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.

**All code will be scanned through MOSS system ( <https://theory.stanford.edu/~aiken/moss/> ) to detect plagarism. Please feel free to discuss. But don’t look at anyone one’s code or show your code to anyone one either. If any plagarism is detected, strict action will be taken. You have been warned.**

**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. Design: simplified CPU**

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

1. No off-chip memory: The instructions of the program are assumed to be in the memory.
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. Design: Enhance CPU**

1) Once the design is completed, verify that the design works as expected. The memory file is provided, along with the expected output (**Final\_values.txt**).

2) Opcode 9 needs to implement a multiplication function as implemented in homework 6. Add four states Mul0 Mul1 Mul2 and Mul3 to accommodate the 4 states in HW6. Add more states as needed. You have to incorporate the HW6 multiplier. If you were unable to get yours to work. Sample code will be provided.

**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 and include the provided test bench. It may use the *‘include* directive to include other files, if you wish, but they must also be submitted with Wolfware. In addition, a second program will be used to test your code that will not be provided.

Instructions set:

0x01 ADD

0x02 OR

0x09 MULL

0x0A Neg

0x3 JUMP

0x4 JUMPZ

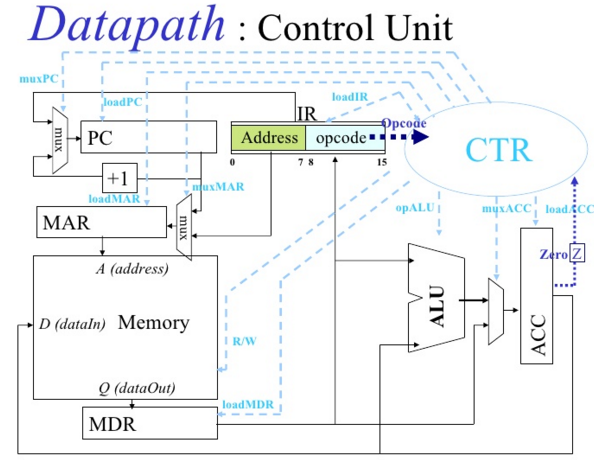
0x5 LOAD

0x6 STORE

**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, 2 bit input, 0 = or, 1 = a+b, 2 = Mul, 3 = negate

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 suggested opcode representation for proper operation

parameter op\_add=8'b001;

parameter op\_or= 8'b010;

parameter op\_jump=8'b011;

parameter op\_jumpz=8'b100;

parameter op\_load=8'b101;

parameter op\_store=8'b110;

parameter op\_mull=8'b1001;

parameter op\_neg=8'b1010;

MUL will be similar to ADD. You will have mul\_1 and mul\_2 to get the two parameters.

Then you will have mul\_3 , 4, 5 , 6 that would the 4 state machine states in HW6.

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 for verification**

|  |
| --- |
| @0000 0B05 // LOAD 11 acc = 0010  @0001 0C01 // ADD 12 acc = 0001 + 0010 = 0011  @0002 0A02 // OR 10 acc = 0011 | 0100 = 0111  @0003 0d06 // STORE 13 Store 0111 to 0x000d  @0004 0d09 // MULL 13 acc = acc \* 0x7 = 49 = 110001  @0005 0D0A // NEG acc = 1110 => ignore address.  @0006 0603 // JUMPZ 6 => alu not zero so nothing happens.  @0007 0e06 // STORE 14  @0008 0804 // JMP 08 // this will halt the code.  @0009 0000  @000a 0004 //A data = 4  @000b 0002 //B data = 2  @000c 0001 //C data = 1  @000d 0000 //Store value  @000e 0000 //Store value => all goes well location shoudl have 0xE |

**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("0x000e %d\n",proj1\_tb.dut.ram\_ins.mem256x16[16'h000e]);  $finish;  end |