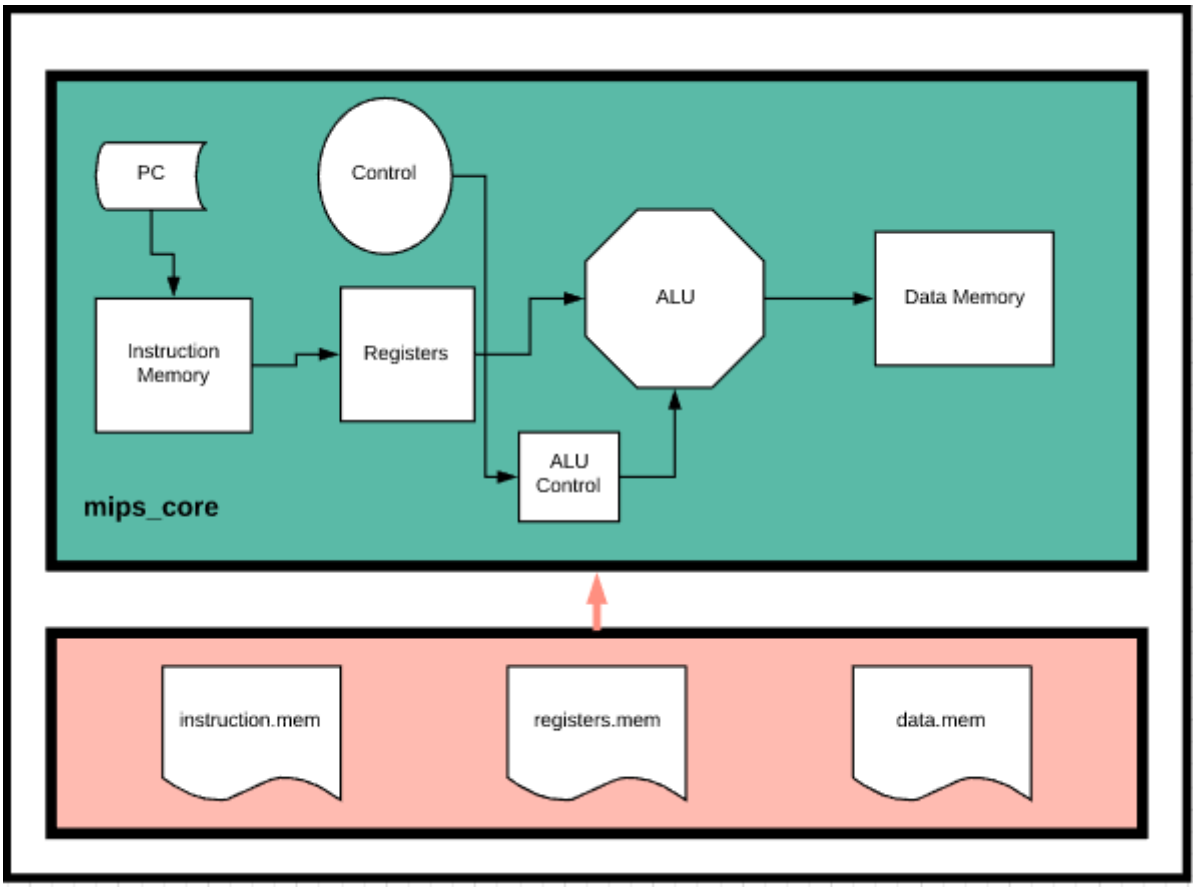
Gökçe DEMİR
141044067

BİL331 FİNAL PROJE RAPORU

1.Giriş

1.1



1.2

<u>Single Cycle DataPath'e Giren Bir Instruction'ın Geçtiği Yollar :</u>

Datapath'e gelecek olan instruction input olarak instruction.mem isimli dosyadan okunur. Bunu gerçekleştirmek için program counter devreye girer. Program counter'ın değerine göre instruction getirilir.

Okunan 32 bitlik instruction, instruction memory modülünde parçalara ayrılır. Bu parçalama işlemi instructionun tipine göre değişir. 3 tip instructionu farklı şekillerde parçaladım. Aşağıda 32 biti parçalara ayrılmış şekilde şematize ettim.

R-type instructions

31-26	25-21	20-16	15-11	10-6	5-0
Opcode	Rs	Rt	Rd	Shift amount	function

I-type instructions

31-26	25-21	20-16	15-0
Opcode	Register s	Register t	Immediate

J-type instructions

31-26	25-0
Opcode	Target address

Parse edilen rt ,rd ve rs'in 5 bitlik adresleri register modülüne input olarak gönderilir. Register modülü rs, rt ve rd 'in contentini registers.mem isimli dosyadan okuyarak bulur. Bunlar daha sonra aluya input olarak gönderilir. Alu control unitten gelen değerlere bakarak registers contentlerini kullanarak işlemini gerçekleştirir. R-type instructionlarının memory bloğuyla işleri yok. Memory bloğunda store instructionlar ve load instructionları kullanılır. Gelen sinyaller ve register tiplerine göre gerekli hesaplamlar alu tarafından yapılıp gerek memorye gerek registera yazılır. Jump instructionlarında da memory ve register ile işlem yapılmaz sadece program counterin değeri güncellenir.

1.3

Single cycle datapathde bulunan tüm modülleri yazdım ve bu modüllerin testbenchlerini yazarak test ettim. Clock kullanmada

sıkıntı yaşadığım için instructionların hesaplanmasında sorunlar çıkabiliyor. Bunu düzeltmeye vakit bulamadım.

2.Method

mips_core.v :

module mips_core(input clock) → input olarak clock alır. Clock sinyalinin değişimine göre instructionları alır. Başlangıçta program counter sıfıra set edilir. Ardından instruction methodundan gelen instruction ile instrcution parser işlemi gerçekleşir. Diğer tüm methodlar burada çağrılır. Bu modül top modüldür ayrıca.

Mips_core'un testbecnch'inde de instruction sayisi *2 kadar clock yazılıp method test edilir.

mips_registers.v :

module mips_registers

(read_data_1 → output olarak rs'in contentini döndürür

read_data_2 → output olarak rt'in contentini döndürür

write_data → registera yazılacak olan data input olarak gelir

read_reg_1 → 5 bitlik rs'in adresini getiren input

read_reg_2 → 5 bitlik rt'in adresini getiren input

write_reg → registera yazılacak olan datanın adresi

signal_reg_write → control unit tarafından üretilen sinyal

clk) → yazma ve okuma işlemlerinin senkronize bir şekilde gerçekleşmesine yardımcı olur.

Registers.mem isimli dosya okunarak rs ve rt'İn contentleri elde edilir. Aynı zamanda registera yazma işlemi gerçekleştiğinde de write_datayı registera yazar.

Mips_registers_testbench'de test edilip registers yazıldığı görülmüştür.

mips_instr_mem.v :

module mips_instr_mem

(instruction → 32 bitlik output olarak instruction çevirir

program_counter →32 bitlik input olarak gelir

)

İnput olarak aldığı program counterin gösterdiği instrucition memorydeki instructionu getirip instruction isimli outputa atar.

Mips_inst_mem_testbenchde de test edilip instructionları instruction.mem isimli file okuduğu görülmüştür.

Gökçe DEMİR
141044067

Alu.v :

module alu

(zero→ sonuç 0 ise 0 output olarak çıkar

ALU_result→ control sinyallerine göre gerekli işlemleri yapıp alu resulta yüklenir.

read_data_1→ rs contenti 32 bit input

read_data_2→ rt contenti 32 bit input

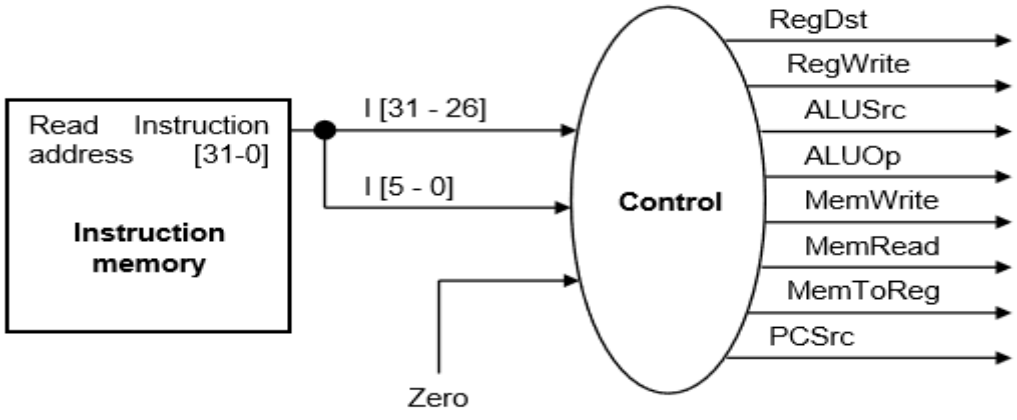
shamt → shift amount r type instrucionlarda kullanılan input

ALU_control→ control unit tarafından üretilen kontrol sinyali olan input.

Gerekli işlemleri yapıp alu resulta atar. Eger alu resultumuz 0 ise sifir registeri outputumuz olur.

Alu testbenchde test edilip işlemleri doğru yaptığı görülmüştür.

Control_unit.v :



Gökçe DEMİR
141044067

module control_unit

(output RegDst → sonucun hangi registera yazılacağını söyler (rt veya rd)

output Branch → branch yapılıp yapılamayacağını söyler

output MemRead → register okuma yapılıp yapılamayacağı söyler

output MemtoReg → memoryeden registera data aktarılırsa 1 olur

output [2:0] ALUOp, → bu output alu_control'a input olarak gider

output MemWrite → memorye yazma yapılacaksa 1 olur (store ins.)

output ALUSrc → sign extend veya rt secilir

output RegWrite → registera yazma yapılacaksa 1 olur (load ins.)

output jump, //jump

output bne, //brach not equal

output jal, //jump and link

output lui,

output lbu,

output lhu,

output sb,

output sh,

input [5:0] opcode → instruction[31:26]

Gökçe DEMİR
141044067

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

Burada ki sinyaller uygulanmıştır.

testbenchde test edilip çalıştığı gözükmüştür.

Alu_control_unit.v :

module alu_control_unit(

ALU_control-- > 3 bit control singal

ALUOp → 3 bit

funct) → //6-bit funciton field

Gökçe DEMİR
141044067

3 .RESULT:

Mips testbench result

```
# Loading work.mips_data_mem
add wave -position insertpoint  \
sim:/mips_testbench/clock
VSIM 18> step -current
# Instruction: 00000010000100011001000000100010, PC:        0 RS:        16, RT:        17, regWrite_signal:1, RD_adress: 18, RD:4294967295 mem_address: 4294967295, ALU_REsult= 4294967295
#  shamt =  0, Write_data= 11111111111111111111111111111111, Sign_Extend: 11111111111111111001000000100010
# Instruction: 00000001111011100011100000100000, PC:        1 RS:        96, RT:        12, regWrite_signal:1, RD_adress:  7, RD:       113 mem_address:       113, ALU_REsult=       113
#  shamt =  0, Write_data= 00000000000000000000000001110001, Sign_Extend: 00000000000000000111000001100000
# Instruction: 00000010000011110100000000100000, PC:        2 RS:        16, RT:        96, regWrite_signal:1, RD_adress:  8, RD:        33 mem_address:        33, ALU_REsult=        33
#  shamt =  0, Write_data= 00000000000000000000000000100001, Sign_Extend: 00000000000000001000000100000
# Instruction: 00000010000100011001000000100000, PC:        3 RS:        16, RT:        17, regWrite_signal:1, RD_adress: 18, RD:        33 mem_address:        33, ALU_REsult=        33
#  shamt =  0, Write_data= 00000000000000000000000000100001, Sign_Extend: 11111111111111111001000000100000
# Instruction: 00100010011100110000000000000001, PC:        4 RS:        19, RT:        19, regWrite_signal:1, RD_adress:  0, RD:        36 mem_address:        36, ALU_REsult=        36
#  shamt =  0, Write_data= 00000000000000000000000000100100, Sign_Extend: 00000000000000000000000000000001
# Instruction: 00000010100101010110000101010, PC:        5 RS:        20, RT:        21, regWrite_signal:1, RD_adress: 11, RD:         0 mem_address:         0, ALU_REsult=         0
#  shamt =  0, Write_data= 00000000000000000000000000000000, Sign_Extend: 00000000000000001011000001010
# Instruction: 00101010110011000000000000010000, PC:        6 RS:        22, RT:        32, regWrite_signal:1, RD_adress:  0, RD:         5 mem_address:         5, ALU_REsult=         5
#  shamt =  0, Write_data= 00000000000000000000000000000101, Sign_Extend: 00000000000000000000000000010000
# Instruction: 10001100101001110000000000001010, PC:        7 RS:         5, RT:       113, regWrite_signal:1, RD_adress:  0, RD:        15 mem_address:        15, ALU_REsult=        15
#  shamt =  0, Write_data= 00000000000000000000000000001111, Sign_Extend: 00000000000000000000000000001010
# Instruction: 10101101110110000000000000001101, PC:        8 RS:        12, RT:        24, regWrite_signal:0, RD_adress:  0, RD:        22 mem_address:        22, ALU_REsult=        22
#  shamt =  0, Write_data= 00000000000000000000000000010110, Sign_Extend: 00000000000000000000000000001101
# Instruction: 00000010000100011001000000100000, PC:        9 RS:        16, RT:        17, regWrite_signal:1, RD_adress: 18, RD:        33 mem_address:        33, ALU_REsult=        33
#  shamt =  0, Write_data= 00000000000000000000000000100001, Sign_Extend: 11111111111111111001000000100000

VSIM 19>
```