CSE 331 Computer Organization

Project 2 – ALU with Structural Verilog

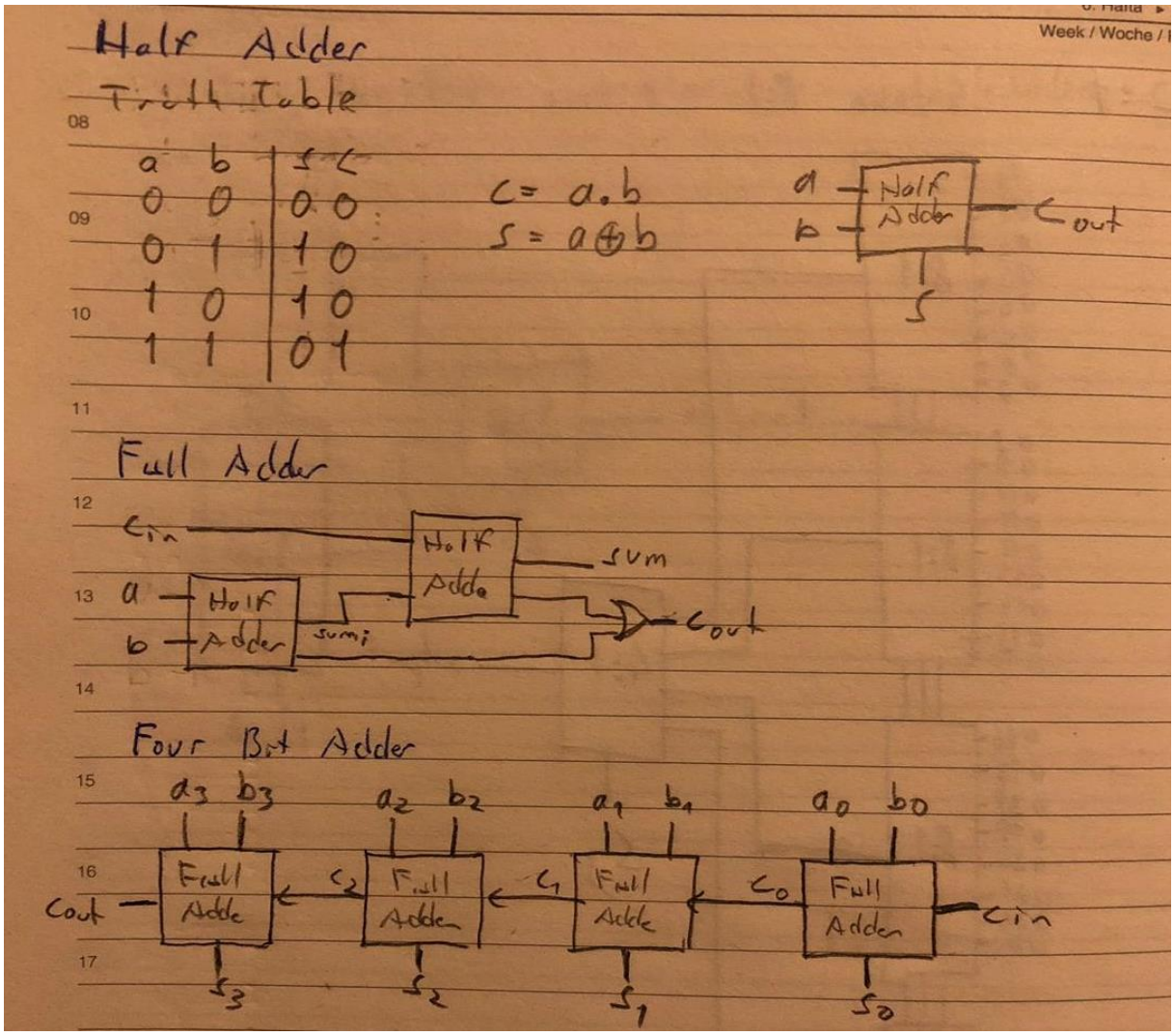
Batuhan TOPALOĞLU 151044026

GTÜ Bilgisayar Mühendisliği Bölümü 2018

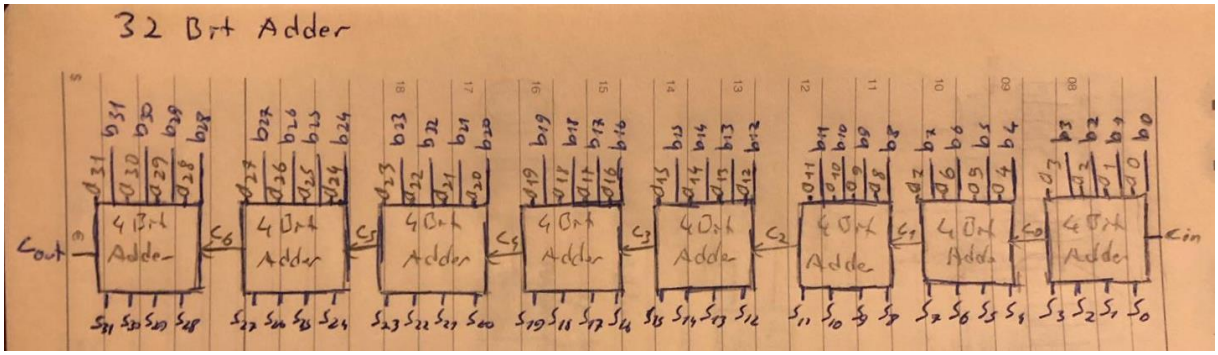
1. Şematik Tasarımlar:

1.1 Toplayıcılar

Lojik kapılarla half adder'ı gerçekledilten sonra half adder kullanarak 1-bit full adder tasarlandı, ardından bu 1-bit lik full adder'lar kullanılarak 4-bit adder tasarlandı. Bu sayede gereksiz tasarım tekrarı yükünden kaçınılmış oldu.

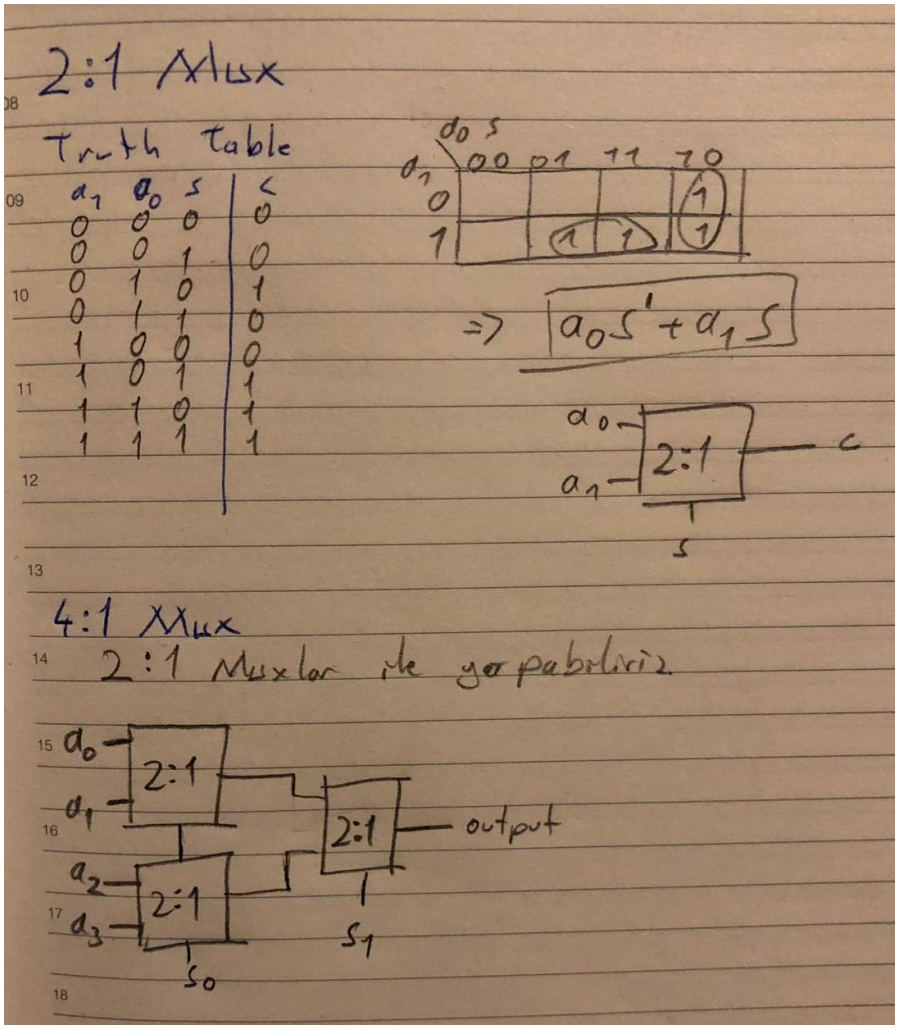


Bu 4-bit'lik adder'lar kullanılarak 32-bit'lik bir adder tasarlandı. Bunun için 8 adet 4-bit adder kullanıldı. Çıkarma işlemi içinde adder kullanıldı.

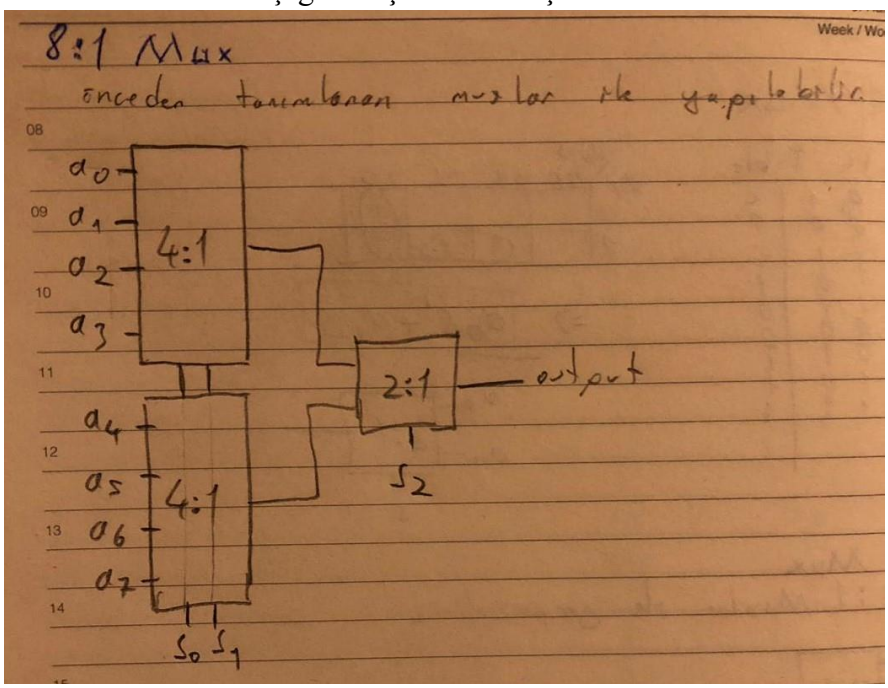


1.2 Seçiciler

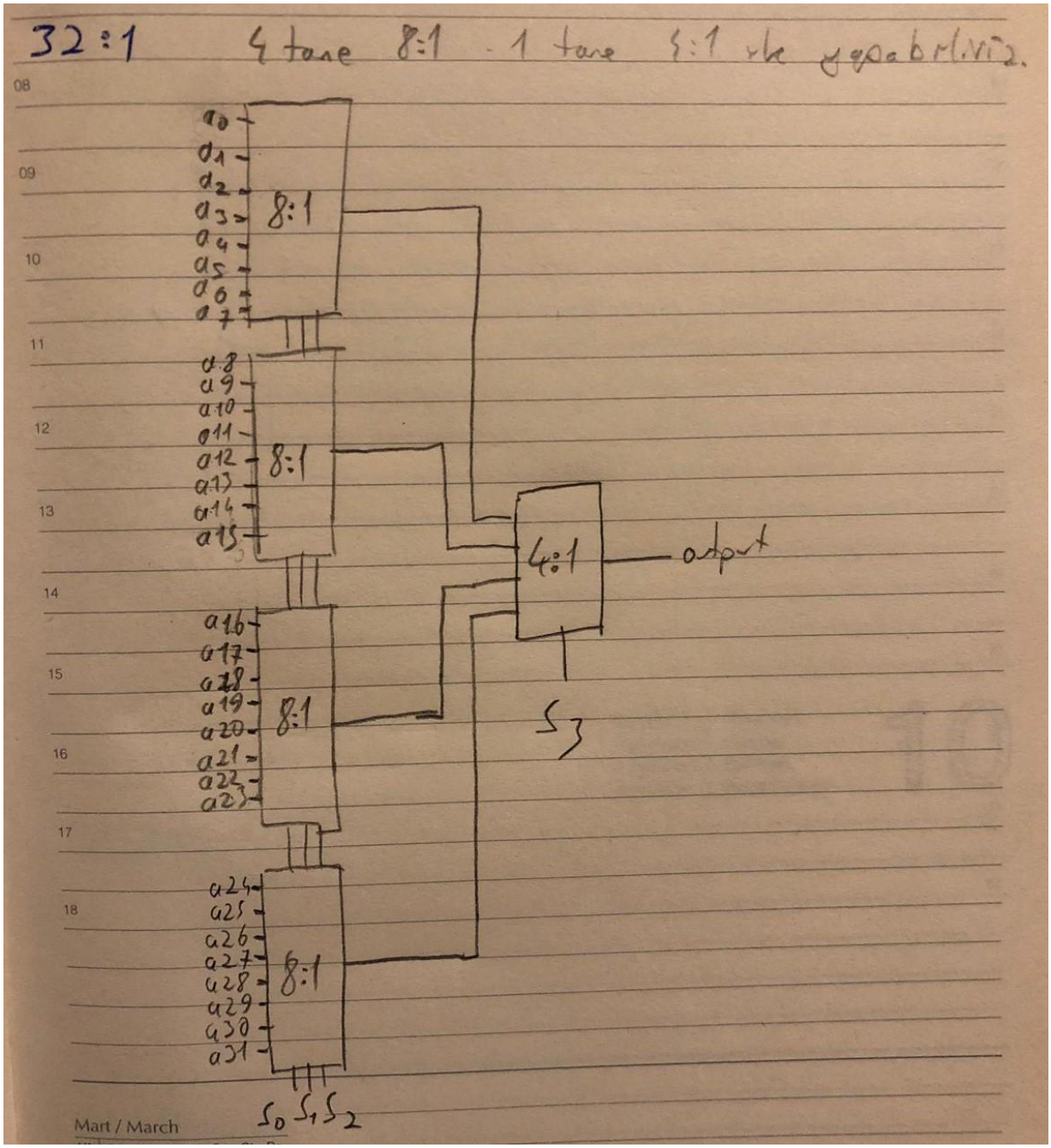
Doğruluk tablosu kullanarak 2:1 mux ‘u lojik kapılar ile tasarlandı. Ardından bu 2:1 mux’lar ile aşağıdaki görselde görüldü gibi 4:1 mux’u tasarlandı.



4:1 ve 2:1 mux’ları aşağıdaki şekilde birleştirerek 8:1 mux tasarlandı.

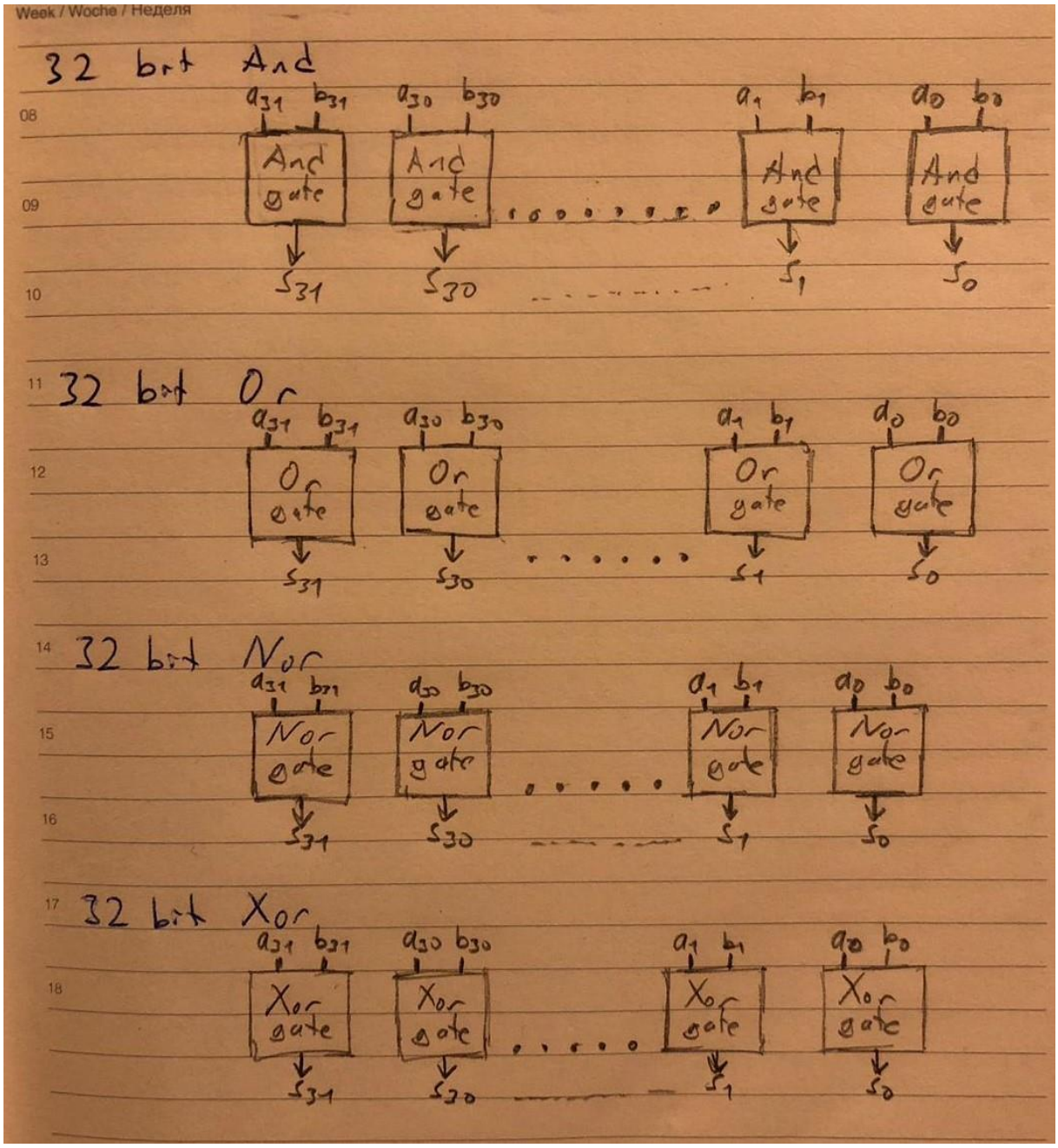


8:1 ve 4:1 mux'lar kullanılarak aşağıdaki şekilde bir 32:1 mux tasarlandı. 8:1 mux'ların seçici bacaklarına aynı s0,s1,s2 değerleri verileceği bu şekilde gösterildi.



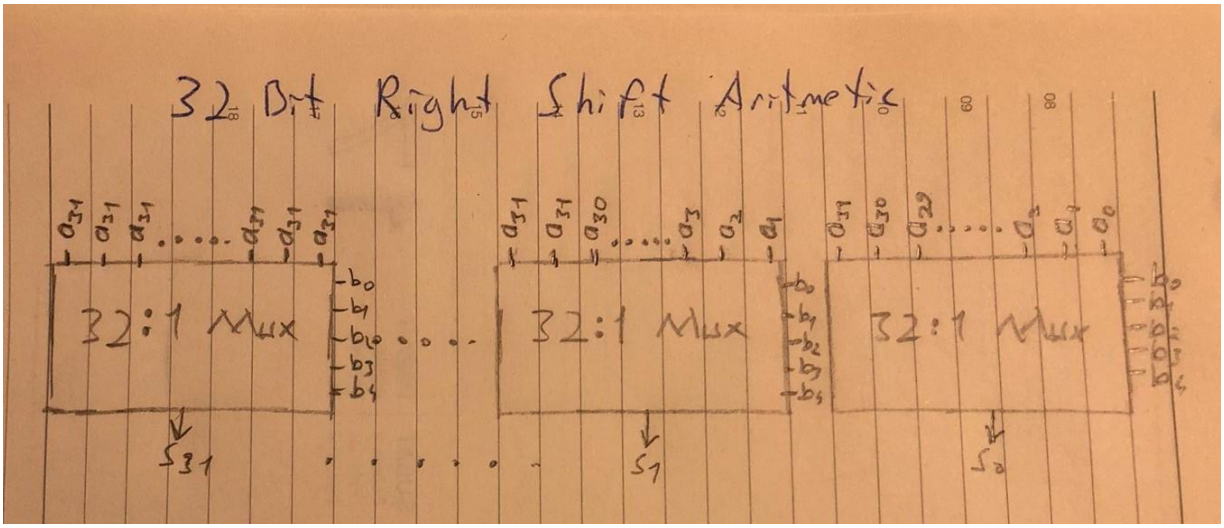
1.3 32-bit lojik kapılar

Burada tasarlanan bütün modüllerin mantığı aynı, 2 adet 32-bit'lik sayının bitlerini tek tek 1-bitlik lojik kapıları kullanarak işleme tabi tutup sonucu çıktı değişkeninin uygun indeksine yazmak.

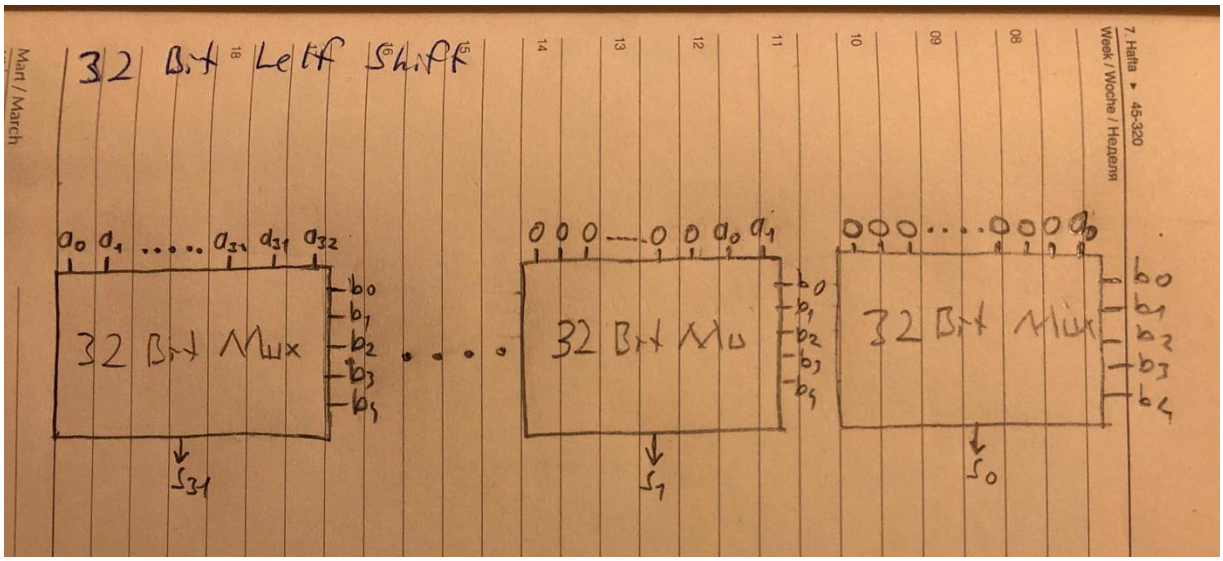


1.4 Shift operatörleri

Aritmetik sağa shift modülü parametre olarak aldığı 32-bit'lik 'a' binary sayısını diğer bir parametresi olan 'b' değeri kadar sağa shift eder. Bunu yaparken b'nin 0'dan farklı değeri için a değerinin shift edilmiş halinde boş kalan en değerli bitlerinin olduğu kısıma shift etme adeti kadar 'a' nın en değerli bitinin (a31) değerini koyar. Eğer b'nin değeri 31'den büyükse shift işleminin sonucundaki değerin bütün bitleri '0' olucak şekilde ayarlandı.



32-Bit lojik left shift operatörü 32-bit'lik 'a' değerini 32-bit'lik 'b' değeri kadar sola shift eden modüldür. 'b' nin 0 dan farklı her değeri için 'a' nın en değerliksiz bitinden başlamak üzere boş kalan bitlere '0' değeri atanır. Eğer b'nin değeri 31'den büyükse shift işleminin sonucundaki değerin bütün bitleri '0' değeri olucak şekilde ayarlandı.



2.Modüller:

2.1 module half_adder(sum, carry_out, a, b);

Doğruluk tablosu ve karnaugh haritasını çıkardıktan sonra xor ve and kapıları kullanarak gerçekledik. Sum toplamın sonucu, carry_out elde çıkışı, a ve b toplanacak 1bitlik sayılar.

2.2 module full_adder(sum, carry_out, a, b, carry_in);

Lojik kapılar ile yaptığımız half-adder ve bir adet or lojik kapısı kullanarak gerçekledik. Sum toplamın sonucu, carry_out elde çıkışı, carry_in elde girişi, a ve b toplanacak 1bitlik sayılar.

2.3 module _4bit_adder (S,C,A,B,C0);

Tasarladığımız full adder 1-bit iki değeri toplayabildiği için 4 adet full-adder kullanarak 4-bit lik bir adder elde ettik. S toplamın sonucu, C elde çıkışı, C0 elde girişi, A ve B toplanacak 4bitlik sayılar.

2.4 module _32bit_Adder(sum,carry_out,a,b,carry_in);

Elimizdeki 4-bir'lik adder'lardan 8 tanesi ardışıl olarak kullanarak 32 bitlik bir adder elde ettik. Sum toplamın sonucu, carry_out elde çıkışı, carry_in elde girişi, a ve b toplanacak 32 bitlik değerler.

2.5 module _32bit_Sub(sum,carry_out,a,b,carry_in);

32-bit'lik adder'ın elde bit girişine 1 gönderip A-B senaryosu için B'nin bütün bitlerinin not'ını alarak A+ (B')+1 üzerinden çıkarma işlemi yapabiliyoruz. Sum 32-bit çıkarmanın sonucu, carry_out elde çıkışı, carry_in elde girişi, a ve b çıkarılacak 32-bitlik değerler.

2.6 module _2_1_mux(out,a,s);

Öncelikle doğruluk tablosu ve karnaugh haritasını çıkarak lojik kapılar kullanarak nasıl gerçekleyebileceğimiz bulduk ve daha sonra bu "a.0s' + a1.s" ifadesini and, or ve not kapıları gerçekledik. 'out' seçim sonucu 1-bit, a 2-bitlik üzerinden seçim yapılacak değer ve s ise 1-bitlik seçim girişi.

2.7 module _4_1_mux(out,a,s);

Bir alt seviyede kullanabileceğimiz bir 2:1 mux'umuz olduğu için lojik kapılar düzeyine inmeden 3 adet 2:1 mux ile 4:1 mux'u tasarladık. Tasarım detayına 1.2 başlığından erişilebilir. 'out' seçim sonucu 1bit, a 4-bitlik üzerinden seçim yapılacak değer ve s ise 2-bitlik seçim girişi.

2.8 module _8_1_mux(out,a,s);

Elimizdeki 4:1 ve 2:1 mux'larda 2 adet 4:1 ve 1 adet 2:1 olandan kullanarak 8-bitlik bir veriyi seçebiliriz. Bunun tasarım 1.2 de mevcut. Modül parametreleri ise ; 'out' seçimin sonucu 1-bitlik değer, a 8-bitlik üzerinden seçim yapılacak değer ve s ise 3-bitlik seçim girişi.

2.9 module _8_1_V2_mux(out,a0,a1,a2,a3,a4,a5,a6,a7,s);

Bu modülün diğer 8:1 muxtan tek farklı 8 adet'lik seçim eleamanları bir dizi olarak değil ayrı ayrı alıyor olması , bu sayede farklı bit dizlerinin elemanları aynı mux'a gönderebiliyoruz. 'alu32' modülü içerisinde kullanımı görebilirsiniz . 'out' seçimin sonucu 1-bitlik değer,a'lar aralarından seçim yapılacak bitler ve s ise 3-bitlik seçim girişi.

2.10 module_32_bit_mux(out,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16,a17,a18,

a19, a20, a21,a22,a23,a24,a25,a26,a27,a28,a29,a30,a31,s);

Bu modülde _8_1_V2_mux ile aynı sebepten ötürü seçilecek değerleri bit bir alıyor. Arka planda 4 adet 8:1 ve 1 adet 4:1 mux üzerinden çalışıyor. Sistem tasarımı 1.2 başlığından görülebilir. 'out' seçimin sonuc biti a değerleri seçilecek değerler ve s ise 5-bitlik seçim girişi.

2.11 module _32_bit_and(out,a,b);

İnput olarak alınan a ve b 32-bitlik değerlerinin aynı indeksteki bitlerinin tek tek and kapısına sokar ve sonuçları bit bit out değerinin aynı indeksine değer olarak koyar. => and a1(out[0],a[0],b[0])

2.12 module _32_bit_or(out,a,b);

İnput olarak alınan a ve b 32-bitlik değerlerinin aynı indeksteki bitlerinin tek tek 'or' kapısına sokar ve sonuçları bit bit out değerinin aynı indeksine değer olarak koyar. => or a1(out[0],a[0],b[0])

2.13 module _32_bit_nor(out,a,b);

İnput olarak alınan a ve b 32-bitlik değerlerinin aynı indeksteki bitlerinin tek tek nor kapısına sokar ve sonuçları bit bit out değerinin aynı indeksine değer olarak koyar. => nor a1(out[0],a[0],b[0])

2.14 module _32_bit_xor(out,a,b);

İnput olarak alınan a ve b 32-bitlik değerlerinin aynı indeksteki bitlerinin tek tek xor kapısına sokar ve sonuçları bit bit out değerinin aynı indeksine değer olarak koyar. => xor a1(out[0],a[0],b[0])

2.15 module _32_bit_shift_left(out,a,b);

İnput olarak alınan a değerini diğer bir input olan b nin değeri kadar sola shift eden modül. 'out' a'nın b kadar shift edilmiş hali. Shift etme işlemini 32:1 mux lardan 32 adet kullanarak yapıyorum. 'b' nin değerinin 31 den büyük olduğu her durumda 'out' değerinin bütün bitleri 0 olucaktır. Çünkü değerler 32 bitlik bloklar halinde.Örneğin 'out[0]' değerini seçecek mux sadece b'nin 0 a eşit olduğu durumda a[0] değerine eşit olucaktır , bunu haricindeki her durumda 0 olucaktır. Bu kademe kademe out[31] e kadar gidecektir. Detaylar _32_bit_shift_left.v dosyası incelenerek daha rahat anlaşılabilir.

2.16 module _32_bit_shift_right(out,a,b);

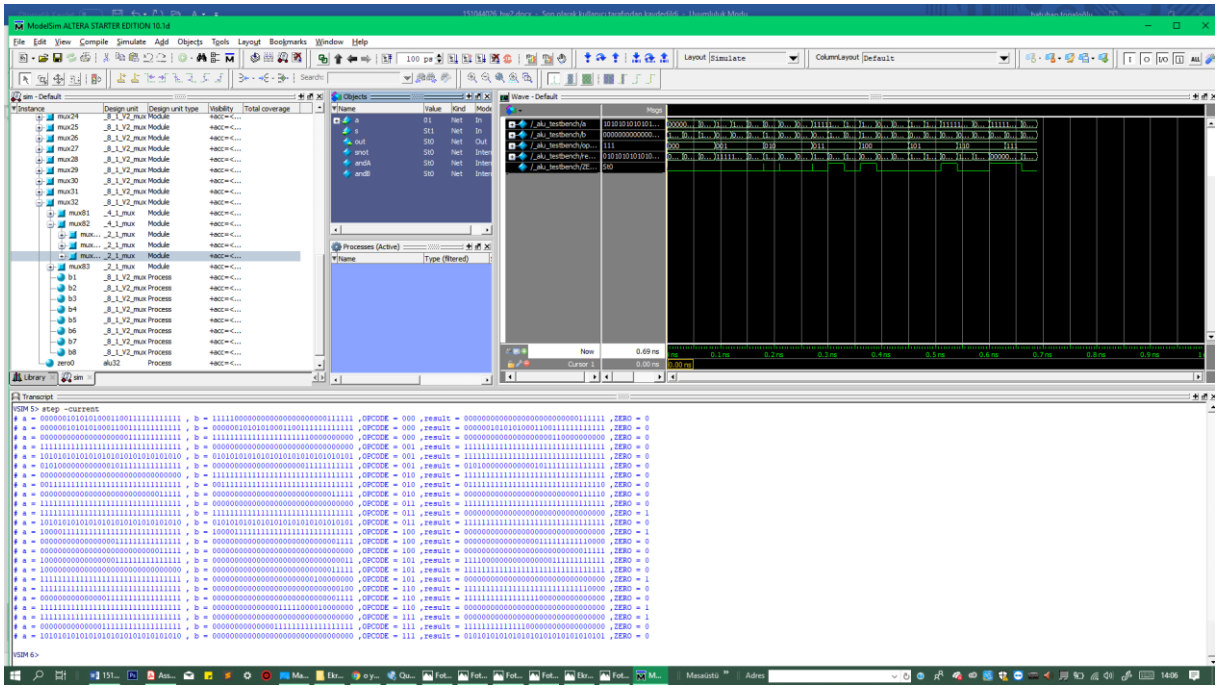
Bu shift modülü 32-bitlik a'yı 32-bitlik b'nin değeri kadar sağa aritmetik shift yapar. Aritmetik shift'in farklı boş kalan bitlere en değerlikli bit olan a[31]'in değerinin konulmasıdır. 'b' değerinin 31 den büyük olduğu her durumda out'un bütün bitleri '0' değeri olucak şekilde ayarlandı. Detaylar _32_bit_shift_right.v dosyası incelenerek daha rahat anlaşılabilir.

2.17 module alu32(result,Zero,a,b,opCode);

Bu projediki ana modül alu32. Yazılan tüm modüllerin amacı buradaki işlemleri gerçekleştirebilmek. Burada input olarak gelen a ve b değerleri tanımlı 8 işlemin hepsine gönderili ve temp resultlar elde edilir. Gelen operasyon koduna göre elde edilen bu result değerlerinin istenilen değer mux'lar yardımı ile seçilerek output olarak verilir. Zero 1bitlik bir output eğer resultın bütün bitleri 0 sa Zero değeri 1 oluyor. 'result' 32-bit işlem sonucu , a ve b 32-bit üzerlerinde işlemlerin gerçekleştirileceği değerler. 'opCode' yapılacak işlemi belirleyen operasyon kodu.

3.Modelsim Simülasyon Sonuçları:

24 farklı test case'ini '_alu_testbench' modülünü kullanarak testler gerçekleştirildi. Tanımlar müdüller eksiksiz olarak çalışmakta. Çıktının bütün bitlerinin 0 olması durumda ZERO biti 1 olmaktadır. (Modelsim de testlerin gerçeklenme süresi çok uzun sürmektedir, sebebi hakkında bir fikrim bulunmamakta.)



Büyütürsek :

```
VSIM 5> step -current
# a = 00000010101010001100111111111111 , b = 11111000000000000000000000111111 ,OPCODE = 000 ,result = 00000000000000000000000000111111 ,ZERO = 0
# a = 00000010101010001100111111111111 , b = 00000010101010001100111111111111 ,OPCODE = 000 ,result = 00000010101010001100111111111111 ,ZERO = 0
# a = 00000000000000000000001111111111 , b = 11111111111111111111110000000000 ,OPCODE = 000 ,result = 00000000000000000000110000000000 ,ZERO = 0
# a = 11111111111111111111111111111111 , b = 00000000000000000000000000000000 ,OPCODE = 001 ,result = 11111111111111111111111111111111 ,ZERO = 0
# a = 10101010101010101010101010101010 , b = 01010101010101010101010101010101 ,OPCODE = 001 ,result = 11111111111111111111111111111111 ,ZERO = 0
# a = 01010000000000001011111111111111 , b = 00000000000000000000111111111111 ,OPCODE = 001 ,result = 01010000000000001011111111111111 ,ZERO = 0
# a = 00000000000000000000000000000000 , b = 11111111111111111111111111111111 ,OPCODE = 010 ,result = 11111111111111111111111111111111 ,ZERO = 0
# a = 00111111111111111111111111111111 , b = 00111111111111111111111111111111 ,OPCODE = 010 ,result = 01111111111111111111111111111110 ,ZERO = 0
# a = 00000000000000000000000000001111 , b = 00000000000000000000000000001111 ,OPCODE = 010 ,result = 00000000000000000000000000001110 ,ZERO = 0
# a = 11111111111111111111111111111111 , b = 00000000000000000000000000000000 ,OPCODE = 011 ,result = 11111111111111111111111111111111 ,ZERO = 0
# a = 11111111111111111111111111111111 , b = 11111111111111111111111111111111 ,OPCODE = 011 ,result = 00000000000000000000000000000000 ,ZERO = 1
# a = 10101010101010101010101010101010 , b = 01010101010101010101010101010101 ,OPCODE = 011 ,result = 11111111111111111111111111111111 ,ZERO = 0
# a = 10000111111111111111111111111111 , b = 10000111111111111111111111111111 ,OPCODE = 100 ,result = 00000000000000000000000000000000 ,ZERO = 1
# a = 00000000000000000011111111111111 , b = 00000000000000000000000000001111 ,OPCODE = 100 ,result = 00000000000000000011111111110000 ,ZERO = 0
# a = 00000000000000000000000000001111 , b = 00000000000000000000000000000000 ,OPCODE = 100 ,result = 00000000000000000000000000001111 ,ZERO = 0
# a = 10000000000000000111111111111111 , b = 00000000000000000000000000000011 ,OPCODE = 101 ,result = 11110000000000000111111111111111 ,ZERO = 0
# a = 10000000000000000000000000000000 , b = 00000000000000000000000000001111 ,OPCODE = 101 ,result = 11111111111111111111111111111111 ,ZERO = 0
# a = 11111111111111111111111111111111 , b = 00000000000000000000000100000000 ,OPCODE = 101 ,result = 00000000000000000000000000000000 ,ZERO = 1
# a = 11111111111111111111111111111111 , b = 00000000000000000000000000000100 ,OPCODE = 110 ,result = 11111111111111111111111111110000 ,ZERO = 0
# a = 00000000000000001111111111111111 , b = 00000000000000000000000000001111 ,OPCODE = 110 ,result = 11111111111111110000000000000000 ,ZERO = 0
# a = 11111111111111111111111111111111 , b = 00000000000000001111000010000000 ,OPCODE = 110 ,result = 00000000000000000000000000000000 ,ZERO = 1
# a = 11111111111111111111111111111111 , b = 00000000000000000000000000000000 ,OPCODE = 111 ,result = 00000000000000000000000000000000 ,ZERO = 1
# a = 00000000000000001111111111111111 , b = 00000000000000001111111111111111 ,OPCODE = 111 ,result = 11111111111111110000000000000000 ,ZERO = 0
# a = 10101010101010101010101010101010 , b = 00000000000000000000000000000000 ,OPCODE = 111 ,result = 01010101010101010101010101010101 ,ZERO = 0
```