// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Datapath

// Programmer : Ngan Ngoc Pham

// Date : Nov 1st, 1998

// Description : Datapath of the MIPS processor

// See text book page 414 for block diagram

// Issue : None

//

// Modified : October 4, 2002 (Jon Abell) (fix overflow bug)

// Modified : October, 2010 (P. Franzon/ R.Widialaksono) (make

// little more compatible with current 406 styles)

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// This is removed so that we can use synopsys on the datapath

//`include "memory.v"

// Note : This code is highly structured with a rich hierarchy.

// In reality, the hierarchy is a little too rich ñ a flatter hierarchy

// would be better.

// This is best synthesized at the datapath module level ñ i.e. Assume

// that is the 'leaf' module (P. Franzon)

module datapath ( Clock, // Master Clock

Resetn, // Reset - active low

PCWriteCond, // Used in Branch instruction

PCWrite, // Enable a write to PC

IorD, // Selector for Memory Address

MemRead, // Enable a read from Memory

MemWrite, // Enable a write to Memory

MemtoReg, // Selector for the Write data of RegFile

MemData, // Data obtained from memory

Address, // Address of Memory

WriteData, //Data to be written to memory

IRWrite, // Enable a write to Intruction Register

CauseWrite, // Enable a write to cause register

IntCause, // selector for source of cause (int or exctn)

EPCWrite, // Enable a write to Exception Program Counter

PCSource, // selector for PC source

ALUOp, // ALUcontrol operating mode

ALUSrcB, // selector for ALU source B

ALUSrcA, // selector for ALU source A

RegWrite, // Enable a write to Register File

RegDst, // Selector for the reg to be written in RegFile

Opcode, // Opcode of an instruction

overflowRegOut, // Buffered overflow

LoadByte // CHANGE

);

// Port declaration

input Clock, Resetn, PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite;

input [31:0] MemData;

output [31:0] Address;

output [31:0] WriteData;

input CauseWrite, IntCause, EPCWrite, ALUSrcA, RegWrite, RegDst, LoadByte; // CHANGE

input [1:0] ALUOp, ALUSrcB, PCSource;

output overflowRegOut;

output [5:0] Opcode; // used as input for FSM

// Net and registers

// Internal variables

wire PCenable, // Enable a write to PC

zero, // zero output from ALU

overflowALUout, // overflow output from ALU

CauseIn; // input of Cause register

reg Cause, // Output from Cause register

overflowRegOut; // overflow output from Overflow Register

reg [31:0] PC; // Output of PCreg

wire [31:0] NextPC; // Next PC value

wire [2:0] Operation; // ALU operation

wire [4:0] WriteReg; // Register to be written in Register File

wire [31:0] MemData, // Output from memory

MDRout, // Output from MDR

Instruction, // Output from Instruction Register

ReadData1, // Output 1 from RegFile

Aout, // Output from register A

ReadData2, // Output 2 from RegFile

Bout, // Output from register B

ALUOut, // Output from ALU

ALURegOut, // Output from ALUReg

EPCout, // Output from EPC

Address, // Address to be read from/write to in memory

RegWriteData, // Data to be written to Register File

SEAddr, // Sign\_extended Address

SE\_SLAddr, // Sign\_extended and Shifted\_Left Addr

JumpAddr, // Addr to jump to

Byte, // CHANGE

ALUinA, // input A of the ALU

ALUinB; // Input B of the ALU

wire [7:0] byteMuxOut; // CHANGE

// Program Counter Override ~~~~~~~~~~~~~~~~~~~~~~~

always@(posedge Clock)

begin

if (!Resetn) PC <= 0;

else if (PCenable) PC <= NextPC;

end

// Declaring all reg-32bit with enable ~~~~~~~~~~~~~~~~~~~~~~~

// Instruction Register

Register\_32bitWE IR ( .Clock(Clock),

.Enable(IRWrite),

.Input(MemData),

.Output(Instruction));

// Exception Program Counter

Register\_32bitWE EPC ( .Clock(Clock),

.Enable(EPCWrite),

.Input(ALUOut),

.Output(EPCout));

// Declaring all reg-32bit without enable ~~~~~~~~~~~~~~~~~~~~~

// Memory Data Register

Register\_32bit MDR ( .Clock(Clock),

.Input(MemData),

.Output(MDRout) );

// Outputs from Register File A and B

Register\_32bit A ( .Clock(Clock),

.Input(ReadData1),

.Output(Aout) );

Register\_32bit B ( .Clock(Clock),

.Input(ReadData2),

.Output(Bout) );

// ALUbuffer

Register\_32bit ALUReg ( .Clock(Clock),

.Input(ALUOut),

.Output(ALURegOut) );

// Register File ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

RegFile RF (.Clock(Clock),

.ReadReg1(Instruction[25:21]),

.ReadReg2(Instruction[20:16]),

.RegWrite(RegWrite),

.WriteData(RegWriteData),

.WriteReg(WriteReg),

.ReadData1(ReadData1),

.ReadData2(ReadData2) );

// ALUcontrol ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ALUcontrol ALUcontrol ( .Funct(Instruction[5:0]),

.ALUOp(ALUOp),

.Operation(Operation) );

// ALU ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ALU32Bit ALU (.A(ALUinA),

.B(ALUinB),

.Operation(Operation[1:0]),

.Binvert(Operation[2]),

.result(ALUOut),

.zero(zero),

.overflow(overflowALUout) );

// Overflow Register ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

always @(posedge Clock)

overflowRegOut <= overflowALUout;

// Cause Register ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

always @(posedge Clock)

if (CauseWrite) Cause <= CauseIn;

// Muxes and miscellaneous ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Mux2\_1 for Memory address

mux2to1\_32Bit MemMux ( .select(IorD),

.in0(PC),

.in1(ALURegOut),

.out(Address) );

// Mux2\_1 for Write Register of the Reg File

mux2to1\_5Bit RFmux1 ( .select(RegDst),

.in0(Instruction[20:16]),

.in1(Instruction[15:11]),

.out(WriteReg) );

// Mux2\_1 for Write Data of the Reg File

mux4to1\_32Bit RFmux2 ( .select0(MemtoReg),

.select1(LoadByte),

.in0(ALURegOut),

.in1(MDRout),

.in2(Byte), // CHANGE

.in3(Byte),

.out(RegWriteData) );

mux4to1\_8Bit byteMux ( .select0(Instruction[0] ^ ReadData1[0]),

.select1((Instruction[0] & ReadData1[0]) + Instruction[1] + ReadData1[1]),

.in0(MDRout[31:24]),

.in1(MDRout[23:16]),

.in2(MDRout[15:8]),

.in3(MDRout[7:0]),

.out(byteMuxOut) ); // CHANGE

SignExtend\_8to32 SE2 ( .In(byteMuxOut), // CHANGE

.Out(Byte) );

// Mux for ALU inA

mux2to1\_32Bit ALUmuxA ( .select(ALUSrcA),

.in0(PC),

.in1(Aout),

.out(ALUinA) );

// Mux for ALU inB

mux4to1\_32Bit ALUmuxB ( .select0(ALUSrcB[0]),

.select1(ALUSrcB[1]),

.in0(Bout),

.in1(32'h0000\_0004),

.in2(SEAddr),

.in3(SE\_SLAddr),

.out(ALUinB) );

// Mux4-1 for next PC

mux4to1\_32Bit PCmux ( .select0(PCSource[0]),

.select1(PCSource[1]),

.in0(ALUOut),

.in1(ALURegOut),

.in2(JumpAddr),

.in3(32'hc000\_0000),

.out(NextPC));

// Mux2\_1 for Cause Register

mux2to1 CauseMux ( .select(IntCause),

.in0(1'b0),

.in1(1'b1),

.out(CauseOut) );

// Sign\_extender for data from Instruction Reg

SignExtend SE1 ( .In(Instruction[15:0]),

.Out(SEAddr) );

// Left\_shifter for SEAddr

shift\_left2 SL1 ( .In(SEAddr),

.Out(SE\_SLAddr) );

// Logic for JumpAddr

wire [27:0] temp;

assign temp = Instruction[25:0] << 2;

assign JumpAddr = { PC[31:28], temp }; // Watch out for this one

// Logic for PCEnable

assign PCenable = (zero & PCWriteCond) | PCWrite;

// Logic for outputOpcode

assign Opcode = Instruction[31:26];

//!add Combinational Logic

assign WriteData = Bout;

endmodule

/\* Module \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* NAME : Cla\_4bit

\*

\* DESCRIPTION : A 4-bit carry-lookahead module

\*

\*

\* NOTES :

\*

\*

\* REVISION HISTORY :

\* Date Programmer Description

\* ---- ---------- -----------

\* 8-26-96 Srisai Rao Created

\*

\*M\*/

/\*==== Declarations =====================================================\*/

module Cla\_4bit ( c, ci, a, b ) ;

/\*---- Inputs -----------------------------------------------------------\*/

/\*---- Outputs ----------------------------------------------------------\*/

input ci;

input [3:0] a, b;

/\*---- Outputs ----------------------------------------------------------\*/

output [3:0] c;

/\*---- Registers --------------------------------------------------------\*/

reg [3:0] c;

reg g0, g1, g2, g3;

reg p0, p1, p2, p3;

/\*==== Operation ========================================================\*/

always@ ( ci or a or b )

begin

g0 = a[0] & b[0];

g1 = a[1] & b[1];

g2 = a[2] & b[2];

g3 = a[3] & b[3];

p0 = a[0] | b[0];

p1 = a[1] | b[1];

p2 = a[2] | b[2];

p3 = a[3] | b[3];

c[0] = g0 | ( p0 & ci );

c[1] = g1 | ( p1 & g0 ) | ( p1 & p0 & ci );

c[2] = g2 | ( p2 & g1 ) | ( p2 & p1 & g0 ) | ( p2 & p1 & p0 & ci );

c[3] = g3 | ( p3 & g2 ) | ( p3 & p2 & g1 ) | ( p3 & p2 & p1 & g0 ) | ( p3 &

p2 & p1 & p0 & ci );

end

endmodule

/\* Module \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* NAME : shift\_left

\*

\* DESCRIPTION : 2-bit left shift

\*

\*

\* NOTES :

\*

\*

\* REVISION HISTORY :

\* Date Programmer Description

\* ---- ---------- -----------

\* 8-26-96 Srisai Rao Created

\* 11-03-98 Irfan Mazhar/Ngan Pham Revised

\*

\*M\*/

/\*==== Declarations =====================================================\*/

module shift\_left2( Out,

In ) ;

/\*---- Inputs -----------------------------------------------------------\*/

input[31:0] In;

/\*---- Outputs ----------------------------------------------------------\*/

output[31:0] Out;

/\*---- Registers --------------------------------------------------------\*/

reg[31:0] Out;

/\*==== Operation ========================================================\*/

always@( In )

begin

Out = (In << 2 ) ;

end

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Register\_32bit

// Programmer : Ngan Ngoc Pham

// Date : Oct 21, 1998

// Description : A 32-bit register without enable

// Note : none

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module Register\_32bit (Clock, Input, Output);

// Port declaration

input [31:0] Input; // new input

input Clock; // syn clock

output [31:0] Output; // output

// Internal variables

reg [31:0] Output;

// Initialization register

// Initial statements are not supported by Synopsys, so this is removed

//initial

// Output = 32'h0000;

// Code for the flip-flops

always @(posedge Clock)

Output <= Input;

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Register\_32bit

// Programmer : Ngan Ngoc Pham

// Date : Oct 21, 1998

// Description : A 32-bit register with enable

// Note : none

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module Register\_32bitWE (Clock, Enable, Input, Output);

// Port declaration

input Enable; // enable signal

input [31:0] Input; // new input

input Clock; // syn clock

output [31:0] Output; // output

// Internal variables

reg [31:0] Output;

// Initialization register

// Initial statements are not supported by Synopsys, so this is removed

//initial

//Output = 32'h0000;

// Code for the flip-flops

always @(posedge Clock)

if (Enable) Output <= Input;

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : RegFile

// Programmer : Ngan Ngoc Pham

// Date : Oct 23, 1998

// Description :

// This is the verilog description for a register file

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module RegFile( Clock, // Syn clock

ReadReg1, // first reading-reg

ReadReg2, // second reading-reg

RegWrite, // write enable

WriteData, // data to be written

WriteReg, // register to be written

ReadData1, // data read from first reg

ReadData2 // data read from second reg

);

// Port declaration

input Clock, RegWrite;

input [4:0] ReadReg1, ReadReg2, WriteReg;

input [31:0] WriteData;

output [31:0] ReadData1, ReadData2;

// Internal variable

reg [31:0] register[31:1]; // 31 32-bit regs ($0=0)

// Logic for reading register, register $0 is 0

assign ReadData1 = |ReadReg1 ? register[ReadReg1[4:0]]: 32'h0;

assign ReadData2 = |ReadReg2 ? register[ ReadReg2[4:0] ]: 32'h0;

// Logic for writing register

always @(posedge Clock)

begin

if (RegWrite && (|WriteReg))

register[ WriteReg[4:0] ] <= WriteData;

end

endmodule

// This is a 4-to-1 Mux

module mux4to1 (select0, select1, in0, in1, in2, in3, out);

// Port declration

input select0, select1; // select lines

input in0, in1, in2, in3; // inputs

output out;

reg out;

always @(select0 or select1 or in0 or in1 or in2 or in3)

case ({select1, select0})

2'b00 : out = in0;

2'b01 : out = in1;

2'b10 : out = in2;

2'b11 : out = in3;

endcase

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Mux2to1\_5bit

// Programmer : Ngan Ngoc Pham

// Description : 5-bit Mux2\_to\_1

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module mux2to1\_5Bit (select, in0, in1, out);

// Port declaration

input select;

input [4:0] in0, in1;

output [4:0] out;

assign out = select ? in1 : in0;

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Mux2to1

// Programmer : Ngan Ngoc Pham

// Description : Mux 2\_to\_1

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module mux2to1 (select, in0, in1, out);

// Port declaration

input select, in0, in1;

output out;

assign out = select ? in1 : in0;

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Mux2to1\_32bit

// Programmer : Ngan Ngoc Pham

// Description : 32-bit Mux2\_to\_1

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module mux2to1\_32Bit (select, in0, in1, out);

// Port declaration

input select;

input [31:0] in0, in1;

output [31:0] out;

assign out = select ? in1 : in0;

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : ALU32Bit

// Programmer : Ngan Ngoc Pham

// Date : Oct 21, 1998

// Description : See specification in text book, page 355

// Note : Modified to Dr Franzon suggestion

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module ALU32Bit (A, B, Operation, Binvert, result, zero, overflow);

// port declaration

input [31:0] A, B; // input numbers

input[1:0] Operation; // op-code

input Binvert; // sub ?

output [31:0] result;

output zero, overflow;

reg [31:0] result;

reg overflow;

// internal variables

wire [31:0] AND, OR, XOR, ADD\_SUB; // result of each arithmetic

wire ov; // overflow signal from adder

// Logic for arithmetics

assign AND = A & B;

assign OR = A | B;

assign XOR = A ^ B;

Adder\_32bit adder(A, B, Binvert, Binvert, ADD\_SUB, ov);

// Logic for result

// See text book page 355 for specifications

always @(Operation or AND or OR or XOR or ADD\_SUB)

begin

case (Operation)

2'b10: result = ADD\_SUB;

2'b00: result = AND;

2'b01: result = OR;

2'b11: result = XOR;

default: result = 32'hffff;

endcase

end

// Logic for zero

assign zero = ~|result;

// Logic for overflow

always @(Operation or Binvert or ov)

begin

overflow = 1'b0;

overflow = ~Binvert & ~Operation[0] & Operation[1] & ov;

end

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : ALUcontrol

// Programmer : Ngan Ngoc Pham

// Date : Oct 21, 1998

// Description : See specification in text book, page 355

// Note : none

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module ALUcontrol (Funct, ALUOp, Operation);

// Port declaration

input [5:0] Funct; // Function code from instruction

input [1:0] ALUOp; // ALU operation mode

output [2:0] Operation; // operation coding for ALU

reg [2:0] Operation;

// Logic for ALUcontrol

// See text book page 355 for specifications

always @(Funct or ALUOp)

case (ALUOp)

2'b00: Operation = 3'b010; // always add

2'b01: Operation = 3'b110; // always substract

default:

case (Funct)

6'b10\_0000: Operation = 3'b010; // dec 32 = add

6'b10\_0010: Operation = 3'b110; // dec 34 = sub

6'b10\_0100: Operation = 3'b000; // dec 36 = and

6'b10\_0101: Operation = 3'b001; // dec 37 = or

6'b10\_0110: Operation = 3'b011; // dec 38 = xor

6'b10\_1010: Operation = 3'b111; // dec 42 = slt

default : Operation = 3'b111; // unexpected case

endcase

endcase

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Mux4to1\_32Bit

// Programmer : Ngan Ngoc Pham

// Description : 32-Bit Mux4\_to\_1

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module mux4to1\_32Bit (select0, select1, in0, in1, in2, in3, out);

// Port declration

input select0, select1; // select lines

input [31:0] in0, in1, in2, in3; // inputs

output [31:0] out;

reg [31:0] out;

always @(select0 or select1 or in0 or in1 or in2 or in3)

case ({select1, select0})

2'b00 : out = in0;

2'b01 : out = in1;

2'b10 : out = in2;

2'b11 : out = in3;

endcase

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Mux4to1\_8Bit CHANGE

// Programmer : John Sizemore

// Description : 8-Bit Mux4\_to\_1

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module mux4to1\_8Bit (select0, select1, in0, in1, in2, in3, out);

// Port declration

input select0, select1; // select lines

input [7:0] in0, in1, in2, in3; // inputs

output [7:0] out;

reg [7:0] out;

always @(select0 or select1 or in0 or in1 or in2 or in3)

case ({select1, select0})

2'b00 : out = in0;

2'b01 : out = in1;

2'b10 : out = in2;

2'b11 : out = in3;

endcase

endmodule

/\* Module \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* NAME : Sign-Extend 8 to 32

\* DESCRIPTION : A 8-to-32 bit sign-extension module

\*

\*

\* NOTES : // CHANGE

\*

\*

\* REVISION HISTORY :

\* Date Programmer Description

\* ---- ---------- -----------

\* 12-02-11 John Sizemore Created

\*/

/\*==== Declarations =====================================================\*/

module SignExtend\_8to32( Out,

In ) ;

/\*---- Inputs -----------------------------------------------------------\*/

input[7:0] In;

/\*---- Outputs ----------------------------------------------------------\*/

output[31:0] Out;

/\*---- Registers --------------------------------------------------------\*/

reg[31:0] Out;

/\*==== Operation ========================================================\*/

always@( In )

begin

Out = { { 24 {In[7] } }, In };

end

endmodule

/\* Module \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* NAME : Sign-Extend

\*

\* DESCRIPTION : A 16-to-32 bit sign-extension module

\*

\*

\* NOTES :

\*

\*

\* REVISION HISTORY :

\* Date Programmer Description

\* ---- ---------- -----------

\* 8-26-96 Srisai Rao Created

\* 11-03-98 Irfan Mazhar/Ngan Pham Revised to new version

\*

\*M\*/

/\*==== Declarations =====================================================\*/

module SignExtend( Out,

In ) ;

/\*---- Inputs -----------------------------------------------------------\*/

input[15:0] In;

/\*---- Outputs ----------------------------------------------------------\*/

output[31:0] Out;

/\*---- Registers --------------------------------------------------------\*/

reg[31:0] Out;

/\*==== Operation ========================================================\*/

always@( In )

begin

Out = { { 16{ In[15] } }, In };

end

endmodule

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : Carry Look Ahead Adder

// Programmer : Ngan Ngoc Pham

// Date : Dec 22, 1998

// Description : An implementation of the carry look ahead

// adder. See Patterson & Hennessy, 2nd Edition

// page 241 for details.

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module Adder\_32bit (a, b, ci, binv, result, overflow);

// Port declarations

input [31:0] a, b; // Two 32\_bit input number

input ci; // carry in

input binv; // 1 if add and 0 if substract

output [31:0] result; // result of the calculation

output overflow; // overflow

// Internal variables

wire [31:0] c; // c = inverted b

wire [31:0] bnew; // input used for 2 number

wire [31:0] carry0; // carry bits for case ci = 1'b0

wire [31:0] carry1; // carry bits for case ci = 1'b1

wire [31:0] carry; // carry bits

wire [31:0] y; // result of first XOR 2 inputs

// preparing input for sub

assign c = ~b;

// Select second input

assign bnew = binv ? c : b;

// Calculating carry look ahead carryin bits

Cla\_4bit cla00(carry0[3:0], 1'b0, a[3:0], bnew[3:0]); // without ci

Cla\_4bit cla01(carry0[7:4], 1'b0, a[7:4], bnew[7:4]);

Cla\_4bit cla02(carry0[11:8], 1'b0, a[11:8], bnew[11:8]);

Cla\_4bit cla03(carry0[15:12], 1'b0, a[15:12], bnew[15:12]);

Cla\_4bit cla04(carry0[19:16], 1'b0, a[19:16], bnew[19:16]);

Cla\_4bit cla05(carry0[23:20], 1'b0, a[23:20], bnew[23:20]);

Cla\_4bit cla06(carry0[27:24], 1'b0, a[27:24], bnew[27:24]);

Cla\_4bit cla07(carry0[31:28], 1'b0, a[31:28], bnew[31:28]);

Cla\_4bit cla10(carry1[3:0], 1'b1, a[3:0], bnew[3:0]); // with ci

Cla\_4bit cla11(carry1[7:4], 1'b1, a[7:4], bnew[7:4]);

Cla\_4bit cla12(carry1[11:8], 1'b1, a[11:8], bnew[11:8]);

Cla\_4bit cla13(carry1[15:12], 1'b1, a[15:12], bnew[15:12]);

Cla\_4bit cla14(carry1[19:16], 1'b1, a[19:16], bnew[19:16]);

Cla\_4bit cla15(carry1[23:20], 1'b1, a[23:20], bnew[23:20]);

Cla\_4bit cla16(carry1[27:24], 1'b1, a[27:24], bnew[27:24]);

Cla\_4bit cla17(carry1[31:28], 1'b1, a[31:28], bnew[31:28]);

// Logic to select carry bits

assign carry[3:0] = ci ? carry1[3:0] : carry0[3:0];

assign carry[7:4] = carry[3] ? carry1[7:4] : carry0[7:4];

assign carry[11:8] = carry[7] ? carry1[11:8] : carry0[11:8];

assign carry[15:12] = carry[11] ? carry1[15:12] : carry0[15:12];

assign carry[19:16] = carry[15] ? carry1[19:16] : carry0[19:16];

assign carry[23:20] = carry[19] ? carry1[23:20] : carry0[23:20];

assign carry[27:24] = carry[23] ? carry1[27:24] : carry0[27:24];

assign carry[31:28] = carry[27] ? carry1[31:28] : carry0[31:28];

// Logic for overflow

assign overflow = carry[31] ^ carry[30];

// Calculating the result

assign y = a ^ bnew;

assign result = y ^ {carry[30:0],ci};

endmodule