Laboratuvar Çalışması 0x8

Tek Ritimli İşlemci Tasarımı

I. Amaç

Bu labın amacı

- Basitleştirilmiş bir komut setine göre tek ritimli işlemci tasarlamak. GTE20 (Gebze Teknik Elektronik 2020) komut kümesi, 22 komut.
- İşlemci içerisindeki farklı parçaların birbiri ile nasıl bağlandığını öğrenmek.
- Tasarlanan işlemciyi gerçekleyip, belirlenen komut setine göre derlenmiş bir programı çalıştırmak.
- İşlemcinin performansını ölçmek ve doğrulamasını yapmak.

II. Uyarılar

1. Raporunuza her problem için RTL şemasını ekleyin.
2. Sadece son problem için utilization raporu, Fmax değerini ekleyin.
3. Kodlarınızın başına daha önce verilen proje şablonunu ekleyin.
4. Devrelerinizi ayrı ayrı sentezleyip, devrelerde latch oluşmadığına emin olun.
5. Bu işlemcide data memory kullanmayacağız ve dolayısıyla LDR / STR gibi komutlar bulunmayacaktır.

III. Problemler

Problem 1 - Instruction Fetch ünitesi

Bu problemde işlemcinin ilk parçası PC (program counter) adında, bağlanan *komut hafızası*ndan (instruction memory), bir sonra çalıştırılacak komutun adresini hesaplayan üniteyi tasarlayacaksınız.
Bu ünite:

- **pc** sinyalinde register bulunacak.
- senkron olarak bağlanmış, aktif-low olarak çalışan **reset** sinyali geldiği zaman **pc** değerini 0 layacak.
- eğer dışarıdan gelen **pc_update** sinyali aktif ise, **pc** değerine yine dışarıdan gelen **pc_new** değeri atanacak
- eğer **pc_update** sinyali aktif değil ise, yeni **pc** değeri olarak, bir önceki **pc** değerinin 4 fazlasını atayacak.

Örnek bir modul başlığı aşağıda verilmiştir.

```
module fetch (
    input logic clk, reset,
    output logic [31:0] pc,
    // execute dan gelen pc güncellemeleri
    input logic pc_update, // execute pc_update sinyaline baglanacak
    input logic [31:0] pc_new // execute sonic sinyaline baglanacak
);
```

1. Bu modülü test etmek için öncelikle basit bir testbench yazınız. **pc** değerinin her clock ile 4 arttığını, **pc_update** sinyalini aktive ettiğinizde de, **pc** değerinin gönderdiğiniz **pc_new** sinyaline eşit olduğunu gözlemleyiniz.
2. Testbenchinizde, 64 satırlık, 32-bitlik komut hafızası oluşturun. Lab ile birlikte verilen hafıza değerleriyle yükleyin. Gelen **pc** sinyalinin ilk 30 bitini bu hafızanın adresi olarak verin. Hafızanın çıktısını da komut olarak belirleyip, testbenchinizde hafızaya yüklediğiniz değerleri gözlemleyin. Bu testbench, aynı zamanda tüm işlemciyi simüle etmek için kullanılacaktır.

Problem 2 - Instruction Decode ünitesi

Bu problemde Lab 07, Problem 3 de tasarladığınız üniteyi *instruction decode* olarak kullanıp, Problem 1 de tasarladığınız üniteyle birbirine bağlayacaksınız.

- Bunun için öncelikle işlemcinin ana bağlantılarını (top-level) tutacak olan bir devre oluşturun.
- **Devre ismi olarak p2 değil, soyisim (bireysel) veya soyisimlerinizi (grup ise) kullanın. İşlemcinizin ismi bu isim olacak, ve raporunuzun cover sayfasında bu isim bulunacaktır.**
- Bu devre içerisinde fetch ve decode ünitelerinizi çağırın. Devrenin **pc** çıkışına fetch ünitesinin **pc** çıkışını, **komut** girişini de decode ünitesinin **komut** girişini bağlayın. Decode ünitesi Lab 07 Problem 3 te tasarladığınızla aynı olacaktır. **we**, **rd** ve **rd_data** sinyallerini çıkardıysanız burada eklemeniz gerekmektedir.
    - Lab 07 Problem 2 deki devreniz Register File görevi görecek olup, komutlar için registerların değerlerini sağlayacak ve tutacak hafıza birimi olacaktır.
    - Bu kodunuzdaki küçük bir değişiklik, initial blogunu kaldırmanız ve registerları **reset** aktif olduğu zaman 0 lamanız olacaktır.
    - Yine okumalarınız **clk** dan bağımsız olacaktır. Yazma işlemi ise **clk** rising edgine bağlı olacaktır.
    - Bununla alakalı bir örneği githubdaki repoda bulabilirsiniz.
- Problem 1 için yazdığınız ikinci testbenchdeki çağırılan devreyi güncelleyip, bu yeni devrenizi çağırın.
- Komut hafızasına yüklediğiniz değerlere göre decode ünitesinin çıkışındaki değerlerin doğru ayrıldığına her bir komut için 1-2 komut oluşturup test ederek emin olun.
- **Bu devrenizi de Quartus da sentezleyip, herhangi bir latch oluşmadığına emin olun.**

Not: Aşağı katmandaki modüllerin iç sinyallerini görmek için ModelSim de **Instance** penceresinden istediğiniz modülü seçip, **Objects** penceresinden bu modül içi sinyalleri **Waveform** unuza çekebilirsiniz.

```
module gte20 (
```

```
    input  logic clk, reset,
    input  logic [31:0] komut, // decode a baglanacak
    output logic [31:0] pc,    // fetch e baglanacak
    output logic hata          // decode a baglanacak
);

module decode (
    input  logic clk, reset,
    input  logic [31:0] komut,
    output logic [6:0] opcode,
    output logic [3:0] func, // alu veya branch ops
    output logic [31:0] rs1_data,
    output logic [31:0] rs2_data,
    output logic [31:0] imm,
    output logic hata,
    // execute dan gelen rd güncellemeleri
    input  logic we,            // sonucu rd ye yaz enable (we)
    input  logic [31:0] rd_data // rd sonucu (sonuc)
);
```

Problem 3 - Execute ünitesi

İşlemcinin son parçası olarak, komutları çalıştırıp, sonuçlara göre gerekli yerleri güncellemek için kullanılacak execute ünitesini tasarlayacaksınız.

1. Execute ünitesi tamamen combinational olacaktır.
2. Daha önce tasarladığınız ALU ünitesini temel alabilirsiniz. *
3. **func** ve **opcode** girişlerinize göre rs1_data, rs2_data veya imm sinyallerinin hangilerini kullanarak sonuçları elde edeceğinizi hesaplayacak bir mux tasarlayacaksınız.
    a. Eklerde verilen tabloda bütün komutlar mevcuttur.
    b. Örnek olarak (`SUB R2, R6, R2` komutu) eğer **func** değeri `1000 (SUB)` ve **opcode** `0000001 (R tipi)` ise
    `sonuc = rs1_data - rs2_data`
    `hata  = 0`
    `pc_update = 0`
    `we = 1`
    olarak belirlenecektir.
    c. Örnek olarak (`BLT R2, R6, #16` komutu) eğer **func** değeri `100 (BLT)` ve **opcode** `0001111 (B tipi)` ise
    `sonuc = 16 (pc_new den pc yi güncelleyecek)`
    `hata  = 0`
    `pc_update = eğer $signed(R2) < $signed(R6) ise 1, değilse 0`
    `we = 0`
    olarak belirlenecektir.
    d. Tablodaki geri kalan bütün kombinasyonlar için tekrarlayın.
    Not: Burada isterseniz **func** girişini, isterseniz de **opcode** girişini ana *case* iniz olarak kullanabilirsiniz. Diğerini *case* içerisinde *if, else if* olarak kullanabilirsiniz.
4. Devrenizde **pc_update** ve **we** aynı anda aktif olmaması gerekir.

5. **Bu devrenizi de Quartus sentezleyip, herhangi bir latch oluşmadığına emin olun.**
6. Bu devre için ayrıca bir testbench yazmanız isteğinizde bağlıdır. Olası problemlerin tespiti için basit bir testbench yazmanız tavsiyemdir.

```
module execute (
    input  logic [31:0] rs1_data, rs2_data,
    input  logic [31:0] imm,
    input  logic [3:0] opcode,
    input  logic [3:0] func, // alu veya branch ops
    output logic [31:0] sonuc, // duruma gore ya rd datası ya yeni pc adresi
    output logic pc_update, // branch basarili, pc yi güncelle
    output logic we, // rd yi güncelle
    output logic hata
);
```

Problem 4 - Tek ritimli işlemci

Bu problemde yapacağınız Problem 3 te tasarladığınız execute ünitesini, Problem 2 de tasarladığınız ana devre içerisinde çağırıp, gerekli bağlantıları yapmaktır. Hata çıkışı hem decode, hem de execute den gelebildiği için, işlemcinin hata çıkışına, başka isimler verdiğiniz decode ve execute hata çıkışlarını OR layıp öyle atayın.

> Örnek olarak
> ...
> `.hata(hata_decode),`
> ...
> `assign hata = hata_execute | hata_decode;`

1. Problem 2 de güncellediğiniz testbenchi olduğu gibi kullanabilirsiniz.
2. Eğer bir yanlışlık yoksa komut hafızanızı, verilen **fib20.mem** programı ile yüklediğinizde, simülasyon sonucunda R6 register ınızda 6765 görmeniz gerekmektedir.
3. Devrenin RTL şemasını çıkarın, Ne kadar register ve LUT harcadığına bakın, ve sonuçları **yorumlayın**.
4. Fmax ı ekleyin ve yorumlayın. Nasıl iyileştirilebileceğini, data memory eklenince ne gibi bir değişme yaşanacağı vs.

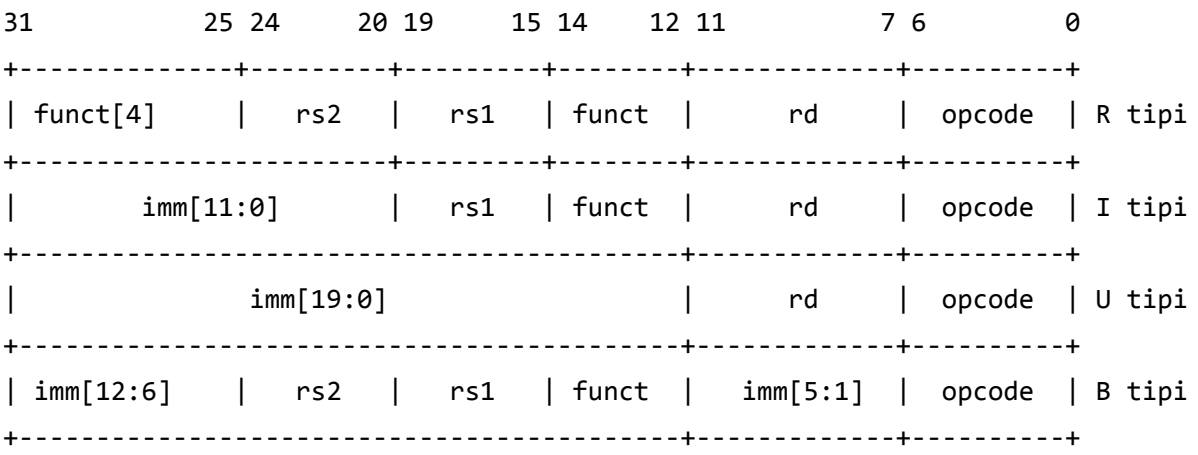
IV. Ekler

EK A - GTE20 - Komut tipleri ve operasyon kodları

Bu komut tipleri ve operasyon kodları Lab 07 de verilenlerle aynıdır.

Operasyon tipi	Operasyon kodu (opcode)
R Tipi	0000001

I Tipi	0000011
U Tipi	0000111
B Tipi	0001111



EK B - GTE20 - Komut kümesi açıklaması

R tipi komutlar - İki operandı da register olan aritmetik ve lojik operasyonlar için kullanılır.
Örnek olarak `ADD R2, R4, R6` komutu için; `(R2 = R4 + R6)`
`rd = 2, rs1 = 4, rs2 = 6, funct = 0000` olarak belirlenmelidir. Geri kalan değerler 0 lanmalıdır.

I tipi komutlar - İlk operandı register, ikinci operandı immediate olan aritmetik ve lojik operasyonlar için kullanılır.
Örnek olarak `EORI R12, R24, #176` komutu için: `(R12 = R24 ^ 176)`
`rd = 12, rs1 = 24, imm = 176, funct = 0100` olarak belirlenmelidir. Geri kalan değerler 0 lanmalıdır.

`imm` - **zero extended** olarak kullanılacaktır.

U tipi komutlar - Verilen immediate sayıyı, belirtilen registera yazmak için kullanılır. (Sadece **MOV** komutu)
Örnek olarak `MOV R18, #423` komutu için: `(R18 = 413)`
`rd = 18, imm = 413` olarak belirlenmelidir. Geri kalan değerler 0 lanmalıdır.
imm - **zero extended** olarak kullanılacaktır.

B tipi komutlar - Branch operasyonlari için kullanılır. **`Bcond rs1 rs2, dest_address`** şeklinde kullanılır. Eğer **cond** verilmemişse, rs1, rs2 değerlerine bakılmaksızın **dest_addresse** gidecektir.
Örnek olarak `BLTU R2, R3, #16` komutu için; (unsigned olarak varsayılan R2, unsigned olarak varsayılan R3 den küçükse 16 adresine git)
`rd = 0, rs1 = 2, rs2 = 3, imm = 16, funct = 0110` olarak belirlenmelidir.

imm - **zero extended** olarak kullanılacaktır.

Not: U versiyonları unsigned, diğerleri signed olarak hesaplanacaktır.

EK C - GTE20 komut kümesi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0000001	ADD
0100000	rs2	rs1	000	rd	0000001	SUB
0000000	rs2	rs1	111	rd	0000001	AND
0000000	rs2	rs1	110	rd	0000001	OR
0000000	rs2	rs1	100	rd	0000001	EOR
0000000	rs2	rs1	001	rd	0000001	LSL
0000000	rs2	rs1	101	rd	0000001	LSR
0100000	rs2	rs1	101	rd	0000001	ASR
imm[11:0]		rs1	000	rd	0000011	ADDI
imm[11:0]		rs1	111	rd	0000011	ANDI
imm[11:0]		rs1	110	rd	0000011	ORI
imm[11:0]		rs1	100	rd	0000011	EORI
imm[11:0]		rs1	001	rd	0000011	LSLI
imm[11:0]		rs1	101	rd	0000011	LSRI
imm[19:0]				rd	0000111	MOV
imm[12:6]	rs2	rs1	011	imm[5:1]	0001111	B
imm[12:6]	rs2	rs1	000	imm[5:1]	0001111	BEQ
imm[12:6]	rs2	rs1	001	imm[5:1]	0001111	BNE
imm[12:6]	rs2	rs1	100	imm[5:1]	0001111	BLT
imm[12:6]	rs2	rs1	101	imm[5:1]	0001111	BGE
imm[12:6]	rs2	rs1	110	imm[5:1]	0001111	BLTU
imm[12:6]	rs2	rs1	111	imm[5:1]	0001111	BGEU

EK D - GTE20 assembly ile Fibonacci serisi 20. sayısının hesabı

fib20.mem

```
00000000000000000000000000000111 // MOV r0, #0       // ADDR: 0 // zero register
00000000000000000000010000010000111 // MOV r1, #1       // ADDR: 4 // one register
00000000000000001001101001000111 // MOV r9, #19      // ADDR: 8 // loop counter
00000000000000000000001000000111 // MOV r4, #0       // ADDR:12 // initials fib0
00000000000000000000010010000111 // MOV r5, #1       // ADDR:16 // initials fib1
// LOOP:
00000000010100100000001100000001 // ADD r6, r4, r5   // ADDR:20 // fib2 = fib0 + fib1
00000000000000010100000100000011 // ADD r4, r5, #0   // ADDR:24 // fib0 = fib1 + 0
00000000000000011000000101000011 // ADD r5, r6, #0   // ADDR:28 // fib1 = fib2 + 0
01000000000101001000010010000001 // SUB r9, r9, r1   // ADDR:32 // loopcounter--
00000000000000100100101010000111 // BNE r9, r0, #20  // ADDR:36 // if (loopcounter != 0) go to address 20
00000000000000000011101000001111 // B #40            // ADDR:40 // go to address 40 (self) --> while(1);
// Simulasyon sonunda r6 sonucu gosterecektir.
```