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 Prof Daniel O'Hagan

Tutored by Stephen Paine and Randy Cheng

Day 3 – 19 July 2017

Outline

Outline

Pipelines

Streaming Processors

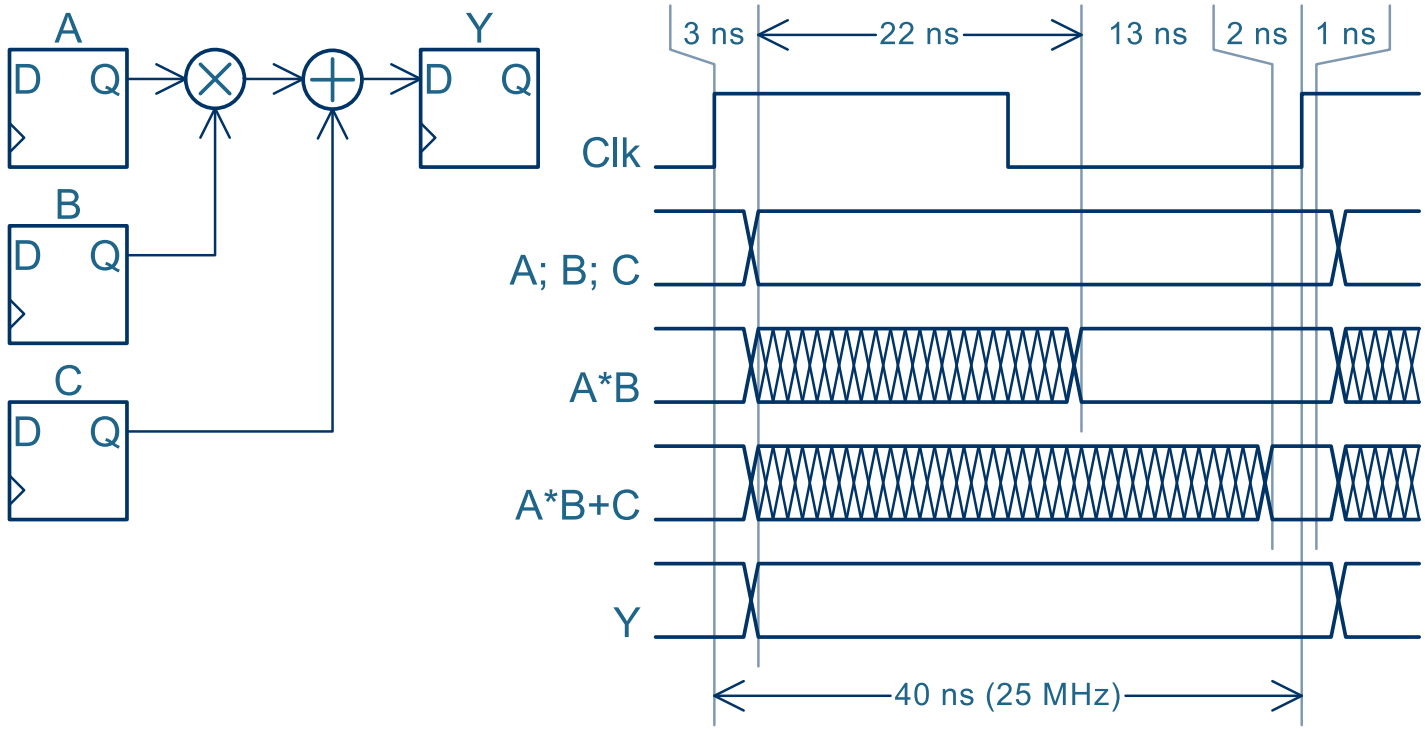
Memory-Mapped Bus

Example: Virtual JTAG MM-Bus Control Interface

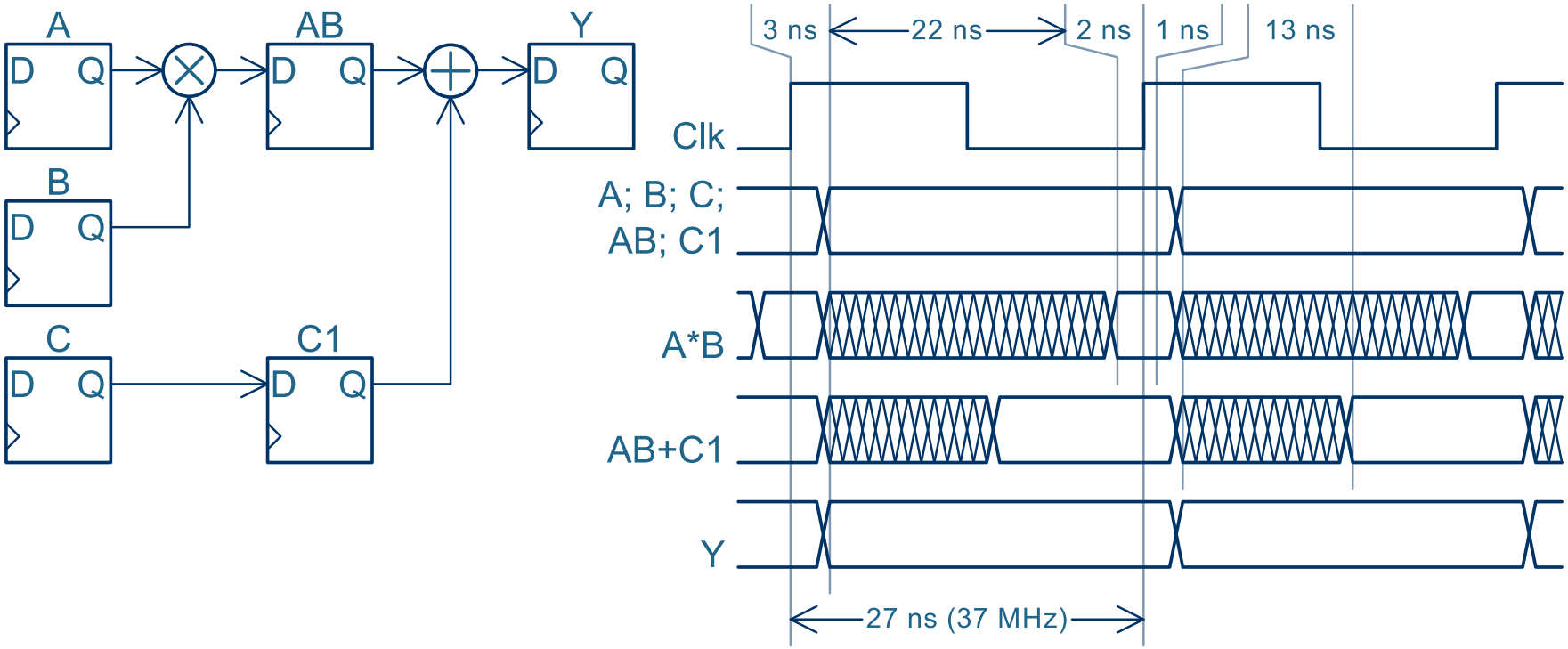
Advanced Simulation

Practical

Simple Pipelines



Simple Pipelines



Simple Pipelines

- Gain throughput at the cost of latency and resources
- Often easier to use arrays, especially with long chains
- Keep the index equal to the stage which assigns the value

```
always @(posedge Clk) begin
 // Stage 1
 AB <= A*B;
 C1 <= C;

 // Stage 2
 Y <= AB + C1;
end
```

Simple Pipelines

- Gain throughput at the cost of latency and resources
- Often easier to use arrays, especially with long chains
- Keep the index equal to the stage which assigns the value

```
reg [7:0]C[1:0];

always @(posedge Clk) begin
 // Stage 0
 C[0] <= C_Input;

 // Stage 1
 AB   <= A*B;
 C[1] <= C[0];

 // Stage 2
 Y <= AB + C[1];
end
```

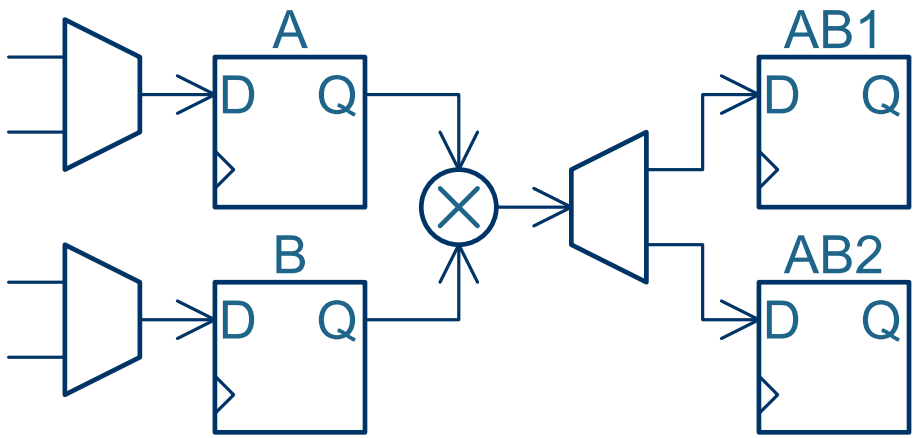
Simple Pipelines

- Gain throughput at the cost of latency and resources
- Often easier to use arrays, especially with long chains
- Keep the index equal to the stage which assigns the value

```
reg [7:0]C[1:0];

always @(posedge Clk) begin
 // Stage 0
 C[0] <= C_Input;

 // Stage 1
 AB   <= A*B;
 C[1] <= C[0];

 // Stage 2
 Y <= AB + C[1];
end
```

Sharing Resources

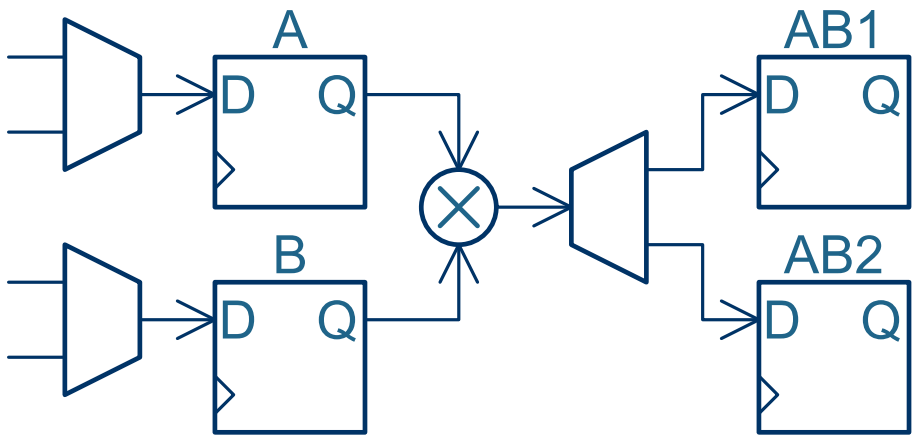

```
reg  [ 7:0]A, B;
wire [15:0]AB = A * B;

State1: begin
 AB2   <= AB;
 A     <= A1;
 B     <= B1;
 State <= State2;
end

State2: begin
 AB1   <= AB;
 A     <= A2;
 B     <= B2;
 State <= State1;
end
```



- Gain resource efficiency at the cost of throughput

Sharing Resources

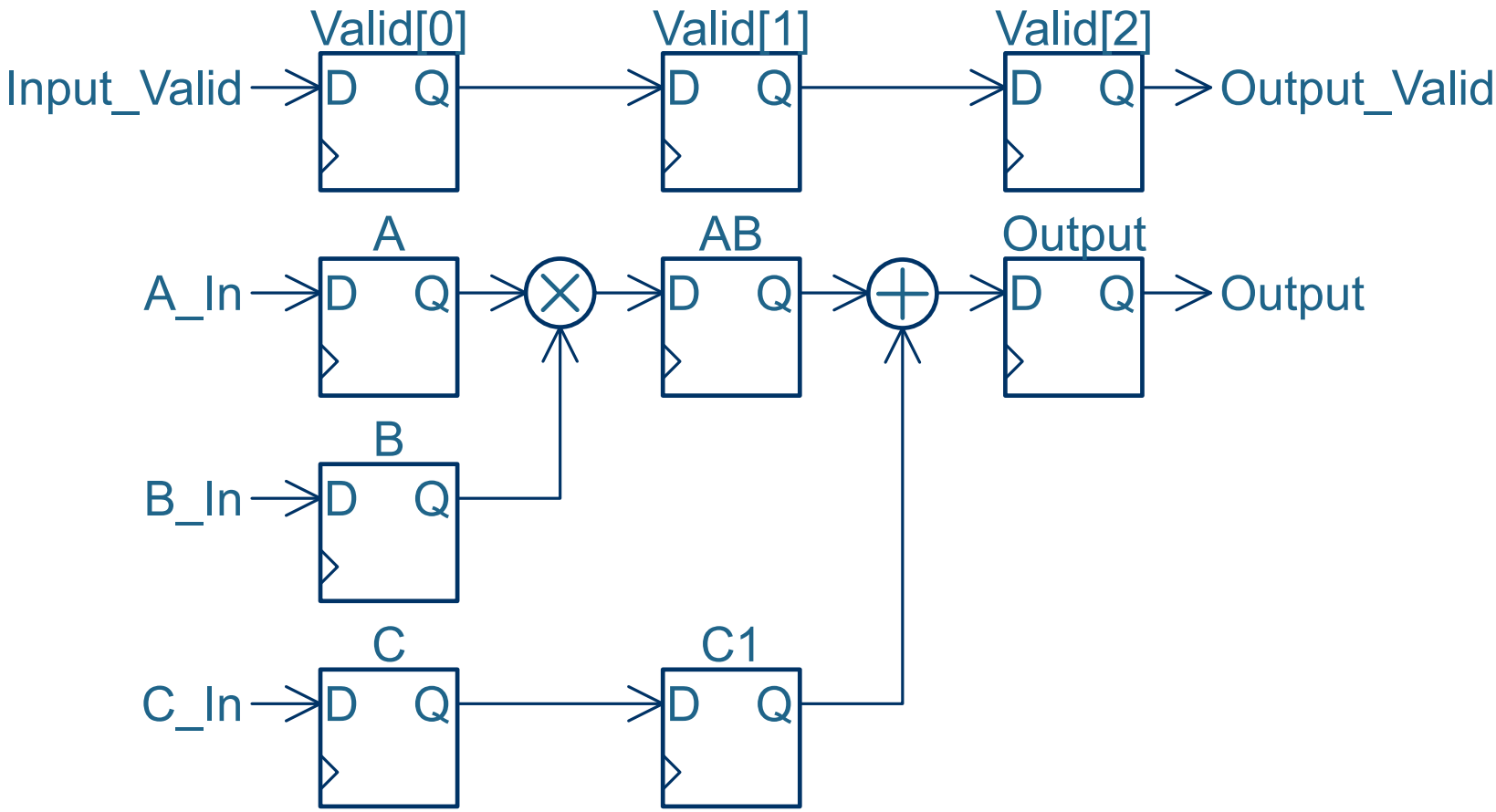

```
reg  [ 7:0]A, B;
wire [15:0]AB = A * B;

State1: begin
 AB2   <= AB;
 A     <= A1;
 B     <= B1;
 State <= State2;
end

State2: begin
 AB1   <= AB;
 A     <= A2;
 B     <= B2;
 State <= State1;
end
```



- Gain resource efficiency at the cost of throughput

Outline

Stream Pipeline



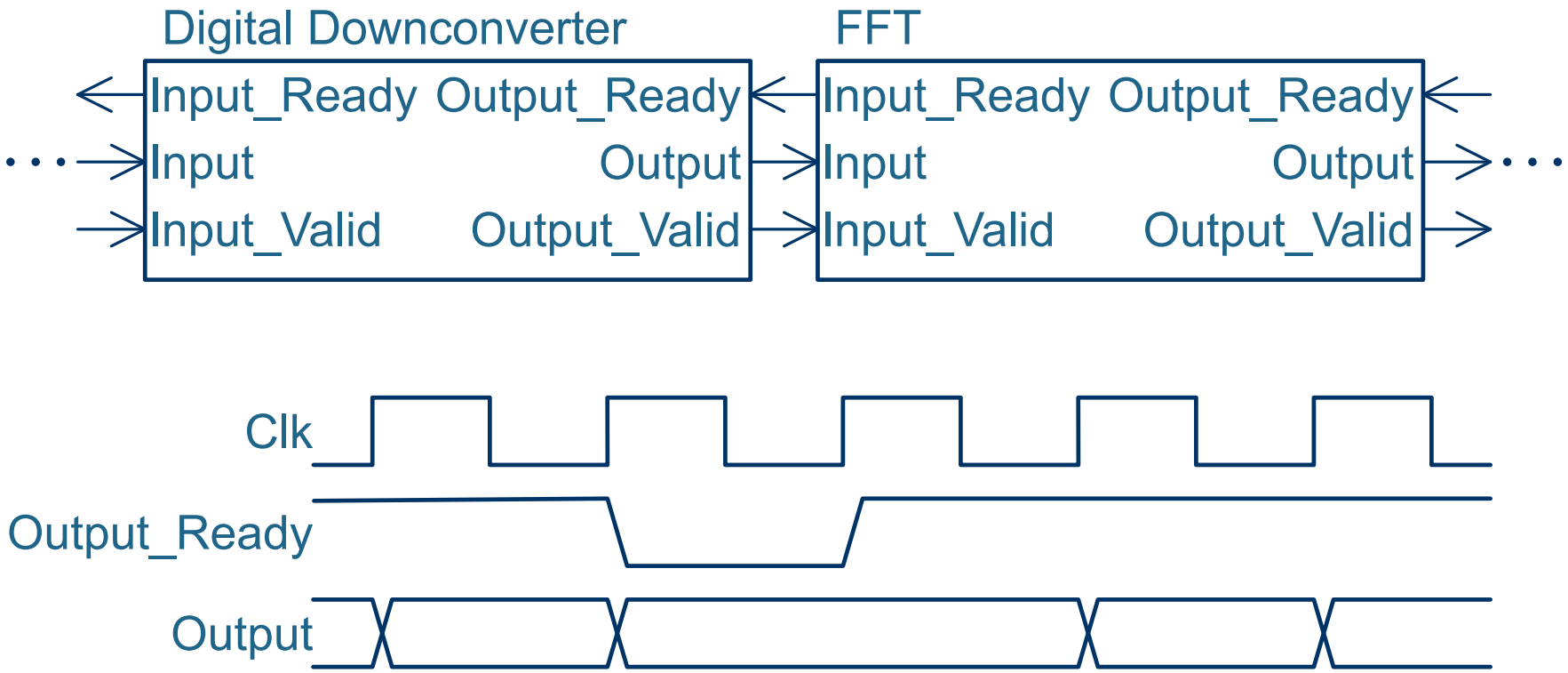
Stream Pipeline


```
reg [2:0]Valid;

always @(posedge Clk) begin
 // Input
 A     <= A_In; B <= B_In; C <= C_In;
 Valid <= {Valid[1:0], Input_Valid};

 // Stage 1
 AB <= A*B;
 C1 <= C;

 // Stage 2
 Output <= AB + C1;
end

assign Output_Valid = Valid[2];
```

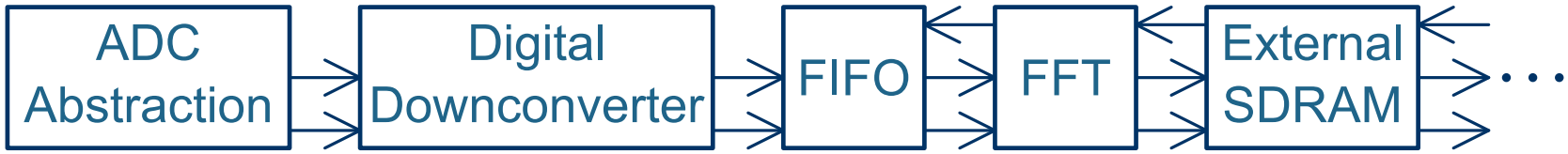
Back-pressure


```
reg [2:0]Valid;

always @(posedge Clk) begin
 if(Output_Ready) begin
  // Input
  ...

  // Stage 1
  ...

  // Stage 2
  Output <= AB + C1;
 end
end

assign Input_Ready  = Output_Ready;
assign Output_Valid = Valid[2];
```

Streaming Processor

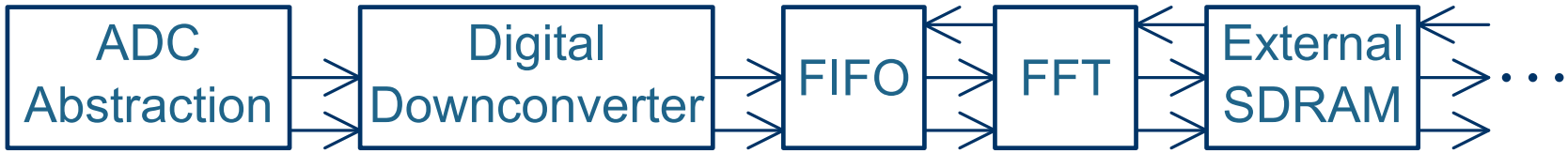


FIFO Queues



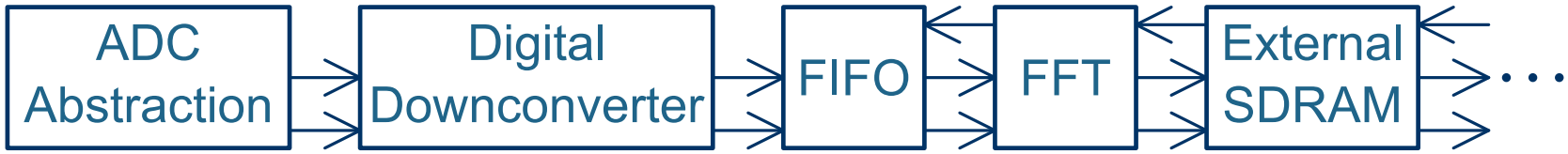
- Back-pressure makes it possible to move all the FIFO queues into a single queue
- Generally put the queue where the least amount of data is (saves resources)

FIFO Queues



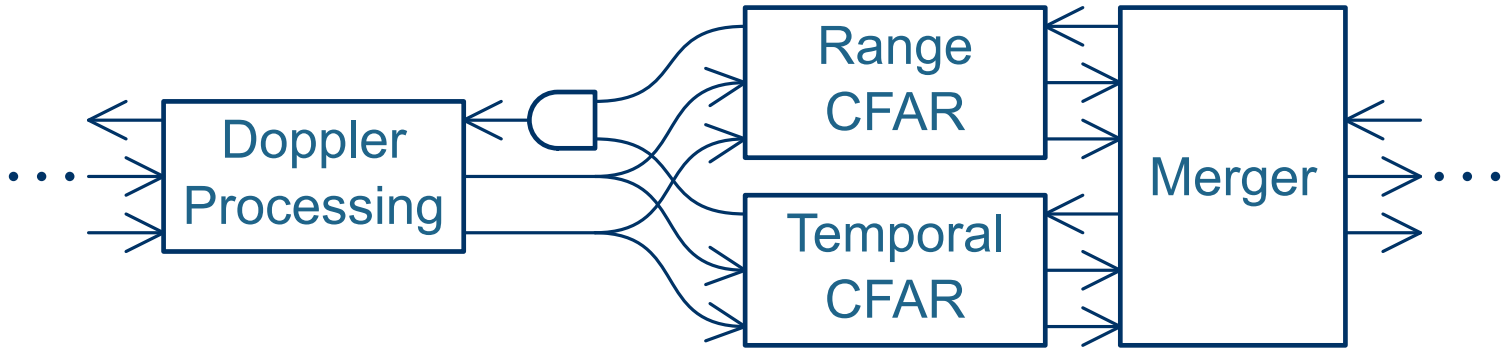
- Back-pressure makes it possible to move all the FIFO queues into a single queue
- Generally put the queue where the least amount of data is (saves resources)

FIFO Queues



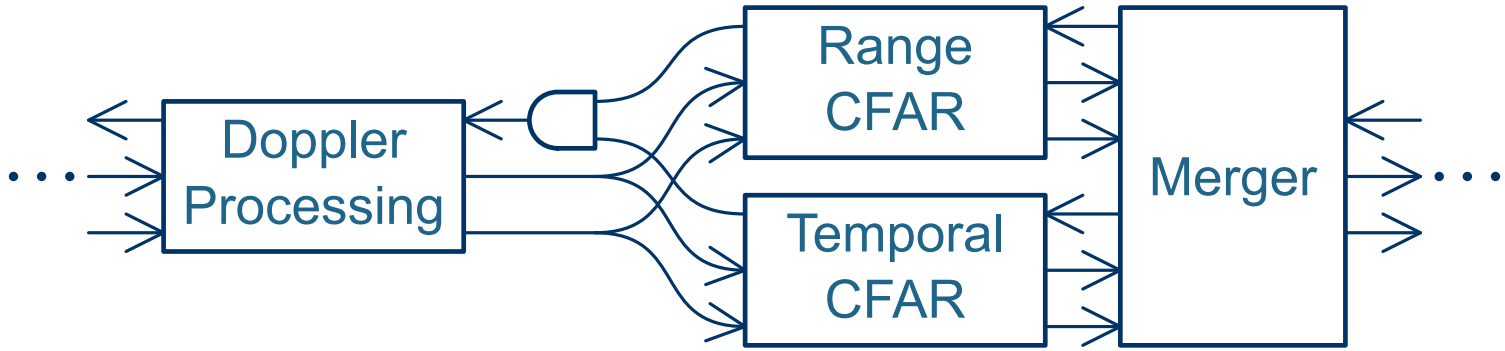
- Back-pressure makes it possible to move all the FIFO queues into a single queue
- Generally put the queue where the least amount of data is (saves resources)

Complex Structures



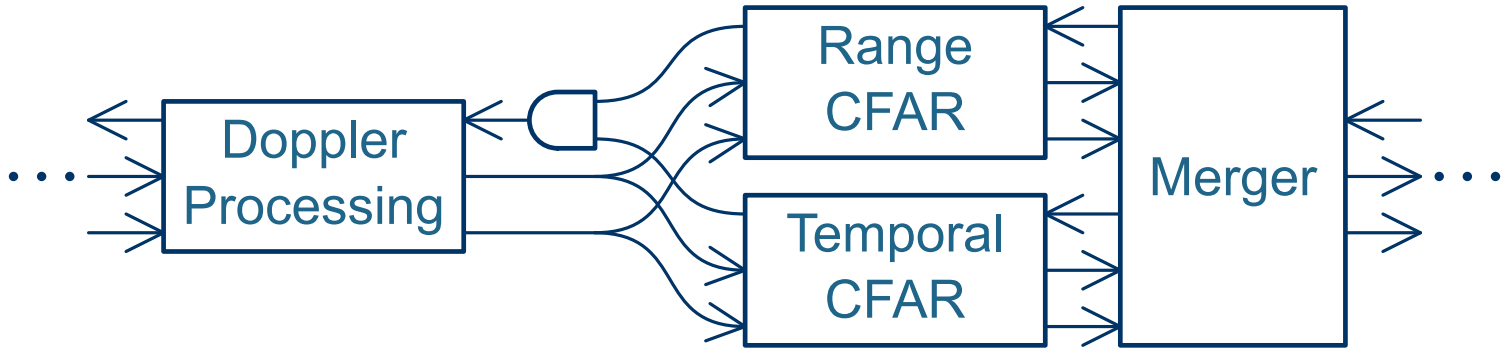
- Gate the “Valid” with the “Ready”
- The “Merger” can take various forms:

Complex Structures



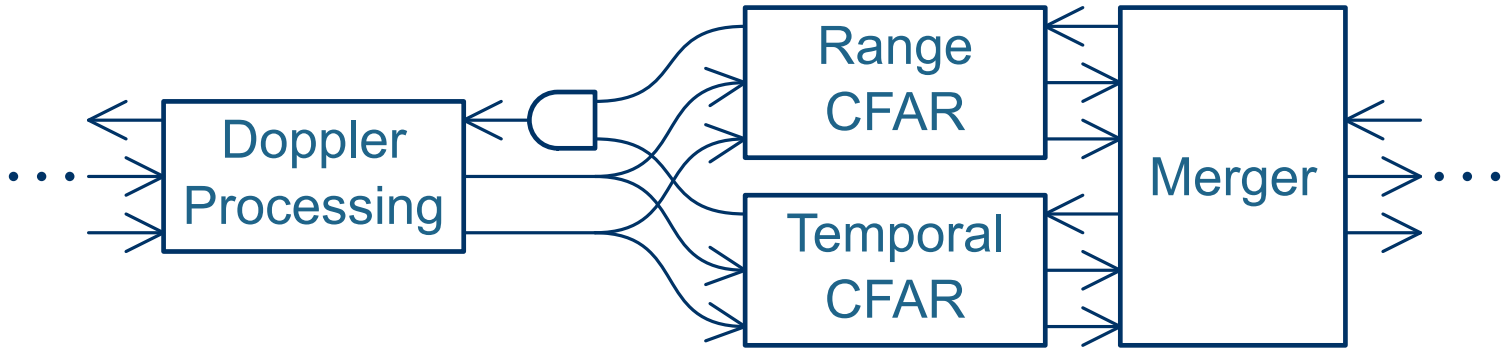
- Gate the “Valid” with the “Ready” (if either sink is not ready, both sinks must see a “not valid” input)
- The “Merger” can take various forms:

Complex Structures



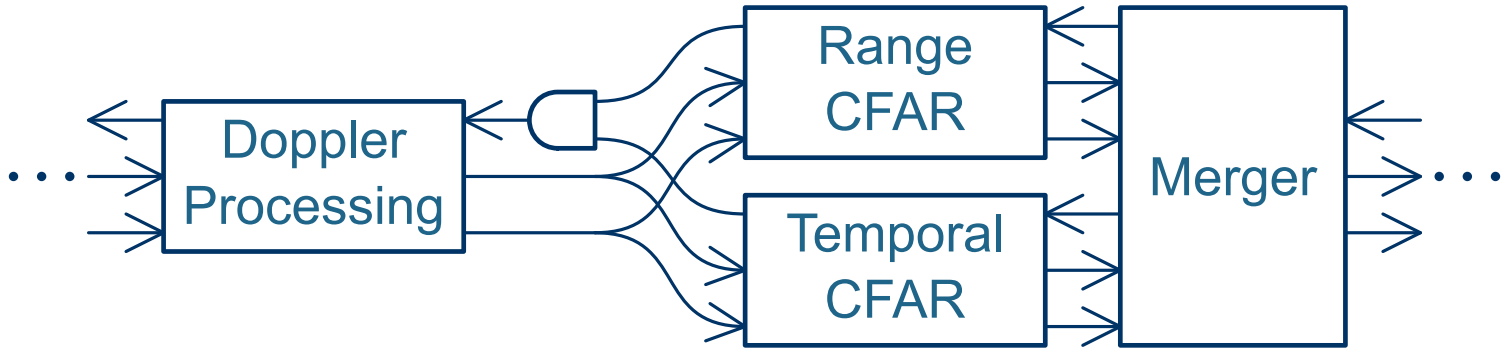
- Gate the “Valid” with the “Ready”
- The “Merger” can take various forms:
  - Interleave
  - Alternate (sent one logical unit from the one, then the other, then repeat)
  - Combine through calculation
  - etc.

Complex Structures



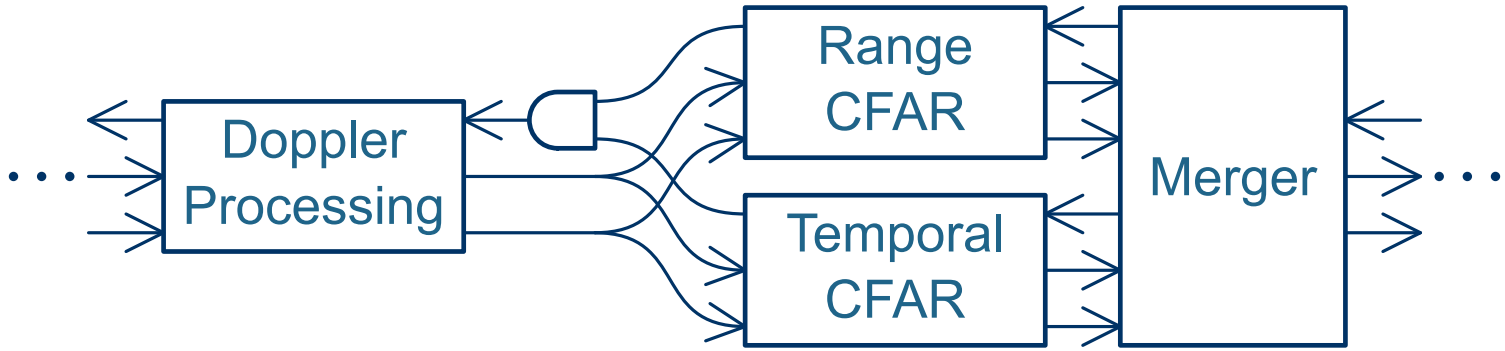
- Gate the “Valid” with the “Ready”
- The “Merger” can take various forms:
  - Interleave
  - Alternate (sent one logical unit from the one, then the other, then repeat)
  - Combine through calculation
  - etc.

Complex Structures



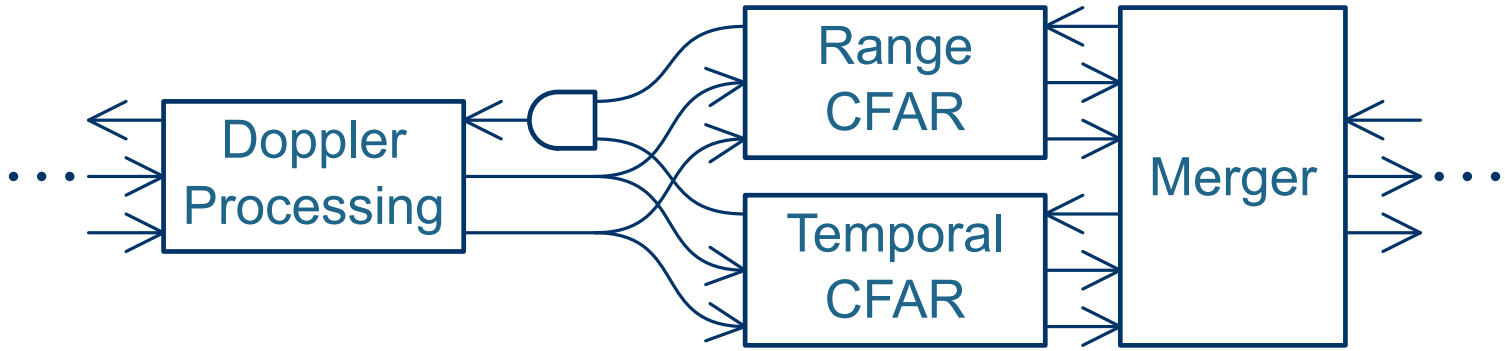
- Gate the “Valid” with the “Ready”
- The “Merger” can take various forms:
  - Interleave
  - Alternate (sent one logical unit from the one, then the other, then repeat)
  - Combine through calculation
  - etc.

Complex Structures



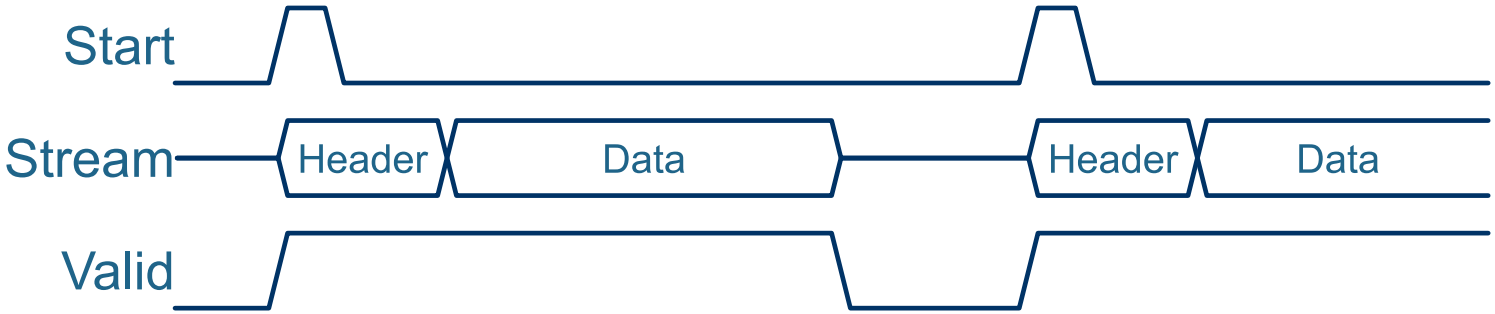
- Gate the “Valid” with the “Ready”
- The “Merger” can take various forms:
  - Interleave
  - Alternate (sent one logical unit from the one, then the other, then repeat)
  - Combine through calculation
  - etc.

Complex Structures



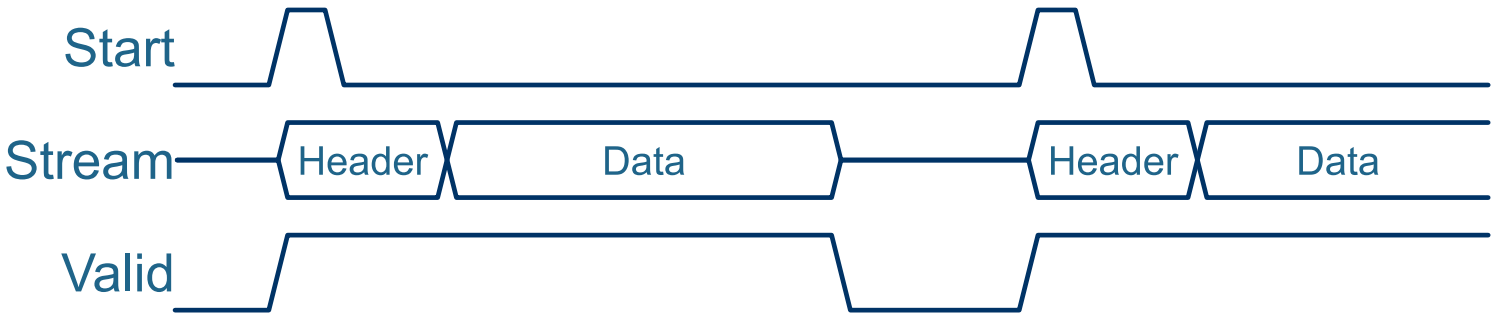
- Gate the “Valid” with the “Ready”
- The “Merger” can take various forms:
  - Interleave
  - Alternate (sent one logical unit from the one, then the other, then repeat)
  - Combine through calculation
  - etc.

Packet-based Streams



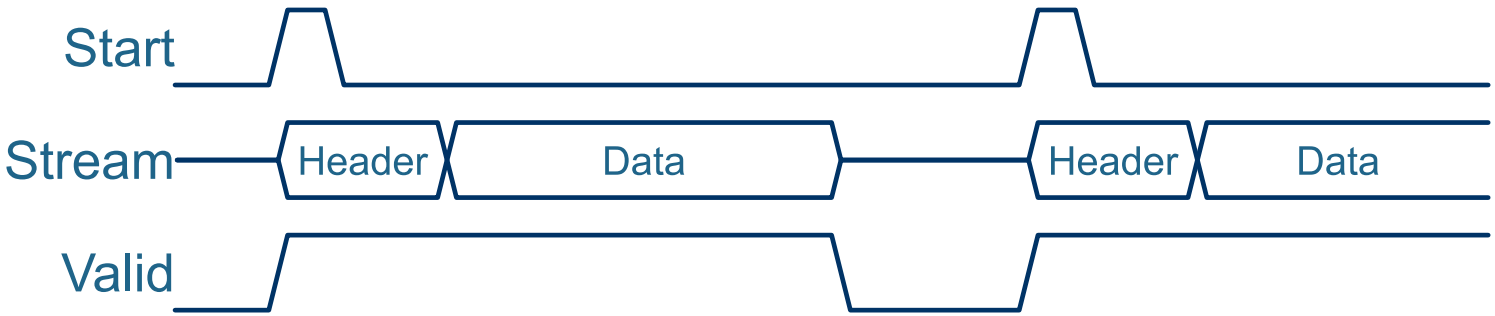
- Add a start-of-packet strobe (valid for one clock-cycle only)
- Natural fit for RADAR: one packet per PRI
- Natural fit for packet-based communication
  (eg. Ethernet or UDP – makes it easy to implement Ethernet-based FPGA-in-the-loop testing)
- The header can contain all sorts of metadata...

Packet-based Streams



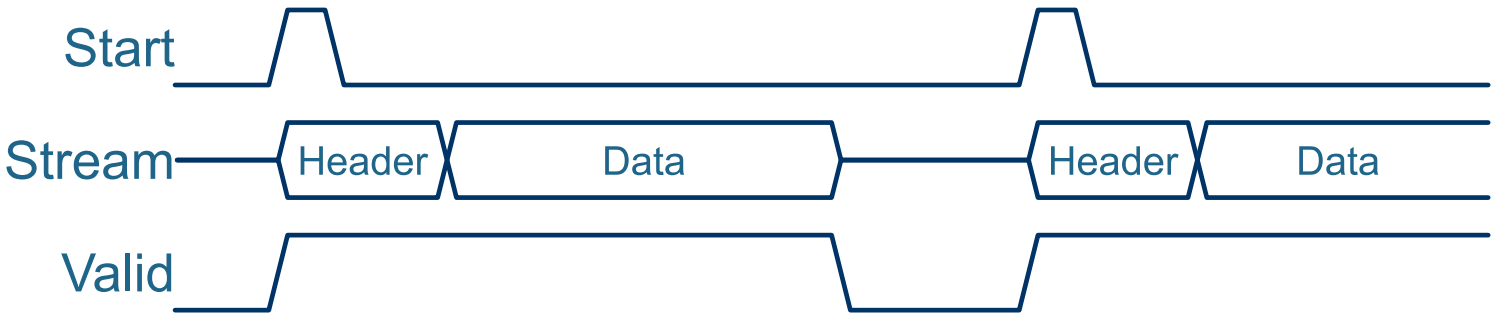
- Add a start-of-packet strobe (valid for one clock-cycle only)
- Natural fit for RADAR: one packet per PRI
- Natural fit for packet-based communication (eg. Ethernet or UDP – makes it easy to implement Ethernet-based FPGA-in-the-loop testing)
- The header can contain all sorts of metadata...

Packet-based Streams



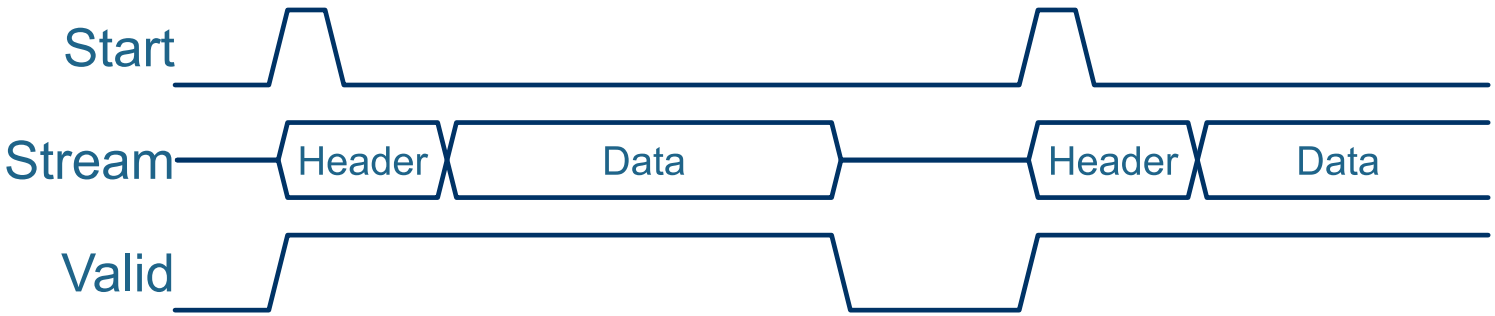
- Add a start-of-packet strobe (valid for one clock-cycle only)
- Natural fit for RADAR: one packet per PRI
- Natural fit for packet-based communication (eg. Ethernet or UDP – makes it easy to implement Ethernet-based FPGA-in-the-loop testing)
- The header can contain all sorts of metadata...

Packet-based Streams



- Add a start-of-packet strobe (valid for one clock-cycle only)
- Natural fit for RADAR: one packet per PRI
- Natural fit for packet-based communication
  (eg. Ethernet or UDP – makes it easy to implement Ethernet-based FPGA-in-the-loop testing)
- The header can contain all sorts of metadata...

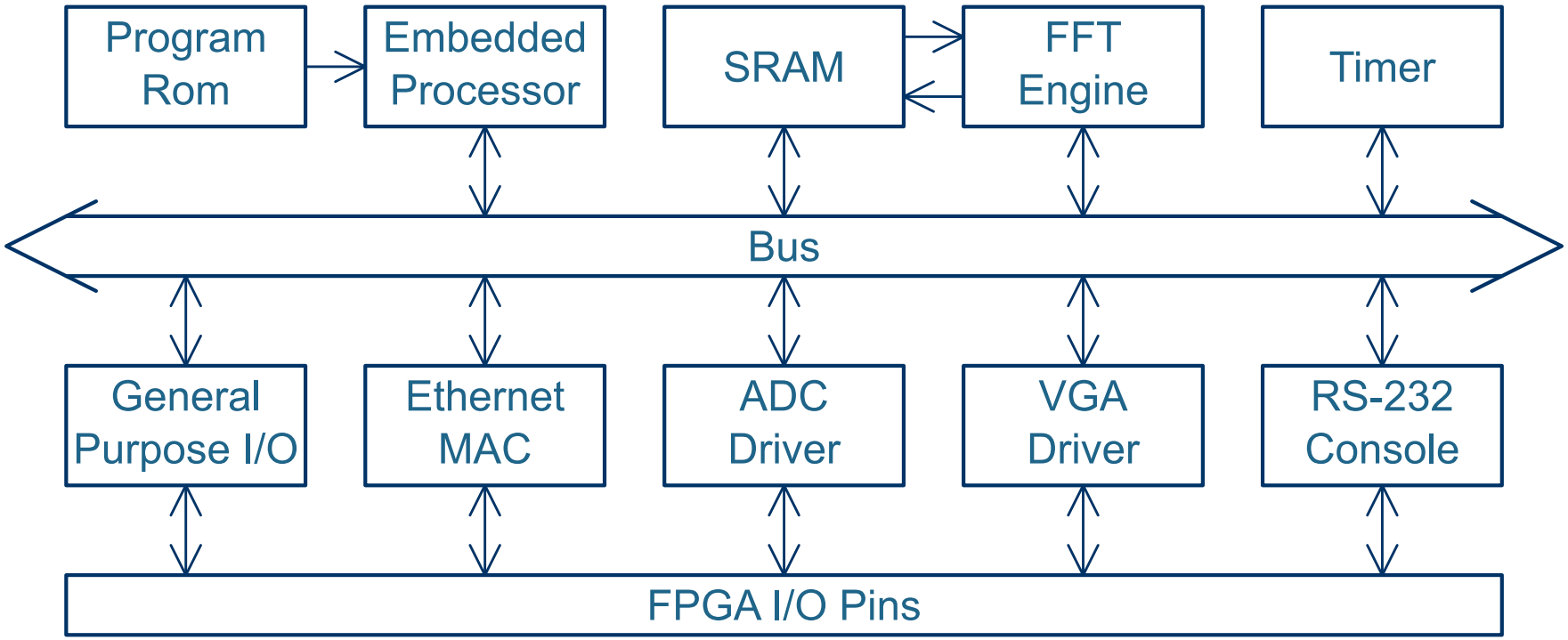
Packet-based Streams



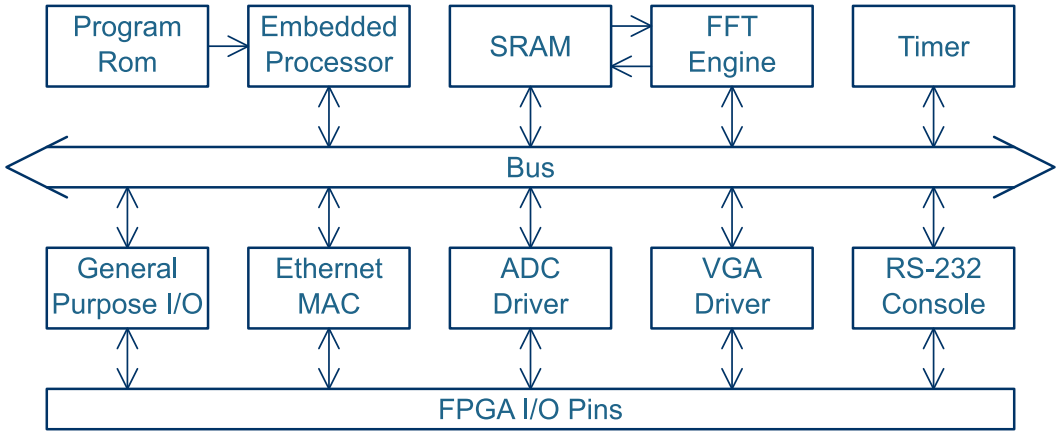
- Add a start-of-packet strobe (valid for one clock-cycle only)
- Natural fit for RADAR: one packet per PRI
- Natural fit for packet-based communication
  (eg. Ethernet or UDP – makes it easy to implement Ethernet-based FPGA-in-the-loop testing)
- The header can contain all sorts of metadata...

Outline

Memory-Mapped Bus

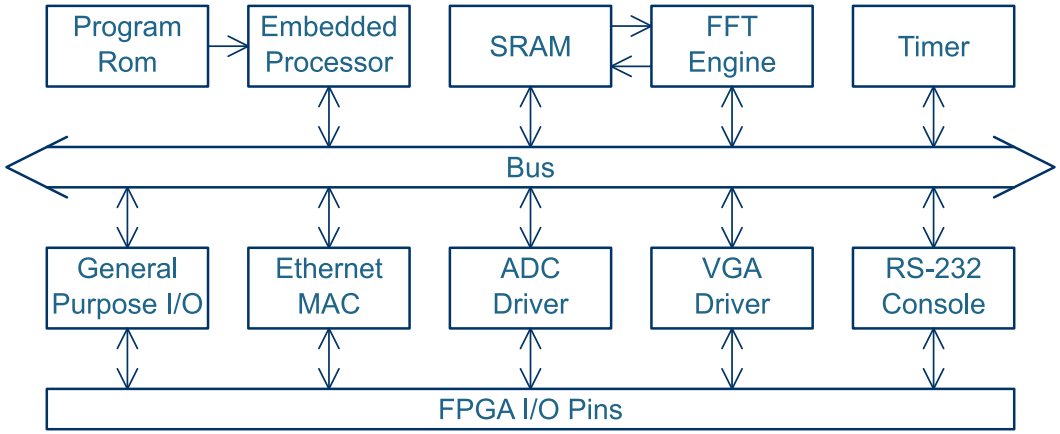


Memory-Mapped Bus



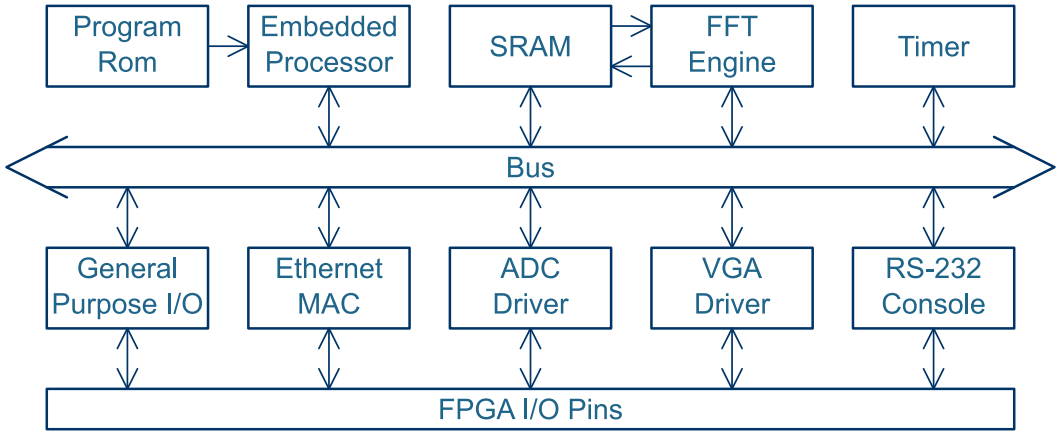
- Every node on the bus has an address range allocated
- Generally used for writing control registers, reading system status and accessing memory
- Altera Qsys uses Avalon
- Xilinx IP Integrator and ARM processors use AXI
- Many open-source projects use Wishbone

Memory-Mapped Bus



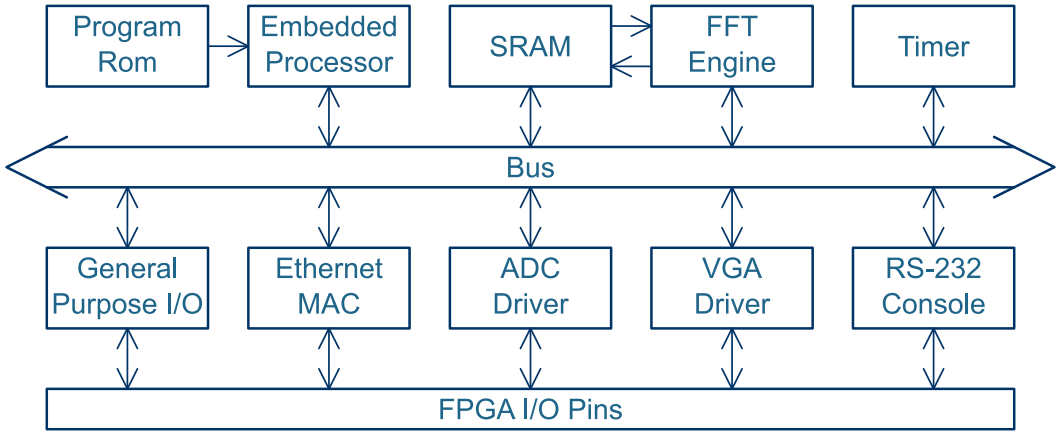
- Every node on the bus has an address range allocated
- Generally used for writing control registers, reading system status and accessing memory
- Altera Qsys uses Avalon
- Xilinx IP Integrator and ARM processors use AXI
- Many open-source projects use Wishbone

Memory-Mapped Bus



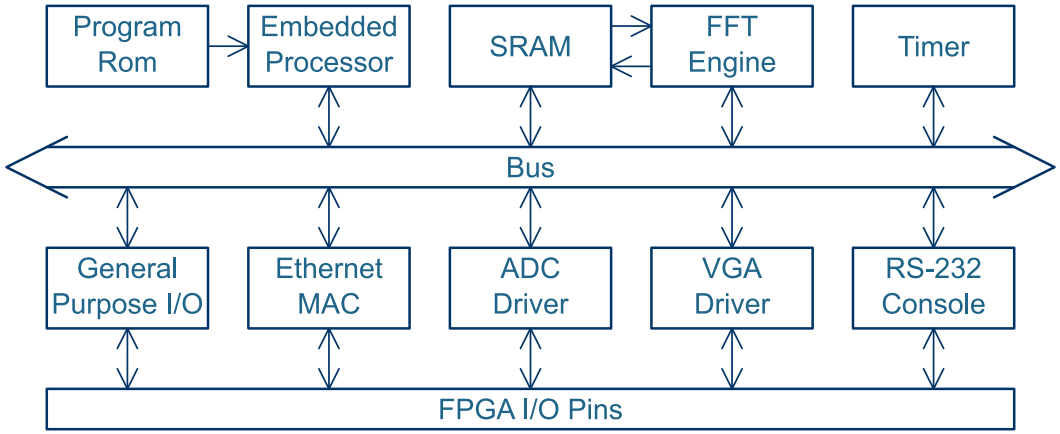
- Every node on the bus has an address range allocated
- Generally used for writing control registers, reading system status and accessing memory
- Altera Qsys uses Avalon
- Xilinx IP Integrator and ARM processors use AXI
- Many open-source projects use Wishbone

Memory-Mapped Bus



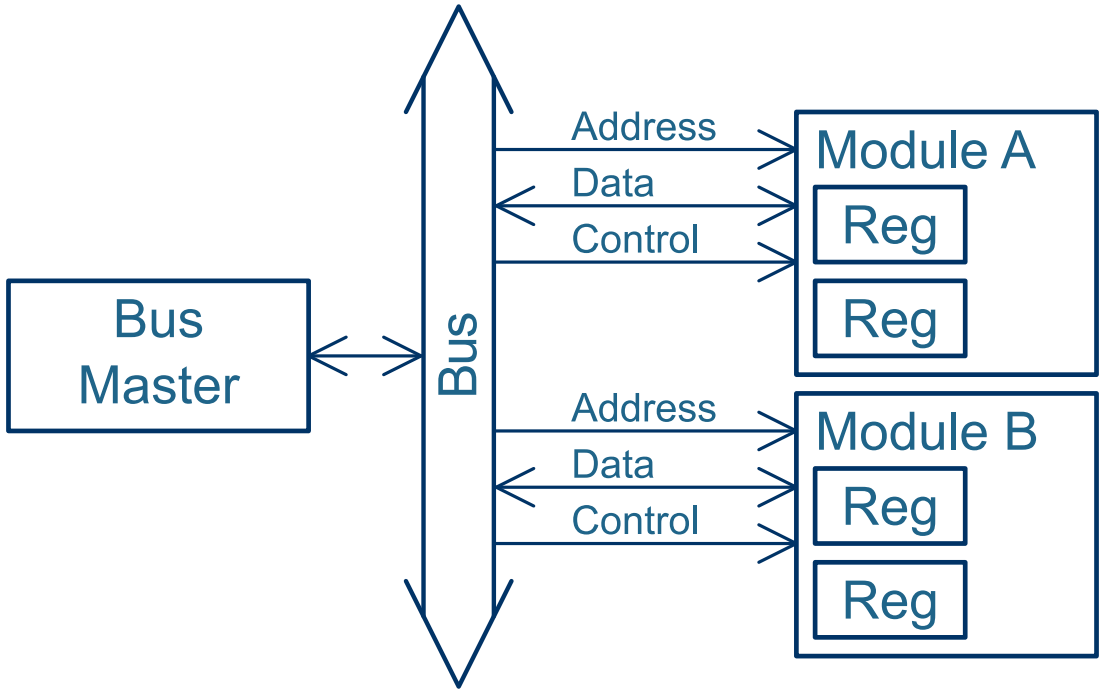
- Every node on the bus has an address range allocated
- Generally used for writing control registers, reading system status and accessing memory
- Altera Qsys uses Avalon
- Xilinx IP Integrator and ARM processors use AXI
- Many open-source projects use Wishbone

Memory-Mapped Bus

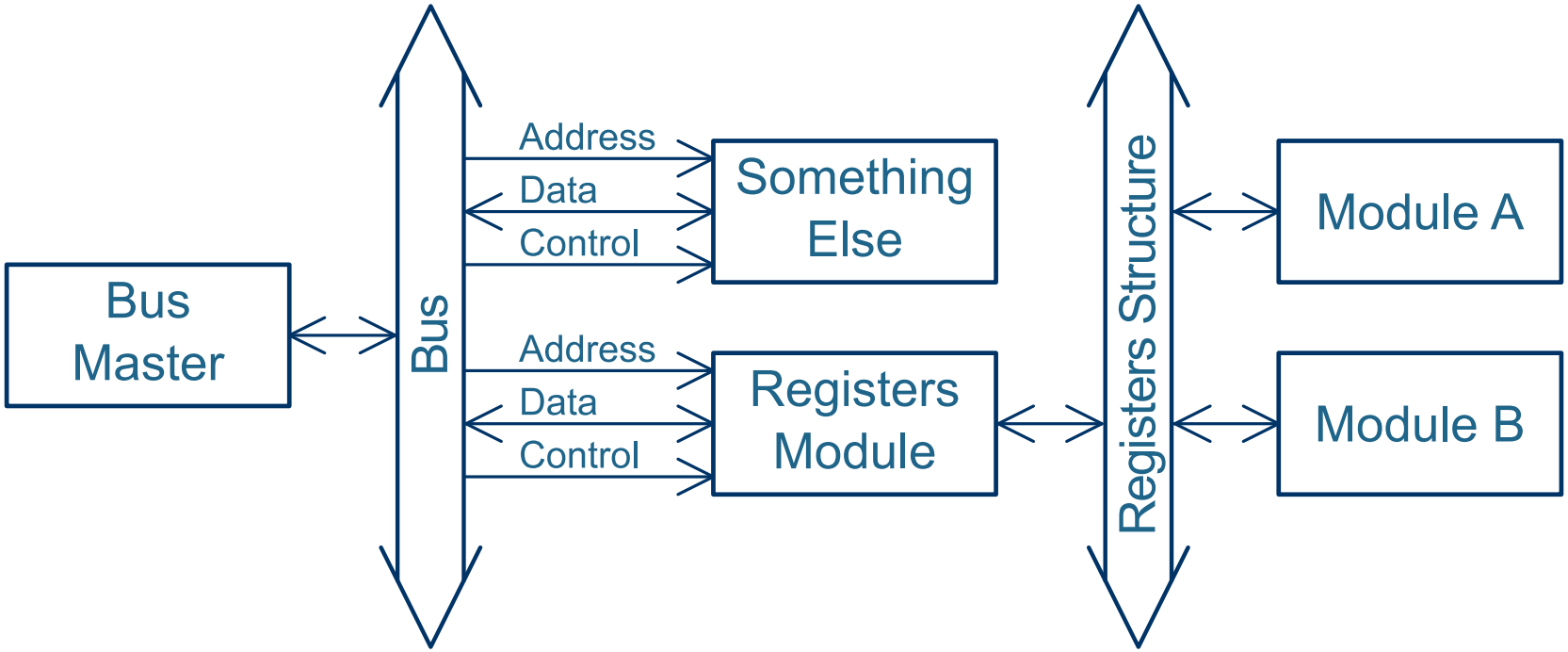


- Every node on the bus has an address range allocated
- Generally used for writing control registers, reading system status and accessing memory
- Altera Qsys uses Avalon
- Xilinx IP Integrator and ARM processors use AXI
- Many open-source projects use Wishbone

Memory-mapped Architectures



Memory-mapped Architectures



Structures

- SystemVerilog only
- Define these once, in global space: typically in the “Registers Module” source file

```
typedef struct{
 logic [ 1:0]Buttons;
 logic [ 9:0]Switches;
 logic [31:0]StatusBits;
} RD_REGISTERS;

typedef struct{
 logic [ 9:0]LEDs;
 logic [31:0]NCO_Frequency;
 logic [ 2:0]Bandwidth;
} WR_REGISTERS;
```

Structures

- SystemVerilog only
- Define these once, in global space: typically in the “Registers Module” source file

```
typedef struct{
 logic [ 1:0]Buttons;
 logic [ 9:0]Switches;
 logic [31:0]StatusBits;
} RD_REGISTERS;

typedef struct{
 logic [ 9:0]LEDs;
 logic [31:0]NCO_Frequency;
 logic [ 2:0]Bandwidth;
} WR_REGISTERS;
```

Structures


```
// struct typedefs go here

module Registers(
 input Clk, Reset,

 input  [15:0]Address,
 output [31:0]RdData,
 input  [31:0]WrData,
 input        WrEn,

 input  RD_REGISTERS RdRegisters,
 output WR_REGISTERS WrRegisters
);
...
```

Structures


```
// In the top-level module...

RD_REGISTERS RdRegisters;
WR_REGISTERS WrRegisters;

Registers Registers_Inst(
 Clk, Reset,
 Address, RdData, WrData, WrEn,
 RdRegisters, WrRegisters
);

NCO NCO_Inst(
 Clk, Reset,
 WrRegisters.NCO_Frequency,
 RdRegisters.StatusBits[15]
);
```

Structures

- You can map a structure to an input port and then use only what you need within the submodule
- You can not map the same structure to an output port of more than one module – you need to map the structure members individually
- Luckily, most registers are control registers, and therefore input ports of the target modules

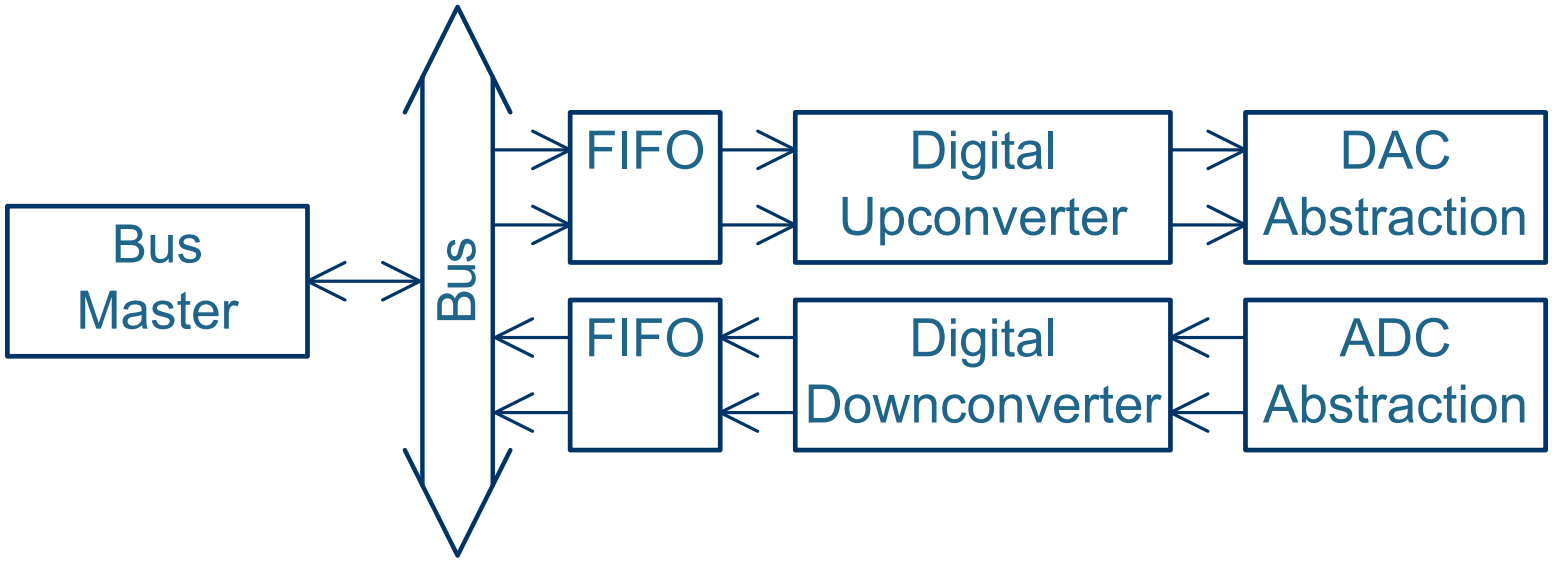
Structures

- You can map a structure to an input port and then use only what you need within the submodule
- You can not map the same structure to an output port of more than one module – you need to map the structure members individually
- Luckily, most registers are control registers, and therefore input ports of the target modules

Structures

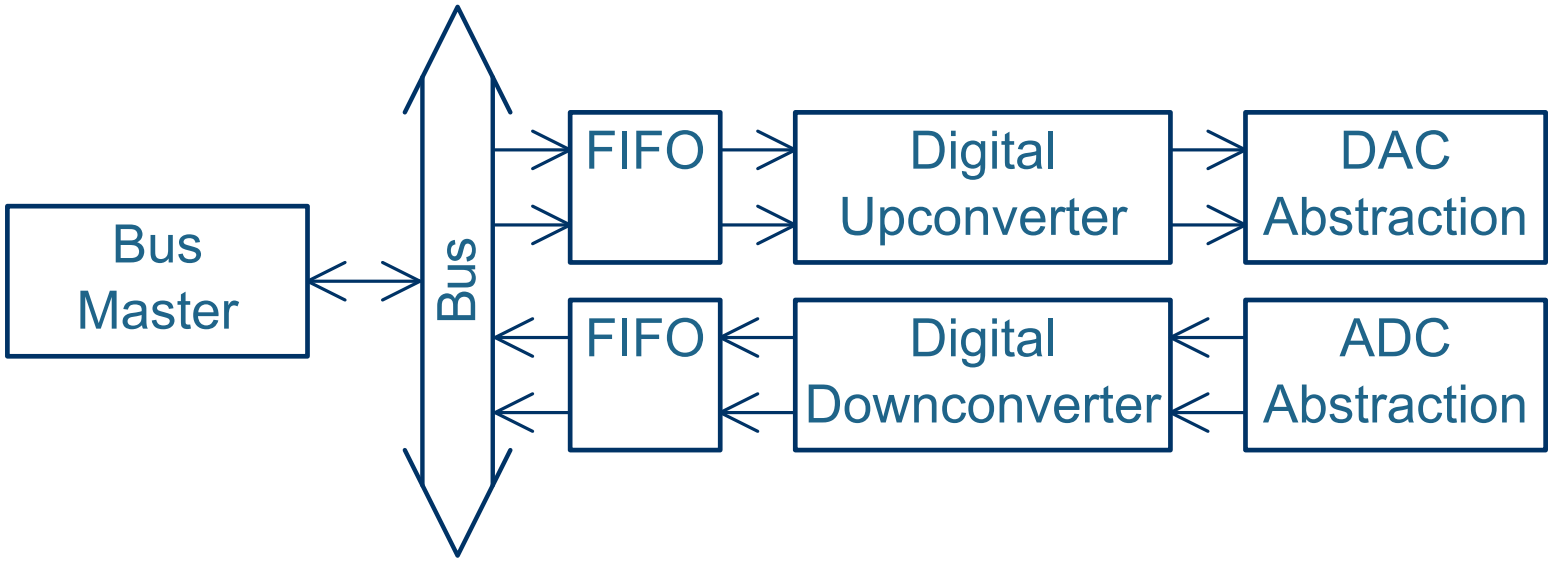
- You can map a structure to an input port and then use only what you need within the submodule
- You can not map the same structure to an output port of more than one module – you need to map the structure members individually
- Luckily, most registers are control registers, and therefore input ports of the target modules

Memory-mapped Streams



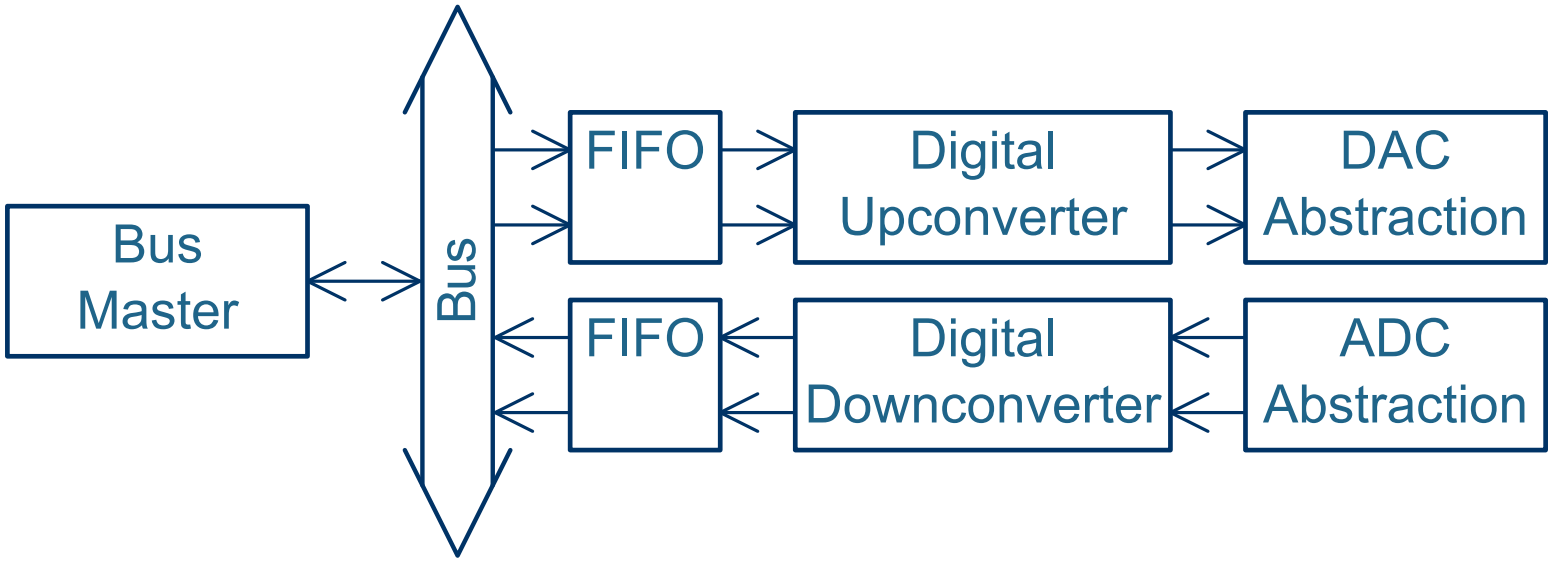
- It is often convenient to allocate a bus address to a stream
- A write to that address injects one unit into the stream
- A read from that address reads one unit from the stream
- Most often unidirectional and FIFO-buffered with feed-back registers

Memory-mapped Streams



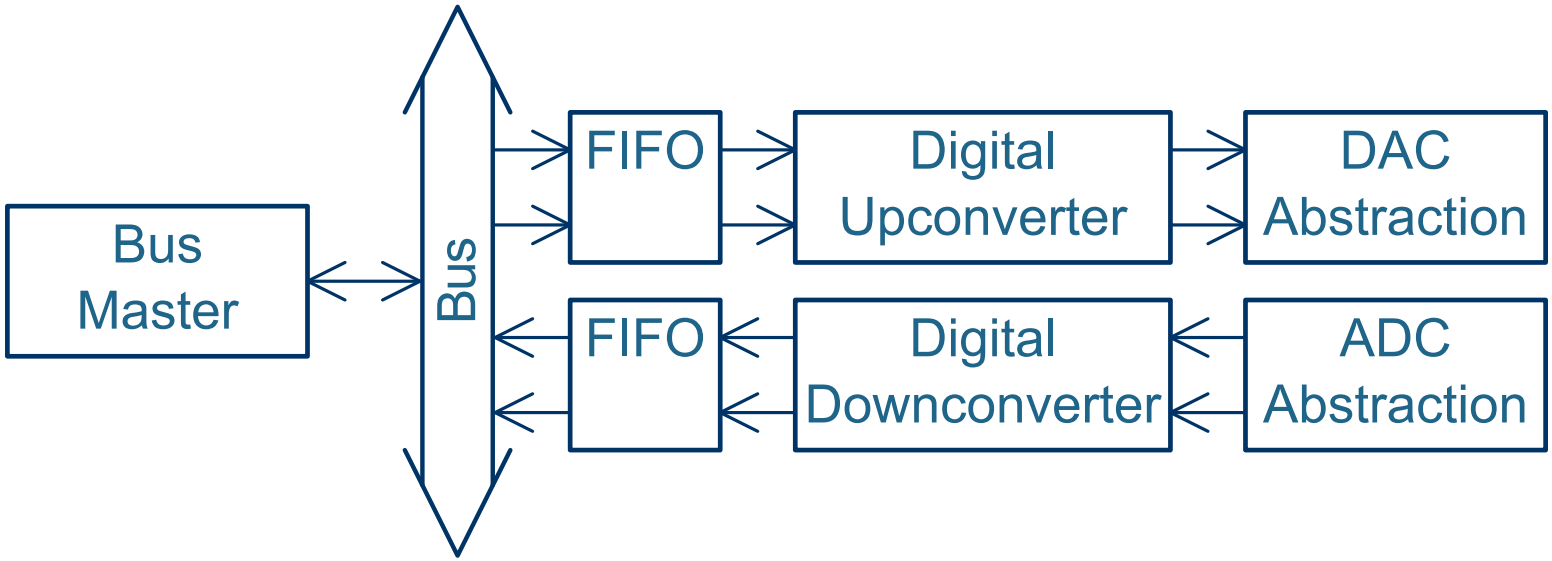
- It is often convenient to allocate a bus address to a stream
- A write to that address injects one unit into the stream
- A read from that address reads one unit from the stream
- Most often unidirectional and FIFO-buffered with feed-back registers

Memory-mapped Streams



- It is often convenient to allocate a bus address to a stream
- A write to that address injects one unit into the stream
- A read from that address reads one unit from the stream
- Most often unidirectional and FIFO-buffered with feed-back registers

Memory-mapped Streams

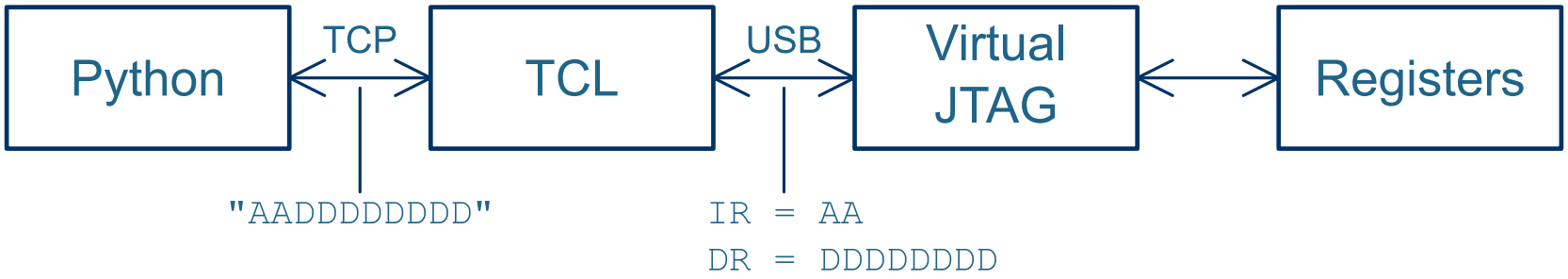


- It is often convenient to allocate a bus address to a stream
- A write to that address injects one unit into the stream
- A read from that address reads one unit from the stream
- Most often unidirectional and FIFO-buffered with feed-back registers

Coffee Break...

Outline

Virtual JTAG MM-Bus



This section has been adapted from Idle Logic Labs

http://idlelogiclabs.com/2012/04/15/talking-to-the-de0-nano-using-the-virtual-jtag-interface/

Virtual JTAG MM-Bus (Verilog)


```
wire tck, tdi; reg tdo;
wire Capture, Shift, Update;

wire WriteEnable; wire [6:0]Address; reg [31:0]Data;

sld_virtual_jtag #(
 .sld_auto_instance_index("NO"),
 .sld_instance_index     (0), // Make sure these don't clash
 .sld_ir_width           (8)
)virtual_jtag_0(
 .tck(tck), .tdi(tdi), .tdo(tdo),
 .ir_in             ({WriteEnable, Address}),
 .virtual_state_cdr(Capture),
 .virtual_state_sdr(Shift  ),
 .virtual_state_udr(Update )
);
```

Virtual JTAG MM-Bus (Verilog)


```
always @(posedge tck) begin
 case(1'b1)
  Capture: begin
   case(Address)
    7'h0   : {Data[30:0], tdo} <= {20'd0, ~nKEY, SW};
    7'h1   : {Data[30:0], tdo} <= {22'd0, LED};
    7'h2   : {Data[30:0], tdo} <= { 8'd0, SevenSegments};
    7'h3   : {Data[30:0], tdo} <= {22'd0, DutyCycle};
    default: {Data[30:0], tdo} <= 32'h_XXXX_XXXX;
   endcase
  end
```

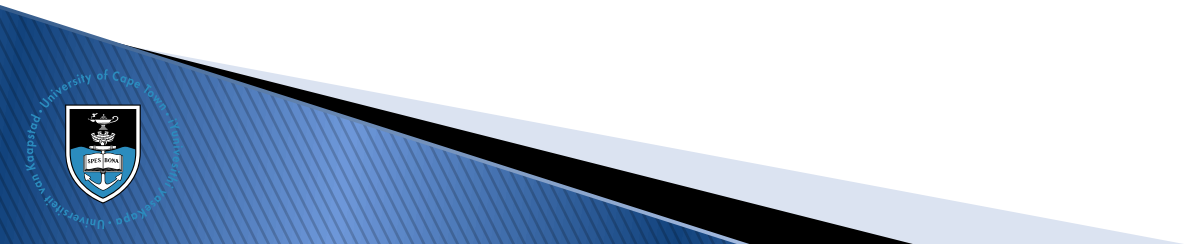
Virtual JTAG MM-Bus (Verilog)


```
  Shift: {Data, tdo} <= {tdi, Data};

  Update: begin
   if(WriteEnable) begin
    case(Address)
     7'h1    : LED           <= Data[ 9:0];
     7'h2    : SevenSegments <= Data[23:0];
     7'h3    : DutyCycle     <= Data[ 9:0];
     default:;
    endcase
   end
  end

  default:;
 endcase
end
```

Virtual JTAG MM-Bus (TCL)

- Mostly the same as the “WriteData.tcl” script:
  1. Select the first available hardware
  2. Select the first device on the chain
  3. Open the device
  4. And then:

```
proc OnClientConnect {Channel Address Port} {
 # This is the socket connection event handler
}

socket -server OnClientConnect 12288
vwait forever

close_device
```



Virtual JTAG MM-Bus (TCL)

- Mostly the same as the “WriteData.tcl” script:
  1. Select the first available hardware
  2. Select the first device on the chain
  3. Open the device
  4. And then:

```
proc OnClientConnect {Channel Address Port} {
 # This is the socket connection event handler
}

socket -server OnClientConnect 12288
vwait forever

close_device
```

Virtual JTAG MM-Bus (TCL)

- Mostly the same as the “WriteData.tcl” script:
  1. Select the first available hardware
  2. Select the first device on the chain
  3. Open the device
  4. And then:

```
proc OnClientConnect {Channel Address Port} {
 # This is the socket connection event handler
}

socket -server OnClientConnect 12288
vwait forever

close_device
```

Virtual JTAG MM-Bus (TCL)

- Mostly the same as the “WriteData.tcl” script:
  1. Select the first available hardware
  2. Select the first device on the chain
  3. Open the device
  4. And then:

```
proc OnClientConnect {Channel Address Port} {
  # This is the socket connection event handler
}

socket -server OnClientConnect 12288
vwait forever

close_device
```

Virtual JTAG MM-Bus (TCL)

- Mostly the same as the “WriteData.tcl” script:
  1. Select the first available hardware
  2. Select the first device on the chain
  3. Open the device
  4. And then:

```
proc OnClientConnect {Channel Address Port} {
 # This is the socket connection event handler
}

socket -server OnClientConnect 12288
vwait forever

close_device
```

Virtual JTAG MM-Bus (TCL)

- Mostly the same as the “WriteData.tcl” script:
  1. Select the first available hardware
  2. Select the first device on the chain
  3. Open the device
  4. And then:

```
proc OnClientConnect {Channel Address Port} {
 # This is the socket connection event handler
}

socket -server OnClientConnect 12288
vwait forever

close_device
```

Virtual JTAG MM-Bus (TCL)


```
proc OnClientConnect {Channel Address Port} {
 fconfigure $Channel -buffering line

 puts "Connection from $Address, port $Port"

 while 1 {
  gets $Channel Data

  set DataLength [string length $Data]

  # Strip the header (to be used for something else later)
  set Header [string range $Data 0 1]
  set Data   [string range $Data 2 [expr {$DataLength - 1}]]

  # Strip the trailing newline
  set Data    [string trimright $Data]
  set NumBits [expr {($DataLength - 2) * 4}]
```

Virtual JTAG MM-Bus (TCL)


```
if {$NumBits > 0} {
 puts "Writing $Data to IR $Header"

 device_lock -timeout 1000
  device_virtual_ir_shift \
   -instance_index 0      \
   -ir_value "0x$Header"  \
   -no_captured_ir_value

  set Result [             \
   device_virtual_dr_shift \
    -dr_value $Data        \
    -instance_index 0      \
    -length $NumBits       \
    -value_in_hex          \
  ]
```

Virtual JTAG MM-Bus (TCL)


```
    # Force a DR update by shifting IR
    device_virtual_ir_shift \
     -instance_index 0      \
     -ir_value       0      \
     -no_captured_ir_value

   device_unlock

   puts $Channel $Result

  } else {
   break
  }
 }
}
```

Virtual JTAG MM-Bus (Python)


```
import time
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM);
sock.connect(("localhost", 12288));

while True:
  # Read the buttons and switches
  # (the new-line flushes the stream)
  msg = "0000000000\n"; sock.sendall(msg.encode('utf-8'));
  Buttons = sock.recv(10); # TCL appends \r\n to the data
  Buttons = Buttons[0:8];  # This strips it
  print(Buttons);
```

Virtual JTAG MM-Bus (Python)


```
  # Set the LEDs to the buttons and switches
  msg = "81" + Buttons.decode() + "\n";
  sock.sendall(msg.encode('utf-8'));
  sock.recv(10);

  # Set the PWM duty-cycle to the buttons and switches
  msg = "83" + Buttons.decode() + "\n";
  sock.sendall(msg.encode('utf-8'));
  sock.recv(10);

  # Write the time to the 7-segment display
  msg = "8200" + time.strftime("%H%M%S") + "\n";
  sock.sendall(msg.encode('utf-8'));
  sock.recv(10);

  time.sleep(0.1);
sock.close();
```

Note...

- The same script can be used to write data to SDRAM
- The header can be adapted to support larger address words, or more than one Virtual JTAG instance ID
- In essence: anything that can open a TCP socket can now be used to control your FPGA
- You can control your FPGA remotely
- Note, however, that the virtual JTAG interface has a lot of overhead, so if you have lots of data to transfer, keep the transactions as large as possible (32-bit transactions get about 1 kb/s, whereas sending 64 MiB all at once gets about 3 Mb/s)

Note...

- The same script can be used to write data to SDRAM
- The header can be adapted to support larger address words, or more than one Virtual JTAG instance ID
- In essence: anything that can open a TCP socket can now be used to control your FPGA
- You can control your FPGA remotely
- Note, however, that the virtual JTAG interface has a lot of overhead, so if you have lots of data to transfer, keep the transactions as large as possible (32-bit transactions get about 1 kb/s, whereas sending 64 MiB all at once gets about 3 Mb/s)

Note...

- The same script can be used to write data to SDRAM
- The header can be adapted to support larger address words, or more than one Virtual JTAG instance ID
- In essence: anything that can open a TCP socket can now be used to control your FPGA
- You can control your FPGA remotely
- Note, however, that the virtual JTAG interface has a lot of overhead, so if you have lots of data to transfer, keep the transactions as large as possible (32-bit transactions get about 1 kb/s, whereas sending 64 MiB all at once gets about 3 Mb/s)

Note...

- The same script can be used to write data to SDRAM
- The header can be adapted to support larger address words, or more than one Virtual JTAG instance ID
- In essence: anything that can open a TCP socket can now be used to control your FPGA
- You can control your FPGA remotely
- Note, however, that the virtual JTAG interface has a lot of overhead, so if you have lots of data to transfer, keep the transactions as large as possible (32-bit transactions get about 1 kb/s, whereas sending 64 MiB all at once gets about 3 Mb/s)

Note...

- The same script can be used to write data to SDRAM
- The header can be adapted to support larger address words, or more than one Virtual JTAG instance ID
- In essence: anything that can open a TCP socket can now be used to control your FPGA
- You can control your FPGA remotely
- Note, however, that the virtual JTAG interface has a lot of overhead, so if you have lots of data to transfer, keep the transactions as large as possible (32-bit transactions get about 1 kb/s, whereas sending 64 MiB all at once gets about 3 Mb/s)

Outline

Pipelines

Streaming Processors

Memory-Mapped Bus

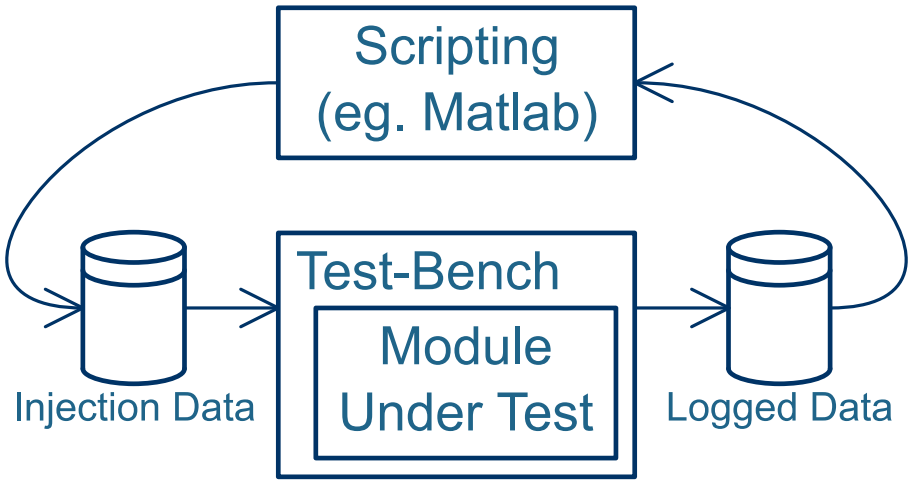
Example: Virtual JTAG MM-Bus Control Interface

Advanced Simulation

Practical

THE RADAR MASTERS COURSE

Advanced Simulation



File I/O


```
`timescale 1ns/1ns
module File_IO_TB;

integer Input_File;
integer Output_File;

initial begin
 Input_File  = $fopen("Test_File.txt", "rb");
 Output_File = $fopen("Output.txt"   , "wb");
 $fwrite(
  Output_File,
  "Time [ns], Data [Hex], Data [Binary]\n"
 );
end

reg Clk = 0; always #5 Clk <= !Clk; // 100 MHz
```

File I/O


```
reg [7:0]Byte; reg [15:0]Data; integer Result;

always @(posedge Clk) begin
 Result = $fread(Byte, Input_File); Data[ 7:0] = Byte;
 Result = $fread(Byte, Input_File); Data[15:8] = Byte;

 if(Result < 1) begin
  $fclose(Input_File); $fclose(Output_File); $stop;
 end

 $fwrite(Output_File, "%d, %04X, %s\n", $time, Data, Data);
end

endmodule
```

File I/O

Consult the Verilog-2001 standard:

- Section 17.2.4: “Reading data from a file”
- Section 17.2.8: “Loading memory data from a file”
- `$fread` reads any size register, but in big-endian
- `$fscanf` reads formatted data, including raw binary
- `$fwrite` works similar to `fprintf` from C, except that `%u` writes raw binary data
- `$fmonitor` logs to file whenever the signals change

File I/O

Consult the Verilog-2001 standard:

- Section 17.2.4: “Reading data from a file”
- Section 17.2.8: “Loading memory data from a file”
- `$fread` reads any size register, but in big-endian
- `$fscanf` reads formatted data, including raw binary
- `$fwrite` works similar to `fprintf` from C, except that `%u` writes raw binary data
- `$fmonitor` logs to file whenever the signals change

File I/O

Consult the Verilog-2001 standard:

- Section 17.2.4: “Reading data from a file”
- Section 17.2.8: “Loading memory data from a file”
- `$fread` reads any size register, but in big-endian
- `$fscanf` reads formatted data, including raw binary
- `$fwrite` works similar to `fprintf` from C, except that `%u` writes raw binary data
- `$fmonitor` logs to file whenever the signals change

File I/O

Consult the Verilog-2001 standard:

- Section 17.2.4: “Reading data from a file”
- Section 17.2.8: “Loading memory data from a file”
- `$fread` reads any size register, but in big-endian
- `$fscanf` reads formatted data, including raw binary
- `$fwrite` works similar to `fprintf` from C, except that `%u` writes raw binary data
- `$fmonitor` logs to file whenever the signals change

File I/O

Consult the Verilog-2001 standard:

- Section 17.2.4: “Reading data from a file”
- Section 17.2.8: “Loading memory data from a file”
- `$fread` reads any size register, but in big-endian
- `$fscanf` reads formatted data, including raw binary
- `$fwrite` works similar to `fprintf` from C, except that `%u` writes raw binary data
- `$fmonitor` logs to file whenever the signals change

File I/O

Consult the Verilog-2001 standard:

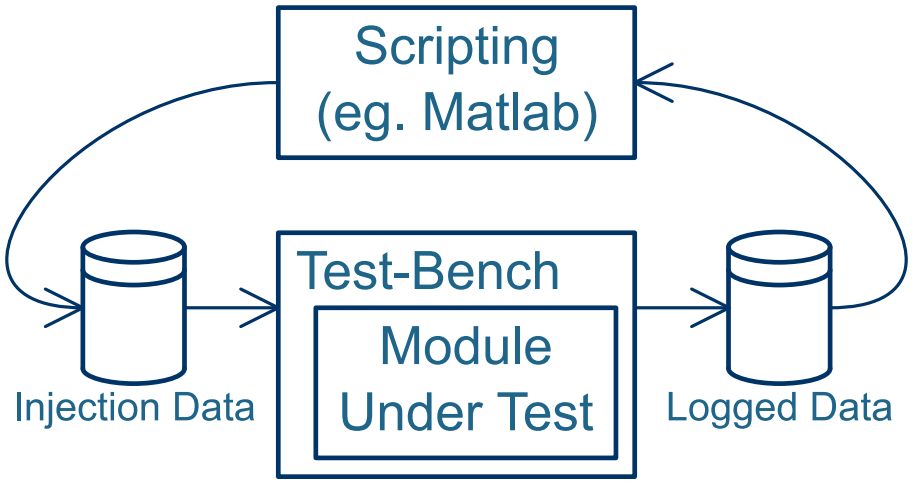
- Section 17.2.4: “Reading data from a file”
- Section 17.2.8: “Loading memory data from a file”
- `$fread` reads any size register, but in big-endian
- `$fscanf` reads formatted data, including raw binary
- `$fwrite` works similar to `fprintf` from C, except that `%u` writes raw binary data
- `$fmonitor` logs to file whenever the signals change

File I/O

Consult the Verilog-2001 standard:

- Section 17.2.4: “Reading data from a file”
- Section 17.2.8: “Loading memory data from a file”
- `$fread` reads any size register, but in big-endian
- `$fscanf` reads formatted data, including raw binary
- `$fwrite` works similar to `fprintf` from C, except that `%u` writes raw binary data
- `$fmonitor` logs to file whenever the signals change

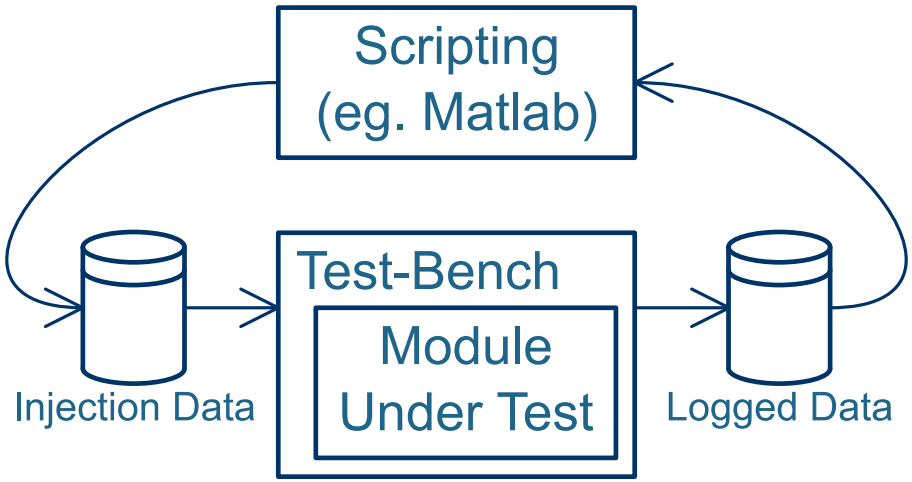
Data Injection (ADXL345_TB)



- Create a test file “ADXL345_TB.dat”:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Data Injection (ADXL345_TB)



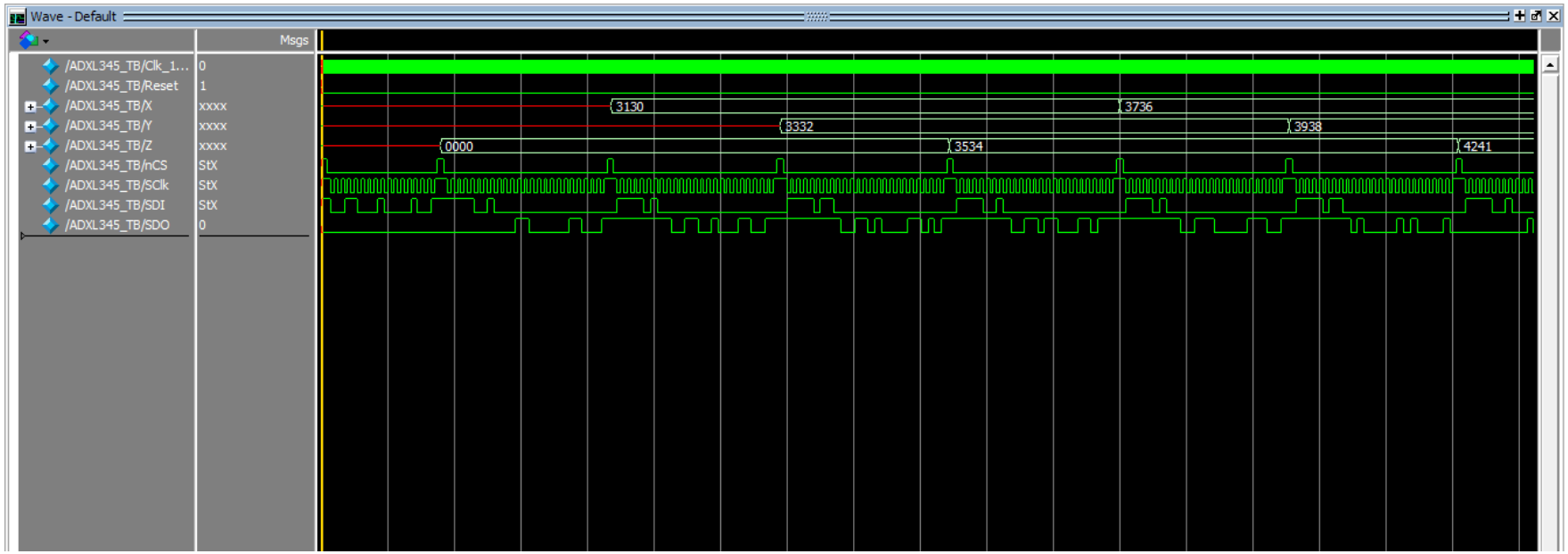
- Create a test file “ADXL345_TB.dat”:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

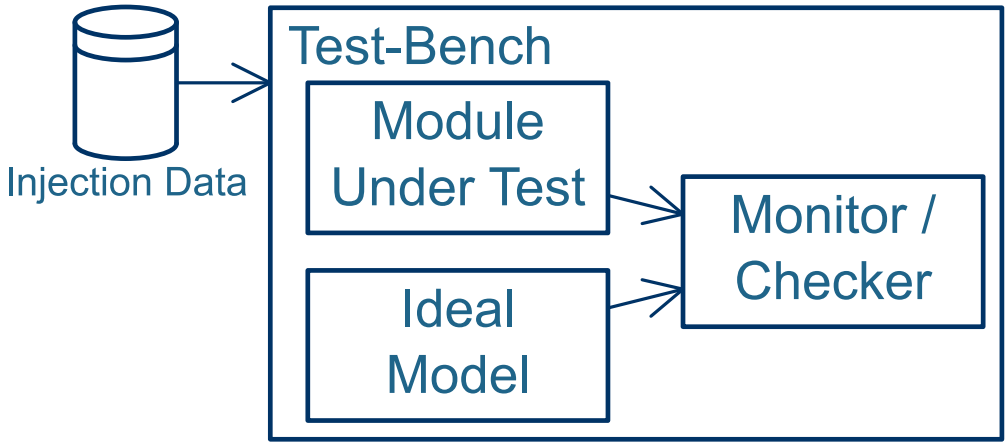
Data Injection (ADXL345_TB)


```
integer   File, j;
reg [15:0]Data;

initial begin
 File = $fopen("../Peripherals/ADXL345_TB.dat", "rb");

 for(j = 0; j < 16; j = j + 1) @(negedge SClk);
 forever begin
  for(j = 0; j <  8; j = j + 1) @(negedge SClk);
  $fread(Data, File);
  for(j = 0; j < 16; j = j + 1) begin
   @(negedge SClk);
   #40 {SDO, Data[15:1]} <= Data;
  end
 end
end
```

Data Injection (ADXL345_TB)

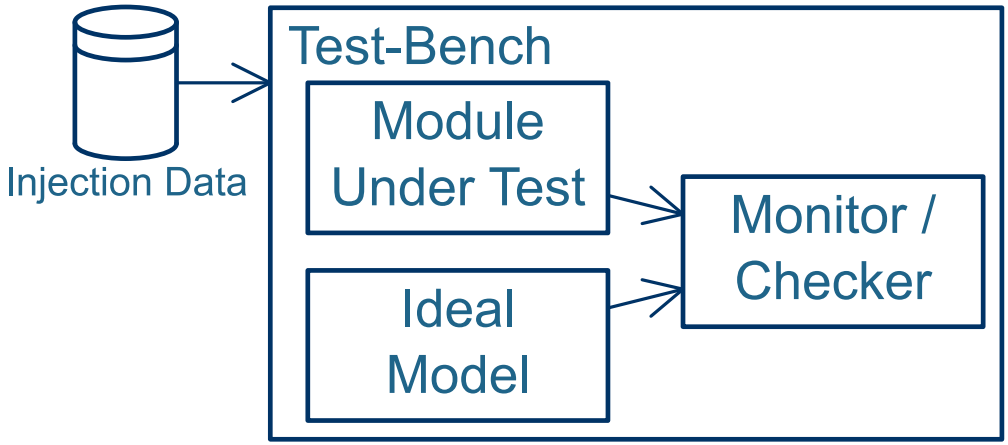


Automated Test-benches



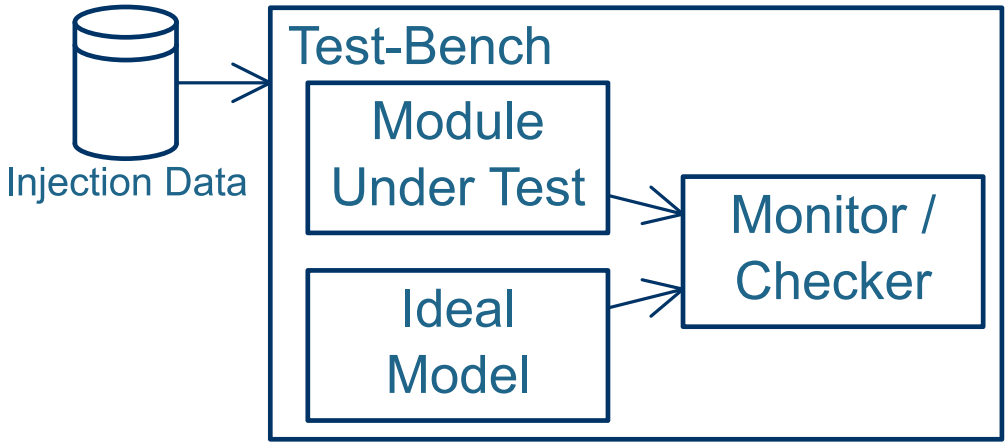
- Very useful, but time-consuming to write...
- The “Ideal Model” can be read from a file, or calculated in the test-bench
- The “Monitor” issues errors (with `$error`), warnings (with `$warning`) or information (with `$info`) as appropriate

Automated Test-benches



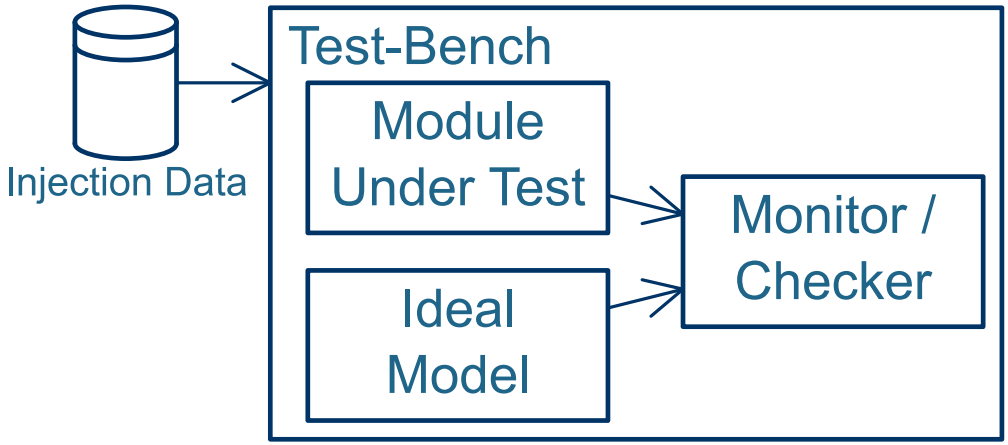
- Very useful, but time-consuming to write...
- The “Ideal Model” can be read from a file, or calculated in the test-bench
- The “Monitor” issues errors (with `$error`), warnings (with `$warning`) or information (with `$info`) as appropriate

Automated Test-benches



- Very useful, but time-consuming to write...
- The “Ideal Model” can be read from a file, or calculated in the test-bench
- The “Monitor” issues errors (with `$error`), warnings (with `$warning`) or information (with `$info`) as appropriate

Automated Test-benches



- Very useful, but time-consuming to write...
- The “Ideal Model” can be read from a file, or calculated in the test-bench
- The “Monitor” issues errors (with `$error`), warnings (with `$warning`) or information (with `$info`) as appropriate

Outline

Pipelines

Streaming Processors

Memory-Mapped Bus

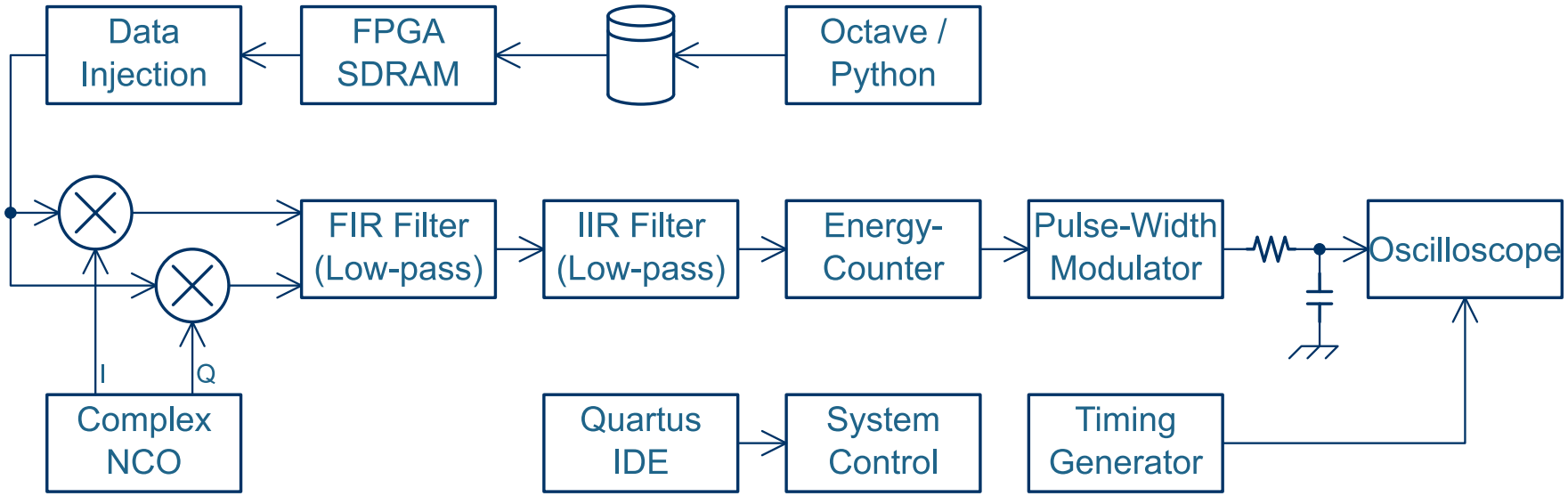
Example: Virtual JTAG MM-Bus Control Interface

Advanced Simulation

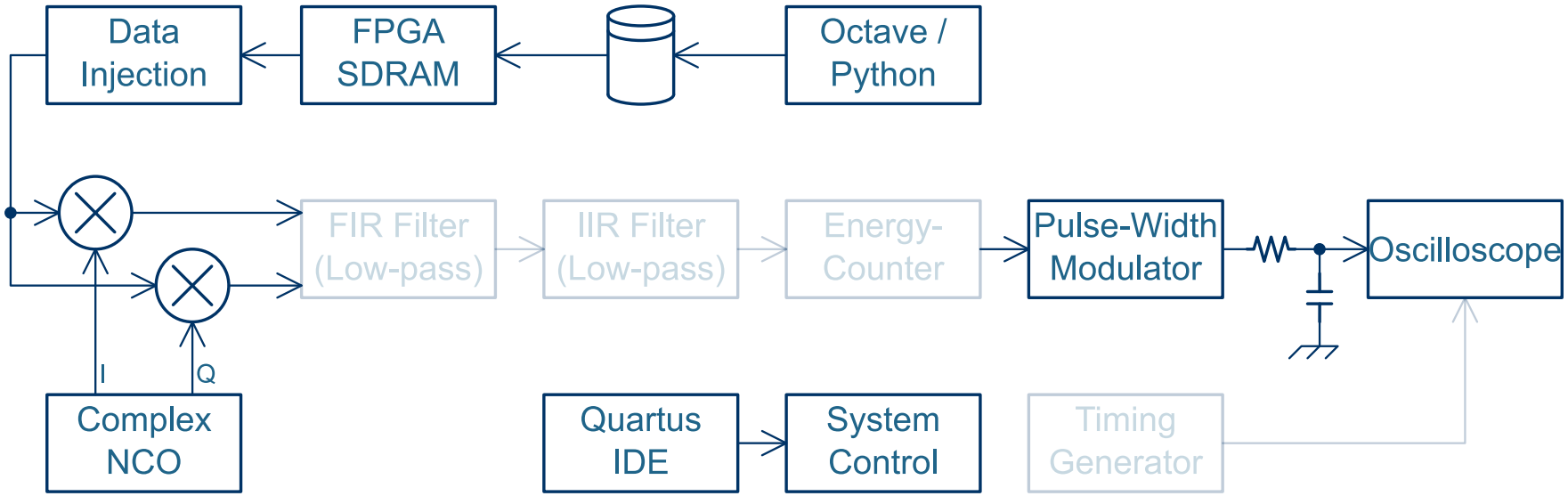
Practical

THE RADAR MASTERS COURSE

Practical


Data Injection
FPGA SDRAM
Octave / Python
FIR Filter (Low-pass)
IIR Filter (Low-pass)
Energy-Counter
Pulse-Width Modulator
Oscilloscope
I
Q
Complex NCO
Quartus IDE
System Control
Timing Generator

Practical


Data Injection
FPGA SDRAM
Octave / Python
FIR Filter (Low-pass)
IIR Filter (Low-pass)
Energy-Counter
Pulse-Width Modulator
Oscilloscope
I
Q
Complex NCO
Quartus IDE
System Control
Timing Generator

Agenda

1. Modify the injection module to inject 8-bit data at 100 MSps
2. Implement a complex NCO (100 MSps, 12-bit phase, 9-bit amplitude) and verify through simulation
3. Implement a complex mixer (real input; complex output) and verify through simulation
4. Ensure that the design meets timing requirements
5. Use JTAG to control the NCO frequency
   (Your choice of Source-and-Probes / Virtual JTAG / etc.)
6. Use injection data and mixer settings that result in low frequency components (<20 kHz) and display using the Oscilloscope

Agenda

1. Modify the injection module to inject 8-bit data at 100 MSps
2. Implement a complex NCO (100 MSps, 12-bit phase, 9-bit amplitude) and verify through simulation
3. Implement a complex mixer (real input; complex output) and verify through simulation
4. Ensure that the design meets timing requirements
5. Use JTAG to control the NCO frequency
   (Your choice of Source-and-Probes / Virtual JTAG / etc.)
6. Use injection data and mixer settings that result in low frequency components (<20 kHz) and display using the Oscilloscope

Agenda

1. Modify the injection module to inject 8-bit data at 100 MSps
2. Implement a complex NCO (100 MSps, 12-bit phase, 9-bit amplitude) and verify through simulation
3. Implement a complex mixer (real input; complex output) and verify through simulation
4. Ensure that the design meets timing requirements
5. Use JTAG to control the NCO frequency
   (Your choice of Source-and-Probes / Virtual JTAG / etc.)
6. Use injection data and mixer settings that result in low frequency components (<20 kHz) and display using the Oscilloscope

Agenda

1. Modify the injection module to inject 8-bit data at 100 MSps
2. Implement a complex NCO (100 MSps, 12-bit phase, 9-bit amplitude) and verify through simulation
3. Implement a complex mixer (real input; complex output) and verify through simulation
4. Ensure that the design meets timing requirements
5. Use JTAG to control the NCO frequency
   (Your choice of Source-and-Probes / Virtual JTAG / etc.)
6. Use injection data and mixer settings that result in low frequency components (<20 kHz) and display using the Oscilloscope

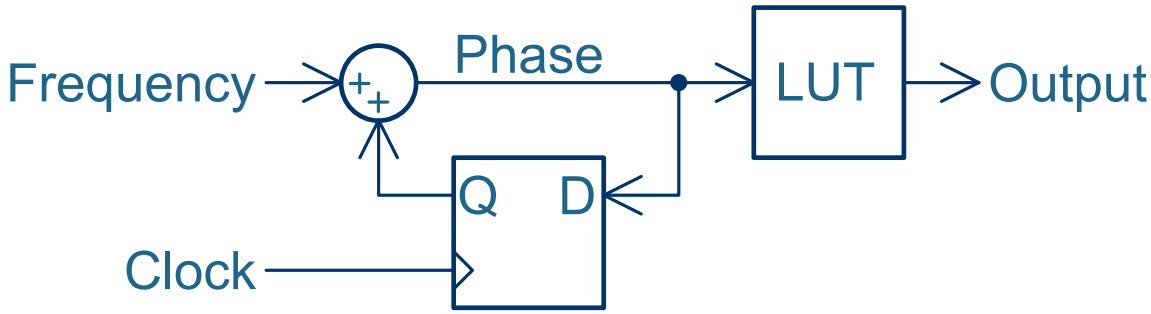
Agenda

1. Modify the injection module to inject 8-bit data at 100 MSps
2. Implement a complex NCO (100 MSps, 12-bit phase, 9-bit amplitude) and verify through simulation
3. Implement a complex mixer (real input; complex output) and verify through simulation
4. Ensure that the design meets timing requirements
5. Use JTAG to control the NCO frequency
   (Your choice of Source-and-Probes / Virtual JTAG / etc.)
6. Use injection data and mixer settings that result in low frequency components (<20 kHz) and display using the Oscilloscope

Agenda

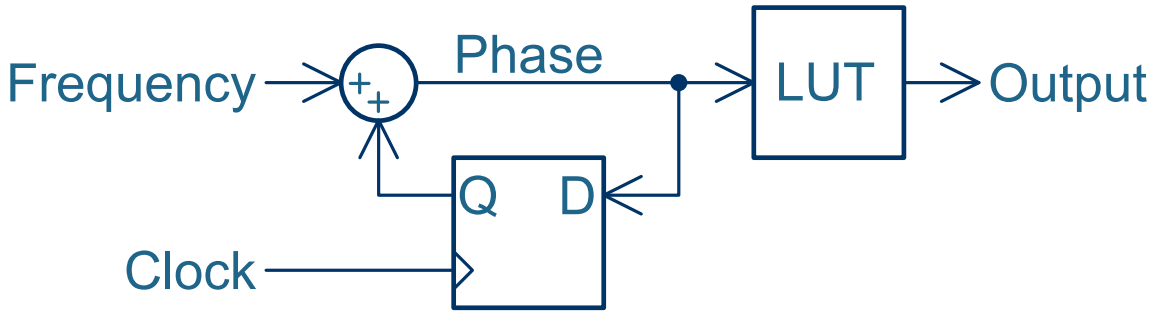
1. Modify the injection module to inject 8-bit data at 100 MSps
2. Implement a complex NCO (100 MSps, 12-bit phase, 9-bit amplitude) and verify through simulation
3. Implement a complex mixer (real input; complex output) and verify through simulation
4. Ensure that the design meets timing requirements
5. Use JTAG to control the NCO frequency
   (Your choice of Source-and-Probes / Virtual JTAG / etc.)
6. Use injection data and mixer settings that result in low frequency components ($<$20 kHz) and display using the Oscilloscope

Numerically-controlled Oscillator



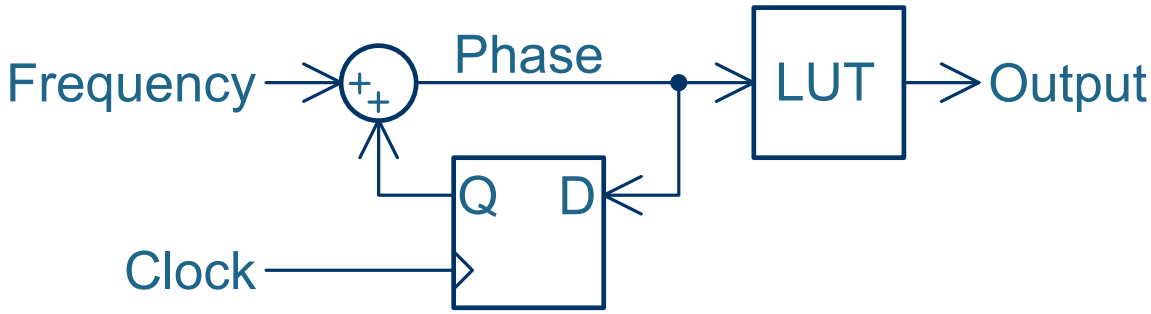
- Integrate frequency to obtain phase
- Use on-chip memory blocks as a look-up table to convert phase to the intended waveform
- Use dual-port ROM to obtain sine and cosine from the same LUT

Numerically-controlled Oscillator



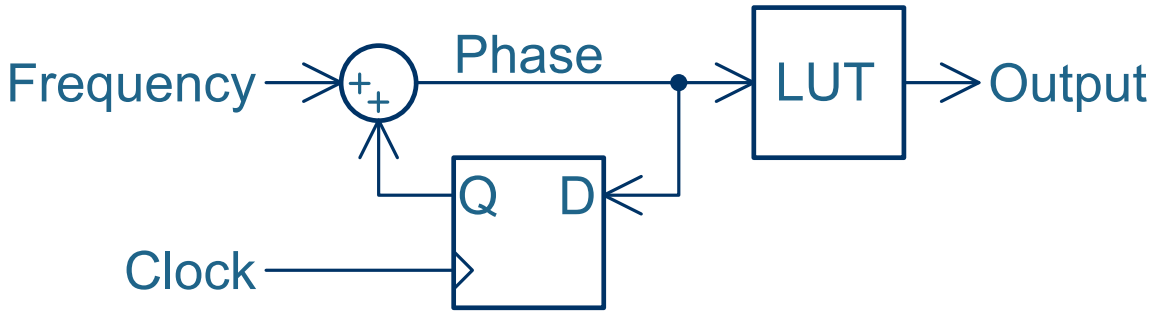
- Integrate frequency to obtain phase
- Use on-chip memory blocks as a look-up table to convert phase to the intended waveform
- Use dual-port ROM to obtain sine and cosine from the same LUT

Numerically-controlled Oscillator



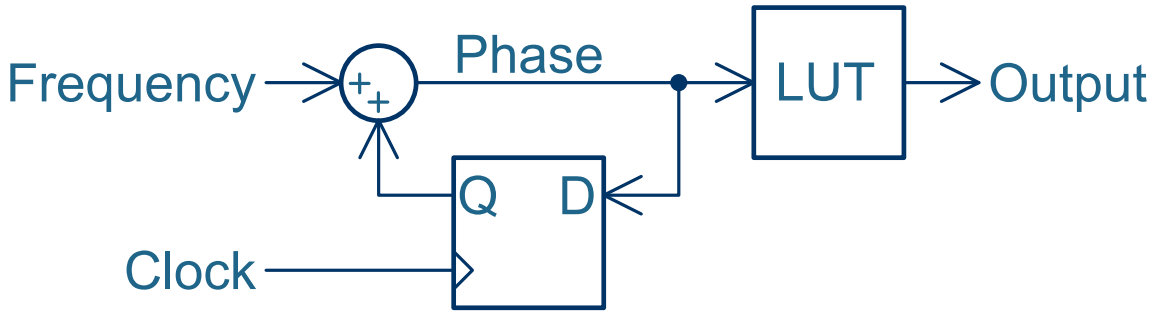
- Integrate frequency to obtain phase
- Use on-chip memory blocks as a look-up table to convert phase to the intended waveform
- Use dual-port ROM to obtain sine and cosine from the same LUT

Numerically-controlled Oscillator



- Integrate frequency to obtain phase
- Use on-chip memory blocks as a look-up table to convert phase to the intended waveform
- Use dual-port ROM to obtain sine and cosine from the same LUT

Numerically-controlled Oscillator



$$f_{out} = f_s \cdot \frac{f_{in}}{2^N}, \quad N = 32, \quad f_s = 100 \text{ MHz}$$

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 Prof Daniel O'Hagan

Tutored by Stephen Paine and Randy Cheng

Day 3 – 19 July 2017