Information ∝ Uncertainty ∝ 1/pi

I(x) = log2(1/pi) bits (amt of info received)
N equally probable choices -> M choices (N>M)
I(x) = log2(N/M) bits
EG. N=4, I(x)=log2(1/0.25)=2bits; 4 events (00,01,10,11)
Octal System (8-bit)
- Bin, take 3 bits as 1 octal digit.
EG. $127_8 = 8^2 + 2(8) + 7 = 87$

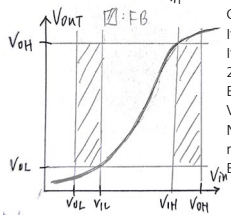
2's Complement $\quad log_b a = log_c a / log_c b$
0011 = 3 | 1100 | 1101 = $-2^3 + 2^2 + 1 = -3$

Bin Range:
Given x bits, encode 2^x choices
Unsigned: 0 to $2^x - 1$ | Signed: -2^{x-1} to $2^{x-1} - 1$

Combinational Device:
1. INPs 2. O/Ps 3. Fn Spec (TT) 4. Tpd

SD: If given valid INP, guarantees valid OP

$V_{IL} - V_{OL} \neq V_{OH} - V_{IH}$ may not



1. |Gain| > 1
Gain=(Voh-Vol)/(Vih-Vil)
If >1: finite NM,
If =1: zero NM
2. Non-Linear Gain
Best if ∞ gain like step fn, Vih & Vil can be shifted st NM↑, ↓ size of INV region
BUFFER; INV -_
SOP: Tpd max 3 gates
Not necc most eff way

MOSFET: Ids ∝ w/l of FET
(drain is usually centre, pfet & nfet converge)
CMOS Recipe: (Else Short Circuit)
1. ALL PU -> PFET; ALL PD -> NFET
2. #PFET = #NFET
3. For every PFET in //, NFET in serie Viceversa
Tpd = Time delay from valid INP to valid OP (max value, only for acyclic circuit)
Tcd = Time delay from invalid INP to invalid OP
For cyclic, signal can propagate back in circuit so using longest path to cal is not accurate.

16 possible 2-inp gates. For x-inp, 2^{2^x} *gates*

Universal Gates:
- INV+OR+AND (SOP)
- NAND | - NOR | - MUX

Building a CMOS Circuit:
- Find **INVERSE** of expr you want
- PULLDOWN: AND: nfet series | OR: nfet //
- PULLUP: opposite to match nfet
- Combine; if given PULLDOWN circuitry to find expression, rmb to **INVERT**

Mux (implemented using logic gates)
- Expensive but universal; hardcodes truth table
- 2^k data INP and k bits sel INP, only 1 O/P

Decoder: (opp of Mux)
- k sel inp, 2^k possible data outputs
- selected output i is HIGH, rest of data output is LOW.

ROM (uses decoder)
- Convention: pulldown -> 1 in TT cus of INV
- For N-inp boolean fn,
size of ROM ≈ 2^N x #outputs

When asked to simplify circuit
1. Boolean exp -> reduction/kmap 2. Demorgan

Must not violate **dynamic discipline** so that we don't store invalid information

DD states that: Tsetup = 2 Tpd | Thold = Tpd
Tsetup = Min time voltage on D needs to be stable BEFORE clock edge changes from 0 to 1
Thold = Min time voltage on wire D needs to be stable AFTER clock edge changes from 0 to 1
Tpd is propagation delay of D-latch
★ For graph: rmb to take input to D instead of overall inp & rmb to inverse when needed!

Sequential Timing:

INP -> CL -> Reg:
For inp: $T_s = T_{pd,CL} + T_{s,Reg}$
$T_H = T_{H,Reg} - T_{cd,CL}$

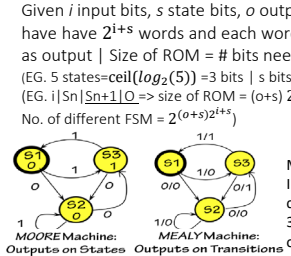
R1 -> CL -> OP:
For whole circuit: $T_{pd} = T_{pd,R1} + T_{pd,CL}$
$T_{cd} = T_{cd,R1} + T_{cd,CL}$

Min CLK Period:
From reg to reg (check all routes for max val)
R1 -> CL -> R2: $T_{CLK,min} = T_{pd,R1} + T_{pd,CL} + T_{s,R2}$

Min Tcd of CL:
$T_{H,down\ reg} < T_{cd,CL} + T_{cd,up\ reg}$
> Ts & Th for INP only concerns paths till 1st reg
> Tpd & Tcd for whole circuit only concerns circuit downstream of last reg b4 output

Metastable State - Properties
> corresponds to a invalid logic level (b/w 0&1)
> unstable equilibrium (smol △ will cause it to accelerate towards stable 0/1)
> will settle to valid 0/1 eventually (may take arbitrarily long)
> EVERY bistable system exhibits at least 1 metastable state
> Inevitable risk of synchronisation (our devices always have a fixed point voltage Vm where Vin=Vm implies Vout=Vm), can only mimimise its probability by **introducing more delays** between INP & output hopefully signal somehow resolves itself b4 reaching output.
- To fix **clock skew** (clock does not arrive at all regs at same time, Tskew is max diff in CLK signal arrival times across all FlipFlops):
R1>CL1>R2: $T_{cd,R1} + T_{cd,CL1} > T_{H,R2} + T_{skew}$
$T_{pd,R1} + T_{pd,CL1} + T_{S,R2} < T_{CLK} + T_{skew}$

FSM: required when same INP produces multiple different O/P
Arcs leaving a state must be:
- Mutually exclusive (no 2 choices for 1 inp)
- Collectively exhaustive (specify OP for all INP)
Equivalence: FSMs are equivalent iff every input sequence yield identical output sequence
Implementing FSM as ROM:
Given *i* input bits, *s* state bits, *o* output bits, we have have 2^{i+s} words and each word has *o* bits as output | Size of ROM = # bits needed for TT
(EG. 5 states=ceil($log_2(5)$) =3 bits | s bits = 2^s states)
(EG. i|Sn|Sn+1|O => size of ROM = (o+s) 2^{i+s} |Outputs No. of different FSM = $2^{(o+s)2^{i+s}}$)



Turing Machine: solves the FINITE problem of FSM
> FSM + doubly ∞ tape | can solve () check
> can read & write at tape in every step
> can solve problems w infinitely many states
y=Ti[x] ; y is output after arrow executes TM func specs; Ti is TM func spec that dictates machine to go left or right according to state; x is int input

★ WRITE FIRST THEN MOVE ARROW

Church's Thesis: **Every discrete fn computable by ANY realisable machine is computable by some TM.**
Halt fn is incomputable; not all well-defined int fn are computable
Universal fn: **U(k,j)=Tk(j)**; k is input (a program to tell us which computable fn Tk we want to compute); j is data tape Tk to perform at.
With Tu, don't need to make so many TM to perform each fn cause it can emulate behaviour of any TM (like our comp)

Von Neumann Model
PC: addr of next inst to be executed (last 2 bits:00)
(INP/OP ↔ CPU ↔ Main Mem)
- Central Processing Unit:
1. ALU 2. Control Unit (interprets comp instr) 3. PC 4. Registers (store temp operands & results)
- Memory: storage of N words of W bits. (W is fixed, N vary) Store data & instr. If a word is 32bit, address can be 32 bit and at most 2^{32} entries. 2^{30} is 1GB so at most 4GB entries. 1KB= 2^{10}B, 1MB= 2^{20}B
- INP/OP: Devices for communicating w outside world | - Bus that connects all 3 together

***β* Instructions**: (Low 2 address bits are ignored)
FOR BRANCHING: (up to 32767 instr b4/after)
const = (label - <addr of BNE/BEQ>)/4 -1
(no of instr lines b/w br & label instr -1)
(not including BR but including first line of instruction of function 'label') | Label = address (of some place)
Literal = constant (cal from label & add (BNE/BEQ))
★ PUSH/POP takes 2 instructions

Interpreter:
- Computes exact instr
- Done after execution; slows program execution
- Decision made during run time, after exec

Compiler:
- Translate high-level lang to low-level assemblr mac lang
- Generate prog to compute
- Done before execution, slows program dev
- Decisions made during compile time

★ **Both serve same purpose but are different!**
Assembler: takes assembler code outputted by either 2 above and translate into binary machine code.
UASM -> interpreted from left to right as least to most significant bytes
. = <address> (lines after this is stored at this address) | (x=...) stored in symbol table |
y: (labels; symbols that take the value of current mem add)
".+4" => "mem location of current instr+4"
write 16-bit (WORD) and 32-bit(LONG) words
Little (Big) Endian: least(most) significant bit is stored at lowest memory address

. = 0x0	0x3	0x2	0x1	0x0
1234	04	03	02	01
	01	02	03	04

(Little used in ***β***) (Big)

Memory device has content & addr, **BOTH** are 32 bits
In ***β***, data always comes in 32 bits (not bytes). The addr of the entire of each row, by convention, is the smallest byte index of that row, converted into 32 bits.
32bit addr can only hold up to 2^{32}byte indexes, cannot house all 2^{32} permutations of 32 bit of info per row, can only house $2^{32}/4$ of possible permutations

C Language: Ref Operator &: address of a variable
Pointers: Variables that store address of a variable
(int var=17; int* p=&var;)
Dereference Op *: Gives value of addr of a pointer
*p=17 | Array name is a fixed pointer to addr of 1st element

Stack & Pointer
- **SP**: points to first unused stack location
- **BP**: points to base of the stack (or first item pushed to the stack, registers)
- **LP**: return address to caller (always same for same recursive call) {usually at deallocate(), can make use of LP location + 4(# lines from deallocate to inst) to find address of instruction)
- **PUSH(Rc):** Add value in Reg[Rc] onto stack (↑SP by 4 & write data to address as pointed by SP-4)
- **POP(Rc):** Pop value on top of stack onto Reg[Rc]
- **Allocate(k):** Reserve k words of stack
- **Deallocate(k):** Release k words of stack
R0 is reserved as the register for the return value of the function getting called

Building the Beta

Classes of Instructions:
1. ALU: OP & OPC
2. LD & ST
3. Branches
4. Exceptions

Register File (RF):
- use of multiplexers (combinational logic) and registers (sequential logic)
- 2 combinational READ ports, 1 clocked WRITE port
Write Enable, Write Address and Write data have to be set Ts before clock rising edge and set Th after clock rising edge. (DD) | ADD(R0,R1,R0) works, read first then write at the end of clock cycle

Data Paths:
WERF - 1 when you wanna write into reg, 0 if not
Output of ALU goes in as Write Data to Register File
Sign Extend 16 bits constant into 32 bits. (Copy bit 16 to [32:17])
BSEL - what goes into B port of ALU, RD2 or SXT(Constant)
WR (1bit) - 0 if read, 1 if write
RA2SEL - select Rb/Rc to go into Read Port 2
WDSEL - select what goes into Write Port of Reg File
ASEL - select what goes into A port of ALU
*PC+4 & C -- combi logic --> (PC+4)+4*SXT(C) [at PCSEL=1]*
Ra (32bit) -- NOR gate --> Z=1 if all bits of Ra = 0
For BEQ: Z=1: branch, PCSEL=1
PCSEL=0 -> Incremented PC value is chosen
PCSEL=2 -> Value of Reg A is chosen
(expression) ? value if true : value if false
EG. implement STX(Rc,Rb,Ra) which is a shortcut for
ADD(Rc,Rb,Rb) Mem[Reg[Ra]+Reg[Rn]] <- Reg[Rc]
ST(Rc,0,Rb) Need to amend data path & reg file
RF needs another RA/RD port which could eliminate RA2SEL.
EG. implement LDX(Ra,Rb,Rc) which is a shortcut for
ADD(Rb,Ra,Ra) Reg[Rc] <- Mem[Reg[Ra]+Reg[Rb]]
LD(Ra,0,Rc) let ALUFN= '+', WERF=1, WDSEL=2, rest=0

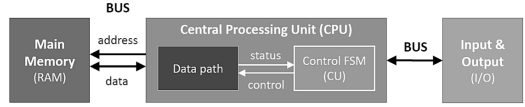
★ BSEL&RA2SEL can be merged if choose apppropriate values for 'dc' entries for RA2SEL

Synchronous/software interrupts:
- Illegal OPCODE in instr word
- Reference to non-existent memory
- Divide by 0

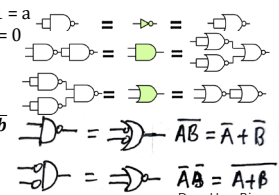
Asynchronous/hardware interrupts:
- User hits a key
- A packet comes via network
- In general: input from I/O hardware device

IllOp: (forced by hardware) PUSH(XP)
fetch inst at Mem[Reg[XP-4], fill result reg
POP(XP)
JMP(XP)

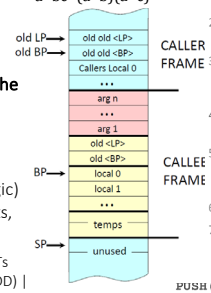
Exception Handling: - Do not execute current instr
- Save current PC (actl PC+4) -> Exception pointer Reg (R30)
- Load PC with exception pointer (0x4 for synch ex, 0x8 for asynch ex, both 32bits)



a+1 = 1 | a+0 = a | a+a = a | a1 = a
a0=0 | aa = a | a+$\bar{a}$ = 1 | a$\bar{a}$ = 0
a+ab=a | a+$\bar{a}$b=a+b
a(a+b)=a | a($\bar{a}$+b)=ab
ab+$\bar{a}$b=b | (a+b)($\bar{a}$+b)=b
DM: $\bar{a} + \bar{b} = \overline{ab}$ | $\overline{ab} = \bar{a} + \bar{b}$
Commutative: a+b=b+a| ab=ba
Associative: (a+b)+c=a+(b+c); (ab)c=a(bc)
Distributive: a(b+c)=ab+ac; a+bc=(a+b)(a+c)



Dec	Hex	Bin
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000



1. Caller pushes arguments onto stack, in reverse order
2. Caller branches to callee, putting return address (current PC) into LP
3. Callee pushes LP & BP, sets BP=SP, allocates local vars, pushes any used registers
4. Callee performes computation, leaving return value in R0
5. Callee pops registers, (deallocates local vars,) sets SP=BP, pops BP & LP
6. Callee branches to return address (LP)
7. Caller pops/deallocates arguments from stack

Calling Sequence

```
PUSH(arg_n)          | push args,
...                  | Last arg first
PUSH(arg_1)
BEQ(R31,f, LP)       | Call f.
DEALLOCATE(n)        | Clean up!
...                  | (f's return value in r0)
```

Entry Sequence

```
f:  PUSH(LP)         | Save LP and BP
    PUSH(BP)         | in case we make new calls.
    MOVE(SP,BP)      | set BP=frame base
    ALLOCATE(nlocals)| allocate locals
    (push other regs)| preserve any regs used
```

Return Sequence

```
    (pop other regs) | restore regs
    MOVE(val, R0)    | set return value
    MOVE(BP,SP)      | strip locals, etc
    POP(BP)          | restore CALLER's linkage
    POP(LP)          | (the return address)
    JMP(LP,R31)      | return.
```

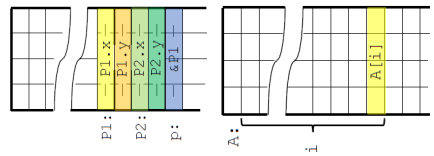
C code:

```
struct Point
{ int x, y; }
P1, P2, *p;

...
P1.x = 157;

...
p = &P1;

p->y = 157;
```

β Assembly:

```
P1: .=.+8
P2: .=.+8
p:  .=.+4
x=0                  | Offset for x component
y=4                  | Offset for y component
...
ADDC(r31,157,r0)     | r0 <- 157
ST(r0,P1+x)          | P1.x = 157
...
ADDC(r31,P1,r3)      | Put &P1 in r3
ST(r3,p)             | Store r3 in p
ST(r0,y,r3)          | p->y = 157;
```

2 int input for point structures so they need 8 bytes (2 words) each
y has an offset of 4 bytes because its one row below p.x.



Note that the interrupted instruction has not been executed. So if the exception handler wishes to execute the interrupted instruction, it will have to subtract 4 from the value in the XP register before performing a JMP(XP) to resume execution of the interrupted program.

Read-only Memory (ROM) (64 location) indexed by 6-bit OPCODE field is easiest way to generate the appropriate control signals for current instruction.
WR always have defined value, ensuring only write to memory during ST inst, and **WERF** is also well-defined except during RESET. During RESET, set WR=0, rest dc.

CPU Design Tradeoffs:
Max Performance: measured by no. of instructions executed per second
MIPS (Millions of Inst/sec) =
Clock Rate (MHz) / Clocks per inst (CPI)
Benchmark Runtime =
#inst executed (in millions) / MIPS
Min Cost: measured by size of circuit
Best Perf/Price: measured by ratio of MIPS to size.
GOAL: Perform with SRAM speed at cost of DISK
Locality of Reference: Reference to memory location X at time t implies that reference to X+△X at t+△t becomes more probable as △X,△t approaches zero
Evidence: local stack frame grows nearby to one another, program inst are near to each other, data (eg. arrays) also nearby one another.

Cache Idea:
- Cache contains temporary copies of selected memory locations A and its content Mem[A].
1. Look for requested info (from CPU) in cache
2. If in cache (hit), return contents to CPU
3. Otherwise (cache miss), look for info in physical memory, and subsequently, the disk

$$T_{avg} = \alpha T_c + (1-\alpha)(T_c + T_m) = T_c + (1-\alpha)T_m$$

α is hit ratio (the amount of information found in cache)
Tc is cache access time and Tm is memory access time.

Fully Associative Cache (FA):
- TAG: all bits of addr A | DATA: all bits of content at A: Mem[A]
- Expensive; SRAM + comparator at each row
- Real Fast: parallel look up
- Flexible: memory + content can be stored on any TAG-DATA row
- Needs **Replacement Strategy** to decide which cache line to write to when cache is full
- Good when cache size is small, Less important when big

Direct Mapped Cache (DM):
- TAG: T-upper bits of A |DATA: all bits of content at A: Mem[A]
- Lower k-bits of A decides which row of DM cache
- Made of SRAMs; but cheaper than FA since only one comparator circuit per DM cache
- No PARALLELISM, but fast mapping between address & cache line index
- Inflexible: a unique combi of K-bits of A is mapped to exactly one row of DM cache
- Suffers from **contention**, 2 different addresses can be mapped to same location if both has same K-lower bits. Selecting K-lower bits for mapping is better than T-upper bits due to LOR but does not completely eliminate contention

$k = log_2(\#rows)$ EG. 3bits if 8 rows
FA cache size = any; DM cache size = 2^k

N-way Associative Cache:
- **N** DM caches in parallel
- Cells in same 'row' belong to same set
- Cells in same 'column' of DM caches belong in same cache line
- Given same k-bits lower address, can be stored in any N cache lines in same set
- However, different combi of k-bits lower address will have to be direct mapped on a different set
- Lookup operation: Parallel op of N-DM cache, each with 2^k lines in cache line.
- M-size N-way cache: no. of rows = M/N; then calculate bits needed
EG. 4-word 2-way => 2 rows, 1 bit

Replacement Strategy:
1. LRU: Least Recently Used
- Good when N is small, need to keep ordered list of N items (N! orderings)
- Overhead = O(log2(N!)) = O(Nlog2(N) "LRU bits" for FA (/set for NW) | Complex logic to re-order list after each cache access
2. FIFO/LRR: Least Recently Replaced
- Replace oldest item
- Overhead is O(log2N) bits/set. (just need one pointer that tells us oldest item within each set; diff pointers for diff sets(NW))
3. Random
- Use pseudo random generator to get reproducible behaviour | Good when N is huge
- Overhead=log2(N)/cache to point to which line to replace
No best replacement strategy, one replacement strategy can be better depending on cases.

Increase Block Size
- Take advantage of locality, reduces size of tag memory
- Increase size of data field, no. of data words in each cache line = block size, always a power of 2.
- Fetch a blocks of words tgt on a cache miss, trading increased cost of miss against increased probability of future hits.
In β architecture, address is 30bits if word addressing is used. **Add 2 extra bits of storage in cache: V & D.**
V: Valid Bit - indicates if particular cache row contains data from memory and not empty or redundant value. Only check cache lines with valid bit=1
D: Dirty Bit - set to 1 iff CPU writes to cache and it hasn't been stored to memory (memory is outdated)
LRU Bit: (Not written) present in each cache line (for FA) and each cell in N-way regardless of block size because R/W with block size more than 1 is always done in parallel.
#bits required for each LRU indicator is log2(N) where N=number of ways
EG. 4-way, 8 rows (4x8 items in total) so 32 LRU cells, each size of 2 bits.

Cache Writes:
1. Write-through
CPU writes are done in cache first by setting TAG=Addr, DATA= new Mem[Addr] in available cache line, but also written to main memory immediately. Stalls CPU until write to memory is complete. Memory never outdated.
2. Write-Behind
CPU writes are also cached, and write to main memory is immediate but is buffered. CPU keeps executing next inst while writes are completed (in order) in background
3. Write-Back (Most used)
CPU writes are also cached, but not immediately written to main memory. MM contents can be 'stale'. When entry in cache, safely just write into cache, leaving memory entry incorrect. But when cache is replaced, write into memory. This requires dirty bit in cache.

OS Multiplexing: At first P1 running, when done, OS interrupts, saves all states of P1 to Mem. OS loads states of P2 to regs and let P2 run task, after finish, interrupt, save, load back P1 states.
Interrupt: (IRQ=1 & PC=0), XP <- PC+4, PC<- Xaddr. Execute interrupt handler at Xaddr.
Beta IRQ Handling: β always check for IRQ b4 each inst fetch. On IRQ(j), stop j, XP <- PC+4, PC<- Xaddr. Xaddr is addrs of handler code, saves user states in corr jth cell of Proctable. After process handling exception returns, reinstall user saved state, return PC to XP-4.
Handler (located at lower mem addr): Reset-0x80000000, Illop-0x80000004, Xaddr-0x80000008 | What if got interrupt call during OS saving user states? Setup proced NEVER interrupts a kernel. If MSB of PC=0, user; allow intrpt. If =1, kernel mode, no intrpt. IRQ not allowed till MSB back to 0.
Programs Comm w OS: known as SVC. Call OS form code by executing particular ILLOP -> user-mode SVC (provide arg in reg, OS returns result in R0). EG. ask for mouse input. (I/O controlled by kernel) | **Sleep:** If buffer empty, kernel exits prog & execute others, will not return till key is pressed. Status 0: active program. When buffer filled, IRQ code for I/O contains wakeup, change status back to 0.

Latency (b/w interrupt and service), task must be done before deadline.
Weak: fixed order, kernel does not stop task upon interrupt.
Strong: higher p bits of PC for priority (3b for 8lvl), handlers w lower priority interrupted **only by higher** priority

Virtual Memory - When looking for particular memory location, look for PPN then PO
- Each program has their own Virtual Memory (all their PCs start from 0x0000 within their own programs)
- VA (addrs generated by programs) | PA: real addrs
- Every VA can be mapped into every PA, but not all VA has corr. PA in the RAM (may be in disk).
Demand Paging: Upon start-up, everything on disk, VA corr to PA on disk.
As programs execute, bring pages into RAM, update MMU st VA corr to PA in RAM
- 1 entry in MMU needed for every virtual page. If R=1, data in RAM, R=0, page-fault exception (no PA in RAM cause data not in RAM). D=1, data to write to disk before removing from RAM
Page Fault: 1. Find page to replace (LRU) 2. Write to disk if dirty 3. Fetch requested page from disk 4. Write to RAM 5. Update MMU
MMU PgTbl in RAM, PgTbl Pointer points to first entry of PgTbl section in RAM. But Slow, so use TLB.
TLB is small, FA cache for mapping certain VPN to PPN. Context# + VPN index rows of TLB, has D,R,PPN.
Context Switching: Add a reg to hold index of current context st don't have to flush TLB when change context.
Case 1: Cache > MMU (Cache stores VA+data) +: no MMU access when HIT, -: need to flush cache when context switch
Case 2: MMU > (Cache>RAM)/Disk (Cache stores PA+data) +: no flush, -: MMU access all the time | if miss in TLB, def not in cache, go to PgTbl.
Case 3: indexing of cache lines use PO, indexing of PgTbl use VPN. Each cache line stores single data (not page) & addrs of data. TAG of cache=PO, content=PA. PA from cache checked with PPN (from MMU) + PO. **MMU access and cache access done in parallel.** If cache miss, machine immediately fetch data from either mem/disk, update MMU & cache at same time.

Scheduling: 1) If finite dL, check if period > completion time. If period shorter, cannot be solved
2)Check if period < dL, if yes: period is 'realistic' dL | 3) Check %Load (SVC time (s) x freq (1/s))
Qn Weak **priority**, what's the max svc time of N | look at task w shortest gap b/w svc time & dL (eg task X) max svc time of N = dL(x) – SVC(x) [if N is ongoing, others cannot interrupt, may miss dL]
Qn Strong Priority (T>N>A>J>C) (1) N meets its dL: in N's period, how often T happens, max svc time = dL/period - #(T) | (2) A meets its dL: in A's period, T&N happened #times. (A'sdL/P - #(T) - A)/#(N) | (3) J
(4) dc bout C cus no dL **Qn Is weak scheduling able to meet constraints?**: Check that if a particular task cannot be interrupted, the svc time of this task cause others to miss dL. If YES, cannot be solved.

★ Using our procedure linkage convention, how far (in terms of addresses) can a callee be from its caller? > The entry point of the callee must be reachable by a branch, i.e., it must be within approximately 2^15 words of the call instruction. Can overcome by using LDR to load target address into a reg and the JMP using the register.
★ If there is a bound on the no. of states, you can discover its behaviour. For a k-state FSM, every state can be reached in < k steps.
★ If storage for variable is located at address more than 0x8000, (add needs to fit in 15bit) 16-bit constant field isn't large enough so can use LDR to load a 32-bit address into a reg and use LD to fetch data. EG. var: LONG(0x12468)... LDR(var,R0) LD(R0,0,R0)
★ Checking if circuit can be made:
if unbounded time: Yes, can make
Else: if arbiter: NO
else: YES
★ Branches (b) use a PC-relative displacement while jumps (j) use absolute addresses.
★ Shift left i bits ⇒ x 2^i (shift right ⇒ ÷ 2^i)
★ BR address that made the call f() is the value of LP-4.
★ R1->CL->R2: If you had a faster vers of CL with tcd=0, can you substitute? Check $T_{H,R2} \leq T_{cd,R1} + T_{cd,CL}$. After CLK change, input to R2 needs to hold for $T_{H,R2}$.
★ New state for registers -> wait 1 clock cycle for the D to pass to Q.
★ Universal FSM will have some fixed number of states built into its design, won't have enough states to emulate machines with more than N states.
★ CPU can only have finite registers cause registers must be encoded in instructions
★ Cannot enumerate functions of continuous variables (eg sinx). There are only 16 2inp combi fn but infinitely many continuous 2-inp functions.
★ TMs differ only in their FSMs.
★ Contents of memory location holding instr: 32 bits instruction code
★ If MOVE(BP,SP) were deleted, procedure would work just fine
If DEALLOCATE is deleted, even tho stack is reset correctly by MOVE(BP,SP), that happens after POP(R1) and POP(R2) so they will not restore the values of R1 and R2 from values pushed onto stack during entry sequence.
★ What is the value of BP at the time the stack snapshot was taken? old BP + (no of words in a stack)
★ What is the worst case delay for states A&B to be valid after inp x is changed to be valid?
= tpd (x to reg) + 1/clockrate
★ Computer is not a turing machine, is a FSM

Macro	Definition
BEQ(Ra, label)	BEQ(Ra, label, R31)
BF(Ra, label)	BF(Ra, label, R31)
BNE(Ra, label)	BNE(Ra, label, R31)
BT(Ra, label)	BT(Ra, label, R31)
BR(label, Rc)	BEQ(R31, label, Rc)
BR(label)	BR(label, R31)
JMP(Ra)	JMP(Ra, R31)
LD(label, Rc)	LD(R31, label, Rc)
ST(Rc, label)	ST(Rc, label, R31)
MOVE(Ra, Rc)	ADD(Ra, R31, Rc)
CMOVE(c, Rc)	ADDC(R31, c, Rc)
PUSH(Ra)	ADDC(SP, 4, SP) ST(Ra, -4, SP)
POP(Rc)	LD(SP, -4, Rc) SUBC(SP, 4, SP)
ALLOCATE(k)	ADDC(SP, 4*k, SP)
DEALLOCATE(k)	SUBC(SP, 4*k, SP)

C code:
```
int A[100];
...
A[i] += 1;
```

β Assembly:
```
A:    .=.+4*100    | Leave room for 100 ints
      ...
      LD(i,r1)
      MULC(r1,4,r2)     | index -> byte offset
      LD(r2,A,r0)       | A[i] -> R0
      ADDC(r0,1,r0)     | increment
      ST(r0,A,r2)       | A[i] <- R0
```