NC State University	ECE 310	Fall 2015
ECE Department	Design of Complex Digital Systems	Kamal Sundaresan

Project #1: Simple CPU

1. Introduction

The main purpose of this project is to let you start dealing with more complex designs, and become familiar with some of the elements used within a CPU.

2. Learning Objectives

- Complete a design involving separate control and datapath with multiple modules
- Complete a design that includes most of the elements to be used in the CPU

3. Project Report

You are expected to turn in a report after the end of this project. Follow the project report format on the course web-site. Be sure to include all items listed in that report format for full credit.

4. Wolfware Submission

You also need to submit your Verilog code electronically through Wolfware as **proj1.v**. This file should contain a module called proj1. It may use the *‘include* directive to include other files, if you wish, but they must also be submitted with Wolfware. A test bench needs to be submitted as well. The memory file is provided, along with the expected output (**Final_values.txt**). In addition, a second program will be used to test your code that will not be provided.

5. Design: simplified CPU

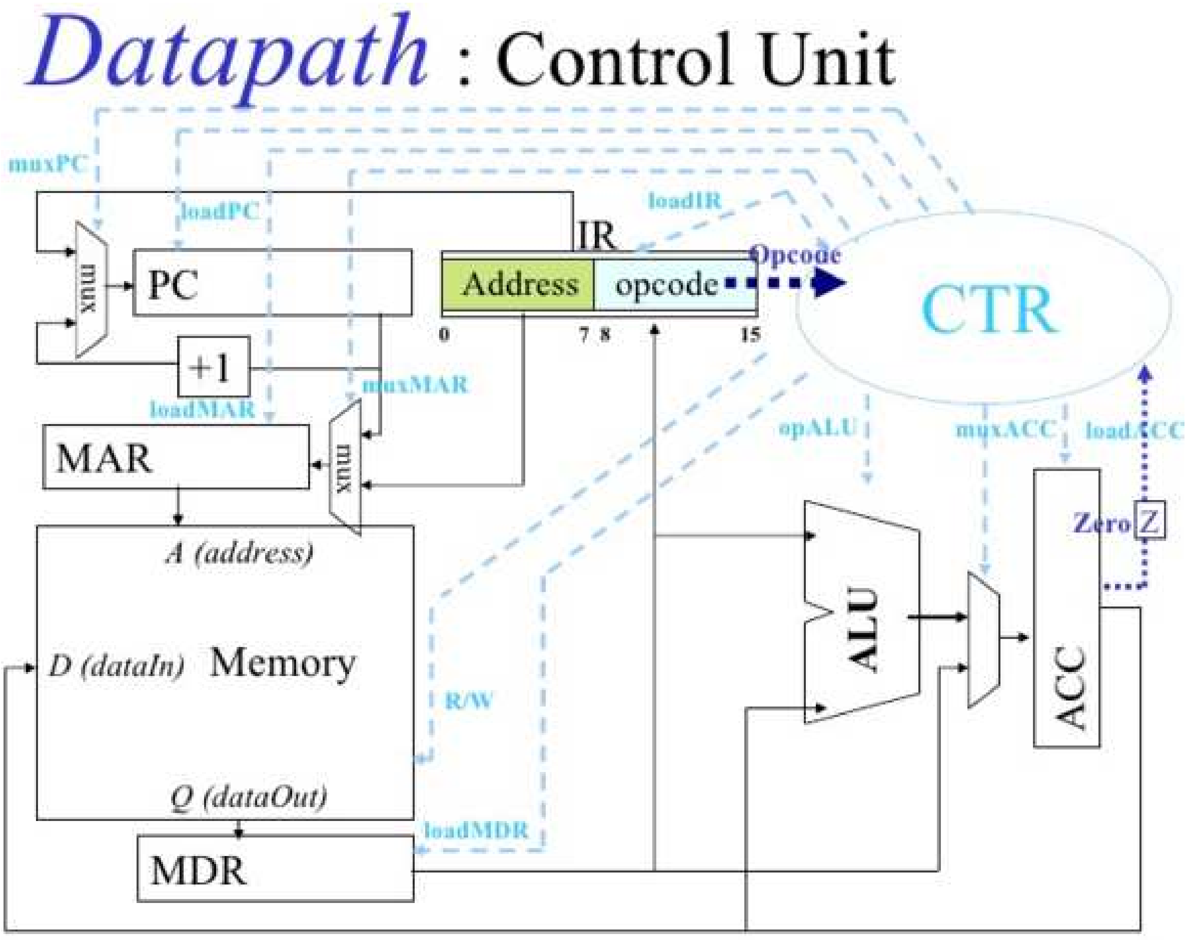
The microcontroller designed in this assignment is a simplified version of a microcontroller. Specifically, four simplifications are considered as follows:

1) No off-chip memory: The instructions of the program are assumed to be in the cache.
2) The programs consist of valid instructions ONLY, i.e., you do not have to perform error checking to detect bad instructions
3) No overflow detection is required.
4) The CPU has a synchronous reset rst. That resets all registers to zero.

5.1 Top-level module:

```
module proj1(
        clk,
        rst,
         MemRW_IO,
         MemAddr_IO,
        MemD_IO
                    );

        input clk;
        input rst;
         output MemRW_IO;
         output [7:0]MemAddr_IO;
        output [15:0]MemD_IO;
```

CPU Schematic



Here are the modules that needs to be coded up

All instructions are assumed to present in a memory

Module 1
The memory module:

```
module  ram(
        we,
        d,
        q,
        addr
            );
```

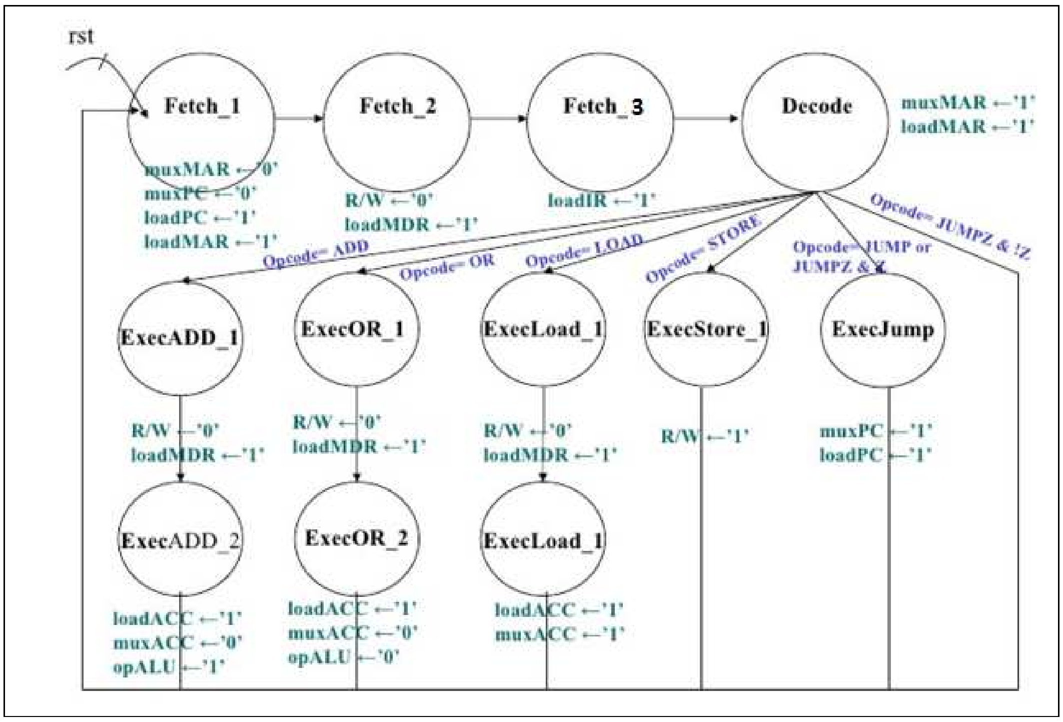
We => 1 bit read / write enable
D => 16 bit data input
Q => 16 bit data output
Addr => 8 bit input address

Module 2
Alu module

```
module alu(
        A,
        B,
        opALU,
        Rout
);
```

A => 16 bit input 1
B => 16 bit input 2
opALU => 1 bit input
    1 A + B
    0 A ^ B
Rout => 16 bit output

Module 3
Controller



```
module ctr (
	clk,
	rst,
	 zflag,
	opcode,
	muxPC,
	muxMAR,
	muxACC,
	loadMAR,
	loadPC,
	loadACC,
	loadMDR,
	loadIR,
	opALU,
	MemRW
);

	input clk;
	input rst;
	input zflag;
	input [7:0]opcode;
	output reg muxPC;
	output reg muxMAR;
	output reg muxACC;
	output reg loadMAR;
	output reg loadPC;
	output reg loadACC;
	output reg loadMDR;
	output reg loadIR;
	output reg opALU;
	output reg MemRW;



//These opcode representation need to be followed for proper operation
parameter op_add=8'b001;
parameter op_or= 8'b010;
parameter op_load=8'b011;
parameter op_store=8'b100;
parameter op_jump=8'b101;
parameter op_jumpz=8'b110;
```

we move through the states at each clock cycle. There are only two exceptions. At decode we have to see what is the opcode and go to the next state accordingly.

At Jumpz you have to look at the zeroflag. If the flag is high, we have to execute to go to the exec jump state or go to fetch_1 state.

When at each of the state you will have to set all the appropriate outputs as shown in the finite state machine.

Module 4
Register bank

```
module registers(
	clk,
	rst,
	PC_reg,
	PC_next,
	IR_reg,
	IR_next,
	ACC_reg,
	ACC_next,
	MDR_reg,
	MDR_next,
	MAR_reg,
	MAR_next,
	Zflag_reg,
	zflag_next
			);

input wire clk;
input wire rst;
output reg  [7:0]PC_reg;
input wire  [7:0]PC_next;

output reg  [15:0]IR_reg;
input wire  [15:0]IR_next;

output reg  [15:0]ACC_reg;
input wire  [15:0]ACC_next;

output reg  [15:0]MDR_reg;
input wire  [15:0]MDR_next;

output reg  [7:0]MAR_reg;
input wire  [7:0]MAR_next;

output reg Zflag_reg;
input wire zflag_next;
```

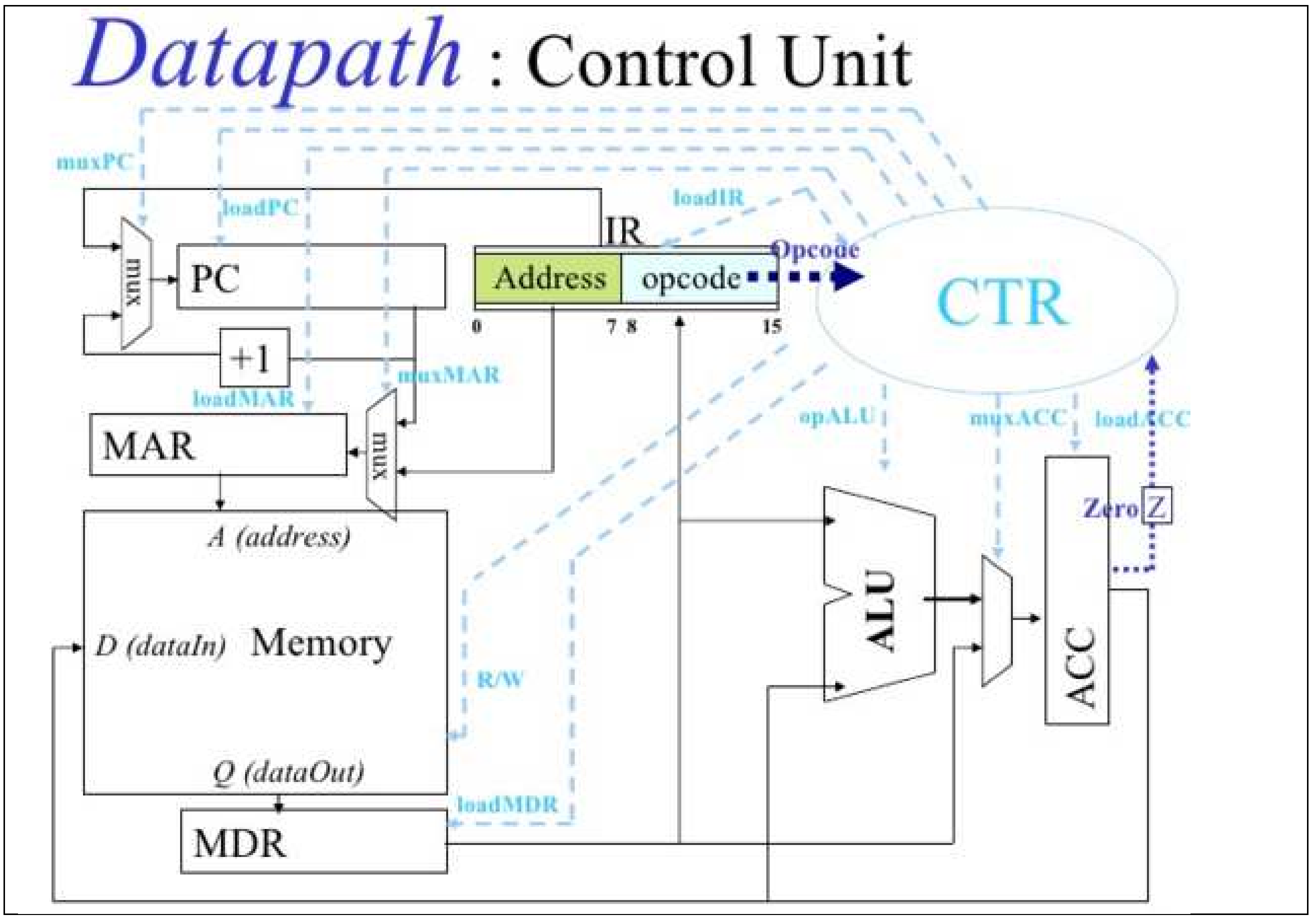
This is a very simple module. At reset set all registers to zero. At all other clocks cycles, All it does is at each rising edge of clock, it grabs the next value and stores it in the registers.

Module 5
Data path: In this module the next values are generated for all the registers and the singles to drive all the muxes.

```
module datapath(
        clk,
        rst,
        muxPC,
        muxMAR,
        muxACC,
        loadMAR,
        loadPC,
        loadACC,
        loadMDR,
        loadIR,
        opALU,
                zflag,
                opcode,
        MemAddr,
        MemD,
        MemQ
                  );

       input clk;
       input  rst;
       input  muxPC;
       input  muxMAR;
       input  muxACC;
       input  loadMAR;
       input  loadPC;
       input  loadACC;
       input  loadMDR;
       input  loadIR;
       input  opALU;
       output   zflag;
       output   [7:0]opcode;
       output   [7:0]MemAddr;
       output   [15:0]MemD;
       input   [15:0]MemQ;

reg  [7:0]PC_next;
wire  [15:0]IR_next;
reg  [15:0]ACC_next;
```

```
wire  [15:0]MDR_next;
reg  [7:0]MAR_next;
reg zflag_next;

wire  [7:0]PC_reg;
wire  [15:0]IR_reg;
wire  [15:0]ACC_reg;
wire  [15:0]MDR_reg;
wire  [7:0]MAR_reg;
wire zflag_reg;

wire  [15:0]ALU_out;
```

//one instance of ALU
// one instance of register.



//code to generate
[7:0]PC_next;
Only change if loadpc is enabled.
Mux pc decides between pc+1 or branch address
Reset address is 0, Hence nothing for the datapath to do at reset.

[15:0]IR_next;
Gets value of mdr_reg if loadir is set

[15:0]ACC_next;
Only change when loaddacc is enabled.
Muxacc decides between mdr_reg and alu out

[15:0]MDR_next;
Gets value from memeory, if load mdr is set

[7:0]MAR_next;
Only change if loadmar is enabled.
Mux mar decides between pcreg or IR[15:8]reg

zflag_next;
Decide based on the content of acc_reg

//needs to generate the following outputs
//set this outputs based on the registered value and not the next value to prevent glitches.

```
  output   zflag; => based on ACC reg
  output   [7:0]opcode; => based on IR_reg
  output   [7:0]MemAddr => Same as MAR_reg
  output   [15:0]MemD => Same as ACC reg
```

Module 6
High level module

```
module proj1(
        clk,
        rst,
         MemRW_IO,
         MemAddr_IO,
        MemD_IO
                    );

        input clk;
        input rst;
```

```
            output MemRW_IO;
            output [7:0]MemAddr_IO;
           output [15:0]MemD_IO;
```

//one instance of memory
//one instance of controller
//one instance of datapath1

//these are just to observe the signals.
assign MemAddr_IO = MemAddr;
assign MemD_IO = MemD;
assign MemRW_IO = MemRW;

The program to be loaded in to memory

```
@0000 0B03    // LOAD  11   acc = 0010
@0001 0C01    // ADD   12   acc = 0001 + 0010 = 0011
@0002 0A02    // XOR   10   acc = 0011 ^ 1000 = 1011
@0003 0D04    // STORE 13   Store 1011 to 0x00d
@0004 0606    // JUMPZ 6
@0005 0D03    // LOAD  13
@0006 0605    // JUMP  6
@0007 0904    // STORE 13
@0008 0000
@0009 0000
@000a 0008    //A data = 8
@000b 0002    //B data = 2
@000c 0001    //C data = 1
@000d 0000    //Store value
```

Testbench snippet

```
always
     #5  clk =  !clk;

initial begin
clk=1'b0;
rst=1'b1;
$readmemh("memory.list", proj1_tb.dut.ram_ins.mem256x16);
#20 rst=1'b0;
#435
$display("Final value\n");
$display("0x00d %d\n",proj1_tb.dut.ram_ins.mem256x16[16'h000d]);
$finish;
end
```