© 2024 Burak Soner

ELEC 305

Digital System Design Lab

Fall 2024

Lecture 2:

Revisiting Fundamentals

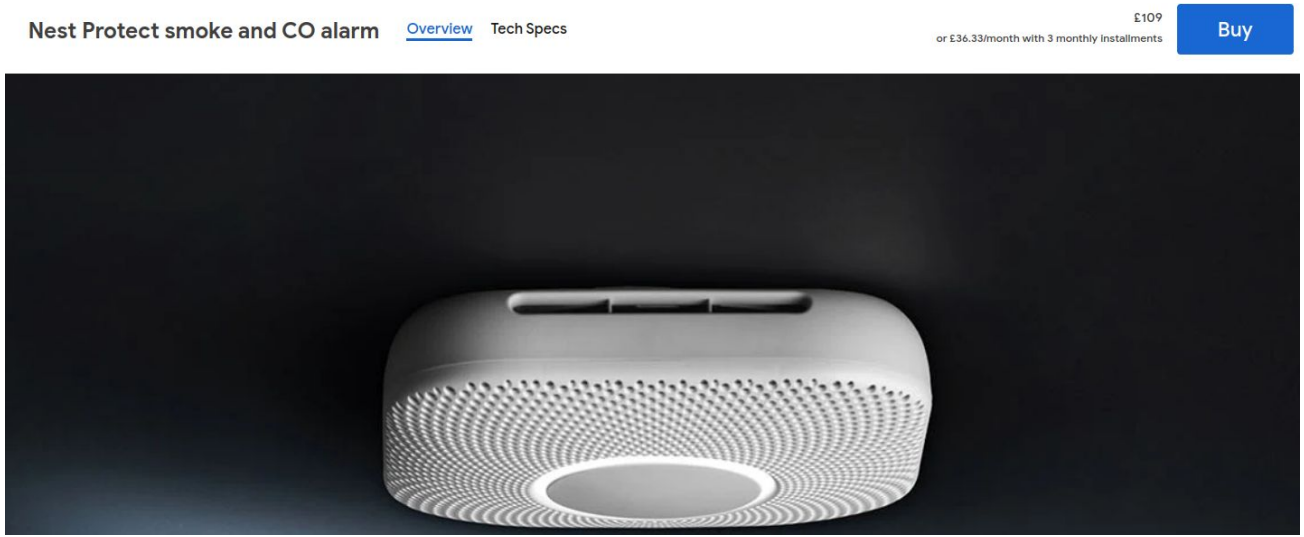
Outline

- In this lecture we will design a digital system for an example task (a fire detector) and revisit our fundamentals along the way by dissecting each design choice
- Specifically, we'll talk about…
  - Digital vs. analog
  - Using processors vs. application-specific circuits, and software design vs. hardware design
  - Combinational and sequential logic
  - Intro to HDLs and how to use them for realizing digital circuits on FPGAs
- We will treat all of these only lightly in this lecture though, detailed treatment will follow in later lectures and labs. Think of this lecture as an intro to the topics the course covers as well as review of some pre-requisite material.

© 2024 Burak Soner

Task - Fire Detector

- We basically want a system that predicts whether or not there is a fire in a room
- Let’s break down an example product: [Nest by Google]

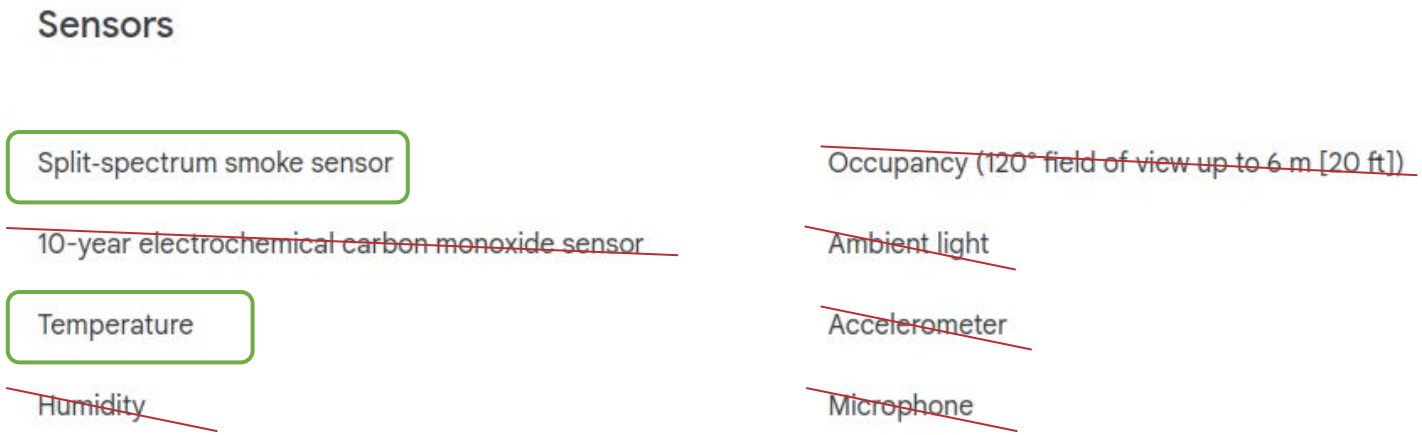


© 2024 Burak Soner

Task - Fire Detector

© 2024 Burak Soner

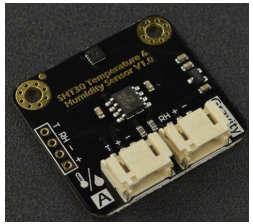
- Temperature and Smoke sensors → These are enough for us now, ignore the others
  (we're not trying to build a product, this is just a case study)



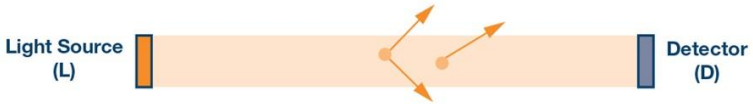
© 2024 Burak Soner

Task - Fire Detector

- We can buy the temperature sensor in TR → SHT30 gives analog voltage output proportional to the temperature
  (it also gives out the humidity level on a separate channel, but ignore that)



- The smoke sensor is a bit hard to get, assume we built one based on ADIs advice here using LEDs and photodetectors which gives analog voltage output proportional to “smoke density”



- Voltage outputs from these sensors typically have a clear bijective (1-to-1 and onto) mapping to the physical quantities they measure

© 2024 Burak Soner

Design - Algorithm

- Naive attempt at a fire detection algorithm using these 2 sensors:

  →*TA*: temperature, *SA*: smoke density

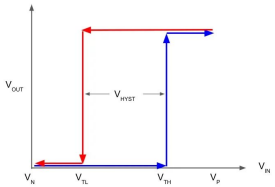
  →*X* = *TA**p1 + *SA**p2 + b1 : affine combination of *SA* and *TA*

  →*Y* = *TA*(t) - *TA*(t-100ms) : rise in temperature over 100ms

  →`if` (*X* > *THD1*) `and` (*Y* > *THD2*) `then` “fire detected, start alarm”

  →`if` (*X* < *THD3*) `and` (*Y* < *THD4*) `then` “fire extinguished / cooling, stop alarm”

  →*THD3-4* has hysteresis against *THD1-2* (i.e., *THD4*<<*THD2*, *THD3*<<*THD1*)

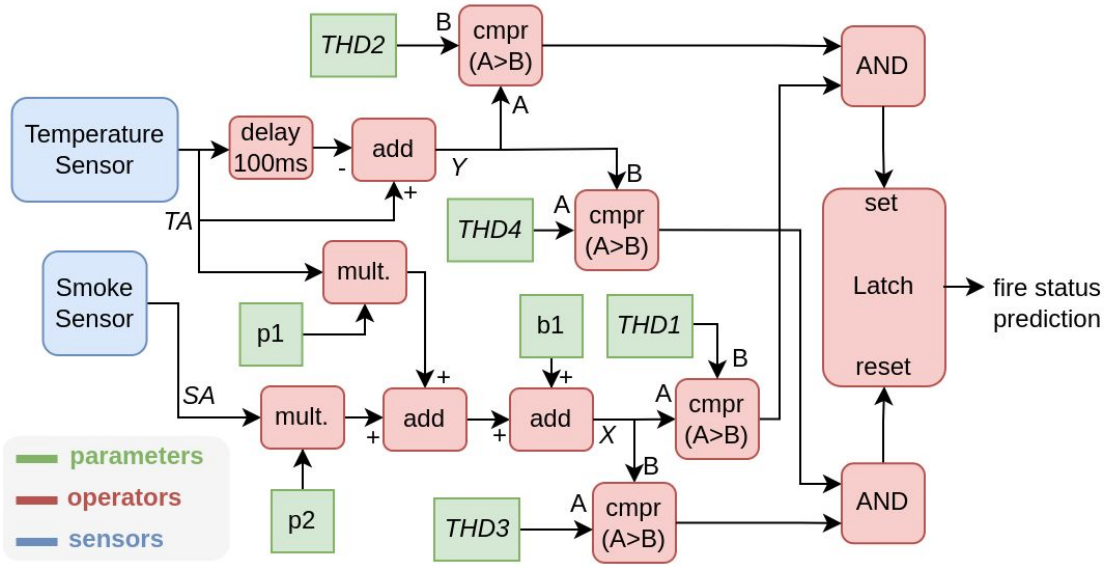


- Through some calibration we can find OK values for p1, p2, b1 and *THD1-4*

© 2024 Burak Soner

Design - Algorithm

- OK we have the sensors and an algorithm to make sense of the sensor readings
- everything works on paper
- let's start designing its realization



© 2024 Burak Soner

Design - Analog vs. Digital

- First design choice → analog vs. digital (we'll choose digital of course, but still, let's investigate)
  - Analog: Continuous-time, "continuous-valued". The physical world is mostly analog.
  - Digital: Discrete-time, discrete-valued. Computers are mostly digital nowadays.

- Our algorithm inputs (sensors) are analog. We can digitize them straight away and use digital computation OR keep them as is and use analog computation

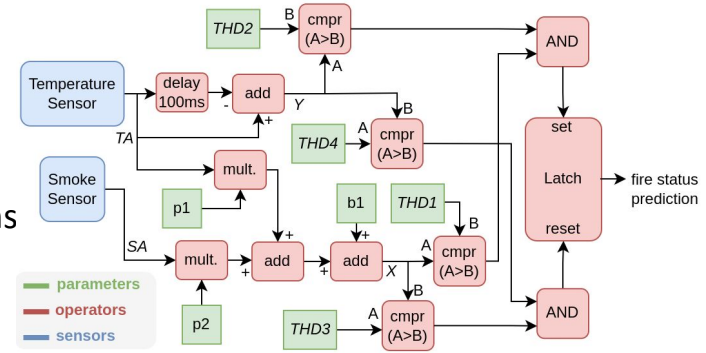
- let's consider the analog case first (we will not start designing analog circuits now, but we will consider them as modules to make sense of the design in the analog domain)

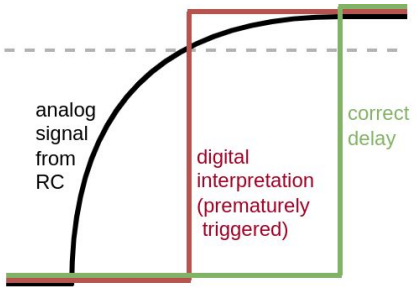
© 2024 Burak Soner

Design - Analog vs. Digital

Analog computing

- Constant levels (parameters) can be realized with some resistive dividers + buffers.
- There are 6 different operators, we need to design analog realizations for each:
  - add ⟶ passive avg (with resistors) + 1 amplifier of gain=2
  - comparison ⟶ differential amplifier (2 transistors)
  - mult ⟶ typically 4+ transistors (a.k.a. “modulator”)
  - delay ⟶ other than RC delay, which is not really a delay! ⟶⟶⟶⟶⟶⟶⟶⟶⟶ it’s tough to build, but possible (see bucket brigade or CTD)
  - AND, latch⟶ these are inherently digital but they have analog implementations for a given “logic” voltage level. An AND gate is 2 transistors in series, and a latch is a bistable configuration of those (typically called SR)





© 2024 Burak Soner

Design - Analog vs. Digital

Analog computing

- OK we **were** able to build the system in analog fashion, what's the problem?
- Alongside design challenges with certain components (e.g., a simple analog delay is much harder its digital counterpart), analog designs suffer significantly from external disturbances, noise and loss.
- Specifically, since analog values are continuous, minor inaccuracies such as tolerances, parasitics, thermal effects etc., change the information they carry.
- Furthermore, signal losses are always present and typically vary unpredictably. Together, these cause the signals to always be "dirty", i.e., you never have a deterministic output like 0/1 as in digital.
- In our fire detector example we could tune the parameters assuming clean signals or a more realistic certain set of "dirty" signals and the system could then fail (either false positives or false negatives) in the case of unexpected amounts of noise and loss due to such disturbances.

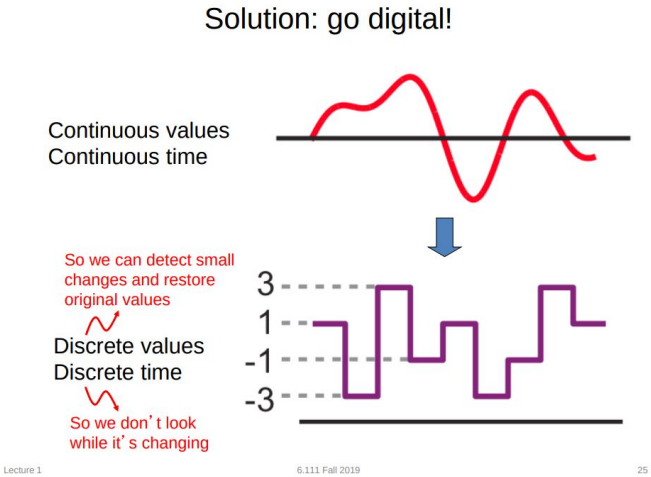
© 2024 Burak Soner

Design - Analog vs. Digital

© 2024 Burak Soner

Motivation for digital computing

- Noise and inaccuracy are unavoidable though, so what can we do?
- The digital abstraction is a “workaround” to this
- If our application allows us to settle for a few distinct voltage levels instead of the whole voltage range, we might recover the correct signal from its noisy mix and avoid error
- For example, if we can get by with only 3 distinct levels, e.g., {0, 0.5, 1} V, we can treat 0.3V as 0.5V and 0.1V as 0V, and so on. This way, if there’s a < 0.25V disturbance, we’re good!



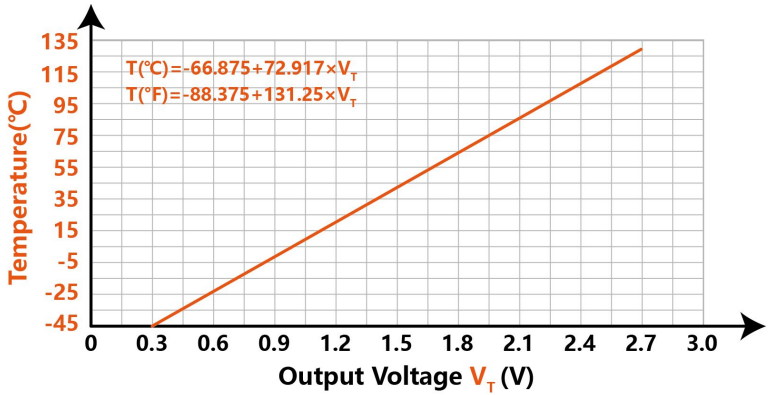
- Next step ⟶ let’s try to realize a digital design for our system

Design - Analog vs. Digital

Digital computing

- To realize our design with digital computation, we first need to “digitize” the analog inputs, i.e., sample them in time and discretize (“quantize”) the values

  → Note: We will not cover sampling in detail in this course (consider taking DSP: ELEC 303 for that if you haven’t), but we will cover quantization and its effects in detail.



- Temperature and smoke density have slow dynamics, so sampling them at a modest ADC clock of 1 kHz would be more than enough.

- SHT30 transfer characteristics are shown on the right. Let’s assume our smoke sensor has a similar response curve, and that 30%-70% of the total range would correspond to safe and dangerous smoke density level limits respectively

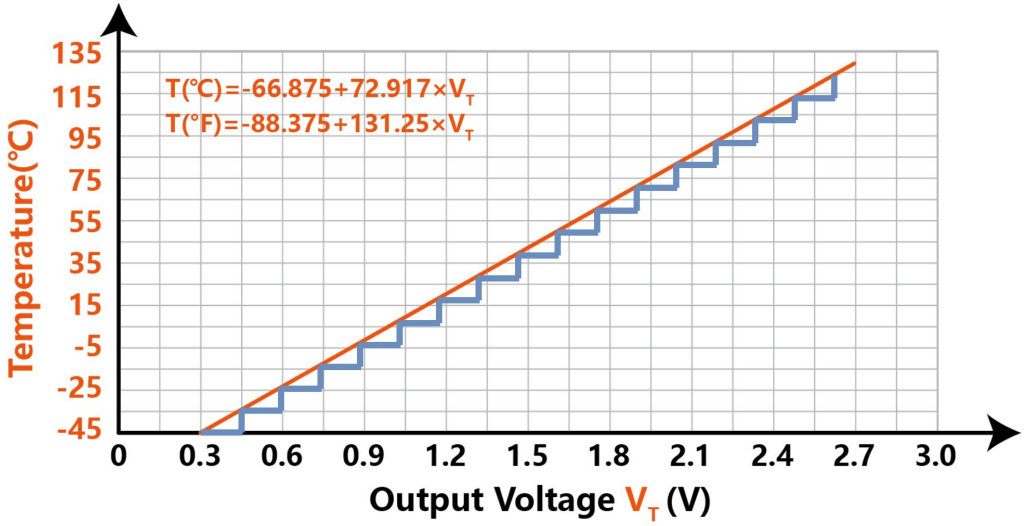
© 2024 Burak Soner

Design - Analog vs. Digital

© 2024 Burak Soner

Digital computing

- Quantization → Let's chop this 0.3V-2.7V voltage range up to 32 pieces and represent it with a uniform fixed-point number representation (5-bits)



- This means we'll have the following values to work with (other values will be rounded to these somehow): {0.300, 0.375, 0.450, … , 2.550, 2.625}

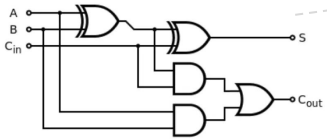
  →Note how we don't have 2.7 anymore, that would require a 33rd value

- We can now treat this set of 32 values as a 5-bit digital value set (i.e., {b00000, b00001, b00010, …, b11110, b11111}) and design around it

© 2024 Burak Soner

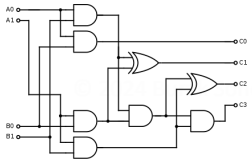
Design - Analog vs. Digital

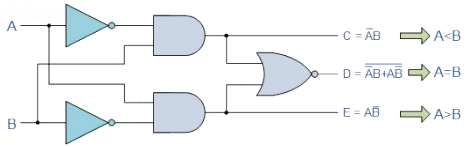
Digital computing

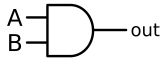
- Let's review our operators again like we did in the analog case:
  - add ⟶ Recall the full adder 

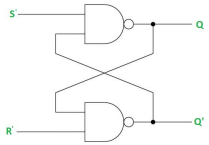
(don't worry we'll review these when the time comes)

this is the 1-bit version of course, we'll need the 5-bit version but it's the same thing



  - mult ⟶ same approach, different circuit for multiplication:
  - cmpr ⟶ 

  - AND ⟶ trivial: 

  - Latch ⟶ 

- Overall, the design complexity turned out to be significantly lower than the analog case because we were able to use the **digital abstraction** and do gate-level design!

- delay: 1 kHz clock + a 5-bit latch + a counter triggering at 100

© 2024 Burak Soner

Design - Analog vs. Digital

Comparison

Now let's take a step back and compare what we have in digital vs. analog

- Digital (mostly) saves us from the detrimental effects of noise, we get deterministic outputs (to be fair the output of the task was binary like yes-fire / no-fire so it's inherently more amenable to a digital design, anyhow →analog = noise problems)
- Digital looks simpler (thanks to the gate abstraction), but it actually has **many more transistors**, which means it spends more power, and probably takes up more area
  - 5-bit full adder requires on the order of 100s of transistors. The analog adder needed only 1!!
- This also implies analog chips should be cheaper than their digital counterparts → Yes… but not really! Once you're in production, the material costs are naturally lower yes, but the development effort drives the costs up in analog!
- The easier development process drove costs down for digital, especially in CMOS, and led to the famous Moore's law (exponential scaling in number of transistors per mm^2), which basically meant this for hard analog tasks: "if you can do it fast / resolute enough in digital, don't bother with the analog design, you'll be better off in the long run"

© 2024 Burak Soner

Design - Analog vs. Digital

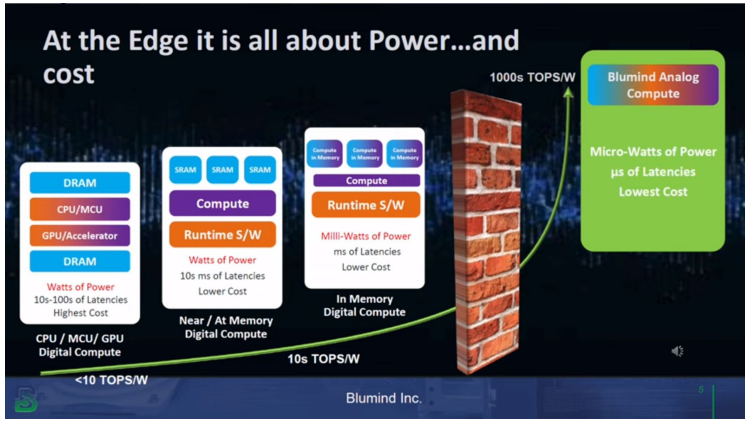
Comparison

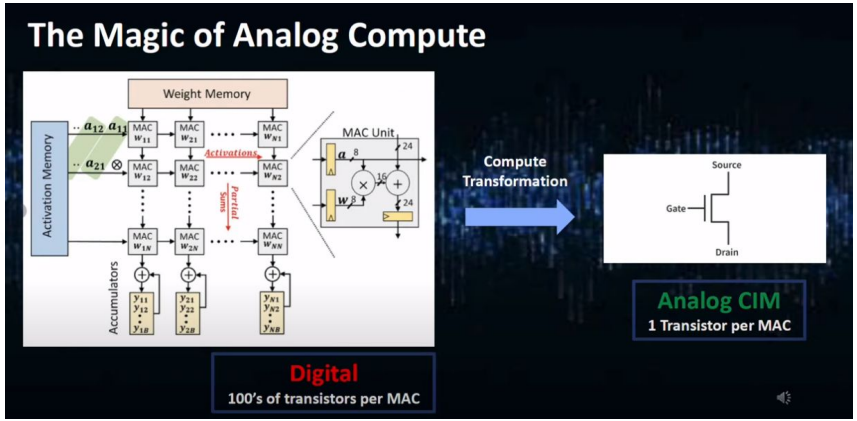
- *"So… do we always choose digital?!"* →No, proven behavioral design + financial means (and time!) to support the physical design effort + millions / billions of volume could mean analog is better. But if you're prototyping or need re-configurability after deployment, digital is probably better.
- This is typically a very complex analysis since the business implications are huge though, so don't take my word for it.
- Today, power and signal path (radio & comms) chips are mostly analog, and recently the analog AI accelerator market is growing (e.g., see Mythic and Blumind) together with neuromorphic designs.
- There are also designs based on different materials / processes such as carbon nanotube FETs, memristors, phase-change memory etc. and chipmakers are trying to find ways to increase efficiency with tricks like compute-in-memory (a.k.a. in-memory compute? the terms are relatively new).
- > 50% of the market is still digital and mixed-signal (where the reconfigurable parts are mostly digital)

© 2024 Burak Soner

Design - Analog vs. Digital

Blumind's presentation at tinyML Asia 2023 had a few informative slides about this, click on the screenshots if you want to see the video:





Many startups like Blumind have popped up in the last 10 years, trying to build the ultimate edge AI accelerator with the best TOPS/Watt. Movidius (acquired by intel) was probably one of the first, with their "Green Computing" vision processors ("Myriad", not an analog design but it was revolutionary at the time).

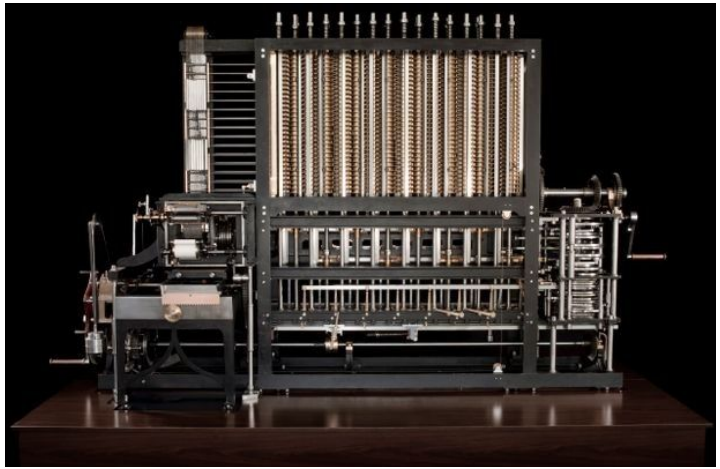
We haven't seen a "winner" yet, the problem → requirement variation is just too high for different apps

© 2024 Burak Soner

Design - Analog vs. Digital

Bonus: Other types of digital systems

- We only discussed electrical approaches so far, and more specifically voltage-based signaling
- That's a bit unfair, the first digital computer was mechanical! → The Babbage Engine
- Since an adder is easier to build with gears than mul and div (sound familiar?), Babbage designed a machine that realizes the method of finite differences:
  - basically polynomial approx allowing arbitrary ops with addition only, of course with a certain approx error
  - Input numbers with levers, turn the wheel, get your answer
- This is a clever design optimization, we'll frequently do stuff like this to work our way around constraints



© 2024 Burak Soner

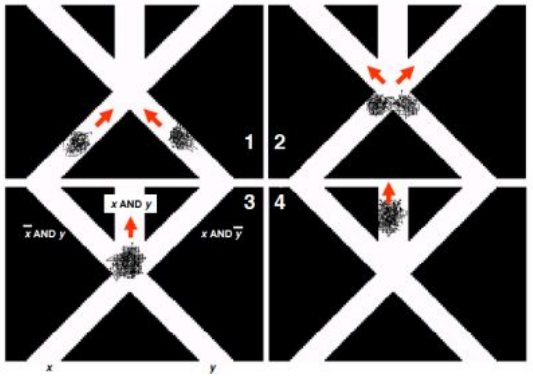
Design - Analog vs. Digital

© 2024 Burak Soner

Bonus: Other types of digital systems

- Mechanical/electrical analogies are well-known by now though, so maybe this wasn't a shock. Well how about a swarming-behaviour-of-soldier-crabs digital computer from 2012 !!

> "Back in the early 80s, a couple of computer scientists … studied how it might be possible to build a computer out of billiard balls… This information is processed through gates in which the billiard balls either collide and emerge in a direction that is the result of the ballistics of the collision, or don't collide and emerge with the same velocities. Now … a couple of pals have built what is essentially billiard ball computer using soldier crabs. "We demonstrate that swarms of soldier crabs can implement logical gates when placed in a geometrically constrained environment," they say"



- A true digital computer, but not a practical (or animal/environmentally friendly) one for sure. Takeaway: there are other ways to realize digital systems, not just voltage-based, not even just electrical!

Design - Analog vs. Digital

© 2024 Burak Soner

Bonus: Other types of digital systems (and more)

OK these were interesting for sure, but let's get back to the practical approaches and recap:

- Voltage-based signaling is extremely amenable to CMOS (dominant manufacturing technology today) and most components consume approx. 0 power in resting state (e.g., think non-volatile SSDs), making it the dominant approach. **We'll exclusively focus on voltage-based in this course.**
- However computing based on current amplitude, or the phase or frequency instead of amplitude, are also possible, and all have different applications (think BJTs vs. FETs)
- Other emerging technologies for the curious:
  - Optical computing based on fiber modes, nanophotonics, …
  - Reservoir computing with organic-electrical hybrids → Brainoware (extremely interesting)
  - Quantum computing
  - …

© 2024 Burak Soner

Design - Circuits vs. Processors

- OK we chose digital, let's look back at our design → we built dedicated circuits for our task
- This isn't the only way though, we're all more familiar with using processors for realizing algorithms like this in the digital realm.
- Specifically for a task like ours (fire detector), we interface with physical signals, so we utilize embedded processors (microcontrollers, "MCUs")
- On an MCU, this algorithm would not take longer than ≈ 50 lines of code!

- Let's have a go on an Arduino:

```
// define the quantize_to_5bits() function
void setup(){
// define sensor transfer characteristics
// define p1, p2, b1, THD1, THD2, THD3, THD4
int delay_counter = 0;
pinMode(1, OUTPUT);}

void loop() {
  uint8_t TA = quantize_to_5bits(analogRead(A0));
  uint8_t SA = analogRead(A1);
  delay(100); // unit is ms
  uint8_t TA_100msdelay = analogRead(A0);
  int X = TA*p1 + SA*p2 + b1;
  int Y = TA - TA_100msdelay;
  if((X > THD1) && (Y > THD2)){
    digitalWrite(1, HIGH)
  }
  else if((X < THD3) && (Y < THD4)){
    digitalWrite(1, LOW)
  }
  delay(1);  // for stability at approx. 1kHz sampling
}
```

(this is of course not exactly the same implementation, the timing's off, we're not using 5-bits etc., but it serves our purpose of analyzing processor implementations)

© 2024 Burak Soner

Design - Circuits vs. Processors

- That was much easier than designing a digital circuit with gates etc. (higher level of abstraction!) and it's still digital.
- However, this implementation failed to realize a few things (alongside precise timing):
  - Parallelization: the computation of X and Y use the same arithmetic resources on the processor, but on the circuit we can just synthesize two separate units and compute them simultaneously to gain speed if needed.
  - Arbitrary arithmetic: the smallest data type in arduino is 8-bits, but with our circuit design we could go down to 5 bits and use "just enough" resources for the task if cost was an issue.
  - Task specificity: it's not just the arithmetic that's wasteful, the processor has TONS of extra overhead (it has to "boot" to a stable state for starters). This is unavoidable as the processor is useless without this specific overhead. This means power, space, … all sorts of extra costs.
- Of course, the arduino is an especially weak processor, and some processors (especially custom MCUs or massively parallel ones like GPUs or ones with programmable fabric in them) do allow this sort of customization so this comparison doesn't generalize well, but it does show our point.

© 2024 Burak Soner

Design - Circuits vs. Processors

- So in general, when would we choose to build circuits instead of software on processors?
- Naturally, the reasons differ with respect to your design goals, but here are a few:
  - **First step towards an ASIC:** Probably the most popular reason outside the defence sector. When you're going to build a system in the milions (think of something like a 555, ≈1B/year 🤯), every transistor counts, so you scrap all unnecessary components and build an Application Specific Integrated Circuit (ASIC). Building and verifying your digital circuit for this task is a much stronger verification before the ASIC phase compared to doing that on an MCU because you will surely not put that MCU in your ASIC!
  - **High performance:** Software is flexible but its performance is bounded by the hardware architecture of the processor. For instance, the AVX-512 extensions on modern processors have convoluted software implementations that get the best performance out of them for 512-bit vector ops (refs: 1, 2). However, if you want anything larger (e.g., image processing), you're out of luck. Processor makers cannot support that sort of flexibility continuously unless it's economically feasible (it's almost always not!). FPGAs (on which we build arbitrary circuits, not processor-friendly software) typically fill this gap.

© 2024 Burak Soner

Design - Circuits vs. Processors

- Reasons geared towards mission-critical apps like aviation and defence (more refs: reddit, vhdlwhiz, fpgainsights):
  - **Flexibility:** Changing requirements over the course of a project might render your previous processor choice obsolete (legacy comm protocols, power requirements, wider parallelization, …), incurring a lot of technical debt. The custom circuit (FPGA) approach is flexible here since you can upsize/downsize your design as needed, without system/board-level changes.
  - **Security / anti-tampering:** Reverse-engineering is **significantly** easier for software than an FPGA bitstream, and doing so in a semantically meaningful way on FPGAs (e.g., getting behavioral HDL code from the bitstream) is even harder. One reason: compiling software is significantly less complex than implementation + place-and-route on the FPGA. There are also a lot of encryption or obfuscation based protection methods too. In general, there is a lot of interesting ongoing work on this (1, 2, 3, 4).
  - **Reliability and testing:** More abstraction means easier design, but harder testing! Hardware can be verified more reliably than software for this reason (+ thanks to companies like Xilinx you have super-accurate device models, that's part of why Vivado is huge). Furthermore, when you have flexible hardware, you can do things like thermal-aware design to avoid failures and further improve long-term reliability.

© 2024 Burak Soner

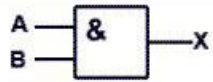
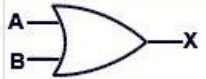
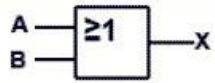
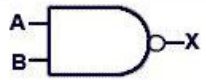
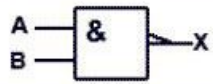
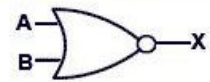
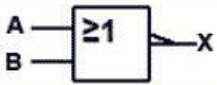
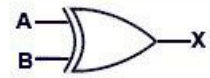
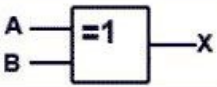
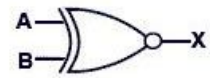
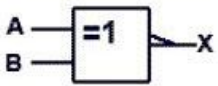
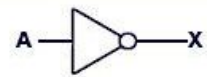
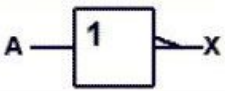
Design - Logic Components

- OK we've decided on a custom digital circuit for this task, let's examine components. There are basically two types of logic components: combinational and sequential
  - Combinational logic = output depends only on input
    - e.g., button pushed, LED turns on and stays on as long as the button is down, turns off when button is up
  - Sequential logic = output depends on input + "state" (memory)
    - e.g., button pushed, LED turns on, however this time it turns off when the button is pressed again
- Sequential logic has memory! Depending on how the memory is arranged, how it can be accessed and the data it holds, different types of "automata" can be built: Finite state machines, "Pushdown" machines (≈ FSM + stack), random-access machines (very similar to current Von Neumann computers with CPU + RAM), Turing machines.
- Check John E. Savage's book for more on automata theory if you're curious. My knowledge on this topic is limited so I don't want to mislead anyone here with a light treatment.

© 2024 Burak Soner

Design - Logic Components

- Recall the basic set of logic gates →→→→→
- A gate is defined by an input-output function (truth table) + temporal response (propagation delay)
- These are our “lego pieces”, we connect them in various ways to realize both combinational and sequential logic, hence, our digital circuits
- However, some of these are easier to manufacture and integrate than others (e.g., NAND flash) and there are also simplification algorithms (e.g., SOP and POS) so we typically do not rely on all of them at the same time

ANSI Symbol	IEC Symbol	NAME
		AND
		OR
		NAND
		NOR
		XOR
		XNOR
		NOT

© 2024 Burak Soner

Design - Logic Components

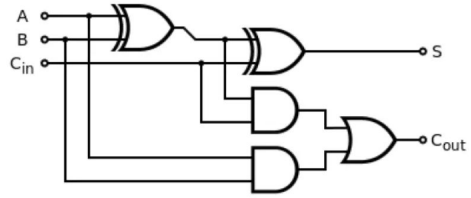
- Let's choose two example components from our fire detector example to study combinational and sequential logic over: the add component and the delay component
- Addition is combinational, the operation has no state. It just takes two inputs and computes the output based on a set of logic rules
- Specifically, our 5-bit adder needs to do the following:
  - Do 5 x 1-bit additions
  - Manage the carry bit in each step
  - Push out the 6-bit result (5 bit + 1 carry)
- Note that the circuit does this **"non-stop"**, there is no state change to wait for, it keeps outputting A+B as fast as possible, this is the essence of combinational circuits!

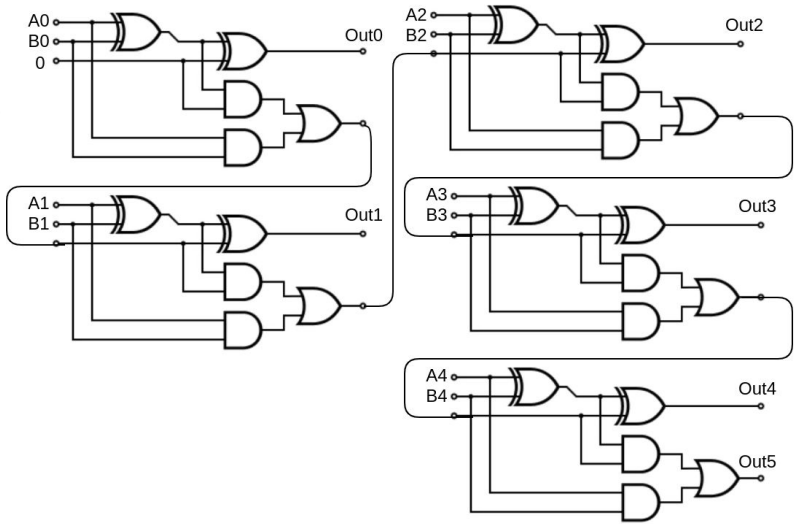
© 2024 Burak Soner

Design - Logic Components

Combinational Circuit (add)

- Recall the 1-bit full adder on the right
- To extend this to 5-bits, we connect a 0 to the first C_{in} bit, and connect the C_{out} of each consecutive adder to the C_{in} of the next. The final C_{out} is the additional mandatory 1 bit resulting from the addition operation
  - Recall: addition of two N-bit numbers creates an additional bit, output gets represented by N+1 bits. Multiplication of two N-bit numbers becomes 2N bits.



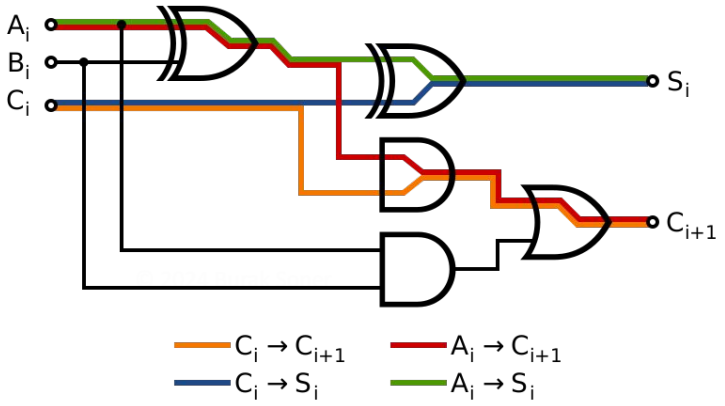


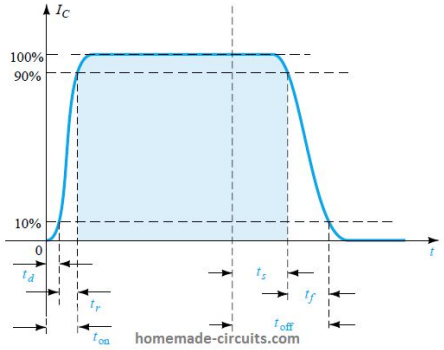
© 2024 Burak Soner

Design - Logic Components

Combinational Circuit (add)

- The logic is OK. But there is one more aspect, mostly covered in a “ceremonial” manner in introductory courses like ELEC 205: **propagation delay**
- Real circuits have finite bandwidth, you can’t shift voltages up and down between logic levels in 0 time (such a square wave requires ∞ bandwidth!)
- The rising and falling times make up the delay, and the worst case delay of this circuit as a whole (considering all input→output links) defines the latency





Note: we all know the response is never this clean, especially with smaller transistors. However we’re abstracting out that part and settling for a single latency number to simplify things. Physical designers do a lot of black magic there when they actually build a chip with gates like this but that’s beyond our scope here.

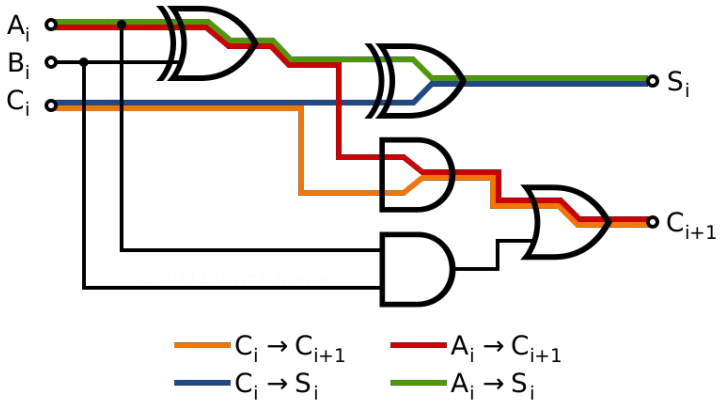
© 2024 Burak Soner

Design - Logic Components

Combinational Circuit (add)

- Even though this is purely combinational, the higher-level app typically at least saves the values in time as a series, so the system is typically state-based (has memory)
- This typically means the circuit is clocked at a certain speed. As expected, this clock speed is dictated by the worst case delays in the circuit (we can't try saving the output of the next cycle before this cycle's output settles!)

Note: There's one important "lookahead" point here, FPGAs implement gates with look-up tables inside "configurable logic blocks" (CLBs) to be able to make them programmable. However these CLBs naturally have different propagation characteristics since they are different than actual fixed gates, so the delay we compute on paper will not be equal to the delay on the FPGA. Furthermore it will change from FPGA to FPGA due to the way the CLBS are set up on the chip. Managing the delay on an ASIC requires a lot of black magic as discussed earlier so that's a whole other topic. We'll see more on managing delay and clocks in FPGAs later.



- In the case of this N-bit full adder, our gate delay is:

$$t_p = 4 + 2(n-2) + 2 = 2n + 2$$

where n is per-gate delay

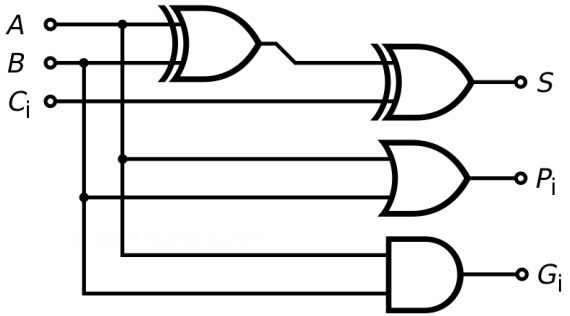
(assuming all gates have the same delay)

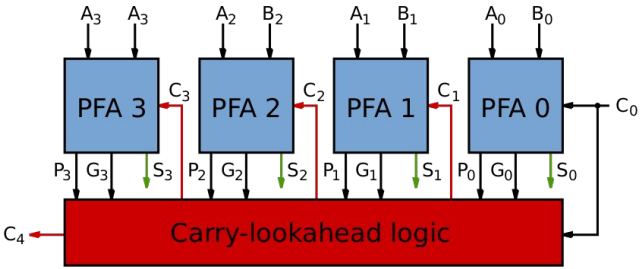
© 2024 Burak Soner

Design - Logic Components

Combinational Circuit (add)

- Notice something here ⟶ Our full adder has a worst case delay of 12 unit gate delays in 5-bit, and it grows linearly with number of bits (in 32-bit it has 66, that's huge!!)
- Our current design is called a "ripple carry adder", and it's a naive approach at an adder circuit with a large word width.
- Faster versions are available, such as the "carry lookahead adder" which uses a partial version of the full adder as the building block and adds a "carry lookahead logic" component and attains O(1) in 6 (constant) unit delays. One downside ⟶ this needs gates with more than 2 inputs for n>2.





© 2024 Burak Soner

Design - Logic Components

- OK we've characterized a combinational component, went over the preliminary design aspects, we'll dive deeper in later lectures and labs
- Let's do the same for the delay function to review sequential components
- The delay function can be implemented as follows:
  - a 1 kHz clock
  - a 5-bit latch
  - a counter triggering at 100, saves the delayed 5-bit value to the latch at each trigger
- Leaping ahead a bit → the counter already contains latches, so let's just have a look the counter and the clock in detail.

© 2024 Burak Soner

Design - Logic Components

© 2024 Burak Soner

Clock

- The clock is not exactly a digital component, it's rather something like a signal source. Generating a stable / accurate clock of a desired freq on a chip is a **loaded** topic and it's not in the scope of this course, we will only see basics.
- The clock generation process starts from an oscillation source: a crystal, a tank circuit, and in some niche high-freq use cases (>10 GHz) ⟶ cavity resonators (DRO, Gunn etc.). Since oscillators typically cannot exactly produce the desired signal (harmonics and exact frequency), we add some signal conditioning (e.g., filters) and scalers (multiplier / dividers).
- Also sometimes, when the speed of these "source oscillators" are not enough for the application (e.g., crystals are typically limited at ≈200 MHz, but you want to power a processor at 3 GHz), we may use something called a phase-locked loop (PLL).
- PLLs are not only used to multiply clocks though, they have a pretty vast application space, and designing a good one for a given set of application requirements is very hard work ⟶ check these extra pll references and the last chapters of the famous "High Speed Digital Design: A Handbook of Black Magic" book for more info on clock generation

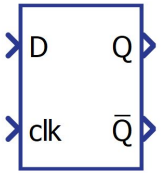
Design - Logic Components

© 2024 Burak Soner

Sequential Circuit (counter)

- The simple free-running counter is the canonical example of a sequential circuit. Its current count value is its state, and it jumps between states at each clock tick.
- In other words, its input is always the same: tick, tick, tick, … and its output is only dependent on its current state (remember Moore vs. Mealy finite state machines? This is a Moore's FSM)

- Our counter FSM of limit=100 therefore needs 100 states. To represent 100 different states with binary logic, we'll need 7 bits (6 makes 64 states available, not enough, and 8 →256 states, too much).
- These state bits are typically stored in circuits via flip-flops, the most popular one being a D flip-flop.



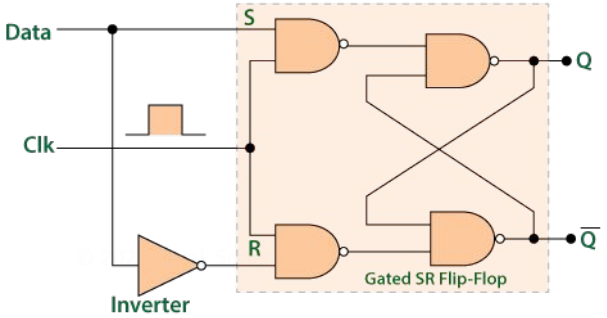
Clock	D	Q	Q'	Description
↓ » 0	X	Q	Q'	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1

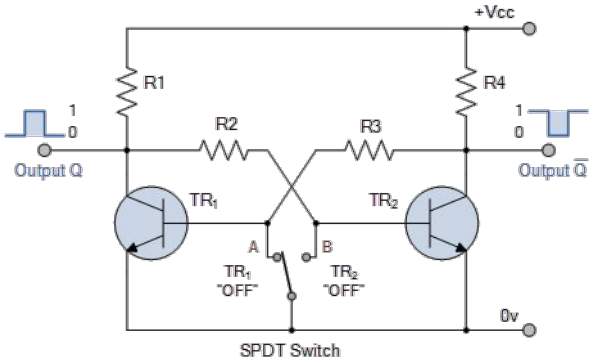
© 2024 Burak Soner

Design - Logic Components

Sequential Circuit (counter)

- How does a digital component do this though? How does the flip-flop “latch” onto a certain value and retain that? We do not know of any gates that can do this.
- Answer → bistable multivibrator!
- The feedback connection on the SR part makes this possible. It’s a circuit that is stable at two points and unstable at all others. The trigger pushes the output between those two states.
- The transistor version is a bit more intuitive. You might remember this sort of feedback behavior from lab work on “Schmitt triggers” in introductory circuits courses.





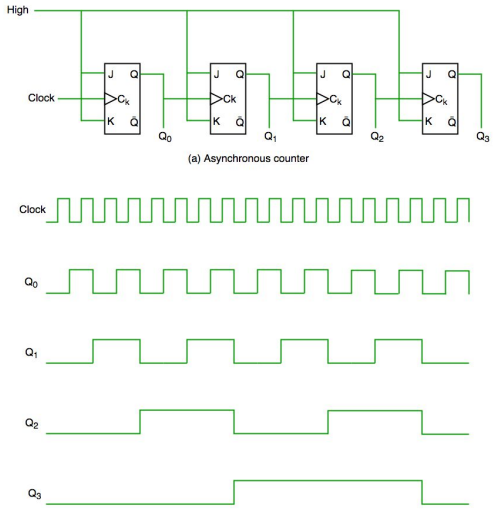
© 2024 Burak Soner

Design - Logic Components

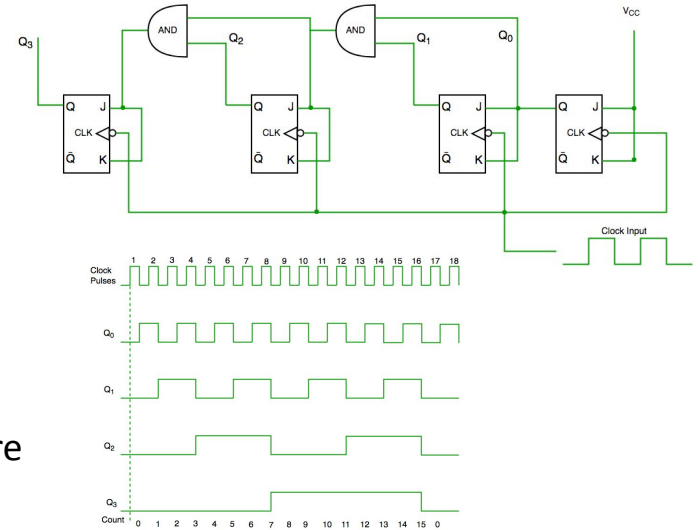
Sequential Circuit (counter)

- There are two main types of counter circuits: synchronous and asynchronous, they have different use cases, in our case it doesn't really matter because the clock frequency is very low

Asynchronous Counter (4-bit)



Synchronous Counter (4-bit)



the 7-bit versions are simple extensions

© 2024 Burak Soner

Design - Logic Components

© 2024 Burak Soner

Sequential Circuit (counter)

- The propagation delay issue and related design aspects are valid also for sequential circuits just like combinational circuits
- However, the fact that a slow clock dictates the operation of the circuit makes things easier in the case of our fire detector → the delays are orders of magnitude smaller than one clock period, so the delays become negligible.
- The main challenges in sequential circuits arise from issues related to clock management (e.g., CDC) and other application-level issues (sampling etc.).
- Terminology note → we put together flip-flops to build **registers**. The register abstraction is important because it will be our most basic unit of memory!

Design - HDLs and FPGAs

- OK we analyzed digital vs. analog designs, investigated using processors vs. custom circuits and dove a bit deeper into how our digital circuit can be implemented. We are ready to deploy this realization.
- Old school ⟶ we can deploy this using discrete logic ICs (the 74 series)

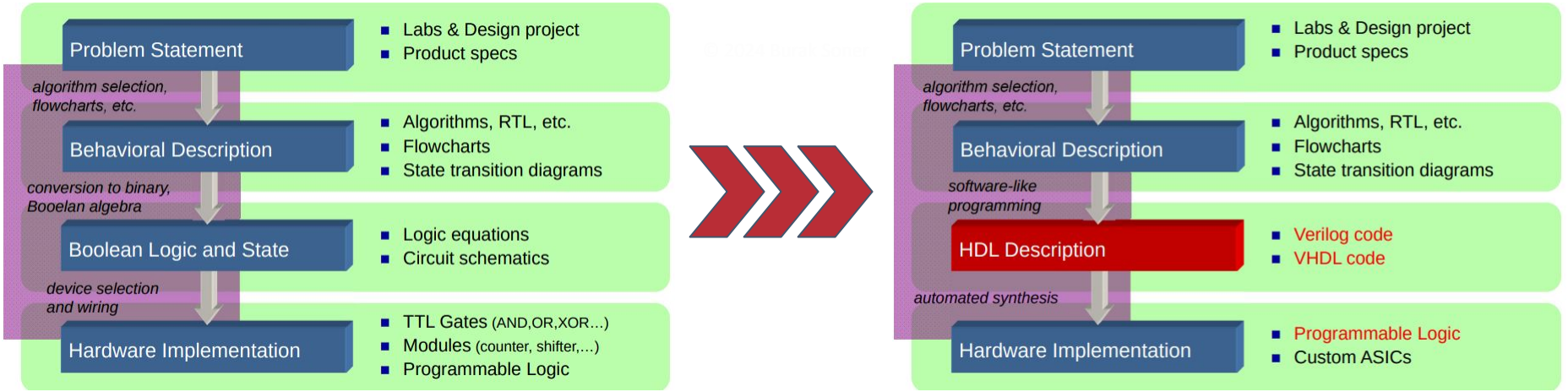
Part number	Units	Description	Input	Output	Pins	Datasheet
74x00	4	quad 2-input NAND gate			14	SN74LS00
74x01	4	quad 2-input NAND gate; different pinout for 74H01		open-collector	14	SN74LS01
74x02	4	quad 2-input NOR gate			14	SN74LS02
74x03	4	quad 2-input NAND gate		open-collector	14	SN74LS03
74x04	6	hex inverter gate			14	SN74LS04
74x05	6	hex inverter gate		open-collector	14	SN74LS05

- This might actually be an option for our small fire detector system example, but we can all imagine ⟶ for anything larger, this approach will not be scalable and that's certainly not how people build larger systems like processors etc. nowadays…

© 2024 Burak Soner

Design - HDLs and FPGAs

- The need is clear → another level of abstraction on top of gates, a language, so that we can define and simulate the “behavior” of such circuits clearly.
- The language is to serve as both the input of an automated “gate netlist generator” for easy deployment, as well as a specification of the behavioral model of the circuit.



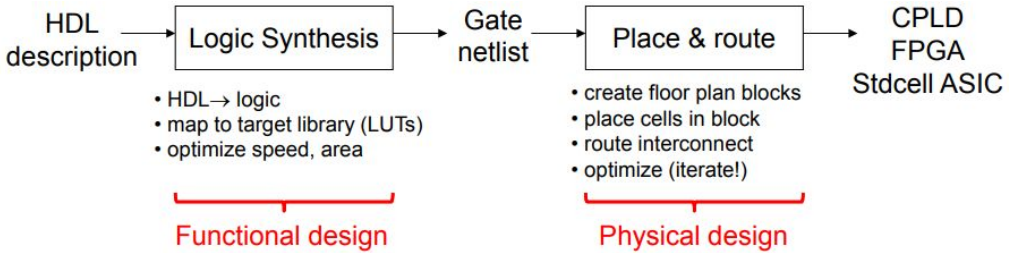
© 2024 Burak Soner

Design - HDLs and FPGAs

- We do have languages like this now, they are called hardware description languages (HDLs). Digital designers typically use these while designing circuits instead of netlist drawings like we did earlier. The most popular ones are VHDL and Verilog (and also SystemVerilog)

- HDLs are not even restricted to digital! There is an analog version of Verilog called Verilog-A which analog designers use to describe and simulate circuits (SPICE fashion)

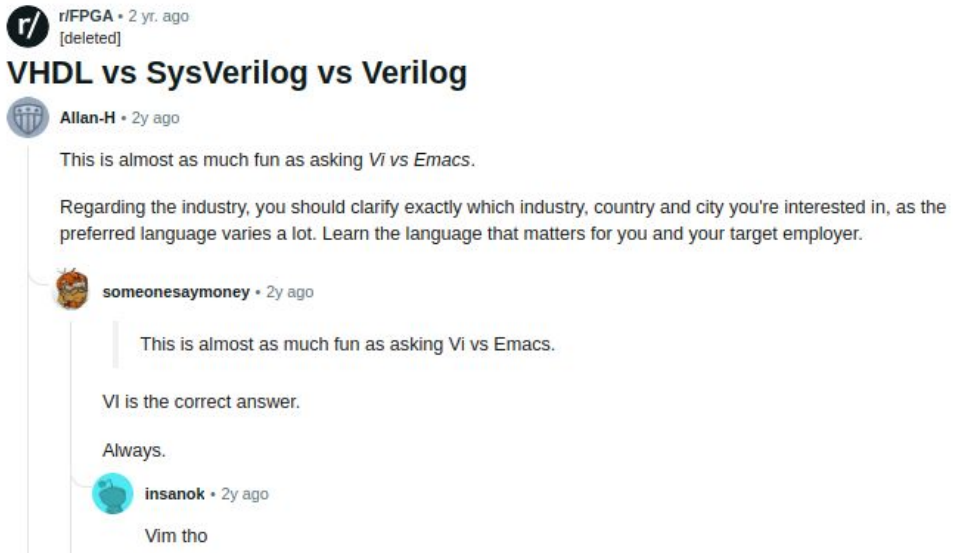
- This way the design procedure gets separated into two automated parts which can be improved independently ⟶ synthesizing a gate-level netlist + realizing a physical layout for that netlist on a chosen piece of hardware (an FPGA, or an ASIC with a pre-determined underlying structure)



© 2024 Burak Soner

Design - HDLs and FPGAs

- The VHDL vs. Verilog (+SysVerilog) debate is endless, highly resembles any debate on any programming language or software tool, and in my opinion it's a bit funny



- The debate is also pointless of course, as these two HDLs were not created differently for "artistic" reasons, they serve different purposes.
- Industry-dependent and geographical (cultural?) differences typically dictate the choice

© 2024 Burak Soner

Design - HDLs and FPGAs

© 2024 Burak Soner

- VHDL was invented by and for the defence industry (specifically, US DoD) and tied to MIL-STD-454 →→→
- The dominance of the US over the global defence arena might have been the reason for this → most non-US defence industries are also highly inclined towards VHDL as opposed to Verilog, the latter sees wider use in commercial sectors.

The Department of Defense (DOD) is engaged in a number of programs which require VHDL (VHSIC Hardware Description Language) models of ASIC's and systems. Specifically, the details of the deliverable VHDL models are expressed in a combination of documents such as MIL-STD-454, the VHDL Data Item Description (VHDL-DID (DI-EGDS-80811)) and any additional requirements specified in any given Contract Deliverable Data Items ("CDRLs" or "data items").

VHDL data items capture the behavior and structure of an electronic system, subsystem, or device. The primary purpose of these data items is to document hardware designs in a machine executable, simulatable, and hierarchical format. VHDL models themselves must be inspected to insure that they meet the requirements specified in the contract or VHDL-DID, as applicable. The VHDL-DID may be tailored by the contract requirements for some applications.

For acceptance, VHDL simulation models provided to the Government as CDRLs must satisfy some known acceptance and verification criteria and procedure. These criteria and procedures are the purpose of this document.

- We will study VHDL almost exclusively in this course, not because of its ties with the defence industry, but mostly because it's a very explicit (formally called "strongly typed") language that allows to picture the hardware more clearly (less abstractions).

Design - HDLs and FPGAs

- Example VHDL code from an online source for an up counter is shown on the right
- We'll dive into this in more detail later on, but for now let's point out an important difference with programming languages:
  - This is not code that gets executed line by line like in C, Python, … The whole source describes a circuit (the entity!), with input/output ports, signals (on wires) and "processes" inside the behavioral definition signifying the sequential components.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- FPGA projects using Verilog code VHDL code
-- fpga4student.com: FPGA projects, Verilog projects, VHDL projects
-- VHDL project: VHDL code for counters with testbench
-- VHDL project: VHDL code for up counter
entity UP_COUNTER is
    Port ( clk: in std_logic; -- clock input
           reset: in std_logic; -- reset input
           counter: out std_logic_vector(3 downto 0) -- output 4-bit counte
     );
end UP_COUNTER;

architecture Behavioral of UP_COUNTER is
signal counter_up: std_logic_vector(3 downto 0);
begin
-- up counter
process(clk)
begin
if(rising_edge(clk)) then
    if(reset='1') then
         counter_up <= x"0";
    else
        counter_up <= counter_up + x"1";
    end if;
 end if;
end process;
 counter <= counter_up;

end Behavioral;
```


© 2024 Burak Soner

Design - HDLs and FPGAs

- Verilog is a commercial effort, invented at a company called Gateway Design Automation (acquired by Cadence)
- At that point VHDL was open, Verilog was proprietary. Realizing this would prevent widespread adoption, Verilog was converted to an open IEEE standard (#1364).
- System Verilog (superset of Verilog) followed this with #1800.

A Tale of Two HDLs

VHDL	Verilog
ADA-like verbose syntax, lots of redundancy (which can be good!)	C-like concise syntax
Extensible types and simulation engine. Logic representations are not built in and have evolved with time (IEEE-1164).	Built-in types and logic representations. Oddly, this led to slightly incompatible simulators from different vendors.
Design is composed of entities each of which can have multiple architectures. A configuration chooses what architecture is used for a given instance of an entity.	Design is composed of modules.
Behavioral, dataflow and structural modeling. Synthesizable subset...	Behavioral, dataflow and structural modeling. Synthesizable subset...
Harder to learn and use, not technology-specific, DoD mandate	Easy to learn and use, fast simulation, good for hardware design

Lecture 2 6.111 Fall 2019 43

Design - HDLs and FPGAs

- Once the HDL source is ready, we feed it to two automatic tools in series:
  1) gate netlist generator, which is typically called “synthesis”, and
  2) place-and-route, which is typically called “implementation”

- The synthesis output is generic, it’s the gate-level design that we drew earlier and can be implemented anywhere (discrete digital ICs, FPGAs, ASIC). We’ll have a better idea about this after having a look at VHDL fundamentals so let’s focus on implementation for now.

- Alongside the synthesis output, the implementation tool takes in chip constraints, and plans a layout of the synthesized circuit on the die using its “resources”.

- For ASICs this is typically a free-for-all situation (although there are some standards), and in FPGAs these “resources” are called configurable logic blocks (CLBs)

© 2024 Burak Soner

Design - HDLs and FPGAs

- We will study FPGA internals in detail during the lab tutorial, so let's summarize the basics here.
- The CLB unit is the core of the FPGA as it realizes the letter P (Field **Programmable** Gate Array). It can mimic most of the fundamental digital units we covered earlier (gates, or more complicated units like the D flip-flop). It typically has a complicated design with many resources inside to maintain versatility.

Table 2-1: **Logic Resources in One CLB**

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM(1)	Shift Registers(1)
2	8	16	2	256 bits	128 bits

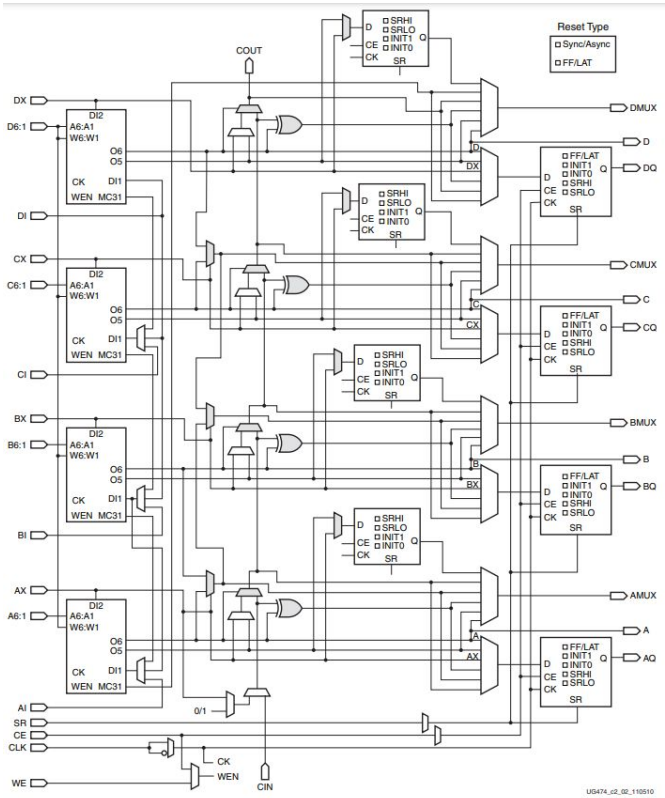
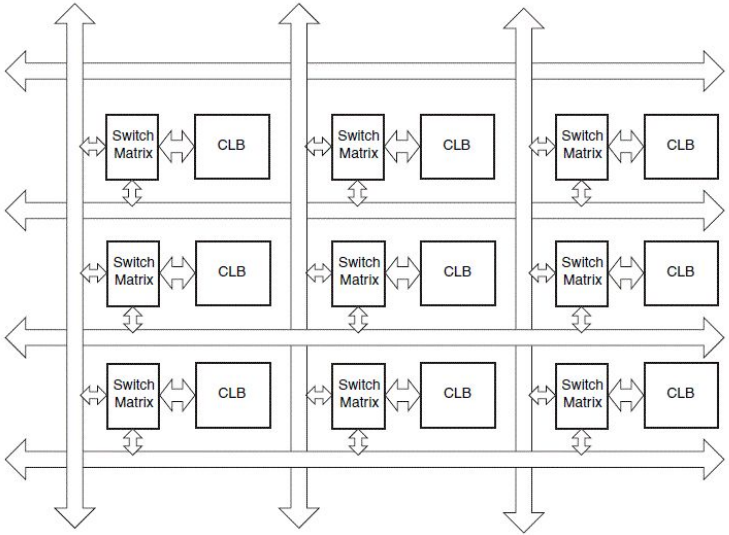


Figure 2-3: **Diagram of SLICEM**

© 2024 Burak Soner

Design - HDLs and FPGAs

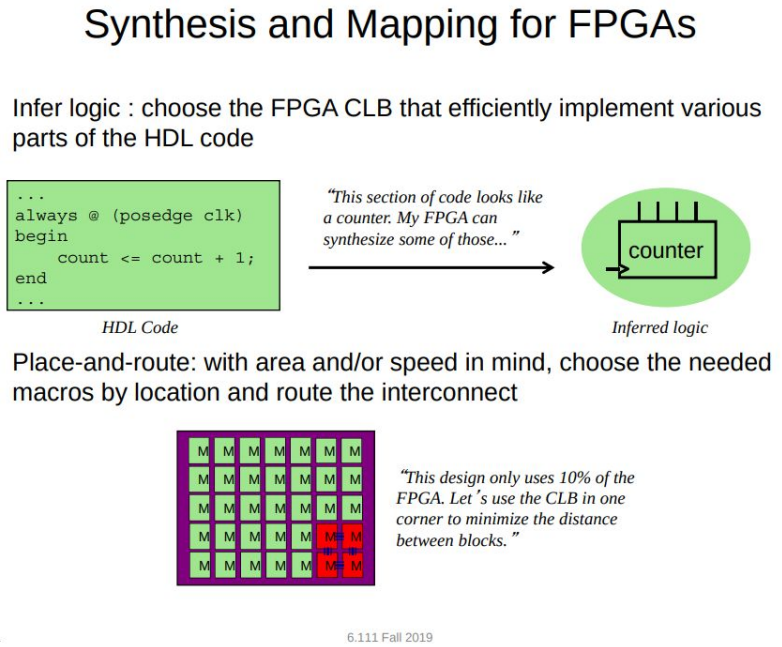
- Between these programmable CLBs is another programmable component, the “interconnect”
- The implementation tool takes the gate-level design, assigns the resources inside the CLBs to certain roles as per the needs of the design, and then uses the switch matrices on the interconnect buses (typically 2D, see right) to connect those configured CLBs together and realize the circuit.



© 2024 Burak Soner

Design - HDLs and FPGAs

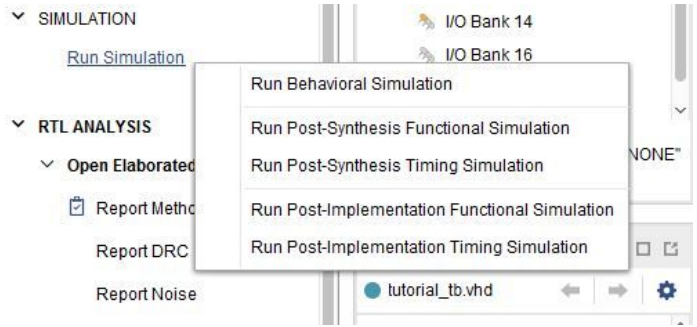
- As you can imagine, finding the optimal combination of CLB assignments and interconnect configurations is not trivial, especially so for large circuits. Implementation tools typically work iteratively and require a significant amount of computation with high-fidelity physical models.
- This is part of why Vivado is such a huge software package. It's literally building a circuit on the FPGA automatically like this.



Design - HDLs and FPGAs

- Once the implementation phase is completed, the designer then has the option to upload the generated bitstream to program the FPGA and finally realize the circuit in hardware.
- However, there is a crucial step before this that any serious project must consider: simulation.
- The reason is simple: by simulating this design before deployment in the development tool, you can 1) give the circuit arbitrary inputs very easily (as opposed to doing this via a signal generator + logic analyzer on a desktop lab unit), and 2) debug internal signals alongside outputs in response to those inputs.

- A few types of simulation is typically available:



- Behavioral and post-synthesis simulations are better for uncovering your coding bugs since they are faster (no routing info), but are typically inaccurate for timing. Post-implementation simulations are closer to the real case.

© 2024 Burak Soner

Design - HDLs and FPGAs

- Synthesis, implementation and accurate simulation are **fascinating** technologies, and we can safely say these are among the primary “accelerators” of the modern semiconductor industry.

- While they certainly sound like topics that only people lik 30-year Xilinx veterans would know something about (especially the place-route and post-implementation simulations), there are successful open source efforts (FOSS) in this domain! Some examples †:

  - Icarus Verilog ⟶ https://steveicarus.github.io/iverilog/
  - YosysHQ’s “nextpnr” ⟶ https://github.com/YosysHQ/nextpnr + https://arxiv.org/pdf/1903.10407.pdf
  - A python-based “modern” HDL ⟶ https://github.com/amaranth-lang/amaranth
  - EDA playground’s online simulator ⟶ https://edaplayground.com/

- The board support is not great in these though, so handle with care.

† special thanks to İhsan Kehribar for introducing me to these

© 2024 Burak Soner

Design - Recap

- OK we've covered almost every aspect (except hardware testing), let's recap this intro:
  - We laid out task requirements and chose sensors
  - We compared and contrasted digital vs. analog realizations of the fire detector
  - We investigated a (very simple) processor implementation of this algorithm and discussed what advantages could be leveraged if this was instead implemented via a custom circuit
  - We analyzed the modules in the circuit design and discussed their implementation details (using combinational and sequential logic components, gate-level design)
  - We investigated deployment options for the circuit. We discussed HDLs as a scalable alternative to gate-level representation, and how FPGA deployment is the natural choice. We briefly discussed FPGA structures as well as FPGA toolkits which allow for automatic netlist generation from HDLs as well as place-and-route and simulation.

© 2024 Burak Soner

© 2024 Burak Soner

next → mandatory lab tutorial

we'll have a look at Vivado + VHDL projects