YILDIZ TEKNİK ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

SÜPERSKALER MİMARİLERDE PERFORMANS ARTIŞI İÇİN BİR MODEL

Bilg. Bil. Müh. Emre ÖZER

F.B.E. Bilgisayar Bilimleri Mühendisliği Anabilim Dalında hazırlanan

YÜKSEK LİSANS TEZİ

Tez Danışmanı : Yrd. Doç. Dr. Selim AKYOKUŞ

İSTANBUL, 1996

İÇİNDEKİLER

TEŞEKKÜR

Tezimin başlangıcından sonuna kadar bana yardımlarını esirgemeyen , hem tez sırasında hem de aldığım derslerde kendisinden çok şey öğrendiğim değerli hocam **Doç. Dr. Bülent Örencik**' e teşekkürlerimi sunmayı bir borç bilirim. Ayrıca, desteklerinden dolayı Bölüm Başkanımız **Prof. M. Yahya Karslıgil**' e ve tez danışmanım **Yrd. Doç. Dr. Selim Akyokuş**' a teşekkür ederim.

TÜRKÇE ÖZET

Süperskaler işlemciler RISC tabanlı ve birden çok işhattı olan işlemcilerdir Birden fazla komut aynı anda işlemciye yayınlanabilir. Bu komutlar arasında veri bağımlılığı ve kaynak çatışmalarının ortadan kaldırılmış olması gerekmektedir. Bir çevrim süresinde süperskaler işlemciye ne kadat çok komut yayılanırsa, performans o denli artacaktır. Özellikle, komutlar program sırası dışında işlemciye yayınlandığında, süperskaler işlemciler en yüksek performansa ulaşırlar.

Program sırası dışında komut yayınlandığında oluşacak bir kesme durumunda kesmenin kotarılması, program sırasına göre komut yayınlayan bir işlemcideki kesmenin kotarılmasına göre daha karmaşık olmaktadır. Kesmelerin kotarılması için geliştirilmiş olan çeşitli çözüm yöntemleri vardır. Bu yöntemler yeniden sıralama tamponu, tarih dosyası , gelecek dosyası gibi yöntemlerdir. Bu yöntemlerin hepsi kesmeleri precise olarak kotarmaktadır.

Bu tezde kesme kotarma işlemi için farklı bir yöntem önerilmiştir. Bu yöntem, kesmelere imprecise olarak yaklaşmaktadır. Yönteme Gölge Tutucu Yığın yöntemi denmiştir. Önerdiğimiz bu yöntem yapılan simülasyonla önceki yöntemlerle hem performans hem de maliyet olarak karşılaştırıldı. Elde edilen verilere dayanarak gölge tutucu yığın yönteminin başarılı olduğu sonucuna varılmıştır.

ABSTRACT

Superscalar processors are RISC-based machines with multiple instruction pipelines. Many instructions can be issued simultaneously and many results can be generated per cycle. Independent instructions can be executed in parallel in superscalar processors. Superscalar processors achieve high performance as they simultaneously dispatch multiple independent instructions and obtain more than one result per cycle.

Potentially, out-of-order issue of instructions results in higher performance than in-order issue; but in this case the interrupt handling mechanism becomes much more complicated. There are some interrupt handling hardware solutions which use precise interrupt model. These methods are reorder buffer, history file, future file which are all precise interrupt handling schemes.

In this thesis, an imprecise interrupt handling mechanism is proposed for superscalar processors which employ out-of-order issue policy to achieve high performance. The proposed scheme is called shadow latch stacks. We made performance and hardware cost comparison between shadow latch stack and the other schemes. The simulation study showed that shadow latch stack seemed to be promising in superscalar processors.

I. GİRİŞ

Bilgisayar mimarisindeki gelişim son derece hızlı olmaktadır. Bilgisayar mimarisi , hem donanım organizasyonuyla hem de donanım üzerinde çalışacak yazılımlarla ilgilenmektedir. Donanım açısından bakıldığında bilgisayar mimarisinde komut kümesi, adresleme yapıları, saklayıcılar (registers), bellek yapısı, önbellek (cache) organizasyonu, veri ve adres yolları mimarinin temel yapı taşlarını oluşturmaktadır. Bu yapı taşları, tasarlanan bir mikroişlemcinin performansını ve maliyetini önemli ölçüde etkilemektedir.

Mikroelektronik teknolojisinde yaşanan hızlı değişimler, bilgisayar mimarisinde son derece hızlı ilerlemelere sebeb olmuştur. Daha küçük boyutlu mikroişlemciler geliştirilmiştir. Yeni teknolojiyle geliştirilen küçük bir mikroişlemci eski teknolojiyle geliştirilmiş olan daha büyük bir mikroişlemciyle aynı sayıda transistör içerebilmektedir. Daha eski teknolojiyle üretilen mikroişlemcilerde boyut büyük olduğundan, performansı artırmak için eklenecek olan her donanım, sisteme ek maliyet getirecektir. Bu sebeble, bilgisayar mimarları performans / maliyet ilişkisini gözönüne alarak düşük performansa sahip mikroişlemciler üretmişlerdir. Mikroelektronik geliştikçe daha yüksek performanslı mimariler geliştirmek mümkün olmaya başladı.

İlk tasarlanan mimarilerden olan Von Neumann mimarisidir. Bu mimarinin en önemli özelliği ardışıl (sequential) makina olmasıdır. Ardışıl makina, yazılmış olan bilgisayar programını programdaki sıraya göre işleten makinaya denmektedir. Programdaki sıra, program sayıcı (program counter) tarafından yapılmaktadır. Program sayıcı kavramı daha sonra geliştirilmiş olan mimarilerde de kullanılmıştır.

Von Neumann mimarisi ardışıl bir mimari olduğundan dolayı , mimarinin kullanıldığı sistem yavaş çalışacaktır. Bu sebeble Von Neumann mimarisindeki bir işlemciyi hızlandırmak için çeşitli yöntemler önerilmiştir. Bu yöntemlerden birisi öngörü (lookahead) tekniğidir. Öngörü tekniğinde belirli sayıda program komutu işlemci içerisine çekilir. Böylece bir bellek erişimiyle bir yerine birden çok komut daha hızlı olarak işlemci içerisine çekilir. Öngörü tekniği dışında önerilmiş olan diğer bir teknik ise çok sayıda fonksiyonel birimin işlemci içerisinde bulundurulmasıdır. Fonksiyonel birimler işlemci içerisinde komutları işleten (execute) birimlerine verilen addır. ALU (Arithmetic Logic Unit), çarpma / bölme devresi, dallanma / döngü birimi gibi birimler fonksiyonel birimlerdir. Fonksiyonel birimlerin artırılması ile işlemci içerisinde hazır durumda bekleyen komutlar aynı fonksiyonel birimi kullanan önceden gönderilmiş olan komutların tamamlanmasını beklemeden aynı fonksiyona sahip olan ve o anda boş olan başka bir fonksiyonel birimi kullanabilmektedirler. Bu durum işlemci içinde komutların daha hızlı olarak çalışmasını sağlar. Öngörü ve çoklu fonksiyonel birimler dışında çok önemli bir diğer teknikte işhattı (pipeline) tekniğidir. İşhattı yapısı tıpkı yürüyen bant yapısına benzer. Yürüyen bandın belirli noktalarında bulunan birimler belirli bir görevi yapmaktadır. Bant üzerinde ilerleyen her nesne bir birimde işlem gördükten sonra , bir sonraki birime başka bir işlem için gönderilir. İşhattı yapısında ise işlemci kendi

içinde fonksiyonel olarak çeşitli katlara bölünmüştür. Gelen komutlar bir katta işlem gördükten sonra diğer kata gönderilirler ve en son kat işlemi bitirdikten sonra komut işlemciden işletilmiş olarak çıkar. İşhattı yapısıyla işlemcinin performansı büyük ölçüde artmaktadır. Bu yapı bir çok ticari mikroişlemci tarafından kullanılmış ve halen kullanılmaya devam edilmektedir.

Tüm bu bahsedilen tekniklere rağmen işlemci bir çevrim süresi içinde sadece bir komut gönderme yapabilir. Bir çevrim süresinde bir komut gönderen işlemcilere **skaler** işlemciler denir. Skaler işlemciler hem RISC hem de CISC yapısında olabilir.

Süperskaler işlemciler, skaler işlemcilerin daha geliştirilmiş olan bir üst sınıfıdır. Süperskaler işlemcilerde bir çevrim süresi içinde, skaler işlemcilerden farklı olarak, işlemciye birden fazla komut göndermek mümkün olmaktadır. Bu özelliğinden dolayı işlemciye **süperskaler** işlemci denmektedir. Süperskaler işlemcilerin , skaler işlemcilere göre ölçülebilir bir performans üstünlüğü bulunmaktadır. Süperskaler işlemciler, işlemci içerisinde paralleliği sağlayan tek mimari yapısı değildir. Süperişhattı ve VLIW mimarileri işlemci içinde komut paralelliğini sağlayan yapılardır. Süperskaler mimari ile süperişhattı ve VLIW mimarileri karşılaştırılarak aralarındaki benzerlik ve farklar ortaya konulacaktır.

Sırasız komut yayınlayabilen bir süperskaler mimaride kesme oluştuğunda , kesmenin kotarılması karmaşık donanım çözümleri gerektirmektedir. Getirilen donanım çözümleri, skaler mimarilerde önerilen çözüm yöntemleri olan precise yöntemlerin temelde benzerleridir. Bu yöntemler dışında , hem performansı arttıracak hem de maliyeti fazla olmayan bir yöntem önerilebilir. Gölge tutucu yığını adı verilen bu yöntem kesmeleri imprecise olarak kotarır. Bu yöntem, kesme oluştuğunda işhattında bulunup tamamlanmamış olan komutları tutucu yığınlarda saklarlar. Kesme servis programından dönüşte ise tutucu yığınlarında bulunan bilgiler ana tutuculara geri yüklenirler. Böylece işhattı yeniden çalıştırılmaya hazır hale gelir. Bu yöntem, özellikle sırasız komut yayınlayabilen bir süperskaler işlemcide performansı büyük ölçüde arttırabilmektedir.

II. SKALER MİMARİLER

Skaler mimariler , bir çevrim süresi boyunca sadece bir komut gönderebilen mimarilerdir. Skaler yapıdaki işlemciler RISC veya CISC mimarinde olabilir. RISC ve CISC komut kümesine göre sınıflandırılmış, birbirinden farklı organizasyona sahip mimarilerdir.

2.1 CISC Mimarisi

CISC mimarisinde komut sayısı RISC mimarisine göre oldukça fazladır. Birçok farklı fonksiyon için komut bulunmaktadır. Çok sayıda komut olduğundan komut formatı değişken formattadır. Bellek erişimleri için birçok adresleme modu düşünülmüştür. Saklayıcı sayısı azdır. CISC mimarisinde her komut daha alt mikrokomutlara bölünmüştür. Bu mikrokomutlar mikroprogram ROM' undan işletilirler. CISC mimarisinde hem komutlar için hem de veriler için birleşik önbellek vardır; bu nedenle aynı veri ve komut yolunu kullanırlar. Çok sayıda işlem için bir komut varolduğundan derleyici (compiler) geliştirme CISC mimarisinde daha basit olmaktadır. CISC mimarisinde işhattı yapısı bulunmamaktadır. CISC mimarisinin genel yapısı Şekil 2.1' de gösterilmiştir (Hwang , 1993).

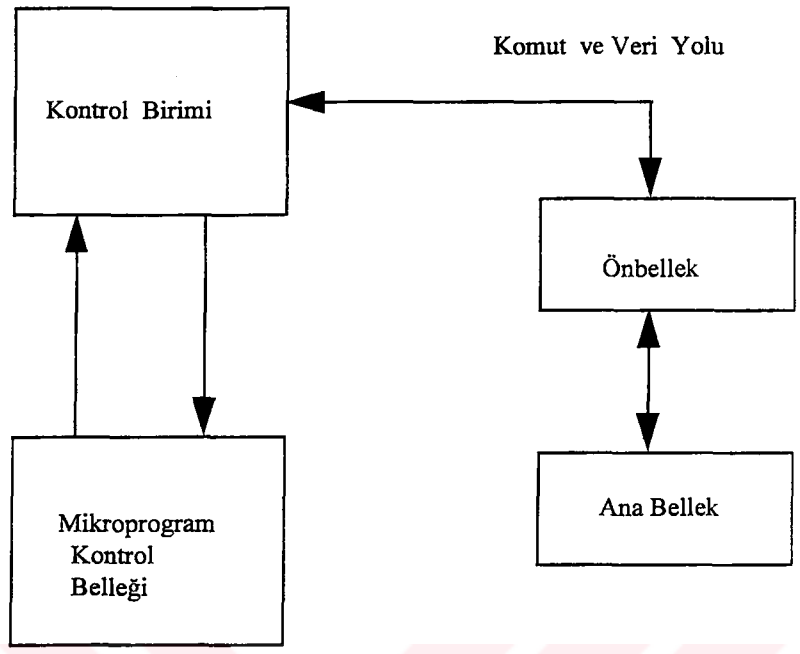
Tipik bir CISC mimarisi 120 ila 350 komut arasında komut içerebilmektedir. Saklayıcı sayısı ise 8 ila 24 arasındadır (Hwang , 1993).

CISC mimariye örnek olarak Intel 486 (Intel , 1989) işlemcisi verilebilir. 486 32 bitlik mimariye sahip ve işlemci içinde bellek yönetim birimi (memory management unit), kayan-nokta birimi ve önbellek bulunmaktadır. Önbellekte hem veri hem de program bulunmaktadır. Bellek yönetim biriminde hem segmentasyon hem de sayfalama (paging) yapılabilmektedir. Sayfalama birimi yazılımla kontrol edilebilmektedir. İşlemci içindeki önbellek 8KB büyüklüğündedir. Kayan nokta birimi tamsayı birimiyle paralel çalışmaktadır ve çeşitli aritmetik işlemleri icra etmektedir. Bu işlemler arasında tanjant, sinüs, cosinüs, logaritma gibi fonksiyonları işletebilmektedir.

486 mikroişlemcisinde 8 adet 32 bitlik saklayıcı bulunmaktadır. Bu saklayıcıların 16 bitlik kısımları saklayıcı olarak tasarlanmıştır. Ayrıca, işlemci içinde 6 adet segment saklayıcısı, program sayıcı ve bayrak saklayıcısı bulunmaktadır. 486 değişken komut formatına ve 11 adet adresleme moduna sahiptir.

2.2. RISC Mimarisi

CISC mimarilerinin komut sayısı çok fazla olduğundan, komut formatı değişken uzunlukta olmaktadır. Değişken uzunluklu komutların kodlarının çözülebilmesi, sabit uzunluklu komut formatına göre daha karmaşık ve uzun süre gerektirmektedir. Bu sebeble, RISC mimarisi önerilmiştir. RISC



Şekil 2.1 CISC Mimarisinin Genel Yapısı.

mimarisinde komut sayısı azdır ve yapıları oldukça basittir. Komutlar az sayıda olduğundan sadece temel işlemler için komutlar bulunmaktadır.

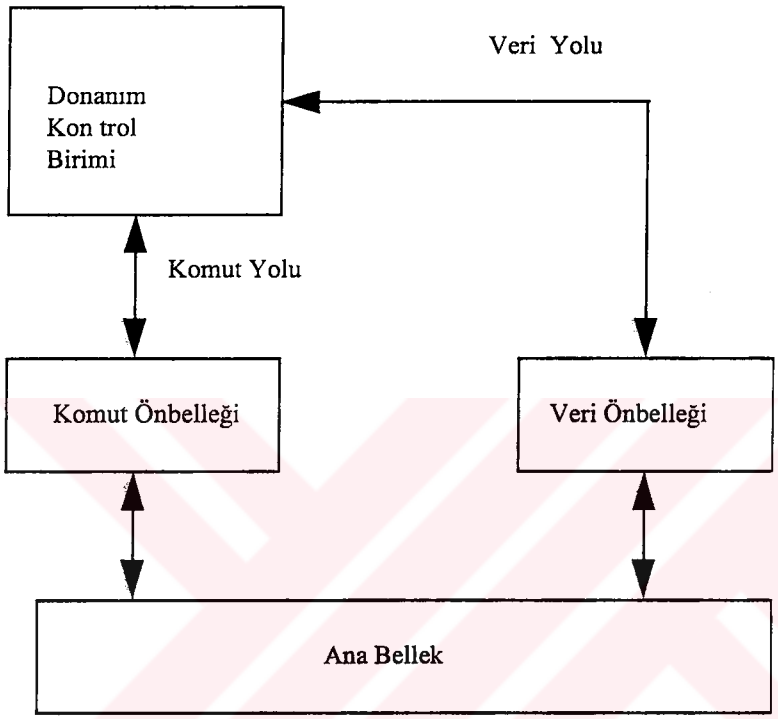
RISC mimarisinde komutlar sabit uzunluklu formattadır ve her komut tek çevrim zamanında işletilmektedir. Adresleme modlarının sayısı CISC' e göre daha azdır. Komutların tamamı saklayıcı üzerinden işlem yaparlar. Bellek erişimi load ve store komutlarıyla yapılmaktadır. Komutlar, mikroprogram ROM yerine doğrudan donanım tarafından işletilirler. Bu da işlemcinin daha hızlı komut işletmesini sağlar. İşlemci içinde bir mikroprogram olmadığından, RISC işlemcilerindeki saklayıcı sayıları artırılmıştır. Komut ve veriler için ayrı önbellekler bulunmaktadır. RISC mimarilerde işhattı yapısı kullanılmaktadır. Sadece temel işlemler için komutlar olduğundan derleyici geliştirmek CISC' de derleyici geliştirmekten daha karmaşıktır. RISC mimarisinin genel yapısı Şekil 2.2' de gösterilmiştir.

Tipik bir RISC mimarisinde 100' den az sayıda komut vardır ve komutların tümü 32 bitlik komut formatındadır. Saklayıcı sayısı 32' den fazladır ve 3 ila 5 arasında basit adresleme modu içermektedir.

RISC mimarisine örnek olarak MOTOROLA 88100 (Alsup , 1990) işlemcisi verilebilir. 88100 işlemcisi 32 bitlik mimariye sahip olan ve işlemci içerisinde kayan nokta birimi, bellek yönetim birimi ve önbellekler bulunmaktadır. İki adet önbellek bulunmaktadır. Her iki önbellekte 16KB büyüklüğünde olup veri ve program önbelleği olarak ayrılmışlardır. Önbelleklerin her birinde bellek yönetim birimi

bulunmaktadır. İşlemci içinde 32 adet 32 bitlik saklayıcı bulunmaktadır. İşlemci içinde 4 katlı komut işhattı yeralmaktadır. Komut formatı sabit ve 32 bit uzunluğundadır.

CISC ve RISC mimarilerinin özellikleri Tablo 2.1' de karşılaştırmalı olarak verilmiştir. RISC mimarisinde işhattı tekniği CISC mimarisine göre önemli ölçüde üstünlük sağlamaktadır. İşhattı yapısı, hem skaler hem de süperskaler işlemcilerde performans artışına sebeb olduğundan mimari olarak kritik öneme sahiptir.



Şekil 2.2 RISC Mimarisinin Genel Yapısı.

TABLO 2.1

	CISC	RISC
Komut Sayısı	Çok	Az
Komut Uzun.	Değişken	Sabit
Saklayıcı Sayısı	Az	Çok
Kontrol Yöntemi	Mikroprogram	Donanım
Derleyici	Kolay	Az
İşhattı	Yok	Var
Komut Başına Çevrim	> 1	= 1
Önbellek	Birleşik	Ayrı
Adresleme Modu	12 - 24	3 - 5

2. 3 İŞHATTI

İşhattı yapısında işlemci içerisinde bulunan tüm birimler ardarda gelecek şekilde tertiplenmiştir. Her birimin işlemci içerisinde farklı görevi vardır ve bu birimlere işhattı katları denmektedir. Her işhattı katı, diğer işhattı katlarıyla aynı anda çalışabilmektedir. Veri ve komutlar bir katta işlem gördükten sonra diğer kata gönderilirler ve en son katta ise yapılan işlem tamamlanır. Bu şekilde çalışan işhattı yapısına lineer işhattı yapısı denir.

Lineer işhattı senkron veya asenkron yapıda olabilir. Asenkron lineer işhattında bir katta yapılan işlem sonunda elde edilen sonuç, bir sonraki kata gönderilmeden önce sonucun hazır olduğunu belirten bir işaret gönderilir. Bir sonraki kat, sonucu aldığında kendinden önceki kata sonucun alındığını bildiren alındı mesajını gönderir. Asenkron lineer pipeline yapısı görüldüğü üzere elsıkışma (handshaking) protokolünü kullanmaktadır.

Senkron lineer işhattı yapısı Şekil 2.3' de gösterilmiştir. Her işhattı katından önce bir tutucu (latch) bulunmaktadır. Her saat darbesinde tutucular girişleri tutarlar. Belli bir süre giriş bilgilerini saklarlar. İşhattı katları kombinasyonel devreler olduklarından belli bir gecikme süresi içerisinde işlem yapmaktadırlar. Bu nedenle girişlerin tutucular tarafından belli bir süre tamponlanması gerekmektedir.

TABLO 2.1

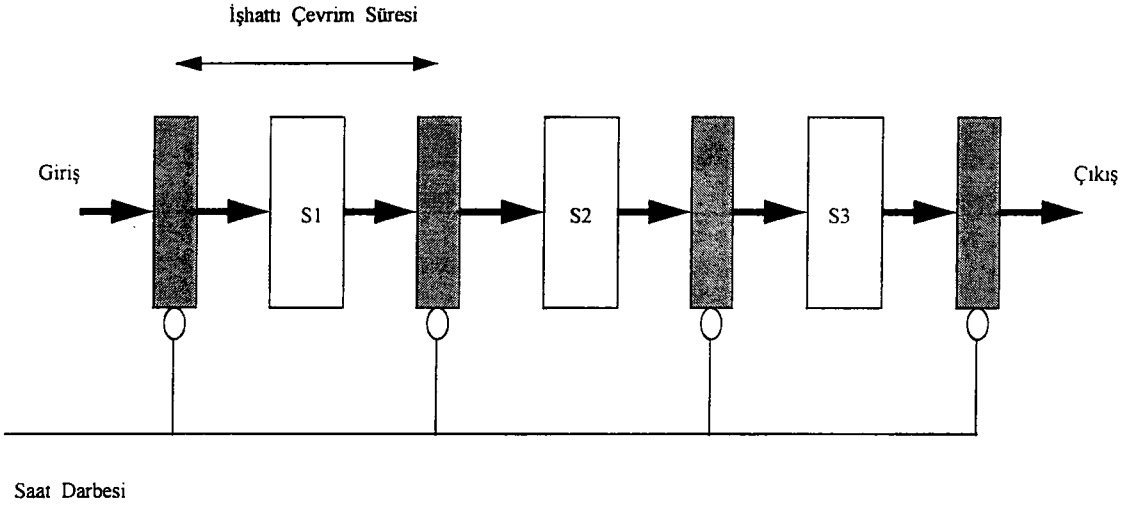
	CISC	RISC
Komut Sayısı	Çok	Az
Komut Uzun.	Değişken	Sabit
Saklayıcı Sayısı	Az	Çok
Kontrol Yöntemi	Mikroprogram	Donanım
Derleyici	Kolay	Az
İşhattı	Yok	Var
Komut Başına Çevrim	> 1	= 1
Önbellek	Birleşik	Ayrı
Adresleme Modu	12 - 24	3 - 5

2. 3 İŞHATTI

İşhattı yapısında işlemci içerisinde bulunan tüm birimler ardarda gelecek şekilde tertiplenmiştir. Her birimin işlemci içerisinde farklı görevi vardır ve bu birimlere işhattı katları denmektedir. Her işhattı katı, diğer işhattı katlarıyla aynı anda çalışabilmektedir. Veri ve komutlar bir katta işlem gördükten sonra diğer kata gönderilirler ve en son katta ise yapılan işlem tamamlanır. Bu şekilde çalışan işhattı yapısına lineer işhattı yapısı denir.

Lineer işhattı senkron veya asenkron yapıda olabilir. Asenkron lineer işhattında bir katta yapılan işlem sonunda elde edilen sonuç, bir sonraki kata gönderilmeden önce sonucun hazır olduğunu belirten bir işaret gönderilir. Bir sonraki kat, sonucu aldığında kendinden önceki kata sonucun alındığını bildiren alındı mesajını gönderir. Asenkron lineer pipeline yapısı görüldüğü üzere elsıkışma (handshaking) protokolünü kullanmaktadır.

Senkron lineer işhattı yapısı Şekil 2.3' de gösterilmiştir. Her işhattı katından önce bir tutucu (latch) bulunmaktadır. Her saat darbesinde tutucular girişleri tutarlar. Belli bir süre giriş bilgilerini saklarlar. İşhattı katları kombinasyonel devreler olduklarından belli bir gecikme süresi içerisinde işlem yapmaktadırlar. Bu nedenle girişlerin tutucular tarafından belli bir süre tamponlanması gerekmektedir.



Şekil 2. 3 Senkron Lineer İşhattı

Şekil 2.3' de üç katlı lineer bir işhattı yapısı görülmektedir. Şekilde koyu renkle gösterilen kutular tutucuları simgelemektedir. Her tutucu devresi bir saatle senkron bir biçimde çalışmaktadır. İşhattının saat çevrim süresi, işhattındaki en uzun gecikmeye sahip olan katın gecikmesi ile tutucu gecikmesinin toplamından ibarettir. Diğer katlar ne kadar hızlı olursa olsun, işhattı çevrim süresi en uzun gecikmeye sahip olan katın gecikmesine bağlı olacaktır. İşhattındaki her kat bir kez dolduğunda, her çevrim süresinde bir komut işletilmiş olmaktadır.

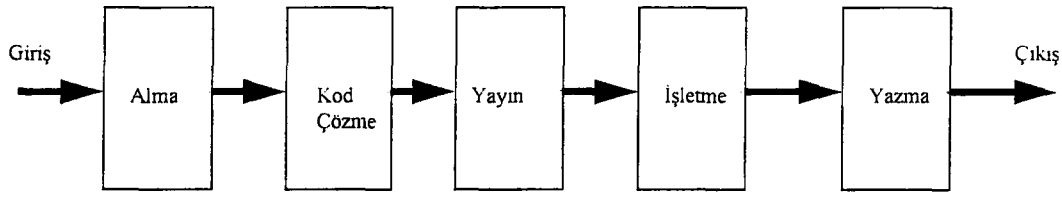
k katlı bir işhattına sahip bir mimarinin işhattını kullanmayan bir mimariye göre hızlanma oranı k kat sayısına ve programdaki komut sayısına bağlıdır. Programdaki komut sayısı arttıkça işhattının kullanımı artacağından hızlanma oranı artacaktır. Aynı şekilde, işhattındaki k kat sayısı artıkça hızlanma oranı artacaktır fakat kat sayısının artması, pratikte maliyeti arttıracağından ve işhattının kontrolünü zorlaştıracağından dolayı, kat sayısı sınırlı kalmak zorundadır. Yapılan istatistiklere göre pratikte k kat sayısının optimal değeri 2 ile 15 arasında bir değerdir (Hwang , 1993).

2. 3. 1. KOMUT İŞHATTI YAPISI

Bir komutun işhattında işletilmesi, işhattında çeşitli ara işlemlere bölünmüştür. Bu ara işlemler şu şekildedir :

- Alma (Fetch)
- Kod Çözme (Decode)
- Yayın (Issue)
- İşletme (Execute)
- Yazma (Store)

Alma katında komutlar önbellekten alınırlar ve kod çözme katına iletilirler. Kod çözme katında komutların kodları çözülür ve komutların kullanacakları kaynaklar belirlenir. Saklayıcılar, fonksiyonel birimler komutların kullanacakları kaynaklardır. Yayın katında ise kaynaklar rezerve edilirler ve komutların kullanacağı operandlar saklayıcılardan okunur.. İşletme katında komutlar bir veya birden çok çevrim süresinde işletilirler. Son kat olan yazma katında ise işletme katında elde edilen sonuçlar saklayıcılara yazılırlar. Bellekle ilgili load ve store işlemleri load / store fonksiyonel biriminde yapıldığı kabul edilir. 5 katlı bir komut işhattı Şekil 2.4' de gösterilmiştir.



Şekil 2. 4 5 Katlı Komut İşhattı

2 . 3. 2. İŞHATTININ GETİRDİĞİ DARBOĞAZLAR

İşhattı yapısı kullanıldığı sistemde büyük performans artışına sebeb olmaktadır. Buna rağmen bazı durumlarda işhattı çalışma mekanizması işhattından beklenen verimi vermemektedir. Böyle durumlarda işhattı yapısını kullanan işlemcilerdeki performans, kullanmayan işlemcilere göre düşük olabilmektedir. Bu da işhattı tekniğinin getirdiği avantajları ortadan kaldırmaktadır. Bahsedilen bu durumlara işhattının ortaya çıkarttığı darboğazlar denmektedir. Bu darboğazlar veri bağımlılığı (data dependency), dallanma (branching), kesmeler (interrupts) durumlarıdır. Bu darboğazların giderilebilmesi için çeşitli çözümler önerilmiştir. Bu önerilen yöntemler daha ileriki bölümlerde ayrıntılı olarak işlenecektir.

2. 3. 2. 1. VERİ BAĞIMLILIĞI

İşhattında bulunan farklı komutlar ortak saklayıcıları veya ortak bellek gözlerini kullanabilirler. Eğer programdaki tüm komutlar programdaki sıraya göre işletiliyorsa herhangi bir problem doğmaz. Ancak komutlar programdaki sıraya göre işletilmiyorsa, yani program sırasına göre sonra gelen bir komut kendinden önce gelen komutlardan önce işletilmişse işlem sonunda yanlış sonuçlar elde edilecektir. Komutların kullandıkları ortak saklayıcılar ve ortak bellek gözlerine yapılan işlemlerin programdaki sıraya

göre düzenlenmesi gerekmektedir. Farklı komutlar arasında meydana gelen bu probleme veri bağımlılığı problemi denmektedir. Pipeline' da veri bağımlılığı üç şekilde olabilir :

- Yazmadan sonra Okuma
- Yazmadan sonra Yazma
- Okumadan sonra Yazma

Bir programda b komutunun a komutundan sonra geldiğini ve her iki komutun da aynı saklayıcı üzerinden işlem yaptığını varsayalım. Eğer a komutu o saklayıcıya yazma ve b komutu ise aynı saklayıcıdan okuma yapıyorsa, Yazmadan sonra Okuma tehlikesi ortaya çıkar. Bu durumda b komutunun okuma işlemini, a komutunun yazma işleminden önce yapmasını engellemek gerekir. Eğer a komutu ve b komutu aynı saklayıcıya yazma yapıyorsa, Yazmadan sonra Yazma tehlikesi olur. b komutunun a komutundan önce yazma yapmasının engellenmesi gerekir. Aksi takdirde, b komutundan sonra gelen komutlar saklayıcıdan yanlış değer okuyacaktır. Son olarak Okumadan sonra Yazma tehlikesini inceleyelim. Eğer b komutu, a komutunun yazma işlemini tamamlanmasını beklemeden okuma yaparsa, Okumadan sonra Yazma tehlikesi ortaya çıkar. Bundan dolayı, b komutunun a komutu yazma yapana kadar bekletilmesi gerekir.

Tehlike durumlarını tespit etmek ve ortadan kaldırmak için iki yöntem vardır. Bunlar statik komut düzenleme (Static Instruction Scheduling) ve dinamik komut düzenleme (Dynamic Instruction Scheduling) yöntemleridir. Her iki yöntemin de amacı, komutların işhattına girmeden önce aralarındaki veri bağımlılıklarını tespit etmek ve çözümlemek, kullandıkları kaynak çatışmalarını önerdikleri çözüm mekanizmasına göre ortadan kaldırmaktadır. Kaynak çatışması başka bir komut tarafından kullanılan kaynağı, işhattına gönderilmek üzere olan bir komutun kullanmak istemesi nedeniyle ortaya çıkar.

Statik komut düzenleme yönteminde komutlar arasındaki veri bağımlılığı ve kaynak çatışmaları yazılımla tespit edilir ve çözümlenir. Bu yazılım genelde derleyicidir. Derleyici programdaki komutları tarayarak birbirleri arasında veri bağımlılığı olan komutları tespit eder. Daha sonra tespit edilen bu komutların, veri bağımlılığını ortadan kaldıracak şekilde aralarını açar. Aralarını açmadan kasıt birbirlerine yakın ve veri bağımlılığı bulunmayan komutlar arasına, bu komutlarla veri bağımlılığı bulunmayan diğer komutların yerleştirilmesidir. Örnek olarak aşağıdaki kod parçasını alalım :

1 ADD R1,R2	1	Çevrim Süresi
2 LD R5,[R6]	2	"
3 LD R7,[R8]	2	"
4 DIV R7,R5	3	"

Bu kodda görüldüğü üzere DIV komutu R7 ve R5 saklayıcılarını kullanmaktadır fakat her iki saklayıcıda DIV komutundan önce gelen LD R5,[R8] ve LD R7,[R8] komutları tarafından kullanılmaktadır. Bu nedenle DIV komutuyla iki LD komutu arasında veri bağımlılığı problemi bulunmaktadır. DIV

komutunun pipeline' a gönderilebilmesi için her iki LD komutun da tamamlanması gerekmektedir. Her iki LD komutunun tamamlanması pipeline' da üç pipeline çevrim süresi alır. 3 numaralı LD komutu pipeline'a gönderildikten sonra sonraki pipeline çevrim süresinde, DIV komutu pipeline' a gönderilmeyecektir; çünkü 3 numaralı LD komutu henüz tamamlanmamıştır. Bu nedenle 1 numaralı ADD komutu her iki LD komutundan da bağımsız olduğundan DIV komutuyla LD komutları arasına yerleştirilerek bir pipeline çevrim süresi kazanılmış olur. Derleyicinin oluşturacağı yeni kod şöyle olacaktır :

```
1 LD   R5,[R6]
2 LD   R7,[R8]
3 ADD  R1,R2
4 DIV  R7,R5
```

Dinamik komut düzenleme yönteminde ise veri bağımlılığı ve kaynak çatışmaları derleyici tabanlı bir çözüm yerine, doğrudan işlemci içerisinde bulunan özel donanım çözümleri yapılmaktadır. Veri bağımlılığı programın çalışma zamanında (run-time) tespit edilip çözümlenmektedir. Önerilen donanım çözümleri arasında Tomasulo Algoritması, Skorbord (Scoreboarding) ve Dağıtım Yığını (Dispatch Stack) mekanizmaları ayrıntılı olarak incelenecektir.

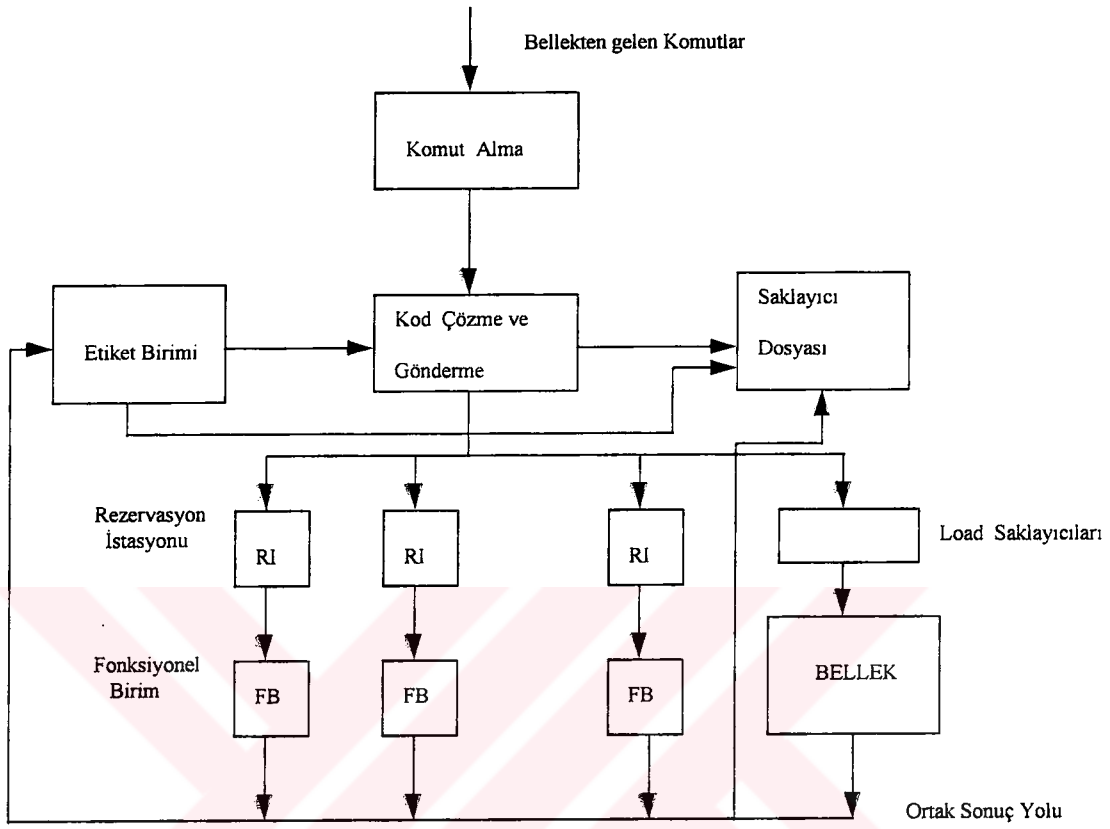
1 - TOMASULO ALGORİTMASI

Komutlar kodları çözülüp işhattında alma katına geldiğinde, veri bağımlılıklarına bakılır. Eğer komutun kullanacağı operandlar başka komutlar tarafından kullanılıyorsa, komut her fonksiyonel birime dağıtılmış olan rezervasyon istasyonuna (reservation station) gönderilirler. Rezervasyon istasyonları komutların bekletildiği tampon alan olarak tasarlanmış donanım yapılarıdır. Komutlar rezervasyon istasyonlarında veri bağımlılıkları çözülene kadar bekletilirler. Tüm operandları hazır olan ve rezervasyon istasyonunda bekleyen bir komut hemen ilgili fonksiyonel birime gönderilir. Tomasulo Algoritmasının model mimarisi Şekil 2.5' de gösterilmiştir (Sohi, 1990).

Bu mimaride etiket birimi, fonksiyonel birimlere gönderilen her komut için bir etiket değeri atar ve bunu kendi içinde saklayıcı numarasıyla birlikte saklar. Komutlar rezervasyon istasyonuna, kullandıkları saklayıcıların etiketleriyle beraber gönderilirler.

Komutlar tamamlandığında komutun sonucuyla birlikte etiketi de ortak sonuç yolunda bulunur ve bu etiket yardımıyla etiket birimi güncellenir. Tüm rezervasyon istasyonlarında kendi operandlarını bekleyen komutlar, tamamlanan komutun etiketine göre operandların hazır olup olmadıklarını anlarlar. Hazır durumda olan komutlar ise fonksiyonel birimlere gönderilirler.

Tomasulo Algoritmasında bellek de özel bir fonksiyonel birim gibi düşünülmektedir. Tomasulo Algoritması IBM 360 / 91 makinasının kayan-nokta (floating-point) biriminde uygulanmış bir tekniktir.



Şekil 2. 5 Tomasulo Algoritması Mimari Modeli

2 - SKORBORD

Skorbord ilk defa CDC6600 makinasında kullanılmıştır. CDC6600 makinasında çoklu fonksiyonel birimler bulunmaktadır. Her fonksiyonel birim için komut tamponları bulunmaktadır. Komutlar fonksiyonları birimlere kaynak saklayıcıları hazır olmazsa bile gönderilir ve komut tamponlarında bekletilir. Kaynak saklayıcıları hazır olduğunda komut işletilmeye başlanır. Komutların bekletilmesini ve işletilmeye hazır hale getirilmesini skorbord denilen mekanizma yapar. Skorbord, fonksiyonel birimlerde bekleyen komutlar tarafından kullanılan veya kullanılacak olan saklayıcıların durumlarını izler. Eğer bir komutun ihtiyaç duyduğu saklayıcılar hazır hale gelirse, skorbord komutun işletilmesi için işaret gönderir. Aynı şekilde, bir fonksiyonel birim bir komutu tamamladığında anında skorborda güncelleme yapabilmesi için işaret verir.

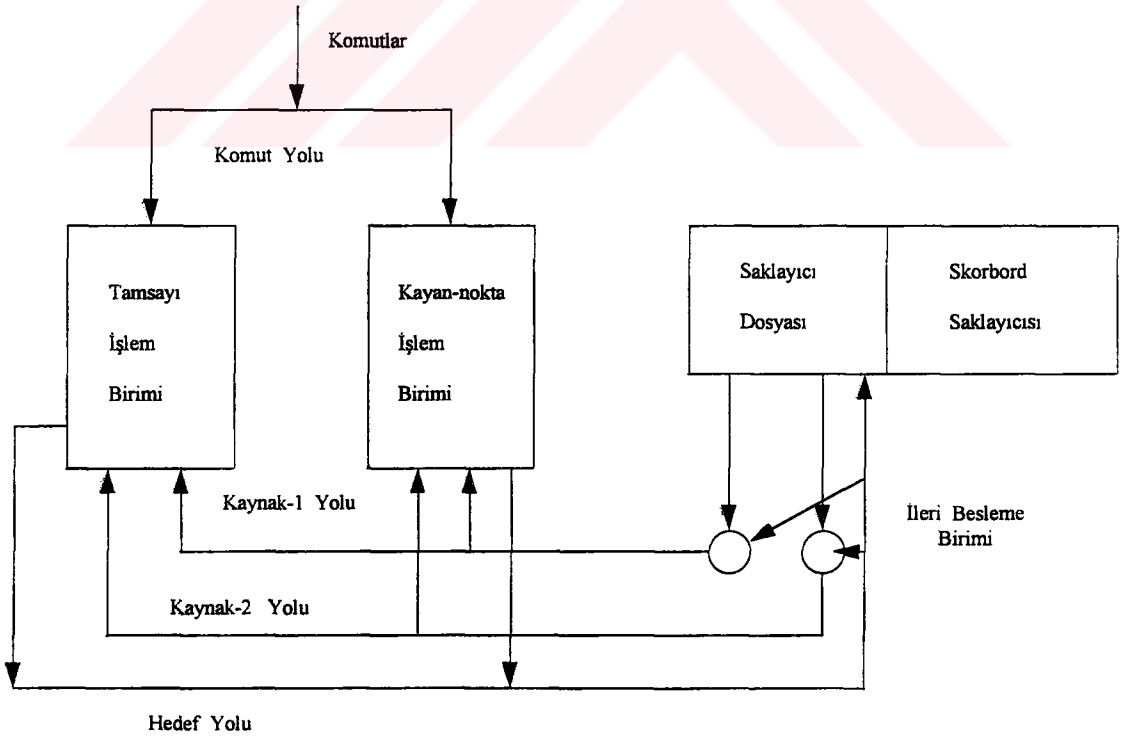
Skorbord mekanizması Motorola' nın 88000 RISC (Melear, 1989) işlemcisinde de kullanılmaktadır. 88000 işlemcisinin yapısı Şekil 2.6' da gösterilmektedir. Skorbord saklayıcısı 32 bitlik

bir saklayıcıdır. Saklayıcı dosyasında her saklayıcı için bir bitlik bir alan mevcuttur. Bir komut fonksiyonel birime gönderildiğinde kullandığı hedef saklayıcı için skorbordda ilgili bit set edilir. Komut tamamlanıp komutun sonucu yazıldığında, skorborddaki ilgili bit resetlenir. Skorbordda bir saklayıcıya ilişkin set edildiğinde, aynı saklayıcıyı kullanmak isteyen ve sonra gelen bir komut o saklayıcıyı kullanamayacaktır. Bu komut skorborddaki ilgili bit resetleninceye kadar komut işletime sokulmaz.

3 - DAĞITIM YIĞINI

Dağıtım yığını (Acosta et al, 1986) , mekanizmasının en önemli özelliği her çevrim süresi içerisinde bir veya daha çok komutu fonksiyonel birimlere gönderilebilmesidir. Tomasula Algoritması ve Skorbord mekanizmaları ise sadece bir komut gönderebilmektedir. Dağıtım yığını mekanizmasının bir diğer özelliği ise komutların program sırasına göre gönderilmesinin zorunlu olmamasıdır. Dağıtım yığını model mimarisi Şekil 2.7' de gösterilmiştir.

Mimaride komutlar bellekten çekilip kodları çözüldükten sonra dağıtım yığınına alınırlar. Dağıtım yığınındaki komutların kullandıkları tüm saklayıcılar hakkında bilgiler tutulur. Bu bilgiler arasında bir saklayıcının programdaki sıraya göre daha önce gelen ve tamamlanmamış komutlar tarafından kaç kez hedef saklayıcısı ve kaynak saklayıcısı olarak kullanıldığı bilgileri vardır. Bunun yanında komutun kullandığı kaynak saklayıcıların önceki komutlar tarafından kaç kez hedef saklayıcısı olarak



Şekil 2. 6 MOTOROLA 88000 RISC İşlemcisi

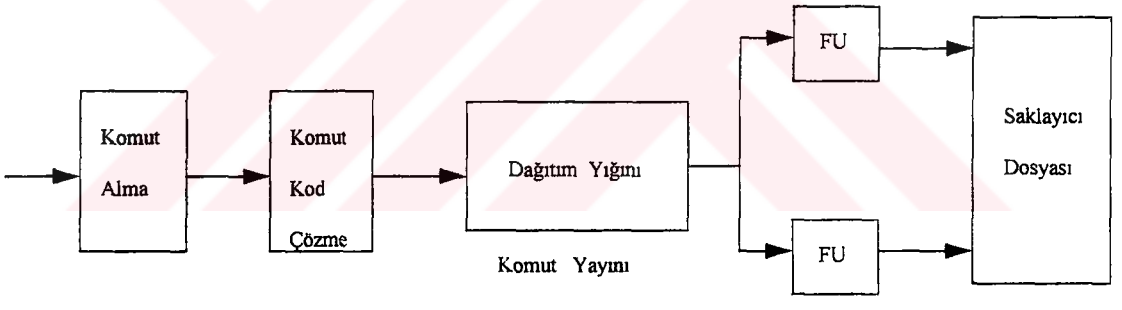
kullanıldığını belirten sayılarla, aynı komutun kullandığı hedef saklayıcısının önceki komutlar tarafından kaç kez kaynak ve hedef saklayıcısı olarak kullanıldığını belirten sayılar toplanarak o komut için tüm veri bağımlılıklarını ifade eden toplam bir sayı elde edilir.

Her çevrim süresinde dağıtım yığını taranarak , bu toplam değerin 0 olduğu komutlar bulunur. Eğer bir komut için bu değer 0 olmuşsa o komutla ilgili fonksiyonel birime gönderilir. Her çevrim süresinde aynı anda fonksiyonel birimlere gönderilebilecek bir veya daha çok komut bulunabilir. Bazı çevrim sürelerinde fonksiyonel birimlere gönderilebilecek komut bulunamayabilir.

Dinamik komut düzenlemede önerilmiş olan yöntemler donanım tabanlı yöntemlerdir. Donanım tabanlı yöntemler programın çalışma anında veri bağımlılıklarını tespit ettiği için daha hızlıdır. Statik komut düzenlemede ise mekanizma, derleyici tabanlı olduğundan dinamik komut düzenlemeye göre daha yavaştır ve kullanılan makinaya bağlıdır. Buna karşın, dinamik komut düzenleme daha karmaşık donanım kullandığından sisteme ek maliyet getirecektir.

2 . 3. 2. 2. DALLANMA

İşhattı yapısında en çok problem yaratan darboğazlardan biri de dallanmadır.



Şekil 2. 7 Dağıtım Yığını Organizasyonu

Dallanma komutları işhattına girdiklerinde önlerinde iki olasılık vardır. Dallanma komutunun sonucuyla ya dallanma komutundan sonra gelecek olan komut işhattına gelecektir ya da dallanma komutunun işaret edeceği yeni adresten itibaren komutlar işhattına gönderilecektir. Dallanma komutunun sonucu çalışma anında belli olacağından komutun ne yönde dallanacağı kesin olarak bilinemez. Ancak istatistiklere göre tahmin yürütülebilir. Eğer yapılan tahmin yanlış çıkarsa, yanlışlıkla işhattına gönderilen tüm komutların işhattından boşaltılması gerekecektir. Bu durumda işhattı kullanımından sağlanması gereken performans düşecektir.

İki tip dallanma komutu vardır. Şartsız ve şartlı dallanma komutları bulunmaktadır. Şartsız dallanma komutlarında dallanma, verilen adrese hiç bir şarta bağlı olmadan yapılır. İşlemci şartsız

dallanma komutuna herhangi bir komut olarak davranabilir. Şartlı dallanma komutunda ise işlemci hangi yönde dallanacağına karar vermek zorundadır. Elde edilecek olan şarta göre dallanacak adres belirlenecektir.

26 program üzerinde yapılan istatistiklere göre bir programdaki komutların % 10 ila % 30' u dallanma komutu olarak belirlenmiştir. Bu dallanma komutlarının % 60' ında dallanma gerçekleşmiştir. Sonuç olarak işhattının performansı % 34' e düşmüştür (Lilja , 1988).

Dallanma komutlarının bir diğer olumsuz etkisi önbellek üzerine olmaktadır. Özellikle ileri doğru yapılan dallanmalarda dallanmanın gideceği hedef adresi önbellekte bulunmayabilir. Bu durumda önbellek ıskası (cache miss) olur. Bu durumda önbelleğin ana bellekten yeni komutları alması için doldurulması gerekecektir. Geriye doğru dallanmada ise genellikle dallanma hedef adresi önbellek içerisinde bulunacağından herhangi bir problem ortaya çıkmayacaktır. Geriye doğru dallanmalar özellikle döngü komutları sırasında meydana gelmektedir.

Dallanma komutlarından kaynaklanan performans düşüşünü bir ölçüde azaltmak için çeşitli yöntemler önerilmiştir. Bu yöntemler; döngü tamponları (loop buffers), gecikmeli dallanma (delayed branch), dallanma hedef tamponu (branch target buffer) yöntemleridir.

1 - DÖNGÜ TAMPONLARI

Döngü tamponu küçük boyutlu ve hızlı bir tampon bellektir ve işhattının komut alma katında bulunur. Döngü tamponuna bellekten getirilen komutlar yerleştirilir. Eğer çoklu döngü tamponu varsa, bu tamponlar da komutlarla doldurulur. Tamponlarda tutulan komutlar program sırasına göre birbirlerini takip eden komutlardır.

Dallanma komutu olduğunda dallanma hedef adresi döngü tamponunda ise çok hızlı bir şekilde bu adrese ulaşmak mümkündür. Eğer dallanma bir döngü tarafından meydana gelmişse, döngünün tüm komutlarına bu tampon bellekten erişilir. Döngü tamponunun kullanıldığı işlemciler arasında CDC Star-100, Cray-I işlemcileri bulunmaktadır.

2 - GECİKMELİ DALLANMA

Gecikmeli dallanma kavramı, dallanma komutunun yorumlanıp anlaşılmasıyla hedef adrese gitmesi arasındaki boş kalan sürede, işhattına yararlı komutlar gönderme esasına dayanır. Yararlı komutlar ise veri bağımlılığı olmayan veya ortadan kaldırılmış olan komutlardır. Bu komutların bulunup dallanma komutunun hemen ardına yerleştirilmesi derleyici tarafından yapılır. Derleyici, komutlar arasındaki veri bağımlılığını belirleyip dallanma komutlarından sonra bu komutları yerleştirir. Dallanma komutu işhattına gönderildikten sonra, bu komutlar işhattına gönderilir. Böylece işhattının boşta kalacağı sürelerde yararlı komutlar işletilecektir. Derleyici dallanmanın arkasına yerleştirecek komut bulamadığı

durumlarda NOP (No Operation) komutunu yerleştirir. Gecikmeli dallanma yöntemi MIPS R4000 ve MOTOROLA MC88110 işlemcilerinde kullanılmıştır.

3 - DALLANMA HEDEF TAMPONU

Dallanma hedef tamponu küçük bir asosiyatif önbellektir (Lee et al, 1984), (Lilja, 1988). Dallanma hedef tamponu içerisinde daha önceden işletilmiş olan dallanma komutunun adresi, dallanmanın yapılıp yapılmadığı hakkında öngörü (prediction) bilgisi ve dallanmanın hedef adresi tutulur. Komutlar alma katına geldiklerinde , alma katı komutun adresiyle tampondaki adresleri asosiyatif olarak karşılaştırılır. Eğer adresler aynıysa tamponun ilgili alanındaki öngörü bilgisine bakılarak dallanmanın ne yönde yapılacağına karar verilir. Dallanma komutu gerçekten işletildikden sonra dallanma hedef tamponundaki öngörü bilgisi ve hedef adres alanları güncellenir.

2.3.2.3. KESME ve AYKIRI DURUMLAR

Kesmeler, normal program akışını değiştiren olaylardır. Bir prosesin işletilmesi esnasında kesme oluşursa, prosesin işletilmesi durdurulur ve kesme servis programına gidilir. Kesme servis programına gitmeden önce işlemci içerisindeki durum bilgisi (state information) saklanmalıdır. Durum bilgisi içinde program sayıcı, saklayıcılar ve bellek vardır. Kesme servis programından dönüşte, daha önce saklanmış durum bilgisi tekrar işlemci içerisine yüklenir ve kesintiye uğramış prosese, kalınan noktadan devam edilir.

Kesmeler ; program kesmeleri ve dış kesmeler olarak iki ayrı sınıfta incelenebilir. Program kesmeleri aykırı durumlar (exception conditions) olarak adlandırılır. Aykırı durumlar işlemci içerisinde oluşan kesmelerdir. Örnek olarak; taşma (overflow), sayfa hatası (page fault) , sıfıra bölme (divide by zero) , hatalı komut işletme (illegal opcode execution) aykırı durumlardır. Dış kesmeler ise komutlardan kaynaklanan kesmeler değildirler. Tamamen dış olaylardan kaynaklanan, genellikle çalışan prosesle ilgisi olmayan kesmelerdir. Giriş-Çıkış ve zamanlayıcı kesmeleri dış kesmelere örnek verilebilir.

İşhattı kullanan mimarilerde kesme oluştuğunda saklanacak olan proses durumu ardışıl mimarilerdeki proses durumundan farklıdır. İşhattında program sırasına göre sonra gelen komutlar kendilerinden önce gelen komutlardan önce tamamlanabilirler. Böyle bir durumda kesme oluştuğunda kesmenin olduğu komutun program sayıcı adresini saklamanın yararı olmayacaktır. Çünkü, kesmenin oluştuğu komuttan sonraki komutlar işletilmiş olabilir. Eğer bu durum gözönüne alınmayıp program sayıcı saklanır ve kesme servis programından dönüşte, kalınan noktadan proses işleilmeye devam edilirse daha önce işletilmiş komutlar bir kez daha işletilmiş olur. Bu nedenle yanlış sonuçlar ortaya çıkacağından, proses doğru çalışmadan tamamlanmış olur.

İşhattı kullanan bir işlemcide oluşan kesmeden sonra saklanması gereken proses durumunun ardışıl mimariyi kullanan işlemcideki proses durumuyla aynı olmasını sağlamak gerekir. Böyle bir durumun sağlanması durumunda kesme **precise** adını alır (Smith et al, 1988). Kesmenin precise olması için aşağıdaki üç şartı sağlaması gerekir :

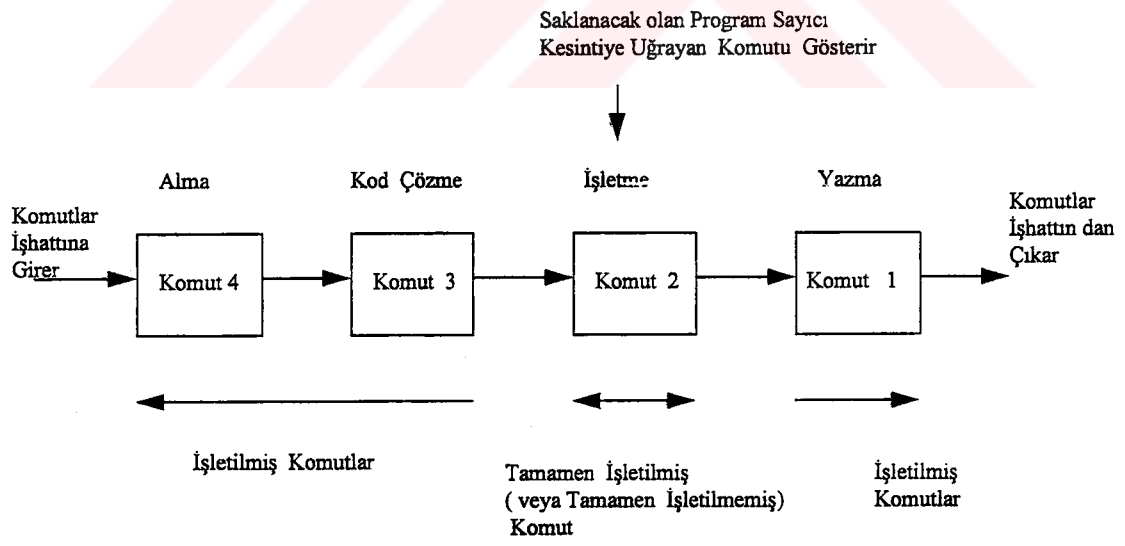
1. Saklanan program sayıcısının gösterdiği komuttan önce gönderilmiş tüm komutlar tamamlanmış ve proses durumunu doğru olarak değiştirmiş olmalıdır.
2. Saklanan program sayıcısının gösterdiği komuttan sonra gönderilmiş olan tüm komutlar tamamlanmamış ve proses durumunu değiştirmemiş olmalıdır.
3. Kesintiye uğramış olan komut ya hiç işletilmemiş olmalı ya da tamamlanmış olmalıdır.

(Walker et al, 1995)

Bu üç şartın işhattı yapısında gösterilimi Şekil 2.8' de gösterilmiştir. Eğer saklanmış olan proses durumu bu üç şartı sağlamıyorsa, kesme **imprecise** kesme olarak adlandırılır.

Precise kesmeleri kotarmak imprecise kesmeleri kotarmaktan daha kolay olmaktadır. Precise kesmelerde yapılması gereken, işlemci içindeki proses durumunun yukarıda değinilen üç şartı sağlamaktır. Fakat, imprecise kesmelerde saklanması gerekli olan proses durumu çok olduğundan böyle bir çözümün gerçekleştirilmesi oldukça büyük miktarda ek donanım gerektirmektedir. Ayrıca, imprecise kesmeler precise kesmelerin sağlaması gereken şartlara bağlı olmayacağından daha yüksek performansa sahiptirler.

Kesme kotarma işlemi, işlemcilerde iki şekilde yapılabilir. Bunlar; işlemcinin durumunu saklama ve işlemci durumunu ardışıl mimari modeline uygun olarak düzenleme şeklindedir.



Şekil 2. 8 Precise Kesme Şartlarının İşhattın daki Gösterilimi

İşlemci durumunu saklama için çeşitli donanım çözüm yöntemleri bulunmaktadır. Bu donanım yöntemleri arasında yığın kullanımı, gölge saklayıcıları (shadow registers) , gölge yığınları (shadow stacks) , kontrolnoktası donanımı (checkpointing hardware) bulunmaktadır.

1 - YIĞIN YÖNTEMİ

Yığın, proses durumunu saklamak için düşünülmüş en basit yöntemdir. Kesme geldiğinde program sayıcı ve diğer bilgiler belleğe atılırlar. Belleğe erişim çok yavaş olduğundan yığın yöntemi, kesmelerin kotarılması için çok basit ve oldukça yavaş bir yöntemdir. Intel 8086 işlemcisi kesme kotarma için yığın yöntemini kullanmışlardır.

2 - GÖLGE SAKLAYICILARI YÖNTEMİ

Bu yöntemde işlemci içindeki bazı saklayıcılar için yedek saklayıcılar bulunmaktadır. Bu yedek saklayıcılara gölge saklayıcıları denmektedir. İşlemci içinde kesme olduğunda bazı saklayıcıların içerikleri gölge saklayıcılarda tutulur. Kesme servis programı tamamlandıktan sonra, gölge saklayıcılarda tutulan bilgiler tekrar ana saklayıcılara yüklenirler.

Gölge saklayıcıları yöntemi MOTOROLA 88100 (Diefendorff et al, 1989) işlemcisinde kullanılmıştır. 88100 işlemcisinde işlemci durum saklayıcısı (processor status register) , skorbord saklayıcısı, komut takibinde kullanılan bazı işaretçiler (pointers) ve bellek erişiminde kullanılan saklayıcılar için gölge saklayıcılar bulunmaktadır.

Herhangi bir kesme oluştuğunda, işlemci içindeki işlemci durum saklayıcısının bir biti set edilir. Bu bitin set edilmesiyle, komut alma katı komut alımını durdurur. Gölge saklayıcılara ana saklayıcıların içerikleri saklanır ve kesme servis programına gidilir. Eğer içiçe (nested) kesmeler olursa, gölge saklayıcıların içerikleri ana belleğe aktarılmak zorunluluğu ortaya çıkar. Kesme servis programından dönüşte, işlemci durum saklayıcısındaki ilgili bit resetlenir ve gölge saklayıcılar, içeriklerini ana saklayıcılara yüklerler.

3 - GÖLGE YIĞINLAR YÖNTEMİ

Gölge yığınlar yöntemi, gölge saklayıcılar yöntemine çok benzemektedir. Gölge saklayıcılar yöntemindeki gölge saklayıcısının yerine gölge yığın yerleştirilmiştir. Böylece içiçe kesmeler olduğunda, saklayıcıların içerikleri bellek yerine gölge yığınlara atılmaktadırlar.

Gölge yığınlar yöntemi IBM RISC System / 6000 (Grohoski, 1990) işlemcisinde kullanılmıştır. İşlemcideki tamsayı biriminde load, store ve tuzak (trap) komutları kesmeye yolaçan komutlar olarak değerlendirilmektedir. Kesmeye yolaçan komutlardan biri fonksiyonel birimlere

gönderileceği zaman, bu komutun adresiyle birlikte komutun yaptığı değişiklikleri düzeltmek için gerekli olan bazı durum saklayıcıları program sayıcı yığınına (program counter stack) atılırlar. Eğer bu komut kesme oluşturursa, program sayıcı yığınında saklanmış olan bu bilgiler geri yüklenirler. Eğer komutta kesme olmamışsa program sayıcı yığınında komutla ilgili bilgiler iptal edilirler.

4 - KONTROLNOKTASI DONANIMI

Kontrolnoktası donanımı yönteminde (Hwu et al, 1987) belli zaman periyotlarında, proses durumu örneklenerek bellekte saklanırlar. Program içerisinde örnekleme yapılan noktalara kontrolnoktaları denir. Programın herhangi bir noktasında kesme olduğunda kesme gelmeden önceki ilk kontrolnoktasındaki proses durumu bellekten işlemciye aktarılır. Kesme servis programından dönüşte, işleme başlanacak nokta, yapılan bu nokta olacaktır. Proses durumları ya ana belleğe ya da diske yapılabilir. Bu yöntem . precise kesmeyi sağlamak için belleğe çok fazla durum saklamak zorunda kalacaktır. Belleğe erişim hem yavaştır hem de saklanan durumlar bellekte çok büyük yer işgal edecektir. Bu nedenle kontrolnoktası donanımı yöntemi herhangi bir ticari işlemcide henüz kullanılmamıştır.

İşlemci durumunu ardışıl mimari modeline uygun olarak düzenleyen yöntemler ise şunlardır :

- Sıralı Komut Tamamlama (In-order Instruction Completion)
- Yeniden Düzenleme Tamponu (Reorder Buffer)
- Tarih Dosyası (History File)
- Gelecek Dosyası (Future File)

(Smith et al, 1988)

Tüm bu yöntemler precise kesme yaklaşımını temel alırlar.

1 - SIRALI KOMUT TAMAMLAMA

Komutlar, eğer kendilerinden önce gönderilmiş tüm komutlar kesme olmadan tamamlanmışlarsa, proses durmunu değiştirebilirler. Mimarideki tüm fonksiyonel birimlerdeki işhattı gecikmeleri sabittir. Komutların program sırasına göre tamamlanmasını sağlamak için sonuç kaydırmalı saklayıcı (result shift register) yapısını kullanır. Sonuç kaydırmalı saklayıcının uzunluğu, mimarideki en uzun fonksiyonel birimdeki işhattının uzunluğu kadardır.

Bir komut fonksiyonel birime gönderileceğinde komutun kaç çevrim süresi sonunda işhattından çıkacağı bulunur. Komut j çevrim süresinde biterse , sonuç kaydırmalı saklayıcının j. satırına komutla ilgili bilgiler yazılır ve bu komut j. satırdan önceki tüm satırları rezerve edilir. Yani bu komut, sonra gelecek olan ve daha kısa bir komutun sonuç kaydırmalı saklayıcıya yerleştirilmesine izin vermez. Böylece komutların program sırasına göre tamamlanması sağlanmış olur. Her işhattı çevrim süresinde sonuç kaydırmalı saklayıcı bir satır yukarı doğru kaydırılır. Komut birinci satırdan da çıktıktan sonra

aykırı durum olup olmadığına bakılır. Eğer aykırı durum varsa, kontrol lojiği işhattında bulunan diğer komutları iptal eder. Sonuç kaydırmalı saklayıcı Şekil 2.9' da gösterilmiştir.

Kat	Fonk. Birim	Hedef Saklayıcısı	Geçerli Biti	Program Sayıcı
1				
n				

Hareket Yönü

Şekil 2. 9 Sonuç Kaydırmalı Saklayıcı

Kesmenin olduğu komutun program sayıcı adresi doğrudan sonuç kaydırmalı saklayıcıdan bulunarak saklanır. Store komutları, kendilerinden önce gelen tüm komutların aykırı durum olamadan tamamlanmasını beklemeden load / store fonksiyonel birimine gönderilir.

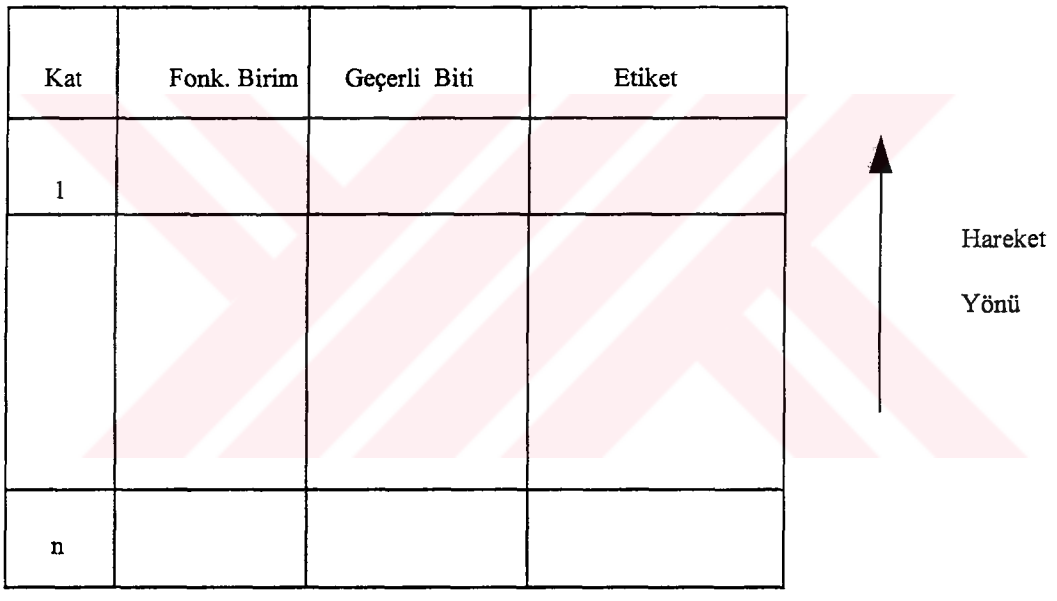
2 - YENİDEN DÜZENLEME TAMPONU

Sıralı komut tamamlama yönteminin olumsuz tarafı , daha hızlı ve veri bağımlılığı olmayan bazı komutların fonksiyonel birimlere gönderilmesine izin vermemesidir. Yeniden düzenleme tamponu yöntemi bu olumsuz yanı ortadan kaldırmak için önerilmiş olan bir yöntemdir. Bu yöntemde komutların program sırasına dışında (out-of-order) tamamlanmasına izin verilir. Komutlar proses durumunu değiştirmeden önce, yeniden düzenleme tamponu yardımıyla program sırasına göre düzenlenirler.

Yeniden düzenleme tamponu, baş ve son (head and tail) işaretçileri olan dairesel bir bellektir. Baş ve son işaretçileri arasında olan satırlar, geçerli satır olarak kabul edilirler. Bir komut gönderileceği zaman, son işaretçi tarafından işaret edilen satıra komutla ilgili bilgiler yazılır ve son işaretçi bir arttırılır. Tamponun son satırına gelindiğinde son işaretçi tamponun başını işaret eder. Yeniden düzenleme tampon yönteminde sonuç kaydırmalı saklayıcı yapısı da kullanılmaktadır. Sonuç kaydırmalı saklayıcının, sıralı komut tamamlama yöntemindeki sonuç kaydırmalı saklayıcıdan farkı, etiket alanının bulunmasıdır. Sonuç kaydırmalı saklayıcıdaki etiket bilgisi, yeniden düzenleme tamponunda, ilgili komutun satır numarasıdır.

Bir komut tamamlandığında hem sonuçlar hem de aykırı durum bilgisi yeniden düzenleme tamponuna gönderilir. Sonuç kaydırmalı saklayıcıda bulunan etiket yardımıyla, yeniden düzenleme tamponundaki ilgili satıra gidilir. Sonuçlar ve aykırı durum bilgisi yeniden düzenleme tamponuna yazılırlar , sonuçlar doğrudan saklayıcı dosyasına yazılmazlar.

Eğer baş işaretçinin işaret ettiği satıra sonuç yazılmışsa, bu satırla ilgili aykırı durum bilgisine bakılır. Komutta aykırı durum meydana gelmişse, komut gönderilmesi durdurulur ve kesme kotarma işlemi başlatılır. Eğer aykırı durum yoksa, yeniden sıralama tamponunda bulunan sonuç, saklayıcı dosyasına yazılır. Yeniden düzenleme tamponu ve ilgili sonuç kaydırmalı saklayıcı Şekil 2.10' da gösterilmektedir.



Sonuç Kaydırmalı Saklayıcı

	Satır No	Hedef Saklayıcı	Sonuç	Aykırı Durum	Geçerli Biti	Program Sayıcı
	1					
BAŞ →	2					
SON →	8					

Yeniden Düzenleme Tamponu

Şekil 2. 10 Yeniden Düzenleme Tampon Organizasyonu

Yeniden düzenleme tampon yönteminin benzeri Intel P6 işlemci içerisinde kullanılmıştır.

3 - TARİH DOSYASI

Tarih dosyası yönteminde, komutların sonuçları doğrudan saklayıcı dosyasına yazılır. Fakat ortaya çıkacak olan kesmelerde proses durumunun tutarlı (consistent) olabilmesi için komutların kullandıkları hedef saklayıcıları tarih tamponunda tutulur.

Tarih dosyası, yeniden düzenleme tampon yapısına benzemektedir. Yöntem, yeniden düzenleme yönteminde olduğu gibi sonuç kaydırmalı saklayıcı yapısını kullanmaktadır. Tarih dosyasında hedef saklayıcısının sonucu yerine eski değerini saklayan bir alan vardır. Bir komut fonksiyonel birime gönderildiğinde komutla ilgili bilgiler tarih tamponuna yazılır. Aynı anda komutun kullandığı hedef saklayıcısının eski değeri saklayıcı dosyasından okunarak tarih tamponuna yazılır. Komutlar tamamlandığında doğrudan saklayıcı dosyasına yazılırlar. Eğer baş işaretçinin gösterdiği satır geçerli ise ve aykırı durum yoksa, o satıra artık ihtiyaç olmayacaktır. Eğer aykırı durum varsa, diğer komutların fonksiyonel birimlere gönderimi durdurulur. Baş ve son işaretçileri arasındaki tüm hedef saklayıcıların eski değerleri , saklayıcı dosyasına yeniden yüklenir. Tarih tamponu ve ilgili sonuç kaydırmalı saklayıcı Şekil 2.11' de gösterilmiştir.

Kat	Fonk. Birim	Hedef Saklayıcısı	Geçerli Biti	Etiket
1				
n				

Hareket Yönü

Sonuç Kaydırmalı Saklayıcı

	Satır No	Hedef Saklayıcı	Eski Değer	Aykırı Durum	Geçerli Biti	Program Sayıcı
	1					
BAŞ →	2					
SON →	8					

Tarih Tamponu

Şekil 2. 11 Tarih Dosyası Organizasyonu

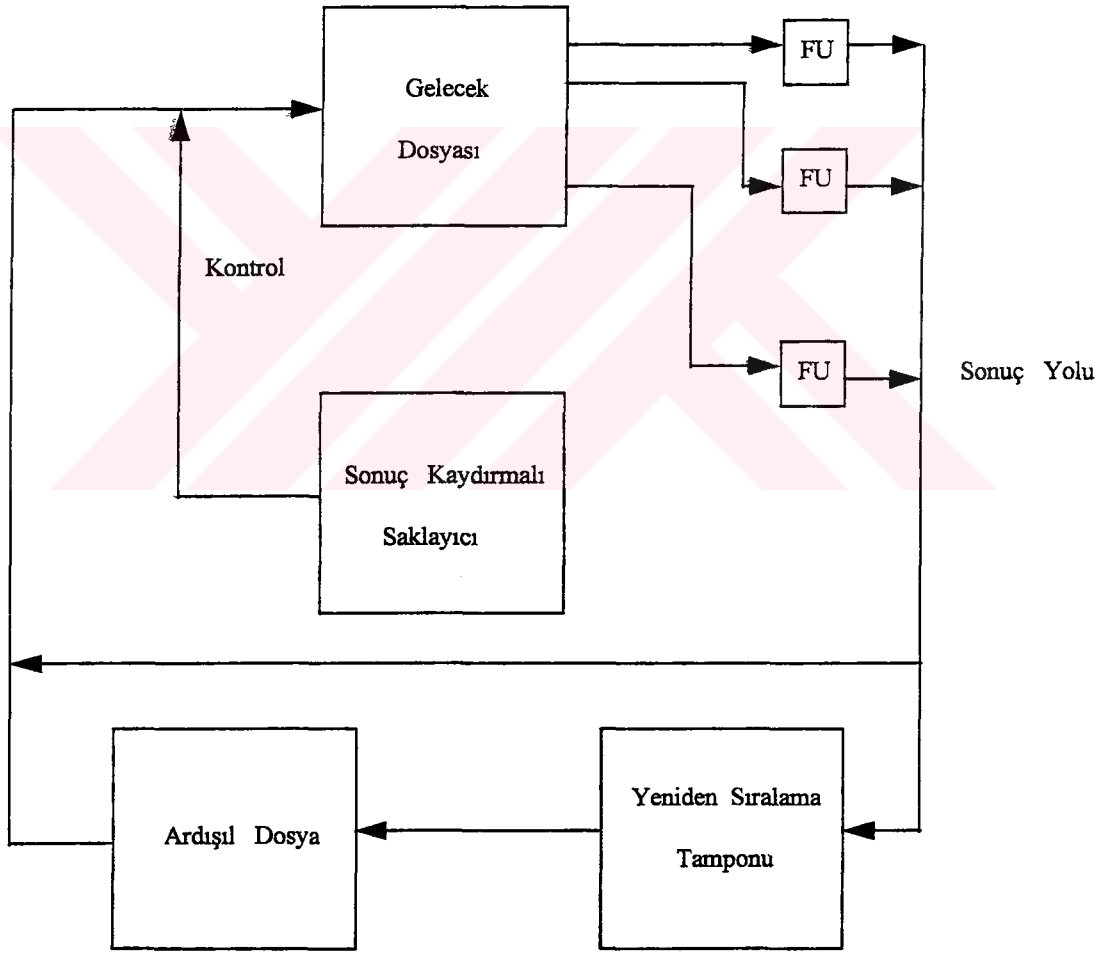
Tarih dosyası yöntemi MOTOROLA 88110, UltraSparc ve PowerPC603 işlemcilerinde kullanılmıştır.

4- GELECEK DOSYASI

Gelecek dosyası yöntemi tarih dosyası yöntemine benzemektedir. Gelecek dosyası yönteminde , tarih dosyasından farklı olarak iki ayrı saklayıcı dosyası kullanılmaktadır. Bu dosyalardan bir tanesi ardışıl

dosya diye adlandırılan ve ardışıl modele göre komutların yazma yapabildiği saklayıcı dosyasıdır. Diğer dosya ise gelecek dosyası diye adlandırılan ve komutların anında sonuçları yazdığı saklayıcı dosyasıdır.

Komutlar gönderilip tamamlandığında sonuçlar gelecek dosyasına yazılır. Gelecek dosyası yönteminde ayrıca komutların sonuçlarının tutulduğu bir yeniden sıralama tamponu bulunmaktadır. Yeniden sıralama tamponunun amacı, ardışıl dosyaya yapılacak olan değişiklikleri ardışıl yapıya uygun olarak yerine getirmektir. Yeniden sıralama tamponunda baş pointer' ın gösterdiği satırdaki komut geçerli ise ve aykırı durum yoksa, o satırdaki tamamlanmış komutun sonucu ardışıl dosyaya yazılır. Eğer bu komutta aykırı durum varsa, baş ve son işaretçileri arasındaki satırların hedef saklayıcı değerleri gelecek dosyasından ardışıl dosyaya yazılarak tutarlılık sağlanmış olur. Gelecek dosyası yönteminin yapısı Şekil 2.12' de gösterilmektedir.



Şekil 2. 12 Gelecek Dosyası Organizasyonu

Gelecek dosyası yöntemi herhangi bir ticari işlemcide henüz kullanılmamıştır. Bahsedilen tüm bu kesme kotarma yöntemleri precise kesme modeline göre geliştirilmiş yöntemlerdir. Yani kesme

oluştuğunda, kullanılan yöntem hangisi olursa olsun , kesintiye uğramış komutun program sayıcı adresinin bulunup saklanması gerekir. Değinilen bu yöntemler skaler işlemciler için makul çözüm yöntemleridir.

III. SÜPERSKALER MİMARİ YAPISI

Skaler işlemciler aynı anda sadece bir komut gönderebilen işlemcilerdir. Süperskaler işlemciler ise skaler işlemcilerin yapısında bulunan komutları aynı anda işleterek performansı artırırlar. Kullanılan komut yapıları, genel amaçlı mikroişlemcilerde bulunan komut yapılarıdır. Bu sebeble, süperskaler işlemciler skaler işlemcilerle kod uyumludur.

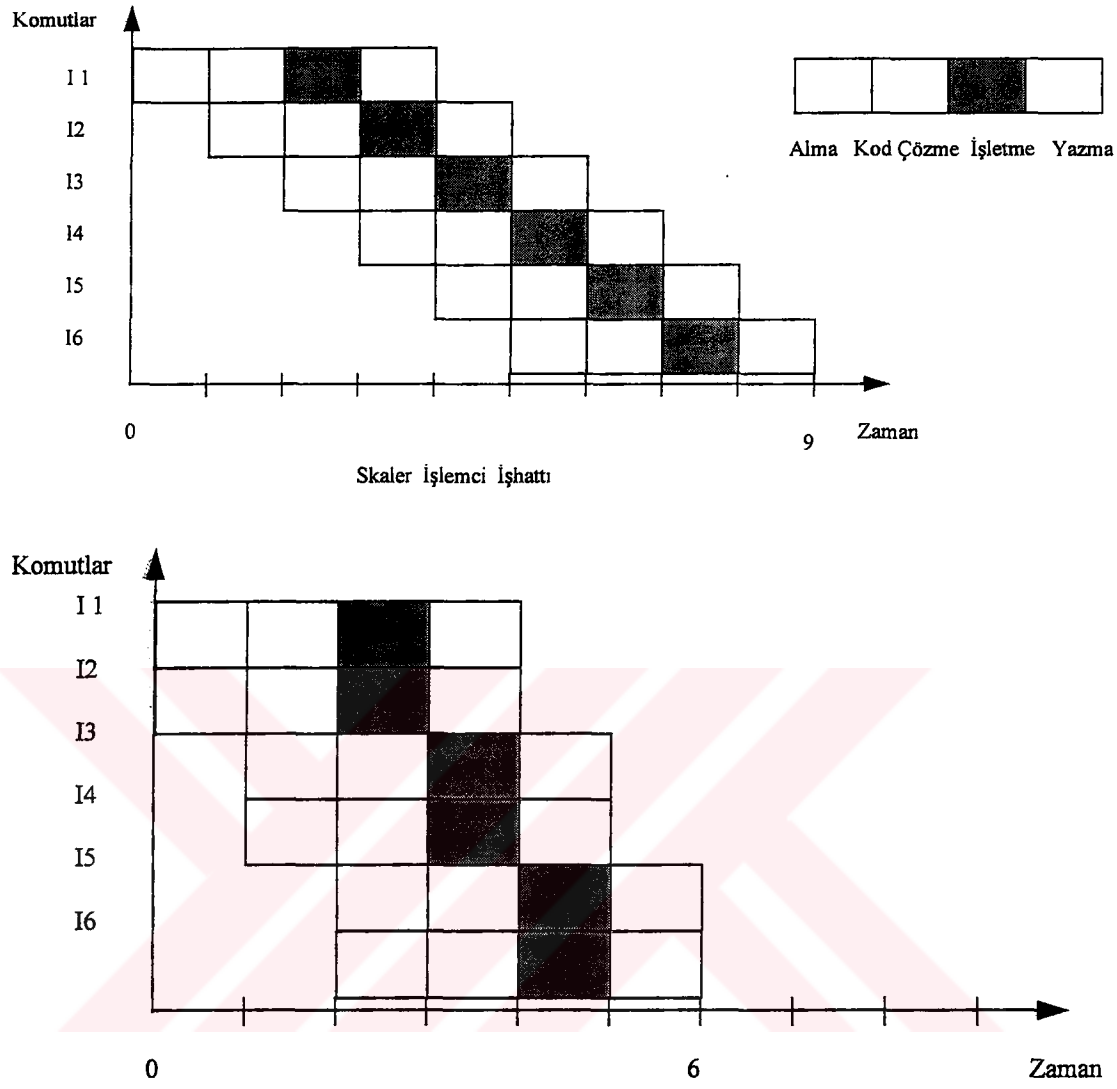
Süperskaler işlemcilerin aynı anda birçok komut gönderebilmesi için komutların birbirlerinden bağımsız olması gerekir. Bağımsız komutların bulunabilmesi tamamiyle işletilen programlara bağlıdır. Yapılan çalışmalara göre aynı anda üçten fazla komut gönderebilen bir süperskaler yapı çok fazla avantajlı olmamaktadır. Bu sebeble bir süperskaler işlemcinin gönderebileceği komut sayısı 2 ila 5 arasında sınırlıdır.

Süperskaler tekniği hem RISC hem de CISC mimarilerine uygulanabilir. Ancak RISC mimarisi süperskaler tekniğe çok daha uygun olduğu için, ticari süperskaler işlemciler RISC mimarisi özelliğini taşırlar.

3.1. SÜPERSKALER İŞLEMCİLERDE İŞHATTI YAPISI

n dereceli bir süperskaler işlemci her saat çevriminde n adet komut gönderebilir. Buna karşın bir RISC veya CISC skaler işlemcide n=1' dir. Bir süperskaler işlemcinin her an gönderebilecek n adet komut bulabilmesi mümkün olmayabilir. Böyle durumlarda işlemcideki işhatlarının beklemesi gerekecektir. Skaler bir işlemcideki işhattı yapısı ile süperskaler işlemci işhattı yapısı Şekil 3.1' de verilmektedir.

Süperskaler işlemcilerdeki işhattı yapısı skaler RISC işlemcisindeki işhattı yapısına benzemektedir. n komut gönderebilen bir süperskaler işlemcide n adet komut işhattı ve çoklu fonksiyonel birimler bulunmaktadır. Fonksiyonel birimler çoklu işhatları tarafından ortak kullanılabilmektedir. 2 komut gönderebilen bir süperskaler işlemcinin yapısı Şekil 3.2. ' de gösterilmiştir. Şekilde de görüldüğü üzere her işhattının kendi komut alma, kod çözme ve yazma katları bulunmaktadır. Komutlar tek bir komut önbelleğinden alınmaktadırlar.

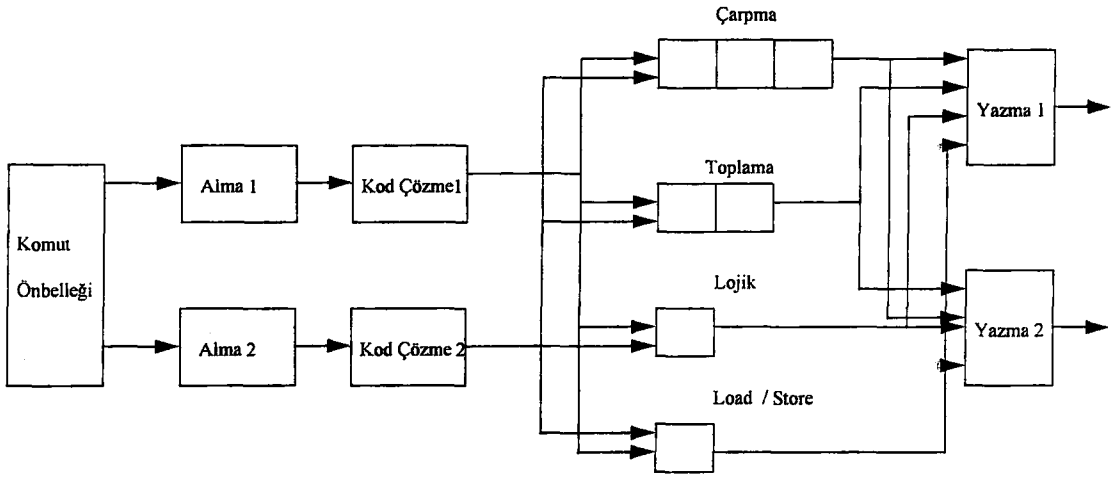


Şekil 3.1 Skaler ve Süperskaler İşhattı Yapıları

3.1.1. SÜPERSKALER İŞLEMCİLERDE VERİ BAĞIMLILIĞI ve DALLANMA

Süperskaler işlemcilerde olan veri bağımlılığı problemi Bölüm 2.3.2.1' de değinilen skaler işlemcilerdeki veri bağımlılığı problemiyle temelde aynıdır. Ancak süperskaler işlemcilerde aynı anda birbirinden bağımsız komutların bulunabilmesi için, komutlar arasında veri bağımlılığının olmaması

gerekir. Bu nedenle, veri bağımlılığı problemi süperskaler işlemcilerde skaler işlemcilere göre daha büyük performans kayıplarına yolaçmaktadır.



Şekil 3.2 2 Komut Yayınlayabilen bir Süperskaler İşlemci

Bir işlemin işletilmesi ne kadar uzun sürerse, işlemcinin aynı anda yayınlayabilecek komutları bulabilmesi ihtimali o kadar azalır. Bazı durumlarda bağımsız komut bulunamayabilir ve işhatlarına hiç komut yayınlanamaz..

Süperskaler işlemcilerde veri bağımlılığını çözmek yazılımla yapılmaya kalkışılırsa, ortaya skaler işlemcilerdekine benzer sorunlar yaşanır. Derleyici tabanlı çözümler eksiklikleri, komutların işlemci içerisindeki gecikme sürelerini (latency time) tam olarak belirleyememeleridir. Bu gecikmeleri belirlemek ancak donanım tabanlı çözümlerle çalışma anında olabilir.

Süperskaler işlemcilerde veri bağımlılığı için kullanılan donanım çözüm yöntemleri, skaler işlemcilerdeki yöntemlerin daha geliştirilmiş versiyonlardır. Bu yöntemler arasında çok bitli skorbord yöntemi , dağıtım yığını ve Tomasulo Algoritması yönteminin geliştirilmiş versiyonları bulunmaktadır.

Süperskaler işlemcilerde dallanma problemi, tıpkı skaler işlemcilerde olan dallanma problemiyle aynıdır. Ancak, dallanmanın getirdiği darboğaz süperskaler işlemcilerde daha büyük olmaktadır. n komut yayınlayabilen bir süperskaler işlemcide bir dallanma komutu, n adet komutun işhatlarına yayınlanmasını engelleyecektir.

Süperskaler işlemciler dallanma problemini donanım tabanlı dallanma öngörü (branch prediction) yöntemiyle yapılmaktadır. Dallanma öngörü donanımları ile birden fazla öngörü yapılabilmektedir (Johnson, 1991).

3.2. KOMUT YAYINLAMA YÖNTEMLERİ

Çoklu işhattı kullanımı ile süperskaler işlemciler aynı anda birçok komutu işhatlarına yayınlayabilir. Komutların işhatlarına yayınlanması ve bu komutların tamamlanması üç şekilde yapılabilir :

1- Sıralı Yayınlama ve Sıralı Tamamlama (In-order Issue with In-order Completion)

2- Sıralı Yayınlama ve Sırasız Tamamlama (In-order Issue with Out-of-order Completion)

3- Sırasız Yayınlama ve Sırasız Tamamlama (Out-of-order Issue with Out-of-order Completion)

1- SIRALI YAYINLAMA ve SIRALI TAMAMLAMA

En basit komut yayınlama yöntemidir. Komutlar program sırasına göre yayınlanıp program sırasına göre tamamlanırlar. Bu yöntemin süperskaler işlemcideki çalışma düzeni Şekil 3.3' de gösterilmektedir.

Şekildeki süperskaler işlemcinin iki adet kod çözme ve yazma katı bulunmaktadır. Komutlar üç adet fonksiyonel birimlerde işletilirler. Komutlardan I1 komutu iki çevrim zamanında tamamlanır. I3 , I4 ve I5,I6 komutları aynı fonksiyonel birimleri kullanılır. I5 komutu ise I4 komutuyla veri bağımlıdır. Komutlar işlemciye programdaki sıraya göre gönderilebilir ve aynı sırada tamamlanırlar. Kod çözme katında komutların kodları bu sıraya göre çözülüp, yazma katında ise aynı sıraya göre tamamlanırlar. Kod çözme katından da görüldüğü gibi, eğer komutlar aynı fonksiyonel birimlere gönderilmişse komut gönderme işlemi durdurulur. Bu örnekte, programın tamamlanması sekiz çevrim süresi almaktadır.

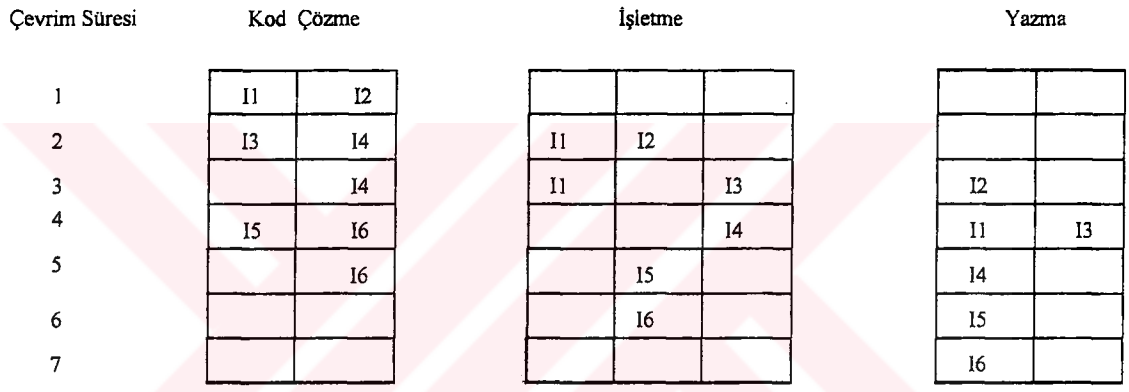
Çevrim Süresi	Kod Çözme		İşletme			Yazma	
1	I1	I2					
2	I3	I4	I1	I2			
3	I3	I4	I1				
4		I4			I3	I1	I2
5	I5	I6			I4		
6		I6		I5		I3	I4
7				I6			
8						I5	I6

Şekil 3. 3 Sıralı Yayınlama ve Sıralı Tamamlama

Bu yöntemin performansı çok düşük olduğundan skaler işlemcilerde bile kullanılmazlar.

2 - SIRALI YAYINLAMA ve SIRASIZ TAMAMLAMA

Bu yöntemde, komutlar program sırasına göre yayınlanır ama program sırasına göre tamamlanmak zorunda değildir. Sırasız tamamlama kullanılmakla sırasız tamamlamaya karşı önemli performans artışı sağlanmış olur. Çünkü, sonra gelen bağımsız komutlar işhatlarına yayınlanabilir. Ancak yayınlanacak olan ve sonra gelen komutlar, kendilerinden önce gelen komutların yaptıkları değişiklikleri etkilememelidir. Komutlar fonksiyonel birimleri aynı anda kullanmak isterlerse, komut yayını durdurulur. Bir önceki yöntemde kullanılan örneğin sıralı yayınlama ve sırasız tamamlama yönteminde uygulanması Şekil 3.4' da gösterilmektedir.



Şekil 3.4 Sıralı Yayınlama ve Sırasız Tamamlama

Bu yöntemle programın tamamlanma süresi yedi çevrim süresine inmiştir. Sırasız tamamlama yöntemi daha çok donanım kullanacaktır. Veri bağımlılıklarını belirlemek için karmaşık bir donanım kullanılması gerekecektir.

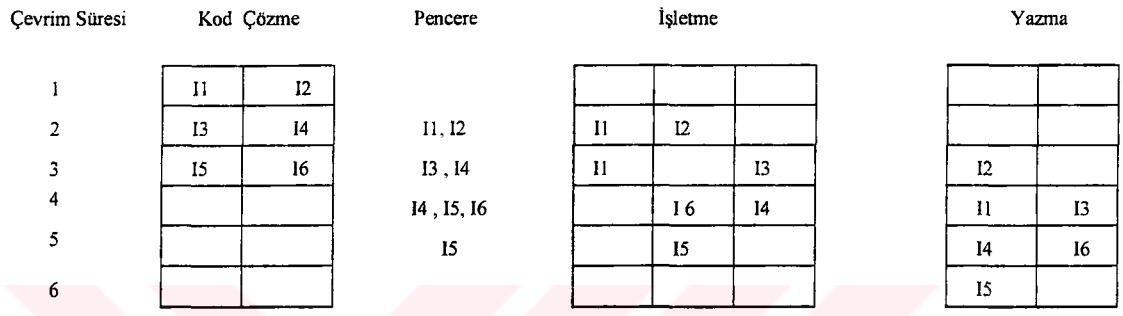
Komutlar sırasız şekilde tamamlandığından, aykırı durumlar oluştuğunda işlemci durumunun ardışıl mimari modeline uyması gerekecektir. Bu durumda kesintiye uğramış komuttan sonra gelen komutların yaptığı değişikliklerin düzeltilmesi gerekmektedir. Bunu yapacak ek bir donanım kullanma zorunluluğu ortaya çıkacaktır.

3 - SIRASIZ YAYINLAMA ve SIRASIZ TAMAMLAMA

Komutlar program sırası dışında yayınlanır ve sırasız olarak tamamlanır. Sırasız komut yayınlayabilmek için komutlar arasında veri bağımlılığının olmaması gerekir. Ancak, bundan sonra

komutlar işhatlarına yayınlanabilirler. Komutlar sırasız yayınlandığı için, doğal olarak komutların tamamlanması da sırasız olacaktır.

Komutlar, kod çözme birimine sıralı bir biçimde gelir. Kod çözme birimi kodunu çözdüğü komutları bir komut penceresine (instruction window) gönderir. Komut penceresinde komutların arasındaki veri bağımlığı ve kaynak çatışması olmayacak şekilde sırasız olarak fonksiyonel birimlere komutlar yayınlanır. Şekil 3.5' de daha önce kullanılan örneğin sırasız yayınlama ve sırasız tamamlama yönteminde uygulanışı görülmektedir.



Şekil 3.5 Sırasız Yayınlama ve Sırasız Tamamlama

Görüldüğü üzere sırasız yayınlama ve sırasız tamamlama yönteminde programın tamamlanması altı çevrim süresi almaktadır. Bu üç yöntemin içinde en yüksek performansa sahip olan yöntem sırasız yayınlama ve sırasız tamamlama yöntemidir.

3.3. SÜPERİŞHATTI ve VLIW TEKNİKLERİ

Süperskaler mimari tekniği aynı anda bir veya daha çok komutu yayınlama özelliğine dayanmaktadır. Süperskaler tekniği dışında, komut paralelliğini artıracak başka teknikler de bulunmaktadır. Bu teknikler süperişhattı ve VLIW (Very Long Instruction Word) teknikleridir.

3.3.1. VLIW MİMARİSİ

VLIW tekniğinde tüm komutlar yüzlerce bit uzunluğunda ve her komut birçok işlemi yerine getirebilecek kapasitededir. Mimaride çoklu fonksiyonel birimler bulunmaktadır. Her komuttaki belli bir alan işlemci içerisinde ayrı bir işlem yapmaktadır. Tipik bir VLIW işlemcisi ve komut formatı Şekil 3.6' de gösterilmektedir.

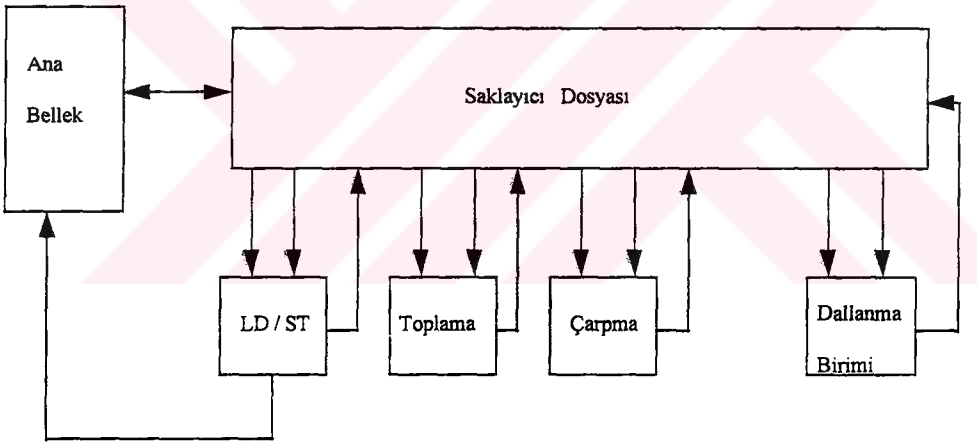
VLIW tekniğinde süperskaler tekniğinden farklı olarak komutların kodlarının çözülmesi daha kolaydır. Çünkü, VLIW işlemcide komutlar arası veri bağımlılığı ve bir komuttaki hangi işlemlerin aynı anda işletileceği derleme anında yapılır. Halbuki, süperskaler işlemcide ise, veri bağımlılığının çözülmesi ve hangi komutların aynı anda gönderileceği çalışma anında belli olur.

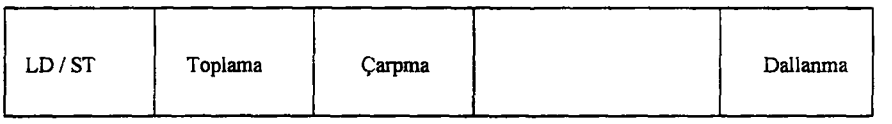
VLIW işlemcisinde komutların bazı alanlarında kullanılmayan işlemler olabilir. Bu nedenle VLIW işlemcisindeki kod yoğunluğu, süperskaler işlemciye göre daha kötüdür. VLIW işlemcilerinin farklı komut formatı olduğundan birçok paralel olmayan işlemciyle kod uyumlu değildir. Halbuki, süperskaler işlemciler skaler işlemcilerle kod uyumludurlar.

VLIW tekniği donanım yapısı ve komut kümesi olarak basit bir tekniktir. Aynı anda birçok işlem yapabilmesi, özellikle bilimsel uygulamalar için uygundur. Ama kod uyumsuzluğu nedeniyle VLIW mimarisi henüz ticari olarak bir işlemcide kullanılmamıştır.

3 . 3. 2. SÜPERİŞHATTI

m dereceli bir süperişhattı kullanan işlemcideki işhattı çevrim süresi 1/m' dir. İşhattına bir komut, her temel çevrim zamanında değil her 1/m çevrim süresinde gönderilir. Bu durumda

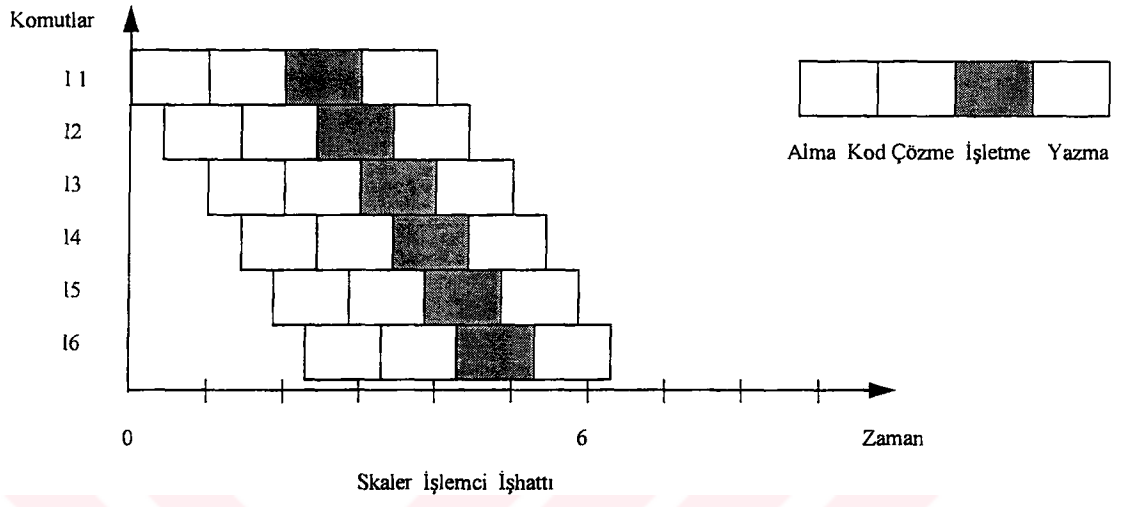




Komut Formatı

Şekil 3. 6 VLIW İşlemci ve Komut Formatı

süperişhattı kullanan bir mimarinin çevrim süresi 1/m' dir. Böylece bir temel çevrim süresi içinde m adet komut işhattına yayınlanmış olur. Süperişhattı kullanan ve m=2 olan bir işlemcinin komutları işletmesi Şekil 3.7' de gösterilmektedir.



Şekil 3. 7 m= 2 için Süperişhattı Yapısı

Süperişhattı tekniği çok uzun süreden beri kullanılan bir yöntemdir. CDC7600, Cray 1, MIPS R4000 ve Digital Alpha işlemcileri süperişhattı tekniğini kullanan işlemcilerdir.

Süperişhattı ve süperskaler işlemcileri arasındaki performans olarak karşılaştırıldığında, süperskaler işlemcilerin performans olarak biraz üstünlüğü vardır. Süperişhattında komutların gönderildiği ilk anlarda , süperskaler tekniğine göre , daha büyük miktarda bir gecikme bulunmaktadır. Ancak süperişhattı derecesi arttırıldığında performans , süperskaler işlemcinin performansına çok yaklaştığı yapılan simülasyonlara göre belirlenmiştir. (Jouppi et al, 1989)

IV. SÜPERSKALER İŞLEMCİLERDE KESME ve AYKIRI DURUMLAR

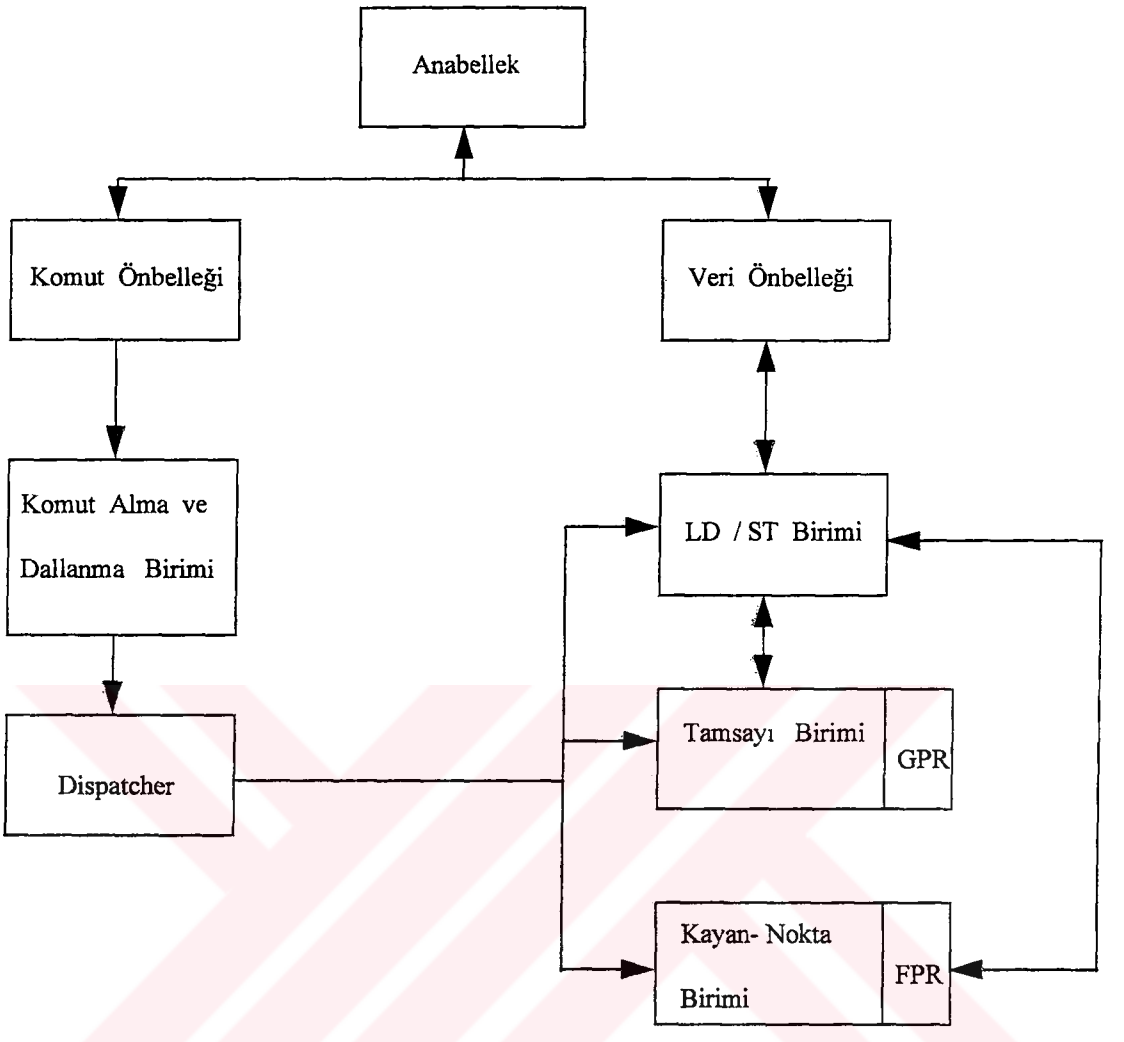
Süperskaler işlemcilerin performanslarını etkileyen etkenlerin en önemlilerinden biri de kesme ve aykırı durumlardır. Özellikle, sırasız komut yayınlama ve tamamlama yöntemini kullanan bir süperskaler işlemci içinde kesme olduğunda, bazı komutların işlemci durumunda yaptığı değişikliklerin düzeltilmesi gerekir. Eğer kesmeden sonra gelen komutlar hala işhatlarında bulunuyorsa, işhatlarından atılmaları gerekir. Ayrıca, kesmenin oluştuğu komuttan önce gelen komutlar da işhatlarında olabilir. Bu durumda bu komutların tamamlanması gerekecektir. Hatta, kesmenin olduğu komuttan önceki komutlar işhatlarına hiç gönderilmemiş olabilir. Precise kesme modeline uymak için bu komutların işhatlarına yayınlanıp tamamlanması gerekecektir.

Süperskaler işlemcilerde kesme ve aykırı durumları kotarmak için precise kesme modelini kullanan yeniden sıralama tamponu ve tarih dosyası yöntemleridir. Yeniden sıralama tamponu ve tarih dosyası yöntemleri, skaler işlemcilerdeki yeniden sıralama ve tarih dosyasına benzemektedir. Süperskaler işlemcilerde bu yöntemler biraz değiştirilerek kullanılmıştır. Mesela, üç komut yayınlıyabilen bir süperskaler işlemcide tarih dosyasının bu üç komutun kullandığı hedef saklayıcılarının değerlerini okumak için üç adet saklayıcı portunun olması gerekir. Aynı şekilde, üç komut tamamlandığında eğer yeniden sıralama tampon yöntemi kullanılıyorsa, yeniden sıralama tamponuna aynı üç sonucun yazılabilme imkanının olabilmesi gerekir.

Bazı süperskaler işlemcilerde kullanılan kesme ve aykırı durum modelleri incelenip, kullandıkları kesme kotarma yöntemleri ayrıntılı olarak gösterilecektir.

4.1. POWERPC 603.

PowerPC 603 işlemcisi POWERPC mimarisini kullanan bir süperskaler işlemcidir. İşlemci içinde tamsayı, kayan-nokta, dallanma ve Load / Store birimleri bulunmaktadır. Veri ve komut önbellekleri ayrı ayrıdır. İşlemcinin yapısı Şekil 4.1.' de gösterilmiştir.



Şekil 4.1 POWERPC 603 İşlemcisinin Blok Diyagramı

Şekil 4.1' deki dispatcher birimi, komut belleğinden gelen komutları fonksiyonel birimlere dağıtan birimdir.

POWERPC 603 işlemcisinde kesmeler, aykırı durumu lojiği tarafından gözlenir. Tamamlanma birimi (Completion Unit) olarak adlandırılan bir birim, komutların gönderilmesinden tamamlanmasına kadar tüm komutları takip eder. Bir komut gönderileceği zaman, her komut için bir tamamlanma saklayıcısı tamamlanma birimi tarafından atanır. Komutlar tamamlandığında, sonuçlarını doğrudan saklayıcı dosyasına yazarlar. Komutlar sırasız olarak tamamlandığından, işlemcinin precise kesme modelini sağlayabilmesi için komutların sıralı olarak tamamlanmasının sağlanabilmesi gerekir. Tamamlanma saklayıcısı sıralı komut tamamlamak için her komutla ilgili yeterli bilgiyi saklarlar. Tamamlanma saklayıcıları , komutların hedef saklayıcılarının eski değerlerini saklayan bir tür tarih dosyası sayılabilirler.

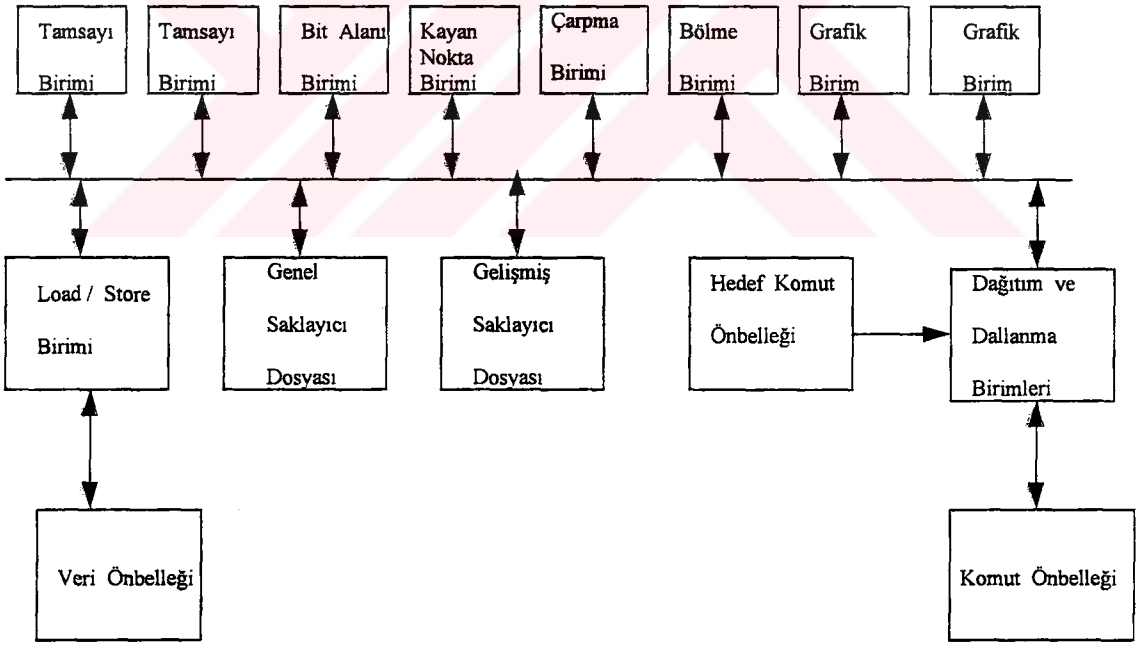
Tamamlanma saklayıcıları her komut için tahsis edilirler. Eğer tamamlanma biriminde tahsis edilecek saklayıcı kalmazsa, komut gönderimi durdurulur.

4. 2. MOTOROLA 88110

MOTOROLA 88110 işlemcisi, içinde 10 adet fonksiyonel birim bulunan ve aynı anda fonksiyonel birimlere 2 komut yayınlayabilen süperskaler mimeriye sahip bir işlemcidir. Precise kesme modelini kullanmaktadır. Sırasız komut yayınlama ve sırasız komut tamamlama yöntemlerine izin vermektedir. İşlemcide veri ve komut önbellekleri ayrıdır. Şekil 4.2' de MOTOROLA 88110 işlemcinin blok diyagramı görülmektedir.

88110 işlemcisi precise kesme modelini kullanır. Bir komutta aykırı durum olduğunda, aykırı durum olan komuttan önce gelen tüm komutların tamamlanması beklenir. Daha sonra, komut gönderimi ve işletimi durdurulur. İşlemci, tarih dosyası yardımıyla aykırı duruma yolaçan komuta kadar gerekli düzeltmeler yapılır. Tarih dosyasında gerekli düzeltmelerin yapılabilmesi için komutlarla ilgili yeterli bilgiler tutulmaktadır.

Bir dış kesme olduğunda, komut işletimi durdurulur ve tüm bitmemiş komutlar iptal edilir. Sırasız olarak tamamlanmış olan komutların yaptığı değişiklikler düzeltilir.



Şekil 4.2 MOTOROLA 88110 İşlemcisinin Blok Diyagramı

4. 3. Intel P6

Intel P6 işlemcisi ayrı veri ve komut önbelleklerine sahip olan süperskaler bir işlemcidir. İşlemci içerisinde komutlar üç adet paralel kod çözücü tarafından sıralı olarak yayınlanır. Komutların yayını işlemci içindeki alma / kod çözme birimi tarafından yapılmaktadır. Komutlar önce komut havuzuna (instruction pool) gönderilir. Dağıtım / İşletim birimi ise komut havuzundan komutları alır ve fonksiyonel birimlere gönderirler. Tamamlanan komutlar ise tekrar komutuna komut havuzuna gönderilir. Dağıtım / İşletim birimi her saat çevriminde 3 komut gönderimi yapabilmektedir. Tamamlanan komutların sonuçları komut havuzuna yazılırlar.

Tamamlanmış komutları emekli etme (retire) birimi, komut havuzundan alır. Emekli etme birimine giden komutların sonuçları sonradan gelen komutlar tarafından ihtiyaç duyulabileceğinden, bu birimdeki sonuçlar doğrudan rezervasyon istasyonları tarafından alınabilmektedir.

Emekli etme biriminin amacı, emekli edilmiş olan komutları program sırasına göre komut havuzundan alıp, komutların sonuçlarını program sırasına göre saklayıcı dosyasına yazmaktır. Böylece precise kesme modeli sağlanmış olur. Bir kesme oluştuğunda, kesmenin oluştuğu komuttan önceki tüm komutlar komut havuzundan silinirler. Komut havuzundan silinen komutların yaptıkları değişiklikler saklayıcı dosyasına yansımadığından işlemci içindeki durum precise olmaktadır.

Intel P6 işlemcisinde kullanılan kesme kotarma yöntemi, yeniden düzenleme tampon yöntemidir. Çünkü komutların sonuçları doğrudan saklayıcı dosyasına yazılmayıp, komut havuzunda tutulmaktadır. Intel P6 işlemcisinin yapısı Şekil 4.3' de gösterilmiştir.

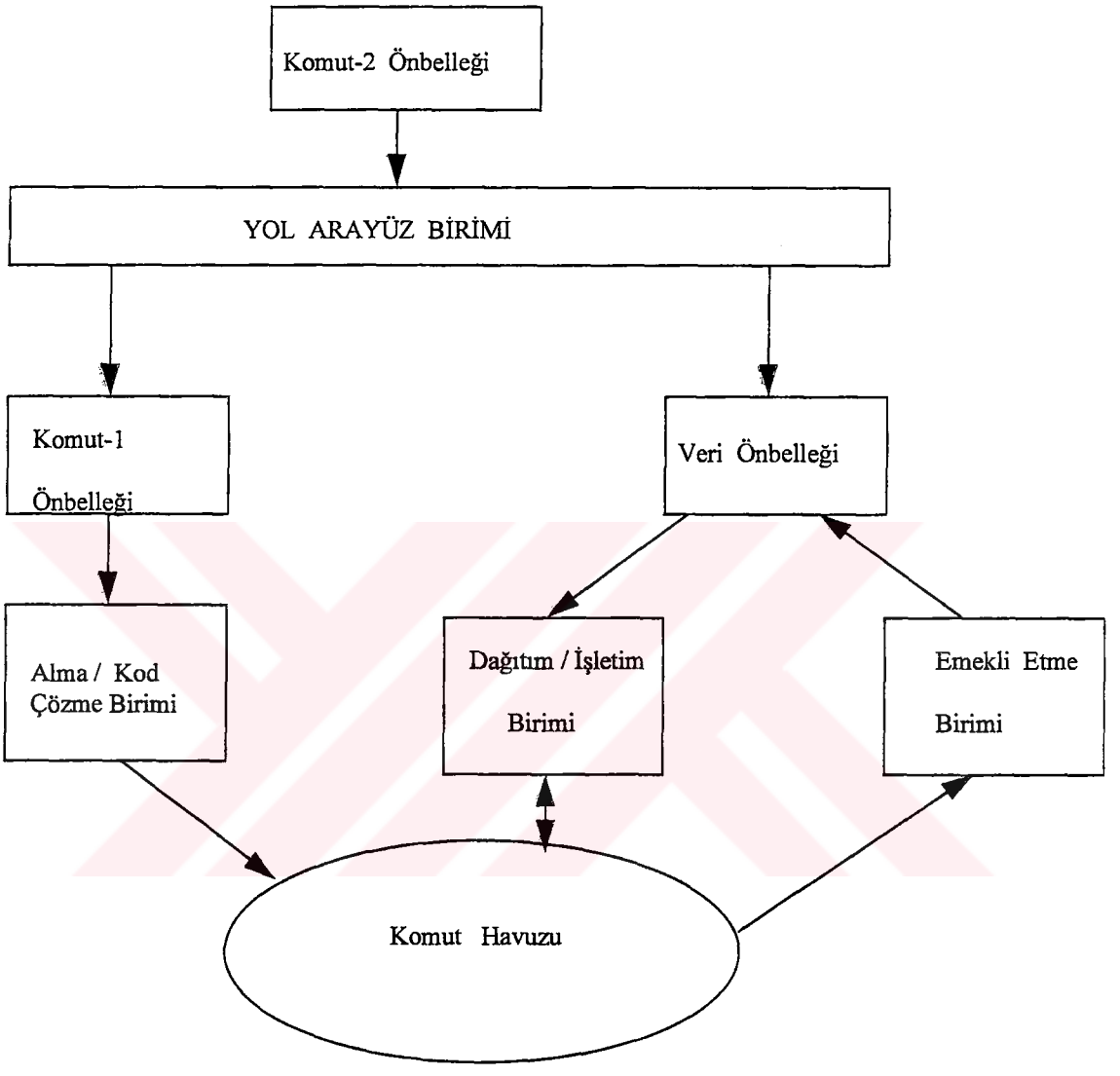
Çeşitli işlemcilerin kesme kotarma yöntemleri incelenmiştir. Bu işlemcilerin hepsi precise kesme modelini kullanmışlardır. Precise kesme modelindeki hedef, kesme oluşan komuttan sonraki tüm komutların yaptığı değişiklikler düzeltilir. Böylece işlemci durumu ardışıl mimari modeline uygun hale gelir.

Bu yöntemler dışında performansı artıracak ve süperskaler mimari yapısına uygun yeni bir kesme modeli önerilebilir. Bu model, precise olmayıp tamamen imprecise bir modeldir. Şimdi yeni önerilen bu yöntem incelenecektir.

4 . 4. GÖLGE TUTUCU YIĞINLAR YÖNTEMİ

Gölge Tutucu Yığınlar adı verilen yeni yöntem, kesmeleri imprecise olarak kotaran bir yöntem olarak önerilmiştir. Precise kesme yöntemlerinde, kesmenin olduğu komuttan sonra gelen komutların yaptıkları değişiklikler düzeltilir. Kesmenin oluştuğu komutun program sayıcı adresi bulunarak kesme servis programına gidilir. Kesme servis programından dönüşte program sayıcı adresinden itibaren komutlar tekrar işletilir. Kesme olmadan önce, kesme oluşacak komuttan sonraki komutlardan bazıları tamamlanmış olabilir. Bu komutlar bağımsız komutlar olduğundan, işletilmesinde bir mahsur olmadığı

için işhatlarına gönderilmişlerdir. Kesme geldiğinde ise bu komutların yaptıkları değişiklikler düzeltilir. Kesme servis programından dönüşte bu komutlar tekrar işletilecektir. Bu durumda yararlı pipeline çevrim süreleri boş yere harcanmış olacaktır.

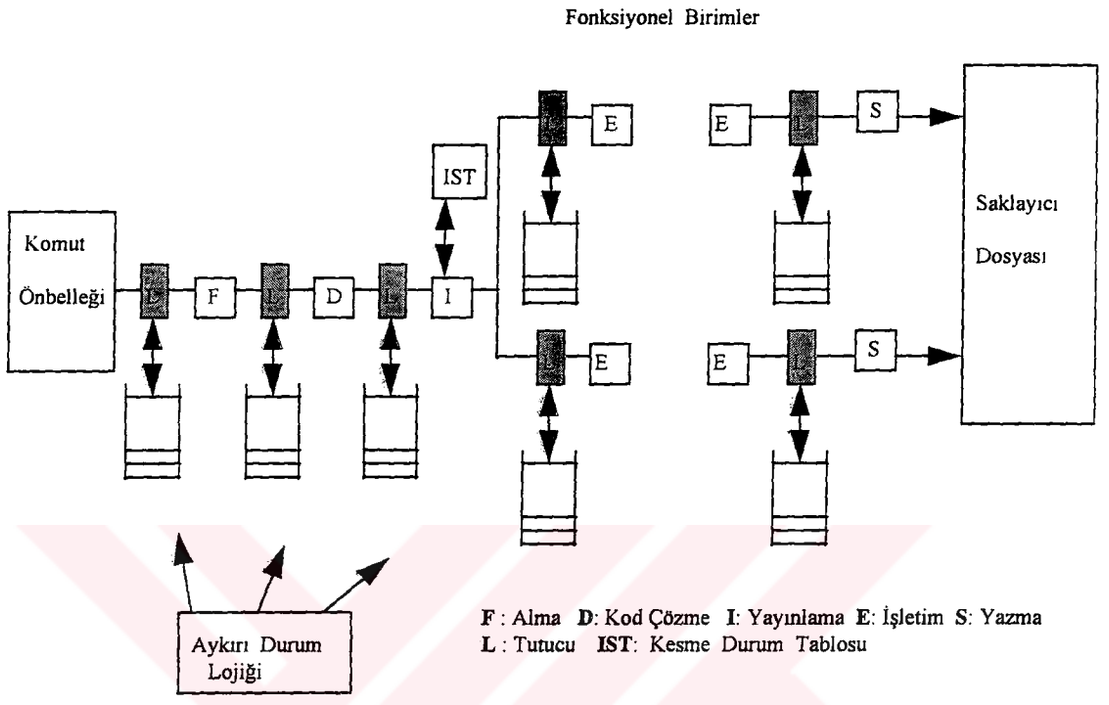


Şekil 4.3 Intel P6 İşlemcisi

Önerilen yeni yöntemde kesme oluştuğunda, kesme oluşan komuttan sonra gelen komutların yaptığı değişiklikler düzeltilmezler ve tüm işhatları dondurulur. Dondurulan işhatlarındaki komutların saklanabilmesi için işhatlarındaki her tutucu için bir tutucu yığını eklenmiştir. Önerilmiş olan bu yöntemin mimari yapısı Şekil 4.4' de 2 komut yayınlayabilen bir mimari için gösterilmiştir.

Mimaride komutlar komut önbelleğinden alınıp, kodları çözülüp yayınlama katındaki komut penceresine gönderilir. Komut penceresinde komutlar arasındaki veri bağımlılığı çözülüp komutlar

fonksiyonel birimlere gönderilir. Yayınlama katı bir veya daha çok komutu fonksiyonel birimlere gönderilebilmektedir.



Şekil 4.4 2 Komut Yayınlayabilen ve Gölge Tutucu Yığını Kullanan bir Süperskaler Mimari Modeli

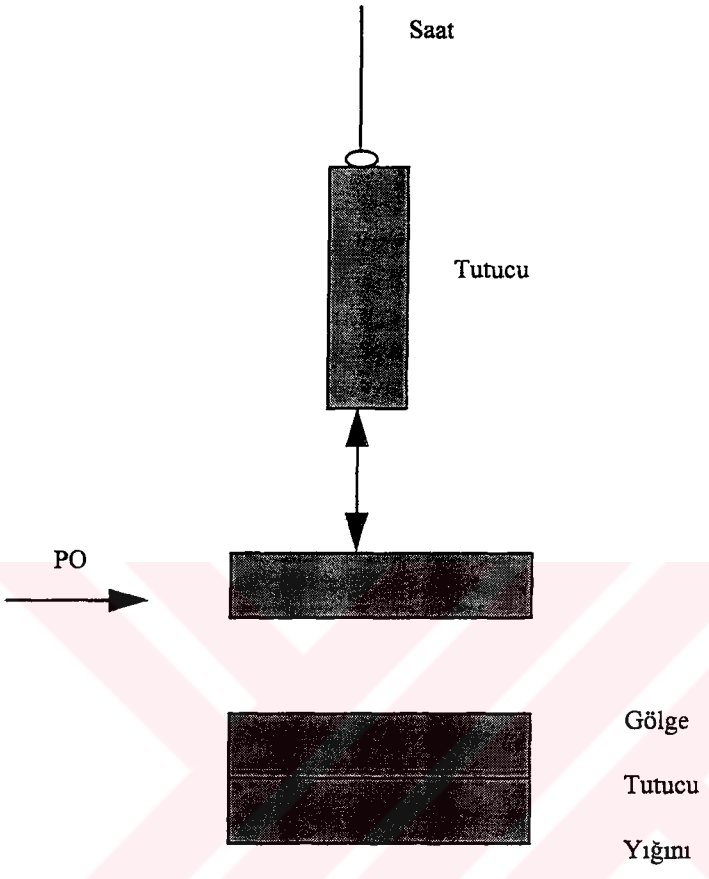
Bu mimaride debug ve breakpoint kesmeleri dışında tüm dış kesmeler ve aykırı durumlar imprecise olarak kotarılmaktadır. Sadece debug ve breakpoint kesmeleri precise olarak kotarılacaktır ve bu kesmelerin debug ve breakpoint komutları tarafından üretildiği kabul edilmektedir. Ayrıca yayınlama katından önce olacak aykırı durumlar, imprecise olarak kotarılacaktır.

Imprecise bir kesme meydana geldiğinde, komut yayını durdurlur ve tüm işhatları dondurulur. İşhatlarında yarım kalmış olan komutlar gölge tutucu yığınlara atılarak saklanırlar. İşhatlarındaki her tutucuya bir gölge tutucu yığın eşlik etmektedir. (Şekil 4.5)

Gölge tutucu yığın , LIFO (Last-in-First-Out) şeklinde düşünülmüş bir yapıdadır. Yığın olarak düşünülmesi içiçe oluşacak kesmelerde işlemcilerin durumlarının yığınlarda saklanabilmesi içindir.

Kesme sırasında, kesme servis programına gidilebilmesi için ne tür bir kesme olduğu ve hangi komutta meydana geldiği bilinmelidir. Çünkü kesme servis programı kesmeyi kotarırken kesmeye yolaçan durumu ortadan kaldırmalıdır. Mesela, bir taşma (overflow) olduğunda , kesme servis programı taşan

saklayıcıyı bularak işlemci içindeki en büyük sayıyı o saklayıcıya yazarak taşmanın nedenini ortadan kaldırılır. Bu sebeble işhatlarında bulunan komutlara ait olan bazı bilgiler kesme durum tablosu adı



Şekil 4. 5 Gölge Tutucu Yığını

verilen bir tabloda tutulurlar. Bu tablo Şekil 4.6' de görülmektedir.

Program sayıcı adresi kesmenin oluştuğu komutu bulmak için kullanılır. Fonksiyonel birimler ise kesmenin oluştuğu komutun hangi fonksiyonel birimde gerçekleştiğini bulmak için kullanılır. İşhattı gecikmesi ise bir komutun, gönderileceği fonksiyonel birimde kaç işhattı çevrim süresinde biteceğini veren sayıdır. Her işhattı katının bir çevrim süresi aldığı kabul edilmektedir. İşhattı gecikme sayısı komut yayınlama katına geldiğinde kesme durum tablosuna yazılır ve her çevrim süresinde bu sayı bir azaltılır. İşhattı gecikme sayısı fonksiyonel birimdeki hangi işhattı katında kesme olduğunu bulmak için kullanılır. İşhattı gecikme sayısı 0 olduğunda, kesme durum tablosunda o satır geçersiz yapılır ve aynı satır sonra gelecek olan komutlar için kullanılabilir.

Imprecise bir kesme olduğunda, kesmenin olduğu fonksiyonel birim ve ilgili işhattı katı bilgisi aykırı durum lojiğine gelir. Bu bilgiler kesme durum tablosundaki fonksiyonel birim ve işhattı gecikme sayılarıyla asosiyatif olarak karşılaştırılır. Eğer uyuşma varsa, kesmeye yolaçan komutun program sayıcı

adresi elde edilir ve kesme servis programı gerekli düzeltmeleri yapar. Kesme servis programına gitmeden önce yayınlama katındaki komut penceresi ve kesme durum tablosu saklanmak zorundadır. Servis programından dönüşte, işlemci kesme durum tablosu ve komut penceresi tekrar yüklenir ve gölge tutucu yığındaki yarım kalmış komutlar tutuculara geri yüklenirler. Bundan sonra işlemci yeniden çalışmaya hazır hale gelir.

Program Sayıcı	Fonk. Birim	İşhattı Gecikmesi	Geçerli Biti

Şekil 4. 6 Kesme Durum Tablosu

Precise kesmeleri kotarmak için de şöyle bir çözüm yöntemi önerilmiştir. Yayınlama katındaki komut penceresinde komut incelenir ve precise bir kesme olup olmadığına karar verilir. Eğer kesme precise ise komut alma durdurulur. Precise kesme komutundan önceki komutların tamamlanması beklenir. Daha sonra precise kesmeye yolaçacak komut ve aynı fonksiyonel birimi kullanan ve sonra gelen diğer komutlar yayınlanır.

Daha sonraki adımlar, imprecise kesmede takip edilen adımlarla aynıdır. Önerilen mimari yapıda, precise kesmelerin nadiren olduğu düşünülürse, precise kesmeler performansı büyük miktarda etkilemeyecektir.

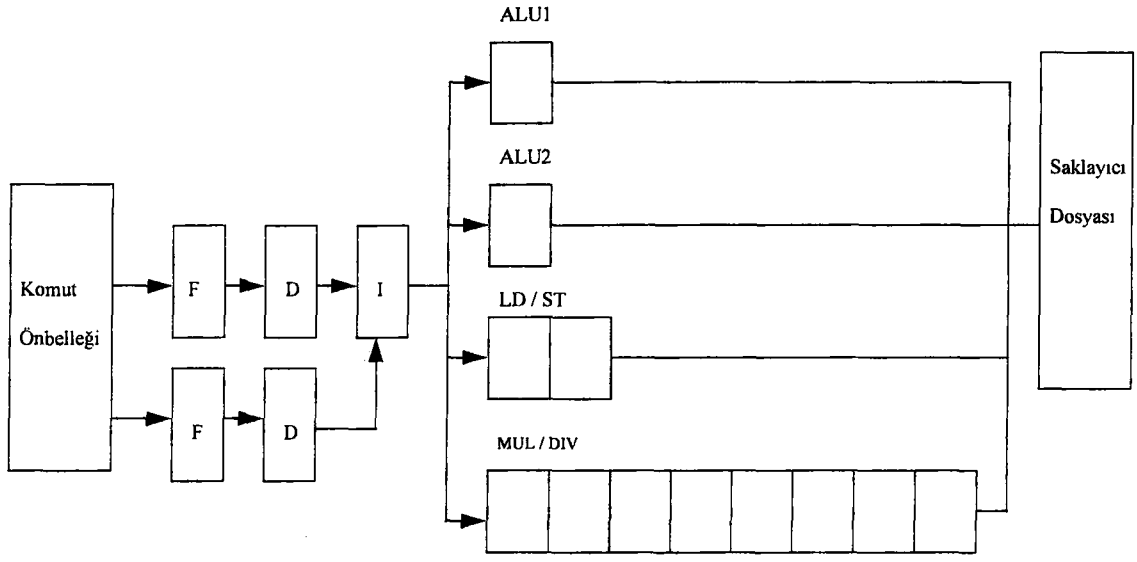
Önerilen bu yöntemin, diğer yöntemlere nazaran daha yüksek performansa sahip olduğu yapılan simülasyonla belirlenmiştir. Simülasyonla ilgili ayrıntılı bilgiler Bölüm V' de anlatılacaktır. Yöntemin donanım maliyeti ise diğer yöntemlere göre aşırı abartılı olmadığı görülmüştür.

V. SİMÜLASYON

Bu bölümde süperskaler mimarilerde kesme kotarma problemine karşı getirilmiş olan yöntemlerle kendi önerdiğimiz yöntem karşılaştırılacaktır. Önermiş olduğumuz gölge tutucu yığın yöntemiyle, yeniden sıralama tamponu ve tarih dosyası karşılaştırılacaktır. Bu karşılaştırma hem performans olarak hem de donanım maliyeti olarak yapılacaktır. Performansı ölçmek için seçilen bir mimari model üzerinde her bir yöntem simüle edilmelidir. Simülasyonu yapmak için çeşitli benchmark programların bu mimari model üzerinde her bir yöntem için denenmelidir. Her üç yöntem için her bir benchmark programın kaç makina çevrim süresi aldığı hesaplanır. Elde edilen verilere göre yöntemler içinde en yüksek performansa sahip olan yöntem belirlenir.

5 . 1. MİMARİ MODEL

Simülasyonu yapılacak olan mimari model aynı anda iki komut gönderebilen süperskaler bir mimari modeldir. Model mimaride 4 adet fonksiyonel birim bulunmaktadır. İki adet ALU birimi, bir adet Load / Store birimi ve bir adet çarpma / bölme devresi bulunmaktadır. Model mimari tipik bir RISC mimari yapısındadır. Tüm komutlar işlemleri saklayıcılar üzerinden yaparlar. Bellek üzerinden yapılacak olan işlemler ise load ve store komutlarıyla yapılmasına izin verilmektedir. Mimari model Şekil 5.1' de görülmektedir.



Şekil 5.1. Mimari Model

Mimari modelde komutlar komut önbelleğinden alınır ve kodları çözülür. Kodları çözülen komutlar komut yayınlama katına (I) gelirler. Burada tutulan komutlar arasında veri bağımlılığı problemi ortadan kaldırılan komutlardan iki tanesi aynı anda fonksiyonel birimlere gönderilebilir. Fonksiyonel birimlerde işletilen komutlar saklayıcı dosyasına sonuçları yazarlar. Fonksiyonel birimlerden ALU' lar tek çevrim süresinde, LD / ST birimi iki çevrim süresinde, MUL / DIV birimi ise sekiz çevrim süresinde komutları işletirler.

Seçilem mimari modelin veri yolu ve adres yolu uzunlukları 32 bit olarak kabul edilmektedir. Ayrıca mimaride 16 adet 32 bitlik saklayıcının olduğu varsayılmaktadır. Mimarideki komut formatının uzunluğu 32 bittir. İşlemci içinde 11 komuttan oluşan temel komut kümesi bulunmaktadır. Bu komutlar mimari model üzerinde denenecek olan benchmark programlarını işletebilmek için seçilmiş temel komutları temsil etmektedir. Komutlar ve yerine getirdiği fonksiyonlar Tablo 5.1' de gösterilmiştir.

Tablo 5.1. Komut Kümesi

KOMUT	FONKSİYON
ADD	Toplama İşlemi
LD	Load İşlemi
ST	Store İşlemi
LOOP	Döngü İşlemi
MUL	Çarpma İşlemi
DIV	Bölme
SUB	Çıkarma
HLT	Dur
NOP	No operation
MOV	Taşıma
BR	Dallanma

5.2. PERFORMANS ANALİZİ

Mimari model üzerinde kesme kotarma problemine çözüm olacak üç yöntem modellenecektir. Öncelikle yeniden sıralama tamponu ve tarih dosyası yöntemleri mimari model üzerinde modellenecek, ardından yeni önerilen gölge tutucu yığın yöntemi aynı mimari üzerinde modellenecektir. Daha sonra elde edilen veriler ile yöntemlerin performansları belirlenecektir. Yeniden sıralama tamponu ve tarih dosyası yöntemlerinin karşılaştırma için seçilmesinin sebebi, her iki yöntemin de ticari olarak süperskaler işlemcilerde kullanılan en sık yöntemler olmasıdır.

Performans analizi yapabilmek için çeşitli benchmark programların model üzerinde denenmesi gerekmektedir. Benchmark programlarının seçiminde , en çok bilinen benchmark programlar olması önemli etken olmuştur. Bunun yanısıra aynı konu üzerinde yapılan çalışmalarda kullanılan benchmark

programlar da bu seçim de etkili olmuştur (Wang et al , 1993). Seçilen benchmark programları şunlardır :

- Fibonecci Sayıları
- Loop Programı
- Matris Çarpımı
- Asal Sayılar

(BYTE , 1987)

Seçilen bu benchmark programları önce yüksek seviyeli bir dil olan C dilinde kodlanmış, sonra da mimari modelin makina kodlarına dönüştürülmüştür. Her programın C ve makina dilindeki kodları aşağıda verilmektedir.

Fibonecci Sayıları :

C Kodu:
```
fib[0]=1; fib[1]=1;
for(i=2;i<50; I++) fib= fib [ i-1]+fib [I-2];
```

Makina Kodu:
```
        MOV  R1,1
        ST [200],R1     ; Veriler 200. bellek adresinden başlar.
        ST [201],R1
        MOV R2,50
        MOV R3,200
        LD R4,[R3]
        ADD R3,1
        LD R5,[R3]
DONGU : ADD R4,R5
        ADD R3,1
        ST [R3],1
        MOV R6,R5
        MOV R5,R4
        MOV R4,R6
        LOOP  R2,DONGU
        HLT
```

Fibonecci programındaki veri sayısı 50' dir. Yani 50' ye kadarki Fibonecci sayılarını bulmaktadır.

Loop :

C Kodu : for (I=0; i<50;i++)

```
{
  a[i]+=a[i]*b[i];
  c[i]+=c[i]/d[i];
 }
```

Makina Kodu:

```
        MOV R1,200   ; a dizisinin bellek adresi
        MOV R2,400   ; b          "
        MOV R3,600   ; c          "
        MOV R4,800   ; d          "
        MOV R5,50
DONGU:  LD R6,[R1]
        LD R7,[R2]
        MUL R7,R6
        ADD R6,R7
        ST [R1],R6
        LD R8,[R3]
        LD R9,[R4]
        MOV R10,R8
        DIV R10,R9
        ADD R8,R10
        ST [R3],R8
        ADD R1,1
        ADD R2,1
        ADD R3,1
        ADD R4,1
        LOOP R5, DONGU
        HLT
```

Loop denilen bu benchmark programı 50 kez tekrarlanan çarpma ve bölme işlemlerinden ibarettir.

Matris Çarpımı :

C Kodu : for (i=0;i<3;i++)

```
{
```

```
        for (j=0;j<3;j++)
          for (k=0;k<3;k++)  c[i][j]+=a[i][k]*b[k][j];
      }
```

Makina Kodu :

```
          MOV R10,3
          MOV R11,3
          MOV R12,3
          MOV R1,0
DONGU1:   MOV R2,0
DONGU2:   MOV R3,0
DONGU3:   MOV R4,R1
          MUL R4,3
          MOV R5,R4
          ADD R4,R2
          ADD R5,R3
          MOV R6,R3
          MUL R6,3
          ADD R6,R2
          LD R7,[R5]
          LD R8,[R6]
          MUL R7,R8
          LD R9,[R4]
          ADD R9,R7
          ST [R4],R9
          ADD R3,1
          LOOP R12, DONGU3
          ADD R2,1
          LOOP R11, DONGU2
          ADD R1,1
          LOOP R10, DONGU1
          HLT
```

Matris çarpımı programı 3x3' lük iki matrisin çarpımını hesaplayan programdır.

Asal Sayılar :

C Kodu : a[0]=0;

```
for ( i=1;İ<50; I++) a[i]=1;
for ( i=1;i<25;i++)
   for ( j=1; j <25/i;j++)  a[i*j]=0;
```

Makina Kodu:

```
          MOV R1,0
          ST [R1],0
          LD R3,50
          MOV R2,1
DONGU0:   ST [R2],1
          ADD R2,1
          LOOP R3,DONGU0
          MOV R3,1
          MOV R4,25
DONGU1:   MOV R5,25
          DIV R5,R3
          MOV R6,1
DONGU2:   MUL R3,R6
          ST [R3],0
          ADD R6,1
          LOOP R5,DONGU2
          ADD R3,1
          LOOP R4,DONGU1
          HLT
```

Asal sayılar programı 1' den 50' ye kadar asal olan sayıları hesaplar.

Benchmark programlarının simülasyonu her yöntem için ayrı ayrı C dilinde hazırlanmıştır. Simülasyon programlarının kaynak kodları EKLER Bölümünde bulunmaktadır. Benchmark programları her yöntem için ayrı ayrı değerlendirilmelidir. Yeniden sıralama tamponu ve tarih dosyası yöntemleri skaler işlemcilerdeki yeniden sıralama ve tarih dosyası yöntemleriyle aynıdır. Fakat aynı anda iki komut yayınlayabilen bir süperskaler işlemci için iki adet sonuç kaydırmalı saklayıcı kullanmak zorunluluğu vardır. Çünkü işlemci aynı anda iki komut yayınlayabildiği için komutların işhatlarında takibi ancak iki adet sonuç kaydırmalı saklayıcı kullanmak ile mümkün olabilmektedir.

Her benchmark programında performans analizi yapabilmek için program içinde kesmelerin oluşması gerekmektedir. Oluşacak kesmeler, program içersinde rastgele seçilen komutların yolaçacağı kesmeler olacaktır. Kesme oluştuğunda her yöntem kendi kesme kotarma işlemini yapacak ve benchmark programını tamamlayacaktır. Benchmark programının işletilme süresi, o programın makina çevrim süresini belirler. Yeniden sıralama tamponu, tarih dosyası ve gölge tutucu yığın yöntemlerinin performans karşılaştırılmaları Tablo 5.2.' de verilmiştir.

Tablo 5.2. Üç Yöntemin Performans Karşılaştırılması (Birim : Makina Çevrim Süresi)

	Yeniden Sıralama Tamponu	Tarih Dosyası	Gölge Tutucu Yığını
FIBONECCI	214	213	209
LOOP	1456	916	906
MATRİS	1054	787	759
ASAL	1417	1416	1416
TOPLAM	**4141**	**3332**	**3290**

Simülasyon sırasında yeniden sıralama tamponunun ve tarih dosyasının uzunluğu 10, gölge tutucu yığın yöntemindeki kesme durum tablosunun uzunluğu olarak 12 alınmıştır. Yeniden sıralama ve tarih dosyasının uzunluğunun 10 olarak seçilmesi bu değerin en optimal değer olmasıdır (Smith et al, 1988). Kesme durum tablosunun uzunluğunun 12 olması mimarideki toplam işhattı kat sayısının 12 olmasıdır. (ALU1=1, ALU2=1, LD/ST=2, MUL/DIV=8)

Tablo 4.1.' den görüldüğü üzere üç yöntemden en yüksek performansa sahip yöntem, en az sayıda makina çevrim süresi harcamış olan gölge tutucu yığın yöntemidir.

Makina kodlarına çevrilmiş programların aynı anda iki komut yayınlayabilen bir süperskaler mimaride simüle edilebilmesi için fonksiyonel birimlere iki komut yayınlamak gerekir. Bu nedenle C dilinde yazılmış bir benchmark programı önce model mimarinin makina koduna çevrilir, sonra makina kodları aynı anda iki komut yayınlayabilecek şekilde yeniden düzenlenir. Komutlar yeniden düzenlendiği

için fonksiyonel birimlere sırasız yayınlanabilmektedir. Sırasız komut yayınlamak için komutlar arasında veri bağımlılığının ve kaynak çatışmasının olmaması gerekir. Kullanılan benchmark programlar yapı itibariyle veri bağımlılığına müsait olduğu için bazı işhattı çevrim sürelerinde fonksiyonel birimlere yayınlanabilecek komut bulunamayabilir. Bu durumda, fonksiyonel birimlere NOP (No Operation) komutu gönderilir. Tablo 4.1' den de görüldüğü üzere ilk üç benchmark programda gölge tutucu yığını yönteminin bariz üstünlüğü vardır. Bunun sebebi, bu programlarda pekçok komutun fonksiyonel birimlere sırasız olarak yayınlanabilmesidir. Asal sayılar programında ise fonksiyonel birimlere yayınlanan komutlar çoğunlukla sıralı olarak yayınlanmıştır. Bu nedenle, gölge tutucu yığın yönteminin harcadığı çevrim sayısı, diğer yöntemlerin harcadığı çevrim sayılarıyla hemen hemen aynıdır. Sonuç olarak, süperskaler mimarilerde komutların sırasız yayınlanma oranı arttıkça, gölge tutucu yığın yönteminin diğer yöntemlere göre performansı daha yüksek olmaktadır.

5.3. DONANIM MALİYETİ

Donanım maliyeti, her yöntemin kesme kotarma işlemi için kullandıkları tüm donanımların toplam bit sayısı hesaplanacaktır. Daha sonra her yöntemin donanım maliyeti karşılaştırılarak en iyi maliyet - performans ikilisine sahip olan yöntem bulunacaktır.

Yeniden Sıralama Tampon Yönteminin Donanım Maliyeti

Yeniden sıralama tamponu yöntemi, iki adet sonuç kaydırmalı saklayıcı ve bir yeniden sıralama tamponu kullanmaktadır. Sonuç kaydırmalı saklayıcıda bulunan alanlar şunlardır:

Kat	**Fonk. Birim**	**Geçerli Biti**	**Etiket**	
3	2	1	4	(Bit)

Mimarideki en uzun pipeline kat sayısı 8 olduğundan, sonuç kaydırmalı saklayıcını uzunluğu 8 olacaktır. Buna göre **Kat** adlı alan 3 bitle ifade edilebilir. Mimaride toplam 4 adet fonksiyonel birim olduğundan **Fonk. Birim** 2 bitle temsil edilebilir. **Etiket** alanı yeniden sıralama tamponundaki bir satırı ifade göstermektedir. Yeniden sıralama tamponunun uzunluğu 10 olduğu için **Etiket** alanı 4 bitle temsil edilebilir. Böylece, bir sonuç kaydırmalı saklayıcıdaki (SKS) toplam bit sayısı :

SKS = 8 * (3+2+1+4) = 80 bit.

İki adet SKS kullanıldığından ;

SKS_toplam = 2 * 80 = 160 bit.

Yeniden sıralama tamponunda ise şu alanlar bulunmaktadır :

Etiket Hedef Saklayıcı Sonuç Aykırı Durum Geçerli Biti Program Sayıcı

4	4	32	4	1	32

Mimaride toplam 16 adet saklayıcı olduğundan **Hedef Saklayıcı** alanı 4 bitle gösterilmiştir. **Sonuç** alanı ise hedef saklayıcının içeriğidir , saklayıcılar 32 bitlik olduğundan **Sonuç** alanı 32 bittir. **Aykırı Durum** bilgisi mimarinin tanımlamış olduğu kesme sayısına bağlı bir parametredir. Mimarideki kesme sayısı olarak 16 olarak kabul edilirse, **Aykırı Durum** alanı 4 bitle temsil edilebilir. **Program Sayıcı** , adres yolunun uzunluğunun aynısıdır. Bu durumda yeniden sıralama tamponunun (YST) toplam bit sayısı :

YST = 10 * (4+4+32+4+1+32) = 810 bit.

Sonuç olarak, **YST yönteminin donanım maliyeti**:

YST_toplam = SKS_toplam + YST = 160 + 810 = 970 bit.

Tarih Dosyası Yönteminin Donanım Maliyeti

Tarih dosyası (TD) yöntemi de YST yöntemi gibi iki adet SKS kullanır. Fakat tarih dosyasındaki SKS' de Hedef Saklayıcı alanı bulunmaktadır. Tarih dosyasındaki SKS yapısı şöyledir :

Kat	**Fonk. Birim**	**Hedef Saklayıcı**	**Geçerli Biti**	**Etiket**	
3	2	4	1	4	(Bit)

Buna göre iki adet SKS' nın toplam bit sayısı :

SKS_toplam = 2 * 8 * (3+2+4+1+4) = 224 bit.

Tarih dosyasında bulunan alanlar şunlardır :

Etiket	**Hedef Saklayıcı**	**Eski Değer**	**Aykırı Durum**	**Geçerli**	**Program Sayıcı**
4	4	32	4	1	32 (Bit)

Eski Değer alanı hedef saklayıcının fonksiyonel birime gönderilmeden önceki değerdir. Tarih dosyasının toplam bit sayısı :

TD = 10 * (4+4+32+4+1+32) = 810 bit.

Sonuç olarak, **TD yönteminin donanım maliyeti**:

TD_toplam = SKS_toplam + TD = 224 + 810 = 1034 bit.

Gölge Tutucu Yığın Yönteminin Donanım Maliyeti

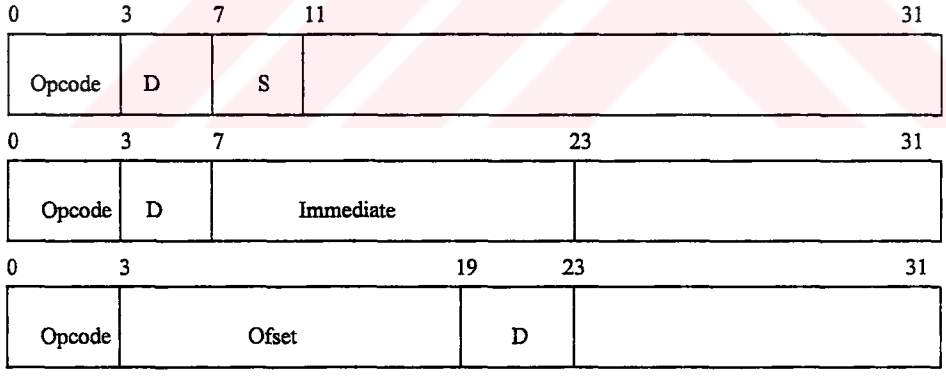
Gölge tutucu yığın yöntemi bünyesinde kesme durum tablosunu ve gölge tutucu yığınlarını barındırmaktadır. Kesme Durum Tablosundaki (KDT) alanlar aşağıdaki gibidir :

Program Sayıcı	**Fonk. Birim**	**İşhattı Gecikmesi**	**Geçerli Biti**	
32	2	3	1	(Bit)

İşhattı Gecikmesi değeri mimarideki en uzun işhattı katının uzunluğu belirler. Model mimarideki en uzun işhattı katının uzunluğu 8' dir. Bu sebeble, **İşhattı Gecikmesi** 3 bitle gösterilebilir. KDT' nin uzunluğu için optimal değeri 12 olarak kabul edilir. Çünkü , mimari modelin işhatlarında toplam kat sayısı 12 olduğundan işlemci içerisinde en fazla 12 komut olacaktır. Bu nedenle, KDT tablosunun uzunluğu 12 olarak tespit edilmiştir. KDT' nin toplam bit sayısı :

KDT = 12 * (32+2+3+1) = 456 bit.

Gölge tutucu yığınların donanım maliyetini hesaplayabilmek için komut formatını tanımlamak gerekir. Komut formatı 32 bitlik formattadır. Komut formatı Şekil 5. 2' de gösterilmiştir.



Şekil 5.2. Komut Formatı

Komut sayısı 11 adet olduğundan komutun **opcode** kısmı için 4 bit ayrılmıştır. **D** ve **S** kısımları hedef ve kaynak saklayıcıyı göstermektedir. Her biri 4 bitten oluşmuştur. **Immediate** kısmı 16 bitlik ivedi değeri temsil etmektedir. **Ofset** ise 16 bitlik, Load ve Store komutları için gerekli olan ofset adresidir.

Bu bilgilerden faydalanarak gölge tutucu yığınların bit uzunlukları hesaplanacaktır. Mimarideki komut alma (F), komut kod çözme (D), ALU1, ALU2, LD / ST, MUL / DIV kat ve fonksiyonel birimlerdeki her tutucu için bir gölge tutucu yığın kullanmak gerekir. 32 bitlik bir komut, komut alma

katında 32 bitlik bir tutucuya ihtiyaç duyacaktır. Kod çözme katında ise 4 bitlik opcode çıkartılır ve geri kalan 28 bit tutucuya gönderilir. Tüm fonksiyonel birimlere; hedef - kaynak saklayıcıları, hedef saklayıcısı - immediate değer, ofset - hedef saklayıcı ikililerinden biri gönderilecektir. Bu ikililerin en uzunu 20 bittir ve tutucunun uzunluğu 20 bitlik olmalıdır. Buna göre yığın büyüklüğü 1 olan Gölge Tutucu Yığının (GTY) toplam bit sayısı :

	F	D	ALU1	ALU2	LD/ST	MUL/DIV	
GTY =	2*32	2*28	20	20	2*20	8*20	= 360 bit.

Yığın büyüklüğüne bağlı olarak GTY yönteminin donanım maliyeti Tablo 5.3.' de gösterilmektedir. Yığın büyüklüğü, GTY yönteminin donanım maliyetini büyük ölçüde belirlemektedir. Yığın büyüklüğü 1 olduğunda donanım maliyeti, hem YST hem de TD yöntemlerinden daha az bulunmaktadır. Yığın uzunluğu, tamamen mimarinin izin vereceği içiçe kesmelerin sayısına bağlıdır. İşlemcilerde dış kesmeler maskelenebileceğinden en kötü durumda iki iç kesmenin içiçe olacağı düşünülerek , yığın büyüklüğü 2 olarak seçilmiştir. Donanım maliyeti açısından önerdiğimiz gölge tutucu yığın yöntemi , yığın büyüklüğü 2 için, diğer yöntemlerin bit sayısına yakın olmasına rağmen bit sayısı fazladır. (140 ila 210 bit) Fakat performans olarak bakıldığında önerdiğimiz yöntem diğer yöntemlere göre daha verimli ve daha hızlıdır.

Tablo 5.3. Gölge Tutucu Yığın Donanım Maliyeti

Yığın Büyüklüğü	KDT	GTY	Donanım Maliyeti
1	456	+ 1*360	816
2	456	+ 2*360	1176
3	456	+ 3*360	1536
4	456	+ 4*360	1896
5	456	+ 5*360	2256

VI. SONUÇ

Yapılan bu çalışmada süperskaler mimari yapıları ayrıntılı olarak incelendi ve süperskaler işlemcilerde kesme ve aykırı durum problemi için yeni bir çözüm modeli önerildi. Öncelikle skaler işlemcilerde kesme problemi ve getirilen çözümler değerlendirildi ve sonra süperskaler işlemcilerde kesme ve aykırı durum problemleri üzerinde duruldu. Süperskaler işlemcilerde kesme problemine çözüm olarak getirilmiş yöntemler kesmeleri, precise kesme modeline göre kotarmaktadır.

Süperskaler işlemcilerde kesme problemine imprecise bir yöntem geliştirerek bir çözüm bulunabileceği düşünüldü. **Gölge Tutucu Yığınlar** adlı imprecise bir yöntem ortaya atıldı. Önerilen yeni bu yöntem; yeniden sıralama ve tarih dosyası yöntemleriyle performans ve maliyet olarak karşılaştırıldı. Her yöntem çeşitli benchmark programlarıyla denenerek performansları makina çevrim süresi olarak belirlendi. Elde edilen sonuçlara göre önerdiğimiz yeni yöntemin ötekilere kıyasla performans üstünlüğü sağladığı görüldü. Performans analizi sonuçlarına göre gölge tutucu yığınlar yöntemi bazı benchmark programlar için bariz üstünlük sağlamakta, bazı benchmarklarda ise performans diğer yöntemlerle hemen hemen aynı olmaktadır. Bunun sebebi, yöntemin performans olarak üstün gözüktüğü benchmark programlardaki komutların diğer benchmark programlara göre daha fazla sırasız olarak yayınlanabilmesidir. Sonuç olarak, benchmark programdaki sırasız yayınlanacak komutların sayısı ne kadar çoksa, gölge tutucu yığınlar yönteminin performansı o derece artacaktır; bu da doğal olarak süperskaler işlemcinin performansını arttıracaktır.

Maliyet analizi ise yöntemlerin kullandıkları donanımlara göre yapılmıştır. Her yöntem için kesme kotarma işlemi yapılırken kullanılan donanımların bit olarak büyüklükleri hesaplandı. Önerdiğimiz gölge tutucu yığın yönteminin donanım bit büyüklüğü, seçilecek olan yığın uzunluğuna bağlıdır. Yığın uzunluğu ise mimarinin izin verdiği içiçe kesme sayısına bağlıdır. Yapılan incelemede önerilen yeni yöntemde yığın uzunluğu iki için, kullandığı donanım miktarının, diğer yöntemlerin kullandıkları donanım miktarından çok fazla olmadığı tespit edilmiştir. İçiçe oluşabilecek kesmeleri maskeleyebilmek işlemci içinde mümkün olabilmektedir. Bu özellikten faydalanarak gölge tutucu yığın uzunluğu optimum bir değere getirilebilir. Eğer oluşabilecek içiçe kesmeleri gözönüne alarak , yığın uzunluğu ikiden fazla bir değerde seçilirse, donanım maliyeti artacaktır. Yöntemin donanım maliyeti yığın uzunluğuyla lineer olarak arttığından sisteme aşırı bir yük getirmeyecektir. Abartılı donanım maliyeti olmaması ve sağladığı performans artışı ile, gölge tutucu yığınlar yönteminin süperskaler bir işlemcide imprecise kesme kotarma işlemini başarıyla yapabileceği sonucuna varıldı.

KAYNAKLAR

1- Acosta , R.D. , Kjelstrup, J. , and Torng , H.C. , 1986. An Instruction Issue Approach to Enchanting Performance in Multiple Functional Unit Processors, IEEE Trans. on Computers, Vol. C-35, No.9, September.

2- Alsrup, M. , 1990. Motorola' s 88000 Family Architecture, IEEE Micro, April.

3- BYTE, 1987. High-Tech Horsepower, July.

4- Diefendorff, K. , and Allen, M. , 1992. Organization of the Motorola 88110 Superscalar RISC Microprocessor, IEEE Micro, April.

5- Grohoski, G. F. , 1990. Machine Organization of the IBM RISC System / 6000 Processor, IBM J. Res. DEVELOP., Vol. 34, No.1, January.

6- Hwang, K. , 1993. Advanced Computer Architecture. McGraw-Hill International Editions.

7- Hwu, M.W. , and Patt, Y. N. , 1987. Checkpointing Repair for Out-of-Order Execution Machines, IEEE Trans. on Computers, Vol. C-36, No.12, December, pp. 1515-1522.

8- Intel, 1989. Intel I486 Microprocessors, Intel Coporation, April.

9- Johnson, M. , 1991. Superscalar Microprocessors Design, Prentice-Hall, Englewood Cliffs, New Jersey.

10- Jouppi, N.P. , and Wall, D. W. ,1989. Available Instruction- Level Parallelism for Superscalar and Superpipelined Machines, Proc. Int. Conf. Arch. Support for Prog. Lang. and OS.

11- Lee, J.K.F. , and Smith, A.J. , 1984. Branch Prediction Strategies and Branch Target Buffer Design, IEEE Computer, January.

12- Lilja, D. J. , 1988. Reducing the Branch Penalty in Pipelined Processors, IEEE Computer July.

13- Melear, C. , 1989. The Design of the 88000 RISC Family, IEEE Micro , April.

14- Smith, J.E. , and Pleszkun, A. R. , 1988. Implementing Precise Interrupts in Pipelined Processors, IEEE Trans. on Computers, Vol. 37, No.5 , May.

15- Sohi, G.S. , 1990. Instruction Issue Logic for High-Performance Interruptible Multiple Functional Unit, Pipelined Computers, IEEE Trans. on Computers, Vol.39, No.3, March.

16- Walker, W. , Cragon , H. G. , 1995. Interrupt Processing in Concurrent Processors, IEEE Micro, June.

17- Wang, C-J. , and Emnett, F. , 1993. Implementing Precise Interruptions in Pipelined RISC Processors, IEEE Micro, August.

EKLER

Bu bölümde C dilinde yazılmış olan simülasyon programlarının kaynak kodları bulunmaktadır. Kullanılan fibonecci, loop, matris çarpımı ve asal sayılar gibi benchmark programlarının yeniden sıralama tamponu, tarih dosyası ve gölge tutucu yığın yöntemleri için yazılmış olan kaynak kodları aşağıda verilmektedir. Her kaynak programı **bench.h** kütüphane dosyasında bulunan fonksiyonları kullanmaktadır. Kaynak kodlarının benchmark programlara ve yöntemlere göre dağılımı şu şekildedir :

1- Fibonecci Programı :
- Yeniden Sıralama Tampon Yöntemi İçin
- Tarih Dosyası Yöntemi İçin
- Gölge Tutucu Yığın Yöntemi İçin

2- Loop Programı :
- Yeniden Sıralama Tampon Yöntemi İçin
- Tarih Dosyası Yöntemi İçin
- Gölge Tutucu Yığın Yöntemi İçin

3- Matris Çarpımı Programı:
- Yeniden Sıralama Tampon Yöntemi İçin
- Tarih Dosyası Yöntemi İçin
- Gölge Tutucu Yığın Yöntemi İçin

4- Asal Sayılar Programı:
- Yeniden Sıralama Tampon Yöntemi İçin
- Tarih Dosyası Yöntemi İçin
- Gölge Tutucu Yığın Yöntemi İçin

/* **Bench.h** kütüphane dosyasında bulunan fonksiyonlar yazılan tüm kaynak programları tarafından kullanılmaktadır. **sembol_tablosu** fonksiyonu makina kodunda yazılmış olan programı tokenlarına ayırarak bir tabloya yazar. Mimari modeldeki her komut için bir fonksiyon oluşturulmuştur. Bu fonksiyonlar **exec** fonksiyonu ile çağrılarak programdaki ilgili komutun çevrim zamanı bulunarak sembol tablosuna yazılır. Benchmark programının kullanıldığı yöntemde kaç çevrim süresi aldığını **calculate** fonksiyonu hesaplar. */

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>


typedef struct {
   char tab[6][7];
}SYMBOL;

/* DEFINES FOR COMMANDS  */

#define CMD_ADD          0
#define CMD_SUB          1
#define CMD_HLT          2
#define CMD_ST           3
#define CMD_LD           4
#define CMD_DIV          5
#define CMD_NOP          6
#define CMD_LOOP         7
#define CMD_MUL          8
#define CMD_MOV          9
#define CMD_BR           10

int exec_cycle;

/* A STRING TABLE WHICH CORRESPONDS WITH THE ABOVE DEFINITIONS FOR
COMMANDS */

char *ins[]={"ADD", "SUB","HLT","ST", "LD","DIV","NOP","LOOP","MUL","MOV", "BR" };

int komut(char *s)
{
        int i=0;
        while ((i<11) && (strcmp(s,ins[i])!=0))  i++;
        if (i==11) return -1;
        else return i;
}

int mnem(char *st)
{
        int i,f;
        for(i=0;i<11;i++) {
                if (strcmp(ins[i],st)==0) {
                        f=1;
                        break;
```

```
                }
                else f=0;
        }
        return f;
}

int cmd_add(void)
{
 exec_cycle=1;
 return exec_cycle;
}

int cmd_sub(void)
{
 exec_cycle=1;
 return exec_cycle;
}

int cmd_hlt(void)  /* should halt the program */
{
 exec_cycle=1;
 return exec_cycle;
}

int cmd_st(void)
{
  exec_cycle=2;
  return exec_cycle;
}
int cmd_ld(void)
{
  exec_cycle=2;
  return exec_cycle;
}

int cmd_div(void)
{
  exec_cycle=8;
  return exec_cycle;
}

int cmd_nop(void)
{
  exec_cycle=1;
  return exec_cycle;
}

int cmd_loop(void)
{
   exec_cycle=1;
   return exec_cycle;
}

int cmd_mul(void)
```

```
{
   exec_cycle=8;
   return exec_cycle;
}
int cmd_mov(void)
{
 exec_cycle=1;
 return exec_cycle;
}

int cmd_br(void)
{
 exec_cycle=1;
 return exec_cycle;
}

void exec(int l,SYMBOL sym[])
{
        int te,pc=0,i,sayi;
        char s[7],str[7];

        int (*cmd_ptr[11])(void);

        cmd_ptr[CMD_ADD ] = cmd_add;
        cmd_ptr[CMD_SUB ] = cmd_sub;
        cmd_ptr[CMD_HLT ] = cmd_hlt;
        cmd_ptr[CMD_ST]  = cmd_st;
        cmd_ptr[CMD_LD]  = cmd_ld;
        cmd_ptr[CMD_DIV]  = cmd_div;
        cmd_ptr[CMD_NOP]  = cmd_nop;
        cmd_ptr[CMD_LOOP] = cmd_loop;
        cmd_ptr[CMD_MUL]  = cmd_mul;
        cmd_ptr[CMD_MOV]  = cmd_mov;
        cmd_ptr[CMD_BR]  = cmd_br;


        while (pc<l) {
                for(i=0;i<7;i++) s[i]='\0';
                strncpy(s,sym[pc].tab[1],6);
                te = komut(s);
                sayi=(*cmd_ptr[te])();
                for(i=0;i<7;i++) str[i]='\0';
                itoa(sayi,str,10);
                strncpy(sym[pc].tab[5],str,6);
                pc++;
        }
}

int sembol_tablosu(FILE *prgfp,SYMBOL sym[])
{

        int l,co,i;
        char str[7],*p;
        int c;
```

```
        clrscr();

        l=0;
        do {
                co=0;
                c=' ';
                while (c!='\n') {
                        c=fgetc(prgfp);
                        if (c!=' ') {
                                for(i=0;i<7;i++) str[i]='\0';
                                p=str;
                                while ((c!=' ') && (c!=',') && (c!='\n')) {
                                        *p++= (char) c;
                                        c=fgetc(prgfp);
                                }
                                if (co==0) {
                                        if (mnem(str)==1) {
                                                strncpy(sym[l].tab[1],str,6);
                                                co=1;
                                        }
                                        else {
                                        strncpy(sym[l].tab[0],str,6);
                                        }
                                }
                                else strncpy(sym[l].tab[co],str,6);
                                co++;
                        }
                }
                l++;
        } while (strcmp(sym[l-1].tab[1],"HLT")!=0);

        fclose(prgfp);
        return l;
}

int calculate(int n,int total_cycles,int cycles[])
{
  int i;

  for(i=0;i<n;i++) {
    if (cycles[i]>0) (cycles[i])--;
  }
  total_cycles++;
  return total_cycles;
}
```

```
/* Yeniden Sıralama Tamponu için Fibonecci Sayıları Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}REORDER_TABLE;

# define LOOP_NUMBER            15
# define LOOP_DIFFERENCE   7
# define REORDER_SIZE      10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
REORDER_TABLE reorder[REORDER_SIZE];
int total_cycles,po;
int cycles[1000];

void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) reorder[i].exception=1;
   else reorder[i].exception=0;
}

void initialize()   /* Initializes the reorder buffer table */
{
 int i;

 for(i=0;i<REORDER_SIZE;i++) {
   reorder[i].tag=0;
   reorder[i].pc=-1;
   reorder[i].valid=-1;
   reorder[i].dirty=-1;
   reorder[i].cycle=-1;
 }
}

void inicycles()
{
 int i;

 for(i=0;i<1000;i++) cycles[i]=0;
}
```

```
void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
  while (l>LOOP_NUMBER) l-=LOOP_DIFFERENCE;
  return l;
}
 /* Yeniden Sıralama Yönteminin Simülasyonunu yapar. */
void sim()
{
  FILE *fp;
  int i,j,max,k,l,x0,t,x1,flag,temp,instruction_no;
  char c,str[7],*p,*issue[2];

  initialize();
  total_cycles=0;
  po=0;
  fp=fopen("FIBREISS.dat","r");
  flag=0;
  k=0;
  do {
   c=' ';
   c=fgetc(fp);
   if (c!='.') {
    for(j=0;j<2;j++) strcpy(issue[j],"");
    for(j=0;j<2;j++) {
          for(i=0;i<7;i++) str[i]='\0';
          p=str;
                if (c==' ') c=fgetc(fp);
                while ((c!=' ') && (c!='\n')) {
                      *p++= (char) c;
                      c=fgetc(fp);
                }
         strncpy(issue[j],str,6);
    }  /* end of for  */

    if (flag==1) {
        for(l=0;l<REORDER_SIZE;l++) {
          if (reorder[l].cycle>0) (reorder[l].cycle)--;
          if (reorder[l].cycle==0) reorder[l].valid=1;
        }
        if (reorder[po].valid==1) {
         if (reorder[po].exception==1) {
           fclose(fp);
           fp=fopen("FIBREEXC.DAT","r");
           introutine(reorder[po].pc);
           initialize();
           inicycles();
           k=0;
         }
```

```
            else {
             while ((reorder[po].valid==1) && (reorder[po].dirty==0)) {
              reorder[po].dirty=1;
              po++;
              if (po>REORDER_SIZE-1) po=po%REORDER_SIZE;
             }
            }
           }
           total_cycles=calculate(k,total_cycles,cycles);
      }
      if (strcmp(issue[0],"NOP")!=0) {
           x0=atoi(issue[0]);
           x1=x0-1;
           instruction_no=generate_pc(x0)-1;
           t=instruction_no;
           temp=atoi(sym[t].tab[5]);
           cycles[k++]=temp;
           if (x1>REORDER_SIZE-1) x1%=REORDER_SIZE;
           iniexception(instruction_no,x1);
           reorder[x1].pc=instruction_no;
           reorder[x1].valid=0;
           reorder[x1].dirty=0;
           reorder[x1].cycle=temp;
           reorder[x1].tag=x0;
      }
      else  cycles[k++]=1;
      if (strcmp(issue[1],"NOP")!=0) {
           x0=atoi(issue[1]);
       x1=x0-1;
           instruction_no=generate_pc(x0)-1;
           t=instruction_no;
           temp=atoi(sym[t].tab[5]);
           cycles[k++]=temp;
           if (x1>REORDER_SIZE-1) x1%=REORDER_SIZE;
           iniexception(instruction_no,x1);
           reorder[x1].pc=instruction_no;
           reorder[x1].valid=0;
           reorder[x1].dirty=0;
           reorder[x1].cycle=temp;
           reorder[x1].tag=x0;
      }
      else  cycles[k++]=1;
      flag=1;
     }
    } while (c!='.');
    fclose(fp);

    max=cycles[0];
    for (i=1;i<k;i++) {
    if (cycles[i]>max) max=cycles[i];
    }
    total_cycles+=max;
    printf("TOTAL CYCLES OF FIBONECCI PROGRAM WITH REORDER BUFFER : %d
",total_cycles);
```

```
}

int main()
{
        int i,j,l;
        FILE *prgfp;

        prgfp = fopen("FIBONEC.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
                exit(1);
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strcpy(sym[i].tab[j],"");
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

```
/* Tarih Dosyası için Fibonecci Sayıları Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}HISTORY_TABLE;

# define LOOP_NUMBER     15
# define LOOP_DIFFERENCE   7
# define BUFSIZE           10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
HISTORY_TABLE his[BUFSIZE];
/*int exec_cycle;*/
int total_cycles;
int cycles[1000],po;


void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) his[i].exception=1;
   else his[i].exception=0;
}

void initialize()  /* Initializes the history table */
{
 int i;

 for(i=0;i<BUFSIZE;i++) {
   his[i].tag=0;
   his[i].pc=-1;
   his[i].valid=-1;
   his[i].dirty=-1;
   his[i].cycle=-1;
 }
}

void inicycles()
{
 int i;
```

```
 for(i=0;i<1000;i++) cycles[i]=0;
}

void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
 while (l>LOOP_NUMBER) l-=LOOP_DIFFERENCE;
 return l;
}

void sim()
{
  FILE *fp;
  int t,i,j,k,l,x0,flag,temp,instruction_no,x1,max;
  char c,str[7],*p,*issue[2];

  initialize();
  total_cycles=0;
  po=0;
  fp=fopen("FIBISSUE.dat","r");
  flag=0;
  k=0;
  do {
   c=' ';
   c=fgetc(fp);
   if (c!='.') {
    for(j=0;j<2;j++) strncpy(issue[j],"",5);
    for(j=0;j<2;j++) {
          for(i=0;i<7;i++) str[i]='\0';
          p=str;
                if (c==' ') c=fgetc(fp);
                while ((c!=' ') && (c!='\n')) {
                      *p++= (char) c;
                      c=fgetc(fp);
                }
         strncpy(issue[j],str,6);
    }  /* end of for  */

    if (flag==1) {
         for(l=0;l<BUFSIZE;l++) {
          if (his[l].cycle>0) (his[l].cycle)--;
          if (his[l].cycle==0) his[l].valid=1;
         }
         if (his[po].valid==1) {
          if (his[po].exception==1) {
            fclose(fp);
            fp=fopen("FIBEXC.DAT","r");
            introutine(his[po].pc);
            initialize();
            inicycles();
```

```
            k=0;
           }
           else {
            while ((his[po].valid==1) && (his[po].dirty==0)) {
             his[po].dirty=1;
             po++;
             if (po>BUFSIZE-1) po=po%BUFSIZE;
            }
           }
          }

          total_cycles=calculate(k,total_cycles,cycles);
    }
    if (strcmp(issue[0],"NOP")!=0) {
          x0=atoi(issue[0]);      /* Instruction number  */
          x1=x0-1;
          instruction_no=generate_pc(x0)-1;
          t=instruction_no;
          temp=atoi(sym[t].tab[5]);
          cycles[k++]=temp;
          if (x1>BUFSIZE-1) x1%=BUFSIZE;
          iniexception(instruction_no,x1);
          his[x1].pc=instruction_no;
          his[x1].valid=0;
          his[x1].dirty=0;
          his[x1].cycle=temp;
          his[x1].tag=x0;
    }
    else  cycles[k++]=1;
    if (strcmp(issue[1],"NOP")!=0) {
          x0=atoi(issue[1]);
          x1=x0-1;
          instruction_no=generate_pc(x0)-1;
          t=instruction_no;
          temp=atoi(sym[t].tab[5]);
          cycles[k++]=temp;
          if (x1>BUFSIZE-1) x1%=BUFSIZE;
          iniexception(instruction_no,x1);
          his[x1].pc=instruction_no;
          his[x1].valid=0;
          his[x1].dirty=0;
          his[x1].cycle=temp;
          his[x1].tag=x0;
    }
    else  cycles[k++]=1;
    flag=1;
  }
} while (c!='.');
fclose(fp);

max=cycles[0];
for (i=1;i<k;i++) {
  if (cycles[i]>max) max=cycles[i];
}
```

```
  total_cycles+=max;
  printf("TOTAL CYCLES OF FIBONECCI PROGRAM WITH HISTORY FILE : %d ",total_cycles);
}

int main()
{
        int i,j,l;
        FILE *prgfp;

        prgfp = fopen("FIBONEC.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
                exit(1);
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strncpy(sym[i].tab[j],"",6);
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

```
/* Gölge Tutucu  Yığınlar için Fibonecci Sayıları Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        int pc;
        int pipe_delay;
        int valid;
} IST;

# define  IST_SIZE      10
# define LOOP_NUMBER    15
# define LOOP_DIFFERENCE  7


/* GLOBAL DEGISKENLER */

SYMBOL sym[250];        /* table to keep symbol information */
IST  ist[IST_SIZE];
int total_cycles;
int cycles[1000];

int generate_pc(int i)
{
  while (i>LOOP_NUMBER) i-=LOOP_DIFFERENCE;
  return i;
}

int search_ist()
{
  int i=0;

  while ((i<IST_SIZE) && (ist[i].valid!=1)) i++;
  if (i>=IST_SIZE) return -1;
  else return i;
}

void ini_ist()
{
  int i;

  for(i=0;i<IST_SIZE;i++) {
     ist[i].pc=-1;
     ist[i].pipe_delay=-1;
     ist[i].valid=1;
  }
}

void update_ist()
```

```
{
  int i,te;

  for(i=0;i<IST_SIZE;i++) {
     te=ist[i].pipe_delay;
     if (te>0) {
           te--;
           ist[i].pipe_delay=te;
           if (te==0) ist[i].valid=1;
     }
  }
}

void sim()
{
  FILE *fp;
  int i,max,j,k,x0,flag,instr_po,temp,entry;
  char c,str[7],*p,*issue[2];

  total_cycles=0;
  ini_ist();
  fp=fopen("FIBISSUE.dat","r");
  flag=0;
  k=0;
  do {
   c=' ';
   c=fgetc(fp);
   if (c!='.') {
    for(j=0;j<2;j++) strcpy(issue[j],"");
    for(j=0;j<2;j++) {
      for(i=0;i<7;i++) str[i]='\0';
          p=str;
                if (c==' ') c=fgetc(fp);
                while ((c!=' ') && (c!='\n')) {
                      *p++= (char) c;
                      c=fgetc(fp);
                }
          strncpy(issue[j],str,6);
    }  /* end of for  */
    if (flag==1) {
         total_cycles=calculate(k,total_cycles,cycles);
         update_ist();
    }
    if (strcmp(issue[0],"NOP")!=0) {
         x0=atoi(issue[0]);
         instr_po=generate_pc(x0)-1;
         temp=atoi(sym[instr_po].tab[5]);
         cycles[k++]=temp;
         entry=search_ist();
         if (entry!=-1) {
           ist[entry].pc=instr_po;
           ist[entry].pipe_delay=temp;
           ist[entry].valid=0;
         }
```

```
      }
      else  cycles[k++]=1;
      if (strcmp(issue[1],"NOP")!=0) {
           x0=atoi(issue[1]);
           instr_po=generate_pc(x0)-1;
           temp=atoi(sym[instr_po].tab[5]);
           cycles[k++]=temp;
           entry=search_ist();
           if (entry!=-1) {
             ist[entry].pc=instr_po;
             ist[entry].pipe_delay=temp;
             ist[entry].valid=0;
           }
      }
      else  cycles[k++]=1;
      flag=1;
     }
   } while (c!='.');
   fclose(fp);
   max=cycles[0];
   for (i=1;i<k;i++) if (cycles[i]>max) max=cycles[i];
   total_cycles+=max;
   printf("TOTAL CYCLES OF FIBONECCI PROGRAM WITH SHADOW LATCH STACK : %d
",total_cycles);
}

int main()
{
          int i,j,l;
          FILE *prgfp;

          prgfp = fopen("FIBONEC.SRC", "r");
          if (prgfp == NULL) {
                    printf(" Can't open %s file for input\n");
                    exit(1);
          }

          clrscr();

          for(i=0;i<250;i++) {
             for(j=0;j<6;j++) strcpy(sym[i].tab[j],"");
          }

          l=sembol_tablosu(prgfp,sym);
          exec(l,sym);
          sim();
          return 0;
}
```

```
/*  Yeniden Sıralama Tamponu için Loop  Programı  */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        char komut[6];
} ISSUE;

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}REORDER_TABLE;

# define  LOOP_NUMBER      21
# define  LOOP_DIFFERENCE   16
# define  BUFSIZE           10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
REORDER_TABLE reorder[BUFSIZE];
ISSUE issue[2];
int total_cycles,k;
int cycles[3000];
int current_tag;

void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) reorder[i].exception=1;
   else reorder[i].exception=0;
}

void initialize()  /* Initializes the history table */
{
 int i;

 for(i=0;i<BUFSIZE;i++) {
   reorder[i].tag=-1;
   reorder[i].exception=-1;
   reorder[i].pc=-1;
   reorder[i].valid=-1;
   reorder[i].dirty=-1;
   reorder[i].cycle=-1;
 }
}
```

```
void inicycles()
{
 int i;

 for(i=0;i<3000;i++) cycles[i]=0;
}

void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
 while (l>LOOP_NUMBER) l-=LOOP_DIFFERENCE;
 return l;
}

int make_issue(FILE *fp)
{
 int i,j;
 char c,*p,str[7];

 c=' ';
    c=fgetc(fp);
    if (c!='.') {
     for(j=0;j<2;j++) strncpy(issue[j].komut,"",5);
     for(j=0;j<2;j++) {
            for(i=0;i<7;i++) str[i]='\0';
            p=str;
                    if (c==' ') c=fgetc(fp);
                    while ((c!=' ') && (c!='\n')) {
                            *p++= (char) c;
                            c=fgetc(fp);
                    }
           strncpy(issue[j].komut,str,6);
     }  /* end of for  */
     return 1;
   }
   else return 0;
}

int find_tag()
{
 int l;

 l=0;
 while ((l<BUFSIZE) && (reorder[l].tag!=current_tag)) l++;
 if (l>=BUFSIZE) return -1;
 else return l;
}

int suspend(FILE *fp)
```

```
{
 int l,j,bufno;

do {
 for(l=0;l<BUFSIZE;l++) {
           if (reorder[l].cycle>0) (reorder[l].cycle)--;
           if (reorder[l].cycle==0) reorder[l].valid=1;
 }
  bufno=find_tag();
  if (bufno!=-1) {
         if (reorder[bufno].valid==1) {
          if (reorder[bufno].exception==1) {
            fclose(fp);
            fp=fopen("LOOPREXC.DAT","r");
            introutine(reorder[bufno].pc);
            initialize();
            inicycles();
            k=0;
            current_tag++;
            return -1;
          }
          else {
          while ((reorder[bufno].valid==1)  && (bufno!=-1)) {
           reorder[bufno].dirty=1;
           current_tag++;
           bufno=find_tag();
          }
         }
        }
   }
 total_cycles++;
 j=0;
 while ((j<BUFSIZE) && (reorder[j].dirty==0)) j++;
 if (j<BUFSIZE) return j;
} while (1);
}

void sim()
{
 FILE *fp;
 int max,t,bufno,i,l,c,x0,flag,temp,instruction_no,x1,test;

 initialize();
 l=0;
 current_tag=1;
 total_cycles=0;
 fp=fopen("LOOPREISS.dat","r");
 flag=0;
 do {
  test=make_issue(fp);
  if (test!=0) {
   if (flag==1) {
        for(c=0;c<BUFSIZE;c++) {
         if (reorder[c].cycle>0) (reorder[c].cycle)--;
```

```
        if (reorder[c].cycle==0) reorder[c].valid=1;
       }

  bufno=find_tag();
  if (bufno!=-1) {
       if (reorder[bufno].valid==1) {
        if (reorder[bufno].exception==1) {
          fclose(fp);
          fp=fopen("LOOPREXC.DAT","r");
          introutine(reorder[bufno].pc);
          initialize();
          inicycles();
          k=0;
          current_tag++;
          test=make_issue(fp);
        }
        else {
         while ((reorder[bufno].valid==1)  && (bufno!=-1)) {
          reorder[bufno].dirty=1;
          current_tag++;
          bufno=find_tag();
         }
        }
       }
 }
       total_cycles=calculate(k,total_cycles,cycles);
}
  if (strcmp(issue[0].komut,"NOP")!=0) {
       x0=atoi(issue[0].komut);      /* Instruction number  */
       x1=x0-1;
       instruction_no=generate_pc(x0)-1;
       t=instruction_no;
       temp=atoi(sym[t].tab[5]);
       cycles[k++]=temp;
       l=0;
       while ((l<BUFSIZE) && (reorder[l].dirty==0)) l++;
       if (l>=BUFSIZE) {
        printf("Reorder buffer line match\n");
        l=suspend(fp);
       }
       if (l!=-1) {
        iniexception(instruction_no,l);
        reorder[l].pc=instruction_no;
        reorder[l].valid=0;
        reorder[l].dirty=0;
        reorder[l].cycle=temp;
        reorder[l].tag=x0;
       }
  }
  else  cycles[k++]=1;
 if (l!=-1) {
  if (strcmp(issue[1].komut,"NOP")!=0) {
       x0=atoi(issue[1].komut);
       x1=x0-1;
```

```
            instruction_no=generate_pc(x0)-1;
            t=instruction_no;
            temp=atoi(sym[t].tab[5]);
            cycles[k++]=temp;
            l=0;
            while ((l<BUFSIZE) && (reorder[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("Reorder buffer line match\n");
              l=suspend(fp);
            }
            iniexception(instruction_no,l);
            reorder[l].pc=instruction_no;
            reorder[l].valid=0;
            reorder[l].dirty=0;
            reorder[l].cycle=temp;
            reorder[l].tag=x0;
        }
        else  cycles[k++]=1;
        flag=1;
      }
     }
    } while (test!=0);
    fclose(fp);
    max=cycles[0];
    for (i=1;i<k;i++) {
    if (cycles[i]>max) max=cycles[i];
    }
    total_cycles+=max;
    printf("TOTAL CYCLES OF LOOP PROGRAM WITH REORDER BUFFER : %d ",total_cycles);
    getch();
}

int main()
{
            int i,j,l;
            FILE *prgfp;

            prgfp = fopen("LOOP.SRC", "r");
            if (prgfp == NULL) {
                        printf(" Can't open %s file for input\n");
                        exit(1);
            }

            clrscr();

            for(i=0;i<250;i++) {
               for(j=0;j<6;j++) strncpy(sym[i].tab[j],"",6);
            }

            l=sembol_tablosu(prgfp,sym);
            exec(l,sym);
            sim();
            return 0;
}
```

```
/* Tarih Dosyası için Loop Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        char komut[6];
} ISSUE;

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}HISTORY_TABLE;

# define LOOP_NUMBER      21
# define LOOP_DIFFERENCE   16
# define BUFSIZE           10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
HISTORY_TABLE his[BUFSIZE];
ISSUE issue[2];
int total_cycles,k;
int cycles[2000];
int current_tag;

void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) his[i].exception=1;
   else his[i].exception=0;
}

void initialize()  /* Initializes the history table */
{
 int i;

 for(i=0;i<BUFSIZE;i++) {
   his[i].tag=-1;
   his[i].exception=-1;
   his[i].pc=-1;
   his[i].valid=-1;
   his[i].dirty=-1;
   his[i].cycle=-1;
 }
}
```

```
void inicycles()
{
 int i;

 for(i=0;i<2000;i++) cycles[i]=0;
}

void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
 while (l>LOOP_NUMBER) l-=LOOP_DIFFERENCE;
 return l;
}

int make_issue(FILE *fp)
{
 int i,j;
 char c,*p,str[7];

 c=' ';
    c=fgetc(fp);
    if (c!='.') {
     for(j=0;j<2;j++) strncpy(issue[j].komut,"",5);
     for(j=0;j<2;j++) {
            for(i=0;i<7;i++) str[i]='\0';
            p=str;
                    if (c==' ') c=fgetc(fp);
                    while ((c!=' ') && (c!='\n')) {
                            *p++= (char) c;
                            c=fgetc(fp);
                    }
           strncpy(issue[j].komut,str,6);
     }  /* end of for  */
     return 1;
   }
   else return 0;
}

int find_tag()
{
 int l;

 l=0;
 while ((l<BUFSIZE) && (his[l].tag!=current_tag)) l++;
 if (l>=BUFSIZE) return -1;
 else return l;
}

int suspend(FILE *fp)
```

```
{
 int l,j,bufno;

do {
 for(l=0;l<BUFSIZE;l++) {
            if (his[l].cycle>0) (his[l].cycle)--;
            if (his[l].cycle==0) his[l].valid=1;
 }
  bufno=find_tag();
  if (bufno!=-1) {
          if (his[bufno].valid==1) {
           if (his[bufno].exception==1) {
             fclose(fp);
             fp=fopen("LOOPEXC.DAT","r");
             introutine(his[bufno].pc);
             initialize();
             inicycles();
             k=0;
             current_tag++;
             return -1;
           }
           else {
           while ((his[bufno].valid==1)  && (bufno!=-1)) {
            his[bufno].dirty=1;
            current_tag++;
            bufno=find_tag();
           }
          }
         }
   }
  total_cycles++;
  j=0;
  while ((j<BUFSIZE) && (his[j].dirty==0)) j++;
  if (j<BUFSIZE) return j;
 } while (1);
}

void sim()
{
  FILE *fp;
  int max,t,bufno,i,l,c,x0,flag,temp,instruction_no,x1,test;

  initialize();
  l=0;
  current_tag=1;
  total_cycles=0;
  fp=fopen("LOOPISS.dat","r");
  flag=0;
  do {
   test=make_issue(fp);
   if (test!=0) {
    if (flag==1) {
          for(c=0;c<BUFSIZE;c++) {
           if (his[c].cycle>0) (his[c].cycle)--;
```

```
          if (his[c].cycle==0) his[c].valid=1;
         }

   bufno=find_tag();
   if (bufno!=-1) {
        if (his[bufno].valid==1) {
         if (his[bufno].exception==1) {
           fclose(fp);
           fp=fopen("LOOPEXC.DAT","r");
           introutine(his[bufno].pc);
           initialize();
           inicycles();
           k=0;
           current_tag++;
           test=make_issue(fp);
         }
         else {
          while ((his[bufno].valid==1)  && (bufno!=-1)) {
           his[bufno].dirty=1;
           current_tag++;
           bufno=find_tag();
          }
         }
        }
  }
        total_cycles=calculate(k,total_cycles,cycles);
}
   if (strcmp(issue[0].komut,"NOP")!=0) {
        x0=atoi(issue[0].komut);     /* Instruction number  */
        x1=x0-1;
        instruction_no=generate_pc(x0)-1;
        t=instruction_no;
        temp=atoi(sym[t].tab[5]);
        cycles[k++]=temp;
        l=0;
        while ((l<BUFSIZE) && (his[l].dirty==0)) l++;
        if (l>=BUFSIZE) {
         printf("History buffer line match\n");
         l=suspend(fp);
        }
        if (l!=-1) {
         iniexception(instruction_no,l);
         his[l].pc=instruction_no;
         his[l].valid=0;
         his[l].dirty=0;
         his[l].cycle=temp;
         his[l].tag=x0;
        }
  }
  else  cycles[k++]=1;
 if (l!=-1) {
  if (strcmp(issue[1].komut,"NOP")!=0) {
       x0=atoi(issue[1].komut);
       x1=x0-1;
```

```
            instruction_no=generate_pc(x0)-1;
            t=instruction_no;
            temp=atoi(sym[t].tab[5]);
            cycles[k++]=temp;
            l=0;
            while ((l<BUFSIZE) && (his[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("History buffer line match\n");
              l=suspend(fp);
            }
            iniexception(instruction_no,l);
            his[l].pc=instruction_no;
            his[l].valid=0;
            his[l].dirty=0;
            his[l].cycle=temp;
            his[l].tag=x0;
        }
        else  cycles[k++]=1;
        flag=1;
       }
      }
    } while (test!=0);
    fclose(fp);
    max=cycles[0];
    for (i=1;i<k;i++) {
    if (cycles[i]>max) max=cycles[i];
    }
    total_cycles+=max;
    printf("TOTAL CYCLES OF LOOP PROGRAM WITH HISTORY FILE : %d ",total_cycles);
    getch();
}

int main()
{
            int i,j,l;
            FILE *prgfp;

            prgfp = fopen("LOOP.SRC", "r");
            if (prgfp == NULL) {
                        printf(" Can't open %s file for input\n");
                        exit(1);
            }

            clrscr();

            for(i=0;i<250;i++) {
               for(j=0;j<6;j++) strncpy(sym[i].tab[j],"",6);
            }
            l=sembol_tablosu(prgfp,sym);
            exec(l,sym);
            sim();
            return 0;
}
```

```
/* Gölge Tutucu Yığınlar için Loop Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        int pc;
        int pipe_delay;
        int valid;
} IST;

# define  IST_SIZE      10
# define LOOP_NUMBER    21
# define LOOP_DIFFERENCE 16


/* GLOBAL DEGISKENLER */

SYMBOL sym[250];        /* table to keep symbol information */
IST  ist[IST_SIZE];
int total_cycles;
int cycles[2000];

int generate_pc(int i)
{
  while (i>LOOP_NUMBER) i-=LOOP_DIFFERENCE;
  return i;
}

int search_ist()
{
  int i=0;

  while ((i<IST_SIZE) && (ist[i].valid!=1)) i++;
  if (i>=IST_SIZE) return -1;
  else return i;
}

void ini_ist()
{
  int i;

  for(i=0;i<IST_SIZE;i++) {
     ist[i].pc=-1;
     ist[i].pipe_delay=-1;
     ist[i].valid=1;
  }
}

void update_ist()
```

```
{
  int i,te;

  for(i=0;i<IST_SIZE;i++) {
     te=ist[i].pipe_delay;
     if (te>0) {
           te--;
           ist[i].pipe_delay=te;
           if (te==0) ist[i].valid=1;
     }
  }
}

void sim()
{
  FILE *fp;
  int max,i,j,k,x0,flag,instr_po,temp,entry;
  char c,str[7],*p,*issue[2];

  total_cycles=0;
  ini_ist();
  fp=fopen("LOOPISS.dat","r");
  flag=0;
  k=0;
  do {
   c=' ';
   c=fgetc(fp);
   if (c!='.') {
    for(j=0;j<2;j++) strncpy(issue[j],"",6);
    for(j=0;j<2;j++) {
           for(i=0;i<7;i++) str[i]='\0';
           p=str;
                 if (c==' ') c=fgetc(fp);
                 while ((c!=' ') && (c!='\n')) {
                       *p++= (char) c;
                       c=fgetc(fp);
                 }
          strncpy(issue[j],str,6);
    }  /* end of for  */
    if (flag==1) {
         total_cycles=calculate(k,total_cycles,cycles);
         update_ist();
    }
    if (strcmp(issue[0],"NOP")!=0) {
         x0=atoi(issue[0]);
         instr_po=generate_pc(x0)-1;
         temp=atoi(sym[instr_po].tab[5]);
         cycles[k++]=temp;
         entry=search_ist();
         if (entry!=-1) {
           ist[entry].pc=instr_po;
           ist[entry].pipe_delay=temp;
           ist[entry].valid=0;
         }
```

```
      }
      else  cycles[k++]=1;
      if (strcmp(issue[1],"NOP")!=0) {
           x0=atoi(issue[1]);
           instr_po=generate_pc(x0)-1;
           temp=atoi(sym[instr_po].tab[5]);
           cycles[k++]=temp;
           entry=search_ist();
           if (entry!=-1) {
             ist[entry].pc=instr_po;
             ist[entry].pipe_delay=temp;
             ist[entry].valid=0;
           }
      }
      else  cycles[k++]=1;
      flag=1;
     }
   } while (c!='.');
   fclose(fp);
   max=cycles[0];
   for (i=1;i<k;i++) if (cycles[i]>max) max=cycles[i];
   total_cycles+=max;
   printf("TOTAL CYCLES OF LOOP PROGRAM WITH SHADOW LATCH STACK : %d
",total_cycles);
   getch();
}

int main()
{
          int i,j,l;
          FILE *prgfp;

          prgfp = fopen("LOOP.SRC", "r");
          if (prgfp == NULL) {
                    printf(" Can't open %s file for input\n");
                    exit(1);
          }

          clrscr();

          for(i=0;i<250;i++) {
            for(j=0;j<6;j++) strcpy(sym[i].tab[j],"");
          }

          l=sembol_tablosu(prgfp,sym);
          exec(l,sym);
          sim();
          return 0;
}
```

```
/* Yeniden Sıralama Tamponu için Matris Çarpımı Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        char komut[6];
} ISSUE;

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}REORDER_TABLE;

# define LOOP1     16
# define LOOP2     19
# define LOOP3     22
# define FIRSTLOOP  22    /* Address of the first loop */
# define SECONDLOOP 24
# define THIRDLOOP  26
# define BUFSIZE    10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
REORDER_TABLE reo[BUFSIZE];
ISSUE issue[2];
int total_cycles,k;
int cycles[2000];
int current_tag;
int l1=0,l2=0,l3=0;

void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) reo[i].exception=1;
   else reo[i].exception=0;
}

void initialize()  /* Initializes the history table */
{
 int i;

 for(i=0;i<BUFSIZE;i++) {
   reo[i].tag=-1;
   reo[i].exception=-1;
   reo[i].pc=-1;
```

```
    reo[i].valid=-1;
    reo[i].dirty=-1;
    reo[i].cycle=-1;
  }
}

void inicycles()
{
 int i;

 for(i=0;i<2000;i++) cycles[i]=0;
}

void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
 int i,m;

 m=l;
 if (m>FIRSTLOOP) {
   i=LOOP1*l1+LOOP2*l2+LOOP3*l3;
   m=m-i;
   if (m==FIRSTLOOP) l1++;
   else if (m==SECONDLOOP) l2++;
          else if (m==THIRDLOOP)  l3++;
   return m;
 }
 else {
   if (m==FIRSTLOOP) l1++;
   return m;
 }
}

int make_issue(FILE *fp)
{
 int i,j;
 char c,*p,str[7];

 c=' ';
   c=fgetc(fp);
   if (c!='.') {
     for(j=0;j<2;j++) strncpy(issue[j].komut,"",5);
     for(j=0;j<2;j++) {
          for(i=0;i<7;i++) str[i]='\0';
          p=str;
               if (c==' ') c=fgetc(fp);
               while ((c!=' ') && (c!='\n')) {
                    *p++= (char) c;
                    c=fgetc(fp);
               }
```

```
            strncpy(issue[j].komut,str,6);
      }  /* end of for  */
      return 1;
    }
    else return 0;
}

int find_tag()
{
 int l;

 l=0;
 while ((l<BUFSIZE) && (reo[l].tag!=current_tag)) l++;
 if (l>=BUFSIZE) return -1;
 else return l;
}

int suspend(FILE *fp)
{
 int l,j,bufno;

do {
 for(l=0;l<BUFSIZE;l++) {
            if (reo[l].cycle>0) (reo[l].cycle)--;
            if (reo[l].cycle==0) reo[l].valid=1;
 }
  bufno=find_tag();
  if (bufno!=-1) {
          if (reo[bufno].valid==1) {
           if (reo[bufno].exception==1) {
             fclose(fp);
             fp=fopen("MATREXC.DAT","r");
             introutine(reo[bufno].pc);
             initialize();
             inicycles();
             k=0;
             current_tag++;
             return -1;
           }
           else {
           while ((reo[bufno].valid==1)  && (bufno!=-1)) {
            reo[bufno].dirty=1;
            current_tag++;
            bufno=find_tag();
           }
          }
         }
    }
  total_cycles++;
  j=0;
  while ((j<BUFSIZE) && (reo[j].dirty==0)) j++;
  if (j<BUFSIZE) return j;
} while (1);
}
```

```
void sim()
{
  FILE *fp;
  int max,t,bufno,i,l,c,x0,flag,temp,instruction_no,x1,test;

  initialize();
  l=0;
  current_tag=1;
  total_cycles=0;
  fp=fopen("MATREISS.dat","r");
  flag=0;
  do {
   test=make_issue(fp);
   if (test!=0) {
    if (flag==1) {
          for(c=0;c<BUFSIZE;c++) {
           if (reo[c].cycle>0) (reo[c].cycle)--;
           if (reo[c].cycle==0) reo[c].valid=1;
          }

    bufno=find_tag();
    if (bufno!=-1) {
         if (reo[bufno].valid==1) {
          if (reo[bufno].exception==1) {
            fclose(fp);
            fp=fopen("MATREXC.DAT","r");
            introutine(reo[bufno].pc);
            initialize();
            inicycles();
            k=0;
            current_tag++;
            test=make_issue(fp);
          }
          else {
           while ((reo[bufno].valid==1)  && (bufno!=-1)) {
           reo[bufno].dirty=1;
           current_tag++;
           bufno=find_tag();
           }
          }
         }
   }
   total_cycles=calculate(k,total_cycles,cycles);
  }
    if (strcmp(issue[0].komut,"NOP")!=0) {
         x0=atoi(issue[0].komut);     /* Instruction number  */
         x1=x0-1;
         instruction_no=generate_pc(x0)-1;
         t=instruction_no;
         temp=atoi(sym[t].tab[5]);
         cycles[k++]=temp;
         l=0;
         while ((l<BUFSIZE) && (reo[l].dirty==0)) l++;
```

```
            if (l>=BUFSIZE) {
              printf("Reorder buffer line match\n");
              l=suspend(fp);
            }
            if (l!=-1) {
             iniexception(instruction_no,l);
             reo[l].pc=instruction_no;
             reo[l].valid=0;
             reo[l].dirty=0;
             reo[l].cycle=temp;
             reo[l].tag=x0;
            }
        }
        else  cycles[k++]=1;
       if (l!=-1) {
        if (strcmp(issue[1].komut,"NOP")!=0) {
            x0=atoi(issue[1].komut);
            x1=x0-1;
            instruction_no=generate_pc(x0)-1;
        t=instruction_no;
            temp=atoi(sym[t].tab[5]);
            cycles[k++]=temp;
        l=0;
            while ((l<BUFSIZE) && (reo[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("Reorder buffer line match\n");
              l=suspend(fp);
            }
            iniexception(instruction_no,l);
            reo[l].pc=instruction_no;
            reo[l].valid=0;
            reo[l].dirty=0;
            reo[l].cycle=temp;
            reo[l].tag=x0;
        }
        else  cycles[k++]=1;
        flag=1;
      }
     }
    } while (test!=0);
    fclose(fp);
    max=cycles[0];
    for (i=1;i<k;i++) {
    if (cycles[i]>max) max=cycles[i];
    }
    total_cycles+=max;
    printf("TOTAL CYCLES OF 3x3 MATRIX MULTIPLICATION\n");
    printf("PROGRAM WITH REORDER BUFFER : %d ",total_cycles);
    getch();
}

int main()
{
        int i,j,l;
```

```
        FILE *prgfp;

        prgfp = fopen("MATRIX.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
                exit(1);
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strncpy(sym[i].tab[j],"",6);
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

```
/* Tarih Dosyası için Matris Çarpımı Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        char komut[6];
} ISSUE;

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}HISTORY_TABLE;

# define LOOP1     16
# define LOOP2     19
# define LOOP3     22
# define FIRSTLOOP  22    /* Address of the first loop */
# define SECONDLOOP 24
# define THIRDLOOP  26
# define BUFSIZE    10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
HISTORY_TABLE his[BUFSIZE];
ISSUE issue[2];
int total_cycles,k;
int cycles[2500];
int current_tag;
int l1=0,l2=0,l3=0;

void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) his[i].exception=1;
   else his[i].exception=0;
}

void initialize()  /* Initializes the history table */
{
 int i;

 for(i=0;i<BUFSIZE;i++) {
   his[i].tag=-1;
   his[i].exception=-1;
   his[i].pc=-1;
```

```
    his[i].valid=-1;
    his[i].dirty=-1;
    his[i].cycle=-1;
  }
}

void inicycles()
{
 int i;

 for(i=0;i<2500;i++) cycles[i]=0;
}

void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
 int i,m;

 m=l;
 if (m>FIRSTLOOP) {
   i=LOOP1*l1+LOOP2*l2+LOOP3*l3;
   m=m-i;
   if (m==FIRSTLOOP) l1++;
   else if (m==SECONDLOOP) l2++;
          else if (m==THIRDLOOP)  l3++;
   return m;
 }
 else {
   if (m==FIRSTLOOP) l1++;
   return m;
 }
}

int make_issue(FILE *fp)
{
 int i,j;
 char c,*p,str[7];

 c=' ';
   c=fgetc(fp);
   if (c!='.') {
     for(j=0;j<2;j++) strncpy(issue[j].komut,"",5);
     for(j=0;j<2;j++) {
       for(i=0;i<7;i++) str[i]='\0';
          p=str;
                if (c==' ') c=fgetc(fp);
                while ((c!=' ') && (c!='\n')) {
                        *p++= (char) c;
                        c=fgetc(fp);
                }
```

```
            strncpy(issue[j].komut,str,6);
        }  /* end of for  */
        return 1;
      }
      else return 0;
}

int find_tag()
{
 int l;

 l=0;
 while ((l<BUFSIZE) && (his[l].tag!=current_tag)) l++;
 if (l>=BUFSIZE) return -1;
 else return l;
}

int suspend(FILE *fp)
{
 int l,j,bufno;

do {
 for(l=0;l<BUFSIZE;l++) {
            if (his[l].cycle>0) (his[l].cycle)--;
            if (his[l].cycle==0) his[l].valid=1;
 }
  bufno=find_tag();
  if (bufno!=-1) {
          if (his[bufno].valid==1) {
           if (his[bufno].exception==1) {
             fclose(fp);
             fp=fopen("MATRXEXC.DAT","r");
             introutine(his[bufno].pc);
             initialize();
             inicycles();
             k=0;
             current_tag++;
             return -1;
           }
           else {
      while ((his[bufno].valid==1)  && (bufno!=-1)) {
            his[bufno].dirty=1;
            current_tag++;
            bufno=find_tag();
           }
          }
         }
    }
  total_cycles++;
  j=0;
  while ((j<BUFSIZE) && (his[j].dirty==0)) j++;
  if (j<BUFSIZE) return j;
} while (1);
}
```

```
void sim()
{
  FILE *fp;
  int max,t,bufno,i,l,c,x0,flag,temp,instruction_no,x1,test;

  initialize();
  l=0;
  current_tag=1;
  total_cycles=0;
  fp=fopen("MATRXISS.dat","r");
  flag=0;
  do {
   test=make_issue(fp);
   if (test!=0) {
    if (flag==1) {
          for(c=0;c<BUFSIZE;c++) {
           if (his[c].cycle>0) (his[c].cycle)--;
           if (his[c].cycle==0) his[c].valid=1;
          }

    bufno=find_tag();
    if (bufno!=-1) {
          if (his[bufno].valid==1) {
           if (his[bufno].exception==1) {
             fclose(fp);
             fp=fopen("MATRXEXC.DAT","r");
             introutine(his[bufno].pc);
             initialize();
             inicycles();
             k=0;
             current_tag++;
             test=make_issue(fp);
           }
           else {
            while ((his[bufno].valid==1)  && (bufno!=-1)) {
             his[bufno].dirty=1;
             current_tag++;
             bufno=find_tag();
            }
           }
          }
   }

          total_cycles=calculate(k,total_cycles,cycles);
  }
    if (strcmp(issue[0].komut,"NOP")!=0) {
          x0=atoi(issue[0].komut);     /* Instruction number  */
          x1=x0-1;
          instruction_no=generate_pc(x0)-1;
          t=instruction_no;
          temp=atoi(sym[t].tab[5]);
          cycles[k++]=temp;
          l=0;
```

```
            while ((l<BUFSIZE) && (his[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("History buffer line match\n");
              l=suspend(fp);
            }
            if (l!=-1) {
             iniexception(instruction_no,l);
             his[l].pc=instruction_no;
             his[l].valid=0;
             his[l].dirty=0;
             his[l].cycle=temp;
             his[l].tag=x0;
            }
       }
       else  cycles[k++]=1;
      if (l!=-1) {
       if (strcmp(issue[1].komut,"NOP")!=0) {
            x0=atoi(issue[1].komut);
            x1=x0-1;
            instruction_no=generate_pc(x0)-1;
       t=instruction_no;
            temp=atoi(sym[t].tab[5]);
            cycles[k++]=temp;
       l=0;
            while ((l<BUFSIZE) && (his[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("History buffer line match\n");
              l=suspend(fp);
            }
            iniexception(instruction_no,l);
            his[l].pc=instruction_no;
            his[l].valid=0;
            his[l].dirty=0;
            his[l].cycle=temp;
            his[l].tag=x0;
       }
       else  cycles[k++]=1;
       flag=1;
      }
     }
    } while (test!=0);
    fclose(fp);
    max=cycles[0];
    for (i=1;i<k;i++) {
    if (cycles[i]>max) max=cycles[i];
    }
    total_cycles+=max;
    printf("TOTAL CYCLES OF 3x3 MATRIX MULTIPLICATION\n");
    printf("PROGRAM WITH HISTORY FILE : %d ",total_cycles);
    getch();
}

int main()
{
```

```
        int i,j,l;
        FILE *prgfp;

        prgfp = fopen("MATRIX.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
                exit(1);
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strncpy(sym[i].tab[j],"",6);
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

```
/* Gölge Tutucu Yığınlar için  Matris  Çarpımı Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        int pc;
        int pipe_delay;
        int valid;
} IST;

# define  IST_SIZE   12
# define  LOOP1      16
# define  LOOP2      19
# define  LOOP3      22
# define  FIRSTLOOP   22     /* Address of the first loop */
# define  SECONDLOOP  24
# define  THIRDLOOP   26


/* GLOBAL DEGISKENLER */

SYMBOL sym[250];        /* table to keep symbol information */
IST  ist[IST_SIZE];
int total_cycles;
int cycles[2500];
int l1=0,l2=0,l3=0;

int generate_pc(int l)
{
 int i,m;

 m=l;
 if (m>FIRSTLOOP) {
   i=LOOP1*l1+LOOP2*l2+LOOP3*l3;
   m=m-i;
   if (m==FIRSTLOOP) l1++;
   else if (m==SECONDLOOP) l2++;
        else if (m==THIRDLOOP)  l3++;
   return m;
 }
 else {
   if (m==FIRSTLOOP) l1++;
   return m;
 }
}

int search_ist()
{
  int i=0;
```

```
  while ((i<IST_SIZE) && (ist[i].valid!=1)) i++;
  if (i>=IST_SIZE) return -1;
  else return i;
}

void ini_ist()
{
  int i;

  for(i=0;i<IST_SIZE;i++) {
     ist[i].pc=-1;
     ist[i].pipe_delay=-1;
     ist[i].valid=1;
  }
}

void update_ist()
{
  int i,te;

  for(i=0;i<IST_SIZE;i++) {
     te=ist[i].pipe_delay;
     if (te>0) {
          te--;
          ist[i].pipe_delay=te;
          if (te==0) ist[i].valid=1;
     }
  }
}

void sim()
{
  FILE *fp;
  int max,i,j,k,x0,flag,instr_po,temp,entry;
  char c,str[7],*p,*issue[2];

  total_cycles=0;
  ini_ist();
  fp=fopen("MATROISS.dat","r");
  flag=0;
  k=0;
  do {
   c=' ';
   c=fgetc(fp);
   if (c!='.') {
    for(j=0;j<2;j++) strncpy(issue[j],"",6);
    for(j=0;j<2;j++) {
          for(i=0;i<7;i++) str[i]='\0';
          p=str;
                if (c==' ') c=fgetc(fp);
                while ((c!=' ') && (c!='\n')) {
                      *p++= (char) c;
                      c=fgetc(fp);
```

```
                    }
                strncpy(issue[j],str,6);
        }  /* end of for  */
        if (flag==1) {
            total_cycles=calculate(k,total_cycles,cycles);
            update_ist();
        }
        if (strcmp(issue[0],"NOP")!=0) {
            x0=atoi(issue[0]);
            instr_po=generate_pc(x0)-1;
            temp=atoi(sym[instr_po].tab[5]);
            cycles[k++]=temp;
            entry=search_ist();
            if (entry!=-1) {
              ist[entry].pc=instr_po;
              ist[entry].pipe_delay=temp;
              ist[entry].valid=0;
            }
        }
        else  cycles[k++]=1;
        if (strcmp(issue[1],"NOP")!=0) {
            x0=atoi(issue[1]);
            instr_po=generate_pc(x0)-1;
            temp=atoi(sym[instr_po].tab[5]);
            cycles[k++]=temp;
            entry=search_ist();
            if (entry!=-1) {
              ist[entry].pc=instr_po;
              ist[entry].pipe_delay=temp;
              ist[entry].valid=0;
            }
        }
        else  cycles[k++]=1;
        flag=1;
      }
    } while (c!='.');
    fclose(fp);
    max=cycles[0];
    for (i=1;i<k;i++) if (cycles[i]>max) max=cycles[i];
    total_cycles+=max;
    printf("TOTAL CYCLES OF 3X3 MATRIX MULTIPLICATION\n");
    printf(" WITH SHADOW LATCH STACK : %d ",total_cycles);
    getch();
}

int main()
{
        int i,j,l;
        FILE *prgfp;

        prgfp = fopen("MATRIX.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
                exit(1);
```

```
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strcpy(sym[i].tab[j],"");
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

```
/* Yeniden Sıralama Tamponu için Asal Sayılar Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        char komut[6];
} ISSUE;

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}REORDER_TABLE;

# define LOOP1     3
# define LOOP2     4
# define LOOP3     9
# define FIRSTLOOP  7     /* Address of the first loop */
# define SECONDLOOP 16
# define THIRDLOOP  18
# define BUFSIZE    10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
REORDER_TABLE reo[BUFSIZE];
ISSUE issue[2];
int total_cycles,k;
int cycles[2500];
int current_tag;
int l1=0,l2=0,l3=0;

void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) reo[i].exception=1;
   else reo[i].exception=0;
}

void initialize()  /* Initializes the history table */
{
 int i;

 for(i=0;i<BUFSIZE;i++) {
   reo[i].tag=-1;
   reo[i].exception=-1;
   reo[i].pc=-1;
```

```
    reo[i].valid=-1;
    reo[i].dirty=-1;
    reo[i].cycle=-1;
  }
}

void inicycles()
{
 int i;

 for(i=0;i<2500;i++) cycles[i]=0;
}

void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
 int i,m,loopsize=50;

 m=l;
 if (m>FIRSTLOOP) {
  if (l1==loopsize) l1--;
   i=LOOP1*l1+LOOP2*l2+LOOP3*l3;
   m=m-i;
   if (m==FIRSTLOOP) l1++;
   else if (m==SECONDLOOP) l2++;
          else if (m==THIRDLOOP)  l3++;
   return m;
 }
 else {
   if (m==FIRSTLOOP) l1++;
   return m;
 }
}

int make_issue(FILE *fp)
{
int i,j;
char c,*p,str[7];

c=' ';
   c=fgetc(fp);
   if (c!='.') {
    for(j=0;j<2;j++) strncpy(issue[j].komut,"",5);
    for(j=0;j<2;j++) {
          for(i=0;i<7;i++) str[i]='\0';
          p=str;
                if (c==' ') c=fgetc(fp);
                while ((c!=' ') && (c!='\n')) {
                      *p++= (char) c;
                      c=fgetc(fp);
```

```
                }
            strncpy(issue[j].komut,str,6);
        } /* end of for */
        return 1;
      }
      else return 0;
}

int find_tag()
{
 int l;

 l=0;
 while ((l<BUFSIZE) && (reo[l].tag!=current_tag)) l++;
 if (l>=BUFSIZE) return -1;
 else return l;
}

int suspend(FILE *fp)
{
 int l,j,bufno;

do {
 for(l=0;l<BUFSIZE;l++) {
            if (reo[l].cycle>0) (reo[l].cycle)--;
            if (reo[l].cycle==0) reo[l].valid=1;
 }
  bufno=find_tag();
  if (bufno!=-1) {
          if (reo[bufno].valid==1) {
           if (reo[bufno].exception==1) {
             fclose(fp);
             fp=fopen("PRIMEXC.DAT","r");
             introutine(reo[bufno].pc);
             initialize();
             inicycles();
             k=0;
             current_tag++;
             return -1;
           }
           else {
           while ((reo[bufno].valid==1)  && (bufno!=-1)) {
            reo[bufno].dirty=1;
            current_tag++;
            bufno=find_tag();
           }
           }
          }
    }
  total_cycles++;
  j=0;
  while ((j<BUFSIZE) && (reo[j].dirty==0)) j++;
  if (j<BUFSIZE) return j;
} while (1);
```

```
}

void sim()
{
  FILE *fp;
  int max,t,bufno,i,l,c,x0,flag,temp,instruction_no,x1,test;

  initialize();
  l=0;
  current_tag=1;
  total_cycles=0;
  fp=fopen("PRMREISS.dat","r");
  flag=0;
  do {
   test=make_issue(fp);
   if (test!=0) {
    if (flag==1) {
          for(c=0;c<BUFSIZE;c++) {
           if (reo[c].cycle>0) (reo[c].cycle)--;
           if (reo[c].cycle==0) reo[c].valid=1;
          }

    bufno=find_tag();
    if (bufno!=-1) {
          if (reo[bufno].valid==1) {
           if (reo[bufno].exception==1) {
             fclose(fp);
             fp=fopen("PRIMEXC.DAT","r");
             introutine(reo[bufno].pc);
             initialize();
             inicycles();
             k=0;
             current_tag++;
             test=make_issue(fp);
           }
           else {
            while ((reo[bufno].valid==1)  && (bufno!=-1)) {
             reo[bufno].dirty=1;
             current_tag++;
             bufno=find_tag();
            }
           }
          }
   }
  total_cycles=calculate(k,total_cycles,cycles);
  }
    if (strcmp(issue[0].komut,"NOP")!=0) {
          x0=atoi(issue[0].komut);     /* Instruction number  */
          x1=x0-1;
          instruction_no=generate_pc(x0)-1;
          t=instruction_no;
          temp=atoi(sym[t].tab[5]);
          cycles[k++]=temp;
          l=0;
```

```
            while ((l<BUFSIZE) && (reo[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("Reorder buffer line match\n");
              l=suspend(fp);
            }
            if (l!=-1) {
             iniexception(instruction_no,l);
             reo[l].pc=instruction_no;
             reo[l].valid=0;
             reo[l].dirty=0;
             reo[l].cycle=temp;
             reo[l].tag=x0;
            }
        }
        else  cycles[k++]=1;
       if (l!=-1) {
        if (strcmp(issue[1].komut,"NOP")!=0) {
            x0=atoi(issue[1].komut);
            x1=x0-1;
            instruction_no=generate_pc(x0)-1;
         t=instruction_no;
            temp=atoi(sym[t].tab[5]);
            cycles[k++]=temp;
         l=0;
            while ((l<BUFSIZE) && (reo[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("Reorder buffer line match\n");
              l=suspend(fp);
            }
            iniexception(instruction_no,l);
            reo[l].pc=instruction_no;
            reo[l].valid=0;
            reo[l].dirty=0;
            reo[l].cycle=temp;
            reo[l].tag=x0;
        }
        else  cycles[k++]=1;
        flag=1;
       }
      }
     } while (test!=0);
     fclose(fp);
     max=cycles[0];
     for (i=1;i<k;i++) {
     if (cycles[i]>max) max=cycles[i];
     }
     total_cycles+=max;
     printf("TOTAL CYCLES OF PRIME NUMBERS\n");
     printf("PROGRAM WITH REORDER BUFFER : %d ",total_cycles);
     getch();
}

int main()
{
```

```
        int i,j,l;
        FILE *prgfp;

        prgfp = fopen("PRIME.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
                exit(1);
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strncpy(sym[i].tab[j],"",6);
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

```
/* Tarih  Dosyası için Asal Sayılar Programı  */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        char komut[6];
} ISSUE;

typedef struct {
        int tag;
        int exception;
        int pc;
        int valid;
        int dirty;
        int cycle;
}HISTORY_TABLE;

# define  LOOP1     3
# define  LOOP2     4
# define  LOOP3     9
# define  FIRSTLOOP  7     /* Address of the first loop */
# define  SECONDLOOP  16
# define  THIRDLOOP  18
# define  BUFSIZE    10

/* GLOBAL DEGISKENLER */

SYMBOL sym[250]; /* table to keep symbol information */
HISTORY_TABLE his[BUFSIZE];
ISSUE issue[2];
int total_cycles,k;
int cycles[2500];
int current_tag;
int l1=0,l2=0,l3=0;

void iniexception(int j,int i)
{
   if (strcmp(sym[j].tab[4],"E")==0) his[i].exception=1;
   else his[i].exception=0;
}

void initialize()  /* Initializes the history table */
{
 int i;

 for(i=0;i<BUFSIZE;i++) {
   his[i].tag=-1;
   his[i].exception=-1;
   his[i].pc=-1;
```

```
    his[i].valid=-1;
    his[i].dirty=-1;
    his[i].cycle=-1;
  }
}

void inicycles()
{
 int i;

 for(i=0;i<2500;i++) cycles[i]=0;
}

void introutine(int counter)
{
  strncpy(sym[counter].tab[4],"",6);
}

int generate_pc(int l)
{
 int i,m,loopsize=50;

 m=l;
 if (m>FIRSTLOOP) {
  if (l1==loopsize) l1--;
   i=LOOP1*l1+LOOP2*l2+LOOP3*l3;
   m=m-i;
   if (m==FIRSTLOOP) l1++;
   else if (m==SECONDLOOP) l2++;
          else if (m==THIRDLOOP)  l3++;
   return m;
 }
 else {
   if (m==FIRSTLOOP) l1++;
   return m;
 }
}

int make_issue(FILE *fp)
{
int i,j;
char c,*p,str[7];

c=' ';
   c=fgetc(fp);
   if (c!='.') {
     for(j=0;j<2;j++) strncpy(issue[j].komut,"",5);
     for(j=0;j<2;j++) {
       for(i=0;i<7;i++) str[i]='\0';
           p=str;
                  if (c==' ') c=fgetc(fp);
                  while ((c!=' ') && (c!='\n')) {
                          *p++= (char) c;
                          c=fgetc(fp);
```

```
                    }
                strncpy(issue[j].komut,str,6);
        }  /* end of for  */
        return 1;
      }
      else return 0;
}

int find_tag()
{
 int l;

 l=0;
 while ((l<BUFSIZE) && (his[l].tag!=current_tag)) l++;
 if (l>=BUFSIZE) return -1;
 else return l;
}

int suspend(FILE *fp)
{
 int l,j,bufno;

do {
 for(l=0;l<BUFSIZE;l++) {
              if (his[l].cycle>0) (his[l].cycle)--;
              if (his[l].cycle==0) his[l].valid=1;
 }
  bufno=find_tag();
  if (bufno!=-1) {
            if (his[bufno].valid==1) {
             if (his[bufno].exception==1) {
               fclose(fp);
               fp=fopen("PRIMEXC.DAT","r");
               introutine(his[bufno].pc);
               initialize();
               inicycles();
               k=0;
               current_tag++;
               return -1;
             }
             else {
        while ((his[bufno].valid==1)  && (bufno!=-1)) {
              his[bufno].dirty=1;
              current_tag++;
              bufno=find_tag();
             }
             }
            }
     }
  total_cycles++;
  j=0;
  while ((j<BUFSIZE) && (his[j].dirty==0)) j++;
  if (j<BUFSIZE) return j;
} while (1);
```

```
}

void sim()
{
  FILE *fp;
  int max,t,bufno,i,l,c,x0,flag,temp,instruction_no,x1,test;

  initialize();
  l=0;
  current_tag=1;
  total_cycles=0;
  fp=fopen("PRIMEISS.dat","r");
  flag=0;
  do {
   test=make_issue(fp);
   if (test!=0) {
    if (flag==1) {
         for(c=0;c<BUFSIZE;c++) {
          if (his[c].cycle>0) (his[c].cycle)--;
          if (his[c].cycle==0) his[c].valid=1;
         }

    bufno=find_tag();
    if (bufno!=-1) {
         if (his[bufno].valid==1) {
         if (his[bufno].exception==1) {
           fclose(fp);
           fp=fopen("PRIMEXC.DAT","r");
           introutine(his[bufno].pc);
           initialize();
           inicycles();
           k=0;
           current_tag++;
           test=make_issue(fp);
         }
         else {
          while ((his[bufno].valid==1)  && (bufno!=-1)) {
           his[bufno].dirty=1;
           current_tag++;
           bufno=find_tag();
          }
         }
        }
   }

        total_cycles=calculate(k,total_cycles,cycles);
  }
    if (strcmp(issue[0].komut,"NOP")!=0) {
        x0=atoi(issue[0].komut);     /* Instruction number  */
        x1=x0-1;
        instruction_no=generate_pc(x0)-1;
        t=instruction_no;
        temp=atoi(sym[t].tab[5]);
        cycles[k++]=temp;
```

```
            l=0;
            while ((l<BUFSIZE) && (his[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("History buffer line match\n");
              l=suspend(fp);
            }
            if (l!=-1) {
             iniexception(instruction_no,l);
             his[l].pc=instruction_no;
             his[l].valid=0;
             his[l].dirty=0;
             his[l].cycle=temp;
             his[l].tag=x0;
            }
        }
        else  cycles[k++]=1;
       if (l!=-1) {
        if (strcmp(issue[1].komut,"NOP")!=0) {
            x0=atoi(issue[1].komut);
            x1=x0-1;
            instruction_no=generate_pc(x0)-1;
         t=instruction_no;
            temp=atoi(sym[t].tab[5]);
            cycles[k++]=temp;
         l=0;
            while ((l<BUFSIZE) && (his[l].dirty==0)) l++;
            if (l>=BUFSIZE) {
              printf("History buffer line match\n");
              l=suspend(fp);
            }
            iniexception(instruction_no,l);
            his[l].pc=instruction_no;
            his[l].valid=0;
            his[l].dirty=0;
            his[l].cycle=temp;
            his[l].tag=x0;
        }
        else  cycles[k++]=1;
        flag=1;
       }
      }
     } while (test!=0);
     fclose(fp);
     max=cycles[0];
     for (i=1;i<k;i++) {
     if (cycles[i]>max) max=cycles[i];
     }
     total_cycles+=max;
     printf("TOTAL CYCLES OF PRIME NUMBERS PROGRAM WITH HISTORY FILE : %d
",total_cycles);
     getch();
}

int main()
```

```
{
        int i,j,l;
        FILE *prgfp;

        prgfp = fopen("PRIME.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
                exit(1);
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strncpy(sym[i].tab[j],"",6);
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

```
/* Gölge Tutucu Yığınlar için Asal Sayılar Programı */

#include<stdio.h>
#include<conio.h>
#include"bench.h"
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

typedef struct {
        int pc;
        int pipe_delay;
        int valid;
} IST;

# define IST_SIZE      12
# define LOOP1          3
# define LOOP2        4
# define LOOP3          9
# define FIRSTLOOP      7
# define SECONDLOOP   16
# define THIRDLOOP      18


/* GLOBAL DEGISKENLER */

SYMBOL sym[250];        /* table to keep symbol information */
IST  ist[IST_SIZE];
int total_cycles;
int cycles[3000];
int l1=0,l2=0,l3=0;

int generate_pc(int l)
{
  int i,m,loopsize=50;

  m=l;
  if ( m>FIRSTLOOP) {
     if (l1==loopsize) l1--;
     i=LOOP1*l1+LOOP2*l2+LOOP3*l3;
     m=m-i;
     if (m==FIRSTLOOP) l1++;
     else if ( m==SECONDLOOP) l2++;
             else if (m==THIRDLOOP) l3++;
             return m;
  }
  else {
    if (m==FIRSTLOOP) l1++;
    return m;
  }
}

int search_ist()
{
```

```
  int i=0;

  while ((i<IST_SIZE) && (ist[i].valid!=1)) i++;
  if (i>=IST_SIZE) return -1;
  else return i;
}

void ini_ist()
{
  int i;

  for(i=0;i<IST_SIZE;i++) {
    ist[i].pc=-1;
    ist[i].pipe_delay=-1;
    ist[i].valid=1;
  }
}

void update_ist()
{
  int i,te;

  for(i=0;i<IST_SIZE;i++) {
    te=ist[i].pipe_delay;
    if (te>0) {
        te--;
        ist[i].pipe_delay=te;
        if (te==0) ist[i].valid=1;
    }
  }
}

void sim()
{
  FILE *fp;
  int max,i,j,k,x0,flag,instr_po,temp,entry;
  char c,str[7],*p,*issue[2];

  total_cycles=0;
  ini_ist();
  fp=fopen("PRIMEISS.dat","r");
  flag=0;
  k=0;
  do {
   c=' ';
   c=fgetc(fp);
   if (c!='.') {
    for(j=0;j<2;j++) strncpy(issue[j],"",6);
    for(j=0;j<2;j++) {
        for(i=0;i<7;i++) str[i]='\0';
        p=str;
            if (c==' ') c=fgetc(fp);
            while ((c!=' ') && (c!='\n')) {
                *p++= (char) c;
```

```
                        c=fgetc(fp);
                    }
                strncpy(issue[j],str,6);
        }  /* end of for  */
        if (flag==1) {
            total_cycles=calculate(k,total_cycles,cycles);
            update_ist();
        }
        if (strcmp(issue[0],"NOP")!=0) {
            x0=atoi(issue[0]);
            instr_po=generate_pc(x0)-1;
            temp=atoi(sym[instr_po].tab[5]);
            cycles[k++]=temp;
            entry=search_ist();
            if (entry!=-1) {
              ist[entry].pc=instr_po;
              ist[entry].pipe_delay=temp;
              ist[entry].valid=0;
            }
        }
        else  cycles[k++]=1;
        if (strcmp(issue[1],"NOP")!=0) {
            x0=atoi(issue[1]);
         instr_po=generate_pc(x0)-1;
            temp=atoi(sym[instr_po].tab[5]);
            cycles[k++]=temp;
            entry=search_ist();
            if (entry!=-1) {
              ist[entry].pc=instr_po;
              ist[entry].pipe_delay=temp;
              ist[entry].valid=0;
            }
        }
        else  cycles[k++]=1;
        flag=1;
      }
    } while (c!='.');
    fclose(fp);
    max=cycles[0];
    for (i=1;i<k;i++) if (cycles[i]>max) max=cycles[i];
    total_cycles+=max;
    printf("TOTAL CYCLES OF PRIME NUMBERS PROGRAM WITH SHADOW LATCH STACK :
%d ",total_cycles);
    getch();
}

int main()
{
        int i,j,l;
        FILE *prgfp;

        prgfp = fopen("PRIME.SRC", "r");
        if (prgfp == NULL) {
                printf(" Can't open %s file for input\n");
```

```
            exit(1);
        }

        clrscr();

        for(i=0;i<250;i++) {
           for(j=0;j<6;j++) strcpy(sym[i].tab[j],"");
        }

        l=sembol_tablosu(prgfp,sym);
        exec(l,sym);
        sim();
        return 0;
}
```

ÖZGEÇMİŞ

Doğum Tarihi	18 Ekim 1971
Doğum Yeri	Ankara
Mezun Olduğu Lise	50. Yıl Tahran Lisesi
Mezuniyet Tarihi	Haziran 1988
Mezun Olduğu Üniversite	Yıldız Teknik Üniversitesi
Mezuniyet Tarihi	Haziran 1993
Mezun Olduğu Bölüm	Bilgisayar Bilimleri Mühendisliği Bölümü