

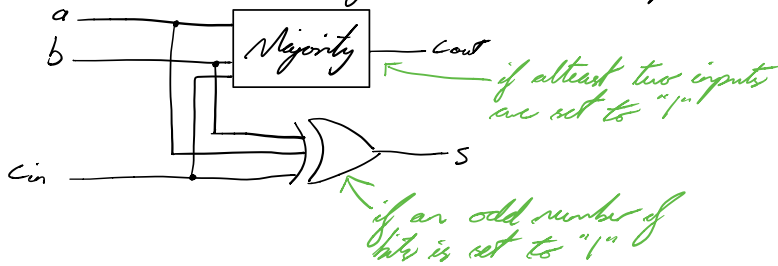
Combinational Logic Building Blocks

• Binary Addition

$$\begin{array}{r} 6 \quad 110 \\ 3 \quad 011 \\ \hline 1001 \end{array}$$

• Adder

- Counts the number of "1" bits on its input



$c[i]$	$b[i]$	$a[i]$	$c[i+1]$	$s[i]$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

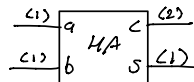
```

module FullAdder2(a, b, cin, cout, s);
    input a, b, cin;
    output cout, s;
    wire s = a ^ b ^ cin;
    wire cout = (a & b) | (a & cin) | (b & cin);
endmodule
    
```

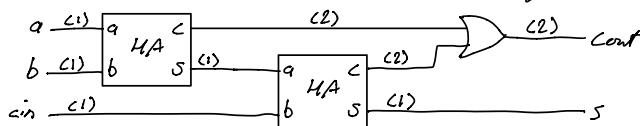
← carry and sum

• Half Adder

Outputs "sum" and "carry over"



- Can combine half adders to make full adder



- Can combine multiple Full Adders to add multi-bit buses, "cin" carries carry over from previous adder

• Negative Integers (2's Complement)

$$0011 = 3$$

1) Flip each bit 1100

2) Add one 1101

$$1101 = -3$$

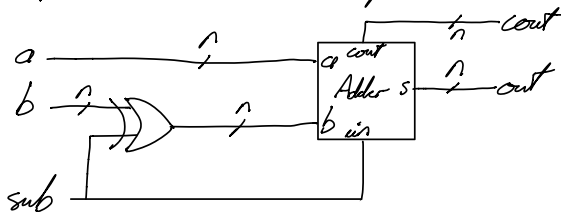
(same for negative to positive)

• Represent negative number as: $-x = 2^n - x$

$$x + (-y) = x + (2^n - y), n \text{ bit number}$$

$$\begin{aligned} 4 - 3 &= 4 + (-3) \\ &= 4 + (16 - 3) \text{ mod } 16 \\ &= 4 + (15 - 3 + 1) \\ &= 0100 + (1111 - 0011) + 0001 \\ &= 0100 + 1100 + 0001 \text{ complements} \\ &= 0001 \end{aligned}$$

• Adder/Subtractor (2's Complement)



• Overflow
• Check that the carry in does not equal the carry out

— Datapath State Machines

• Counter

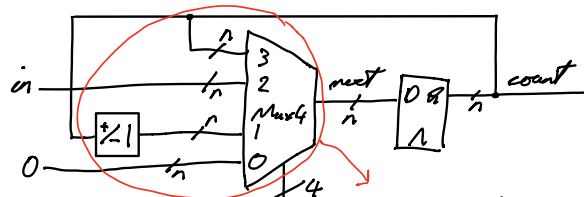
- Increment output by 1 on each clk input
- A "case x" statement allows for don't cares

Ex: case x (a)

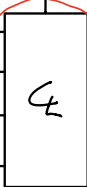
$$4'b000x: b = 4'b0010$$

if "a" is 0001 or 0000, b is set to 0010

• Up/Down Counter



rst
up
down
load

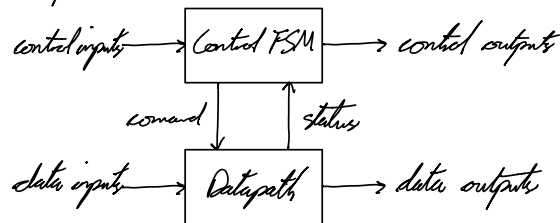


counter logic

Up/Down, load, rst in one-hot

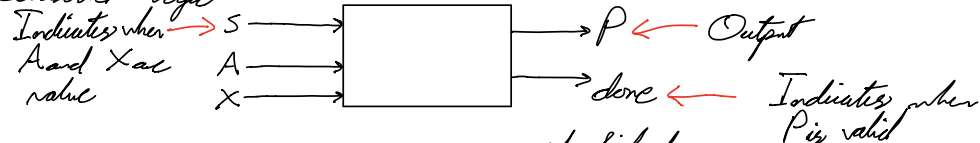
State	Next State				
	rst	up	down	load	else
q	0	q + 1	q - 1	input	q

- Datapath/Control Partitioning
- Control inputs/outputs and data inputs/outputs wiring

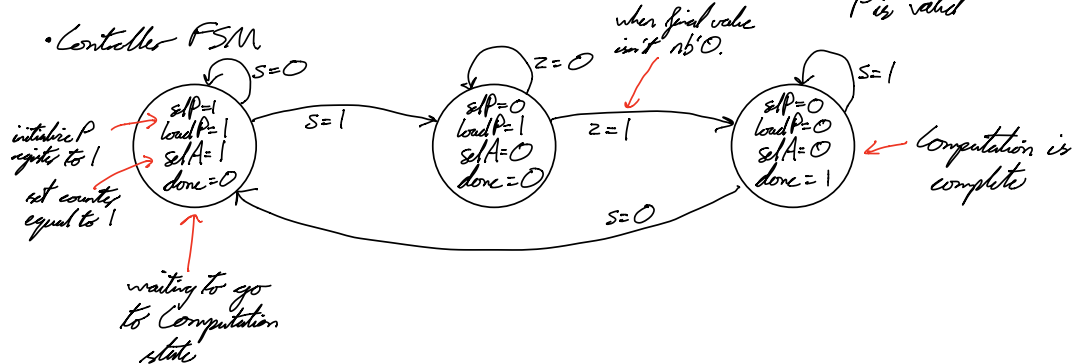


Ex: Compute X^A

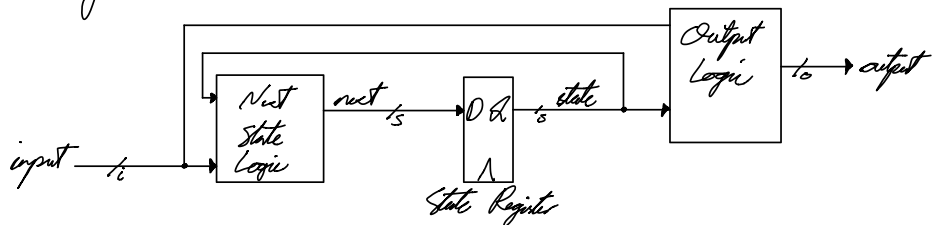
- Controller logic



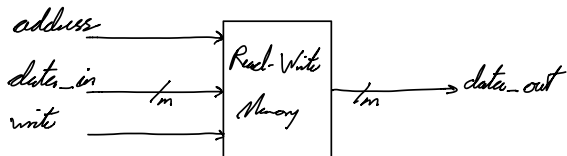
- Controller FSM



- FSM logic



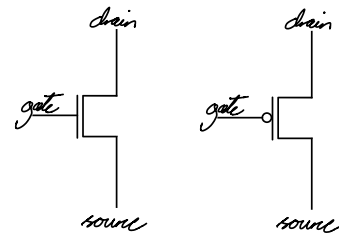
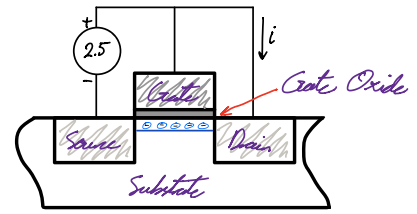
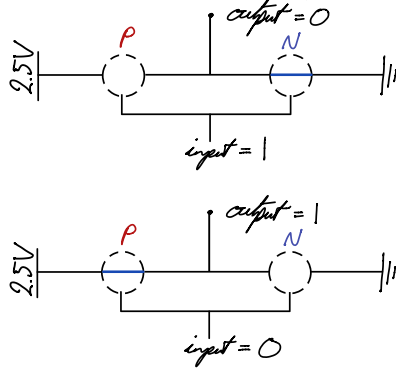
RAM (Random Access Memory)



- Address chooses latch to write to, simultaneously choosing latch to read from

Complementary Metal-Oxide-Semiconductor

- Amount of current flowing from *source* to *drain* depends on voltage applied to gate
- A *wire* is effectively created when *gate* receives *voltage*
- Can create **NOT** Gate from **P/N** FETs



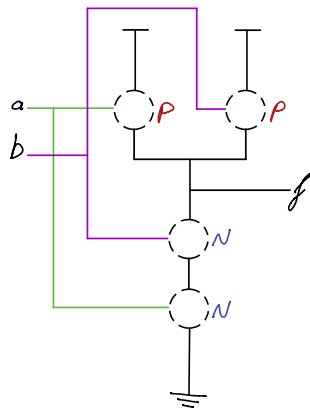
NFET

PFET

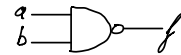
PFET has opposite function to NFET

CMOS Logic Gate Design

- Can build a gate to realise $f(x)$ using **NFET** network that is closed when $f(x)$ is *true* and **PFET** network that is closed when $f(x)$ is *false*
- Ex: $f(a,b) = a \cdot b$ NAND Gate

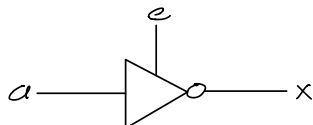


a	b	f
0	0	1
0	1	1
1	0	1
1	1	0



Tri-State Inverter

- Disconnects output when $e = 0$
- Acts like normal inverter when $e = 1$

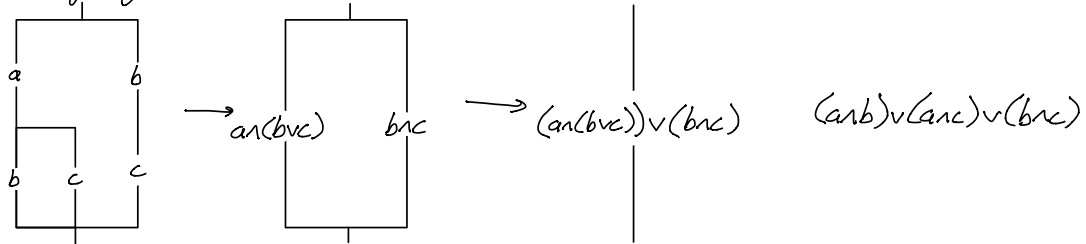


e	a	x
0	0	z
0	1	z
1	0	1
1	1	0

assign $x = e ? \sim a : 1'bz;$

- Series-Parallel Networks
 - Networks of **switches** in series or parallel
 - Can be simplified using Boolean logic

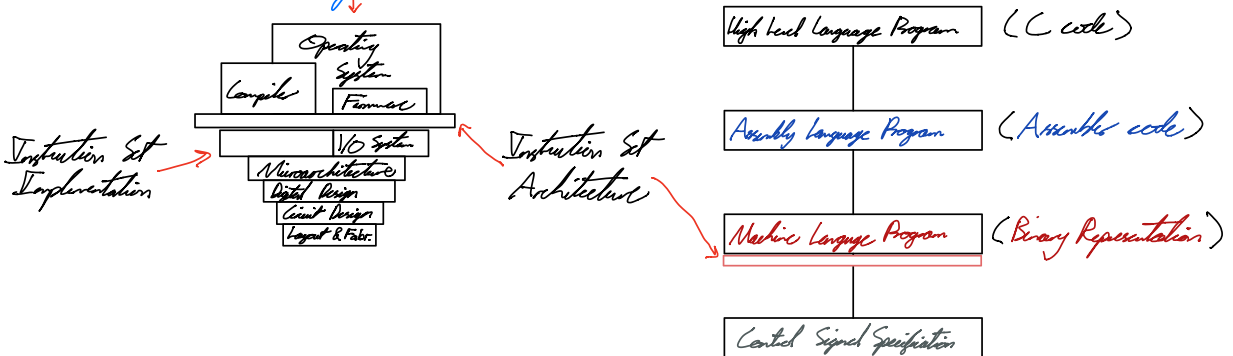
Ex: Majority Function



- Uses fewer transistors than distinct AND and OR gates
- Larger circuits must be decomposed into multiple simpler gates (> 6 inputs)

ARM Assembly Programming

- Putting code and processing is the result of multiple **transformations** which **exist** between **layers**



ARM Syntax

C code: $f = (g+h) - (i+j)$

ARM Assembly:
 ADD a, g, h
 ADD b, i, j
 SUB f, a, b

- Data Retrieved
LDR $R5, [R2, \#8]$, $R2 = 100$
 $\rightarrow$ copies data from address 108 to R5

- Data Write
STR $R5, [R2, \#8]$, $R2 = 100$
 $\rightarrow$ copies data from R5 into address 108

• Encoding Instructions

ADD r3, r3, #4 → r3 = r3 + 4

Condition	ALU	Immediate	ADD	S	Rn	Rd	Operand2
↓	↓	↓	↓	↓	↓	↓	↓
E	I	I	4	0	3	3	4
↓	↓	↓	↓	↓	↓	↓	↓
1110	00	1	0100	0	0011	0011	{960, 36100}

• Instructions

MVN moves register → register
 MOV moves constant → register
 ADD adds 2 registers → register
 SUB subtracts 2 registers → register

• Operands

LSL shifts bits Left by constant
 LSR shifts bits Right by constant

• Branch Instructions

CMP does register = register
 B unconditionally branch
 BNE branch if CMP is not equal
 BEQ branch if CMP is equal
 XY multiple "do if" instructions

"Else:" location pointed to by branch

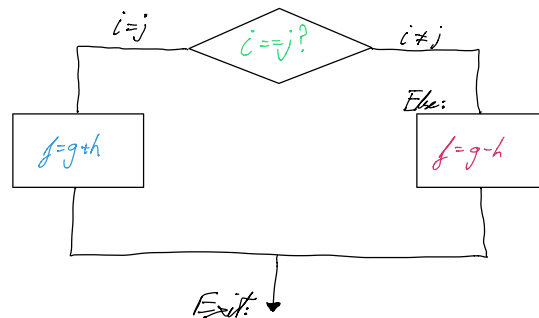
• Branch Target Encoding

$$PC_{\text{target}} = PC_{\text{branch}} + 8 + (4 \cdot \text{"Target Address"})$$

↑ ↑ ↑
 Target Program Current Program Branch Offset
 Count Count times 4 bytes

CMP R3, R4
 BNE Else
 ADD r0, r1, r2
 B Exit
 Else: SUB r0, r1, r2
 Exit:

→ goto "Else" if i ≠ j
 → goto "Exit"



C Code Compiler

• Function Calls in ARM

- **Extern** linkage means function is visible to external files
- **Static** linkage means function can only be called by function in current file

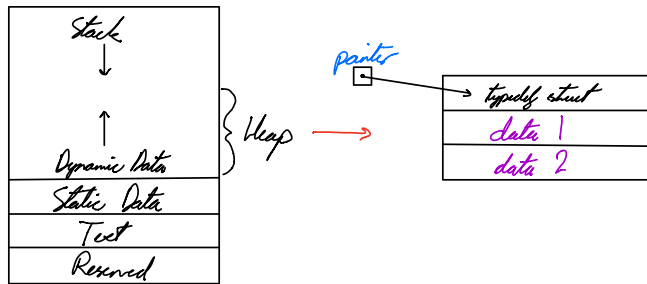
• Operation of a Function Call

1. Put parameters where function can find them
 - ARM uses R0 to R3 for first four function inputs
 - Ex: MOV R0, #8
 2. Transfer control to the function
 - **BL** (branch and link instructions) to transfer control to subroutine
 - remembers where function was called from
 - ARM uses R14 to store program counter of instruction to return to, the **link register**
 - Ex: BL ProcedureAddress
 - $R14 = PC + 4$; $PC = \text{ProcedureAddress}$;
 3. Acquire storage for function
 - Data is saved in **Stack** - located in memory - starting with higher addresses
 - ARM: Callee **pushes** registers to stack, then **pops** registers from stack
 - Ex: STR r6, [sp, #8]
 - Move **stack pointer** down to allocate space for this step
 4. Perform task
 - ARM instructions to execute function tasks
 5. Put result where calling program can access it
 - ARM uses R0 to store **return value**
 - Write contents of memory back into registers
 - Ex: LDR r5, [sp, #4]
 - Move stack pointer back up to allow for future use
 6. Return control to point of origin
 - ARM uses R15 to store **program counter**
 - Copy LR into PC (R14 into R15)
- To ensure that data written into registers isn't lost during function:
- Data can be duplicated just before function call, or immediately afterwards (before or after BL - **caller save** or **callee save**)
 - ARM uses callee save
- ARM Calling Conventions
- Return result in r0
 - r0-r3, r12 are not saved to stack by callee
 - r4-r11 are saved to stack by callee during function call
 - **r0-r3** take argument 1 to 4 respectively; Ex: r0 = first argument
- If there are **more than 4 arguments**, extra arguments are saved to stack by caller
- Ex: MOV r4, #30 → saves 30 to stack
STR r4, [sp] .pointer = 0
- Then load from stack prior to use of argument

- ARM has special loading instructions for reading and writing a **byte**
 - LDRB** $R_n, [R_d, \#x] \rightarrow$ "x" extends upper 24 bits of hexadecimal value at $MEM[R_d]$ and assigns that to R_n
 - STRB** $R_n, [R_d, \#0] \rightarrow$ assigns $MEM[R_d] = R_n$
 - LDRSB** $R_n, [R_d]$ $\rightarrow$ sign extends upper 24 bits of hexadecimal value at $MEM[R_d]$ and assigns that to R_n

Heap

- Structures are created and removed from the heap
- Each data is stored one byte after previous, and all **data** is stored after the **pointer**

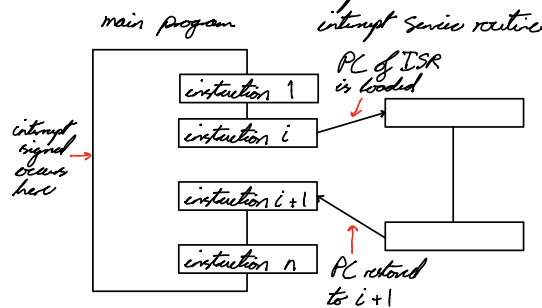


Connecting I/O to the Computer

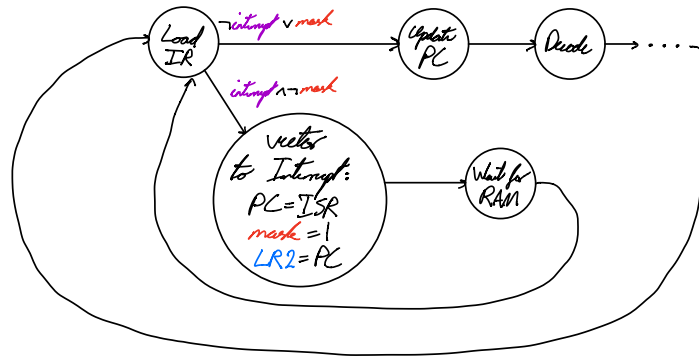
- Memory Mapped Registers are located within I/O device and have their **data** transferred to Memory or Processor using **LDR** and **STR**
- Duration between inputs (I/O changes) can be used for:
 - 1) Polling: do nothing
 - 2) Interrupts: allow background tasks to occur

Interrupt

- A function can be called in response to an interrupt signal (Interrupt Service Routine)



- During the "load instruction" state, the **interrupt signal** diverts the state machine to set the program counter to the ISR
- A signal called **mask** is set to 1 indicating that the interrupt has occurred
- The program counter is saved to a secondary **link register** for restoration to main program
- Return to main using an instruction that sets the **next** program counter, restores the contents of the status register, and **unmask** interrupts



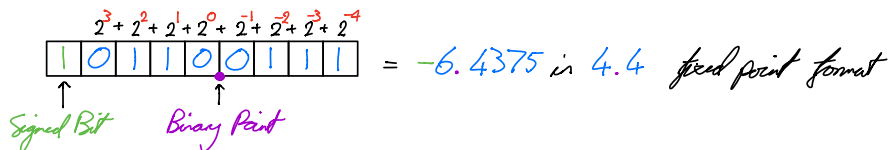
- ARM Exceptions
 - Processing anomalies such as "Reset" or other Interrupts must be handled without crashing
 - These Processor Modes each have their own stack pointer and link register that are set by the hardware and restored by the software
 - ARM stores address for vectors in stack at predefined addresses
- Interrupt Controller
 - Allows for software control of interrupts
 - Specific registers responsible for:
 - enabling interrupts on this interface: CPU Interface Control Register
 - sending highest priority interrupts to CPU: Interrupt Priority Mask Register
 - holding ID of peripheral causing current interrupt: Interrupt Acknowledge Register
 - writing ID of interrupt to be cleared: End of Interrupt Register

———— Multitasking

- Operating system will alternate between running different programs (Threads) using interrupts
- A Thread is a location in memory and set of instructions (stack pointer and program counter)
- Multithreading allows hardware to run multiple threads simultaneously
- Thread Count is number of threads that can be run simultaneously

———— Fixed Point

- Representation of Fraction numbers



- Error; Closest representation of number with defined fixed point format will have an error value
 - Define $R(x)$ as binary number used to represent x where $R(x)$ is chosen to minimize error
 - Define $V(y)$ as decimal value of y

Absolute Error: $e_a = |V(R(x)) - x|$

Relative Error: $e_r = \left| \frac{V(R(x)) - x}{x} \right|$

- Registers
 - Floating point coprocessor provides 32-bit registers: **S0-S31**
 - Registers **S(i) + S(i+1)** can be referred as 64-bit register **D(i)**

Microarchitecture

- Interface between software and hardware, the way in which a given ISA is implemented in a particular processor

Performance

- Many ways of quantifying performance (#instructions/second, etc.)
- Basic Response Time metric:

$$\text{Performance}(X) = \frac{1}{\text{Execution Time}(X)}$$

- Processor Performance Equation:

$$\frac{\text{CPU Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Clock Cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Clock Cycle}}$$

- Cycles Per Instruction:

$$\text{CPI} = \sum_{j=1}^n \text{CPI}_j \cdot F_j, \text{ where } F_j \text{ is the normalized instruction frequency for instructions of type } j$$

Pipelining

- Rather than running through each instruction following the completion of the previous, **overlap** multiple instructions to **decrease program time**
- Add **registers** between each **stage** of the instruction pipeline allowing each stage to **process a different instruction**

Instruction **Fetch** — instruction fetched from memory

Instruction **Decode** — registers used in instruction are decoded

EXecute — shift and ALU operation

MEMory access — data access to/from memory

Write Back — write value into destination register

- Hazards

Structural Hazards

- Two or more instructions wanting to use the same hardware simultaneously

- Solutions:

- **Stall**: one instruction goes first
- **Replicate Hardware**

Data Hazards

- Later instruction requiring data from preceding instruction that has not finished

- Solutions:

- **Stall**: **delay** until data has been written/processed (**NOP** instruction(s))
- **Bypass**: send processed data to hardware used by next instruction

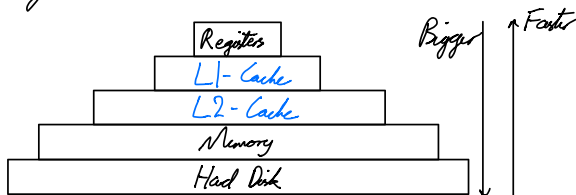
- Control Hazards
 - Branch instructions only change PC after finishing, subsequent instructions will have started by that point
 - Solutions:
 - Stall: delay until branch has executed (NOP instruction(s))
 - Predict "Not Taken": always assume branch is not taken; if incorrect — NOP the 2 subsequent instructions
 - Delayed Branches: use a branch that executes after N instructions

RAM

- Form of memory that can be accessed in any order
- SRAM: 6 transistors to store a single bit
 - Smaller SRAM is faster
- DRAM: 1 transistor and 1 capacitor to store a single bit
 - Cheaper, longer access time than SRAM
- Flash Memory
 - A Floating Gate MOSFET can have its threshold voltage (V_t) adjusted
 - Arrange these transistors into NOR and NAND which store memory
 - NAND has higher density, NOR is quicker
 - Higher density / lower access speed than DRAM

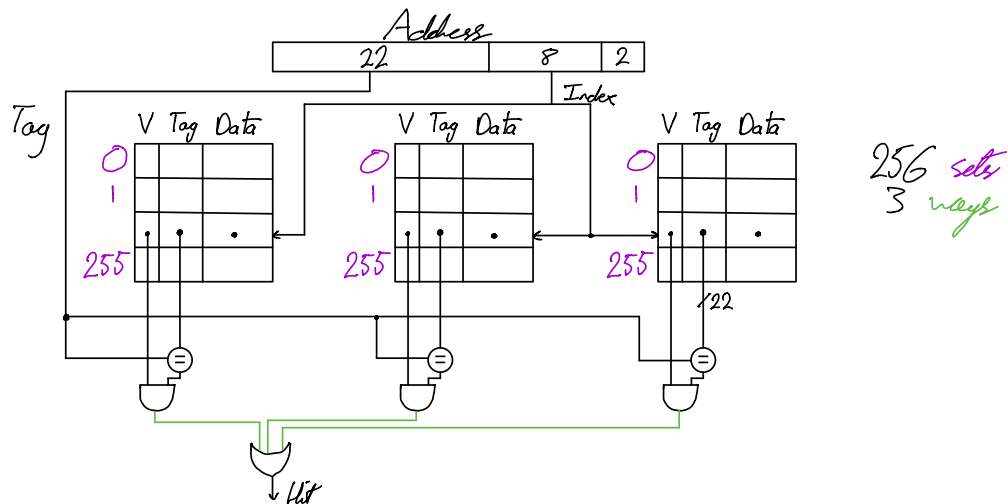
Caches

- Stages between hard drive and CPU registers are implemented to allow for faster data access



- Temporal Locality: a location that is read, then accessed again by program
- Spatial Locality: nearby locations that are being accessed by program
- Program Controlled Cache
- Hardware Controlled Cache
 - Hardware transfers certain data to cache when program accesses data
 - Cache Miss occurs when necessary data was not moved to cache; processor will proceed to load data into cache at a certain address (tag)
- Cache Blocks are chunks of data that are moved to cache
 - Predetermined size, begin at address 0x0

- When Cache is full, "evict" block that has been unused for the longest period of time
- **Reuse Distance** of a block is number of unique blocks accessed between accesses of block in question
- Direct Mapped Cache
 - Mapping memory addresses to cache addresses predeterminedly
 - Only need to compare tags
- Fully Associative Cache
 - A new line can be placed anywhere in cache
 - Each retrieval must be compared with desired memory value
- Set Associative Cache
 - Memory blocks are mapped to sets of cache locations and then placed into any of those locations
 - Each position in cache is defined by a **set** and **way**



- Cache Size:
 - tag bits = address bits - index bits - valid bits - block offset bits
 - Total Bits = sets $\cdot$ (tag bits + valid bits + data size bits)

- Writing to Memory
 - **Write-Through** will store to both the cache and memory
 - **Write-Back** will only write to cache
 - make the **dirty bit** of the cache location
 - **Write Miss** occurs if value to be read from cache, to memory, is not found
- Multilevel Caches
 - Can decrease **Miss Penalty** by adding another cache

Virtual Memory

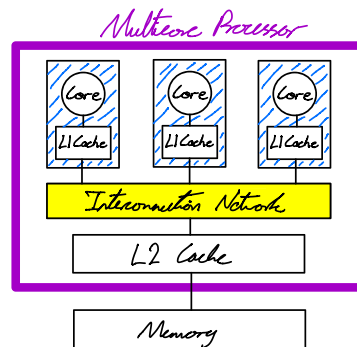
- Divide memory into **pages** so that holes in memory (created from closed programs) can be used
- **Fragmented** memory has holes everywhere
- Store extra program data on hard drive
- **Virtual Address** points to data in memory or hard drive depending on if there was overflow at time of storage
 - Allows each program to "see" memory differently

Page Table

- Translates virtual address to physical address
- Each program gets own page table
- Usually separate into **books** of page tables (decreases size of memory usage)
- **Translation Lookaside Buffer** is a special cache for page table addresses

Parallel Computing (Multiple Cores)

- Cores can operate using independent sets of instructions (**MIMD**) or a single set of instructions across all cores (**SIMD**)



- Program can send function call to different core (using same memory for data) and then wait for return value
 - Requires programs that are coded for multiple processors or **threads** that can do this **actively**
- **Threads** are programmed "jumps" to other cores
 - A thread may be forced to pause while another is altering memory, this is done with a programmed **lock**
 - locks diminish parallel processing potential

Amdahl's Law

- Some parts of processing can never be enhanced, leading to idea that more cores is not always better

$$\text{Speedup}_{\text{Overall}} = \frac{\text{ExecTime}_{\text{old}}}{\text{ExecTime}_{\text{New}}} = \left[(1 - \text{Fraction}_{\text{Enhanced}}) + \frac{\text{Fraction}_{\text{Enhanced}}}{\text{Speedup}_{\text{Enhanced}}} \right]^{-1}$$

Cache Coherence

- Every time a processor updates its L1 cache, it invalidates that cache value in all other caches, and then stores the value to its cache