# System-on-Chip Design Supervision Problem (Set 1)

Wei Song 06/03/2016

This answer provides a summary of key points in each question. It is the understanding of the supervisor, which does not necessarily always correct and does not represent the view of lecturers. It is expected to expand the key points into detailed description when similar questions are asked in exams.

## SP 1. SoC Components and Bus Structure

**Q1: Wire up a push button to a GPIO pin and describe the action of (or sketch out the code for) a device driver that sets up the GPIO device and returns how many times it has so far been pressed. Describe firstly the polling code. Then write a few sentences about how the interrupt driven solution would be implemented and when it should be used instead. [SoC-SP1.4]**

- Understand the structure of a GPIO, see lecture note 1, GPIO.
- The button must be connected to a pull-up or pull-down resistor.
- The GPIO should be set as an input port.
- Use an edge detector to identify button action.
- For the polling code, poll frequency is around 10~100 Hz.
- Adding edge detection and a counter to the timer interrupt service function is a good idea.
- For the interrupt method, still use timer interrupt for edge detection. When a press is detected, timer triggers an interrupt. The button interrupt service function implements the counter.
- For the interrupt method, make sure interrupt enable and mask registers are set properly.
- Even better if the edge detector is implemented in hardware.

**Q2: Sketch RTL Verilog code for a 3-input non-preemptive static prioritized arbiter. Then extend it into a non-preemptive round-robin arbiter. [Optional] How can we make the arbiter to support up to N inputs, where N is a design time parameter? [Extended SoC-SP1.15]**

The followings are my SystemVerilog implementations (not checked by simulators yet).

```
function int first_one(input logic [2:0] r);
  int i;
  for(i=0; i<3; i++) if(r[i]) return i;
  return 0;
endfunction

module arbiter_static (input logic [2:0] req, output logic [2:0] gnt);
  int g;
  assign g = first_one(req);
  assign gnt = req[g] << g;
endmodule

module arbiter_roundrobin  (input logic [2:0] req, output logic [2:0] gnt, input logic clk, rst);
  int p, p_rr, g;
  logic [2:0] req_rr;
  always_ff @(posedge clk or posedge rst)
```

```
    if(rst) g <= 0; else if(|req) g <= p;

  assign req_rr = {req, req} >> g;
  assign p_rr = first_one(req_rr);
  assign p = p_rr + g >= 3 ? p_rr + g – 3 : p_rr + g;
  assign gnt = req[p] << p;
endmodule

module arbiter_staticN #(parameter N=3)(input logic [N-1:0] req, output logic [N-1:0] gnt);
  int g;
  function int first_one(input logic [N-1:0] r);
    int i;
    for(i=0; i<N; i++) if(r[i]) return i;
    return 0;
  endfunction

  assign g = first_one(req);
  assign gnt = req[g] << g;
endmodule
```

## SP 3. SoC Busses, Partition and Technology

**Q3: Consider multiple busses with bridges [SoC-SP3.2]:**

1. **In current SoCs, what is a bus and how does it compared with the 1980's concept of a motherboard bus (such as ISA or PCI bus)?**

The busses in current SoCs are unidirectional point-to-point wires, which do not use bidirectional busses and tri-states gates as the old motherboard busses do. At functional level, the SoC busses remain largely arbitrated, meaning only one master driving the bus at any time.

2. **How might the destination port for a transaction over such a bus be decided?**

Addresses are used in the same way as old busses to identify the destination.

3. **What is a bus bridge, what transactions might it support and what internal operations might it implement?**

A bus bridge connects two busses so that remote transactions can be performed. A bus bridge can be used to communicate between two busses using two different standards, protocols, bandwidth, etc. A bus bridge may also be used to build a hierarchical bus system that supports multiple bus masters active simultaneously at different levels. Also, a bus bridge can be used to do burst transactions, such as a DMA.

4. **If a SoC is designed with a number of bridged busses, what are the main aspects that determine the allocation of initiators and targets to the bus?**

Most of time the goal is to minimize inter-bus communication. So it would be better to put targets on the same bus with their common initiator. Also some time putting simultaneous bus masters on different busses can increase overall throughput.

5. **What is the difference between a network-on-chip (NoC) and multiple bridged busses?**

No fundamental differences. We can see NoC as a more regulated and segmented hierarchical bus system that every network link between routers and network interfaces is a separate bus.

6. **What form of bus protocol is needed for good performance on a SoC that uses a number of bridged busses or clock domains?**

Support variable transaction latency, handshaking and out-of-order. The underlying reason is that it is better not to assume the protocol used on the other side of a bus bridge.

7. **How is contention for destinations handled in a SoC that uses a number of bridges busses compared with a NoC?**

Every bus and NoC is different. Generally speaking, a bus is normally centrally arbitrated that the master nodes are required to have buffers to store data when they are waiting for an arbitration grant. In a NoC, arbitration is handled in a distributed way that every router allocates its own resources. So the buffering scheme is more distributed that every router and network interface has its own buffer.

**Q4: DRAM and cache [SoC-SP3.4]:**

1. **Why is DRAM not commonly integrated in the same die as a part of a SoC?**

Since DRAM is volatile memory that needs regular refresh to store data, it is built on dedicated fab lines and so not ideal to be mixed with general digital logic on a SoC.

2. **Considering accesses to DRAM, why should out-of-order read responses ideally be supported by a SoC bus or NoC?**

To achieve the optimal bandwidth, the DRAM controller normally reorders memory accesses into bunches of accesses to the same region (same row for example). As a result, a SoC or NoC is better to support to out-of-order accesses to reorder the memory accesses.

**Q5: Cell library [SoC-SP3.6]:**

1. **Give a short list of logic cells found in a standard cell library.**

Combinational logic: AND, NAND, OR, NOR, XOR, INV, MUX, BUF
Sequential logic: Latch, Flip-Flop
Tristate special (not available in all libraries): pullup, pulldown, tristate
Physical cells (not available in frontend library): power gate

2. **List five types of information that should be stored for each cell in a library.**

Supply voltage, temperature, area, I/O pins, function, driving strength, latency/energy related to input transition time/input signals/output load, leakage related to input signals, setup/hold time for registers/latches.

3. **[Optional] When estimating the latency of a gate (implemented using a standard cell) in a circuit, what information stored in the cell library is related and what is the relationship (no equation needed)?**

The latency of a gate is related to many parameters. Normally the following hold true:

Latency increases with low supply voltage, high temperature, small driving strength, long input transition time and large output load.

## SP 3. Silicon Energy, Power and Technology

**Q6: Dynamic voltage and frequency scaling [Power.3]:**

1. **Give a formula for estimating the dynamic power dissipation related to supply voltage, capacitance, and clock frequency.**

Dynamic power is linear with $aCV^2f$.

2. **What is dynamic clock gating and compare this to a technique where software writes to a control register that turns off a clock generator.**

Dynamic clock gating is normally automatically done by tools which insert clock controls to a single pipeline stage or a state machine. The regional clock tree is still running in this case. The latter one is also called coarse-grained clock gating that designers add clock controls on the roots of regional clock trees. When a component is idle, the whole clock region is turned off which includes the regional cock tree.

3. **For a fixed supply voltage, quantify the power benefits of frequency scaling.**

If not considering clock tree, frequency scaling reduces dynamic power but uses the same amount of dynamic energy. If the energy consumption of clock trees is considered, a clock tree consumes less dynamic energy with lower clock frequency. So there is a small energy saving.

4. **Give two ways that the supply voltage to a region may be varied.**

Power gating can be used to totally shut down a power region. Multiple supply voltages can be selected at run time to achieve voltage scaling.

5. **Using variable supply voltage, quantify the power benefit of frequency scaling.**

Frequency scaling allows supply voltage to be dropped when calculation load is low. The drop of supply voltage reduces the dynamic energy consumption, according to $aCV^2f$.