```
/*
 File: paths.v
 Created: 4/5/1998
 Modules contained: datapath, controlpath

 The condition codes are ordered as ZCNV

 Changelog:
 9 June 1999 : Added stack pointer
 4/16/2001: Reverted to base code (verBurg)
 4/16/2001: Added the "addsp" instruction. (verBurg)
*/

/*
 module datapath

   This is the datapath for the WilE240.  Mainly just instantiates stuff
 in modules.v.
 */
module datapath (IR, SP, CondCodes, cPoints, clock, reset, ALUSrcA, ALUSrcB,
         ALUresult, PC, MemAddr, MemData, RegSelA, RegSelB, RegSelC,
         RegC, addr_p, pport, stbl, ackl, busy, pe, clk25);
   output [15:0]     IR, SP;
   output [3:0]      CondCodes;
   output [15:0]     ALUSrcA, ALUSrcB, ALUresult,PC,MemAddr,MemData;
   output [2:0]      RegSelA, RegSelB;
   output [15:0]     RegC;
   input [2:0]       RegSelC;
   input [13:0]      cPoints;
   input         clock, reset;

   input [7:0]       pport;
   output [7:0]      addr_p;
   input         stbl, clk25;
   output        ackl, busy, pe;

   wire [15:0]       RfileA, RfileB;
   wire [15:0]       PC, MemData, MemAddr, IR, SP;
   wire [15:0]       ALUSrcA, ALUSrcB, ALUresult;
   wire [15:0]       MemOut, NewMDR, MemIn;
   wire [5:0]        LoadLines;
   wire [3:0]        NewCondCodes, ALUop;
   wire [1:0]        SrcASel, SrcBSel;
   wire          RegLoad, CCLoad, SPLoad, PCLoad, MDRLoad, MDwrite;
   wire          MARLoad, IRLoad, DataMemLoad, MuxMDRSel;
   wire          WE, RE;

   // Assign wires
   assign        MDRLoad = MDwrite & RE;
   assign        MuxMDRSel = cPoints[1];
   assign        RegSelA = IR[5:3];
   assign        RegSelB = IR[2:0];
   assign        WE = cPoints[0];
   assign        RE = cPoints[1];
   assign        MemIn = MemData;

   // Declare the modules that we need:
   RegFile rfile(RfileA, RfileB, RegC, ALUresult, RegLoad, IR[5:3], IR[2:0],
         RegSelC, clock, reset); // IR's are RegSels.

   TriDrive a(ALUresult, MDwrite, NewMDR);
   TriDrive b(MemOut,    RE,      NewMDR);
```

```
   //TriDrive c(MemData,   WE,      MemInOut);

   mux4_1_16bit MuxA(cPoints[9:8],ALUSrcA, RfileA, SP, PC, MemData),
                MuxB(cPoints[7:6],ALUSrcB, RfileB, SP, PC, MemData);

   ALU alu(ALUresult, NewCondCodes, ALUSrcA, ALUSrcB, cPoints[13:10]);

   decoder RegLoadDecoder ({IRLoad, MARLoad, MDwrite, PCLoad, SPLoad,
                RegLoad}, cPoints[5:3]);

   clocked_reg_16 PCReg(PC,  ALUresult, PCLoad, clock, reset);

   clocked_reg_16 MDR(MemData, NewMDR, MDRLoad, clock, reset),
                  MAR(MemAddr, ALUresult, MARLoad, clock, reset),
     IRReg(IR, ALUresult, IRLoad, clock, reset),
     SPReg(SP, ALUresult, SPLoad, clock, reset);

   clocked_reg_4 CCreg(CondCodes, NewCondCodes, cPoints[2], clock);

`ifdef synthesis
   Memory_256_16bit DataMem(MemOut, MemIn, MemAddr, WE, addr_p, busy,
                pport, stbl, ackl, pe, clk25, reset);
`else
   Memory_256_16bit DataMem(MemOut, MemIn, MemAddr, WE, addr_p, busy,
                pport, stbl, ackl, pe, clk25, reset, clock);
`endif
endmodule


/*
 module controlpath

 This is the FSM for the WilE240.  Any modifications to the ISA
 or even the base implementation will most likely affect this module.
 (Hint, hint.)
 */
module controlpath (out, CCin, IRIn, clock, reset, currState, nextState, w);
   input [3:0]   CCin;
   input [15:0]  IRIn;
   output [13:0] out;
   output [9:0]  currState,nextState;
   output    w;  // this is for simulation only: dump reg file
   input     clock,reset;
   reg [9:0]     currState, nextState;
   reg [13:0]    out;
   reg       w;

`include "params.v"

   always @(posedge clock or negedge reset)
     if (~reset)
       currState <= `fetch;
     else
       currState <= nextState;

   // order of control points:
   // {ALU fn, AmuxSel, BmuxSel, DestDecode, CCLoad, RE, WE}

   always @(currState or CCin or IRIn) begin
      w = 0;
      case (currState)
    `fetch: begin
```

```
      out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
      nextState = `fetch1;
   end
   `fetch1: begin
      out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
      nextState = `fetch2;
   end
   `fetch2: begin
      out = {fA, muxMDR, 2'bxx, destIR, noLoad, noLoad, noLoad};
      nextState = `decode;
   end
   `decode: begin
      out = {4'bxxxx, 2'bxx, 2'bxx, destNone, noLoad, noLoad, noLoad};
      nextState = IRIn[15:6];
   end
   `ldi: begin
      out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
      nextState = `ldi1;
   end
   `ldi1: begin
      out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
      nextState = `ldi2;
   end
   `ldi2: begin
      out = {fA, muxMDR, 2'bxx, destReg, loadCC, noLoad, noLoad};
      nextState = `fetch;
   end
   `add: begin
      out = {fAplusB,muxReg,muxReg,destReg,loadCC,noLoad,noLoad};
      nextState = `fetch;
   end
   `sub: begin
      out = {fAminB,muxReg,muxReg,destReg,loadCC,noLoad,noLoad};
      nextState = `fetch;
   end
   `incr: begin
      out = {fAplus1,muxReg,2'bxx,destReg,loadCC,noLoad,noLoad};
      nextState = `fetch;
   end
   `decr: begin
      out = {fAmin1,muxReg,2'bxx,destReg,loadCC,noLoad,noLoad};
      nextState = `fetch;
   end
   `ldr: begin
      out = {fB, 2'bxx,muxReg,destMAR,noLoad,noLoad,noLoad};
      nextState = `ldr1;
   end
   `ldr1: begin
      out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,memRD,noLoad};
      nextState = `ldr2;
   end
   `ldr2: begin
      out = {fA, muxMDR,2'bxx,destReg,loadCC,noLoad,noLoad};
      nextState = `fetch;
   end
   `bra: begin
      out = {fA, muxPC,2'bxx,destMAR,noLoad,noLoad,noLoad};
      nextState = `bra1;
   end
   `bra1: begin
      out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,memRD,noLoad};
      nextState = `bra2;
```

```
    end
    `bra2: begin
       out = {fA, muxMDR,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `brn: begin
       out = {fA, muxPC,2'bxx,destMAR,noLoad,noLoad,noLoad};
       if (CCin[1])
         nextState = `brn2;
       else
         nextState = `brn1;
    end
    `brn1: begin
       out = {fAplus1, muxPC,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `brn2: begin
       out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,memRD,noLoad};
       nextState = `brn3;
    end
    `brn3: begin
       out = {fA, muxMDR,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `brz: begin
       out = {fA, muxPC,2'bxx,destMAR,noLoad,noLoad,noLoad};
       if (CCin[3])
         nextState = `brz2;
       else
         nextState = `brz1;
    end
    `brz1: begin
       out = {fAplus1, muxPC,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `brz2: begin
       out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,memRD,noLoad};
       nextState = `brz3;
    end
    `brz3: begin
       out = {fA, muxMDR,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `stop: begin
       out = {8'bxx,destNone, noLoad, noLoad, noLoad}; // same as above
       nextState = `stop; // This is to avoid a latch
       // quick 347-type debug: dump the register file:
       $display("%0d cycles", (($time+10)/20));
       w = 1;
       #1 $finish;
       //$stop;  // have to delay to let the register dump happen
    end
    `brc: begin
       out = {fA, muxPC,2'bxx,destMAR,noLoad,noLoad,noLoad};
       if (CCin[2])
         nextState = `brc2;
       else
         nextState = `brc1;
    end
    `brc1: begin
       out = {fAplus1, muxPC,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
```

```
    end
    `brc2: begin
       out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,memRD,noLoad};
       nextState = `brc3;
    end
    `brc3: begin
       out = {fA, muxMDR,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `brv: begin
       out = {fA, muxPC,2'bxx,destMAR,noLoad,noLoad,noLoad};
       if (CCin[0])
         nextState = `brv2;
       else
         nextState = `brv1;
    end
    `brv1: begin
       out = {fAplus1, muxPC,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `brv2: begin
       out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,memRD,noLoad};
       nextState = `brv3;
    end
    `brv3: begin
       out = {fA, muxMDR,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    // logical functions:
    `andOp: begin
       out = {fAandB,muxReg,muxReg,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    `notOp: begin
       out = {fAnot,muxReg,2'bxx,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    `orOp: begin
       out = {fAorB,muxReg,muxReg,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    `xorOp: begin
       out = {fAxorB,muxReg,muxReg,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    // illogical func--er, Comparison functions:
    `cmi: begin
       out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
       nextState = `cmi1;
    end
    `cmi1: begin
       out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
       nextState = `cmi2;
    end
    `cmi2: begin
       out = {fAminB, muxReg, muxMDR, destNone, loadCC, noLoad, noLoad};
       nextState = `fetch;
    end
    `cmr: begin
       out = {fAminB,muxReg,muxReg,destNone,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
```

```
    // Shift functions:
    `ashr: begin
       out = {fAashr,muxReg,2'bxx,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    `lshl: begin
       out = {fAshl,muxReg,2'bxx,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    `lshr: begin
       out = {fAlshr,muxReg,2'bxx,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    `rol: begin
       out = {fArol,muxReg,2'bxx,destReg,loadCC,noLoad,noLoad};
       nextState = `fetch;
    end
    // Data movement functions:
    `mov: begin
       out = {fB, 2'bxx, muxReg,destReg,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `lda: begin
       out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
       nextState = `lda1;
    end
    `lda1: begin
       out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
       nextState = `lda2;
    end
    `lda2: begin
       out = {fA, muxMDR, 2'bxx, destMAR, noLoad, noLoad, noLoad};
       nextState = `lda3;
    end
    `lda3: begin
       out = {4'bxxxx, 2'bxx, 2'bxx, destNone, noLoad, memRD, noLoad};
       nextState = `lda4;
    end
    `lda4: begin
       out = {fA, muxMDR, 2'bxx, destReg, loadCC, noLoad, noLoad};
       nextState = `fetch;
    end
    `sta: begin
       out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
       nextState = `sta1;
    end
    `sta1: begin
       out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
       nextState = `sta2;
    end
    `sta2: begin
       out = {fA, muxMDR, 2'bxx, destMAR, noLoad, noLoad, noLoad};
       nextState = `sta3;
    end
    `sta3: begin
       out = {fB, 2'bxx, muxReg, destMDR, noLoad, noLoad, noLoad};
       nextState = `sta4;
    end
    `sta4: begin
       out = {4'bxxxx, 2'bxx, 2'bxx, destNone, noLoad, noLoad, memWR};
       nextState = `fetch;
    end
```

```
    `str: begin
       out = {fA, muxReg,2'bxx,destMAR,noLoad,noLoad,noLoad};
       nextState = `str1;
    end
    `str1: begin
       out = {fB, 2'bxx,muxReg,destMDR,noLoad,noLoad,noLoad};
       nextState = `str2;
    end
    `str2: begin
       out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,noLoad,memWR};
       nextState = `fetch;
    end
    // Stack based ops:
    `jsr: begin
       out = {fAmin1, muxSP,2'bxx,destSP,noLoad,noLoad,noLoad};
       nextState = `jsr1;
    end
    `jsr1: begin
       out = {fA, muxSP,2'bxx,destMAR,noLoad,noLoad,noLoad};
       nextState = `jsr2;
    end
    `jsr2: begin
       out = {fAplus1, muxPC,2'bxx,destMDR,noLoad,noLoad,noLoad};
       nextState = `jsr3;
    end
    `jsr3: begin
       out = {fA, muxPC, 2'bxx, destMAR,noLoad,noLoad,memWR};
       nextState = `jsr4;
    end
    `jsr4: begin
       out = {4'bxxxx, 2'bxx,2'bxx,destNone,noLoad,memRD,noLoad};
       nextState = `jsr5;
    end
    `jsr5: begin
       out = {fA, muxMDR,2'bxx,destPC,noLoad,noLoad,noLoad};
       nextState = `fetch;
    end
    `ldsf: begin
       out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
       nextState = `ldsf1;
    end
    `ldsf1: begin
       out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
       nextState = `ldsf2;
    end
    `ldsf2: begin
       out = {fAplusB, muxMDR, muxSP, destMAR, noLoad, noLoad, noLoad};
       nextState = `ldsf3;
    end
    `ldsf3: begin
       out = {4'bxxxx, 2'bxx,2'bxx,destNone, noLoad, memRD, noLoad};
       nextState = `ldsf4;
    end
    `ldsf4: begin
       out = {fA, muxMDR, 2'bxx, destReg, noLoad, noLoad, noLoad};
       nextState = `fetch;
    end
    `ldsp: begin
       out = {fA, muxReg, 2'bxx, destSP, noLoad, noLoad, noLoad};
       nextState = `fetch;
    end
    `pop: begin
```

```
      out = {fA, muxSP, 2'bxx, destMAR, noLoad, noLoad, noLoad};
      nextState = `pop1;
   end
   `pop1: begin
      out = {fAplus1, muxSP, 2'bxx, destSP, noLoad, memRD, noLoad};
      nextState = `pop2;
   end
   `pop2: begin
      out = {fA, muxMDR, 2'bxx, destReg, noLoad, noLoad, noLoad};
      nextState = `fetch;
   end
   `push: begin
      out = {fAmin1, muxSP, 2'bxx, destMAR, noLoad, noLoad, noLoad};
      nextState = `push1;
   end
   `push1: begin
      out = {fA, muxReg, 2'bxx, destMDR, noLoad, noLoad, noLoad};
      nextState = `push2;
   end
   `push2: begin
      out = {fAmin1, muxSP, 2'bxx, destSP, noLoad, noLoad, memWR};
      nextState = `fetch;
   end
   `rtn: begin
      out = {fA, muxSP, 2'bxx, destMAR, noLoad, noLoad, noLoad};
      nextState = `rtn1;
   end
   `rtn1: begin
      out = {fAplus1, muxSP, 2'bxx, destSP, noLoad, memRD, noLoad};
      nextState = `rtn2;
   end
   `rtn2: begin
      out = {fA, muxMDR, 2'bxx, destPC, noLoad, noLoad, noLoad};
      nextState = `fetch;
   end
   `stsf: begin
      out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
      nextState = `stsf1;
   end
   `stsf1: begin
      out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
      nextState = `stsf2;
   end
   `stsf2: begin
      out = {fAplusB, muxMDR, muxSP, destMAR, noLoad, noLoad, noLoad};
      nextState = `stsf3;
   end
   `stsf3: begin
      out = {fA, muxReg, 2'bxx, destMDR, noLoad, noLoad, noLoad};
      nextState = `stsf4;
   end
   `stsf4: begin
      out = {4'bxxxx, 2'bxx, 2'bxx, destNone, noLoad, noLoad, memWR};
      nextState = `fetch;
   end
   `addsp: begin
      // get IMM value by putting PC on MAR:
      out = {fA, muxPC, 2'bxx, destMAR, noLoad, noLoad, noLoad};
      nextState = `addsp1;
   end
   `addsp1: begin
      // add one to PC and store back, read memory from PC:
```

```
       out = {fAplus1, muxPC, 2'bxx, destPC, noLoad, memRD, noLoad};
       nextState = `addsp2;
    end
    `addsp2: begin
       // add MDR with SP and store into SP:
       out = {fAplusB, muxMDR, muxSP, destSP, noLoad, noLoad, noLoad};
       nextState = `fetch;
    end
    //`addsp3: begin
       // put content of selected register on MDR:
       //out = {fA, muxReg, 2'bxx, destMDR, noLoad, noLoad, noLoad};
       // put contents of MAR into SP:
       //nextState = `fetch;
    //end
    `stsp: begin
       out = {fA, muxSP, 2'bxx, destReg, noLoad, noLoad, noLoad};
       nextState = `fetch;
    end
    `neg: begin
       out = {fAnot, muxReg, 2'bxx, destReg, noLoad, noLoad, noLoad};
       nextState = `neg1;
    end
    `neg1: begin
       out = {fAplus1, muxReg, 2'bxx, destReg, noLoad, noLoad, noLoad};
       nextState = `fetch;
    end
    default: begin
       out = 14'bx;
       nextState = `fetch;
    end
      endcase
   end // always @ (currState or CCin or IRIn)

endmodule
```