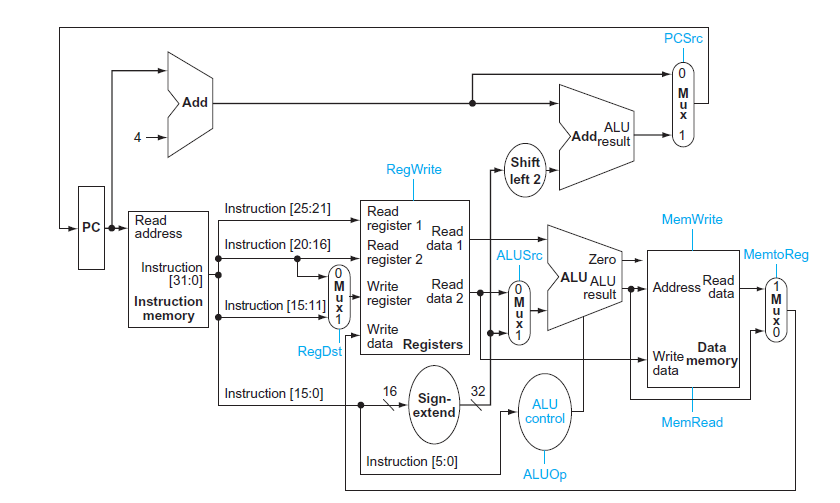
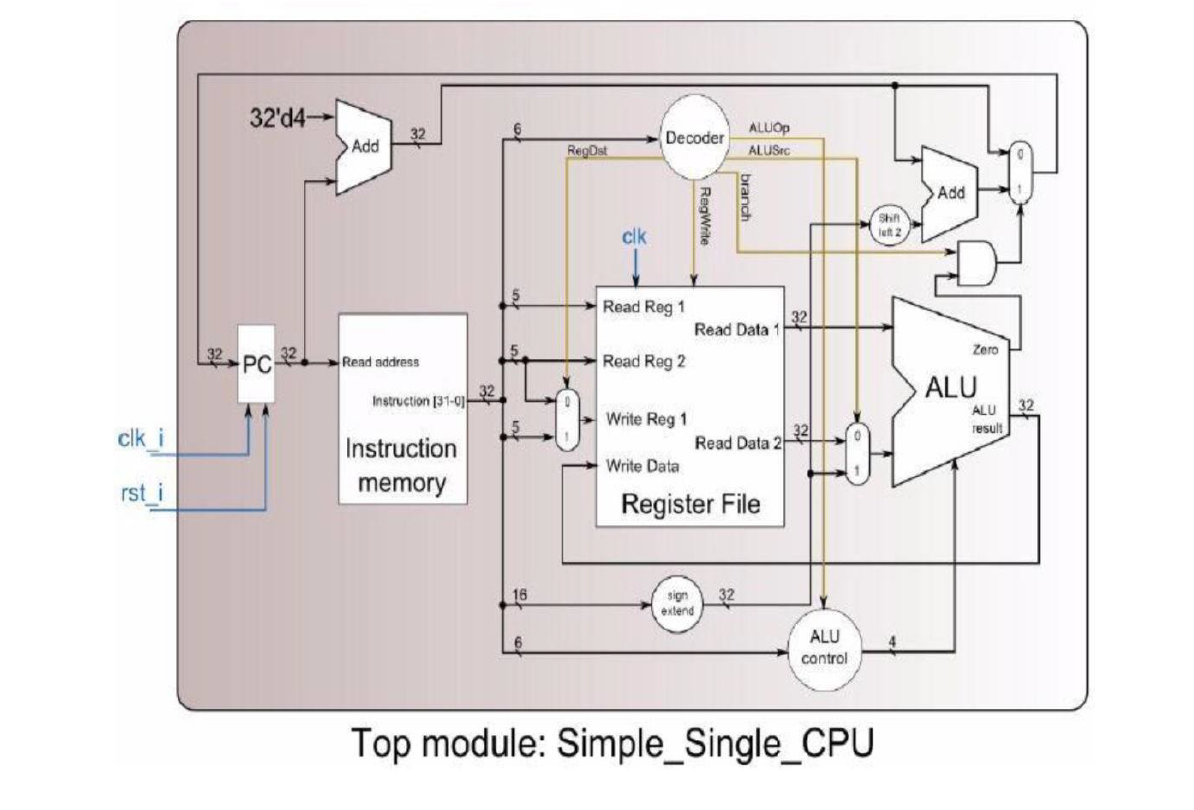
**Computer Organization**

**Architecture diagrams:**

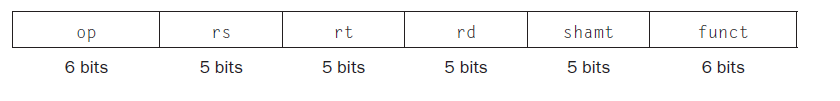
**根據課本完整的電路設計與作業提供的電路:**



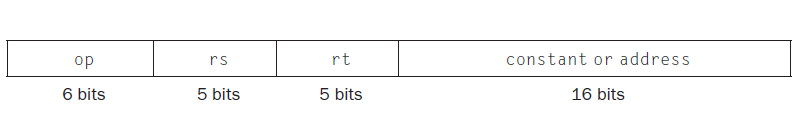
****

**我們可以知道這一次我們實踐的僅僅包括4個Stage，分別是IF，ID，EX，WB而對於memory的使用(MEM)這個stage在這邊事先不考慮的，因此ALU的結果只接傳給要被寫入的Reg。**

**整個跑的流程是，PC把要被Decoder的地址傳出，傳出的值我設為pc\_out\_o，接著這個值會跑去兩個地方，一個是去decoder被解析，一個是去PC\_source去得到下一個指令的PC位置，decoder的話，一開始會先從地址抓出32bits的指令，先著把指令根據課本與目前作業用到的指令來看，可以粗略區分成R-type:**

****

**與I-type:**

****

**這兩個Type最大的區別就是，R-type處理的指令，會用到三個Register，並且有op跟funct同時與ALU\_control做溝通，相反的，I-type處理的指令只有兩個Register以及一個16bits的常數，ALU的溝通只能依靠op的部分。抓好指令後，Decoder要做得就是分析這個指令並且輸出給RegDst，RegWrite，branch，ALUop，ALUsrc這幾個mux需要的控制信號，個別來分析，Regdst只要是看是不是R-type，是的話就要抓Mux的1資料(因為選擇1資料裡面包含的是指令[15:11])。RegWrite則是要看這個指令需不需要最後WB的stage來把值存回某個register，branch為1時，如果Zero也是1，就會執行jump，而ALUOP則是Decoder用來告訴ALU\_CRL這是什麼指令，並且委由ALU\_CRL傳送實際ALU要做的指令，最後ALUsrc則是告訴ALU除了吃Data1以外，Data2的來源如果是來自最後16位則輸出1，否則如果是來自Reg2就輸出0。**

**而我們要實作的有add，sub，slt，or，and，beq，addi，slti，其中前面五個使用的是R-type，後面三個使用的是I-type。**

**Decoder分析好後，接著RF就會根據讀到的Register地址來輸出這個地址的值，以及在最後處理寫回Register的部分。SE則是把最後16位的數值變成32位，要注意的是，如果第16位sign bit是0，表示這個數是正數，因此左半邊就加16個0即可，然而如果第16位sign bit是1，表示這個數是負數，負數的話往左邊擴展的bits就要全部填入1。**

**ALUCrtl得到最後6位的funct及ALUOP以後就要告訴ALU要做什麼事，以這個實驗來看的話，因為R-type都會有funct對應的值，所以ALUCrtl處理R-type可以直接看funct的值來做判斷，相反的，I-type因為最後十六位要看成一整個數值，因此I-type的opcode就會經由Decoder變成對應的ALUOP值，因為R-type指令有對應的ALU指令，所以我們只看最後三個指令，beq，addi，slti，beq因為有用到Zero輸出1的條件才跳，所以我們ALU要執行的就是減法看看結果是不是0，而addi就只是add一個數字而已，所以ALU要做的是add，而slti只是拿一個數字來比較，因此ALU要做的只是slt。最後ALU輸出的結果再跑回去RF，如果有要寫入站存器就寫入，沒有的話，這一整個single cycle就結束運算了。**

**接著再回去討論PC輸出往上跑去adder的地方，第一個碰到的adder是輸出PC+4的加法器，就是當沒有用到jump指令的時候，下一個輸入指令的地址就是PC+4，而這個PC+4會在往下碰到一個adder，把最後16位經過擴展成32位的offset相加，如果有確定要使用beq而且兩值經過sub結果為0，就會選擇PC+4+offset，反之以任何一項不滿足，就只會使用PC+4當作下一個指令的地址。**

**以上大致就是一個指令跑在這一整個不包含memory的CPU過程。**

**Hardware module analysis:**

1. **PC:**

**誠上面所述，PC僅僅是作為下個指令地址的輸入與把指令地址輸出的一個中間元件。**

**Code:**

**module ProgramCounter(**

**clk\_i,**

**rst\_i,**

**pc\_in\_i,**

**pc\_out\_o**

**);**

**//I/O ports**

**input clk\_i;**

**input rst\_i;**

**input [32-1:0] pc\_in\_i;**

**output [32-1:0] pc\_out\_o;**

**//Internal Signals**

**reg [32-1:0] pc\_out\_o;**

**//Parameter**

**//Main function**

**always @(posedge clk\_i) begin**

**if(~rst\_i)**

**pc\_out\_o <= 0;**

**else**

**pc\_out\_o <= pc\_in\_i;**

**end**

**endmodule**

**Features:**

**考慮到實際電路不可能一直運行，因此只有在clock由低電位往高電位跑時才進行把輸入的指令地址輸出。**

1. **IM:**

**IM主要就是負責把指令地址輸入後轉成該地址下的指令輸出給後面元件使用。**

**Code:**

**module Instr\_Memory(**

**pc\_addr\_i,**

**instr\_o**

**);**

**//I/O ports**

**input [32-1:0] pc\_addr\_i;**

**output [32-1:0] instr\_o;**

**//Internal Signals**

**reg [32-1:0] instr\_o;**

**integer i;**

**//32 words Memory**

**reg [32-1:0] Instr\_Mem [0:32-1];**

**//Parameter**

**//Main function**

**always @(pc\_addr\_i) begin**

**instr\_o = Instr\_Mem[pc\_addr\_i/4];**

**end**

**//Initial Memory Contents**

**initial begin**

**for ( i=0; i<32; i=i+1 )**

**Instr\_Mem[i] = 32'b0;**

**$readmemb("CO\_P2\_test\_data2.txt", Instr\_Mem); //Read instruction from "CO\_P2\_test\_data1.txt"**

**end**

**endmodule**

**Features:**

**可以看到最一開始的時候我們已經把所有指令放到對應的地址去了，所以當吃到指令地址後，會把指令地址/4直接得到array裡的某一項當作指令(因為MIPS一個單位是32bits=4bytes)。**

1. **Decoder:**

**Decoder主要把吃到的指令分析，並且輸出Mux接收的訊號，讓後面的運算可以得到正確的安排。**

**Code:**

**module Decoder(**

**instr\_op\_i,**

**RegWrite\_o,**

**ALU\_op\_o,**

**ALUSrc\_o,**

**RegDst\_o,**

**Branch\_o**

**);**

**//I/O ports**

**input [6-1:0] instr\_op\_i;**

**output RegWrite\_o;**

**output [3-1:0] ALU\_op\_o;**

**output ALUSrc\_o;**

**output RegDst\_o;**

**output Branch\_o;**

**//Internal Signals**

**//Parameter**

**wire RegWrite\_o;**

**wire [3-1:0] ALU\_op\_o;**

**wire ALUSrc\_o;**

**wire RegDst\_o;**

**wire Branch\_o;**

**wire rtype;**

**wire beq;**

**wire addi;**

**wire slti;**

**wire ori;**

**wire lui;**

**//Main function**

**//process ALUop field**

**assign rtype = (instr\_op\_i==0);**

**assign beq = (instr\_op\_i==4);**

**assign addi = (instr\_op\_i==8);**

**assign slti = (instr\_op\_i==10);**

**//process output signal**

**assign RegWrite\_o = ((rtype | addi )| slti);**

**assign ALUSrc\_o = (addi | slti );//1 to use original 16bits**

**assign RegDst\_o = rtype;//1 for rd**

**assign Branch\_o = beq;**

**//only use when there is other command than R-type**

**//by book==> rtype-->100 beq-->001 and addi subi doesn't influence the add or sub operation but we still only can use ALU\_op\_o as the command to alu**

**//so addi-->ALU need add-->0010-->010**

**//so slti-->ALU need slt-->0111-->111**

**//so beq-->ALU need sub-->0110-->110**

**assign ALU\_op\_o[2] = (beq | slti );**

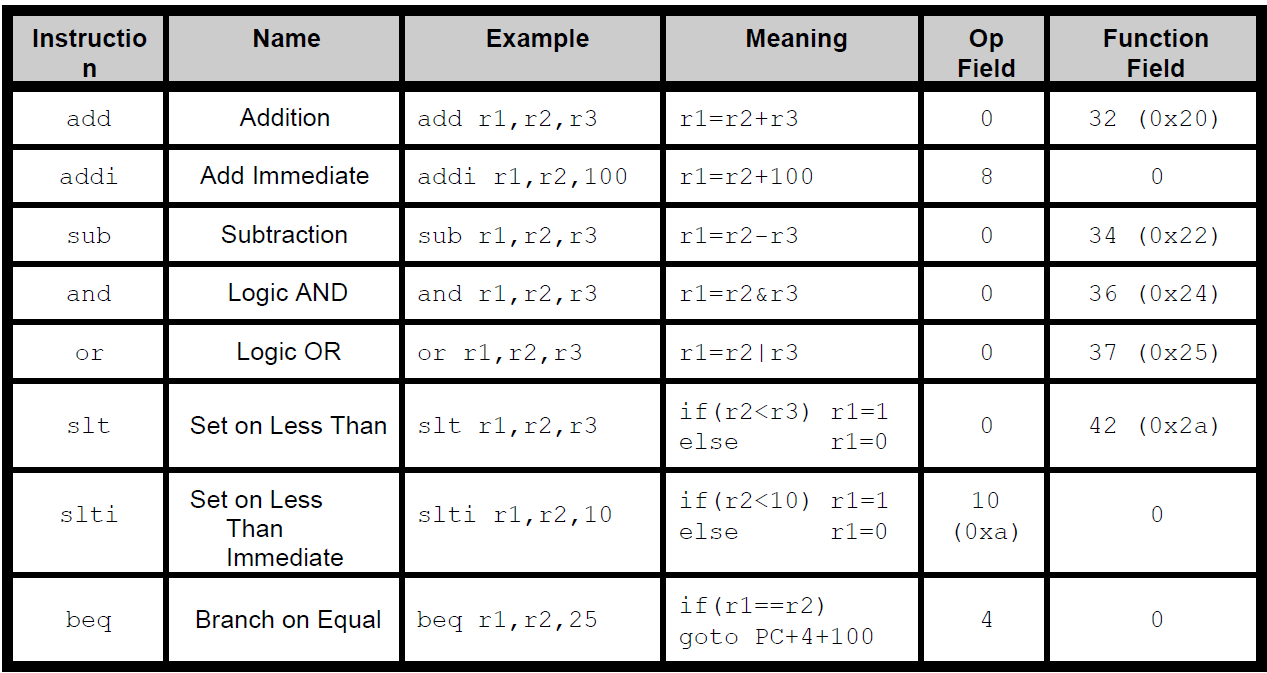
**assign ALU\_op\_o[1] = ((beq | addi )| slti);**

**assign ALU\_op\_o[0] = slti;**

**endmodule**

**Features:**

**根據:**

****

可以根據Op Field大致分成4種指令，R-type，addi，slti，beq，分成這4種指令之後，再根據各自的需求去分配各個Mux訊號的輸出值，比較需要注意的是，ALUOP我在這邊設計的方式是使用與ALU到時候需要用的指令化簡最後三個bit得到的結果，像是beq會用到sub，sub在ALU的指令是0110，那麼我在這邊就會讓ALUOP的值變成110，方便我等一下輸出指令時寫法更簡單。所以，ALUop的值:

R-type:000

Beq:110

Addi:010

Slti:111

1. **RF:**

**RF主要負責把吃到的Register地址轉成地址下的值輸出，並且在最後WBstage時，如果有要寫入Register的話就負責執行。**

**Code:**

**module Reg\_File(**

**clk\_i,**

**rst\_i,**

**RSaddr\_i,**

**RTaddr\_i,**

**RDaddr\_i,**

**RDdata\_i,**

**RegWrite\_i,**

**RSdata\_o,**

**RTdata\_o**

**);**

**//I/O ports**

**input clk\_i;**

**input rst\_i;**

**input RegWrite\_i;**

**input [5-1:0] RSaddr\_i;**

**input [5-1:0] RTaddr\_i;**

**input [5-1:0] RDaddr\_i;**

**input [32-1:0] RDdata\_i;**

**output [32-1:0] RSdata\_o;**

**output [32-1:0] RTdata\_o;**

**//Internal signals/registers**

**reg signed [32-1:0] Reg\_File [0:32-1]; //32 word registers**

**wire [32-1:0] RSdata\_o;**

**wire [32-1:0] RTdata\_o;**

**//Read the data**

**assign RSdata\_o = Reg\_File[RSaddr\_i] ;**

**assign RTdata\_o = Reg\_File[RTaddr\_i] ;**

**//Writing data when postive edge clk\_i and RegWrite\_i was set.**

**always @( posedge rst\_i or posedge clk\_i)**

**begin**

**// $display( "count%d\n",RegWrite\_i);**

**if(rst\_i == 0) begin**

**Reg\_File[0] <= 0; Reg\_File[1] <= 0; Reg\_File[2] <= 0; Reg\_File[3] <= 0;**

**Reg\_File[4] <= 0; Reg\_File[5] <= 0; Reg\_File[6] <= 0; Reg\_File[7] <= 0;**

**Reg\_File[8] <= 0; Reg\_File[9] <= 0; Reg\_File[10] <= 0; Reg\_File[11] <= 0;**

**Reg\_File[12] <= 0; Reg\_File[13] <= 0; Reg\_File[14] <= 0; Reg\_File[15] <= 0;**

**Reg\_File[16] <= 0; Reg\_File[17] <= 0; Reg\_File[18] <= 0; Reg\_File[19] <= 0;**

**Reg\_File[20] <= 0; Reg\_File[21] <= 0; Reg\_File[22] <= 0; Reg\_File[23] <= 0;**

**Reg\_File[24] <= 0; Reg\_File[25] <= 0; Reg\_File[26] <= 0; Reg\_File[27] <= 0;**

**Reg\_File[28] <= 0; Reg\_File[29] <= 0; Reg\_File[30] <= 0; Reg\_File[31] <= 0;**

**end**

**else**

**begin**

**if(RegWrite\_i)**

**begin**

**Reg\_File[RDaddr\_i] <= RDdata\_i;**

**end**

**end**

**end**

**endmodule**

**Features:**

**在最一開始rst訊號為0時，所有的Register會先歸0，接著配合PC是在asserted輸出值，RF也只會在asserted去寫入Register如果有要寫入的訊號為1時。**

1. **SE:**

**SE主要負責把16位的資料轉成32位，要注意的是當sign bit為1時，擴展後左半邊的bit都要是1，如果0，則擴展後填入的左半邊都是0，之所以要這樣主要是因為sign bit為0，表示是正數，那麼當我們往左擴展更高位數字時，對於原本的數字來說這些擴展的bit並不會是原本數字組成所需要的bit，因此只能輸入0，然而對於負數來說，往左邊擴展bits時，因為二補數得到的方式是所有bits翻轉再+1，因此對於負數來說，可以想成原本是正數，正數擴展是填入0，擴展完後變負數，所有bits由0🡪1，1🡪0的翻轉最後再加1，這個結果再+1，就是負數擴展的結果，可以發現這樣的過程中其實相當於就是往左擴展填入的bit都為1，因為對於正數來說這些更高位的bit都是0，因此變成負數時，更高位的bits只能都變成1。**

**Code:**

**module Sign\_Extend(**

**data\_i,**

**data\_o**

**);**

**//I/O ports**

**input [16-1:0] data\_i;**

**output [32-1:0] data\_o;**

**//Internal Signals**

**reg [32-1:0] data\_o;**

**//Sign extended**

**always @(data\_i) begin**

**data\_o[15:0] <= data\_i[15:0];**

**//positive**

**if (data\_i[15]==0)**

**data\_o[31:16] <= 16'b0;**

**//negative**

**else**

**data\_o[31:16] <= ~(16'b0);**

**end**

**endmodule**

**Features:**

**我的寫法就是直接創造32bits的空間，接著把右半邊16bits直接填入原先的16bits，接著判斷原先16bits的sign bit也就是[15]是正還負，是正數的話就把左半邊16bits填入0，否則就填入~(16’b0)，也就是16個bit都是0的值把每個位元從0🡪1，因此可以得到16個bit為1的值，看起來有點麻煩但是卻只能這樣做，詳細問題在”Probles you met and solutions”中有提到。**

1. **ALU\_ctrl:**

**ALU\_ctrl主要是當作Decoder與ALU之間的橋樑，負責把Decoder希望ALU做的事情藉由ALU\_ctrl來達成。**

**Code:**

**module ALU\_Ctrl(**

**funct\_i,**

**ALUOp\_i,**

**ALUCtrl\_o**

**);**

**//I/O ports**

**input [6-1:0] funct\_i;**

**input [3-1:0] ALUOp\_i;**

**output [4-1:0] ALUCtrl\_o;**

**//Internal Signals**

**reg [4-1:0] ALUCtrl\_o;**

**//Parameter**

**//Select exact operation**

**always @(funct\_i,ALUOp\_i) begin**

**case(ALUOp\_i)**

**//0: ALUCtrl\_o <= 4'b1000;//addi or beq or slti**

**0:**

**case(funct\_i)**

**32: ALUCtrl\_o <= 4'b0010;//add**

**34: ALUCtrl\_o <= 4'b0110;//sub**

**36: ALUCtrl\_o <= 4'b0000;//and**

**37: ALUCtrl\_o <= 4'b0001;//or**

**42: ALUCtrl\_o <= 4'b0111;//slt**

**endcase**

**default: //process not R-type**

**begin**

**ALUCtrl\_o[3] <= 0;**

**ALUCtrl\_o[2:0] <= ALUOp\_i[2:0];**

**end**

**endcase**

**end**

**endmodule**

**Features:**

**可以看到，因為上面我說分四大類是根據opcode，並且對應輸出不一樣的ALUOP，故一開始先判斷是不是R-type🡺ALUOP=0，接著，因為我在設計上不是R-type的指令對應ALUOP就是ALU動作指令的後三位，因此可以直接把輸出的四位元最高位放0，後三位直接放入ALUOP的值。**

1. **ALU**

**在這次作業中可以進行add,sub,or,and,nor,slt的動作，詳細在上次結報中已寫過，不贅述。**

1. **MUX**

**在這個電路裡，會用到很多選擇器MUX，主要是為了配合指令而有多種不同選擇。實踐方式很簡單，如果訊號是1，輸出就把存在MUX1的資料丟出，否則就丟出存在MUX0的資料。**

**Code:**

**module MUX\_2to1(**

**data0\_i,**

**data1\_i,**

**select\_i,//and gate**

**data\_o**

**);**

**parameter size = 0;**

**//I/O ports**

**input [size-1:0] data0\_i;**

**input [size-1:0] data1\_i;**

**input select\_i;**

**output [size-1:0] data\_o;**

**//Internal Signals**

**reg [size-1:0] data\_o;**

**//Main function**

**always @(data0\_i or data1\_i or select\_i)**

**begin**

**if (select\_i==0) begin**

**data\_o <= data0\_i;**

**end**

**else begin**

**data\_o <= data1\_i;**

**end**

**end**

**endmodule**

**Features:**

**當輸入訊號select為1，丟出data1，否則丟出data0。**

1. **Adder**

**使用上次ALU實踐的add\_ALU模組，不贅述。**

1. **Shifter**

**當指令中含有地址的值時，會因為要有更多空間，而又MIPS最小單位是4byte為一個單位，所以這些指令的地址量就會是以4byte為一個單位，故最後在做實際地址的運算時，必須把地址左移2位，才能拿到實際的值。**

**Code:**

**module Shift\_Left\_Two\_32(**

**data\_i,**

**data\_o**

**);**

**//I/O ports**

**input [32-1:0] data\_i;**

**output [32-1:0] data\_o;**

**//shift left 2**

**assign data\_o=data\_i<<2;**

**endmodule**

**Features:**

**可以看到就直接使用<<2就可以了，或是，因為我們知道data\_i在這個作業中是由16bits擴展得到的，所以不會oerflow，故也可以改寫成:**

**assign data\_o[32-1:2]=data\_i[30-1:0];**

**assign data\_o[1:0]=0;**

**這樣子也可以得到一樣的效果。**

1. **CPU**

**CPU主要就是把上面的所有元件組成最上面的電路圖，為了不搞混變數名稱，所以我命名變數的原則主要是以最後\_i當作前面名稱的輸入，\_o當作前面名稱的輸出。**

**Code:**

**module Simple\_Single\_CPU(**

**clk\_i,**

**rst\_i/\***

**,temp**

**,temp2**

**,temp3**

**,temp4**

**,temp5**

**,temp6**

**,temp7**

**,temp8\*/**

**);**

**//I/O port**

**input clk\_i;**

**input rst\_i;**

**/\***

**output [31:0]temp;**

**output [31:0]temp2;**

**output [31:0]temp3;**

**output [31:0]temp4;**

**output [31:0]temp5;**

**output [31:0]temp6;**

**output [31:0]temp7;**

**output [31:0]temp8;**

**\*/**

**//Internal Signles**

**//fetch-->decoder/register-->ALU/branch-->memory-->write back**

**// wire for PC**

**wire [31:0] pc\_in\_i;**

**wire [31:0] pc\_out\_o;//for adder1 and IM**

**// wire for adder PC+4**

**wire [31:0] sum\_o\_add1;**

**// wire for IM**

**wire [31:0] instr\_o;**

**// wire for MUX\_write\_reg**

**wire [4:0] data\_o\_write\_reg;**

**// wire for RF**

**wire [31:0] RSdata\_o;**

**wire [31:0] RTdata\_o;**

**// wire for ALU**

**wire [31:0] result\_o;**

**wire zero\_o;**

**// wire for SE**

**wire [31:0] data\_o\_SE;**

**// wire for Mux\_\_alu\_src**

**wire [31:0] Src\_data\_o;**

**// wire for Decoder**

**wire RegWrite\_o;**

**wire [2:0] ALU\_op\_o;**

**wire ALUSrc\_o;**

**wire RegDst\_o;**

**wire Branch\_o;**

**// wire for ALUCtrl**

**wire [3:0] ALUCtrl\_o;**

**//Internal Signles**

**//for branch**

**wire and\_out;**

**and AND(and\_out,Branch\_o,zero\_o);**

**// wire for shift-left**

**wire [31:0] data\_o\_shift;**

**// wire for adder address offset**

**wire [31:0] sum\_o\_adder2;**

**/\***

**assign temp=ALU.result;**

**assign temp2=ALU.src1;**

**assign temp3=ALU.src2;**

**assign temp4=ALU.zero1;**

**assign temp5=instr\_o[15:0];**

**assign temp6=data\_o\_SE;**

**assign temp7=ALUCtrl\_o;**

**assign temp8= and\_out;**

**\*/**

**//assign temp=RegWrite\_o;**

**//Greate componentes**

**ProgramCounter PC(**

**.clk\_i(clk\_i),**

**.rst\_i (rst\_i),**

**.pc\_in\_i(pc\_in\_i) ,**

**.pc\_out\_o(pc\_out\_o)**

**);**

**Adder Adder1(**

**.src1\_i(32'd4),**

**.src2\_i(pc\_out\_o),**

**.sum\_o(sum\_o\_add1)**

**);**

**Instr\_Memory IM(**

**.pc\_addr\_i(pc\_out\_o),**

**.instr\_o(instr\_o)**

**);**

**MUX\_2to1 #(.size(5)) Mux\_Write\_Reg(**

**.data0\_i(instr\_o[20:16]),**

**.data1\_i(instr\_o[15:11]),**

**.select\_i(RegDst\_o),**

**.data\_o(data\_o\_write\_reg)**

**);**

**Reg\_File RF(**

**.clk\_i(clk\_i),**

**.rst\_i(rst\_i) ,**

**.RegWrite\_i (RegWrite\_o),**

**.RSaddr\_i(instr\_o[25:21]) ,//address**

**.RTaddr\_i(instr\_o[20:16]) , //address**

**.RDaddr\_i(data\_o\_write\_reg) ,**

**.RDdata\_i(result\_o) ,**

**.RSdata\_o(RSdata\_o) ,**

**.RTdata\_o(RTdata\_o)**

**);**

**Decoder Decoder(**

**.instr\_op\_i(instr\_o[31:26]),**

**.RegWrite\_o(RegWrite\_o),**

**.ALU\_op\_o(ALU\_op\_o),**

**.ALUSrc\_o(ALUSrc\_o),**

**.RegDst\_o(RegDst\_o),**

**.Branch\_o(Branch\_o)**

**);**

**ALU\_Ctrl AC(**

**.funct\_i(instr\_o[5:0]),**

**.ALUOp\_i(ALU\_op\_o),**

**.ALUCtrl\_o(ALUCtrl\_o) //do add/subb etc**

**);**

**Sign\_Extend SE(**

**.data\_i(instr\_o[15:0]),**

**.data\_o(data\_o\_SE)**

**);**

**MUX\_2to1 #(.size(32)) Mux\_ALUSrc(**

**.data0\_i(RTdata\_o),**

**.data1\_i(data\_o\_SE),**

**.select\_i(ALUSrc\_o),**

**.data\_o(Src\_data\_o)**

**);**

**ALU ALU(**

**.clk(clk\_i),**

**.rst\_n(rst\_i),**

**.src1(RSdata\_o),**

**.src2(Src\_data\_o),**

**.ALU\_control(ALUCtrl\_o),**

**.result(result\_o),**

**.zero(zero\_o)**

**);**

**Adder Adder2(**

**.src1\_i(sum\_o\_add1),**

**.src2\_i(data\_o\_shift),**

**.sum\_o(sum\_o\_adder2)**

**);**

**Shift\_Left\_Two\_32 Shifter(**

**.data\_i(data\_o\_SE),**

**.data\_o(data\_o\_shift)**

**);**

**MUX\_2to1 #(.size(32)) Mux\_PC\_Source(**

**.data0\_i(sum\_o\_add1),**

**.data1\_i(sum\_o\_adder2),**

**.select\_i(and\_out),**

**.data\_o(pc\_in\_i)**

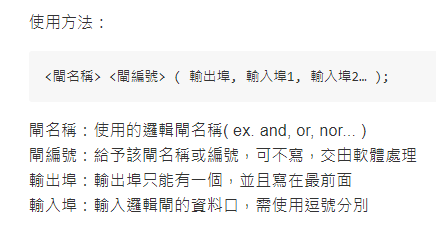
**);**

**endmodule**

**Features:**

**基本上最上面的電路元件都有了，唯獨在下個指令地址處理的route中，有一個and gate沒有在上面，根據:**

[**https://hom-wang.gitbooks.io/verilog-hdl/content/Chapter\_03.html**](https://hom-wang.gitbooks.io/verilog-hdl/content/Chapter_03.html)

****

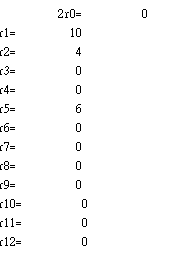
**我們可以知道Verilog這個語言有內建邏輯閘，因此and gate可以用上面的語法輕鬆寫在CPU的程式裡作為一個元件，同理，adder其實也可以!所以我在我的程式中為了表示有這兩種寫法，adder使用的是上一次ALU中我寫的add ALU，and gate則是使用內建的and gate。**

**Finished part:**

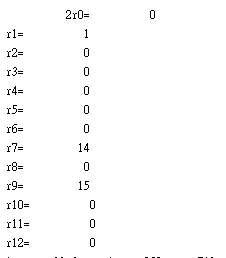
**目前來看，以這個作業來說，我完成了and，or，sub，slt，add，addi，slti，beq的指令在不考慮MEM stage的single cycle CPU裡面運行。**

**最後結果:**

**1:**

****

**2:**

****

**Problems you met and solutions:**

**Q1.PC的起始值要怎麼設定呢?**

**A1.在PC的.v檔中，**

**if(~rst\_i)**

**pc\_out\_o <= 0;**

**因此在最一開始的時候，指令地址是把0輸出，接著根據指令，傳回下一個指令地址給PC來進行一整個CPU的模擬。**

**Q2.ALUOP在課本上是用2bit，但是在這裡卻使用到了3bit?**

**A2.根據我詢問助教得到的答案，因為剛好分出了四種指令，因此可以使用one-hot encoding的方式用000，001，010，100來區別這四種指令，另外，根據我與助教交流的過程中，我大概知道了其實ALUOP傳給ALUCRL的這個部分是可以不需要的，畢竟可以直接由Decoder去做判斷要做什麼即可，也就是說ALUOP的功能不過就是把指令又多了一層的整理，要用2bit來寫入也是可以，沒有一定的標準在。**

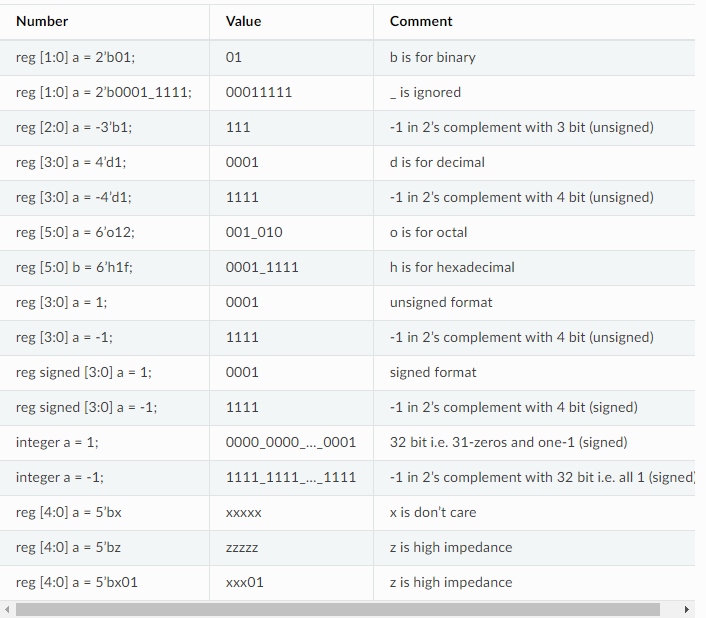
**Q3.在寫SE時，可以看到我判斷是負數後，我是這樣寫:**



**請問我右半邊的寫法可以寫成比較簡單的<=16’b1嗎?**

**A3.**

**根據:** [**https://verilogguide.readthedocs.io/en/latest/verilog/datatype.html#number-representation**](https://verilogguide.readthedocs.io/en/latest/verilog/datatype.html#number-representation)



**我們可以發現寫16’b之後應該要寫出16個bits各自的值，否則不寫的值會由高位到低位依序填入0，因此16’b1得到的是0000000000000001，而不會是1111111111111111，這是在寫硬體語言需要注意的地方，16’b1!=16個bit 1。因此並不能這樣寫，要得到16個bit為1，用~(16’b0)應為最快的方式。**

**Summary:**

**覺得這一次的作業難度最難的就是要把CPU的輸入輸出填完，其他很多元件其實都已經寫好了，剩下沒寫好的想一下也可以寫出來，另外比較困難的則是clk的部分，有時候輸出的值是錯誤的會跟clk有關，因為某些元件可能有delay的狀況，這種時候就常常會有取值錯誤的問題，這也是我並不是很喜歡寫硬體語言的原因，光是要把clk的錯誤修正就會花非常大量的時間。不過這個作業裡我覺得我對CPU更加的熟悉了!opcode，ALUop，function，ALUcontrol input這四個名詞我之前都蠻容易搞混的，但經過自己這樣設計之後，我已經可以知道opcode提供的是給decorder資訊，ALUop提供的是給ALU\_control資訊，function提供的是R-type的資訊，ALUcontrol input提供的是ALU要做什麼動作的訊號。不過我覺得收穫最大的是，我覺得我對CPU的興趣經過這次作業後變更多了，雖然我知道實際上CPU的運作一定更複雜，但是能夠對硬體的東西再次感到興趣，覺得很開心，這也是為什麼我會去找助教問ALUOP有沒有什麼標準的配置方式，因為真的覺得很好玩!不過Verilog還是不太好玩就是了，因為每次debug都要多考慮clk的問題，其實蠻累人的。**