Addressing Modes

Source/Destination Operands

- Source supports all 7 addressing modes, destination only supports 4
    - Source: 1 (register), 5 (memory), 1 (machine)
    - Destination: 1 (register), 3 (memory)

#	Addressing Mode	Assembly code & examples		S/D	Data stored in
1	Register Direct	`Rs`	`ADD.W` **`R4,`** `R10`	S/D	Register
2	Indexed mode	`x(Rs)`	`ADD.W` **`6(R4),`** `R10`	S/D	Memory
3	Register Indirect	`@Rs`	`ADD.W` **`@R4,`** `R10`	S	Memory
4	Indirect Auto-increment	`@Rs+`	`ADD.W` **`@R4+,`** `R10`	S	Memory
5	PC Relative (Symbolic)	`label`	`ADD.W` **`cnt,`** `R10`	S/D	Memory
6	Absolute mode	`&label`	`ADD.W` **`&cnt,`** `R10`	S/D	Memory
7	Immediate mode	`#n`	`ADD.W` **`#100,`** `R10`	S	Machine code

#	Add. Mode	Description	Size (word) Default: 1 word for instruction
1	**`R4`**	Contents of register used as operand	
2	**`6(R4)`**	Address of operand = Register contents + offset number x *(e.g. 6)*	+1 for each operand in this mode **2^{nd}/3^{rd} word is the offset number**
3	**`@R4`**	Address of operand = Register contents	
4	**`@R4+`**	Address of operand = Register contents Register = Register contents ++ (+1 if `.b`) (+2 if `.w`)	
5	**`cnt`**	Address of operand = PC + PC value. After PC moves to 2^{nd} / 3^{rd} word, the add. of operand will be address of 2^{nd} / 3^{rd} word + its content value	+1 for each operand in this mode **2^{nd}/3^{rd} word's address + content is the operand address**
6	**`&cnt`**	Address of operand = Contents of next instruction (PC++)	+1 for each operand in this mode **2^{nd}/3^{rd} word's content is the operand address**
7	**`#100`**	Operand = Contents of next instruction (PC++) (like literally if the instruction is 100 then use 100)	+1 if used as source (only source) **2^{nd} word is the operand**

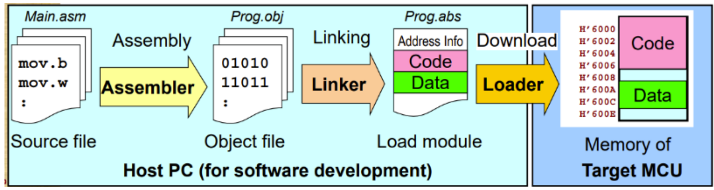
Two-operand arithmetic instructions					
op	src	Ad	b/w	As	dest
b/w: 1bit 0=w, 1=b			Ad: 1 bit As: 2 bit		

2 (SR)	**#4**
2 (SR)	**#8**
3 (CG)	**#0**
3 (CG)	**#1**
3 (CG)	**#2**
3 (CG)	**#–1**

Values available in CG

Stages in developing an assembly program

Stage	Description	
Text editor	Edit the text-based mnemonics in source file (*.asm)	Done on Host PC
Assembler	Converts mnemonics in source file into machine code. • Produces an object file (*.obj)	
Linker	Combines several obj files (e.g. from libraries) • Produces a load module that contains machine code and address info (*.abs)	
Loader	Uses load module's address info to download instructions and data constants into appropriate memory areas for execution	Use to load to target MCU



Group	Operator	Description
1	+, -, ~, !	Unary plus, minus, 1's complement, Logical NOT
2	*, /, %	Multiplication, Division, Modulo
3	+, -	Addition, Subtraction
4	<<, >>	Shift left, Shift right
5	<, <=, >, >=	Less than, Less than or equal to, Greater than, Greater than or equal to
6	=[=], !=	Equal to, Not equal to
7	&	Bitwise AND
8	^	Bitwise exclusive OR (XOR)
9	|	Bitwise OR

Precedence in Expressions

Assembly Language Syntax

- Maximum 200 characters, any chars beyond are truncated
- All statements must begin with a label / blank / asterisk / semicolon
- One or more blanks (tab/space) must separate each field

	Label	Mnemonic	Operand List	Comment
Range	Contain up to 128 alphanumeric characters (A-Z, a-z, 0-9, _ and $)		Contain 0, 1 or 2 operands (separated by comma)	
Syntax	• Case sensitive • First character cannot be a number • Can be followed by a colon • On a line by itself is valid	• Many opcode mnemonics require suffix to indicate size (`.b`/`.w`)	View operand table below • Can contain expressions (handled by assembler before convert to machine code)	• Column 1: begin with **asterisk * or semicolon ;** • Other columns: begin with **ONLY semicolon ;**
Types	1. **Address Label**: stores add. of memory 2. **Value label**: .equ creates value labels	1. Assembler Directives 2. Executable Instructions	1. Symbols 2. Constants (unsigned)	

E.g.	`Fri .equ 5`
	`Week .byte 7`
	`Delay dec.w R15 ; The Delay Loop`
		`.data`
		`.space`
		`mov.w R8, R10`
			`mov.b R5, R6`
			`dec.w R15`
			`ret (no operand)`	

Operand Representations

Representation	Decimal	Hexadecimal	Octal	Binary
Limit/Range	-2147483648 to 4294967295	8 Hexadecimal digits	8 Hexadecimal digits	32 binary digits
Suffix		'H' / 'h' or preceded by '0x'	'Q'/'q' or preceded by '0'	'B'/'b'
i.e.	1000, -32678	78h, 0x78, 0A234H	078, 0x78, 78Q	0000b, 11110000B

For **Hexadecimal**, if value starts with alphabet (A-F), **MUST add 0 in front**. Assembler treats anything that starts with alphabet as a label.

Assembly Directives

	Syntax		Description	Other info
Sections Location Counter $	.text		Stores code (executable instructions)	Initialised sections (like in ROM/FLASH)
	.data		Stores data	
	.sect	.sect “section name”	• Create a new section (with string as name) • continue program from there until it hits another section	
	.bss	.bss symbol, size(byte) [,alignment]	• Section created for you by the assembler • Reserves size bytes in the already initialised .bss section (Used to store uninitialized variables)	Can appear anywhere without affecting section contents of current section (e.g. if it’s inside .data, it will just continue in .data after that line)
	.usect	Symbol .usect “section name”, size(byte) [,alignment]	• Start a new uninitialized area (Reserve space) (Create another section that doesn’t exist)	
Initialising Values	.byte	.byte value$_1$[,…, value$_n$]	One or more successive bytes in current section	
	.double	.double floating_pt_value	48-bit MSP430 floating-point constant	
	.float	.float floating_pt_value	32-bit MSP430 floating-point constant	
	.space	.space size(byte)	• Reserves space (in bytes) in the current section • A label points to beginning of the reserved space	
	.string	.string “string$_1$”[,…, “string$_n$”]	One or more text strings	
	.word	.word value$_1$[,…, value$_n$]	One or more 16-bit integers	
Symbol Table	.set	Symbol .set value	Equates a constant value to a symbol (can be address/value) - “It’s like a text replacement tool!!”	.set and .equ are Identical, can be used interchangeably
	.equ	Symbol .equ value		
Library ref/def	.ref	.ref func	Reference symbols defined outside of the program	
	.def	.def func	Label a function in the current module	
	.global		Declares a symbol to be external so other modules can use it - Acts as .def for defined symbols and .ref for undefined symbols	
End	.end		End program – anything that comes after it will not be assembled by the assembler	

Executable Instructions (1/2)

	Instruction & Syntax		Purpose & Side effects (If any)	Status bits
Data Movement	mov	`mov.x src,dst` `mov.b R10, R11`	Move from source to destination - Source not modified	NIL
		`(.x = byte/word)`		
Arithmetic	sxt	`sxt reg` `sxt R10`	Sign extension: copies bit 7 into bits 8-15 • E.g. Useful when you want to perform arithmetic between a byte and a word (convert 8-bit to 16-bit)	N/Z: if N/Z, else reset C: NOT(Z) V: Reset
	add	`add.x src,dst` `add.w #2, R13`	Adds source to the destination - Result stored in destination	N/Z: if N/Z else reset C: if carry else reset V: if overflow occurs else reset
	addc	`addc.x src,dst` `add.w #2, R13 (R13=R13+2+c)`	Adds source + carry flag to destination - Often used to support higher-order addition (adding numbers beyond 16-bit)	N/Z: if N/Z else reset C: if carry from MSB else reset V: if overflow occurs else reset

	dadd	`dadd.x src,dst`
		`R10=0x1234, R11=0x6789`
		`dadd.w R10, R11 R11=0x8023`
			Adds source to the destination decimally
			- Treats hexadecimal values as DECIMAL (base 10)
			- Result stored in destination (as hexadecimal)
				N/Z: Set if MSB/Z, else reset
				C: if result > 9999 / >99
				V: Undefined

	sub	`sub.x src,dst` `sub.b R10, R11`	Subtracts source from destination - Result stored in destination - Executed via adding 2's complement of src to dst	N/Z: if N/Z else reset C/V: if carry/overflow else reset
	subc	`subc.x src,dst` `subc.b #5, R12 (R12=R12-5-C)`	Subtracts source and carry flag from destination - Often used to support higher-order subtraction	
Logical and Bit Mani-pulation	>>>>	`Example: R10=01010101`		

	and	`and.x src,dst`
		`and.b #11110000b, R10`
		`R10=01010000b`
			Bit clear using 0 (AND)
			- Logic 0 is used to **clear** specific bits in dst
				N/Z: if MSB/Z else reset
				C: NOT(Z)
				V: Reset

	bis	`bis.x src,dst`
		`bis.b #11110000b, R10`
		`R10=11110101b`
			Bit set (OR)
			- Logic 1 is used to **set** specific bits in dst
				NIL

	xor	`xor.x src,dst`
		`xor.b #11110000b, R10`
		`R10=10100101b`
			Bit complement (XOR)
			- Logic 1 is used to **complement** specific bits in dst
				N/Z: if MSB/Z else reset
				C: NOT(Z)
				V: if both operands are negative

	bic	`bic.x src,dst` `bic.b #11101100b, R11`	Bit clear (NOT(src) AND dst) - Logic 1 is used to **clear** specific bits in dst	NIL
	OTHER EMULATED INSTRUCTIONS			
	`clrc, clrn, clrz (clear c/n/z flags)` `setc, setn, setz (set c/n/z flags)`		`clr.b / clr.w (clear destination)` `inv.b / inv.w (invert destination)`	

Executable Instructions (2/2)

		(.x = byte/word)		
Rotation	RLA	`rla.x src,dst` `rla.b R11`	Arithmetic Left Shift (Rotate Left Arithmetically) - Put 0 in LSB - Push MSB to carry flag	N/Z: if N/Z else reset C: loaded from MSB V: MSB change
	RRA	`rra.x src,dst` `rra.b R11`	Arithmetic Right Shift (Rotate Right Arithmetically) - Right Shift but keep the sign (MSB copied to MSB) - Push LSB into carry flag	N/Z: if N/Z else reset C: loaded from LSB V: reset
	RLC	`rlc.x src,dst` `rlc.b R11`	Rotate Left through Carry - Push carry flag to LSB - Push MSB to carry flag	N/Z: if N/Z else reset C: loaded from MSB V: if sign bit change
	RRC	`rrc.x src,dst` `rrc.b R11`	Rotate Right through Carry - Push carry flag to MSB - Push LSB to carry flag	N/Z: if N/Z else reset C: loaded from LSB V: reset
	swpb	`swpb dst` `swpb R11`	Swap bytes (lower-order byte & higher-order byte) - E.g. 0xAABB will become 0xBBAA - Only supports word-sized operations	NIL
Program Control	**Conditional Jumps (8 instr)** $PC_{new} = PC_{old} + 2 + PC_{offset} \times 2$ (PC_{old} is the address of the jump instr) ($PC_{old}+2$ is the address of the offset)		JEQ/JZ: Z=1 | JNE/JNZ: Z=0 | JC: C=1, JNC: C=0 | JN: N=1 *(Signed)* | JGE: N=V, JL: N≠V *(Signed)*	JMP: Unconditionally | Signed: JGE/JL/JN Unsigned: JLO/JHS
	bit	`bit.x src, dst` `bit.w #0x0080, R11`	Bit testing - Source & Operand are logically AND'ed - Only affect status bits - Source & destination not affected - Used to check for certain bits of dest before a conditional jump	N/Z: if MSB=1/Z else reset C: NOT(Z) V: reset
	cmp	`cmp.x src, dst` `cmp.w #0x0014, R11`	Compare values - Source subtracted from destination - Source & destination not affected - Used to compare two values before a conditional jump	N/Z: if N/Z else reset C: if carry else reset V: if overflow else reset
			OTHER EMULATED INSTRUCTIONS	
	tst	`tst.x dst`	When you want to test negative/zero value - Emulated instruction using cmp - Destination not affected	N/Z: if N/Z else reset C: set V: reset
	br	`br dst`	Allows branch without any constraint (same as jmp??) - Actual instruction is `mov dst,R0`	

Subroutines & the Stack

The Stack is used for:			
Subroutine calls (store return address)	Subroutine parameters	Temporary storage for local variables	Interrupts & Exception Handling

- Data not removed when popping, SP just moves downwards (to top of stack).
- When **subroutine ends**, the stack pointer must be **pointing at return address**.
    - ^ Any push MUST have appropriate pop **before RET is executed**.

Interesting history to understand : [Why does the stack grow downward? - Software Engineering Stack Exchange]

Instructions for Push/Pop		
`push.w`	SP dec by 2, move src to @SP	PRE-DECREMENT (dec first then do stuff)
`pop.w`	move @SP to dest, SP inc by 2.	POST-INCREMENT (do stuff first then incr)

Stack Pointer (R1/SP)
Pre-decrement, post-increment scheme
Always point to RAM location and is **always even**

Instructions for Subroutines	
`call`	**Pushes return address (2 bytes) onto system stack**, then PC jump to effective address. *SP dec by 2 BEFORE moving the src into top of stack.*
`ret`	Return from subroutine by **popping the return address from stack into PC**. *SP inc by 2 AFTER moving the stack value to the dst.*

Parameters Passing Methods for Subroutines

Type	Register	Memory	Stack
Description	Parameters placed into register before calling subroutine	• Region in memory treated like mailbox • Parameters in a block at predefined memory location • Start address of the memory block is passed to subroutine via an address register	• Parameters pushed onto stack before calling subroutine • Retrieved from stack within subroutine *(Most favoured method for parameter passing but can use combination of methods)*
Used …	when no. of parameters is small	for passing large number of parameters – "Pass by reference"	for passing large number of parameters (as long as no stack overflow)
Pros	Very efficient		• Most general method – no registers needed • Support recursive programming
Cons	• Reduces number of available registers within subroutine • Lacks generality	Not as efficient as register	• Parameters pushed on stack must be removed by main/calling program immediately after returning from subroutine • If not, will cause stack overflow

Stack Frame

- **Temporary memory space for local variables** (-> only belong within the subroutine)
- Created on entry into subroutine and released on exit from subroutine
- Accessed during subroutine using SP (via appropriate positive displacements – indexed addressing mode!)

Peripherals connect to a processor in 2 ways
1. Loosely coupled — via external bus (USB, Firewire), via network (Ethernet, ATM), via port (serial, parallel)
2. Tightly coupled — via fast internal bus (graphics e.g. GPU)

Input Output Interfacing

Parallel	Serial
More than 1 bit of information can be transferred simultaneously (Multiple wires)	Only 1 bit of data to be transmitted at a time (can use single wire / pair of wires)
• Faster data transfers • Hard to communicate using wireless connection, not economical • Wires will generate interference, increase transmission time because need to re-transmit	• Less expensive • Less efficient / slower (But new serial interfaces like USB 3.2 can support up to 20Gbits/s) • More robust: no cross talk & no limitations at high frequencies/long range
e.g. Printers, CPU Internal bus	e.g. USB

Modes of Transfers

Simplex mode	Half duplex	Full duplex
one direction only	both directions but only one direction at a time (T/R)	both directions simultaneously (sometimes called async transmission)
e.g. temperature sensor sending data	e.g. walkie talkie (single channel, 2 way radio)	e.g. telephone

Polled vs Interrupt-driven

Polled I/O (Program controlled)	Interrupt-driven I/O
CPU initiates/controls/terminates data transfer	**External device initiates** data transfer **CPU controls/terminates** data transfer
CPU keeps polling the port (issues command to I/O module), waiting for data to be available, until I/O operation complete	Interrupt: external hardware event - Executes special routine called ISR (Interrupt Service Routine)
Useful when data transfer must be completed before program can continue	Useful when timing of data transfer cannot be known beforehand / infrequent
e.g. wait for input from mouse/keyboard *e.g. wait for switch to be pressed*	*e.g. sending data to a printer* *e.g. getting data from a temperature sensor*
Advantages	
Minimum hardware interface circuitry Programmer has complete control	Efficient use of CPU, provides service at request of peripheral
Easy to test and debug	CPU can continue other tasks between interrupts
Disadvantages	
Inherently inefficient, CPU can't do anything until data transfer is complete	More hardware interface circuitry required between I/O and processor
Waste processor time if CPU faster than I/O	More complex & difficult to debug

Synchronous: **Common clock** used between transmitter and receiver to synchronise the bit transfers

Asynchronous: **No common clock**: one event will trigger another

- Each data bit comprises of
  - 1 start bit
  - 7/8 bits of information
  - (optional) 1 parity bit — Parity error if any
  - 1, 1.5 or 2 stop bits — framing error if stop bit = 0
- Parameters must be specified before transmission
  - Baud rate (Max number of bits per second) – 1/T
  - Number of data & stop bits
  - Optional Parity bit
- Each bit is represented by Mark (1) or Space (0)
- Idle: remains in Mark (1)
- Start bit = Space (0)
- Stop bit = Mark (1)

Memory Mapped I/O

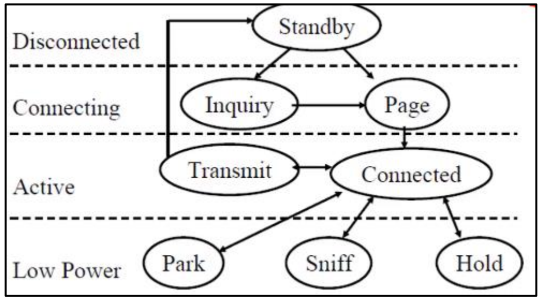
- Peripheral registers accessed in the same way as memory
- Set of peripheral registers are provided by I/O ports and peripheral modules so that
  - Peripheral can be configured (e.g. async data format)
  - Peripheral can be used (to output value to I/O port)
- MSP430
  - P1DIR: Set to bit 1 = output; set to bit 0 = input
  - P1OUT: Stores output data
  - P1IN: Shows port 1 pin states

Isolated I/O (e.g. Intel 8086)

- Two separate address spaces (one for memory, one for I/O)
- Two separate control lines (read & write)
- Separate instructions for data transfer e.g. IN/OUT for I/O

Bluetooth - Short-range radio connection technology that is low power & lower cost, not limited to "line of sight". Operating frequency ~2.4GHz

Piconet	Type of BT network consisting a set of BT devices sharing same channel / frequency band • 1 master • Max 7 active slaves (up to 255 parked nodes) • Slave can transmit only when a master requests
Scatternet	Interconnecting multiple (~10) piconets



BT Operational Status

Packet Format
Access Code
Header
Payload

FH-TDD-TDMA	
- FH Sequence is determined by the master in the piconet (function of the master's BT add) - Masters in different piconets in same area uses different hop frequencies (-> physical channels) - **Collision is possible** if two piconets **use the same physical channel**	
FH: Frequency Hopping	
Purpose	• Minimises interference between wireless communication • Makes jamming and interception difficult, but **can't completely overcome**
Implementation	• Transmit radio signals by rapidly changing carrier frequency among many distinct frequencies occupying a large spectral band • Hop rate: 1600 hop/s – each physical channel occupied for **0.625ms**
More implementation	• **Master transmits during even numbered slots**; slaves transmit during odd • Packets can be **1/3/5 slots** long. Hopping is deferred during data transmission.
TDD: Time Division Duplexing	
Data transmission alternates between two directions (transmitter & receiver) – half duplex	
TDMA: Time Division Multiple Access	
Access technique when more than two devices sharing the piconet	

Power Down Modes: Inactive states (Low Power) – **Sniff > Hold > Park**	
Sniff	• Low power mode • Slave listens after fixed sniff intervals
Hold	• Low power mode • Place the link between master and slave in hold mode for a specified time
Park	• Very low power mode • Wakes up periodically & listens to beacons from master • 8-bit parking member address and loses its 3-bit active member address

Bluetooth Low Energy (Bluetooth LE)

- Lower energy, longer battery life
- Used for applications that do not need to exchange large amounts of data and can run on battery power for years at cheaper cost
- e.g. proximity detectors, location, health devices & sensors, IoT applications

Secondary Memory Subsystems

Register (Processor) → Cache Memory (sRAM) → Main Memory (dRAM/ROM) → Secondary Memory (Hard disk, SSD)

External memory consists of peripheral storage devices that are accessible to the process via I/O controllers.

Types of external secondary memory	
Magnetic	• Mechanical hard disk • Floppy disk
Optical	• CD-ROM • DVD • BLU-RAY
Semiconductor	• Memory cards • SSD

Characteristics of Magnetic Hard disk	
Platters	• Multiple "disks" on a common spindle • Covered with thin magnetic film • Rotate on spindle at constant rate • Can be single/double-sided (1/2 surfaces)
Read/write heads	• Mounted on movable arm • Close proximity to the surface • During r/w, head is stationary, platter rotates • May be a single r/w head or separate ones

- Striped by cylinder
  - Data stored at same relative position on different surfaces
- Striped by platter
  - Data stored on the same platter, across different tracks

Data Organising & Formatting	
Tracks	Data is stored on the surface of the platter in concentric rings called Tracks
Cylinders	Set of all the tracks in the same relative position on the platter (# no. of tracks = # no. of cylinders)
Sectors	Tracks divided into multiple sectors • May be separated by a small gap to reduce precision requirement • Minimum data block size = ONE SECTOR

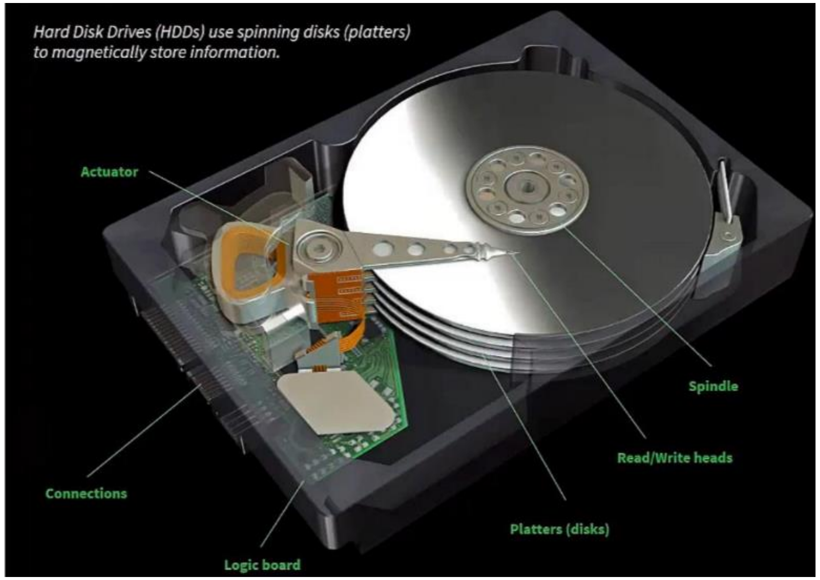
- Most modern hard disks have multiple platters per disk
  - Data is **striped by cylinder**
    - Reduces head movement
    - Increases speed

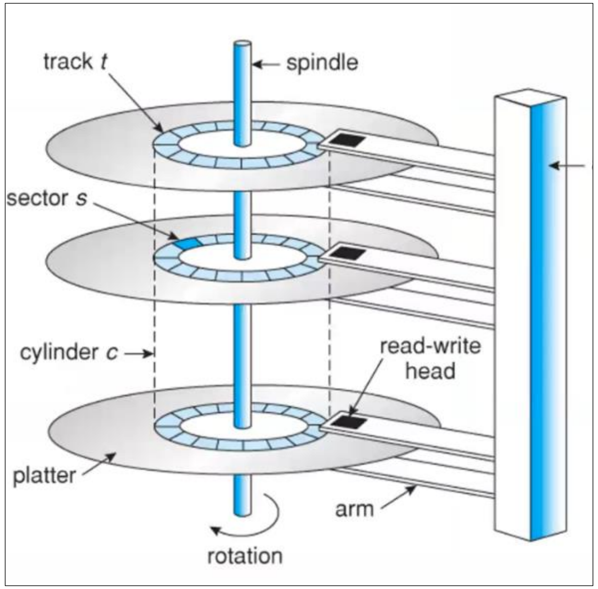
- Access time can be a bottleneck when accessing random files on the disk

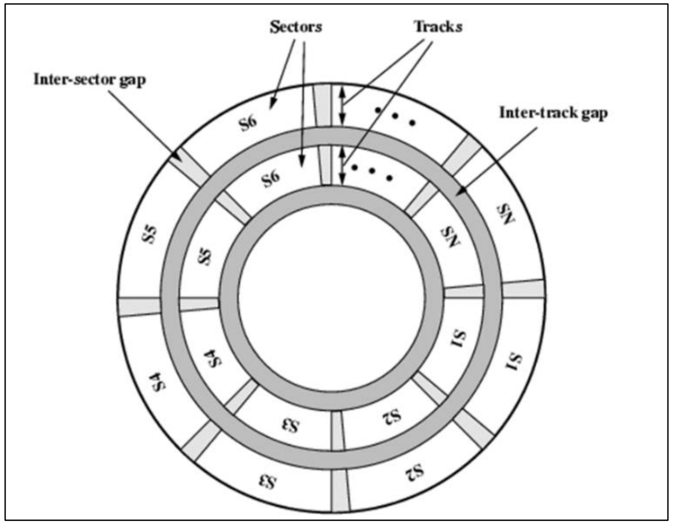
Speed stuff	
Seek Time (Ts)	Time it takes for the head to move to the correct track
Rotation Speed	Revolutions per minute (RPM) or Revolutions per second (RPS) || **RPS = RPM/60**
Rotational Delay (Tr)	Time it takes for the disk to rotate until head reaches starting position of target sector **Average rotational delay = 0.5 * 1/RPS seconds** (0.5 * amount of time for 1 revolution)
Access Time (Ta)	Time from request to the time the head is in position **Ta = Ts + Tr.**
Transfer Time (Tt)	Time required to transfer the data after head is positioned Dependent on rotation speed, number of bytes per track, number of bytes to transfer
Track Density (Dt)	Number of sectors per track
Sector Density (Ds)	Number of bytes per sector
Track-to-track access t	Time taken to move from one track to successive track
Ttotal = *Ta + Tt* = **Ts + Tr + Tt**	

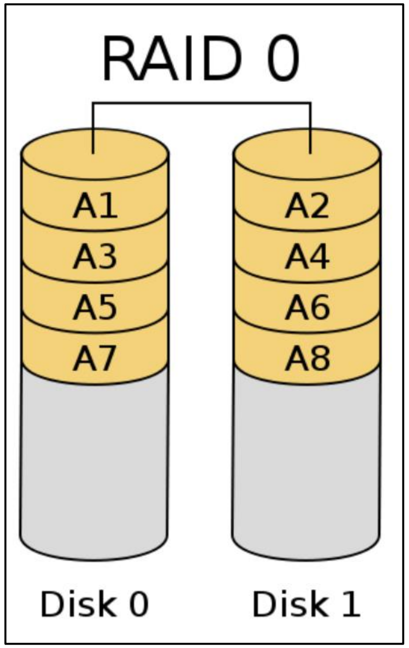
TYPES OF RAID
RAID 0
No redundancy
Minimum 2 disks
Striped across all disks "Round Robin striping"
A: Increased speed (Disks seek in parallel)
RAID 1
Mirrored Disks: Data written **identically** to 2 or more disks
Minimum 2 disks
2 copies of each stripe on separate disks
A: Faster read operations, but write same spd
A: Recovery is simple
D: Higher cost
RAID 10
Mirroring and striping (stripes of mirrors)
Minimum 4 disks
Only half capacity is for data storage
A: Fast read/write operation

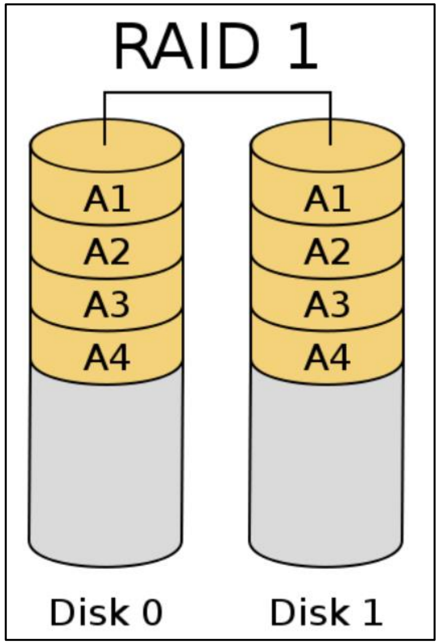
Some nice pictures for Secondary Memory Subsystems:

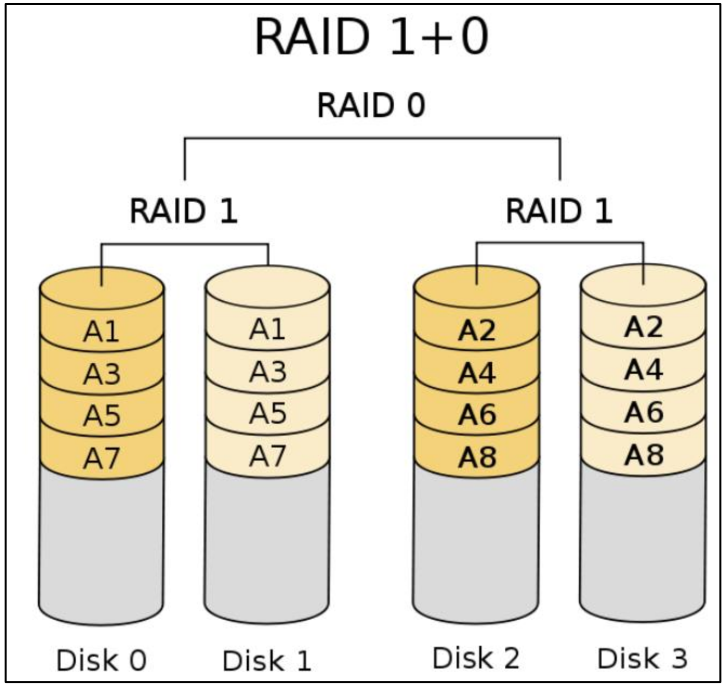












Primary Memory Subsystems

Cache Memory

- Small (optional) amount of fast memory between main memory and CPU
- Single/Multi-level
  - CPU > L1 > L2 > L3 > Main Mem
- Cache operations handled by <u>cache controller</u>
  - CPU requests content -> Check if data avail in cache -> if not present -> fetch block from mem and store in cache -> CPU read from cache

Access Time (Latency)	Memory Cycle time	Transfer Rate
Time from control request sent to memory to time that data available/stored (R/W)	Access Time + additional time before memory access, mainly for RAM	Rate at which data can be transferred into/out of memory • 1/Memory Cycle time for RAM

Bus – communication pathway between 2 or more devices

- Address: No. of address lines (width) determines max memory capacity
- Data: No. of data lines determines bits that can be transferred at once
- Control: Supports asynchronous & synchronous
  - Typical control lines include:
    Memory operations, I/O operations, Interrupts, Clock, Reset

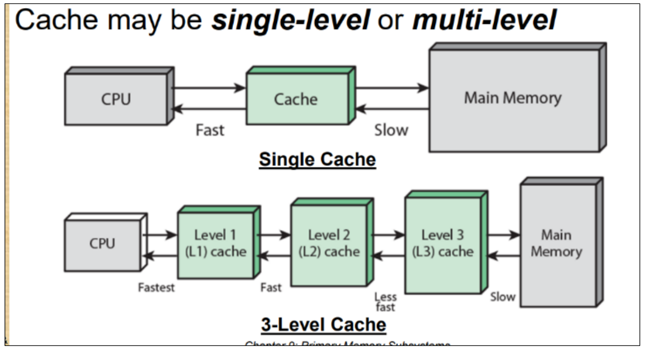
Synchronous: controlled by clock	**Asynchronous**: events one after another
Simpler to implement and test	More complex diff devices operate diff speed
	A slow device may hold up the entire bus
Less flexible as all devices on the bus are tied to a fixed clock rate	Very flexible as a mix of slow/fast devices can share a bus

<u>Basic Memory cell</u> - 2 operations – **read & write**
3 terminals – **Select, Control, Data** (nxt pg for pic)

Memory needed in <u>F</u>D<u>E</u> cycle:
Fetch & Execute

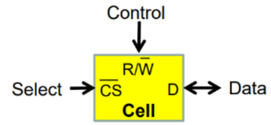
	RAM		ROM
Content	Can be read from/written to any time.		Not easily changed. (Once changed, contents can be read but not normally written to.)
Volatility	Normally **volatile** (data retained when power is on).		Non-volatile
Built using:	**Transistors** - Viewed as digital switch with Source (S), Gate (G), Drain (D) - Only when gate is closed (G=1), current is allowed to flow from S to D.		A **single transistor switch** for the bit line • Either connected or disconnected • Other end of the bit line connected to power supply through a resistor
	Static RAM (SRAM) – commonly used in **Cache**	**Dynamic RAM (DRAM)** used in **Main Memory**	**Types of commonly used ROM**
Made by:	• 2 cross-connected inverters (4/<u>6 transistors</u>) in a feedback loop to prevent leakage • Typically uses CMOS cell (low power)	• <u>A transistor and a capacitor</u>	**<u>PROM</u>**: Programmable ROM • Usually written once, at time of manufacture
Implementation	Word Line & Bit Line • Two transistors controlled by word line acts as switches between cell & bit lines (selects cell) • To Read/write: read/applied from/to bit line	(both R/W refreshes). Select using word line to • Read: use sense amplifiers on bit lines to identify high/low voltage • Write: dis/charge capacitor using bit line	**<u>EPROM</u>**: Erasable Programmable ROM • Use special transistors instead of fuse • **Remove all charge using UV light** **<u>EEPROM</u>**: Electrically Erasable PROM • **Individual cells** content can erase electrically
Simplicity		Simpler to build (less components)	• More expensive than EPROM
Power	Needs power to retain state	**Charge can leak and must be <u>refreshed</u>**	**<u>FLASH</u>** (based on EEPROM, but only **erase blocks**)
Speed	Faster than DRAM / short access time	Slower than SRAM / long access time	• Read block -> erase block -> write block
Density (of data per space) & $$	Need several transistors per cell so lower density & more expensive	Higher density & lower cost Higher capacity, larger memory units	• Greater density & lower cost outweighs block write. e.g. Cell phones, Cameras, SSD

Some nice pictures for Main Memory Subsystems:

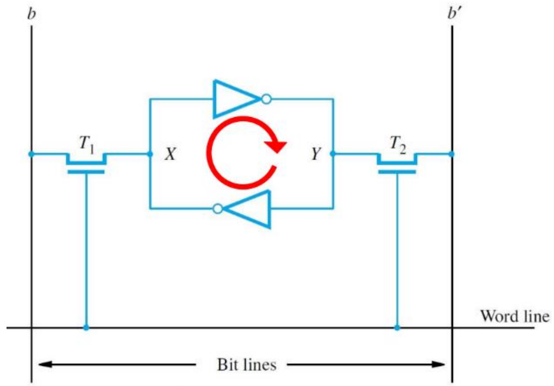


A basic memory cell has 3 terminals:

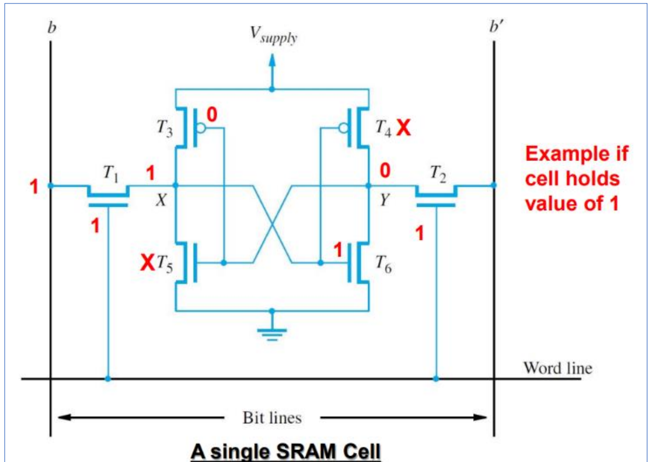
- **Select**: Activates/Selects a cell for reading/writing
- **Control**: Indicate if a read or write operation is required
- **Data**: Indicate the logic to be stored (Write operation) or the logic that has been stored (Read operation)



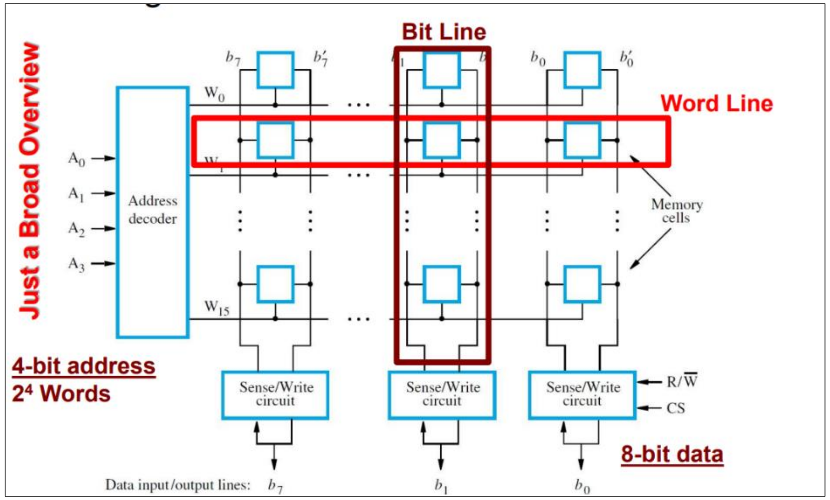
SRAM diagrams

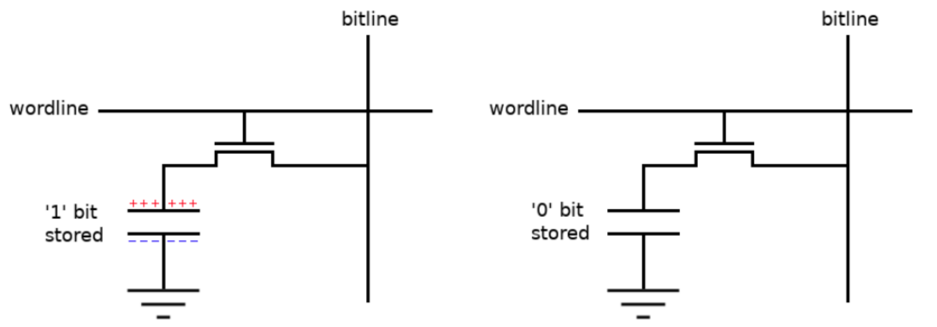


A single SRAM Cell
A logical view of the cell

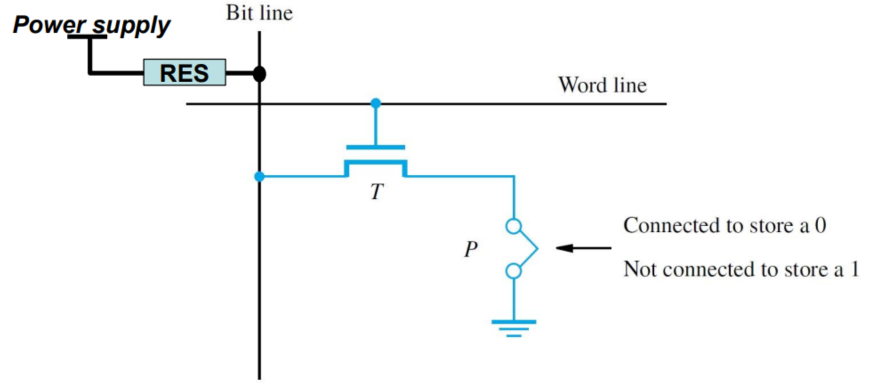


A single SRAM Cell
Actual Implementation using transistors





A single DRAM Cell



A single ROM Cell

Memory Management

Type	Uni program systems	Multi Program systems
Description	Only executing ONE program at one time	More than 1 program can be executed concurrently
	No concurrency	**Concurrently but need not be simultaneously**
	Memory split into two: One for OS (monitor), one for currently executing program ("user")	"User" memory is sub-divided and shared among active processes → **Memory management**
Problem	I/O operations required by program can be very slow vs CPU, making system **inefficient**	

Process	A program in execution
Long-term queue	queue of **process requests**, typically stored on disk
Intermediate queue	queue of **existing processes** that was **temporarily kicked out of memory**

Modern computers only **execute programs loaded into main memory**

- Main memory limited → Fetch the program to be executed from 2ndary mem to the main mem → SWAPPING

Swapping: is an I/O Operation btwn disk & mem

If no processes in memory is ready:

- Swap out a blocked process (main mem -> int. queue)
- Swap in a ready process or new process (int./long-term queue -> main mem)

Partitioning: splitting memory into sections to allocate to processes. Two Methods:

Type	Fixed-size partitioning	Dynamic partitioning
Descr.	• Size of each partition is fixed • Partition may or may not be equal size • Process fitted into smallest partition "best fit"	• When process placed into main memory → exact required memory is allocated to the process
Problem	Some wasted memory	• A hole at end of memory (too small to use) • When process swaps out, the new process swapped in may be smaller, **creating more holes**

aka

External fragmentation: Solutions
Coalesce • Join adjacent holes into one large holes
Compaction • From time to time, OS go through memory and move all holes into one free block

Paging: *Used to **overcome the problem of "holes"** in basic partitioning*

- **Pages**: programs/processes divided into equal sized small chunks
- **Frames**: main memory divided into small chunks of equal sizes
- **Size of Frame == Size of Page ------> page/frame offset is the same**
- Required number of frames are allocated to process
- OS maintains the list of free frames
- **Page table**: needed to keep track of memory allocation
  - (process doesn't need contiguous page frames)
- **Logical Addresses (Virtual)**
  - A location relative to beginning of program
- **Physical Address**
  - Actual location in memory
- **Memory Management Unit (of the OS)**
  - translates between logical / physical address

LOGICAL ADDRESS is divided into:	
Page number (p)	**Page offset (d)**
used as an index into a page table	relative address within a page

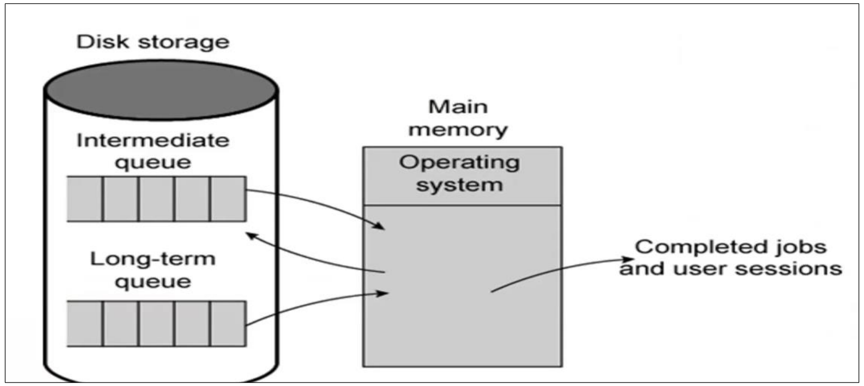
- If the whole logical address is n-bits: **logical address space = 2^n**
- If the total number of items in 1 page uses m-bits: **page size = 2^m**
- Bits left to store page number = n – m → **Number of pages = 2^{n-m}**

e.g. if n=16 bits, m=10 bits
- Total number of logical addresses = 2^{16} = 65536
- Total number of items in 1 page = 2^{10} = 1024
- Total number of pages = 2^{16-10}=2^6 = 64

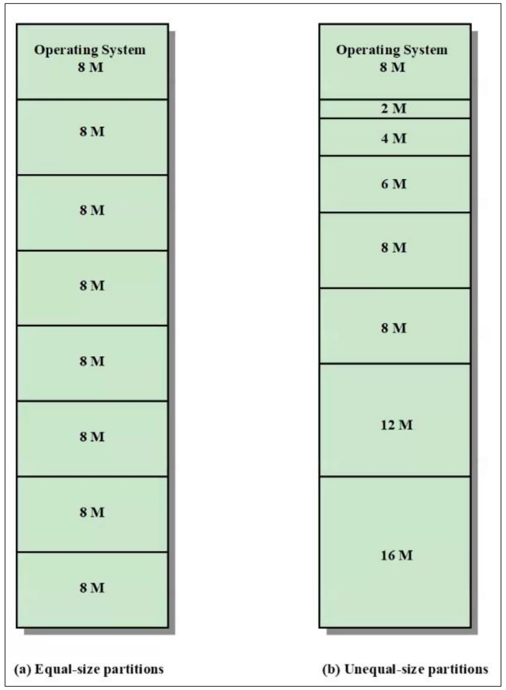
Example: The logical address space contains 128KB, the physical address space contains 64KB. The page size is 2KB. How many frames & pages are there in the system respectively?

- Pages divide program/processes (logical address space). There are 128/2=64 pages.
- Frames divide the main memory (physical address space). There are 64/2=32 frames.

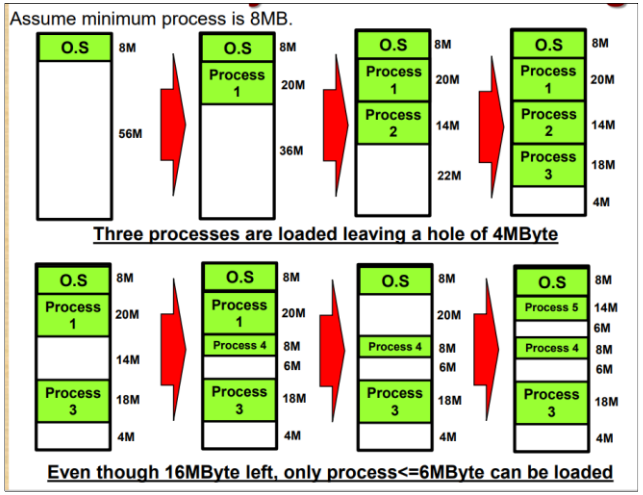
Some nice pictures for Memory Management:



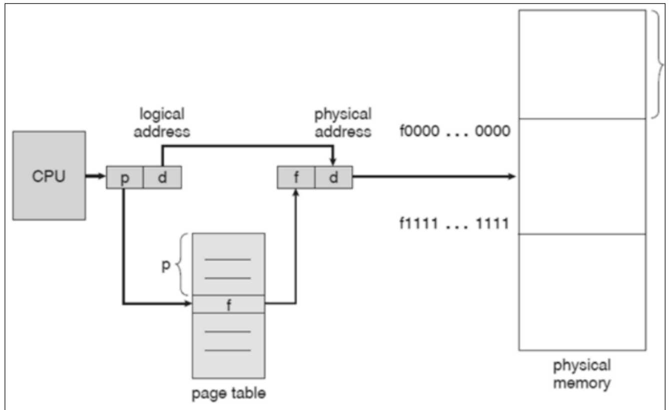
Swapping: processes moving to/from intermediate queue



Fixed partitioning

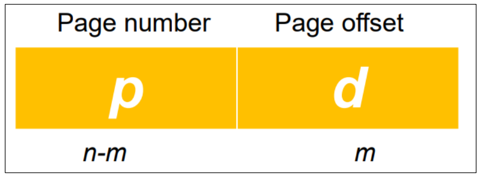


The two issues with dynamic partitioning
(hole at the end & holes created during swapping)

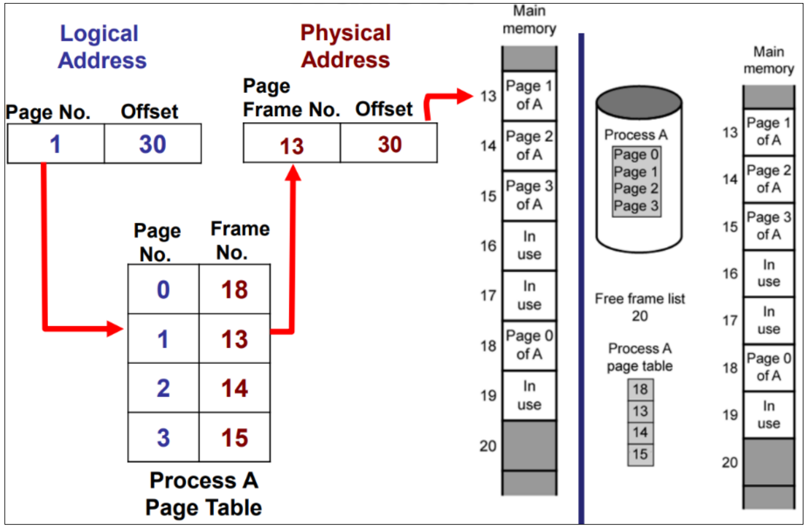


p	Page number
d	Page offset
f	Page frame number

Paging Hardware



Address Translation Scheme idk



Logical/Physical Address Translation

Quick Summary:

- **Uni-program and Multi-program**
- **Swapping:** I/O operation between the hard disk and main memory
- **Partitioning:** split memory into sections to allocate the process; can cause fragmentations/holes
- **Paging:** divide process and memory into page and frames; overcome the problem of holes