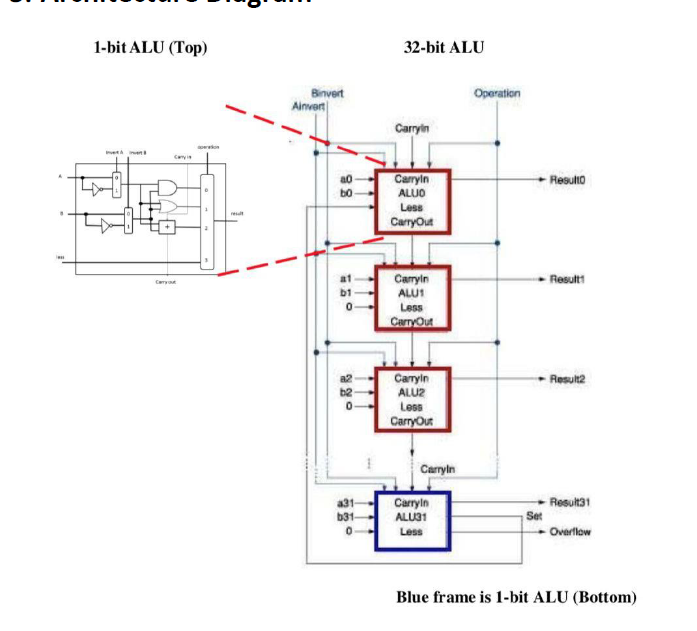
**0810740張又仁**

**Computer Organization**

**Architecture diagrams:**

****

**我先把1-bit alu(top)做完，接著再實現一個一次做出4-bit的alu，再用4-bit alu實現32bit 的add ,sub,or,and,nor跟slt，最後再把這6個電路合併到alu裡面，根據operation選擇需要的結果!**

**Hardware module analysis:**

**Alu\_top:**

**我認為這個最基本電路只是根據指定要求去算出單個bit的結果!**

**Code:**

**module alu\_top(**

**clk,**

**src1, //1 bit source 1 (input)**

**src2, //1 bit source 2 (input)**

**less, //1 bit less (input)**

**A\_invert, //1 bit A\_invert (input)**

**B\_invert, //1 bit B\_invert (input)**

**cin, //1 bit carry in (input)**

**operation, //operation (input)**

**result, //1 bit result (output)**

**cout //1 bit carry out(output)**

**//,ap**

**);**

**input clk;**

**input src1;**

**input src2;**

**input less;**

**input A\_invert;**

**input B\_invert;**

**input cin;**

**input [2-1:0] operation;**

**output result;**

**output cout;**

**//output ap;**

**wire in1,in2;**

**assign in1=(A\_invert^src1);**

**assign in2=(B\_invert^src2);**

**assign cout=((in1&in2)|(in2&cin))|(in1&cin);**

**wire op\_less;**

**assign op\_less=less;**

**wire op\_add;**

**assign op\_add=(in1^in2)^cin;**

**wire op\_or;**

**assign op\_or=in1|in2;**

**wire op\_and;**

**assign op\_and=in1&in2;**

**reg r;**

**always@(src1 or src2 or operation or op\_less or op\_add or op\_or or op\_and or clk or less ) //operation (input))**

**begin**

**if(operation==2'b11)**

**begin**

**//less**

**r=op\_less;**

**end**

**else if(operation==2'b10)**

**begin**

**//add**

**r=op\_add;**

**end**

**else if(operation==2'b01)**

**begin**

**//or**

**r=op\_or;//xor 0==>not change xor 1==>invert!**

**end**

**else//2'b00**

**begin**

**//and**

**r=op\_and;**

**end**

**end**

**//assign ap=op\_add;**

**assign result=r;**

**endmodule**

**Features:**

1. **assign in1=(A\_invert^src1);**

**assign in2=(B\_invert^src2);**

**in1 in2才是實際要做運算的輸入，可以發現我用^來代替選擇器的功能，因為我們知道x^0=x x^1=x’，所以^(xor)可以做為很好的選擇功能**

1. **assign cout=((in1&in2)|(in2&cin))|(in1&cin)**

**cout表示carry out，我對於進位最熟悉的就是把三個輸入兩兩做一次&最後再Or，這樣就可以得到cout了，雖然老師有講過更快的進位ahead(也就是只做前4個bit後面直接用第四個bit的cout來推倒!)但是，我認為目前不考慮實際電路運作的狀況，就沒有嘗試實現了!**

1. **因為總共有4個可以選擇的對象，所以我先用wire把他們的”電路”直接建立好最後進入到always裡面再讓Reg result去做選擇!我原本有考慮把前面的對象也用reg處理，但是發現結果有些問題(下面問題與解決有提到)，因此才改寫成以上分成wire代表選項跟reg代表從中選一，這樣的好處就是，reg選擇時不會出現選項還停留在以前資料的狀況，另外也比較符合實際電路!**

**Alu\_4:**

**這個模組則是要把alu\_top合併成4個為一組!**

**Code:**

**module alu\_4(**

**clk,**

**srcA, //4 bit**

**srcB, //4 bit**

**less, //1 bit**

**A\_invert, //1 bit**

**B\_invert, //1 bit**

**cin, //1 bit**

**operation, //2 bit**

**result1, //4 bit**

**cout1 //1 bit**

**//,ap**

**);**

**input clk;**

**input [4-1:0] srcA, srcB;**

**input cin, less, A\_invert, B\_invert;**

**input [2-1:0] operation;**

**output cout1;**

**output [4-1:0]result1;**

**// output [4-1:0]ap;**

**wire [4-1:0] co;**

**alu\_top a1(**

**.clk(clk)**

**,.src1(srcA[0]),**

**.src2(srcB[0]),**

**.less(less),**

**.A\_invert(A\_invert),**

**.B\_invert(B\_invert),**

**.cin(cin),**

**.operation(operation),**

**.result(result1[0]),**

**.cout(co[0])**

**//,.ap(ap[0])**

**);**

**alu\_top a2(**

**.clk(clk)**

**,.src1(srcA[1]),**

**.src2(srcB[1]),**

**.less(0),**

**.A\_invert(A\_invert),**

**.B\_invert(B\_invert),**

**.cin(co[0]),**

**.operation(operation),**

**.result(result1[1]),**

**.cout(co[1])**

**//,.ap(ap[1])**

**);**

**alu\_top a3(**

**.clk(clk)**

**,.src1(srcA[2]),**

**.src2(srcB[2]),**

**.less(0),**

**.A\_invert(A\_invert),**

**.B\_invert(B\_invert),**

**.cin(co[1]),**

**.operation(operation),**

**.result(result1[2]),**

**.cout(co[2])**

**// ,.ap(ap[2])**

**);**

**alu\_top a4(**

**.clk(clk)**

**,.src1(srcA[3]),**

**.src2(srcB[3]),**

**.less(0),**

**.A\_invert(A\_invert),**

**.B\_invert(B\_invert),**

**.cin(co[2]),**

**.operation(operation),**

**.result(result1[3]),**

**.cout(co[3])**

**//,.ap(ap[3])**

**);**

**assign cout1=co[3];**

**endmodule**

**Features:**

**可以發現只要實現1-bit的基本元件，4-bit的實現就變得很簡單!唯一要注意的就是co[x]會是第x+1元件的cin。另外，or and的實現可以直接打出來，add的實現可以根據Lab0得到!less 的實現則比較特別，因為less 是Slt用來判斷對應bit的值，而又slt只有0，1，故less可以在進去模組時直接輸入!**

**And\_32:**

**我認為這個模組沒什麼難度，直接讓最基本alu接收and指令即可!**

**Code:**

**module and\_32(**

**clk,src1,src2,result,cout,zero,over**

**);**

**input clk;**

**input [32-1:0] src1;**

**input [32-1:0] src2;**

**output [32-1:0] result;**

**output zero;**

**output cout;**

**output over;**

**wire [32-1:0] result;**

**wire [8-1:0] co;**

**reg zero;**

**wire cout;**

**reg over=0;**

**alu\_4 a41 (**

**.clk(clk)**

**,.srcA(src1[4-1:0]), //4 bit**

**.srcB(src2[4-1:0]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(1'b0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[4-1:0]), //4 bit**

**.cout1(co[0]) //1 bit**

**);**

**alu\_4 a42(**

**.clk(clk)**

**,.srcA(src1[8-1:4]), //4 bit**

**.srcB(src2[8-1:4]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(1'b0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[8-1:4]), //4 bit**

**.cout1(co[1]) //1 bit**

**);**

**alu\_4 a43(**

**.clk(clk)**

**,.srcA(src1[12-1:8]), //4 bit**

**.srcB(src2[12-1:8]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(1'b0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[12-1:8]), //4 bit**

**.cout1(co[2]) //1 bit**

**);**

**alu\_4 a44(**

**.clk(clk)**

**,.srcA(src1[16-1:12]), //4 bit**

**.srcB(src2[16-1:12]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(1'b0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[16-1:12]), //4 bit**

**.cout1(co[3]) //1 bit**

**);**

**alu\_4 a45(**

**.clk(clk)**

**,.srcA(src1[20-1:16]), //4 bit**

**.srcB(src2[20-1:16]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(1'b0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[20-1:16]), //4 bit**

**.cout1(co[4]) //1 bit**

**);**

**alu\_4 a46(**

**.clk(clk)**

**,.srcA(src1[24-1:20]), //4 bit**

**.srcB(src2[24-1:20]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(1'b0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[24-1:20]), //4 bit**

**.cout1(co[5]) //1 bit**

**);**

**alu\_4 a47(**

**.clk(clk)**

**,.srcA(src1[28-1:24]), //4 bit**

**.srcB(src2[28-1:24]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(1'b0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[28-1:24]), //4 bit**

**.cout1(co[6]) //1 bit**

**);**

**alu\_4 a48(**

**.clk(clk)**

**,.srcA(src1[32-1:28]), //4 bit**

**.srcB(src2[32-1:28]), //4 bit**

**.less(1'b0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1'b0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b0), //2 bit**

**.result1(result[32-1:28]), //4 bit**

**.cout1(co[7]) //1 bit**

**);**

**assign cout=0;**

**always@(src1 or src2 or clk)**

**begin**

**if(result==0)**

**zero=1;**

**else**

**zero=0;**

**end**

**endmodule**

**Features:**

**這是我第一個實現的32-bit模組，由於上面我已經製作好了4-bit為一組的alu，因此在這邊我總共要弄8次去實現一個32-bit!**

**Or\_32:**

**我認為這個模組跟and\_32很像，只是把指令由and變成or!**

**Code:**

**module or\_32( clk,src1,src2,result,cout,zero,over**

**);**

**input clk;**

**input [32-1:0] src1;**

**input [32-1:0] src2;**

**output [32-1:0] result;**

**output zero;**

**output cout;**

**output over;**

**wire [32-1:0] result;**

**wire [8-1:0] co;**

**reg zero;**

**wire cout;**

**reg over=0;**

**alu\_4 a41 (**

**.clk(clk)**

**,.srcA(src1[4-1:0]), //4 bit**

**.srcB(src2[4-1:0]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[4-1:0]), //4 bit**

**.cout1(co[0]) //1 bit**

**);**

**alu\_4 a42(**

**.clk(clk)**

**,.srcA(src1[8-1:4]), //4 bit**

**.srcB(src2[8-1:4]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[8-1:4]), //4 bit**

**.cout1(co[1]) //1 bit**

**);**

**alu\_4 a43(**

**.clk(clk)**

**,.srcA(src1[12-1:8]), //4 bit**

**.srcB(src2[12-1:8]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[12-1:8]), //4 bit**

**.cout1(co[2]) //1 bit**

**);**

**alu\_4 a44(**

**.clk(clk)**

**,.srcA(src1[16-1:12]), //4 bit**

**.srcB(src2[16-1:12]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[16-1:12]), //4 bit**

**.cout1(co[3]) //1 bit**

**);**

**alu\_4 a45(**

**.clk(clk)**

**,.srcA(src1[20-1:16]), //4 bit**

**.srcB(src2[20-1:16]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[20-1:16]), //4 bit**

**.cout1(co[4]) //1 bit**

**);**

**alu\_4 a46(**

**.clk(clk)**

**,.srcA(src1[24-1:20]), //4 bit**

**.srcB(src2[24-1:20]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[24-1:20]), //4 bit**

**.cout1(co[5]) //1 bit**

**);**

**alu\_4 a47(**

**.clk(clk)**

**,.srcA(src1[28-1:24]), //4 bit**

**.srcB(src2[28-1:24]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[28-1:24]), //4 bit**

**.cout1(co[6]) //1 bit**

**);**

**alu\_4 a48(**

**.clk(clk)**

**,.srcA(src1[32-1:28]), //4 bit**

**.srcB(src2[32-1:28]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b01), //2 bit**

**.result1(result[32-1:28]), //4 bit**

**.cout1(co[7]) //1 bit**

**);**

**assign cout=0;**

**always@(src1 or src2 or clk)**

**begin**

**if(result==0)**

**zero=1;**

**else**

**zero=0;**

**end**

**endmodule**

**Fetures:**

**與and\_32一樣，因為只需要對最基本元件下or的指令即可達成，因此在一開始cin,每個less,invert的輸入都直接填0即可!**

**Add\_32:**

**這是第一個必須考慮到怎麼處理over跟cout的32-bit模組，但是cout原則上不難，比較麻煩的是怎麼判斷over的部分花了一段時間。**

**Code:**

**module add\_32(clk,src1,src2,result,cout,zero,over//,a**

**);**

**input clk;**

**input [32-1:0] src1;**

**input [32-1:0] src2;**

**output [32-1:0] result;**

**output zero;**

**output cout;**

**output over;**

**//output [32-1:0]a;**

**wire [8-1:0] co;**

**reg zero;**

**wire cout;**

**wire over;**

**alu\_4 a41 (**

**.clk(clk)**

**,.srcA(src1[4-1:0]), //4 bit**

**.srcB(src2[4-1:0]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(0), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[4-1:0]), //4 bit**

**.cout1(co[0]) //1 bit**

**//,.ap(a[4-1:0])**

**);**

**alu\_4 a42(**

**.clk(clk)**

**,.srcA(src1[8-1:4]), //4 bit**

**.srcB(src2[8-1:4]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(co[0]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[8-1:4]), //4 bit**

**.cout1(co[1]) //1 bit**

**// ,.ap(a[8-1:4])**

**);**

**alu\_4 a43(**

**.clk(clk)**

**,.srcA(src1[12-1:8]), //4 bit**

**.srcB(src2[12-1:8]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(co[1]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[12-1:8]), //4 bit**

**.cout1(co[2]) //1 bit**

**//,.ap(a[12-1:8])**

**);**

**alu\_4 a44(**

**.clk(clk)**

**,.srcA(src1[16-1:12]), //4 bit**

**.srcB(src2[16-1:12]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(co[2]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[16-1:12]), //4 bit**

**.cout1(co[3]) //1 bit**

**//,.ap(a[16-1:12])**

**);**

**alu\_4 a45(**

**.clk(clk)**

**,.srcA(src1[20-1:16]), //4 bit**

**.srcB(src2[20-1:16]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(co[3]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[20-1:16]), //4 bit**

**.cout1(co[4]) //1 bit**

**//,.ap(a[20-1:16])**

**);**

**alu\_4 a46(**

**.clk(clk)**

**,.srcA(src1[24-1:20]), //4 bit**

**.srcB(src2[24-1:20]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(co[4]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[24-1:20]), //4 bit**

**.cout1(co[5]) //1 bit**

**//,.ap(a[24-1:20])**

**);**

**alu\_4 a47(**

**.clk(clk)**

**,.srcA(src1[28-1:24]), //4 bit**

**.srcB(src2[28-1:24]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(co[5]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[28-1:24]), //4 bit**

**.cout1(co[6]) //1 bit**

**// ,.ap(a[28-1:24])**

**);**

**alu\_4 a48(**

**.clk(clk)**

**,.srcA(src1[32-1:28]), //4 bit**

**.srcB(src2[32-1:28]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(0), //1 bit**

**.cin(co[6]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[32-1:28]), //4 bit**

**.cout1(co[7]) //1 bit**

**//,.ap(a[32-1:28])**

**);**

**assign cout=co[7];**

**assign over=co[7]^(src1[31]^src2[31]^result[31]);//two same sign^==>0 but result not the same sign! overflow for positive co[7]==0 for neg co[7]==1 so need ^co[7]**

**always@(src1 or src2 or clk)**

**begin**

**if(result==0)**

**zero=1;**

**else**

**zero=0;**

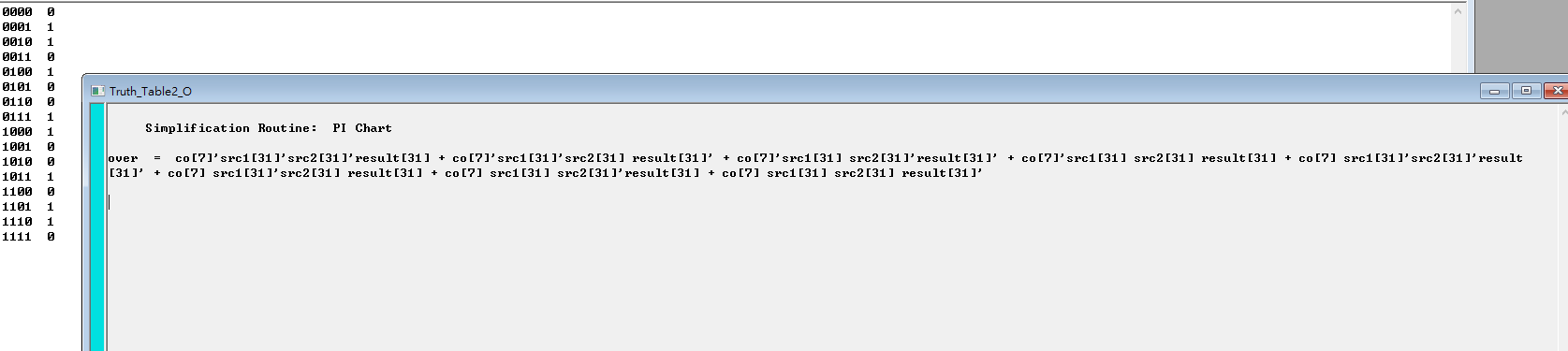
**end**

**endmodule**

**Features:**

1. **assign over=co[7]^(src1[31]^src2[31]^result[31]);**

**我處理overflow的方式就是判斷是否有正+正或負+負但是結果卻是負或正，因此重點放在result[31]，也就是最高位元或者說代表正負的位元，以及src1[31]第一個輸入的正負及src2[31]第二個輸入的正負彼此之間的關係，但是很容易被忽略的是，也要考慮cout[7]，因為sign的位置也有可能被進位導致最後結果只看result[31]不準，根據LogicAid:**

**可以發現我們寫出與我所打的關係式一致的結果。**

**或者用這個方式理解:**

**如果sign有被進位且cout=1則表示原本的輸入至少一個為1，保證結果為沒有overflow，如果sign有被進位且cout=0，則表示原本的輸入沒有1，保證overflow，如果sign沒有被進位且cout=1，則表示原本有兩個輸入1(負數)，保證overflow，如果sign有被進位且cout=1，則表示至少有一個輸入為1，保證沒有overflow，我們可以列出:sign進位^cout**

**又sign進位可以表示成src1[31]^src2[31]^result[31](根據真值表及logic aid)，所以可以再次得證我列的算式的正確性。**

1. **同前面的方式，zero則是直接看result有沒有=0。**
2. **Cout直接取co[7]即可。**

**Sub\_32:**

**我認為減法算式要稍微想一下的地方，但是可以藉由二補數就是全部invert再加1的概念來修改add\_32即可!**

**Code:**

**module sub\_32(clk,src1,src2,result,cout,zero,over**

**);**

**input [32-1:0] src1;**

**input [32-1:0] src2;**

**output [32-1:0] result;**

**output zero;**

**output cout;**

**output over;**

**input clk;**

**wire [32-1:0] result;**

**wire [8-1:0] co;**

**reg zero;**

**wire cout;**

**wire over;**

**alu\_4 a41 (**

**.clk(clk)**

**,.srcA(src1[4-1:0]), //4 bit**

**.srcB(src2[4-1:0]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(1), //1 bit for 2 complement**

**.operation(2'b10), //2 bit**

**.result1(result[4-1:0]), //4 bit**

**.cout1(co[0]) //1 bit**

**);**

**alu\_4 a42(**

**.clk(clk)**

**,.srcA(src1[8-1:4]), //4 bit**

**.srcB(src2[8-1:4]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[0]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[8-1:4]), //4 bit**

**.cout1(co[1]) //1 bit**

**);**

**alu\_4 a43(**

**.clk(clk)**

**,.srcA(src1[12-1:8]), //4 bit**

**.srcB(src2[12-1:8]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[1]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[12-1:8]), //4 bit**

**.cout1(co[2]) //1 bit**

**);**

**alu\_4 a44(**

**.clk(clk)**

**,.srcA(src1[16-1:12]), //4 bit**

**.srcB(src2[16-1:12]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[2]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[16-1:12]), //4 bit**

**.cout1(co[3]) //1 bit**

**);**

**alu\_4 a45(**

**.clk(clk)**

**,.srcA(src1[20-1:16]), //4 bit**

**.srcB(src2[20-1:16]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[3]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[20-1:16]), //4 bit**

**.cout1(co[4]) //1 bit**

**);**

**alu\_4 a46(**

**.clk(clk)**

**,.srcA(src1[24-1:20]), //4 bit**

**.srcB(src2[24-1:20]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[4]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[24-1:20]), //4 bit**

**.cout1(co[5]) //1 bit**

**);**

**alu\_4 a47(**

**.clk(clk)**

**,.srcA(src1[28-1:24]), //4 bit**

**.srcB(src2[28-1:24]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[5]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[28-1:24]), //4 bit**

**.cout1(co[6]) //1 bit**

**);**

**alu\_4 a48(**

**.clk(clk)**

**,.srcA(src1[32-1:28]), //4 bit**

**.srcB(src2[32-1:28]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[6]), //1 bit**

**.operation(2'b10), //2 bit**

**.result1(result[32-1:28]), //4 bit**

**.cout1(co[7]) //1 bit**

**);**

**assign cout=co[7];**

**assign over=co[7]^(src1[31]^(src2[31]^1)^result[31]);**

**always@(src1 or src2 or clk)**

**begin**

**if(result==0)**

**zero=1;**

**else**

**zero=0;**

**end**

**endmodule**

**Features:**

1. **可以發現我在第一個4-bit alu裡面加了一個起始cin，這主要是因為二補數需要加一個1，而在每個alu都把B\_invert設為1，這樣減法就可以用加分來完成!**
2. **assign over=co[7]^(src1[31]^(src2[31]^1)^result[31]);**

**與add不一樣也特別的是src2[31]^1的部分，因為sub有把B給invert所以^1才是實際進去的輸入!**

**Nor\_32:**

**Nor的真值表如圖:**

****

**所以可以發現基本上nor(p,q)=~p&~q，就可以把nor想成是把and跟invert結合!**

**Code:**

**module nor\_32(**

**clk,src1,src2,result,cout,zero,over**

**);**

**input clk;**

**input [32-1:0] src1;**

**input [32-1:0] src2;**

**output [32-1:0] result;**

**output zero;**

**output cout;**

**output over;**

**wire [32-1:0] result;**

**wire [8-1:0] co;**

**reg zero;**

**wire cout;**

**reg over=0;**

**alu\_4 a41 (**

**.clk(clk)**

**,.srcA(src1[4-1:0]), //4 bit**

**.srcB(src2[4-1:0]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[4-1:0]), //4 bit**

**.cout1(co[0]) //1 bit**

**);**

**alu\_4 a42(**

**.clk(clk)**

**,.srcA(src1[8-1:4]), //4 bit**

**.srcB(src2[8-1:4]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[8-1:4]), //4 bit**

**.cout1(co[1]) //1 bit**

**);**

**alu\_4 a43(**

**.clk(clk)**

**,.srcA(src1[12-1:8]), //4 bit**

**.srcB(src2[12-1:8]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[12-1:8]), //4 bit**

**.cout1(co[2]) //1 bit**

**);**

**alu\_4 a44(**

**.clk(clk)**

**,.srcA(src1[16-1:12]), //4 bit**

**.srcB(src2[16-1:12]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[16-1:12]), //4 bit**

**.cout1(co[3]) //1 bit**

**);**

**alu\_4 a45(**

**.clk(clk)**

**,.srcA(src1[20-1:16]), //4 bit**

**.srcB(src2[20-1:16]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[20-1:16]), //4 bit**

**.cout1(co[4]) //1 bit**

**);**

**alu\_4 a46(**

**.clk(clk)**

**,.srcA(src1[24-1:20]), //4 bit**

**.srcB(src2[24-1:20]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[24-1:20]), //4 bit**

**.cout1(co[5]) //1 bit**

**);**

**alu\_4 a47(**

**.srcA(src1[28-1:24]), //4 bit**

**.srcB(src2[28-1:24]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[28-1:24]), //4 bit**

**.cout1(co[6]) //1 bit**

**);**

**alu\_4 a48(**

**.srcA(src1[32-1:28]), //4 bit**

**.srcB(src2[32-1:28]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(1), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(0), //1 bit**

**.operation(0), //2 bit**

**.result1(result[32-1:28]), //4 bit**

**.cout1(co[7]) //1 bit**

**);**

**assign cout=0;**

**always@(src1 or src2 or clk)**

**begin**

**if(result==0)**

**zero=1;**

**else**

**zero=0;**

**end**

**endmodule**

**Features:**

**如上面所說，因為nor是~p&~q，所以這次A,B都要invert，接著再執行and。**

**Slt\_32:**

**Set on less than最主要的核心想法就是把A-B看會不會<0，會的話slt=true!**

**Code:**

**module slt\_32(clk,src1,src2,result,cout,zero,over**

**);**

**input clk;**

**input [32-1:0] src1;**

**input [32-1:0] src2;**

**output [32-1:0] result;**

**output zero;**

**output cout;**

**output over;**

**wire [32-1:0] result;**

**wire [8-1:0] co;**

**reg zero;**

**wire cout;**

**wire over;**

**wire sign;**

**alu\_4 a41 (**

**.clk(clk)**

**,.srcA(src1[4-1:0]), //4 bit**

**.srcB(src2[4-1:0]), //4 bit**

**.less((((src1[31]^(src2[31]^1)^sign)==1)&((co[7]^(sign)==0))|((co[7]^(sign)==1)&(src1[31]^(src2[31]^1)^sign)==0))), //1 bit not over then we can just see result max be o and if over flow then slt will be need +1**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(1), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[4-1:0]), //4 bit**

**.cout1(co[0]) //1 bit**

**);**

**alu\_4 a42(**

**.clk(clk)**

**,.srcA(src1[8-1:4]), //4 bit**

**.srcB(src2[8-1:4]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[0]), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[8-1:4]), //4 bit**

**.cout1(co[1]) //1 bit**

**);**

**alu\_4 a43(**

**.clk(clk)**

**,.srcA(src1[12-1:8]), //4 bit**

**.srcB(src2[12-1:8]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[1]), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[12-1:8]), //4 bit**

**.cout1(co[2]) //1 bit**

**);**

**alu\_4 a44(**

**.clk(clk)**

**,.srcA(src1[16-1:12]), //4 bit**

**.srcB(src2[16-1:12]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[2]), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[16-1:12]), //4 bit**

**.cout1(co[3]) //1 bit**

**);**

**alu\_4 a45(**

**.clk(clk)**

**,.srcA(src1[20-1:16]), //4 bit**

**.srcB(src2[20-1:16]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[3]), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[20-1:16]), //4 bit**

**.cout1(co[4]) //1 bit**

**);**

**alu\_4 a46(**

**.clk(clk)**

**,.srcA(src1[24-1:20]), //4 bit**

**.srcB(src2[24-1:20]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[4]), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[24-1:20]), //4 bit**

**.cout1(co[5]) //1 bit**

**);**

**alu\_4 a47(**

**.clk(clk)**

**,.srcA(src1[28-1:24]), //4 bit**

**.srcB(src2[28-1:24]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[5]), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[28-1:24]), //4 bit**

**.cout1(co[6]) //1 bit**

**);**

**alu\_4 a48(**

**.clk(clk)**

**,.srcA(src1[32-1:28]), //4 bit**

**.srcB(src2[32-1:28]), //4 bit**

**.less(0), //1 bit**

**.A\_invert(0), //1 bit**

**.B\_invert(1), //1 bit**

**.cin(co[6]), //1 bit**

**.operation(2'b11), //2 bit**

**.result1(result[32-1:28]), //4 bit**

**.cout1(co[7]) //1 bit**

**);**

**assign sign=src1[30]&(src2[30]^1)|((src1[30]|(src2[30]^1))&(src1[29]&(src2[29]^1)))|((src1[30]|(src2[30]^1))&(src1[29]|(src2[29]^1))&(src1[28]&(src2[28]^1)))|((src1[30]|(src2[30]^1))&(src1[29]|(src2[29]^1))&(src1[28]|(src2[28]^1))&co[6]);**

**assign cout=0;**

**assign over=co[7]^(sign);**

**always@(src1 or src2 or clk)**

**begin**

**if(result==0)**

**zero=1;**

**else**

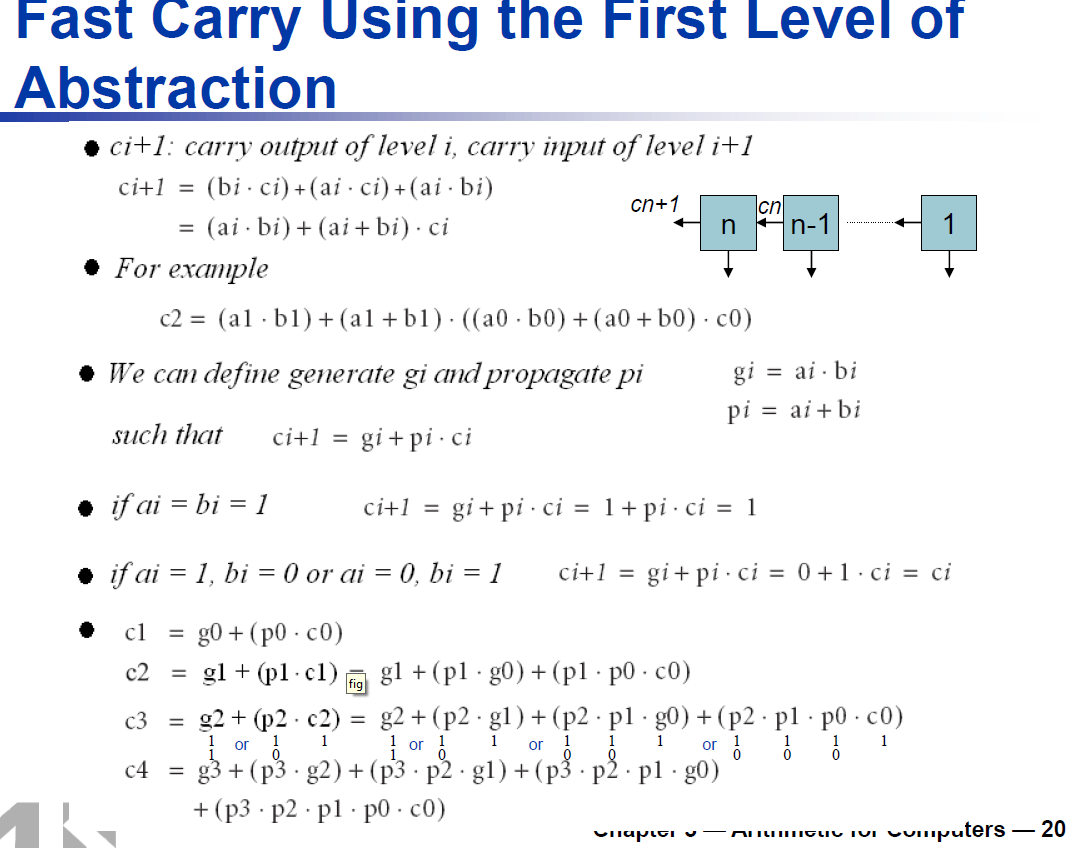
**zero=0;**

**end**

**endmodule**

**Features:**

1. **誠如上面所說，slt主要就是相減再看大小，所以我基於sub的架構去修改成slt，不同的是，由於slt不考慮carryout跟overflow的flag，故可以設為0，或是像我這邊去考慮slt overflow的部分，那就會遇到result不能直接用的問題(不像sub)，考慮到模組的general，所以就只好根據講義:**

****

**特別做一個不需要用到其他carry out的算法!再直接帶入由add推到的over=sign^cout!**

1. **比較特別的是less在第一個units的值不像前面一樣都直接設0，而是要看相減後是否大於0來決定，另外bit2~31可以直接less=0是因為slt只有0,1的結果!**

**Alu:**

**這個模組就是整合以上六個功能的top模組!**

**Code:**

**module alu(**

**clk, // system clock (input)**

**rst\_n, // negative reset (input)**

**src1, // 32 bits source 1 (input)**

**src2, // 32 bits source 2 (input)**

**ALU\_control, // 4 bits ALU control input (input)**

**result, // 32 bits result (output)**

**zero, // 1 bit when the output is 0, zero must be set (output)**

**cout, // 1 bit carry out (output)**

**overflow // 1 bit overflow (output)**

**//,as**

**//,result1,result2,result3,result4,result5,result6**

**);**

**input clk;**

**input rst\_n;**

**input [32-1:0] src1;**

**input [32-1:0] src2;**

**input [4-1:0] ALU\_control;**

**output [32-1:0] result;**

**output reg zero;**

**output reg cout;**

**output reg overflow;**

**//output wire [32-1:0] as;**

**reg [32-1:0] result;**

**//output**

**wire [32-1:0] result1;**

**//output**

**wire [32-1:0] result2;**

**//output**

**wire [32-1:0] result3;**

**//output**

**wire [32-1:0] result4;**

**//output**

**wire [32-1:0] result5;**

**//output**

**wire [32-1:0] result6;**

**wire [6-1:0] zero1;**

**wire [6-1:0] cout1;**

**wire [6-1:0] overflow1;**

**and\_32 an1(.clk(clk),.src1(src1),.src2(src2),.result(result1),.cout(cout1[0]),.zero(zero1[0]),.over(overflow1[0]));**

**or\_32 an2(.clk(clk),.src1(src1),.src2(src2),.result(result2),.cout(cout1[1]),.zero(zero1[1]),.over(overflow1[1]));**

**add\_32 an3(.clk(clk),.src1(src1),.src2(src2),.result(result3),.cout(cout1[2]),.zero(zero1[2]),.over(overflow1[2])**

**//,.a(as)**

**);**

**sub\_32 an4(.clk(clk),.src1(src1),.src2(src2),.result(result4),.cout(cout1[3]),.zero(zero1[3]),.over(overflow1[3]));**

**nor\_32 an5(.clk(clk),.src1(src1),.src2(src2),.result(result5),.cout(cout1[4]),.zero(zero1[4]),.over(overflow1[4]));**

**slt\_32 an6(.clk(clk),.src1(src1),.src2(src2),.result(result6),.cout(cout1[5]),.zero(zero1[5]),.over(overflow1[5]));**

**always@( posedge clk or negedge rst\_n )**

**begin**

**if(!rst\_n)**

**begin**

**result=0;**

**zero=0;**

**cout=0;**

**overflow=0;**

**end**

**else**

**begin**

**if(ALU\_control==4'b0000)//and**

**begin**

**result=result1;**

**cout=cout1[0];**

**zero=zero1[0];**

**overflow=overflow1[0];**

**end**

**else if(ALU\_control==4'b0001)//or**

**begin**

**result=result2;**

**cout=cout1[1];**

**zero=zero1[1];**

**overflow=overflow1[1];**

**end**

**else if(ALU\_control==4'b0010)//add**

**begin**

**result=result3;**

**cout=cout1[2];**

**zero=zero1[2];**

**overflow=overflow1[2];**

**end**

**else if(ALU\_control==4'b0110)//sub**

**begin**

**result=result4;**

**cout=cout1[3];**

**zero=zero1[3];**

**overflow=overflow1[3];**

**end**

**else if(ALU\_control==4'b1100)//nor=~p&~Q**

**begin**

**result=result5;**

**cout=cout1[4];**

**zero=zero1[4];**

**overflow=overflow1[4];**

**end**

**else if(ALU\_control==4'b0111)//slt**

**begin**

**result=result6;**

**cout=cout1[5];**

**zero=zero1[5];**

**overflow=overflow1[5];**

**end**

**if(result==0)**

**zero=1;**

**else**

**zero=0;**

**end**

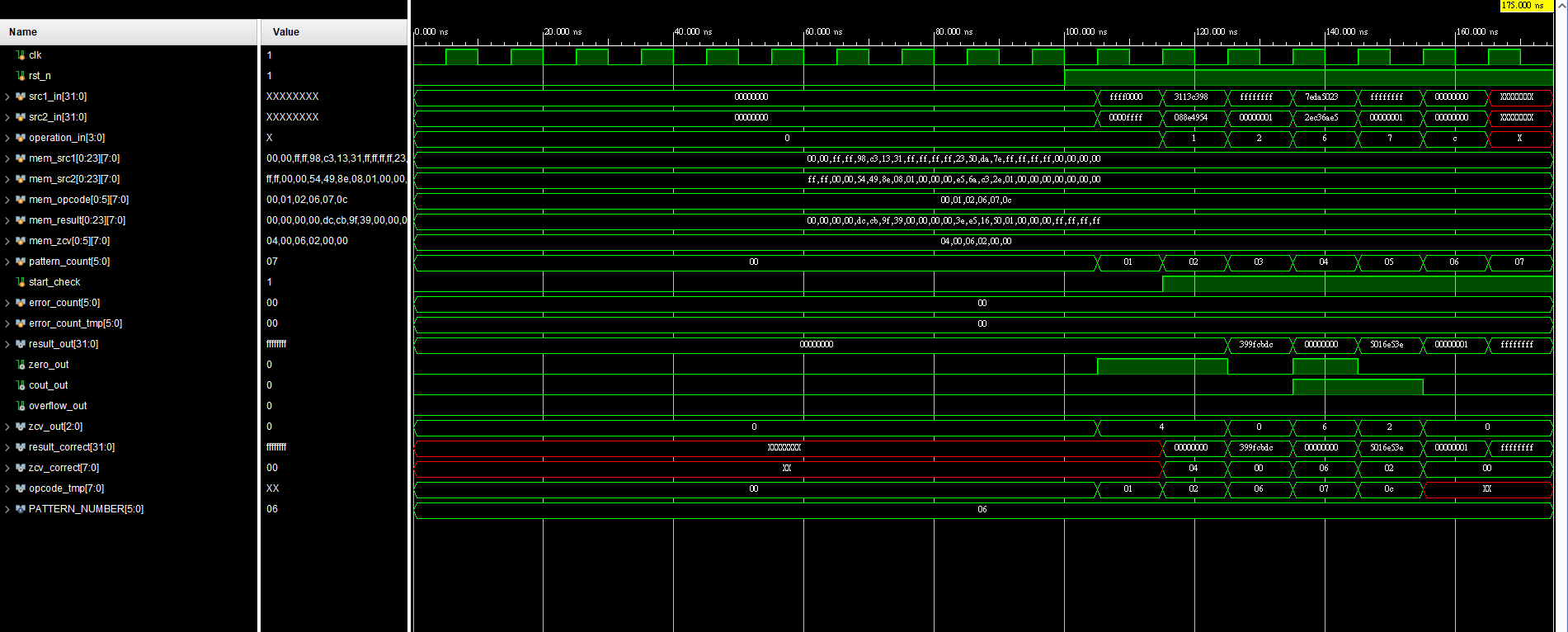
**end**

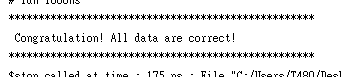
**endmodule**

**Features:**

**與alu\_top類似的設計概念，以wire接選項再以reg接result，另外zero的值我也有放在這邊做判斷(可有可無，僅僅用來做測試)。**

**Experiment result:**

****

****

**Problems you met and solutions:**

1. **alu\_top 的always @(parameter)**

****

**這個問題主要是困擾我很久的問題，雖然整個alu的設計跟coding並不複雜，但是我想verilog這種比較偏硬體的硬體語言最麻煩的就是必須考慮”條件觸發”，因為實際上硬體的CPU也就那麼大，不可能所有任務都要無時無刻執行，因此”條件式觸發”就成為實現硬體必須考慮的部分。而我所遇到的問題就是如果沒有正確填入parameter(或是修改相關的程式)，那麼result可能會有以下三種結果(根據我不斷嘗試跟debug)**

**第一種，訊號為X或是未定義行為(錯誤結果)，也就是說可能在執行這個block的同時有訊號也在”要變化”的過程，根據:**

[**https://www.cnblogs.com/oomusou/archive/2010/07/30/blocking\_vs\_nonblocking.html**](https://www.cnblogs.com/oomusou/archive/2010/07/30/blocking_vs_nonblocking.html)

**通常這樣的錯誤會跟non-blocking與blocking的code同時存在有關!或者，X也可能來自於同個變數在不同module中一下是reg一下是wire的情況，而當執行always時恰好處於wire狀況要變化，這時候非常明顯的衝突就會產生!可以想像，就是一個通時擔任reg右同時擔任wire但是因為處在不同module而無法抓到的錯誤!**

**處理方式:確保每個input output在不同module中是擔任reg還是wire(嚴謹一點的話，或許可以把整個電路畫出來)**

**第二種，訊號延遲，也就是說可能在需要對值的時候，卻沒有觸發always先把該變化的做變化，而是等到對值完後才觸發。或者，也非常有可能來自於reg暫存的關係!**

**處理方式:確保always的parameter可以在需要對值的時候有被觸發，同時確認電路中的reg與result的關係，因為reg不具備wire的即時性，因此如果有訊號被滯留，就很有可能發生延遲甚至變成X!**

**第三種，訊號不變，通常這個問題比較少見(我只有一次)，主要原因就是parameter沒有正確用到具有”影響result”的會變化的變數，或者就是單純coding沒有正確打出這個parameter與result的關係!**

**處理方式:檢查所有具有”功能性質”的coding，確保每個變數都有被正確使用，同時確認parameters是否是需要考慮變化的對象!**

**另外，假如能夠確保自己的coding只是差在always parameter不知道要搬涵的對象有誰(我最後花最久時間看的地方)，根據:**

[**https://stackoverflow.com/questions/6009998/verilog-always-block-using-symbol**](https://stackoverflow.com/questions/6009998/verilog-always-block-using-symbol)

**我們可以知道其實要解決parameter最乾脆簡單的方式就是使用alwyas@(\*)來讓系統自己評估什麼變數需要考慮!(只是相對的，這個處理方法就比較不嚴謹，無法實現更準確的電路)**

1. **slt在overflow的狀況**

**雖然這個問題並不能從測資裡看出來，而且助教也說不需要考慮，但是我還是認為這個問題是值得被討論的，在沒有overflow的時候，我只需要看兩個數相減的結果是否<0即可(也就是看result[31]==1)但是如果overflow的話我認為依然可以看出，因為我們已經限定輸入都是32bits所以就算是overflow也頂多是把sign的位置+1當作這兩個數字運算後需要的部分，因此最後可以發現overflow的話slt就看是否不是<0!**

**處理方式:** ****，要注意的是，因為我在slt是使用sub的架構去看最後結果的最高位元，而sub架構相對於是把B invert，所以一樣會有src2[31]^1的部分來得到加法中src的實際最高位元!另外，由於slt在result的答案與sub不同，因此實際的code可以參考我貼在slt\_32的部分!

**Summary:**

1. **Verilog需要用電路的角度分析code**
2. **用8個4-bit實現似乎不太好debug，應該先合併成16再用16去實現32!**
3. **確認好always block無誤**

**以上差不多就是我在這次實驗覺得比較重要的地方，剩下很多就是關於verilog這個程式常常會有讓我覺得不安的感覺，可能我真的不太熟悉他，畢竟網路上好像也沒什麼資源去查verilog(不像c,c++,java等)，另外在這一次的實驗中我真的花了超久的時間debug always的問題，其中我也學會了如何去看module裡面更深層資料的值，所以我的code有很多comment，主要就是用來測試我哪些做對哪些做錯，算是蠻深刻的學習吧!**