Register Transfer Level

- Something between the logic level and the architecture level
- A convenient way to describe synchronous sequential systems
- State diagrams for pros
Hierarchy of Designs

- The design of a digital system happens in many different levels of abstraction
  - Physical level
  - Electronic level
  - Logic level
  - Architectural level
  - System level
Theory-Practice

- In theory, theory and practice are the same.
- In practice they are not
- Real systems tend to have many states with many transitions that depend on many inputs
- Simple state diagrams are not enough; we need a more powerful language
Modularity

- Using better notation is not enough
- We have to use another design concept called modularity
- We partition our design into many modules each one with its own specifications
Advantages

• Division of labor among members of the team
• Reuse the design of the modules in other systems
• Keep the size manageable.
They are not unrelated

- Modules and hierarchies of abstraction are not unrelated
- They both try to hide unnecessary information.
What we do here

• Design simple modules
• Implement our designs in the logic level
• Explore various possibilities
Components

- Registers (collections of F-F)
- Operations on registers (using combinational circuits)
- Control (someone to boss everything around)
Registers

- Registers are collections of F-F that can execute LOAD and other operations
- Sometimes the other operations are incrementing, decrementing, shifting, etc
- Sometimes these operations are done with the help of separate combinational circuits
For Example

• A 16-bit incrementer requires 16 half adders
  – more hardware if we seek efficiency

• If we need two counters that do not increment at the same time we might decide to let them share the incrementer.
Transfers

• This is done by transferring the data to the incremeniter

• and then transferring the output of the incremeniter back to the register
Transfer Notation

• We use statements like
  – \( R_1 \leq R_2 \)

• e.g. \( R_2 \) is copied into \( R_1 \)

• The transfer may be conditional
  – if \( (T_1=1) \) then \( (R_1 \leq R_2) \)
Anything Goes

– R1 <= R2+R3
– R3 <= R3+1
– R4 <= shl R4
– R5 <= 0

• The similarity with Verilog is obvious...
Like Verilog

• We want to describe
  – Transfer operations
  – Arithmetic operations
  – Logic operations
  – Shift operations
Clocked Transfer

• The transfer happens only at the edge of the clock
• Before the clock the combinational circuits are computing the input to the F-F
• After the clock we are computing the F-F input for the next state.
All Transfers at Once

- Since all transfers happen at once
- The natural procedural assignment is the non-blocking one:
  - R1 \leq R2
  - R2 \leq R1
Loops

• There are two uses of the loops:
  – describe test benches
  – describe repeated hardware
Example: for loop

module decoder (IN, Y);
    input [1:0] IN;    //Two binary inputs
    output [3:0] Y;   //Four binary outputs
    reg [3:0] Y;
    integer I;       //control variable for loop
    always @ (IN)
        for (I = 0; I <= 3; I = I + 1)
            if (IN == I) Y[I] = 1;
            else Y[I] = 0;
endmodule
The Equivalent

- The **for** loop can be replaced by
  - if (N=00) F[0]=1; else F[0]=0;
  - if (N=01) F[1]=1; else F[1]=0;
Synthesis

- The hardware compiler should know how to do:
  - \( B = A + C \) // addition
  - \texttt{assign} \( Y = S \ ? \ I1 : I0 \) // 2-1 MUX
  - \texttt{case} ... // larger MUX
  - \texttt{always @ (posedge ... )} // edge triggered F-F
The Design Process

- Like any design process, we have two phases
  - Compose
  - Verify
- Composition is (nowadays) done in a HDL
- Verification is done (mostly) with simulations
Various Simulations

• A simulation is just an approximation of the reality

• There are several kinds of approximations
  – RTL simulations
  – Gate level simulations
  – Electronic level simulations
Fig. 8-1 Process of HDL Simulation and Synthesis
Algorithmic State Machines

- Synchronous sequential circuits can be thought of as having two parts
  - The *data* part that is concerned with the processing of the contents of the registers
  - The *control* part that is concerned with the sequencing of states
The Datapath

• Contains all the registers
• All the arithmetic etc logic that operates on the data
• All the outputs (to the world and to the logic unit)
• Receives commands from the control unit
Control Unit

- Contains all the “state” F-F
- All the logic to decide the next state
- Receives feedback from the data path
- Generates commands for things to happen in the datapath or just outputs its state
Fig. 8-2 Control and Datapath Interaction
ASM

• Part of the work to be done is to
  – Define a set of states
  – The operations that take place in every state
  – And the transitions between the states

• All this in a way that they solve a problem

• This is called an ASM
ASM Charts

- An ASM can be described with a HDL
- Can also be described with a kind of flowchart
  - Similar to s/w flowcharts, but adapted to hardware
State Box

- Represents a state (of course!)
- Contains
  - The symbolic state name
  - The binary state name (if available)
  - Any number of unconditional operations
Fig. 8-3 State Box

(a) General description
(b) Specific example
Decision Box

- Represents a decision (of course!)
- Normally a binary decision
- Has one incoming arrow and two outgoing
- The condition is written inside
- The outcome on the outgoing arrows
Conditional Box

- Represents a conditional statement (of course!)
- Always follows a decision box
- Contains the operations that will be executed if we reach it
Fig. 8-5 Conditional Box

(a) General description

Register operation or output

From exit path of decision box

(b) Example with conditional box

START

\[ T_1 \quad 001 \]

\[ E \]

\[ 0 \quad 1 \]

\[ R \leftarrow 0 \]

\[ T_2 \quad 010 \]

\[ F \leftarrow E \]
ASM Block

- It is not what happens to an author that cannot compose an ASM chart
- Represents a complete state
- Contains one state box
- And all associated decision and condition boxes
Fig. 8-6 ASM Block
State Diagram

- A state diagram does exactly the same as an ASM chart
- The ASM chart is better suited for real problems that may have more detail
- The diagram that follows does the same as the chart before
Fig. 8-7 State Diagram Equivalent to the ASM Chart of Fig. 8-6
Timing

- All F-F, both state F-F and registers are connected to a common clock and triggered in the same fashion.
- All operations within an ASM block take place at once, but the results are stored in the F-F at the clock edge.
As a Result

- Max one assignment per F-F per ASM block
- The F-F do not change value in between clock edges
- A state is the time between the (triggering) clock edges
Fig. 8-8 Transition Between States
Design Example

• Design a sequential circuit that has
  – A counter: A[4:1]
  – Two F-F: E and F
  – An input S

• When the system is in the initial state and the input S becomes 1 the system goes in the counting state and F and A are reset
Design Example

• If the input S is 0 and the system is in the initial state, it remains in the initial state

• If in the counting state, E is set is $A[3]$ is 1, o/w reset

• If in the counting state, we go to the output state if $A[3:4] == 11$
Fig. 8-9 ASM Chart for Design Example
Design the Datapath

- We can design it with state tables etc
- We can design it in an *ad hoc* fashion
  - This often the best
Assumptions

• We have:
  – A counter A with synchronous clear
  – Two flip-flops E and F of the J-K variety
  – An input S

• And also
  – The system has three inputs T0, T1, T2 that correspond to the three states
Let’s Fry Some Bits

• The counter counts when in state T1
• The counter is reset when in state T0 and S=1
Fig. 8-10 Datapath for Design Example
Fry more Bits

- F-F E is set when in state T1 and A3=1
- F-F E is reset when in state T1 and A3=0
- F-F F is set when in state T2
- F-F F is reset when in state T0 and S=1
Control Logic

• We will see a few techniques to design the control logic

• There are many we will not see:
  – Either proprietary
  – Or historical
State Table

• We need the state table
• The state table can be big
• For this abysmally small example it has 32 entries
• We compress it
# The Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Inputs</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1 G0</td>
<td>S A3 A4</td>
<td>G1 G0</td>
<td>T1 T2 T3</td>
</tr>
<tr>
<td>0 0</td>
<td>0 x x</td>
<td>0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>1 x x</td>
<td>0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>x 0 x</td>
<td>0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>x 1 0</td>
<td>0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>x 1 1</td>
<td>1 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 1</td>
<td>x x x</td>
<td>0 0</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>
Fig. 8-11 Register Transfer Level Description of Design Example
The logic

• With a bit of symbolic manipulation
  – $D_1 = G_1' \ G_0 \ A_3 \ A_4$
  – $D_0 = G_1' \ G_0 + S \ G_1' + G_1 \ G_0' + S \ G_0'$

• And
  – $T_0 = G_1' \ G_0'$
  – $T_1 = G_1' \ G_0$
  – $T_2 = G_1 \ G_0$
With Maps

- The F-F inputs are
  - $D1 = G1' G0 A3 A4$
  - $D0 = G1'G0 + G0'S$

- And the outputs are
  - $T0 = G0'$
  - $T1 = G1'G0$
  - $T2 = G1$
With Intuition

• The F-F inputs are:
  – $D1 = T1 \ A3 \ A4$
  – $D0 = T0 \ S + T1$

• And the outputs are
  – $T0 = G0'$
  – $T1 = G1'G0$
  – $T2 = G1$
Fig. 8-12 Logic Diagram of Control
With Verilog

- Behavioral description
  - the most abstract
  - h/w compiler is the king

- Structural description
  - we have to do the work
  - we give all the detail
module Example_RTL (S,CLK,Clr,E,F,A);
//Specify inputs and outputs
//See block diagram Fig. 8-10
  input S,CLK,Clr;
  output E,F;
  output [4:1] A;
//Specify system registers
  reg [4:1] A;  //A register
  reg E, F;  //E and F flip-flops
  reg [1:0] pstate, nstate;  //control register
//Encode the states
  parameter T0 = 2'b00, T1 = 2'b01, T2 = 2'b11;
The Control Section

//State transition for control logic
//See state diagram Fig. 8-11(a)
always @(posedge CLK or negedge Clr)
    if (~Clr) pstate = T0;  //Initial state
    else pstate <= nstate;  //Clocked operations
always @ (S or A or pstate)
    case (pstate)
        T0: if(S) nstate = T1;
        T2: nstate = T0;
        default: nstate = T0;
    endcase
always @(posedge CLK)
  case (pstate)
    T0: if(S)
      begin
        A <= 4'b0000;
        F <= 1'b0;
      end
    T1:
      begin
        A <= A + 1'b1;
        if (A[3]) E <= 1'b1;
        else E <= 1'b0;
      end
    T2: F <= 1'b1;
  endcase
Multiplier

- An extremely useful thing
- An astonishingly complex thing to do if speed is important
- An outrageously tricky thing to do with floating point arithmetic
Back to Elementary School

• Let’s multiply 23 by 19

  10111
  x 10011
  --------
  10111
  10111
  10111

  110110101
Still in Elementary School

- The first number is called multiplicand
- The second is called multiplier
- The result is called product
Register Configuration

- We store the multiplicand in Reg. B
- We store the multiplier in Reg. Q
- Reg. Q will be shifted out to oblivion
- The product will be stored half in the accumulator A and half in Q
Fig. 8-13 Block Diagram of Binary Multiplier
Fig. 8-14 ASM Chart for Binary Multiplier
Control Logic

• We have three choices
  – Binary
  – Gray Code
  – One-hot (1 F-F per state)

• We have three inputs:
  – S, Q0(?), Z
We know this stuff...

• We can distill the ASM chart into a simple state diagram
• We do the tables
• Simplify
• And we are done
(a) State diagram

\[ S = 0 \]
\[ S = 1 \]
\[ Z = 1 \]
\[ Z = 0 \]

\[ T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \]

\( T_0 \): Initial state

\( T_1 \): \( A \leftarrow 0, \ C \leftarrow 0, \ P \leftarrow n \)

\( T_2 \): \( P \leftarrow P - 1 \)

\[ \text{if } (Q_0) = 1 \text{ then } (A \leftarrow A + B, \ C \leftarrow C_{\text{out}}) \]

\( T_3 \): shift right\( CAQ, \ C \leftarrow 0 \)

(b) Register transfer operations

Fig. 8-15 Control Specifications for Binary Multiplier
How it looks like

- Q0 can be dealt with separately since it does not affect the sequence of states but only the commands to the datapath (e.g. the output of the control logic)
Fig. 8-16 Control Block Diagram
## The Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1 G0</td>
<td>S Z</td>
<td>G1 G0</td>
<td>T0 T1 T2 T3</td>
</tr>
<tr>
<td>0 0</td>
<td>0 X</td>
<td>0 0</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>1 X</td>
<td>0 1</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>X X</td>
<td>1 0</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>X X</td>
<td>1 1</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>1 1</td>
<td>X 0</td>
<td>1 0</td>
<td>0 0 0 1 0</td>
</tr>
<tr>
<td>1 1</td>
<td>X 1</td>
<td>0 0</td>
<td>0 0 0 0 1</td>
</tr>
</tbody>
</table>
Simplify

• Assuming binary state assignment
  – \( D_1 = G_1' G_0 + G_1 G_0' + G_1 Z' \)
  – \( D_0 = G_1 G_0' + G_0' S \)

• Assuming Gray state assignment
  – \( D_1 = G_0 + G_1 Z' \)
  – \( D_0 = G_1 G_0 + G_1' G_0' S \)
Using the Outputs

• We need a decoder for this since there is not much minimization for the output circuits

• Plugging the outputs in
  – $D_1 = T_1 + T_2 + T_3 Z'$
  – $D_0 = T_2 + T_0 S$

• Does not matter if we use Gray or binary
Fig. 8-17 Logic Diagram of Control for Binary Multiplier Using a Sequence Register and Decoder

© 2002 Prentice Hall, Inc.
M. Morris Mano
DIGITAL DESIGN, 3e.
Some Observations

- We cannot always avoid a full decoder
- Decoders need one gate per state
- Control logic does not always offer much of a chance for simplification
One F-F per State

- It is not always as costly as it looks
- Permits direct implementation from the ASM chart
- It makes sense!
Fig. 8-18 One Flip-Flop Per State Controller
module mltp(S,CLK,Clr,Binput,Qinput,C,A,Q,P);
  input S,CLK,Clr;
  input [4:0] Binput,Qinput; //Data inputs
  output C;
  output [4:0] A,Q;
  output [2:0] P;
  //System registers
  reg C;
  reg [4:0] A,Q,B;
  reg [2:0] P;
  reg [1:0] pstate, nstate; //control register
  parameter T0=2'b00, T1=2'b01, T2=2'b10, T3=2'b11;
  //Combinational circuit
  wire Z;
  assign Z = ~P; //Check for zero
always @(negedge CLK or negedge Clr)
  if (~Clr) pstate = T0;
  else pstate <= nstate;
always @(S or Z or pstate)
  case (pstate)
    T0: if (S) nstate = T1;
    T1: nstate = T2;
    T2: nstate = T3;
    T3: if (Z) nstate = T0;
        else nstate = T2;
  endcase
always @(negedge CLK)
case (pstate)
  T0: B <= Binput; //Input multiplicand
  T1: begin
    A <= 5'b00000;
    C <= 1'b0;
    P <= 3'b101; //Initialize counter to n=5
    Q <= Qinput; //Input multiplier
  end
  T2: begin
    P <= P - 3'b001; //Decrement counter
    if (Q[0])
      {C,A} <= A + B; //Add multiplicand
  end
  T3: begin
    C <= 1'b0; //Clear C
    A <= {C,A[4:1]}; //Shift right A
    Q <= {A[0],Q[4:1]}; //Shift right Q
  end
endcase
Design with MUX

• Allow us to minimize the number of components
• Quite simple to do.
Fig. 8-19 Example of ASM Chart with Four Control Inputs
The Advantages

• MUXes allow us to break the combinational circuit in many simpler smaller ones
• Are among the few ways to design a 3 level combinational circuit
• Really useful when we have many variables and not all of them affect every state.
## The Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Input Conditions</th>
<th>MUX 1</th>
<th>MUX 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1 G0</td>
<td>G1 G0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>w'</td>
<td>0</td>
<td>w</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
<td>x</td>
<td>1</td>
<td>x'</td>
</tr>
<tr>
<td>0 1</td>
<td>1 1</td>
<td>x'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>y'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>yz'</td>
<td>yz' + yz = y</td>
<td>yz</td>
</tr>
<tr>
<td>1 0</td>
<td>1 1</td>
<td>yz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>y'z</td>
<td>y'z' + yz = y'</td>
<td>y'</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
<td>y</td>
<td>y + y'z' = y + z'</td>
<td>y'</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>y'z'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fig. 8-20 Control Implementation with Multiplexers
With One-Hot

- We do this with the tried and true method (glare at the chart til your eyes get dry)
  - $T_0 = T_0 w' + T_2 y'$
  - $T_1 = T_0 w + T_3 y'z$
  - $T_2 = T_3 y + T_1 x + T_2 yz'$
  - $T_3 = T_3 y'z' + T_1 x' + T_2 yz$
Count the Ones

- Design a circuit that counts the ones in a register R1 and stores the result in counter R2
- The circuit keeps counting and shifting the contents of R1 out till R1 is all zeros
- R2 is initialized to all 1s.
Fig. 8-21 ASM Chart for Count-of-Ones Circuit
We Also Need

- A F-F to store the bit shifted out of R1
- A combinational circuit to check if all bits of R1 are zero.
Fig. 8-22 Block Diagram for Count-of-Ones
## The Table

<table>
<thead>
<tr>
<th>Pr. St.</th>
<th>N. St.</th>
<th>Inp. Cond</th>
<th>MUX1</th>
<th>MUX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1 G0</td>
<td>G1 G0</td>
<td>Sʼ</td>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
<td>Z</td>
<td>1</td>
<td>Zʼ</td>
</tr>
<tr>
<td>0 1</td>
<td>1 1</td>
<td>Zʼ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
<td>Eʼ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fig. 8-23 Control Implementation for Count-of-Ones Circuit
Fig. P8-10  Control State Diagram for Problems 8-10 and 8-11
Fig. P8-20  ASM Chart for Problems 8-20