gerekli işlemler yapılır.

Interaction between Datapath and Control Unit



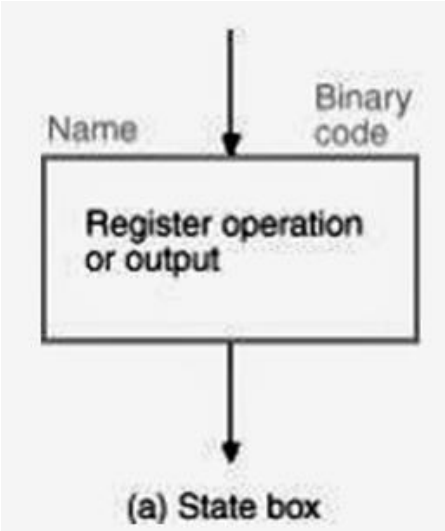
4. Algoritmic State Machines (ASM)

Bir dijital devrelerin tasarımında en zorlayıcı kısmı donaın bağantılarının formüle edilme sürecidir. Bir akış diyagramı sıralı sistem adımları ve karar mekanizmalarını belirlemek için uygun bir seçenektir.

ASM chart üç temel elementten oluşur:

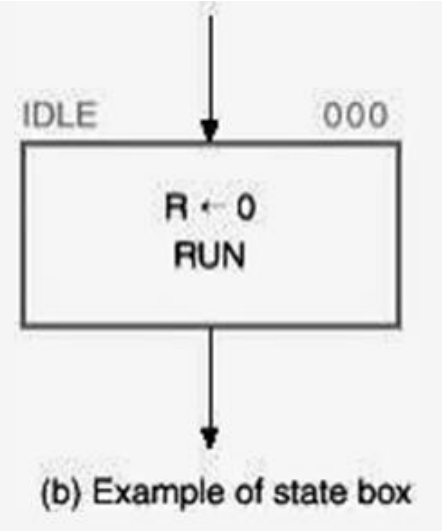
- 1. İfade Kutusu
- 2. Karar Kutusu
- 3. Koşullu çıktı kutusu

İfade kutusu şu şekilde gösterilir:



Control unit bu state box içerisinde iken, state box register transfer işlemleri veya aktif olacak çıkış sinyallerini içerir. Dolaylı olarak bir çıkış sinyalinin aktivasyonu demek , sinyalin değerinin 1 olması anlamına gelir Bir state box ın ismi sol üst köşesinde yer alır. Bir State box ın binary kodu ise o state in sağ üst köşesine yazılır.

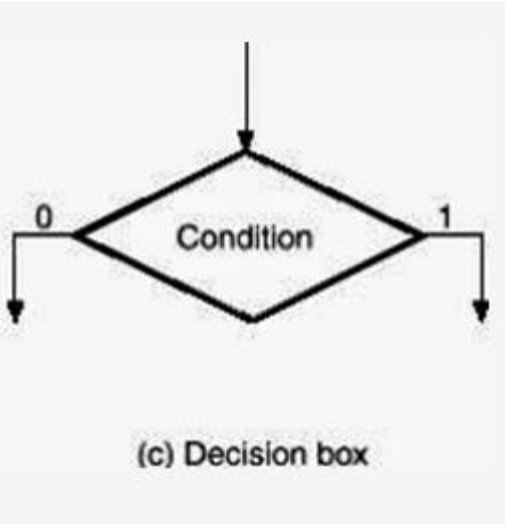
Aşağıda bir örnek state box ı inceleyelim;



(b) Example of state box

state box ın ismi IDLE binary kodu 000 ve state box un içinde R registırını sıfırlayan bir register işlemi ve bir RUN ifadesi yer alıyor. Control birimi IDLE da olduğu müddetçe meydana gelen her clock pulse ı sırasında R registerındaki tüm bitler 0 lanacak Buradaki RUN ifadesi ise şu anlama gelir: Control birimi IDLE da olduğu zaman zarfında output RUN sinyali 1 olmalıdır. RUN göründüğü her state box ta 1 değeri alır görünmediği her state box da 0 değerini alır.

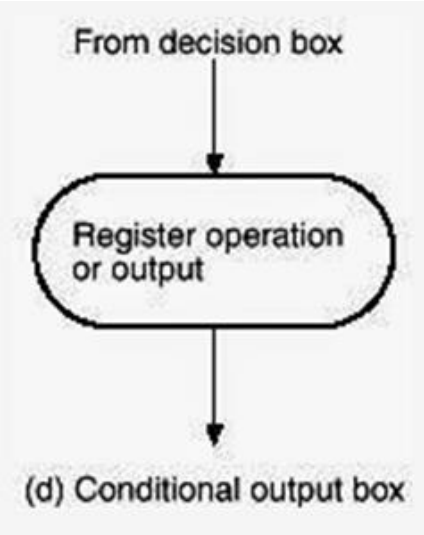
- Decision box, yani karar kutusu diye adlandıracağımız yapı ise şu şekildedir;



(c) Decision box

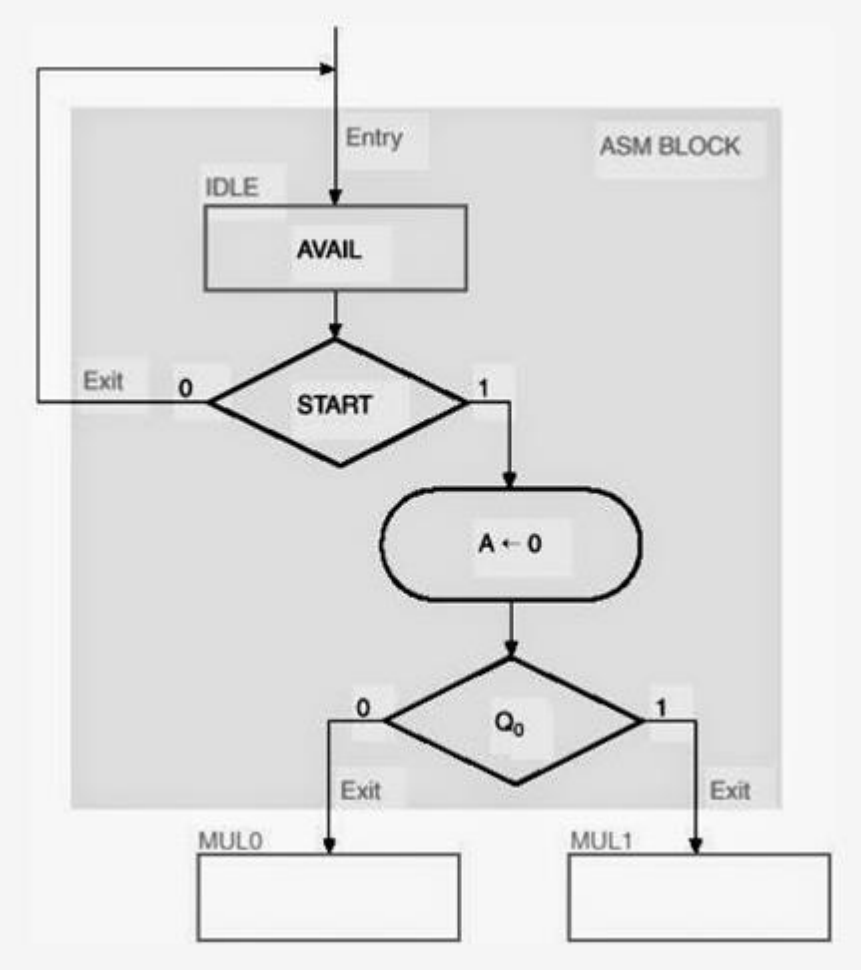
Decision box ın 1 girişi ve 2 adet çıkışı bulunmaktadır. Giriş şartı tek bir binary değer veya tek bir boolean ifadesi olabilir. Her seferinde yalnızca 1 çıkış yolu aktif olabilir. Gelen Boolean ifadesi doğru ise 1 numaralı çıkış ucundan aksi takdirde 0 numaralı çıkış ucundan yoluna devam eder.

- Conditional output box ise şu şekilde gösterlir



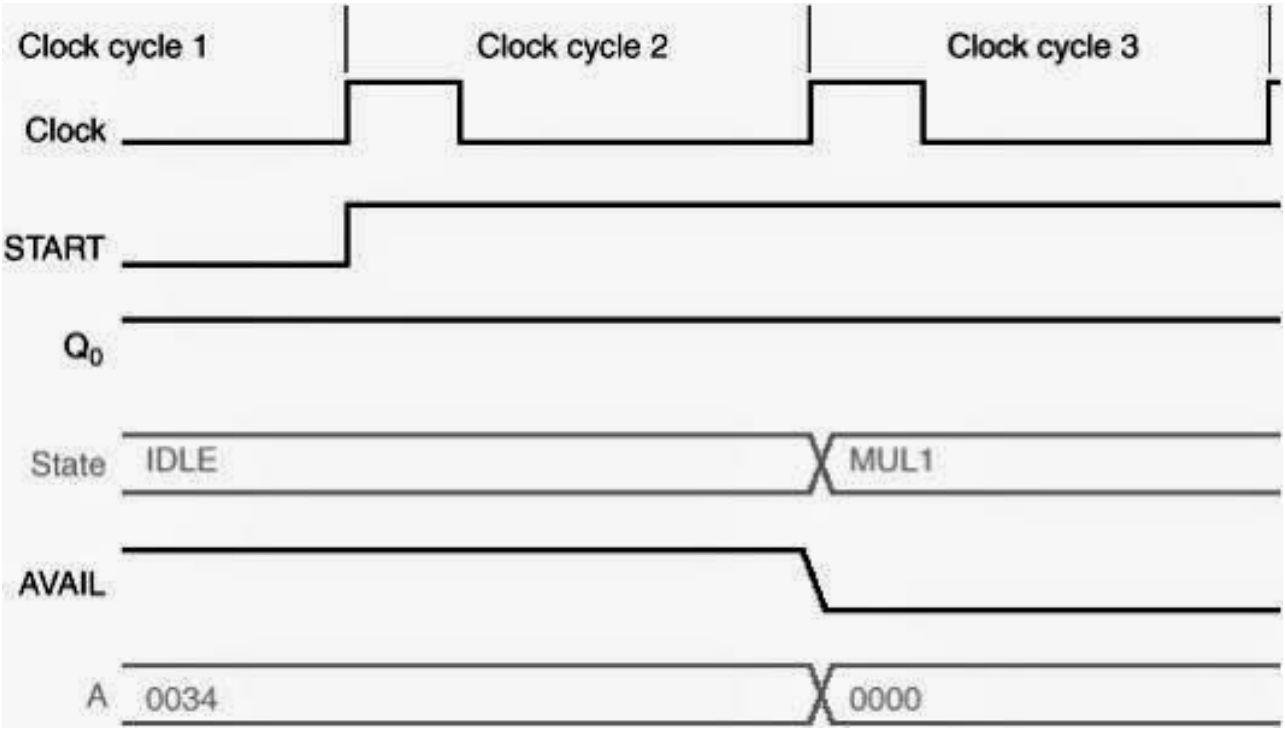
Conditional output box ın giriş yolu muhakkak bir veya daha fazla decision box dan geçmiş bulunması gerekir. Geri kalan özellikleri state box gibidir.

Bir örnek üzerinde inceleme yapalım;



Şekil üzerinden durumları inceleyelim;
state IDLE durumunda iken AVAIL çıkışı 1 olacaktır. Eğer start 0 olursa next state tekrar IDLE olacaktır ve start değişkeni 0 da kaldığı müddetçe sonsuz bir döngü oluşacaktır. Start 1 olduğunda bir clock pulse inde A registerındaki datalar 0 olcaktır. Daha sonra Q0 ın durumuna göre state ya MUL0 ya da MUL1 olacaktır.

şimdi durumun zamanlama durumuna bir göz atalım;



Şekli incelersek;
ilk gelen clock pulse sırasında aynı anda start girişi de 1 olmuş ancak state conditional output box a gidip A registerındaki 0034 olan değeri 0000 yapmamıştır. Bunun nedeni yükselen kenar muhabbetidir. start girişinin 1 olduğu sırada clock pulse de 1 olmuş gibi gözükür ama aslında şöyle bir durum vardır: Clock un değeri aniden 1 e çıkamaz eğer yakından analiz edilcek olursa bir eğimle yukarı çıkar ancak çok kısa bir süre içinde olduğu için genelde bu tür şekillerde düz bir çizgiyle gösterilir. Yani sonuç olarak sistem start ın 1 olduğu anda clock pulse ini 1 olarak algılmayacağından hiçbir değişiklik yapmaz ancak 2. Clock pulse ine kadar.. 2. clock pulse süresinde, ilk clock pulse sırasında sıfırlanamayan A registerı burada 0 lanacak ve Q0 değeri en baştan beri 1 değerinde olduğu için sistem bunu algılayacak ve direk MUL1 state box ına geçiş yapacaktır. Bir de state artık IDLE değilde MUL1 olduğu ve MUL1 de AVAIL ifadesi bulunmadığı için AVAIL in değeri yine bu clock pulse sırasında 0 olacaktır. Burada dikkatleri üzerine çekmek istediğimiz bir sonuç var. Şekle dikkat edilirse İkinci clock pulse inin yükselen kenarında start artık 1 olarak algılandığı için state conditional state boxa ulaşıp A registerını 0 laması ile state durumunun MUL1 ulaşması eşzamanlı olarak gerçekleşir.

Binary çarpma işlemi

Şimdi çarpma işleminin ASM chartını oluşturarak çarpma işlemini yapan devreyi/devreleri tasarlayalım. Bunun için önce binary tabanda çarpma işleminin nasıl işlediğine bir göz atalım.

Örneğin;
23 ile 19 sayısını yani 10111 ile 10011 sayısını çarpalım. genel ilerleyiş adımlarımız şöyledir

- İlk işleme başlarken tüm bitleri 0 olan bir sayıyı yazarız. Buradaki 0 ların sayısı çarpılan sayının bit sayısı kadar olması işimizi kolaylaştıracaktır.
- Daha sonra çarpan sayıya bakılır
    - En sağdan ilk biti 1 ise çarpılan bir önceki sayı ile toplanır(ilk bit için bu sayı en başta yazdığımız 0 sayısıdır) ve elde edilen sayı toplama işleminden sonra bir bit sağa kaydırılır.
    - En sağdan ilk biti 0 ise sadece bir bit sağa kaydırılır
- Daha sonra çarpan sayının tüm bitleri kontrol edilinceye kadar bu işlem devam ettirilir.

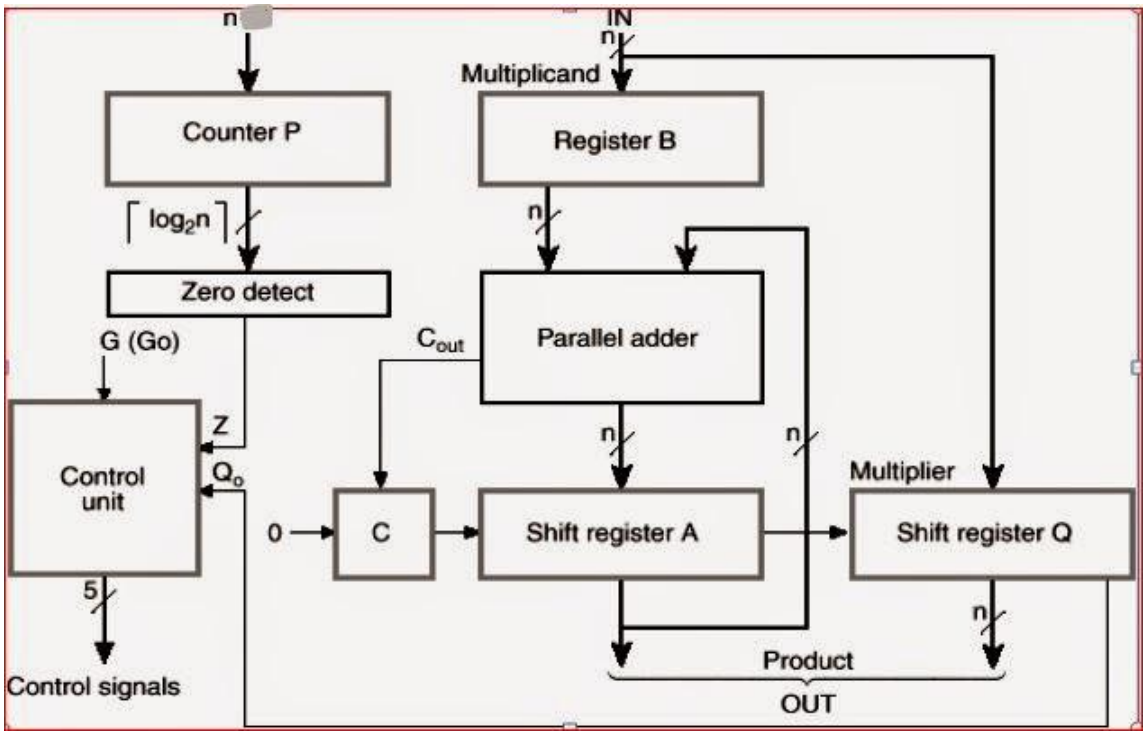
Aşağıda bu işlemin yapılış aşamaları bulunmaktadır.

23	10111	Multiplicand
19	10011	Multiplier
	00000	Initial partial product
	10111	Add multiplicand, since multiplier bit is 1
	10111	Partial product after add and before shift
	010111	Partial product after shift
	10111	Add multiplicand, since multiplier bit is 1
	1000101	Partial product after add and before shift[a]
	1000101	Partial product after shift
	01000101	Partial product after shift
	001000101	Partial product after shift
	10111	Add multiplicand, since multiplier bit is 1
	110110101	Partial product after add and before shift
437	0110110101	Product after final shift

a. Note that overflow temporarily occurred.

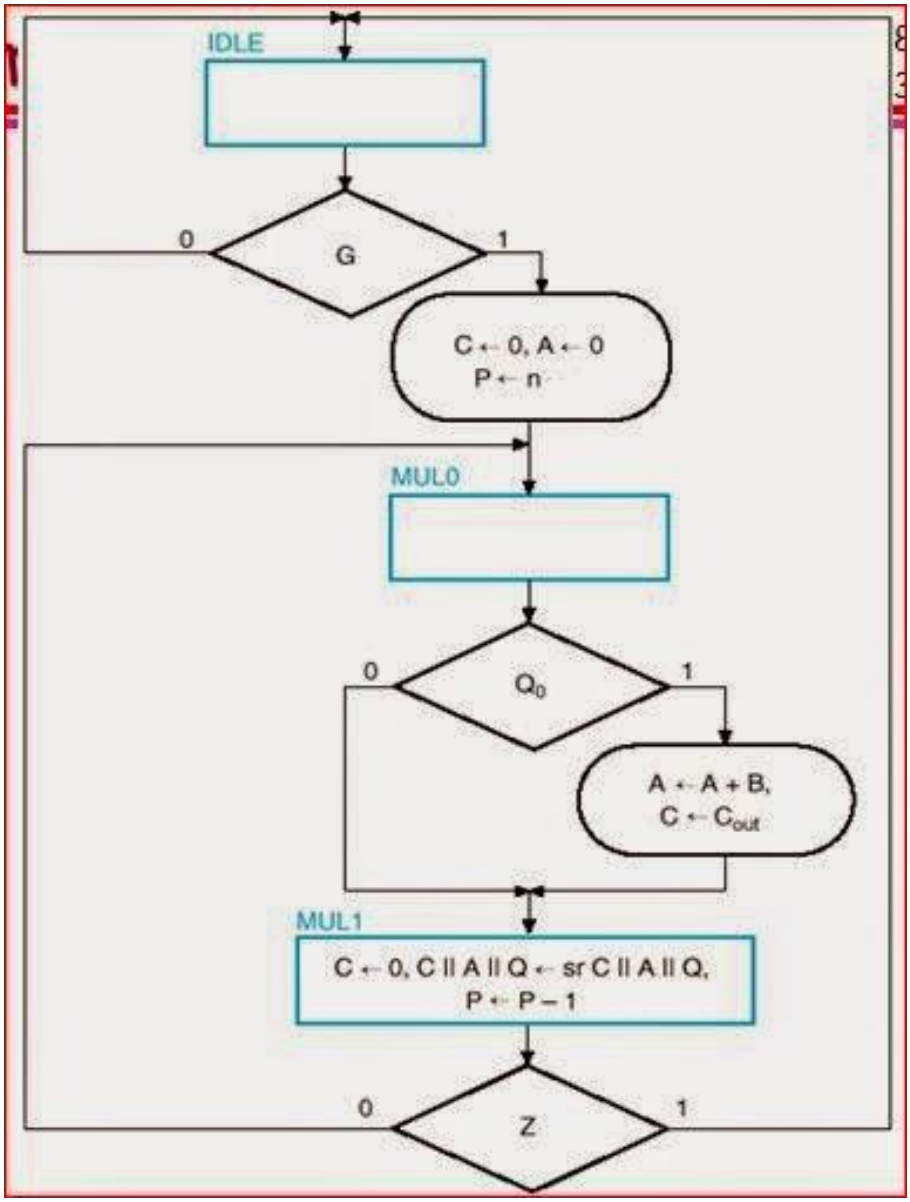
bu işlem sırasında geçici olarak bir taşma(overflow) durumu olsa da toplama işlemi ile kaydırma işlemi aynı anda gerçekleştiği için bir sorun oluşturmamaktadır.

Çarpma işleminin genel algoritmasını gördüğümüze göre bu algoritmayı sağlayan block diagrama geçiş yapabiliriz.



Çarpılan sayı Register B ye yüklenmiş ve çarpan sayı register Q da yüklenmiş durumda. İlk değerimiz 0 ise Register A içinde yüklü durumdadır. Sonucumuz Register A ve Register Q ile içindeki değerler birleştirilerek okunacaktır. Şekilde bulunan C filp-flopu ise anlık taşma durumları için kullanılmıştır. Taşma olmadığı durumda içinde sıfır barındırır. Counter P ise çarpan sayının her biti çarpıldıktan sonra register Q dan alınan bir sinyaldan sonra n den 0 sayarak 0 ulaştığında zero detect sayesinde 0 ı yakalayıp control unit e ilettiğinde çarpma işleminin durmasını sağlar. Aksi takdirde çarpma işlemi durmayacak ve bize sağlıklı sonuçlar üretmeyecektir. Çarpma işlemi control unit e bağlı durumda olan G(go) selecti 1 olduğunda başlayacaktır. Aksi takdirde sistem ilk durumunda kalacak ve işlem başlamayacaktır.

Şimdi de bu işlemi ASM chart üzerinde nasıl ifade edebiliriz ona bakalım. Aşağıdaki şekilde binary çarpma işlemini yapan ASM chart bulunmaktadır.



- Durum(state)—durum için state terimini kullnacağım— IDLE da iken G(go) select 0 olduğu müddetçe state sürekli IDLE da kalacaktır.
- State IDLE da iken G select 1 olursa conditional output box da gösterilen işlemler yapılacaktır ve aynı zamanda state artık MUL0 da olacaktır. Conditional output box da C registerındaki ve A registerındaki datalar sıfırlanacak. Counter P de n sayısından sinyal geldikçe 0 a kadar sayacaktır.
- State MUL0 da iken Q0 selecti 0 sa hiç bir işlem olmadan state MUL1 de olacaktır. Q0 selecti 1 ise conditional output box daki işlemler yapılarak state MUL1 de olacaktır. ((Buradaki Q0 çarpan sayının en sağdaki bitidir. Hatırlayacağınız gibi çarpan sayının son biti 0 sa sadece kaydırma 1 ise çarpılan sayıyı önceki işlem sonucu ile toplayarak kaydırma işlemine tabi tutuyorduk))Ardından MUL1 de yapılacak işlemler state box ın içerisinde gösterilmiştir.

- MUL1 deki || işareti birleştirme işaretidir. sonuçta okuyacağımız sayı C,A ve Q registerında okuyacağımız değer olacağından bunları birleştirmemiz gerekecektir.
- state MUL1 de Z select e bakılır.
    - Z select, counterP de sayılan değer sıfıra ulaştığında 1 değerini alarak bize sinyal veren dedektördür. Z select 0 ı yakaladığında 1 olduğu için işlem bitmiş olacağından tekrar IDLE a eğer işlem bitmemişse yani Z select 0 ise tekrar çarpanın son bitine bakıp kaydırma ve toplama işlemi yapılan MUL0 a gidecektir.

HARDWİRED CONTROL

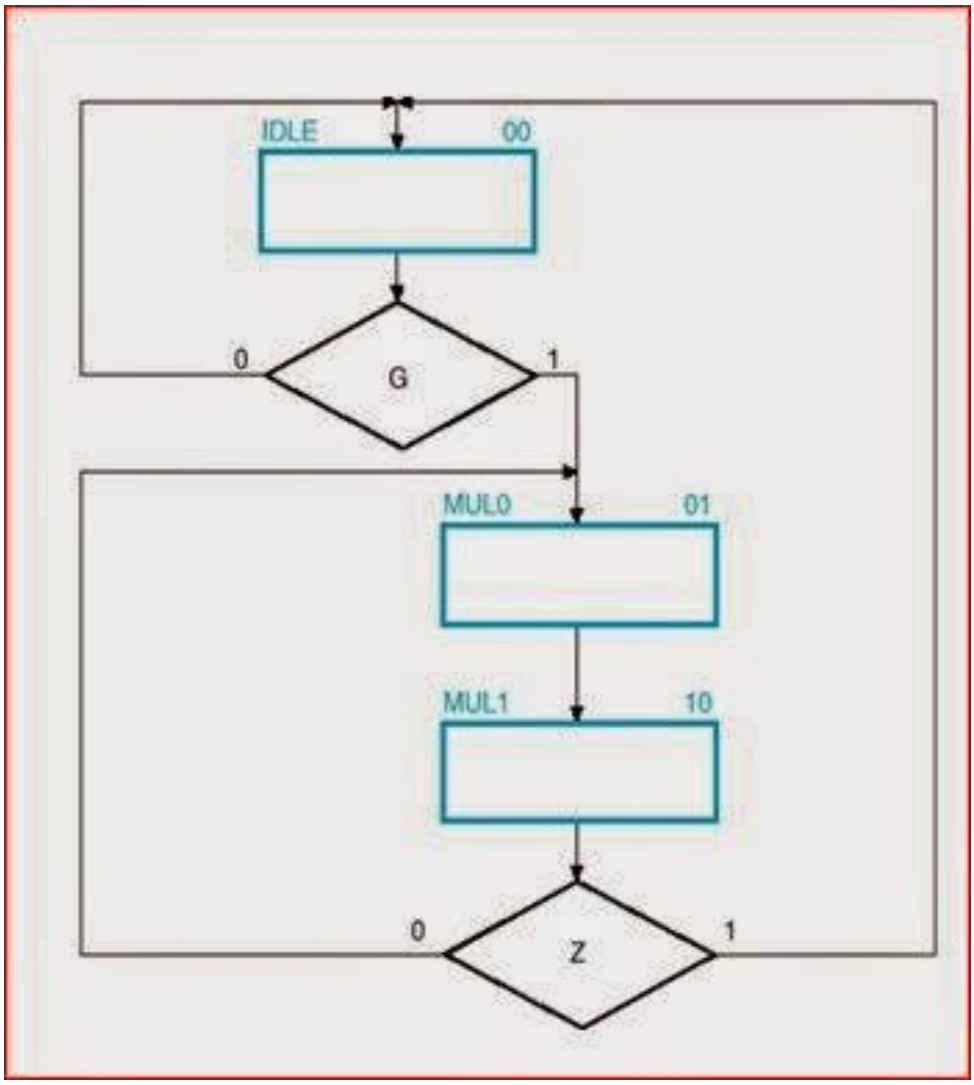
Bir control unit i gerçeklerken, iki ayrı etki düşünülmelidir. Mikroişlemlerin kontrolü ve control unit ve mikroişlemlerin sıralanması. Kontrol sinyallerini üreten parça oluşturulmalı ve ardından daha sonra ne olacağını tanımlayan parça oluşturulmalıdır.

Kontrol sinyallerini ASM chart üzerinden temellendiririz. Çarpma işlemi için gerekli olan datapath aşağıdaki tabloda gösterilmiştir

Block Diagram Module	Microoperation	Control Signal Name	Control Expression
Register A:	$A \leftarrow 0$	Initialize	$IDLE \cdot G$
	$A \leftarrow A + B$	Load	$MUL0 \cdot Q_0$
	$C \| A \| Q \leftarrow sr\ C \| A \| Q$	Shift_dec	MUL1
Register B:	$B \leftarrow IN$	Load_B	LOADB
Flip-Flop C:	$C \leftarrow 0$	Clear_C	$IDLE \cdot G + MUL1$
	$C \leftarrow C_{out}$	Load	—
Register Q:	$Q \leftarrow IN$	Load_Q	LOADQ
	$C \| A \| Q \leftarrow sr\ C \| A \| Q$	Shift_dec	—
Counter P:	$P \leftarrow n$	Initialize	—
	$P \leftarrow P$	Shift_dec	—

Bir kontrol sinyali bir veya daha fazla register ı aktive etmek için kullanılabilir. Örneğin Register A için konuşacak olursak, tabloda 3 adet mikroişlem gözükmektedir. Bunlar; Clear,Load, Shift.mikroişlemleridir. Register A daki clear(sıfırlama) mikroişlemi, C flip-flop unun sıfırlanması(clear) işlemi ve Counter P ye load(yükleme) işlemi aynı anda gerçekleştiği için bu işlemler tek bir sinyalle kontrol edebiliriz. Bu sinyali *Initialize* diye isimlendirdik. Fakat C flip-flop unun sıfırlnma işlemi ayrıca MUL1 içinde de olduğundan bunu ayırmak durumundayız. Dolayısıyla *Initialize* sinyalini A registerını sıfırlamak ve Counter P ye yükleme yapmak için kullanabiliriz. *Initialize* aktif olması için ise bir boolean ifade bulmamız gerekirse ve ASM chart dan yararlanacak olursak içinde bulunduğu state IDLE ve G 1 olduğunda aktif olacağı için IDLE.G olarak tanımlayabiliriz. Diğer ifadeleri de aynı şekilde düşünebilirsiniz.

Sadece sıralı sistemi görmek için conditional output box ları kaldırarak ASM chart ı yeniden çizecek olursak aşağıdaki şekli elde ederiz.



Q0 ın çıkışına bağlı olan conditional output box kaldırıldığı için oradaki Q0 seçiminden sonra aynı stateye gittikleri için etkisiz hale gelmesinden dolayı onu da kaldırdık.Şimdi bu devreyi gerçeklemenin 3 ayrı metodunu inceleyelim

Sıralı register ve decoder methodu

Bu metodu ancak küçük boyuttaki sıralı devreler için kullanırız. Büyük boyuttaki devreler için çok kullanışsız bir metoddur.

Son durumda binary çarpma işleminin sıralı ASM chart ın da IDLE, MUL0 ,MUL1 olmak üzere 3 adet state i ve G,Z olmak üzere 2 adet de girişi vardır. Bu ASM chart ı register ve decoder ile gerçeklemek için 2 adet flip-flopa ve 2 ye 4 decoder e ihtiyacımız var. 3 adet state olduğundan decoder in yalnız 3 adet çıkışını kullanacağız. O halde devreyi gerçeklemek

için üstteki şekilden uyarlanan state table a bakalım

Present state			Inputs		Next state		Decoder Outputs		
Name	M_1	M_0	G	Z	M_1	M_0	IDLE	MUL0	MUL1
IDLE	0	0	0	×	0	0	1	0	0
	0	0	1	×	0	1	1	0	0
MUL0	0	1	×	×	1	0	0	1	0
MUL1	1	0	×	0	0	1	0	0	1
	1	0	×	1	0	0	0	0	1
—	1	1	×	×	×	×	×	×	×

Üstteki ASM chart ı inceleyerek Present state(şimdiki durum) ve Next state(sonraki durum) un değerlerini inceleyerek test edin ve doğrulayın. Buradaki flip-floplarımız next state başlığı altındaki M1 ve M0 dır. Decoder outputlar ise o anki state hangi pozisyonda ise o state isminin değeri 1 diğerlerinin ki ise 0 olmak üzere tabloya yerleştirilmiştir. Burada kullanacağımız flip-flop D tipi flip-floptur ve bu flip flopların girişlerini tayin etmek için yukarıdaki tabloda M1 ve M0 değerlerini sadeleştirmemiz gerekecek. BUnu karnough yöntemi ile de yapabilsek de bunu kullanmadan direk tablo üzerinden de çıkarabiliriz.

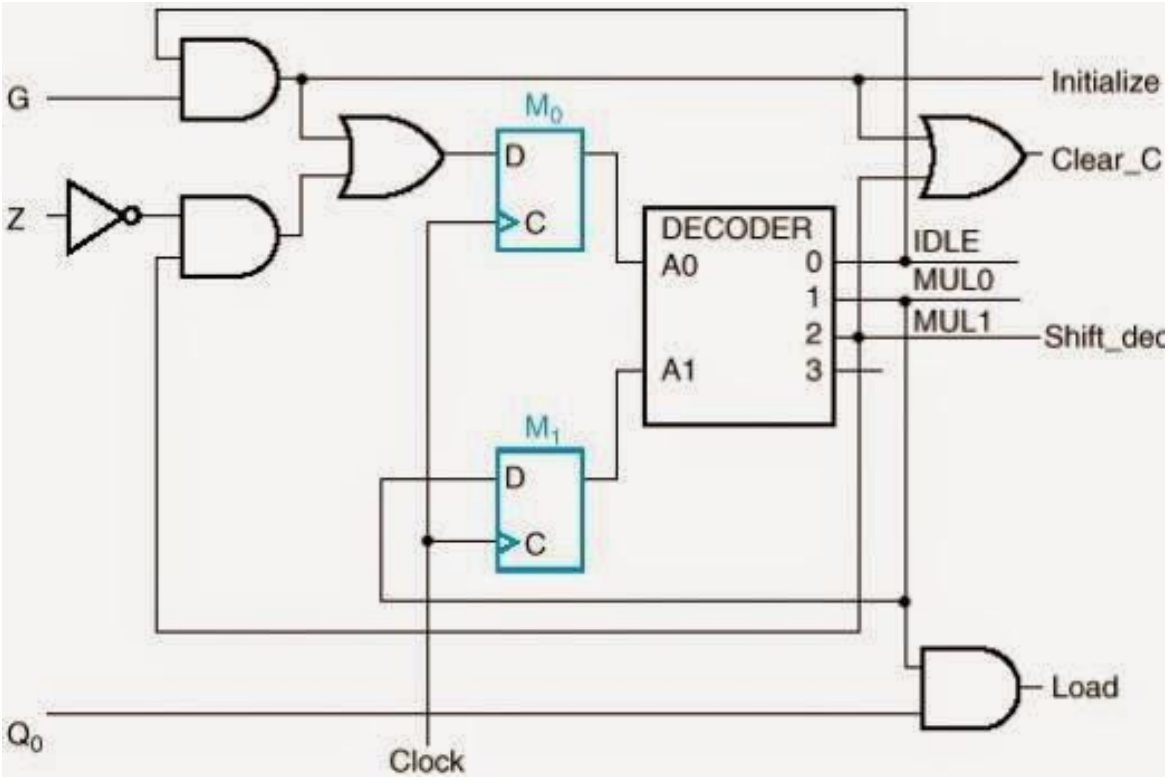
Örneğin M1 değerinin direk MUL0 değerine eşit olduğu görülüyor

- M1=MUL0

M0 değeri için ise tabloya baktığımızda

- M0=IDLE.G+ MUL1.Z’ olduğunu görebilirsiniz.

O halde artık binary çarpma işlemini bu verileri kullanarak devreye dökebiliriz. Aşağıda bu devrenin yapılmış hali verilmiştir.

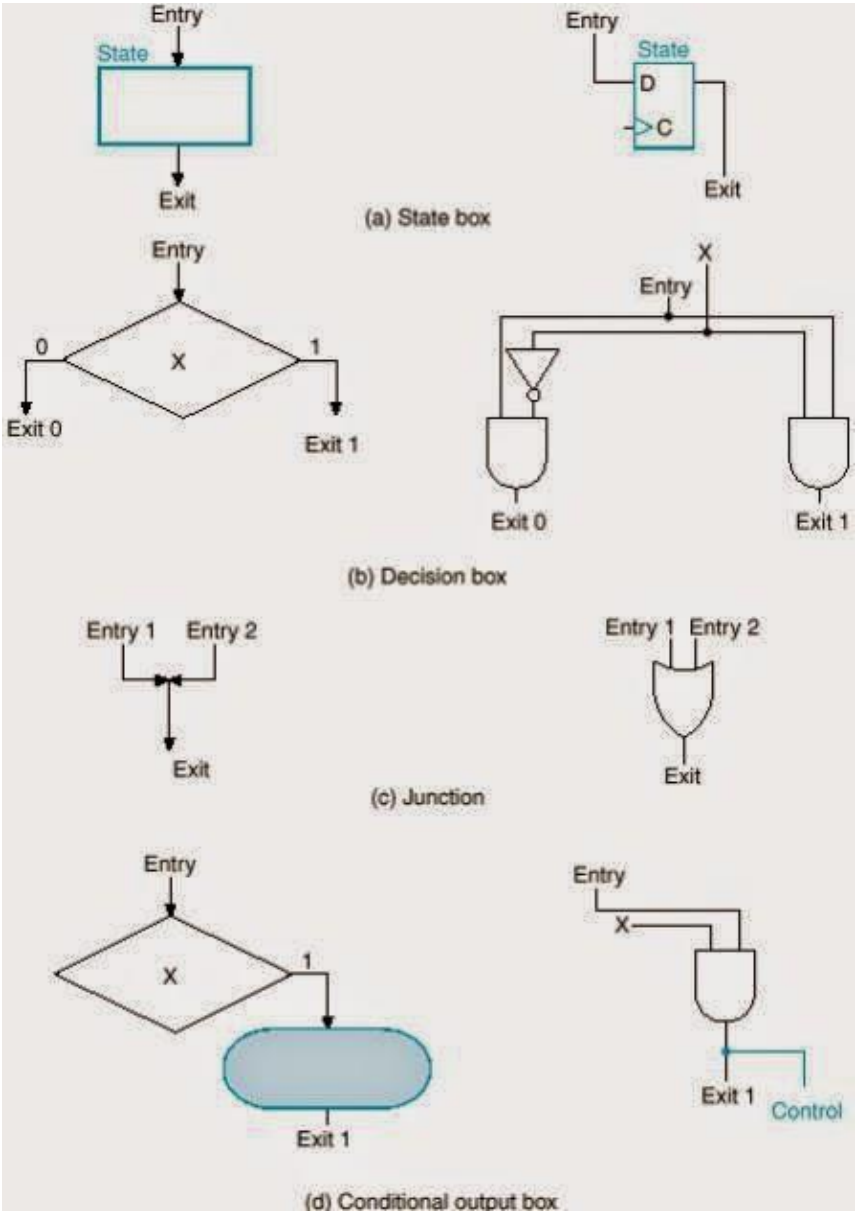


Decoder in 3 adet çıkışını control çıkışlarını oluşturmak için kullnadık. Örneğin Clear_C için önceki tablolardan hatırlayacağınız gibi IDLE.G+MUL1 ifadesini kullanmıştık ve bunu bu devrede gerçekledik. Yİne aynı şekilde tablolardan load ın durumunu da inceleyiniz. Decoder çıkışlarını aynı zamanda flip-flop un girişleri için de kullandığımıza da dikkat edin.

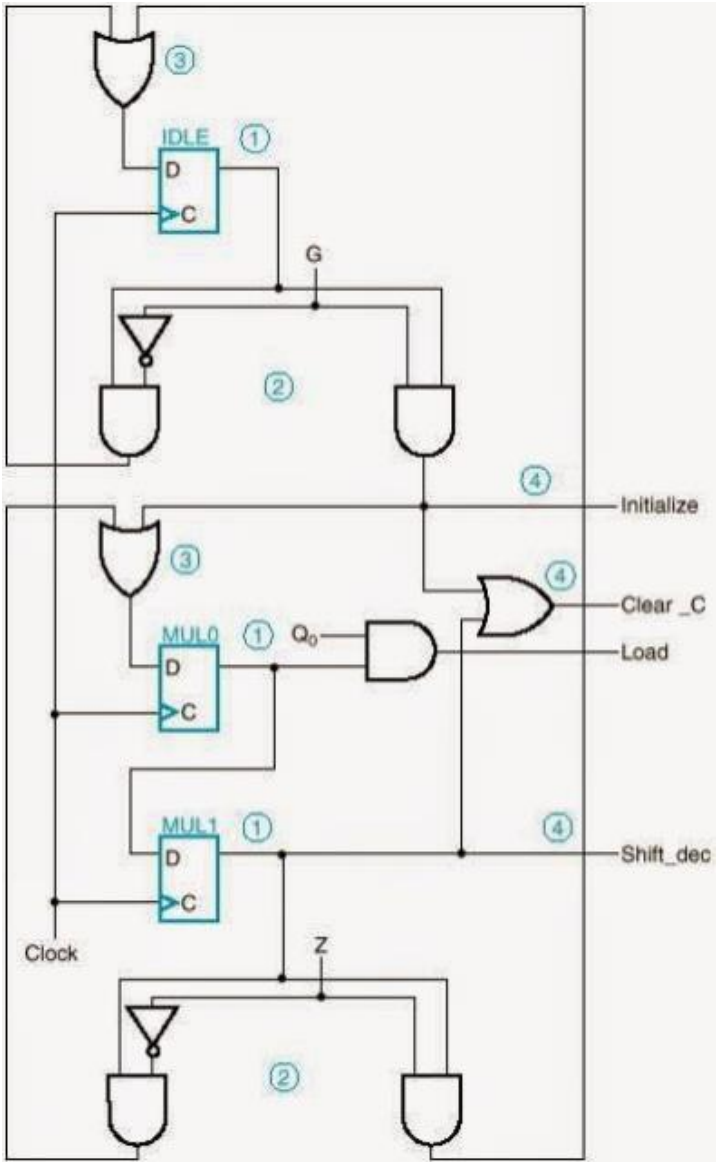
Evet artık elimizde binary düzeyde çarpma işlemine ait bir devre var. Mutlu olmanız gerekir.!!! Şimdi bir de bu devreyi başka bir metod ile çözüme kavuşturalım

Her state için bir Flip-flop yöntemi

Bu yöntem oldukça kullanışlı ve yapılması basit bir yöntemdir.Bu yöntemde belirli dönüşümler uygulayacağız. Bunlar aşağıdaki dönüşüm tablosunda verilmiştir.



Şekilde görüldüğü gibi ASM chart ta verilen her state box yerine bir d tipi flip-flop, her decision box için ise şekilde görülen devre parçası eklenecektir. Şimdi gelin bunu binary çarpma işlemi için kullandığımız ASM chart için yapalım. Aşağıda devrenin yapılmış şekli bulunmaktadır.



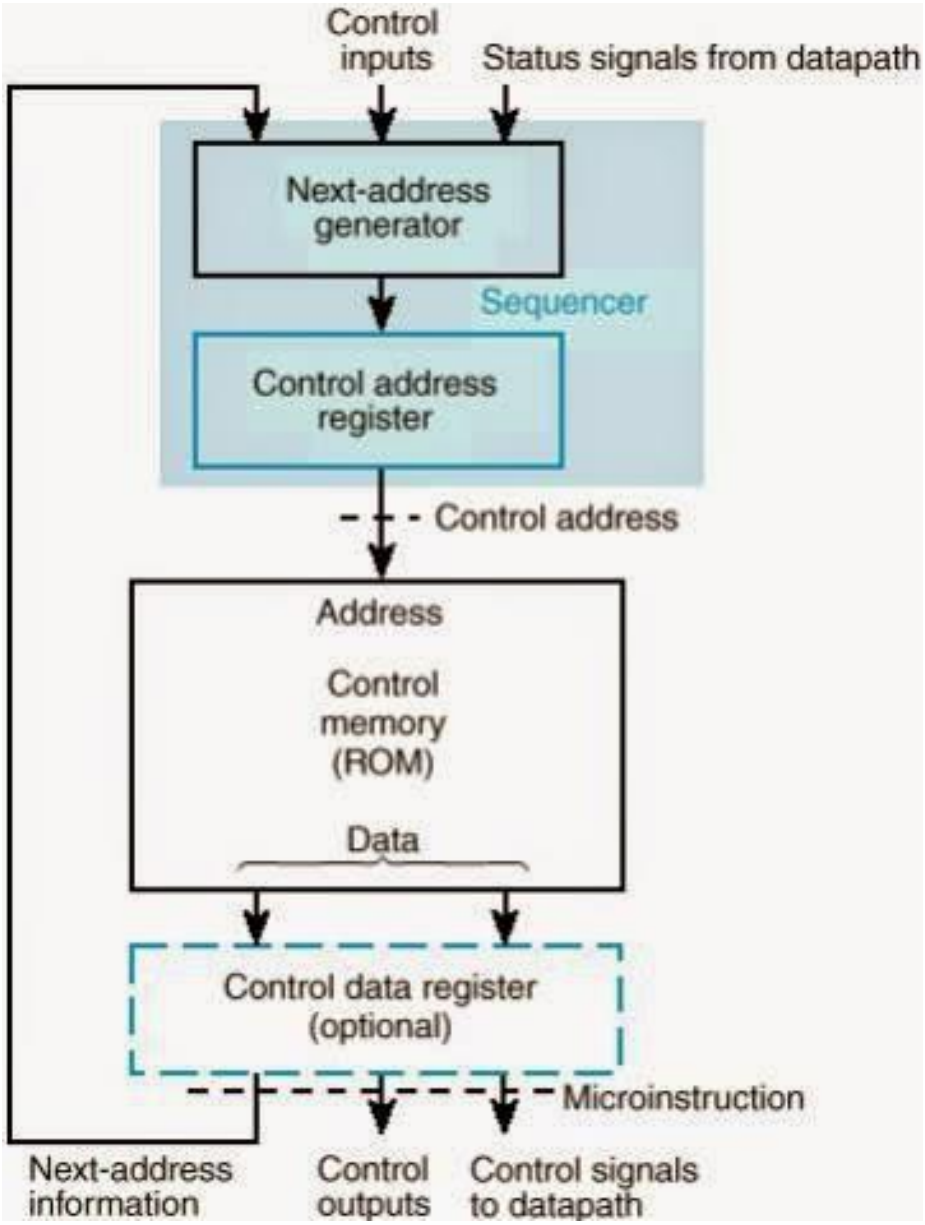
IDLE state box ı için bir flip flop kullnadık ve bunu takip eden G select için gerekli devre parçasını ekledik. Eğer bir girişe birden fazla sinyal geliyorsa bunu da or kapısı kullanarak hallettik. Yine Clear_C, load, Shift_dec sinyallerinin bağlanışına dikkat ediniz. Bu bağlanmalar önceki tablolarda verdiğimiz boolean ifadelerle özdeş olmalıdır.Sonuç olarak bu devre de register ve decoder yöntemi ile ypamış olduğumuz devre ile aynı işi yapacaktır.

Şimdi de bu devreyi mikroprogramlama yöntemi ile gerçekleştirelim

5. Microprogrammed control yöntemi

Bir control unit hafızasında depoladığı *word* şeklindeki değerlerle binary kontrolü sağlayabilir ki biz buna mikroprogramlama ile kontrol yöntemi diyeceğiz. Kontrol memory de bulunan her word sitem için bir veya daha fazla mikroişlemi belirleyen *mikrokomutlar* içerir. Sıralı bir mikrokomut bir mikroprogramı meydana getirir.Bu mikroprogramlar genellikle sistem dizayn edildiği sırada düzenlenir ki bu yüzden ROM da depolanır.Nadir de olsa RAM de depolanabilir.Eğer RAM tercih edilirse yazılabilir bir kontrol memory olur. ROM da ise sadece okuma işlemi yapabildiğimizi hatırlayınız.

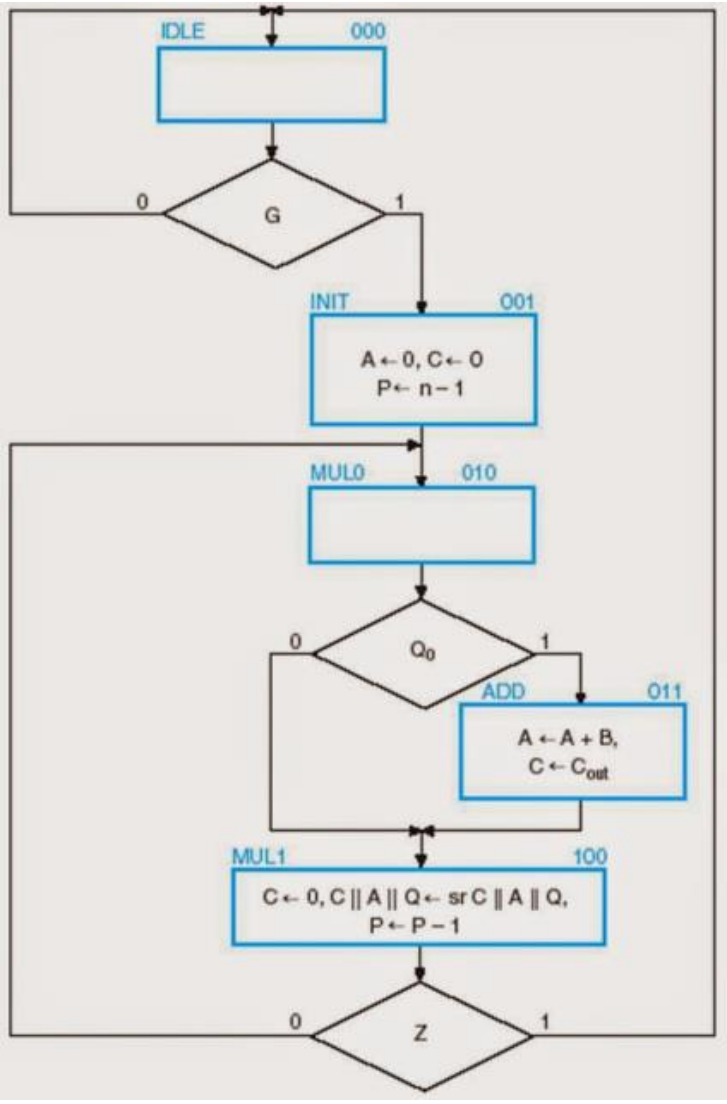
Aşağıdaki şekil genel bir mikroprogramlama kontrol ün yapılanışını gösterir



Bu şekilde kontrol memory ROM olarak tercih edilmiştir.

- The control address register(CAR) mikrokomutların adresini belirler.
- Koyuulması seçimli olan control data register(CDR), datapath ve control unit tarafından güncel olarak yürütülen mikrokomutları tutar
- Control word ün bir görevi yürütülecek sonraki mikrokomutun adresini tanımlamaktır.
- Bir mikrokomut yürütüldüğü zaman, next address genarator sonraki adresi üretir. Bu adres sonraki clock pulse i sıraında CAR a transfer edilir
- CAR bazen sequncer olarak da isimlendirilir.

Şimdi binary çarpma işlemini bir de bu yöntemle gerçekleştirelim. Bunun için çarpma işleminin sağlandığı ASM chartı tekrar hatırlayalım



Bu ASM chart içindeki conditional output box ları mikroprogramlama yöntemi ile gerçekleştirmek için state box a çevirdiğimize dikkat edin

Datapath için dört tane sinyale ihtiyacımız var . Bunlar; Initialize, Load, Clear_C, ve Shift_dec.Şimdi bunları datapath içine kod olarak yazabilmemiz için bir tablo oluşturalım. Aşağıdaki şekilde bu tablo verilmiştir.

TABLE 8-3
Control Signals for Microprogrammed Multiplier Control

Control Signal	Register Transfers	States in Which Signal is Active	Micro-instruction Bit Position	Symbolic Notation
Initialize	$A \leftarrow 0, P \leftarrow n-1$	INIT	0	IT
Load	$A \leftarrow A+B, C \leftarrow C_{out}$	ADD	1	LD
Clear_C	$C \leftarrow 0$	INIT, MUL1	2	CC
Shift_dec	$C\|A\|Q \leftarrow sr\ C\|A\|Q, P \leftarrow P-1$	MUL1	3	SD

Bu tabloda hangi sinyalin hangi işlemleri yaptığı register transfers başlığı altında verilmiştir.Ayrıca bu sinyallerin hangi state durumunda aktif olduğu da belirtilmiştir. Bu state ler in yukarıdaki ASM chart baz alınarak yapıldığını doğrulayın.
Şimdi bu sinyallerin datapath in hangi bitinde yer alacağına keyfi olarak değer verebiliriz. 4 adet sinyalimiz olduğu için 4 bitlik datapath bize yeterli olacaktır. O halde 0-3 arasında bu değerleri keyfi olarak sıralayabiliriz ki bu değerleri tabloda görmeniz mümkündür. Ayrıca verdiğimiz bu keyfi değerler için sembolik isimlendirme yapılmıştır. Bunu da tabloda görmeniz mümkündür.
Durumu yeniden ele alırsak şu çıkarımları yapabiliriz

- INIT state inde Initialize, Clear_C işlemleri
- ADD state inde Load işlemi
- MUL1 state indeClear_C ve Shift_dec işlemleri vardır

Bu yapıyı yine bir control word yardımıyla çözebiliriz bu control word ün formatı şu şekildedir

- Bu control word içindeki datapath az önceki tabloda keyfi değerler verdiğimiz sembolik isimleri IT,LD,CC,SD olan sinyalleri içerir.

- SEL bölümündeki 2 bit ise G,Z ve Q0 olmak üzere 3 değişkeni kontrol eden mux un seçme uçlarıdır.
- NXTADD0, ilgilendiği bir selectin 0 olduğu durumda gideceği adresi tutan bölümdür
- NXTADD1, ilgilendiği bir selectin 1 olduğu durumda gideceği adresi tutan bölümdür
- NXTADD1, NXTADD0, SEL ve DATAPATH bölümlerinin control word içindeki diziliş şekilleri keyfidir.

Şimdi de 2 bitlik SEL datasını hangi durumlarında ne olacağına karar verelim. Bunu da şu tabloda gösterelim

TABLE 8-4
SEL Field Definition for Binary Multiplier Control Sequencing

SEL		
Symbolic notation	Binary Code	Sequencing Microoperations
NXT	00	$CAR \leftarrow NXTADD0$
DG	01	$\overline{G}: CAR \leftarrow NXTADD0$ $G: CAR \leftarrow NXTADD1$
DQ	10	$\overline{Q_0}: CAR \leftarrow NXTADD0$ $Q_0: CAR \leftarrow NXTADD1$
DZ	11	$\overline{Z}: CAR \leftarrow NXTADD0$ $Z: CAR \leftarrow NXTADD1$

2 bitlik mux ile 4 adet çıkışı yönetebiliriz. SEL bitlerinin;

- 00 olması durumunda Control Address registera (CAR) nxtadd0 yazılacaktır ki bu şartsız geçiş demektir
- 01 olması durumunda G select 0 ise nxtadd0, G select 1 ise nxtadd1 CAR a yazılacaktır
- 10 olması durumunda Q0 select 0 ise nxtadd0, Q0 select 1 ise nxtadd1 CAR a yazılacaktır
- 11 olması durumunda Z select 0 ise nxtadd0, Z select 1 ise nxtadd1 CAR a yazılacaktır.

Buradaki dizilim ve seçim yine keyfidir.

Artık devremizi kurabiliriz.Aşağıda bu devrenin yapılmış hali gösterilmektedir.

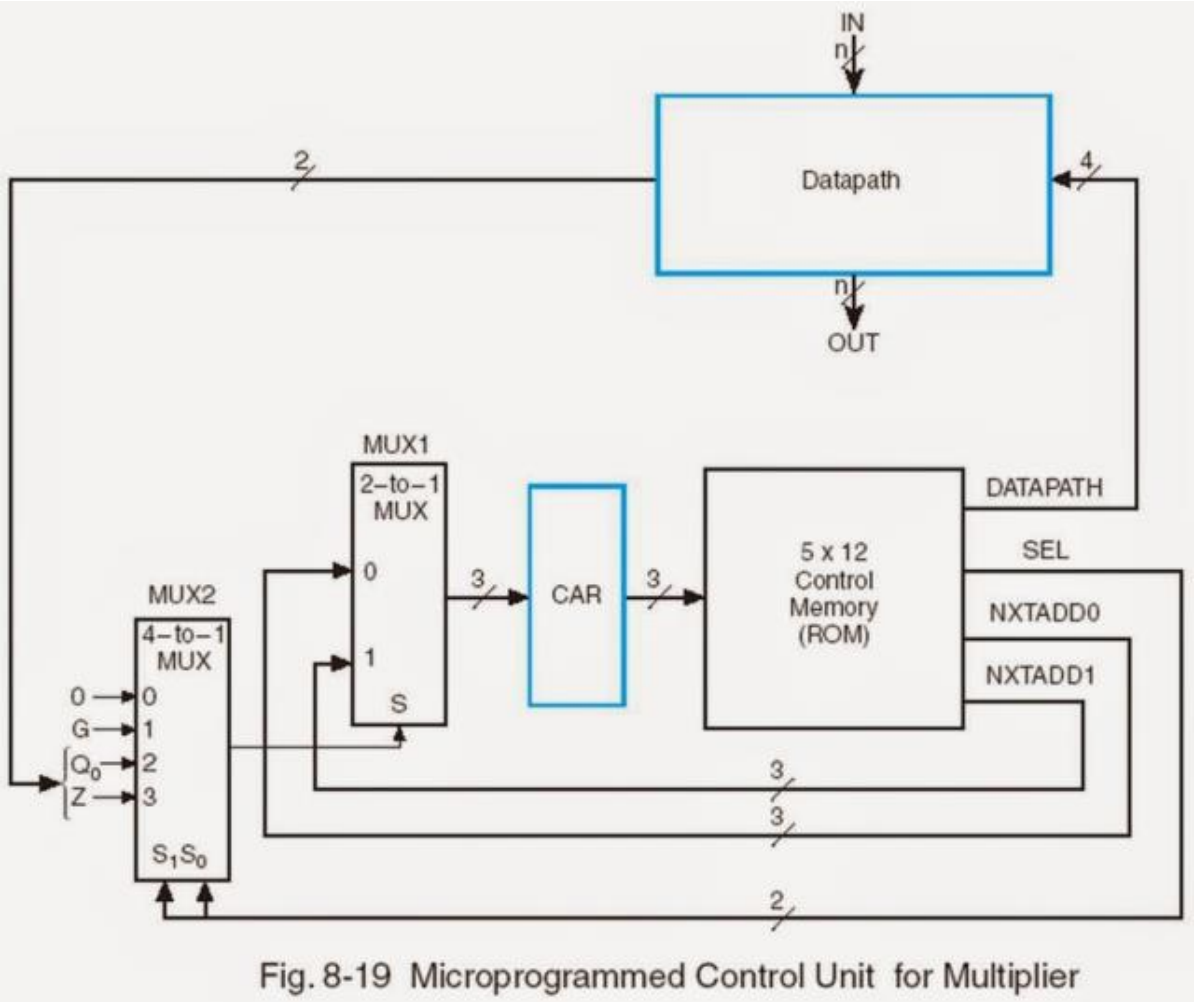


Fig. 8-19 Microprogrammed Control Unit for Multiplier

ROM un uzunluğu control wordün uzunluğu 12 bit olduğu için 12 dir. 5 adet state olduğu için de ROM 5 adet word içermekte.

- Datapath den çıkan bilginin sadece Q0 ve Z yi etkilediğine dikkat ediniz. G yi etkilemez çünkü G yi dışarıdan sistemin çalışmasını veya durmasını istediğimizde biz seçeriz.
- SEL in durumlarına göre tablodaki değerleri sağladığına dikkat ediniz.

Şimdi de binary çarpma işlemi mikroprogramı için register transferlerine bakalım. aşğıda şekilde verilen mikrokomutlar ASM chart daki statelerle uyuşmakta olduğunu kontrol ediniz

TABLE 8-5
Register Transfer Description of Binary Multiplier Microprogram

Address	Symbolic transfer statement
IDLE	$G: CAR \leftarrow \text{INIT}, \overline{G}: CAR \leftarrow \text{IDLE}$
INIT	$C \leftarrow 0, A \leftarrow 0, P \leftarrow n-1, CAR \leftarrow \text{MUL0}$
MUL0	$Q_0: CAR \leftarrow \text{ADD}, \overline{Q_0}: CAR \leftarrow \text{MUL1}$
ADD	$A \leftarrow A+B, C \leftarrow C_{\text{out}}, CAR \leftarrow \text{MUL1}$
MUL1	$C \leftarrow 0, C \| A \| Q \leftarrow \text{sr}\ C \| A \| Q, Z: CAR \leftarrow \text{IDLE}, \overline{Z}: CAR \leftarrow \text{MUL0}, P \leftarrow P-1$

şimdi de control word içine kodları nasıl yazacağımıza bakalım

TABLE 8-6
Symbolic Microprogram and Binary Microprogram for Multiplier

Address	NXTADD1	NXTADD0	SEL	DATAPATH	Address	NXTADD1	NXTADD0	SEL	DATAPATH
IDLE	INIT	IDLE	DG	None	000	001	000	01	0000
INIT	—	MUL0	NXT	IT, CC	001	000	010	00	0101
MUL0	ADD	MUL1	DQ	None	010	011	100	10	0000
ADD	—	MUL1	NXT	LD	011	000	100	00	0010
MUL1	IDLE	MUL0	DZ	CC, SD	100	000	010	11	1100

Tablonun sol tarafında sembolik olarak ifade edilirken sağ tarafında binary kodlarla belirtilmiştir.Şimdi adım adım inceleyelim

1. Adres IDLE da iken G select 0 olduğunda state yeniden IDLE; G select 1 olduğunda state INIT olacağı için nxtadd0 a IDLE , nxtadd1 e INIT yazdık.ayrıca state IDLE da bir register işlemi olmadığı için datapath kısmına none yazdık
2. Adres INIT konumunda iken herhangi bir seçim olmadığı için sonraki adrese direk geçiş yapacaktır. Bunun için nxtadd0 a MUL0 yazdık . Yeri gelmişken şöyle sorabilir siniz. MUL0 ı neden nxtadd0 yazdık da nxtadd1 e yazmadık? 2 bitlik Sel in hangi durumlarda ne olcağını belirttiğimiz tabloda sel 00 olduğunda nxtadd0 ın CAR a yazılacağını belirttik ve devremizi de ona göre tasarladık. AZ önce kurmuş olduğumuz devreye bakarsanız sel 00 olduğunda MUX2 den çıkan sinyal MUX1 den nxtadd0 sinyalinin çıkmasına neden olacaktır. En başta oraya nxtadd1 yazsaydık devreyi ona göre tasarlardık ve buradaki tabloda mul0 ı nxtadd1 e yazardık. Ayrıca adres INIT deyken IT ve CC olarak sembolize ettiğimizi hatırladığınız ınitialize ve Clear_C işlemleri olduğu için datapath e onları yazdık.
3. aynı şekilde diğer adımları da kendiniz takip ederek tablodaki değerler ile ASM chart daki değerlerin uyuştuğunu doğrulayınız.
4. Sağ taraftaki asıl kısımda yani kod kısmında sol kısımda olan sembolik isimlendirmelerin binary kodları yerine yazmaktan başka birşey yapmadık

5. Datapath kodlamasından bir örnek verecek olursak örneğin tablodaki 2. satırda IT ve CC sembolleri var. Bunu önceki tablolardan hatırlayacağınız gibi keyfi olarak sıralamıştık ve sıralamamız 0 numaralı bitten 3 numaralı bite şöyleydi : IT,LD,CC,SD burada sadece IT ve CC aktif olduğu için datapath e yazacağımız kod 0101 olacaktır. Aynı şekilde diğer karşılaştırmalar size bırakılmıştır.

6. A Simple Computer Architecture

((((((Bu konuyu bu yazıları okuduktan sonra veya önce okursanız bazı şeyler kafanızda daha iyi yer tutacaktır

1-SEP İşlemci Tasarım Aşamaları-1.pdf

2-SEP İşlemci lemci Tasarım Aşamaları-2.pdf

3- ayrıca daha fazla döküman için http://www.mcu-turkey.com/))))

Programlanabilir sistemlerde komutlar genellikle memory de depolanır. Bu RAM veya ROM olabilir. Bu komutların sırayla çalışabilmesi için çalışacak komutların hafızadaki adreslerinin elde edilmesi gerekir. Bir bilgisayarda bu adres program counter(PC) denilen registerdan sağlanır.

PC paralel yükleme özelliğine sahip olması gerekir.

Depolama kaynakları

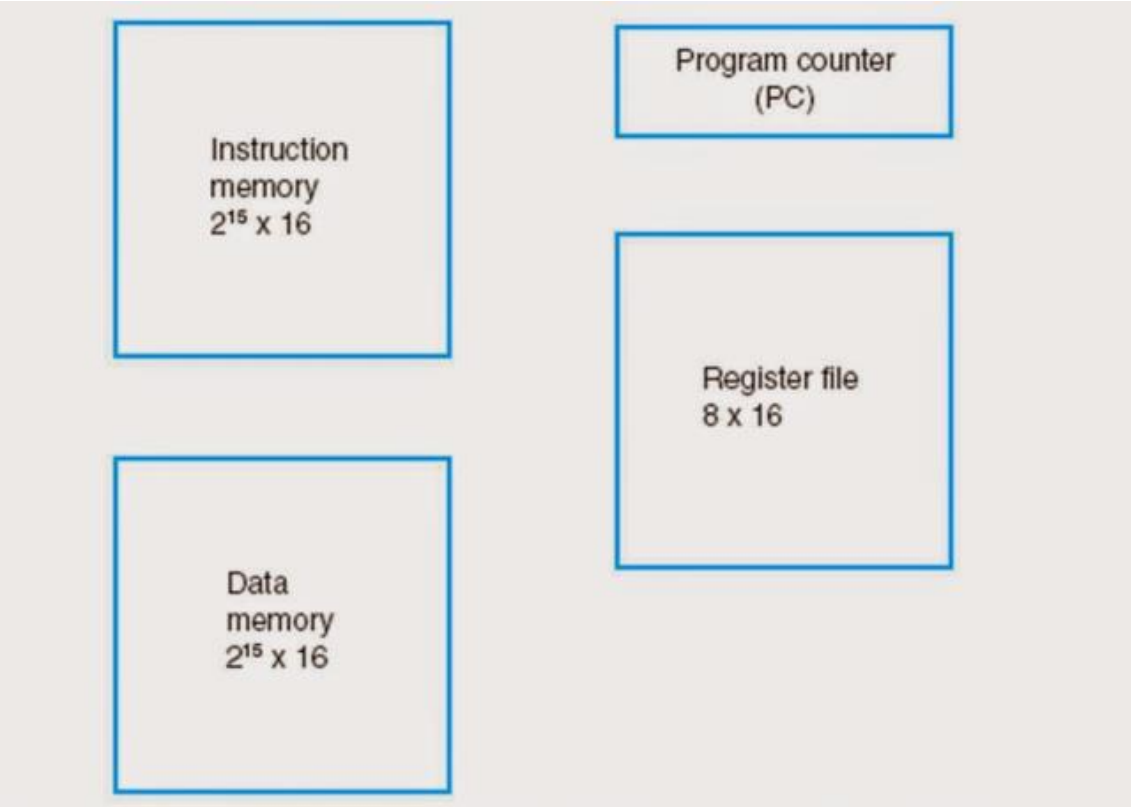


Fig. 8-22 Storage Resource Diagram for a Simple Computer

Bu mimarinin 2 adet memory içerdiğine dikkat edin. Biri komutları tutmak için kullanılırken diğeri dataların depolanması için kullanılır. Aslına bakarsanız ikisini de aynı memory içinde depolayabilirdik. Bunu ileride tartışmakla birlikte burada en azından iki farklı görevde memory nin olduğunun bilinmesidir. Ayrıca program counter (pc) kaç bitlik ise register file da o kadar bit olmalı . 16 bit registerlar ve 16 bit program counter gibi..

İNSTRUCTİON FORMAT

Bir komutun formatı genel larak bir dikdörtgen bir kutu şeklinde betimlenir. Sonra kendi aralarında alanlara ayrılır. Her alan register adresi, sabit değer, operation code gibi özel bir göreve işaret eder.

Bir komutun operation code kısmı, opcode diye kısaltılır, komutun toplam çıkarma kaydırma hangi işlemleri yapacağına karar veren kısımdır.Bir komutun opcode için gerekli olan bit sayısı komut setinde bulunan toplam işlem sayısının bir fonksiyonudur. opcode bit sayısı 2^m farklı işlem için en azından m bitten oluşmak zorundadır. Dizayn eden her kimse her bir işlem için bir kod işaretler. Örneğin,bir tanesi toplama işlemi olan maximum 128 ayrı işlem yapan bir bilgisayar düşünün. O halde opcode 7 bitten oluşmalıdır. toplama işlemi içn de 0000010 kodu işaretlenmiş olsun. 0000010 kodu control unit tarafından ortaya çıkarılınca toplama işlemi gerçekleşecektir.

Basit bir bilgisayar için 3 tane komut formatı aşağıdaki gibidir.

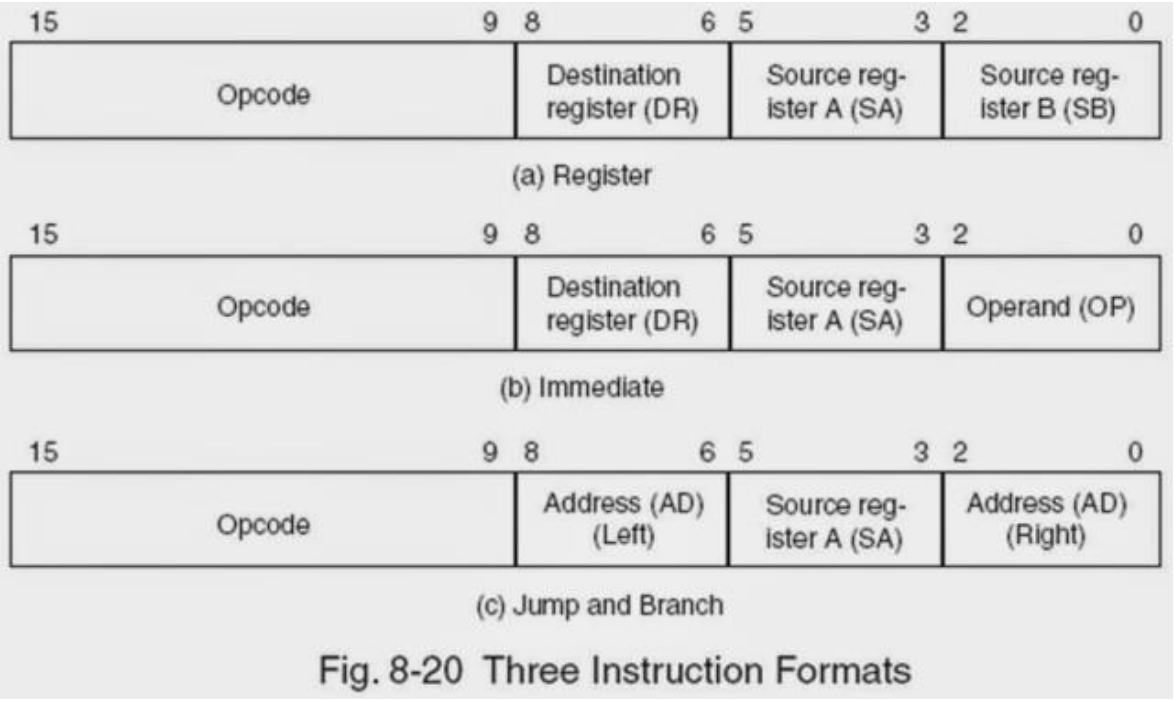


Fig. 8-20 Three Instruction Formats

R0 dan R7 ye olmak üzere bilgisayarın 8 adet registera sahip olduğu varsayılmıştır. a daki şekilde sonuç için 1 register ve işlem kaynağı için 2 adet register kaynağı ayrılmıştır.

Opcode içinde bulunan kodlar ayrıca registerların kullanım şeklini de belirler. Yani demek istediğimiz şudur. Opcode daki kodlara göre registerların kullanım şeklinin a daki gibi mi b deki gibi mi yoksa c deki gibi mi olacağı belirlenir. Yoksa bir bilgisayarın içinde 3 ayrı yerde bu şekillerin ifade ettiği donanımlar yoktur.(en azından şimdilik :))

a daki şekilde; SA 010 seçilerek R2 yi, SB 011 seçilerek R3 ü, DR de 001 seçilerek R1 i ifade ediyor olsun. opcode ise toplama işlemini işaret eden bir kod getirmiş olsun R2 ve R3 biti toplanıp R1 register ına yazılacaktır.

b deki şekilde SA R7 yi , DR R2 yi işaret etsin ve OP 011 olsun. 011 onluk tabanda 3 e işaret etmektedir.opcode da toplama işlemini işaret ediyorsa R7 deki data ile 3 datası toplanarak R2 ye yazılacaktır.

c deki şekilde , hiçbir register file ve memory içeriği değişmez. Memoryden yakalanan komutun hangi sırada işleneceğine etki eder.Buna örnek olarak;
jump ve branch komutlarını verebiliriz. Bir adet kaynak registerı SA ve dağıtıcı adres alanı AD bulunmaktadır. eğer bir branch komutu(dallanma) meydana gelirse yeni adres PC nin güncel içeriği ile 6 bitlik AD içeriğinin toplanmasından elde edilen sonuç olacaktır. Bu adresleme metodu PC relative olarak adlandırılır ve bu 6 bitlik AD alanına 2's complement olarak adress offset denir. 2's complementi korumak için bu 6 bitlik AD ye sign extension uygulanarak 16 bitlik forma dönüştürülür.
Örneğin;
PC değeri 55 olsun. R6 nın içeriğinin 0 olması durumunda gerçekleşecek branch komutu sonrası yeni adresimizin 35 olması istensin. opcode branch on zero komutunu belirlemelidir. SA ise R6 yı temsil etmelidir. 6 bit olan AD ise 2's complement gösteriminde -20 yi göstermelidir ki 55 +(-20)=35 olabilsin. R6 nın içeriği 0 ise PC nin içeriği 35 olacaktır. Aksi taktirde yani R6 nın içeriği 1 ise pc saymaya devam edeck ve 55 ten 56 ya geçecektir.

İNSTRUCTİON SPECİFİCATİONS

komut specification lar her ayrı komutun sistem tarafından çalıştırılacak komutları tanımlar. Her komut için kısaltılmış ifadesi mnemonic ile birlikte bir opcode belirlenmiştir. mnemonic opcode için sembolik gösterimlerde kolaylık açısından kullanılabilir. Bu sembolik ifadeler assembler denilen program saysinde tekrar binary komutlar a çevrilir. basit bir bilgisayar için aşağıda bu kodlar tablo halinde verilmiştir.

Instruction Specifications for the Simple Computer

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	R[DR] ← R[SA]	N, Z
Increment	0000001	INC	RD,RA	R[DR] ← R[SA] + 1	N, Z
Add	0000010	ADD	RD,RA,RB	R[DR] ← R[SA] + R[SB]	N, Z
Subtract	0000101	SUB	RD,RA,RB	R[DR] ← R[SA] − R[SB]	N, Z
Decrement	0000110	DEC	RD,RA	R[DR] ← R[SA] − 1	N, Z
AND	0001000	AND	RD,RA,RB	R[DR] ← R[SA] ∧ R[SB]	N, Z
OR	0001001	OR	RD,RA,RB	R[DR] ← R[SA] ∨ R[SB]	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	R[DR] ← R[SA] ⊕ R[SB]	N, Z
NOT	0001011	NOT	RD,RA	R[DR] ← $\overline{R[SA]}$	N, Z
Move B	0001100	MOVB	RD,RB	R[DR] ← R[SB]	
Shift Right	0001101	SHR	RD,RB	R[DR] ← sr R[SB]	
Shift Left	0001110	SHL	RD,RB	R[DR] ← sl R[SB]	
Load Immediate	1001100	LDI	RD, OP	R[DR] ← zf OP	
Add Immediate	1000010	ADI	RD,RA,OP	R[DR] ← R[SA] + zf OP	
Load	0010000	LD	RD,RA	R[DR] ← M[SA]	
Store	0100000	ST	RA,RB	M[SA] ← R[SB]	
Branch on Zero	1100000	BRZ	RA,AD	if (R[SA] = 0) PC ← PC + se AD	
Branch on Negative	1100001	BRN	RA,AD	if (R[SA] < 0) PC ← PC + se AD	
Jump	1110000	JMP	RA	PC ← R[SA]	

komutların ve dataların binary biçimde memory e yerleşimi ise aşağıdaki gibidir

TABLE 10-9
Memory Representation of Instructions and Data

Decimal Address	Memory Contents	Decimal Opcode	Other Fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2, SB:3	R1 ← R2 – R3
35	0100000 000 100 101	32 (Store)	SA:4, SB:5	M[R4] ← R5
45	1000010 010 111 011	66 (Add Immediate)	DR:2, SA:7, OP:3	R2 ← R7 + 3
55	1100000 101 110 100	96 (Branch on Zero)	AD: 44, SA:6	If R6 = 0, PC ← PC – 20
70	0000000011000000	Data = 192. After execution of instruction in 35, Data = 80.		

Bu tabloda ayrı formatta 4 adet kodun depolanması görüntülenmiştir.

- adres 25 de R2 registerındaki datadan R3 registerında datanın çıkarılıp R1 registerına yazılma işlemi var. Bir üstteki tablodan kontrol edebileceğiniz gibi çıkarma (sub) işleminin kodu 0000101 olarak belirlendiği için memory içeriğindeki opcode kısmı 0000101 (onluk tabanda da 5 olmak üzere) olarak yazılmıştır.memory içerisinde geriye kalan kod ise register adreslerinden oluşmakta.001 hedef fegisterı olan R1 i ,010 kaynak registerı olan R2 ve 011 kodu ise R3 register ını işaretlemektedir.
- adres 35 de ise R5 registerındaki içeriğin R4 ün işaret ettiği memory hücresine yazımı vardır ki biz buna store diyoruz. Store kodunun yine bir önceki tablodan değerini kontrol ederek opcode un 0100000 (onluk tabanda 32) olacağına karar veriyoruz.R4 registerının içeriğinin 70 ve R5 inki ise 80 olduğunu varsayalım. Bu kod çalıştıktan sonra memory nin 70 inci adresinde 80 değeri depolanacaktır.
- adres 45 de ise onluk tabanda 3 sayısının R7 registerındaki data ile toplanıp R2 registerına yüklenme işlemi var. buradaki opcode —-artık yukarıdaki tablodan

  kontrol edin demeyeceğim —- 1000010 dir diğer kodlar ise registerları işaret eder. 3 sayısının 011 olduğuna dikkat ediniz.
- adres 55 de ise branch komutu gerçekleşmektedir. opcode 1100000(onluk tabanda 96) kaynak registerı R6 olarak belirlenmiştir. AD(soldaki) nin içeriğinin 101 ve AD(sağdaki) 100 olduğuna dikkat ediniz. Bu ikisini bir araya getirip sign extension uyguladığımızda onluk tabanda (-20) sayısını ifade eden 1111111111101100 2's complement değerine ulaşmış oluruz.Eğer R6 nın içeriği 0 ise PC değerine (-20)

eklenerek 35 değerine ulaşır. Eğer R6 nın içeriği 0 değilse PC nin yeni değeri 55+1=56 olacaktır. burada şöyle düşünebiliriz. sign extension geçirmiş bir branch komutunun en soldan 10 biti birdir. en sağdaki 6 bit ise bizim AD(left) ve AD(right) da yazılan kodların birleşimidir. eğer bu ikisinin birleşimi 000000 olursa o halde sign extensiondan geçen kısım şöyle olacaktır. 1111111111000000 bu da -64 e işaret eder. örneğin az önceki gibi pc nin içeriğinin 20 geriye gitmesi istensin. yani sign extensiondan geçtikten sonra -20 olsun. düşünmemiz gereken şu olacaktır. -64 e kaç

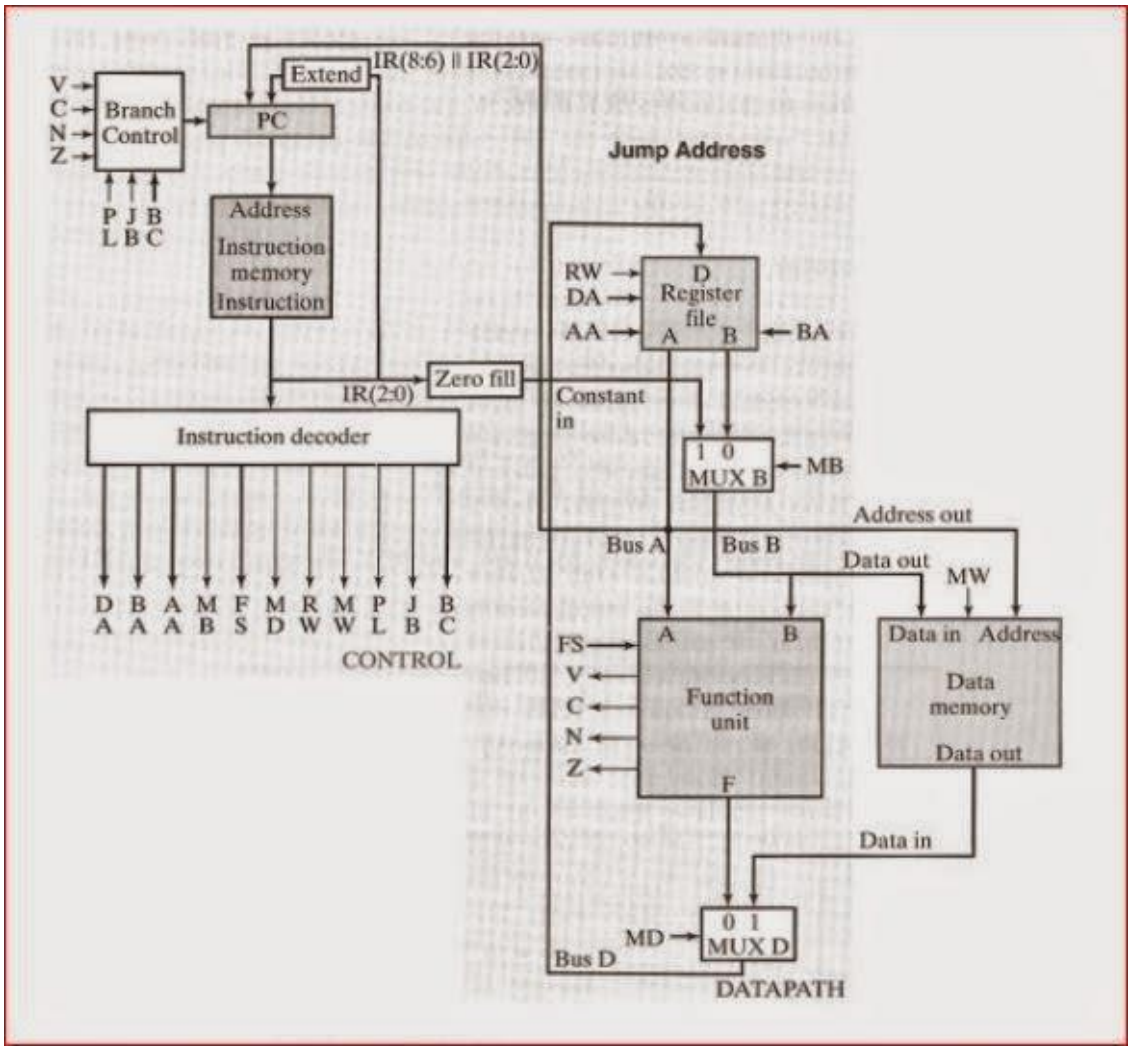
eklersem -20 olur? cevap 44 olacaktır. bilenleri tebrik ediyorum her neyse.. yani AD(left) ve AD(right) da yazan kodların 6 bitlik birleşimi 44 olması lazım bu da 101100 sayısına işaret eder. yukarıdaki örnekten bunu kontrol edebilirsiniz. diyeceğim odur ki bundan sonra branch yaparken kolaylık olması bakımından hep -64 e kaç eklemeliyim diye düşünebiliriz..

Tabloda görülen bu değerle keyfi olarak yerleştirilmiştir. Sonuçta belirleyeceğiniz koda göre donanım tasarlayacağınızdan burada keyfi değerler vermenizin sakıncası yoktur. Ancak biz datapath tasarımında kullandığımız kodlara göre devam ediyoruz. Burada keyfi değer verdi iseniz ilerideki tüm işlemler bu kodlara göre tasarlayacağınızı unutmayınız.

Çoğu bilgisayarda memory nin içeriği 64 bite kadar çıkmaktadır. Bu yüzden daha fazla operand ve adres tutulabilir.

Bir işlem bilgisayar hafızasında binary olarak tutulan bir kod tarafından belirlenir. Bilgisayar içindeki control unit belirli olan adresi veya program counter tarafından sağlanan adresleri memoryden komuta erişmek için kullanır. Daha sonra opcode bitlerini ve komutdaki diğer bilgileri gerekli mikroişlemi gerçekleştirmesi için çözer.

SİNGLE CYCLE HARDWİRED CONTROL



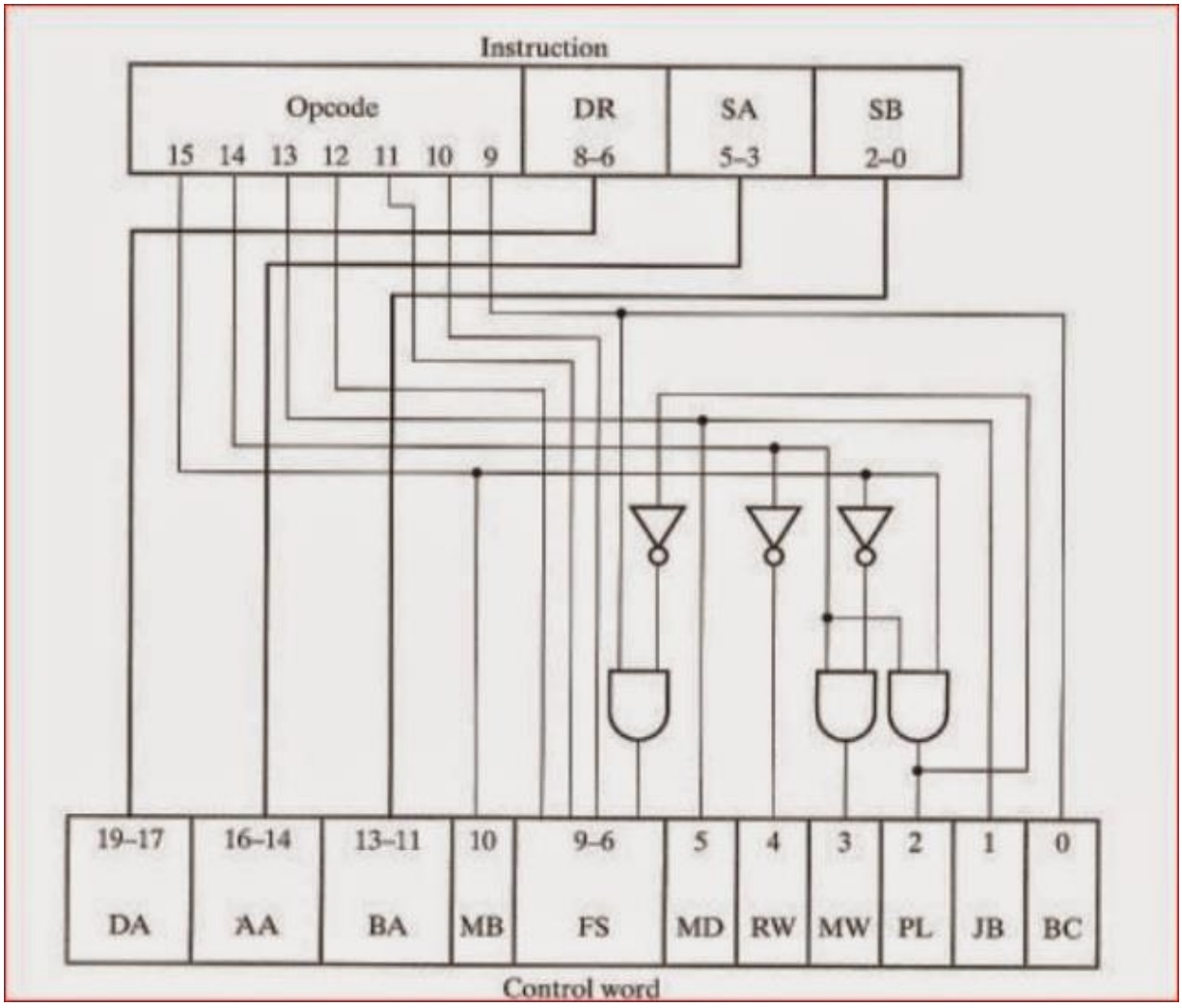
Buradaki datapath yapısı önceki kısımlarda tasarladığımız datapath ile tamaman aynı. Buradaki datapath in adress out ,data out, data in kısmına data memory bağlanmıştır.Bir adet MW adında kontrol sinyali bulunmaktadır. Eğer MW 1 ise memory ye yazma işlemi yapılacaktır. Yama işlemi yok memory den bilgi çekilecekse veya hiçbir işlem yapılmayacaksa MW nin 0 olması gerekir.

Control unit şeklin sol kısmında bulunmaktadır. Aslında sadece control unit olarak düşünülmemelidir. bir adres girişi ve bir komut çıkışı ile birlikte instruction memory (komutları hafızada tutmak için kullanılır). de bu yapının içinde uygunluk için gösterilmiştir.

- PC instruction memory ye adresleri sağlar ve bu adresdeki kodlar instruction memory nin çıkışından çözülmek üzere instruction decoder e gelir.
- instruction memory nin çıkışındaki komutlar aynı zamanda Extend ve zerofill e de gider.

- Buradaki extend yapısını şöyle açıklayabiliriz: bir önceki kısımda eğer bir branch komutu geldiyse soldaki ve sağdaki AD bitlerini birleştirerek 6 bitlik bir değer elde etmiştik. Burada bu iki değeri birleştirmek için extend yapısı kullanıldı.
- Bir registerdaki değerle sabit bir sayıyı işleme sokan bir instruction format görmüştük. Bunun için instruct un en sağdaki 3 biti ayrılmıştı. Bizim registerlar 16 bitlik olduğu için 3 bitlik bir sayı ile işleme sokulabilmesi için zerofill yapısı ile bu 3 bitlik değerin sol kısmına 13 adet sıfır eklenerek işleme hazır hale getirilir. örneğin gelen sabit sayı 110 ise zerofill yapsından sonra 0000000000000110 olacaktır.
- PC her clock cycle ında güncellenir. PC nin davranışı opcode tarafından tanımlanan komplex bir register yapısıdır.
- Eğer bir jump komutu gerçekleşirse PC nin yeni değeri Bus A nın taşıdığı değer olur.
- Eğer bir branch komutu gelirse PC nin yeni değeri, önceki değeri ile extend den gelen datanın toplamı olur. Aksi takdirde PC nin değeri 1 artar.

İnstruction decoder



- Şekilde görüldüğü gibi DA,AA ve BA ; DR,SA, ve SB değerlerine eşittir.
- Eğer bir jump veya branch komutu varsa PL=1 olmaktadır ve PC yüklenir
- Eğer PL=0 sa PC değeri 1 Arttırılır.

MB, MD, RW, MW, PL ,JB, BC komutlarının instruction un 15,14 ve 13 üncü bitlerine göre şekil aldığını bulduk. buna göre aşağıdaki tablo düzenlenmiştir.

TABLE 10-10
Truth Table for Instruction Decoder Logic

Instruction Function Type	Instruction Bits				Control Word Bits						
	15	14	13	9	MB	MD	RW	MW	PL	JB	BC
Function unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
Memory read	0	0	1	X	0	1	1	0	0	X	X
Memory write	0	1	0	X	0	X	0	1	0	X	X
Function unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
Conditional branch on zero (Z)	1	1	0	0	X	X	0	0	1	0	0
Conditional branch on negative (N)	1	1	0	1	X	X	0	0	1	0	1
Unconditional Jump	1	1	1	X	X	X	0	0	1	1	X

datapath tasarımından da hatırlayacağınız gibi FS 0000 olduğunda register A daki değer hiçbir değişikliğe uğramadan ALU dan çıkıyordu. herhangi bir branch komutunda da Bus A dan gelen değerin değişmesini istemediğimiz için FS=0000 olmak zorunda.

Örnek komutlar ve program

Aşağıdaki tabloda 6 adet komut listelenmiştir.

TABLE 10-11
Six Instructions for the Single-Cycle Computer

Operation code	Symbolic name	Format	Description	Function	MB	MD	RW	MW	PL	JB	BC
1000010	ADI	Immediate	Add immediate operand	$R[DR] \leftarrow R[SA] + zf\ I(2{:}0)$	1	0	1	0	0	0	0
0010000	LD	Register	Load memory content into register	$R[DR] \leftarrow M[R[SA]]$	0	1	1	0	0	1	0
0100000	ST	Register	Store register content in memory	$M[R[SA]] \leftarrow R[SB]$	0	1	0	1	0	0	0
0001110	SL	Register	Shift left	$R[DR] \leftarrow slR[SB]$	0	0	1	0	0	1	0
0001011	NOT	Register	Complement register	$R[DR] \leftarrow \overline{R[SA]}$	0	0	1	0	0	0	1
1100000	BRZ	Jump/Branch	If $R[SA] = 0$, branch to PC + se AD	If $R[SA] = 0$, $PC \leftarrow PC + seAD$, If $R[SA] \neq 0, PC \leftarrow PC + 1$	1	0	0	0	1	0	0

binary komutlar kolaylık açısından sembolik olarak isimlendirilmiştir.

- şimdi ilk komutu inceleyelim yani add immediate(ADI) yı
    - ADI komutu instruction memory nin çıkışındadır. Bu komutun ilk üç biti 100 olduğundan instruction decoder şu çözümlemeleri yapacaktır:
        - MB=1, MD=0, RW=1 ve MW=0.
    - Opcode un son üç biti zerofill vasıtasıyla 16 bite tamamlanacaktır.
    - MB=1 olduğundan zerofill e tabi tutulan bu kod Bus B ye gelecektir (**SİNGLE CYCLE HARDWİRED CONTROL** şeklinden takip ediniz**)**
    - MD=0 olduğundan function unit den gelen çıkış seçilecektir.
    - opcode un son 4 biti 0010 FS yi belirleyecektir ki bu da toplama işlemini işaret eder.
    - Dolayısıyla zerofillenmiş Bus B deki data ile register file dan SA registerından gelen data toplanacaktır ve sonuç Bus D ye iletilecektir.
    - Son olarak MW=0 olduğundan memory ye yazım işlemi yapılmayacaktır.
    - Bu anlattığımız tüm işlemler yalnızca bir clock cycle içinde olacağına dikkat ediniz.

diğer kodları aynı şekilde takip edip test edebilirsiniz.
Örneğin;
83 – (2 + 3) işlemini yaptıralım.
R[R3]=248
M[248]=2
M[249]=83
olsun ve sonuç M[250]'ye yazılsın. uygulamamız gereken kodlar şu şekilde olacaktır.

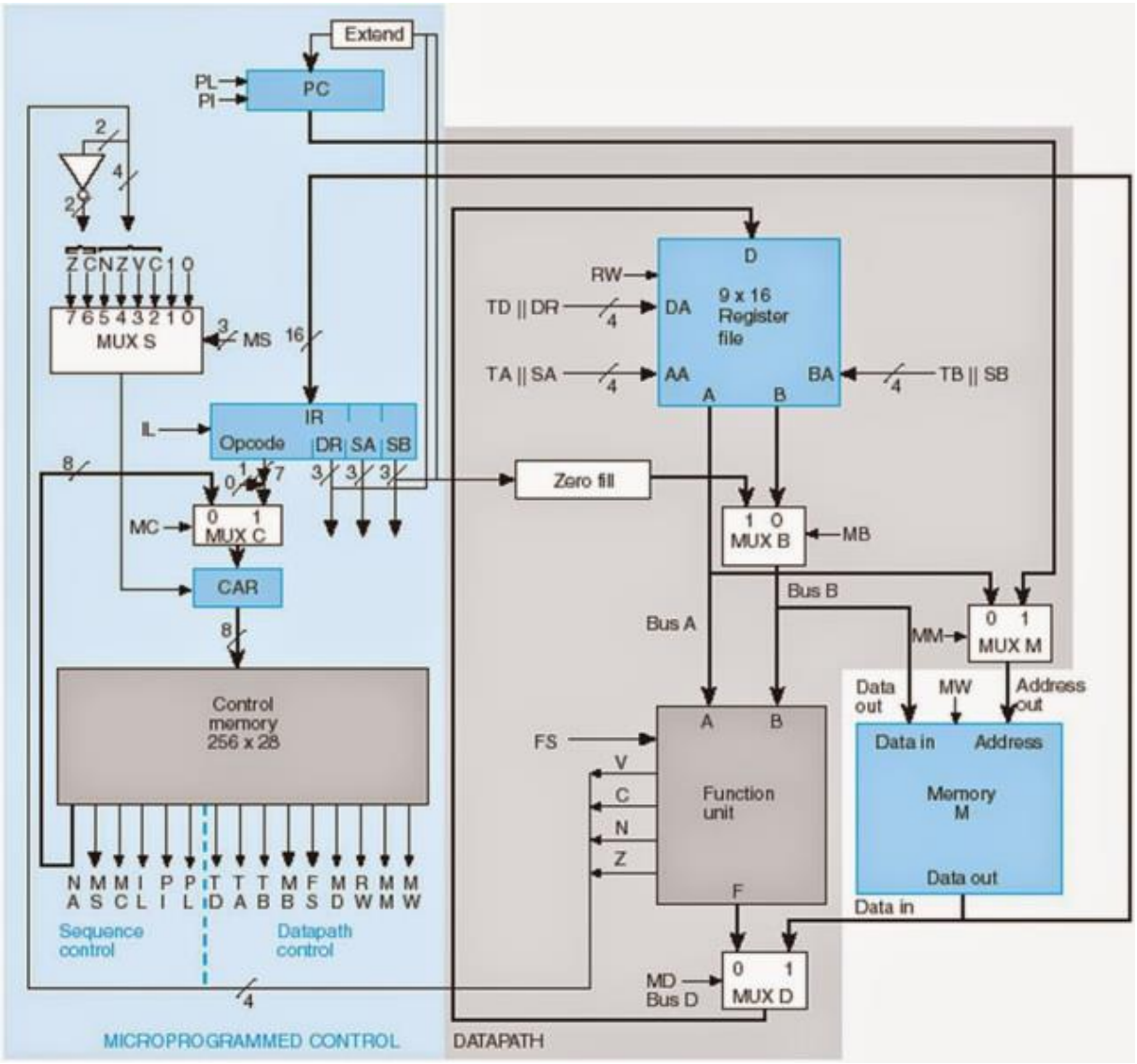
```
                ; R3=248
LD  R1,R3       ; R1=2          // R3 registerında bulunan datanın memoryde işaret ettiği yerde bulunan değerin R1 e yazılması

ADI R1,R1,3     ; R1=2+3=5      // R1 e R1+3 değerinin yazılması
NOT R1,R1       ; R1=R1'        // R1 in 2's complementi için önce tersinin alınması
INC R1,R1       ; R1=-5         // R1 e 1 eklemekle 2's comlement elde edilmesi
INC R3,R3       ; R3=249        // 83 değerine ulaşmak için R3 ün 1 arttırılması
LD  R2,R3       ; R2=83         //R3 registerında bulunan datanın memoryde işaret ettiği yerde bulunan değerin R2 e yazılması
ADD R2,R2,R1    ; R2=78         // 83-5 işleminin yapılması
INC R3,R3       ; R3=250        // sonucu 250 . memory hücresine yazmak için R3 ün arttırılması
ST  R3,R2       ; M[250]=78     //R3 ün işaret ettiği memory hücresine R2 datasının yazılması
```

((((buraya kadar mano nun3. edition daki kodlar ve şekiller kullanılmıştır. Buradan sonra 2. edition daki şekiller ve kodlar kullanılacaktır.))))

7. Multiple-cycle microprogrammed control

Bu yapıyı gerçeklemek için single cycle yapısından yaralanacağız. sadece bu yapıdaki elemanlar modifiye etmemiz gerekecek aşağıda üzerinde multiple cycle tasarımı ve mikrokomut formatı bulunmaktadır.



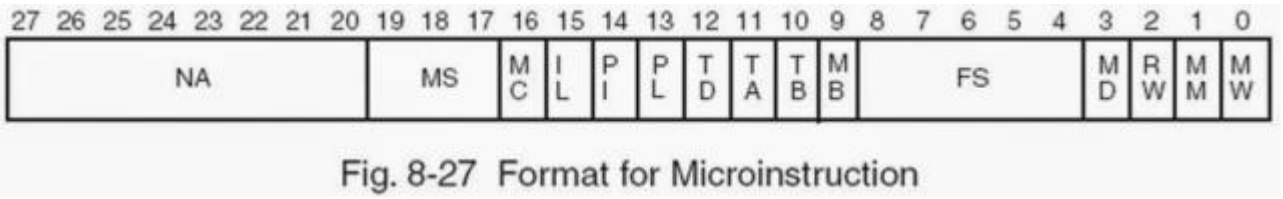


Fig. 8-27 Format for Microinstruction

Bu tasarımda daha önce birbirinden ayrı olarak bulunduğunu söylediğimiz instruction memory ve data memory tek bir M memory de yeniden yerleştirilmiştir.

not: hep karıştırıldığı için söylüyorum data registerlarda tutulur, komutlar ise instruction memory de tutulur.

- komutları fetch(yakalamak) için,PC ,memory e adres kaynağıdır. datanın yani bilginin yakalanması için ise Bus A memory için adres kaynağıdır. Hangi kaynaktan adresin seçileceği ise Mux M in işidir. Mux M yi de MM sinyali ile kontrol ettiğimize dikkat edin.
- multiple cycle da bir komut işlenirken adından da belli olacağı üzerebirden fazla clock pulse ne ihtiyaç duyulur. bir komut çalışırken dataların kaybolmaması için geçici olarak bir register da tutulması icap eder. geçici olarak kullanılan bu register kullanıcıya genellikle görünür değildir. Bu basit tasarıma complex bir yapı eklemekten kaçınmak için bir tek geçici register R8 i register file içine ekleriz. R8 i adreslemek için şekilde görüldüğü gibi her adres girişinin sol kısmına bir bit eklenmiştir. Örneğin
    - TA=0 ise AA=TA||SA=0||SA 8 registerden (R0… R7) birini işaret eder.
    - TA=1 ise AA=TA||SA=1||SA R8'i işaret eder.
    -

Aşağıdaki tablo datapath in yeni sinyalleri ile birlikte özeliklerini vermektedir.

TABLE 8-9
Control Word Information for Datapath

TD	TA	TB	MB		FS		MD	RW	MM	MW	
Select	Select	Select	Select	Code	Function	Code	Select	Function	Select	Function	Code
R[DR]	R[SA]	R[SB]	Register	0	$F = A$	00000	FnUt	No write (NW)	Address	No write (NW)	0
$R8$	$R8$	$R8$	Constant	1	$F = A + 1$	00001	Data In	Write (WR)	PC	Write (WR)	1
					$F = A + B$	00010					
					$F = A + B + 1$	00011					
					$F = A + \overline{B}$	00100					
					$F = A + \overline{B} + 1$	00101					
					$F = A - 1$	00110					
					$F = A$	00111					
					$F = A \wedge B$	01000					
					$F = A \vee B$	01010					
					$F = A \oplus B$	01100					
					$F = \overline{A}$	01110					
					$F = B$	10000					
					$F = \text{sr } B$	10100					
					$F = \text{sl } B$	11000					

datalar için geçici bir register a ihtiyaç duyduğumuz gibi komutları(instruction) ları da aynı nedenden ötürü geçici olarak tutmamız gerekir.bunun için tasarımda görmüş olduğunuz instruction register(IR) kullanılacaktır.IR ye yükleme yalnızca memoryden data okuma sırasında olur. IR nin bir kontrol sinyali IL vardır. IL , sadece memory'den instruction fetch edildiği zaman aktif olur. Instruction fetch: "program memory" den PC'nin gösterdiği instruction alınıp, IR'ye yazılması işlemine denir.
PC nin değeri:Eğer PI=1 ise; PC instruction fetch edildiğinde bir artar.

Eğer PL=1 ise; PC=PC+se(DR||SB) ile yüklenir. Bu jump veya branch instructionlarda olur. Böylece single cycle computerdeki branch kontrol elimine edilmektedir!!

- CAR tarafından adreslenen memory nin içinde mikrokomutlar tutulmaktadır. CAR ya bir arttırılır(increment) yada next adress logic in tanımladığı değer ile yüklenir. CAR a gelen bu iki bilgiden hangisinin seçileceği MUX C nin işidir. kaynaklardan biri, mevcut

mikrokomut alanında bulunan 8 bitlik NA, diğeri ise IR nin içindeki 7 bitlik opcode. bu opcode sol kısmına 1 sıfır alarak 8 bit haline gelir. mu seçimi de MC sinyali üstlenir.

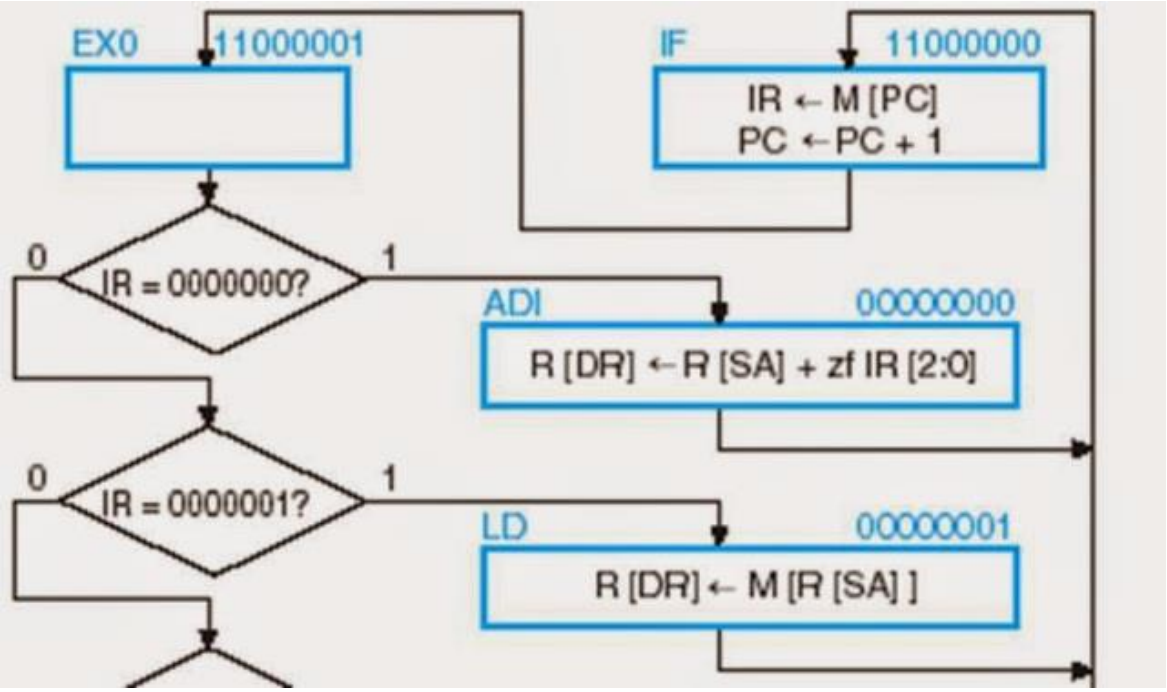
- control memory de bulunan mikroprogramıın başlangıcında komutun çalışması için ilk olarak CAR a opcode yüklenir. Örneğin opcode 0000001 ise control memory deki proagram 0000001 ile yüklenir.
- CAR ın arttırılacağı mı yoksa yükleme işleminin mi yapılcağını MUX S belirleyecektir. bu mux mikrokomutun içindeki MS alanındaki 3 bit ile kontrol edilir. MS in değerine bağlı olarak 8 değişken vaya sabit seçilebilir. MUX S den gelen çıkış 0 olursa CAR arttırılır, MUX S den gelen çıkış 1 olursa CAR yüklenir. Aşağıdaki tabloda bunu inceleyebilirsiniz.

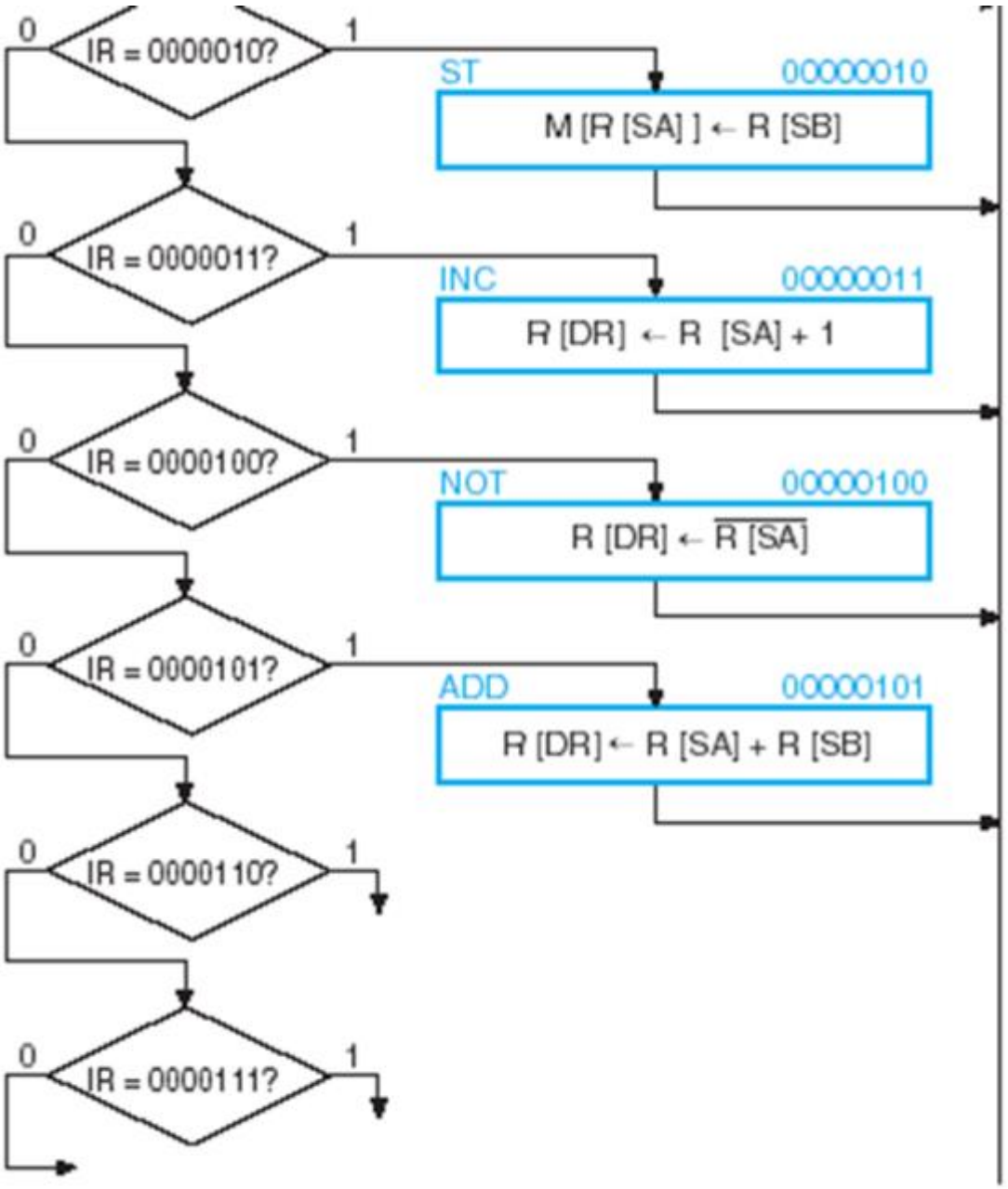
TABLE 8-10
Control Information for Sequence Control Fields

MS			MC		IL	PI	PL			
Action	Symbolic Notation	Code	Select	Symbolic Notation	Action	Symbolic Notation	Action	Symbolic Notation	Action	Symbolic Notation	Code
Increment *CAR*	CNT	000	NA	NXA	No load	NLI	No load	NLP	No load	NLP	0
Load *CAR*	NXT	001	Opcode	OPC	Load instr.	LDI	Increment PC	INP	Load PC	LDP	1
If $C = 1$, load *CAR*; else increment *CAR*	BC	010									
If $V = 1$, load *CAR*; else increment CAR	BV	011									
If $Z = 1$, load *CAR*; else increment *CAR*	BZ	100									
If $N = 1$, load *CAR*; else increment *CAR*	BN	101									
If $C = 0$, load *CAR*; else increment *CAR*	BNC	110									
If $Z = 0$, load *CAR*, else increment *CAR*	BNZ	111									

Microprogram design

komutları yakalamak ve çalıştırmak adına mikroprogram yazımına başlamak için artık tüm bilgilere sahibiz desek yeri olur umarım. Bu adımı hemen geçmek yerine register transferlerini ve komutların sırasını tanımlayabilmek adına ASM Chart kullanacağız. Bu ASM chart şu şekildedir





(iki resmi birleşik gibi düşünün)

Her komutun işlenişi iki adımdan oluşuyor: instruction fetch(yakalama) ve instruction execution(çalışma).

- instruction fetch state IF de meydana gelir(sağ üst köşede)
- PC memory M deki komutların adresini içerir. state IF nin bitimindeki clock pulse nde bu adres memory e uygulanır ve IR ye yüklenir.Aynı clock pulse i PC nin değerinin güncellenmesine ve yeni state EX0 a geçilmesine sebep olur.
- EX0 state inde CAR, 0||IR işleminin sonucu yüklenir(soluna “0” eklenerek CAR’a yüklenir)

- IR den gelen opcode değerine bağlı olarak— burada 7 bitlik opcode olduğundan 128 farklı komut gerçekleştirilebilir– komut sayısı değişecektir.
- 128 komutun hepsi gösterilmemiştir. yalnızca 8 adet durum ASM chart da gösterilmiştir. Gösterimin devam ettiği şekilden de bellidir.
- ilk opcode için konuşacak olursak: 0000000, “Add immediate” komutu için tanımlanmıştır. Aynı şekilde diğer durumlar için de tasarlanmıştır. Aşağıda buradaki ASM chart için sembolik olarak mikroprogram gösterilmiştir.
-

TABLE 8-11
Symbolic Microprogram for Fetch and Execution of Six Instructions

Address	NXT ADD	MS	MC	IL	PI	PL	TD	TA	TB	MB	FS	MD	RW	MM	MW
IF	EX0	CNT	—	LDI	INP	NLP	—	—	—	—	—	—	NW	PC	NW
EXO	—	NXT	OPC	NLI	NLP	NLP	—	—	—	—	—	—	NW	—	NW
ADI	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	Constant	$F = A + B$	FnUt	WR	—	NW
LD	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	—	—	Data	WR	MA	NW
ST	IF	NXT	NXA	NLI	NLP	NLP	—	SA	SB	Register	—	—	NW	MA	WR
INC	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	—	$F = A + 1$	FnUt	WR	—	NW
NOT	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	—	$F = \overline{A}$	FnUt	WR	—	NW
ADD	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	SB	Register	$F = A + B$	FnUt	WR	—	NW

yukarıdaki tabloda neden bahsettiğimizi biraz daha açıklayalım.

- ADI komutunda yani add immediate. ne iş yaprdı hemen hatırlayacak olursak; bir register kaynağından gelen bir data ile sabit bir sayıyı toplayan ve onu da bir registera yazan bir komuttu. peki bunun için hangi girişler ne olmalı? lütfen şimdi yazacaklarımı yukarıdaki multiple cycle tasarım şeklinden takip edin .
    - bu toplama işleminde kaynak registerı olarak sadece BusA yı kullanabiliriz. bunun için toplanacak sayı SA girişinden belirtilir. sabit sayı ise zerofill den sonra Mux B ye uğramktadır. bu sayıyı alabilmemiz için MuxB deki seçme sinyali olan MB selectin constant ı yani sabit sayıyı seçmesi gerekir. Aslına bakacak olursanız bu da 1 dir. Daha sonra control unit in uçlarına ulaşan iki datayı toplama işlemine sokmak için FS selectin de toplama kodunu belirtmesi gerekir. control unitten çıkan sonucun alınabilmesi için MuxD yi kontrol eden MD selectin de function unit i belirtmesi gerekir ki aslına bakarsanız bunu sağlayan binary kod da 0 dır. Ve artık sonucun yazılacağı register da DA tarafından belirtilir. Yazma işleminin olabilmesi için de RW nin 1 olması gerekir.
- LD komutunun da ne iş yaptığını hatırlayalım.BusA dan gelen datanın işaret ettiği memory de ki datayı kaynak registerına yazıyordu. DR ve SA nın belirtilmesi gerekir. SB ye gerek yoktur. Memeory e adres gösterebilmemiz için Mux M nin BusA dan gelen datayı seçmesi gerekir. Mux D nin de memory den gelen datayı seçmesi gerekir.
- ST komutu da SA nın işaret ettiği adrese SB den gelen datanın yazılmasını emrediyordu. bunun için neler gerektiğini de şekle bakarak siz bulun hadi

bakalım herşeyi bizden bekkmeyin adam olun neyse bu böyle gider.(((yazar bazen saçmalar)))

Burada ilk önce adresler ve next adresler için state isimleri kullandık. Bu sembolik mikroprogramı binary mikroprograma dönüştürürsek aşağıdaki tabloyu elde ederiz

TABLE 8-12
Binary Microprogram for Fetch and Execution of Six Instructions

Address	NXT ADD	MS	MC	IL	PI	PL	TD	TA	TB	MB	FS	MD	RW	MM	MW
192	193	000	0	1	1	0	0	0	0	0	00000	0	0	1	0
193	000	001	1	0	0	0	0	0	0	0	00000	0	0	0	0
000	192	001	0	0	0	0	0	0	0	1	00010	0	1	0	0
001	192	001	0	0	0	0	0	0	0	0	00000	1	1	0	0
002	192	001	0	0	0	0	0	0	0	0	00000	0	0	0	1
003	192	001	0	0	0	0	0	0	0	0	00001	0	1	0	0
004	192	001	0	0	0	0	0	0	0	0	01110	0	1	0	0
005	192	001	0	0	0	0	0	0	0	0	00010	0	1	0	0

ilginçtir ki multiple cycle ı single cycle ile karşılaştırdığımızda single cycle bir komut için tek clock pulse kullanırken multiple cycle 3 clock pulse i kullanır. Peki bir clock pulse inde tüm işlemi halletmek varken neden multiple cycle kullanalım işte bu sorunun cevabı şimdi anlatacağımız komutların single cycle de olmaması

ilk komutumuz 0000110 kodu ile “load register indirect”(LRI) . Bu komutta SA registerını değeri memory de bir adrese karşılık geliyor. Daha sonra memorydeki bu adresin işaret ettiği registera gidiliyor. yani bir register değerine dolaylı yoldan ulaşmış olduk. sembolik olarak şu şekilde ifade edilebilir:

R[DR] = M[M[R[SA]]]

Bu kodun ASM chartı şu şekildedir

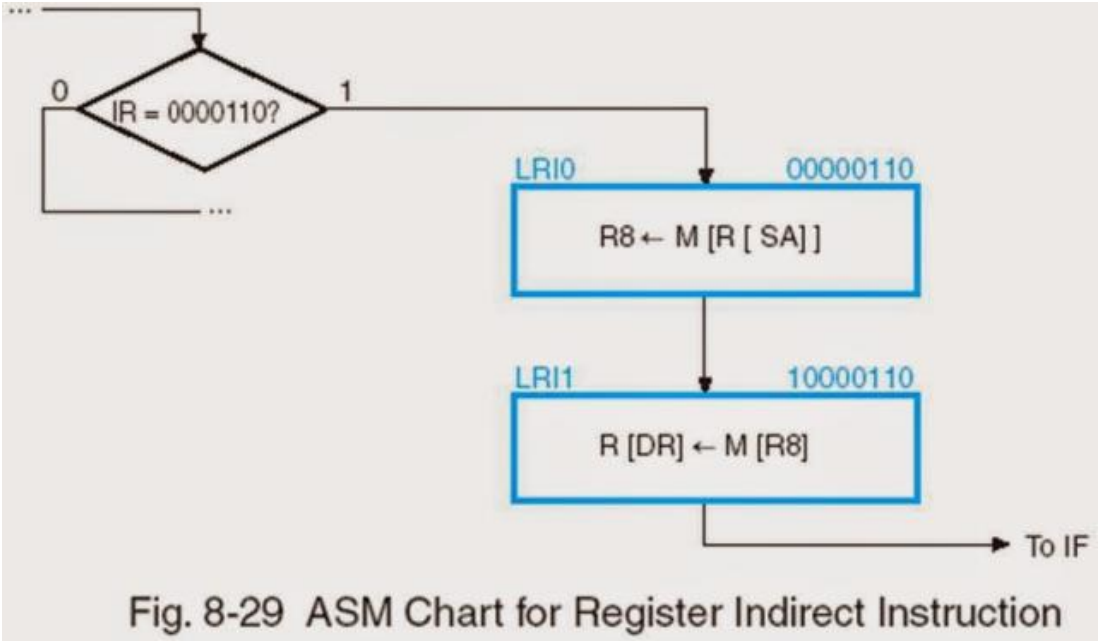
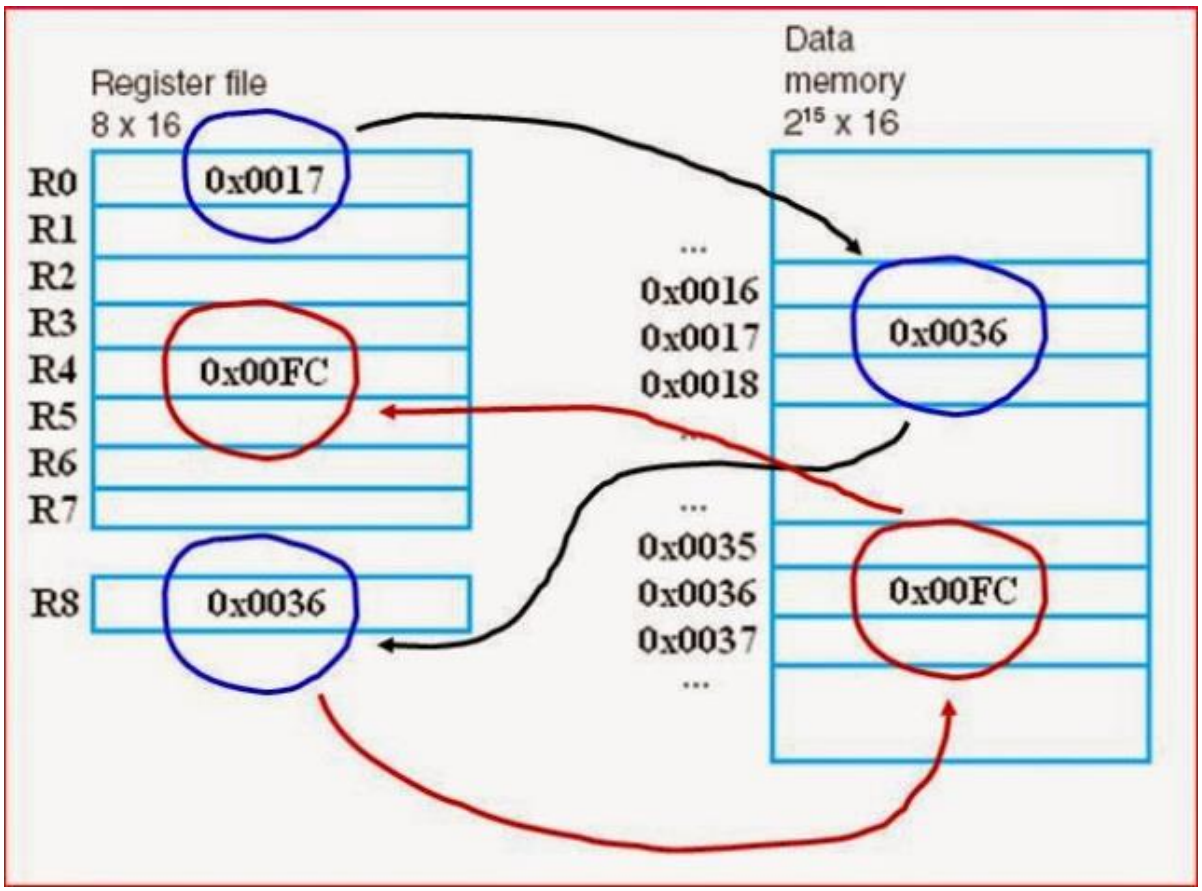
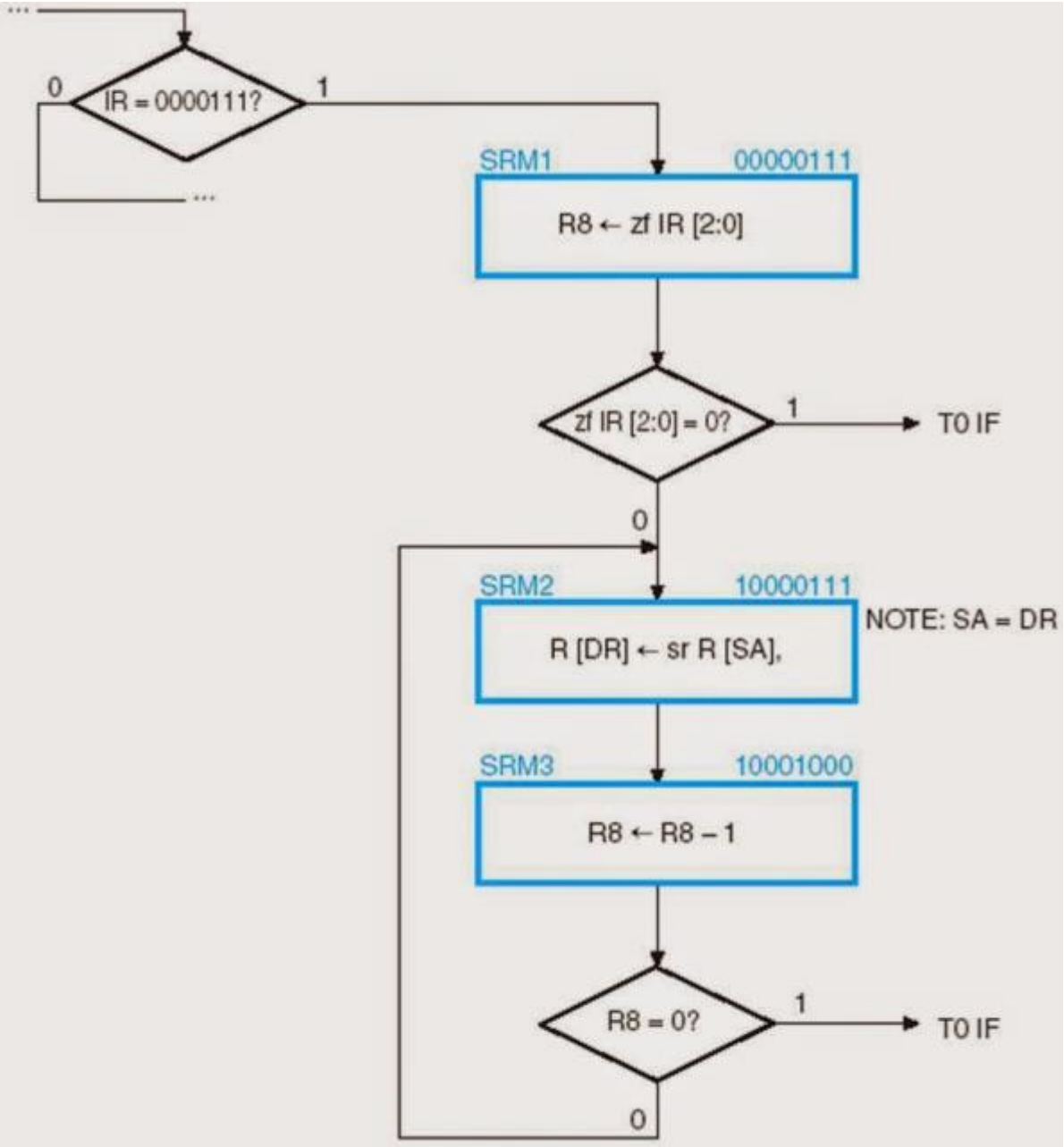


Fig. 8-29 ASM Chart for Register Indirect Instruction

Aynı işlemi 2 adet LD komutu ile yapabilirdik, ancak 6 clock gerekirdi. Böyle 4 clock gerekiyor. hala kafanızda bu işlem nasıl oluyor diye soru varsa aşağıdaki görseli inceleyebilirsiniz
örnek olarak
R[R4]çM[M[R[R0]]] işlemi için
İşlem sonunda R[R4]=0x00FC olur.



multiple cycle ı seçmemizin ikinci nedeni olan komut ise 0000111 opcode u ile “shift right multiple” komutudur. Bu komutun çalışması için mutlaka hedef registerı DA ile kaynak registerı SA birbirine eşit olması gerekmektedir. Aşağıda bu kodun ASM chartı bulunmaktadır



Register R8 kaydırılacak kalan bit pozisyon sayısını tutar.. İlk olarak, 0-7 arasında kaydırma miktarı R8 e yerleştirilir. Değer R8 e yüklendiğinde 0 mı değil mi diye kontrol edilir. eğer sıfırsa kaydırma işlemi yapılmayacak anlamına gelir. bu durumda state IF ye geri döner. eğer R8 0 değilse ilk önce bir adet right shift uygulanır ve R8 in değeri 1 azaltılır. daha sonra R8 tekrar kontrol edilir ve R8 sıfır oluncaya kadar bu döngü devam eder.

Bu instruction'ı single cycle computer yapısında yapmak mümkün değil.bu komut 2s+3 clock cycle gerektirir. s kaydırma miktarıdır. bu komut right shift komutu s defa kullanılarak yapılabiliridi. Ancak s>4 olduğu durumlarda bu komut daha hızlı çalışır.

Mikroprogramlı kontrolün (Microprogrammed control) ün bir avantajı; hardware yapısının bir defa oluşturulduktan sonra, başka hiçbir ek bağlantı ve elemana ihtiyaç duymadan,

mikrokomutlarla (microinstruction) istenilen program komutlarının (instruction) elde edilebilmesidir. Sadece kontrol memory'nin değiştirilmesi ile işlemler farklılaştırılabilmektedir.

İkinci avantajı, çok kompleks komutları gerçekleştiebilmek için daha büyük control belleği kullanmak yeterli iken, hardwired kontrolde , donanım oldukça karmaşıklaşabilir ve maliyetler oldukça artar.

Öte yandan eğer komutlar çok basit ise mikroprogram kullanmak lüks olabilir, hardwired kontrol daha ucuz olabilir.

8. Instruction Set Architecture

Bir bilgisayarda genelikle çeşitli komutlar ve birden fazla komut formatı(instruction format) bulunur. bunun basit örneklerini önceki kısımlarda işledik. Şimdi ticari amaçlı bilgisayarlarda bulunan tipik komutlarla bu gösterimleri genişletelim. Bitler field(alan) denilen gruplara ayrılır. instruction formatlarda bulunan tipik alanlar şöyledir:

1. Opcode field, sergilenecek işlemi belirler,
2. Address field, memorydeki bir adresi veya başka bir yerdeki adresi sağlar,
3. Mode field, yorumlanacak olan adres field ın yolunu belirler

Basic Computer Operation Cycle

Bir bilgisayardaki control unit programdaki her bir komutu aşağıdaki sırayla çalıştırmak için tasarlanmıştır.

1. memoryden komutu control register a fetch et,
2. komutu çöz
3. operandları komut kullanarak yerleştir
4. eğer gerekliyse memoryden operandları fetch et
5. işlemci registerında işlemi çalıştır
6. belli bir yerde sonuçları depola
7. sonraki komutu çalıştırmak için 1. adıma geri dön

Register Set

Register set, programcı tarafından erişilebilen CPU daki tüm registerlardan oluşmaktadır. CPU ayrıca instruction register ve register file dak registerlar gibi sadce mikroprogramcı tarafından görülebilen başka registerlar da içermektedir.

Bu register set gerçek bir CPU için comleks gelmeye başlayabilir. iki adet register ekleriz: Processor Status register (PSR) ve Stack pointer (SP).

Processor Status register ALU dan C,N,V,Z değerleri tarafından seçilen flip flopları içerir. bu durum bitleri programın akışını belirleyen kararları vermek için kullanılır.

Operand Addressing

X=(A+B)(C+D) işlemini3,2,1 ve 0 adres kullanarak değerlendirelim.

Three-address Instructions

ADD T1,A,B M[T1] ß M[A] + M[B]

ADD T2,C,D M[T2] ß M[C] + M[D]

MUL X,T1,T2 M[X] ß M[T1] x M[T2]

Bu formatın avantajı kodların kısa olması , dezavantajı ise yazılacak binary kodların çok fazla bite sahip olması

Two-address Instructions

MOVE T1,A M[T1] ß M[A]
ADD T1,B M[T1] ß M[T1] + M[B]
MOVE X,C M[X] ß M[C]
ADD X,D M[X] ß M[X] + M[D]
MUL X,T1 M[X] ß M[X] x M[T1]

One-address Instructions

LD A ACC ß M[A]
ADD B ACC ß ACC + M[B]
ST X M[X] ß ACC
LD C ACC ß M[C]
ADD D ACC ß ACC + M[D]
MUL X ACC ß ACC x M[X]
ST X M[X] ß ACC

ACC= akümülatör. bir operand ve işlem sonucunun yazılacağı adresi tutar. Tüm işlemler ACC registerı ile memory operand arasında olur.

Zero-address Instructions

Bir ADD komutunu gerçekleştirmek için tüm üç adres de gizlenmek zorundadır. bu amaca ulaşmanın uygun bir yolu stack(yığın) denilen yapıyı kullanmak. bu stack şu söze göre çalışır " son giren ilk çıkar"

Bu stack ın en tepesindeki değere (top of stack) TOS denilmiştir. ondan sonra gelen TOS-1 denilmiştir. bu formatta iki adet komut kullanılır

PUSH X TOS ß M[X] yığını yükler
POP X M[X] ß TOS yığından çıkarır.

Yine aynı işlemi yapalım

X=(A+B)(C+D)

PUSH A TOS ß M[A]
PUSH B TOS ß M[B]
ADD TOS ß TOS + TOS_{-1}
PUSH C TOS ß M[C]
PUSH D TOS ß M[D]
ADD TOS ß TOS + TOS_{-1}
MUL TOS ß TOS x TOS_{-1}
POP X M[X] ß TOS

Addressing Architectures

Bunlar arasından three adress i seçtik. Tüm operandlar memoryden direk geliyor ve işlem sonucu direk olarak memory e gönderiliyor.Önceki örnekte bu seçenek için 3 adet komut kullanmıştık. fakat her memory adres için komutlarda extra bir word görünmek zorunda kalırsa bu sayı 4 e çıkar. komutları fetch etme ve sonucu depolamak için memeory e erişim için 21 adet access gerekir. bu da 21 clock pulse demektir. Komut sayısı az ancak omutun çalıştırılması uzun sürmektedir. bu yapı mimarisi yeni tasarımlarda bu yüzden kullanılmamaktadır. buna karşın *three adress register to register* veya l*oad/store* mimarisi yalnızca tek adres kullanılarak yapılmaktadır. Bu tasrımda şu kodlar karşımıza çıkar;

X=(A+B)(C+D)
LD R1,A R1 ß M[A]
LD R2,B R2 ß M[B]
ADD R3,R1,R2 R3 ß R1 + R2
LD R1,C R1 ß M[C]
LD R2,D R2 ß M[D]
ADD R1,R1,R2 R1 ß R1 + R2
MUL R1,R1,R3 R1 ß R1 x R2
ST X,R1 M[B] ß R1

Toplamda 8 komut bulunmaktadır ve memory e erişim 21 den 18 e düşmüştür.

ADDRESSING MODES

Bilgisayarlar aşağıdaki iki adres modundan birini veya ikisini birden kullanabilir.

1. Kullanıcıya program esnekliği kazandırmak için memory e pointer lar vasıtasıyla ulaşmak
2. instruction daki adres alanının bit saysını azaltmak

Bu çeşitli adresleme modlarının kullanılabilirliği birkaç komutla program yazma kabiliyetini programcıya kazandırır.

Bazı bilgisayarlarda komutun adresleme modu ayrı bir binary kod ile kontrol edilir. Diğer bilgisayarlar ise işlemi ve adresleme modununun ikisini birden design eden yaygın bir binary kod kullanır. Ayrı adresleme modu alanı ile birlikte bir instruction format örneği şu lekildedir.

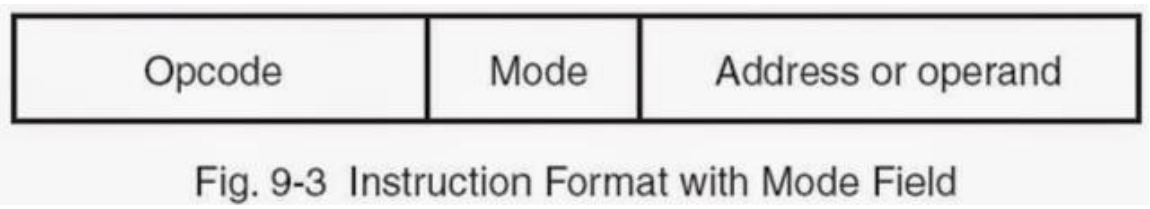


Fig. 9-3 Instruction Format with Mode Field

Opcode sergilenecek olan işlemi tayin eder. Mode alanı ise işlem için gerekli olan operandları yerleştirmek için kullanılır. instruction içinde adres alanı olabilir de olmayabilir de. Eğer bir adres alanı varsa bir memory adresini veya bir işlemci register ını belirtir. Daha

da ötesinde, bir instruction bir adres alanından daha fazla alana sahip olabilir. Bu durumda her adres alanı kendi özel adres modu ile ilişkilendirilir.

İmplied mode

Bu modda bir adres alanına ihtiyaç yoktur. Bu modda operand, opcode un tanımında dolaylı olarak belirlenir. Daha önce adı geçen akümülatör buna bir örnektir.

Immediate mode

Bir immediate mode instruction da adres alanı yerine operand alanı bulunmaktadır. Bu kullanışlıdır örneğin başlangıç işemlerinde register lara sabit değerleri yükleme gibi

Register and register-indirect modes

instruction daki adres alanı bir memory location ı veya bir işlemci register ını belirleyebilir. Adres alanı işlemci registerlarını belirlediği zaman instruction un register modda olduğunu anlarız. Bu modda operandlar bilgisayarın işlemcisi içine yerleşen register lar içindedir. Registerlar instruction format içinde bulunan register adres alanından seçilir.

Register indirect modda ise , instruction, içeriği memorydeki operandın adresini veren işlemcideki register ı belirler. Diğer bir ifade ile seçilen register, operandın kendisini direk olarak tutmaktansa operandın memory deki adresini tutar. register-indirect mode instruction u kullanmadan önce programcı mutlaka işlemci register içinde bulunan kullanılabilir memory adresi sağlamalıdır. Bu modun avantajı instruction daki adres alanının bir register seçmek için birkaç bit kullanıyor olması . Eğer operanda direk erişim sağlansaydı bit sayısı artacaktı.

Autoincrement veya autodecrement mode ise register ın, memory e erişmek için adres kullanımından sonra veya önce arttırılma işlemi hariç register-indirect mod ile benzerdir. bir örnek biraz daha netlik kazandırır umarım

ADD (R1)+, 3 M[R1] ß M[R1] + 3, R1 ß R1+1

Direct addressing mode

Şekil üzerinden konuşalım

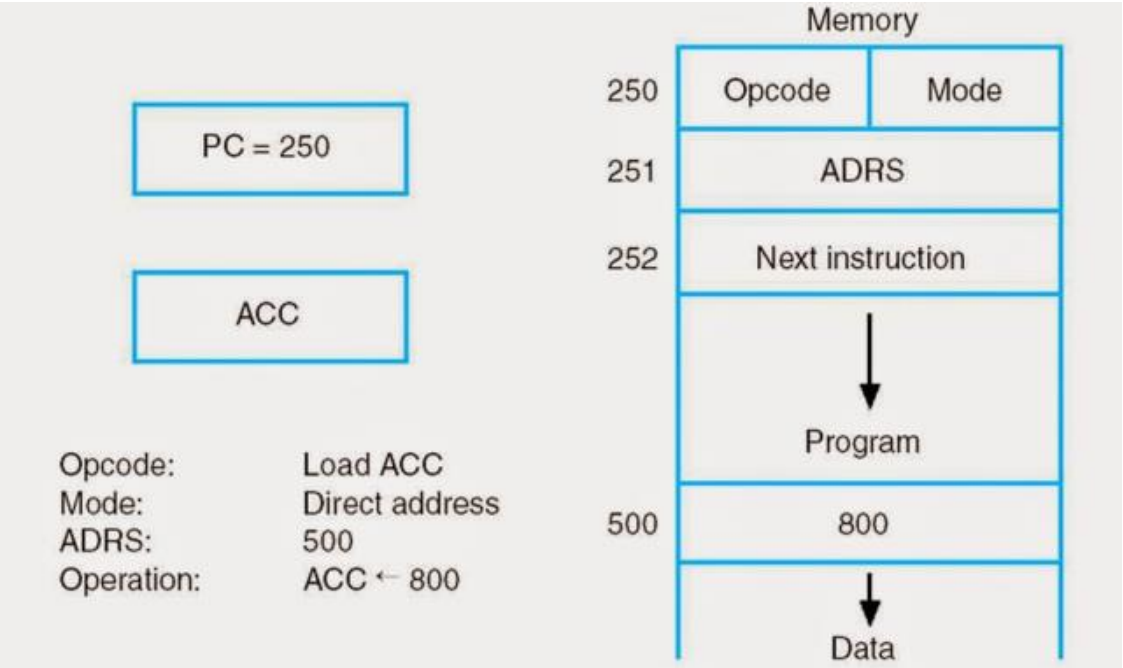


Fig. 9-4 Example Demonstrating Direct Addressing for a Data Transfer Instruction

memory deki instruction iki word den oluşmukta, İlki, adres 250 de , "load to ACC" için opcode ve direct adresi belirleyen mode alanı. İnstruction daki ikinci word ise , adres 251 de, ADRS olarak sembolize edilen ve 500 değerini tutan adres alanını içerir. PC ise iki memory erişimi sağlanarak memoryden getirilen komut adresini tutar. Aynı anda veya ilk erişim sağlandıktan sonra PC değeri 251 e artırılır. Komutun çalışmasının sonucu şu işlemle gösterilibilir

ACC ß M[ADRS]

ADRS=500 ve M[500]=800 olduğundan , ACC 800 numarasına ulaşır. Bu komut çalıştırıldıktan sonra program içindeki sonraki komutun adresi olan 252 numarasını tutar.

Şimdi aşağıda gösterilen bir branch-type instruction düşünelim.

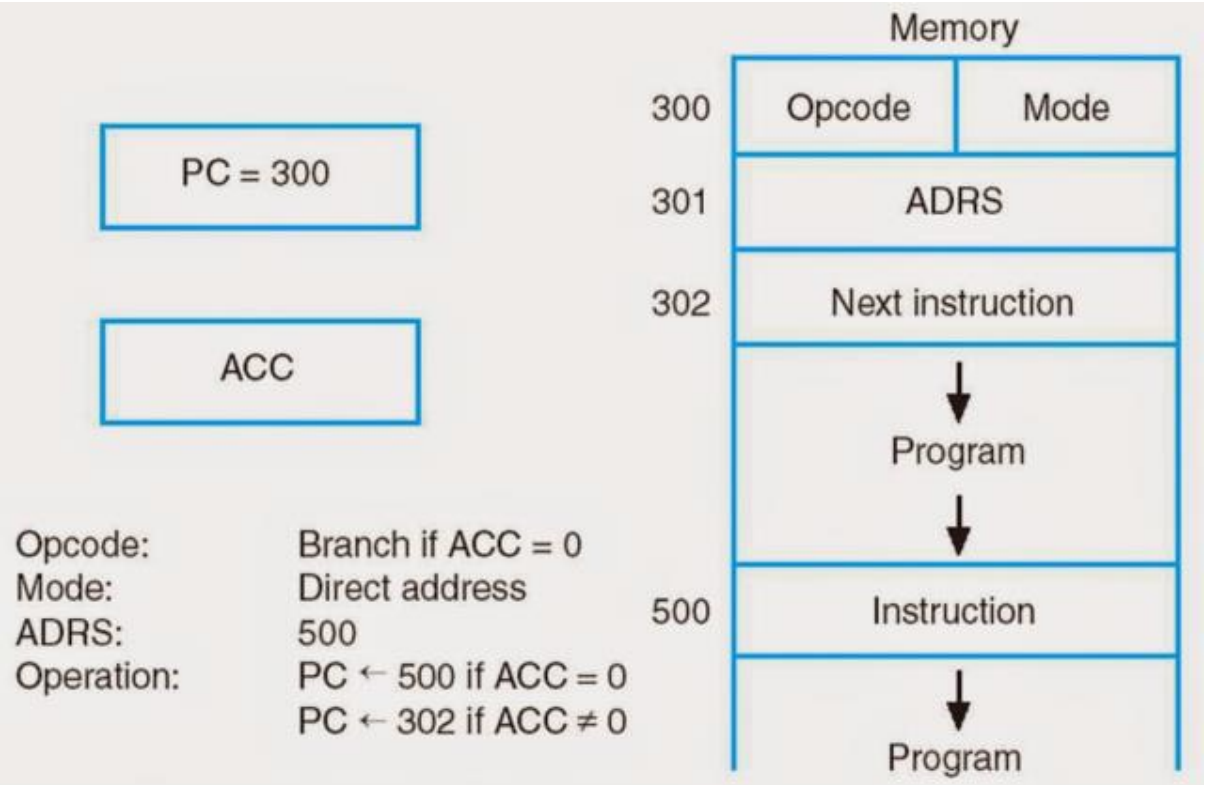


Fig. 9-5 Example Demonstrating Direct Addressing in a Branch Instruction

Eğer ACC nin içeriği 0 a eşitse ADRS de olan değere gidilir aksi takdirde program sıralı olarak next instruction ile devam eder.

Indirect addressing mode

instruction daki adres alanı memoyde depolanan efective adresdeki adresini verir. Şekilde Mode kısmında direct adres yazıyor ama bu örnek için indirect yazdığını düşünün.

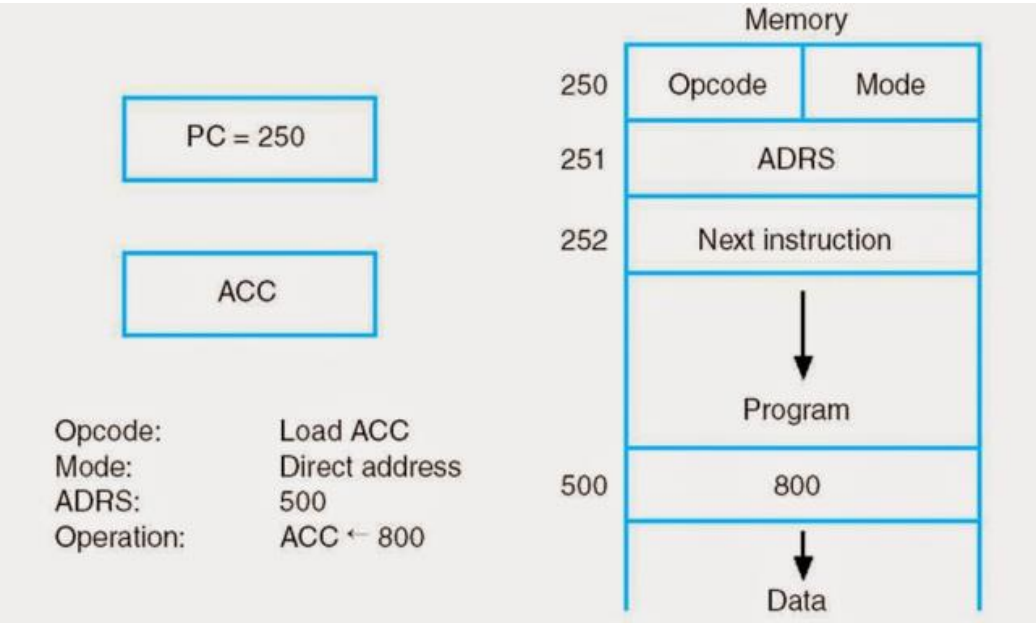


Fig. 9-4 Example Demonstrating Direct Addressing for a Data Transfer Instruction

ADRS=500 ve M[ADRS]=800 olduğundan , effective adres 800 dür. bu şu anlama gelir

ADRS=500
M[ADRS]=800
ACC ß M[800] şekilde 800. adres gösterilmemiştir. 800. adresdeki bilgi ACC ye yüklenir.

Relative addressing mode
Effective address=PC içeriği + instruction ın Address kısmı

Açıklık getirecek olursak;
Yukarıdaki şekilde PC=250 ve ADRS=500 olduğunu varsayarsak
PC=252+500=752 olacaktır.

<u>Adres modlarının özeti</u>
Aşağıdaki şekilde gösterilen komutlarda farklı modların etkisini araştıracağız.

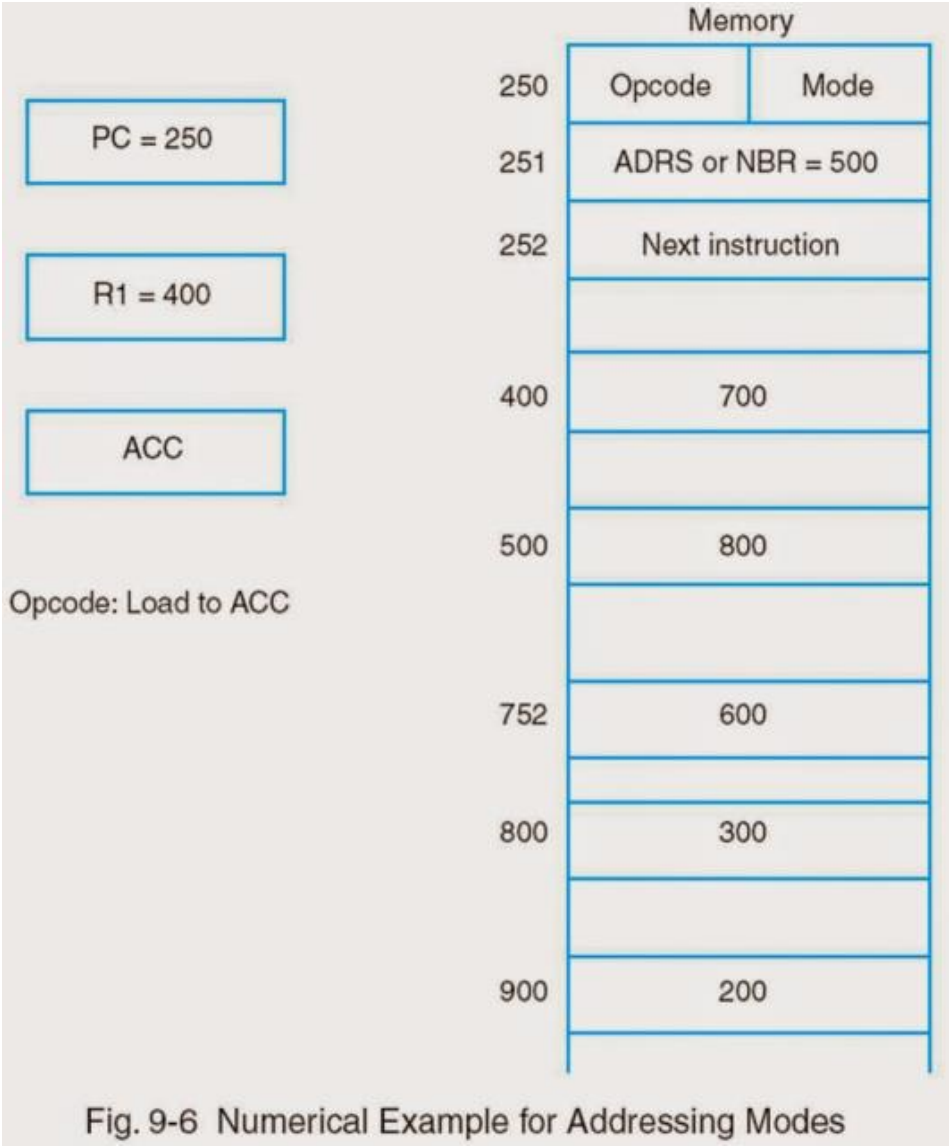


Fig. 9-6 Numerical Example for Addressing Modes

- Direct modda effective adres 500 ve ACC ye yüklenecek operand 800
- İmmediate modda ACC ye yüklenecek operand 500
- İndirect modda effective adres 800 ve operand 300
- relative modda effective adres 752 ve operand 600
- index modda effective adres 900 ve ACC ye yüklenecek değer 200 (R1 index register varsayılmıştır)
- register-indirect modda efective adres R1 in içeriği olur ve ACC ye yüklenecek değer 700 dür

Aşağıdaki tablo bunları göstermektedir

TABLE 9-1
Symbolic Convention for Addressing Modes

			Refers to Figure 9-6	
Addressing mode	**Symbolic convention**	**Register transfer**	**Effective address**	**Contents of *ACC***
Direct	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Immediate	LDA #NBR	$ACC \leftarrow NBR$	251	500
Indirect	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relative	LDA $ADRS	$ACC \leftarrow M[ADRS+PC]$	752	600
Index	LDA ADRS (R1)	$ACC \leftarrow M[ADRS+R1]$	900	200
Register	LDA R1	$ACC \leftarrow R1$	—	400
Register indirect	LDA (R1)	$ACC \leftarrow M[R1]$	400	700

Tablodaki LDA load-to-accumulator opcode için semboldür.

Eveeeet şimdi bunları özetleyen bir örnek yapma zamanı geldi..

ÖRNEK1:

aşağıdaki tablolarda komutlar, registerların durumu ve memory içerikleri verilmiştir ve sıralı olarak bu kodlar çalışacaktır.içerikler hex formatında gösterilmiştir. gerçekleşecek mikroişlemleri ve sonucundaki değişiklikleri bulalım.

ADDRESS	CONTENTS	COMMENT
00	ADD	Direct 2 address
01	26	
02	28	
03	LD	İndirect 1 address
04	2F	
05	MUL	İndirect 1 address
06	2E	
07	PUSH	Relative 1 address
08	40	
09	PUSH	Relative 1 address
0A	40	
0B	MUL	0 adress
0C	POP	Register 1 adress
0D	2	
0E	MUL	Register 1 adress
0F	2	

PC	00
ACC	00
SP	A7
R1	00
R2	0F
R3	00

register içeriği

24	02
25	05
26	03
27	07
28	01
29	0A
2A	1B
2B	32
2C	04
2D	35
2E	2C
2F	24
40	61
41	0D
42	06
48	02
49	04
4A	05
4B	0A

memory içeriği

Şimdi neler mi olacak gelin beraber inceleyelim

- ilk olarak ADD komutu gerçekleşecektir. yanında direct 2 address yazıyor bu demek oluyor ki bu işlem için alttaki iki adresi kullan sonucu da ilk adrese yaz. 26. ve 28. adreslere bakıyoruz ve 03 ve 01 datasını buluyoruz ikisini toplarsak 04 olur bunu da tekrar 26. memory hücresine yaz.
    - M[26]=M[26]+M[28]
- daha sonra LD komutu gerçekleşir. yanında indirect adress yazıyor. bu adresleme metoduna yukarıda değindiğimiz gibi dolaylı yoldan dataya ulaşıyorduk. alttaki adresi

indirect adres olarak kullanıyoruz. Bakıyoruz 2F yazıyor. Memory deki 2F e bakıyoruz 24 yazıyor.daha sonra memory deki 24. adrese gidiyoruz ve 02 datasını buluyoruz bunu da ACC yani akümülatöre yüklüyoruz

- ACCßM[M[2F]] ; ACCß02(hex)

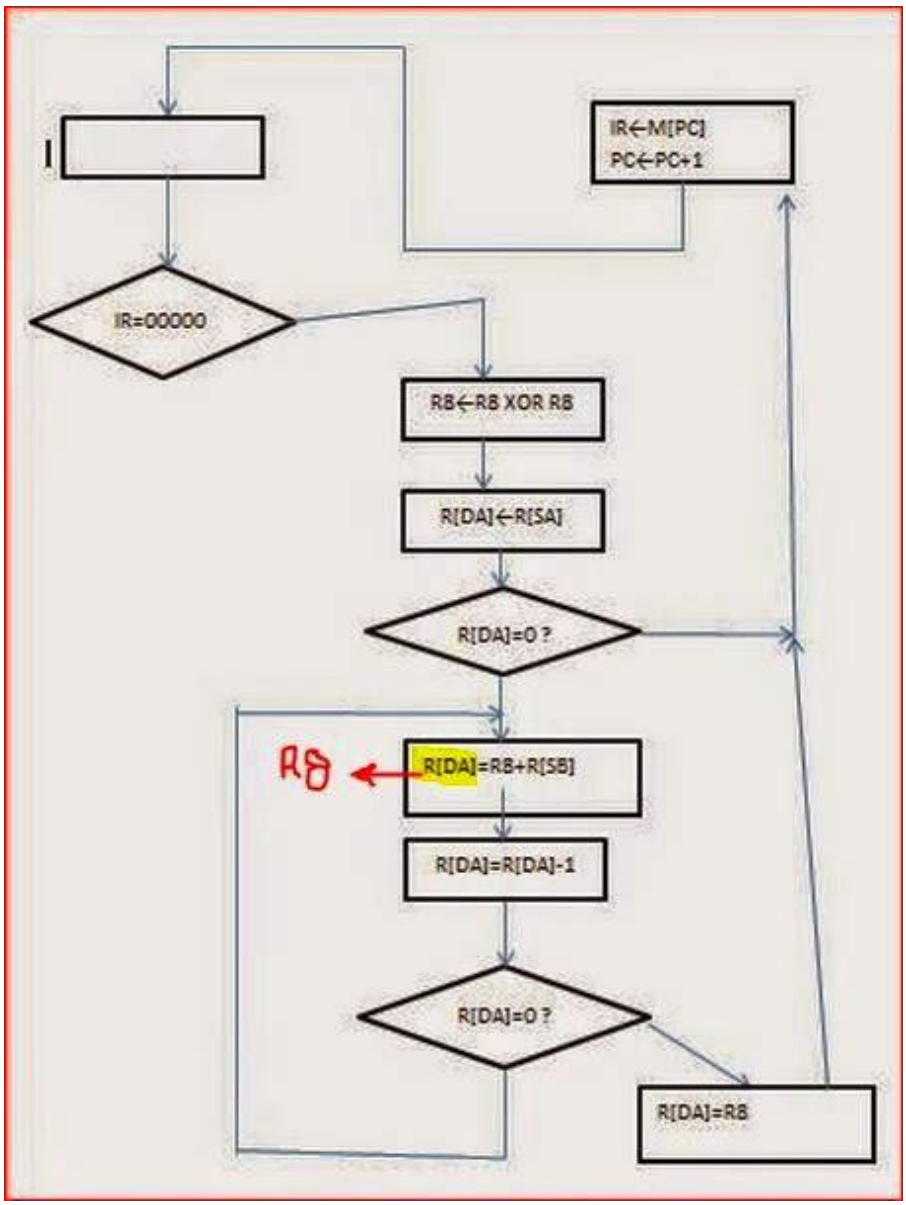
- daha sonra MUL komutu gerçekleşir. yanında yine indirect yazıyor. yine aynı metodla aşağıdaki işlemin gerçekleşeceğini buluruz
  - ACCß ACCX [M[M[2E]] ; 02X04 ; ACCß08(hex)
- daha sonra PUSH komutu gerçekleşir. yanında reletive 1 address yazıyor. PUSH komutu sonraki adresi algılar. Dolaysıyla aşağıdaki işlem gerçekleşir
  - TOSßM[40+09]; M[49]=04
- daha sonra Yine PUSH komutu gerçekleşir. aynı mantıkla gidecek olursak;
  - TOSß[40+0B]; M[4B]=0A
- daha sonra MUL komutu gerçekleşir. yanında 0 adres yazıyor. bu demektir ki Push-pop mimarisindeki tos ve tos-1 adresleri işleme sok.
  - TOSßTOSXTOS-1 ;0AX04=28(hex)
- daha sonra pop komutu gerçekleşir. yanında register 1 address yazıyor. Bu da aşağıdaki adresin gösterdiği registerı pop komutu için kullan demektir.
  - R2ßTOS=28(hex)
- daha sonra MUL komutu gerçekleşir. yanında register 1 adres yazıyor. bu da mul işelmi için aşağıdaki adresin göstediği registerı kullan demektir
  - ACCßACCxR2 ; 08(hex) x 28(hex) =140

olur…

ÖRNEK2;

yukarılarda bir yerlerde resmi olan multiple cycle microprogrammed computer şeklini baz alalım. Bunun içine işaretsiz binary sayıları tekrar toplama metodu ile çarpma işlemi yapan bir mikroprogram yazılmıştır. .örneğin 5×4 işlemini ele alırsak 5+5+5+5 işelmini yapmamız gerekecek. Çarpılan sayı R[SA] da, çarpan sayı R[SB] de ve çarpımda R[DA] da tutulsun.

a) bu komut için ASM chart oluşturun. R[SA] =4 ve R[SB]=5 varsayın



tekrar şekli çizmeye üşendiğimden yanlışı üzerinde düzelttim R[DA] yerine R8 olacak..
kendi el yordamımla word de birşeyle çizdim ama açılayarak çizimi tamamlamak istiyorum.
İlk önce normal multiple cycle işlemlerinin başlaması için pc yüklemesi falan var daha sonra R8 i R8 le xor layıp R8 e yeniden yazdık. Peki bunun amacı neydi? bunun amacı R8 in içindeki datayı sıfırlamak. Bir sayıyı kendisiyle xor işelmeine sokarsanız sonuç 0 olcaktır. Daha sonra

SA nın belirlemiş olduğu registerdaki değeri DA ya yazdık yani çarpılanı. SA da 4 vardı. eğer bu değer sıfırsa zaten işleme sokmaya gerek yoktur çünkü çarpma işleminde 0 la çarparsanız

sonuç sıfır olacaktır çok ilginç değil mi eğer sıfır değilse işleme sokacağız. peki nedir bu işlem ? çarpılanı çarpan kere kendisi ile toplamak . DA ya R8+SB demişiz.ilk seferde 0+5=5 olacaktır daha sonra DA nın değeri 1 azaltılıyor eğer DA sıfır değilse tekrar işlem tekrarlanıyor.R8 in içinde artık 5 var bu kez 5+5=10 işlemi gerçekleşecek daha sonra 10+5=15 daha sonra 15+5=20 olduğunda DA nın değeri de her saferde 1 azaltığından artık 0 a gelmiştir ve döngüden çıkılır. Çarpmanın sonucunun DA da tutulması istendiğinden en son R8 de biriken sonucun DA ye yazılma işlemi vardır.

b) bu komut için gerekli kodlar sembolik olarak ifade edelim

1. PUSH R8 TOS←R8
2. PUSH R8 TOS←R8
3. XOR TOS←TOS XOR TOS-1
4. POP R8 R8←TOS
5. PUSH SA TOS←SA
6. POP DA DA←TOS R[DA]←R[SA]
7. PUSH R8 TOS←R8
8. PUSH SB TOS←SB
9. ADD TOS←TOS+TOS-1
10. POP R8 R8←TOS
11. PUSH DA TOS←DA
12. INC TOS←TOS-1
13. POP DA DA←TOS R[DA]←R[DA]-1
14. PUSH R8 TOS←R8
15. POP DA DA←TOS R[DA]←R8 SONUÇ R[DA] YA YAZILDI.

c) bu tasarım için mikro komut programını yazalım

ADRES	N.ADRES	MS	MC	IL	PI	PL	TD	TA	TB	MB	FS	MD	RW	MM	MW
192	193	000	0	1	1	0	0	0	0	0	00000	0	0	1	0
193	000	001	1	0	0	0	0	0	0	0	00000	0	0	0	0
7	8	000	0	0	0	0	1	1	1	0	01100	0	1	0	0
8	192	100	0	0	0	0	0	0	0	0	00000	0	1	0	0
9	10	000	0	0	0	0	1	1	1	0	00010	0	1	0	0
10	9	111	0	0	0	0	0	0	0	0	00110	0	1	0	0
11	192	001	0	0	0	0	0	0	0	0	00000	0	1	0	0

Asm chartta state lere adres vermeyi unuttuğumdan burada veriyorum;

192 pc nin 1 artrıldığı state in adresi
193 boş bıraktığım ama aslında daha önce bahsattiğim IR nin soluna bir bit 0 eklendiği state adresi
7 adresi R8 in xor işemine girdiği state in adresi
8 adresi R[DA] ya R[SB] nin yazıldığı state in adresi
9 adresi R8 e R8 + R[SB] işleminin olduğu state adresi
10 adresi R[DA] nın 1 azaltığı statein adres
11 adresi R8 in R[DA] ya yazıldığı state adresi

artık ne olduğuna bakabiliriz

- 192 den 193 e geçiş sırasında herhangi bir koşul yoktur. Bunun için MS=000 olarak belirtilmiştir.hatırlayacaksınız ki MuxS den gelen çıkış 0 ise car 1 arttrılıyordu 1 ise car yükleniyordu. burada 192 den 193 e geçmesi için car ın 1 artması yeterlidir. MS 000 olduğunda muxS 0 sinyalini çıkışa verecektir.dolayısıyla diğer komut adresimiz 193 olacaktır
- 193 adresinde MS 001 dir. dolayısıyla muxS 1 sinyalini çıkışa verecek ve car 1 arttırlmayıp muxC den gelen data ile yüklenecektir. MuxC yi kontrol eden MC nin 1 olduğunu görüyoruz. Bu demektir ki car IR den gelen data ile yüklenecektir. bu da dolaylı olarak PC nin işaret ettiği adresin yüklenmesi demektir.
- PC değeri 1 artarak 7 adresine ulaşır. Asm chartı incelersek daha sonraki state e koşulsuz geçiş vardır. bu yüzden car ın 1 arttırılması bize yetecektir. bunun için MS 000 dır. .Ayrıca bu adreste yapılan işlem R8 i xor lamaktır. R8 i TD,TA,TB registerları aktif olduğunda kullanabildiğimizi hatırlayın. FS ise xor işlemini işaret eder
- PC değeri 1 artarak 8 adresine ulaşmıştır. Burada bir koşul vardır. eğer R[DA] 0 sa başa dönüyor değilse 1 artarak 9 adresine gidiyor. burada mantıklı olanı status registerını kullanmaktır. eğer Z bayrağı 1 se R[DA] 0 demektir ki bu durumda next adres yüklenir next adres de 192 olduğu için başa döner bu da bizim istediğimizi karşılar. eğer Z bayrağı sıfırsa R[DA] sıfır değildir bu durumda Z 0 olduğu için car 1 arttırılarak adres 9 a gider ki bu da bizim hoşumuza gider :). o halde MS i Z bayrağını seçecek şekilde yani 100 yaparız.MC 0 olmalıdır ki car next adresle yüklensin Ayrıca 8 adresinde R[DA] ya R[SA] yüklemesi vardır.
- adres 9 da ise koşulsuz olarak adres 10 a gidileceğinden car ın 1 artması bize kafi gelecektir. car ın 1 artması için MS nin 000 olması gerekir. Ayrıca burada R[DA] nın 1 azaltılması işlemi vardır.FS nin bu işlemi sağladığını kontrol ediniz.
- adres 10 da önümüzde iki seçenek vardır. Eğer DA 0 sa adres 11 e değilse tekrar 9 a gidecektir. bunun için yine status registerlarına gideriz. bunu sağlayacak olan Z' bayrağıdır. kuralımız neydi? MS nin çıkışı 1 se car next adresle yükleniyordu 0 sa pc 1 artıyordu. Z' bayrağı ne zaman 1 olur? sonuç 0 olmadığında. Sonuç sıfır değilse adres 9 a gitmemiz gerekir. bu yüzden next adrese 9 yazdık. Aksi takdirde pc 1 artarak 11 e gidecektir.dalayısıyla MS 111 olacaktır ayrıca MC 0 olmalıdır ki car next adresle yüklensin . bu işlemi Z bayrağı kullanarak yapamaz mıydık neden ters yollara gidip

fantezi yaptık? eğer Z bayrağını kullanarak yapsaydık next adrese 11 yazmamız gerekecekti. tama burda sıkıntı yok . Amaaaa!!! aksi takdirde adresin 9 a gitmesi lazım. burada pc 1 artarsa yine 11 oluyor sıkıntı burda işte !! ayrıca adres 10 dayken yapılan işlemleri tablodaki değerler ile kontrol ediniz. bu değerleri daha önce datapath işlemlerinde nasıl yaptığımızı gösterdiğimiz için burada sadece adresleme işlemine değiniyorum…

- adres 11 de yine koşulsuz geçiş vardır ancak burada 1 arttırlması değil 192 ye gitmemiz gerekiyor bu yüzden car 1 olması gerekir. bunu da MS 001 olduğunda sağlarız. MC 0 olmalıdır ki car next adresle yüklensin.
- NOT: Artık derslerimizin sonuna ulaştık. Sadece notlar içinde daha sonra eklnecek dediğim yerler eklenecek. Her ne kadar bu iş burda bitmez ise de en azından biraz ufkumuzun açılacağını düşünüyorum. Risc ve cisc mimari, pipelend mimari gibi yerlere değinmedim ama bunları bir şekilde öğrenmenizin yararı olacaktır. Umarım yararı olmuştur.