TCES 330

Lecture 14

When you use the Compiler to perform a full compilation, the Compiler uses a series of modules to process the design and, ultimately, to create one or more programming files.

- Analysis & Synthesis, which first checks the design files and the overall design for errors, and builds a single design database that integrates all the design files in a design hierarchy. Analysis & Synthesis performs logic synthesis to minimize the logic of the design, and performs technology mapping to implement the design logic using device resources such as logic elements. Analysis & Synthesis provides comprehensive Verilog HDL and VHDL language support..

- The Fitter, which fits (that is, places and routes) the logic of a design into a device. You must perform analysis and synthesis on a design before using the Fitter. When you use the Fitter on a design, the Fitter runs an Auto Fit compilation by default. In Auto Fit mode, the Fitter attempts to meet your timing constraints and not to beat them; it automatically decreases compilation time by turning off some optimization steps if they are not required for the design to meet the timing constraints. You can also decrease compilation time by directing the Fitter to reduce Fitter effort after meeting a design’s timing requirements, or to make only one fitting attempt.

What Quartus ‘Compile’ Mean?

- The Assembler, which converts the Fitter’s device, logic cell, and pin assignments into a programming image for the device. The Assembler is the Compiler module that completes project processing by converting the Fitter’s device, logic cell, and pin assignments into a device programming image, in the form of one or more Programmer Object Files (.pof), SRAM Object Files (.sof), Hexadecimal (Intel-Format) Output Files (.hexout), Tabular Text Files (.ttf), and Raw Binary Files (.rbf), from a successful fit.

- The Design Assistant, which checks the reliability of a design based on a set of design rules. You can use the Design Assistant after either performing Analysis & Synthesis or using the Fitter on a design.

- The TimeQuest Timing Analyzer, which analyzes, debugs, and validates the timing performance of all logic in a design. You must complete Analysis & Synthesis and Fitting on a design before performing a full timing analysis. However, you can run an early timing estimate to obtain preliminary timing data without completing fitting.

Quartus ‘Compile’, cont.

State Machines, Continued

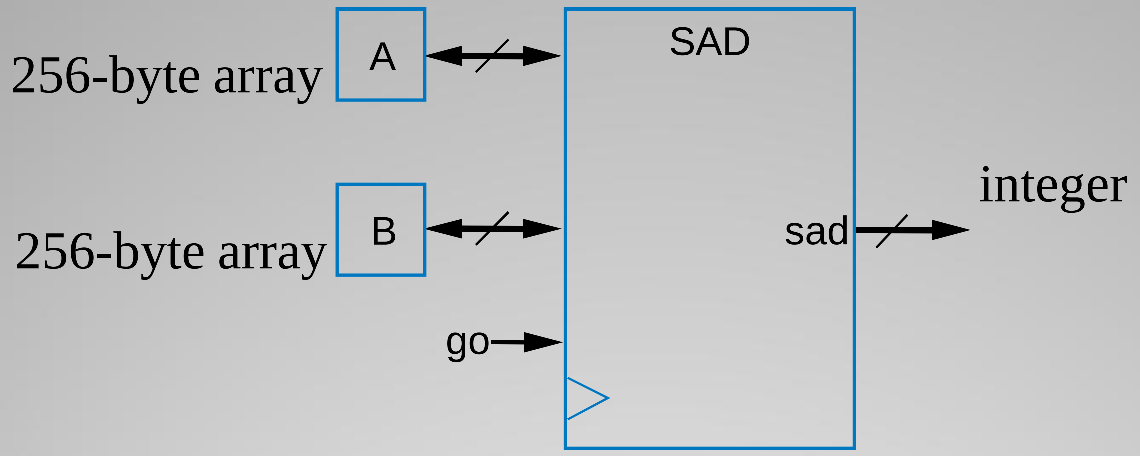
SAD Processor

Mythbusters Frame 1

Mythbusters Frame 2

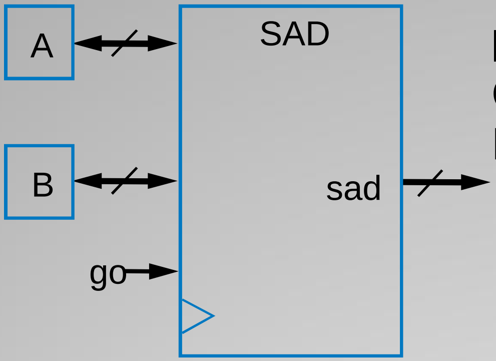
- Compare small patches, one frame to the next.
- If changes are minor, just send the difference between the patches.
- If changes are not minor, send the whole patch.
- ‘Patches’ are 16 x 16 pixel blocks (256 pixels).
- In our simplified example each pixel is one byte (8 bits).
- We’ll take the difference between each pair of pixels
- We’ll take the absolute value of this difference.
- We’ll sum up all these values for the patch (SAD).
- If this value is less than some threshold, we’ll just send the difference, otherwise the whole patch.

One Aspect of Video Compression

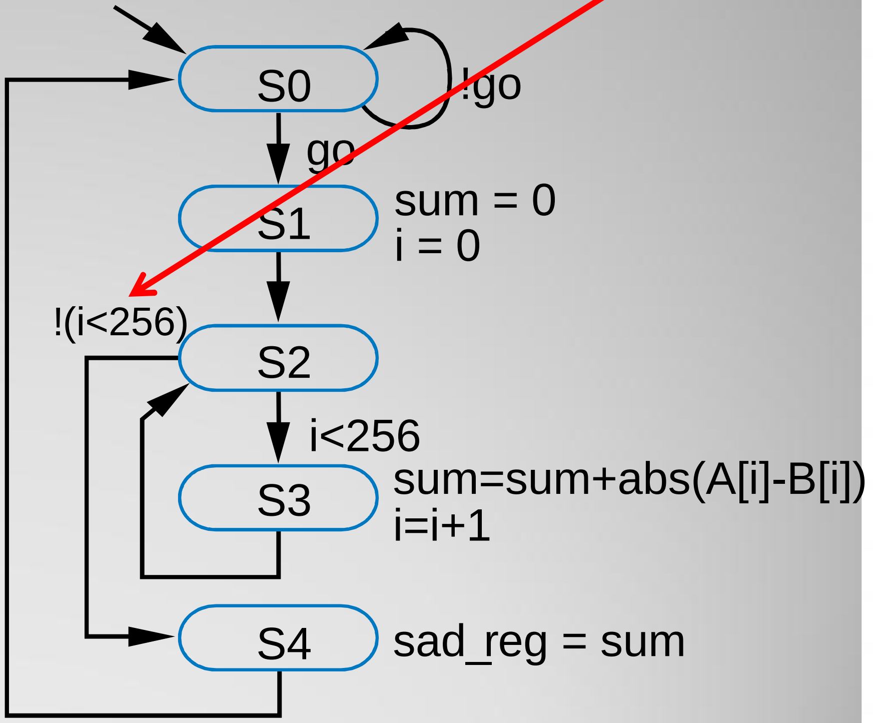


- Want fast sum-of-absolute-differences (SAD) component
  - When *go=1*, sums the absolute value of the differences of element pairs in arrays *A* and *B*, outputs that sum
  - $$sad = \sum_{0}^{255} |A_i - B_i|$$

SAD Circuit



Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)



- S0: wait for go
- S1: initialize sum and index
- S2: check if done (i>=256)
- S3: add difference to sum, increment index
- S4: done, write to output sad_reg

From F. Vahid's *Digital Design*

SAD State Machine

Verilog Tasks and Functions

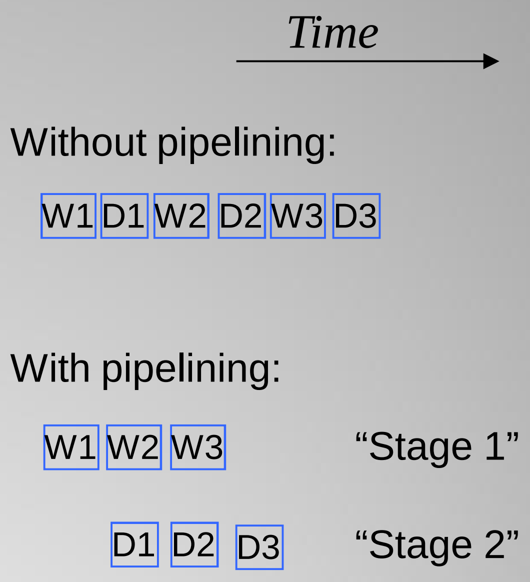
- Verilog Task
  - A task is declared by the keyword task and it comprises a block of statements that ends with the keyword endtask. The task must be included in the module that calls it. It may have input and output ports (these are not the ports of the module that contains the task). The task ports are used only to pass values between the module and the task.
- Verilog Function
  - A function is declared by the keyword function and it comprises a block of statements that ends with the keyword endfunction. The function must have at least one input and it returns a single value that is placed where the function is invoked.

```
module mux16to1 (W, S16, f);
   input [15:0] W;    // 16 inputs
   input [3:0]  S16; // 4 bits of select
   output reg f;     // 1 bit of output

   always @( W, S16 )
     case ( S16[3:2] )
         0: mux4to1( W[3:0],   S16[1:0], f );
         1: mux4to1( W[7:4],   S16[1:0], f );
         2: mux4to1( W[11:8],  S16[1:0], f );
         3: mux4to1( W[15:12], S16[1:0], f );
     endcase
   end // always
// Task that specifies a 4-to-1 multiplexer
task mux4to1;
   input [0:3] X;
   input [1:0] S4;
   output reg g;

   case (S4)
     0: g = X[0];
     1: g = X[1];
     2: g = X[2];
     3: g = X[3];
   endcase
endtask
endmodule
```

Use of a Task in Verilog

Pipelining

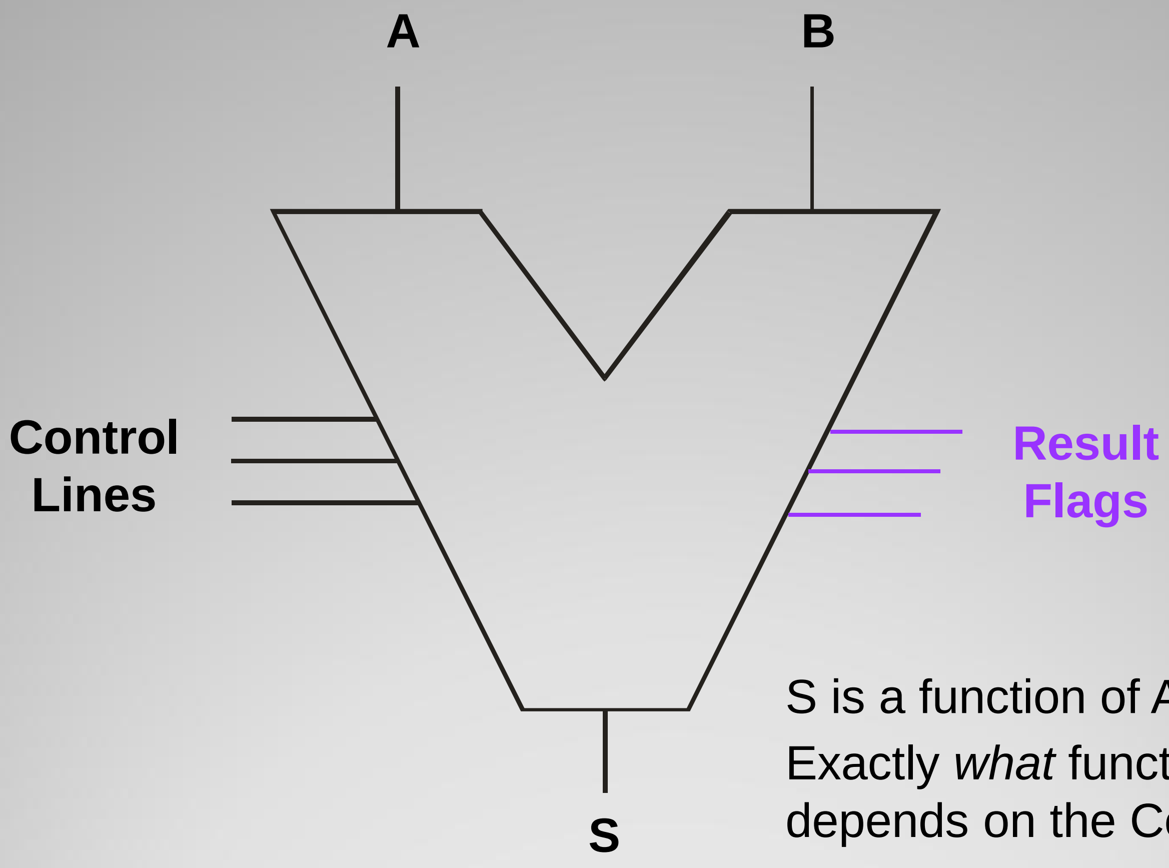
- Intuitive example: Washing dishes with a friend, you wash, friend dries
  - You wash plate 1
  - Then friend dries plate 1, while you wash plate 2
  - Then friend dries plate 2, while you wash plate 3; and so on
  - You don't sit and watch friend dry; you start on the next plate
- Pipelining: Break task into stages, each stage outputs data for next stage, all stages operate concurrently (if they have data)



More Datapath

Counters/Timers

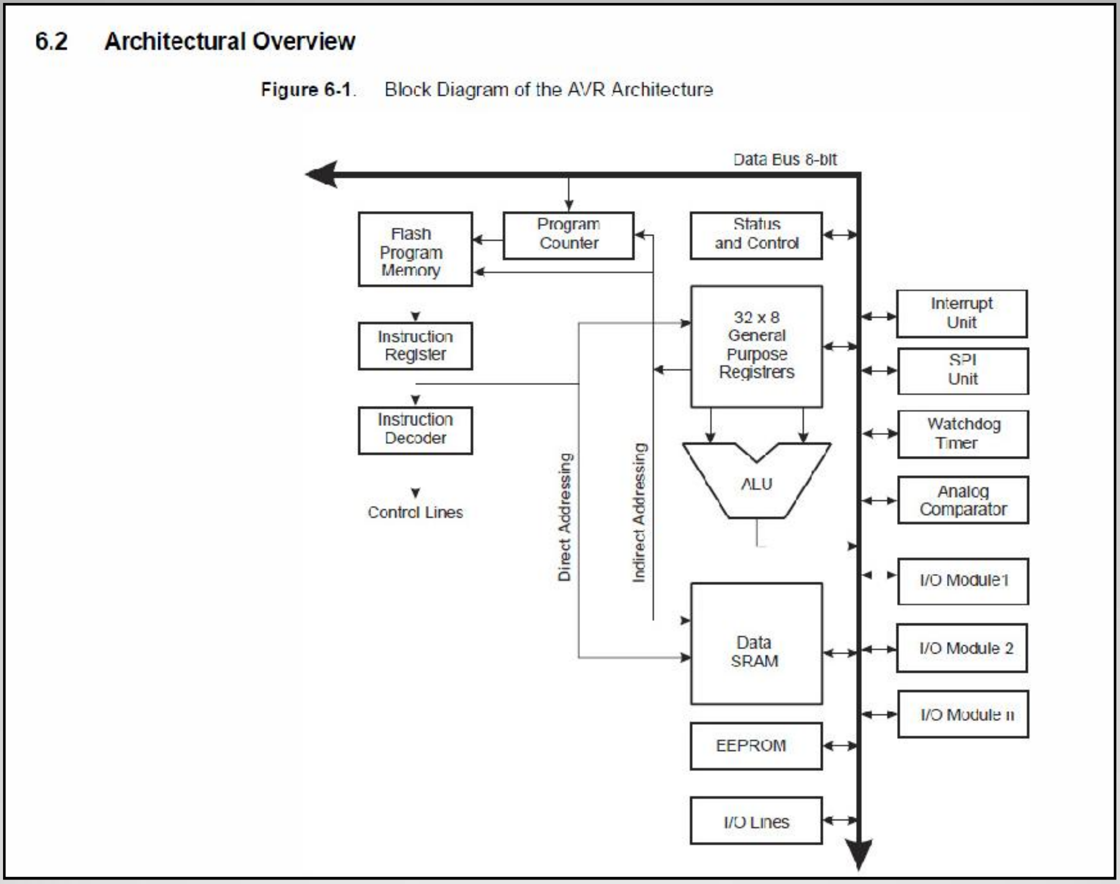
- What do we count? Clock pulses, always!
- Not necessarily <u>every</u> clock pulse.
- If we want to count the output pulses from some other circuit, we use those pulses to enable our clock pulse counter.
- We use counters to add delays or timers to our circuits
  - Not for-loops
  - Not while-loops
  - Not wait-statements
- All modern µProcessor controllers have built-in timers (counters).



S is a function of A and B

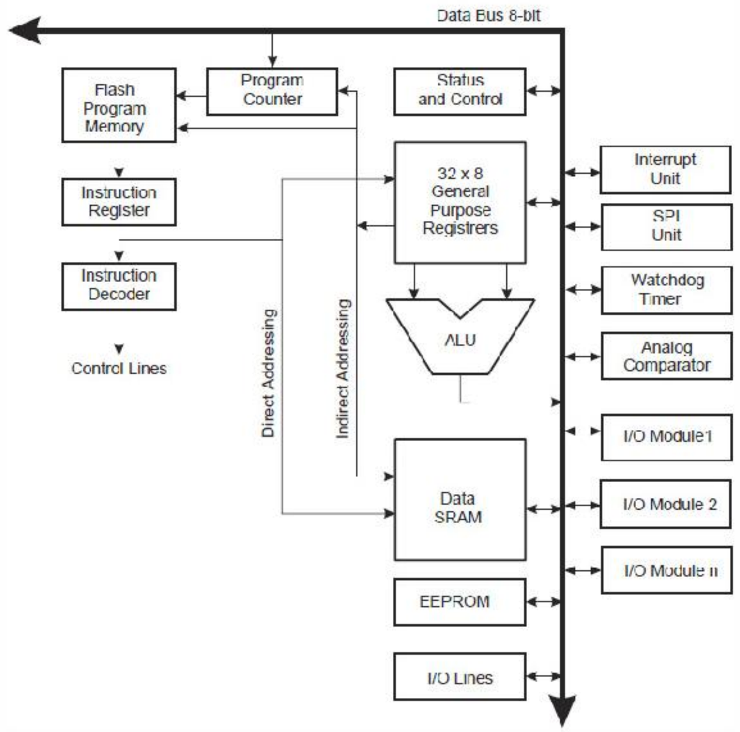
Exactly *what* function depends on the Control Lines

Arithmetic Logic Unit (ALU)



6.2 Architectural Overview

Figure 6-1. Block Diagram of the AVR Architecture



Atmel 168 Microprocessor

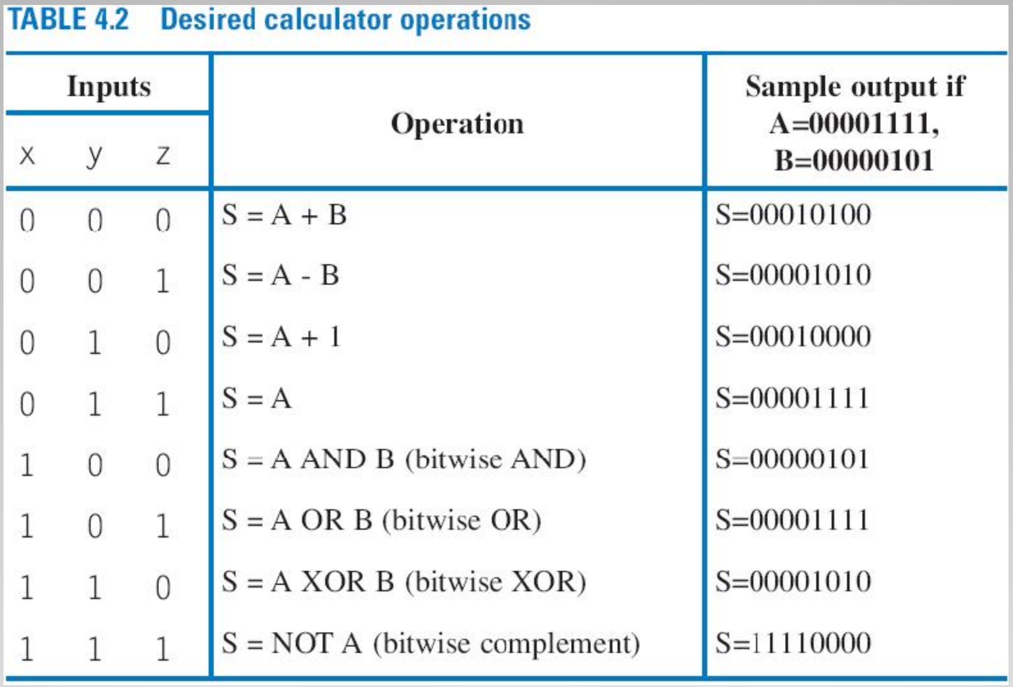


TABLE 4.2 Desired calculator operations

Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

Example Arithmetic Logic Unit

Simple ALU in Verilog

```
module ALU( Sel, A, B, S );
  input [2:0] Sel;  // function select
  input [7:0] A, B;    // input data
  output reg [7:0] S;  // ALU output (result)

  always @ * begin
    S = 0;  // default value
    case ( Sel )
      3'h0: begin
        S = A + B;  // ADD
      end
      3'h1: begin
        S = A - B;  // SUBTRACT
      end
      3'h2: begin
        S = A + 1'b1;  // INCREMENT
      end
      3'h3: begin
        S = A;  // FALL-THROUGH
      end
```

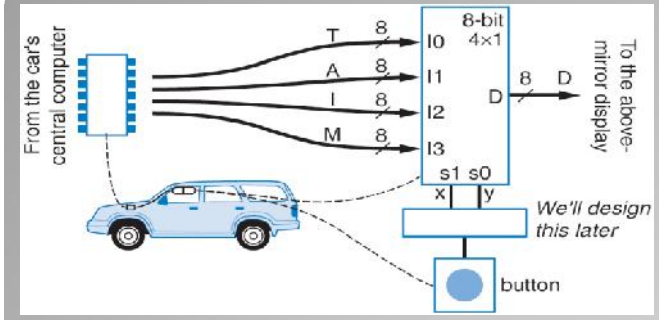
```
3'h4: begin
        S = A & B;  // bitwise AND
      end
      3'h5: begin
        S = A | B;  // bitwise OR
      end
      3'h6: begin
        S = A ^ B;  // bitwise XOR
      end
      3'h7: begin
        S = ~ A;  // bitwise complement
      end
    endcase
  end // always
endmodule
```

ALU Improvements

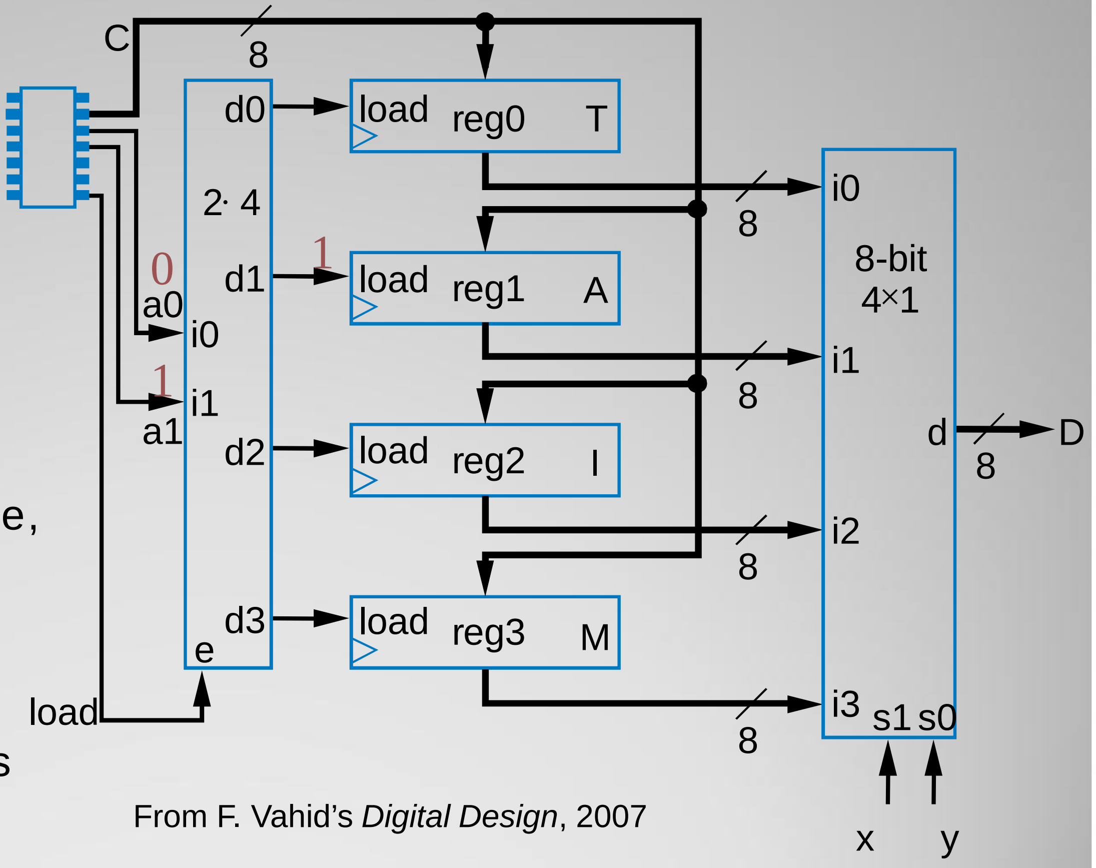
- We would write more general Verilog (N bits)
- We would output the usual flags
  - Zero
  - Overflow
  - Negative
  - Etc.
- Additional functions
  - Compare (signed and unsigned)
  - Shifts and Rotates left and right
  - Increment/decrement
  - ...

Register Files

Recall: Above-Mirror Display

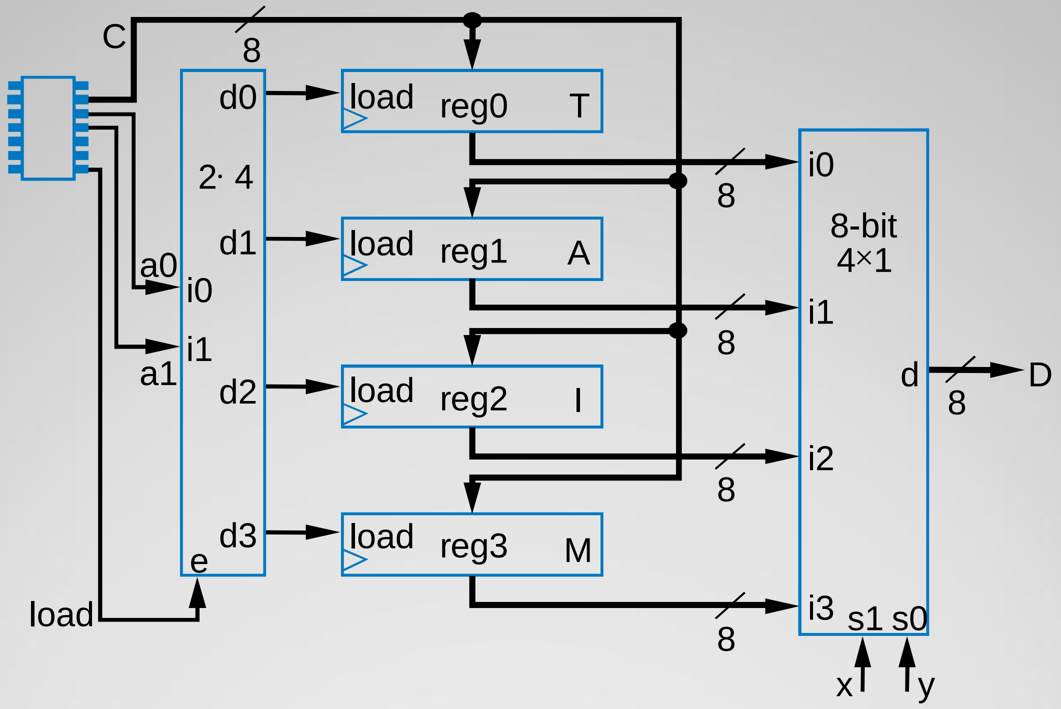


- Recall: Four simultaneous values from car's computer
- To reduce wires: Computer writes only 1 value at a time, loads into one of four registers
  - Was: 8+8+8+8 = 32 wires
  - Now: 8 +2+1 = 11 wires

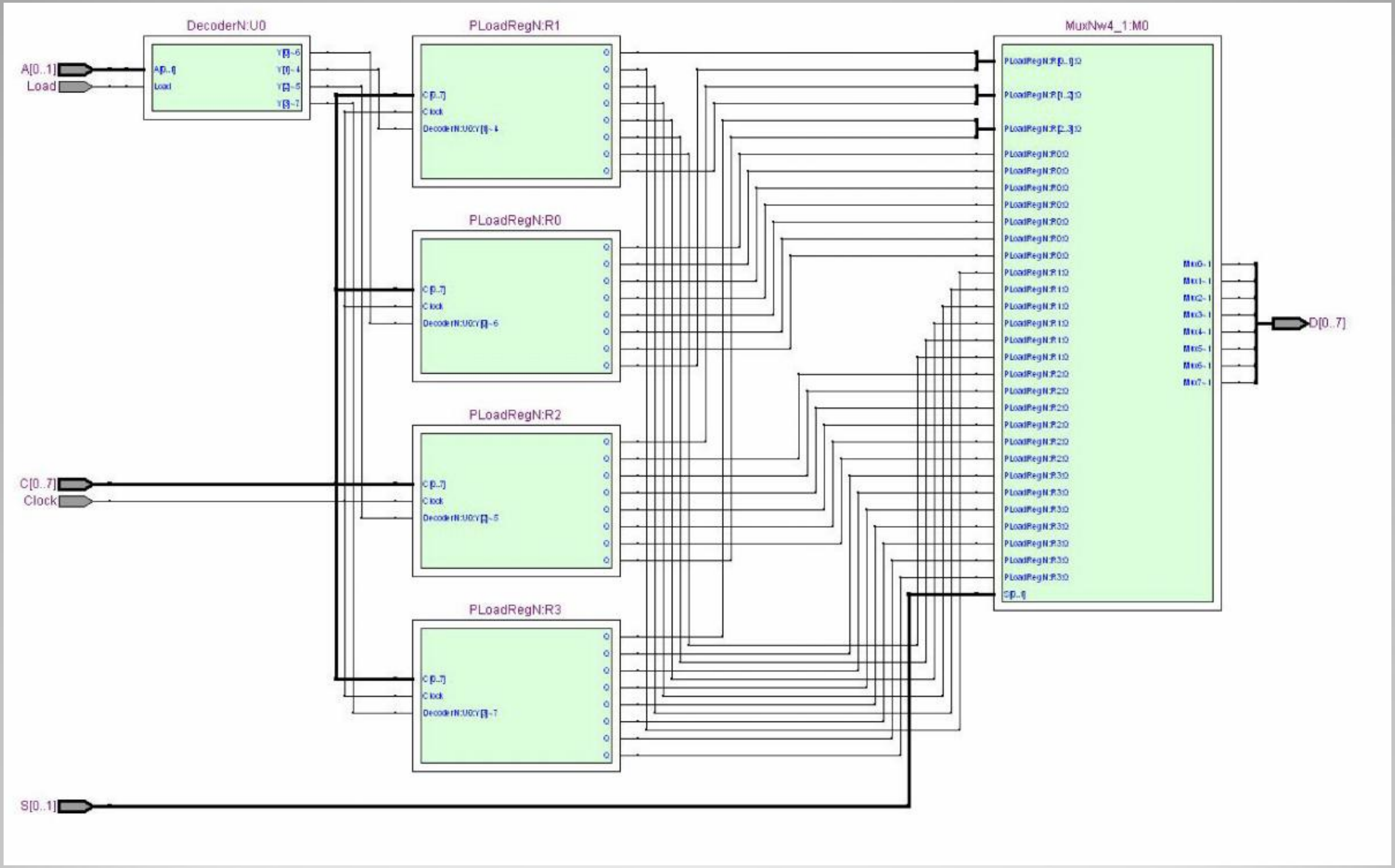


From F. Vahid's *Digital Design*, 2007

- Our solution to the over the mirror display problem
- This is called a register bank
- In this case we have four 8-bit registers
- <u>This design does not scale well</u>



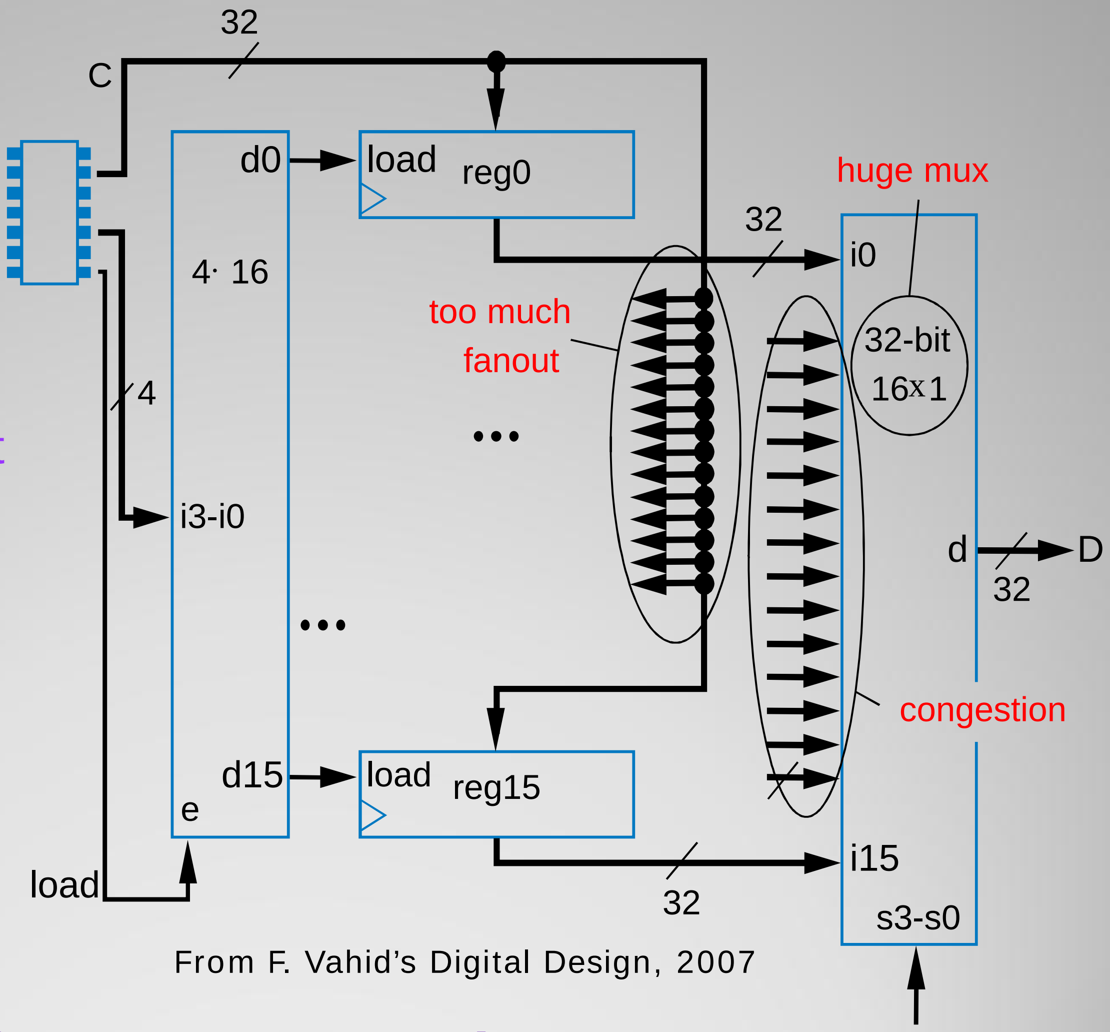
Register Bank



What Quartus Built

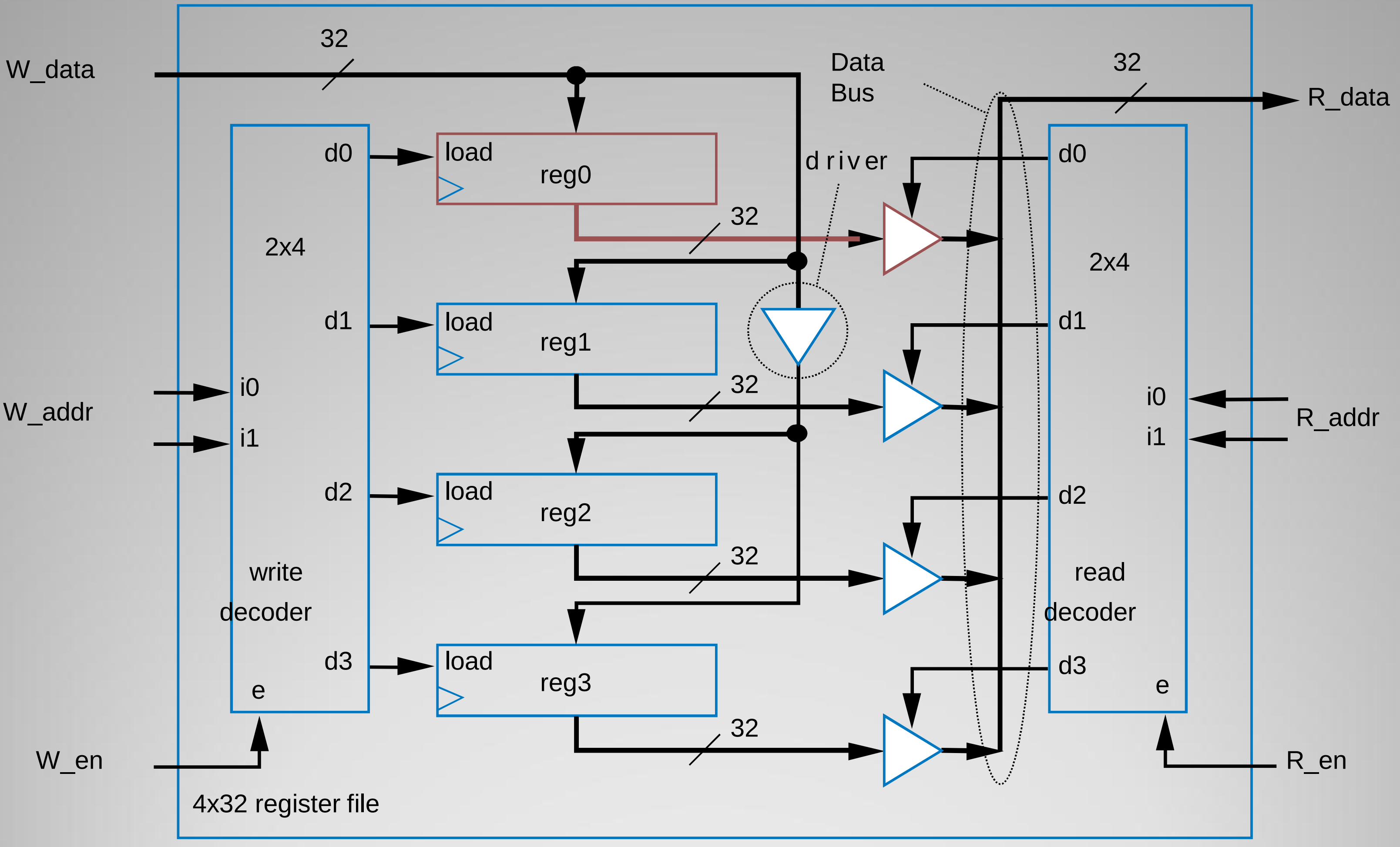
What if we had 16 32-bit registers to deal with?

- Lots of input fan-out
- Mux would have to have lots of input lines
- Large Muxes are slow



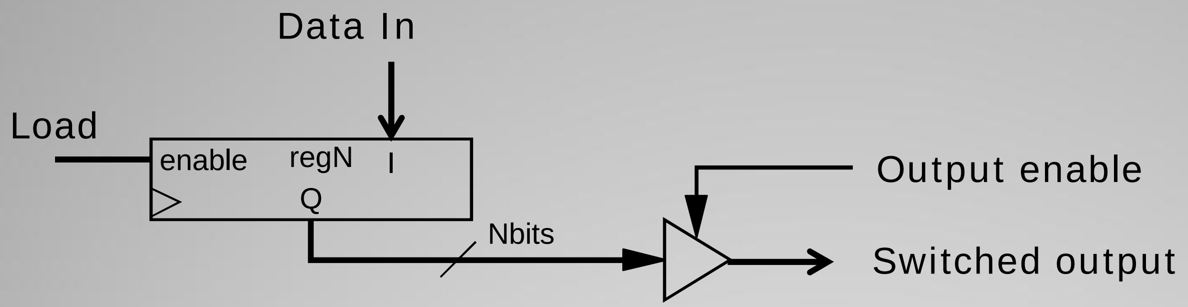
From F. Vahid's Digital Design, 2007

Wide Register Bank



From F. Vahid's Digital Design, 2007

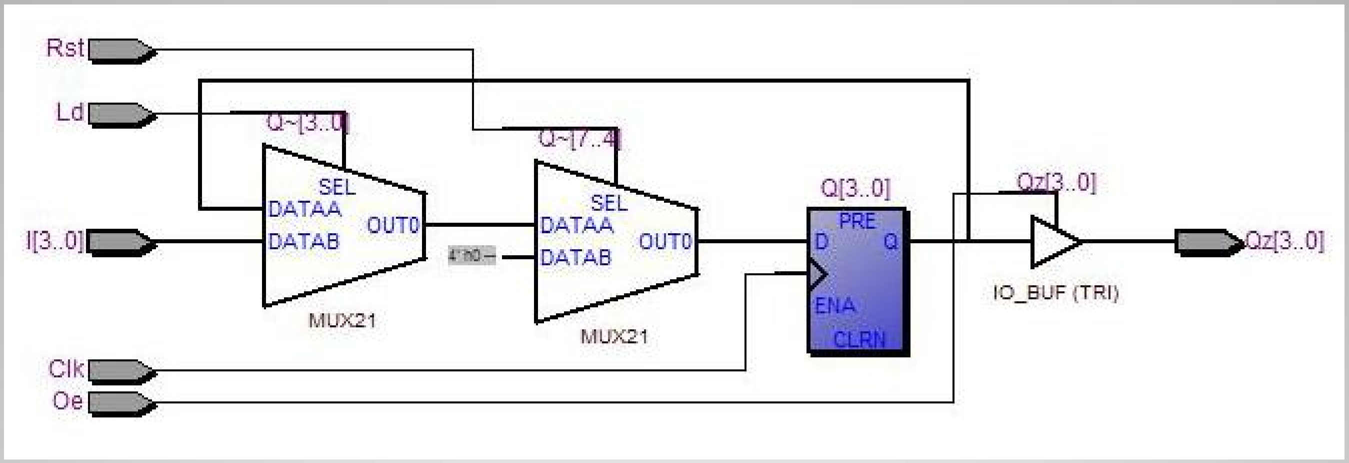
Register with Output Enable



Register with OE Detail

```
module RegisterOE4( Clk, Rst, Ld, I, Oe, Qz );
  input Clk;
  input Rst;
  input Ld;       // load signal
  input Oe;       // output enable (this is new)
  input [3:0] I;  // data to load

  output [3:0] Qz; // switched output
  reg [3:0] Q;     // standard register variable

  // Register Procedure
  always @(posedge Clk) begin
    if (Rst == 1)     // reset
      Q <= 0;
    else if (Ld == 1) // load
      Q <= I;
  end

  // Output
  assign Qz = Oe ? Q : 4'bzzzz;
endmodule
```

4-Bit Register with Output Enable



What Quartus Built

How do we generalize the RegisterOE module to an N-Bit Register?

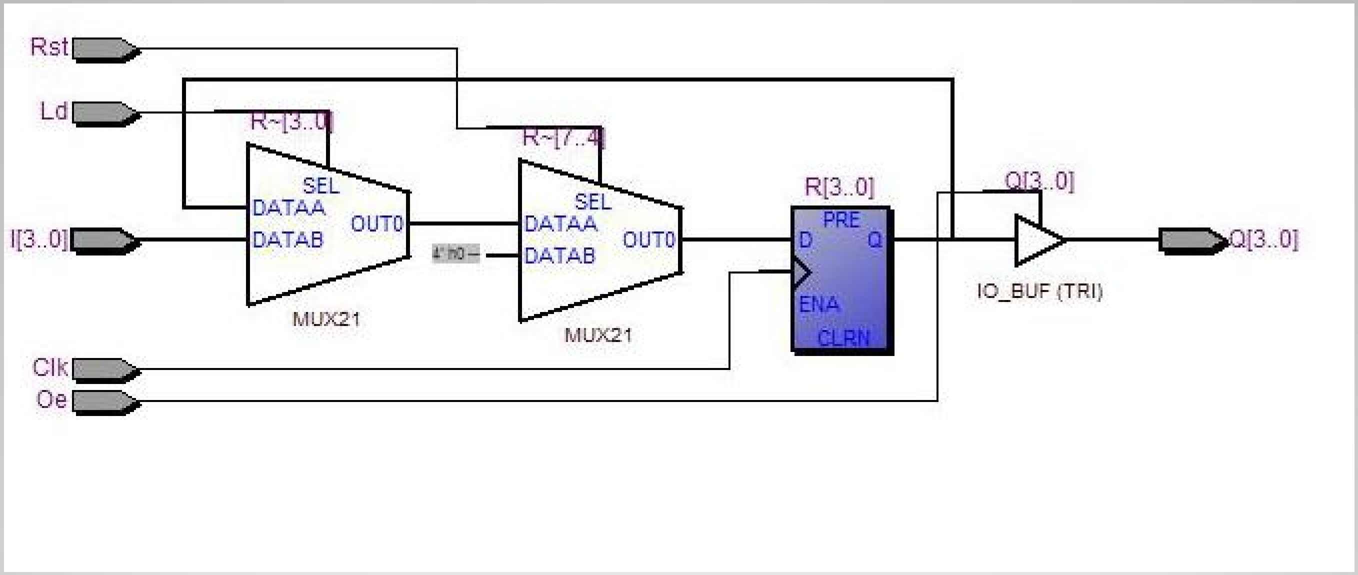
```
module RegisterOE( I, Ld, Oe, Clk, Rst, Q );
   input [31:0] I;   // data to load
   input Ld;         // load signal
   input Oe;         // output enable (this is new)
   input Clk, Rst;
   output [31:0] Q;   // our output
   reg [31:0] R;      // the basic register
   // Register Procedure
   always @(posedge Clk) begin
      if (Rst == 1)
         R <= 0;
      else if (Ld == 1)
         R <= I;
   end

   // Output
   assign Q = Oe ? R : 32'hZZZZZZZZ;

endmodule
```

These sizes must agree (and N'bZ is not valid)

Register with Output Enable

```
module RegisterOEN( Clk, Rst, Ld, I, Oe, Qz );
  parameter N = 4;

  input Clk;
  input Rst;
  input Ld;         // load signal
  input Oe;         // output enable (this is new)
  input [N-1:0] I;  // data to load

  output [N-1:0] Qz;  // switched output
  reg [N-1:0] Q;    // the basic register

  // Register Procedure
  always @(posedge Clk) begin
    if (Rst == 1)
      Q <= 0;
    else if (Ld == 1)
      Q <= I;
  end

  // Output
  assign Qz = Oe ? Q : {N{1'bZ}};

endmodule
```

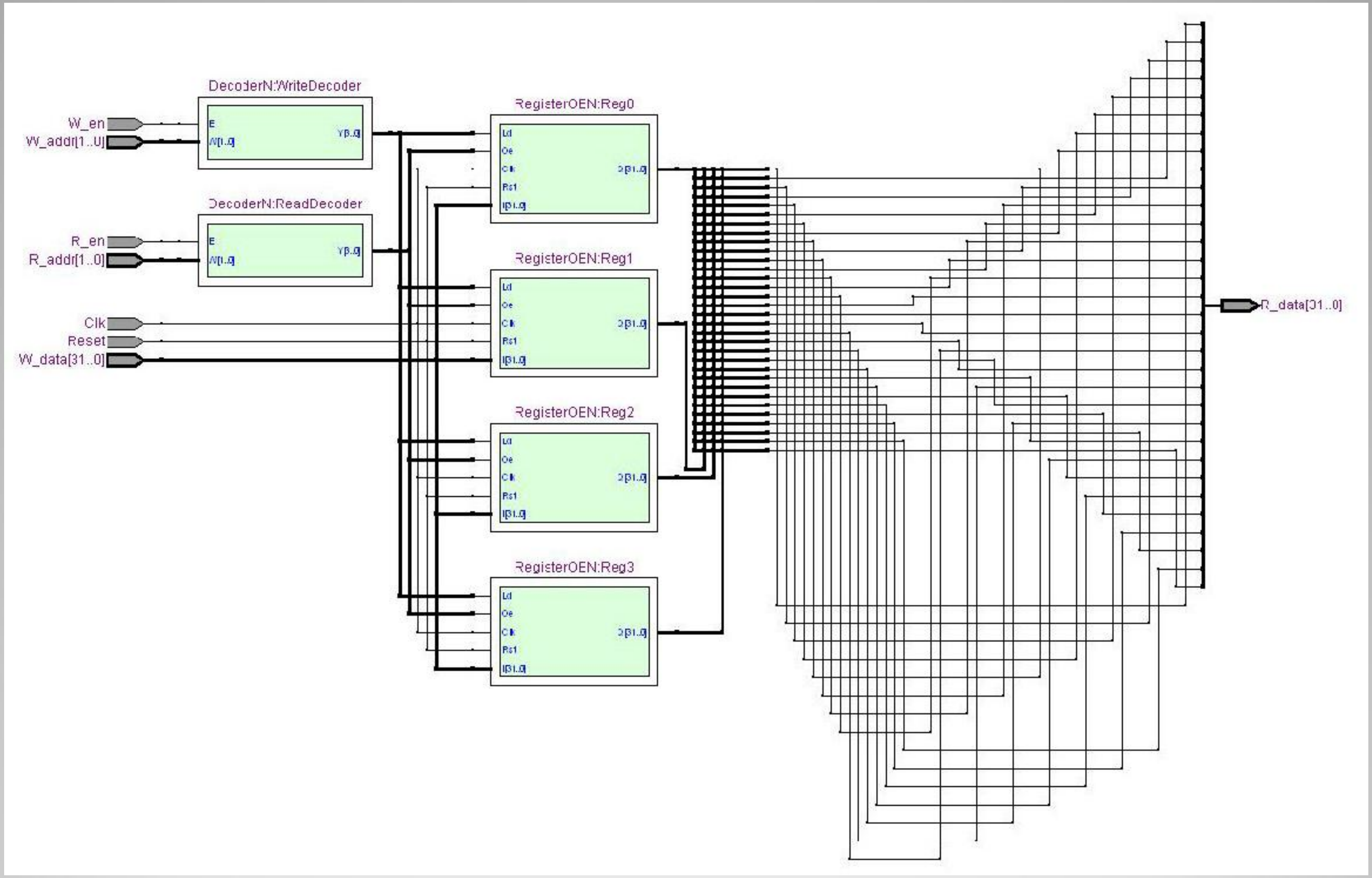
RegisterOEN



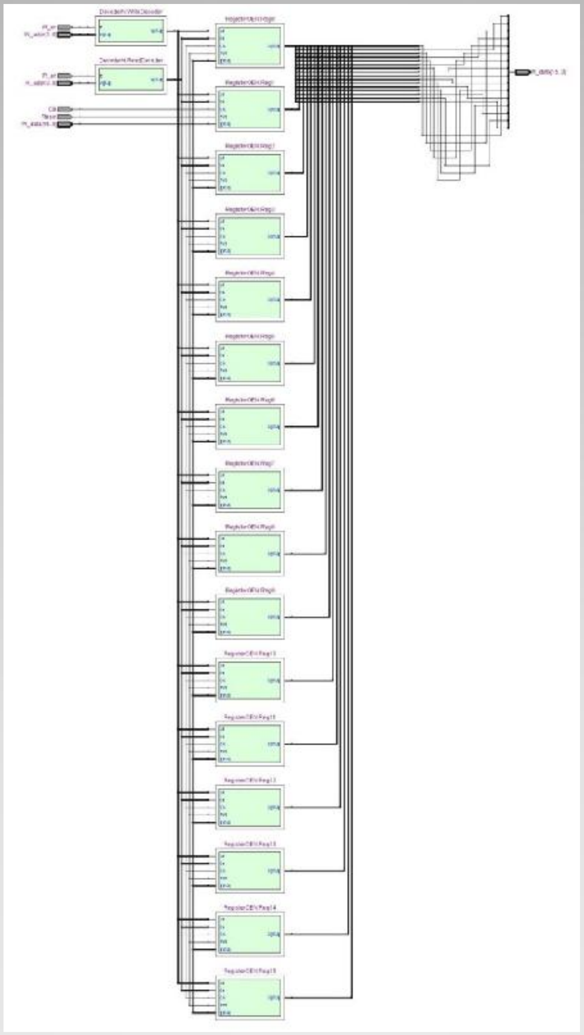
RegisterOEN with N = 4

Building the Register File

- For our register file we'll need:
  - Decoder for write
  - Decoder for read
  - Registers with output enable
  - Drivers (Quartus will take care of these for us)



Register File (4 Reg X 32 Bits)

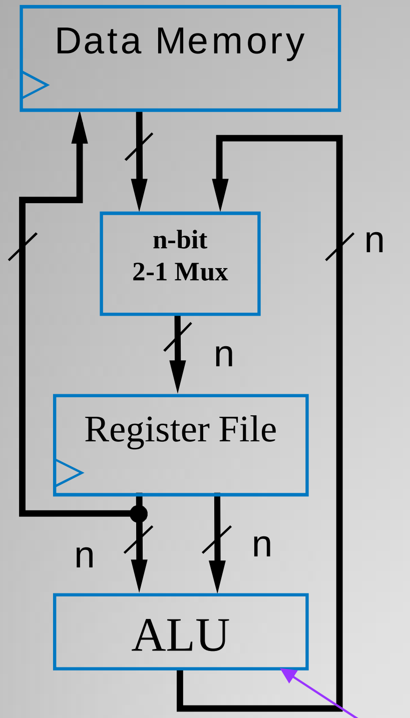


Register File (16 Reg X 16 Bits)

```
Fitter Status : Successful - Wed May 12 12:54:15 2010
Quartus II Version : 9.0 Build 132 02/25/2009 SJ Full Version
Revision Name : RegisterFile
Top-level Entity Name : RegisterFile
Family : Cyclone II
Device : EP2C35F672C6
Timing Models : Final
Total logic elements : 356 / 33,216 ( 1 % )
    Total combinational functions : 260 / 33,216 ( < 1 % )
    Dedicated logic registers : 256 / 33,216 ( < 1 % )
Total registers : 256
Total pins : 44 / 475 ( 9 % )
Total virtual pins : 0
Total memory bits : 0 / 483,840 ( 0 % )
Embedded Multiplier 9-bit elements : 0 / 70 ( 0 % )
Total PLLs : 0 / 4 ( 0 % )
```

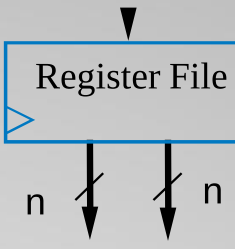
Register File (16 Reg X 16 Bits) Fit Summary

Our Programmable Processor's Datapath

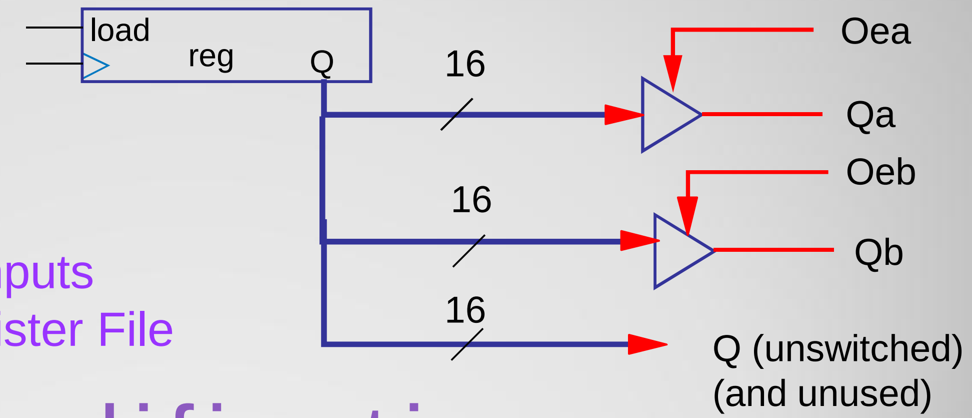


ALU has two inputs
fed by the Register File

So, our Register File needs two independent outputs.



We can accomplish this by adding a second register output line, controlled by a second output enable.



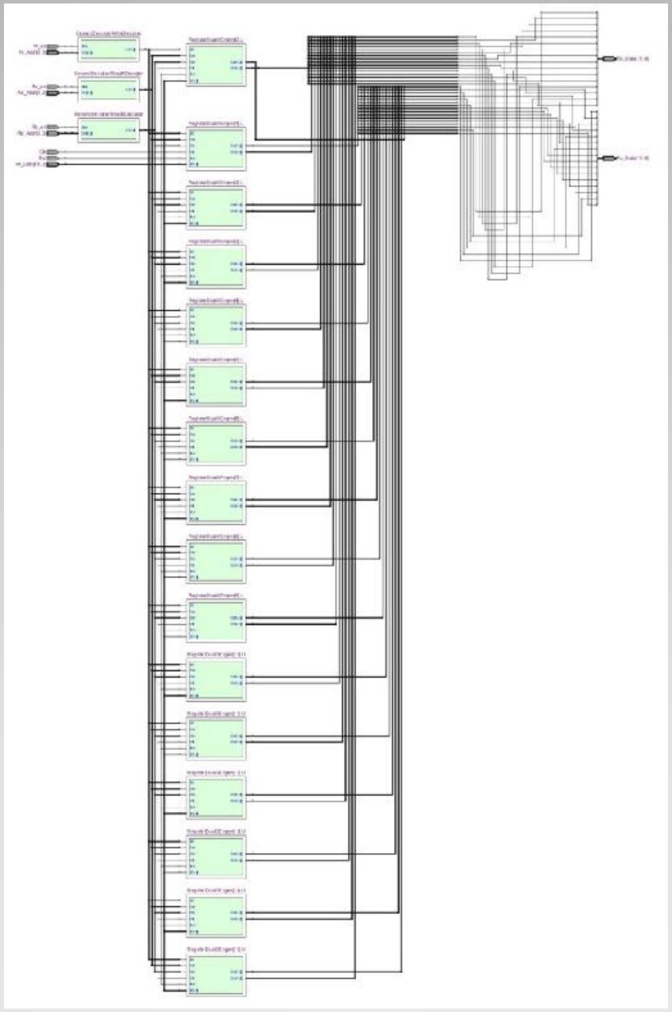
One More Modification

```
module RegisterDualOEN( I, Ld, Oea, Oeb, Clk, Rst, Qa, Qb );
  parameter N = 16;
  input [N-1:0] I;  // input data
  input Oea;       // channel a output enable
  input Oeb;       // channel b output enable
  input Ld;        // register load enable
  input Clk;       // input clock
  input Rst;       // synchronous reset signal

  output [N-1:0]Qa; // channel a output
  output [N-1:0]Qb; // channel b output

  // Your code goes here!
```

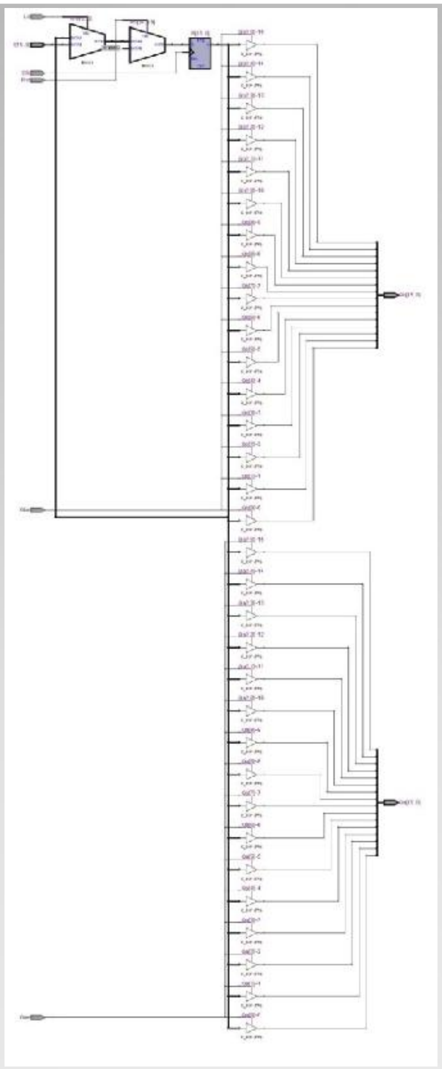
RegisterDualOEN

```
module RegisterFile(  Clk, Rst, W_en, W_Addr, W_Data, Ra_Addr, Rb_Addr, Ra_en, Rb_en, Ra_Data, Rb_Data );
  input Clk;                      // system clock
  input Rst;                      // system reset
  input W_en;                     // write enable
  input [3:0] W_Addr;             // write address
  input [15:0] W_Data;            // data to write
  input [3:0] Ra_Addr, Rb_Addr;   // A side and B side read addresses
  input Ra_en, Rb_en;             // A side and B side read enables

  output [15:0] Ra_Data, Rb_Data;    // A side and B side outputs (switched)
  output [15:0] RQ0;                 // Register 0 output (unswitched)

  wire [15:0] Ra_d , Rb_d; // read enables (decoder outputs)
  wire [15:0] W_d ;        // write enable (decoder output)
  wire [255:0] Q;          // holds the unswitched register outputs

  genvar I;  // we are going to generate the registers

  // instantiate two read decoders and a write decoder
  GenericDecoder #(.M(4)) WriteDecoder( W_Addr, W_en, W_d );    // write decoder
  GenericDecoder #(.M(4)) ReadADecoder( Ra_Addr, Ra_en, Ra_d ); // A side read decoder
  GenericDecoder #(.M(4)) ReadBDecoder( Rb_Addr, Rb_en, Rb_d ); // B side read decoder

  // instantiate the 16 dual-output registers
  generate
    for ( I=0; I<16; I=I+1 ) begin:rgen
      RegisterDualOE U( Clk, Rst, W_d[I], W_Data, Ra_d[I], Rb_d[I], Q[I*16+15:I*16], Ra_Data, Rb_Data );
    end
  endgenerate

endmodule
```

The Dual Output Register File



Register File with Dual Outputs



RegisterDualOE