Design Studies – Part 2

- Data queue
- Packet FIFO
- Priority queue

Jon Turner
Data Queue

- A queue is a data structure that stores a set of values so they can be retrieved in the same order they were stored
  - operations are enqueue and dequeue
  - separate dataIn, dataOut ports allow simultaneous enqueue & dequeue (so, one "client" can write, while another reads)
  - status signals: empty and full
  - implement using an array of data registers, plus a pair of of pointers and a counter
 VHDL Specification

define entity queue is port ( 
    clk, reset: in std_logic;
    enq, deq : in std_logic;
    dataIn : in word;
    dataOut : out word;
    empty, full : out std_logic);
end queue;
architecture arch of queue is
constant qSize: integer := 16;
type qStoreTyp is array(0 to qSize-1) of word;
signal qStore: qStoreTyp;
subtype pointer is std_logic_vector(3 downto 0);
subtype counter is std_logic_vector(4 downto 0);
signal readPtr, writePtr: pointer;
signal count: counter;
begin
    process (clk) begin

if rising_edge(clk) then
  if reset = '1' then
    readPntr <= (others => '0'); writePntr <= (others => '0');
    count <= (others => '0');
  else
    if enq = '1' and deq = '1' then
      if count = 0 then
        qStore(int(writePntr)) <= dataIn;
        writePntr <= writePntr + 1; count <= count + 1;
      else
        qStore(int(writePntr)) <= dataIn;
        readPntr <= readPntr + 1; writePntr <= writePntr + 1;
      end if;
    elsif enq = '1' and count /= qSize then
      qStore(int(writePntr)) <= dataIn;
      writePntr <= writePntr + 1; count <= count + 1;
    elsif deq = '1' and count /= 0 then
      readPntr <= readPntr + 1; count <= count - 1;
    end if;
  end if;
end process;

dataOut <= qStore(int(readPntr));
empty <= '1' when count = 0 else '0';
full <= '1' when count = qSize else '0';
end arch;
# Simulation Results

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>enq</th>
<th>deq</th>
<th>dataIn</th>
<th>readPtrn</th>
<th>writePtrn</th>
<th>count</th>
<th>dataOut</th>
<th>empty</th>
<th>full</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>02</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>02</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>02</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Incoming values**: The values that are currently being input into the system.
- **Outgoing values**: The values that are being output from the system.
- **Queue full condition**: The condition where the queue is full, as indicated by the stored values.
- **Stored values**: The values that are currently stored in the queue.

*Note: The diagram illustrates the simulation results of a system with incoming and outgoing values, as well as a queue that can store data.*
Exercises

1. Modify the queue VHDL to replace the counter with a single flip flop that is high whenever the queue is full.

2. Modify the queue VHDL to eliminate the counter by allowing a maximum of 15 values to be stored in the queue, instead of 16.

3. Draw a block diagram of the data queue. Assume that the \texttt{qStore} is implemented by a dual port memory. This is a memory with a data input, a data output plus a \texttt{writeAddress} input and a \texttt{readAddress} input plus a \texttt{writeEnable} and a \texttt{readEnable}. Show all registers in your diagram. You can show the control logic as a block labelled \texttt{controller} with signals going to the other components. The input and output sides of the memory can operate independently and concurrently.

4. Write a VHDL module that implements a \textit{double-ended queue} (sometimes called a \textit{deque}). This module implements a \texttt{writeFront} operation and a \texttt{writeBack} operation. It should be equipped with two data outputs, one of which outputs the value at the front of the queue, and another that outputs the value at the back of the queue. It can be implemented in a way that is very similar to the data queue.
Packet FIFO

- Store variable length packets in FIFO order
  - packets received and sent word-by-word
- Left interface signals include
  - start-of-packet – high during first clock tick of packet
  - ack output high if enough room to store arriving packet
  - error output high if packet length out of range (3 to 6)
- Right interface signals include
  - ok2send – when high, circuit sends packet if it has one
Interface Timing

**left interface**

- **clk**
- **sopIn**
- **dIn**: x04, x01, x02, x03, --
- **ack**

**right interface**

- **clk**
- **ok2send**
- **sopOut**
- **dOut**: x04, x01, x02, x03, --
Block Diagram

- `pStore 16x4`
- `inCnt`, `wPntr`, `rPntr`, `outCnt`
- `ctlLeft idle busy errState`
- `ctlRight idle busy`
- `dIn`, `dOut`
- `nPkt`, `nWords`
Controller State Machines

**ctlLeft**
- **idle**
  - **condition:** inCnt=1
  - **action:**
    - increment nWords
    - increment wPtr
    - write to pStore
- **busy**
  - **condition:** inCnt>1
  - **action:**
    - decrement inCnt
    - increment nWords
    - increment wPtr
    - write to pStore

**errState**
- **condition:** sopIn=1 and (din=3 or din=6)
  - **action:** none

**ctlRight**
- **idle**
  - **condition:** outCnt=1
  - **action:**
    - decrement nWords
    - increment rPtr
    - decrement nPkts
- **busy**
  - **condition:** outCnt>1
  - **action:**
    - decrement outCnt
    - decrement nWords
    - increment rPtr

**Condition:**
- eq: cond=1 and cond=6 and nWords+nPkt>15
  - **action:**
    - load inCnt=dIn=1
    - increment nWords
    - increment wPtr
    - write to pStore
    - assert ack

**Condition:**
- eq: cond=1 and cond=6 and nWords+nPkt>15
  - **action:**
    - load outCnt=dOut=1
    - decrement nWords
    - increment rPtr
    - decrement nPkts
VHDL

entity pFIFO is port(
    clk, reset: in std_logic;
    -- left interface
dIn: in word;
    sopIn: in std_logic;
    ack, errSig: out std_logic;
    -- right interface
doOut: out word;
    sopOut: out std_logic;
    ok2send: in std_logic);
end pFIFO;
architecture arch1 of pFIFO is
constant psSiz : integer := 16;
subtype register is std_logic_vector(4 downto 0);
-- number of words left to arrive/leave
signal inCnt, outCnt: word;
-- pointer to next word to write/read
signal wPntr, rPntr: register;
-- number of words/packets stored
signal nWords, nPkts: register;
-- Packet store
type pStoreType is array(0 to psSiz-1) of word;
signal pStore: pStoreType;
-- state machines controlling input and output
type stateType is (idle, busy, errState);
signal leftCtl, rightCtl: stateType;

-- auxiliary signals
signal inRange, validArrival, enoughRoom: std_logic;
signal nWordsPlus, nWordsMinus: std_logic;
signal nPktsPlus, nPktsMinus: std_logic;
begin
  inRange <= '1' when dIn >= x"3" and dIn <= x"6" else '0';
  enoughRoom <= '1' when ('0' & dIn) + nWords <= psSiz
                   else '0';
  validArrival <= '1' when leftCtl = idle and sopIn = '1' and
                   inRange = '1' and enoughRoom = '1'
                   else '0';
process (clk) begin -- process for left controller
    if rising_edge(clk) then
        ack <= '0';
        if reset = '1' then
            leftCtl <= idle;
            inCnt <= (others => '0');
            wpntr <= (others => '0');
        else
            if leftCtl = idle then
                if validArrival = '1' then
                    pStore(int(wpntr)) <= din;
                    inCnt <= inCnt - 1; wpntr <= wpntr + 1;
                    ack <= '1'; leftCtl <= busy;
                elsif sopIn = '1' and inRange = '0' then
                    leftCtl <= errState;
                end if;
            elsif leftCtl = busy then
                pStore(int(wpntr)) <= din;
                inCnt <= inCnt - 1; wpntr <= wpntr + 1;
                if inCnt = 1 then leftCtl <= idle; end if;
            end if;
        end if;
    end if;
end process;
-- outputs of left controller
nPktsPlus <= '1' when leftCtl = busy and inCnt = 1 else '0';
nWordsPlus <= '1' when leftCtl = busy or validArrival = '1' else '0';
errSig <= '1' when leftCtl = errState else '0';
-- process for updating nWords and nPkts
process(clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      nWords <= (others => '0'); nPkts <= (others => '0');
    else
      if nWordsPlus > nWordsMinus then
        nWords <= nWords + 1;
      elsif nWordsPlus < nWordsMinus then
        nWords <= nWords - 1;
      end if;
      if nPktsPlus > nPktsMinus then
        nPkts <= nPkts + 1;
      elsif nPktsPlus < nPktsMinus then
        nPkts <= nPkts - 1;
      end if;
    end if;
  end if;
end process;
process (clk) begin  -- process for right controller
    if rising_edge(clk) then
        sopOut <= '0';
        if reset = '1' then
            rightCtl <= idle;
            outCnt <= (others => '0'); rPtrn <= (others => '0');
        else
            if rightCtl = idle then
                if ok2send = '1' and nPkts > 0 then
                    outCnt <= pStore(int(rPtrn));
                    rightCtl <= busy; sopOut <= '1';
                end if;
            elsif rightCtl = busy then
                outCnt <= outCnt - 1;
                rPtrn <= rPtrn + 1;
                if outCnt = 1 then
                    rightCtl <= idle;
                end if;
            end if;
        end if;
    end if;
end process;

-- outputs of right controller
nPktsMinus <= '1' when rightCtl = busy and outCnt = 1 else '0';
nWordsMinus <= '1' when rightCtl /= idle else '0';
dOut <= pStore(int(rPtrn));
end arch1;
Exercises

1. Why is a separate process used to update the nWords and nPkts registers in the packet FIFO? Wouldn’t it be simpler if the left and right controllers simply updated these registers directly?

2. In the packet FIFO, the right-hand interface is a little different from the left-hand interface. Consider a version of the packet FIFO where the right-hand interface uses an acknowledgment from the downstream client, rather than an ok2send signal. In this version, the FIFO attempts to send a packet whenever it has one to send and the downstream circuit sends an acknowledgement, when it is able to accept the packet. Modify the VHDL for the right-hand controller to implement such an interface.

3. Consider a version of the packet FIFO that uses a single controller, rather than two separate controllers. Draw a state transition diagram for such a controller that includes the following four states:
   - idle – nothing to do
   - incoming – handling an incoming packet
   - outgoing – handling an outgoing packet
   - inAndOut – handling an incoming and outgoing packet at the same time

   You may assume that the arriving packets all have a length that is within the allowed range (so you can skip the error state).
Priority Queue

- Priority queue stores (key, value) pairs and always makes pair with smallest key available
  - operations – insert new pair, delete pair with smallest key
  - inputs – clk, reset, insert, delete, key, value
  - outputs – smallKey, smallValue, empty, full, busy

- Implement as two rows of cells
  - keys in bottom row are sorted, keys in columns are sorted
  - occupied cells to left, occupied top cell must have occupied cell below it
  - insert by shifting into top row then doing per-column swap
entity priQueue is port(
    clk, reset: in std logic;
    insert, delete: in std_logic;
    key, value: in word;
    smallKey, smallValue : out word;
    busy, empty, full: out std logic);
end priQueue;
architecture arch1 of priQueue is
constant rowSize: integer := 4;
type cell is record
    valid: std_logic;
    key,value: word;
    end record cell;
type rowTyp is array(0 to rowSize-1) of cell;
signal top, bot: rowTyp;
type state_type is (ready, colSwap);
signal state: state_type;
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset = '1' then
        for i in 0 to rowSize-1 loop
          top(i).valid <= '0'; bot(i).valid <= '0';
        end loop;
        state <= ready;
      elsif state = ready and insert = '1' then
        if top(rowSize-1).dp /= '1' then
          top(1 to rowSize-1) <= top(0 to rowSize-2);
          top(0) <= ('1',key,value);
          state <= colSwap;
        end if;
      elsif state = ready and delete = '1' then
        if bot(0).valid /= '0' then
          bot(0 to rowSize-2) <= bot(1 to rowSize-1);
          bot(rowSize-1).valid <= '0'; state <= colSwap;
        end if;
      end if;
    end if;
  end process;
end;

make all cells empty initially
shift top row right and insert new value
shift bottom row left
elsif state = colSwap then
  for i in 0 to rowSize-1 loop
    if top(i).valid = '1' and
      (bot(i).valid = '0' or top(i).key < bot(i).key) then
      bot(i) <= top(i); top(i) <= bot(i);
    end if;
  end loop;
  state <= ready;
end if;
end process;

smallKey <= bot(0).key;
smallValue <= bot(0).value;
empty <= not bot(0).valid;
full <= top(rowSize-1).valid;
baby <= '1' when state = colSwap else '0';
end arch1;

compare and swap columns

async assignments to outputs, but values only depend on state & registers (so, effectively synchronous)
Exercises

1. Unlike the simple data queue and the packet FIFP, the priority queue does not support simultaneous insert and delete operations. How would you modify the design to allow simultaneous insert and delete requests to be handled. Note that the circuit does not need to do both operations simultaneously, so long as it processes both before raising the ready signal to indicate that it has finished.

2. How would you modify the priority queue so that it could completely process an insert of a delete in a single clock cycle? For this part do not try to handle simultaneous concurrent inserts and deletes.