2 İş Hattı (*Pipeline*)

İş hattında (*pipeline*) birden fazla iş (örneğin komutlar) paralel olarak aynı anda yürütülürler.

Bir iş hattının verimli olarak çalışabilmesi için

1. Farklı veriler üzerinde defalarca tekrarlanan işler (*task*) olması gerekir,
2. İşler paralel yürütülebilen küçük alt işlere bölünebilmeli.

İş hattına **örnek**: Bir otomobil fabrikasındaki üretim/montaj bandı

Burada iş/görev (*task*) bir otomobilin montajının yapılmasıdır.

Bu iş farklı otomobiller için sürekli tekrar edilir.

İş (otomobilin montajı), küçük alt işlemlerden oluşur; kapıların takılması, tekerleklerin montajı, camların takılması.

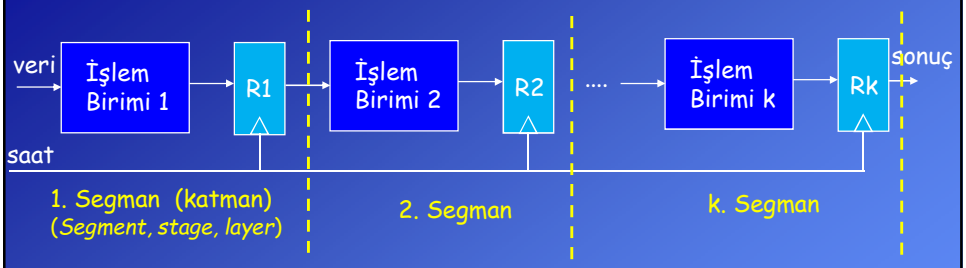
Bu alt işlemlerin her biri için iş hattında (montaj bandı) bir istasyon oluşturulur.

Bu istasyonlarda aynı anda paralel olarak farklı otomobiller üzerinde çalışılır.

Örneğin i. işçi bir otomobilin camını takarken aynı anda (i+1). sıradaki işçi bir önceki otomobilin tekerleklerini takmaktadır.

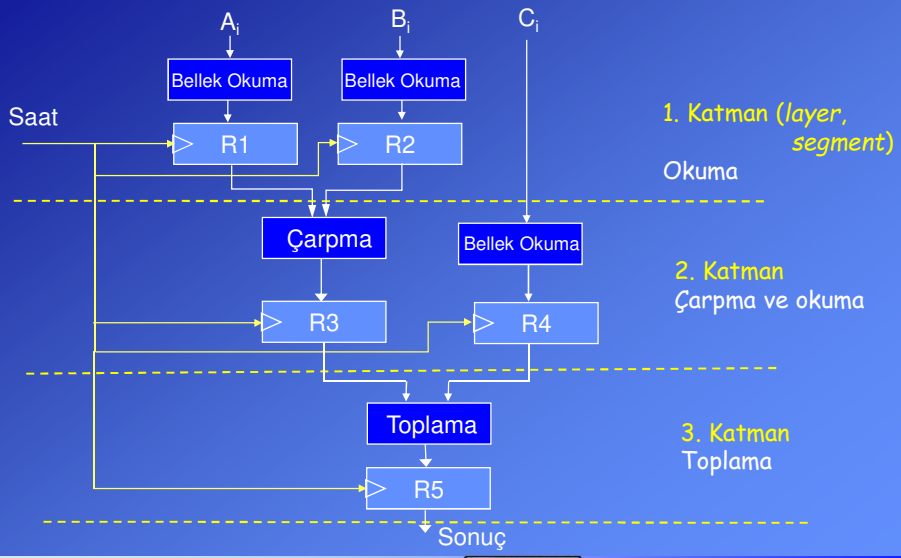
Örnek: Bir otomobil fabrikasındaki üç istasyonlu üretim/montaj bandı

2.1 Bir iş hattının genel yapısı:



- Her katman (işlem birimi) belli, sabit bir işi yapar.
- Her saat çevriminde (*clock cycle*) işlem birimi farklı veriler (iş) üzerinde çalışır. (*Saat işareti konusunda Sayısal Devreler Ders Notları Bölüm 6'da bilgi bulabilirsiniz.*)
- R1, R2,..., Rk gibi saklayıcılar ara sonuçları tutarlar.
- Tüm segmanlar ortak bir saat işareti ile denetlenirler ve eş zamanlı çalışırlar.
- Bir önceki verinin bütün adımları tamamlanmadan (sonuç üretilmeden) önce iş hattının girişinden yeni veriler alınır.
- İş hattının bütün segmanları dolduktan sonra her saat çevriminde çıkışta yeni bir sonuç üretilir.

Örnek: A, B ve C dizisinin elemanları önce bellekten okunacak ardından aşağıdaki işlem yapılacaktır. $A_i*B_i + C_i$ i=1,2,3,....



Örnek (devamı):

- Bu örnekte görev üç alt işleme bölünmüştür: Okuma, çarpma, toplama
- Dizilerin farklı bellek birimlerinde bulunduğu ve paralel olarak aynı anda okunabildiği varsayılmıştır.
- C dizisinin elemanlarının okunmasına bir saat darbesi sonra başlanmaktadır.

Üç segmanlı olarak tasarlanan iş hattının çalışması:

Saat Çevrimi	1. Segman		2. Segman		3.Segman	
	R1	R2	R3	R4	R5	
1	A_1	B_1	-	-	-	
2	A_2	B_2	A_1*B_1	C_1	-	
3	A_3	B_3	A_2*B_2	C_2	$A_1*B_1 + C_1$	(İlk sonuç)
4	A_4	B_4	A_3*B_3	C_3	$A_2*B_2 + C_2$	
5	A_5	B_5	A_4*B_4	C_4	$A_3*B_3 + C_3$	

Not: Verinin önceden hazır olduğu veya bellek okuma süresinin diğer işlemlere göre çok kısa olduğu sistemlerde bellekten okuma ayrı bir alt işlem olarak ele alınmaz.

Bu durumda sadece aritmetik işlemi yapan iş hattı 3 yerine 2 katmanlı olarak tasarlanabilirdi.

2.2 Dört Segmanlı Bir İş Hattının Uzay-Zaman Diyagramı (*Space-Time Diagram*)

Bir iş hattında belli bir anda hangi işin hangi segmanda işlem gördüğünü göstermek için uzay-zaman diyagramları (zamanlama diyagramı) kullanılır.

Aşağıdaki örnek tabloda, saat çevrimleri (adımlar) sütunlara, segmanlar satırlara, o anda yapılan iş (*task*) (veya işleme giren veriler) de tablonun içine yazılmıştır.

Örnek:
(4 segman)

Zaman → Saat Çevrimi (adımlar)

Segman	1	2	3	4	5	6	7
1	T1	T2	T3	T4	T5	T6	
2		T1	T2	T3	T4	T5	T6
3			T1	T2	T3	T4	T5
4				T1	T2	T3	T4

1nci iş (T1) 4 saat çevrimi (segman sayısı k=4) sonunda tamamlandı.

k.'dan sonraki her saat çevriminde yeni bir iş tamamlanır.

Dört iş (T4) 7 saat çevriminde tamamlanmıştır.

Dört Segmanlı Bir İş Hattının Uzay-Zaman Diyagramı (*Space-Time Diagram*) Devamı

Uzay-zaman diyagramları farklı şekilde de oluşturulabilir.

Aşağıdaki diyagramda saat çevrimleri (adımlar) sütunlara, veriler (işler) satırlara, o anda etkin olan segman da tablonun içine yazılabilir.

Zaman ⟶ Saat Çevrimi (adımlar)

İşler	1	2	3	4	5	6	7
T1	S1	S2	S3	S4			
T2		S1	S2	S3	S4		
T3			S1	S2	S3	S4	
T4				S1	S2	S3	S4

1nci iş (T1) 4 saat çevrimi (segman sayısı k=4) sonunda tamamlandı.

k.'dan sonraki her saat çevriminde yeni bir iş tamamlanır.

Dört iş (T4) 7 saat çevriminde tamamlanmıştır.

2.3 İş Hattının sağladığı hızlanma (*Speedup*):

İş hattındaki tüm segmanlar eşzamanlı (*synchronous*) işlem yaptığından, saat işaretinin periyot uzunluğu (çevrim zamanı) (*cycle time*) **en yavaş segmanın** gerek duyduğu çalışma zamanı (gecikmesi) tarafından belirlenir.

Çevrim zamanı (cycle time) (saat işaretinin periyodu) t_p aşağıdaki gibi hesaplanır:

$$t_p = \max(\tau_i) + d_r = \tau_M + d_r$$

t_p: çevrim zamanı (*cycle time*)

τ_i : i. katmandaki devrenin gecikmesi

τ_M : en büyük gecikme (en yavaş katman)

d_r : saklayıcıların gecikmesi

Hızlanma (*Speedup*):

k: İş hattındaki katman (segman) sayısı
t_p: saat periyodu (En yavaş birime göre ayarlanır.)
n: İş sayısı (işin tekrar sayısı)
1nci işin (T1) tamamlanması için k adet saat darbesi gereklidir.
Buna göre 1nci işin tamamlanma süresi: $T(1) = k \cdot t_p$
Kalan n-1 işin tamamlanması içi (n-1) çevrim gereklidir. Süre: $(n-1)t_p$
Tüm işlerin (n adet) toplam süresi: $(k+n-1)t_p$
t_n :İş hattı kullanılmasaydı bir işin süresi

Hızlanma (*Speedup*): $S = \frac{\text{İş hattı } \textbf{olmadan} \text{ gereken süre}}{\text{İş hattı } \textbf{ile} \text{ gerekli olan süre}}$ $S = \frac{n \cdot t_n}{(k+n-1) \cdot t_p}$

İş sayısı çok artarsa: $n \to \infty$ $\lim_{n\to\infty} S = \frac{t_n}{t_p}$

Eğer $t_n = k \cdot t_p$ varsayımı yapılırsa
(ana işi k adet eşit süreli küçük alt işleme bölmek mümkünse ve saklayıcı gecikmeleri göz ardı edilirse):

$S_{max} = k$ (**Teorik** maksimum hızlanma)

Hızlanma ile ilgili yorumlar:

İş hattının verimini arttırmak için bir işi mümkün olduğu kadar **eşit** (en azından yakın) sürelerdeki **küçük** (kısa süreli) alt işlere bölmek gerekir.

Eğer alt işlemlerin süreleri kısa olursa saat işaretinin çevrim süresi de kısalır.

Hatırlatma; en yavaş birim çevrim süresini belirler.

İş hattındaki katman sayısının etkileri:

Olumlu:

- Eğer iş **çok sayıda**, **kısa süreli** alt işlere bölünebiliyorsa segman sayısını arttırmak saat işaretini (t_p) hızlandırır ve iş hattının verimini arttırır.

$\lim_{n\to\infty} S = \frac{t_n}{t_p}$ $S_{max} = k$

Olumsuz:

- İş hattının maliyeti artar. Her katmanın sonuna yerleştirilen saklayıcılar ve ek bağlantılar; maliyet, enerji tüketimi, boyut açısından sisteme yük getirir.
- İlk baştaki 1. iş için bekleme süresi artar. $T(1) = k \cdot t_p$
- Komut iş hattında dallanma cezaları artar. Dallanma cezaları "2.5 İş Hattında Oluşabilen Sorunlar" bölümünde ele alınacaktır.

Bir iş hattı tasarlanırken bütün bu olumlu ve olumsuz noktalar birlikte dikkate alınmalıdır.

İşi alt işlemlere bölmenin hızlanma üzerindeki etkisi:

Eğer ana iş kısa süreli küçük alt işlere bölünebiliyorsa sisteme daha hızlı bir saat işareti uygulanabilir.

Örnek olarak toplam süresi 100 ns olan bir T işini ele alalım.

Bu işin farklı şekillerde alt işlere bölünebildiği varsayılmıştır.

Durum A: İş 2 eşit katmana bölünüyor.

Saklayıcıların gecikmesinin 5 ns olduğu varsayılırsa saat çevrimi t_p = 50+5 = 55 ns

Durum B: İş 3 adet dengesiz katmana bölünüyor.

Saat çevrimi t_p = 50+5 = 55 ns (en yavaş katman τ_M = 50ns)

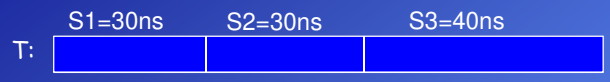
İş hattında daha fazla katman olmasına rağmen Durum A'ya göre bir hızlanma sağlanmamıştır.

Ayrıca iş hattının maliyeti de artmıştır.

İlk işin tamamlanma süresi uzamıştır. $T(1) = k \cdot t_p$

İşi alt işlemlere bölmenin hızlanma üzerindeki etkisi: (devamı)

Durum C: İş 3 adet yakın süreli katmana bölünüyor.



Saat çevrimi t_p = 40+5 = 45 ns (en yavaş katman τ_M =40ns)

Saat işareti Durum A ve B'ye göre hızlanmıştır.

Sonuç:

İş hattının hızlanma sağlayabilmesi için ana işi kısa süreli ve dengeli alt işlere bölmek gerekir.

Önemli olan saat çevriminin (t_p) süresini düşürebilmektir.

Örneğin; yukarıdaki iş, her biri 20ns süreli 5 adet alt işleme bölünebilirse saat işaretinin periyodu 25ns olur.

2.4 Komut İş Hattı (*Instruction Pipeline*)

Komut düzeyinde paralellik (*Instruction-Level Parallelism*)

Merkezi işlem birimleri her komutu işlerken belli alt işlemleri tekrar ederler.

Bir komutun MİB'te işlenme sürecine komut çevrimi (*instruction cycle*) denir.

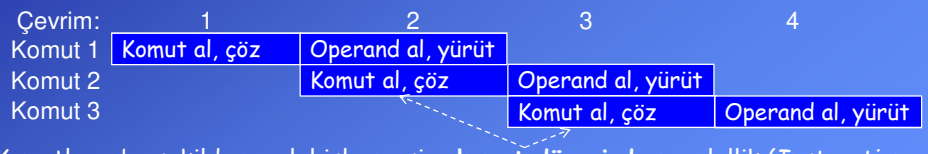
Komut çevriminin genel olarak alt çevrimleri: Komut alma ve çözme, operand alma, yürütme, kesme (Bkz, yansı 1.18).

En basit iş hattı yapısı iki katmanlı olarak kurulabilir:

1) Komut alma ve çözme 2) Operandları alma ve komut yürütme

Komut yürütme birimi belleğe erişmediği zamanlarda komut alma birimi sıradaki komutu bellekten alarak bir komut saklayıcısına yazar.

Böylece o andaki komut yürütülürken sonraki komut bellekten paralel olarak okunur.



Komutların bu şekilde paralel işlenmesine **komut düzeyinde** paralellik (*Instruction-Level Parallelism*) denir.

Hatırlatma; iş hattındaki hızlanmayı arttırmak için iş hattını çok sayıda kısa süreli katmandan oluşturmak gerekir.

Komut İş Hattı (*Instruction Pipeline*) (devamı)

Verimi arttırmak için komut işleme daha küçük alt işlemlere bölünerek 6 segmanlı bir iş hattı oluşturulabilir:

1. Komut alma (*Fetch instruction*) (FI):
2. Komut çözme (Decode instruction) (DI):
3. Operand adresi hesabı (*Calculate addresses of operands*) (CO)
4. Operand alma (*Fetch operands*) (FO)
5. Komut yürütme (*Execute instruction*) (EI)
6. Sonucu yazma (*Write operand*) (WO)

Bu kadar ayrıntılı bölmeleme ise aşağıdaki problemler nedeniyle verimli olmaz:

- Segmanların süreleri farklıdır.
- Her komut bütün alt işlemlere gerek duymaz.
- Değişik segmanlar aynı anda bellek erişimine gerek duyar.

Bu nedenle bazı alt işlemler birleştirilerek komut iş hatları daha az (örneğin 4 veya 5), dengeli segmanla oluşturulur.

Örneğin 80486'da 5 katmanlı bir iş hattı bulunmaktaydı.

Daha çok segmana sahip iş hattı içeren işlemciler de bulunmaktadır.

Örneğin Pentium 4 ailesinin işlemcilerinde 20 katmanlı iş hatları bulunmaktadır.

Bu işlemcilerde komut çevrimin alt işlemleri de daha küçük işlemlere bölünmüştür.

2.4.1 Örnek Bir Komut İş Hattı (4 katmanlı)

1. FI (*Fetch Instruction*): Komut alma; Program sayacının (PC) işaret ettiği sıradaki komutu bellekten oku.
2. DA (*Decode, Address*): Komutu çöz , operandların adreslerini hesapla.
3. FO (*Fetch Operand*): Operandları al (bellekten, saklayıcılardan).
4. EX (*Execution*): Yürütme (İşlem yapılır, saklayıcılar güncellenir. Dallanma komutlarında PC de bu katmanda güncellenir.)

- Komut alma ve operand alma işlemlerinin aynı anda yapılabilmesi için komut ve veri belleklerinin ayrı oldukları varsayılmıştır.
- Belleğe yazma işlemleri bu örneklerde göz ardı edilmiştir.
- Bu iş hattına sahip örnek bir MİB'tir. Daha gerçekçi örnekler "2.4.2 İş Hattına Sahip Örnek Bir RISC İşlemci" bölümünde verilmiştir.

2.4.1 Örnek Bir Komut İş Hattı (devamı)

A) İdeal Durum: Programda Dallanma ve operand bağımlılığı yoktur.

Komut iş hattının zaman diyagramı (ideal durum):

Saat çevrimi / Komutlar (iş "*task*")	Adımlar 1	2	3	4	5	6	7	8
1	FI	DA	FO	EX				
2		FI	DA	FO	EX			
3			FI	DA	FO	EX		
4				FI	DA	FO	EX	
5					FI	DA	FO	EX

İlk komut tamamlandı. 4 çevrim İş hattı doldu.

Bir saat çevrimi sonra ikinci komut tamamlandı.

İlk komut 4 çevrim sonunda tamamlandı (k=4).
4ncü çevrimden sonra her çevrimde yeni bir komut tamamlanır.

Komut sayısı sonsuza yaklaştığında bir komutun tamamlanma süresi de 1 saat çevrimine yaklaşır (yansı 2.9 "Hızlanma").

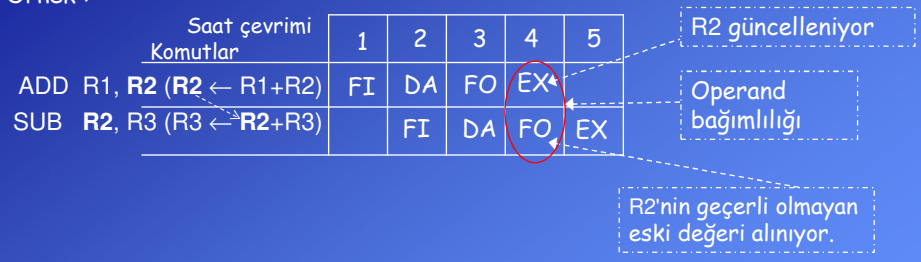
2.4.1 Örnek Bir Komut İş Hattı (devamı)

B) İş Hattında Oluşabilen Sorunlar (*Pipeline Hazards*) (*Conflicts*)

B.1 Veri Çatışması (*Data Conflict*), Operand Bağımlılığı (*Operand Dependency*):

Bir komutun kaynak operandı diğer bir komutun sonucuna bağlıdır.

Örnek :



Programın yanlış çalışmasını önlemek için çeşitli çözüm yöntemlerinin uygulanması gereklidir.

Örneğin; iş hattı durdurulabilir (*stall*) veya komutlar arasına NOOP (No operation) komutları yerleştirilebilir.

Olası çözüm yöntemleri "2.5 İş hattında oluşan sorunlar ve çözümleri" bölümünde ele alınacaktır.

2.4.1 Örnek Bir Komut İş Hattı (devamı)

B.2 Denetim Sorunları (*Control Hazards*):
Dallanma ve Kesmeler (*Branches, Interrupts*)

İş hattında komutlar paralel olarak yürütüldüğünden bir dallanma komutu işlenirken bellekte ondan sonra gelen ancak dallanma nedeniyle yürütülmeyecek olan komut (veya komutlar) da iş hattına alınmış olur.

Eğer önlem alınmazsa programın mantığı gereği yürütülmemesi gereken komutlar da yürütülmüş olur.

Örnek:

```
1.          Komut_1
2.          JUMP Hedef
3.          Komut_3
            :
4.  Hedef   Komut_4
```

Koşulsuz dallanma komutu (BRA / JUMP)

Bellekte dallanmanın peşindeki komut (*next instruction*). Programa göre **yürütülmemesi gerekir**.

Dallanmanın hedefi, dallanmadan sonra yürütülecek komut (*target instruction*).

Koşulsuz dallanma komutu JUMP işlenirken Komut_3 de iş hattına girmiş olur.

Programın yanlış çalışmasını önlemek için iş hattını durdurmak (*stall*) ve Komut_3 çalışmadan önce iş hattını boşaltmak gerekir.

a. Koşulsuz Dallanma (*Unconditional Branch*)

Saat çevrimi / Komutlar	1	2	3	4	5	6	7
Komut 1	FI	DA	FO	EX			
Koşulsuz Dallan 2		FI	DA	FO	EX		
Komut 3			~~FI~~	-	-		
Hedef Komut 4						FI	DA

Komut çözüldüğünde dallanma olduğu anlaşılır.

Dallanılacak adres alınıyor (Mutlak ya da bağıl).

PC (program sayacı) güncelleniyor.
PC = Hedef (dallanılacak adres)

Sorun: Bu komut boşuna alındı.
Bu komut yürütülmemeli!
İş hattından silinecek.

Dallanma cezası (*Branch penalty*)
İş hattı durdurulacak ve boşaltılacak.

Dallanmadan sonra gidilen hedef komut (Dallanmanın hedefi)

Koşulsuz dallanma komutu çözüldüğü (anlaşıldığı) anda olası önlemlerden biri iş hattına yeni komut alma işlemini (FI katmanını) durdurmaktır.

Dallanma komutunun yürütülmesi sonucu hedef komutun adresi hesaplanıp program sayacı (PC) güncellendikten sonra komut alma işlemi tekrar başlar.

b. Koşullu Dallanma (*Conditional Branch*):

Koşullu dallanma komutları yürütülürken iki durum oluşur;

1. Koşul yanlıştır (dallanma olmaz), 2. koşul doğrudur (dallanma olur)

b1. Koşullu dallanma (koşul yanlışsa):

Koşul doğru değilse iş hattını durdurmaya veya boşaltmaya gerek yoktur, çünkü program bir sonraki komut ile devam edecektir.

Saat çevrimi / Komutlar	1	2	3	4	5	6
1	FI	DA	FO	EX		
Koşullu Dallan. 2		FI	DA	FO	EX	
3			FI	DA	FO	EX

Önceki komut bayrakları (koşulları) belirliyor.

PC değişmedi.
Dallanma gerçekleşmedi.

Dallanmanın peşindeki komut yürütülüyor.

Koşula bakılmaksızın bir sonraki komut alındı.

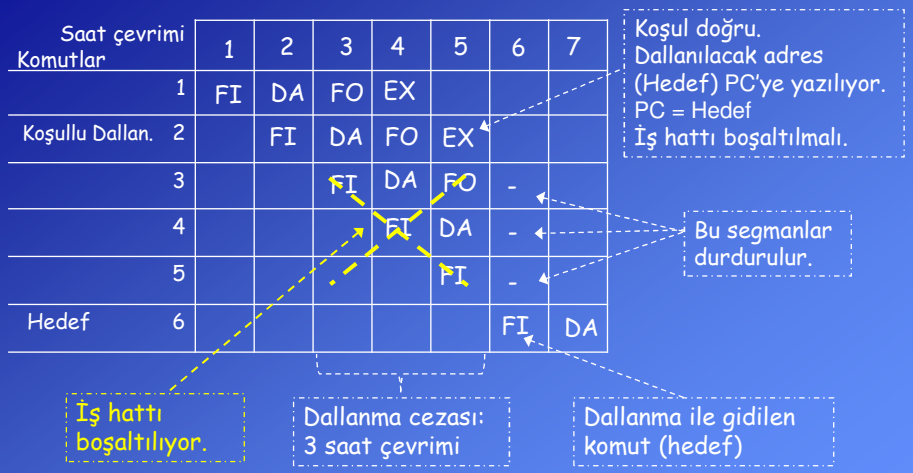
Dallanma olmadığı için boşaltılmayacak (ceza yok).

Koşulun doğru olup olmadığı ancak önceki komut yürütüldükten sonra belli olur.

Eğer koşul yanlışsa (dallanma yoksa) dallanma cezası oluşmaz.

Eğer koşul doğru çıkarsa çözüm yöntemlerine gerek duyulur (sonraki yansılar).

b2. Koşullu dallanma (koşul doğru ise):



Dallanma cezasının süresi iş hattındaki segmanların sayısına ve işlevlerine bağlıdır.

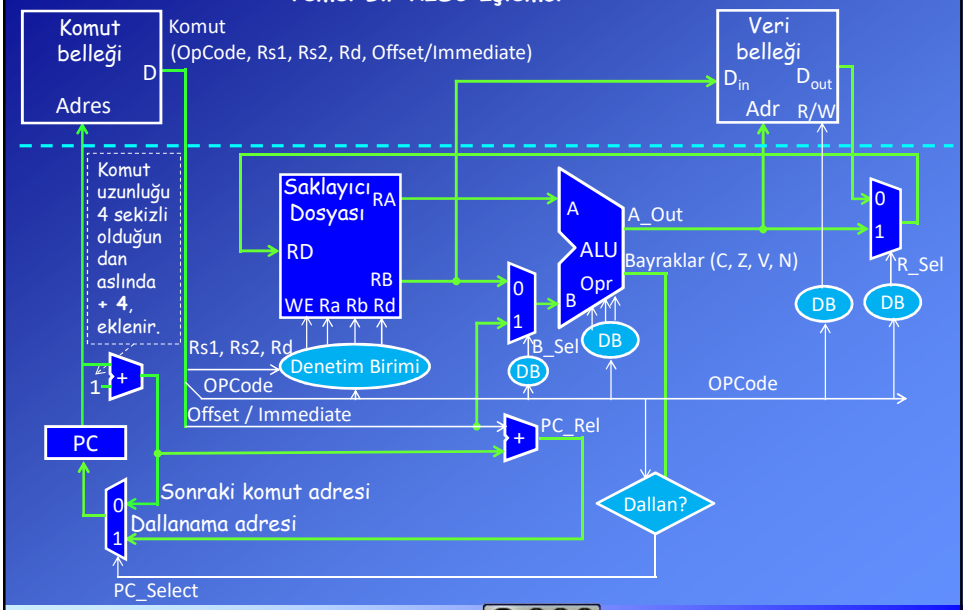
Bu örnek iş hattında dallanma cezası 3 saat çevrimidir; ancak başka yapılardaki iş hatlarında bu süre farklı olabilir (2.5.3. Denetim Sorunları (*Control Hazards*))

2.4.2 İş Hattına Sahip Örnek Bir RISC İşlemci

- Sabit uzunlukta komutlar (genellikle 32 bit).
  Komut alma ve çözme işlemleri basittir (iş hattında yarar sağlar).
- Komutların çoğu sadece saklayıcılar üzerinde işlem yapar. Sadece bellek okuma/yazma (*load/store*) komutları saklayıcılar ve bellek arasında işlem yapar.
- Adresleme kipleri kısıtlıdır.
- Bazı örnek komutlar:

• ADD	Rs1,Rs2,Rd	*Rd ← Rs1 + Rs1*	
ADD	R3, R4, R12	*R12 ← R3 + R4*	
• ADD	Rs,S2,Rd	*Rd ← Rs + S2*	(S2: İvedi (*immediate*) veri)
ADD	R1, #$1A, R2	*R2 ← R1 + $1A*	
• LDL	S2(Rs),Rd	*Rd←M[Rs + S2]*	Load long (32 bit)
LDL	$500(R4), R5	*R5 ← M[R4 + $500]*	
• STL	S2(Rs), Rm	*M[Rs + S2] ← Rm*	Store long (32 bit)
STL	$504(R6), R7	*M[R6 + $504] ← R7*	
• BRU	Y	*PC←PC + Y*	Unconditional branch
BRU	$0A	*PC←PC + $0A*	Bağıl dallanma (Y: Offset)
• Bcc	Y	*If (cc) then PC←PC + Y*	Conditional branch
BGT	$0A	*If greater, then PC←PC + $0A*	

Temel Bir RISC İşlemci



İş hattına sahip RISC Örnekleri

İş hattına sahip RIDC işlemciler farklı şekillerde tasarlanmaktadır.

Örneğin;

- ARM7'nin iş hattı 3 katmana sahiptir
  IF: (*Instruction fetch*); Komut al
  DR: (*Decode and read registers*); Komutu çöz, operandları saklayıcılardan oku
  EX: (*Execution*); ALU işlemi, bellek erişimi (eğer gerekli ise) sonucu saklayıcılara yaz.
- MIPS R3000: 5 katmanlı
- MIPS R4000: 8 katmanlı (*superpipelined*)
- ARM Cortex-A8: 13 katmanlı

Örnek Bir 5 Katmanlı RISC İş hattı

Bu derste, iş hattı ile ilgili kavramları açıklamak için örnek bir 5 katmanlı RISC iş hattı kullanılacaktır.

1. *Instruction fetch* (IF):

   Sıradaki komutu bellekten al, program sayacını (PC) arttır (komut uzunluğu kadar). Eğer bir komut 4 sekizli ise, PC ← PC + 4.

2. *Instruction Decode, Read registers* (DR)

   Komutu çözerek tüm birimlere gidecek olan denetim işaretlerini oluştur ve saklayıcı dosyasında ilgili saklayıcıları (operandları) oku.

3. *Execute* (EX)

   ALU işlemlerini yap, dallanma komutlarının hedef adreslerini hesapla.

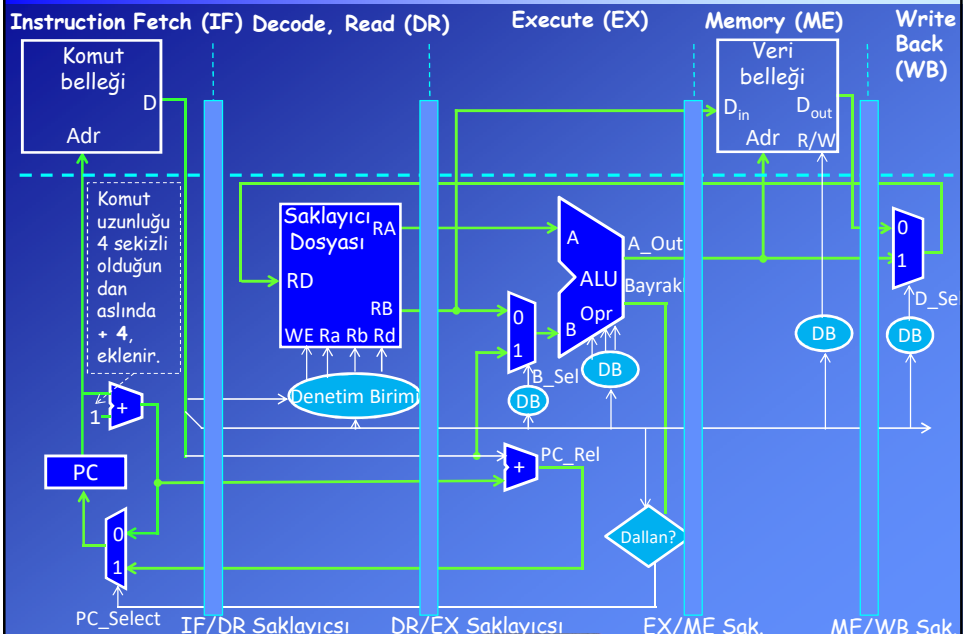
4. *Memory* (ME)

   Eğer gerekli ise veri belleğine eriş (sadece load/store komutları)

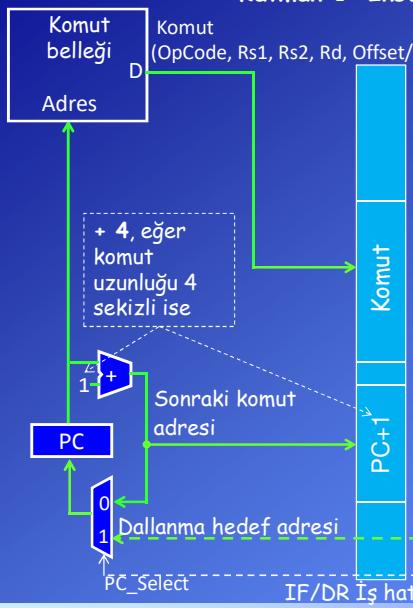
5. *Write back* (WB)

   Sonuçları saklayıcı dosyasına yaz.

Beş Katmanlı Bir RISC İş Hattı



Katman 1: *Instruction Fetch* (IF), Komut Alma

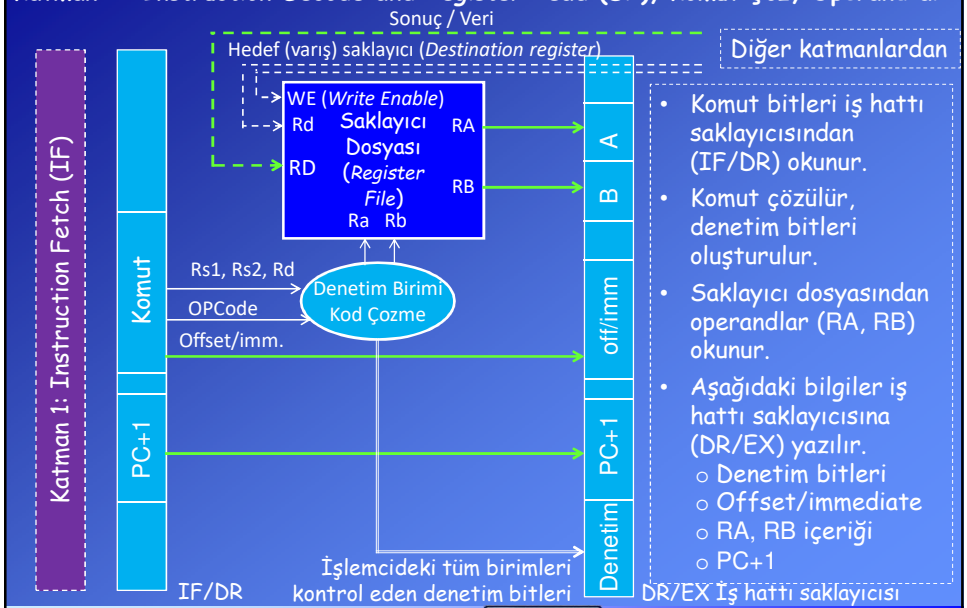


PC'nin o andaki değeri sıradaki komutun komut belleğindeki adresini gösterir

- Komut, komut belleğinden okunur.
- PC arttırılır (şimdilik dallanma olmadığı varsayılıyor).
- Komutu oluşturan bitler (op code, Rs1, Rs2, Rd, S2, offset) iş hattı saklayıcısına (IF/DR) yazılır.
- PC+1 değeri iş hattı saklayıcısına (IF/DR) yazılır. Daha sonraki katmanlarda dallanma adresi hesabında kullanılır.
- Eğer dallanma varsa, PC_Select=1 olur, dallanma hedef adresi PC'ye yazılır.

Diğer katmanlardan

Katman 2: *Instruction Decode and Register Read* (DR), Komut Çöz, Operand al

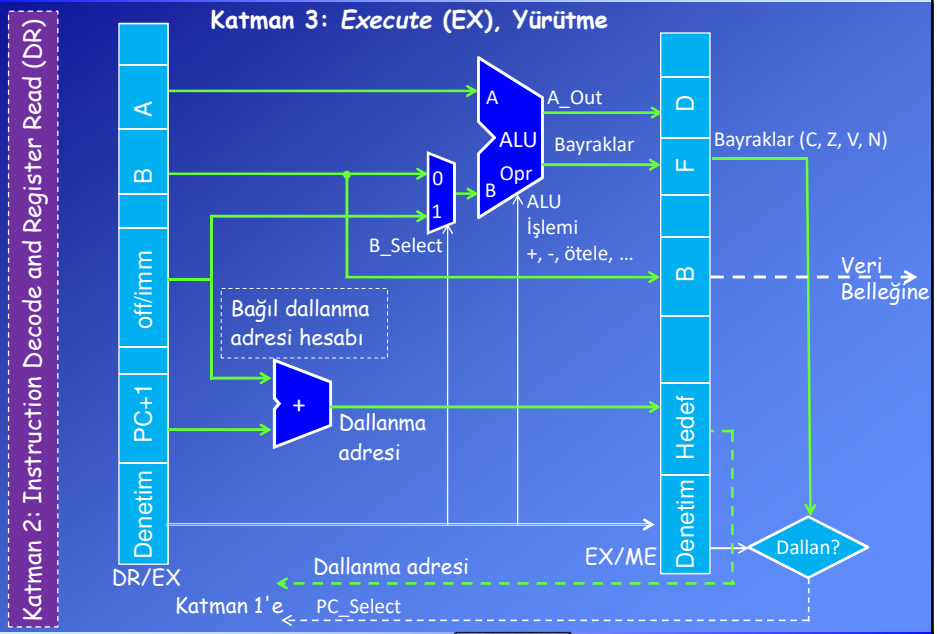


- Komut bitleri iş hattı saklayıcısından (IF/DR) okunur.
- Komut çözülür, denetim bitleri oluşturulur.
- Saklayıcı dosyasından operandlar (RA, RB) okunur.
- Aşağıdaki bilgiler iş hattı saklayıcısına (DR/EX) yazılır.
  - Denetim bitleri
  - Offset/immediate
  - RA, RB içeriği
  - PC+1

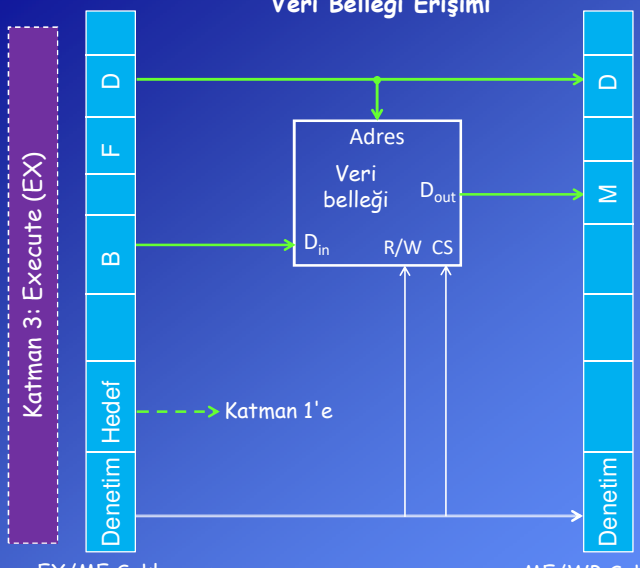
Katman 3: *Execute* (EX), Yürütme

- Denetim bitleri ve veriler (offset/immediate, RA, RB) iş hattı saklayıcısından (DR/EX) okunur.
- ALU işlemleri yapılır.
  ALU, LOAD/STORE komutları için gerekli olan adres hesaplarını da yapar.
  Örneğin; LDL $500(R4), R5 *R5 ← M[R4 + $500]*
  İvedi veri $500 ile R4 saklayıcısını içeriği ALU tarafından toplanır.
- Dallanma komutları için dallanma hedef adresi hesaplanır.
  Örneğin; BGT $0A *Büyükse, PC ← PC + $0A*
  Bu örnek işlemcide, dallanma hedef adresi hesabı için ALU'dan ayrı bir toplayıcı kullanılmaktadır.
- Dallanma olup olmayacağına karar verilir (denetim bitleri ve ALU'dan gelen bayrak değerleri kullanılır).
- Aşağıdaki bilgiler iş hattı saklayıcısına (EX/ME) yazılır.
  - Denetim bitleri
  - ALU'da oluşan sonuç (D) ve bayraklar (F)
  - Belleğe yazma işlemi için RB değeri (B)
  - Dallanma hedef adresi (Hedef)

Katman 3: *Execute* (EX), Yürütme

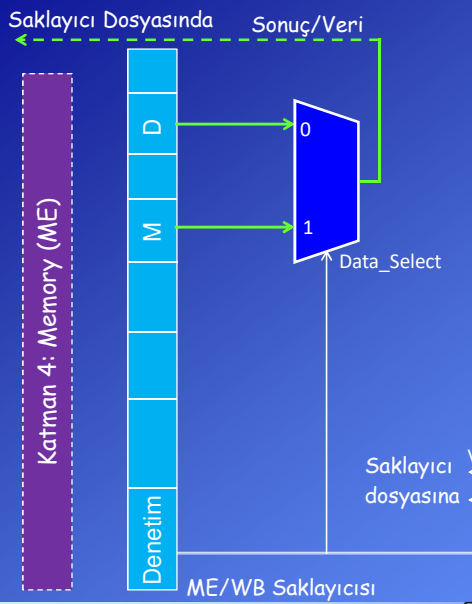


Katman 4: Memory (ME), Veri Belleği Erişimi



- Veri belleği adresi (ALU'dan gelen sonuç) D iş hattı saklayıcısından (EX/ME) okunur.
- Veri B (STORE komutu için) iş hattı saklayıcısından (EX/ME) okunur.
- Eğer gerekli ise bellek okuma/yazma işlemi yapılır.
- Aşağıdaki bilgiler iş hattı saklayıcısına (ME/WB) yazılır.
  - Denetim bitleri
  - Bellek erişimi sonucu (M)
  - ALU'dan gelen sonuç (aktarma) (D)

Katman 5: Write Back (WB), Saklayıcılara Yazma



- ALU'dan gelen sonuç (D) iş hattı saklayıcısından (ME/WB) okunur.
- Bellek erişiminin sonucu (M) iş hattı saklayıcısından (ME/WB) okunur.
- Veri seçilerek (M veya D) saklayıcı dosyasındaki hedef saklayıcıya yazılır.
- Denetim bilgileri (Rd, WE) saklayıcı dosyasına gönderilir.

Örnek RISC iş hattının zaman diyagramı (ideal durum):

İdeal Durum: Programda dallanma ve veri bağımlılığı yoktur.

Saat çevrimi / Komutlar	1	2	3	4	5	6	7	8
1	IF	DR	EX	ME	WB			
2		IF	DR	EX	ME	WB		
3			IF	DR	EX	ME	WB	
4				IF	DR	EX	ME	WB

İlk komut tamamlandı.
5 çevrim
İş hattı doldu.

Bir saat çevrimi sonra
ikinci komut tamamlandı.

İlk komut 5 çevrim sonunda tamamlandı (k = 5).

5nci çevrimden sonra her çevrimde yeni bir komut tamamlanır.

Komut sayısı sonsuza yaklaştığında bir komutun tamamlanma süresi de 1 saat çevrimine yaklaşır (yansı 2.9 "Hızlanma").

IF ve ME katmanları aynı anda belleğe erişmek isterler.

Bu bellek çatışması sorununu çözmek için komut ve veri bellekleri ayrılmıştır (Harvard mimarisi).

2.5 İş Hattı Sorunları (*Pipeline Hazards (Conflicts)*) ve Çözümleri

İş hattında 3 tür sorunla karşılaşılır.

1. Kaynak Çatışması (*Resource Conflict*), Yapısal Sorun (*Structural Hazard*):

İş hattında aynı anda işlenen iki komut aynı kaynağa (bellek, ALU) gerek duyarsa kaynak çatışması oluşur.

2. Veri Çatışması (*Data Conflict*), Veri Bağımlılığı (*Data Dependency*):

Henüz güncellenmemiş olan veri erkenden kullanılmak istenirse sorun ortaya çıkar.

3. Denetim Sorunları (*Control Hazards*) Dallanma (*Branch*), Kesme (*Interrupt*):

İş hattında bir dallanma komutu işlenirken bellekte ondan sonra gelen, ancak dallanma nedeniyle yürütülmeyecek olan komut (veya komutlar) da iş hattına alınmış olur.

Dallanma sonrası iş hattına alınması gereken **hedef komut adresi**, MİB dallanma komutunu yürütene kadar (PC güncellenmeli) belli değildir.

Koşullu dallanma problemi: Bayrakları değiştiren son komut yürütülünceye kadar bayrak değerleri belli olmadığından dallanmanın olup olmayacağı (büyük?, eşit?) belli değildir.

İş hattını durdurmak bu sorunları çözer ancak performansı düşürür.

Daha verimli çözüm yöntemleri bulunmaktadır.

2.5.1. Kaynak Çatışması (*Resource Conflict*), Yapısal Sorun (*Structural Hazard*):

İş hattında aynı anda işlenen iki (veya daha fazla) komut aynı kaynağa (bellek, ALU) gerek duyarsa kaynak çatışması oluşur.

a) Bellek çatışması: İki farklı segmanda aynı bellek modülüne erişilmek istenirse
Örneğin komut alma ile operand okuma/yazma aynı anda olamaz.

Çözümler:

- Komutların belli bölümleri paralel değil, peş peşe seri işlenir. İş hattının belli segmanları durdurulur. Bu çözüm performansı düşürür.
- Harvard mimarisi: Komutlar ve veriler için ayrı bellek
- Komut kuyruğu veya cep bellek: Bir komut işlenirken belleğe erişilmediği anlarda sıradaki komutlar bellekten okunarak bir kuyruğa yazılır.

b) İşlem birimi (ALU, FPU) çatışması: İki farklı segmanda aynı işlem birimine (*Arithmetic Logic Unit*- ALU, *Floating Point Unit*- FPU) gerek duyulursa.

Çözümler:

- İşlem birimlerinin sayısı arttırılır. Örneğin adres hesabı ve veri işleme için iki ayrı ALU kullanılır.
- İşlem birimleri de iş hattı olarak tasarlanarak paralellik sağlanır. Örnek FPU

2.5.2. Veri Çatışması (*Data Conflict*), Veri Bağımlılığı (*Data Dependency*):

Bir veri hazır (güncel) olmadan önce kullanılmaya çalışırsa veri çatışması olur.

Bu sorun çözülmezse iş hattında çalışan program yanlış sonuç üretebilir.

Örnek:

ADD R1, R2, R3 $R3 \leftarrow R1 + R2$
SUB R3, R4, R5 $R5 \leftarrow R3 - R4$

İş hattında veri bağımlılığı

Saat çevrimi / Komutlar	1	2	3	4	5	6
ADD R1,R2,**R3**	IF	DR	EX	ME	WB	
SUB **R3**,R4,R5		IF	DR	EX	ME	WB

ADD komutunun sonucu saklayıcı dosyasına (R3) yazıldı.

SUB komutu, R3 saklayıcısını güncellenmeden önce okur.
R3 henüz bir önceki ADD komutunun sonucunu içermiyor.

2.5.2. Veri Çatışması (*Data Conflict*), devamı

Üç farklı tipte veri çatışması (*data hazard*) oluşabilir:

- **Yazmadan sonra okuma (*Read after write*) (RAW):** Bu türe gerçek bağımlılık (*true dependency*) da denir.

  Bir komut, bir saklayıcıyı veya bellek gözünü değiştirmektedir. Daha sonra gelen bir komut da aynı saklayıcı veya bellek gözünü okumaktadır.

  Eğer iş hattı nedeniyle okuma işlemi yazmadan önce yapılırsa veri çatışması sorunu oluşur.

- **Okumadan sonra yazma (*Write after read*) (WAR):** Anti bağımlılık da denir.

  Bir komut, bir saklayıcıyı veya bellek gözünü okumaktadır. Daha sonra gelen bir komut da aynı saklayıcı veya bellek gözüne yazmaktadır.

  Eğer yazma işlemi okumadan önce yapılırsa veri çatışması sorunu oluşur.

- **Yazmadan sonra yazma (*Write after write*) (WAW):** Çıkış bağımlılığı da denir.

  İki komut aynı saklayıcıyı veya bellek gözüne yazmaktadır.

  Eğer yazma işlemleri programda belirtilenden farklı sırada olursa veri çatışması sorunu oluşur.

Veri çatışması sorununun çözümleri:

A) Durdurma, Donanım Kilidi (*Hardware interlock*) (donanım tabanlı çözümler):

Bir donanım, iş hattındaki tüm komutları (denetim bitlerini) izler. Veri bağımlılığı olan komutların iş hattına girmesi geciktirilir.

İş hattının komut alma segmanı (IF) gerekli saat çevrimi kadar durdurulur (*stall*).

Örnek:

Saat çevrimi / Komutlar	1	2	3	4	5	6	7	8	9
ADD R1,R2,**R3**	IF	DR	EX	ME	WB				
SUB **R3**,R4,R5		IF	-	-	-	DR	EX	ME	WB

Önce R3'e yazılır, sonra okunur. Yazma ve okuma farklı saat çevrimlerinde.

Veri çatışması sezildi. IF/DR.Rs1 = DR/EX.Rd

İş hattında komutun ilerlemesi durduruldu. 3 saat çevrimi gecikme oluştu.

İş hattının durdurulması:

IF/DR saklayıcısına yükleme izni verilmez.

DR katmanına NOOP (*No Operation*) komutunun denetim bitleri yazılır.

PC'nin güncellenmesine izin verilmez.

Veri çatışması sorununun çözümleri (devamı):

Saklayıcı dosyasına (*register file*) erişim sorunun çözümü:

Saklayıcı dosyasına, aynı saat çevriminde hem okuma ham de yazma için erişilebilir.

Saat çevriminin ilk yarısında (saat işaretinin çıkan kenarında) veri yazılır, ikinci yarısında (inen kenar) ise okunur.

Bu yöntem, gecikme (durdurma) süresini 3 çevrimden 2 çevrime düşürür.

Saat çevrimi / Komutlar	1	2	3	4	5	6	7	8
ADD R1,R2,**R3**	IF	DR	EX	ME	WB			
SUB **R3**,R4,R5		IF	-	-	DR	EX	ME	WB

Önce R3'e ilk yarıda yazılır, sonra ikinci yarıda okunur.

Veri çatışması sezildi. IF/DR.Rs1 = DR/EX.Rd

Yaz

Oku

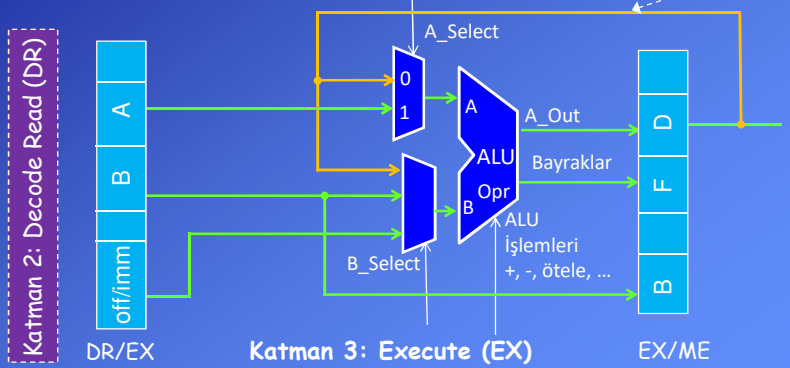
Veri çatışması sorununun çözümleri (devamı):

B) Operand yönlendirme (*Operand forwarding or Bypassing*) (Donanım):

EX katmanının çıkışı (EX/ME saklayıcısı) ile ALU girişleri arasında doğrudan bir bağlantı (*bypass*) oluşturulur.

A_Select ve B_Select seçme girişleri iş hattındaki çatışma sezme (*hazard detection*) birimi tarafından belirlenirler. Bu birim, ya saklayıcı dosyasından okunan verinin ya da bir önceki ALU işleminin sonucunun ALU girişlerine aktarılmasını sağlar.

Forwarding (Bypass)



Operand yönlendirme EX/ME saklayıcısından ALU'ya (devamı):

Eğer çatışma sezme birimi bir önceki ALU işleminin hedef saklayıcısının şimdiki ALU işleminin kaynağı olduğunu sezerse, denetim birimi ALU'nun girişine saklayıcıdan gelen değeri değil, ALU'nun çıkışından doğrudan gelen değeri (*bypass*) yönlendirir.

Örnek:

Komutlar / Saat çevrimi		1	2	3	4	5
ADD R1, R2, **R3**;	R3←R1 + R2	IF	DR	EX	ME	WB
SUB **R3**, R4, R5;	R5←R3 - R4		IF	DR	EX	ME

R3'ün geçerli olmayan önceki değeri okunuyor.
Bu geçersiz değer EX segmanında kullanılmayacak.

İş hattı denetim birimi ALU'nun girişine saklayıcıdan DR'de alınan geçersiz değeri değil, ALU'nun çıkışından doğrudan gelen değeri (*bypass*) yönlendirir (A_Select = 0).

Eğer sorunu operand yönlendirme ile çözmek mümkün olursa iş hattını durdurmaya gerek olmaz ve performans düşmez.

Bellekten okuma çatışmasının (*Load-use data hazard*) operand yönlendirme ile çözülmesi:

Bellekten okuma çatışması (*Load-use data hazard*):

Load komutları da veri çatışmasına neden olur.

Örnek:

Bellekten okunan veri saklayıcı dosyasına (R1) yazıldı.

Bellekten okuma çatışması

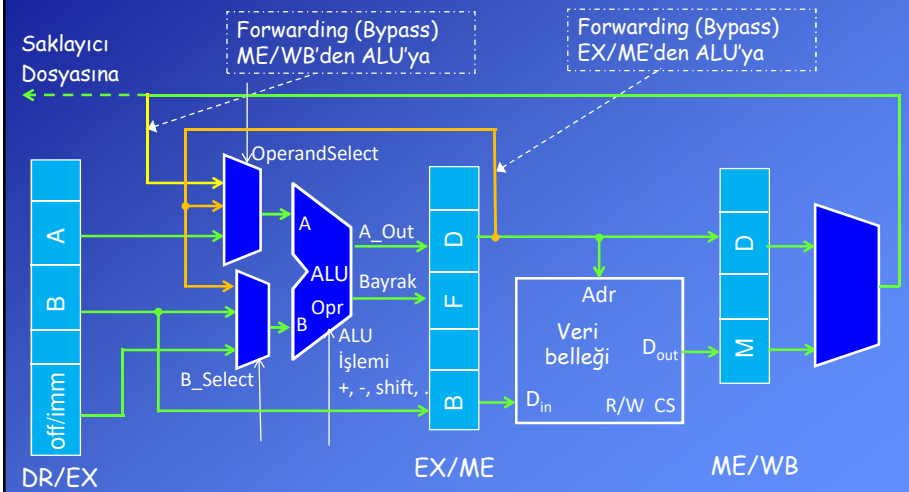
Komutlar / Saat çevrimi		1	2	3	4	5	6
LDL $500(R4), **R1**	*R1 ← M[R4 + $500]*	IF	DR	EX	ME	WB	
ADD **R1**, R2, R3	*R3 ← R1 + R2*		IF	DR	EX	ME	WB

ADD komutu R1 saklayıcısını güncellenmeden önce okur.
R1'deki değer güncel (geçerli) değildir.

Operand yönlendirme ME/WB saklayıcısından ALU'ya :

Bellekten yükleme çatışması nedeniyle oluşan gecikmeyi azaltmak için ME katmanının çıkışından (ME/WB saklayıcısı) ALU'nun girişine doğrudan bir bağlantı oluşturulur.

Ancak bir saat çevrimi gecikme hala gereklidir.



Bellekten okuma çatışması (*Load-use data hazard*) (devamı):

Operand yönlendirme (*forwarding*) + 1 çevrim gecikme ile çözüm:

Örnek:

	Saat çevrimi / Komutlar	1	2	3	4	5	6	7
LDL	$500(R4), R1	IF	DR	EX	ME	WB		
ADD	R1, R2, R3		IF	-	DR	EX	ME	WB

R1'in geçerli olmayan önceki değeri okunuyor.
Bu geçersiz değer EX segmanında kullanılmayacak.

İş hattı denetim birimi ALU'nun girişine saklayıcıdan DR'de alınan geçersiz değeri değil, ALU'nun çıkışından doğrudan gelen değeri (*bypass*) yönlendirir.

Veri çatışması sorununun çözümleri (devamı):

C) NOOP (*No Operation*) komutları eklemek (Yazılım temelli):

Derleyici, çatışmaya neden olan komutlar arasına gerektiği kadar NOOP komutu ekler.

Bu çözümün etkisi iş hattını durdurmak ile aynıdır.

Örnek:

Saat çevrimi / Komutlar	1	2	3	4	5	6	7	8
ADD R1,R2,**R3**	IF	DR	EX	ME	WB			
NOOP		IF	DR	EX	ME	WB		
NOOP			IF	DR	EX	ME	WB	
SUB **R3**,R4,R5				IF	DR	EX	ME	WB

Derleyici ekler.

İlk yarıda R3'e yazılır, sonra ikinci yarıda okunur.

NOOP bir makine dili komutu olduğundan, iş hattında diğer komutlar gibi tüm katmanlardan geçerek aynı şekilde işlenir.

NOOP komutları nedeniyle gecikme oluştuğundan sistemin performansı düşer.

Veri çatışması sorununun çözümleri (devamı):

D) Optimize edilmiş çözüm (Yazılım temelli):

Derleyici, eğer mümkünse programda uygun komutların yerini değiştirerek bu komutları çatışmaya beden olan komutların arasına yerleştirir.

Bu değişiklik algoritmayı değiştirmemeli ve başka çatışmalara neden olmamalı.

Örnek:

```
STL  $00(R6), R1   M[R6 + $00] ← R1
STL  $04(R6), R2   M[R6 + $04] ← R2
ADD  R1, R2, R3    R3 ← R1 + R2
SUB  R3, R4, R5    R5 ← R3 – R4
```

Saat çevrimi / Komutlar	1	2	3	4	5	6	7	8
ADD R1,R2,**R3**	IF	DR	EX	ME	WB			
STL $00(R6), R1		IF	DR	EX	ME	WB		
STL $04(R6), R2			IF	DR	EX	ME	WB	
SUB **R3**,R4,R5				IF	DR	EX	ME	WB

Derleyici taşıdı.

İlk yarıda R3'e yazılır, sonra ikinci yarıda okunur.

NOOP eklemeye göre performans iyileşmiştir.

Bu çözümde NOOP komutları veya durdurma nedeniyle oluşan gecikmeler yoktur.

2.5.3. Denetim Sorunları (Dallanmalar, Kesmeler) (*Control Hazards*):

Örnek RISC işlemcide dallanma komutlarının (branch/jump) hedef adresleri Yürütme katmanında (*Execution* - EX) hesaplanır (yansı 2.30).

Hedef adres EX/ME iş hattı saklayıcısına yazılır.

Dallanma kararı yürütmeden sonra oluşan bayrak değerlerine göre Bellek katmanında (*Memory* - ME) verilir (yansı 2.30).

EX katmanından sonra dallanmaya ilişkin karar (PC_Select) ve hedef adres, 1. katman IF'e gönderilir.

IF katmanında önce PC'nin işaret ettiği komut okunur, sonra PC güncellenir.

Bu işlemler sırasında dallanma komutunun altında yer alan (dallanmanın hedefi olmayan) sıradaki komutlarda iş hattına alınmış olur.

Halbuki dallanma olduğunda bu komutların atlanması gerekirdi.

Bu durumda, ya bir donanım birimi iş hattını durdurup boşaltmalı ya da derleyici temelli bir çözüm olan gecikmeli dallanma (*delayed branch*) uygulanmalı.

İş hattına alınan gereksiz komutlar WB katmanına ulaşmadan önce durdurulmalılar.

İşlemcideki değişiklikler WB katmanında yapılır.

Koşullu Dallanma Sorunları:

Örnek:

```
100  SUB    R1, R2, R1      R1 ← R1 - R2
104  BGT    $1C             Branch if greater ($108 + $1C = $124 Hedef adres)
108  ADD    R1, R1, R2
10C  ADD    R3, R4, R2
110  STL    $00(R5), R2
114  LDL    $0A(R6), R1
...
124  STL    $00(R6), R2     BGT'nin hedefi
```

Eğer dallanma gerçekleşirse bu komutların atlanması gerekir.

Hatırlatma: Bcc koşullu dallanma komutları son ALU işleminde oluşan bayrak değerlerine göre davranırlar.

Örneğin; BGT komutu, işaret "N" (*Negative*) ve taşma "V" (*Overflow*) bayraklarını değerlendirir.

Koşullu Dallanma Sorunları (devamı):
Örnek (devamı): **Eğer dallanma olursa**

Hedef adres ($108 + $1C = $124) EX katmanında hesaplandı ve EX/ME saklayıcısına yazıldı.

Dallanma kararı verildi (EX'ten sonra). "Dallanma var"

Hedef adres EX/ME saklayıcısından IF katmanına gider.

Komutlar										
SUB R1, R2, R1	IF	DR	EX	ME	WB					
BGT $1C		IF	DR	EX	ME	WB				
ADD R1, R1, R2			IF	DR	EX	ME	WB			
ADD R3, R4, R2				IF	DR	EX	ME	WB		
STL $00(R5), R2					IF	DR	EX	ME	WB	
Hedef: STL $00(R6), R2						IF	DR	EX	ME	WB

Bu komutlar atlanmalıydı.

İş hattı durdurup boşaltılmalı veya yazılım temelli çözümler uygulanmalı.

PC, IF katmanının sonunda güncellenir. PC← $124 (Hedef)

BGT komutunun hedefi olan komut alınır.

Eğer iş hattını durdurma çözümü uygulanırsa bu örnek işlemcide, **dallanma cezası 3 çevrimdir.**

Koşullu Dallanma Sorunları (devamı):
Örnek (devamı): **Eğer dallanma olmazsa**

Hedef adres ($108 + $1C = $124) EX katmanında hesaplandı ve EX/ME saklayıcısına yazıldı.

Dallanma kararı verildi (EX'ten sonra). "Dallanma yok"

Hedef adres EX/ME saklayıcısından IF katmanına gider.

Komutlar										
SUB R1, R2, R1	IF	DR	EX	ME	WB					
BGT $1C		IF	DR	EX	ME	WB				
ADD R1, R1, R2			IF	DR	EX	ME	WB			
ADD R3, R4, R2				IF	DR	EX	ME	WB		
STL $00(R5), R2					IF	DR	EX	ME	WB	
LDL $0A(R6), R1						IF	DR	EX	ME	WB

PC, IF katmanının sonunda güncellenir. PC← PC+1 (Sıradaki komut) **Dallanmanın hedef adresi değil.**

Sıradaki komut

Eğer dallanma olmazsa dallanma cezası oluşmaz.

Dallanma cezasının azaltılması:

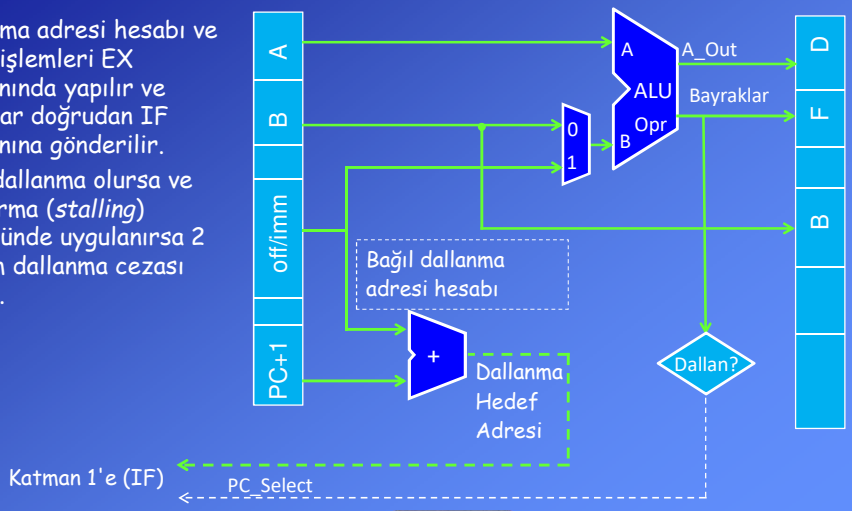
Koşullu dallanma:

Execute (EX) katmanı değiştirilir.

Yürütme (*Execute* EX) katmanı

Dallanma adresi hesabı ve karar işlemleri EX katmanında yapılır ve sonuçlar doğrudan IF katmanına gönderilir.

Eğer dallanma olursa ve durdurma (*stalling*) çözümünde uygulanırsa 2 çevrim dallanma cezası oluşur.



Dallanma cezasının azaltılması (devamı):

Koşullu dallanma (devamı) : Dallanma olursa

Örnek:

Hedef adres ($108 + $1C = $124) hesaplandı. Dallanma kararı alındı (EX'te).

Hedef adres IF katmanına gönderildi.

Komutlar

SUB R1, R2, R1	IF	DR	EX	ME	WB				
BGT $1C		IF	DR	EX	ME	WB			
ADD R1, R1, R2			IF	DR	~~EX~~	~~ME~~	~~WB~~		
ADD R3, R4, R2				IF	~~DR~~	~~EX~~	~~ME~~	~~WB~~	
Hedef: STL $00(R6), R2					IF	DR	EX	ME	WB

Bu komutlar atlanmalıydı.

İş hattı durdurup boşaltılmalı veya yazılım temelli çözümler uygulanmalı.

PC, IF katmanının sonunda güncellenir. PC← $124 (Hedef)

BGT komutunun hedefi olan komut alınır.

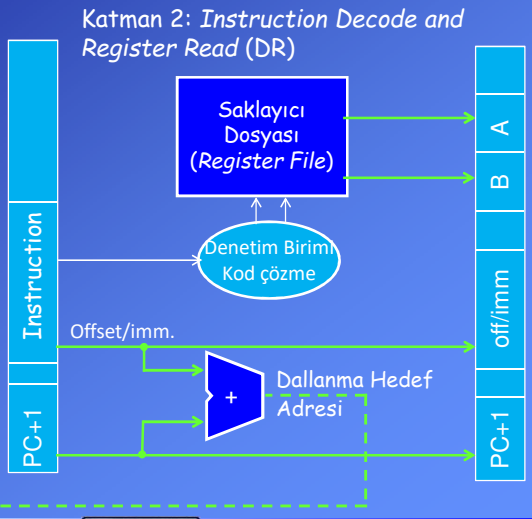
Eğer iş hattını durdurma çözümü uygulanırsa bu örnek iş hattında, **dallanma cezası 2 çevrim** olur.

Dallanma cezasının azaltılması (devamı):

Koşulsuz dallanma:

Bayrak değerlerine gerek olmadığından, dallanma hedef adresi hesabı DR katmanına aktarılabilir.

Bu iyileştirmeden sonra koşulsuz dallanma komutunun (BRU) **dallanma cezası 1 çevrim** olur.



Dallanma cezasının azaltılması (devamı):

Koşulsuz dallanma (devamı) :

Örnek:

Hedef adres ($108 + $1C = $124) hesaplandı

Hedef adres IF katmanına gönderildi.

Komutlar								
SUB R1, R2, R1	IF	DR	EX	ME	WB			
BRU $1C		IF	DR	EX	ME	WB		
Bu komut atlanmalı. ADD R1, R1, R2			IF	DR	EX	ME	WB	
Hedef: STL $00(R6), R2				IF	DR	EX	ME	WB

İş hattı durdurup boşaltılmalı veya yazılım temelli çözümler uygulanmalı.

PC, IF katmanının sonunda güncellenir. PC← $124 (Hedef)

BRU komutunun hedefi olan komut alınır.

Dallanma hedef adresi hesabının DR katmanına taşınmasından sonra, örnek iş hattında, **koşulsuz dallanmanın cezası 1 çevrim** olur.

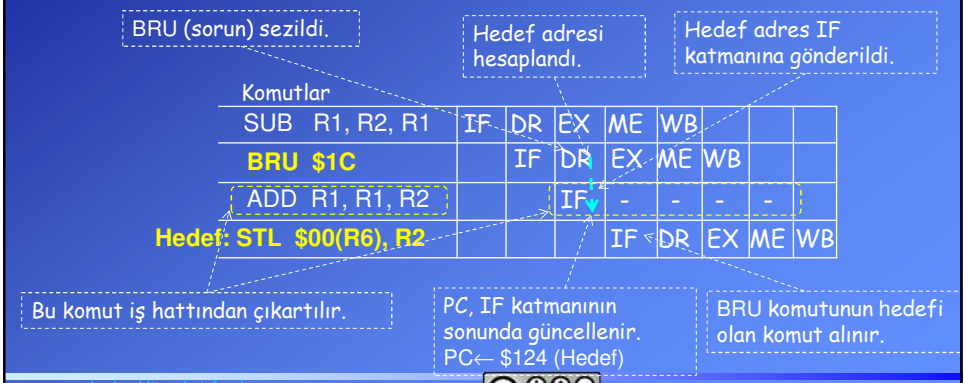
Denetim Sorunlarının (*Control Hazards*) Çözümleri:

A) Durdurma/boşaltma (*Stalling/flushing*) (donanım tabanlı):

Ek bir donanım birimi, sorunu sezer ve dallanmanın hedefi olan komut alınıncaya kadar iş hattını durdurur.

Hem koşulsuz hem de koşullu dallanmalarda uygulanabilir.

Örnek: Koşulsuz dallanma, hedef adres hesabı DR'de yapılıyor.



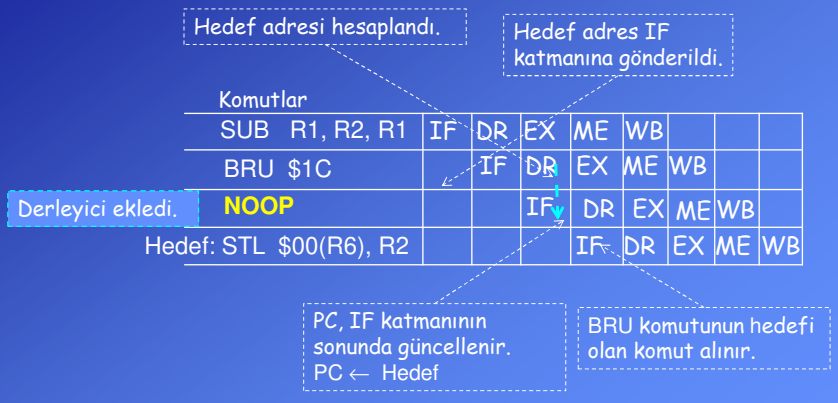
Denetim Sorunlarının (*Control Hazards*) Çözümleri (devamı):

B) NOOP (*No Operation*) komutlarının eklenmesi (Yazılım tabanlı):

Derleyici, dallanma komutundan sonra gerektiği kadar NOOP komutu ekler.

Bu çözümün etkisi iş hattını durdurmakla aynıdır.

Örnek: Koşulsuz dallanma, hedef adres hesabı DR'de yapılıyor.

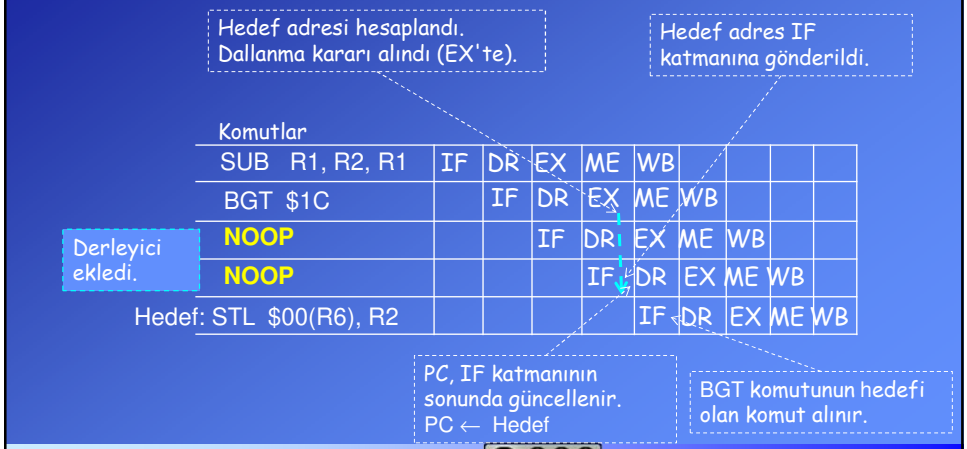


B) NOOP (*No Operation*) komutlarının eklenmesi (devamı):

Gerekli olan NOOP komutlarını sayısı iş hattını ne kadar durdurmak gerektiğine bağlıdır.

Örnek: Koşullu dallanma; adres hesabı ve dallanma kararı işlemleri EX'te.

Bu durumda 2 çevrim gecikmeye gerek olduğundan 2 tane NOOP eklenir.



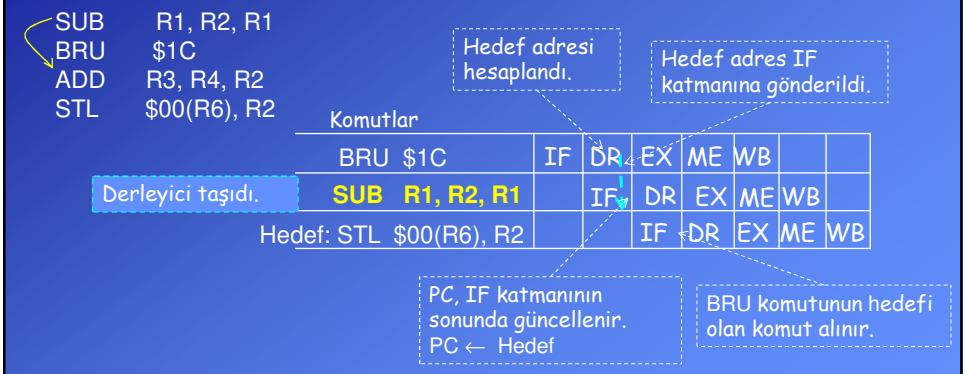
Denetim Sorunlarının (*Control Hazards*) Çözümleri (devamı):

C) Optimize Çözüm (Software-based):

Derleyici, eğer mümkünse programda uygun komutların yerini değiştirerek bu komutları dallanma komutunun peşine yerleştirir.

Bu değişiklik algoritmayı değiştirmemeli ve başka çatışmalara neden olmamalı.

Örnek: Koşulsuz dallanma, hedef adres hesabı DR'de yapılıyor.



Eğer optimize çözüm mümkünse **dallanma cezası oluşmaz**.

C) Optimize Çözüm (devamı):

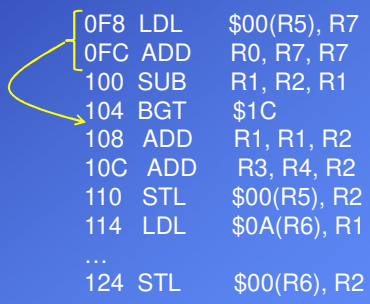
Taşınması gerekli olan komutlarını sayısı iş hattını ne kadar durdurmak gerektiğine bağlıdır.

Bu değişiklik algoritmayı değiştirmemeli ve başka çatışmalara neden olmamalı.

Örnek: Koşullu dallanma; adres hesabı ve dallanma kararı işlemleri EX'te.

Bu durumda 2 çevrim gecikmeye gerek olduğundan 2 tane komut, dallanma komutunun peşine taşınır.

Bu 2 komut, dallanma komutunun peşine taşınabilir.



Komutların sırasını değiştirmek ile ilgili önemli noktalar:

Dallanmadan **önce gelen** gelen bir komut dallanmadan sonraya kaydırılabilir.

Dallanmanın koşulu veya hedef adresi kaydırılan komuta bağlı olmamalı.

Bu yöntem (eğer mümkünse) her zaman performansı arttırır (NOOP'a göre).

Özellikle **koşullu dallanmalarda** bu yöntem dikkatli uygulanmalı.

Dallanmanın bağlı olduğu koşulu belirleyen komut dallanmadan sonraya taşınamaz.

Bu durumda NOOP eklenir.

Diğer seçenekler:

Derleyici taşımak üzere şu komutları seçebilir:

- **Dallanmanın hedefinden** (gidilecek yerden)

- Taşınan komut dallanma gerçekleşmese de çalışacaktır. Bu programı etkilememeli.
- Dallanma gerçekleşirse performans artar.

- **Dallanma komutunun peşinden** (dallanma olmazsa devam edilen kol)

- Taşınan komut dallanma gerçekleşse de çalışacaktır. Bu programı etkilememeli.
- Dallanma gerçekleşmezse performans artar.

Denetim Sorunlarının (*Control Hazards*) Çözümleri (devamı):

D) Dallanma Öngörüsü (*Branch Prediction*):

Hatırlatma: Dallanma komutları nedeniyle iş hattında iki temel problem oluşur.

1. Dallanma komutunu hedef adresi iş hattının IF'den sonraki katmanlarında hesaplanır.

   Bu nedenle, dallanma sonucu **hangi hedef komutun iş hattına alınacağı**, işlemci hedef adresi hesaplayana kadar **belli değildir**.

   PC ← PC + offset

   a) Eğer adres hesabı EX katmanında yapılır ve sonuç EX/ME saklayıcılarından IF katmanına gönderilirse (yansı 2.30), dallanma cezası: 3 çevrim.

   b) Eğer adres hesabı EX katmanında yapılır ve sonuç doğrudan IF katmanına gönderilirse (yansı 2.51), dallanma cezası: 2 çevrim.

   c) Eğer adres hesabı DR katmanında yapılır ve sonuç doğrudan IF katmanına gönderilirse (yansı 2.53), dallanma cezası: 1 çevrim (sadece koşulsuz dallanma komutları için geçerlidir).

   Hedef adresi önceden belirleyip bu sorunu çözmek için **dallanma hedef tablosu** (*branch target table*) kullanılır (yansı 2.64).

   Dallanma hedef tablosu IF katmanında yer alan, dallanma komutlarının ve bu komutların dallanacağı hedeflerin adreslerini tutan bir cep bellektir (*cache*).

Dallanma komutları nedeniyle iş hattında iki temel problem oluşur (devamı):

2. **Koşullu dallanma** sorunu: Dallanmadan önceki komut yürütülünceye kadar bayrakların değeri belli olmadığından dallanmanın gerçekten olup olmayacağı belli değildir.

   Dallanma olmazsa PC ← PC + 4 (örnek RISC işlemcisi için)

   Dallanma olursa PC ← PC + offset

   a) Eğer dallanma karar lojiği ME katmanındaysa (EX'ten sonra) (yansı 2.30), dallanma cezası: 3 çevrim.

   b) Eğer dallanma karar lojiği EX katmanındaysa (yansı 2.51), dallanma cezası : 2 çevrim.

   Bu problemi çözmek için **dallanma öngörü** (*branch prediction*) yöntemleri kullanılır.

   Koşullu dallanma komutu ile karşılaşıldığında dallanma öngörüsü yöntemleri dallanmanın olup olmayacağını öngörmeye çalışırlar.

   Öngörü sonucuna göre bellekteki bir sonraki komut veya dallanmanın hedefi olan komut iş hattına alınır.

D) Dallanma Öngörüsü (*Branch Prediction*) (devamı):

Koşullu dallanma komutu ile karşılaşıldığında dallanma öngörüsü yöntemleri dallanmanın olup olmayacağını öngörmeye çalışırlar.

Öngörü sonucuna göre bellekteki bir sonraki komut veya dallanmanın hedefi olan komut iş hattına alınır.

Eğer öngörü doğru çıkarsa dallanma cezası olmaz.

Öngörü yanlış olursa iş hattı durdurulur ve boşaltılır.

İki tür dallanma öngörüsü yöntemi vardır; **statik** ve **dinamik**.

Statik dallanma öngörüsü stratejileri:

a) "Her zaman dallanma yok" öngörüsü: Her zaman dallanma olmayacağı öngörülür ve bellekte dallanmadan sonra gelen komut iş hattına alınır.

b) "Her zaman dallanma var" öngörüsü : Her zaman dallanma olacağı öngörülür ve dallanmanın hedefi olan komut iş hattına alınır dallanma hedef tablosu gereklidir.

Programların davranışını inceleyen çalışmalar, koşullu dallanmaların %50'sinden fazlasında dallanmanın gerçekleştiğini göstermişlerdir.

Bu nedenle "her zaman dallanma var" öngörüsü performans açısından daha iyi sonuç vermektedir.

D) Dallanma Öngörüsü (*Branch Prediction*) (devamı):

Önceden komut alma (*Target Instruction prefetch*): **Dallanma hedef tablosu**

"Her zaman dallanma var" stratejisi: Her zaman dallanmanın hedef komutu alınır.

Ancak dallanma adresi hesaplanmadan önce hedef komutun adresi **belli değildir**.

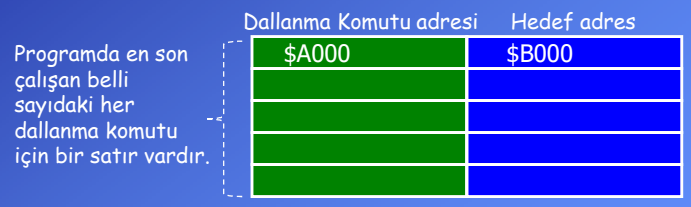
Dallanmanın hedef adresini daha önceden belirleyebilmek **dallanma hedef tablosu (*branch target table*)** kullanılır.

Dallanma hedef tablosu: Son çalışan belli sayıdaki dallanma komutunun adresleri ve son çalıştıklarında nereye gidildiği cep bellekte (*cache memory*) (bkz 6) tutulur.

Son zamanlarda yürütülen her dallanma komutu için ayrı bir satır bulunmaktadır.

Tutulan komut sayısı tablo boyutu ile sınırlıdır.

Tablo sayesinde hedef adres hesaplanmadan önce dallanma komutunun dallanacağı adresteki komutlara erişilebilir.



Örnek:

```
....
$A000  BGT Hedef
....        .....
....        .....
$B000 Hedef ADD ...
```

D) Dallanma Öngörüsü (*Branch Prediction*) (devamı):

Dinamik dallanma öngörüsü stratejileri:

Dinamik dallanma öngörüsü stratejileri o anda çalışan programdaki tüm koşullu dallanma komutlarının geçmişi ile ilgili istatistik tutarak dallanmanın olup olmayacağını öngörmeye çalışırlar.

Programdaki her koşullu dallanma komutu ile bir veya daha fazla sayıda **öngörü biti** (veya sayaç) (*prediction bits*) ilişkilendirilir.

Komutların geçmişi ile ilgili bilgi (daha önceki çalışmalarda dallanma olup olmadığı) sağlayan bu bitler bir **dallanma geçmişi tablosunda** (*branch history table*) tutulur (yansı 2.67).

1 bit dinamik dallanma öngörü yöntemi :

Her koşullu dallanma komutu için dallanma geçmişi tablosunda bir **öngörü biti (p_i)** tutulur.

p_i , i. koşullu dallanma komutunun öngörü bitidir.

Öngörü biti, ilgili komutun son çalışmasında dallanma olup olmadığını gösterir.

Eğer komutun son çalışmasında dallanma olduysa bir sonraki çalışmasında da dallanma olacağı varsayılır.

Algoritma:

i. Koşullu dallanma komutunu al

Eğer ($p_i = 0$) ise öngörü: "dallanma YOK", bellekte sıradaki komutu al

Eğer ($p_i = 1$) ise öngörü: "dallanma VAR", dallanmanın hedefi olan komutu al

Eğer dallanma gerçekten olursa $p_i \leftarrow 1$

Eğer dallanma gerçekten olmazsa $p_i \leftarrow 0$

Dallanma hedef tablosu ve dallanma geçmişi tablosu (*branch history table*):

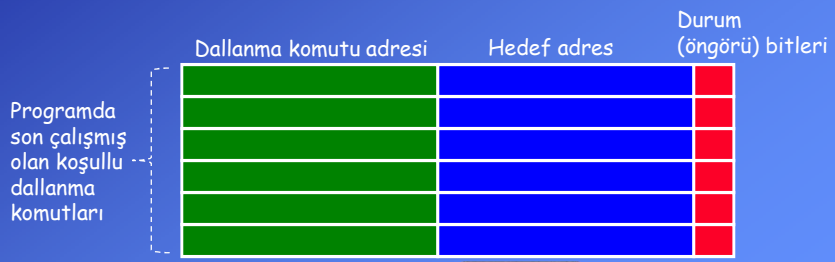
Öngörü bitleri hızlı erişilebilen bir bellekte oluşturulan dallanma geçmişi tablosunda (*branch history table* - BHT) tutulur.

Dallanma geçmişi tablosunda, en son çalışan belli sayıdaki her koşullu dallanma komutu için komutun bellek adresi, hedef adresi ve durum (öngörü) bitleri tutulur.

Öngörü bitleri dallanma komutunun her çalışmasında dallanma olup olmamasına göre değer alırlar.

Koşullu dallanma komutu tekrar çalıştığında bu bitler iş hattı denetim birimi tarafından karar vermek için kullanılır.

Eğer "dallanma VAR" öngörüsü yapılırsa dallanma komutu yürütülmeden önce tablodaki hedef adresi kullanılarak gidilecek olan komut iş hattına alınabilir.



Örnek: 1 bitlik öngörü yöntemi ve döngüler

Öngörü yöntemleri özellikle döngülerde yararlı olur.

Örnek:

```
        counter ← 100       ; saklayıcı veya bellek gözü
LOOP    ----                ; döngüdeki komutlar
        ----
        Decrement counter
        BNZ  LOOP           ; Branch if Not Zero (koşullu dallanma, p biti vardır)
        ----                ; döngüden sonraki komut
```

Program çalışmaya başladığında BNZ komutunun p biti 1'dir (dallanma VAR öngörüsü).

Döngünün ilk çalışmasında BNZ'de doğru öngörü yapılacak ve döngünün başındaki komut iş hattına alınacak.

p bitinin değeri (p=1) döngünün son çalışmasına kadar değişmeyecek.

Döngünün son çalışmasında p biti hâlâ 1'dir ve "dallanma VAR" öngörüsü yapılır; ama counter sıfır olduğu için program döngünün başına dallanmaz ve döngüden sonraki komut ile devam eder (yanlış öngörü). p sıfır yapılır (p ← 0).

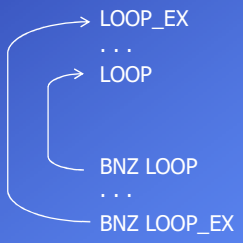
Sonuç olarak 100 defa dönen bir döngüde 99 defa doğru, sadece bir defa yanlış öngörü yapılmış oldu.

Döngüden sonra BNZ'nin p biti 0'dır, çünkü son çalışmada dallanma olmamıştır.

Aynı döngü başka bir döngünün içinde olduğu için tekrar çalıştığında ne olur?

1 bit dinamik dallanma öngörü yöntemindeki sorun:
(İç içe döngüler)

Birden fazla defa çalışan (içteki) döngülerde her defasında iki defa yanlış öngörü yapılır;
biri döngü ilk çalıştığında, diğeri de döngüden çıkarken.



Hatırlayın; önceki örnekte döngüden sonra BNZ'nin p biti 0'dır.

İçteki döngüye tekrar gelindiğinde, ilk çalışmada BNZ'deki öngörü "dallanma YOK" olacaktır (p=0) .

Ancak program dallanarak döngünün başına dönecektir (birinci hata).

Şimdi p biti 1 olur, çünkü dallanma olmuştur (p ← 1).

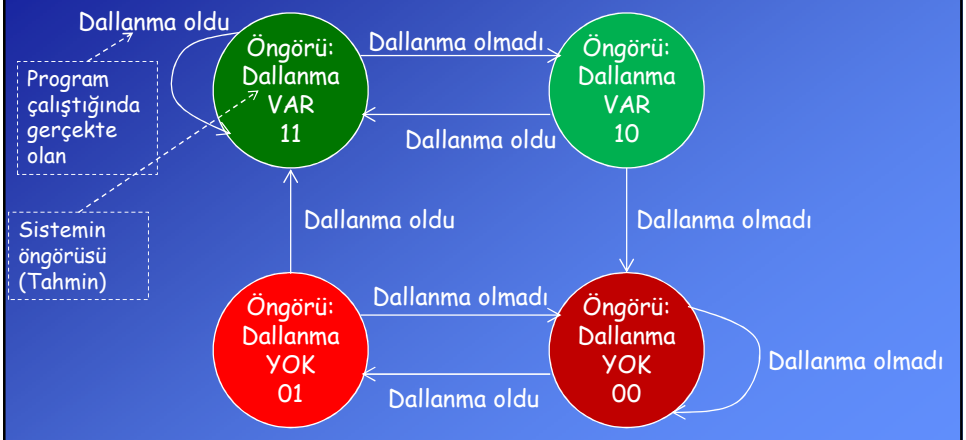
Döngünün son çalışmasına kadar öngörüler doğru olacaktır.

Döngünün son çalışmasında önceki örnekte de gösterildiği gibi yine hatalı öngörü yapılır (ikinci hata).

2 bit dinamik dallanma öngörü yöntemi:

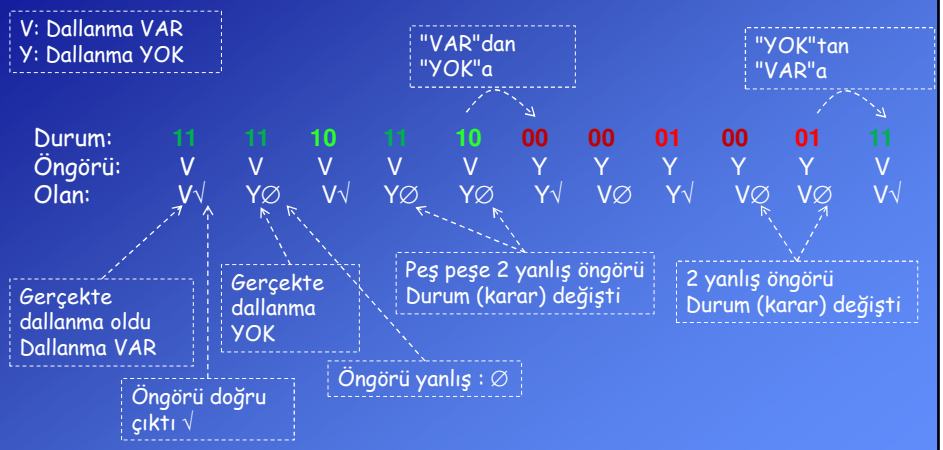
Her koşullu dallanma komutuna iki öngörü (durum) biti atanır.

Eğer komut 11 veya 10 durumlarındaysa "dallanma VAR" öngörüsü yapılır.
Eğer komut 00 veya 01 durumlarındaysa "dallanma YOK" öngörüsü yapılır.



Bu yöntemde ancak **peş peşe iki defa** yanlış öngörü yapılırsa öngörü kararı değişir.

Örnek: 2 bit dinamik dallanma öngörüsü



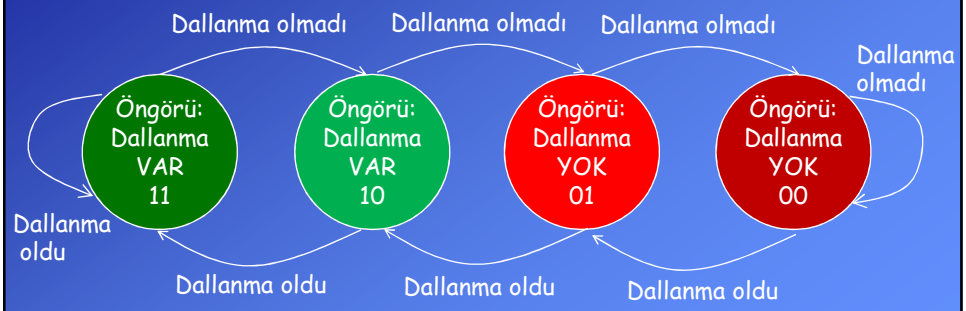
Doyan sayaç (*Saturating counter*):

Diğer bir 2 bit dinamik dallanma öngörü yöntemi:

Dallanma öngörüsü yöntemlerinin sonlu durumlu makineleri (*finite state machine*) farklı şekillerde tasarlanabilir.

Doyan sayaç alternatif bir öngörü yöntemidir. Durum geçişleri farklıdır.

Eğer komut 11 veya 10 durumlarındaysa "dallanma VAR" öngörüsü yapılır.
Eğer komut 00 veya 01 durumlarındaysa "dallanma YOK" öngörüsü yapılır.



Örnek:

Problem:

Bir MİB'te dallanma sorunlarını çözümünde donanım tabanlı yöntemlerin kullanıldığı bir iş hattı (*pipeline*) bulunmaktadır.

Bu MİB'te aşağıda verilen ve iç içe iki döngü içeren kod parçası çalıştırılmaktadır.

```
                  Counter1 ← 10
----> LOOP1       ------                    ; Herhangi bir komut
                  Counter2 ← 10
  --> LOOP2       ------                    ; Herhangi bir komut
                  ------                    ; Herhangi bir komut
                  Counter2 ← Counter2 - 1
  ----------------BNZ   LOOP2               ; Sıfır değilse dallan
                                            (Branch if not zero)
                  ------                    ; Döngüden sonraki komut
  ----------------Counter1 ← Counter1 - 1
                  BNZ   LOOP1               ; Sıfır değilse dallan
                  ------                    ; Döngüden sonraki komut
```

Farklı dallanma öngörüsü yöntemlerinin kullanılması durumunda, yukarıda verilen kod parçasındaki iki dallanma komutunun (BNZ) yürütülmesinde oluşan doğru ve hatalı dallanma öngörülerinin sayılarını veriniz.
Yanıtlarınızı kısaca açıklayınız.

Çözüm:

a. Statik öngörü

i) Her zaman "dallanma var"

BNZ LOOP1: Sadece son yinelemede döngüden çıkarken yanlış öngörü olur; diğer öngörüler doğrudur.

Doğru: 9 Yanlış: 1

BNZ LOOP2: Sadece son yinelemede döngüden çıkarken yanlış öngörü olur; diğer öngörüler doğrudur.

Doğru: 10x9 = 90 Yanlış: 10x1 = 10

Toplam: Doğru: 99 Yanlış: 11

ii) Her zaman "dallanma yok"

BNZ LOOP1: Sadece son yinelemede döngüden çıkarken doğru öngörü olur; diğer öngörüler yanlıştır.

Doğru: 1 Yanlış: 9

BNZ LOOP2: Sadece son yinelemede döngüden çıkarken doğru öngörü olur; diğer öngörüler yanlıştır.

Doğru: 10x1 = 10 Yanlış: 10x9 = 90

Toplam: Doğru: 11 Yanlış : 99

Çözüm (devamı):

b. Bir bitlik dinamik öngörü yöntemi

Dikkat: Her dallanma komutu için ayrı bir öngörü biti kullanılır (Yansılar 2.66, 2.67).

i) Başlangıç kararı "dallanma var"

BNZ LOOP1:
Sadece son yinelemede döngüden çıkarken yanlış öngörü olur; diğer öngörüler doğrudur.

Doğru: 9 Yanlış: 1

BNZ LOOP2:

Döngünün ilk çalışmasında sadece son yinelemede döngüden çıkarken yanlış öngörü olur; diğer öngörüler doğrudur.

Döngüden çıkıldığında öngörü biti p "dallanma yok" olarak değişir. Bu nedenle döngünün 2.-10. çalışmalarında hem ilk hem de son yineleme de hatalı öngörü olur (Yansı 2.69).

Doğru: 9 + 9x8 = 81 Yanlış: 1+ 9x2 =19

Toplam: **Doğru: 90** **Yanlış: 20**

b. Bir bitlik dinamik öngörü yöntemi **(devamı):**

ii) Başlangıç kararı "dallanma yok"

BNZ LOOP1:
İlk ve son yinelemelerde yanlış öngörü olur; diğer öngörüler doğrudur.

Doğru: 8 Yanlış: 2

BNZ LOOP2:
İlk ve son yinelemelerde yanlış öngörü olur; diğer öngörüler doğrudur.

Doğru: 10x8 = 80 Yanlış: 10x2 =20

Toplam: **Doğru: 88** **Yanlış: 22**

c. İki bitlik dinamik öngörü yöntemi:

i) Başlangıç kararı "dallanma var"

BNZ LOOP1: Sadece son yinelemede döngüden çıkarken yanlış öngörü olur; diğer öngörüler doğrudur.

Doğru: 9 Yanlış: 1

BNZ LOOP2: Sadece son yinelemede döngüden çıkarken yanlış öngörü olur; diğer öngörüler doğrudur.

Doğru: 10x9 = 90 Yanlış: 10x1 = 10

Toplam: Doğru: 99 Yanlış: 11

ii) Başlangıç kararı "dallanma yok"

BNZ LOOP1: Birinci, ikinci ve son yinelemelerde yanlış öngörü olur.
Hatırlatma, bu yöntemde karar iki yanlış öngörü-den sonra değişir.

Doğru: 7 Yanlış: 3

BNZ LOOP2: Döngünün ilk çalışmasında; birinci, ikinci ve son yinelemelerde yanlış öngörü olur. Döngünün ilk çalışmasından sonra karar hala "dallanma var" şeklindedir. Bu nedenle, döngünün 2.-10. çalışmasında sadece son yinelemede hatalı öngörü olur.

Doğru: 7+ 9x9 = 88 Yanlış: 3 + 9x1 = 12

Toplam: Doğru: 95 Yanlış: 15