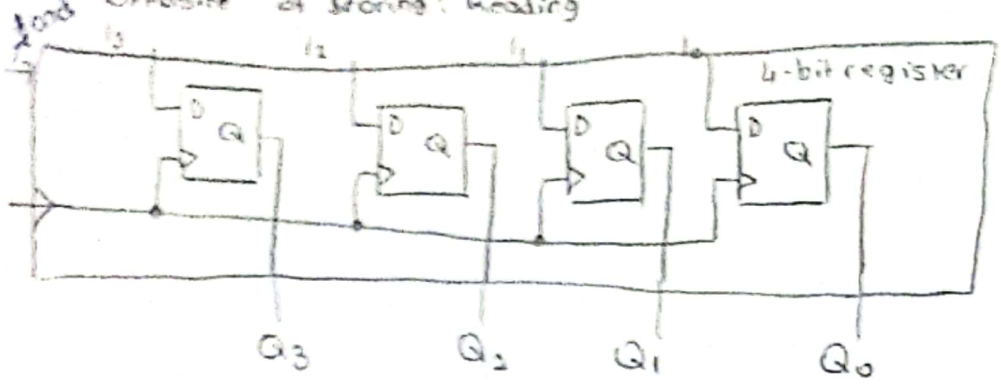
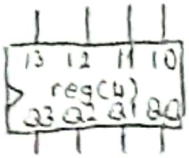
Registers

- N-bit register: Stores N bits, N is the width
- Common widths: 8, 16, 32
- Storing data into register: Loading
- Opposite of storing: Reading



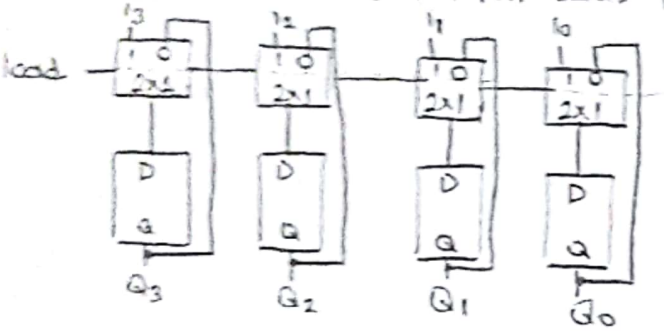
→



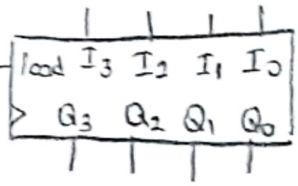
* Basic register loads on every clock cycle
  How extend to only load on certain cycles?

- Register with Parallel Load -

⇒ Add 2x1 mux to front of each flip-flop
⇒ Register's load input selects mux input to pass
- load = 0: existing flip flop value; load = 1: new input value



→



block symbol

Shift Register → (Division)

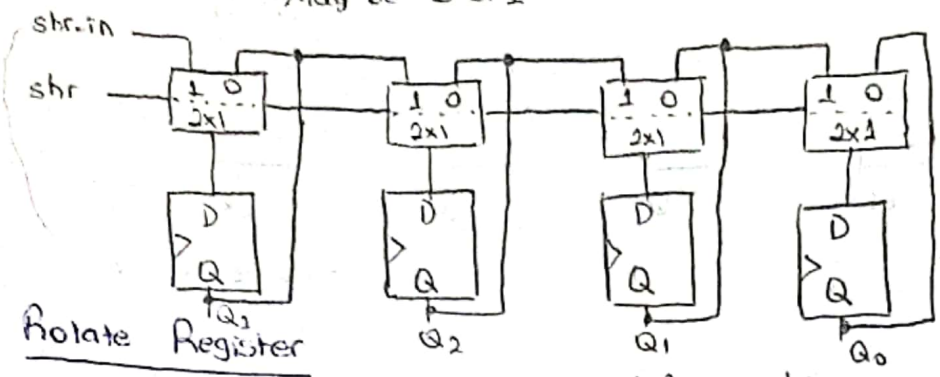
- Shift right
  - move each bit one position right
  - rightmost bits dropped
  - assume 0 shifted into leftmost bit

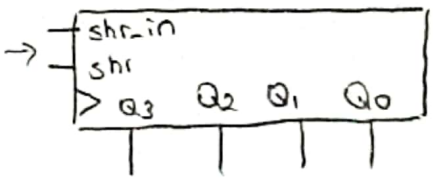
Q1 Do four right shifts on 1001 showing value after each shift

A: 1001 (original)
0100
0010
0001
0000

- Implementation: Connect flip-flop output to next flip-flop's input

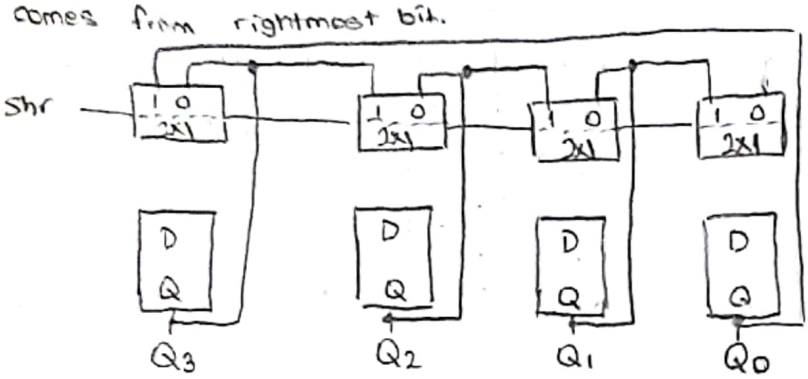
- To allow register to either shift or retain, use 2x1 muxes
  - shr : "0" means retain, "1" shift
  - shr_in : value to shift in
    - May be 0 or 1



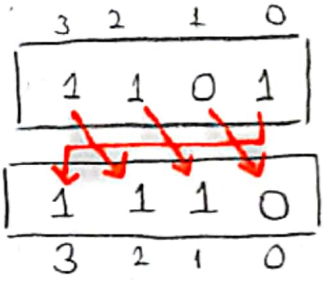


Rotate Register

- Rotate right : Like shift right, but leftmost bit comes from rightmost bit.

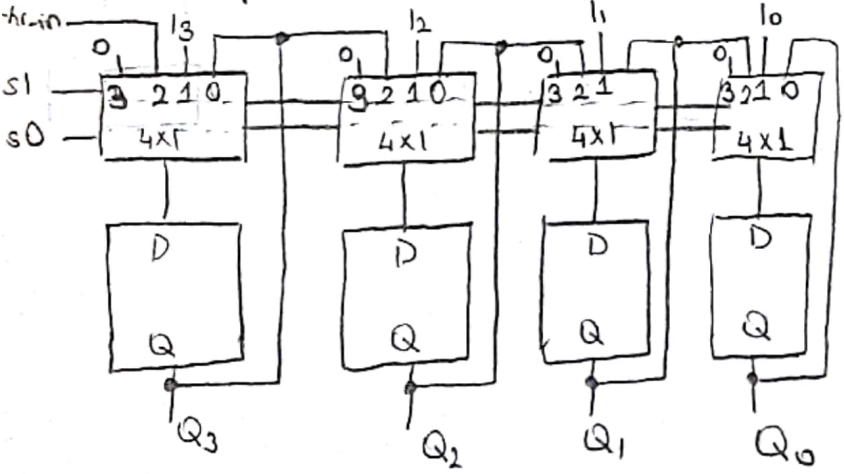


⇒ instead of shr_in use Q0 last bit.



Multifunction Registers

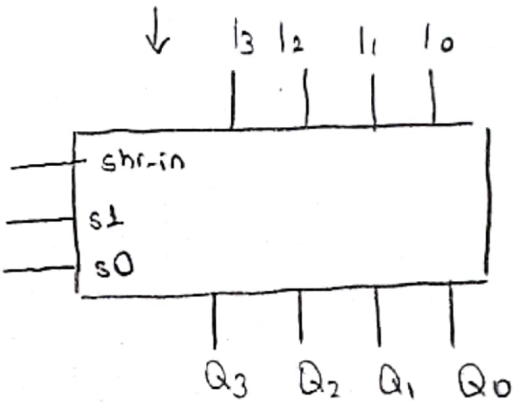
* from number of operation we will understand how many bits stored for select bit!



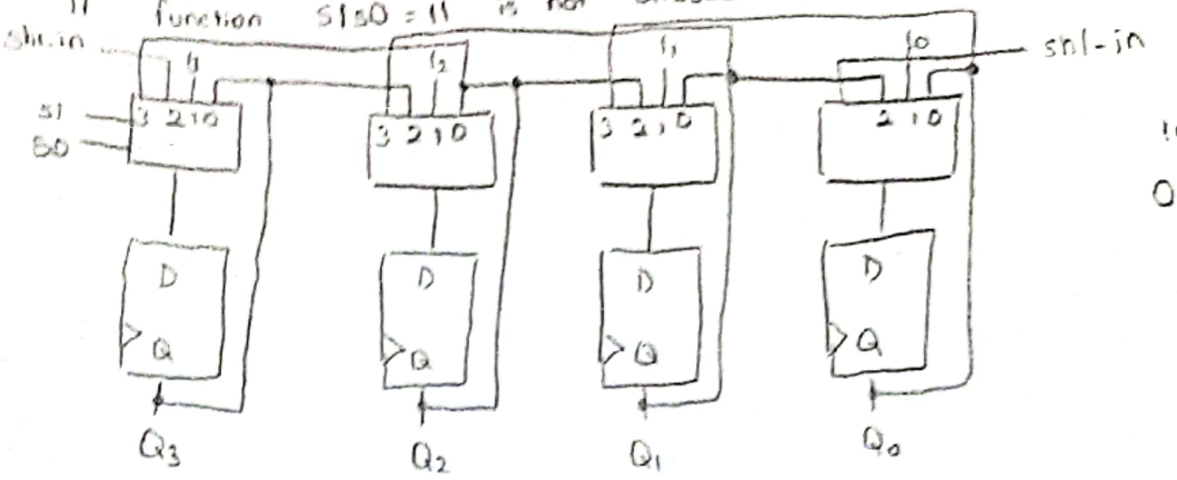
it should be always here

Functions:

S1	S0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	(unused - let's load Qs)



If function S1S0 = 11 is not unused it is shift left:



1010
0101

Multifunction Registers with Seperate Control Inputs

* To apply these priorities we need combinational circuit.

Truth table for combinational circuit;

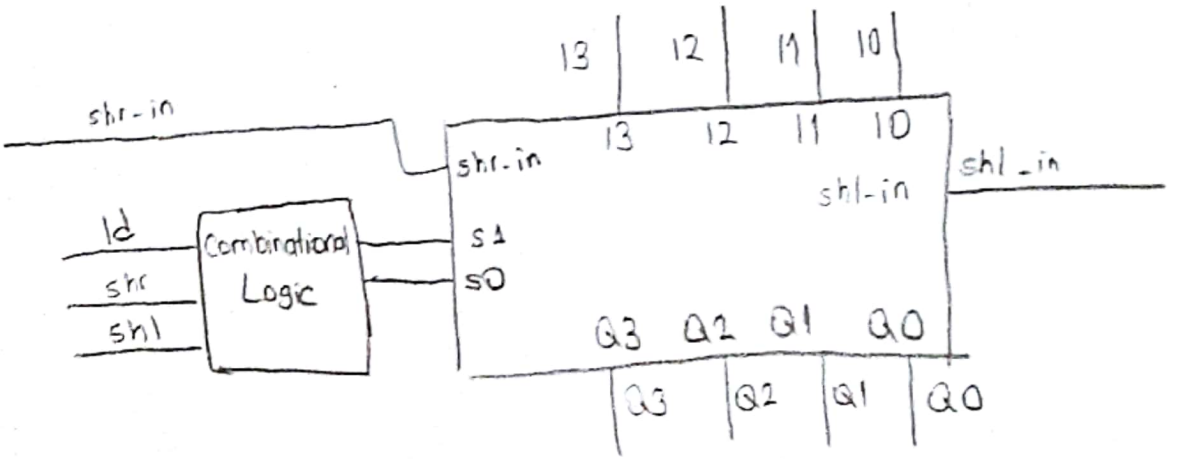
Inputs ld	shr	shl	Outputs S1	S0	Operation
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load

$S1 = ld' * shr' * shl + ld' * shr * shl' + ld' * shr * shl$

$S0 = ld' * shr' * shl + ld$

ld	shr	shl	Operation
0	0	0	Maintain present value
0	0	1	Shift left
0	1	0	Shift right
0	1	1	Shift right - shr has priority over shl
1	0	0	Parallel load
1	0	1	Parallel load - ld has priority
1	1	0	Parallel load - ld has priority
1	1	1	Parallel load - ld has priority

S1	S0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	Shift left



-Register Design Example-

- Desired register operations
  - load, shift left, sychronous clear, sychronous set
  - want unique control input for each operation

Step 1: Determine mux size:

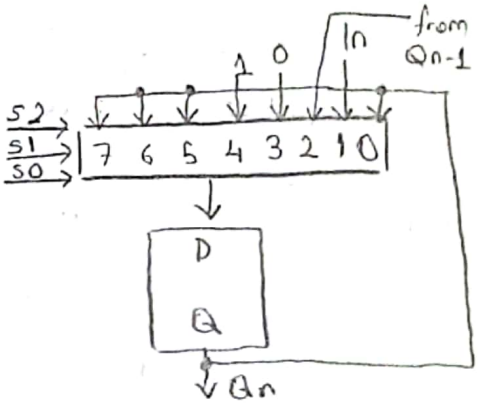
5 operations: above, plus maintain preserve value (don't forget this one!)

→ Use 8x1 mux

Step 2: Create mux operation table

s2	s1	s0	Operation
0	0	0	Maintain present value
0	0	1	Parallel load
0	1	0	Shift left
0	1	1	Sychronous clear
1	0	0	Sychronous set
1	0	1	Maintain present value
1	1	0	Maintain present value
1	1	1	Maintain present value

Step 3: Connect mux inputs



Step 4: Map control lines

Inputs clr	set	ld	shl	Outputs s2	s1	s0	Operation
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s

↳ s2 = clr'*set

s1 = clr'*set'*ld'*shl + clr

s0 = clr'*set'*ld + clr

Adders

- Adds two N-bit binary numbers

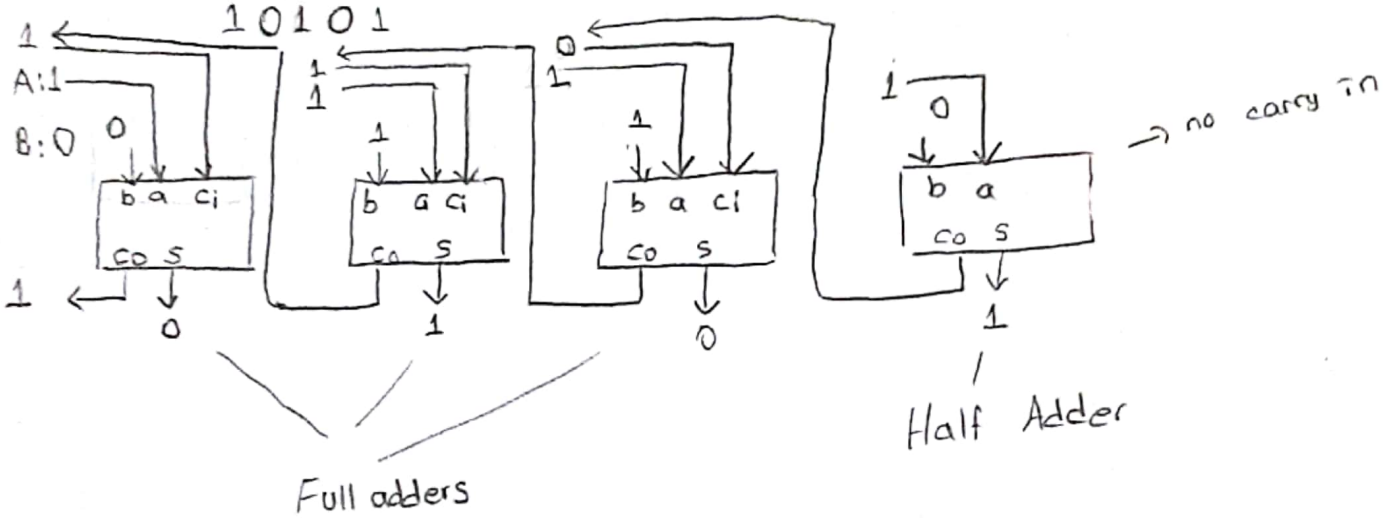
-2 bit adder: adds two 2-bit numbers outputs 3-bit result.

01 + 11 = 100 (1+3=4)

⇒ Alternative adder design: mimic how people do addition by hand.

```
  1 1 0
A: 1 1 1 1
B: 0 1 1 0
-----------
1 0 1 0 1
```

→ Create component for each column



Half adder:

Step 1: Capture the function

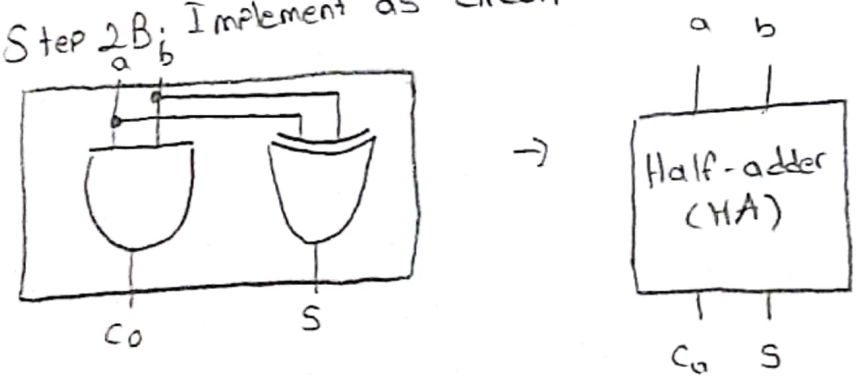
Inputs		Outputs	
a	b	co	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Step 2A: Create equation

→ $Co = ab$

→ $S = a'b + ab'$ (same as S = a xor b)

Step 2B: Implement as circuit



Full Adder

- Full adder : adds 3 bits generates sum and carry.

Step 1: Capture the function

Inputs			Outputs	
a	b	ci	co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Step 2A: Create equations:

$co = a'bc + ab'c + abc' + abc$

$co = a'bc + abc + ab'c + abc + abc' + abc$

$co = bc(a'+a) + ac(b'+b) + ab(c'+c)$

$\boxed{co = bc + ac + ab}$

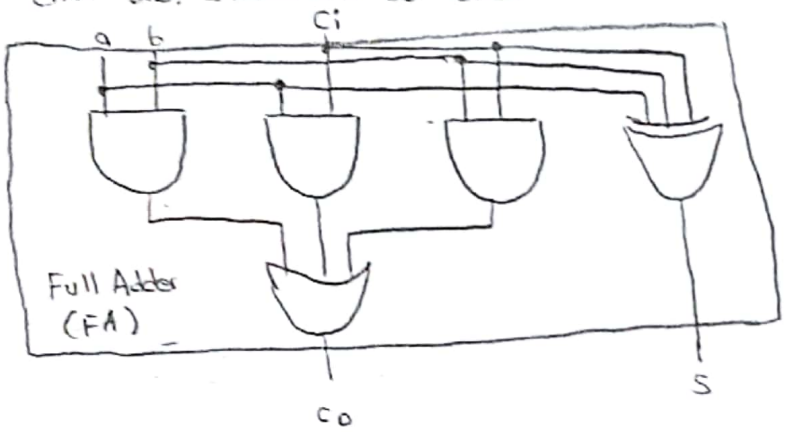
$S = a'b'c + a'bc' + ab'c' + abc$

$S = a'(b'c + bc') + a(b'c' + bc)$

$S = a'(b \text{ xor } c)' + a(b \text{ xor } c)$

$\boxed{S = a \text{ xor } b \text{ xor } c}$

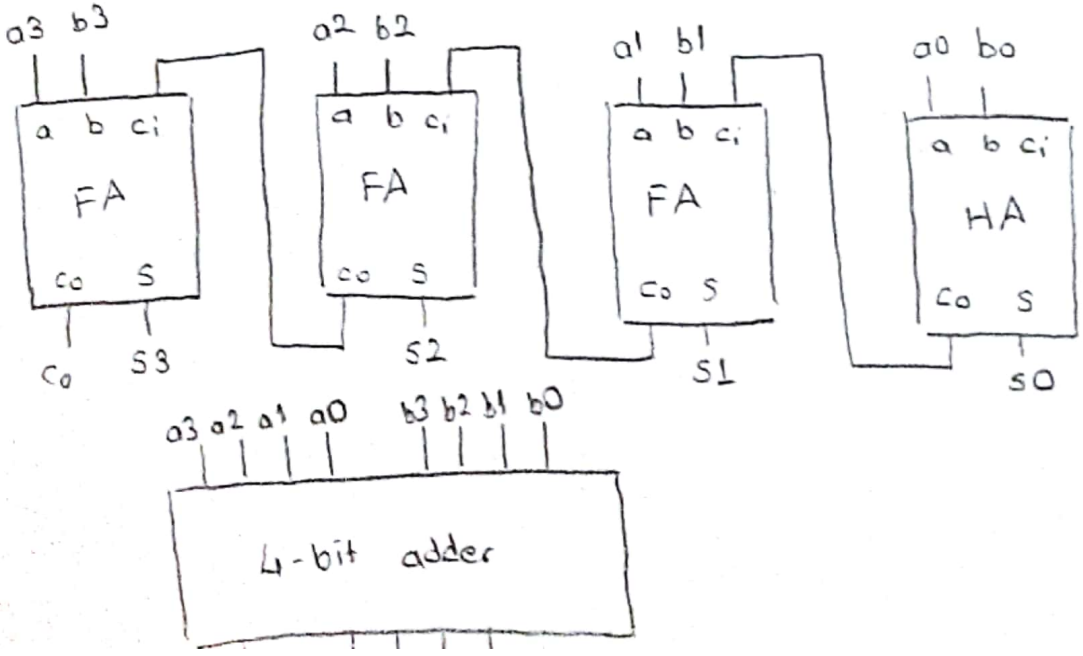
Step 2B: Implement as circuit



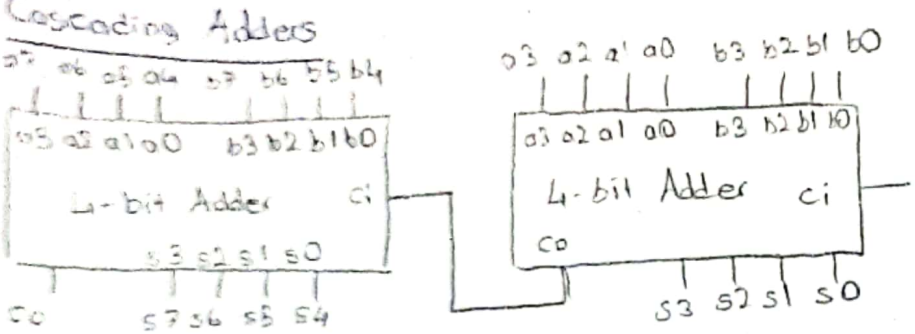
Carry Ripple Adder

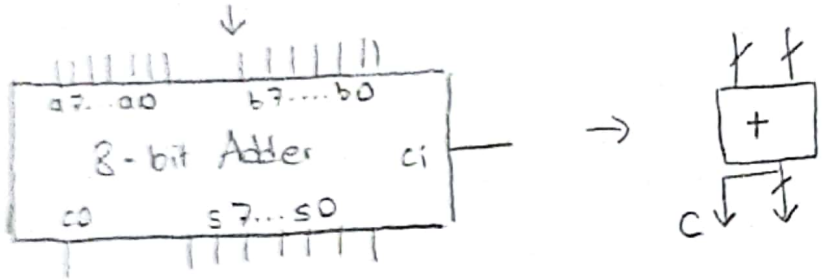
→ Using half adder and full adders, we can build adder that adds like we would by hand, called a carry-ripple adder.

- 4 bit adder shown : Adds two 4-bit numbers, generates 5-bit output.



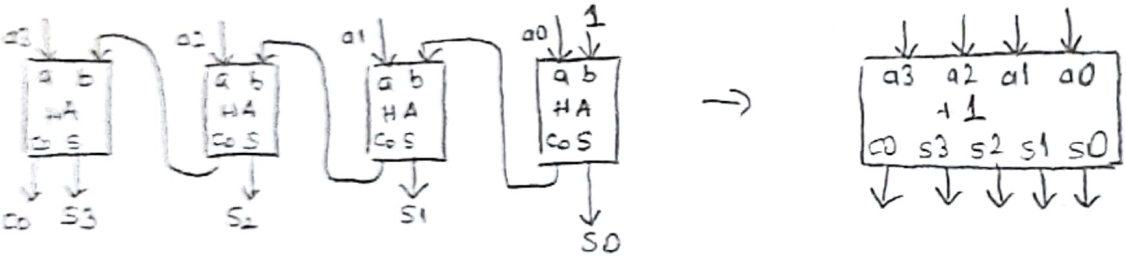
Cascading Adders





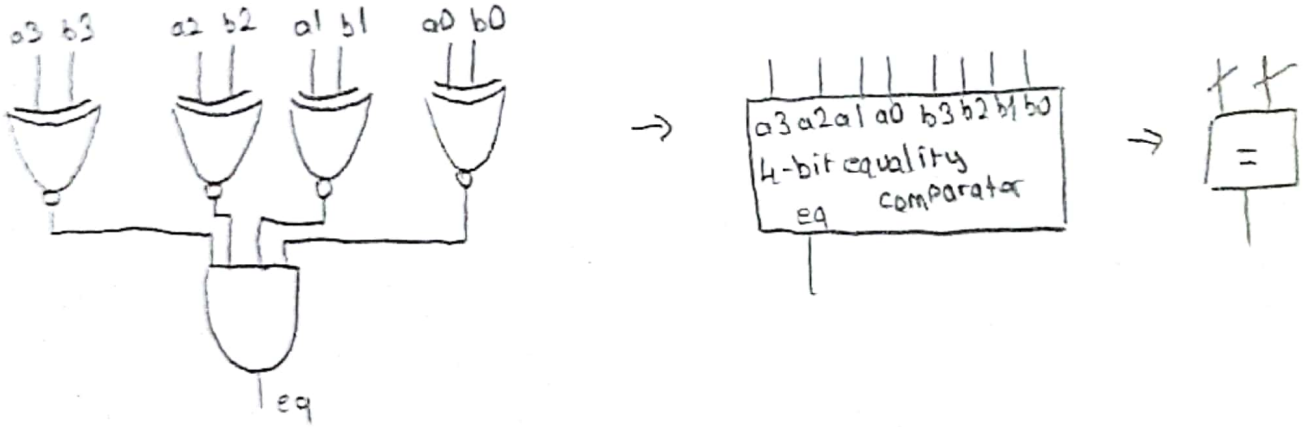
Incrementer

- Adds 1 to input A



Comparators

- N-bit equality comparator. Outputs 1 if two N-bit numbers are equal
  - 4 bit equality comparator with inputs A and B
    - a3 must equal b3, a2=b2, a1=b1, a0=b0
    - Two bits are equal if both 1, or both 0
    - eq = (a3b3 + a3'b3')*(a2b2 + a2'b2')*(a1b1 + a1'b1')*(a0b0 + a0'b0')
  - Note that function inside Paranthesis is XNOR
    - eq = (a3 XNOR b3)*(a2 XNOR b2)*(a1 XNOR b1)*(a0 XNOR b0)



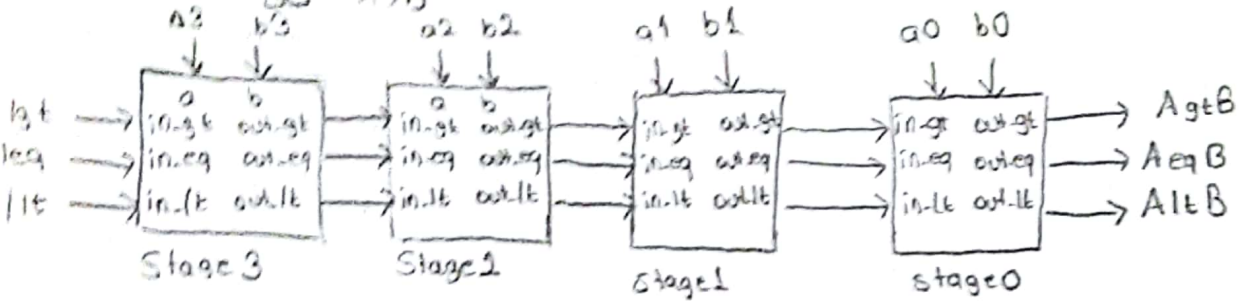
Magnitude Comparator

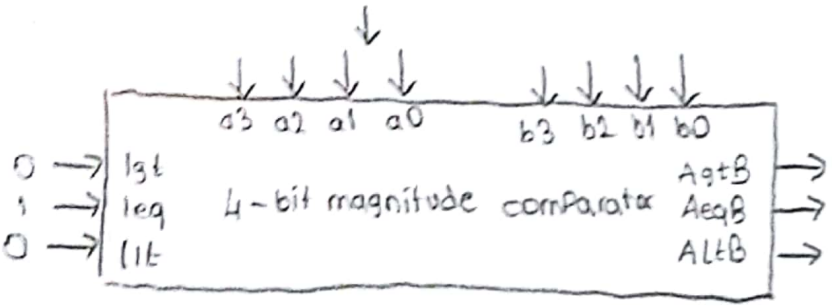
- N-bit magnitude comparator
  Takes N-bit inputs A and B, outputs whether A>B, A=B or A<B

A = 1011 B = 1001

1011 1001 Equal
1011 1001 Equal
1011 1001 Not Equal

A>B

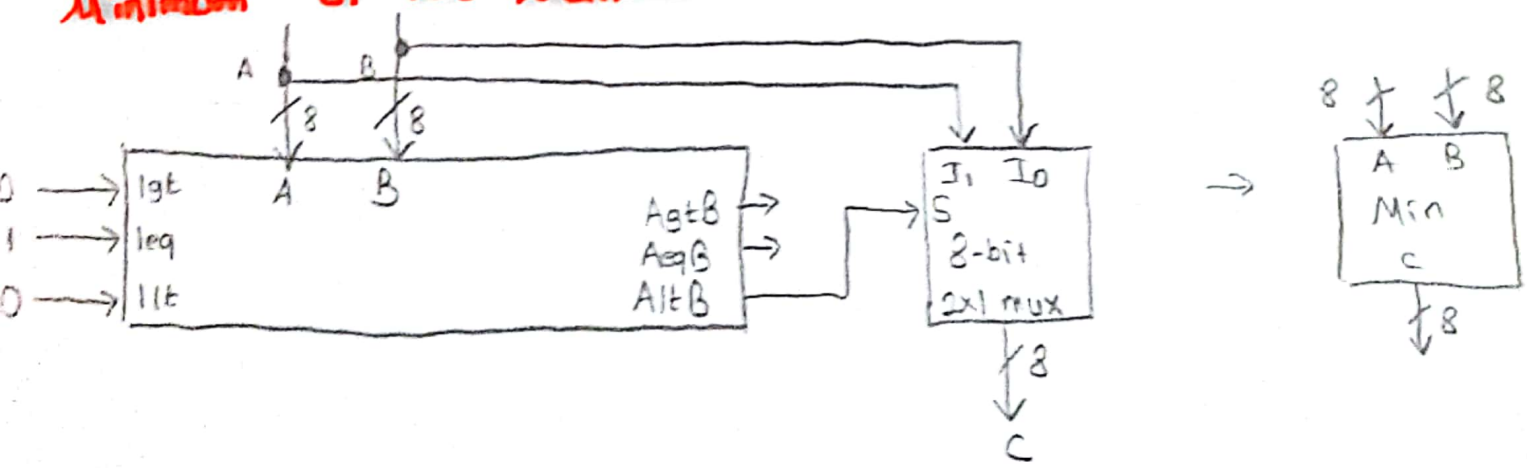




Each stage:

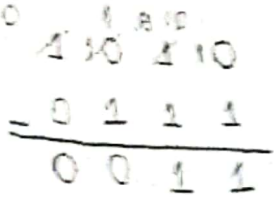
- out_gt = in_gt + (in_eq * a * b')
- out_lt = in_lt + (in_eq * a' * b)
- out_eq = in_eq * (a XNOR b)

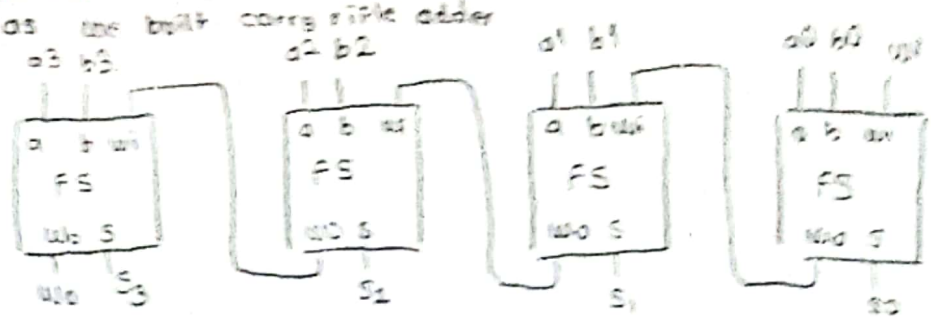
Minimum of two Numbers



Subtractors and Signed Numbers

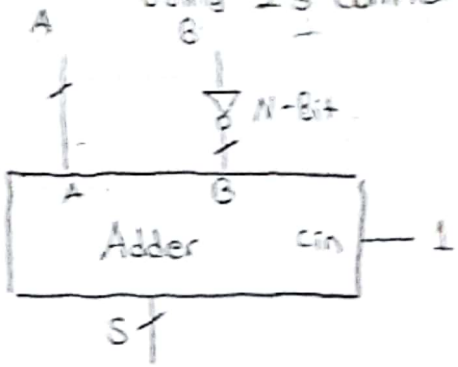
- Can build subtractor as one built carry ripple adder





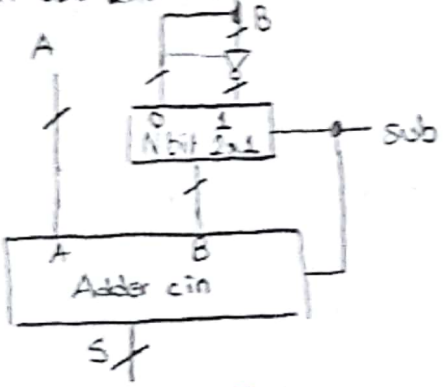
OR

Using 2's complement



Adder / Subtractor

- Adder/subtractor: control input determines whether add or subtract
  - Can use 2x1 mux - sub input passes either B or inverted B

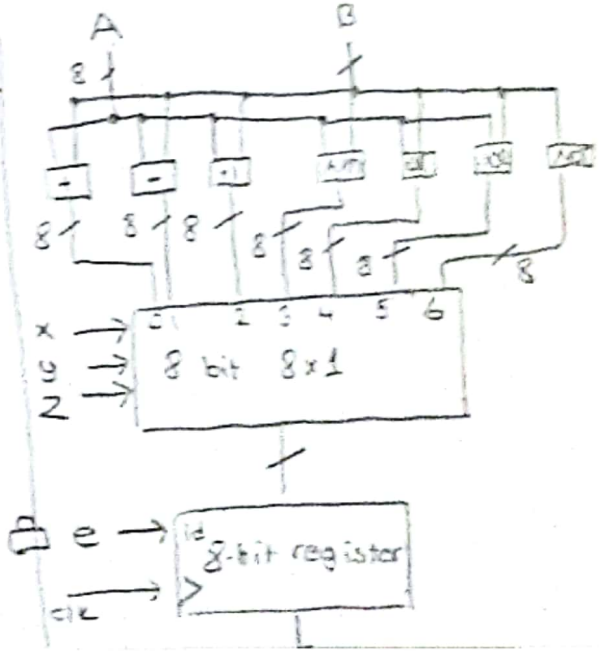


- Arithmetic Logic Unit -

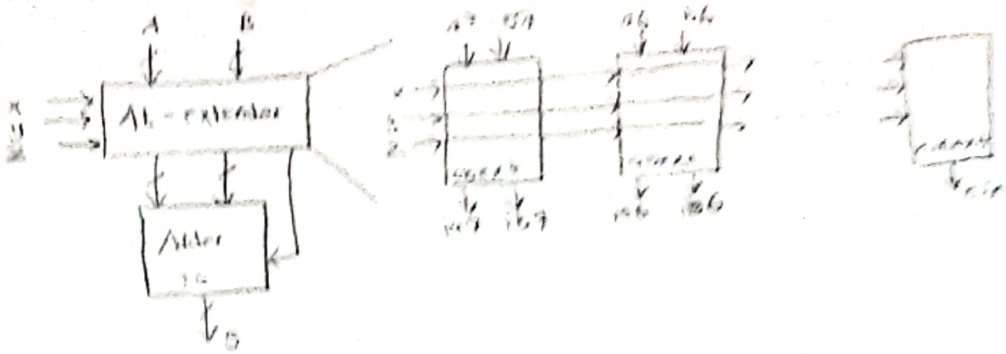
- ALU: Component that can perform various arithmetic (add, subtract, increment etc.) and logic (AND, OR etc.) operations based on control inputs

Inputs x y z	Operations	Sample Outputs if A=00001111 B=00000101
0 0 0	S=A+B	00010100
0 0 1	S=A-B	00001010
0 1 0	S=A+1	00010000
0 1 1	S=A	00001111
1 0 0	S=A AND B	00000101
1 0 1	S=A OR B	00001111
1 1 0	S=A XOR B	00001010
1 1 1	S=NOT A	11110000

Without using ALU

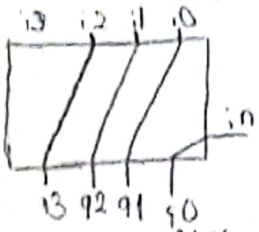


ALU1

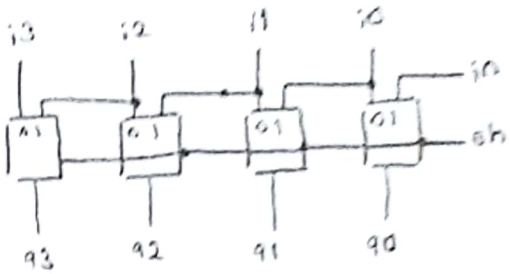


- xyz = 000 Want S = A+B Just pass a to ia, b to ib, and set cin = 0
- xyz = 001 Want S = A-B Pass a to ia, b' to ib, and set cin = 1
- xyz = 010 Want S = A+1 pass a to ia, set ib = 0, and set cin = 1
- xyz = 011 Want S = A pass a to ia, set ib = 0 and set cin = 0
- xyz = 100 Want S = A AND B : set ia = a # b, b = 0, and cin = 0
- Others likewise
- Based on above create logic for ia(x,y,z,a,b) and ib(x,y,z,a,b) for each above. and create logic for cin (x,y,z)

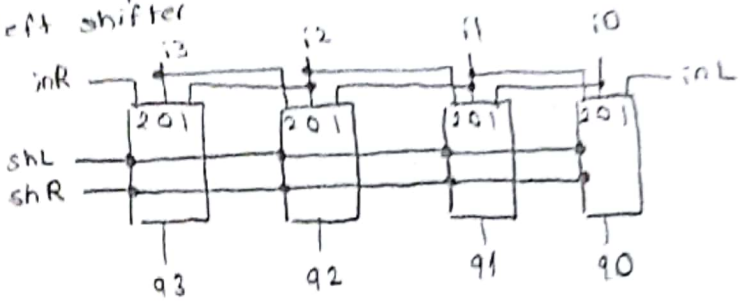
Shifters



Left shifter



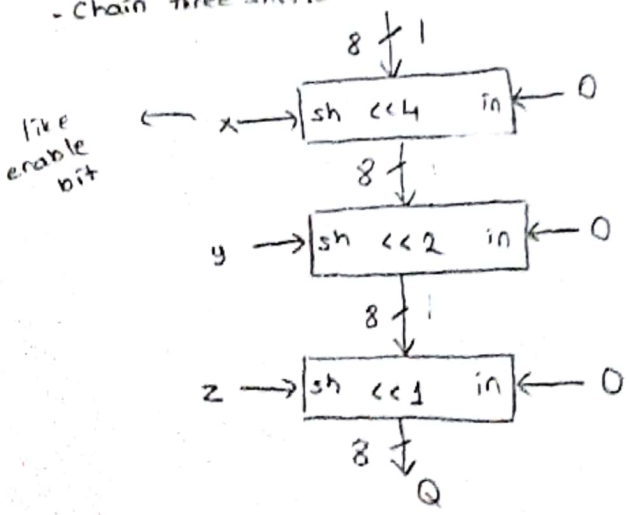
→ Shifter with left shift or no shift



→ shifter with left shift, right shift and no shift.

Barrel Shifters

- A shifter that can shift by any amount
  - 4bit barrel left shift can shift left by 0,1,2,3 positions.
- More elegant desing
  - Chain three shifters: 4:2 and 1



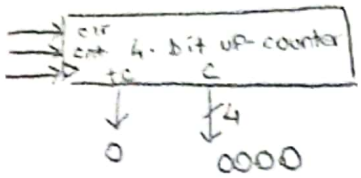
Counters and Timers

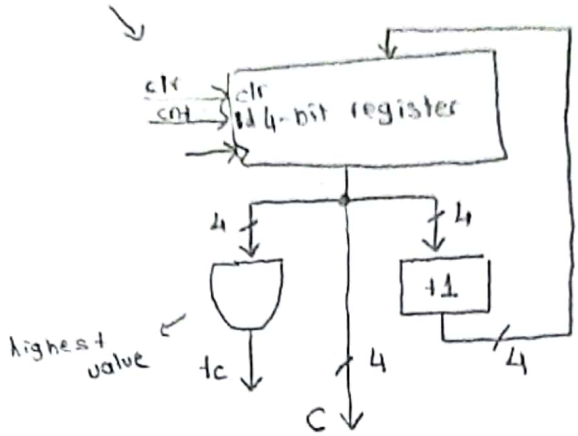
- N-bit UP counter : N-bit register that can increment to its own value on each clock cycle

  -0000, 0001, 0010, 0011, ..., 1110, 1111, 0000.

- Internal Design

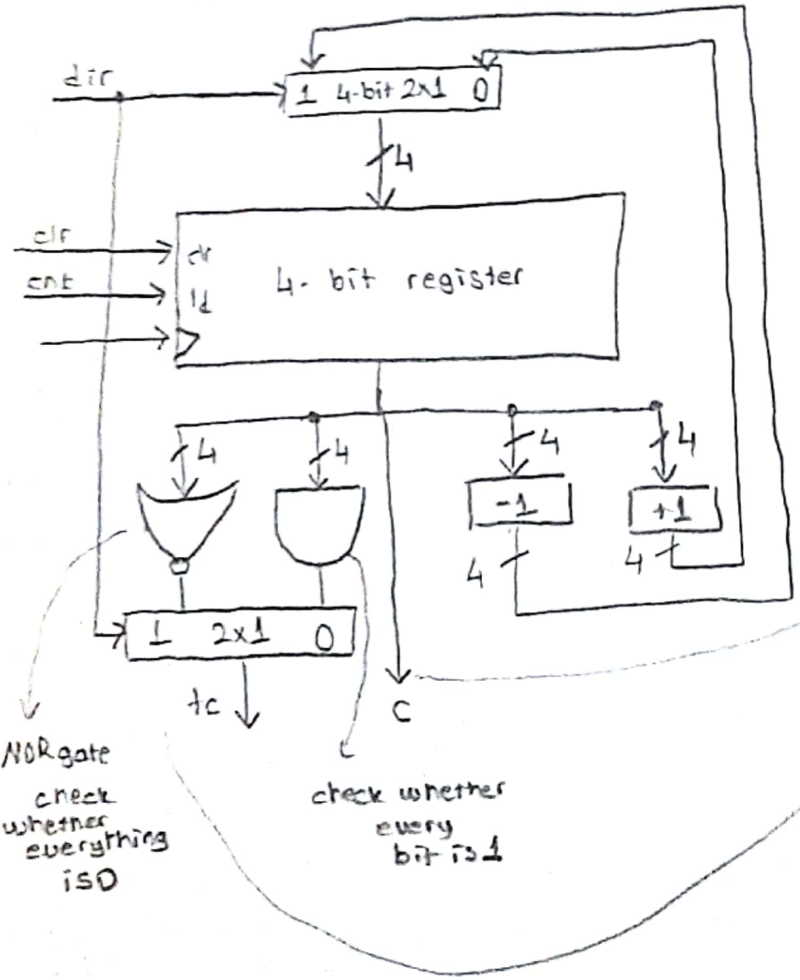
  -Register, incrementer, and N-input AND gate to detect terminal input.





UP/Down Counter

- Can count either UP or down
  - Includes both incrementer and decrementer
  - Use dir input to select, via 2x1 mux: dir=0 means UP



What if we count

0 to 10

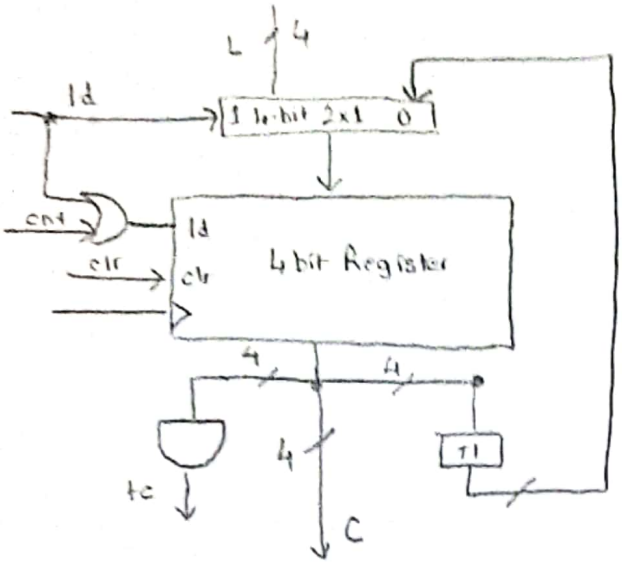
↓

1010

Can we detect with gates?

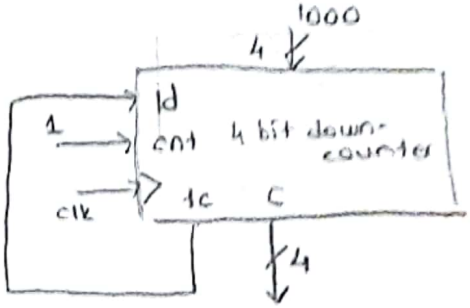
but we cannot change original tc value create own tc value or connect it to C!

Counter with Load



Counter with Parallel Load

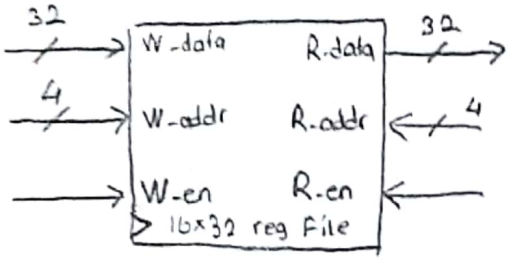
- Useful to create pulses at specific multiples of clock.
- Ex: Pulse every 9 clock cycles
  - Use 4 bit down counter with parallel load
    - Set Parallel load input to 8
    - Use terminal count to reload
      - When count reaches 0, next cycle loads 8.
    - Why load 8 and not 9 Because 0 is included in count sequence
      - 8, 7, 6, 5, 4, 3, 2, 1, 0 → 9 counts

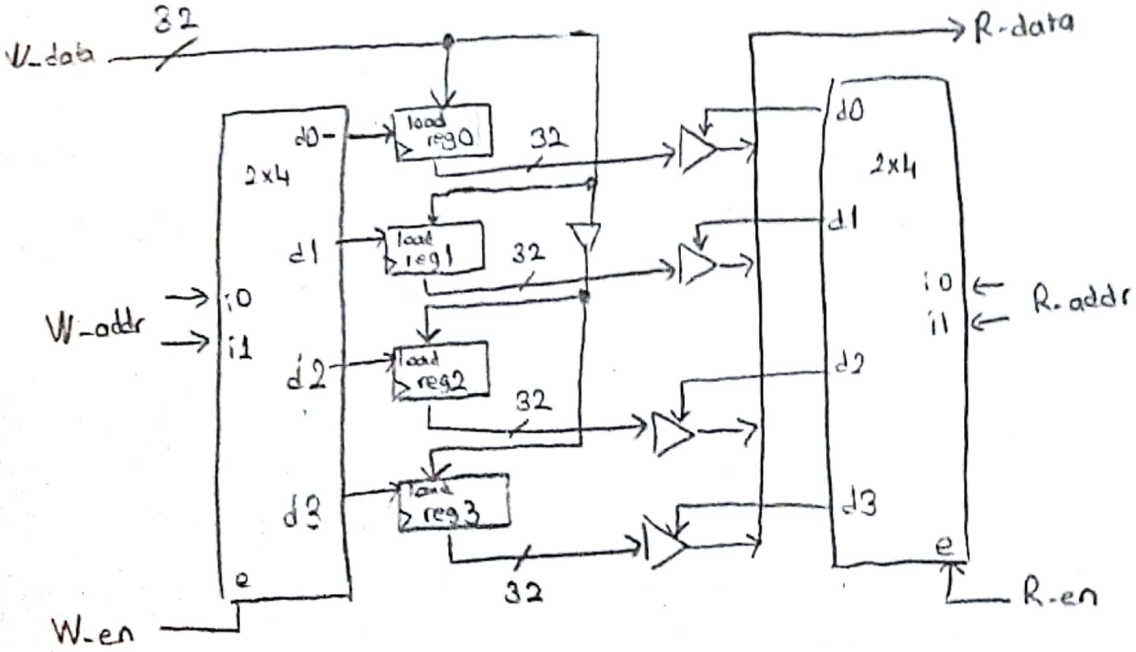


Register Files

M x N register file: Efficient design for one at a time write/read of many registers

- Consider 16 32-bit registers





4x32 reg-file

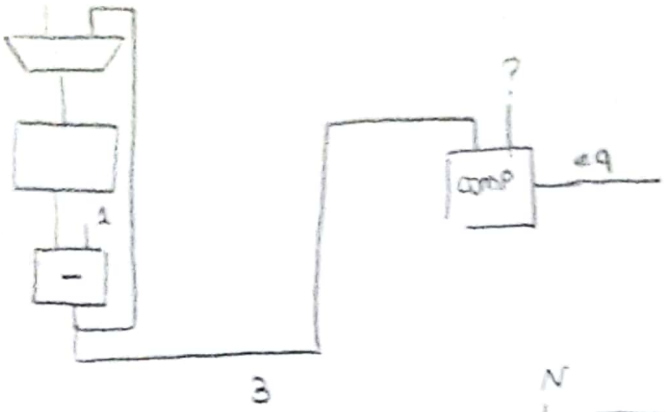
Ex/ Final question hocam dedi

5 → 2 | 3
6 → 3

even → number/2
odd → [number/2]

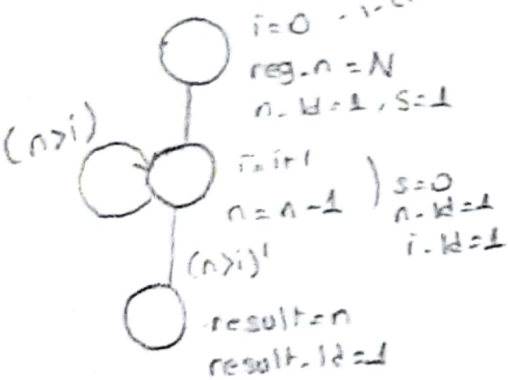
(you can not use shift and division)

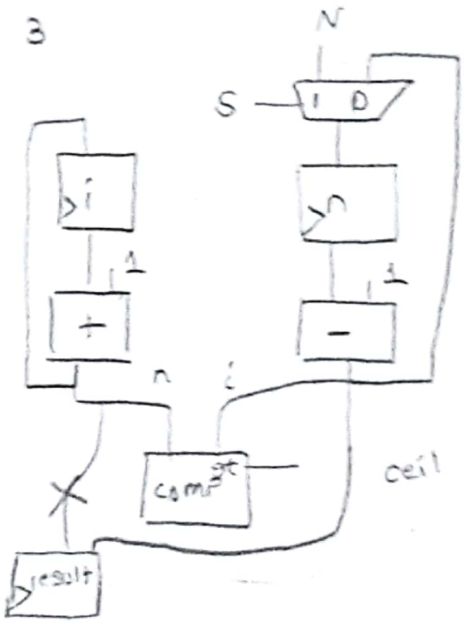
→ HLSM, datapath, FSM

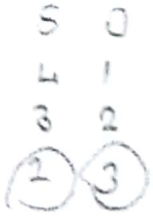


n	i
6	0
(	)
5	1
(	)
4	2
(	)
3	= 3 ⇒ ✓

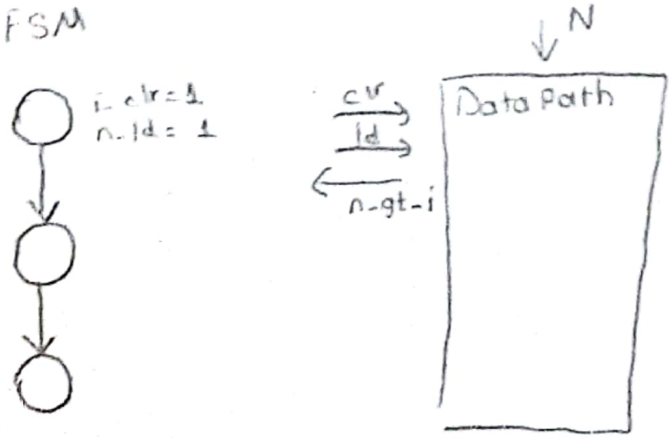
HLSM







FSM

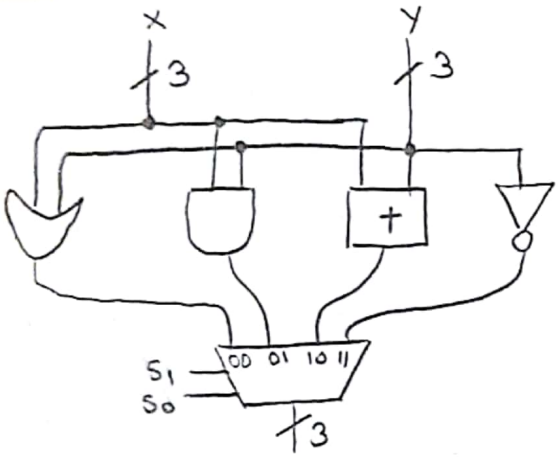


Study Questions

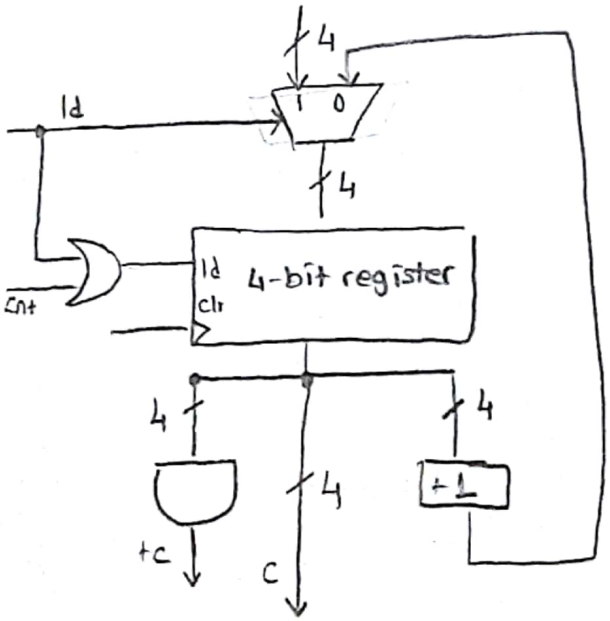
1. Implement a 3-bit ALU for the following operations.

S1	S0	Operation
0	0	x OR y
0	1	x AND y
1	0	x + y
1	1	y'

Answer:



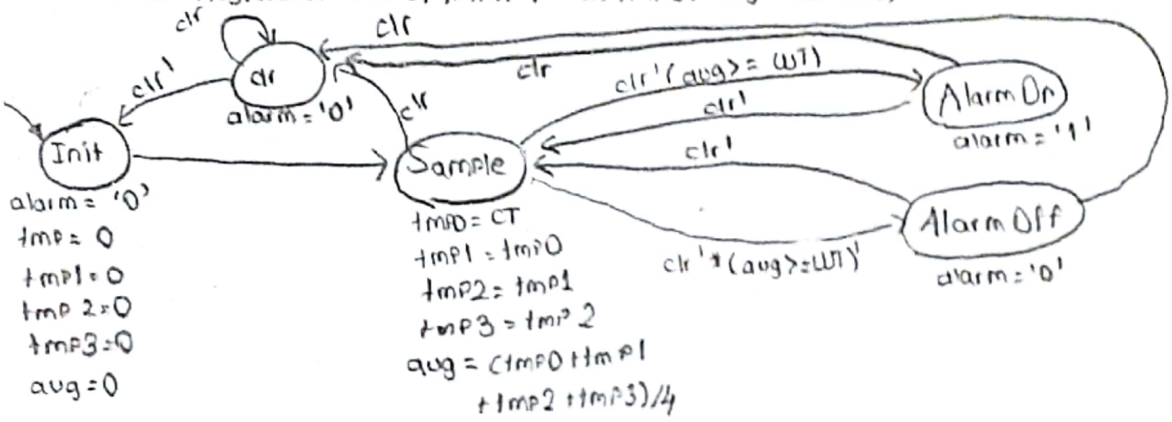
2. Design a 4 bit up counter which counts beginning from a load input and signals when a counting round finishes before starting a next round.



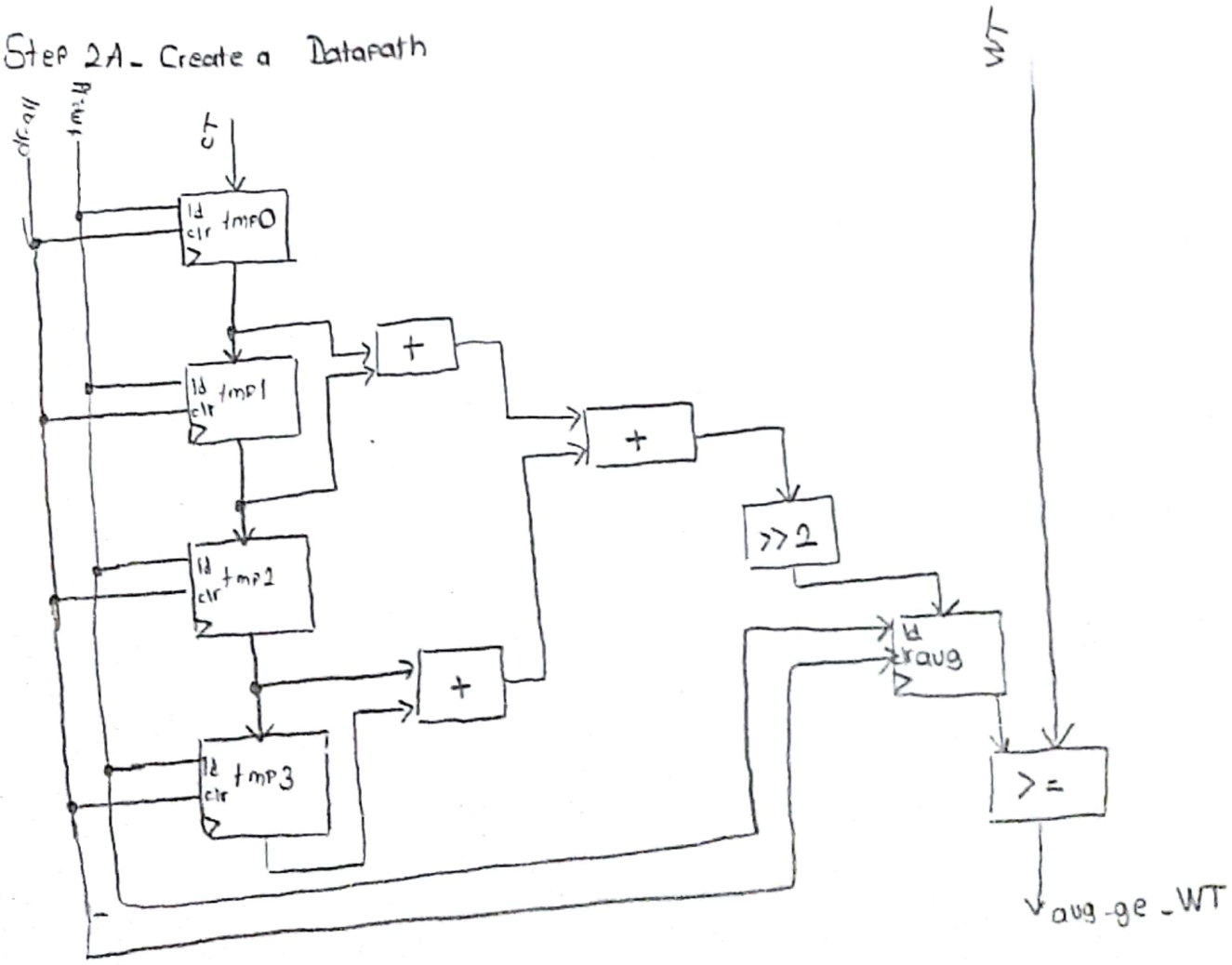
3. Use RTL design process to create an alarm system that sets a single-bit output alarm to 1 when the average temperature of four consecutive samples meets or exceeds a user defined threshold value. A 32-bit unsigned input CT indicates the current temperature, and a 32-bit unsigned input WT indicates the warning threshold. A single-bit input clr when 1 disables alarm and the sampling process. Start by capturing the desired behaviour as an HLSM, and then convert to a controller / datapath.

Step 1 - Capture a HLSM.

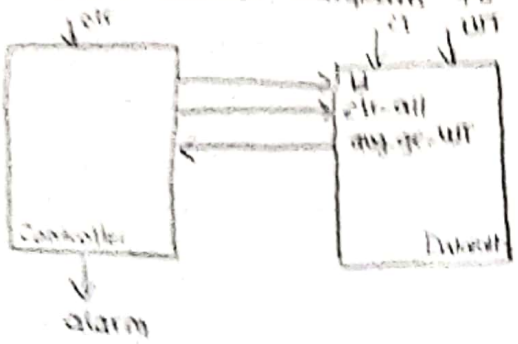
Inputs : CT, WT (32 bits) ; clr (bit)
Outputs: alarm (bit)
Local Registers: tmp0, tmp1, tmp2, tmp3, avg (32 bits)



Step 2A - Create a Datapath



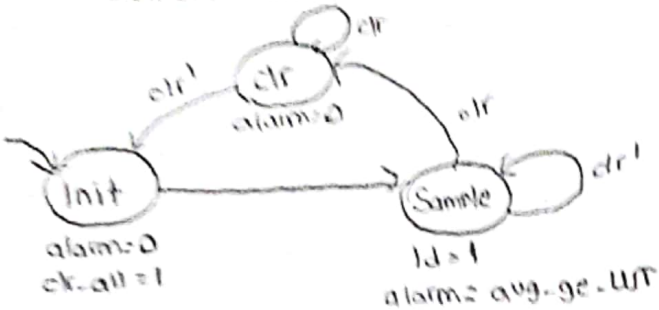
Step 2B - Connect datapath to controller



Step 2C - Derive controller's FSM

Inputs: clr, dec_ge_wt

Outputs: alarm, clr_all, ld



4) Implement a 4-bit register with the functionality specified in the fol. table. A is the current value of the register, and B is the loaded value.

S1	S0	Action	Output
0	0	Load	B
0	1	Keep current value	A
1	0	if (B>A) load B/2 else load B*2	if (B>A) B/2 ; else B*2
1	1	if (B<A/2) load B else keep current value	if (B<A/2) B; else A