T.C.

İSTANBUL ÜNİVERSİTESİ – CERRAHPAŞA

MÜHENDİSLİK FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

2021 – 2022 BAHAR DÖNEMİ

BİLGİSAYAR MİMARİSİ DERSİ

PIPELINED 16-BIT RISC İŞLEMCİ

TASARIM VE TEST ÖDEVİ

ASIM KAYMAK – 1306180004

AYŞE ÖZGÜR – 1306170069

YAHYA BOYALI – 1306180003

YUNUS KARA – 1306180061

YUSUF UMUT BULAK – 1306180025

İçindekiler Tablosu

GENEL BAKIŞ

Projede verilen görev olan 16-bit Pipelined RISC işlemcinin Logisim programında oluşturulmasına geçilmeden önce bu işlemcinin veri yolu (datapath) manuel olarak çizilmiş ve doğruluğundan da emin olunmuştur. Çizilen veri yolunun doğruluğundan emin olduktan sonra aynı tasarım Logisim üzerinde de çizilmiştir. Fakat doğrudan çizim yapmadan önce tüm yapıyı küçük modül devrelere bölüp önce o yapıların tasarımları tamamlanmıştır. Devamında çizilen 16-bit Pipelined RISC işlemciye tek komutluk testler uygulanarak uygun çıktılar beklenmiştir. Tek komutluk testlerin doğru çalıştığının kontrolü sağlandıktan sonra test dosyalarındaki komutlar test edilmiş ve rapora eklenmiştir.

1.1. PLANLAMA VE GÖREV DAĞILIMI

Yapılan ortak online toplantılar sonucunda aşağıda tabloda verilen görevler paylaştırılmış ve her görev o kişi tarafından yapılmıştır. Haftalık toplantılarla görevlerde yaşanılan problemler tartışılmış ve ortak çözümler üretilmiştir. Bireysel görevlerin tamamlanmasının ardından tüm grup üyeleri yapılan devre modüllerini kullanarak birlikte tüm işlemcinin veri yolunu tasarlamıştır. Veri yolunun tasarlanıp tamamlanmasının ardından yine tüm grup üyeleri testleri birlikte gerçekleştirmiş ve test sonuçlarının doğruluğu kontrol edilmiştir. Tüm görevler bitirildikten sonra raporlama aşamasına geçilmiş ve ödev sürecinin tamamı detaylıca ilgili dosyada raporlanmıştır.

Görev \ Grup Üyesi	Asım Kaymak	Ayşe Özgür	Yahya Boyalı	Yunus Kara	Yusuf Umut Bulak
16-bit Register		X			
Registers		X	X		
Control Unit	X				
ALU Control	X				
ALU	X				
R-Type Control	X				
Extender		X			
Instruction Fetch Buffer				X	
Instruction Decode Buffer				X	
Execution Buffer					X
Memory Buffer					X
Forward Hazard			X		
PC Control			X		
İşlemci Veri Yolu	X	X	X	X	X
Testler	X	X	X	X	X
Rapor	X	X	X	X	X

2. 16-BIT PIPELINED RISC İŞLEMCİ VERİ YOLU

Pipeline ya da Türkçe ismiyle boru hattı işlemci yapısında amaç komut döngü süresini (CPI) azaltmaktır. Böylece birim döngü zamanına düşen komut sayısı (IPC) artmış ve diğer yapılara göre daha aynı sürede daha fazla komut çalıştırılmış olur. Bu yapıyla birlikte işlemci tek bir komutun işini hızlandırmakla uğraşmaz, toplu komutların işlerini hızlandırır. Yapıda 5 durum bulunmakta:

1- Getir (Fetch) : Komutlar, komut belleğinden getirilir.
2- Çöz (Decode) : Getirilen komutlar okunur ve çözülür.
3- Yürüt (Execute / ALU) : Bellek adresleri hesaplanır.
4- Bellek (Memory) : Veri belleklerinden gerekli veriler okunur.
5- Yaz (Write Back) : Veri belleğine ya da registerlara gerekli veriler yazılır.

Bir komut herhangi bir adımı bitirip bir sonraki adıma geçerken işlemcinin boşta kalıp tek bir komutla vakit kaybetmemesi için bir sonraki komut bir önceki adıma aktarılır ve bu süreç komutlar bitene kadar devam eder. Bunu aşağıdaki görsel üzerinden daha detaylı görebiliriz.

Clock cycle / Instr. No.	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

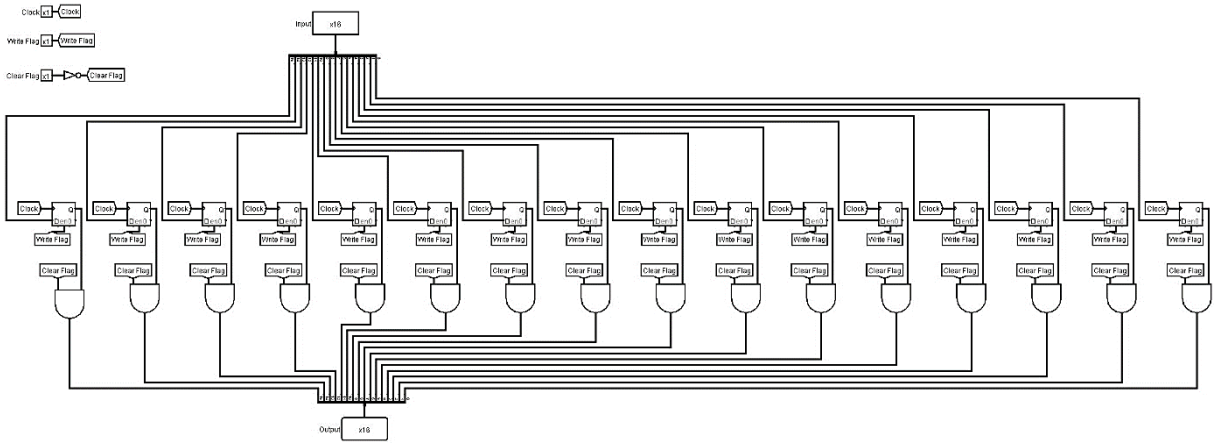
Tek çekirdekli işlemcilerde, bir komut bitince diğer komut çalışmaya başlar. Her saat vuruşunda bir komut girer, bir komut çıkar. Yani buyruk başına düşen çevrim sayısı 1'dir. Boru hattı yöntemi ise çoklu buyrukların örtüşmeli yürütümüdür. Birbirini bağlayan komutlar haricinde bir buyruğun işlemi gerçekleştirilirken diğer komut işleme girebilir. Günümüzde daha hızlı işlemci tasarımında kullanılan önemli bir yöntemdir.

Boru hattı işlemcinin komut döngü süresini azaltır bundan dolayı da birim döngü zamanına düşen komut sayısı artar. Boru hattında bütün aşamalardan geçmek gerekir. Tek bir komutun işini değil toplu komutların işlerini hızlandırır. Kısacası toplamda üretilen işi artırır. Olası hızlanma boru hattındaki aşama sayısına bağlı olarak değişir.

Projenin planlanmasında tüm işlemci sisteminin veri yolu çizimi için gerekli olan bileşenler, farklı devre parçaları olarak düşünülüp küçük modüller halinde planlanarak çizilmiştir. Tüm proje küçük modüllere parçalanmış ve önce o modüller yapılmıştır. Çizilen bileşenler aşağıdaki gibidir.

2.1. 16-BIT REGISTER

16 Bit Register için girdi olarak 16 bit veri, verinin yazım bayrağı ve saat darbesi alınmıştır. Bu girdilere ek olarak Register içeriğini sıfırlamak yani temizlemek için de bir Reset bayrağı kullanılmıştır. D Flip-flop devreleri kullanılarak Registerın 16 bit çıkışına veriler kaydedilmiştir. Böylelikle uygun girdilerle aşağıdaki devre çizimiyle birlikte veri Registera kaydedilmiştir. Register devresinin genel görünümü aşağıdaki gibidir.

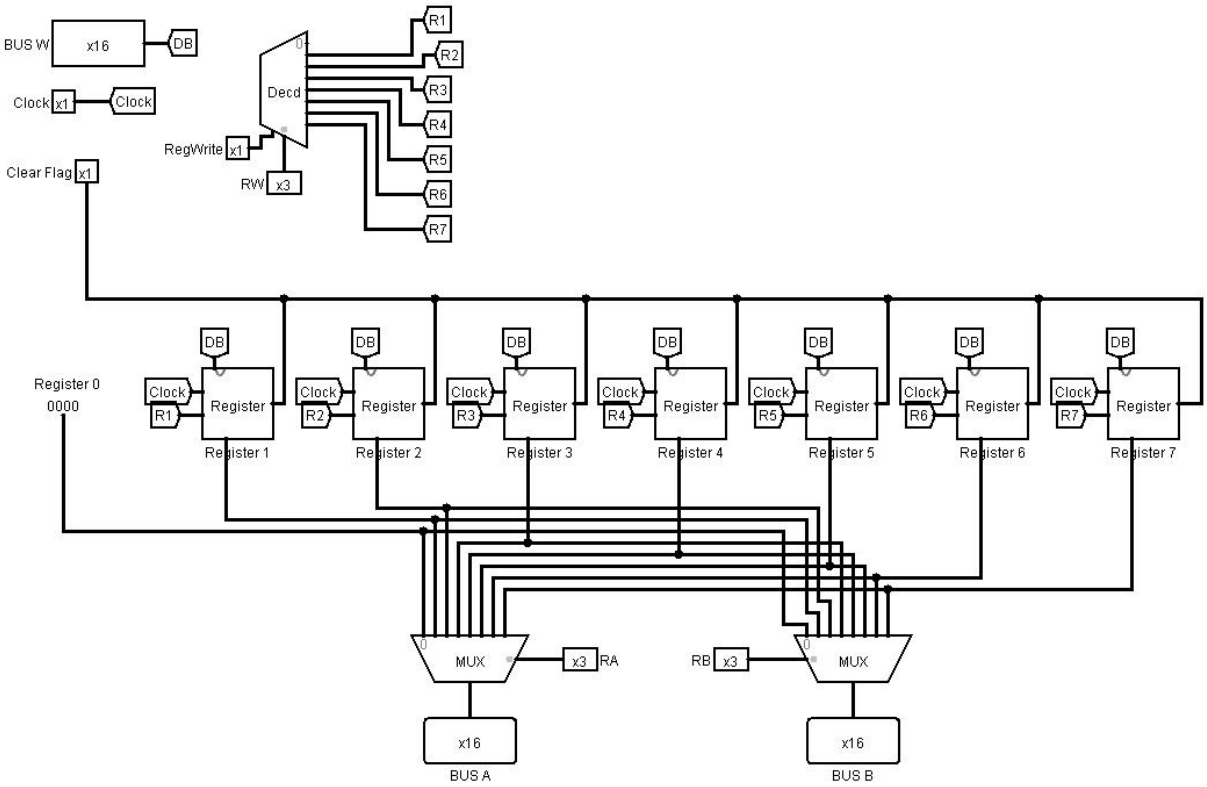


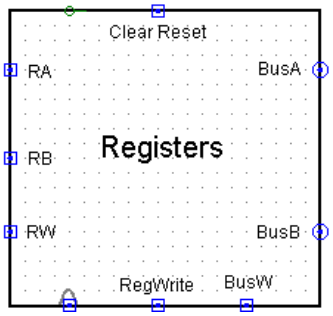
2.2. REGISTERS

Registerların tutulduğu Register kümesi (Registers) için ilk olarak ödevde verilen toplamda 8 adetten oluşan Register devre üzerinde oluşturulmuştur. Fakat ilk Registerın değeri değiştirelemeyen Zero Register olmasından dolayı aslında 7 adet Register devreye eklenmiştir. İlk Register olan Zero Register constant vALUe ve geriye kalan 7 Register ise 2.1'de oluşturulan Registerlar kullanılarak eklenmiştir.

Toplam Register sayısı 8 adet olduğundan seçilen Register'ın girdisi için 3 bitlik bir alan seçilmiştir. Bu sayede ödevde verilen komut setindeki yapıyla da uyum sağlanmıştır. Böylece hangi Register'ın seçildiği kontrolü sağlanmştır. Bu kontrolü sağlarken 3x8 Decoder bileşeni kullanılmıştır. Decodera gelen Register seçimiyle verinin yazılması gereken Registerın Write Flagi 1 yapılmış ve istenen Registera istenen verinin yazılması tamamlanmıştır.

Register kümesinden Buss A ve Buss B olmak üzere 2 adet 16 bitlik veri çıkışının sağlanabilmesi kontrolü ise Multiplexer kullanılarak sağlanmıştır. Son olarak devre üzerinde işlemleri kolaylaştırmak adına tüm Register değerlerini sıfırlayan bir Clear Reset bayrağı eklenmiştir. Registers devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.



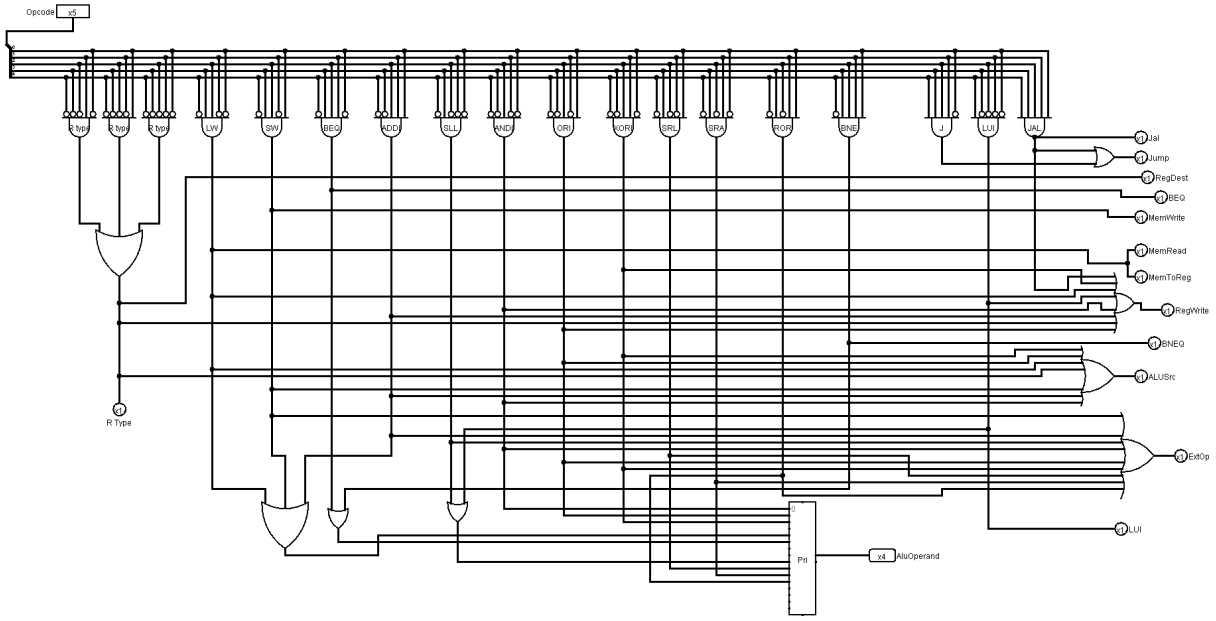


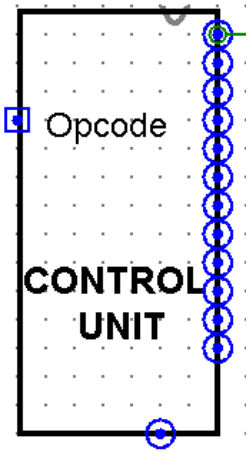
2.3. CONTROL UNIT

Kontrol birimi (Control Unit) için öncelikle ödevde verilen komutların listesi ve komut formatları detaylıca incelenmiştir. Bu inceleme sonunda komutlardaki ilk 5-bitin opcode olduğu ve R-type komutları için function bit uzunluğunun 2-bit olduğu görülmüştür. Devamında her bir komut için bir ALU operanda ihtiyaç duyulduğundan ödevde verilen komut listesine uygun bir çözüm getirilmiştir. Toplamda 9'dan fazla ALU operanda ihtiyaç olacağı bilindiğinden ALU operand için uygun bit uzunluğunun 4 bit olmasına karar verilmiştir. Böylece ihtiyaca göre 16 ALU operand karşılığı sağlanmıştır. Devamında buradan gelen komuta uygun olarak işlemci üzerindeki kontrol bayraklarının durumunun ayarlaması yapılmıştır. Aşağıdaki tabloda her komut için değiştirilen kontrol bayrak durumları ve kullanılan ALU operand değerleri gösterilmiştir.

	ALU OPERAND	ALU FUNCTION	REG DEST	REG	MEM	MEM READ	MEM TO REG	JAL	JUMP	BEQ	BNEQ	ALU SRC	EXT OP	LUI
AND	0000	AND	1	1	0	0	0	0	0	0	0	1	0	0
OR	0001	OR	1	1	0	0	0	0	0	0	0	1	0	0
XOR	0010	XOR	1	1	0	0	0	0	0	0	0	1	0	0
NOR	0011	NOR	1	1	0	0	0	0	0	0	0	1	0	0
ADD	0100	ADD	1	1	0	0	0	0	0	0	0	1	0	0
SUB	0101	SUB	1	1	0	0	0	0	0	0	0	1	0	0
SLT	0110	SLT	1	1	0	0	0	0	0	0	0	1	0	0
SLTU	0110	SLT	1	1	0	0	0	0	0	0	0	1	0	0
JR	0111	JR	1	1	0	0	0	0	0	0	0	1	0	0
ANDI	0000	AND	0	1	0	0	0	0	0	0	0	1	1	0
ORI	0001	OR	0	1	0	0	0	0	0	0	0	1	1	0
XORI	0010	XOR	0	1	0	0	0	0	0	0	0	1	1	0
ADDI	0100	ADD	0	1	0	0	0	0	0	0	0	1	1	0
SLL	1000	SLL	0	0	0	0	0	0	0	0	0	0	1	0
SRL	1001	SRL	0	0	0	0	0	0	0	0	0	0	1	0
SRA	1010	SRA	0	0	0	0	0	0	0	0	0	0	1	0
ROR	1011	ROR	0	0	0	0	0	0	0	0	0	0	1	0
LW	0100	ADD	0	1	0	1	1	0	0	0	0	1	1	0
SW	0100	ADD	0	0	1	0	0	0	0	0	0	1	1	0
BEQ	0101	SUB	0	0	0	0	0	0	0	1	0	0	0	0
BNE	0101	SUB	0	0	0	0	0	0	0	0	1	0	0	0
LUI	1000	SLL	0	1	0	0	0	0	0	0	0	0	0	1
J	XXXX	XXXX	X	X	X	X	X	X	1	X	X	X	X	X
JAL	XXX	XXXX	X	1	X	X	X	1	1	X	X	X	X	X

Bu tabloya göre Control Unit oluşturulurken R-Type komutların hepsinde bayrak durumlarının aynı olmasından dolayı R-type komutlar bir bütün olarak düşünülüp atama yapılmıştır. 5-bitlik opcode Bit Splitter kullanılarak her bir bitine göre gerekli kapılardan geçirilerek komut algılanmış ve bayraklar algılanan komuta göre değiştirilmiştir. Yine uygun komuta göre tabloda belirtilen ALU operand kodlarının da aktarımı için Priority Encoder kullanılmıştır. Control Unit devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.

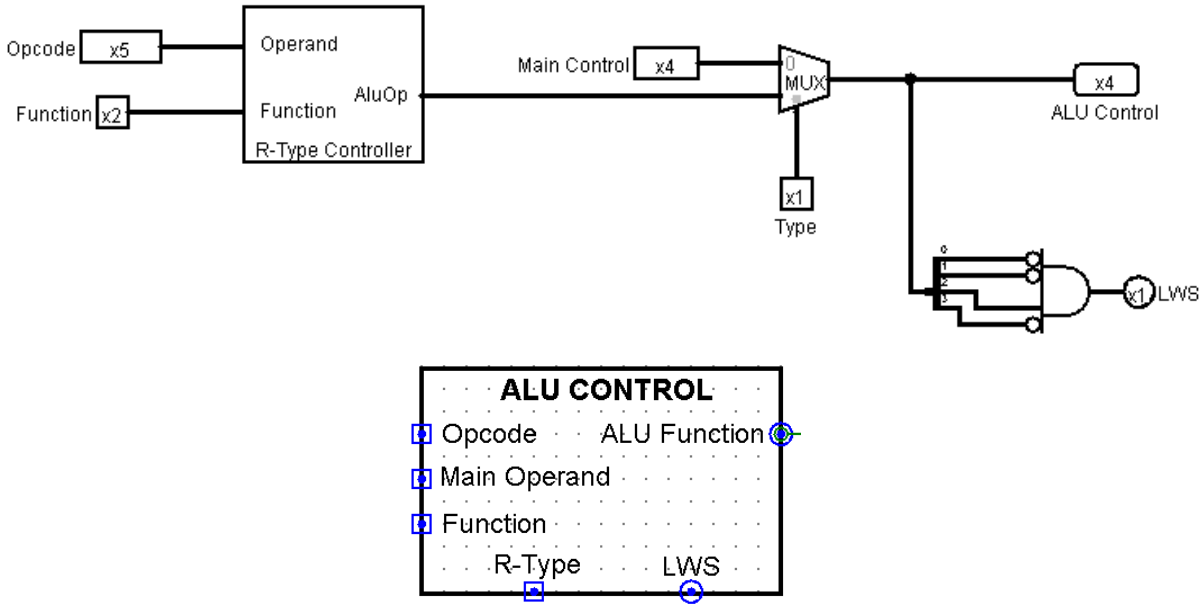




2.4. ALU CONTROL

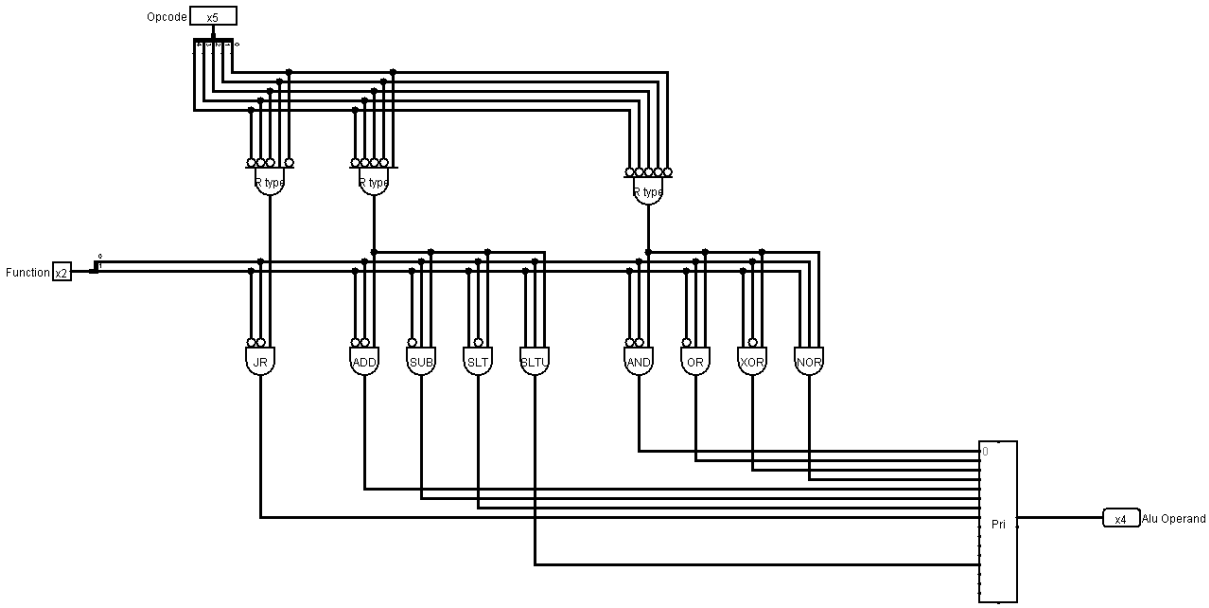
Bu modüldeki asıl amaç basitçe şöyle belirlenmiştir:

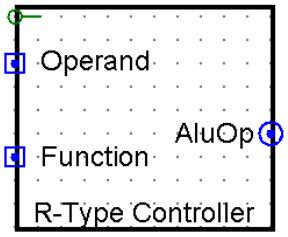
- 3 farklı R-Type opcode ve her biri için ortak function değerleri bulunduğundan dolayı input olarak Opcode ve function alanı eklenmiştir.
- Control unitten gelen R-Type komut bilgisi ve ALU operand kodu ALU control içerisine Main Control olarak eklenmiştir. R-Type komutları için ALU operand ayarlamalarının yapılabilmesi için normalde işlemcilerde bulunmayan bir component çözümü getirilmiştir.
- Gelen komut eğer R-Type bir komutsa yapı içerisindeki R-Type Control devresinin çıktı ALU Operandı kullanılmıştır. Eğer R-Type bir komut değilse doğrudan Control Unit devresinden çıktı olarak gelen ve ALU Control içerisine Main Controle bağlanan ALU Operandı kullanılmıştır. Kısaca iki farklı ALU Operand değerinden birisi komutun yapısına göre Multiplexer kullanılarak seçilmiş ve ALU Function olarak çıktı olarak ALU devre birimine iletilmiştir. Devre içerisinde ekstradan J-Type komutlar için de bazı durum bayraklarının ataması da yapılmıştır. ALU Control devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.



2.4.1. R-TYPE CONTROL

R-Type modülü ALU Control içerisinde yer alıp girdi olarak gelen opcode ve function değerlerinin alınması ile ilgili komutun algılanmasını sağlamıştır. Komutun algılamasından sonra çıktı olarak yine Priority Encoder modülü kullanılarak ALU operand çıktısı üretilmiştir. R-Type Control devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.





2.5. ALU

ALU devresi içerisinde ilk olarak 16-bitlik Buss_A girişi ve Buss_B girişi eklenmiştir. Bu sayede yapılmak istenen ikili işlemlerin hepsi için gerekli girdi alanı oluşturulmuştur. Yukarıda oluşturulan listede ALU Functionlar incelendiğinde 11 adet farklı ALU Function olduğu görülmüştür.

ALU devresinin iç mekaniğindeki yapı ise şöyle kurulmuştur:

- Her ALU işlemi için yukarıda bahsedilen 11 farklı işlemin her biri koşulsuz olarak yapılmakta ve bu işlemlerin sonuçları da 4x16 Multiplexer a yazılmaktadır.

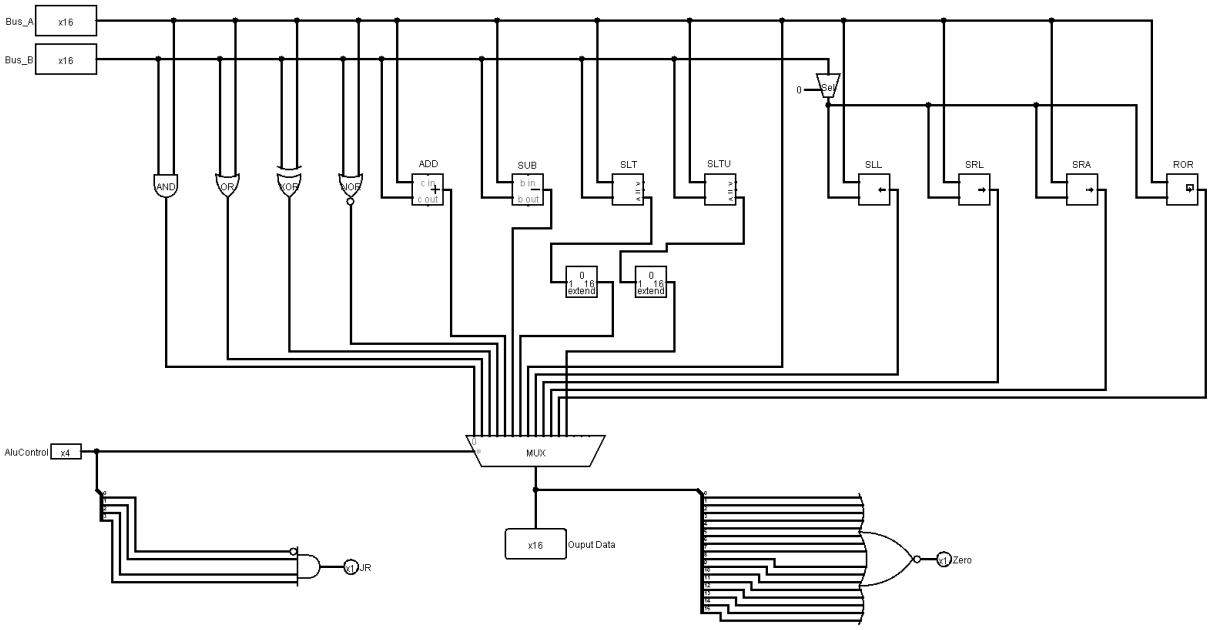
- 4x16 Multiplexer'a selector olarak ALU Control biriminden gelen ALU Function kodu bağlanmaktadır. Böylece hangi işlemin sonucunun alınacağı seçilebilir olmuştur. Örneğin add işlemi için yukarıda belirtilen tabloda ALU Function değeri 0100 olarak gelmektedir. ALU içerisinde yapılan toplama işleminin sonuç bağlantısı 4x16 Multiplexer içerisinde Input-4 alanına bağlanmalıdır ki ALU'dan sonuç verisi olarak bu bilgiye erişilebilir olunsun.

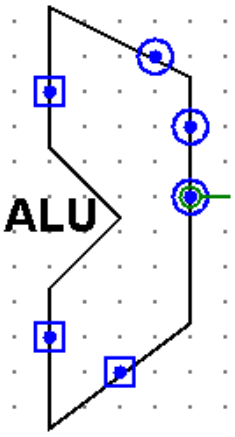
- Eğer işlem sonucu 0 çıktıysa Zero kontrol bayrağının durumunu 1 yapılıyor. Bu işlem ise tüm bitleri birbirleriyle OR işlemine sokarak yapılıyor. Bu sayede komutun J-Type bir komut olduğu algılanıyor ve doğrudan Program Counter Control ile bağlantı sağlanıyor.

- SLT ve SLTU komutlarında sonuç olarak Multiplexer içerisinde bağlandıkları yer aynı olması gerekirken yapılan işlemler farklı olduğundan burada ekstradan bir 1x2 Multiplexer kullanarak bu işlemlerin kontrolü sağlanıyor. SLT ve SLTU işlemlerinde kullanılan modüller (Comparator) aynı olsa bile SLTU için kullanılan comparatorün Numeric Type ayarı "unsigned" olarak kullanılıyor. Böylece SLT için signed karşılaştırma yapılırken SLTU için unsigned karşılaştırma istenildiği gibi yapılıyor.

- Shifting işlemlerinde Buss_B her ne kadar 16-bit olsa bile bunun sadece ilk 5 bitine ihtiyaç duyuluyor. Verilen komut seti yapısına bakıldığında shifting işlemlerinin I-Type olarak alındığını ve bu işlem için gerekli olan immediate datanın 5 bit olduğu görülüyor. 5-bitlik immediate veri Bit Extender kullanılarak 16-bite çevriliyor ve Buss_B'ye bu şekilde aktarılıyor. Tekrardan sadece ilk 5 bite ihtiyaç duyulduğundan bu sefer Bit Selector kullanılarak ilk 5-bit ayıklanıyor ve shifting operatörlere o şekilde bağlanıyor.

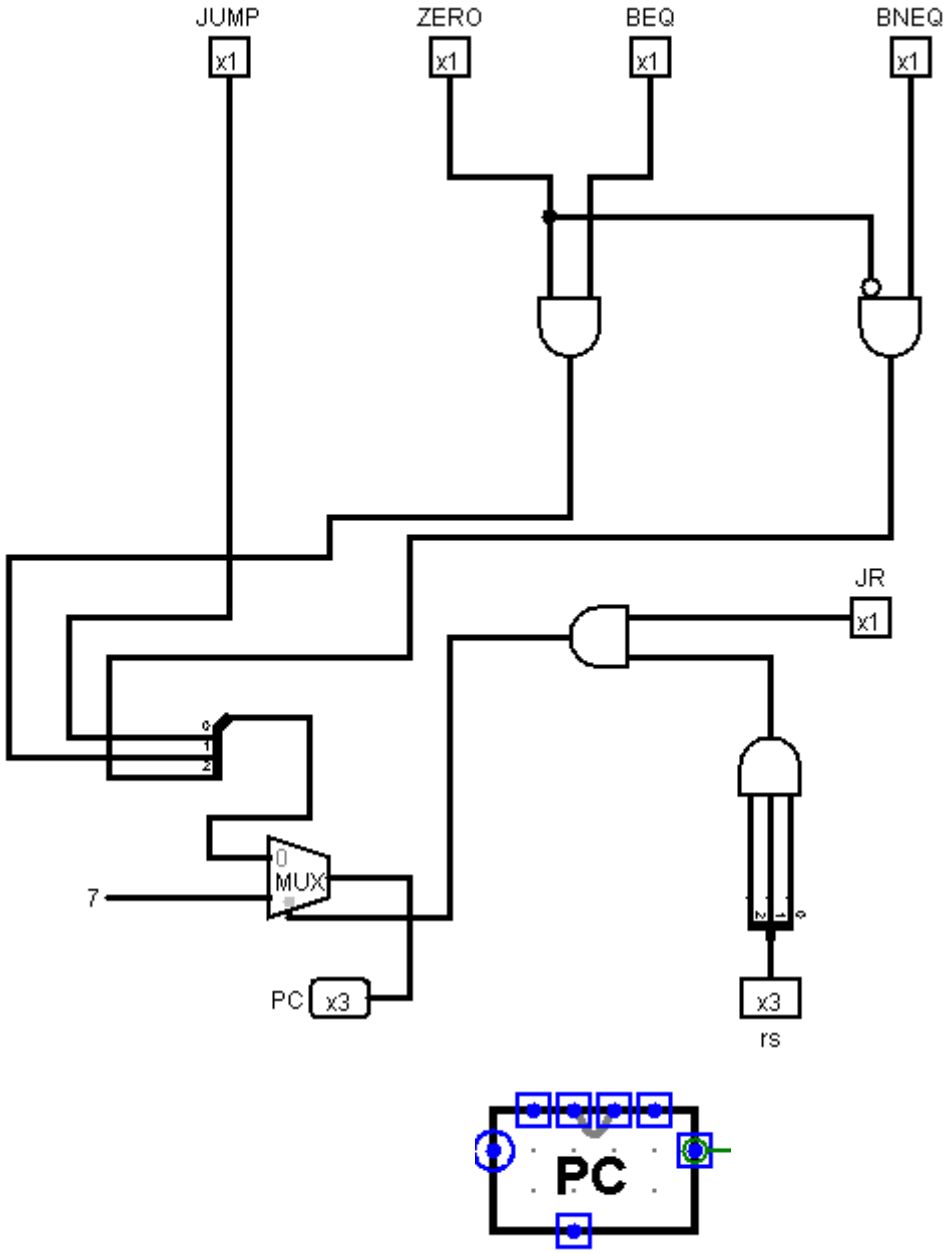
Devrenin daha detaylı iç yapısı ve dış görünümü aşağıya eklenmiştir.





2.6. PROGRAM COUNTER CONTROL

Program Counter Control için öncelikle gereksinimler listelenmiştir. Normal koşullarda teker teker artması gereken program counter değişimi için J-Type komutlara bakıldığında 3 farklı komut daha olduğu ve bunlara ek olarak I-Type komutlardan BEQ ve BNEQ'un olduğu görülmüştür. Normal datapath yapısı ve komutlar da düşünüldüğünde Program Counter'ın çıktı bit uzunluğu 3-bit olarak ayarlanmıştır. Yapı girdi olarak JR, JUMP, ZERO, BEQ, BNEQ durum bayraklarını alıyor ve uygun bayrak durumlarına göre Program Counter çıktısını atıyor. Eğer komut JR komutu ise girdi olarak alınan Rs registerının bilgisi doğrudan Program Countera aktarılıyor. Böylece istenen registera JR komutu ile doğrudan sıçranmış olunuyor. Komut JR komutu değil ise alınan diğer durum kontrol bayraklarının değerlerine göre Program Countera Multiplexer ile seçim yapılarak atama yapılıyor. Böylece Program Counter her komutta uygun olan bir sonraki komutu kendisi içerisinde tutuyor. Program Counter Control devresinin genel yapısı ve dış modül görünümü aşağıya eklenmiştir.



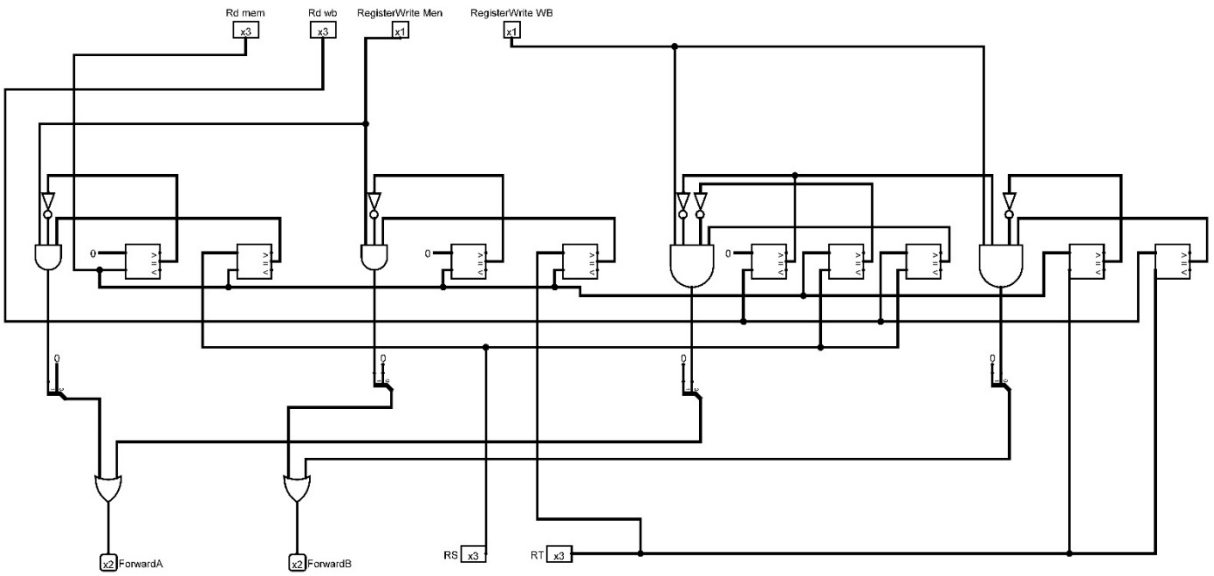
2.7. FORWARD HAZARD

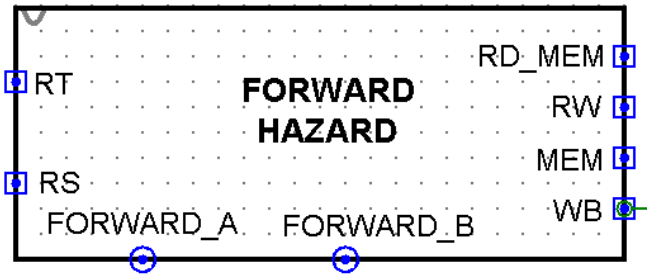
Pipeline mimarisinde çeşitli Çakışma, Hazard durumları ile karşılaşmamız gayet olası bir durum ve büyük bir problem. Kısaca bahsetmek gerekirse:

1. **Structural Hazard:** aynı kaynağı aynı anda kullanmak istenilebilir. Çamaşır yıkama işleminde bahsedildiği üzere tek bir cihazın hem kurutma hem yıkama işlemini yaptığını düşünürsek aynı anda 2 farklı işlem aynı makinayı kullanmak isteyebilir. Bu tarz bir çakışma durumu söz konusu olabilir.
2. **Data Hazards:** örneğin bir işlem bir register'ın sonucunu kullanarak işlem yapmak istiyor olabilir ama bu sonuç daha oluşmamış veya o an farklı bir işlem için bir veri o register'da tutuluyor olabilir bu şekilde istediğimiz sonuçları elde edemeyiz.
3. **Control Hazards:** Dallanma durumlarında ortaya çıkabilecek bir sorun. Örneğin bir dallanma komutu işlenirken bellekte ondan sonra gelen ancak dallanma nedeniyle

yürütülemeyecek olan komutların işleme alınmaması gerekir. Bu tarz komutlar işleme alınırsa programın mantığı akışında bozulmalar meydana gelebilir.

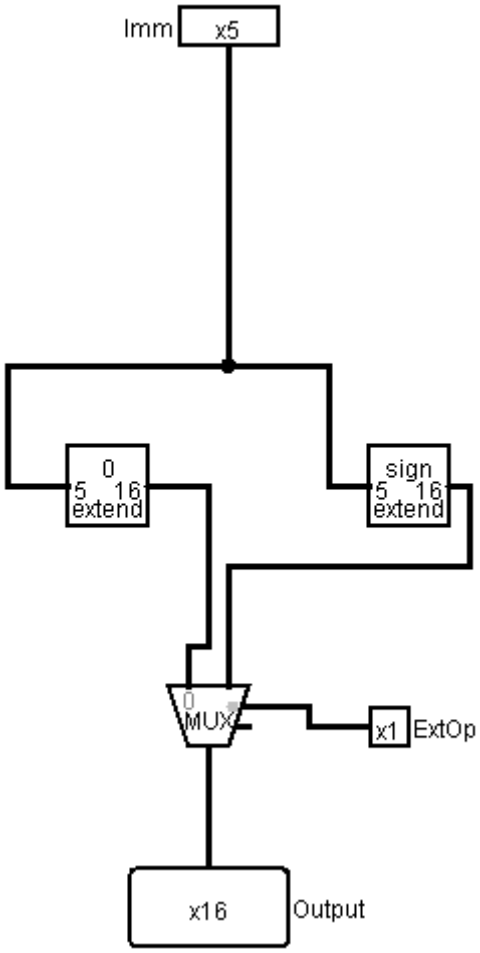
Tüm bu durumların kontrolünün yönetilebilmesi için Forwarding unit kullanılmıştır. Bellek aşamasında alınan RD ile karşılaştırmak için ID buffer'ından alınan RS ve RT kullanıldı. Eğer herhangi bir bağlılık, bağımlılık varsa ALU'ya hangi girdinin gireceğini belirlemek için bu bloğun çıktısı kullanıldı. Datapath üzerinde ALU'ya girdi bilgilerinin seçimlerinin yapıldığı Multiplexer eklenerek Selection bitlerinin bağımlılığı buraya eklendi. Forward Hazard devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.





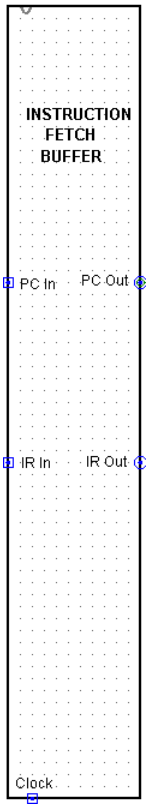
2.8. EXTENDER

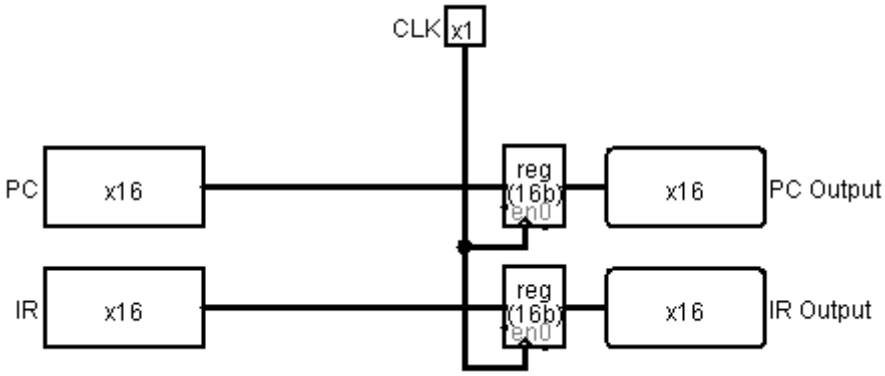
Tüm I-Type komutlara bakıldığında immediate verilerinin 5-bit uzunluğunda olduğu görülmektedir. Bu 5-bitlik verilerin uygun komutlar için ALU'da kullanılabilmesi adına 16-bite çevrilmesi gerekmektedir. Bu çevrimi yaparken verinin signed ya da unsigned olup olmadığının kontrolü de Ext-Op durum kontrol bayrağı ile sağlanmaktadır. Böylece 5-bitlik immediate veri 16-bite çevrilmiştir. Bu çevrimde Bit Extender kullanılmıştır. Extender devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.



2.9. INSTRUCTION FETCH BUFFER

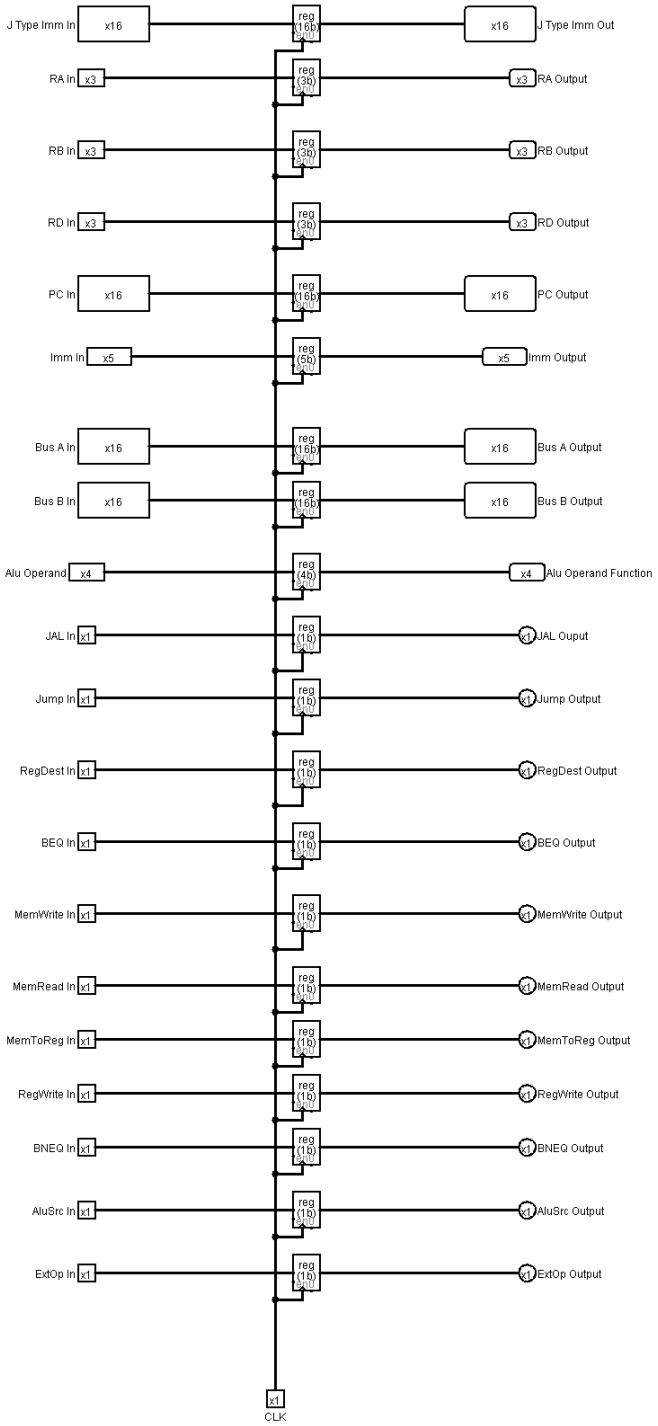
Instruction Fetch Buffer, işlemcinin bir yandan komutları getirirken diğer yandan komutları çalıştırabilmesini sağlar. Bir komut Instruction Fetch Buffer yapısından geçtikten hemen sonra işlemlerine devam ederken bir yandan yeni komut Instruction Fetch Buffer'a gelebilir. Böylece işlemci saat süresi kısaltılmış olur. Instruction Fetch Buffer girdi ve çıktıları değiştirilebilirdir. Yani sabit parametrelere sahip değildir. İşlemci komut yapıları düşünüldüğünde kurduğumuz Instruction Fetch Buffer devresi için sadece Program Counter ve Instruction Memory'den gelen girdilerin kullanılmasının yeterli olacağı anlaşılmıştır. Instruction Fetch Buffer devresinin iç yapısı aşağıda ve modülün dışı görünümü yandaki gibidir.

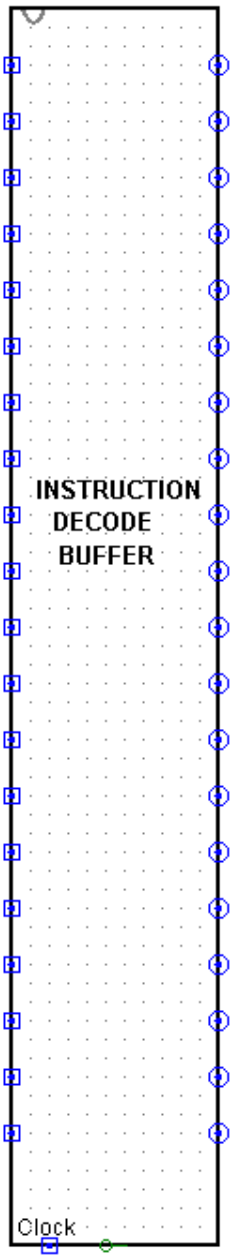




2.10. INSTRUCTION DECODE BUFFER

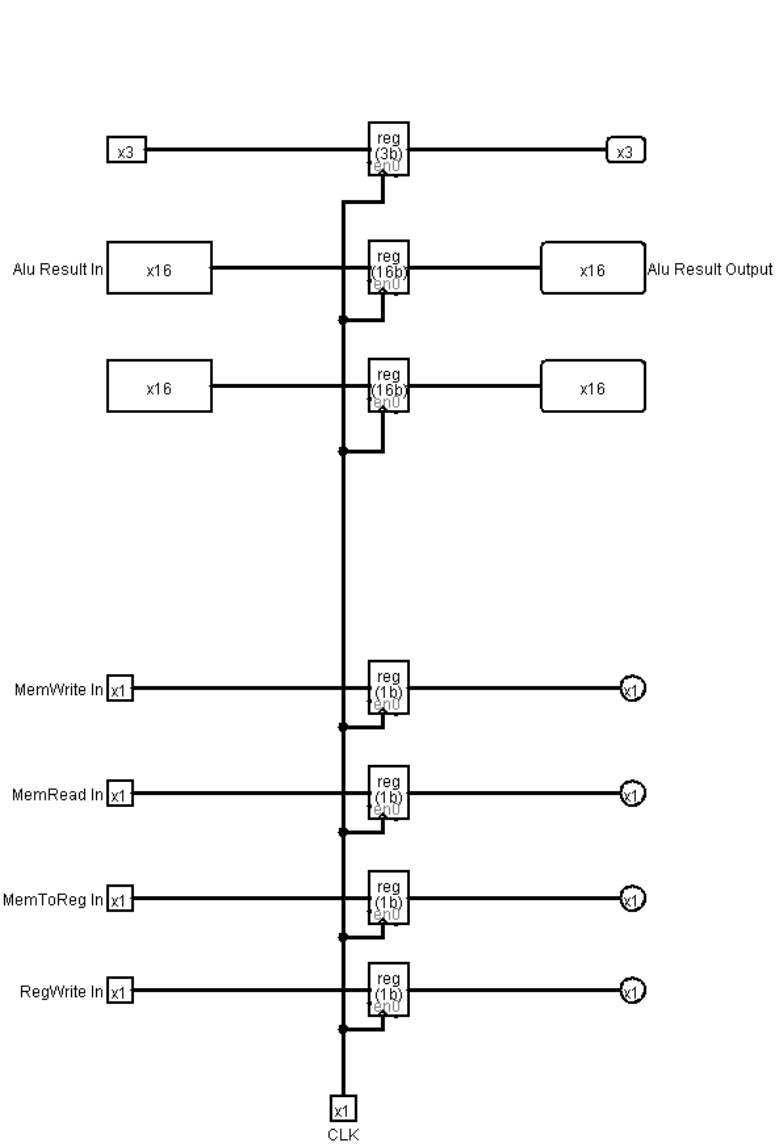
Instruction Decode Buffer, işlemcinin bir yandan komutları çözerken diğer yandan kuyruktaki komutları getirebilmesini sağlar. Instruction Fetch Buffer gibi girdi ve çıktıları özelleştirilebilir. Yani sabit parametrelere sahip değildir. Instruction Decode Buffer devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.

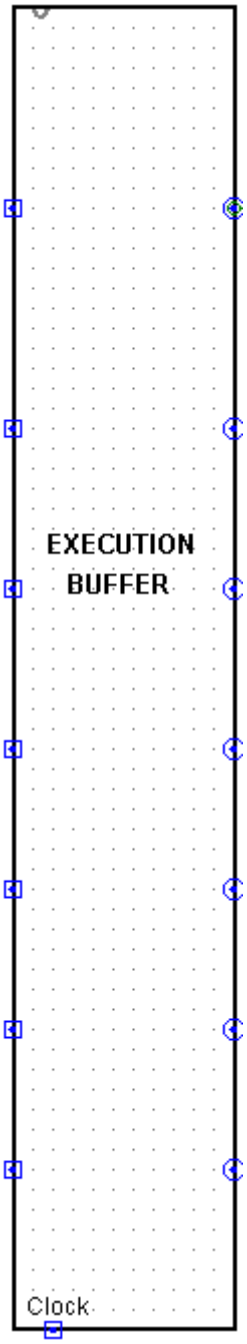




2.11. EXECUTION BUFFER

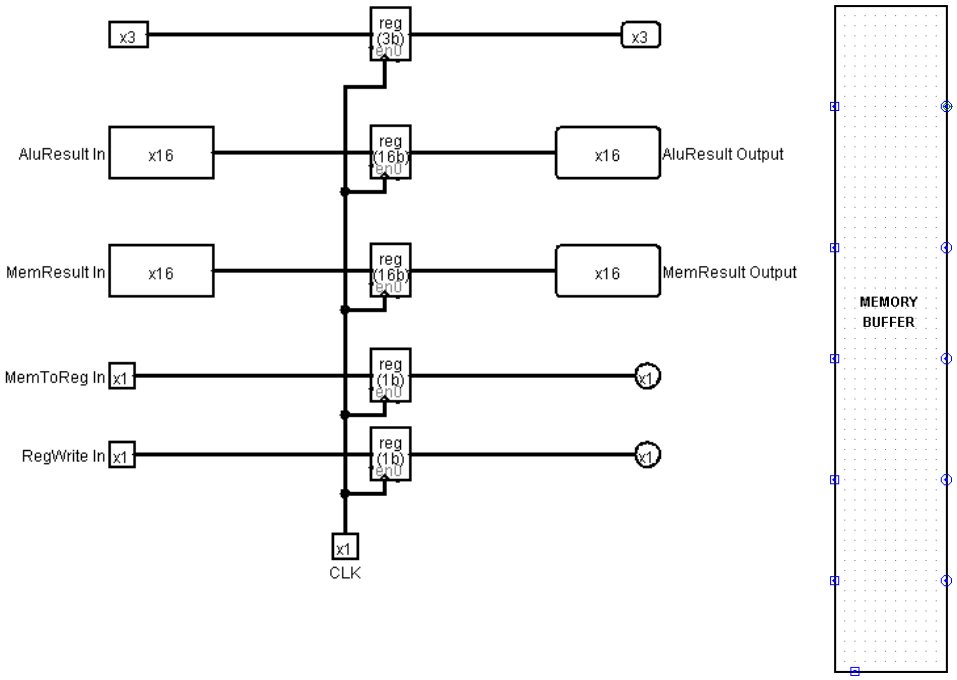
Diğer buffer yapıları gibi Execution Buffer'da da ilk olarak gerekli olabilecek girdilerin listesi oluşturuldu. Komut çalışırken yeni komutların çözülme aşamasına geçerken ihtiyaç duyulan tüm durum ve bilgiler yapıda girdi ve çıktı olarak oluşturuldu. Execution Bufer devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.



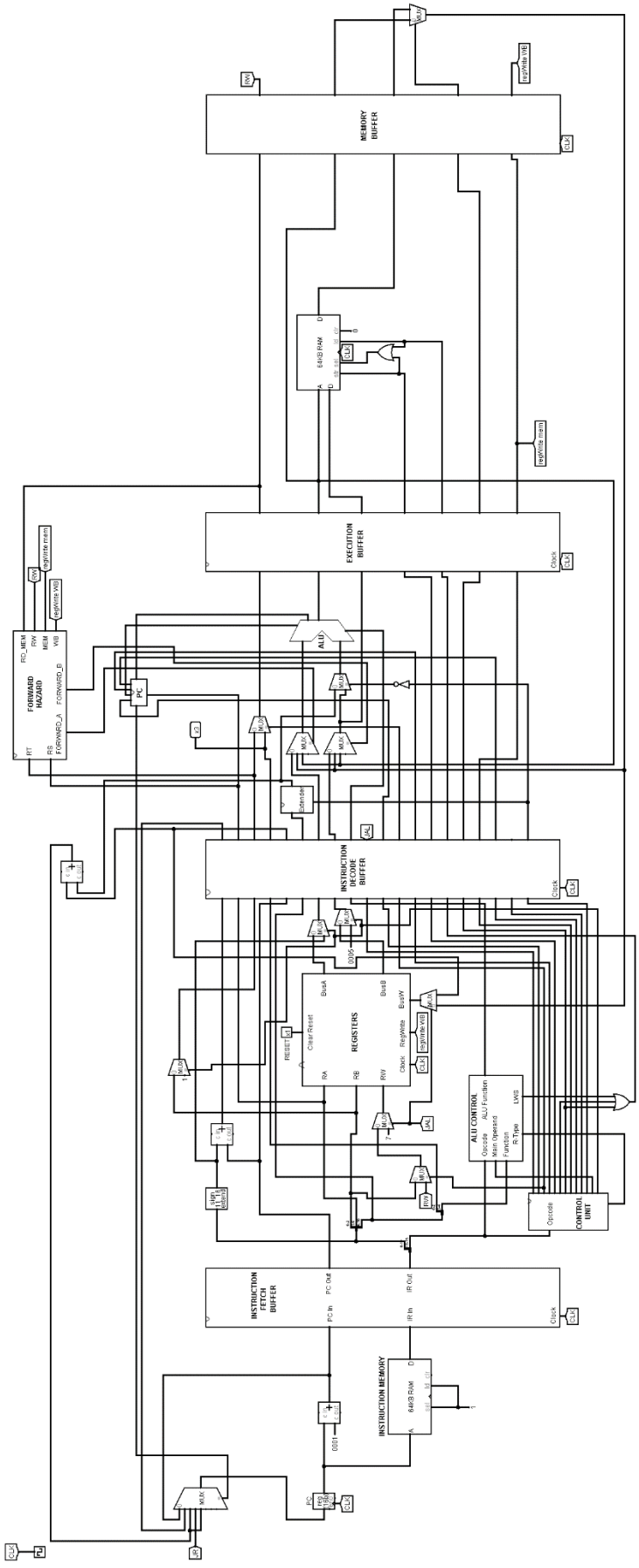


2.12. MEMORY BUFFER

Bu aşamada bir önceki aşama olan Execution Buffer devresinin ALU Result çıkışı veri belleğine adres olarak alınmıştır. Eğer komut aritmetik ise yani R-Type bir komut ise ALU sonucu iletilir. Memory Buffer devresinin daha iyi anlaşılabilmesi için devre iç yapısının görüntüsü ve dış modül görüntüsü aşağıya eklenmiştir.



3. İŞLEMCİ VERİ YOLU (DATAPATH)



4. TEST

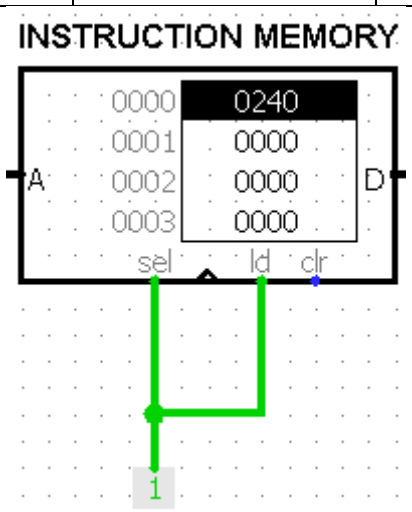
4.1. TEK KOMUTLUK TESTLER

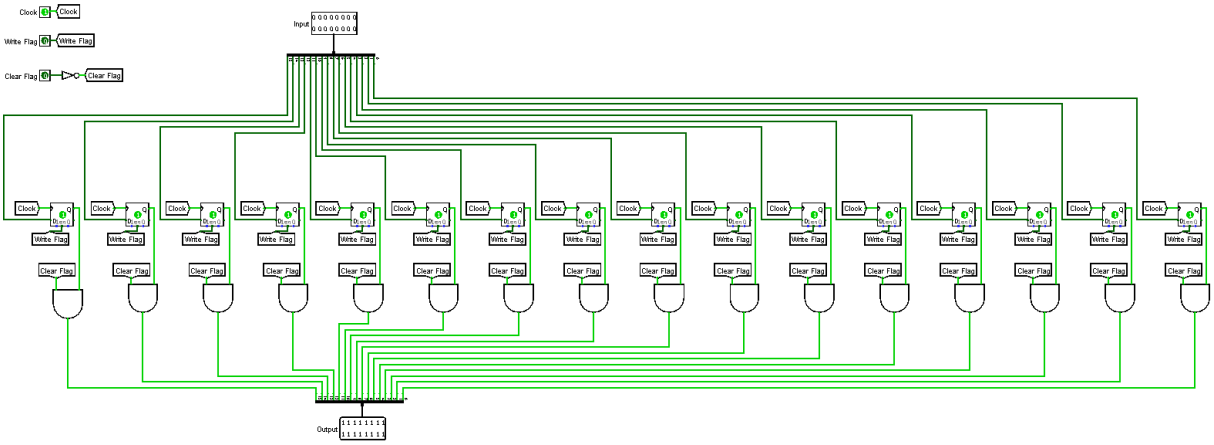
Bu aşamada sadece kurduğumuz veri yolunun doğruluğunu kontrol etmek için tek komutluk testler veya birbirine bağlı iki komutlu testler uygulanmıştır. Bu aşamada Instruction Memory'e doğrudan komutların Hexadecimal karşılıkları Edit Contents seçeneği ile eklenmiş ve Clock çalıştırılmıştır.

4.1.1. R-TYPE KOMUT TESTİ – NOR KOMUTU

NOR komutunun yapısı incelendiğinde şöyle bir komut test edilebilir:

KOMUT	İŞLEM	BINARY DÖNÜŞÜM	HEXADECIMAL DÖNÜŞÜM
NOR R1, R0, R0	R1 = R0 (NOR) R0 R1 = 1111 olmalı.	00000 11 001 000 000	0640

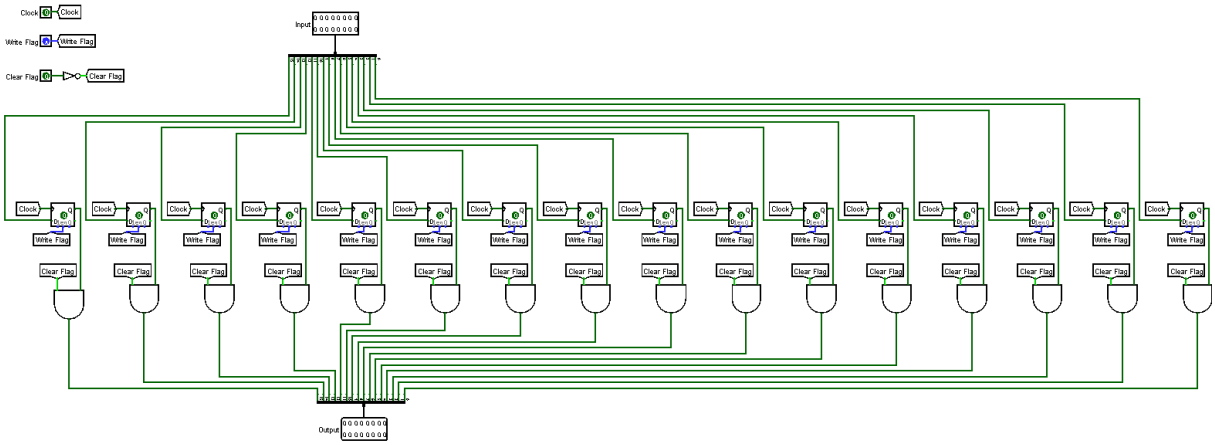




4.1.2. R-TYPE KOMUT TESTİ – SLT

SLT komutunun yapısı incelendiğinde şöyle bir komut test edilebilir:

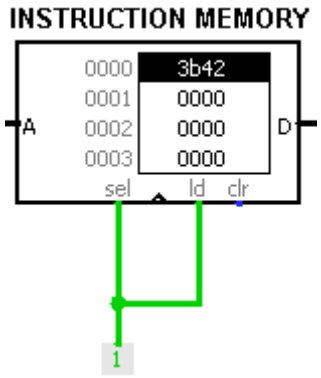
KOMUT	İŞLEM	BINARY DÖNÜŞÜM	HEXADECIMAL DÖNÜŞÜM
NOR R1, R0, R0	R1 = R0 (NOR) R0 R1 = 1111 olmalı.	00000 11 001 000 000	0640
SLT R2, R0, R1	R2 = R1 < R0 R2 = 0000 olmalı	00001 10 010 001 000	0C88

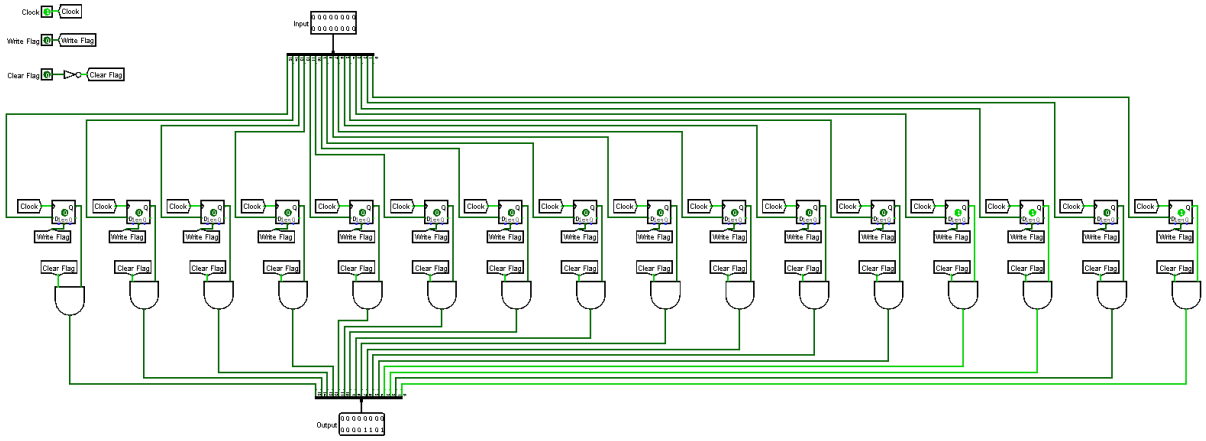


4.1.3. I-TYPE KOMUT TESTİ – ADDI

ADDI komutunun yapısı incelendiğinde şöyle bir komut test edilebilir:

KOMUT	İŞLEM	BINARY DÖNÜŞÜM	HEXADECIMAL DÖNÜŞÜM
ADDI R2, R0, 13	R2 = R0 + 13 R2 = 13 olmalı.	00111 01101 000 010	3B42

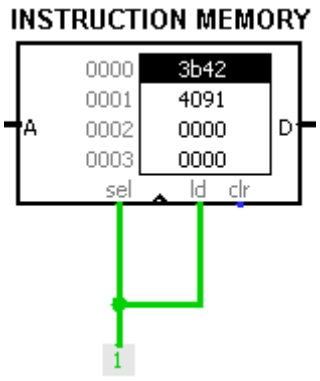


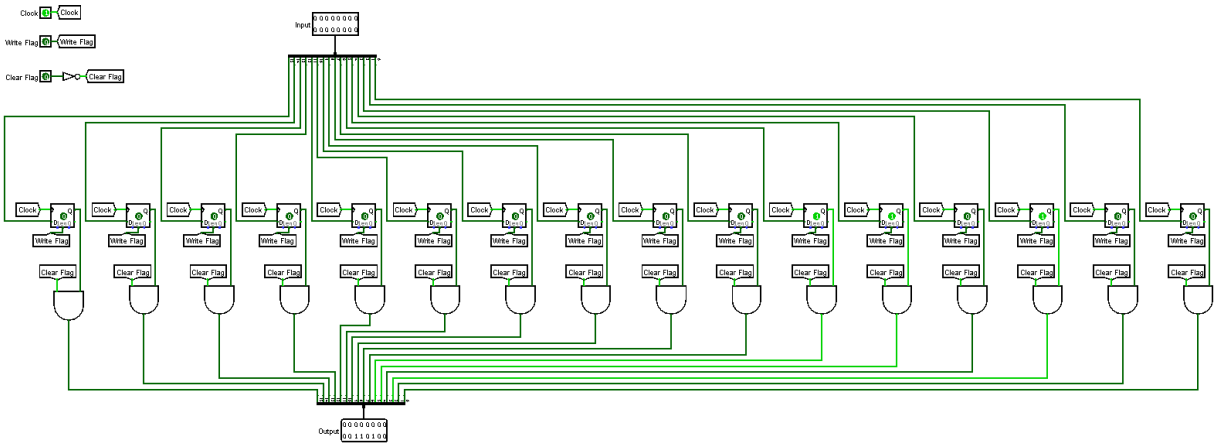


4.1.4. I-TYPE KOMUTLAR – SLL

SLL komutunun yapısı incelendiğinde şöyle bir komut test edilebilir:

KOMUT	İŞLEM	BINARY DÖNÜŞÜM	HEXADECIMAL DÖNÜŞÜM
ADDI R2, R0, 13	R2 = R0 + 13 R2 = 13 olmalı.	00111 01101 000 010	3B42
SLL R1, R2, 2	R1 = R2 << 2 R1 = 26 olmalı.	01000 00010 010 001	4091

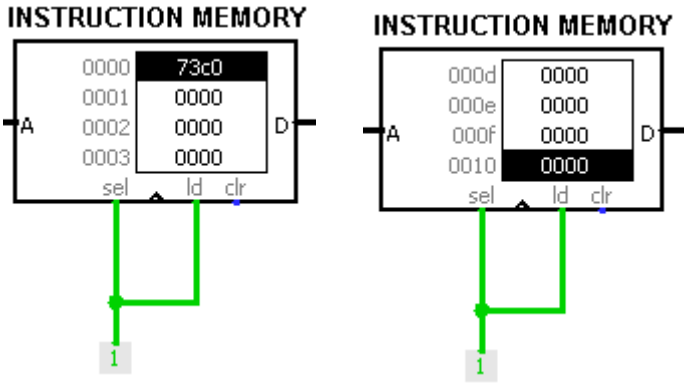




4.1.4. I-TYPE KOMUTLAR – BEQ

BEQ komutunun yapısı incelendiğinde şöyle bir komut test edilebilir:

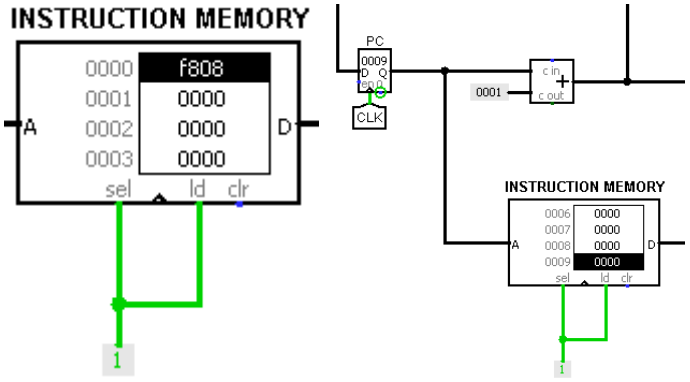
KOMUT	İŞLEM	BINARY DÖNÜŞÜM	HEXADECIMAL DÖNÜŞÜM
BEQ R0, R0, 15	R0 == R0 => 15 I.M. 16'a atlamalı.	01110 01111 000 000	73C0



4.1.5. J-TYPE KOMUT TESTI – J

J komutunun yapısı incelendiğinde şöyle bir komut test edilebilir:

KOMUT	İŞLEM	BINARY DÖNÜŞÜM	HEXADECIMAL DÖNÜŞÜM
J 8	PC = PC + 8 PC = 9 olmalı.	11111 00000001000	F808



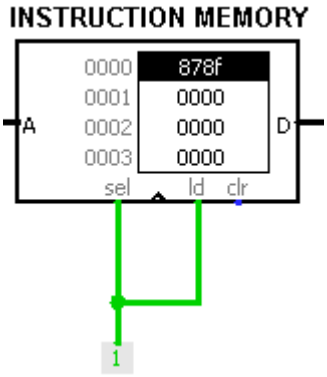
4.2. ÖDEV TESTİ

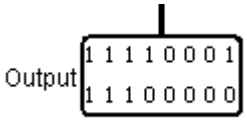
4.2.1. INITIALIZING REGISTERS (TESTING I-TYPE ALU)

Instruction	Hexadecimal	Expected Result
lui 0x78f	**878F**	**r1 = 0xf1e0 (shifted 5 bits left)**
ori r2, r0, 4	**2902**	**r2 = 4 = 0x0004**
addi r3, r0, -2	**3F83**	**r3 = -2 = 0xfffe (sign extension)**
xori r4, r0, -2	**3784**	**r4 = 0x001e (zero extension)**

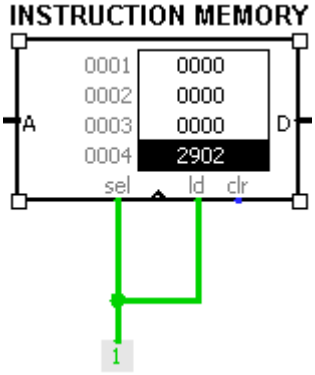
Öncelikle komutlar hexadecimal tabanda yazılmıştır. Ardından Instruction Memory'e yüklenmiş ve çalıştırılmıştır.

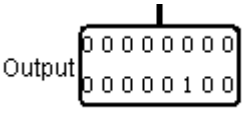
- **lui 0x78f**



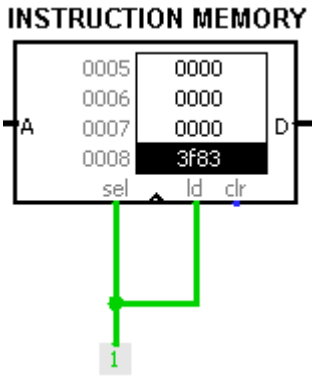


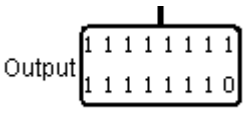
- ori r2, r0, 4



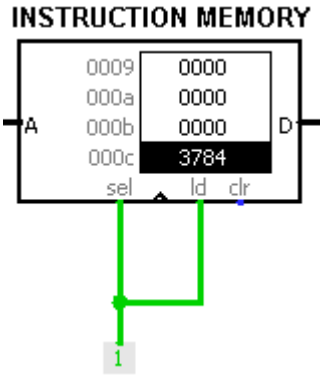


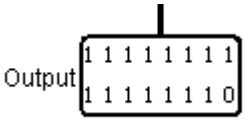
- addi r3, r0, -2





- xori r4, r0, -2





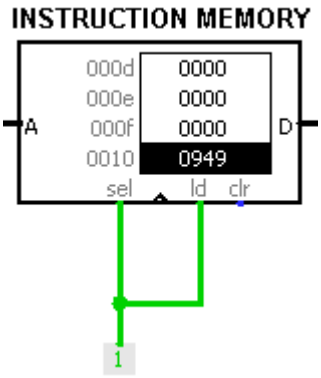
Burada Output verisinin Expected Result değerinden farklı olduğu görülmektedir. Yapımızdaki tüm işlemlerde sign extension yapıldığı için bu durumu düzeltmemiz pek mümkün değildi. Sonucun sign extension olarak istendiğini varsayarsak doğru çıktı verdiğini görebiliriz.

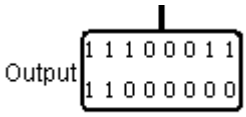
4.2.2. TESTING R-TYPE ALU INSTRUCTIONS (NO RAW HAZARDS – NO FORWARDING)

BİLGİLENDİRME: Burada ödevle alakalı verilen yapıyla bir uyuşmazlık mevcuttur. Normalde shifting operatörler ödev dosyasına baktığımızda I-Type komut olarak sayılmış ve kaydırma değerini immediate değer olarak almış. Fakat aşağıdaki tabloda shifting değer register olarak atanmış. Yapıyla tamamen uyuşmayacağından bu aşamada shifting değerlerin hepsini 2 olarak atayıp işlemlerimize devam ettik. Değer değişikliğine gittiğimiz için o aşaman sonraki tüm Output değerleri de Exptected Result değerinden farklı gelmiş oldu.

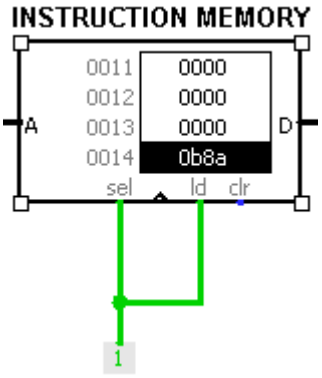
Instruction	Hexadecimal	Expected Result
add r5, r1, r1	0949	r5 = 0xe3c0 (carry is ignored)
sub r6, r1, r2	0B8A	r6 = 0xf1dc
slt r7, r1, r2	0DCA	r7 = 1 (true) r1 < 0
sltu r5, r1, r2	0F4A	r5 = 0 (false)
and r6, r3, r4	019C	r6 = 0x001e
or r7, r1, r2	03CA	r7 = 0xf1e4
xor r5, r1, r3	054B	r5 = 0x0e1e
nor r6, r1, r2	078A	r6 = 0x0e1b
sll r7, r4, r2	????	r7 = 0x01e0
srl r5, r1, r2	????	r5 = 0x0f1e
sra r6, r1, r2	????	r6 = 0xff1e
ror r7, r3, r2	????	r7 = 0xefff

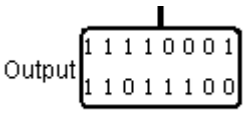
- add r5, r1, r1



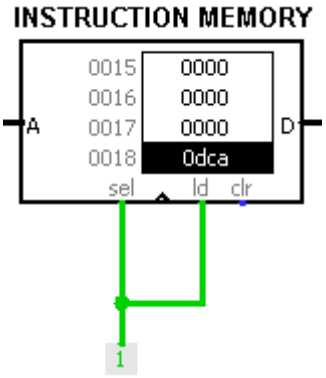


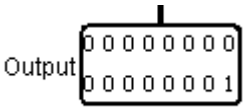
- sub r6, r1, r2



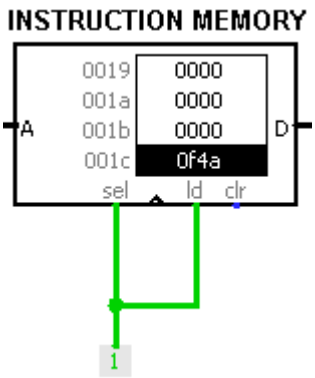


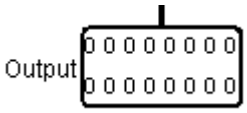
- **`slt r7, r1, r2`**



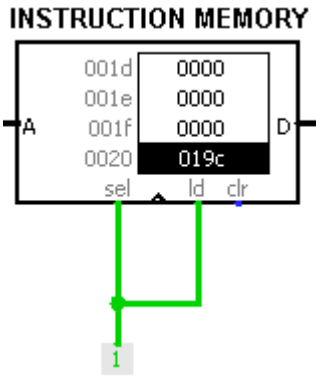


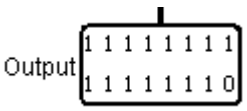
- sltu r5, r1, r2



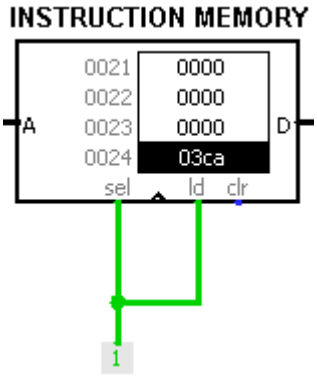


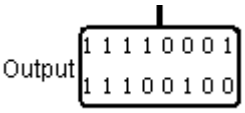
- and r6, r3, r4



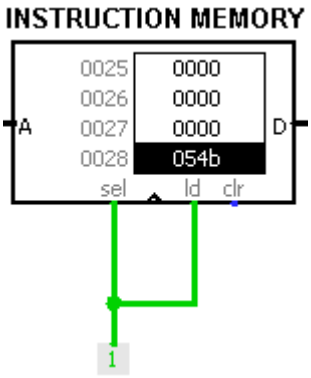


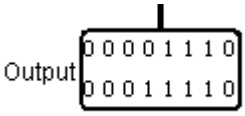
- or r7, r1, r2



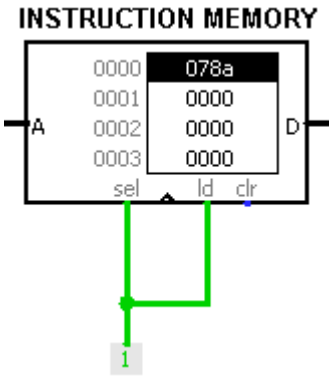


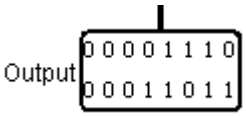
- xor r5, r1, r3



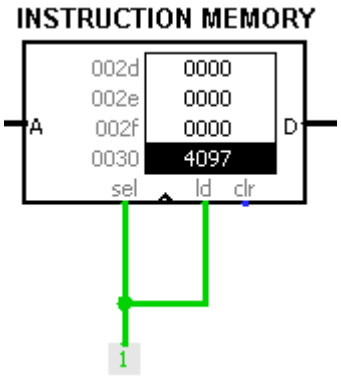


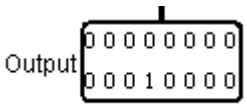
- nor r6, r1, r2



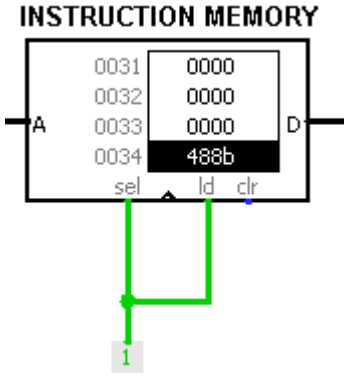


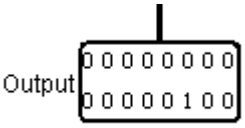
- sll r7, r4, 2 -> 4097



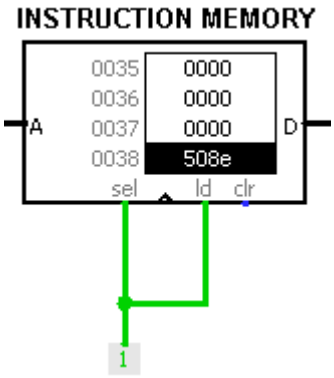


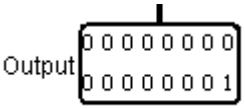
- srl r5, r1, 2 -> 488B



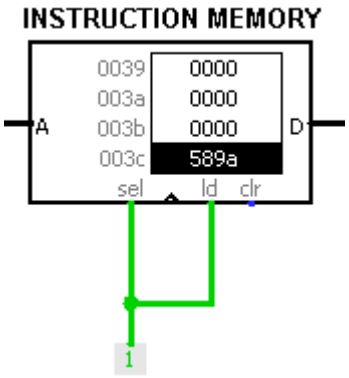


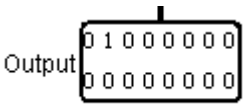
- sra r6, r1, 2 -> 508E





- ror r7, r3, 2 -> 589A



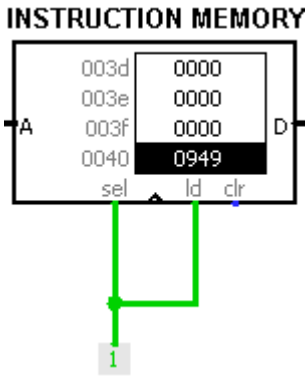


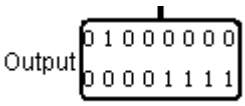
4.2.3. TESTING RAW HAZARDS AND FORWARDING

Değerlerin farklı gelmesi bir önceki aşamadan shifting operatörlerinin değerlerini bizim elle değiştirmemizden kaynaklanıyor. Kontrol edildiğinde doğru çıktılar geldiği görülmüştür.

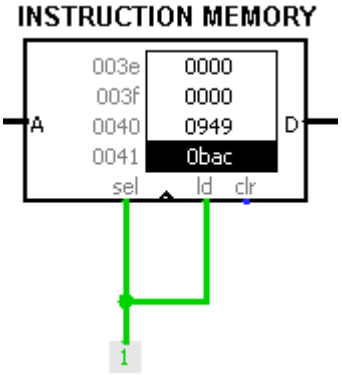
Instruction	Hexadecimal	Expected Result
add r5, r1, r1	**0949**	**r5 = 0xe3c0**
sub r6, r5, r4	**0BAC**	**r6 = 0xe3a2 (depends on add)**
and r7, r5, r6	**01EE**	**r7 = 0xe380 (depends on add/sub)**
ori r5, r5, 0xf	**2BEE**	**r5 = 0xe3cf (depends on add)**

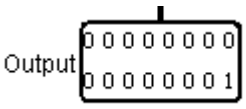
- Add r5, r1, r1



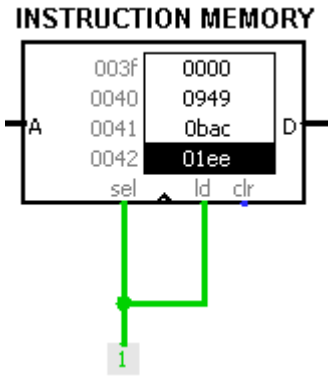


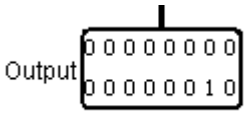
- sub r6, r5, r4



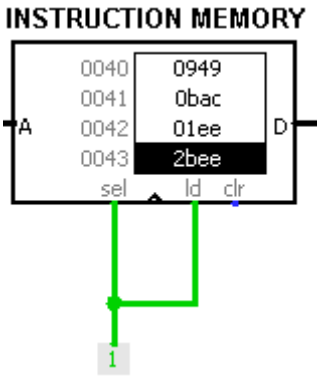


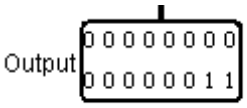
- and r7, r5, r6





- ori r5, r5, 0xf

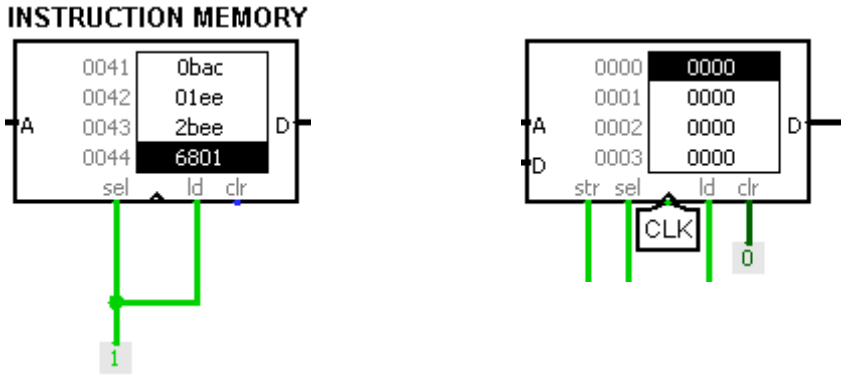




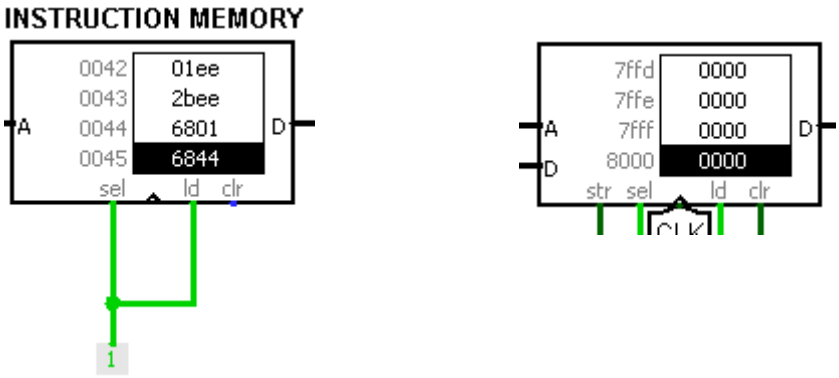
4.2.4. TESTING SW AND LW

Instruction	Hexadecimal	Expected Result
sw r1, 0(r0)	**6801**	**MEM[0] = 0xf1e0**
sw r4, 1(r0)	**6844**	**MEM[1] = 0x001e**
lw r5, 0(r0)	**6005**	**r5 = MEM[0] = 0xf1e0**

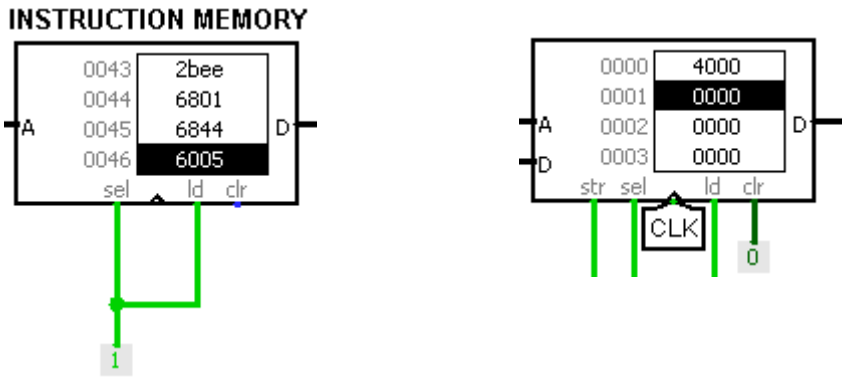
- sw r1, 0(r0)



- sw r4, 1(r0)



- lw r5, 0(r0)

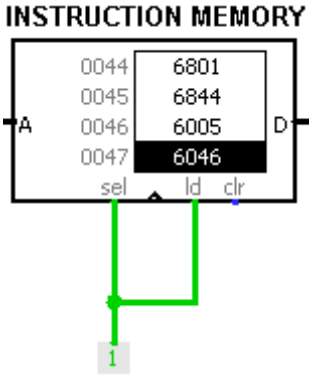


Tüm işlemler sonrasında Memory belleğinin içi aşağıdaki gibi doğru değişmiştir.

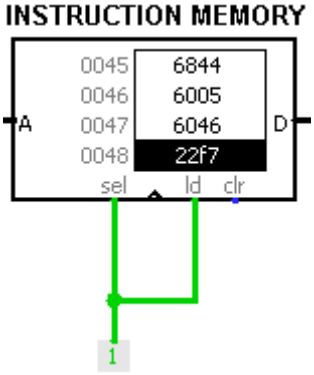
4.2.5. TESTING LOAD DELAY, STALLING PIPELING AND FORWARDING

Instruction	Hexadecimal	Expected Result
lw r6, 1(r0)	6046	r6 = MEM[1] = 0x001e
andi r7, r6, 0xb	22F7	r7 = 0x000a (stall 1 cycle & forward)
sw r7, 2(r0)	6887	MEM[2] = 0x000a (forwarded from andi)
lw r5, 0(r0)	6005	r5 = MEM[0] = 0xf1e0
sw r5, 3(r0)	68C5	MEM[3] = 0xf1e0 (forwarded from lw)

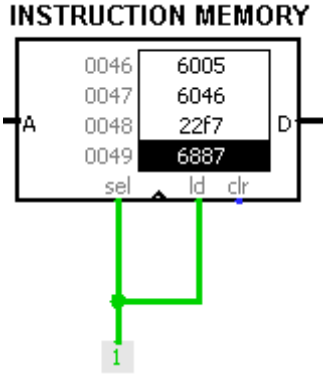
- lw r6, 1(r0)



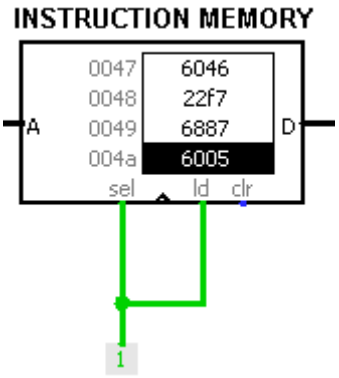
- andi r7, r6, 0xb



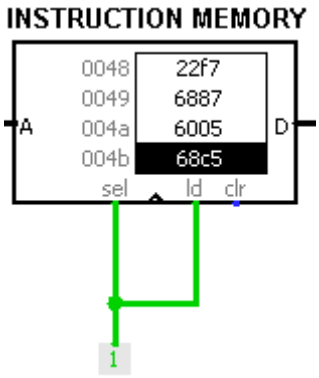
- sw r7, 2(r0)



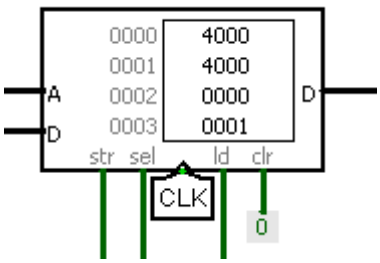
- lw r5, 0(r0)



- lw r5, 3(r0)



Tüm işlemler sonrası Memory belleğinin görünümü aşağıdaki gibi değişmiştir. Farklılıklar bahsedildiği gibi yapının komutlarla uyuşmamasından kaynaklanmıştır. Tüm testler ödev klasörü içerisinde algorithms.txt dosyası kullanılarak tekrarlanabilir.



5. GENEL DEĞERLENDİRME

Proje bizler için pipeline işlemci veri yolu yapısını anlamamızda çok faydalı oldu. Tek çevrimli işlemci, çok çevrimli işlemci ve pipeline işlemciler arasındaki farkları daha iyi kavrayabildik. İşlemcilerde zaman kavramının pipeline yapısıyla birlikte daha da ele alınan önemli bir konu olduğunu gördük.

Pipeline işlemci tasarımında buffer yaklaşımlarıyla genel zamandan kazanç elde edilip işlemlerin ardı ardına beklemeden yapılabildiğini ve bu sayede daha anlık yanıtlar alınabildiğini deneyimledik.

- **Logisim Hakkında**

Logisim'deki devre oluşturup oluşturulan devreyi başka devreler üzerinde de kullanabilmek bizim için çok büyük bir avantaj sağladı. Tüm yapıyı tek seferde tek bir devre üzerinde kurmayı başaramayabilirdik. Bunun yerine Logisim ile kendi küçük devrelerimizi oluşturup onların dış görünümlerini de özelleştirdik. Böylece gerçek pipelined işlemci yapısına da benzer bir yapı ortaya çıkmış oldu. Hali hazırda kendi içerisinde yer alan Multiplexer, Bit Extender, Priority Encoder gibi devreler de işimizi çokça kolaylaştırdı. Simülasyon aşamalarında aktif kabloların ayırt edilebilirliğiyle test aşamalarımızda hiç zorlanmadık.

Logisim'de denenen simülasyonun baştan başlatılamaması bizim için çok büyük bir dezavantaj oluşturdu. Her seferinde test ettiğimiz komut için Simulastion Reset ile tüm simülasyonu sıfırlayıp baştan Instruction Memory düzenlemesi yapmamız gerekiyordu. Bu da aynı işlemleri birçok kez tekrarlama oluyor.