**RTL (REGISTER TRANSFER LEVEL) DESIGN**

We are going to combine controller and datapath component. This combination gives us processor.

**Design processor**: controller + datapath component

Steps to design processor:

1. Capture high level state machine (to get the behaviour of the processor)
2. Create datapath to process the data
3. Connect datapath to a controller
4. Derive controllers behaviour (FSM) by modifying high level state machine
5. Implement controller FSM to a circuit

EXAMPLE: Create a soda dispenser which includes a coin detector who detects and defines the value of coin. The processor asks the coins and compares it with a user defined price (s). If the total inserted coin value is greater than s, than you can dispanse a soda, you will not give return change.

c : coin detected (1 bit)

d : dispanse soda (1 bit)

s : price of the soda (8 bit)

a : inserted coin’s value (8 bit)

STEP 1 : Obtain a high level state machine

Inputs: c, s, a  
Outputs : d

Diagram

Description automatically generatedLocal register: tot (8 bit)

This is high level state machine

tot = 0 is actually tot = 0000 0000

We are using comparisons to move between states. This is the difference between regular FSMs.

Also in “Add” state, we are making calculation.

STEP 2 : Create a datapath to process data (our data is s and a)

Operations:

* sum the coins (tot = tot + a)
* store the tot
* compare the tot with s

Diagram

Description automatically generated

tot\_lt\_s = 1 🡪 if tot is less than s

We have 3 datapath components and blue rectangle is our datapath.

STEP 3 : Connect datapath to controller

A picture containing chart

Description automatically generated

STEP 4 : Derive the behaviour of controller

Diagram

Description automatically generated

This is FSM for the controller.

Controller state table:

* We have 4 states so we need 2 state registers (s1 and s0)
* We have 2 inputs (other than s1 and s0) 🡪 c and tot\_lt\_s
* We have 3 outputs (other than n1 and n0) 🡪 d, tot\_ld, and tot\_clr

init

wait

Add

Disp

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| s1 | s0 | c | tot\_lt\_s | n1 | n0 | d | tot\_ld | tot\_clr |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

tot\_clr = s1’s0’

tot\_ld = s1s0’

d = s1s0

…

You can also evaluate n1 and n0.

Controller:

Diagram

Description automatically generated

**EXAMPLE: Laser-based distance measure**

Chart, line chart

Description automatically generated

When the laser is on, the timer is also turned on. We will count the time between the laser left the device, and the laser returned to the sensor.

Speed of the laser light : m/s

x = V t -----------> 2D = t ----------->

We can find D.

A picture containing text

Description automatically generated

The processes in our system are:

1. Power on the system. Laser will be off and display will show 0.
2. System waits for user to press button (system waits B to become 1).
3. After B = 1, the system turns on the laser for just one clock cycle then the laser will be turned off
4. When the laser turned off (light is travelling), the system starts counting time until a return from sensor
5. After reflection detected, the system calculates the distance and displays it and wait for another button press.

STEP 1 : Create high-level state machine

Inputs: B, S  
Outputs: L, D  
Register: DistReg (distance register)

A picture containing text

Description automatically generated

Button, display, laser light, sensor are all digital components.

Let’s assume that the frequency of clock signal = Hz = 300 MHz  
This means, I am producing clock signal in a second.

1 clock signal için geçen süre =

Mesela 10 tane clock signal saydıysak ve geçen zaman istendiyse:

2D = Dctr : clock signal counter

D =

\* Sensörün bir zaman sonra ışığı göreceğini düşünüyoruz. Yani lazerin objeden döneceğini düşünüyoruz.

Diagram

Description automatically generated

init’e dönmeye gerek yok çünkü L zaten sıfırlanmıştı, D’yi de sıfırlamamıza gerek yok en sonda zaten hesaplıyoruz. init’e dönersek display sadece 1 clock cycle için gözükecek. Yani D sadece ilk açılışta 0 olacak, sonra hep bi değer tutacak.

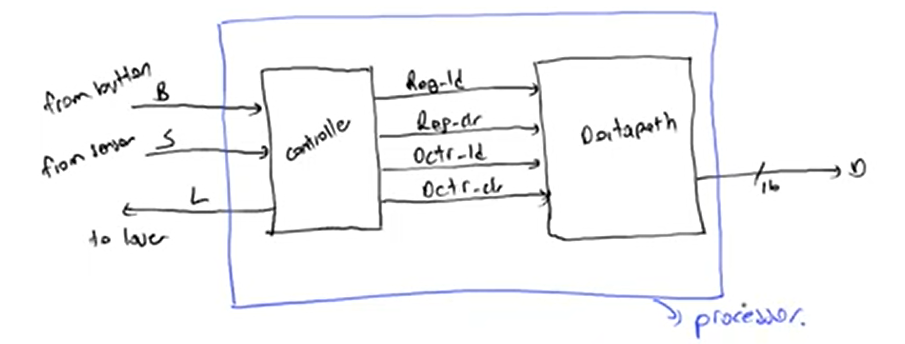
STEP 2 : Create a datapath to process data

* increment (Dctr = Dctr + 1)
* hold value of D

Diagram

Description automatically generated

STEP 3 : Connect datapath to controller



I don’t need to get an output for the L in datapath because L is just 1 bit data. I can easily control a laser light from a controller.

For controller, we have 2 inputs and 5 outputs.

Controller’s main purpose is control Datapath (RTL device).

STEP 4 : Derive the controller’s FSM

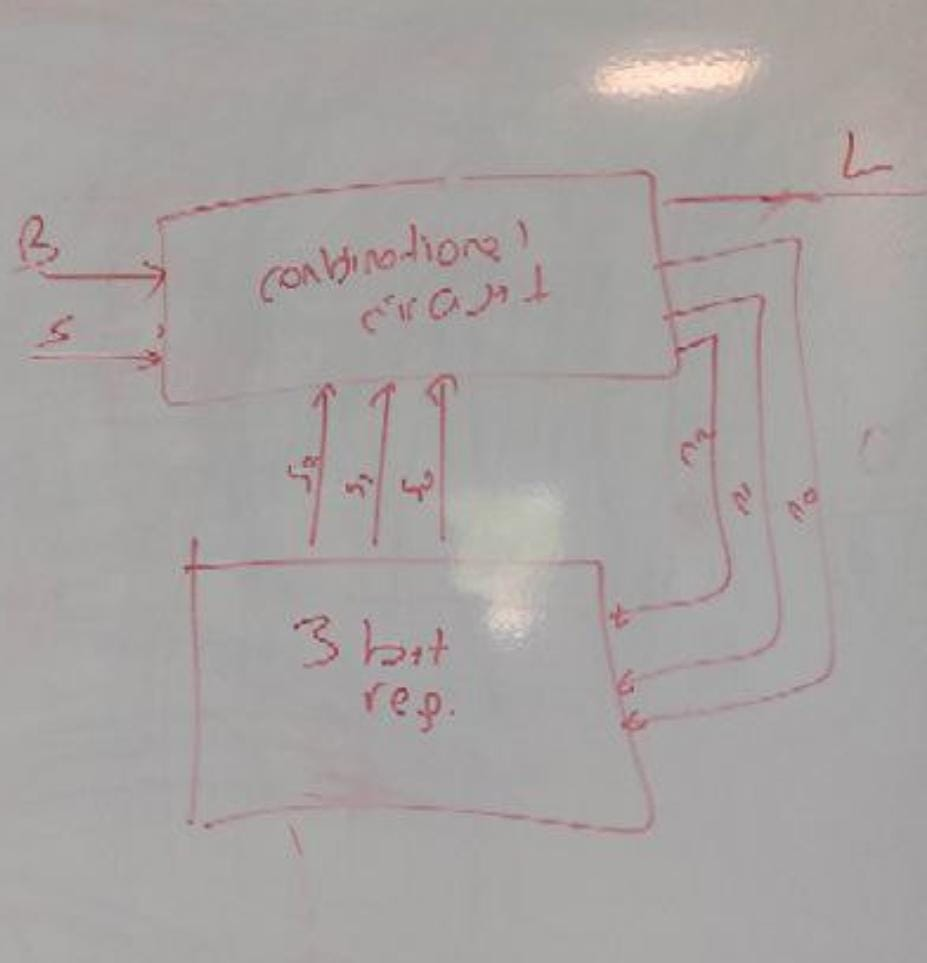
S 1 olunca shift operatoründe hesapladığımız değeri register’a alıyoruz.

Diagram

Description automatically generated

Table’da:

* s2, s1, s0, B, S ---> input
* n2, n1, n0, L, Dctr\_clr, Dctr\_ld, Reg\_clr, Reg\_ld ---> output



**EXAMPLE: Sum of absolute difference**

This system is used in cameras. Each video includes some frames. Each frame is a picture and each picture includes some pixels. Each pixel has a value between 0 and 255 (8 bit value).

You can have 30 frames per second. Since the frames are taken so fast, image may not change in all the frames. Örneğin haber spikerini çekiyosan görüntüde sadece spikerin ağzı hareket eder, diğer kısımlar aynı kalır. No need to process all the pixels. You can only process the changing pixels.

You can take first picture as a base frame, then get the difference between first and second pictures, and you can only process the changing parts.

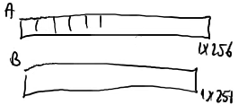
To take differences, usual case is dividing the frames into the subframes because frames could be very large. You cannot process the all frame in 1 operation. Then you can take the differences of the subframes. Subframes are usually 16x16 (256 pixels).

---> Creat a processor to take the sum of differences of a block in consecutive frames.

When you get all the differences for each pixels and sum them, you will define a threshold. If the sum of absolute difference is higher than the threshold, then you can send the new image. If sum of absolute difference is smaller than the threshold, then you can send only the difference.

go: activator (when go is 1, you can make summations)  
A and B: subframes

A and B both include kind of arrays.

These are actually row version of the patch (16x16).

Each item in the array is pixel.

I will subtract pixel at A’s 0th index from pixel at B’s 0th index. Then I will sum all the differences.

STEP 1 : Create high-level state machine (Behaviour of the processor)

Inputs: A (256 bytes), B (256 bytes), go (1 bit)  
Outputs: sad (32 bits - sum of absolute difference)  
Local register: sum, sad\_register, i (9 bits) 🡪 i will show the index of current pixel in block in memory

In state1, we have some data in A and B, then we can make computations. You will start by setting i to 0.

Diagram, whiteboard

Description automatically generatedA picture containing icon

Description automatically generated

Do we need to start from state-1?

* Firstly, we shouldn’t initialize i and sum to 0 at state0. State0 is kind of a loop. At each clock, you return to same operation until go is pressed. So if you initialize i and sum at state0, that means the assignment operation will be repeated at each clock cycle. No need to do that. That works but extra work.
* There is no reason to start from state-1.

sad\_reg’ı 0 yapmamalıyız. 2 patch’in difference’ı 0 olabilir (görüntü hiç değişmemiş olabilir). Yani sad\_reg’i initialize edecek değer yok. -1 olabilir ama ona da ihtiyaç yok. Zaten go 1 olduktan sonra sad\_reg için atama yapıyoruz.

Diagram

Description automatically generated

STEP 2 : Create a datapath to process data

* comparator (i < 256)
* register (to hold the sum)
* substractor (A[i]-B[i])
* absolute-value
* incrementer (i=i+1)
* register (to hold sad\_value)

A picture containing diagram

Description automatically generatedDiagram

Description automatically generated

AB\_addr 🡪 address (index) of the pixels in the block signal

A and B 🡪 These 2 are system itself, not 8 bit values, but memories : 1 pixel of A and B. AB\_addr’i göndereceğiz, o da bana ilgili pixeli gönderecek.

We are not initializing “sad\_reg” anywhere, so “sad\_reg\_clr” is not used.

Here sum is always exists but it is stored when “sad\_reg\_load” is 1. When controller set “sad\_reg\_load” to 1, the value at my input is the result.

Neden “sad\_reg”i kullanıyoruz, zaten sonuç “sum”da saklanıyor?

sum’a sürekli ekleyerek gidiyoruz. Ekleme işleimini 256 kez yapıyoruz. Her çıkan sonucu “toplam budur” diye göndermek mantıksız olur. Bizim istediğimiz son hali.

STEP 3 : Connect datapath to controller

Our processor:

Diagram

Description automatically generated

AB\_rd : AB read. External system with AB\_addr to get the proper pixel.

STEP 4 : Derive the controller’s FSM (Behaviour of the controller)

Whiteboard

Description automatically generated

AB\_rd is to activate the memory, to send the right data. I have to activate that at St3.

STEP 5 : Encode the states

* St0 = 000
* St1 = 001
* St2 = 010
* St3 = 011
* St4 = 100

STEP 6 : State table

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| s2 | s1 | s0 | go | i\_lt\_256 | n2 | n1 | n0 | i\_inc | i\_clr | sum\_clr | sum\_ld | sad\_reg\_ld | AB\_rd |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | x | x | x | x |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | x | x | x | x | x | x |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | x | x | x | x | x | x |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | x | x | x | x | x | x |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | x | 1 | 1 | x | x | x |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | x | 1 | 1 | x | x | x |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | x | 1 | 1 | x | x | x |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | x | 1 | 1 | x | x | x |
| 0 | 1 | 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 0 | 0 | 1 |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 0 | 1 | 0 |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 0 | 1 | 1 |  |  |  |  |  | … |  |  |  |
| 0 | 1 | 1 | 0 | 0 |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 1 | 0 | 1 |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 1 | 1 | 0 |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 1 | 1 | 1 |  |  |  |  |  |  |  |  |  |
| … | … | … | … | … | … | … | … | … | … | … | … | … | … |

For each output, you will get an equation. Then you will use logic circuits to implement these equations in combinational circuit.

Diagram

Description automatically generated with medium confidenceDiagram

Description automatically generated

DESIGN

Diagram

Description automatically generated

🡪 Create high level finite state diagram

Convert Software Programs to High Level State Machine

1. Modify the software by:
   1. assignments
   2. statements (if / if-else)
   3. while loops
2. Convert them to state diagram

If you have for loop, you’ll convert it to while loop; if you have switch case, you’ll convert it to if statement.

**Assignment statement:**

* Assignments are represented as states

Diagram

Description automatically generated

If you have another statement like z = x+10:

Diagram

Description automatically generated

You have to pay attention that, we spend 1 clock signal at assignment state (S1).

If you want to assign x=0, you shouldn’t do computation with x at the same time. You need to do that one at another state. So don’t put “z=x+10” and “x=0” at the same state.

**if statement:**

A picture containing text, hanger, linedrawing

Description automatically generated

**if-else statement:**

Diagram

Description automatically generated

**while loop:**

Diagram, engineering drawing

Description automatically generated

Diagram

Description automatically generatedEXAMPLE

Inputs: uint x, uint y  
Output: uint max

if (x > y) max = x;  
else max = y;

Instead of using whole program for this one, you can design specific processor to handle that program. It will be quite fast, cheap compared to implementing a big system.

For example: there are many specific processors for the cars.

**EXAMPLE: Same problem 🡪 sum of absolute differences**

//Inputs: byte A[256], byte B[256], bit go

//Output: int sad

main(){

uint sum;

short uint i;

while (1){

while (!go);

sum = 0;

i = 0;

while (i < 256){

sum = sum + abs(A[i] - B[i]);

i = i + 1;

}

sad = sum;

}

}

while (!go);

* B is unnecessary because there is no statement (assignment) in the while loop.

Diagram

Description automatically generated

A picture containing diagram

Description automatically generated

Why we do assignments in the end of while loop instead of in separate state?

We are doing nothing with C statement, it is unnecessary. So we can do assignments in C and ignore D (Or ignore C in left figure).

while (i < 256)

Letter

Description automatically generated with medium confidence

At the end we have:

Diagram

Description automatically generated

This is same diagram as we did before:

Diagram

Description automatically generated

Controllerda veri tutmuyoruz ve veriyi işlemiyoruz. Sadece stateleri tutuyoruz.

Controllerda processerımı (datapath bileşenlerimi) çalıştırabileceğim tek bitlik controller outputlarımın assignmentını yapıyorum.

Kukla gibi. Kolu hareket ettirmek için, başı hareket ettirmek için vs. (datapath components) iplerle kuklayı kontrol ediyorsun. Eli controller olarak düşün. Datapathin inputları = kuklanın ipleri.

**Comparators**

a > b , a = b , a < b ---------> 3 outputs

**EQUALITY**

A: a3a2a1a0  
B: b3b2b1b0

You have to compare these 4 bit numbers bit by bit.

We can use XNOR 🡪 0-0 and 1-1 is 1, others are 0.

Diagram

Description automatically generated

This is a datapath component.

**a < b, a > b, a = b**

A = 1011  
B = 1001

I can compare bit by bit starting from left bit. If a3 is greater than b3, that means A is greater than B, no need to look other bits. If a3 and b3 are equal, I can compare a2 and b2…

out\_gt 🡪 a is bigger than b  
out\_lt 🡪 a is less than b  
out\_eq 🡪 a is equal to b

Diagram, schematic

Description automatically generated

*GREEN BOX IS OUR DATAPATH COMPONENT*

out\_gt = in\_gt + (in\_eq)(ab’)

out\_eq = (in\_eq)(a***XNOR***b)

out\_lt = in\_lt + (in\_eq)(a’b)

Diagram

Description automatically generated

If you want to compare 16 bit numbers, you can combine 2 block diagrams of 8 bit comparator above.

Diagram

Description automatically generated

**EXAMPLE: Design a datapath component which finds the minimum of two 8 bit numbers**

Diagram

Description automatically generated

PAY ATTENTION (study by yourself):

* Incrementer
* Decrementer
* ALU