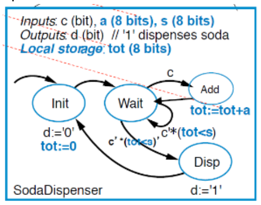
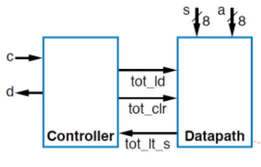
RTL Design

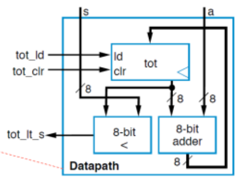
Combinational design (logic level) Behavior: Equations, truth tables
AND + OR + NOT → combinational logic
Controller design Behavior: FSMs | Register + comb.logic = controller
Processor design (Register transfer level) Behavior: High level state machine or pseudocode | Controller + Datapath = Processor
HLSM (high level state machine) multi-bit input/output | Local storage | Arithmetic operations


Inputs: c (bit), a (8 bits), s (8 bits)
Outputs: d (bit) // '1' dispenses soda
Local storage: tot (8 bits)
Init
Wait
Add
Disp
c
tot:=tot+a
c'*(tot<s)
c'*(tot<s)'
d:='0'
tot:=0
d:='1'
SodaDispenser

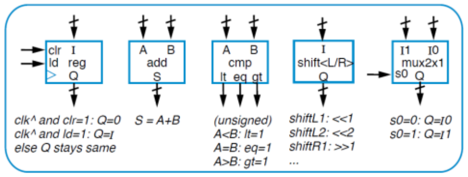
HLSM requires storing data (FSM doesn't)
FSM requires storing the state, but HLSM doesn't
Implementing the HLSM requires registers, and not required in FSM

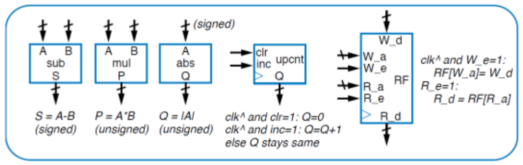


Step 2B

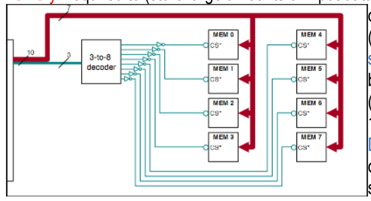


Step 2A





Memory Required to (store large amounts of input data) (use results of previous computations) (Persistent storage) (store instructions) Basic specification K by N (K words of N bits each) (Address = $\log_2 K$) (sz(data)= N) (byte enable=N/8)
1 K = 1024 bytes
DRAM Single transistor and capacitor to store bit | Cheap but slow | refresh required due to leak | Temporary storage. SRAM Memory cell uses flip flop to store bit (register). Requires six transistors to store bit | Holds data when power supplied | Fast

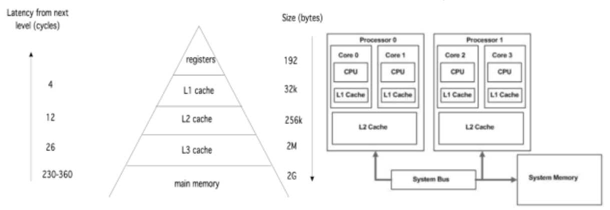


Anything with **bit** is a power of 10, anything with **byte** is a power of 2
Memory hierarchies | Better build up a larger memory from smaller units
Faster storage → cost more per byte → less capacity →require more power
Temporal locality (recent used item being likely to be used in near future)
Spatial locality (items with nearby addresses tended to be used close together)
Cache smaller faster storage device that acts as a staging area for a subset of the data in a larger slower device. For each K, the faster, smaller device at level K serves as a cache for the larger, slower device at level k+1. **direct mapping:** each block in k+1 is mapped to one block in k (faster but more likely to access main memory). **fully associative:** each block in k+1 can be anywhere in k (slower, less likely to access main memory). **n-way associative:** each block in k+1 can be in one of n blocks in k (slower, less likely to access main memory)

Advanced Encryptions standard

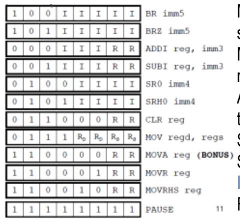
Byte substitution, shift rows, mix columns, add round key, view as alternating XOR key & scramble data bytes. Substitute bytes: A simple substitution of each byte. Uses one table containing a permutation of all 256 8-bit values. Each byte of state is replaced by byte indexed. Designed to be resistant to all known attacks. Shift rows: A circular byte shift in each row. Mix columns: each byte is replaced by a value dependent on all 4 bytes in the column (matrix multiplication). Matrix multiplication (element by element, N * N clock cycle) (Row by row, N clock cycle) (whole matrix, 1 clock cycle). Add round key: XOR state with 128-bits of round key | Processes by column | Inverse for decryption identical. AES key expansion: takes 128-bit key and expands into array of 44/52/60 32-bit words | starting by copying key into first 4 words | Loop creating words that depend on values in previous & 4 places back. Designed to resist known attacks | Design criteria (fast on wide range of CPU, invertible transformation, use round constants to break symmetry, simplicity of description. Decryption: Not identical to encryption since steps done in reverse | Define an equivalent inverse cipher with steps as for encryption. Implementation aspects: can efficiently implement on 8-bit and 32-bit CPU

Optimization performance (latency) | size (correlated with power consumption)
Represent logic function as a sum of products, give best possible performance
Using K-maps always yields the optimal solution | Functions with more than six inputs tend to be too complicated. Iterative improvement: Expand, reduce, reshape, irredundant
Trade performance for size | increase delay for lower number of gate inputs



Power consumption $P_{static} = I_{static} * V_{dd}$ $P_{dynamic} = CV_{dd}^2 f N^3$. Accelerators: Complete more work per clock cycle, reduce clock frequency | Trade performance for power savings. Clock usage: If multiple clocks are available, shut down those that are not used. Reduced power consumption modes: Idle, sleep, tradeoff latency and power consumption. Asynchronous Synchronous circuit → clock period | Asynchronous circuit → propagation delay. Asynchronous circuits change when inputs change while synchronous circuits change on clock edge. Synchronous circuits assume that inputs do not change too close to active clock edge. Main assumptions for asynchronous circuits: Only 1 input changes at a time | Input changes occur sufficiently far apart to allow the circuit to reach a stable state before next change |A stable state is a state circuits will stay in once reached till input change. Advantages: Speed, flexibility, power usage | Cons: Design complexity, glitches. Always to change one bit at one time to avoid race condition→use grey code to avoid. Hazards: A glitch when a signal temporary takes on the wrong value (glitches caused by structure of circuit and propagation delays are called hazards). Static hazard: When signal is not supposed to change its value in response to a specific change in an input, but instead momentarily does change. This happens when one path has a longer propagation delay than the other. Use group to eliminate hazards → add adj expression. Dynamic hazard: When signal suppose to change value, but there's small oscillation. A circuit with dynamic hazard must also contain a static hazard
To avoid dynamic hazards, design two level circuits with no static hazards.

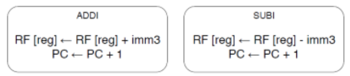
Stepper motor ASIP



1	0	0	I	I	I	I	I	BR imm5
1	0	1	I	I	I	I	I	BRZ imm5
0	0	0	I	I	I	R	R	ADDI reg, imm3
0	0	1	I	I	I	R	R	SUBI reg, imm3
0	1	0	0	I	I	I	I	SR0 imm4
0	1	0	1	I	I	I	I	SRH0 imm4
0	1	1	0	0	0	R	R	CLR reg
0	1	1	1	R_D	R_D	R_S	R_S	MOV regd, regs
1	1	0	0	0	0	R	R	MOVA reg (BONUS)
1	1	0	0	0	1	R	R	MOVR reg
1	1	0	0	1	0	R	R	MOVRHS reg
1	1	1	1	1	1	1	1	PAUSE 11

MOVE reg (move the motor by a number of full steps specified in reg)
MOVERHS reg (move half step) | PAUSE (wait) | CLR reg (clear reg)
ADDI reg, imm3 (add 3 bit unsigned immediate operand to the specified register)
SUBI reg, imm3(subtract) | SR0 imm4 (set lower 4 bit) | SRH0 imm4 (set 4 higher bit)
Implement ASIP: Create FSMD, draw datapath, create FSM for the controller

- `ADDI reg, imm3` `SUBI reg, imm3`



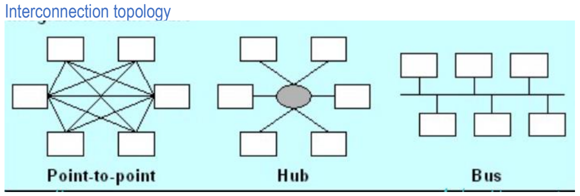
Delay counter: Use clock division, create smaller time interval

HWSW (Hardware software)

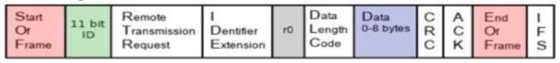
General purpose CPU: Slow but cheap (price and development)
Hardware accelerator: Faster but more expensive |Trade-off performance vs complexity
Energy efficiency: FPGA (1 Gb/J) | ASIC (10 Gb/J)
Mix of architectures: Constraints usually mean that a single choice for all is not suitable (performance, power, cost) | Most applications use a mixture of HW/SW technologies | Functional partition depending on requirements
Synthesis: Allocation, binding, scheduling
Some constraint example: $x_{A,1} + x_{A,2} \geq 1, x_{B,1} + x_{B,2} \geq 1, x_{C,1} + x_{C,2} \geq 1$
Time latency constraints: $5x_{A,1} + 15x_{B,1} + 10x_{C,1} \leq 30$
$10x_{A,2} + 20x_{B,2} + 10x_{C,2} \leq 30$
Objective function (energy): $10x_{A,1} + 6x_{B,1} + 3x_{C,1} + 3x_{A,2} + 8x_{B,2} + 3x_{C,2}$
Goal is to minimize objective function while meet the time constraint
JPEG compression: Discrete cosine transform, quantization, Huffman encoding
Design consideration: Latency, size, power, energy, time to market, profit
Implementations: Microcontroller, Microcontroller + accelerator, Fixed point calculations, compression acceleration

Embedded communications

Interconnection topology



Point to point: Many connections, routing mess, power consumption may high, modification painful expensive, fast communication
Hub (I2C): More cost than bus to central device, central control
Bus: need to access through common medium, protocol (CAN bus), no centralized control, constrained performance (timing issue), cheap, easy to do modification, implements some fault tolerance. | Communicate through a common set of wires
Each bus contains many wires: date wires, address wires, control wire
Bus protocol: Synchronous (all actions happen relative to edges of the clock) | Asynchronous (no centralized clock, actions can happen at any time)
Two types of devices on the bus (Clients and servers), only one device can control the bus at a time | Multiple devices on the bus are distinguished by their addresses (unique)
Serial communication: All nodes can initiate transmission (All nodes can be client), twisted pair bus | Applications (automotive, factory automation, machine control)
Signalling: Recessive (difference between two voltages less than a threshold (logic 1)) Dominant (difference between two voltages greater than a threshold (logic 0))
Better noise rejection
CAN message



Start of frame: for receives to sync what's going on | 11 bit ID: address of device, lower ID, higher priority | RTR, IDE bits: who got to use the device | DLC: how many byte in message | CRC (cyclic redundancy check): Additional bits for error detection (even parity, even number of 1 in the data)
Types of messages: Data frame (majority of messages), remote frame (no data, request data), error frame (when error detected), overload frame (request delay)
Ways to get clearer signal: Subtraction or take average
Arbitration: Nodes sense bus, if value not the same as being transmitted, they stop transmitting
Error detection: Bit error (sender receives a bit not equal to transmit bit) | Bit stuffing (more than 5 consecutive bits the same) | CRC error | Form error | ACK error