計算機結構作業

資工三 B05902023 李澤諺

我實作這次作業所使用的作業系統為 Ubuntu，並使用 iverilog 來編譯這次作業所寫的程式。

本次作業中，我實作各個模組的方式如下：

<table>
<tr><td>Adder.v</td><td>Input 有 32 bits 的 data1_i 和 data2_i，output 則有 32 bits 的 data_o。
將 data1_i 和 data2_i 相加後，將其結果 assign 給 data_o。</td></tr>
<tr><td>ALU.v</td><td>Input 有 32 bits 的 data1_i 和 data2_i，以及 3 bits 的 ALUCtrl_i，output 則有 32 bits 的 data_o，以及 1 bit 的 Zero_o。
首先宣告一個 32 bits 的 reg，將其命名為 data_reg，接著利用 case 敘述，根據 ALUCtrl_i 的值來決定 ALU 要進行何種運算，規則如下：
<table>
<tr><th>ALUCtrl_i</th><th>運算</th></tr>
<tr><td>3’b001</td><td>data_reg = data1_i + data2_i</td></tr>
<tr><td>3’b010</td><td>data_reg = data1_i - data2_i</td></tr>
<tr><td>3’b011</td><td>data_reg = data1_i * data2_i</td></tr>
<tr><td>3’b100</td><td>data_reg = data1_i & data2_i</td></tr>
<tr><td>3’b101</td><td>data_reg = data1_i | data2_i</td></tr>
</table>
運算的結果存在 data_reg 後，將 data_reg 的結果 assign 給 data_o，並判斷 data_reg 的值是否為 0，若是，則將 Zero_o 的值 assign 為 1，否則 assign 為 0。</td></tr>
<tr><td>MUX32.v</td><td>Input 有 32 bits 的 data1_i 和 data2_i，以及 1 bit 的 select_i，output 則有 32 bits 的 data_o。
判斷 select_i 的值為何來決定要將 data_o 的值 assign 為 data1_i 還是 data2_i，若 select_i 為 0，則將 data_o 的值 assign 為 data1_i，若 select_i 的值為 1，則將 data_o 的值 assign 為 data2_i。</td></tr>
<tr><td>Sign_Extend.v</td><td>Input 有 32 bits 的 data_i，output 則有 32 bits 的 data_o。
首先將 data_i 的第 20 位到第 31 位的 bit 拿下來 assign 給 data_o 的第 0 位到第 11 位後(即 immediate 的值)，再判斷 data_i 第 31 位的 bit 為 0 還是 1(即 immediate 的 sign bit)，將 data_o 剩下第 12 位到第 31 位此 20 個 bits 全部填為 immediate 的 sign bit。</td></tr>
<tr><td>Control.v</td><td>Input 有 7 bits 的 Op_i，output 則有 2 bits 的 ALUOp_o，以及 1 bit 的 ALUSrc_o 和 RegWrite_o。</td></tr>
</table>

<table>
<tr><td></td><td>判斷 Op_i 的值為何來決定要 assign 給 ALUOp_o 和 ALUSrc_o 的值，規則如下：
<table>
<tr><th>Op_i</th><th>ALUOp_i</th><th>ALUSrc_o</th></tr>
<tr><td>7’b0110011</td><td>2’b01</td><td>1’b0</td></tr>
<tr><td>7’b0010011</td><td>2’b10</td><td>1’b1</td></tr>
</table>
而由於本次作業所有的 instructions 皆要將結果寫回 register，故 RegWrite 的值恆 assign 為 1’b1。</td></tr>
<tr><td>ALU_Control.v</td><td>Input 有 32 bits 的 funct_i，以及 2 bits 的 ALUOp_i，output 則有 3 bits 的 ALUCtrl_o。
原理為將整個 instruction 作為 funct_i 傳入 ALU_Control 中，並判斷 funct_i 的第 12 位到第 14 位(即 funct 3)、第 25 位到第 31 位(即 funct 7)、ALUOp_i 的值為何，來決定要 assign 給 ALUCtrl_o 的值，規則如下：
<table>
<tr><th>ALUOp_i</th><th>funct 3</th><th>funct 7</th><th>instruction</th><th>ALUCtrl_o</th></tr>
<tr><td>2’b10</td><td colspan="2">(不用判斷)</td><td>addi</td><td>3’b001</td></tr>
<tr><td rowspan="5">2’b01</td><td rowspan="3">3’b000</td><td>7’b0000000</td><td>add</td><td>3’b001</td></tr>
<tr><td>7’b0100000</td><td>sub</td><td>3b’010</td></tr>
<tr><td>7’b0000001</td><td>mul</td><td>3’b011</td></tr>
<tr><td>3’b111</td><td>(不用判斷)</td><td>and</td><td>3’b100</td></tr>
<tr><td>3’b110</td><td>(不用判斷)</td><td>or</td><td>3’b101</td></tr>
</table>
以此決定 ALU 要做何種運算。</td></tr>
<tr><td>CPU.v</td><td>Input 有 1 bit 的 clk_i 和 rst_i 和 start_i，並另外宣告了兩個 32 bits 的 wire，命名為 inst 和 inst_addr，分別用來記錄目前的 instruction 為何以及目前 instruction 在 instruction memory 的 address，並宣告一個 1 bit 的 wire，命名為 tmp，用來接在 ALU 的 Zero_o 之後。
實作方式為將各個 module 的 input 和 output 如 hw4.pdf 中第 2 頁的 datapath 接起來。</td></tr>
</table>