Intro to HW Design & Externs for P4 NetFPGA

CS344 – Lecture 5
Announcements

• Wednesday:
  ○ Static router interoperability pretesting
  ○ Group meetings with instructors (Gates 315)

• Thursday:
  ○ Design challenge project proposals
Outline

- Intro to HW design & FPGAs
- How to build your own stateful extern for P4→NetFPGA
- Design Challenge Projects
Logic Gates

### AND Gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### OR Gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### XOR Gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Software vs Hardware Design

- **Software Design**
  - Functionality as sequence of instructions for CPU
  - Language: C, C++, Python, etc.

- **Hardware Design**
  - Functionality as digital circuit
  - Language: Verilog, VHDL

- **A simple example:**
  ```
  if (a > b) {
    res = a;
  }
  else {
    res = b;
  }
  return res;
  ```
A Simple Example

Software (C)

```c
if (a > b) {
    res = a;
} else {
    res = b;
}
return res;
```

Hardware (Verilog)

```verilog
wire a, b;
reg res;
always @(*)
    if (a > b) begin
        res = a;
    end
    else begin
        res = b;
    end
```

Diagram:

```
<table>
<thead>
<tr>
<th>a</th>
<th>&gt;</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>res</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

---

cmp a b tmp
beqz tmp else
store r0 a
return

else: store r0 b
return
Verilog

• Basic data types:
  ○ `wire`
    ▪ Example: `wire [15:0] B;`
    ▪ Used for combinational (stateless) logic only
  ○ `reg`
    ▪ Example: `reg [7:0] A;`
    ▪ Can be used to hold state

• Example Usage:
  ○ `assign B = {A[7:0], A[7:0]};` // Assignment of bits
  ○ `reg [31:0] Mem [0:1023];` // 1K word memory
Combinational vs Sequential Logic

- **Combinational Logic**
  - Made of logic gates
  - No memory elements
  - Outputs settle to stable values after “short” logic delay

- **Sequential Logic**
  - Combinational circuits and memory elements
  - Used to store state
  - Output depends on inputs and current state
Adding State: Flip-Flop

reg res;
always @(posedge clk)
  if (rst)
    res <= 0;
  else
    res <= res_next;
Register the Output

```verilog
wire a, b;
reg res_next;
always @(*)
    if (a > b) begin
        res_next = a;
    end
    else begin
        res_next = b;
    end
reg res;
always @(posedge clk)
    if (rst)
        res <= 0;
    else
        res <= res_next;
```

![Combinational Logic Diagram](image1)

![Sequential Logic Diagram](image2)
Set Up Time Constraints

Assumes clock is perfectly synchronized at all flip-flops!
Set Up Time Constraints

\[ f(a,b) = (3a^2 + 7ab - 9b^3 = 0) \ ? \ 0 : 1 \]

\[ f(a,b) = (3a^2 + 7ab - 9b^3 == 0) \ ? \ 0 : 1 \]
Set Up Time Constraints

\[ f(a,b) \]

\[ D_{in} \rightarrow D_{out} \]

\[ D_{in} \rightarrow D_{out} \]

\[ clk \quad rst \]

\[ f(a,b) \]

\[ res_{next} \]

\[ clk \quad rst \]

\[ 5\text{ns} \]

\[ 6\text{ns} + 100\text{ps} = 6.1\text{ns} > 5\text{ns} \]
Finite State Machine (FSM)
FSMs Are Everywhere…

FSM Diagram:
- **SRAM**
- **SRAM Control**
- **Parsing**
- **Match/Action Processing**
- **Interface Management**
- **FPGA**
- **SFP**
What is an FPGA?

CLBs

DCM

I/O

BRAM
What is an FPGA?
FPGA Design Flow

- RTL (Verilog) Design
- RTL Behavioral Simulation
- Synthesis
  - Decompose design into well defined logic blocks that are available on FPGA
- Place and Route
  - Figure out exactly which logic blocks to use and how to route between them
- Bitstream Generation
- HW Testing
P4→NetFPGA Extern Function library

- HDL modules invoked from within P4 programs
- Stateful Atoms [1]

<table>
<thead>
<tr>
<th>Atom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>Read or write state</td>
</tr>
<tr>
<td>RAW</td>
<td>Read, add to, or overwrite state</td>
</tr>
<tr>
<td>PRAW</td>
<td>Predicated version of RAW</td>
</tr>
<tr>
<td>ifElseRAW</td>
<td>Two RAWs, one each for when predicate is true or false</td>
</tr>
<tr>
<td>Sub</td>
<td>IfElseRAW with stateful subtraction capability</td>
</tr>
</tbody>
</table>

- Stateless Externs

<table>
<thead>
<tr>
<th>Atom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Checksum</td>
<td>Given an IP header, compute IP checksum</td>
</tr>
<tr>
<td>LRC</td>
<td>Longitudinal redundancy check, simple hash function</td>
</tr>
<tr>
<td>timestamp</td>
<td>Generate timestamp (granularity of 5 ns)</td>
</tr>
</tbody>
</table>

- Add your own!

Build a new extern: \texttt{reg\_srw}

\begin{itemize}
  \item \textbf{Specifications:}
    \begin{itemize}
      \item Single state variable
      \item Can either read or write state
      \item Produces result in 1 clock cycle
      \item Not accessible by control-plane
    \end{itemize}
  \item \textbf{P4 API:}
    \begin{verbatim}
    #define REG_READ 0
    #define REG_WRITE 1
    @Xilinx_MaxLatency(1)
    @Xilinx_ControlWidth(0)
    extern void <reg\_name>_reg\_srw\langle D\rangle(in D newVal,
    in bit opCode,
    out D result);
    \end{verbatim}
\end{itemize}
reg_srw Next State Logic

opcodes:
• READ = 0
• WRITE = 1

```vhdl
wire valid, opcode;
wire [7:0] newVal;
reg [7:0] state_next;
always @(*)
    if (valid & opcode)
        state_next = newVal;
    else
        state_next = state;
```
Finite State Machine

reg_srw Finite State Machine

opcodes:
- READ = 0
- WRITE = 1

```
wire valid, opcode;
wire [7:0] newVal;
reg [7:0] state_next;
always @(*)
    if (valid & opcode)
        state_next = newVal;
    else
        state_next = state;

reg [7:0] state;
always @(posedge clk)
    if (rst)
        state <= 0;
    else
        state <= state_next;
```
Register the Outputs

opcodes:
- READ = 0
- WRITE = 1

valid
opcode
newVal

&

1
0

state_next

D_in  D_out

valid_out

D_in  D_out

clk  rst

state
Register the Outputs

opcodes:
- READ = 0
- WRITE = 1

```vhdl
reg [7:0] state; reg valid_out;
always @(posedge clk)
  if (rst) begin
    state <= 0;
    valid_out <= 0;
  end else begin
    state <= state_next;
    valid_out <= valid;
  end
```

```vhdl
always @(
  posedge clk
) begin
  if (rst) begin
    state <= 0;
    valid_out <= 0;
  end else begin
    state <= state_next;
    valid_out <= valid;
  end
end
```
More Advanced State Machines

```verilog
localparam STATE_1 = 0;
localparam STATE_2 = 1;
reg state, state_next;
reg [1:0] output_1;

always @(*) begin
    // defaults
    state_next = state;
    output_1 = 1;
    case (state)
        STATE_1: begin
            output_1 = 1;
            state_next = STATE_2;
        end
        STATE_2: begin
            output_1 = 2;
            state_next = STATE_1;
        end
    endcase
end

always @(posedge clk) begin
    if (rst)
        state <= STATE_1;
    else
        state <= state_next;
end
```

![State Machine Diagram]

**STATE_1**
- Output 1 = 1

**STATE_2**
- Output 1 = 2
SDNet Extern API

P4 API:
```c
#define REG_READ 0
#define REG_WRITE 1
@Xilinx_MaxLatency(1)
@Xilinx_ControlWidth(0)
extern void <reg_name>_reg_srw<D>(in D newVal,
    in bit opCode,
    out D result);
```

HDL Interface:
```vhdl
module my_reg_srw (  
    input clk,
    input rst,
    input input_VALID,
    input [REG_WIDTH+OP_WIDTH:0] input_DATA,
    output [REG_WIDTH-1:0] output_VALID,
    output [REG_WIDTH-1:0] output_DATA
);  
```
```vhdl
wire valid, stateful_valid, opcode;
wire [REG_WIDTH-1:0] newVal;
assign valid = input_VALID;
assign {stateful_valid, newVal, opcode} = input_DATA;
```

27
Stateful_Valid Signal

bit<8> result;
if (p.hdr.invoke == 1) {
    myReg_reg_srw(0, REG_WRITE, result);
} else {
    result = 32;
}

• Two cases:
  ◦ p.hdr.invoke == 1 \(\Rightarrow\) stateful_valid signal will be set
  ◦ p.hdr.invoke != 1 \(\Rightarrow\) stateful_valid signal will \textit{not} be set

• \texttt{input_VALID} signal \textit{will} be asserted in both cases
State Machine

opcodes:
- READ = 0
- WRITE = 1

```
always @(*)
if (valid & stateful_valid & opcode)
  state_next = newVal;
else
  state_next = state;
```
SDNet Extern API

HDL Interface:

module @MODULE_NAME#@#(
    parameter REG_WIDTH = @REG_WIDTH@,
    parameter OP_WIDTH = 1)
(
    input                  clk_lookup,
    input                  rst,
    input                  tuple_in_@EXTERN_NAME@_input_VALID,
    input  [REG_WIDTH+OP_WIDTH:0]  tuple_in_@EXTERN_NAME@_input_DATA,
    output                 tuple_out_@EXTERN_NAME@_output_VALID,
    output  [REG_WIDTH-1:0]  tuple_out_@EXTERN_NAME@_output_DATA
);

wire valid, stateful_valid, opcode;
wire [REG_WIDTH-1:0] newVal;
assign valid = tuple_in_@EXTERN_NAME@_input_VALID;
assign {stateful_valid, newVal, opcode} = tuple_in_@EXTERN_NAME@_input_DATA;
Adding extern support to P4→NetFPGA

- **Update file:** `$SUME_SDNET/bin/extern_data.py`

```python
extern_data = {
...
 "reg_srw" : {"hdl_template_file": "externs/path/to/EXTERN_reg_srw_template.v", 
              "replacements": {"@EXTERN_NAME" : "extern_name", 
                              "@MODULE_NAME" : "module_name", 
                              "@REG_WIDTH@" : "input_width(newVal)"}},
...
}
```

- **Commands used:**
  - `extern_name` – full name of extern function, determined by SDNet
  - `module_name` – name of the top level extern module, determined by SDNet
  - `input_width(field)` – width in bits of an input field, determined by P4 programmer
Putting it all together: EXTERN_reg_srw_template.v

module @MODULE_NAME#@#(
  parameter REG_WIDTH = @REG_WIDTH@,
  parameter OP_WIDTH = 1)
(
  input                clk_lookup,
  input                rst,
  input                tuple_in_@EXTERN_NAME@_input_VALID,
  input [REG_WIDTH+OP_WIDTH:0] tuple_in_@EXTERN_NAME@_input_DATA,
  output               tuple_out_@EXTERN_NAME@_output_VALID,
  output [REG_WIDTH-1:0] tuple_out_@EXTERN_NAME@_output_DATA
);

// wire and reg declarations
wire valid, stateful_valid, opcode;
wire [REG_WIDTH-1:0] newVal;
reg [REG_WIDTH-1:0] state, state_next;
reg valid_out;

// decoding the inputs
assign valid = tuple_in_@EXTERN_NAME@_input_VALID;
assign {stateful_valid, newVal, opcode} = tuple_in_@EXTERN_NAME@_input_DATA;

// next state logic
always @(*)
  if (valid & stateful_valid & opcode)
    state_next = newVal;
  else
    state_next = state;

// state update / output logic
always @(posedge clk_lookup)
  if (rst) begin
    state <= 0;
    valid_out <= 0;
  end
  else begin
    state <= state_next;
    valid_out <= valid;
  end

// wire up the outputs
assign tuple_out_@EXTERN_NAME@_output_VALID = valid_out;
assign tuple_out_@EXTERN_NAME@_output_DATA = state;
endmodule
Using our new extern: srw_test.p4

// extern declaration
#define REG_READ 0
#define REG_WRITE 1
@Xilinx_MaxLatency(1)
@Xilinx_ControlWidth(0)
extern void myReg_reg_srw(in bit<8> newVal,
in bit opCode,
out bit<8> result);

// match-action pipeline
control TopPipe(...) {

apply {
    . . .
    bit<16> newVal;
    bit opcode;
    if (p.ethernet.etherType > 10) {
        newVal = p.ethernet.etherType;
        opcode = REG_WRITE;
    } else {
        newVal = 0; // unused
        opcode = REG_READ;
    }
    myReg_reg_srw(newVal, opcode, p.ethernet.etherType);
    . . .
}

• **extern-tutorial branch:**
  ○ srw_test project
What we didn’t cover

- Externs with control-plane interface
- BRAM based stateful extern
- Extern C++ implementations
**AXI4 Stream Interface**

- Standardized interface for streaming packets between modules

![Diagram](AXI4_Stream_Interface.png)
AXI4-Stream Handshake

- \text{TVALID} \& \text{TREADY} \rightarrow \text{data is being transferred}
- \text{TVALID} \& \text{TREADY} \& \text{TLAST} \rightarrow \text{the final word of the pkt is being transferred}
always @(*) begin
    // default values
    src_mac_w = 0;
dst_mac_w = 0;
eth_done_w = 0;
src_port_w = 0;
state_next = state;

    case(state)
    /* read the input source header and get the first word */
    READ_MAC_ADDRESSES: begin
        if(valid) begin
            src_port_w = tuser[SRC_PORT_POS+7:SRC_PORT_POS];
dst_mac_w = tdata[47:0];
src_mac_w = tdata[95:48];
eth_done_w = 1;
state_next = WAIT_EOP;
        end
        end case: READ_WORD_1
    WAIT_EOP: begin
        if(valid && tlast)
            state_next = READ_MAC_ADDRESSES;
        end
    endcase // case(state)
end // always @ (*)

always @(posedge clk) begin
    if(reset) begin
        src_port <= {NUM_QUEUES{1'b0}};
dst_mac <= 48'b0;
src_mac <= 48'b0;
eth_done <= 0;
state <= READ_MAC_ADDRESSES;
    end
    else begin
        src_port <= src_port_w;
dst_mac <= dst_mac_w;
src_mac <= src_mac_w;
eth_done <= eth_done_w;
state <= state_next;
    end
end
always @(*) begin
    src_mac_w = 0;
    dst_mac_w = 0;
    eth_done_w = 0;
    src_port_w = 0;
    state_next = state;

    case(state)
        /* read the input source header and get the first word */
        READ_MAC_ADDRESSES: begin
            if(valid) begin
                src_port_w = tuser[SRC_PORT_POS+7:SRC_PORT_POS];
                dst_mac_w = tdata[47:0];
                src_mac_w = tdata[95:48];
                eth_done_w = 1;
                state_next = WAIT_EOP;
            end
        end // case: READ_WORD_1
        WAIT_EOP: begin
            if(valid && tlast) state_next = READ_MAC_ADDRESSES;
        end
    endcase // case(state)
end // always @ (*)

always @(posedge clk) begin
    if(reset) begin
        src_port <= {NUM_QUEUES{1'b0}};
        dst_mac <= 48'b0;
        src_mac <= 48'b0;
        eth_done <= 0;
        state <= READ_MAC_ADDRESSES;
    end
    else begin
        src_port <= src_port_w;
        dst_mac <= dst_mac_w;
        src_mac <= src_mac_w;
        eth_done <= eth_done_w;
        state <= state_next;
    end // else: !if(reset)
end // always @(posedge clk)

header Ethernet_h {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}

parser TopParser(packet_in b, out Parsed_packet p) {
    state start {
        b.extract(p.ethernet);
        transition accept;
    }
}
Design Challenge Projects

- Mini research projects
- Final presentation and demonstration (Wed June 12th 3:30-6:30pm)
- Last year’s projects:
  - NetCache on NetFPGA
  - In-Network compression
  - Network accelerated sorting
  - P4 performance measurement system
  - INT & network cookies
  - Count Min Sketch for packet scheduling

- Proposal due Thursday (4/25)
  - Project title
  - Problem statement and motivation
  - Initial thoughts for system design and final demo

- Project suggestions: Event-Driven P4 programs
Modern Programmable Data-Planes are Limited

Two Key Limitations:

1. Deriving congestion signals for AQM, congestion control, load balancing, and telemetry applications

   - Queue length
   - Queue service rate
   - Queueing delay

   - Packet loss volume
   - Rate of change of queue size
   - Timestamp of buffer overflow/underflow events
   - Per-active-flow buffer occupancy
   - Etc…

2. Performing periodic tasks
   - Periodic state update: Count-Min-Sketch, NumFabric, sPERC, AFD, etc.
   - Periodic packet generation: HULA, BFD, etc.
Modern PISA Switch

Constraints:

- No shared state across pipeline stages
- Limited pipeline stages
- Limited per-packet computation
- Only supports 3 event types
- Limited pipeline bandwidth
- Limited memory
- Limited control-plane bandwidth
# Data-Plane Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingress Packet</td>
<td>Packet arrival</td>
</tr>
<tr>
<td>Egress Packet</td>
<td>Packet departure</td>
</tr>
<tr>
<td>Recirculated packet</td>
<td>Packet sent back to ingress</td>
</tr>
<tr>
<td>Buffer Enqueue</td>
<td>Packet enqueued in buffer</td>
</tr>
<tr>
<td>Buffer Dequeue</td>
<td>Packet dequeued from buffer</td>
</tr>
<tr>
<td>Buffer Overflow / Packet Drop</td>
<td>Packet dropped at buffer</td>
</tr>
<tr>
<td>Buffer Underflow</td>
<td>Buffer becomes empty</td>
</tr>
<tr>
<td>Timer Event</td>
<td>Configurable timer expires</td>
</tr>
<tr>
<td>Control-plane triggered</td>
<td>Control-plane triggers processing logic in data-plane</td>
</tr>
<tr>
<td>Link Status Change</td>
<td>Link goes down / comes up</td>
</tr>
</tbody>
</table>

## Packet & Metadata Events

## Metadata Events
Example Event-Driven Architecture

```c
struct std_timer_t {
    bool trigger;
    bit<32> period;
}

struct std_enq_t {
    bool trigger;
}

struct enq_meta_t {
    bit<8> flowID;
    bit<32> pkt_len;
}

struct std_deq_t {
    bool trigger;
}

struct deq_meta_t {
    bit<8> flowID;
    bit<32> pkt_len;
}
```

Executes when one or more events fire
SUME Event Switch Architecture

Packet Generator

Timer Module

Arbiter

Event Merger

SDNet Pipeline

Output Queues

 nf0
 nf1
 nf2
 nf3
 dma

 Generate packet

 Timer period

 Timer event

 Packet event

 Link status change event

 Enq event

 Deq event

 Drop event
struct sume_metadata_t {
    // request a packet to be generated
    bit<1> gen_packet;
    // which events have fired
    bit<1> link_trigger;
    bit<1> timer_trigger;
    bit<1> drop_trigger;
    bit<1> deq_trigger;
    bit<1> enq_trigger;
    bit<1> pkt_trigger;
    // Current link status (one-hot)
    bit<4> link_status;
    // last timer event (20ns resolution)
    bit<48> timer_now;
    // timer event period (20ns resolution)
    bit<32> timer_period;
    // ports and metadata for events
    bit<8> drop_port;
    bit<8> deq_port;
    bit<8> enq_port;
    bit<32> drop_data;
    bit<32> deq_data;
    bit<32> enq_data;
    // standard metadata fields
    bit<8> dst_port;
    bit<8> src_port;
    bit<16> pkt_len;
};
Project Area Suggestions

- Microburst Detection
- Fast Reroute
- Load Balancing
- Network Monitoring
- Active Queue Management
Extra Slides
Example Event-Driven Architecture

- Timer Module
- Traffic Manager
- Ingress Packet Event
- Timer Event
Example Event-Driven Architecture

- Timer Module
- Traffic Manager
- Ingress Packet Event
- Timer Event
Example Event-Driven Architecture

Diagram showing various components such as Timer Module, Traffic Manager, Ingress Packet Event, Enqueue Event, and Dequeue Event.