1-) RISC mimarisinin geliştirilmesinin gerekçelerini ve RISC mimarisiyle CISC mimarisi arasındaki farkları detaylı açıklayınız.

- Yüksek seviyeli dillerle yazılan programların CISC bilgisayarlarda çalışmaları analiz edilmiştir:
- Operations: CPU tarafından gerçekleştirilen işlemler ve hafıza erişimleri incelenmiştir,
- Operands: kullanım frrekansları ve adresleme modları incelenmiştir,
- Execution sequencing: Kontrol ve pipeline organizasyonu incelenmiştir.
- Yapılan tüm çalışmalarda programların çalışmaları sırasındaki dinamik sonuçlar alınmıştır.

Operations

- Atama deyimleri (assignments) ağırlıklı olarak çalıştırılmaktadır (Data movement instructions).
- Şartlı atlama deyimleri sıklıkla kullanılmaktadır (if, loop, program akış kontrolü).
- Prosedür çağırma ve geri dönme komutları çok zaman harcamaktadır (call, return).
- Bazı HLL komutları çok sayıda makine komutu ile oluşturulmaktadır.

Operands

- Programlarda çalışma sırasında çoğunlukla lokal skalar değişkenler kullanılmaktadır.
- Lokal değişkenlere erişim optimizasyonu yapılmalıdır.

Procedure call/return

- Procedure çağırmalarda parametre sayısı ve içiçe çağırma sayısı önemlidir.
- Procedure'lerden %98'i 6' dan az parametre göndermektedir.
- Procedure'lerin %92'si 6' dan az lokal değişken kullanmaktadır.

Elde edilen sonuçlar

- Yüksek seviyeli dilleri desteklemek için geliştirilen komut kümeleri çok etkili bir yöntem olmamıştır.
- Bunun yerine çok zaman harcayan işlemlerin optimizasyonu daha uygundur.
- Operandların saklanması için çok sayıda register oluşturulmalıdır.
- Şartlı atlama ve procedure çağırma komutlarındaki pipeline performansının artırılması gereklidir.

CISC ve RISC mimarisinin karşılaştırılması

- RISC sabit komut uzunluğuna sahiptir (4 byte).
- RISC az sayıda adresleme moduna sahiptir (5 ten az)
- RISC indirect adresleme modu kullanmaz.
- RISC mimarisinde aritmetik işlemlerde load/store işlemi yapılmaz.
- RISC mimarisinde komutlarda birden fazla hafıza adresleme yapılmaz.
- RISC mimarisinde integer register adresleme için 5 veya daha fazla bit kullanılır (en az 32 integer register).
- RISC mimarisinde floating-point register adresleme için 4 veya daha fazla bit kullanılır (en az 16 integer register).
- 1-2 decode işlemi karmaşıklığını, 3-5 pipeline performansını, 6-7 compiler performansını belirler.

2-) Instruction Pipeline hakkında bilgi veriniz. Pipeline'da performans artışını cebirsel olarak gösteriniz. Branch instruction'larında pipeline performansını düşürmemek için kullanılan yaklaşımları detaylı anlatınız.

- Instruction pipelining bir üretim hattının çalışmasına benzer.
- Bir üretim hattında farklı işler eş zamanlı yapılır.
- Fetch aşamasında hafızaya erişilir ancak execute aşamasında genellikle hafızaya erişilmez.
- Execute aşamasında sonraki komutun fetch edilmesi (prefetch) performansı artırır.
- Fetch aşaması execute aşamasında çok kısa sürer. Birden fazla komut prefetch yapılabilir.
- Branch ve jump komutlarında gerekli olmayan komut prefetch yapılabilir.

Pipelining altı aşamayla ifade edilebilir.

- Fetch instruction (FI): Sonraki komut alınır
- Decode instruction (DI): Komut decode edilir
- Calculate operands (CO): Giriş operandlarının adresleri hesaplanır
- Fetch operands (FO): Giriş operandları alınır
- Execute instruction (EI): Komutun gerektirdiği işlem yapılır
- Write operand (WO): Sonuç operand varsa saklanır
- Bazı aşamalar her komutta kullanılmaz (FO, WO)

Pipeline ile performans artışı aşağıdaki gibi ifade edilir.

$$\tau = max\,[\tau_i] + d = \tau_m + d \qquad 1 \le i \le k$$

burada,

$\tau_i = i$. aşamadaki gecikme

τ_m = tüm aşamalardaki gecikmelerden maksimum olan

k = pipeline'daki aşama sayısı

d = datanın aşamalar arasındaki aktarımı için geçen süre

$\tau_m >> d$ olduğu için d ihmal edilir. $T_{k,n}$, n satır programın k aşamalı pipeline ile çalışma süresidir ve

$T_{k,n} = [k + (n - 1)].\tau$ şeklinde gösterilir. Performans artışı ise

$S_k = T_{1,n} / T_{k,n} = n.k.\tau / [k + (n - 1)].\tau =$ ***n.k / [k + (n - 1)]***

n*→∞** *giderken performans artışı* ***k *olur.*

Branch instruction'larda sonraki performansını düşürmemek için kullanılan yaklaşımlar:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Delayed branch
- Branch prediction

Multiple streams

- Pipeline iki yöndeki komutlarıda fetch eder.
- Birden fazla atlama komutu pipeline'a girerse her atlama komutu için bir stream gerekir.

Prefetch branch target

- Şartlı atlama komutu geldiğinde sonraki komutla birlikte hedef komutta prefetch edilir.
- Target saklanır ve branch taken olursa kullanılır.

Loop buffer

- En son fetch edilen n adet komutu saklayan bir hafıza birimidir.
- Komutlar sıralı saklanır ve hafıza erişimi olmadan alınır.
- Atlama aralığı az olursa veya döngü kısa olursa sürekli buffer üzerinde çalışılır.

Delayed branch

- Komutlar yeniden düzenlenir ve bir atlama komutu olması gerekenden sonra çalışır.
- Bağımsız bir veya birkaç komut, atlama komutu ile ardındaki komutlar arasına alınır.

Branch prediction

- Bir atlama komutunun taken olup olmayacağı tahmin edilir.
- Predict never taken, predict always taken, predict by opcode, taken/not taken switch ve branch history table teknikleri kullanılır.
- İlk üç yöntem statiktir ve programın çalışmasına bağlı değildir. Son iki teknik dinamiktir ve programın çalışması sırasında karar verilir.

Branch prediction – predict never taken

- Atlamaların hiçbir zaman taken olmayacağı varsayılır.
- Yapılan deneysel çalışmalarda %50'den fazla atlamanın taken olduğu görülmüştür.

Branch prediction – predict always taken

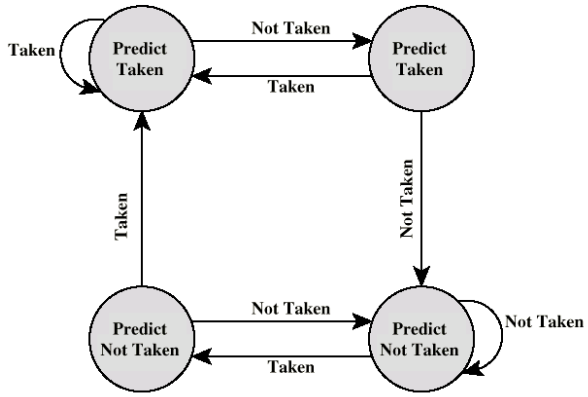
- Atlamaların her zaman taken olacağı varsayılır. Never taken tekniğinde daha başarılı sonuç alınır.

Branch prediction – predict by opcode

- Atlamalar opcode'larına göre taken ve not taken şeklinde gruplandırılır.
- Başarı oranı %75'den büyüktür.

Branch prediction – taken / not taken switch

- Önceki çalışma durumuna bağlıdır. Bir veya birden fazla bit ile her komutun önceki durumu saklanır.
- Bir bitle saklandığı zaman her döngünün başında ve sonunda hata yapılır.
- Bir bitle taken ve not taken arasında her hatalı atlamada sürekli switch yapılır.
- İki bitle iki hata üstüste yapıldığında switch yapılır. Döngülerin başında ve sonundaki hatalarda ortadan kaldırılır. İki bitle yapılan taken ve not taken arasındaki switch aşağıdaki gibidir.



Branch prediction – branch history table

- Branch history table pipeline'ın fetch aşamasıyla ilişkilendirilmiş küçük bir önbellektir.
- Branch miss durumunda yeni bir giriş kaydedilir veya varsa durumu güncellenir.
- Target addres genellikle hedef komutun adresi olur veya hedef komutun kendisi olabilir.
- Execute aşaması, sonucu branch history table devresine iletir.

3-) Aşağıdaki veriyolunda multicycle olarak **bre** ve **jump** komutlarının çalışmasını sonlu durum makinesiyle göstererek anlatınız.