T.C. SANAYİ VE
TEKNOLOJİ BAKANLIĞI

Ayhan Karaca (ayhankaraca352@gmail.com)

Eğitmen: Ömer Güzel

ÇİP TASARIMI UZMANLIK PROGRAMI

MARMARA ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
ELEKTRİK ELEKTRONİK MÜHENDİSLİĞİ

3. YIL 2. DÖNEM

TAM BYPASSLI PIPELINE MODELİN TAMAMLANMASI VE 3 YENİ KOMUT EKLENMESİ

Contents

List of Figures

List of Tables

Chapter 1

GİRİŞ

Bu rapor, Sanayi ve Teknoloji Bakanlığı'nın yürüttüğü Çip Tasarımı Uzmanlığı Programı kapsamında gerçekleştirilen bitirme projesini kapsamaktadır. Proje kapsamında, RISC-V mimarisine uygun 5 aşamalı (5-stage) bir pipelined işlemci tasarımı yapılmıştır. Başlangıçta temel yapısı verilen işlemci tasarımına veri ve kontrol hazard'larını yöneten duraklatma (stall) ve temizleme (flush) mekanizmaları eklenerek mimari tamamlanmıştır. Verilen test dosyası ve buna ek olarak yazılan test dosyaları ile doğrulanmıştır. Bu raporda proje sürecinde yapılan çalışmalar, karşılaşılan problemler ve elde edilen deneyimler özetlenmiştir.

1.1 Motivasyon

Sayısal tasarım alanındaki bilgimi derinleştirmek ve çip tasarımı konusunda uygulamalı deneyim kazanmak amacıyla bu programa başvurdum. İşlemci mimarisi ve donanım tasarımı alanında kariyer hedeflediğim için, programın RTL tasarım, doğrulama ve özellikle RISC-V mimarisi odaklı içeriği benim için oldukça değerliydi. Deneyimli mentorlar eşliğinde gerçek bir proje üzerinde çalışabilme imkânı da bu programa katılmamda önemli bir motivasyon kaynağı oldu.

1.2 Proje Seçimi

İşlemci mimarileri ve donanım düzeyinde tasarım konuları, ilgi alanlarım arasında yer alıyor. Bu nedenle, RISC-V mimarisi gibi açık kaynak ve güncel bir mimari üzerine kurulu bir işlemci tasarımı projesi, teorik bilgilerimi uygulamaya dökmek açısından ideal bir fırsat sundu.

Chapter 2

TASARIM DETAYLARI

Tasarım süreci, eğitim sırasında verilen temel bir işlemci mimarisi üzerinden başlatıldı. Başlangıçta instruction fetch, decode ve execute aşamaları büyük ölçüde hazır durumdaydı. Bu temel yapı üzerine memory ve writeback modülleri eklendi ve veri ve kontrol hazard'larını yönetebilmek için stall, forward ve flush mekanizmalarını ekleyerek tasarım tamamlandı.

Veri bağımlılıklarını çözmek için forwarding ünitesi tasarlandı ve execute aşamasından önce operand'ların gerekli durumlarda önceki aşamalardan alınması sağlandı. Load-use hazard için stall mekanizması kuruldu. Kontrol hazard'ları için ise branching sonrası yanlış tahminleri temizleyen flush yapısı uygulandı. Bu mekanizmalar sayesinde pipeline'da hatalı veri işlenmesi ve performans kaybı minimize edildi.

2.1 Genel Sistem Mimarisi

RTL kodu yazılırken şekil 2.1'da gösterilen mimari dikkate alındı. Bu mimariye ek olarak hazard unit daha sonra tasarlandı. Şekilde görüldüğü gibi işlemci mimarisi 5 aşamadan oluşmaktadır.

2.1.1 Pipeline Stage İmplementasyonu

- **Instruction Fetch (IF):** Bu aşamada, Program Counter (PC), Instruction Memory (IMEM)'e adres sağlar ve ilgili instruction alınır. PC, bir sonraki instruction'ı işaret etmesi için (çünkü her instruction 4 byte'tır) 4 artırılır veya eğer bir branch gerçekleşmişse, branch instruction'ı işaret edecek şekilde güncellenir. Alınan instruction ve güncellenmiş PC, IF/ID pipeline registerına kaydedilir.

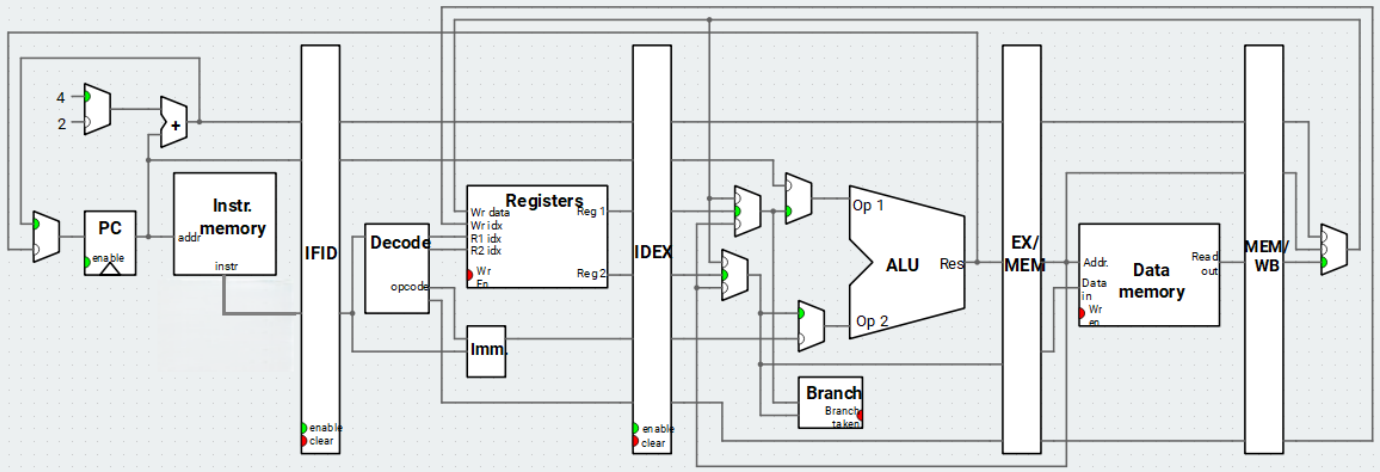

Figure 2.1: Architecture

- **Instruction Decode (ID):** Instruction, türünü (R, I, S, B, U veya J) belirlemek ve işlenen register adresleri ile immediate değerleri çıkartmak için decode edilir. Register file, rs1 ve rs2 alanlarına göre okunur. Immediate değerler gerektiği şekilde genişletilir. Memory write, register write gibi kontrol sinyalleri de bu aşamada üretilir ve bir sonraki aşamaya iletilir.

- **Execute (EX):** Arithmetic Logic Unit (ALU), register file'dan gelen operandlar veya immediate değer kullanılarak gerekli hesaplamayı yapar. Bu hesaplamalar, aritmetik işlemler, mantıksal işlemler, shift işlemleri ve branch karşılaştırmalarını içerir. Branch instruction'lar için, branch hedefi burada hesaplanır ve branch kararı burada verilir.

- **Memory Access (MEM):** Load ve store instruction'lar bu aşamada Data Memory (DMEM) ile etkileşime girer. Kontrol sinyallerine bağlı olarak ya okuma ya da yazma işlemi gerçekleştirilir. Memory ile ilgili olmayan instruction'lar için bu aşama sadece veriyi bir sonraki aşamaya aktarır.

- **Write Back (WB):** Son aşamada, bir işlemin sonucu (ALU veya Data Memory'den) register file'daki hedef register'a yazılır ve instruction'ın yürütülmesi tamamlanır. Bu aşama bir sonraki düşen kenarda gerçekleştirilir. Bunun nedeni, veri bağımlılığı nedeniyle kaybedilen cycle sayısını 1 azaltmaktır. Bu konu, veri hazard'larının ortadan kaldırılması bölümünde ayrıntılı olarak açıklanacaktır.

Her aşama, farklı instruction'lar üzerinde diğerleriyle eş zamanlı çalışarak verimli bir pipeline yürütümü sağlar.

2.1.2 Veriyolu (Datapath)

Aşamalar, kod yazımı sırasında modüller şeklinde yazılmıştır. Tasarımda altı modül bulunmaktadır ve bunlardan biri top modüldür. Bu bölümde, bu altı modülün giriş-çıkışları ve aralarındaki bağlantılar incelenecektir.

- **FETCH MODULE** Bu modüllerde clock ve reset girişleri sabittir çünkü pipeline mimarisinin doğası gereği tüm modüller ardışıl (sequential) tasarımlar içerir. Veriler tüm modüllerde clock'un yükselen kenarında bir sonraki aşamaya geçer ve aktif düşük reset kullanılır. Bu bilgi, sonraki bölümler için de geçerlidir. Clock ve reset girişlerine ek olarak, fetch modülünde 3 giriş daha vardır. Bunlar pc_en_i, next_pc_i ve next_pc_enable_i girişleridir. pc_en_i girişi stall ile ilgilidir. Eğer bu giriş aktifse, bir sonraki instruction alınır. Aksi halde yeni instruction alınmaz ve 1 çevrimlik bir stall meydana gelir. Bunun sebebi Hazardları Elimine Etme bölümünde açıklanacaktır. next_pc_i ve next_pc_enable_i girişleri branching ile ilgilidir. Eğer next_pc_enable_i high seviyedeyse, bir sonraki adres +4 ile artırılmış bir adres değildir. Bunun yerine execute aşamasında hesaplanan adrestir. next_pc_i, bahsettiğim bu adrestir. Bu modülün iki çıkışı vardır. Bunlar pcF_o ve instrF_o'dur. Sinyal isimlerinde o aşamayı belirten harfler bulunur. Bu harfler debug sırasında oldukça faydalıdır. pcF_o çıkışı aslında gerekli bir sinyal değildir, ancak bu sinyali pipeline boyunca ileteceğiz, böylece referans model ile cpu çıktılarımızı karşılaştırırken faydalı olacaktır. instrF_o decode aşamasına gönderilir ve orada decode edilir.

- **DECODE MODULE:** flush_i girişi hazard unit'ten gelen bir sinyaldir. Eğer bu sinyal aktifse, bu aşamadaki instruction üzerinde hiçbir işlem yapılmaz. Doğrudan nop'a dönüştürülür ve bir sonraki aşamaya gönderilir. pcD_i ve pcD_o portlarının bu modül için önemi yoktur. Fetch aşamasında belirtildiği gibi, sadece pipeline boyunca iletilirler. instrD_i, decode edilecek instruction'ı temsil eder. Bu 32-bit giriş, bit alanlarına ayrılarak decode edilir. rdWB_port_i, rd_port_t struct'ının bir örneğidir. Bu struct, destination register'ın adresini (5 bit), içeriğini (32 bit) ve bu adrese yazılıp yazılmayacağını belirten enable (1 bit) sinyalini içerir. Write back aşaması bu modülde düşen clock kenarında gerçekleşir. Daha önce söylediğim gibi, yazma işleminin düşen kenarda yapılmasının sebebi ilerleyen bölümlerde açıklanacaktır. instrD_o çıkışı,

tıpkı pcD portu gibi debug amaçlı olarak çıkışa iletilir. operationD_o çıkışı enum türünde operation_e değişkenidir. ALU modülünün ihtiyaç duyduğu işlemleri içerir. Kodun okunabilirliğini artırmak için tercih edilmiştir. rs1D_idx_o ve rs2D_idx_o çıkışları, instruction decode edildikten sonra elde edilen kaynak register adresleridir. Bu register adreslerindeki veriler işlenmek üzere ex aşamasına gönderilir. rs1D_o ve rs2D_o çıkışları ise bu adreslerdeki 32-bit verileri temsil eder. rdD_addr_o çıkışı, eğer bu instruction write-back yapılacak bir instruction ise, destination register adresini gösterir. rdD_wrt_ena_o ve memD_wr_ena_o çıkış sinyalleri, register veya memory'e yazma işlemi gerekiyorsa aktif olan kontrol sinyalleridir. shamt_dataD_o ve immD_o çıkışları ex aşamasında ihtiyaç duyulan verilerdir. Eğer sonraki aşamada shift işlemleri yapılacaksa, bu shift miktarı decode aşamasında decode edilip sonraki aşamaya gönderilir. Eğer bu instruction I type ya da load-store tipi ise, immediate veri adres hesaplamasında yada ALU işlemlerinde kullanılır.

- **EXECUTE MODULE:** flush_i ve stallE_i giriş sinyalleri hazard unit'ten gelen kontrol sinyalleridir. Eğer bunlardan biri aktifse, execution aşamasındaki instruction devre dışı bırakılır ve işlenmez. İşlense bile branch işlemleri, memory'ye yazma veya register'lara yazma yapılmaz. operationE_i girişi ALU'da hangi işlemin yapılacağını belirleyen giriş sinyalidir. rs1E_i ve rs2E_i girişlerinden gelen veriler operationE_i girişine göre işlenir ve çıkışa aktarılır. rdE_addr_i decode aşamasından gelen ve destination register adresini tutan veridir ve bu aşamada doğrudan çıkışa aktarılır. Çünkü bu veri WriteBack aşamasında gereklidir. rdE_wrt_ena_i ve memE_wrt_ena_i sinyalleri az önce bahsettiğim flush ve stall komutlarına göre tekrar işlenir. Aslında bu veriler decode aşamasında bilinir, ancak hazard unit'ten gelen sinyaller sonucunda instruction geçersiz kılınırsa bu sinyaller yeniden düzenlenebilir. shamt_dataE_i ve immE_i verileri instruction tipine göre gereken verilerdir. Decode kısmı bu sinyallerin execution aşamasında ne işe yaradığını açıklar. operationE_o çıkışı da memory aşamasında gereklidir ve çıkışa iletilir. Load ve store işlemleri memory aşamasında gerçekleştirilir. Bu nedenle bu çıkış memory aşamasında da gereklidir. rdE_port_o çıkışı, Decode aşamasında bahsettiğim rdE_port_i girişine karşılık gelen çıkışları üretir. Bu yüzden bu çıkış aynı struct'ın bir örneğidir. memE_wrt_ena_o, memE_wrt_addr_o ve memE_wrt_data_o verileri sırayla belleğe yazma izini, belleğe yazılacak adres ve belleğe yazılacak verilerdir. Bellek adresi ALU da hesaplanır. stallE_o çıkışı, daha önce bazı sinyaller için açıklandığı gibi, debug amaçlı doğrudan çıkışa aktarılır. Bu çıkış sayesinde test output logunda STALL olarak belirtilir. next_pc_ena_o ve next_pc_o

çıkışları fetch aşamasına gönderilir. Bunlar branch olup olmayacağını ve eğer varsa hangi adrese yapılacağını gösteren çıkış verileridir. Bu veriler ders süresince kombinasyonel olarak fetch aşamasına gönderilmişti. Ama bu projede registerlanarak gönderilmiştir. Bu branch komutlarından sonra iki çevrim kaybı anlamına gelir. Kombinasyonel olarak gönderilseydi dallanma komutlarından sonra bir çevrim kaybı olurdu. Ancak bu implementasyon şeklinin kombinasyonel path delayi arttırabir ve sentez aşamasında zaman ihlallerine sebep olabilirdi. Yine de bu bir tasarım kararıdır ve ihtiyaca göre modifiye edilebilir. Branch komutlarından sonra neden iki çevrim kaybı olduğu hazardları elimine etme bölümünde anlatılacaktır.

- **MEMORY MODULE:** Daha önce bahsettiğim debug için pipeline çıkışına iletilmesi gereken input ve output portlar burada açıklanmayacaktır. operationM_i input'u, load-store türünü belirtir. Bu aşamada load ve store işlemleri bu input verisine göre gerçekleştirilir. rdM_port_i, bir sonraki aşama olan write-back aşamasında ihtiyaç duyulacak hedef register adresini, yazılacak veriyi ve write enable sinyalini içerir. addrM_i ve dataM_o debug amaçlı kullanılan portlardır. Kullanacağımız testte, addrM_i portundan gireceğimiz adresteki veriyi görüntüleyebileceğiz.

- **HAZARD UNIT MODULE:** Bu modül, kombinasyonel olarak tasarlanmış bir modüldür. Bu nedenle clock ve reset sinyalleri içermez. Bu modülün genel amacı, data hazard ve control hazard türlerini tespit etmek ve bu hazard'ları önlemek için bazı kontrol sinyalleri üretmektir. Data hazard tespiti için rs1D_i, rs2D_i, rdM_i, rdE_wr_ena_i, rdM_wr_ena_i ve rdE_i input portlarına ihtiyaç vardır. Bu sinyaller işlenerek forwardA_o ve forwardB_o output sinyalleri üretilir. load-use olarak bilinen hazard türünün tespiti için rs1_f_d_i, rs2_f_d_i, rd_d_e_i, opF_i ve opE_i sinyalleri gereklidir. Bu sinyaller işlenerek pc_en_o ve stallH_o output'ları üretilir. Control hazard tespiti için branch_tkn_i sinyali kullanılır ve gerekli kontrol sinyali olan flush_o üretilir.

- **TOP MODULE:** Top modülde, daha önce bahsettiğim modüller birbirine uygun şekilde bağlanmıştır. Bu modülde veri yönlendirmesi (data forwarding) için mux'lar kullanılmıştır. Hazard unit'ten gelen forwarding sinyallerine göre rs1_final ve rs2_final sinyallerinin değerleri belirlenir. Bu modülde input olarak addr_i kullanılır. Bu, daha önce belirttiğim gibi, memory içeriğine erişmek ve çıktıları komut penceresinde yazdırmak içindir. data_o output portu da bu amaca hizmet eder. pc_o, instr_o, reg_addr_o, reg_data_o ve stall_o portları ise CPU'ya verilen instruction'lara göre üretilen output'lardır. Model doğrulaması, bu çıkışlar gözlemlenerek yapılacaktır.

2.1.3 Hazardların Tespiti ve Elimine Edilmesi

Pipelined bir işlemcide, farklı aşamalardaki instruction'lar arasında bağımlılıklar olduğunda, instruction'ların normal yürütülmesi kesintiye uğrayabilir ve bu durum hazard'lara neden olur. Bu bölümde, tasarlanan yapının hem data hazard hem de control hazard durumlarını özel bir hazard_unit modülü kullanarak nasıl yönettiği açıklanmaktadır.

2.1.3.1 Veri Tehlikeleri ve Forwarding Logic

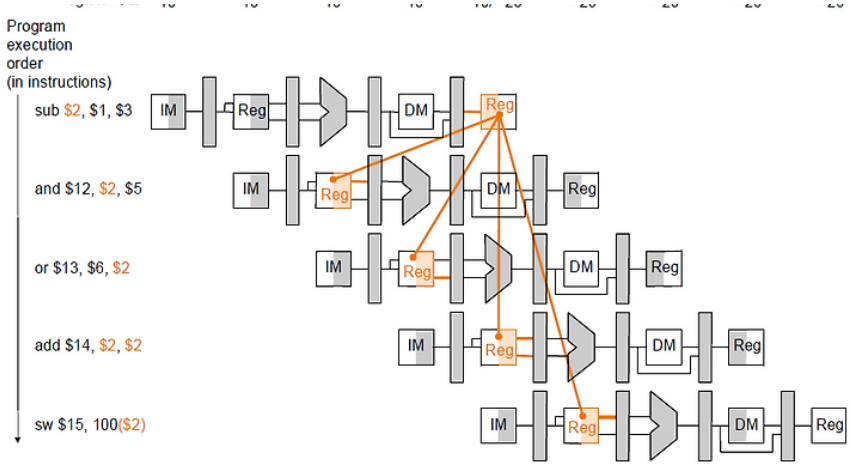

Figure 2.2: Data Hazard

Data hazard'lar, bir instruction'ın henüz yürütülmesini tamamlamamış olan önceki bir instruction'ın sonucuna bağlı olduğu durumlarda ortaya çıkar. Bunu azaltmak için işlemci, bir instruction'ın sonucunun register file'a yazılmasını beklemeden, doğrudan sonraki instruction'lara iletilmesini sağlayan data forwarding yöntemini kullanır.

hazard_unit modülü, Decode aşamasındaki source register'lar (rs1D_i, rs2D_i) ile Execute ve Memory aşamalarındaki destination register'lar (rdE_i, rdM_i) arasında eşleşme olup olmadığını kontrol eden bir forwarding logic bloğu içerir. Forwarding kararı aşağıdaki kriterlere göre verilir:

Always_comb bloğu içerisinde forwardA_o ve forwardB_o değerleri NO_FRWD olarak ilklendirilir. Eğer Decode aşamasındaki komutun source register adreslerinden biri veya ikisi ile ex aşamasındaki hedef register adresi aynı ise veri ex aşamasından aktarılır. Eğer Decode aşamasındaki komutun source register adreslerinden biri veya ikisi ile Memory aşamasındaki hedef register adresi aynı ise ver Memory aşamasından Execution aşamasında ALU girişlerine aktarılır. Bu yüzden core_model modülünde ALU girişlerine muxlar

koyulmuştur. Muxun girişleri Decode aşamasından gelen rs1 veya rs2 verisi, Execution aşamasından gelen rs1 veya rs2 verisi ve Memory aşamasından gelen rs1 veya rs2 verisidir. Muxun seçme girişi ise iki bitlik forward sinyalidir. Çıkışı ise rs1_final veya rs2_final verisidir. Bu ALUda işlenecek veridir. Eğer bir komutun hedef register adresi sonraki gelen iki komutun source registerları ile bağımlı ise bu durumda öncelik Execution aşamasından verinin iletilmesidir. Registera yazma işleminin düşen kenarda yapılmasının sebebi, bir komuttan sonra gelen üçüncü komutun veri bağımlılığında etkilenmemesi içindir. Yani ilk komut Memory aşamsından sonraki ilk düşen kenarda yazma işlemini yaparsa o komuttan sonra gelen üçüncü komut decode aşamasında olur ve decode edilmesi için gereken yükselen kenardan bir önceki düşen kenarda write back olduğu için veri bağımlılığından etkilenmez. Bu durumun da zamanlama ile ilgili dezavantajları bulunur. Bu durum da yine bir tasarım kararıdır. Bu mantık, Read After Write (RAW) hazard'larından kaynaklanan performans kaybını en aza indirir.

2.1.3.2 Load-Use Hazard Tespiti

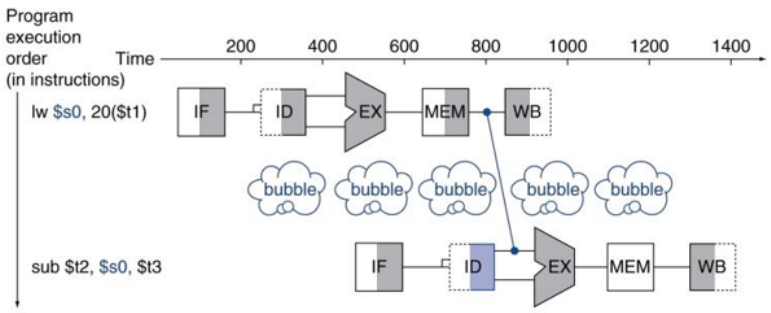

Figure 2.3: Load-use Hazard

Load-use hazard olarak bilinen özel bir data hazard türü, bir load instruction'ın hemen ardından bu yüklenen veriyi kullanan bir instruction geldiğinde meydana gelir. Verinin memory aşamasına kadar hazır olmaması nedeniyle, bu durumda forwarding yeterli olmaz.

hazard_unit, Execute aşamasındaki mevcut instruction'ın bir load instruction olup olmadığını (opE_i) ve bu instruction'ın hedef register'ı (rdE_i) ile Decode aşamasındaki instruction'ın kaynak register'larından (rs1D_i, rs2D_i) herhangi birinin eşleşip eşleşmediğini kontrol eder. Ayrıca decode aşamasında(decode edilmeden hemen önce) komutun opcode u kontrol edilir ve bu aşamadaki komutun load veya store olmaması gerekir. Eğer load veya store ise pipelinei stall etmeye gerek yoktur çünkü sonraki gelen komutta memory işlemidir. Özet olarak Load komutundan sonra load veya store komutu gelirse, bu komut

veri bağımlılığından etkilenmez çünkü o komutta memory aşamasında execute edileceği için bir önceki komuttaki ihtiyaç duyulan veri çoktan registera load edilmiş olur. Eğer bir load-use hazard algılanırsa:

stallH_o sinyali aktif hale getirilerek pipeline bir çevrim boyunca duraklatılır. pc_en_o sinyali pasif hale getirilerek Program Counter sabitlenir. Ex aşamasındaki komut için yazma işlemleri ve branch işlemleri bloke edilir ve bu şekilde pipeline boyunca ilerler. Böylece bu komut nop gibi davranır ve dikkate alınmaz. Test çıktılarında bu komutun yanında STALL! uyarısı verilmiştir.

Bu duraklama, memory'den gelen verinin kullanılabilir hale gelmesini sağlar ve bağımlı instruction'ın doğru şekilde yürütülmesini garanti eder.

2.1.3.3 Kontrol Hazardları ve Flushing

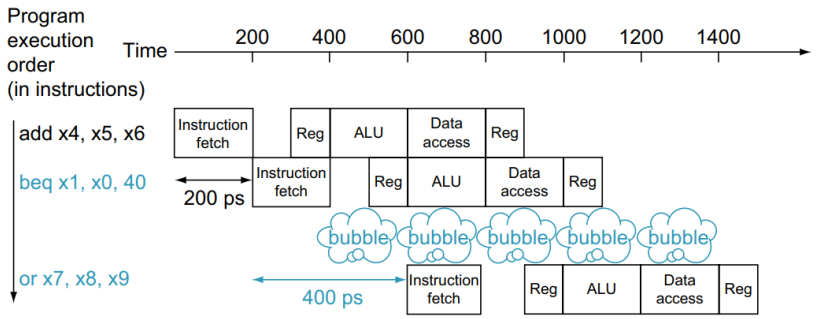

Figure 2.4: Control Hazard

Control hazard'lar, sonucu Execute aşamasına kadar bilinmeyen branch instruction'larından kaynaklanır. Branch'in alınıp alınmadığı belli olmadan sonraki instruction'ların yürütülmesi, hatalı davranışlara yol açabilir.

Bununla başa çıkmak için, hazard_unit bir branch alındığında (branch_tkn_i sinyali high olduğunda) flush sinyali (flush_o) üretir. Bu sinyal, Decode aşamasındaki mevcut instruction'ı geçersiz kılar ve yanlış instruction'ın yürütülmesini engeller. Bu basit branch kontrol yöntemi, doğruluğu garanti altına alırken, alınan her branch için bir çevrimlik bir ceza oluşturur.

next_pc_ena_o ve next_pc_o çıkışları register'lanarak Execution aşamasında output'a iletildiği için, branch alınması durumunda 2 çevrim kaybedilir. Eğer bu sinyaller fetch aşamasına kombinasyonel olarak iletilmiş olsaydı, bu kayıp 1 çevrime indirilebilirdi. Ancak bu durumda kombinasyonel yol gecikmesi artacağı için sistemin clock performansı olumsuz etkilenebilirdi ve bu nedenle mevcut tasarım tercih edilmiştir.

2.1.3.4 Sonuç

Pipelined bir işlemci mimarisinde, ardışık talimatlar aynı anda farklı aşamalarda bulunduğu için veri ve kontrol bağımlılıkları kaçınılmaz hale gelir. Bu durum, "hazard" olarak adlandırılan hatalı veri işleme ya da yanlış kontrol akışı gibi sorunlara yol açar. Eğer bu hazard'lar tespit edilip uygun şekilde elimine edilmezse, işlemci yanlış sonuçlar üretebilir veya ciddi performans kayıpları yaşanabilir. Bu nedenle, veri ve kontrol hazard'larını yönetmek işlemci tasarımında kritik öneme sahiptir.

Chapter 3

TASARIM VE TEST SÜRECİNDE YAŞANAN ZORLUKLAR

Bu proje kapsamında ilk kez pipelined bir mimariyi uyguladığım için, mimarinin genel çalışma mantığını kavramam başlangıçta zaman aldı. Özellikle, talimatların aynı anda farklı aşamalarda çalıştığı bu yapıda veri akışını ve bağımlılıkları doğru şekilde yönetmek başlangıçta zorlayıcıydı.

İmplementasyon sırasında en çok zorlandığım kısımlardan biri, veri hazard'larının doğru şekilde ele alınmasıydı. Özellikle bir komuttan sonra gelen ilk iki komutun da aynı kaynağa ihtiyaç duyması durumunda, testlerde hatalar aldım. Bu durumun çözümünün, öncelikle execute aşamasından forwarding yapılması gerektiğini fark etmem zaman aldı. Bu öncelik kuralını uyguladıktan sonra, veri hazard'larına bağlı hatalar çözüldü.

Test aşamasında ise, riscv-gnu-toolchain ve spike araçlarının kurulumu sırasında ciddi zorluklar yaşamama rağmen bu araçları kurdum ve test dosyaları oluşturdum.

Store komutları ile ilgili testlerde karşılaşılan bir diğer durum ise, verilerin belleğe doğru şekilde yazılmasına rağmen testbench dosyasında yapılan bellek görüntülemelerinde yanlış değerlerin gösterilmesidir. Belleğe store edilen veriler, GTKWave üzerinden gözlemlendiğinde doğru şekilde yazıldığı açıkça görülmektedir. Ancak, testbench içerisinde belirli bir bellek aralığını komut penceresinde yazdırmak için kullanılan display işlemi, verileri hatalı şekilde göstermektedir. Bu nedenle testbenchte farklı bir yaklaşım kullanılmıştır. Uygulanan bütün testlerde Spike aracının ürettiği çıktı ile tasarlanan CPU'nun çıktıları tam örtüşmüştür.

Chapter 4

SONUÇ

Bu proje kapsamında, RISC-V mimarisine uygun, 5 aşamalı pipelined bir işlemci tasarımı gerçekleştirilmiş ve veri ile kontrol hazard'ları uygun forwarding, stall ve flush mekanizmalarıyla yönetilmiştir. Tasarım süreci boyunca mimarinin işleyişi anlaşılmış, karşılaşılan teknik zorluklar sistematik olarak çözülmüştür. Bu proje, işlemci mimarisi alanında hem teorik bilgimi pekiştirmeme hem de pratik tasarım ve doğrulama deneyimi kazanmama katkı sağlamıştır.