CS 3220
Multi-Cycle (CDB-Based) Implementation
Remember LC-2200?

- We need something like this!
Design Strategy

• First build (and test) each non-trivial block
  – ALU, Memory, Registers
  – IR and PC are trivial (just a single register)
• Add all the latches, Ld??? And Dr??? Signals
  – Relatively simple for latches and Ld???
  – For Dr???, we’ll use a “tri”-type wire as a bus
• Wire everything to the bus
• Design control logic
  – Design state machine
  – Figure out how to do dispatch(), test branch conditions
  – Program it all into Verilog
• Voila!
  – Yeah, right!
Data Memory

(* ram_init_file = DMEMINITFILE *)
reg [(DBITS-1):0] dmem[(DMEMWORDS-1):0];
wire MemEnable=! (MAR[(DBITS-1):DMEMADDRBITS]);
wire MemWE=lock&WrMem&MemEnable;
always @(posedge clk)
    if (MemWE)
        dmem[MAR[(DMEMADDRBITS-1):DMEMWORDBITS]]<=memin;
wire [(DBITS-1):0] MemVal=
    MemWE?
    {DBITS{1'bX}}:
    dmem[MAR[(DMEMADDRBITS-1):DMEMWORDBITS]];

Syntactically correct, but can it be correctly synthesized?

It depends!
Yes, if ADDR comes from a flipflop and MemVal goes to a flipflop!
Otherwise, often cannot...
E.g. can’t synthesize when ADDR comes from SW, KEY, some logic, etc.
Using memory on our FPGA chip

- Two types of state bits available
  - Registers, can wire them up in (almost) any way we want and implement (almost) any kind of state
  - Memory, has a pre-defined (memory-like) behavior
- Compiler decides which to use
  - Memory, if Verilog code matches its behavior
  - Registers, in all other situations
- So what’s wrong with testing our memory this way:
  
  ```vhdl
  always @(posedge CLK)
  if(WE)
    mem[!KEY] <= SW;
  assign LEDR = mem[!KEY]; // Continuous output
  ```
- Does not match the behavior of the built-in memory!
This is our MEM module design.
But built-in memory is synchronous

• Reads happen each cycle
  – Not continuous reading like we are used to
  – It has a built-in FF for the read address
• When ADDR comes from FF
  – The FF from design mapped onto built-in FF
  – Can’t do this if ADDR does not come from a FF
Pass-through logic?

- Our design will use a MAR register (FFs)
  - Synchronous memory will work fine with this
- But… Quartus complains about pass-through logic
  - Works correctly, but expensive and slow
- Pass-through logic?
  - Write to “mem” starts at end of cycle in which WE=1
    - Because WE, WADDR, DIN latched into internal FFs first
    - But should be happening during the cycle in this WE=1
  - Reading continuous (every cycle)
    - Using RADDR latched into internal FF that shadows MAR
  - What do we read in the cycle in which we write?
    - Old value or new value?
    - Synthesizer fixes this – adds logic to get new value to output
• We never read and write in the same cycle
  – Fetch in one cycle, then a few cycles, then
  – If LW, read memory, go back to FETCH1
  – If SW, write memory, go back to FETCH1
  – We don’t care what MEM outputs during write!
    • Don’t need the pass-through logic

• Removing the pass-through logic
  – Try letting the Verilog compiler know that we don’t care
    assign DOUT = WE?{DBITS{1'bX}}:mem[ADDR];
    • Still adding the pass-through logic…
  – OK, try a dirty trick…
    • Assignments -> Settings, then select Analysis & Synthesis Settings
    • Click “More Settings”, then find “Add Pass-Through Logic …” option
    • Set it to Off
Other things we’ll need

- Registers
  - Same as memory
  - Use regno as “address”, have only 8 words
  - Note: regno will come from a state machine (not FF)
    - But our registers will be synthesized correctly! Why?

- PC, IR, etc.
  - Almost trivial - each is just a register
tri [32:0] thebus;
assign thebus = DrALU ? ALUval : 32’hzzzzzzzzzz;
assign thebus = DrPC ? PCval : 32’hzzzzzzzzzz;

• Note: trior doesn’t work!
  – Why? Mystery (it should)
  – But tri can’t damage anything anyway:

• There are no internal 3-state drivers in our chip
  – Supported only on external pins, but
  – Quartus II happily converts this code to a selector
parameter BUSZ={DBITS{1'bZ}};
reg [(DBITS-1):0] MAR;  // MAR register  
/* Connect MAR input to the bus */
always @(posedge clk or posedge reset)
  if(reset)
    MAR<=32'b0;
  else if(LdMAR)
    MAR<=thebus;

/* Create PC and connect it to the bus */
reg [(DBITS-1):0] PC;
reg LdPC, DrPC, IncPC;
always @(posedge clk) begin
  if(reset)
    PC<=STARTPC;
  else if(LdPC)
    PC<=thebus;
  else if(IncPC)
    PC<=PC+INSTSIZE;
end
assign thebus=DrPC?PC:BUSZ;
reg signed [31:0] ALUout;
reg [5:0] ALUfunc;
always @(ALUfunc or A or B)
    case(ALUfunc)
        OP2_AND:  ALUout=A&B;
        OP2_OR:   ALUout=A|B;
        ...
        OP2_ADD:  ALUout=A+B;
        OP2_SUB:  ALUout=A-B;
        OP2_EQ:   ALUout=(A==B);
        ...
        OP2_LOHI: ALUout={B[31:16],A[15:0]};
        default:  ALUout={32{1'bX}};
    endcase
assign thebus=DrALU?ALUout:BUSZ;

// A and B registers
reg signed [31:0] A,B;
reg LdA,LdB,DrALU;
always @(posedge clk)
    begin
        if(LdA)
            A <= thebus;
        if(LdB)
            B <= thebus;
    end
parameter S_BITS=5;
parameter [(S_BITS-1):0] 
  S_ZERO  ={(S_BITS){1'b0}},
  S_ONE   ={{(S_BITS-1){1'b0}},1'b1},
  S_FETCH1=S_ZERO,
  S_FETCH2=S_FETCH1+S_ONE,
  S_ALUR1 =S_FETCH2+S_ONE,
...
reg [(S_BITS-1):0] state,next_state;
always @(state or opcode1 or rx or ry or rz or ...) begin
  next_state=state+S_ONE;
  case(state)
    S_FETCH1: {LdIR,IncPC}={1'b1,1'b1};
    S_FETCH2: case(opcode1)
      ...
    endcase
    S_ALUR1: {regno,DrReg,LdB}={ry,1'b1,1'b1};
    ...
  endcase
end
always @(posedge clk or posedge reset)
  if(reset) state<=S_FETCH1;
  else state<=next_state;
Performance depends on several factors
- Clock frequency
- Our state machine (how many cycles each inst takes)
- Instruction mix (which insts are used more often)

Design complexity affected by some of these, too
- May need extra logic to improve frequency
- May need extra logic to remove limitations
- Fancier state machine may use more state or logic