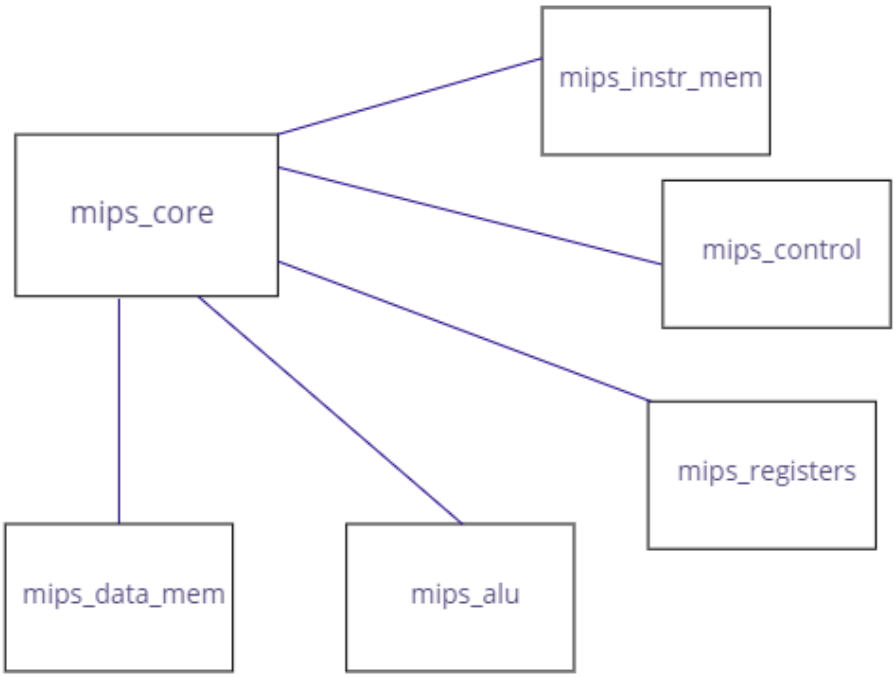
CSE 331 - Computer Organization

Final Project Report – 141044024

1. **Introduction**

   1.1 Big Picture



mips_core : clock inputu alıp, diğer bütün modülleri çağırıp, bu modüllerin birbirlerine verdiği input outputları ayarlar. Genel merkezdir.

mips_instr_mem : PC nin değişimine bağlı olarak instruction outputunu üretir.

mips_control : Gelen instruction'a göre gerekli sinyalleri üretir ve yazılacak register gibi ayarlamaları yapar.

mips_registers : Register'lar üzerinde okuma ve yazma yapar.

mips_alu : rs ve rt content'leri alıp gerekli işlemi yapar.

mips_data_mem : Memory üzerinde okuma ve yazma yapar.

1.2 Life cycle of 1 instruction

Program ilk çalıştırıldığında testbench'teki initial begin kısmında clock'a gelip clock yapmadan önce tüm kodlar derlenip initial'lar çalıştırılıp dosyalar okunur**.** Clock 0 olarak verildiğinde mips_core çalışmaya başlar. PC 0 ile initialize olduğundan **mips_instr_mem** (inputu pc olduğundan)modülü çalışır ve instruction okunur. Okunan bu instruction, mips_core' da fetch edilip önce **mips_control** modülü çağrılır. Bu modül, clock 0 olduğunda(negedge) always bloğunu çalıştıracağından, şimdi always bloğuna girer ve op,fn gibi değerlere bakıp, sig_branch, sig_mem_read, sig_mem_write, sig_mem_to_reg, alu_src, sig_reg_write, pc_source, sig_sb, sig_sh sinyallerine gerekli atamayı yapıp, writeReg'e de rd veya rs adresini atama yapar. Burayla işi bittikten sonra **mips_registers** modülünün posedge clk olan always bloğu çalışıp rs ve rd adresi kullanılarak, rsCont ve rtCont okunur. Daha sonra, always bloğu rs veya rt contentine göre çalışan **mips_alu** modülü çalışıp instructiona göre bir hesap sonucu output olarak üretir. alu_out, hem mem_adress'e hem write_data_mem'e hem de register'ın writeData'sına atanır. Bu aşamada memory nin always bloğun'daki mem_adress ve write_data_mem değiştiği için memory çalışır ve gerekirse sinyallere göre okuma veya yazma yapar. Okuma yaparsa bunu read_data_mem'e atar. writeData'ya alu_out veya read_data_mem atamasının kontrolü ise mips_core'da alttaki always@(alu_out or read_data_mem) bloğunda gerçekleşir. Instructionun durumuna ve sinyallere göre bit representasyonları burada değişir. Registera yazma işlemi clock 1 yapıldığında meydana gelir ve bu aşama geçildikten sonra mips_core'un posedge clock olan always bloğunda PC değiştirilir ; PC arttırılır veya bulunulan istructionun tipine göre jumpAdr atanır vs. Sonraki clock 0 oldugunda yeni instruction çalışmaya başlamış olur. Sonuç olarak Instruction sayısı x 2 adet clock yapılır.

2. METHOD

- **mips_core :**

input:

-> clock

Detailed explanation : Clock değiştikçe programın çalışması sağlanır. 1 instruction için 0 iken sinyal üretme, register okuma, memory okuma, yazma, alu, 1 iken ise register yazma ve PC değiştirme işlemleri uygulanır.

- **mips_instr_mem :**

input

-> program_counter

output

-> instruction

Detailed explanation : PC değiştikçe, instructionu instr_mem'den PC indisinden okur.

- **mips_control :**

input

-> op

-> clk

-> fn

output

-> write_reg, rd, rt

-> sig_mem_read, sig_mem_write, sig_mem_to_reg, sig_sb, sig_sh, sig_reg_write

-> sig_branch

-> pc_source

-> sig_reg_write

Detailed explanation : Instruction yeni çalıştırıldığında sinyallerin belirlenmesi gerektiği için negedge clk da always bloğu çalışır. op ve fn kullanılarak önceden tanımlanan wire'larla eşleştirilip, instructionun ne olduğu anlaşılır. Bu duruma göre, writeReg'e rd, rt ya da 31(return adress) ataması yapılır. Branch instructionlarına göre sig_branch sinyali atanır. Jr instructionu varsa pc_source atanır. Registera yazma yapılacaksa sig_reg_write atanır. Memory okuma yazma, store byte, store half Word ve memoryden registera(lw) gibi işlemler için sig_mem_read, sig_mem_write, sig_mem_to_reg, sig_sb, sig_sh, sig_reg_write sinyalleri atanır.

- **mips_registers :**

  input

  -> write_data

  -> read_reg_1, read_reg_2, write_reg

  -> signal_reg_write, clk

  output

  -> read_data_1, read_data_2

Detailed explanation : negedge clk olan always bloğu, clock 0 iken çalışıp registers' tan read_reg_1 yani rs adresi ve read_reg_2 yani rt adresini verip, read_data_1 ve read_data_2 olarak outputları verir. Posedge clk olan always bloğu ise clock 1 iken çalışıp eğer signal_reg_write(yazma sinyali) 1 ise registers[write_reg] ' e write_data'yı yazar.

- **mips_alu :**

  input

  -> rsCont, rtCont

  -> op, fn, sa, imm

  output

  -> aluOut

Detailed explanation : rs content veya rt content'in değişimine göre çalışan always bloğu instructionu anlayıp, aluOut'a gerekli işlem sonucunu yazar.

- **mips_data_mem :**

  input

  -> mem_adress, write_data

  -> sig_mem_read, sig_mem_write, sig_sb, sig_sh, clk

  output

  -> read_data

Detailed explanation : mem_address or sig_mem_read içeren always bloğu, bunlardan biri değiştiğinde çalışır ve sig_mem_read varsa, memory den adresi mem_adress olan datayı okuyup read_data'ya atar. Posedge clock always'te, clock 1 iken sig_mem_write varsa memory'ye mem_adress olan adreste write_data yı yazar. Yazma işlemini yaparken store byte sinyali (sig_sb) veya store half word sinyali(sh) ' a göre bit representasyonu değişir. Bunlar yoksa normal 32 biti aynı şekilde yazar.

3. RESULT

3.1 Testbench Results

TestBench

```
1   module mips_testbench ();
2   reg clock;
3   wire result;
4
5   mips_core test(clock);
6
7
8   // (Instruction sayisi) x (2) sayisi kadar clock
9   initial begin
10     #50 clock = 0; #50 clock=~clock;
11     #50 clock=~clock; #50 clock=~clock;
12     #50 clock=~clock; #50 clock=~clock;
13     //depend instrucion number
14  end
15
16  // islemler bitince register ve memory sonucunu dosyaya yazma
17  initial begin
18   #2500
19   $writememb("res_registers.mem", test.registers.registers);
20   $writememb("res_data.mem", test.mips_data_mem.data_mem);
21  end
22
23  endmodule
```

3 instruction denedim x 2 = 6 clock, 2500 delay time

Tested Instructions :

```
10010000001000110000000000000011
00001100000000000000000000001011
00000001111011100011100000100000
00000010000011111010000000100000
00001000000000000000000000000111
00100010011100110000000000000001
10010010000010000000000000000010
10010110001010010000000000000100
00000010100101010101100000101010
00101010110011000000000000010000
10100010111011010000000000100000
10100111000110010000000000001000
00000010000100011001000000100010
10101100001000110000000000000001
10001100001000010000000000000011
```

1-) 10010000001000110000000000000011 -> lbu instructionu : rs : 1, rt = 3, imm = 3

rs + imm = 1 + 3 = 4 Memory adresinden okunan bilgiyi rt' ye (3'e) yazar. Registers[3] = 4. → 00…0100

Registers Result

```
1   00000000000000000000000000000000
2   00000000000000000000000000000001
3   00000000000000000000000000000011
4   00000000000000000000000000000100
5   00000000000000000000000000000100
6   00000000000000000000000000000101
```

2-) 00001100000000000000000000001011 -> jal instruction : imm : 11

registers[31] = PC + 1

PC = 11

Registers Result

```
28  00000000000000000000000000011011
29  00000000000000000000000000011100
30  00000000000000000000000000011101
31  00000000000000000000000000011110
32  00000000000000000000000000000010
33
```

3-) 11.instructiona atladı

10100111000110010000000000001000 -> sh instructionu : rs : 24, rt : 25(11001), imm : 8

Mem_adress = 24 + 8 = 32, MEM[32] = rt(25)

Memory Result