

notes.txt

Flip-flop:

- Setup time: minimum time from when input D is stable to rising clock edge
 - input must propagate through internal gates
- Hold time: minimum time for D to remain stable after rising clock edge
 - input must propagate through internal feedback gates

D latch:

- sets memory by input whenever clock is high
- (not on rising edge of clock)

D flop flop

- sets value on rising clock edge
 - and thus, only once per clock cycle
- internally it is two d-latches chained together:
 - the first with an inverter in front of it's clock
 - the second without
- value of D is available at Q immediately after rising clock edge

T flip flop

- on rising edge of clock toggles state if T (input) is high
- if T is low, keeps current value
- IS edge triggered

shift register

- called n-bit shift register
 - n is how many bits it stores
- accepts one bit a a time as input
 - (this is serial)
- outputs all bits at the same time, one pin for each bit
- also has a out pin for chaining multiple together
- internally it is a chain of flip flops
 - previous output goes to next input, so the data shifts down the line

rotate register

- a shift register with it's output connected to it's input
- rotates the stuff in it's buffer

universal shift register

- shift register with lots of other functions
- e.g. clear, clock, shift, right, left, load, read, control

shifter

- just a different combination of wires that moves the bits over
- can be designed to map one of the edge bits to the filler bits

barrel shifter

- shifter that can shift any arbitrary number of bits
- barrel shifter with n shifters can shift between 0 and $(2^n - 1)$ bits
- e.g. 3 shifters goes 0-7

- and uses shifters of 1, 2, and 4 bits

fraction multiplication approximation in binary

- compute $A/3$, error ≤ 5
 - each numerator is picked using:
 - * $\text{numerator} = (2^{\hat{x}} / \text{desired_divisor})$
 - $1/3 = 1/4$, error: $(0.25 - 1/3) = 0.08 \rightarrow 25$
 - $1/3 = 3/8$, error: $(0.375 - 1/3) = 0.04 \rightarrow 12.5$
 - $1/3 = 5/16$, error: $(0.3125 - 1/3) = 0.02 \rightarrow 6$
 - $1/3 = 11/32$, error: $(0.34375 - 1/3) = 0.01 \rightarrow 3$
- 11/32:
 - $(8+2+1)/32$
 - $(2^{\hat{3}} + 2^{\hat{1}} + 2^{\hat{0}})/2^{\hat{5}}$
 - $A/3 = (C_{ii3} + C_{ii1} + C_{ii0})_{ii5}$

counter

- n-bit counter has n bits
 - one design uses n-bit increment-er and n-bit register
- designed to roll over, so that you can chain multiple together
 - has an 'overflow' bit for that reason
 - overflow is bitwise AND of all output bit numbers
 - * (AND is highest representable number)
 - must have a count pin separate from clock pin
 - * so that they can be on the same clock when chained
- down-counter counts down instead of up
 - it's overflow is bitwise NAND of all register bits
- limit a counter to any arbitrary value:
 - make circuit that matches that binary value (outputs 1 only when it gets that value)
 - wire the output of that circuit to the reset pin of the counter

bidirectional counter

- must have different carry bit
- when counting down, you want NOR all inputs as carry (matches 0)

parallel vs serial

- parallel
 - n-bits wide
 - n bits move every clock cycle
 - the bus is n wires plus the clock
 - usually slower clock than serial
- serial
 - one bit wide
 - one bit moves every clock cycle
- you can convert serial data to parallel using a shift register and clock divider

serial addition

- add a number bit by bit
- you store the sum in a shift register because that can accept serial input
- slower than parallel adder, but uses far fewer gates

register file

- pins: clock, w_data, w_addr, w_en, r_data, r_addr, r_en
- more than one register in one chip
- writing
 - w_data is wired to all registers
 - * (buffers are used to make sure the signal doesn't get weak from length)
 - w_addr and w_en go to a decoder which enables one of the registers
- reading
 - the output of every register is wired to the data in port of a driver
 - r_addr is wired to a decoder
 - the output of that decoder is wired to the enable port of each driver
 - the output of every driver is directly connected to r_data
 - this is safe because the decoder ensures that only one of the drivers is ever active

clock divider:

- used to take a fast clock input and output a slower output
- use a count up counter
- if it's a power of 2, you can use an n-bit counter and tc (clock overflow) as clock output
- if it's not a power of 2:
 - up counter
 - * add logic to reset the counter when it reaches the desired number of ticks
 - * connect divided clock output to that reset
 - down counter
 - * set the load input to the desired number of ticks (as constant inputs)
 - * wire reset to load
 - * connect divided clock output to that reset
- the pulse width of the clock output may be different than you expect
 - it will be right for the power of 2 case
 - but for the other case, it is probably wrong
- there will be a phase difference that is very difficult to fix
 - due to gate delay

clock timing:

- to find maximum clock speed, use the formula:
 - $T = (\text{clk-}\lambda Q) + (\text{max } Q\text{-}\lambda D) + \text{setup time} + \text{skew}$
 - * T clock period

- $1/T = \text{clock frequency}$
- * $\text{clk-}\lambda Q$: delay of one flip-flop
 - interval of time after clock rising edge where Q has correct value
- * $\text{max } Q\text{-}\lambda D$: most delayed path from ANY flip-flop output to ANY flip-flop input
 - the input/output don't have to be on the same flip-flop
- * setup time:
 - length of time after the clock rising edge that D must remain the same value in order for the flip-flop to function
- * skew: optional, clock skew of the circuit
 - (if not given, it is 0)

metastability

- happens when
 - you have asynchronous input (e.g. human pres-able button)
 - you break the setup time or hold time
- when output is metastable it is somewhere between 0 and 1
 - you can't predict where it is
- fix for button:
 - wire it into a synchronizer flip-flop and then to the rest of your circuit
 - * (because flip-flop uses the same clock as the rest of your circuit)
 - maybe more than one synchronizer flip-flop for a very clean signal
 - * (at the expense of one clock cycle delay each)

glitching

- outputs that are initially not stable
 - e.g. signal flip flops a little before settling on the correct value
- happens because not all inputs get to the logic at the same time due to gate delay and stuff
- fix by putting a d-latch or flip-flop after the output
 - called registered output
 - this way the output isn't seen until the next clock cycle
 - (the output is assumed to be stable by then. If it isn't, you probably need a longer clock cycle)
 - you can combine that flip flop into the state register

FSM: Finite State Machine

- usually store the state in a state register
 - could also use a collection of flip flops, shift register or whatever
- to prove that two edges are disjunct:
 - AND together the boolean expressions, and simplify
 - you should end up at 0 because it is never possible to satisfy both transitions
- to prove that one condition is always satisfiable
 - OR together all boolean expressions and simplify

- * you should get true because one of them is true for every input
- normal state encoding
 - assign a number to each state
 - use the binary representation of that number
 - allows for the smallest register possible
- one-hot encoding
 - for n states, use a n-bit register
 - each state is n binary bits with one 1 and the rest 0
 - may allow for simpler next-state logic

FSM state reduction

- group states of FSM by what they output
- then, for each input:
 - sub-group the groups by next state according to every combination of that input
- after all this, if 2 states are always in the same group, they are duplicates

FSM -> circuit: (will be on final)

- 1. assign each state a value for the state register to hold
 - don't worry about optimizations yet
- 2. make truth table
 - inputs: current state, inputs to FSM
 - outputs: next state, output
- 3. design circuit using truth table
 - current state + inputs -> next state logic
- if you're asked to match a bit pattern, it is easier to use a shift register
- (or make one out of d-flip-flops)

FSM reducing states:

- group by z-value
- sub-group by arrows pointing into them
- and then by arrows pointing out
- then any states in the same group are duplicates

Mealy vs Moore FSM:

- Moore
 - output is dependent on current state only
 - * when input changes, output does not change until next clock cycle
 - output logic can be separated completely from next state logic
 - * (output logic is then not dependent on inputs)
- Mealy
 - outputs on arrows
 - output is dependent on state and inputs
 - * when input changes, output changes asynchronously
 - * (potentially not to the correct value)
 - means outputs are not lined up with clock edges
 - Mealy and Moore FSMs can be combined

HLFSM: High Level Finite State Machine

- compared to normal FSM:
 - multi-bit input/output
 - local storage
 - arithmetic operations
 - * (FSM can only do boolean, since everything is 1 bit anyway)
 - a state transition can be a more complex expression
 - * e.g. (in_reg < 5)
- convention:
 - := for assignment
 - == for equality comparison
 - '0' for single bit value
 - 0 for integer value
 - "00" for multi-byte value
 - * multi-byte outputs must be local storage

RTL Design: Register Transfer Level Design

- high-level description of the logic involved
- uses multi-byte I/O and generic blocks
 - multi-byte I/O is like arrays
- like a block diagram
 - too complex for a normal FSM, so design with a HLFSM

RAM memory types

- register file
 - discussed previously
- SRAM: Static RAM: Random Access Memory
 - uses 2 inverters back-to-back
- DRAM: Dynamic RAM
 - uses special on-chip capacitors to store data
 - data is encoded in the charge of the capacitor, so it is possible to store multiple bits per cell
 - requires special chip design, so it's usually a separate chip as everything else
 - charge value can decay, so it must be refreshed
 - * (usually taken care of by the chip firmware)
- size:
 - register file: lots of transistors/cell (least dense)
 - SRAM: 6 transistors/cell
 - DRAM: 1 transistor/cell (most dense)
- Speed:
 - register file: fastest
 - SRAM
 - DRAM: slowest
- RAM has setup/hold times just like flip-flops

ROM: Read Only Memory

- non-volatile
- can be very fast
- low power
- does not need to be refreshed (like DRAM)
- PROM: Programmable Read Only Memory
 - can be written to only once
 - implemented using fuses or ant-fuses
- EPROM: Erasable Programmable Read Only Memory
 - written to using higher-than-normal voltage
 - erased by exposure to UV light
 - * meaning chip must have a window
- EEPROM: Electrically Erasable Programmable Read Only Memory
 - EPROM than can be erased electrically

MEMS: Micro Electrical Mechanical System

- tiny chip-sized devices that aren't necessarily gates or stuff
- e.g. gyroscopes

Queue

- FIFO: First in, First Out
- visualize as a circle
- contains limited max size elements
- 2 pointers: front and rear
- push (write):
 - write to address rear
 - increment rear
- pop (read):
 - read from address front
 - increment front
 - does not really destroy content, it is still accessible
- if front or rear reaches size:
 - next value should be 0
- if rear == front:
 - could mean full or empty
 - depends if this happens after read or write
 - after write: it's full
 - after read: it's empty

making a FSM for a Queue is a good exercise

- should probably do this as studying for the exam

RTL tradeoffs

- Latency
 - time from data input to data output in seconds
- Throughput

- how many jobs can you input per second
- pipelining
 - do things in stages, like an assembly line
 - helps with throughput
 - does not help with latency
 - put a pipeline register in in-between steps
 - doesn't make task faster, but you can start a new task before the previous one finishes
- concurrency:
 - do things side by side
 - helps with throughput but not latency
- it is sometimes possible to do both concurrency and pipelining
- component allocation:
 - choice of components used to implement design
 - example: if you have 2 states that need multiplication:
 - * it might be more compact to have them use the same multiplier and simply MUX the inputs
 - * slightly more delay, many fewer gates
- operator binding:
 - how you configure the MUXes in compact allocation
 - if you map them efficiently you can reduce delay
 - e.g. if one input is used in more than one scenario, arrange the inputs such that that one always has the same port
 - * and then, you don't need a MUX there at all
- operator scheduling:
 - introduce new states to preform operations
 - * downside is more states
 - * upside is states do fewer things each
 - * (which means fewer gates)
 - e.g. to reduce the number of multipliers required
 - can reduce the longest path through the circuit which allows you to increase the clock speed
- other optimizations:
 - serial vs concurrent computation
 - * datapath: carry-ripple (serial) vs carry-lookahead adder (parallel)

chip design types

- full custom
 - exactly what it sounds like, gate for gate design
 - very hard to make
 - takes a long time from design to mass production
 - cheapest per chip in large quantities
 - most expensive per chip if you're making only one
 - * (very much in the realm of "Why would you do that!?!")
- standard cell ASIC: Application Specific Integrated Circuit

- has pre laid out cells
- each cell can be made into a gate
- you must still make custom mats to fabricate the circuit
- vs full custom
 - * easier
 - * bigger and slower
 - * more expensive per chip
- gate array
 - structured ASIC
 - gates are already on the chip, they just need to be wired together
 - vs standard cell ASIC
 - * faster and easier to manufacture
 - * bigger and slower
 - * more expensive per chip
 - some use only NAND or NOR gates
 - * since both of those are Turing complete
- FPGA: Field Programmable Gate Array
 - like a gate array that you can reprogram on the fly
 - * at least the original ones were.
 - * modern ones use lookup tables
 - vs gate array
 - * easier
 - * slower
 - * more expensive
 - * bigger
 - no non-recurring costs
- Single IC's
 - like the stuff we did in the first few labs
 - does not scale at all
- SPLD: Single Programmable Logic Device
 - predates FPGAs
 - prefabricated IC with large AND and OR structure
 - has programmable nodes before each AND gate
 - * could be fuse based (not reusable)
 - * or memory based (reusable)
 - used to implement simple logic

how a FPGA works

- ### TODO
- LUT = Look Up Table
 - MUX + memory
 - example: 3 binary inputs = 8 combinations
 - use 8-1 MUX map 8 pre-computed input values
 - if a circuit uses more inputs than a LUT can handle, you must cascade to more than one LUT

- * (just like you do with gates)
 - * this is more efficient than larger memories anyway
 - it is common for a LUT to have fewer than it's maximum inputs
 - * thus the extra data is wasted, but that isn't a big deal
 - you can also have LUTs with more than one bit outputs
 - SM: Switch Matrices
 - AKA programmable interconnect
 - the way that LUTs are connected together
 - also MUX+memory based
 - each output is MUXed to an input
 - CLB: Configurable Logic Block
 - LUT + flip-flop + MUX + memory
 - memory programs MUX to output stored (flip flop) value or logic value
 - chip layout
 - hundreds of thousands of SMs and CLBs in a grid with wires connecting adjacent blocks
- programming a FPGA
- all memory of all blocks is connected sequentially as one big shift register
 - known as a scan chain
 - you shift in your configuration (code) that does stuff and then the FPGA is programmed

TODO: processor internals