Logic Basics

Ohm's Law: $R = \frac{U}{I}$

U = electrical potential (volts, V) R = resistance (ohms, ω) I = current, voltage drop (amps, A)

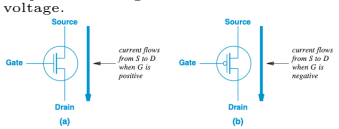
Transistors

MOSFETs have 4 components: Source, Gate, Drain, and Base

PNP/NMOS: Is on when gate is positive. No circle.

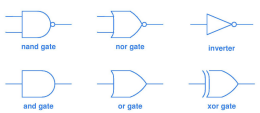
NPN/PMOS: Is on when gate is negative. Has circle.

Generally, transistors are used to pull the output to either a positive voltage, or a zero voltage (1 or 0, on or off). If output is not pulled to one of these, the output is floating and is indeterminate in voltage.



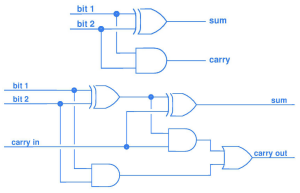
Logic Circuits

Symbols (bottom have 2 inputs)



Adders

Left: Half-adder. Right: Full-adder.

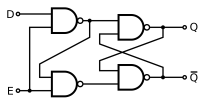


Unlike half-adders, full-adders can receive carry-in bits. To add more bits, chain multiple full-adders together. If final carry at end is 1, there's an overflow error.

Latches & Flip-flops

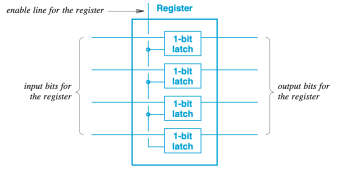
Flip-flop Every time the input switches from 0 to 1, the output switches to the opposite.

Gated D-latch



E/C	D	Q	$\overline{Q}$	
0	X	Q_{prev}	$\overline{Q}_{prev}$	No change
1	0	0	1	Reset
1	1	1	0	Set

- Q - The stored bit
- D - Data/bit to write to Q
- E - The enabler (Must be 1 to enable writing, otherwise nothing changes)
- Operate on the principle of propagation delay.
- Stacking many of them can be used to create a register:



Counters

For each transition from **low to high**, the counter increments the binary output by 1. (Counting the transition from high to low does the same thing, but the lecture used rising-edge counters).

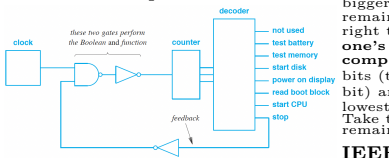
Propagation Delay

The rate at which a transistor switches. Typical 100 ps (picoseconds) **Clock Frequency:** $= 1/f$

Decoders

Take in an n-bit number, and turn on one of 2^n outputs.

Feedback Loops



- Counter increments each time the clock changes from 0 to 1
- Decoder moves to the next instruction each time
- Stop sends a signal back to the and gate which stops counter

Multiplexer / Demultiplexer

Multiplexer turns n signals into a single signal. Chooses which signal to let through. **Demultiplexer** turns a single signal into n signals. The single signal chooses which signal to output.

Software vs Hardware Design

Unlike software, which uses iteration, hardware uses replication. The advantages of replication are increased elegance, higher speed, and increased reliability. Hardware also uses gate minimization, abstraction and power optimization.

Fixed & Programmable Logic

Fixed logic circuits: Pre-determined function. **Programmable logic:** FPGAs (reprogrammable, but still a significant cost to switching functions). **Stored program and re-programmable circuits:** Your computer right now.

Data Encoding

1 Byte = 8 bits. 1 **Byte** encodes a character, integer, or pointer. 1 **Word** is n bytes, determined by the architecture.

Converting between bases

Base 10 to Base N Divide decimal # by # of new base. Take remainder as rightmost digit. Divide quotient of previous divide by new base. Repeat until quotient is zero. **Base N to Base 10** Take each column position of each digit, zero indexed, as n. For each column, do $c \cdot b^n$, where c is the value of the column, and b is the base value in base 10.

Important bases table

Hex	Bin	Dec	Hex	Bin	Dec
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

Important Bytes Table

1 Bit	8 Bits
	1 byte $= 2^3$ bits
1 KBit $= 10^3$ bits	1 Kb $= 2^{10}$ bytes
1 MBit $= 10^6$ bits	1 Mb $= 2^{20}$ bytes
1 MBit $= 10^9$ bits	1 Gb $= 2^{30}$ bytes

Fraction to Binary

1. Let x be the decimal part of our fraction.
2. Let b be our binary output.
3. Multiply x by 2.
4. If $x \geq 1$, subtract 1 from x and append 1 to b.
5. If $x < 1$, append 0 to b.
6. Goto step 3 until x! = 0.

Signed Integers

- **Two's complement:** Flip all bits, add 1. Range is $-(2^{n-1}), +(2^{n-1}-1)$
- **Sign-magnitude:** First bit is sign, rest is magnitude. Range is $-(2^{n-1}-1), +(2^{n-1}-1)$.
- **One's complement:** Flip all bits. Eg, +6 is 0110, so to get -6 go from 0110 → 1001 → 1010. This has 2 zero, positive and negative zero. Range is $-(2^{n-1}-1), +(2^{n-1}-1)$.

Cast from Ints

from smaller into to bigger ints sign-magnitude: Copy the MSB to the bigger ints MSB. Then take the remaining bits and fill them from the right to left.

one's complement and two's complement: Copy the Lowest order bits (the bits except the sign defining bit) and copy them to the other ints lowest order bits.

Take the MSB and make all the remaining bits in the new int the MSB.

IEEE-754 Floats

Can represent values from 2^{-126} to 2^{127} which is 10^{-38} to 10^{38} in decimal. If double precision is used then the range is 2^{-1022} to 2^{1023} which is 10^{-308} to 10^{308} in decimal. Separated into **sign, exponent, and mantissa**. Entire float is represented in binary, and the exponent is biased by $2^{b-1}-1$, where b is the number of exponent bits. Leading 1 is dropped from mantissa. To calculate from IEEE, do $1.(m_2) \times 2^{(e_2)}$. (Convert out of base 2 first). Range is approximately $2^{(-2^b)+2}, 2^{2^b-1}$, where b is the bits dedicated to the exponent, **minus 1**.

Special Values

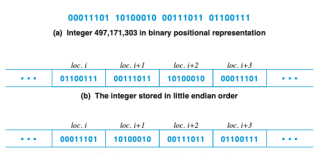
Exponent	Mantissa |	Value
all 1s	all 0s	$\pm\infty$
all 1s	not all 0s	NaN
all 0s	not all 0s	denormalized
all 0s	all 0s	± 0

Binary-coded decimals

Instead of representing decimals using a float, use an arbitrarily long string of bytes, usually as binary ints. Efficiency can be improved using packed BCDs, by putting 2 ints in one byte (each digit occupying a nibble). `0x2D` is used by the textbook to represent a negative, placed at the end of the BCD. For a packed BCD, drop the 2.

Endianess

Endian: The way in which words(group of bytes) are stored in memory. **Big Endian:** Most significant byte first. **Little Endian:** Least significant byte first. Little endian is the most common method of storing data in memory.



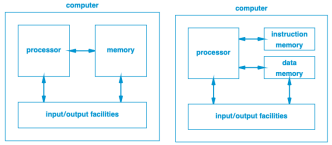
Types of architecture

Von Neumann architecture (left):

- Single memory block which contains both instructions and data.
- Offers complete flexibility: at any time, owner can change how much of the memory is devoted to programs and how much to data.
- More popular.
- Allows dynamically allocating memory between instructions and data.

Harvard architecture (right):

- 2 separate memory. One is used for instruction, one is used for data.
- Inflexible, as you cannot use part of the instructional memory to store data and vise versa.
- Less popular. Sometimes used in small embedded systems.



Von Neumann Bottleneck

On computers running Von Neumann Architecture, time spent accessing memory can limit performance. To avoid the bottleneck, designs are chosen where operands are moved to registers instead of system memory.

Types of processors

A processor is a digital device that can perform a computation involving multiple steps.

Categories based on logic:

- **Fixed logic**: Function fixed in hardware, performs a single task
- **Selectable logic**: Choose one of several fixed functions.
- **Parametrizable logic**: Accepts a set of parameters that control the computation of fixed functions.
- **Programmable logic**: list of instructions provided at runtime (you can code them)

Categories based on Complexity:

- **Co-processors**: Dedicated function. Usually performs a single task at high speed. Used in -> Floating point accelerator. Fixed/Selectable logic.
- **Microcontrollers**: Direct hardware control. Used in -> Elevator doors. Programmable logic.
- **Embedded System Processors**: real-time OS, dedicated hardware. usually more powerful than microcontrollers. Used in -> smart phone Programmable logic.
- **General-purpose Processors**: compatible for multiple systems. Used in -> CPU in a PC. Programmable logic.

Parts of a processor

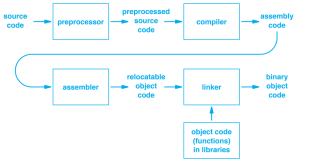
- **Controller**: Responsible for program execution. Steps through the program and coordinates the actions of all other units.
- **Arithmetic logic unit**: Performs all computational tasks. Performs one operation at a time according to controller.
- **Local storage (registers)**: Hold data values such as operands for arithmetic operations and the result.
- **Internal connections**: Transfers data values between units, like from local storage to the ALU. AKA data paths/Bus/Control lines
- **External interface**: Handles all communication between the processor and the rest of the computer system.

Fetch execute cycle

There is a **instruction pointer** which moves through the program performing every step. The cycle never ends while the system is running.

1. Fetch the next instruction
2. Decode the instruction and fetch operands from registers
3. Perform the arithmetic operation specified by the **opcode**
4. Perform memory read or write, if needed
5. Store the result back to the registers
6. go to next instruction, Repeat forever.

Program Translation



- **Preprocessor:** Expands macros, producing modified source program.
- **Compiler:** Translates it to assembly.
- **Assembler:** Translates it to relocatable object code which contains references to external library functions.
- **Linker:** Replaces external function references with its code.

CISC vs RISC

CISC

- Each instruction performs a complex operation
- Instructions may take multiple clock cycles
- Fewer instruction calls

RISC

- Each instruction performs a simple operation
- Instructions all take the same number of clock cycles
- Many instruction calls needed
- Allows for pipelining, as each part of the instruction takes the same amount of time

Pipelines

Allow for more than one instruction to be "processed" at the same time.

Generally 5 stages:

1. Fetch next instruction
2. decode & fetch operands
3. perform arithmetic operation
4. read or write memory
5. store result

Pipeline Stalls

Also known as hazards. 3 main types:

- **Data Hazards:** Waiting for data from an earlier instruction. Can be dealt with using data forwarding (allowing data to be used before it exits the pipeline), re-arranging instruction order.
- **Control Hazards:** Incorrect instruction is in the pipeline. Occurs during jump instructions/branching. Jumps are not executed until the fifth stage, so instructions directly after are fetched inside the pipeline. Can be dealt with using conditional branch prediction, flushing pipeline if prediction is wrong.
- **Structural Hazards:** Resource conflict (usually from external source) (eg somebody else is accessing the same register bank). Can be dealt with by loading data in parallel, eg using multiple banks.

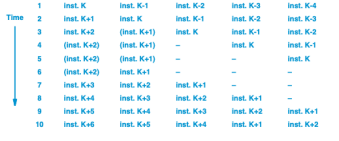


Figure 5.5 Illustration of a pipeline stall. Instruction K+1 cannot proceed until an operand from instruction K becomes available.

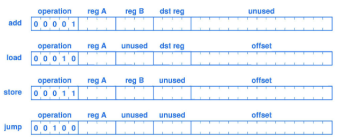
Branching

Moving the instruction pointer to a different location in program. Can be either absolute branch, or relative branch. Absolute branch prediction can be used to try to run code from a branch before the processor has the data needed to evaluate it, speeding up runtime.

Instruction Sets

Generally has the following parts: Operation number, registers, offset.

- **Opcode (operation code)**: Specifies the operation to be performed.
- **Registers**: Specifies the operands and the destination.
- **Offset**: Think of it like array indexes. Can be a signed integer to move backwords.



Design choices

Encoding length

Variable-length encoding can improve instruction density, but **fixed-length** instructions are simpler to implement in hardware, and are thus more performant. Unused bits are ignored by the instruction.

Offsets are used to encode immediate values (generally used for jumping).

Number of Operands

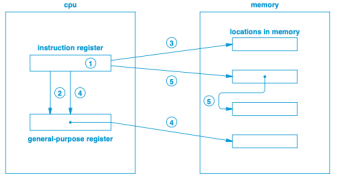
- **Zero operands:** Stack architecture, using push and pop. All operands are implicit.
- **One operand:** Implicit destination (usually a special accumulator register)
- **Two operands:** Specified destination, but uniary operations (eg `add rA, rB #rA=rA+rB`)
- **Three operands:** Specified destination, binary operations

TL;DR, more operands = more flexible instructions, but more space taken up by operands

Implicit vs Explicit Encoding

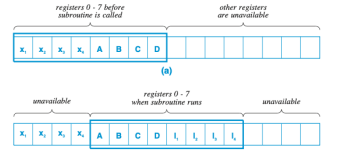
- **Implicit Encoding:** Operand types are always the same for a given opcode. Different opcodes are used for different types.
- **Explicit Encoding:** Operand field specifies what type of operands are being provided.

Operand Adressing Modes



Subroutines and Register Windows

When calling a subroutine, a window viewing the registers will shift between addresses, making some registers of them inaccessible, some new registers available, and keeping some between both calls. This allows for values to be passed to and from the subroutine, while keeping some values separated.



Register Banks

- Allows parallel access within same clock cycle → efficiency
- Some operations require operands from banks
- Register bank conflicts

Register Conflicts

Accessing 2 registers from the same bank simultaneously causes a register conflict. Best case, it causes a stall in the pipeline. Worst case, it causes the system to crash.

Solution

- reassign
- moving registers
- insert an instruction to copy values to the opposite register.

Physical Implementation

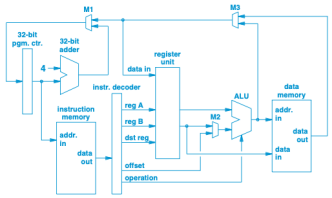


Figure 6.9 Illustration of data paths including data memory.

Instruction	M1	M2	M3
add rX,rA,rB	0	1	0
load rX,off(rA)	0	0	1
store rb,off(rA)	0	0	X
jump off(rA)	1	0	0

- Core loop between M1, 32-bit pgm. ctr., 32-bit adder
- Instruction memory returns instruction at given address
- Instruction decoder takes instruction and decodes it into individual parts
- Register fields used to select registers used in instructions, register unit takes fields and returns contents
- M2 Multiplexer takes auxiliary adding functions (such as adding an offset) and passes it through ALU
- ALU performs operation. For addresses, it's passed directly to data memory to get data out, for operations, data is passed to multiplexer to be stored into register for future use.

Bit Representations

Binary String	Unsigned (positional) Interpretation	Sign Magnitude Interpretation	One's Complement Interpretation	Two's Complement Interpretation
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	−0	−7	−8
1001	9	−1	−6	−7
1010	10	−2	−5	−6
1011	11	−3	−4	−5
1100	12	−4	−3	−4
1101	13	−5	−2	−3
1110	14	−6	−1	−2
1111	15	−7	−0	−1

Figure 3.9 The decimal value assigned to each combination of four bits when using unsigned, sign-magnitude, one's complement, and two's complement interpretations.

Microcode

Allows for simpler hardware design by offloading complexity to software. Macro instructions are made of microcode. Have a different instruction set available internally. Microcontrollers(s) sprocess microcode to emulate macrocode.

Pros

Less prone to errors compared to hardware design. Easier to implement. Ease of extension and modification. Offers another level of abstraction.

Cons

More overhead than dedicated hardware design. Variable cost for macro instructions, depends on number of micro instructions in its implementation. Microcontroller needs to run very fast.

Vertical Microcode

Microcontroller is a centralized unit sequentially excuting code; controls a single functional unit at a time. It must be fast. Think of a macrocode instruction as a function composed of micro instructions run one at a time.

Horizontal Microcode

Control **multiple** functional units simultaneously. Each instruction executes several operations in parallel. Allows greater use of parallelism and the Microcontroller does not need to run at as high a clock rate. Allows simultaneous control of multiple units in the CPU.

Memory & Caching

Memory designed to be hierarchical, with data copied into faster access levels as needed (acting as caching layers): ALU → Registers → Memory → Storage.

RAM Types

SRAM: Similar to a latch, high power consumption, low latency, high access speed.
DRAM: Similar to a capacitor, low power consumption, higher latency, slower access. As time passes the charge dissipates and becomes 0. Requires a refresh circuit.

Measure of Mem performance

- **Density** number of memory cells per square area of silicon. Memory density tends to double approximately every eighteen months. (Moore's Law)
- **Read** and **write** performance.
- **Latency** Time taken to do one operation. A memory system may need extra time between operations, latency alone is insufficient.
- The **read cycle** time and **write cycle** time are used as measures of memory system performance because they assess how quickly the memory system can handle a sequence of requests.

Memory bus: interface width / word size / number of lines

Memory Addressing

Memory is word addressable (r/w ops apply to words), but virtual addresses at processor level are byte addresses.
Let b be the byte-address, and N be the number of bytes in a word. Then

- **word address** $w = \lfloor \frac{b}{N} \rfloor$
- **offset** $= b \mod N$

We use N as powers of 2, to make arithmetic easier (division by 2^x is the same as right shiting by x bits, Modulo equivalent to truncation at x digits).

Byte Alignment

If a piece of data is stored in **a single word**, then it is said to be byte aligned. Good for performance.

Memory Address Space

Address size is the same as word size. **address space** is the set of possible addresses (An n-bit addressable system will have 2^n different memory addresses). If we use byte addressing instead of word addressing, we have a factor of N less memory (N = size of a word).

Calculating alloc. of bits in address

W : words per line, B : blocks/lines in cache, A : bits in architecture.
$|off| = \log_2 W, |blockId| = \log_2 B, |tag| = A - |off| - |blockId|$

Memory Management Unit

Interface that multiple memory chips can connect to, providing a common address space to processor. Implements virtual memory, caching. Combining address spaces generally done either sequentially or interleaved (generally #2). Interleaving done by selecting the least significant bits, assuming 2^N different chips to distribute over.

Caching

The same data is frequently read / written to. It can also reduce the impact of the Von Neumann bottleneck. L1 cache: associated with one particular core. L2 cache: on-chip cache that may be shared. L3 cache: on-chip cache that is shared by multiple cores.
Characteristics: Small - size of cache $\ll$ size of data at producer. Active - contains algorithm that decides data to store and handles requests. Transparent - producer and consumer unaware of cache's existence. Automatic - self-contained.
Hit ratio: $r = \frac{N_{hit}}{N_{hit}+N_{miss}}$
Cache look up cost:
$C_{cache} = rC_h + (1-r)C_m$
Cache always improves performance when $C_m > C_h$ and $r > 0$.
With two caches, $C_{cache} = r_1C_{h1} + r_2C_{h2} + (1-r_1-r_2)C_m$

Replacement Policies

Least Recently Used, Least Frequently Used

Cache Maintenance Policies

Write Through - As soon as value is modified, update all caches.
Write Back - Set a flag when modified (dirty bit), only write back when value in cache is evicted.

Cache Coherence Protocols

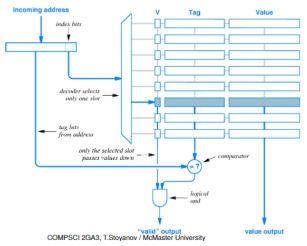
Multiple processors changing the same value means a cache coherence protocol is needed. All caches need to see write ops in same order, and need to be informed immediately when a value is changed. Some cases may lead to cache being flushed. Strategies: common directory, snooping

Direct Mapped Memory Cache

Several cache lines (blocks) containing words from memories. Broken up into Tags and Blocks (both in powers of 2).

- **Offset**: $o = a \mod W$, where o is the offset, a is the address, and W is the number of words per line. (if $W = 2^x$ then x lowest order bits in bin)
- **Block ID**: $b = (a \operatorname{div} W) \mod B$, where B is the number of blocks/lines in the cache.(if $B = 2^x$ then x lowest order bits in bin excluding o)
- **Tag**: $t = (a \operatorname{div} W) \operatorname{div} B$ (everything else)

Total mem needed to implement cache (bytes) = no.of lines × no. of words per line × no.of bytes per word



Self Associative Memory Cache

Maintains *multiple* underlying caches, searches all of them simultaneously through parallelism. For a single DMMC, addresses which collide in a slot will lead to constant cache misses, but with set associative we can reduce the chance of this happening. Full parallelism = Content Addressable Memory (CAM).

Virtual Memory

Addresses sent b/t CPU and MMU are in virtual address space, MMU talks to physical memory in physical address space. Keeps mapping between virtual space i and a pair, *(base, bound)*. *base* points to mem address mapping to vaddress 0, **bound** can be scaled dynamically and points to end of memory space. We split memory space into variable sized chunks to allow us to "use" more memory than is available, as most memory is idle.

Demand Paging

Split program memory up into fixed size chunks called **pages**. Similar to segmentation, more straightforward to implement. Requires cooperation b/t software and hardware. **Software:** Configure hardware - monitor page use - move pages b/t memory and storage - save pages to disk when they need to be swapped out of the resident set
Hardware: Handle address mapping - record when a page is used - detect access attempt to a missing page

Process

1. MMU tries to load address a on page P 2. MMU checks page table $P_t[P]$. Check if *presence bit* is set. 2.5. If not, MMU raises page fault, OS handles fault and loads P from hard drive, if no space left OS swaps out page with *use bit* of 0, if *modified bit* is set, save page, otherise discard. 3. If yes, MMU looks up address and updates *use bit* and *modified bit*. Page table holds pointers to frames, allocates all pages for program's memory space. Also maintains flags: presence bit (if page is resident), use bit (each time address is fetched, clear if page is unused for x cycles), modified bit (set if memory address in page has been written to)

Accessing address a with each page being V bytes long: Page number is $N = a \operatorname{div} V$. Frame address is $F = P_t[N]$. Offset: $O = a \mod V$. Physical address of a is $F + O$.

Translation Lookaside Buffer

Type of cache, we keep a record of $F = P_t[N]$ so if we look it up again we can return the cached version. Parallel lookup through TLB and request to memory. Cache holds virtual addresses with process ID: [ID][Virtual Address] = [address used by cache]

IO Bus

A bus is a digital communication mechanism that allows two or more functional units to transfer control signals or data. Interconnects functional units (includes external devices). A bus standard must specify all the details needed to construct hardware. An access protocol must be defined to permit the sharing of a bus b/w parts. The access protocol specifies how an attached device can determine whether the bus is available or is in use, and how attached devices take turns using the bus.

Parallel vs Serial

Parallel: Multiple data lines, pretty straight forward, multiply lines and clockspeed to get throughput. Lower latency per word, can send in one go. Potentially higher throughput, but possible issues with interference. **Serial:** Single data line, clockspeed = throughput in bits. Higher latency (word sent sequentially), but possibly higher rate of data transfer.

Communication Directionality

Single direction: data can only flow in one direction, ever. Half duplex: one direction at a time. Full duplex: both directions at the same time.

Multiplexing

Take a bunch of bits to be transfered, resize them into chunks, pass them into mux hardware, demux at the other end and reassemble.

Physical Implementation

Bus implemented from parallel wires along edge of board (eg PCIe). Different lines dedicated to different functions: **Control lines:** Take control of bus, request transfers, set signals and interrupts. **Address lines:** Transmit the address in a fetch/store operation. **Data lines:** Transmit data.
Alternatively, multiplex using combined address and data lines together. All devices connected to bus see data, address, control bits sent, only device with matching address responds. Each device on bus has a unique address, with each address corresponding to particular device function. Each socket has a range pre-assigned.

Unified Address Space

IO devices are in same space as memory addresses. Fetch/store to device is same as fetch/store to memory. MMU manages address alloc and raises faults when accessing *hole* in address space.

Bridging

Architecture has one primary bus and several aux busses. Bridges connect busses, provide address translation, relays data. Provides address translation in unified address space, addresses on main bus can be kept constant while being mapped differently on aux bus.

Interrupts

Two options for IO operations: **Option 1:** Each device exposed as an address. CPU performs fetch/store directly. Ops triggered and monitored by CPU. *Programmed I/O*
Option 2: Smart device carries out operations independently. Device must contain processor. CPU sets high-level goals, local processor performs the actual operations and monitors for results. *Interrupt-driven I/O*

Programmed I/O

CPU takes control of all low-level operations on device. Device can be very simple, fixed logic circuits. Sync issues, as CPU runs at a high clock race but device operations are much slower. Result: CPU has to wait for device. Special registers on device used to exchange information: Control and Status Registers (CSRs). Control register corresponds to set of addresses that respond to a store operation. Values of different status variables. Status register corresponds to set of addresses that respond to fetch operation. Repeated polling is used to check for operation status. Inefficient, not commonly used today. The typical usecase for programming in a low-level language is programming instructions for a memory controller.

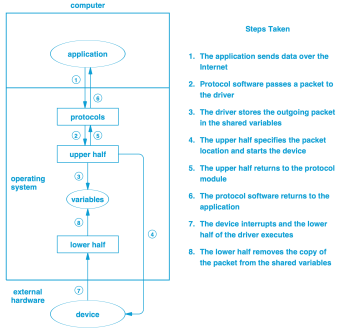
Interrupt-driven I/O

Requires special design for I/O device hardware (on-board processor and ability send interrupts), I/O bus architecture (interrupt signals), processor architecture (context switching), programming paradigm (OS supports interrupts). At boot time, fill IV table by device. IDs can either be by plug-in slot, or set by BIOS. Hotplug devices trigger interrupts, handler allocates device id and interrupt handler for new device. Handling usually done through device drivers.

Interrupt Vectors

Store a vector of interrupt response pointers in memory, so we know how to respond to each device's interrupt.

Device Drivers



Upper Half: Provides OS interface to user-space programs (syscalls)
Lower Half: Runs asynchronously, invoked by interrupts **Communication:** Via shared variables, buffers, and mutex locks (paralellism :D)

Direct Memory Access

Instead of context switching to deal with every byte of memory, give the device an address to a buffer to write a block of information. Put a few addresses in a linked list to make it better.
Alternatively, also include a list of instructions in that linked list.
High-volume devices can otherwise overwhelm the CPU with interrupts. It allows large amounts of data to be sent to devices. It allows the CPU to chain different operations. It allows devices to access memory directly.

Buffering

Collect a bunch of calls together, send them to I/O device together. Given a single call takes M cycles for operations and N cycles for overhead. Thus, K separate calls costs $K \times (M + N)$, while buffered costs $(K \times M) + N$.

Possible Questions

Calculate bits for Tag ID

Assume a **16-bit architecture** where the cache can hold **8 blocks/lines** of memory, each which is **32 bytes long**. How many bits are needed to represent the tag id?

- 16-bit architecture - **2** bytes per word
- 8 blocks - 2^3 blocks, so **3** bits for block id
- 32 bytes per block - $32/2 = 16 = 2^4$ words per block, so **4** bits for offset (or 5 bits if **byte** addressable)
- $16 - 3 - 4 = 9$ bits for the tag id.

Calculate Total GB of RAM Installable

How many GB of RAM can you install on a 32-bit **word** addressable system?

- 32 bit addressable means that one word is 32 bits or 4 bytes
- program counter is the size of a word; can point up to 2^{32} word addresses.
- Each word address is of size 4 (or size 1 if **byte** addressable), thus using $2^{32} \times 4 = 2^{34}$ bytes of total memory.
- $2^{34}/2^{30} = 2^4 = 16$ GB of memory.