Lecture Notes
CS132

Number representation

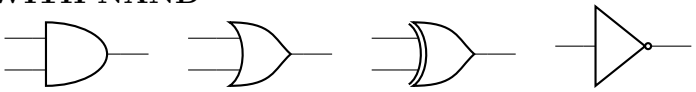
1. **Bases**: value $= \sum_{i=0}^{N-1} \text{digit}_i \times \text{base}^i$, with N being the number of digits. Word is the number of bits that can be processed simultaneously.

2. **Signed Magnitude**: MSB represents sign
   $1101_{2\text{SM}} = (-1 \times 1)(1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -5_{10}$
   values $\in [-2^{N-1}+1, 2^{N+1}-1]$ because $+0 \neq -0$.
   Two's complement: MSB represents negative value:
   $1101_{2\text{TC}} = 1 \times (-8) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -3_{10}$
   values $\in [-2^{N-1}, 2^{N+1}-1]$, one representation of 0.
   Find binary, pad with a leading 0, flip all bits, add 1.
   Subtraction: add negative number to binary of the other number (same num of bits, ignore overflow)
   Biased Form: similar to TC, but shifts all values by subtracting a **bias usually** 2^{N-1} **or** $2^{N-1}-1$ in formula: $(\sum_{i=0}^{N-1} \text{digit}_i \times 2^i) - B$.

3. **Fractions**: $\text{sign} \times \text{mantissa} \times 2^{\text{magnitude}}$ **MEMORISE NUMBERS OF BITS**

Digital Logic

Logical AND, OR, XOR, NOT are represented through: **MEMORISE HOW TO REPRESENT STUFF WITH NAND**



1. Everything can be a NAND, and it's cheap. **REMEMBER LAWS OF BOOLEAN ALGEBRA!**

2. **Karnaugh Maps** (K-maps): cicle 1's in a logic table, to simplify a given expression. Groups can wrap around edges. Can introduce X or "Don't care" cells which can be included in groups if needed to make them more efficient.

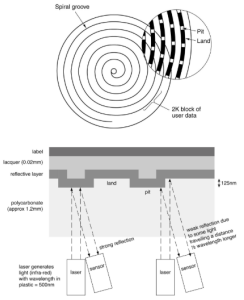
3. 1-bit **Half Adder** given two inputs, produces their sum and the carry. 1-bit **Full Adder** combines a row of N half adders to add N-bit numbers. It works like addition on paper, from right to left. For subtracting, inverse the binary number using TC, add 1, ignore overflow. In a logic circuit, **Z=0**: addition, **Z=1**: Subtraction. (*lecture 05 slide 18*)

4. **Decoder inp×out**: inputs (X_n) decide which outputs (Y_m) are high or low. Few inputs, to many outputs.
   Encoder inp×out: Many inputs to few outputs. Only one input active at a time.

5. **Active high**: activates when 1, **Active low**: activates when 0 is provided to it.

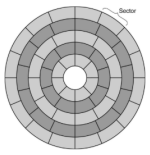
6. **Multiplexer inp-1** (MUX): select which input to output based on signal. Used as a source selection control, parallel to serial converters.
   De-Multiplexer 1-out (MUX): select which output line is active based on signal. Used as control for multiple lights, serial to parallel converters.

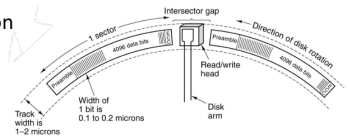
7. HDDs include optical disks such as **CD**:
   Optical Disks → Spiral Track
   - Think CD's... Can store 74 minutes
   - Spiral groove → 6000 tracks per cm
   - The track contains pits and lands → 15000 pits and lands per cm
   - Reads the difference between land and pit by comparing 2 laser signals
   - Lasers wavelength = 500nm

   which can handle large scratches (burst errors)

8. Long term memory: Hard Drives (HDDs)
   Hard Drives (HDDs) → Tracks and Sectors
   - Each disk contains multiple tracks
   - Each track contains multiple sectors → sectors are separated by a gap
   - Each sector contains a preamble, block of data, then an ECC
   - Preamble allows for synchronisation

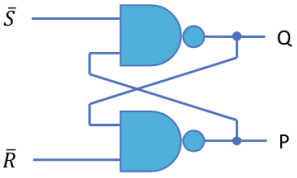
   

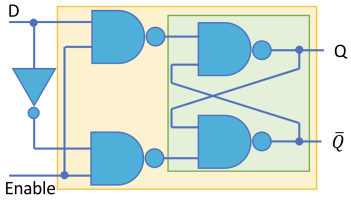
   

Sequential Logic

Computational logic produces output depending on its inputs only, while Sequential logic also depends on its **current state**.

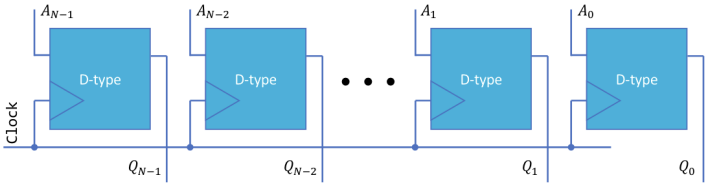
Flip-flop is able to switch on a pulse. Here, S is set, and R - reset, both are active low. Q is the output.



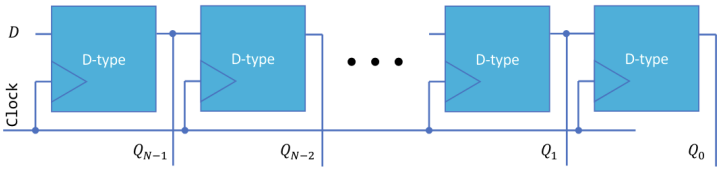


D-Type latch (Delay) extends a flip-flop and acts as a 1-bit memory storage. D is input, Q is output. Enable $0 : Q, \overline{Q}, 1 : D, \overline{D}$

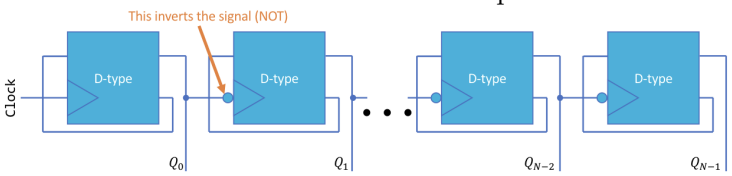
Latches only change on the rising edge of clock input. **N-bit Register** stores N bits on a low-to-high transition of the clock.



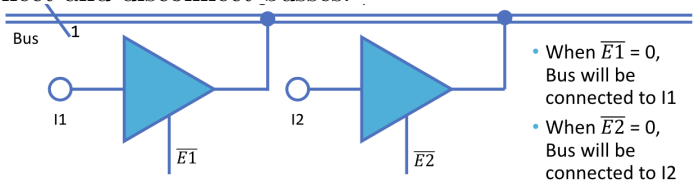
N-bit Shift Register stores N bits by sending N clock pulses and changing D



N-bit Counter each clock it counts up



1. **3-State Logic/buffer** also has enable, which introduces High/Low/Unconnected. Usually uses to connect and disconnect busses.



2. **Logic Integrated Circuits (ICs)** Programmable Array Logic(**PAL**): old, only programmed once, Programmable Logic Array (**PLA**): contains collection of AND gates feeding into OR gates, Field Programmable Gate Array (**FPGA**): contains millions of gates to replicate a chip. **UNDERSTAND PLA** ***lecture 6***

3. **Moore's Law**: For a given size of chip, the number of components would double every year.

4. **MEMORISE TRUTH TABLES FOR ALL DIAGRAMS**

5. **Forbidden** region is the space between logic 0 and 1, in volate, so between 0.8V and 2.8V.

Assembly

1. **Fetch-Decode-Execute** cycle: get current instruction, decode it into machine code, execute, repeat.

2. Assembly depends on processor architecture. High-level lang.→ Low-level Assembler lang. → Machine code→ hardware.

3. **Register Transfer Language (RTL)** is used to describe operations within a microprocessor. Memory is expressed as MS(< location >), and transfer is represented as [< toloc >] ← [< fromloc >]. **Fetch-Execute**
   1. [MAR] ← [PC]
   2. [PC] ← [PC] + 1
   3. [MBR] ← MS([MAR])
   4. [IR] ← [MBR]
   5. [CU] ← [IR(opcode)]
   6. [MAR] ← [PC]
   7. [PC] ← [PC] + 1
   8. [MBR] ← MS([MAR])
   9. [ALU] ← [MBR] + [D0]
   10. [D0] ← [ALU]

4. **RISC-V** has 32 registers, x0 is always 0. Has 6 formats for instructions: **R**: arithmetic, **I**: Shift and Load, **S**: Store, **B**: Branch, **U**: Special case, **J**: Jump.

```
Label :  Opcode Operand(s) # Comment
ADD x3, x1, x2 # Add values at x1, x2
```

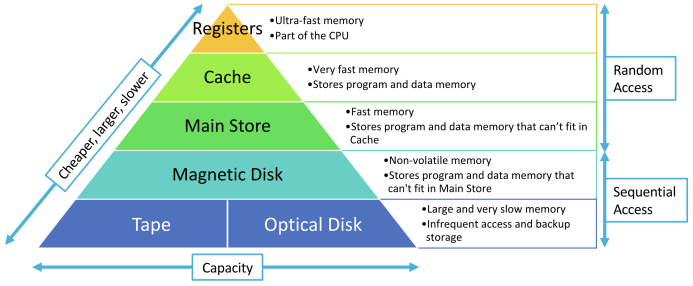
C

1. Primitive datatypes: `char`, `int`, `float`, `double`. Sizes can be dependant on machine, use **sizeof()** to find num of bytes it stores. Integers can be: `short, long, unsigned, signed`. No `boolean`: 0 is false, everything else is true. No classes, just `struct`, only used for variables. Strings are arrays of `char *` variables

2. **Pointers** int* reference the memory address instead of the value, defaulting at 0. void* doesn't have a type. No garbage collection, have to manage memory: `malloc`: Manual Allocation, `calloc`: Clear Allocation, `realloc`: Reallocation. Memory is contiguous.

Memory

1. In memory, have to choose between Capacity, Access time and Cost. **Memory Hierarchy, Types**



Memory Type	Category	Erasure	Writing Mechanism	Volatility
Random Access Memory (RAM)	Read & Write	Electircally (byte level)	Electrically	Volatile
Read-only Memory (ROM)	Read only	✕	Mask	Non-Volatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read (Mostly)	UV Light		
Electronically Erasable PROM (EEPROM)		Electronically (byte level)		
Flash Memory		Electronically (block level)		

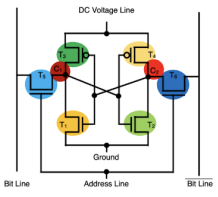
2. **Registers** (in processor) and **Cache** (around processor) are transparent to programmers. Cache stores KBs (L1) and MBs (L2, L3). **Write through**: update cache and all the copies in lower levels of memory. **Write back**: update cache copy, replace blocks in lower levels of memory.

3. **Cache Misses**: **Compulsory** (always occur), **Capacity** (not enough memory), **Conflict** (wrong placement strategy) and **Coherency**: (data was updated by a different processor). **Hit rate** ($h = \frac{\text{Number of times the words are in cache}}{\text{Total number of memory references}}$). **Miss rate** $= 1 - h$.

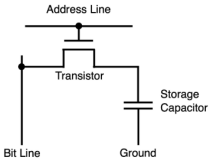
4. **Average memory access time** = Hit time + (Miss Rate × Miss Penalty) where Hit time is the time to hit the memory if in cache, miss penalty denotes time to replace block of memory.

5. Cache size and hit rate are positively correlated (especially L1). Will never get to hit rate of 1.

6. **Main Memory** (RAM): Static RAM (**SRAM**) holds data when power is supplied
   - Holds data whilst power is supplied
   - Usually uses 4 cross-coupled transition:
     - Forms a stable logic state
     - When storing logical 1, C1 is high and C2 is low (T1 and T4 off, T2 and T3 on)
     - When storing logical 0, C1 is low and C2 is high (T1 and T4 on, T2 and T3 off)
     - Address line controls two transistors (T5 and T6), where both transistors being on permits read and write

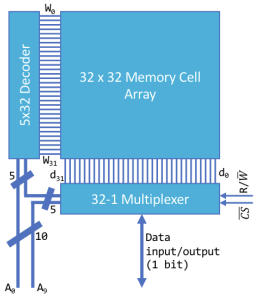


   while **Dynamic** RAM (**DRAM**) is stored as charge in a capacitor (can work in absence of charge)
   - Stores the data as charge in a capacitor
     - Discharges over time → needs to be refreshed periodically
     - Therefore, presence or absence of a charge represents a 0 or a 1
   - Write → Voltage applied to Bit Line
   - Read process is more complex:
     - Sense amplifier detects if 1 or 0, which discharges capacitor
     - Capacitor is restored and refreshed to original state



7. **DRAM** is a lot **cheaper** than **SRAM**, so often used for main memory (Synchronous *SDRAM*, Rambus *RDRAM*, DDR, Cache *CDRAM*).

8. Address Decoding Space: easier to split address into 2 equal parts: row and column address, which minimised space required. Select bits by selecting row/column. 16x8 (16 words each of 8 bits). For 128 of those cells need: 4 address inputs ($\log_2$ words) and 8 data lines(word size), so Use of space vs. number of pins. 1024 cells are either 128x8 (7 address + 8 data = 15 IO pins) or 1024x1 (10 address pins + 1 data pin = 11 IO pins)
   - Often more effective to split the address into 2 equal parts
     - 1 part for the row address
     - 1 part for the column address
   - Minimises the space required for address decoding
   - Maximises space used for memory cells



9. Thermal/Electrical component/transmission circuit noise may affect the data stored, so fix using: if prob. error low, then consecutive err. lower, so either send data twice (expensive) or send parity bits as a summary (to check whether the data is correct) (single transmission, **cheap**)

10. **Parity**: make number of logic 1's even or odd. Either even (start at odd) or odd (start at even) parity (adding single new bit), when reach 1: move to other state, when 0: stay need **dummy bit** (0 in even, 1 in odd parity), repeat and include the parity of the result.

11. **Burst errors** occur in groups, so can use **checksum**: create a collection of parity bits for each part of the message, but wouldn't work if more than 2 errors per block or an even number of errors in a bit-column (because would cancel out the parity check). **UNDERSTAND THIS and Error Correction Code ECC**

I/O

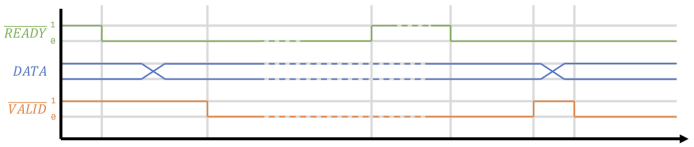
1. **Memory mapped** I/O: Connect components to **address bus** and assign each one a memory address (potentially ≥ 1), so CPU can read/write to those components like usual memory (operations **go through CPU**).
   + easy to implement, less internal CPU logic, can use normal memory instructions.
   - have to reserve memory for IO, small word space CPUs may suffer.

2. **Direct memory access** Controller **(DMAC)** handles IO **(instead of CPU)**. IO sends request to DMAC, which **forwards** it to CPU. CPU initialises and enables DMAC with (I or O?, **start address**, number of words to transfer (count register)), DMAC requests use of busses, CPU sends DMA Acknowledge when ready to surrender busses, which DMAC forwards to IO.

3. **Detached** DMA: DMAC is connected to main bus - simple but inefficient (2 bus cycles per word).
   Integrated DMA: separate DMAC per (≥ 1) IO device - simple but needs more communication with CPU.
   Separate IO bus connected to DMAC: only transfer data to/from main memory, so faster, but more complex and needs more hardware.

4. **Cycle stealing** mode: DMAC uses system busses **when free** (cpu isn't accessing memory), but can miss-detect a clear bus and be interrupted.
   Burst mode: DMAC requests and **locks** CPU out of system busses for a period of time or until completed. CPU can override.

5. **Polling**: ask "are we there yet?", if "yes" - terminate, else 1) wait for a bit and ask again (simple, effective). 2) **"Busy Wait" Polling**: ask again immediately (wastes CPU time and power). 3) **Polling** interleaved with **other tasks**: do something else, and ask again (bad for real-time task like flights).



   CB1: ready control line PRINTER_READY
   CB2: valid control line DATA_VALID !!

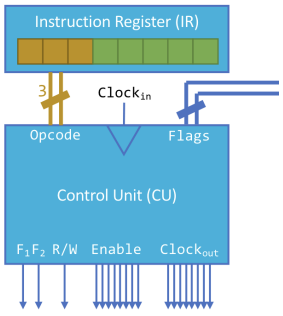
6. **Interrupts**: don't want to wait to complete a task, so complete current instruction, push **Program code** onto a stack, then **Status registers**, run interrupt code, load everything back and continue. Can be nested, using **interrupt request** (IRQ) pin, but **Non-maskable interrupt** (NMI) pin can disable interrupt of that interrupt. Fast response, no wasted cpu time/power, still controlled by cpu, very complex.

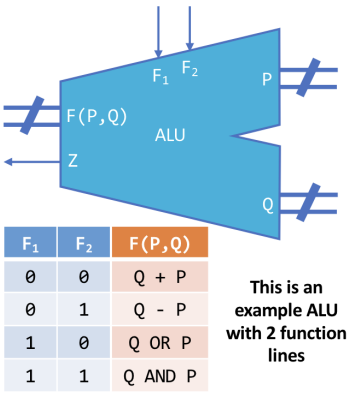
7. Overview:
   • **Memory-mapped I/O**: Devices can be accessed like main memory, but only at special address locations.
   • **Polled I/O**: Scheduling input and output, where the CPU repeatedly checks if this is necessary
   • **Handshaking**: Used to coordinate CPU and I/O devices to synchronise the transfer of data
   • **Interrupts**: Avoids polled I/O by diverting the CPU to a special I/O routine
   • **DMA**: Separate controller can be used instead of the CPU to transfer I/O data into/from memory

Microprocessor

1. Every processor has: data registers, address registers, connecting bus, program counter register (**PC**), instruction register (**IR**), Control Unit (**CU**) that enables/disables components based on instructions in IR, Arithmetic & Logic Unit (ALU) that calculates.

2. **Control Unit** (CU) decodes and executes instructions from IR (opcode-instruction and operand-parameter). Contains opcode bus, line for master clock, flags from CCR, enable and clock line for each component, function lines for ALU.





F_1	F_2	F(P,Q)
0	0	Q + P
0	1	Q - P
1	0	Q OR P
1	1	Q AND P

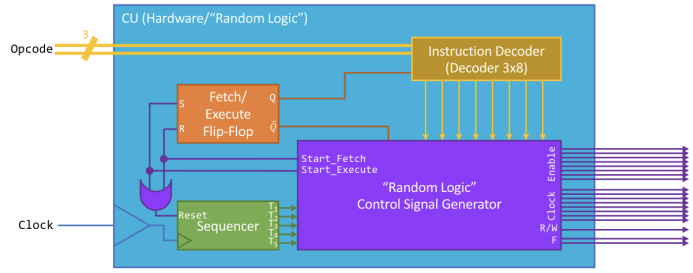
This is an example ALU with 2 function lines

3. **Arithmetic & Logic Unit** (ALU) performs maths and logical calculations. Inputs: 2 connections to the bus for data, multiple function lines to get current operation. Outputs: result transferred to bus through 1 connection, a carry line (for overflow) to CCR.

4. Turing-complete: can solve any problem ignoring runtime/memory etc. Examples: `subleq` (subtract and branch if $\leq$), powerpoint).

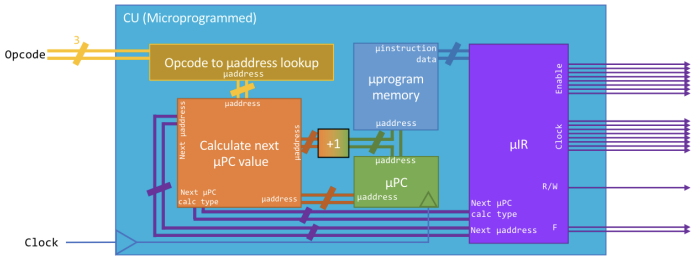
Micro and Macro Instructions

1. **RTL** - theoretical way to represent micro-instructions
2. Faster clock → faster execution → faster processor (each instruction needs clock pulses to execute)
3. Hardware CU design: use logic circuits, boolean logic, called **Random logic**. Very **fast**, and power efficient, but very **complex**, difficult to design and **not flexible** (hard to add new instructions).
4. **Sequencer** is part of hardware CU design, activates a line per clock cycle in order, helps determine how far through an instruction we are.



5. Micro programmed CU design: each machine instruction mapped to micro instruction, those mappings are stored in **ROM** (microprogram memory), called μprogram, where μ is "Mu" and stands for "micro". **Easier** to design and implement, more **flexible**, easy to extend, but **slower**.

   Microinstructions → holds CU ouput values and other files required for control flow of μprogram
   μaddress → location within μprogram memory
   μPC→ CU's internal program counter
   μIR→ CU's internal μinstruction register
   Microprogram routine *to* describes how to generate CU outputs for a macroinstruction, made out of μinstructions and stored in μprogram memory.



RISC/CISC processors

1. **RISC**: try to compute with as few instructions as possible, use less hardware and computation. Few essential instructions which all take up same space, **hardware** based CU, simplifies Fetch-decode-execute. Everything built in hardware: **faster**, energy efficient, but **complex** to build and design, **hard to extend**.
2. **CISC**: combine instructions to make things simpler, no fixed-width instruction set, so many **more instructions** (similar to **high-level** languages). **Microprogram**-based CU (x86, x64). **Simple** to design and implement, **easy to extend** instructions, but **slower** and needs more power.
3. **Thread** is an execution context for a processor, including a stream of instructions. **Parallelism**: use multiple threads, can be done within a single core, **cheaper** to switch **threads** than cores or processes. No limitations on number of threads requested/time held. Sometimes called **logical core**.
4. **Problems** with **Multithreading**:
   Race Conditions: threads update same data at the same time (fix by utilising a **lock**).
   Deadlock: threads lock each other out (fix by being careful threads don't take all the locks)
   Starvation: low priority threads are not scheduled, so might never be run (fix by switching threads explicitly).
5. **Core** is a physical processor that can run code independently, but all cores are on the same physical die and have a small level of L1 cache. **Multicore** is like multithreading, but each core can independently run threads (called **hyperthreading**).

Considerations of processor design

Engineering Limitations

1. To build transistors, need to manipulate manipulate materials at large scales. Can use lasers to pinpoint silicon material and build circuit components. Size of components limited to diameter of laser (Amdahl's Law)
2. **Interconnects**: copper in the interconnects can be fast or dense. Small cross-section: increase in resistance, Large cross-section: larger capacitance on neighbouring wires, so use larger number of stacks with different size wires.
3. **Transistors**: limited by width of the gate, shrinking makes it easy to be impacted by noise, and more noise means more variation in performance.

Energy Limitations

1. To keep chips running, their power density must be limited, because given enough energy atoms start acting quantumly, so as components get smaller, have to "shut off" more parts of the chip, otherwise lower clock speed and utilise more of the chip at slower speed.

Parallelisation Limitations

1. Theoretical speedup of the whole task $S_{\text{latency}}(s) = \frac{1}{(1-p)+p/s}$, where p is proportion of execution time that benefits and s is speedup of part of the program that benefits. Only theoretical limit.
2. **3D**: if algorithm takes n^2 iterations in 1D, n iterations in 2D, then it will take $n^{3/4}$ in 3D. Theoretically mild improvement. Harder to produce 3D cores.
3. **Quantum** are too expensive to design and maintain, too many faults, only faster for **some** algorithms.

Future of processors

1. **High bandwidth memory**: with large data, data transfer speed is bottleneck. **MCDRAM**: multi-channel DRAM: allows multiple ocnnectins between different sections of chip. **HBM2**: memory places onto chip as series of stacks, usd for high performance gpu's.
2. **Specialist Cores**: cores specialised to perform specific tasks (like integrated graphs on CPUs)
3. **GPUs**: large quantity of small, simplistic cores (huge parallelism per device). Cores are slower, but swapping threads hides it, consumes less power than CPU.
4. **Field Programmable Gate Arrays** (FPGAs) are a collection of components that can be turned on and off using transistors, can be hugely power efficient and fast, but takes (hours+) to compile programs efficiently. Used at CERN
5. **Heterogeneous computing**: run code across different specialised machines, Homogeneous: across similar, specialised machines, but need libraries/languages to handle.