**FPGA**

* The design flow for programmable logic in FPGA is RTL Synthesis (controller and datapath) -> logic synthesis (netlist) -> physical design (routing), then the bitstream is produced.
* FPGA consists of configurable logic blocks, a CLB includes LUTs with 4-6 inputs, flip-flops for sequential circuits, muxes, switch matrix to connect LUTs.
* FPGA is more efficient with numerous small LUTs instead of LUTs with high number of inputs
* FPGA is made up of CLBs and switch matrices.
* When you are close to using the max amount of LUTs in the FPGA, the routing step will take a long time, because where the LUTS are placed in the FPGA are not optimized.
  + The routing step interconnects the LUTs
* FPGA is good because you can configure it for many purposes. Although you should choose FPGA over ASIC if you are producing a small amount of products.
  + ASIC is like circuit board
  + ASIC you can optimize the VLSI and make it much faster than the same circuit being implemented on a FPGA.
  + FPGA uses more power and costs more
  + FPGA usually used early in product rollout
* FPGA design flow:
  + Design entry: writing verilog code
    - Designing in text code is much more efficient and is at a higher level compared to a CAD level.
  + Simulation: test the design to make sure it is validated and verified.
    - “Testbench”
    - For ASIC going from simulation to implementation is expensive, so it's important to be sure the design is correct.
  + Implementation: for asic this is putting it on silicon
  + Physical device
* Static timing analysis (STA) tries to figure out worst case delay
  + It’s conservative (always takes the worst delay for a gate)
  + (worst gate delay sum) + (set up time) + (clock to Q delay) = Time period
  + Clock frequency = 1 / (time period)
* Hardware is faster than software
  + Software has a layer of abstraction above the hardware, so it has two things to interact with
  + \*\*\*Software program runs step by step (temporal computation)although modern machines have multiple ALUs so that they can sort of do multiple steps, while hardware can run multiple ‘steps’ at one time (spatial computation)
  + You would use software because it’s easier to develop and it is portable to many different hardwares
  + You don't always use hardware because it is not flexible (it only has one purpose), takes a long time to develop, expensive, only justified if lots of products are produced
  + Reconfigurable computing (FPGA) is the answer.
* Reconfigurable computing
  + Good for data parallelism (execute same computations on many independent data elements)
  + Small and varying bit widths (does not have to be full 64bits like in software, can use a smaller number of bits)
* Downfalls of FPGAs
  + Not as fast as real hardware, consumes more power, and not as easy to design for with software.

**Hardware-Software Communication**

* Keyboards, mice, printers interact with the computer's software.
  + Need a way for hardware to notify software asynchronously about events
  + Want flexibility; no special hw in the cpu for each device
  + Bandwidth
  + Security
* You do not have all of the i/o devices directly connected to the processor. This would take up all of the room on the processor.
  + You use i/o busses (PCIe) to send data from the i/o device to the processor. The processor will give permission to a device to send data. All other devices will then wait.
* Memory mapped i/o allows you to make control registers and i/o device memory appear to be part of the system's main memory.
  + So to access a device's memory, (after it was mapped into address space) you can just use pointers like conventional memory.
  + Interact oddly with caches
    - If the address is cached, the memory mapped address might not go to the right bus.
    - Solution: declare any data structures that are memory-mapped i/o regions as volatile (volatile int \*region)
    - This tells the system it can't keep the address in the cache.
* A processor has multiple i/o busses.
* Handling asynchronous events
  + Polling: processor checks each device to see if it has a request
    - Takes cpu time even if no requests pending
    - Tradeoff between overhead and average response time
  + Interrupts: each device has a wire that can signal the processor
    - When interupted, the processor executes interrupt handler
      * Saves what it was doing, then goes to do whatever the interrupting device wanted, when done it returns
      * Interrupt device passes interrupt number to processor
    - Only address a device when something happens
    - No overhead
  + Polling is better if you care about getting the answer ASAP, and you don't care about power. Or the processor has nothing else to do.
  + Interrupt is better if the processor has other work to do and response time is not critical
    - Performance of interrupt hardware is critical factor on processors for embedded systems
  + Interrupts in linux
    - Write interrupt handler
    - Registering the interrupt handler with the OS.
    - Interaction between the interrupt handler and user programs
      * Handler should run in very little time
      * Handler runs as part of operating system
    - Must have this heading: void int\_handler(int irq, void \*dev\_id, struct pt\_regs \*regs)
      * Inside this function there will be wake\_up\_interruptible(&queue)
      * Interrupt can't access user data

**Virtualization**

* Creating virtual hardware and an operating system on top of your host OS and hypervisor(virtualBox). The hypervisor will help the host OS allocate resources to the virtual OS.
* Virtualize because it's cheaper to emulate multiple machines instead of building a bunch of machines.
  + For example a website hosted on a server will probably not use up an entire server's resources, so you virtualize multiple machines in order to run multiple machines on one server to host many things.
* The virtualized OSs are isolated from each other.
* Have to fake privileged instructions (changing system values)
* Paravirtualization: Guest OS runs at ring 0. Higher performance, lower security. Trusting that the guest OS is going to do the right thing
* Hardware assisted virtualization: guestOS more privileges easier, as the calls are passed directly to the hypervisor

**Memory Systems**

* Main memory (DRAM) sits between the gpu(processor, other stuff) and the storage (HDD).
  + The processor communicates to the DRAM and that communicates to the storage device.
  + The processor cannot communicate directly to the storage because it is in block memory, you cannot just pull a certain amount of bytes out.
* Main memory energy/power is a key system design concern
  + DRAM uses a lot of power because you have to constantly refresh the data stored in there.
  + DRAM uses power even when it is not being used.
* SRAM: static random access memory
  + Holds data as long as there is power
  + Volatile: can't hold data if power is removed
  + Three states; hold, write, read
* DRAM: dynamic random access memory
  + Volatile: loses data when power is removed
* DRAM is smaller and less expensive per bit
* SRAM is faster
* DRAM requires more peripheral circuitry