FPGA Development for Radar, Radio-Astronomy and Communications

THE RADAR MASTERS COURSE

Dept. Electrical Engineering, University of Cape Town
Private Bag, Rondebosch, 7701, South Africa

http://www.rrsg.uct.ac.za

Presented by John-Philip Taylor

Convened by Dr Stephen Paine

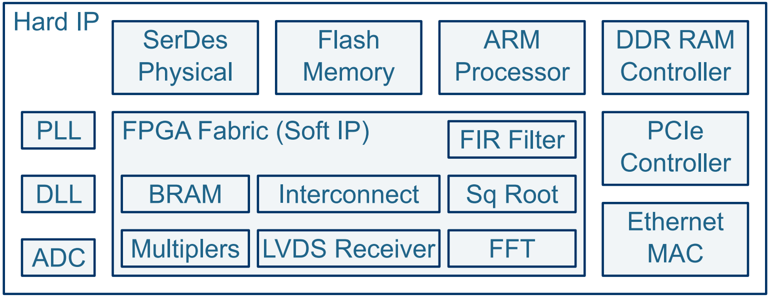
Day 2 – 28 April 2022

Outline

Outline

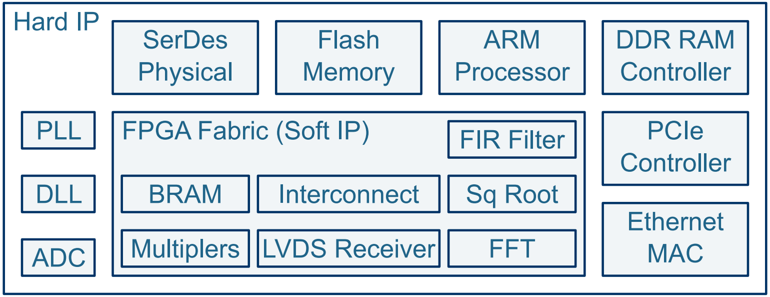
IP Library

- A combination of soft and hard IP
- Collectively known as “Megafunctions” or “IP Cores”



IP Library

- A combination of soft and hard IP
- Collectively known as “Megafunctions” or “IP Cores”



Wizard

- Generally easier to use the IPexpress wizard to generate wrapper modules
- Or one can instantiate the built-in modules directly
- Practical `06 - Data Stream` uses IPexpress to set up an EBR memory block

Wizard

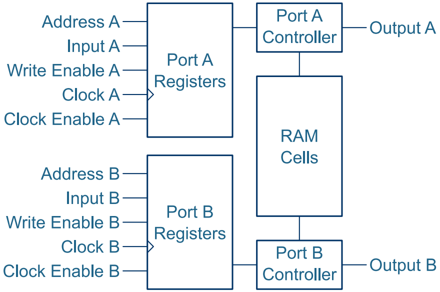
- Generally easier to use the IPexpress wizard to generate wrapper modules
- Or one can instantiate the built-in modules directly
- Practical `06 - Data Stream` uses IPexpress to set up an EBR memory block

Wizard

- Generally easier to use the IPexpress wizard to generate wrapper modules
- Or one can instantiate the built-in modules directly
- Practical `06 - Data Stream` uses IPexpress to set up an EBR memory block

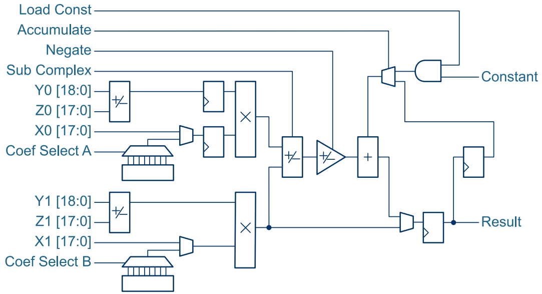
Embedded Components

- RAM / ROM
- DSP blocks
- PLL / DLL blocks
- Processors / SoC (ARM) with bus infrastructure
- Interfaces (DDR Memory / PCIe / SerDes (JESD204) / etc.)



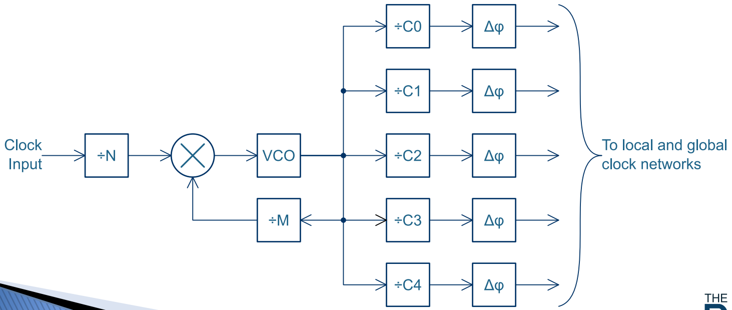
Embedded Components

- RAM / ROM
- DSP blocks
- PLL / DLL blocks
- Processors / SoC (ARM) with bus infrastructure
- Interfaces (DDR Memory / PCIe / SerDes (JESD204) / etc.)



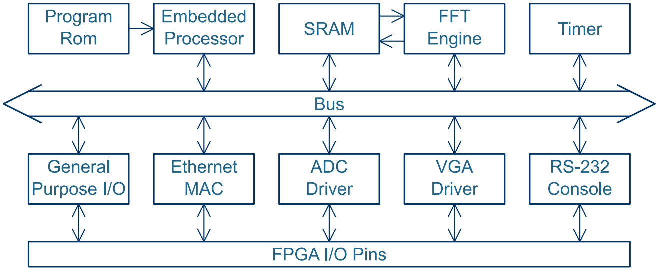
Embedded Components

- RAM / ROM
- DSP blocks
- PLL / DLL blocks
- Processors / SoC (ARM) with bus infrastructure
- Interfaces (DDR Memory / PCIe / SerDes (JESD204) / etc.)



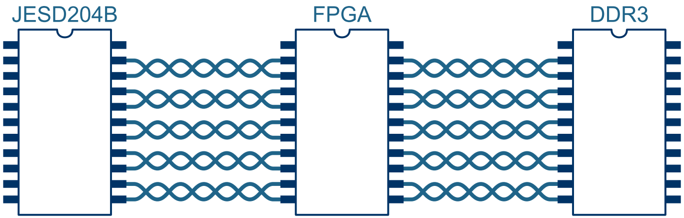
Embedded Components

- RAM / ROM
- DSP blocks
- PLL / DLL blocks
- Processors / SoC (ARM) with bus infrastructure
- Interfaces (DDR Memory / PCIe / SerDes (JESD204) / etc.)



Embedded Components

- RAM / ROM
- DSP blocks
- PLL / DLL blocks
- Processors / SoC (ARM) with bus infrastructure
- Interfaces (DDR Memory / PCIe / SerDes (JESD204) / etc.)

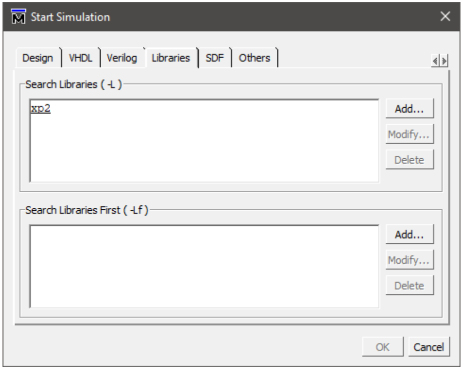


Simulation

- Modelsim can simulate IP modules
- Add the `xp2` library in the “Start Simulation” dialogue box
- Also remember to compile the IP block
- Modelsim cannot understand the `defparam` style of parameters.

Simulation

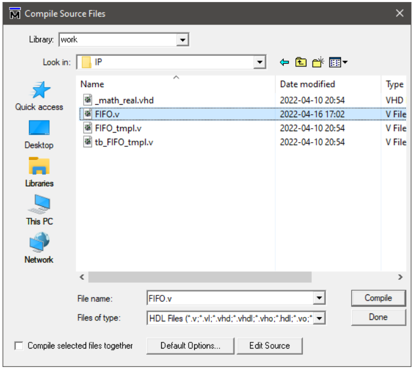
- Modelsim can simulate IP modules
- Add the `xp2` library in the "Start Simulation" dialogue box
- Also remember to compile the IP block
- Modelsim cannot understand the `defparam` style of parameters.



Search Libraries (-L): xp2

Simulation

- Modelsim can simulate IP modules
- Add the `xp2` library in the “Start Simulation” dialogue box
- Also remember to compile the IP block
- Modelsim cannot understand the `defparam` style of parameters.


Compile Source Files

Library: work

Look in: IP

Name	Date modified	Type
_math_real.vhd	2022-04-10 20:54	VHD
FIFO.v	2022-04-16 17:02	V File
FIFO_tmpl.v	2022-04-10 20:54	V File
tb_FIFO_tmpl.v	2022-04-10 20:54	V File

File name: FIFO.v

Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*

Simulation

- Modelsim can simulate IP modules
- Add the xp2 library in the “Start Simulation” dialogue box
- Also remember to compile the IP block
- Modelsim cannot understand the defparam style of parameters.

```
defparam Inst.Param1 = 8'h12;
defparam Inst.Param2 = 8'h23;
Mod Inst(
  // Port assignments
);
```

Simulation

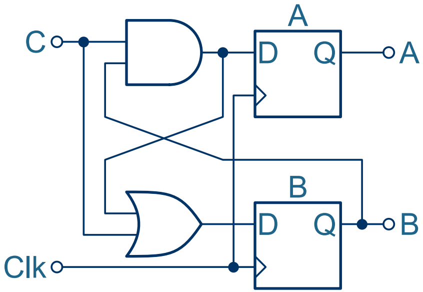
- Modelsim can simulate IP modules
- Add the `xp2` library in the “Start Simulation” dialogue box
- Also remember to compile the IP block
- Modelsim cannot understand the `defparam` style of parameters.

```
Mod #(
  .Param1(8'h12),
  .Param2(8'h23)
) Inst(
  // Port assignments
);
```

Outline

Blocking Statements

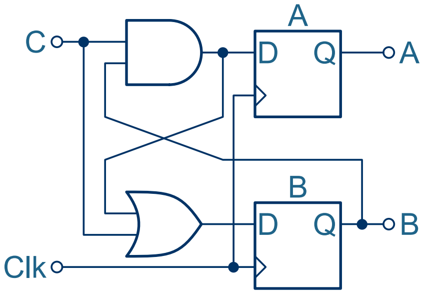
```
reg A, B;

always @(posedge ipClk) begin
  A = C & B;
  B = A | C;
end
```



- Statements are evaluated in order, like a computer program
- Often results in unintentionally long combinational chains
- Note that all registers still change state on the clock edge

Blocking Statements

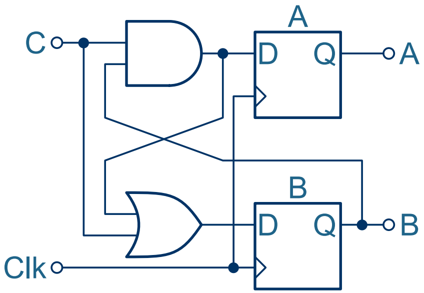
```
reg A, B;

always @(posedge ipClk) begin
  A = C & B;
  B = A | C;
end
```



- Statements are evaluated in order, like a computer program
- Often results in unintentionally long combinational chains
- Note that all registers still change state on the clock edge

Blocking Statements

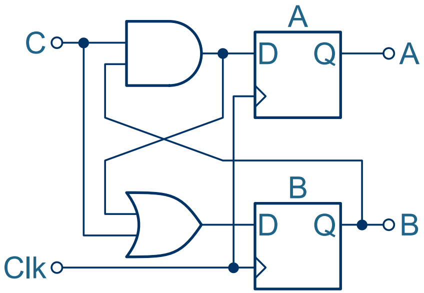
```
reg A, B;

always @(posedge ipClk) begin
  A = C & B;
  B = A | C;
end
```



- Statements are evaluated in order, like a computer program
- Often results in unintentionally long combinational chains
- Note that all registers still change state on the clock edge

Blocking Statements

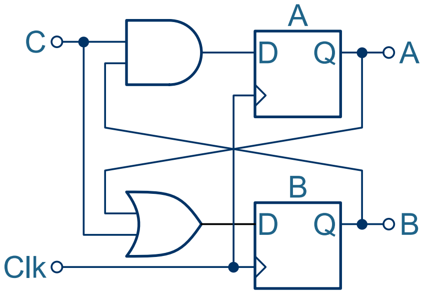
```
reg A, B;

always @(posedge ipClk) begin
  A = C & B;
  B = A | C;
end
```



- Statements are evaluated in order, like a computer program
- Often results in unintentionally long combinational chains
- Note that all registers still change state on the clock edge

Non-Blocking Statements

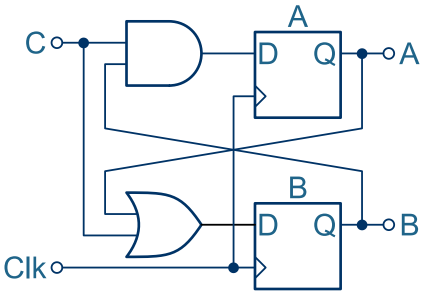
```
reg A, B;

always @(posedge ipClk) begin
  A <= C & B;
  B <= A | C;
end
```



- All right-hand-side expressions are evaluated in parallel
- and then assigned to the left-hand-side on the clock edge
- The order of statements makes no difference to the functionality

Non-Blocking Statements

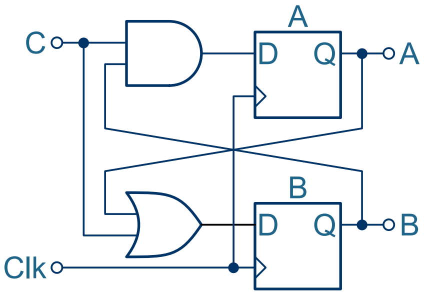
```
reg A, B;

always @(posedge ipClk) begin
  A <= C & B;
  B <= A | C;
end
```



- All right-hand-side expressions are evaluated in parallel
- and then assigned to the left-hand-side on the clock edge
- The order of statements makes no difference to the functionality

Non-Blocking Statements

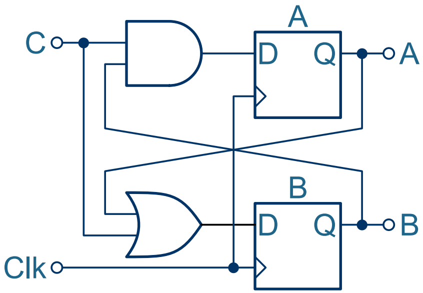
```
reg A, B;

always @(posedge ipClk) begin
  A <= C & B;
  B <= A | C;
end
```



- All right-hand-side expressions are evaluated in parallel
- and then assigned to the left-hand-side on the clock edge
- The order of statements makes no difference to the functionality

Non-Blocking Statements

```
reg A, B;

always @(posedge ipClk) begin
  A <= C & B;
  B <= A | C;
end
```



- All right-hand-side expressions are evaluated in parallel
- and then assigned to the left-hand-side on the clock edge
- The order of statements makes no difference to the functionality

Mixed Mode

Never mix blocking and non-blocking statements in the same always block

Except inside test-benches, where it is sometimes useful...

Mixed Mode

Never mix blocking and non-blocking statements in the same always block

Except inside test-benches, where it is sometimes useful...

Algorithmic Combinational Blocks

- Use blocking assignments: allows algorithmic descriptions
- If not explicitly assigned a new value, the previous value is "remembered" in an "inferred latch" – to be avoided

```
wire [7:0]Byte;  // Assigned outside always block => wire
reg  [2:0]Count; // Assigned inside  always block => reg
integer    n;    // Non-synthesisable type
                 // (compilation-time only)
always @(*) begin
  Count = 0;
  for(n = 0; n < 8; n = n+1) begin
    Count = Count + Byte[n];
  end
end
```

Algorithmic Combinational Blocks

- Use blocking assignments: allows algorithmic descriptions
- If not explicitly assigned a new value, the previous value is “remembered” in an “inferred latch” – to be avoided

```
wire    [7:0]Byte;  // Assigned outside always block => wire
reg     [2:0]Count; // Assigned inside  always block => reg
integer      n;     // Non-synthesisable type
                    // (compilation-time only)
always @(*) begin
  Count = 0;
  for(n = 0; n < 8; n = n+1) begin
    Count = Count + Byte[n];
  end
end
```

Look-up Tables

```
always @(*) begin
  case(BCD)
    4'h0:    SevenSegment = 7'b0111111;
    4'h1:    SevenSegment = 7'b0000110;
    4'h2:    SevenSegment = 7'b1011011;
    4'h3:    SevenSegment = 7'b1001111;
    4'h4:    SevenSegment = 7'b1100110;
    4'h5:    SevenSegment = 7'b1101101;
    4'h6:    SevenSegment = 7'b1111101;
    4'h7:    SevenSegment = 7'b0000111;
    4'h8:    SevenSegment = 7'b1111111;
    4'h9:    SevenSegment = 7'b1101111;
    default:; // This is bad: infers a latch
  endcase
end
```

Look-up Tables

```
always @(*) begin
  case(BCD)
    4'h0:    SevenSegment = 7'b0111111;
    4'h1:    SevenSegment = 7'b0000110;
    4'h2:    SevenSegment = 7'b1011011;
    4'h3:    SevenSegment = 7'b1001111;
    4'h4:    SevenSegment = 7'b1100110;
    4'h5:    SevenSegment = 7'b1101101;
    4'h6:    SevenSegment = 7'b1111101;
    4'h7:    SevenSegment = 7'b0000111;
    4'h8:    SevenSegment = 7'b1111111;
    4'h9:    SevenSegment = 7'b1101111;
    default: SevenSegment = 0; // This is acceptable
  endcase
end
```

Look-up Tables

```
always @(*) begin
  case(BCD)
    4'h0:    SevenSegment = 7'b0111111;
    4'h1:    SevenSegment = 7'b0000110;
    4'h2:    SevenSegment = 7'b1011011;
    4'h3:    SevenSegment = 7'b1001111;
    4'h4:    SevenSegment = 7'b1100110;
    4'h5:    SevenSegment = 7'b1101101;
    4'h6:    SevenSegment = 7'b1111101;
    4'h7:    SevenSegment = 7'b0000111;
    4'h8:    SevenSegment = 7'b1111111;
    4'h9:    SevenSegment = 7'b1101111;
    default: SevenSegment = 7'bXXXXXXX; // This is better
  endcase
end
```

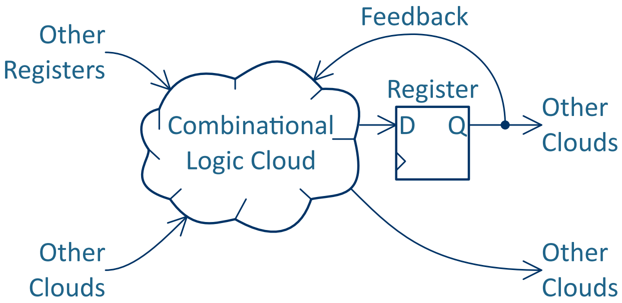
Sparse Case Statements

```
always @(*) begin
  case(Address) // 8-bit address, 16-bit data
    8'h00:   Data = FirmwareVersion;
    8'h01:   Data = BuildDate;
    8'h02:   Data = BuildTime;

    8'h10:   Data = { 6'd0, LED      };
    8'h11:   Data = { 6'd0, Switches};
    8'h12:   Data = {14'd0, Buttons };

    8'h20:   Data = Accelerometer_X;
    8'h21:   Data = Accelerometer_Y;
    8'h22:   Data = Accelerometer_Z;

    default: Data = 0; // This is OK
  endcase
end
```

Sparse Case Statements

```
always @(*) begin
  case(Address) // 8-bit address, 16-bit data
    8'h00:   Data = FirmwareVersion;
    8'h01:   Data = BuildDate;
    8'h02:   Data = BuildTime;

    8'h10:   Data = { 6'd0, LED     };
    8'h11:   Data = { 6'd0, Switches};
    8'h12:   Data = {14'd0, Buttons };

    8'h20:   Data = Accelerometer_X;
    8'h21:   Data = Accelerometer_Y;
    8'h22:   Data = Accelerometer_Z;

    default: Data = 16'hXXXX; // This is better
  endcase
end
```

Register Transfer Logic



Register Transfer Logic

- Use non-blocking assignments:
  easier to relate all calculations to the clock edge

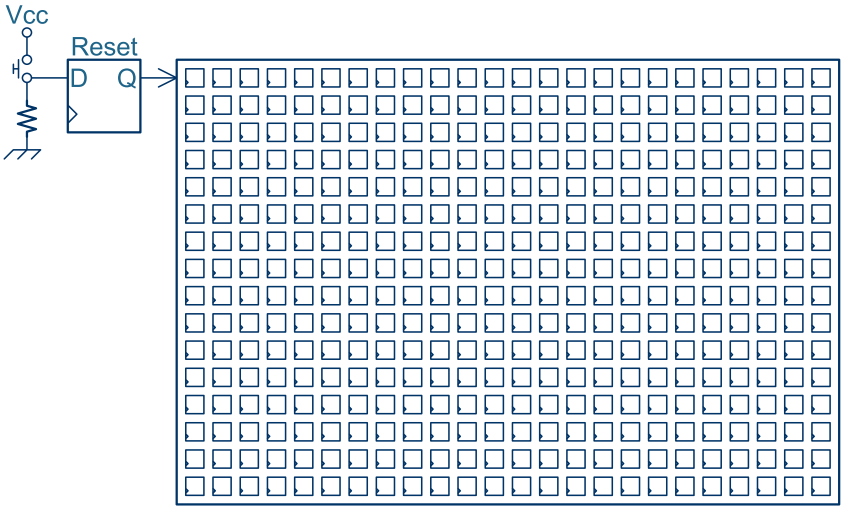
```
reg Reset;

always @(posedge ipClk) begin
  Reset <= ipReset; // Localise the reset

  if(Reset) begin
    // Reset stuff here

  end else if(ipEnabled) begin
    // RTL code goes here
  end
end
```

Register Transfer Logic

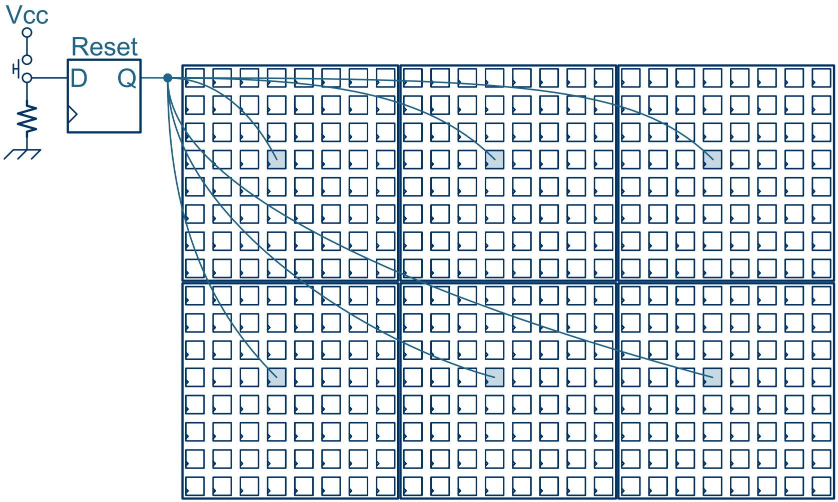
- If not explicitly assigned a new value, the previous value is "remembered" in the register – very useful

```
reg       Reset;
reg [11:0]Count;

always @(posedge ipClk) begin
  Reset <= ipReset; // Localise the reset

  if(Reset) begin
    Count <= 0;

  end else if(ipEnabled) begin
    Count <= Count + 1'b1;
  end
end
```

Synchronous and Local Resets



Synchronous and Local Resets



Outline

Finite State Machines

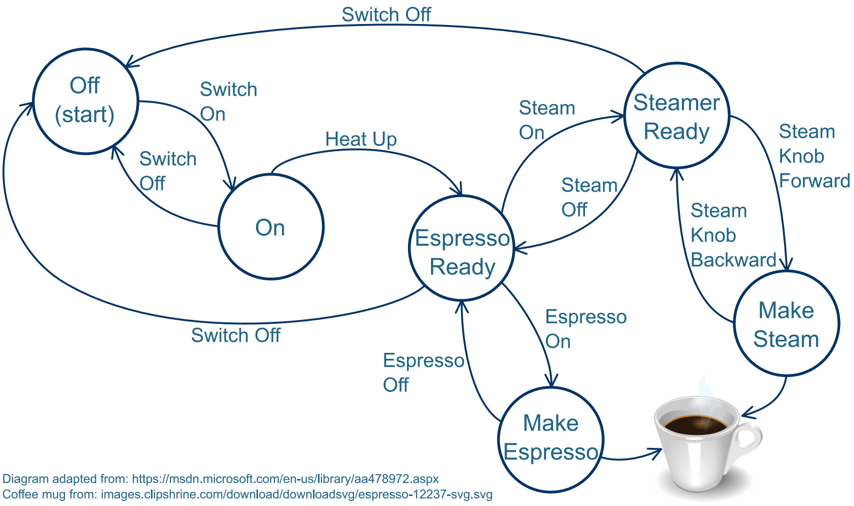


Diagram adapted from: https://msdn.microsoft.com/en-us/library/aa478972.aspx
Coffee mug from: images.clipshrine.com/download/downloadsvg/espresso-12237-svg.svg

Named States

- Example using Gray code:

```
reg [2:0]State;

localparam Off           = 3'b000;
localparam On            = 3'b001;
localparam EspressoReady = 3'b011;
localparam SteamerReady  = 3'b010;
localparam MakeEspresso  = 3'b110;
localparam MakeSteam     = 3'b111;
```

Named States

- Example using one-hot encoding:

```
reg [5:0]State;

localparam Off           = 6'b000001;
localparam On            = 6'b000010;
localparam EspressoReady = 6'b000100;
localparam SteamerReady  = 6'b001000;
localparam MakeEspresso  = 6'b010000;
localparam MakeSteam     = 6'b100000;
```

Named States

- Or let the compiler select the encoding:

```
typedef enum{ // SystemVerilog only
  Off,
  On,
  EspressoReady,
  SteamerReady,
  MakeEspresso,
  MakeSteam
} tState;
tState State;
```

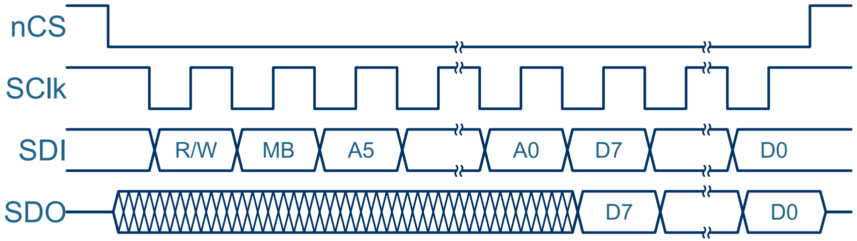
FSM Template

```
always @(posedge ipClk) begin
  if(Reset) begin
    State <= Off;

  end else begin
    case(State)
      Off:           begin ... end
      On:            begin ... end
      EspressoReady: begin ... end
      SteamerReady:  begin ... end
      MakeEspresso:  begin ... end
      MakeSteam:     begin ... end
      default:;
    endcase
  end
end
```

FSM Example

```
                ...
case(State)
  Off: begin
    if(Switch_On) State <= On;
  end

  On: begin
         if(Switch_Off) State <= Off;
    else if(Heat_Up   ) State <= EspressoReady;
  end

  EspressoReady: begin
         if(Switch_Off ) State <= Off;
    else if(Steam_On   ) State <= SteamerReady;
    else if(Espresso_On) State <= MakeEspresso;
  end
                ...
```

SPI Interface



- For the ADXL345 Digital Accelerometer:
  - Maximum SClk frequency is 5 MHz (200 ns period)
  - SDI setup and hold is 5 ns (sampled on rising edge)
  - SClk falling edge to SDO delay is 40 ns

The Abstraction

```
module ADXL345 #(
  parameter Clock_Div = 5 // 5 MHz SClk on a 50 MHz ipClk
)(
  input ipClk, ipReset,

  // 2's Compliment Output
  output reg [15:0]X,
  output reg [15:0]Y,
  output reg [15:0]Z,

  // Physical device interface
  output reg nCS, SClk, SDI,
  input      SDO
);
```

General Structure

```
reg      Reset;
reg [3:0]Clock_Count  = 0;
wire     Clock_Enable = (Clock_Count == Clock_Div);

always @(posedge ipClk) begin
  Reset <= ipReset;

  if(Clock_Enable) Clock_Count <= 4'd1;
  else             Clock_Count <= Clock_Count + 1'b1;

  if(Reset) begin
    // Reset the machine here

  end else if(Clock_Enable) begin
    // State machine goes here
  end
end
```

State Machine

```
reg [ 4:0]Count;
reg [15:0]Data; // (R/W, MB, Address, Byte) or (2 Bytes)

typedef enum {
  Setup,
  ReadX, ReadY, ReadZ,
  Transaction
} STATE;

STATE State;
STATE RetState; // Used for function calls
```

State Machine

```
if(Reset) begin
  nCS   <= 1'b1;
  SClk  <= 1'b1;
  SDI   <= 1'b1;
  State <= Setup;

end else if(Clock_Enable) begin
  case(State)
    Setup: begin
      // SPI 4-wire; Full-res; Right-justify; 4g Range
      Data     <= {2'b00, 6'h31, 8'b0000_1001};
      Count    <= 5'd16;
      State    <= Transaction;
      RetState <= ReadX;
    end
```

State Machine

```
ReadX: begin
  Z        <= {Data[7:0], Data[15:8]};
  Data     <= {2'b11, 6'h32, 8'd0};
  Count    <= 5'd24;
  State    <= Transaction;
  RetState <= ReadY;
end

ReadY: begin
  X        <= {Data[7:0], Data[15:8]};
  Data     <= {2'b11, 6'h34, 8'd0};
  Count    <= 5'd24;
  State    <= Transaction;
  RetState <= ReadZ;
end
```

State Machine

```
ReadZ: begin
  Y        <= {Data[7:0], Data[15:8]};
  Data     <= {2'b11, 6'h36, 8'd0};
  Count    <= 5'd24;
  State    <= Transaction;
  RetState <= ReadX;
end
```

Reading Data

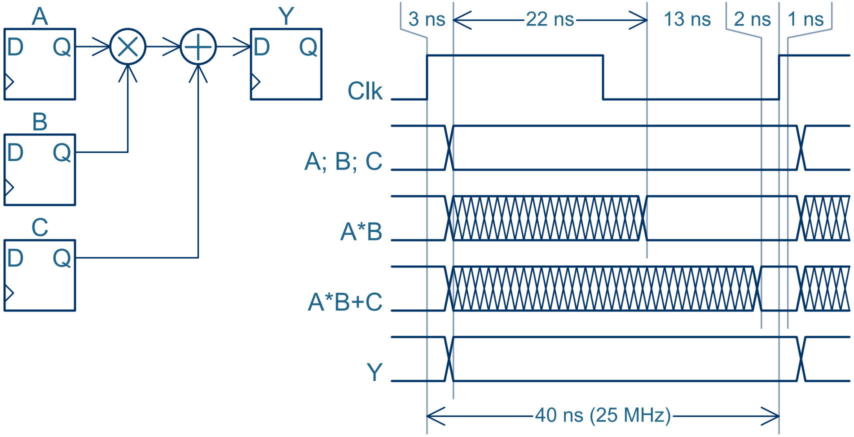
```
Transaction: begin
  if(nCS) begin
    nCS <= 1'b0;

  end else begin
    if(SClk) begin
      if(Count == 0) begin
        nCS   <= 1'b1;
        State <= RetState;
      end else begin
        SClk  <= 1'b0;
      end
      Count <= Count - 1'b1;
      {SDI, Data} <= {Data, SDO};

    end else begin
      SClk <= 1'b1;
end end end
```

THE RADAR MASTERS COURSE

Coffee Break...

Outline

Internal Timing



Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
    - Clock frequencies
    - Asynchronous paths
    - PCB trace delays
    - External device parameters
    - Multi-cycle paths
    - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Synopsis Design Constraints

- De facto industry standard
- TCL based, so one can use TCL scripting within the SDC file
- Only specify what the compiler does not already know:
  - Clock frequencies
  - Asynchronous paths
  - PCB trace delays
  - External device parameters
  - Multi-cycle paths
  - etc.

Asynchronous Pins

- Pins such as LEDs, buttons, RS-232 signals, etc. do not belong to a clock domain
- The compiler must not try to meet timing on these:

```
set_false_path -from [get_registers *} \
               -to   [get_ports opLED*]

set_false_path -to   [get_registers *} \
               -from [get_ports ipSwitch*]

set_false_path -to   [get_registers *} \
               -from [get_ports ipButton*]
```

Asynchronous Pins

- Pins such as LEDs, buttons, RS-232 signals, etc. do not belong to a clock domain
- The compiler must not try to meet timing on these:

```
set_false_path -from [get_registers *} \
               -to   [get_ports opLED*]

set_false_path -to   [get_registers *} \
               -from [get_ports ipSwitch*]

set_false_path -to   [get_registers *} \
               -from [get_ports ipButton*]
```

Clock Specification

```
create_clock -period 100 [get_ports ADC_Clk]
create_clock -period  20 [get_ports Clk1]
create_clock -period  20 [get_ports Clk2]

derive_pll_clocks
derive_clock_uncertainty
```

Clock Groups

- Unless specified otherwise, the compiler assumes that all clocks are related and in the same clock domain
- When clocks are unrelated, all paths between them must be marked as “false paths”
- Do this with clock groups:

```
set_clock_groups -asynchronous \
  -group [get_clocks ADC_Clk]  \
  -group [get_clocks Clk1]     \
  -group [get_clocks Clk2]     \
  -group [get_clocks {SRAM_CLK *altpll_0*}]
```

Clock Groups

- Unless specified otherwise, the compiler assumes that all clocks are related and in the same clock domain
- When clocks are unrelated, all paths between them must be marked as “false paths”
- Do this with clock groups:

```
set_clock_groups -asynchronous \
  -group [get_clocks ADC_Clk]  \
  -group [get_clocks Clk1]     \
  -group [get_clocks Clk2]     \
  -group [get_clocks {SRAM_CLK *altpll_0*}]
```

Clock Groups

- Unless specified otherwise, the compiler assumes that all clocks are related and in the same clock domain
- When clocks are unrelated, all paths between them must be marked as “false paths”
- Do this with clock groups:

```
set_clock_groups -asynchronous \
  -group [get_clocks ADC_Clk]  \
  -group [get_clocks Clk1]     \
  -group [get_clocks Clk2]     \
  -group [get_clocks {SRAM_CLK *altpll_0*}]
```

Outline

IP Library

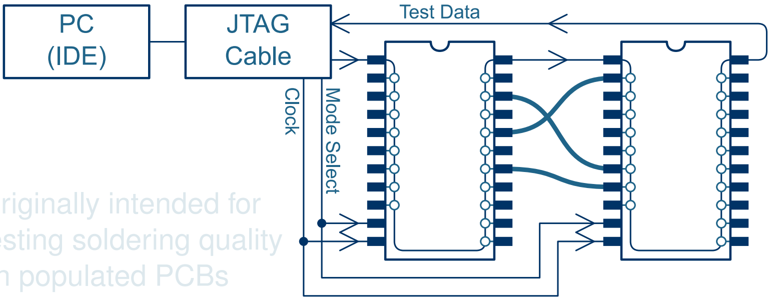
Verilog Processes

Finite State Machines

Timing Constraints

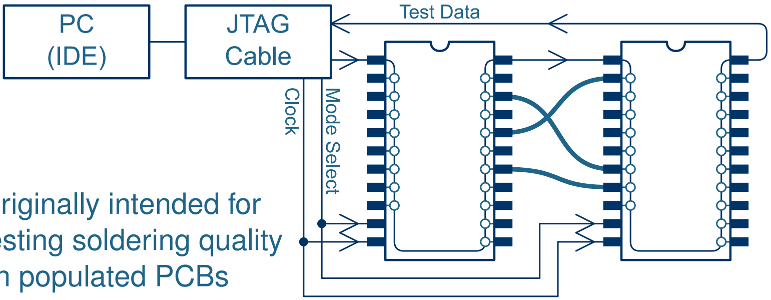
JTAG

JTAG – Joint Test Action Group



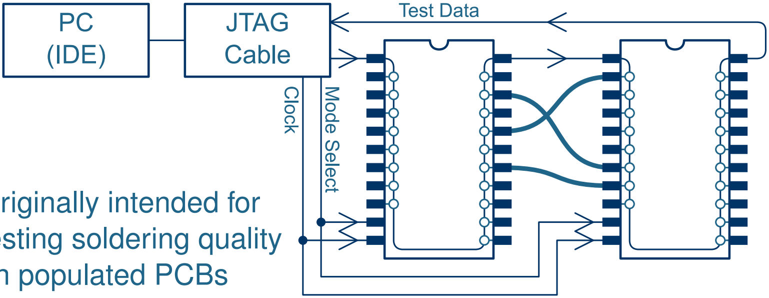
- Originally intended for testing soldering quality on populated PCBs
- Connects to the PC over USB / Ethernet / etc.
- Connected devices form a long chain of shift-registers

JTAG – Joint Test Action Group



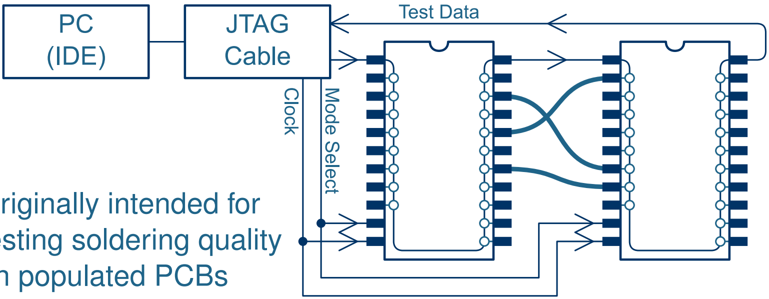
- Originally intended for testing soldering quality on populated PCBs
- Connects to the PC over USB / Ethernet / etc.
- Connected devices form a long chain of shift-registers

JTAG – Joint Test Action Group



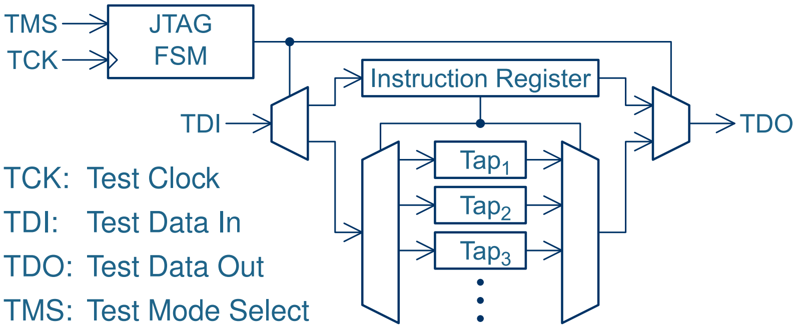
- Originally intended for testing soldering quality on populated PCBs
- Connects to the PC over USB / Ethernet / etc.
- Connected devices form a long chain of shift-registers

JTAG – Joint Test Action Group



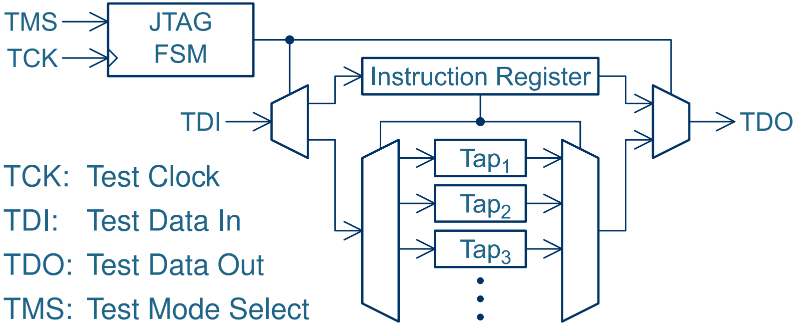
- Originally intended for testing soldering quality on populated PCBs
- Connects to the PC over USB / Ethernet / etc.
- Connected devices form a long chain of shift-registers

JTAG – Joint Test Action Group



- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

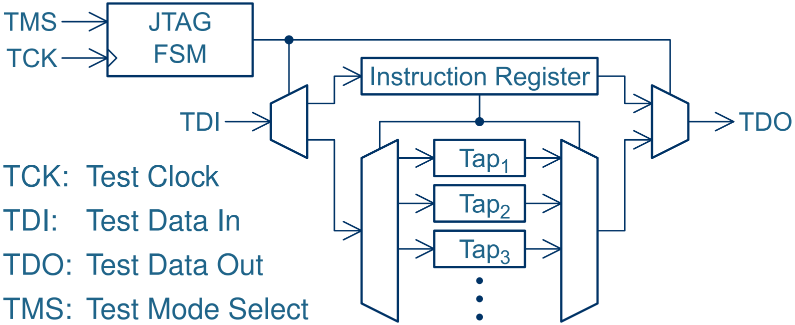
JTAG – Joint Test Action Group



- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

THE RADAR MASTERS COURSE

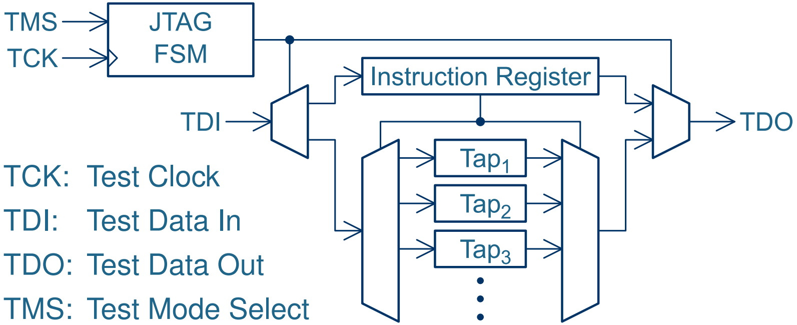
JTAG – Joint Test Action Group



- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

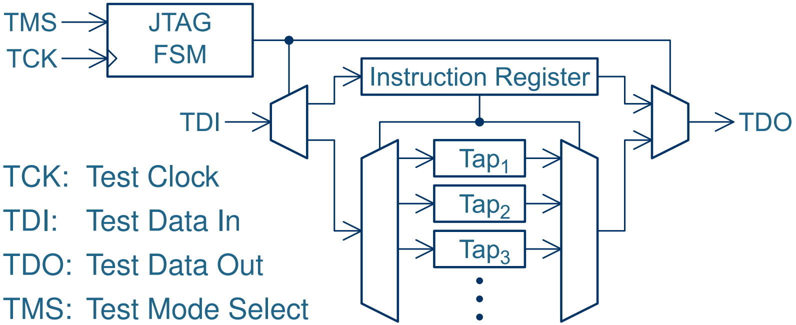
THE RADAR MASTERS COURSE

JTAG – Joint Test Action Group



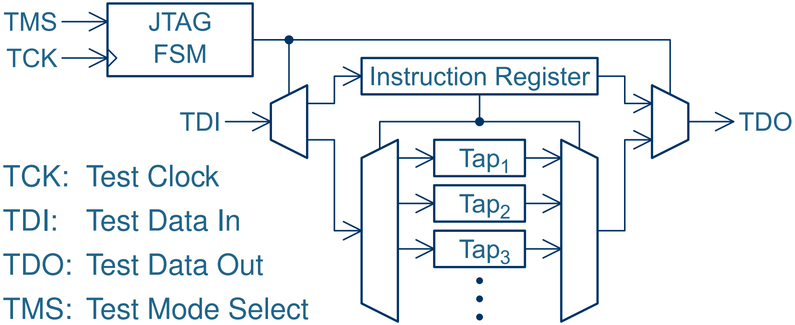
- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

JTAG – Joint Test Action Group



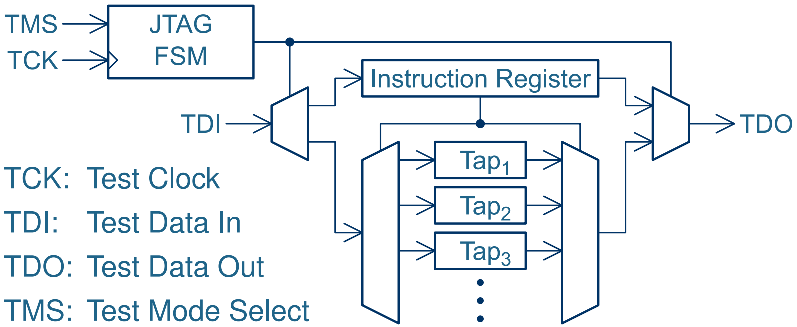
- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

JTAG – Joint Test Action Group



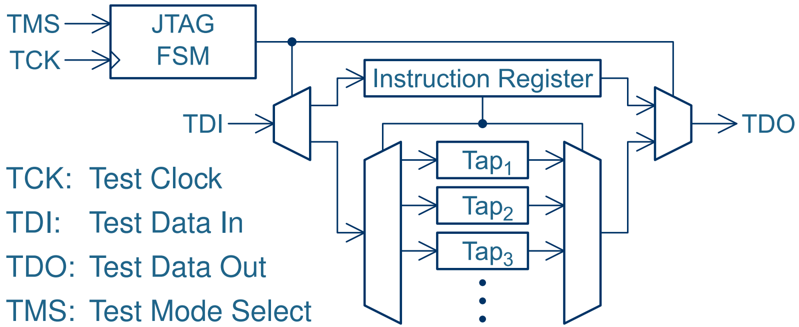
- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

JTAG – Joint Test Action Group



- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

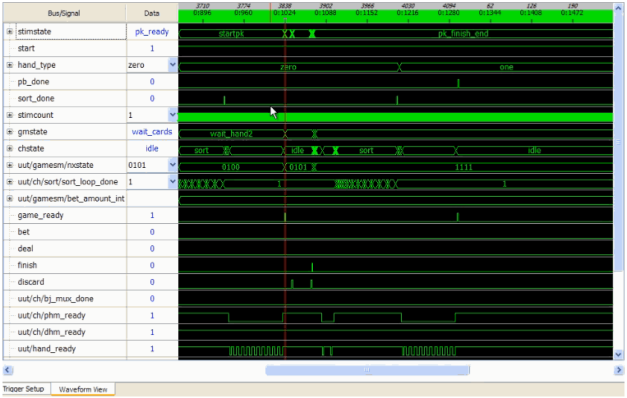
JTAG – Joint Test Action Group



- TCK: Test Clock
- TDI: Test Data In
- TDO: Test Data Out
- TMS: Test Mode Select
- Taps could include: Device pins; Internal flash; Status registers; Debug registers; Virtual JTAG interface; etc.

JTAG Debugging

FPGA IDEs includes powerful JTAG-based debugging tools. Practical `04 – JTAG Debugging` introduces the Lattice Diamond Reveal Analyzer.



Bus/Signal	Data
stimstate	pk_ready
start	1
hand_type	zero
pb_done	0
sort_done	0
stimcount	1
gmstate	wait_cards
chstate	idle
uut/gamesm/nxstate	0101
uut/ch/sort/sort_loop_done	1
uut/gamesm/bet_amount_int	
game_ready	1
bet	0
deal	0
finish	0
discard	0
uut/ch/bj_mux_done	0
uut/ch/phm_ready	1
uut/ch/dhm_ready	1
uut/hand_ready	1

Select References

- Stephen Brown and Zvonko Vranesic
  Fundamentals of Digital Logic with Verilog Design, 2nd Edition
  ISBN 978-0-07-721164-6
- Merrill L Skolnik
  Introduction to RADAR Systems
  ISBN 978-0-07-288138-7
- Mark A. Richards and James A. Scheer
  Principles of Modern Radar: Basic Principles
  ISBN 978-1-89-112152-4
- Deepak Kumar Tala
  World of ASIC
  `http://www.asic-world.com/`
- Jean P. Nicolle
  FPGA 4 Fun
  `http://www.fpga4fun.com/`

FPGA Development for Radar, Radio-Astronomy and Communications

THE RADAR MASTERS COURSE

Dept. Electrical Engineering, University of Cape Town
Private Bag, Rondebosch, 7701, South Africa

http://www.rrsg.uct.ac.za

Presented by John-Philip Taylor

Convened by Dr Stephen Paine

Day 2 – 28 April 2022