Information resolves uncertainty
Encoding-assign representations of info

Quantifying Information

- Given a discrete random variable X
  - With N possible values, and $x_1, x_2, \ldots x_N$
  - Associated probabilities $p_1, p_2, \ldots p_N$
- Information received given x_i:

$$I(x_i) = log_2\left(\frac{1}{p_i}\right) \text{ bits}$$

N choices —Narrow to→ M choices

$$log_2\left(\frac{N}{M}\right)$$

In information theory, entropy H(X) is the average amount of information contained in each piece of data received about the value of X:

$$H(X) = E(I(X)) = \sum_{i=1}^{N} \underbrace{p_i}_{\text{Probability of occurrence of i}} \cdot \underbrace{log_2\left(\frac{1}{p_i}\right)}_{\text{Information } I(x_i)}$$

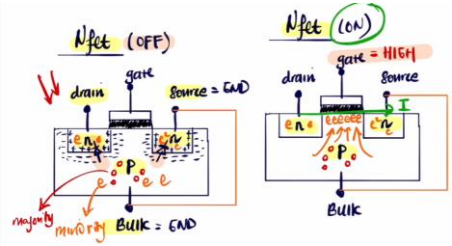
Entropy is the **lower bound** on the number of bits we need to represent this information.

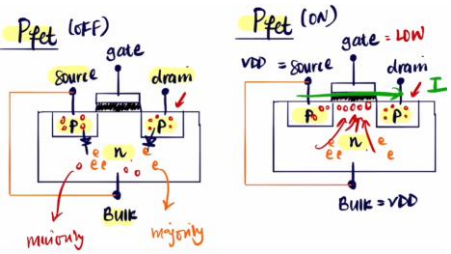
A combinatorial device is a circuit element that has:

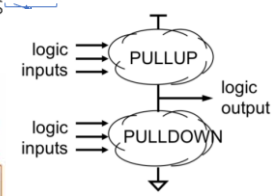
Static discipline
1. One or more digital inputs
2. One or more digital outputs
3. A functional specification that details the value of each output for every possible combination of valid input values(truth table)
4. A timing specification consisting of an upper bound t_{pd} on the required time for the device to compute the specified output values from an arbitrary set of stable valid input values

Proposed fix: separate specifications for inputs and outputs
- Digital output: "0" <= V_{OL}, "1" >= V_{OH}
- Digital input: "0" <= V_{IL}, "1" >= V_{IH}

Gain = $(V_{oh}-V_{ol})/(V_{ih}-V_{il})$

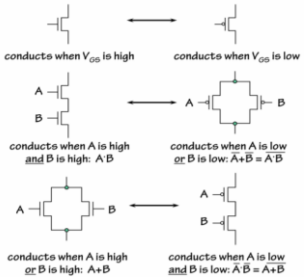






Pullup: All Pfets
Pulldown: All Nfets

CMOS complements

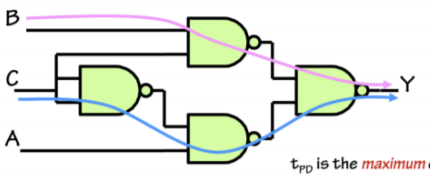


t_{pd}: time delay from valid input to valid output(max)
t_{cd}: time delay from invalid input to invalid output(min)

t_{PD} = 12 nS

t_{CD} = 2 nS

t_{CD} is the *minimum* cumulative contamination delay over all paths from inputs to outputs



t_{PD} is the *maximum* cumulative propagation delay over all paths from inputs to outputs

Absorption: $a+ab=a$, $a+\bar{a}b=a+b$
$a(a+b)=a$, $a(\bar{a}+b)=ab$

Reduction: $ab+\bar{a}b=b$, $(a+b)(\bar{a}+b)=b$

DeMorgan's Law: $\overline{a+b}=\bar{a}\bar{b}$, $\overline{ab}=\bar{a}+\bar{b}$

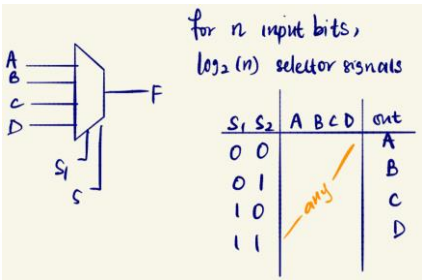
For x-input devices-2^{2^x} *possible devices(gates)*

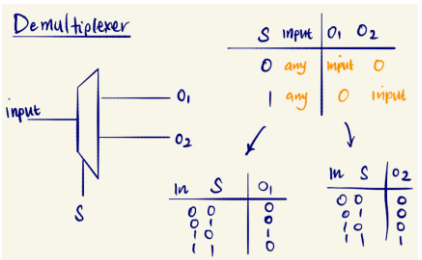
Recall that the contamination time is the period of output validity after the inputs have become invalid. So for nand2:

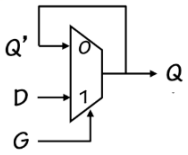
$t_{C\text{-}FALL}$ = time elapsed from when input > V_{IL} to when output < V_{OH}
$t_{C\text{-}RISE}$ = time elapsed from when input < V_{IH} to when output > V_{OL}
$t_C = \min(t_{C\text{-}RISE}, t_{C\text{-}FALL})$

Similarly the propagation time is the period of output invalidity after the inputs have become valid. So for nand2:

$t_{P\text{-}RISE}$ = time elapsed from when input ≤ V_{IL} to when output ≥ V_{OH}
$t_{P\text{-}FALL}$ = time elapsed from when input ≥ V_{IH} to when output ≤ V_{OL}
$t_P = \max(t_{P\text{-}RISE}, t_{P\text{-}FALL})$

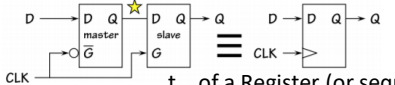




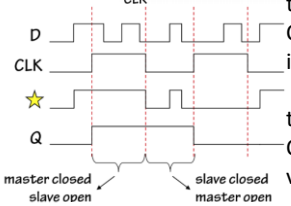


D-latch
1: Write mode, input D
0: Memory mode, input Q'
Dynamic discipline:
t_{setup}= $2t_{PD}$
t_{hold}= t_{PD}
t_{setup}=min. time the voltage on wire D needs to be stable before the clock edge changes from 1 to 0
t_{hold} =min. time the voltage on wire D needs to be stable after the clock edge changes from 1 to 0.

How to reduce FSM states:
1. Two states S_i and S_j are identical if
   (a) States have identical output
   (b) **Every** input ends (transit to) equivalent states
2. Find pairs of equivalent states, merge them



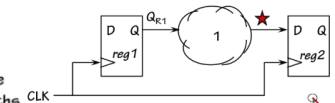
Register/Flip Flop



t_{CD} of a Register (or sequential logic unit with registers and CLs in it) is the time taken for invalid CLK input to produce an invalid final output

$$t_{CD_{master}} > t_{hold_{slave}}$$

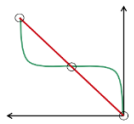
t_{PD} of a Register (or sequential logic unit with registers and CLs in it) is the time taken for valid CLK input to produce a valid final output

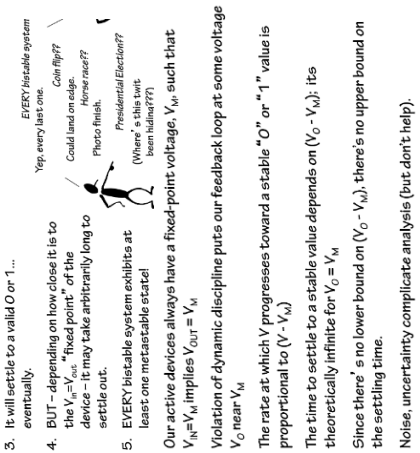


$t_{CD,reg1} + t_{CD,1} > t_{HOLD,reg2}$

$t_{PD,reg1} + t_{PD,1} < t_{CLK} - t_{SETUP,reg2}$

Event of storing invalid value is called the metastable state, as shown in the figure below





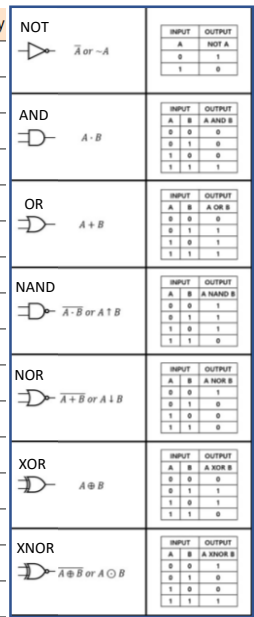
To fix metastable, introduce more delay

Clock skew is the max diff in clock signal arrival times across all flip flops

$$t_{CD,Reg1} + t_{CD,CL1} > t_{hold,Reg2} + t_{skew}$$

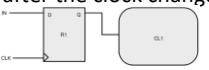
$$t_{PD,Reg1} + t_{PD,CL1} + t_{SETUP,Reg2} < t_{CLK} + t_{skew}$$

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



t_{setup}:time taken for the input to be stable before the clock changes from 1 to 0

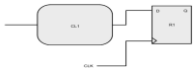
t_{hold}:time taken for the input to be stable after the clock changes from 1 to 0



Input->Register

$t_S = t_{S,R1}$

$t_H = t_{H,R1}$



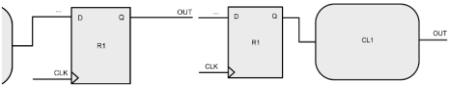
Input->CL->Register

$t_S = t_{PD,CL1} + t_{S,R1}$

$t_H = t_{H,R1} - t_{CD,CL1}$

t_{PD}: time taken to produce a valid output after the CLK rise turns valid

t_{CD}: time taken to produce a invalid output after the CLK rise turns invalid



Register->Output

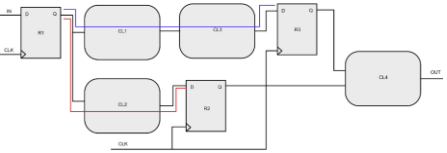
$t_{PD} = t_{PD,R1}$

$t_{CD} = t_{CD,R1}$

Register->CL->Output

$t_{PD} = t_{PD,R1} + t_{PD,CL1}$

$t_{CD} = t_{CD,R1} + t_{CD,CL1}$



$t_{BLUE} = t_{PD,R1} + t_{PD,CL1} + t_{PD,CL3} + t_{S,R3}$

$t_{RED} = t_{PD,R1} + t_{PD,CL2} + t_{S,R2}$

Min CLK period = max(t_{BLUE}, t_{RED})

FSM:

Moore: output is drawn on states and depends on states

Mealy: output is drawn on transitions arcs and depends on both inputs and states

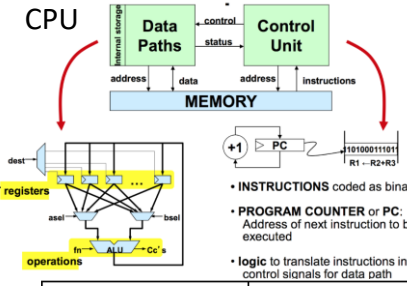
FSM as programmable machine

inputs i	s_N	outputs s_{N+1}	o
0...00	0...00	10110	011
0...01			

2^{i+s} words(input-state combo)

$2^{(0+s)(2)^{(i+s)}}$different FSM(in red box)

CPU



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)

SP points to the available memory location to write to (first unused stack space).

BP points to the base of the stack, or equivalently first item pushed to the stack.

Reg	Sym	Usage
R31	R31	Always zero
R30	XP	Exception Pointer
R29	SP	Stack Pointer
R28	LP	Linkage Pointer
R27	BP	Base of frame Pointer

Von neumann model

1. Central Processing Unit (CPU): containing several registers, as well as logic for performing a specified set of operations on their contents.
2. Memory: storage of N words of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.
3. Input/ Output: Devices for communicating with the outside world.
4. Connection bus that connects all the three components together.

1: Calling Sequence – Arguments

CMOVE(4,R1)

PUSH(R1)

2: Calling Sequence- Branching and Cleanup

BR(square,LP)

DEALLOCATE(1)

HALT()

3: Entry Sequence

PUSH(LP)

PUSH(BP)

MOVE(SP,BP)

Step 4: The Actual Code

PUSH(R2)

LD(BP,-12,R2)

MUL(R2,R2,R0)

5: Exit Sequence -Pop Regs from Actual Code pops whatever register that was used

POP(R2)

6: Standard Exit Sequence

MOVE(BP,SP)

POP(BP)

POP(LP)

JMP(LP)

BR(square, LP) . This means to store the value of PC + 4 to register R28/ LP if R31 is equal to zero (which is always true since R31 is reserved register to 0), and then move PC to the address of 'square

Memory address: 0x168 The initial content of SP is 0. When ALLOCATE(90) is executed, it increases the content of SP by 90 * 4 = 360. Therefore, the stack starts at address 0x168 (in hex).

literal computation of 'label':

(a) Count how many lines of instructions are there between BR and the first line of instruction of the function 'label' (not including BR but including that first line of instruction of function 'label')

(b) There's 2 lines of instructions for each PUSH or POP, and 1 line of instruction for each of every other β instruction.

(c) So the first instruction of function 'square' is actually 3 lines away from BR(Square, LP), 1 line from DEALLOCATE(1), 1 line from HALT(), and 1 line from the first instruction of PUSH - which is ADDC.

The literal (in the 32-bit machine code) is this number (the number of instructions from BR to the first instruction of the function 'square') subtracted by 1, hence the literal is 2.

BP-12 is always the address of the first argument. If there's a second, third argument, then the address of it in the stack will be BP-16, BP-20, etc.

Always PUSH arguments in REVERSE order, meaning push argument n, then argument n-1, then argument n-2, ..., and finally argument 1.

address of "array"? 0x000000EC (59 lines of instructions/ content above "array")

6. A memory contains both instructions and stack
7. Registers are in the CPU, not part of the memory (RAM)
8. We only have limited registers, and the ALU (logic unit) can only access registers to perform computations, so we need stack in the memory, i.e: a temporary space to perform computation and execute function code.
9. Each time you call a function (this case, is function square), the stack grows (arguments pushing, entry sequence, and registers pushing) When the function returns (or ends), the stack diminishes (exit sequence, and pops)
11. When you PUSH(Rx), you are storing the CONTENT of Rx into the memory to the address pointed by SP
12. When you POP(Rx), you are loading the CONTENT of the memory with address pointed by SP back to register Rx

Procedure Linkage

typical "boilerplate" templates

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
```

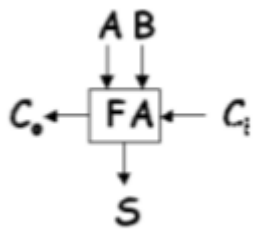
Return Sequence

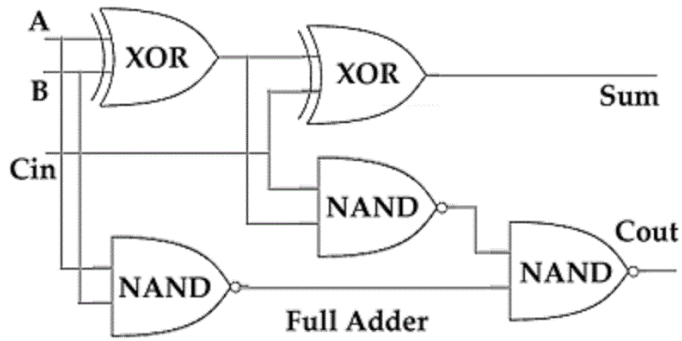
```
(pop other regs)     | restore regs
MOVE(val, R0)        | set return value
MOVE(BP,SP)          | strip locals, etc
POP(BP)              | restore CALLER's linkage
POP(LP)              | (the return address)
JMP(LP,R31)          | return.
```

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.
Done after execution	Done before execution
Decisions are made during run time, after execution	Decisions are made during compile time, before execution
Slows program execution	Slows program development

Reg	Sym	Usage
R31	R31	Always zero
R30	XP	Exception Pointer
R29	SP	Stack Pointer
R28	LP	Linkage Pointer
R27	BP	Base of frame Pointer

C_i	A	B	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





$$S = A \oplus B \oplus C_{in} \qquad C_o = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$$