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.

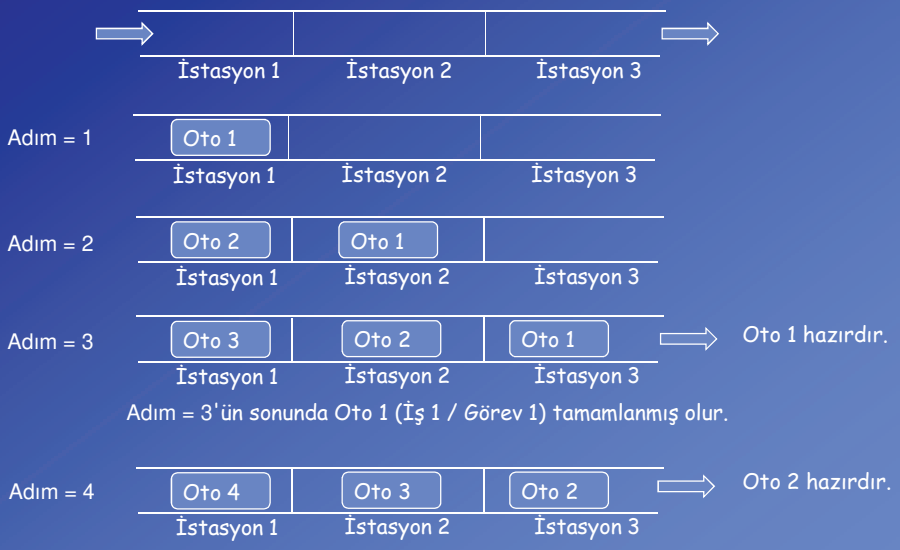
İş (otomobilin montajı), küçük alt işlemlerden oluşur; kapıların takılması, tekerleklerin montajı, camların takılması.

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

Bu istasyonlarda aynı anda paralel olarak farklı otomobiller üzerinde çalışılır.

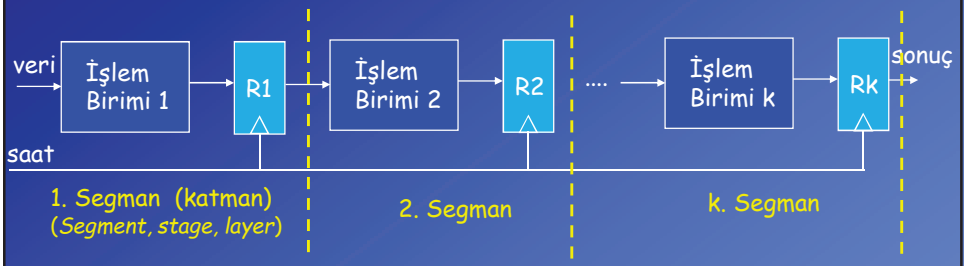
Örneğin i. işçi bir otomobilin camını takarken aynı anda (i+1). sıradaki işçi bir önceki otomobilin tekerleklerini takmaktadır.

Örnek: Bir otomobil fabrikasındaki üç istasyonlu üretim/montaj bandı



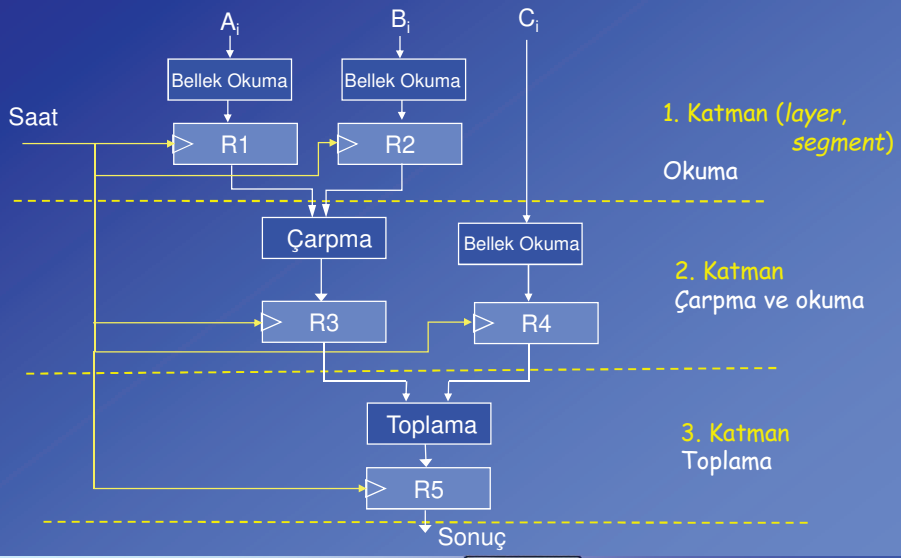
Adım = 3'ten sonra (tüm istasyonlar dolduktan sonra) her adımda yeni bir Oto (İş) tamamlanır.

2.1 Bir iş hattının genel yapısı:



- Her katman (işlem birimi); belli, sabit bir işi yapar.
- Her saat çevriminde (*clock cycle*) işlem birimi farklı bir veri (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 işin bütün adımları tamamlanmadan (sonuç üretilmeden) önce iş hattının girişinden yeni işle ilgili veriler alınır.
- İş hattının bütün segmanları (katmanları) 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*): $S = \frac{\textit{İş hattı } \textbf{\textit{olmadan}} \textit{ gereken süre}}{\textit{İş hattı } \textbf{\textit{ile}} \textit{ gerekli olan süre}}$ $S = \frac{n \cdot t_n}{(k+n-1) \cdot t_p}$

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.

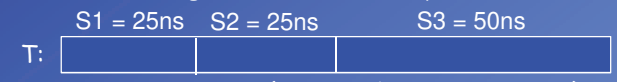
Örnek olarak toplam süresi 100 ns olan bir T işini ele alalım.

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 $\tau_M = 50$ns)

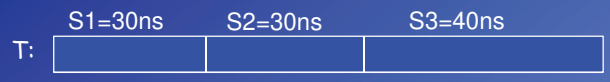
İş hattında daha fazla katman olmasına rağmen Durum A'ya göre bir hızlanma sağlanmamıştır.

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 $\tau_M = 40$ns)

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.

Çevrim:	1	2	3	4
Komut 1	Komut al, çöz	Operand al, yürüt		
Komut 2		Komut al, çöz	Operand al, yürüt	
Komut 3			Komut al, çöz	Operand al, yürüt

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 :

Saat çevrimi / Komutlar	1	2	3	4	5
ADD R1, **R2** (**R2** ← R1+R2)	FI	DA	FO	EX	
SUB **R2**, R3 (R3 ← **R2**-R3)		FI	DA	FO	EX

R2 güncelleniyor

Operand bağımlılığı

R2'nin geçerli olmayan eski değeri alınıyor.

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.

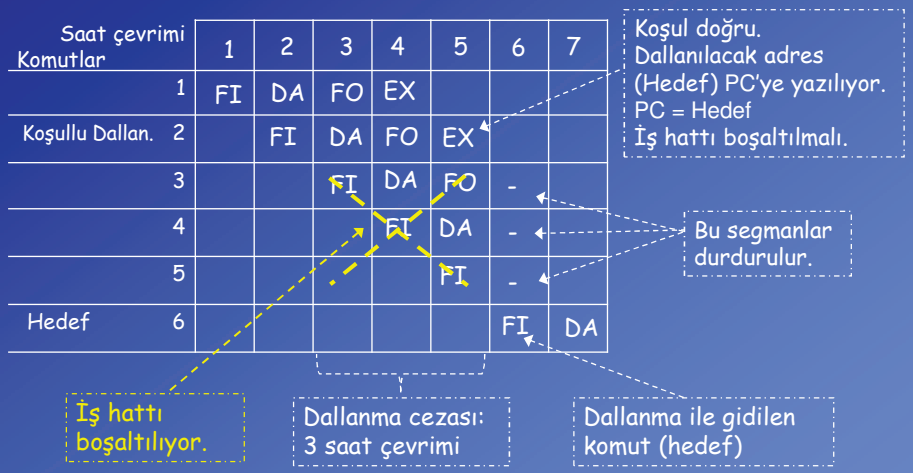
Dallanma olmadığı için boşaltılmayacak (ceza yok).

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):



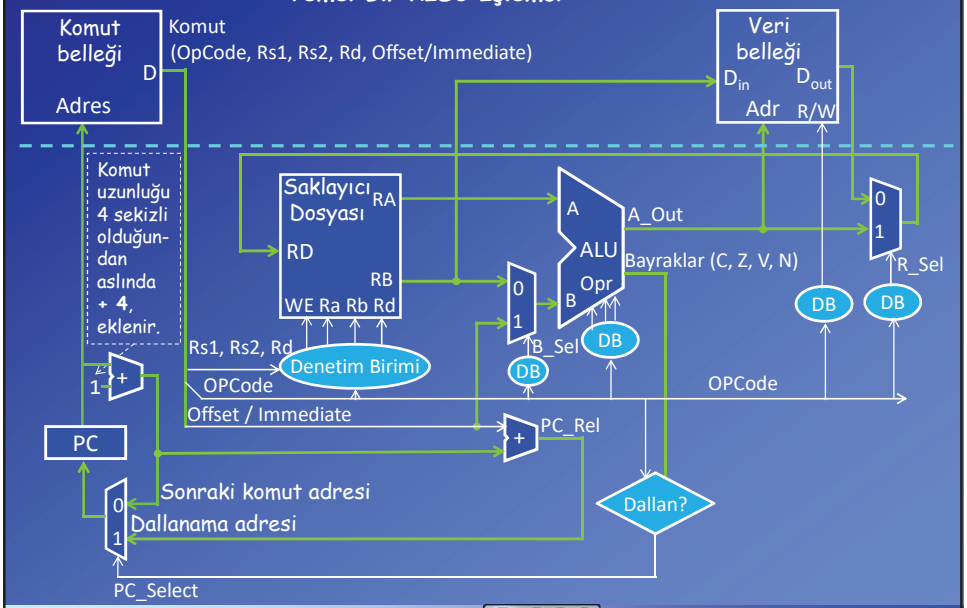
Dallanma cezasının süresi iş hattındaki segmanların sayısına ve işlevlerine bağlıdır.

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 RISC 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