ÖNSÖZ

Günümüzde işlemci tasarımı ve üretimi dünyada en değerli sektörlerden biridir. Özellikle tam bağımsızlığın son derece önemli olduğu savunma sanayii gibi alanlarda entegre devrelerin içeriğine tamamen hakim olunması elzemdir. Hızla gelişen Türkiye'nin en önemli ihtiyaçlarından birisi de artık kendi işlemcilerini üreterek bu alanda da bağımsız hale gelmesidir. Endüstri ve eğitim için açık kaynak olarak University of California Berkeley tarafından geliştirilen RISC-V komut seti ile üreteceğimiz verimli ve endüstriyel standarta sahip işlemcileri savunma sanayii, yerli telefon, yerli bilgisayar, yerli geliştirme kartları gibi ürünlerde kullanmak üzere üreterek ülkemizin kendi işlemcisini üretme eşiğini aşmasını ve bu alanda da bağımsız olması hedeflenmektedir.

İÇİNDEKİLER

SİMGELER VE KISALTMALAR LİSTESİ

RISC	İndirgenmiş Komut Seti Bilgisayarı (Reduced Instruction Set Computer)
CISC	Karmaşık Komut Seti Bilgisayarı (Complex Instruction Set Computer)
ROM	Salt Okunur Bellek (Read Only Memory)
RAM	Rastgele Erişimli Bellek (Random Access Memory)
CPU	Merkezi İşlem Birimi (Central Processing Unit)
HDL	Donanım Tanımlama Dili (Hardware Description Language)
ALU	Aritmetik Mantık Birimi (Arithmetic Logic Unit)
RegFile	Yazmaç Dosyası (Register File)
VCD	Değer Değişim Çıktısı (Value Change Dump)
PC	Program Sayacı (Program Counter)

ŞEKİLLER LİSTESİ

TABLOLAR LİSTESİ

ÖZET

Anahtar kelimeler: İşlemci Tasarımı, Bilgisayar Organizasyonu, RISC-V

Bu çalışmada Verilog HDL dilinde RV64IM bir RISC-V çekirdeği tasarımı yapılması ve simüle edilmesi hedeflenmiştir.

Tasarlanan işlemci çekirdeği çok çevrimli, 5 aşamalı(5-stage) boru hattına(pipeline), 32 adet 64 bit tam sayı yazmaçına(register) sahip olarak, RISC-V komut setinin RV64I(Integer) ve M(Multiplication) çekirdek eklentilerini içerecek şekilde tasarlanmıştır.

İşlemcinin boru hattı uygulamasında karşılaşılan Veri Bağımlılık Sorunu(Data Hazard) 4. veya 5. aşamalardan direkt olarak ihtiyaç olan verinin alınmasıyla çözülmüştür. Bellek okuma(load) buyrukunun hemen ardından gelen veri bağımlı buyrukta ise 1 çevrim beklenmiştir(stall). Dallanma buyruklarında karşılaşılan Denetim Sorunu(Control Hazard) ise hem veri yönlendirmesi hem de 2 çevrim bekleme(stall) yöntemleriyle çözülmüştür.

İşlemcinin bir çevrimi için beş aşama uygulanmıştır. Bunlar; Getir(Fetch), Çöz(Decode), Çalıştır(Execute), Bellek(Memory) ve Geri Yaz(WriteBack) aşamalarıdır.

Sonuç olarak RV64IM buyruklarını işleyen bir işlemci tasarımı elde edilmiş ve başarıyla test ve simüle edilmiştir. Bu tasarımın daha da geliştirilerek endüstriyel, savunma sanayii, kişisel bilgisayarlar gibi alanlarda kullanılması hedeflenmektedir.

1 GİRİŞ

Günümüzde işlemci tasarımı ve üretimi dünyada en değerli sektörlerden biridir. Özellikle tam bağımsızlığın son derece önemli olduğu savuma sanayii gibi alanlarda entegre devrelerin içeriğine tamamen hakim olunması elzemdir. Hızla gelişen Türkiye'nin en önemli ihtiyaçlarından birisi de artık kendi işlemcilerini üreterek bu alanda da bağımsız hale gelmesidir.

Endüstri ve eğitim için açık kaynak olarak University of California Berkeley tarafından geliştirilen RISC-V komut seti ile üreteceğimiz verimli ve endüstriyel standarta sahip işlemcileri savunma sanayii, yerli telefon, yerli bilgisayar, yerli geliştirme kartları gibi ürünlerde kullanmak üzere üreterek ülkemizin kendi işlemcisini üretme eşiğini aşmasını ve bu alanda da bağımsız olması hedeflenmektedir.

1.1 RISC Nedir?

RISC, Reduced Instruction Set Computer, yani İndirgenmiş Komut Seti Bilgisayarı, bir işlemci mimari çeşididir. Bir diğer işlemci mimari çeşidi ise CISC, Complex Instruction Set Computer yani Karmaşık Komut Seti Bilgisayarı'dır.

Bunlar arasındaki farkı kısaca ele alırsak, CISC işlemciler bir buyruğu çalıştırabilmek için birden fazla çevrim(cycle) harcayabilir, birbirinden bağımsız donanımları tek komutta kullanabilir, örneğin tek komutta önce sabit diskten veri okuyup sonra geçici belleğe bu diski yazabilir. Mesela aşağıdaki Assembly komutunu ele alalım:

```
MULT adres1, adres2
```

Bu komut Multiplication yani Çarpma işlemini temsil ediyor. RAM'de adres2'de tutulan sayıyı adres1'de tutulan sayı ile çarpıp adres1'e kaydediyor. Görüldüğü üzere

tek bir işlemci komutu ile A ve B şeklinde iki sayıyı çarpıp tekrar A'ya yazabiliyoruz.

Bu Assembly komutunun C dilindeki karşlığı aşağıdaki gibi olacaktır:

```
// a RAM'de adres1'de tutulsun
// b RAM'de adres2'de tutulsun
a = a * b
```

CISC'te tek bir komutla yazılan MULT işlemi aslında birkaç işlemci döngüsüyle gerçekleşir. Örneğin MULT komutu 4 çevrim gerektirsin ve bizim elimizde 4Hz'lik bir işlemcimiz olsun. Bu durumda MULT komutu tek bir komut olmasına rağmen, 4 çevrim sonra sona ereceği için 1 saniyede ancak 1 tane MULT komutu çalıştırabiliriz.

Peki bu durum RISC'te nasıl? RISC mimarisinde bütün komutlar sadece belirli ve indirgenmiş biri iş gerçekleştirirler. Mesela bir komut aynı anda hem bellekten veri okuyup hem de bir hesaplama işlem yapıp daha sonra da bu sonucu sabit diske kaydetmez. Bellekten veri okuyan komut ayrı, hesaplama işlemini yapan komut ayrı, sabit diske yazan komut ayrıdır. Genelde komutlar 1 çevrimde biter ve tek bir iş yaparlar. Yukarıda CISC ile örneğini verdiğimiz çarpma işlemini RISC mimarisinde gerçekleştirmek için 4 ayrı komuta ihtiyacımız var:

```
lw  a0, 0(x0)   ; A'yı RAM'deki 0 adresinden oku
lw  a1, 3(x0)   ; B'yi RAM'deki 3 adresinden oku
mul a0, a0, a1  ; a0 ile a1'i çarpıp a0'a yaz.
sw  a0, 0(x0)   ; a0 değerini 0 adresine kaydet
```

Gördüğümüz gibi CISC'te 1 komut 4 çevrimde çalışırken RISC'te 4 komut 4 çevrimde çalıştı. Sonuç olarak iki mimari de birbirinden çok farklı değilmiş gibi gözükse de RISC mimarisinin CISC mimarisine kıyasla işlem adımlarının parçalarına ayrılmış ve indirgenmiş olmasının pek çok avantajı var.

1.2 RISC ve CISC Mimarilerinin Farkları

RISC mimarisi CISC mimarisine göre aynı işi yapmak için daha fazla komut gerektiriyor gibi gözükse de bu işlem parçalama şeklinin getirdiği pek çok avantaj var.

RISC mimarisinin CISC mimarisine göre avantajlarını Tablo 1'deki gibi sıralandırabiliriz.

Table 1: RISC'in avantajları

RISC artıları	CISC eksileri
Tek çevrimde gerçekleşen basit komutlar, donanımsal olarak daha az transistör ile gerçekleştirilir	Kompleks komutları icra eden donanımlar daha fazla transistör gerektirir ve karmaşık bir mimariye sahiptir
Her komut tek çevrimde gerçekleştiği için Boru Hattı(Pipelining) uygulaması daha kolaydır	Komutlar farklı çevrimlerde bittiği için Boru Hattı uygulamak zorlaşır. Her komutu tespit için ve bu komutların farklı uzunluklarda olmasıyla gelen karmaşıklık için daha fazla devreye ihtiyaç duyulur.
Az sayıda işlemci komutu ile tüm işlemler gerçekleştirilir. Sade ve kolay bir komut setine sahiptir. Örneğin RISC-V'ın RV64IMAFD komut setinde sadece 159 komut bulunur.	Çok fazla işlemci komutu vardır ve komut seti karmaşıktır. İntel x86-64 mimarisinde yaklaşık 3683 komut bulunur. [1]
Komutlar basit olduğu için donanımsal olarak gerçeklenirken kolayca tasarlanabilirler ve az yer kaplarlar.	Komutlar karmaşık olduğu için donanımsal yapılar daha karmaşık devreler gerektirir ve daha çok yer kaplar.
Transistör sayısı az ve donanımı basit olduğu, çalışma frekansı da düşük olduğu için çok daha az enerji tüketir (4-5W)	Genelde tükettiği enerji miktarı fazladır. (90-100W)

RISC mimarisinin CISC mimarisine göre eksilerini ise Tablo 2'deki gibi sıralandırabiliriz.

Table 2: RISC'in dezavantajları

RISC eksileri	CISC artıları
Derleyici güçlü ve gelişmiş olmalıdır, karmaşık kodları indirgenmiş temel komutlara çevirebilmelidir	Derleyicinin işi kolaydır çünkü karmaşık işlemlerin direkt olarak donanımsal karşılıkları vardır
Daha fazla RAM tüketir	Daha az RAM tüketir
Daha fazla kod boyutu kaplar (aynı işi yapmak için CISC'te 1, RISC'te 4 satır kod yazdık, bu da RISC için derlediğimiz program diskte 3 kelime daha fazla yer kaplar demek)	Daha düşük kod boyutu kaplar

Eskiden işlemciler oldukça sınırlı RAM ve hafıza ile çalıştıklarından CISC mimarisi biraz daha fazla ön plandaydı. Fakat günümüzde RAM ve hafıza bellek boyutları oldukça yüksek olduğu için RISC'in enerji verimliliği, sadeliği, tasarım kolaylığı ve performansı onu daha çok ön plana çıkarıyor.

Özetle derleyicileriniz çok az daha gelişmiş ise, RAM ve hafıza bellek kaynağı sıkıntınız yok ise RISC daha avantajlı diyebiliriz. [2]

1.3 RISC-V(risk fayf) Nedir?

Figure 1.1: RISC-V logosu. [3]

RISC-V, RISC prensipleriyle hazırlanmış, University of California Berkeley tarafından geliştirilen açık kaynak bir Komut Seti Mimarisidir(Instruction Set Architecture). "5" manasına gelen "-V" takısı üniversitenin tasarladığı 5. RISC mimarili komut seti olması sebebiyle eklenmiştir.

RISC-V komut setini işleyen işlemcileri isteyen herkes ister kişisel, ister ticari, ister akademik olmak üzere herhangi bir lisans ücreti ödemeden üretebilir.

Açık kaynak olarak geliştirilmiş RISC-V işlemci çekirdeklerinin tasarımları herkese açık olarak paylaşılmış github.com/riscv hesabı altında paylaşılmış durumdadır: https://github.com/riscv/riscv-cores-list

1.4 RISC-V Komut Tipleri ve Yazmaçlar

32-bit RISC-V instruction formats

Format	Bit																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bits):** Partially specifies which of the 6 types of *instruction formats*.
- **funct7, and funct3 (10 bits):** These two fields, further than the *opcode* field, specify the operation to be performed.
- **rs1 (5 bits):** Specifies, by index, the register containing first operand (i.e., source register).
- **rs2 (5 bits):** Specifies the second operand register.
- **rd (5 bits):** Specifies the destination register to which the computation result will be directed.

Figure 1.2: RISC-V buyruk tipleri ve bit dizilimleri. [4]

RISC-V komut seti üzerindeki komutlar(instruction) 32 bittir. Bu komutların içeriği komutun tipine göre 6 farklı şekilde dizilir. RISC-V komut tiplerini şu şekilde sıralayabiliriz:

1. **Yazmaç-Yazmaç** (Register-Register): Değerler rs1 ve rs2'de belirtilmiş yazmaçlardan okunup tekrar rd'de belirtilen yazmaça yazılır.
2. **Direkt** (Immediate): Değerlerin bir tanesi yine rs1 yazmaçından, diğeri ise direkt olarak bir sayıdan alınır ve sonuç yine rd yazmaçına yazılır. Direkt sayı için ayrılmış bit genişliği 12'dir.
3. **Üst Direkt** (Upper Immediate): 32 bitlik sabit sayılarla işlem yapabilmek için Direkt komutların sağladığı 12 bitlik direkt sayıya ek olarak üst kısımda geriye kalan 20 bitlik değeri belirtmek için kullanılır. Böylece 12 + 20 bit ile 32 bitlik sabit bir sayı üzerinden işlem yapılabilir.
4. **Kayıt** (Store): rs2 yazmaçındaki değer RAM bellekte rs1+direkt adresine yazılır.

5. **Dallanma** (Branch): rs1 ile rs2 yazmaçlarındaki değerler kıyaslanır ve kıyaslama sonucu doğru ise PC(Program Counter) yazmaçına elde edilen değer eklenir. Böylece programda başka bir yere koşullu dallanılmış olur. Ayrıca dallanma komutlarındaki direkt değer 2 ile çarpılarak(en düşük kısmına 1 biti eklenerek) işlenir. Böylece PC ikinin katları olan sayılara dallanbilir. (Standart bir komutun 4 byte ve sıkıştırılmış komutların 2 byte olması hiçbir zaman atlanmayacak tek sayılı adresler oluşturur (1, 3, 21 gibi). PC'yi tüm değerlerle değil de 2 ve katları değerler ile atlatmak atlama kapasitesini 2 katına çıkarır.)
6. **Atlama** (Jump): Jump and Link(JAL) komutu PC'deki bir sonraki komutun(PC+4) değerini rd yazmaçına kaydettikten sonra, PC'ye direkt değeri ekler. Böylece atlanılan yerin adresi kayıt edilerek koşulsuz olarak başka bir yere atlanır. Atlama komutlarında da direkt değer 2 ile çarpılarak(en düşük kısmına 1 biti eklenerek) işlenir. Böylece bir programda 20+1 bitlik bir değer ile(±1 MiB) gibi yüksek bir miktarda ileri veya geri atlayabilir.

RISC-V temel olarak 32 adet tam sayı yazmaçına(register) sahiptir. Bunlar x0'dan x31'e kadardır. x0 yazmaçı sabit olarak 0 değerine sahiptir ve üzerine yazılamaz. Geriye kalan x1-x32 arası 31 adet yazmaç belirli yerlerde kullanılmak üzere Şekil 1.3'teki gibi ayrılmıştır.

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

Figure 1.3: RISC-V komut setindeki yazmaçlar ve ayrıldıkları alanlar.[4]

1.5 Türkiye'de RISC-V

RISC-V 2018-2021 yıllarında Türkiye'de büyük bir ivme yakalamış durumda. Bugün içlerinde Türkiye'den ASELSAN ve SystemPark(DeepControl) firmalarının da bulunduğu dünya çapında 100'den fazla stratejik ve iş ortakları mevcuttur: https://riscv.org/members/

1.6 Verilog Nedir?

Verilog bir HDL(Hardware Description Language) yani Donanım Tanımlama Dilidir. İşlemciyi oluşturacak mantıksal devreleri tasarlamak için bir HDL olan Verilog ve bu dili derlemek, simüle etmek ve devre sentezinde kullanmak için de açık kaynak iverilog(Icarus Verilog) derleyicisi kullanılmıştır.

Verilog dilinde bir bilgisayar programı yazar gibi kod yazarak istenilen mantıksal devreler oluşturulabilir. Örneğin iki giriş alan ve bu girişleri VE mantık işlemine tabi tutup çıkışını dışarıya veren bir devre aşağıdaki gibi tasarlanabilir:

```
module VE_Devresi (output Y, input A, input B);
	assign Y = A & B;
endmodule
```

1.7 Icarus Verilog Derleyicisi

Verilog donanım tasarım dili ile yazdığımız kodları çalıştırabilmek, simüle edebilmek veya devreler sentezlemek için bir derleyiciye ihtiyacımız vardır. Bunun için piyasada ücretli ve ücretsiz pek çok verilog derleyicisi mevcuttur. Bu projede ise açık kaynak ve ücretsiz olan iverilog derleyicisi tercih edilmiştir. (http://iverilog.icarus.com/)

iverilog derleyicisi Pardus GNU/Linux üzerinde aşağıdaki komut ile kurulabilir:

```
$ sudo apt install iverilog
```

Örneğin devre.v adlı bir dosyamız olsun:

```
module hello;
	initial begin
		$display("Hello, World");
		$finish;
	end
endmodule
```

Bu dosyayı aşağıdaki komutla iverilog derleyicisinde derleyebiliriz:

```
$ iverilog devre.v -o devre
```

Derlenme sonucu elde edilen devre dosyası simüle edilmeye hazır bir vvp assembly dosyasıdır. Bu dosyayı çalıştırmak için ise vvp komutunu kullanılmaktadır:

```
$ vvp devre
Hello, World
```

1.8 GTKWave

Verilog ile yazdığımız modüllerin ve devrelerin doğru ve istenilen şekilde çalıştığını test edebilmek için testbench modülleri yazılmaktadır. Bu modüllerin çalışma biçimini incelemek için $display komutu ile konsol çıktısı alınabilir.

Modüllerin çalışma şekillerini görsel olarak görüntülemek için iverilog ile derlenecek verilog dosyasında $dumpfile(“modul.vcd”); $dumpvars(0, Modul); komutları ile .vcd formatında Değer Değişim Çıktısı(Value Change Dump) dosyası elde edilir.

Bu dosyanın gösterimi ve dalgaların incelenebilmesi için ise yine açık kaynak ve ücretsiz olan GTKWave uygulaması tercih edilmiştir.

GTKWave uygulaması Pardus GNU/Linux üzerinde aşağıdaki komut ile kurulabilir:

```
$  sudo apt install gtkwave
```

GTKWave uygulamasının çalışma biçimine örnek olarak yukarıda yazdığımız basit bir VE modülünün dalga çıktılarını inceleyelim:

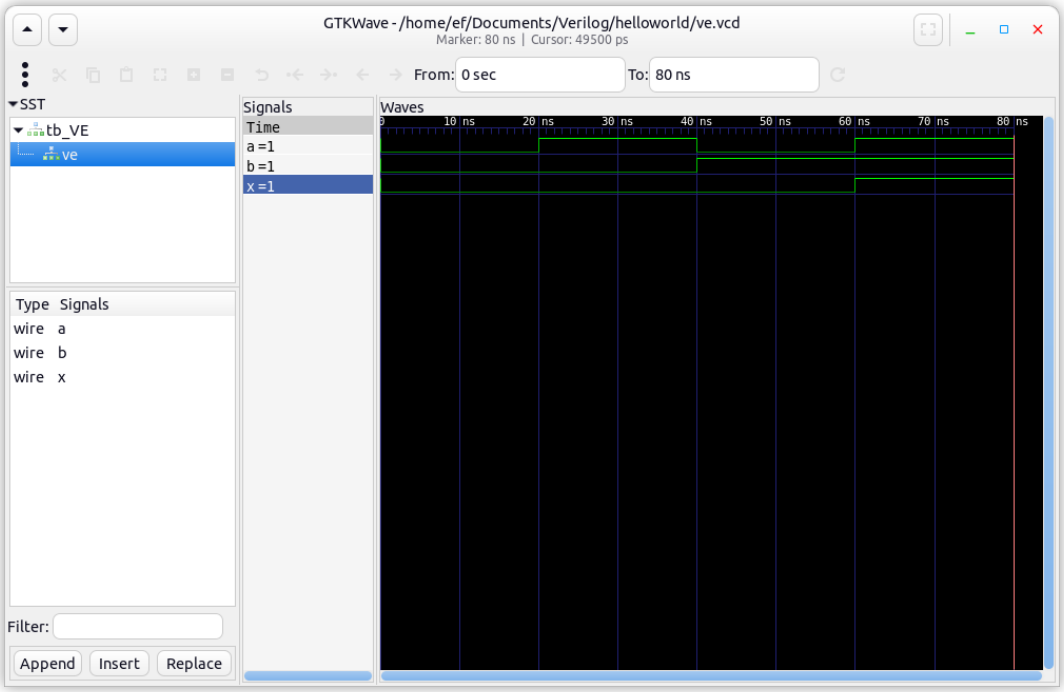

Figure 1.4: GTKWave ile VE modülü .vcd çıktısının incelenmesi.

Yukarıda örnek olarak verdiğimiz VE modülünü ve.v dosyasında, bu modülü test edeceğimiz testbench modülünü ise tb_VE.v dosyasında oluşturduk.

Örnekte kullanılan tb_VE.v dosyasının içeriği:

```
`timescale 1ns / 100ps
`include "ve.v"

module tb_VE();
	wire x;
	reg a;
	reg b;

	// "VE" tipinde "ve" adında bir eleman oluşturduk.
	VE ve(x, a, b);

	initial begin
		a = 0; b = 0;
		#20; // 20 nanosaniye bekle
		a = 1; b = 0;
		#20;
		a = 0; b = 1;
		#20;
```

```
            a = 1; b = 1;
            #20;

            $dumpfile("ve.vcd");
            $dumpvars(0, tb_VE);
      end
endmodule
```

Aşağıdaki komut ile ve.vcd çıktısı elde edilmiştir:

```
$ iverilog tb_ve.v -o tb_ve
$ vvp tb_ve
VCD info: dumpfile ve.vcd opened for output.
```

2 EK MODÜLLER

İşlemci Çekirdeğini tek başına bir dosya şeklinde yazmak istesek çok uzun ve karışık bir dosya halini alabilirdi. Bu yüzden tasarladığımız işlemcide modülerlik esas alınarak belirli bir işi yapan ve işlemcinin iç devresinden bağımsız çalışabilen parçaların ayrı bir modül olması düsturu esas alınmıştır.

Örneğin sürekli kullanılacak olan kodlayıcılar(encoder) bir modül haline getirilmiştir. Bir diğer örnek ise direkt(immediate) buyruk tipli komutlardan direkt değerin kendisini elde etmek için de yine bir modül oluşturulmuştur. Bilgisayar organizasyonunda kullanılacak ve işlemcinin dışında olan RAM ve ROM modülleri de ayrıca tasarlanmıştır.

1.9 ROM

ROM modülü, işlemcinin adresini belirterek istediği program buyruklarını(instruction) işlemciye getirecek(fetch) olan modüldür. Sadece okunabilirdir, veriler işlemci tarafından değiştirelemez.

Modülde tanımlı bellek boyutu 1024 hücredir ve her hücre 32 bit(4 byte) genişliğe sahiptir. Yani 4KB veri kapasiteli bir bellek tanımlanmıştır.

ROM.v dosya içeriği:

```
module ROM(
	input [9:0] ADDRESS,
	output [31:0] DATA
);
	// 10-bit adres, 32-bit hücre boyutu
	reg [31:0] memory [1023:0];
	assign DATA = memory[ADDRESS];
```

```
        initial begin
                memory[0] = 32'h00000013; // nop (add x0 x0 0)
                memory[1] = 32'h00100093; // addi x1 x0 1
                memory[2] = 32'h00100313; // addi x6 x0 1
                memory[3] = 32'h00400613; // addi x12 x0 4
                memory[4] = 32'h0060A023; // sw x6 0(x1)
                memory[5] = 32'h0000A303; // lw x6 0(x1)
                memory[6] = 32'h00158593; // addi x6 x6 1
                memory[7] = 32'h0060A023; // sw x6 0(x1)
                memory[8] = 32'hFEC34AE3; // blt x6 x12 -12
                memory[9] = 32'h03700413; // addi x8 x0 55
                memory[10] = 32'h00800433; // add x8 x0 x8
                memory[11] = 32'h00140413; // addi x8 x8 1
                memory[12] = 32'h00000013; // nop (add x0 x0 0)
        end
endmodule
```

ROM.v modülü içerisinde saklanan salt-okunur(read-only) değerler RISC-V Assembly diliyle derlenmiş bir programın onaltılık taban(hexadecimal) biçimindeki karşılıklarıdır.

İşlemcinin ROM'dan okuyup buyruk olarak çalıştıracağı bu onaltılık sayılar tabandaki sayılar aşağıda belirtilmiş Program.s dosyasından derlenmiştir:

```
start:
        nop
        addi x1, x0, 1
        addi x6, x0, 1
        addi x12, x0, 4
        sw x6, 0(x1) # x6 -> R2 Veri Bağımlılığı (WriteBack)
loop:
        lw x6, 0(x1)
        addi x6, x6, 1      # Load Stall & x6 -> R1 Veri Bağımlılığı
        sw x6, 0(x1)
        blt x6, x12, loop   # Control Hazard & x6 -> R1 Bağ. (WriteBack)
data_dep_test:
        addi x8, x0, 55
        add x8, x0, x8      # x8 -> R2 Veri Bağımlılığı
        addi x8, x8, 1      # x8 -> R1 Veri Bağımlılığı
finish:
        nop
```

Bu program aynı zamanda boru hattı uygulamasında oluşacak şu sorunların da test edilmesini içeren buyruklara sahiptir:

- Bir veya iki önceki buyruğun sonucuna bağımlılıkla oluşan Veri Sorunu(Data Hazard)
- Dallanma(Branch) buyruklarıyla oluşan Denetim Sorunu(Control Hazard)
- Bellekten veri yükleme(Load) buyruğunun sonucuna bağımlılık ile oluşan Yükleme Gecikmesi(Load Stall)

Program.s dosyasının derlenmesiyle elde edilen onaltılık taban biçimindeki assembly komutları:

```
PC          Komut       Tanım
00:         0x00000013  addi x0 x0 0
04:         0x00100093  addi x1 x0 1
08:         0x00100313  addi x6 x0 1
12:         0x00400613  addi x12 x0 4
16:         0x0060A023  sw x6 0(x1)
20:         0x0000A303  lw x6 0(x1)
24:         0x00130313  addi x6 x6 1
28:         0x0060A023  sw x6 0(x1)
32:         0xFEC34AE3  blt x6 x12 -12
36:         0x03700413  addi x8 x0 55
40:         0x00800433  add x8 x0 x8
44:         0x00140413  addi x8 x8 1
48:         0x00000013  addi x0 x0 0
```

ROM modülünün istenilen şekilde çalışıp çalışmadığını test etmek için bir testbench modülü yazılmış, testbench modülü içinde bir ROM modülü oluşturup bu modüle girdiler verip çıktıları kontrol edilmiştir.

tb_ROM.v dosya içeriği:

```
`include "ROM.v"
`timescale 1ns / 100ps
```

```
module tb_ROM;
	reg [9:0] ADDRESS;
	reg [31:0] DATA_OUT;

	ROM rom(ADDRESS, DATA_OUT);

	initial begin
		ADDRESS = 0; #20;
		ADDRESS = 1; #20;
		ADDRESS = 2; #20;
		ADDRESS = 3; #20;
		ADDRESS = 4; #20;
		ADDRESS = 5; #20;
		ADDRESS = 20; #20;
	end

	initial begin
		$dumpfile("rom.vcd");
		$dumpvars(0, tb_ROM);
	end
endmodule
```

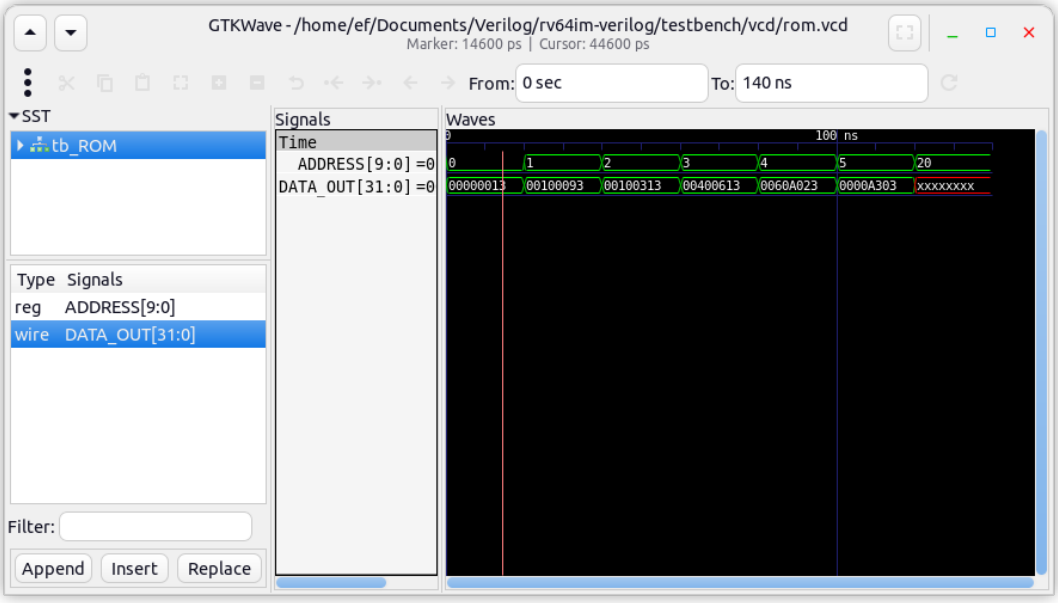

Figure 2.1: tb_ROM.v modülünün dalga analizini yapabileceğimiz rom.vcd dosyasının GTKWave ile görüntülenmesi.

Bu modül iverilog derlenmiş ve tb_ROM.v dosyasındaki $dumpfile ve $dumpvars komutları sayesinde rom.vcd dosyasına dalga çıktılarını çıkartılmış daha sonra

GTKWave uygulaması aracılığıyla oluşturulan rom.vcd dosyasının dalga analizi Şekil 2.1‘deki gibi görüntülenmiştir.

Şekil 2.1‘de görüldüğü üzere ROM modülümüz test edilmek üzere verilen ADDRESS girişlerine beklenen DATA_OUT çıkış değerlerini vermiştir. Böylece modülümüzün düzgün bir biçimde çalıştığı da gözlemlenmiştir.

1.10 RAM

RAM modülü programın çalışması esnasında geçici bellekte veri saklama veya okuma işlemlerinde kullanılacak modüldür. Her biri 64 bit genişliğe sahip 1024 adet hücreden oluşturulmuştur. ROM modülü hücreleri buyruklar 32 bit olduğu için 32 bit genişliğindeyken RAM modülü hücreleri saklanabilen verilen 64 bit olabileceği için 64 bit genişliğe sahiptir.

RAM modülü içindeki hücreler ROM modülünden farklı olarak hem yazılabilir hem okunabilirdir hücrelerdir. WRITE_ENABLE girişi “1” değerini almasıyla DATA_IN girişine verilen 64 bitlik girdi ADDRESS girişine verilen değerdeki adreste bulunan hücreye yazılmaktadır. WRITE_ENABLE girişinin “0” olduğu durumlarda ise ADDRESS adresindeki hücrenin değeri okunup DATA_OUT ile çıktı olarak verilmektedir.

RAM.v dosya içeriği:

```
module RAM(
	input [9:0] ADDRESS,
	input [63:0] DATA_IN,
	input WRITE_ENABLE,
	input CLK,

	output [63:0] DATA_OUT
);

	reg [63:0] memory [1023:0];
```

```
	initial begin
		for(int i=0; i<1024; i=i+1) begin
			memory[i] <= 0;
		end
	end

	assign DATA_OUT = memory[ADDRESS];

	always @(posedge CLK) begin
		if (WRITE_ENABLE)
			memory[ADDRESS] <= DATA_IN;
	end
endmodule
```

RAM modülümüzün de doğru çalışıp çalışmadığını test etmek için bir testbench modülü yazılmıştır.

tb_RAM.v dosya içeriği:

```
`include "RAM.v"
`timescale 1ns / 100ps

module tb_RAM;
	reg [9:0] ADDRESS;
	reg [63:0] DATA_IN;
	reg WRITE_ENABLE;
	reg CLK;
	reg [63:0] DATA_OUT;

	RAM ram(ADDRESS, DATA_IN, WRITE_ENABLE, CLK, DATA_OUT);

	initial begin
		// 1 Adresine 55 verisini yaz.
		ADDRESS = 1;
		DATA_IN = 55;
		WRITE_ENABLE = 1;
		CLK = 1; #20; CLK = 0; #20;

		// 2 Adresine 99 verisini yaz.
		ADDRESS = 2;
		DATA_IN = 99;
		WRITE_ENABLE = 1;
		CLK = 1; #20; CLK = 0; #20;
```

```
            // 1 Adresini oku.
            ADDRESS = 1;
            WRITE_ENABLE = 0;
            DATA_IN = 0;
            CLK = 1; #20; CLK = 0; #20;

            // 2 Adresini oku.
            ADDRESS = 2;
            WRITE_ENABLE = 0;
            DATA_IN = 0;
            CLK = 1; #20; CLK = 0; #20;
        end

        initial begin
            $dumpfile("ram.vcd");
            $dumpvars(0, tb_RAM);
        end
endmodule
```

RAM modülünü test etmek için yazdığımız testbench modülünün iverilog ile derlenmesi sonucu elde ettiğimiz ram.vcd dosyasının dalga analizi şekildeki gibidir:

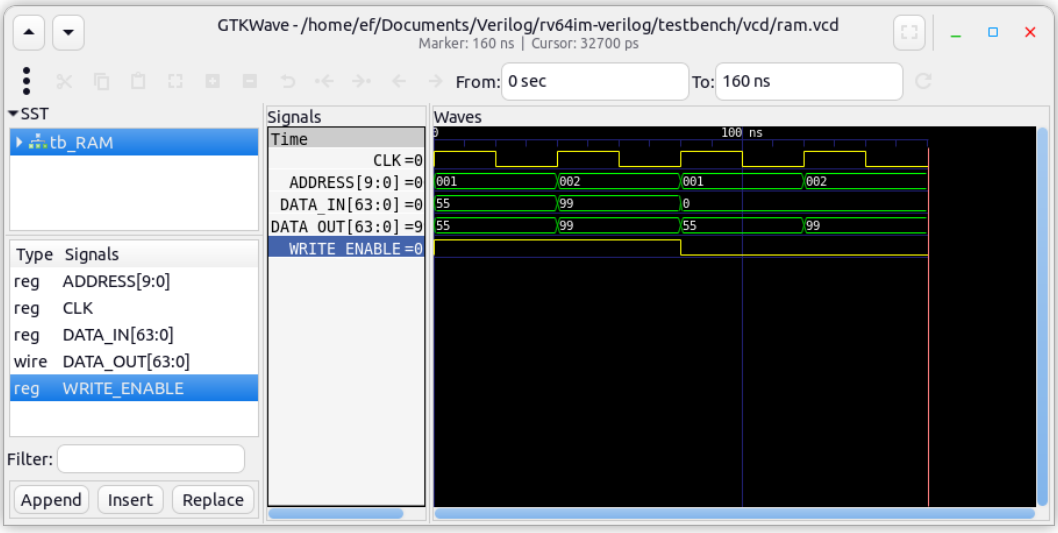

Figure 2.2: tb_RAM.v modülünün dalga analizini yapabileceğimiz ram.vcd dosyasının GTKWave ile görüntülenmesi.

RAM modülümüzün de test için verilen girişlere beklenen çıkışları verildiği ve doğru çalıştığı görülmüştür.

1.11 RegFile

RegFile modülü işlemcinin içerisindeki yazmaçların(register) tutulduğu, bu yazmaçlara okuma ve yazma işlemlerinin gerçekleşmesini sağlayan modüldür. R1(Source Register) ve R2(Source Register) içeriği okunacak yazmaçın hangi yazmaç olduğunu belirten 5 bitlik sayılardır. RD(Destination Register) ise yazılacak yazmaçın hangi yazmaç olduğunu belirten 5 bitlik bir sayıdır. (Şekil 2.5)

Okuma işlemi için gelen komutun tipine göre, R-Tip, S-Tip ve B-Tip komutlarda okunacak iki yazmaçın(R1 ve R2), I-Tip komutlarda ise okunacak tek yazmaçın(R1) kaçıncı yazmaçlar olduğu bilgisi bulunur. U-Tip ve UJ-Tip komutlar yazmaçlardan okuma yapmazlar. (Şekil 2.5)

Yazma işleminde ise gelen komutun tipine göre, R-Tip, I-Tip, U-Tip ve UJ-Tip komutlarda hangi yazmaça(RD) yazılacağı bilgisi bulunur. S-Tip ve B-Tip komutlar yazmaçlara yazma yapmazlar. (Şekil 2.5)

RegFile.v dosya içeriği:

```
module RegFile(
	input [4:0]R1,
	input [4:0]R2,
	input [4:0]RD,
	input [63:0]RD_DATA,
	input WRITE_ENABLE,

	output [63:0]R1_DATA,
	output [63:0]R2_DATA
);
	reg [63:0] REGISTERS[31:0]; // 64 Bit genişliğinde, 32 yazmaç

	integer i = 0;
	initial begin
		// Her yazmaça başlangıçta 0 değeri ataması
		for (i = 0; i< 32 ; i = i + 1) begin
			REGISTERS[i] <= 0;
		end
	end
```

```
        assign R1_DATA = REGISTERS[R1];
        assign R2_DATA = REGISTERS[R2];

        always @(*) begin
                // 0 numaralı yazmaça hiçbir zaman yazılamaz.
                if (WRITE_ENABLE == 1 && RD != 0)
                        REGISTERS[RD] <= RD_DATA;
        end
endmodule
```

RegFile modülü için yazdığımız tb_RegFile modülünün iverilog ile derlenmesi sonucu elde ettiğimiz regfile.vcd dosyasının dalga analizi şekildeki gibidir:

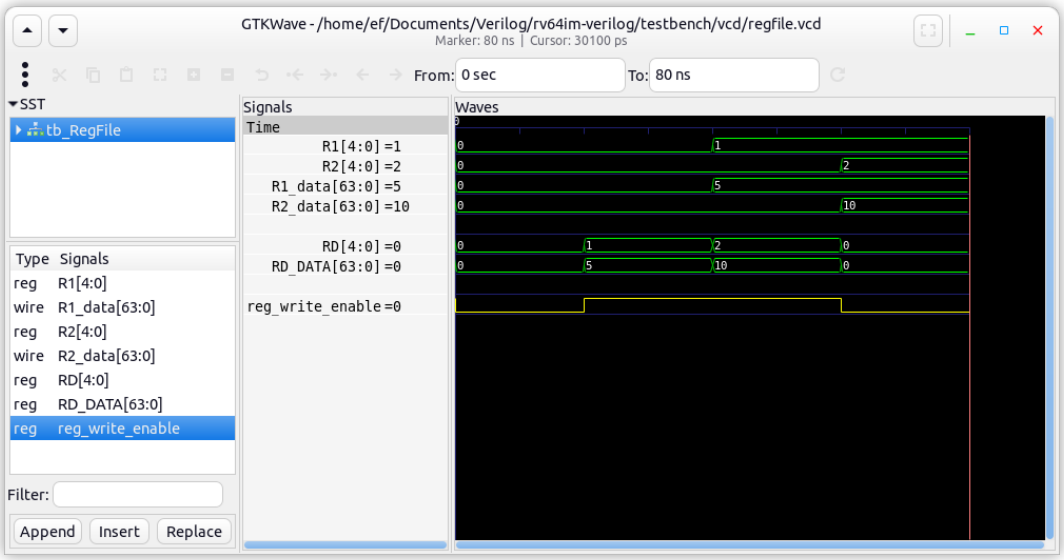

Figure 2.3: tb_RegFile.v modülünün dalga analizini yapabileceğimiz regfile.vcd dosyasının GTKWave ile görüntülenmesi.

RegFile modülümüzün de test için verilen girişlere beklenen çıkışları verildiği ve yazmaçlara okuma/yazma işlemlerinin doğru çalıştığı görülmüştür.

1.12 Kodlayıcılar (Encoders)

CPU'nun iç tasarımında ihtiyacımız olan modüllerden birisi de kodlayıcılardır. Kodlayıcılar girdilerindeki değeri "1" olan telin sırasının ikilik sistemde karşılığını veren devrelerdir.

Örneğin 4-2 lik bir kodlayıcı 4 tel ile giriş alır ve bu girişlerden “1” olan telin sırasının sayısal karşılığını çıktı olarak verir, yani 3. tel “1” diğer teller “0” değerini aldıysa, kodlayıcının çıkış değeri ikilik sistemde 3 sayısının karşılığı olan “11” değerini olacaktır.

Öncelikli Kodlayıcılar(Priority Encoder) ise birden fazla girişin 1 olması durumunda hangisinin seçileceği karışıklığını girişlere öncelik vererek gideren kodlayıcılardır. Örneğin 4-2’lik bir kodlayıcıda 0, 1, 2, 3 numaralı giriş tellerinden 2. tel ve 3. tel “1” olduğunda ikilik sistemde oluşacak olan çıktı “11” yani 3 olacaktır. Çünkü öncelikli kodlayıcılarda en yüksek bit değerine sahip girdiler önceliklidir ve daha düşük bit değerine sahip girdilerin değeri önemsenmez. Yani girişteki 2. tel “1” olduğu için çıktı “10” olmaz.

CPU’nun iç tasarımında 4-2’lik, 8-3’lük ve 16-4’lük kodlayıcılar kullanılmıştır. Örneğin 4-2’lik kodlayıcının Encoders.v dosyasındaki içeriği:

```
module Encoder_4 (
	input [3:0] in,
	output reg [1:0] out
);
	initial begin
		out = 0;
	end
	always @(in) begin
		casex (in)
			4'b1xxx : out = 3;
			4'b01xx : out = 2;
			4'b001x : out = 1;
			4'b0001 : out = 0;

			default: out = 0;
		endcase
	end
endmodule
```

Diğer 8-3 ve 16-4’lük kodlayıcılar da aynı şekilde en önemli bit 1, diğerleri x(don’t care) olacak şekilde Encoders.v dosyasının içerisinde tanımlanmıştır.

Kodlayıcı modüllerinin de doğru çalışıp çalışmadığını test etmek için tb_Encoders.v adında bir testbench modülü yazılmıştır.

tb_Encoders.v dosya içeriği:

```
`include "Encoders.v"
`timescale 1ns / 100ps

module tb_Encoders;
	reg [15:0] in = 0;
	reg [3:0] out4;
	reg [2:0] out3;
	reg [1:0] out2;

	Encoder_16 enc(in, out4);
	Encoder_8 enc8(in[7:0], out3);
	Encoder_4 enc4(in[3:0], out2);

	initial begin
		in = 16'b0000000000000000; #20;
		in = 16'b0000000000000001; #20;
		in = 16'b0000000000000010; #20;
		in = 16'b0000000000000100; #20;
		in = 16'b0000000000001000; #20;
		in = 16'b0000000000010000; #20;
		in = 16'b0000000000100000; #20;
		in = 16'b0000000001000000; #20;
		in = 16'b0000000010000000; #20;
		in = 16'b0000000100000000; #20;
		in = 16'b0000001000000000; #20;
		in = 16'b0000010000000000; #20;
		in = 16'b0000100000000000; #20;
		in = 16'b0001000000000000; #20;
		in = 16'b0010000000000000; #20;
		in = 16'b0100000000000000; #20;
		in = 16'b1000000000000000; #20;
		in = 16'b1000000000000001; #20;
		in = 16'b0000000100000001; #20;
		in = 16'b0000000000000011; #20;
		in = 16'b0000000000000000; #20;
	end

	initial begin
```

```
            $dumpfile("encoders.vcd");
            $dumpvars(0, tb_Encoders);
        end
end
```

tb_Encoders.v modülünün derlenmesi sonucu elde edilen encoders.vcd dosyasının dalga analizi şekildeki gibidir:

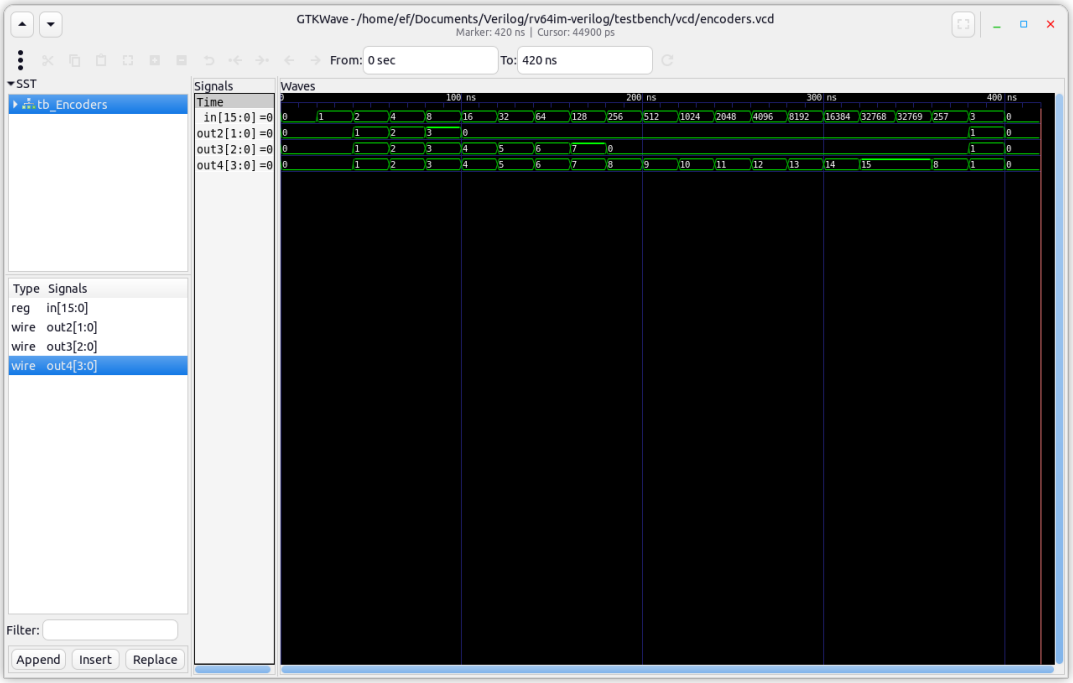

Figure 2.4: tb_Encoders.v modülünün dalga analizini yapabileceğimiz encoders.vcd dosyasının GTKWave ile görüntülenmesi.

Kodlayıcı modüllerimizin de test için verilen girişlere beklenen çıkışları vererek doğru çalıştığı gözlemlenmiştir.

1.13 Direkt Değer Ayrıştırıcısı (Immediate Value Extractor)

RISC-V komut setinde direkt tipteki buyrukların sahip olduğu sayısal değerler buyruğun tipine göre farklı şekillerde dağıtılmıştır. Dolayısıyla gelen buyruğun tipine göre direkt değerin ayrıştırılması gerekmektedir. Bu kısım işlemci içinde de tasarlanabilir olmasına rağmen ayrı bir modüle "buyruğu ver direkt değeri al" yaklaşımı daha sade bir tasarım olması için tercih edilmiştir.

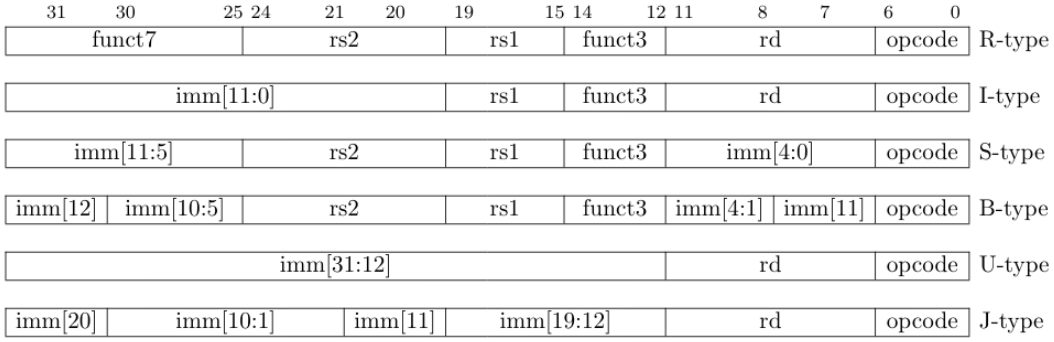

Figure 2.5: RISC-V buyruk tiplerine göre direkt değerlerin dağılımı. [5]

Şekil 2.5'de görüldüğü üzere direkt değerli buyrukların değerlerinin bulunduğu alanlar tiplerine göre farklılık göstermektedir. Bu tiplere göre verilen buyruktaki direkt değeri ayrıştırıp işlemlerde kullanmak üzere 64 bitlik bir sayı haline getirmek için ImmediateExtractor modülü tasarlanmıştır. Direkt değerler en yüksek değerli bitlerine Şekil 2.6'daki gibi genişletilmiştir(sign extended).

Örneğin 5 bitle 01000 şeklinde gösterilen 8 sayısı 000...001000 olarak 64 bite genişletilirken 5 bitle 11000 şeklinde gösterilen -8 sayısı en yüksek bitteki değerle genişletilerek 111...111000 şeklinde 64 bitlik bir sayıya çevirilmiştir. Böylece sayının işareti korunmuştur.

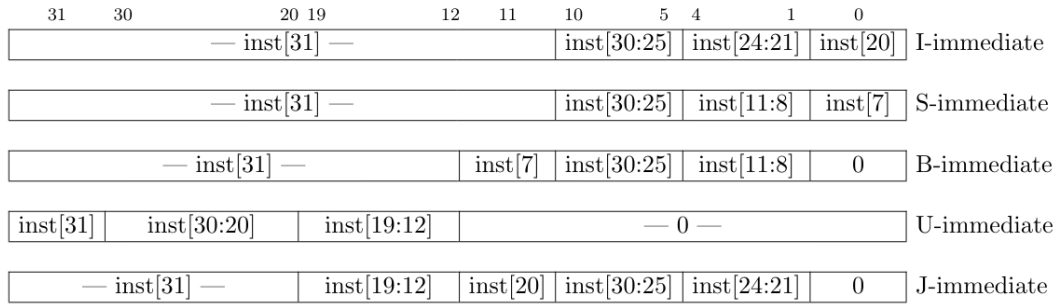

Figure 2.6: Direkt değerlerin tiplerine göre 32 bitlik sayılara genişletilmesi için kullanılan yöntem. 64 bitlik mimaride ise tek fark yüksek değerlikli bitlerin inst[31] bitiyle 64 bite genişletilmesidir.[5]

ImmediateExtractor.v dosya içeriği:

```
module ImmediateExtractor(
       input [31:0] INSTRUCTION,
       input [2:0] SELECTION,
```

```
        output reg signed [63:0] VALUE
);
        // Komutlarda alınabilecek tüm direkt değerleri tanımlayalım
        wire [11:0] IMM_11_0       = INSTRUCTION[31:20];
        wire [19:0] IMM_31_12      = INSTRUCTION[31:12];
        wire [4:0] IMM_4_0         = INSTRUCTION[11:7];
        wire [6:0] IMM_11_5        = INSTRUCTION[31:25];
        wire IMM_11_B              = INSTRUCTION[7];
        wire [3:0] IMM_4_1         = INSTRUCTION[11:8];
        wire [5:0] IMM_10_5        = INSTRUCTION[30:25];
        wire IMM_12                = INSTRUCTION[31];
        wire [7:0] IMM_19_12       = INSTRUCTION[19:12];
        wire IMM_11_J              = INSTRUCTION[20];
        wire [9:0] IMM_10_1        = INSTRUCTION[30:21];
        wire IMM_20                = INSTRUCTION[31];

        // Direkt değerleri komut tipine göre birleştirelim
        wire signed [63:0] Imm_I =
               { {64{IMM_11_0[11]}}, IMM_11_0 };
        wire signed [63:0] Imm_U =
               { {64{IMM_31_12[19]}}, IMM_31_12, 12'h000 };
        wire signed [63:0] Imm_B =
               { {64{IMM_12}}, IMM_11_B, IMM_10_5, IMM_4_1, 1'b0 };
        wire signed [63:0] Imm_S =
               { {64{IMM_11_5[6]}}, IMM_11_5, IMM_4_0 };
        wire signed [63:0] Imm_UJ =
               { {64{IMM_20}}, IMM_19_12, IMM_11_J, IMM_10_1, 1'b0 };

        always @(*) begin
               case (SELECTION)
                      1: VALUE = Imm_I;
                      2: VALUE = Imm_U;
                      3: VALUE = Imm_S;
                      4: VALUE = Imm_B;
                      5: VALUE = Imm_UJ;
                      default : VALUE = 0;
               endcase
        end
endmodule
```

Tasarladığımız ImmediateExtractor modülünün doğru çalışıp çalışmadığını test edebilmek için tb_ImmediateExtractor testbench modülü yazılmıştır.

tb_ImmediateExtractor.v dosya içeriği:

```
`include "ImmediateExtractor.v"
`timescale 1ns / 100ps

module tb_ImmediateExtractor;
	reg [31:0] INSTRUCTION;
	reg [2:0] SELECTION;
	reg signed [63:0] VALUE;

	ImmediateExtractor ie(INSTRUCTION, SELECTION, VALUE);

	initial begin
		INSTRUCTION = 32'h00A00613; // addi x12 x0 10 | Beklenen: 10
		SELECTION = 1; // I Type
		#20;

		INSTRUCTION = 32'h00001337; // lui x6 1 | Beklenen: 4096
		SELECTION = 2; // U Type
		#20;

		INSTRUCTION = 32'h00B323A3; // sw x11 7(x6) | Beklenen: 7
		SELECTION = 3; // S Type
		#20;

		INSTRUCTION = 32'hFEC5CAE3; // blt x11 x12 -12 | Beklenen: -12
		SELECTION = 4; // B Type
		#20;

		INSTRUCTION = 32'h4000006F; // jal x0 1024 | Beklenen: 1024
		SELECTION = 5; // UJ Type
		#20;
	end

	initial begin
		$dumpfile("immediateextractor.vcd");
		$dumpvars(0, tb_ImmediateExtractor);
	end
endmodule
```

tb_ImmediateExtractor.v dosyasının iverilog derleyicisiyle derlenmesiyle elde edilen immediateextractor.vcd dosyasının GTKWave ile dalga analizi şekildeki gibidir:

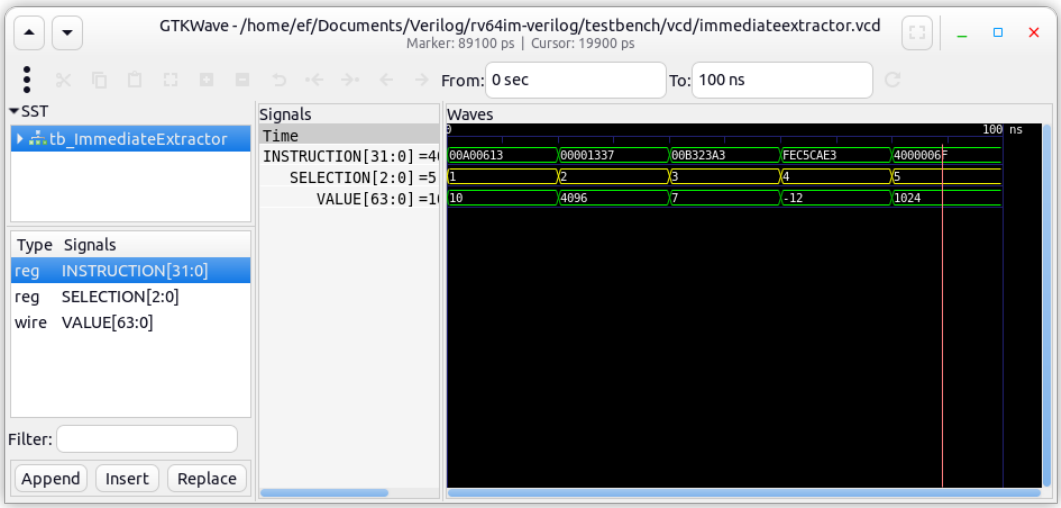

Figure 2.7: tb_ImmediateExtractor.v modülünün dalga analizini yapabileceğimiz immediateextractor.vcd dosyasının GTKWave ile görüntülenmesi.

Immediate Extractor modülümüzün de test için verilen girişlere beklenen çıkışları vererek doğru çalıştığı gözlemlenmiştir.

1.14 Aritmetik Mantık Birimi (Arithmetic Logic Unit)

Her işlemcinin olmazsa olmaz modüllerinden ALU elbette bizim tasarladığımız çekirdekte de mevcuttur. Aritmetik Mantık Birimi X ve Y şeklinde 64 bitlik iki sayı girişi alan, daha sonra bu iki X ve Y sayısını verilen OP(Operasyon tipi) girdisine göre seçilen işleme tabî tutan, daha sonra da 64 bitlik hesaplanmış değeri çıkış olarak veren bir modüldür.

Bu işlemci tasarımında RV64IM çekirdek tasarımındaki komutların kullanacağı hesaplamaları karşılayacak şekilde 14 adet operasyon tipi belirlenmiştir.

ALU.v dosya içeriği:

```
module ALU(
        input [63:0] X,
        input [63:0] Y,
        input [3:0] OP,

        output [63:0] OUTPUT,
        output isEqual
```

```
);
	reg [127:0] RESULT;

	wire signed [63:0] X_signed = X;
	wire signed [63:0] Y_signed = Y;

	assign isEqual = X == Y;

	always @(*) begin
		case (OP)
			0: RESULT <= X + Y; // add
			1: RESULT <= X - Y; // sub
			2: RESULT <= X & Y; // and
			3: RESULT <= X | Y; // or
			4: RESULT <= X ^ Y; // xor
			5: RESULT <= X << Y; // shift left logical
			6: RESULT <= X >> Y; // shift right logical
			7: RESULT <= X_signed >>> Y; // shift right arithmetic
			8: RESULT <= X * Y; // mul
			9: RESULT <= X * Y; // mulh
			10: RESULT <= X / Y; // div
			11: RESULT <= X % Y; // rem
			// set less than(slt)
			12: RESULT <= (X_signed < Y_signed ? 1 : 0);
			13: RESULT <= (X < Y ? 1 : 0); // set less than (sltu)
		endcase
	end

	assign OUTPUT = OP == 9 ? RESULT[127:64] : RESULT[63:0];

endmodule
```

RESULT yazmaçının 128 bit olma sebebi 64 bitlik iki sayıyla yapılan çarpma işleminin sonucunun 128 bitlik bir sayı olabilmesi ve mulh komutunun bu sonucun 127-64 bitlerini, mul komutunun ise bu sonucun 63-0 bitlerini almasıdır. Diğer tüm komutlarda 65. bite taşma ihtimali olmadığı için sadece mulh komutu geldiğinde 127-64 bit arası değerler çıkışa aktarılmıştır.

Tasarladığımız ALU modülünün doğru çalışıp çalışmadığını test edebilmek için tb_ALU testbench modülü yazılmıştır.

tb_ALU.v dosya içeriği:

```
`include "ALU.v"
`timescale 1ns / 100ps

module tb_ALU;
	reg [63:0]a;
	reg [63:0]b;
	reg [3:0]op;
	wire [63:0]result;
	wire isEqual;

	ALU alu(a, b, op, result, isEqual);

	initial begin
		// add
		a = 5;		b = 5;		op = 0; #20;
		// sub
		a = 66;		b = 11;		op = 1; #20;
		// and
		a = 3'b101;	b = 3'b110;	op = 2; #20;
		// or
		a = 3'b101;	b = 3'b110;	op = 3; #20;
		// xor
		a = 3'b110;	b = 3'b010;	op = 4; #20;
		// sll (Shift Left Logical)
		a = 1;		b = 3;		op = 5; #20;
		// srl (Shift Right Logical)
		a = 8;		b = 2;		op = 6; #20;
		// sra (Shift Right Arithmetic)
		a = -8;		b = 2;		op = 7; #20;
		// mul
		a = 6;		b = 5;		op = 8; #20;
		// mulh | beklenen = ...0010 = 2
		a = 64'h8000000000000000;
		b = 4; op = 9; #20;
		// div
		a = 66;		b = 11;		op = 10; #20;
		// rem
		a = 62;		b = 3;		op = 11; #20;
		// slt (Set Less Than)
		a = -1;		b = 9;		op = 12; #20;
		// sltu (Set Less Than Unsigned)
		a = -1;		b = 9;		op = 13; #20;
```

```
        end

        initial begin
                $dumpfile("alu.vcd");
                $dumpvars(0, tb_ALU);
        end
endmodule
```

tb_ALU.v dosyasının iverilog derleyicisiyle derlenmesiyle elde edilen alu.vcd dosyasının GTKWave ile dalga analizi şekildeki gibidir:

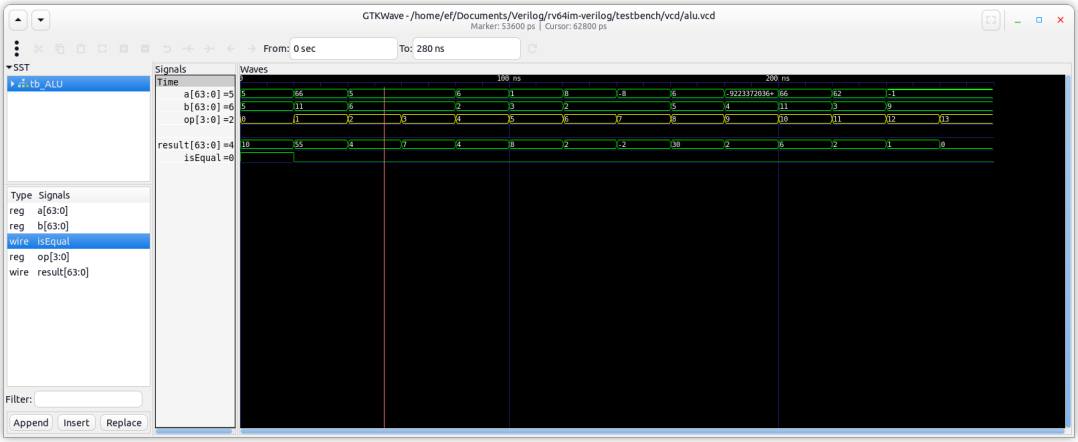

Figure 2.8: tb_ALU.v modülünün dalga analizini yapabileceğimiz alu.vcd dosyasının GTKWave ile görüntülenmesi.

ALU modülümüzün de test için verilen girişlere beklenen çıkışları vererek doğru çalıştığı gözlemlenmiştir.

3 İŞLEMCİ TASARIMINA GİRİŞ

İşlemcinin iç tasarımına geçmeden önce karar vermemiz gereken tasarım konseptleri vardır. Bu tasarım konseptleri işlemci tasarlanmadan önce seçilip yol haritası oluşturulmasından sonra işlemci tasarlanmaya başlanır.

İşlemcimizde kullandığımız iki adet tasarım konsepti vardır. Boru Hattı ve Harvard mimarisi. Boru Hattı yöntemi işlemcinin iç mimarisini büyük oranda etkileyen ve işlemciyi tek çevrimli bir işlemciden çok çevrimli bir işlemci haline getiren bir yöntemdir. Harvard mimarisi ise işlemcinin bilgisayar organizasyonu ile alakalı olup işlemcinin diğer dış modüllerle (ROM ve RAM gibi) çalışma biçimini tanımlar.

1.15 Boru Hattı (Pipeline)

Boru hattı yöntemi gelişmiş bir işlemcinin olmazsa olmazı özelliklerden birisidir. 2004'te çıkarılan İntel Pentium 4 Prescott işlemcisi 31 aşamalı boru hattına sahiptir[5]. Boru Hattının çalışma prensibini anlatmak için ise genelde kıyafetlerini yıkamak isteyen birkaç kişinin bu süreci gerçekleştirme şekli örnek verilir.

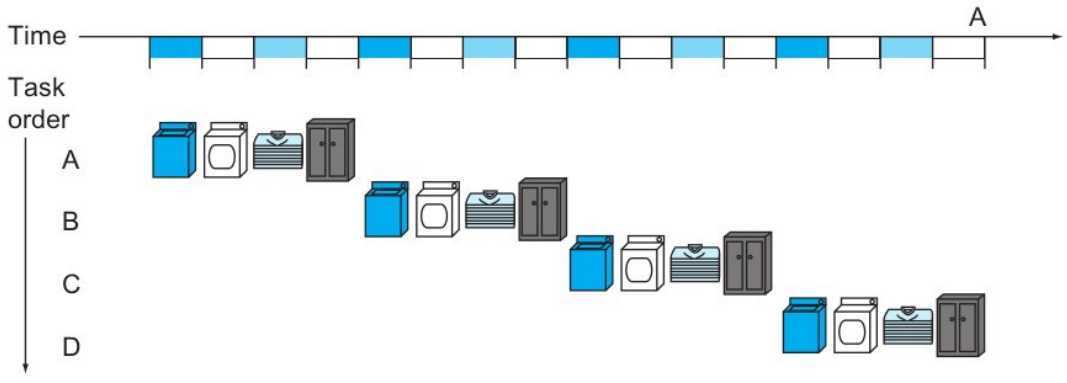

Figure 3.1: Boru hattı prensibi uygulanmamış çamaşır yıkama işinin harcadığı zamanın görselleştirlmiş hali. [6]

Şekil 3.1'de dört adet kişinin kıyafetlerini yıkaması için izlediği adımları incelediğimizde A kişisi kirli kıyafetlerini yıkayıp, kurutup, katlayıp, dolaba yerleştirdikten sonra ancak B kişisi kendi çamaşırını yıkama işlemine başlayabiliyor.

Bu örnekte fark edileceği üzere B kişisinin kıyafetlerini yıkamak için A kişisinin kıyafetlerini katlayıp dolabına asmasını beklemesi gerekmiyor. Hatta bu durum büyük bir vakit kaybına yol açıyor çünkü B kişisi, A kişisi kıyafetlerini çamaşır makinesinden çıkarıp kurutma aşamasına geçtiğinde kendi kıyafetlerini çamaşır makinesine yerleştirebilecekken gereksiz yere beklemiş oluyor.

Şekil 3.1'deki örnek aslında Tek Çevrimli tasarlanmış bir işlemciyi temsil ediyor. Tek Çevrimli tasarlanan bir işlemcide bütün işlemler tek bir çevrimde gerçekleşir. Dolayısıyla işlemcinin o anki çevrimi bitirip diğer çevrime geçmesi için bütün aşamalarının geçilmiş ve tamamlanmış olması gerekiyor. Bu da saat darbe sıklığının daha düşük tutulması gerekliliğini getiriyor.

Örneğin getirdiği buyruk yükle komutu olan ve bellekten veri okuma işlemini gerçekleştiren işlemci, istediği veri gelene kadar diğer bütün donanımları boşta bekletiyor. Böylece büyük bir zaman kaybı olmuş oluyor.

İşte boru hattı yöntemi ve çok çevrimli işlemci tasarımları bu soruna çözüm olarak sunulmuş yöntemlerdir. Boru hattı yöntemiyle işlemcinin içinde yapılan işler parçalara ayrılıp, hepsinin ayrı ayrı saat darbelerinde gerçekleşmesini sağlayarak, beklemeden kaynaklı zaman israfını engellemiş ve birim zamanda daha çok iş gerçekleştirilmiş oluyor.

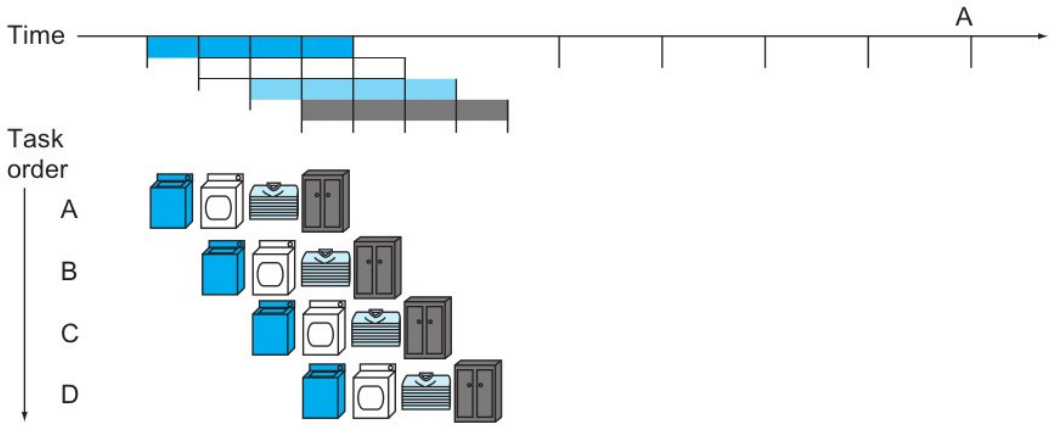

Figure 3.2: Boru hattı yöntemi uygulanan işlemler çok daha kısa bir sürede zaman israfı olmadan bitirilmiş oluyor. [6]

Şekil 3.2'de görüldüğü gibi artık B kişisinin kıyafetlerini çamaşır makinesine atmak için A kişisinin bütün işlerini bitirmesini beklemiyor. Böylece birim zamanda yapılan iş miktarı artmış oluyor.

1.16 Boru Hattı Yönteminin İşlemcide Uygulanışı

Boru Hattı yöntemi uygulanmış işlemciler yaptıkları işlemleri parçalara ayırarak bir çevrimde birden fazla işin aynı anda yapılmasını sağlar. Örneğin bir saat darbesinde hem sıradaki komut getirilip hem de önceden getirilmiş bir komut ALU'da işleme tabi tutulabilir. Böylece bir çevrimin tamamlanması için beklenilmesi gereken asgari süre de azaltılmış olmaktadır. Bunun sonucunda da daha yüksek çevrim hızlarına çıkılabilmektedir.

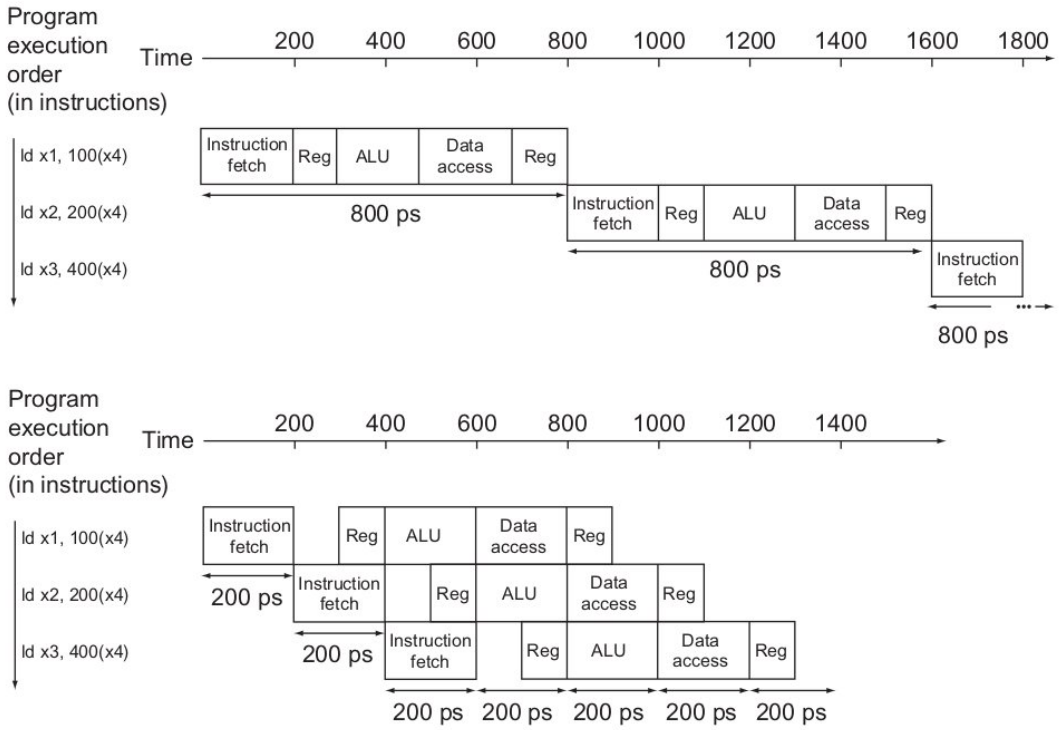

Figure 3.3: Tek Çevrimli ve Çok Çevrimli Boru Hattı yöntemi uygulanmış işlemcilerin buyrukları işleme süreleri arasındaki fark. [6]

Şekil 3.3'te görüldüğü üzere çok çevrimli işlemciler işlemcide gerçekleştirilen aşamaları parçalara ayıp hepsini ayrı ayrı saat darbelerinde tetiklediği için gayet yüksek miktarda bir hızlanma elde edilmiştir.

1.17 İşlemcideki Boru Hattı Aşamaları

Tasarladığımız işlemci 5 aşamalı boru hattına sahiptir. Bu aşamalar ise:

1. Getir (Fetch) : İşlemci ROM bellekten bir sonraki buyruğu getirip bir yazmaça kaydeder. Bu buyruk boru hattının sonuna kadar her aşamaya tek tek aktarılır.
2. Çöz (Decode) : Getirilen buyruk çözümlenir ve parametrelerine ayırılır.
3. Çalıştır (Execute) : Her şeyi bilinen buyruğun çalıştırılması yani sonucun hesaplanmasının gerçekleştiği aşamadır.
4. Bellek (Memory) : Bellekten veri okunan veya belleğe veri yazılan aşamadır.
5. Geri Yaz (Writeback) : Çalıştır veya Bellek aşamalarından elde edilen sonuçlar eğer bir yazmaça kaydedilecekse bu aşamada kaydedilir.

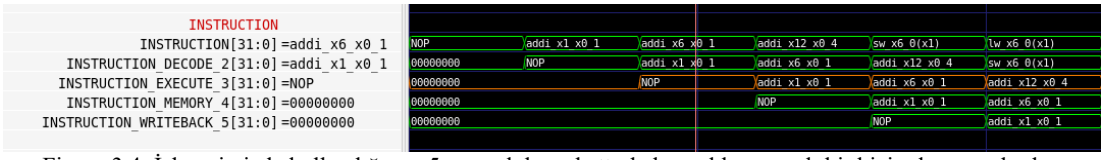

Figure 3.4: İşlemcimizde kullandığımız 5 aşamalı boru hattıyla buyrukların sıralı bir biçimde aşamalardan geçirilmesinin gözlemlenmesi.

1.18 Boru Hattı Yöntemi Sorunları (Pipeline Hazards)

Elbette ki elde ettiğimiz bu hızlanma yanında çözülmesi gereken pek çok sorunu da beraberinde getirmektedir. İşlemcimizde kurduğumuz boru hattında oluşan üç adet sorun şu şekildedir:

1. Veri Bağımlılığı (Data Hazard) : Bu sorun bir komutun girdi olarak kullanacağı yazmaçın bir veya iki önceki komutta değiştirilmiş olmasıdır. İşlenen komutun sonucunun ilgili yazmaça yazılması için üç saat darbesi geçip Geri Yaz(Writeback) aşamasının bitmiş olması gerekir. Veri bağımlılığı sorununda hemen bir veya iki sonraki gelen komut henüz geri yazılmamış bu veriye ihtiyaç duyar.

```
addi x8, x0, 55
add x8, x0, x8 # x8 -> Bir önceki komutta hesaplanacak x8 yazmaçına bağımlı.
```

```
addi x6, x0, 1
addi x12, x0, 4
sw x6, 0(x1) # x6 -> İki önceki komutta hesaplanacak x6 yazmaçına bağımlı.
```

Bu sorunu gidermek için iki yöntem uygulanabilir. Boru hattına üç çevrim boyunca yeni komut almayıp Bellek ve Geri Yaz aşamalarını ilerletip diğer aşamaları bekletmek, veya Çalıştır işleminden elde edilen ve bir sonraki aşamaya aktarılacak olan sonucun aynı zamanda tekrar Çalıştır aşamasına tekrar geri verilmesi.

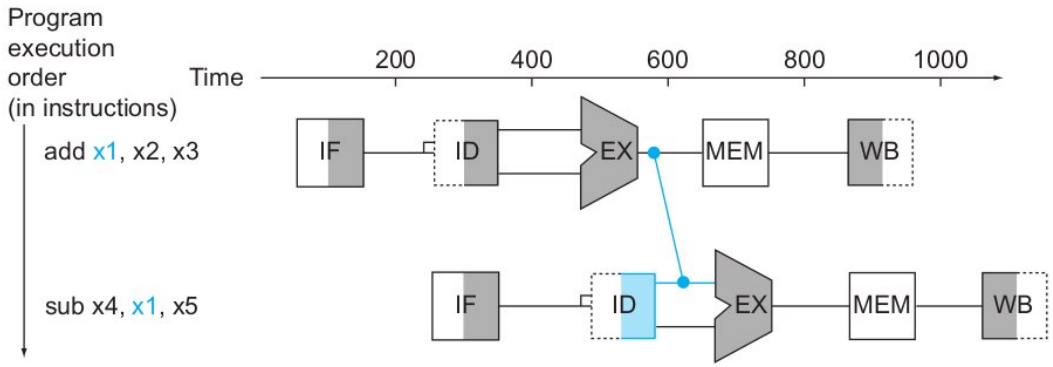

Figure 3.5: Veri bağımlılığı sorununun Çalıştır aşamasının çıkışını tekrar Çalıştır aşamasına geri vererek çözümünün görselleştirilmiş hali. [6]

İşlemcimizin tasarımında Veri Bağımlılığı sorununun giderilmesi için ihtiyaç olan verinin ilgili aşamalardan direkt olarak alınıp kullanılması yöntemi uygulanmıştır.

2. Kontrol Sorunu (Control Hazard) : Bu sorun dallanma(branch) buyruklarında dallanmaya karar verilme aşamasının Çalıştır yani üçüncü aşama olmasından kaynaklanan bir sorundur.

Dallanma komutunda dallanmanın yapılıp yapılmayacağına Çalıştır aşamasında karar verilir ve Program Sayacının(Program Counter) yeni değeri elde edilmiş olur. Fakat bu aşamaya gelene kadar boru hattına iki adet çalışmaması gereken buyruk dahil olmuş olabilir.

```
loop:
        lw x6, 0(x1)
        addi x6, x6, 1
        sw x6, 0(x1)
```

```
        blt x6, x12, loop
exit_loop:
        addi x8, x0, 55
        add x8, x0, x8
```

Örneğin yukarıdaki RISC-V programında "x6 x12'den küçük müdür?" kararı verilene kadar exit_loop kısmındaki iki komut da boru hattına eklenmiş olacaktır.

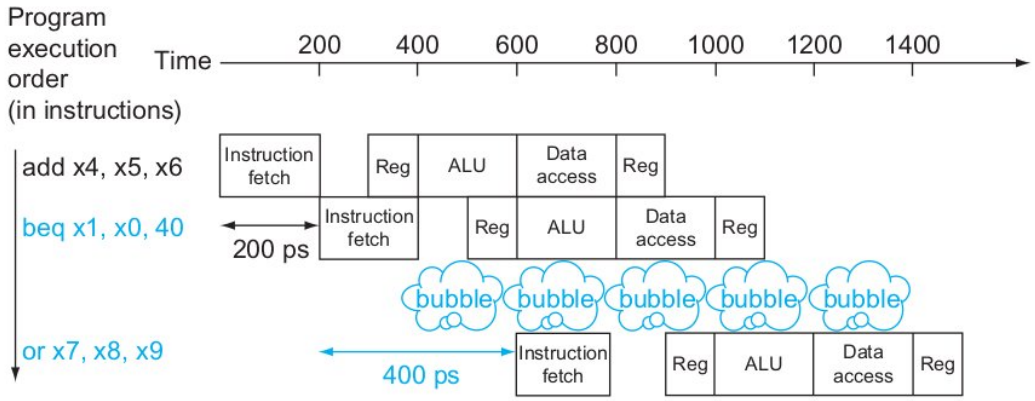

Figure 3.6: Kontrol Sorununun çözülmesi için iki çevrim boyunca boru hattının bekletilmesi yöntemi. [6]

Bu sorunu gidermek için iki yöntem uygulanabilir. Boru Hattına bir dallanma komutu girdiğinde dallanma kararı belli olana kadar, yani iki çevrim boyunca boru hattını bekletmek. Diğer bir yöntem ise Dallanma Tahmini(Branch Prediction) yaparak dallanma kararını önceden verilmiş olan kararlara bakarak verip, örneğin iki aşamalı bir sonlu durum makinesi kullanarak, dallanma komutu daha Çalıştır aşamasına gelmeden Program Sayacını ilgili yere çekmek.

Bu yöntem beraberinde "Yanlış tahmin edildiğinde ne olacak?" sorununu da getirir. Bu sorunun çözümü için ise yanlış tahmin edildiği anlaşılan buyruktan sonra işlenen komutları geri alma işlemi uygulanmalıdır. Yanlış tahminin maaliyeti yüksektir.

İşlemcimizin tasarımında Kontrol Sorununun giderilmesi için dallanma tahmini yapılmamış, boru hattını iki çevrim boyunca bekletme(stall) yöntemi uygulanmıştır.

3. Bellek Yükleme Gecikmesi (Load Stall) : Bu sorun yükleme(load) komutundan sonra gelen komutun bellekten yüklenecek olan veriye bağımlı olma sorunudur.

```
lw x6, 0(x1)
addi x6, x6, 1 # x6'nın bellekten yüklenmesi beklenmelidir.
```

Bu sorun da aynı Veri Bağımlılığı sorunu gibi bekleyerek ya da ara kablo çekilerek çözülebilir. Fakat Veri Bağımlılığına ek olarak Bellek aşamasından Çalıştır aşamasına direkt olarak kablo çekilse bile yükleme(load) komutunun Bellek aşamasına gelmesi için 1 çevrim gecikme kaçınılmazdır.

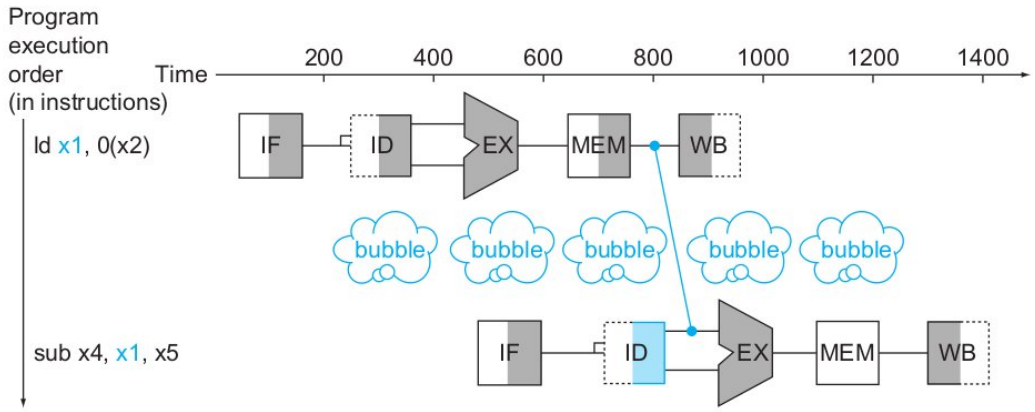

Figure 3.7: Bellek Yükleme Gecikmesi sorununun çözümü için kablo çekilse bile 1 çevrim gecikmenin kaçınılmaz olduğunu gösteren görsel. [6]

İşlemcimizin tasarımında Bellek Yükleme Gecikmesi sorununun çözümü için hem Bellek aşamasından kablo çekilmiş, hem de boru hattı bir çevrim bekletilmiştir.

1.19 İşlemcinin Çalışacağı Bilgisayar Düzeni

Von Neumann Mimarisine göre bir bilgisayar temel olarak iki birimden oluşur. İşlemci ve RAM bellek. İşlemci RAM bellekteki komutları sırasıyla okuyup çalıştırır ve gerektiğinde tekrar RAM belleğe veri yazar ve okur. Eğer program komutlarının okunacağı bellek, verilerin depolandığı bellekten ayrı tutulacaksa bu mimariye de Harvard Mimarisi adı verilir.

İşlemciyi test edeceğimiz bilgisayar organizasyonu Harvard mimarisine göre tasarlanmıştır, yani program komutlarının okunacağı ROM bellek ve verilerin saklanıp okunacağı RAM bellek şeklinde iki ayrı belleğimiz olacaktır.

İşlemcimizin çalıştığı bilgisayar düzeni Şekil 3.8'te görselleştirildiği gibi bir adet RV64IM CPU, bir adet 10-bit adres uzayına sahip RAM ve bir adet 10-bit adres uzayına sahip ROM bellekten oluşmaktadır.

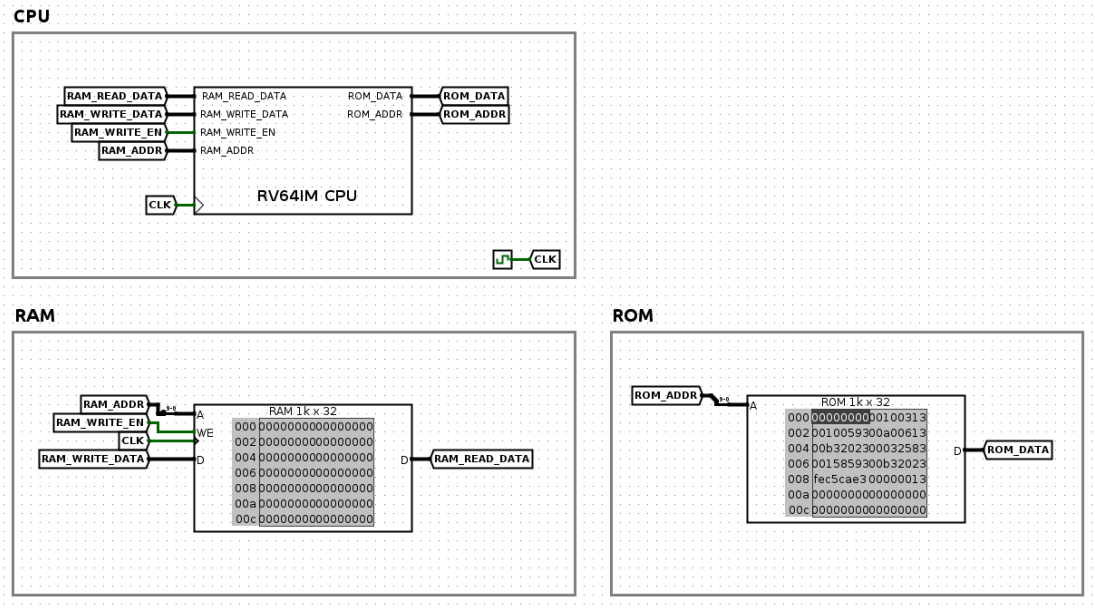

Figure 3.8: RV64IM işlemciye sahip bilgisayarımızın örnek olarak görselleştirilmiş organizasyonu.

Görsel olarak tasarımını gösterdiğimiz bu bilgisayar mimarisinin Verilog dilinde karşılığı tb_Main.v dosyasında aşağıdaki şekilde yazılmıştır:

```
`include "CPU.v"
`include "RAM.v"
`include "ROM.v"

`timescale 1ns / 100ps
module tb_Main;
	wire [9:0] RAM_ADDR;
	wire [63:0] RAM_READ_DATA;
	wire [63:0] RAM_WRITE_DATA;
	wire RAM_WRITE_ENABLE;
	wire [9:0] INSTRUCTION_ADDR;
	wire [31:0] INSTRUCTION;
```

```
	reg CLK = 1;

	CPU cpu(
		.RAM_READ_DATA(RAM_READ_DATA),
		.INSTRUCTION(INSTRUCTION),
		.CLK(CLK),

		.RAM_ADDR(RAM_ADDR),
		.RAM_WRITE_DATA(RAM_WRITE_DATA),
		.RAM_WRITE_ENABLE(RAM_WRITE_ENABLE),
		.RAM_READ_ENABLE(RAM_READ_ENABLE),
		.INSTRUCTION_ADDR(INSTRUCTION_ADDR)
	);
	ROM rom(
		.ADDRESS(INSTRUCTION_ADDR),

		.DATA(INSTRUCTION)
	);
	RAM ram(
		.ADDRESS(RAM_ADDR),
		.DATA_IN(RAM_WRITE_DATA),
		.WRITE_ENABLE(RAM_WRITE_ENABLE),
		.READ_ENABLE(RAM_READ_ENABLE),
		.CLK(CLK),

		.DATA_OUT(RAM_READ_DATA)
	);


	initial begin
		// Test amaçlı 40 Adet Saat Darbesi verilmiştir.
		for(int i=0; i<40; i=i+1) begin
			CLK=1; #20; CLK=0; #20;
		end
	end

	initial begin
		$dumpfile("main.vcd");
		$dumpvars(0, tb_Main);
	end
endmodule
```

tb_Main modülü, İşlemci, RAM ve ROM bellekten oluşan bir bilgisayar organizasyonudur. Bu modülün kendine has Main.v şeklinde bir dosyası yoktur

çünkü sadece başka modülleri test etmek için testbench modülü olarak oluşturulmuştur.

1.20 İşlemci Çekirdeğine Giriş

CPU.v modülü içerisinde tasarladığımız işlemcinin ilk kısımlarında, ilerleyen adımlarda ihtiyacımız olacak sabitleri ve kablolar tanımlanmıştır.

CPU.v dosya içeriği:

```
`include "ALU.v"
`include "RegFile.v"
`include "ImmediateExtractor.v"
`include "Encoders.v"

module CPU (
        input [63:0] RAM_READ_DATA,
        input [31:0] INSTRUCTION,
        input CLK,

        output [9:0] RAM_ADDR,
        output reg [63:0] RAM_WRITE_DATA,
        output RAM_WRITE_ENABLE,
        output [9:0] INSTRUCTION_ADDR
);
        // SABİT SAYI TANIMLAMALARI
        // -- OPCODE Tanımları
        integer OP_R_TYPE          = 7'h33;
        integer OP_R_TYPE_64       = 7'h3B;
        integer OP_I_TYPE_LOAD     = 7'h03;
        integer OP_I_TYPE_OTHER    = 7'h13;
        integer OP_I_TYPE_64       = 7'h1B;
        integer OP_I_TYPE_JUMP     = 7'h6F;
        integer OP_S_TYPE          = 7'h23;
        integer OP_B_TYPE          = 7'h63;
        integer OP_U_TYPE_LOAD     = 7'h37;
        integer OP_U_TYPE_JUMP     = 7'h67;
        integer OP_U_TYPE_AUIPC    = 7'h17;
        // -- Boru Hattında ilerletilen komutların tipleri
        integer TYPE_REGISTER      = 0;
        integer TYPE_LOAD          = 1;
```

```
integer TYPE_STORE         = 2;
integer TYPE_IMMEDIATE     = 3;
integer TYPE_UPPERIMMEDIATE = 4;
integer TYPE_BRANCH        = 5;
// -- Boru Hattı Aşamaları
integer DECODE      = 0;
integer EXECUTE     = 1;
integer MEMORY      = 2;
integer WRITEBACK   = 3;
```

İlerideki aşamalarda kullanılan ve ne olduğu tam olarak belirli olmayan numaraları kullanmak yerine o numaralara isimler vererek kullanılarak kodun daha temiz kalabilmesi amaçlanmıştır.

Her çekirdek eklentisinin getirdiği komutların OPCODE değerlerinin listesi bu adreslerde herkese açık erişilebilir durumdadır: https://github.com/riscv/riscv-opcodes, https://riscv.org/technical/specifications/.

1.21 Sık Kullanılacak Parametrelerin Tanımlanması

İşlemci içerisinde temel olarak ihtiyacımız olacak parametreler gelen komutlardan ayrıştırılıp kendi müstakil kablolarına atanmıştır. Böylece birden fazla yerde aynı özelliğe erişilmek istenildiğinde her seferinde tekrar ayrıştırma işlemi için uğraşılmamış, ilk seferde ayrıştırılmış hazır kablolar kullanılmıştır.

CPU.v dosya içeriği devamı:

```
// Gelen komuttan parametrelerin elde edilmesi
wire [6:0] OPCODE   = INSTRUCTION_EXECUTE_3[6:0];
wire [4:0] RD       = INSTRUCTION_WRITEBACK_5[11:7];
wire [2:0] FUNCT3   = INSTRUCTION_EXECUTE_3[14:12];
wire [4:0] R1       = INSTRUCTION_EXECUTE_3[19:15];
wire [4:0] R2       = INSTRUCTION_EXECUTE_3[24:20];
wire [6:0] FUNCT7   = INSTRUCTION_EXECUTE_3[31:25];

// Elde edilen parametrelerden komut tipinin elde edilmesi
wire R_TYPE         = OPCODE == OP_R_TYPE;
wire R_TYPE_64      = OPCODE == OP_R_TYPE_64;
```

```
wire I_TYPE_LOAD    = OPCODE == OP_I_TYPE_LOAD;
wire I_TYPE_OTHER   = OPCODE == OP_I_TYPE_OTHER;
wire I_TYPE_64      = OPCODE == OP_I_TYPE_64;
wire I_TYPE_JUMP    = OPCODE == OP_I_TYPE_JUMP;
wire I_TYPE  = I_TYPE_JUMP || I_TYPE_LOAD || I_TYPE_OTHER || I_TYPE_64;
wire S_TYPE         = OPCODE == OP_S_TYPE;
wire B_TYPE         = OPCODE == OP_B_TYPE;
wire U_TYPE_LOAD    = OPCODE == OP_U_TYPE_LOAD;
wire U_TYPE_JUMP    = OPCODE == OP_U_TYPE_JUMP;
wire U_TYPE_AUIPC   = OPCODE == OP_U_TYPE_AUIPC;
wire U_TYPE  = U_TYPE_JUMP || U_TYPE_LOAD || U_TYPE_AUIPC;
```

Bu kısımda yine ileride kontrol amaçlı veya yapılacak işlemlere karar verme amaçlı kullanılacak olan komut tiplerinin elde edilmesi sağlanmıştır. Örneğin R_TYPE kablosu, eğer OPCODE kablosunun değeri OP_R_TYPE sabit sayısına eşitse “1” değilse “0” olacaktır.

Çalıştır safhasında kullanılacak parametreler INSTRUCTION_EXECUTE_3 yazmaçında tutulan komuttan elde edilirken Geri Yaz safhasında kullanılacak olan RD parametresi INSTRUCTION_WRITEBACK_5 yazmaçında tutulan komuttan elde edilmiştir.

1.22 Buyrukların Tespit Edilmesi

İşlenecek olan buyruğun tam olarak hangi buyruk olduğuna karar verebilmek için RISC-V Opcode listesindeki buyruklara ait ön tanımlı sabit sayılar kullanılmıştır. Bir buyruğun kimliğinin tespiti için OPCODE ve FUNCT değerleri kullanılmaktadır.

Örneğin R-Tip bir add buyruğu gelip gelmediğini tespit edebilmek için daha önce tanımladığımız ve buyruk R-Tip bir buyruk ise “1” olan R_TYPE kablosunu, add buyruğunun ön tanımlı sabit sayıları olan FUNCT3 için 0 ve FUNCT7 için de yine 0 değerlerine eşit olup olmadığı kıyaslanmıştır.

CPU.v dosya içeriği devamı:

```
// --- Yazmaç-Yazmaç(Register-Register) Komutları (R-Type)
// ---- RV32I:
wire R_add   = R_TYPE && FUNCT3 == 3'h0 && FUNCT7 == 7'h00;
wire R_sub   = R_TYPE && FUNCT3 == 3'h0 && FUNCT7 == 7'h20;
wire R_sll   = R_TYPE && FUNCT3 == 3'h1 && FUNCT7 == 7'h00;
wire R_slt   = R_TYPE && FUNCT3 == 3'h2 && FUNCT7 == 7'h00;
wire R_sltu  = R_TYPE && FUNCT3 == 3'h3 && FUNCT7 == 7'h00;
wire R_xor   = R_TYPE && FUNCT3 == 3'h4 && FUNCT7 == 7'h00;
wire R_srl   = R_TYPE && FUNCT3 == 3'h5 && FUNCT7 == 7'h00;
wire R_sra   = R_TYPE && FUNCT3 == 3'h5 && FUNCT7 == 7'h20;
wire R_or    = R_TYPE && FUNCT3 == 3'h6 && FUNCT7 == 7'h00;
wire R_and   = R_TYPE && FUNCT3 == 3'h7 && FUNCT7 == 7'h00;
// ---- RV32M:
wire R_mul   = R_TYPE && FUNCT3 == 3'h0 && FUNCT7 == 7'h01;
wire R_mulh  = R_TYPE && FUNCT3 == 3'h1 && FUNCT7 == 7'h01;
wire R_rem   = R_TYPE && FUNCT3 == 3'h6 && FUNCT7 == 7'h01;
wire R_div   = R_TYPE && FUNCT3 == 3'h4 && FUNCT7 == 7'h01;
// ---- RV64I:
wire R_addw = R_TYPE_64 && FUNCT3 == 3'h0 && FUNCT7 == 7'h00;
wire R_subw = R_TYPE_64 && FUNCT3 == 3'h0 && FUNCT7 == 7'h20;
wire R_sllw = R_TYPE_64 && FUNCT3 == 3'h1 && FUNCT7 == 7'h00;
wire R_srlw = R_TYPE_64 && FUNCT3 == 3'h5 && FUNCT7 == 7'h00;
wire R_sraw = R_TYPE_64 && FUNCT3 == 3'h5 && FUNCT7 == 7'h20;
// ---- RV64M:
wire R_mulw = R_TYPE_64 && FUNCT3 == 3'h0 && FUNCT7 == 7'h01;
wire R_divw = R_TYPE_64 && FUNCT3 == 3'h4 && FUNCT7 == 7'h01;
wire R_remw = R_TYPE_64 && FUNCT3 == 3'h6 && FUNCT7 == 7'h01;

// -- Direkt(Immediate) Komutlar (I-Type)
// ---- RV32I:
wire I_addi  = I_TYPE_OTHER && FUNCT3 == 3'h0;
wire I_slli  = I_TYPE_OTHER && FUNCT3 == 3'h1 && FUNCT7 == 7'h00;
wire I_slti  = I_TYPE_OTHER && FUNCT3 == 3'h2;
wire I_sltiu = I_TYPE_OTHER && FUNCT3 == 3'h3;
wire I_xori  = I_TYPE_OTHER && FUNCT3 == 3'h4;
wire I_srli  = I_TYPE_OTHER && FUNCT3 == 3'h5 && FUNCT7 == 7'h00;
wire I_srai  = I_TYPE_OTHER && FUNCT3 == 3'h5 && FUNCT7 == 7'h10;
wire I_ori   = I_TYPE_OTHER && FUNCT3 == 3'h6;
wire I_andi  = I_TYPE_OTHER && FUNCT3 == 3'h7;
// ---- RV64I:
wire I_addiw = I_TYPE_64 && FUNCT3 == 3'h0;
wire I_slliw = I_TYPE_64 && FUNCT3 == 3'h1 && FUNCT7 == 7'h00;
wire I_srliw = I_TYPE_64 && FUNCT3 == 3'h5 && FUNCT7 == 7'h00;
wire I_sraiw = I_TYPE_64 && FUNCT3 == 3'h5 && FUNCT7 == 7'h20;
// ---- Load:
```

```
wire I_lb = INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_LOAD &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h0;
wire I_lh = INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_LOAD &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h1;
wire I_lw = INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_LOAD &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h2;
wire I_ld = INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_LOAD &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h3;
// ---- Jump:
wire I_jalr = I_TYPE_JUMP;

// -- Üst Direkt(Upper Immedidate) Komutlar (U-Type):
wire U_lui   = U_TYPE_LOAD;
wire U_auipc = U_TYPE_AUIPC;
wire U_jal   = U_TYPE_JUMP;

// -- Kayıt(Store) Komutları (S-Type):
wire S_sb = INSTRUCTION_MEMORY_4[6:0] == OP_S_TYPE &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h0;
wire S_sh = INSTRUCTION_MEMORY_4[6:0] == OP_S_TYPE &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h1;
wire S_sw = INSTRUCTION_MEMORY_4[6:0] == OP_S_TYPE &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h2;
wire S_sd = INSTRUCTION_MEMORY_4[6:0] == OP_S_TYPE &&
       INSTRUCTION_MEMORY_4[14:12] == 3'h3;

// -- Dallanma(Branch) Komutları (B-Type)
wire B_beq   = B_TYPE && FUNCT3 == 0;
wire B_bne   = B_TYPE && FUNCT3 == 1;
wire B_blt   = B_TYPE && FUNCT3 == 4;
wire B_bge   = B_TYPE && FUNCT3 == 5;
wire B_bltu  = B_TYPE && FUNCT3 == 6;
wire B_bgeu  = B_TYPE && FUNCT3 == 7;
```

Yükleme(Load) ve Kaydetme(Store) komutları diğer komutlardan farklı olarak 4. yani Bellek aşamasında kullanılacak olan komutlar olduğu için Çalıştır(Execute) aşamasındaki komuttan elde edilen OPCODE tanımı kullanılması yerine direkt olarak INSTRUCTION_MEMORY_4 aşamasındaki komuttan elde edilmiştir.

Böylece işlemcinin hangi komutu çalıştırdığı hakkında artık bilgi sahibiyiz. Bundan sonra ise bu kabloları kullanarak hangi işlemi gerçekleştireceğimize karar verebiliriz.

CPU.v dosya içeriği devamı:

```
wire signed [63:0] R1_DATA =
                    DATA_DEPENDENCY_HAZARD_R1 ? ALU_OUT_MEMORY_4 :
                    DATA_DEPENDENCY_HAZARD_R1_WRITEBACK ?
                    (REG_WRITEBACK_SELECTION == 3
                          ? RAM_READ_DATA_WRITEBACK_5
                          : REG_WRITE_DATA_WRITEBACK_5)
                    : R1_DATA_REGFILE;
wire signed [63:0] R2_DATA =
                    DATA_DEPENDENCY_HAZARD_R2 ? ALU_OUT_MEMORY_4 :
                    DATA_DEPENDENCY_HAZARD_R2_WRITEBACK ?
                    (REG_WRITEBACK_SELECTION == 3
                          ? RAM_READ_DATA_WRITEBACK_5
                          : REG_WRITE_DATA_WRITEBACK_5)
                    : R2_DATA_REGFILE;

wire [63:0] R1_DATA_UNSIGNED = R1_DATA;
wire [63:0] R2_DATA_UNSIGNED = R2_DATA;
```

Bu kısımda R1 ve R2 yazmaçlarından okunacak verinin ne olacağına karar verilmiştir. Aslında eğer boru hattı uygulaması yapılmasaydı bu kısım basitçe şu şekilde yazılacaktı:

```
wire signed [63:0] R1_DATA = R1_DATA_REGFILE;
wire signed [63:0] R2_DATA = R2_DATA_REGFILE;
wire [63:0] R1_DATA_UNSIGNED = R1_DATA;
wire [63:0] R2_DATA_UNSIGNED = R2_DATA;
```

Görüldüğü gibi R1 ve R2 yazmaçlarının verileri direkt olarak RegFile'ın çıkışından alınacaktı. Fakat Boru Hattı Yöntemi Sorunları (Pipeline Hazards) başlığı altında bahsettiğimiz sorunlar yüzünden işleme tutulacak R1 ve R2 yazmaçlarının verileri farklı aşamalardan da alınabilir şekilde tasarlanmıştır.

CPU.v dosya içeriği devamı:

```
wire PC_ALU_SEL = (B_beq && R1_DATA == R2_DATA)
                    || (B_bne && R1_DATA != R2_DATA)
                    || (B_blt && R1_DATA < R2_DATA)
                    || (B_bge && R1_DATA >= R2_DATA)
```

```
|| (B_bltu && R1_DATA_UNSIGNED < R2_DATA_UNSIGNED)
|| (B_bgeu && R1_DATA_UNSIGNED >= R2_DATA_UNSIGNED)
|| I_jalr
|| U_jal
;
```

PC_ALU_SEL kablomuz Program Sayacı(Program Counter) bir sonraki saat darbesinde yoluna + 4 ekleyerek devam mı edecek yoksa ALU'da hesaplanan adres değerini mi alacak kararının verildiği kablo. Eğer PC, ALU'nun çıkışına eşitlenecekse bu kablo 1, PC + 4 olarak normal artmaya devam edecekse 0 olacak. PC'yi ileride arttırırken bu kabloyu kontrol edeceğiz.

CPU.v dosya içeriği devamı:

```
assign RAM_WRITE_ENABLE = INSTRUCTION_MEMORY_4[6:0] == OP_S_TYPE;
assign RAM_ADDR = ALU_OUT_MEMORY_4;
```

CPU modülümüzün çıkış parametrelerinden RAM_WRITE_ENABLE eğer Bellek aşamasındaki komutun tipi S-Tip yani Kayıt komutu ise RAM'e yazmayı etkinleştiriyoruz. RAM_ADDR çıkışı ise yazacağımız veya okuyacağımız RAM adresini belirtiyor. Bu iki tür adres de ALU'da offset + yazmaç şeklinde hesaplanan değer sonucunda elde ediliyor. Dolayısıyla ALU_OUT_MEMORY_4 yazmaçının değeri direkt olarak RAM_ADDR çıkışına verilebilir.

1.23 Boru Hattı Sorunlarının Tespiti

Boru Hattı sorunlarının tespiti için Boru Hattı Yöntemi Sorunları (Pipeline Hazards) başlığı altında belirtilmiş çözümler kullanılmıştır.

CPU.v dosya içeriği devamı:

```
// Boru Hattında bulunan komutların: R1, R2, RD ve Komut tipi bilgilerinin
tutulacağı yazmaçlar.
reg [4:0] R1_PIPELINE[3:0];
reg [4:0] R2_PIPELINE[3:0];
reg [4:0] RD_PIPELINE[3:0];
reg [2:0] TYPE_PIPELINE[3:0];
```

```
// TYPE_PIPELINE yazmaçları TYPE_IMMEDIATE, TYPE_REGISTER gibi sabit tanımlı
değerleri almaktadır.
```

Bu yazmaçlarda boru hattındaki komutun R1, R2 ve RD olarak hangi yazmaçları kullandığı ve komutun tipi bilgisi tutulmaktadır. Veri Bağımlılığı ve Bellek Yükleme Gecikmesi sorunlarının tespiti için kullanılmaktadır.

CPU.v dosya içeriği devamı:

```
// Eğer R1 bir önceki komutun(Bellek aşaması) RD'sine bağımlıysa
wire DATA_DEPENDENCY_HAZARD_R1 =
       R1_PIPELINE[EXECUTE] != 0
       && TYPE_PIPELINE[EXECUTE] != TYPE_UPPERIMMEDIATE
       && R1_PIPELINE[EXECUTE] == RD_PIPELINE[MEMORY];

// Eğer R2 bir önceki komutun(Bellek aşaması) RD'sine bağımlıysa
wire DATA_DEPENDENCY_HAZARD_R2 =
       R2_PIPELINE[EXECUTE] != 0
       && TYPE_PIPELINE[EXECUTE] != TYPE_UPPERIMMEDIATE
       && TYPE_PIPELINE[EXECUTE] != TYPE_IMMEDIATE
       && R2_PIPELINE[EXECUTE] == RD_PIPELINE[MEMORY];

// Eğer R1 iki önceki komutun(Geri Yaz aşaması) RD'sine bağımlıysa.
wire DATA_DEPENDENCY_HAZARD_R1_WRITEBACK =
       R1_PIPELINE[EXECUTE] != 0
       && TYPE_PIPELINE[EXECUTE] != TYPE_UPPERIMMEDIATE
       && R1_PIPELINE[EXECUTE] == RD_PIPELINE[WRITEBACK];


// Eğer R2 iki önceki komutun(Geri Yaz aşaması) RD'sine bağımlıysa.
wire DATA_DEPENDENCY_HAZARD_R2_WRITEBACK =
       R2_PIPELINE[EXECUTE] != 0
       && TYPE_PIPELINE[EXECUTE] != TYPE_UPPERIMMEDIATE
       && TYPE_PIPELINE[EXECUTE] != TYPE_IMMEDIATE
       && R2_PIPELINE[EXECUTE] == RD_PIPELINE[WRITEBACK];
```

Yukarıda tanımlı kablolar Çalıştır aşamasındaki bir komutun Bellek veya Geri Yaz aşamasındaki bir komutun sonucuna bağımlı olup olmadığını tespit etmek için kullanılmıştır.

R1 yazmaçının bağımlılığının tespiti için Üst Direkt(Upper Immediate) komutlar dışındaki tüm komutlar, R2 yazmaçının bağımlılığının tespiti için ise hem Üst Direkt(Upper Immediate), hem de Direkt(Immediate) dışındaki komutlarda bağımlılık aranmıştır.

Üst Direkt ve Direkt komutlarında Şekil 2.5'te görüldüğü üzere R1 ve R2 yazmaçının belirtildiği yeri direkt değer işgal ettiği için düşük bir ihtimal de olsa direkt değer R1'in bulunduğu kısımdaki bit değerleri ile önceki komutun RD değeri eşit çıkabilir ve bağımlılık olmadığı halde varmış gibi işleme tabi tutulabilir.

CPU.v dosya içeriği devamı:

```
// Eğer Load komutundan sonra gelen komut, Load komutuna bağımlıysa
wire LOAD_STALL =
              TYPE_PIPELINE[EXECUTE] == TYPE_LOAD
       && (
              (
                     TYPE_PIPELINE[DECODE] != TYPE_UPPERIMMEDIATE
                     && TYPE_PIPELINE[DECODE] != TYPE_IMMEDIATE
                     && (
(R1_PIPELINE[DECODE] != 0 && R1_PIPELINE[DECODE] == RD_PIPELINE[EXECUTE])
|| (R2_PIPELINE[DECODE] != 0 && R2_PIPELINE[DECODE] == RD_PIPELINE[EXECUTE])
                     )
              )
       || (
              TYPE_PIPELINE[DECODE] == TYPE_IMMEDIATE
              && R1_PIPELINE[DECODE] != 0
              && R1_PIPELINE[DECODE] == RD_PIPELINE[EXECUTE]
              )
       );
```

Şekil 3.7'de gösterildiği üzere eğer bellek yükleme komutundan bir çevrim sonra gelen komut bellek yükleme komutunun sonucuna bağlıysa verinin ara kabloyla aktarılmadan önce bellekten getirilmiş olması için boru hattı 1 çevrimlik bekletilmesi gerekmektedir. LOAD_STALL kablosu ise bu bekletme işleminin yapılıp yapılmayacağını kontrol edeceğimiz kablodur. Bu kablolar yazılımsal programlardaki *boolean* tipli yani *true* veya *false* değerler alabilen değişkenler olarak düşünülebilir.

CPU.v dosya içeriği devamı:

```
wire CONTROL_HAZARD_STALL =
       INSTRUCTION_DECODE_2[6:0] == OP_B_TYPE
       || INSTRUCTION_EXECUTE_3[6:0] == OP_B_TYPE;
```

Kontrol Sorununun mevcut olması ise dallanma komutunun Çöz ve Çalıştır aşamalarından geçene kadardır. Çalıştır aşamasından sonra artık artık dallanma yapılıp yapılmayacağı kararı çıktığı için beklemeye gerek kalmayacaktır.

1.24 Ek Modüllerin Tanımlamaları

Bu kısımda İşlemci'de kullanmak üzere oluşturduğumuz ve EK MODÜLLER bölümünde iç tasarımlarından bahsettiğimiz modülleri artık ihtiyacımıza göre ekleyip kullanacağız. İlk modülümüz Direkt Değer Ayrıştırıcısı(Immediate Extractor).

CPU.v dosya içeriği devamı:

```
wire [63:0] IMMEDIATE_VALUE;
wire [2:0] IMMEDIATE_SELECTION;
wire [7:0] immediateSelectionInputs;

assign immediateSelectionInputs[0] = 0;
assign immediateSelectionInputs[1] = I_TYPE;
assign immediateSelectionInputs[2] = U_TYPE_LOAD || U_TYPE_AUIPC;
assign immediateSelectionInputs[3] = S_TYPE;
assign immediateSelectionInputs[4] = B_TYPE;
assign immediateSelectionInputs[5] = U_TYPE_JUMP;
assign immediateSelectionInputs[6] = 0;
assign immediateSelectionInputs[7] = 0;
Encoder_8 immediateSelectionEncoder(immediateSelectionInputs,
IMMEDIATE_SELECTION);
ImmediateExtractor immediateExtractor(INSTRUCTION_EXECUTE_3,
IMMEDIATE_SELECTION, IMMEDIATE_VALUE);
```

Bu modüle Çalıştır aşamasındaki buyruk ve hangi direkt değer tipine göre ayrıştırma işlemini gerçekleştireceğimizi belirtmek için Kodlayıcı ile ikilik sisteme

dönüştürülmüş IMMEDIATE_SELECTION girdileri verilmiştir. Çıkış olarak ise 64 bitlik direkt değer IMMEDIATE_VALUE kablosuna atanmıştır.

Sonraki modülümüz ise en önemli modüllerden biri olan ALU. Fakat ALU'yu eklemeden önce ALU modülünde kullanacağımız girişler hazırlanmıştır.

CPU.v dosya içeriği devamı:

```
wire [15:0] aluOpEncoderInputs;
wire [3:0] ALU_OP;
assign aluOpEncoderInputs[0] = R_add || R_addw || I_addi || I_addiw;
assign aluOpEncoderInputs[1] = R_sub || R_subw;
assign aluOpEncoderInputs[2] = R_and || I_andi;
assign aluOpEncoderInputs[3] = R_or || I_ori;
assign aluOpEncoderInputs[4] = R_xor || I_xori;
assign aluOpEncoderInputs[5] = R_sll || R_sllw || I_slli || I_slliw;
assign aluOpEncoderInputs[6] = R_srl || R_srlw || I_srli || I_srliw;
assign aluOpEncoderInputs[7] = R_sra || R_sraw || I_srai || I_sraiw;
assign aluOpEncoderInputs[8] = R_mul || R_mulw;
assign aluOpEncoderInputs[9] = R_mulh;
assign aluOpEncoderInputs[10] = R_div || R_divw;
assign aluOpEncoderInputs[11] = R_rem || R_remw;
assign aluOpEncoderInputs[12] = R_slt || I_slti;
assign aluOpEncoderInputs[13] = R_sltu || I_sltiu;
assign aluOpEncoderInputs[14] = 0;
assign aluOpEncoderInputs[15] = 0;
Encoder_16 aluOpEncoder(aluOpEncoderInputs, ALU_OP);
```

ALU'da hangi işlemin yapılacağını 16-4'lük kodlayıcı ile ALU_OP kablosuna atamış olduk. Şimdi ise ALU'nun girişlerini elde edelim.

CPU.v dosya içeriği devamı:

```
wire [3:0] aluX1SelectionInputs;
wire [3:0] aluX2SelectionInputs;
wire [1:0] ALU_X1_SEL;
wire [1:0] ALU_X2_SEL;
assign aluX1SelectionInputs[0] = 1;
assign aluX1SelectionInputs[1] =
       B_TYPE || U_TYPE_JUMP || U_TYPE_AUIPC || I_TYPE_JUMP;
```

```
assign aluX1SelectionInputs[2] = U_TYPE_LOAD;
assign aluX1SelectionInputs[3] = 0;

assign aluX2SelectionInputs[0] = 1;
assign aluX2SelectionInputs[1] = S_TYPE || I_TYPE || B_TYPE || U_TYPE;
assign aluX2SelectionInputs[2] = 0;
assign aluX2SelectionInputs[3] = 0;
Encoder_4 aluX1SelectionEncoder(aluX1SelectionInputs, ALU_X1_SEL);
Encoder_4 aluX2SelectionEncoder(aluX2SelectionInputs, ALU_X2_SEL);
```

ALU'ya vereceğimiz girişlerin hangileri olduğunu seçeceğimiz ALU_X1_SEL ve ALU_X2_SEL kabloları kodlayıcıların çıkışlarına atanmıştır. Artık ALU modülünü tanımlayabiliriz.

CPU.v dosya içeriği devamı:

```
reg [63:0] ALU_X1;
reg [63:0] ALU_X2;
wire [63:0] ALU_OUT;
wire isALUEqual;
ALU alu(ALU_X1, ALU_X2, ALU_OP, ALU_OUT, isALUEqual);

always @(*) begin
	case(ALU_X1_SEL)
		0: ALU_X1 <= R1_DATA;
		1: ALU_X1 <= PC_EXECUTE_3;
		2: ALU_X1 <= 0;
		default: ALU_X1 <= 0;
	end

	case(ALU_X2_SEL)
		0: ALU_X2 <= R2_DATA;
		1: ALU_X2 <= IMMEDIATE_VALUE;
		default: ALU_X2 <= 0;
	endcase
end
```

Geriye kalan RegFile modülünde ise yazmaçlara yazma işlemi son aşama olan Geri Yaz aşamasında gerçekleştiği için ve buyruk tipine göre yazım denetimini sağlayabilmek için buyruk tipleri Geri Yaz aşamasındaki buyruk için tekrar elde edilmiştir.

CPU.v dosya içeriği devamı:

```
wire [6:0] OPCODE_WRITEBACK_5 = INSTRUCTION_WRITEBACK_5[6:0];
wire WB_R_TYPE             = OPCODE_WRITEBACK_5 == OP_R_TYPE;
wire WB_R_TYPE_64          = OPCODE_WRITEBACK_5 == OP_R_TYPE_64;
wire WB_I_TYPE_LOAD        = OPCODE_WRITEBACK_5 == OP_I_TYPE_LOAD;
wire WB_I_TYPE_OTHER       = OPCODE_WRITEBACK_5 == OP_I_TYPE_OTHER;
wire WB_I_TYPE_64          = OPCODE_WRITEBACK_5 == OP_I_TYPE_64;
wire WB_I_TYPE_JUMP        = OPCODE_WRITEBACK_5 == OP_I_TYPE_JUMP;
wire WB_I_TYPE             = WB_I_TYPE_JUMP || WB_I_TYPE_LOAD ||
                           WB_I_TYPE_OTHER || WB_I_TYPE_64;
wire WB_U_TYPE_LOAD        = OPCODE_WRITEBACK_5 == OP_U_TYPE_LOAD;
wire WB_U_TYPE_JUMP        = OPCODE_WRITEBACK_5 == OP_U_TYPE_JUMP;
wire WB_U_TYPE_AUIPC       = OPCODE_WRITEBACK_5 == OP_U_TYPE_AUIPC;
wire WB_U_TYPE             = WB_U_TYPE_JUMP || WB_U_TYPE_LOAD ||
                           WB_U_TYPE_AUIPC;

wire REG_WRITE_ENABLE = WB_R_TYPE || WB_R_TYPE_64 || WB_I_TYPE || WB_U_TYPE;
```

RegFile modülünün yazım denetim kablosu da tanımlandıktan sonra bir de hangi verinin yazılacağı denetimini yapmak için REG_WRITEBACK_SELECTION kablosuna kodlayıcı ile değer ataması yapılmıştır.

CPU.v dosya içeriği devamı:

```
wire [3:0] regWritebackSelectionInputs;
wire [1:0] REG_WRITEBACK_SELECTION;

assign regWritebackSelectionInputs[0] = 0;
assign regWritebackSelectionInputs[1] = WB_R_TYPE || WB_R_TYPE_64 ||
      WB_U_TYPE_LOAD || WB_I_TYPE_OTHER || WB_I_TYPE_64;
assign regWritebackSelectionInputs[2] = WB_U_TYPE_JUMP || WB_I_TYPE_JUMP;
assign regWritebackSelectionInputs[3] = WB_I_TYPE_LOAD;
Encoder_4 writeBackSelectionEncoder(regWritebackSelectionInputs,
REG_WRITEBACK_SELECTION);
```

RegFile'daki yazmaça yazılacak veride load komutuyla bellekten okunmuş bir değer atanacaksa, yazmaça yazılacak veri RAM_READ_DATA_WRITEBACK_5, diğer

komutlar için ise REG_WRITE_DATA_WRITEBACK_5 değeri olarak seçilmiştir. Bu değerler Boru Hattı Tasarımı bölümünde tanımlanmıştır.

CPU.v dosya içeriği devamı:

```
wire signed[63:0] R1_DATA_REGFILE;
wire signed [63:0] R2_DATA_REGFILE;

wire [63:0] REG_WRITE_DATA = REG_WRITEBACK_SELECTION == 3
                           ? RAM_READ_DATA_WRITEBACK_5
                           : REG_WRITE_DATA_WRITEBACK_5;

RegFile regFile(R1, R2, RD, REG_WRITE_DATA, REG_WRITE_ENABLE,
R1_DATA_REGFILE, R2_DATA_REGFILE);
```

1.25 Boru Hattı Tasarımı

Tüm parametrelerini ve modüllerini oluşturduğumuz işlemcimizin geriye boru hattı mekanizmasını tanımlamak kaldı.

CPU.v dosya içeriği devamı:

```
// -- 1. Aşama: Getir(Fetch)
reg [9:0] PC = 0;
assign INSTRUCTION_ADDR = PC >> 2;

always @(posedge CLK) begin
	// Program Sayacı(Program Counter) atamaları
	if (PC_ALU_SEL == 1) begin
		PC <= ALU_OUT[9:0];
	end
	else begin
		if (LOAD_STALL == 1 || CONTROL_HAZARD_STALL == 1)
			PC <= PC;
		else
			PC <= PC + 4;
	end

	// Boru Hattı Kontrol Yazmaçları atamaları.
```

```
        if (CONTROL_HAZARD_STALL == 1) begin
                R1_PIPELINE[DECODE] <= 0;
                R2_PIPELINE[DECODE] <= 0;
                RD_PIPELINE[DECODE] <= 0;
                TYPE_PIPELINE[DECODE] <= TYPE_IMMEDIATE;
        end
        else begin
                R1_PIPELINE[DECODE] <= INSTRUCTION[19:15];
                R2_PIPELINE[DECODE] <= INSTRUCTION[24:20];
                RD_PIPELINE[DECODE] <= INSTRUCTION[11:7];

                if (INSTRUCTION[6:0] == OP_R_TYPE
                        || INSTRUCTION[6:0] == OP_R_TYPE_64)
                        TYPE_PIPELINE[DECODE] <= TYPE_REGISTER;

                else if (INSTRUCTION[6:0] == OP_I_TYPE_LOAD)
                        TYPE_PIPELINE[DECODE] <= TYPE_LOAD;


                else if (INSTRUCTION[6:0] == OP_S_TYPE)
                        TYPE_PIPELINE[DECODE] <= TYPE_STORE;

                else if (INSTRUCTION[6:0] == OP_I_TYPE_OTHER
                        || INSTRUCTION[6:0] == OP_I_TYPE_64
                        || INSTRUCTION[6:0] == OP_I_TYPE_JUMP)
                        TYPE_PIPELINE[DECODE] <= TYPE_IMMEDIATE;

                else if (INSTRUCTION[6:0] == OP_B_TYPE[6:0])
                        TYPE_PIPELINE[DECODE] <= TYPE_BRANCH;
        end
end
```

Getir aşamasında PC <= PC + 4 şeklinde arttırılarak bir sonraki komutun ROM'dan çağırılması sağlanmıştır. ROM modülümüz her hücresinde 32 bit (4 byte) veri sakladığı için işlemcimizin ROM modülüne bağlı olan INSTRUCTION_ADDR çıkışına PC değerinin iki kere sağa kaydırılmış, yani 4'e bölünmüş hali atanmıştır.

Getir aşamasında aynı zamanda boru hattı sorunları kontrolünde kullanılacak R1, R2, RD ve Buyruk Tipi yazmaçları da bir sonraki saat vuruşunda DECODE aşamasındaki yazmaçlara yazılmak üzere ayarlanmıştır.

CPU.v dosya içeriği devamı:

```
// -- 2. Aşama: Çöz(Decode)
reg [9:0] PC_DECODE_2 = 0;
reg [31:0] INSTRUCTION_DECODE_2 = 0;

always @(posedge CLK) begin

	if (LOAD_STALL == 1) begin
		INSTRUCTION_DECODE_2 <= INSTRUCTION_DECODE_2;
		PC_DECODE_2 <= PC_DECODE_2;
	end
	else if (CONTROL_HAZARD_STALL == 1) begin
		INSTRUCTION_DECODE_2 <= 32'h00000013;
		PC_DECODE_2 <= PC_DECODE_2;
	end
	else begin
		INSTRUCTION_DECODE_2 <= INSTRUCTION;
		PC_DECODE_2 <= PC;
	end

	if (LOAD_STALL == 1) begin
		R1_PIPELINE[EXECUTE] <= 0;
		R2_PIPELINE[EXECUTE] <= 0;
		RD_PIPELINE[EXECUTE] <= 0;
		TYPE_PIPELINE[EXECUTE] <= TYPE_IMMEDIATE;
	end
	else begin
		R1_PIPELINE[EXECUTE] <= INSTRUCTION_DECODE_2[19:15];
		R2_PIPELINE[EXECUTE] <= INSTRUCTION_DECODE_2[24:20];
		RD_PIPELINE[EXECUTE] <= INSTRUCTION_DECODE_2[11:7];

		if (INSTRUCTION_DECODE_2[6:0] == OP_R_TYPE
			|| INSTRUCTION_DECODE_2[6:0] == OP_R_TYPE_64)
			TYPE_PIPELINE[EXECUTE] <= TYPE_REGISTER;

		else if (INSTRUCTION_DECODE_2[6:0] == OP_I_TYPE_LOAD)
			TYPE_PIPELINE[EXECUTE] <= TYPE_LOAD;

		else if (INSTRUCTION_DECODE_2[6:0] == OP_S_TYPE)
			TYPE_PIPELINE[EXECUTE] <= TYPE_STORE;

		else if (INSTRUCTION_DECODE_2[6:0] == OP_I_TYPE_OTHER
			|| INSTRUCTION_DECODE_2[6:0] == OP_I_TYPE_64
```

```
                || INSTRUCTION_DECODE_2[6:0] == OP_I_TYPE_JUMP)
                TYPE_PIPELINE[EXECUTE] <= TYPE_IMMEDIATE;

        else if (INSTRUCTION_DECODE_2[6:0] == OP_B_TYPE[6:0])
                TYPE_PIPELINE[EXECUTE] <= TYPE_BRANCH;


    end
end
```

Görüldüğü gibi her boru hattı aşamasının o aşamadaki bilgileri tutacak yazmaçları ayrı ayrı tanımlanmıştır. PC ve INSTRUCTION değerlerinin her aşamada ayrı ayrı tutulduğu yazmaçları vardır.

Çöz aşamasında LOAD_STALL durumunda INSTRUCTION_DECODE_2 yazmaçı kendi değerlerini tekrar alırken CONTROL_HAZARD_STALL durumunda yani dallanma buyruğu geldikten sonra yeni komut alınmaz ve boru hattına NOP yani addi x0 x0 0 buyruğu eklenir ve hat devam ettirilir.

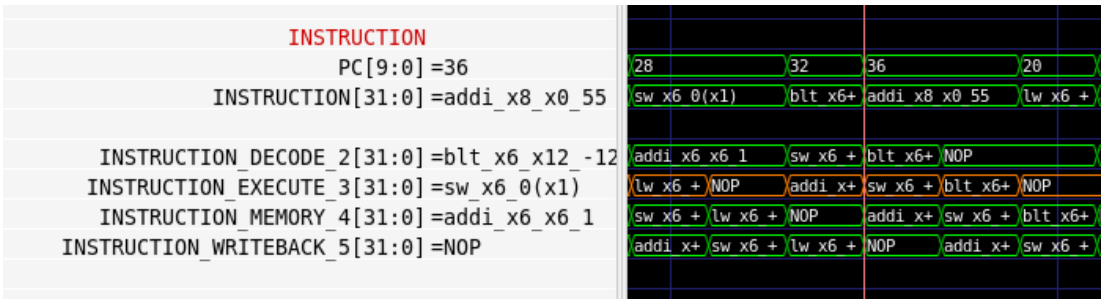

Figure 3.9: Çöz aşamasına dallanma buyruğu blt geldikten sonra sonuç oluşup yeni buyruk gelene kadar NOP eklenmesi.

CPU.v dosya içeriği devamı:

```
// -- 3. Aşama: Çalıştır(Execute)
reg [9:0] PC_EXECUTE_3 = 0;
reg [31:0] INSTRUCTION_EXECUTE_3 = 0;
always @(posedge CLK) begin
        if (LOAD_STALL == 1) begin
                INSTRUCTION_EXECUTE_3 <= 32'h00000013;
                PC_EXECUTE_3 <= PC_EXECUTE_3;
        end
        else begin
```

```
            PC_EXECUTE_3 <= PC_DECODE_2;
            INSTRUCTION_EXECUTE_3 <= INSTRUCTION_DECODE_2;
        end

        R1_PIPELINE[MEMORY] <= INSTRUCTION_EXECUTE_3[19:15];
        R2_PIPELINE[MEMORY] <= INSTRUCTION_EXECUTE_3[24:20];
        RD_PIPELINE[MEMORY] <= INSTRUCTION_EXECUTE_3[11:7];

        if (INSTRUCTION_EXECUTE_3[6:0] == OP_R_TYPE
        || INSTRUCTION_EXECUTE_3[6:0] == OP_R_TYPE_64)
            TYPE_PIPELINE[MEMORY] <= TYPE_REGISTER;

        else if (INSTRUCTION_EXECUTE_3[6:0] == OP_I_TYPE_LOAD)
            TYPE_PIPELINE[MEMORY] <= TYPE_LOAD;

        else if (INSTRUCTION_EXECUTE_3[6:0] == OP_S_TYPE)
            TYPE_PIPELINE[MEMORY] <= TYPE_STORE;

        else if (INSTRUCTION_EXECUTE_3[6:0] == OP_I_TYPE_OTHER
        || INSTRUCTION_EXECUTE_3[6:0] == OP_I_TYPE_64
        || INSTRUCTION_EXECUTE_3[6:0] == OP_I_TYPE_JUMP)
            TYPE_PIPELINE[MEMORY] <= TYPE_IMMEDIATE;

        else if (INSTRUCTION_EXECUTE_3[6:0] == OP_B_TYPE[6:0])
            TYPE_PIPELINE[MEMORY] <= TYPE_BRANCH;
    end
```

Çalıştır aşamasında da eğer LOAD_STALL aktif ise boru hattına NOP buyruğu eklenmiştir. Böylece Çalıştır aşamasından önceki komutlar bir çevrim bekletilmiş, sadece Bellek ve Geri Yaz aşamaları yürütülmüştür.

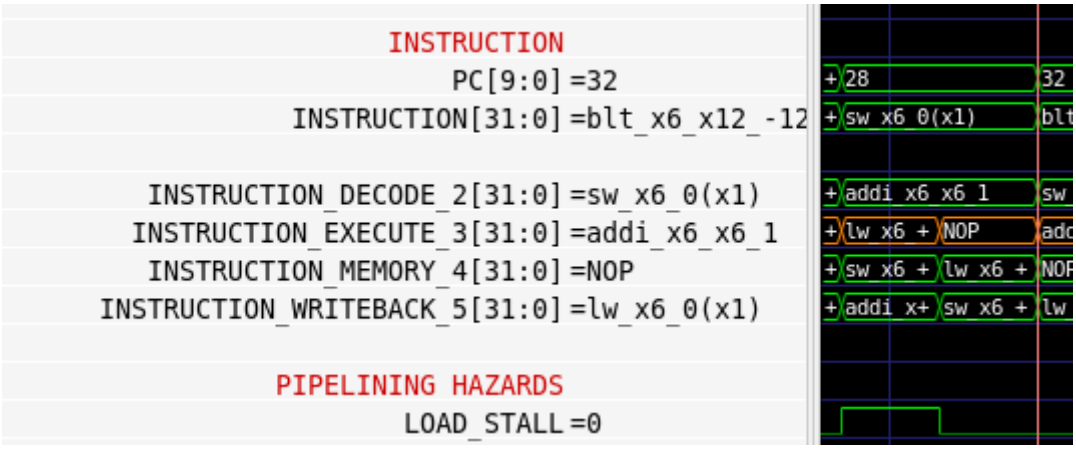

Figure 3.10: Çalıştır aşamasına gelen bağımlı load buyruğu sonrası 1 çevrimlik bekleme için eklenen NOP buyruğu.

CPU.v dosya içeriği devamı:

```
// -- 4. Aşama: Bellek(Memory)
reg [9:0] PC_MEMORY_4 = 0;
reg [31:0] INSTRUCTION_MEMORY_4 = 0;
reg [63:0] ALU_OUT_MEMORY_4 = 0;

always @(posedge CLK) begin
	INSTRUCTION_MEMORY_4 <= INSTRUCTION_EXECUTE_3;
	PC_MEMORY_4 <= PC_EXECUTE_3;

	ALU_OUT_MEMORY_4 <= ALU_OUT;
	RAM_WRITE_DATA <= R2_DATA;

	R1_PIPELINE[WRITEBACK] <= INSTRUCTION_MEMORY_4[19:15];
	R2_PIPELINE[WRITEBACK] <= INSTRUCTION_MEMORY_4[24:20];
	RD_PIPELINE[WRITEBACK] <= INSTRUCTION_MEMORY_4[11:7];

	if (INSTRUCTION_MEMORY_4[6:0] == OP_R_TYPE
	|| INSTRUCTION_MEMORY_4[6:0] == OP_R_TYPE_64)
		TYPE_PIPELINE[WRITEBACK] <= TYPE_REGISTER;

	else if (INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_LOAD)
		TYPE_PIPELINE[WRITEBACK] <= TYPE_LOAD;

	else if (INSTRUCTION_MEMORY_4[6:0] == OP_S_TYPE)
		TYPE_PIPELINE[WRITEBACK] <= TYPE_STORE;

	else if (INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_OTHER
	|| INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_64
	|| INSTRUCTION_MEMORY_4[6:0] == OP_I_TYPE_JUMP)
		TYPE_PIPELINE[WRITEBACK] <= TYPE_IMMEDIATE;

	else if (INSTRUCTION_MEMORY_4[6:0] == OP_B_TYPE[6:0])
		TYPE_PIPELINE[WRITEBACK] <= TYPE_BRANCH;
end
```

Bellek aşamasında artık herhangi bir buyruk bekletme işlemine gerek kalmamıştır çünkü bekleme kontrolünü önceki aşamalar yapmaktadır. Ayrıca önceki aşamalara ek olarak ALU çıktısının sonraki aşamaya aktarılacağı yazmaç olan

ALU_OUT_MEMORY_4 yazmaçı da eklenmiştir. Yine ek olarak RAM modülünün veri girişine bağlı olan RAM_WRITE_DATA çıkışına R2_DATA değeri atanmıştır.

CPU.v dosya içeriği devamı:

```
reg [31:0] INSTRUCTION_WRITEBACK_5 = 0;
reg [63:0] REG_WRITE_DATA_WRITEBACK_5 = 0;
reg [63:0] RAM_READ_DATA_WRITEBACK_5 = 0;

always @(posedge CLK) begin
	INSTRUCTION_WRITEBACK_5 <= INSTRUCTION_MEMORY_4;
	RAM_READ_DATA_WRITEBACK_5 <= RAM_READ_DATA;

	case (REG_WRITEBACK_SELECTION)
		1: REG_WRITE_DATA_WRITEBACK_5 <= ALU_OUT_MEMORY_4;
		2: REG_WRITE_DATA_WRITEBACK_5 <= PC_MEMORY_4 + 4;
	endcase
end
```

Son aşamamız olan Geri Yaz aşamasında artık RAM'den bir değer okunduysa bu değer RAM_READ_DATA_WRITEBACK_5 yazmaçına kaydedilmiştir. Ayrıca RegFile modülünde yazılacak olan yazmaça hangi değerin yazılacağına da karar verilmiştir.

4 SONUÇLAR VE ÖNERİLER

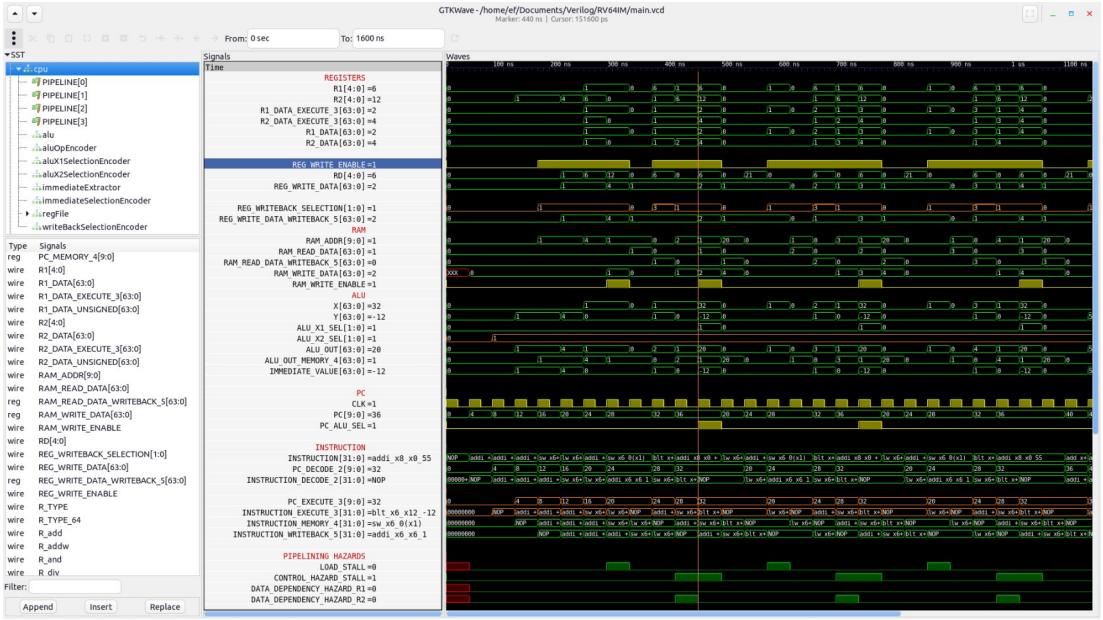

Figure 4.1: Tasarlanan işlemcinin iç bileşenlerinin başarıyla çalışma durumunun GTKWave ile dalga analizleri ile test edilerek tamamlanması.

Bu projede sonuç olarak 64 bit, 5 aşamalı boru hattına sahip, harvard mimarisiyle tasarlanmış bir RV64IM işlemci çekirdeği Verilog dili ile tasarlanıp başarıyla simüle edilmiştir.

Çalışmanın daha da geliştirilerek işlemci çekirdeğinin işletim sistemi çalıştırabilen RV64IMAFDC hale getirilmesi, Sırasız İcra(Out-of-Order Execution) kabiliyetinin eklenmesi, Çok Çekirdekli(Multi Core) çalışma desteği getirilmesi, RAM bellek ile arasındaki iletişimi azaltmak ve hızlandırmak için Önbellek(Cache) eklenmesi ve Önbellek Tutarlılığı(Cache Coherency) sistemleri eklenmesi, Dallanma Tahmini(Branch Prediction) gibi yöntemler eklenmesi, İşletim Sistemi çalıştırmak için kullanılan Ayrıcalık Modları(Privilege Mode) gibi yapılarla geliştirilmesi düşünülmektedir.

Ayrıca bu tasarım bir FPGA üzerinde CoreMark gibi benchmark testlerine tabi tutularak eksik yönleri tespit edilip iyileştirilebilir.

KAYNAKLAR

[1] https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/

[2] https://eminfedar.com/riscv-nedir

[3] https://riscv.org/about/risc-v-branding-guidelines/

[4] https://en.wikipedia.org/wiki/Riscv

[5] RISC-V Instruction Set Manual, https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[6] Computer Organization and Design RISC-V Edition, DAVID A. PATTERSON, JOHN L. HENNESSY