2-1-17

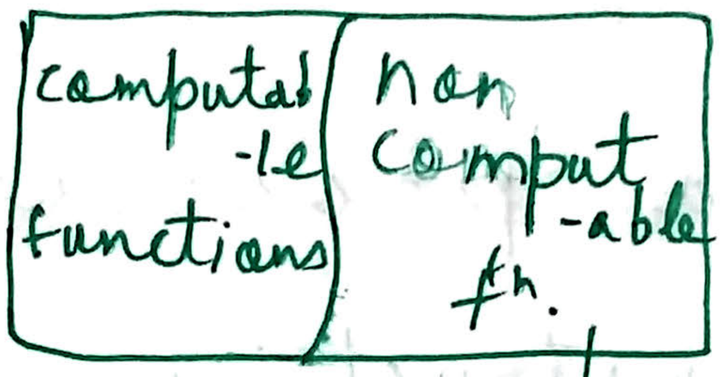
C S O

Servers, Embedded Systems, desktop system

A digital watch may not be called a computing system as it is not programmable. It does only one job.

Can we say that; even the computers are limited; so how can your call it programm-able?

No. Turing Machine model (of a comp-uter)



complete

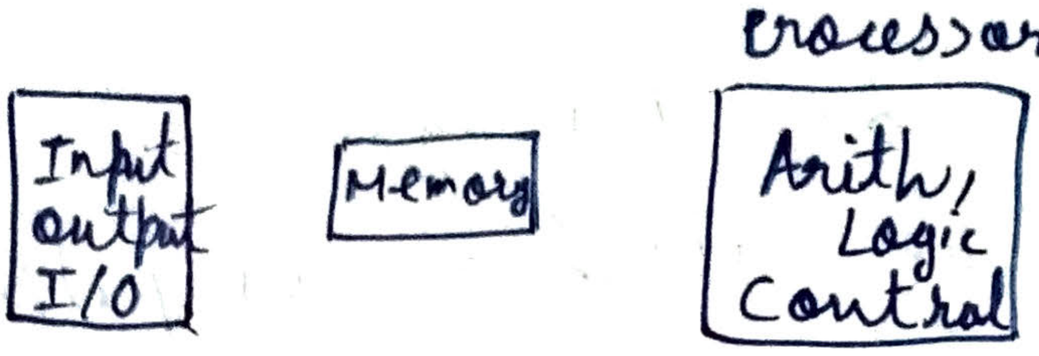
So we call a device a^ computing system if one can realise all the computable funct.

The digital watch was limited. Hence we MAY NOT consider it to be a digital computing system.

Levels of programmability:

digital Watches ⟶ Multiplexers ⟶ Laptops

Basic Units of a computer:



For Embedded systems. Input Output I/O is not same for each of the system.

Internet of Things?

Embedded systems?

To ~~Rnow~~ use a processor, we need to Rnow it's Instruction set.

1. Inst. set Arch. design (ISA)] Course topics
2. processor design
3. Interface with memory
4. I/O devices

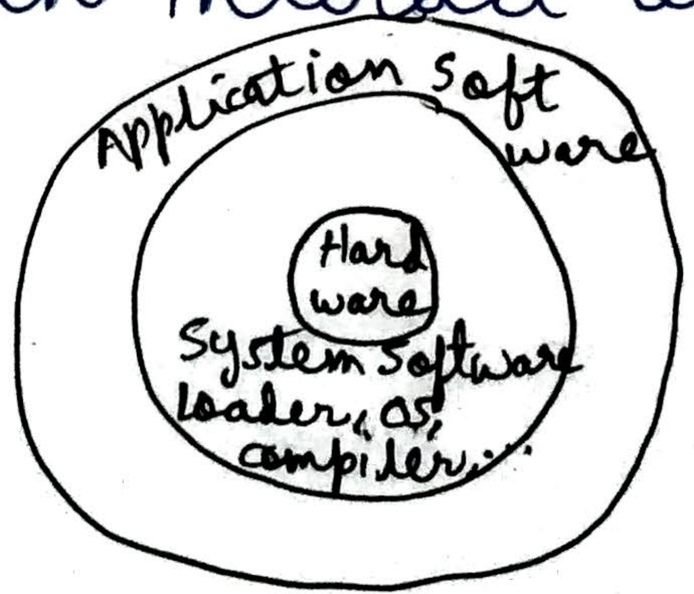
Plug Computer?

[small computers; display, I/O is a problem)

↳ We can use a projector

PICO projector?.

When we execute a compiled program, it gets loaded onto memory and the PC is set accordingly. Processor can interact only with the memory.

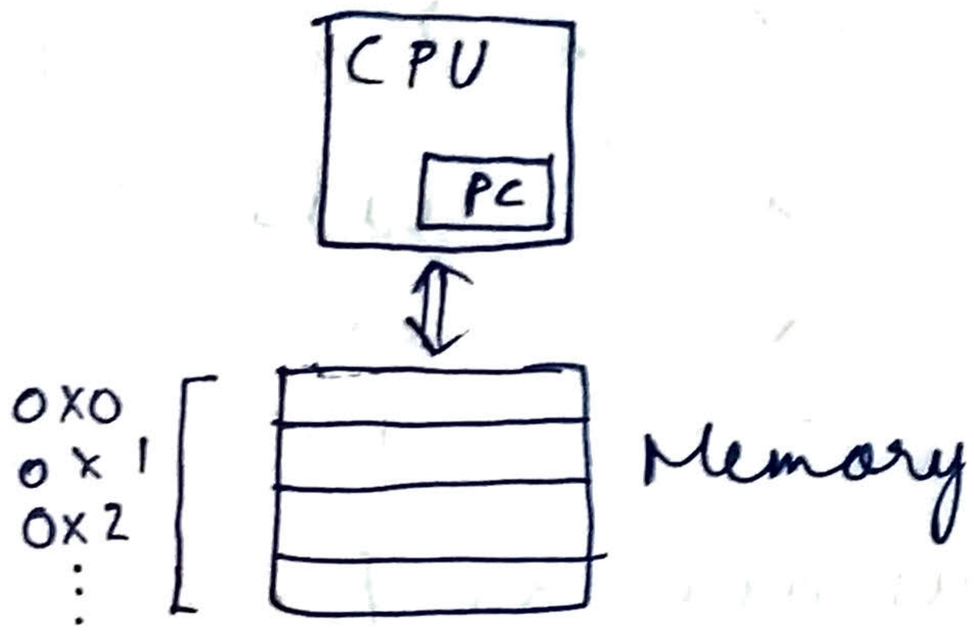


output of the processor is stored in the memory. The OS redirects the stored output to the destination.

We want to design machines which have good

1) performance/speed
2) power (low consumption)
3) size/area
4) cost. (less cost)

Processor Model



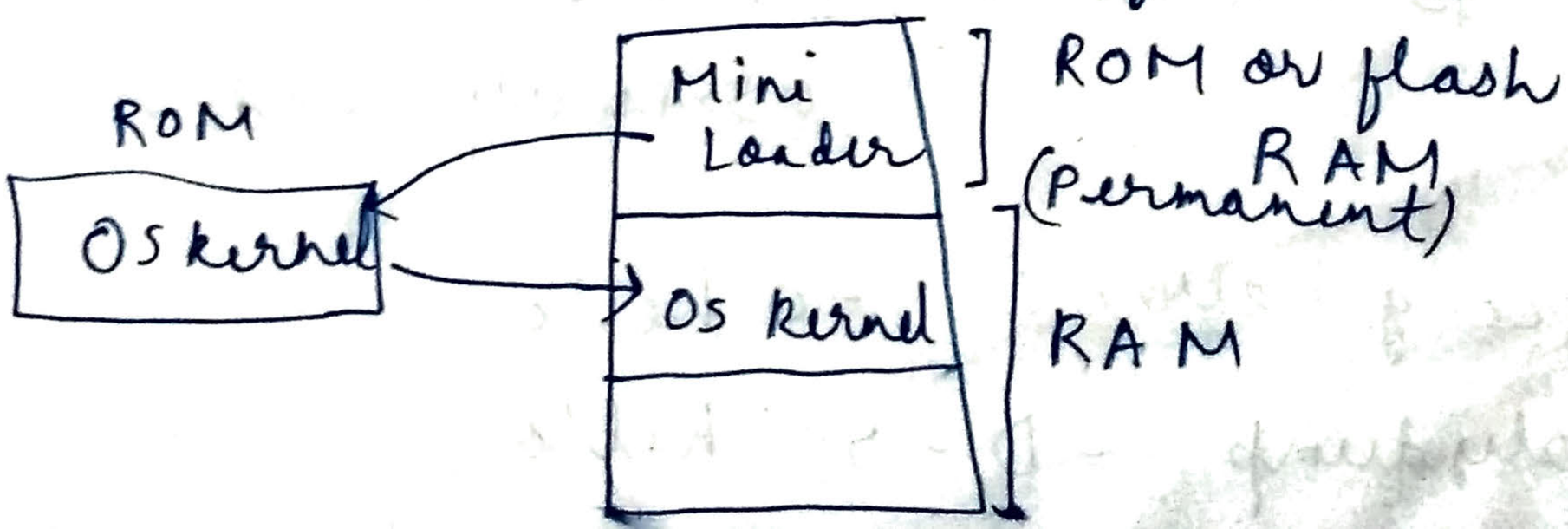
All the processor does is: fetch-decode-execute cycle.

For improving the performance of the processor the memory model should also be understood.

Loader (present as a part of the OS) loads the executable into the main memory.

Bootstrapping problem: initialization problem.

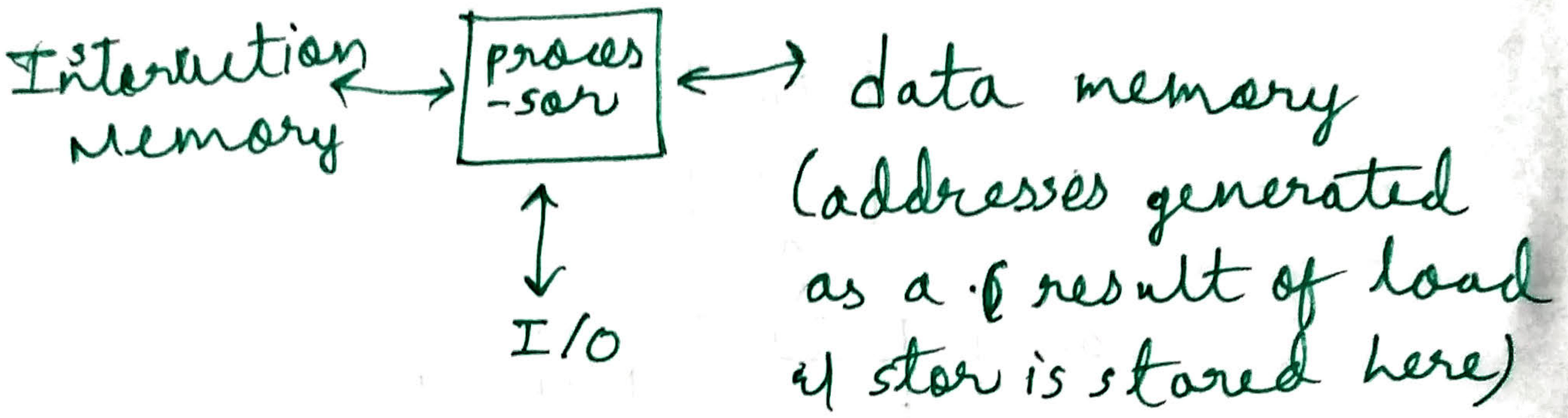
The problem here is that ~~wh~~ ~~comes to~~ when we switch on the computer, the loader must be activated so that meaningful instructions are executed & not some garbage values.



When we compile a ~~run~~ a program in IA-32 processor in Linux and run it in Windows IA-32 processor, it will not run as many instructions like printf, scanf include OS calls which are not same in all the OS's.

Processor models:

Harvard architecture:



Von Neumann or Princeton arch.
(it common)

~~Both memory in~~ Single memory

The advantage with harvard is: we can have // access to both inst, data. But since we have partitioned the memory, there is a limit on how ~~much~~ any instructions can be accomodated.

SOC architectures: System on Chip arch.
(smart phones, etc)

1. gcc -gS hello.c gives hello.s
assembly code.

2. gcc -g hello.c -o hello
objdump -D -S hello

Given a binary string,

B2 Unsigned $(\vec{x}) = \sum_{i=0}^{w-1} 2^i x_i$

B2 Signed $(\vec{x}) = (-1)^{x_{w-1}} \sum_{i=0}^{w-2} 2^i x_i$

B2 One's comp $(\vec{x}) = -(2^{w-1}-1) * x_{w-1} + \sum_{i=0}^{w-2} 2^i x_i$

B2 2's comp $(\vec{x}) = -2^{w-1} x_{w-1} + \sum_{i=0}^{w-2} 2^i x_i$

	U	SM	1's	2's
00	0	0	0	0
01	1	1	1	1
10	2	-0	-1	-2
11	3	-1	0	-1

t_1	00	01	10	11
00	00	01	10	11
01	01	10	11	00
10	10	11	00	01
11	11	00	01	10

·	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

1. Depending on the technology which we use, we may have to use a particular notation
2. The cost of mapping from 0-9 notation to that particular notation should be low.

Unary notation:

⊗	[ ]	1	11	111
[ ]	[ ]	1	11	111
1	1	11	111	1111
11	. . . .			
111				

Cost of conversion is high.

Another thing to be taken care of: Overflow
We can send a signal to indicate this.
There is partial isomorphism b/w 1's comp and binary rep. table.

Overflow:

C_{in}

$x_{w-1} \cdots x_1$

$y_{w-1} \cdots y_1$

C_{out} $S_{w-1} \cdots S_1$

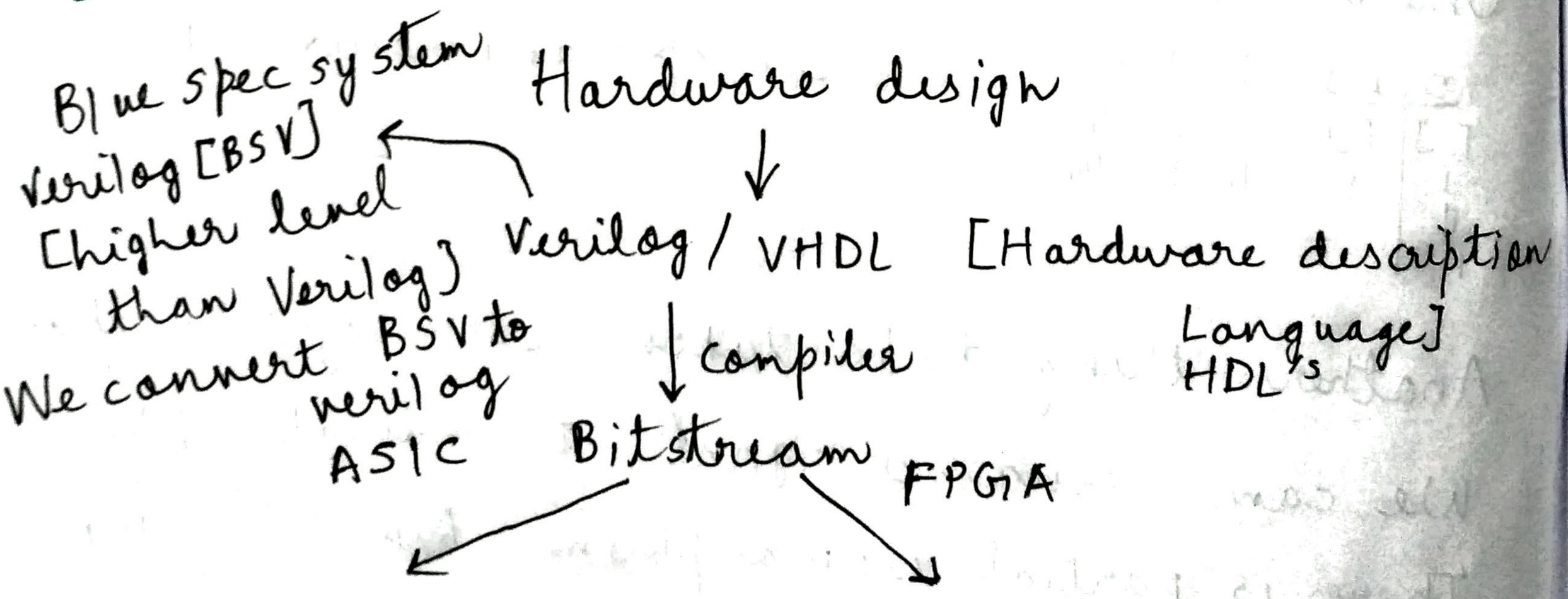
(if (=((Cin ⊕ Cout) 1) "overflow" "No overflow"))

There is a flag register which is used for storing the meta information about a computation.

This flag register is called CVSR in ARM processor.

C	V	Op-1	Op-2
0	0	00	00
0	1	01	01
1	0	11	11
1	1	10	10.

1. Sign-extension
2. Arithmetic & logical shift.



function fa(a, b, cin)
↳ wires (not args)

[s = (a ^ b) ^ cin
[c-out = ———] PTO

↓
names of wires.

ALU design

Data Pack
↓
has info abt the functions to be performed

Control Pack
↓
Takes in a f'n, co-ordinates the f'n execution by taking info from data pack.

n-bit IS Architecture: Operands can be of n-bits.
RISC V Architecture.?

Hardware (H/W) design
↓
BSV source code
↘ Simulation (check for correctness of circ.)
↙ ↘
ASIC [IC fabrication] → Risky
FPGA Re-programm-able hardware.

function:

```
fa ( a, b, c_in);
s = (a ^ b) ^ c_in;
c_out = (a & b) | (c_in & (a ^ b));
return {c_out, s};
```

ordering is important; s is the least significant bit.

Blue spec compiler remembers a ^ b and directly uses it for the next time rather than re-computing it.

But here we didn't specify the no. of bits of the input.

Bit#(1)

Bit #(2) fa (a, b, c-in)

Bit #(1) · s = a ^ b ^ c-in

Bit#(1) c-out ...

```
return {...}
end function.
```

Advantages: 1. Readability

2. Type-check:

Bit#(1) s = a ^ b ^ c_in

1bit ^ 1bit ^ 1bit

1 bit

matched.

3. This is a strongly-typed-language.

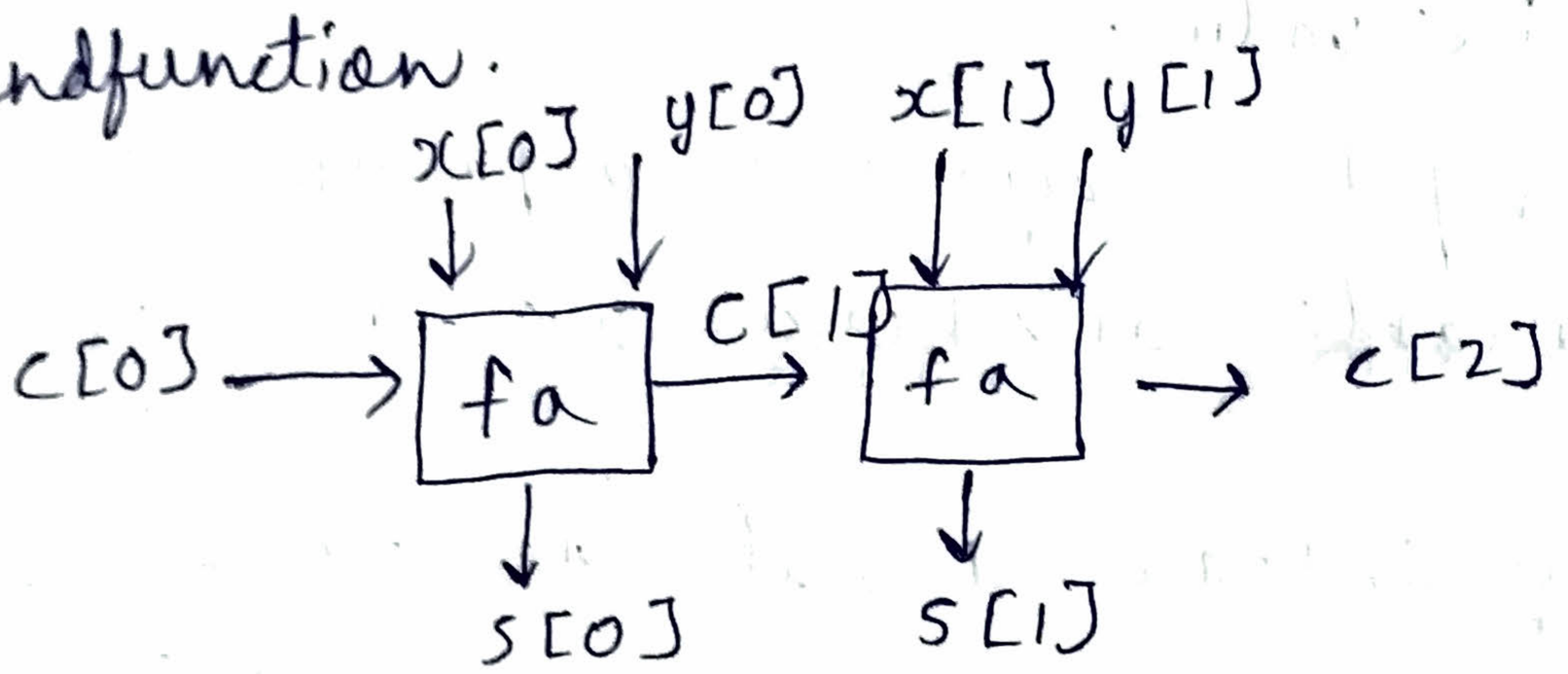
eg of weakly typed:

```
char a = 'A';
int b = a;
```

this is where we say it is weakly typed.

2-bit Ripple Carry Adder:

```
function: Bit#(3) add(Bit#(2) x, Bit#(2) y,
                      Bit#(1) c0);
  Bit#(2) s = 0;
  Bit#(3) c;
  c[0] = c0;
  let cs0 = fa(x[0], y[0], c[0]);
      c[1] = cs0[1]; s[0] = cs0[0];
  let cs1 = fa(x[1], y[1], c[1]);
      c[2] = cs1[1]; s[1] = cs1[0];
  return {c[2], s};
endfunction.
```



When we use let, the type of RHS is evaluated & assigned to cs0. We need not mention it.

W-Bit Ripple Carry adder

(typedef Bit# ↑ #)

```
function Bit#(W+1) addN(Bit#(W) x,
                        Bit#(W) y,
                        Bit#(1) c0);
  Bit#(W) s; Bit#(W+1) c; c[0] = c0;
  for(Integer i=0; i<W; i=i+1)
    begin
      let cs = fa(x[i], y[i], c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
```

end.
return {c[w], s};
end function.

To call this fn:

function #(33) add32 (#(32)x...)
= addN(x, y, c).

① Bit#(w+1) → Error

Bit#(TAdd#(w,1))

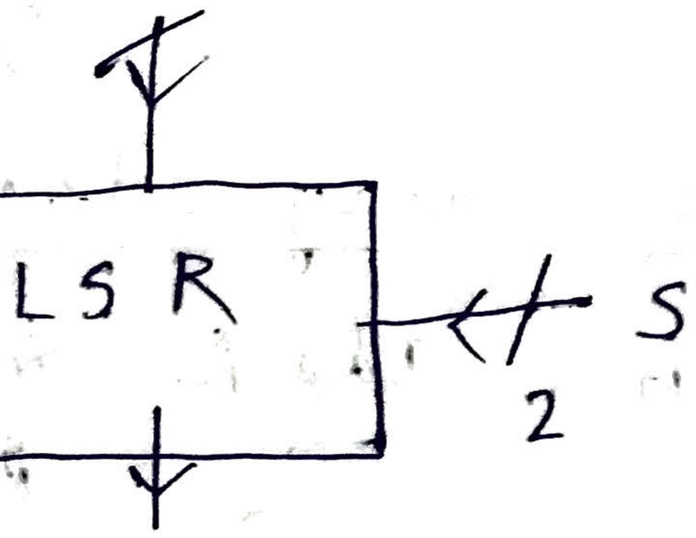
② i < w → Error

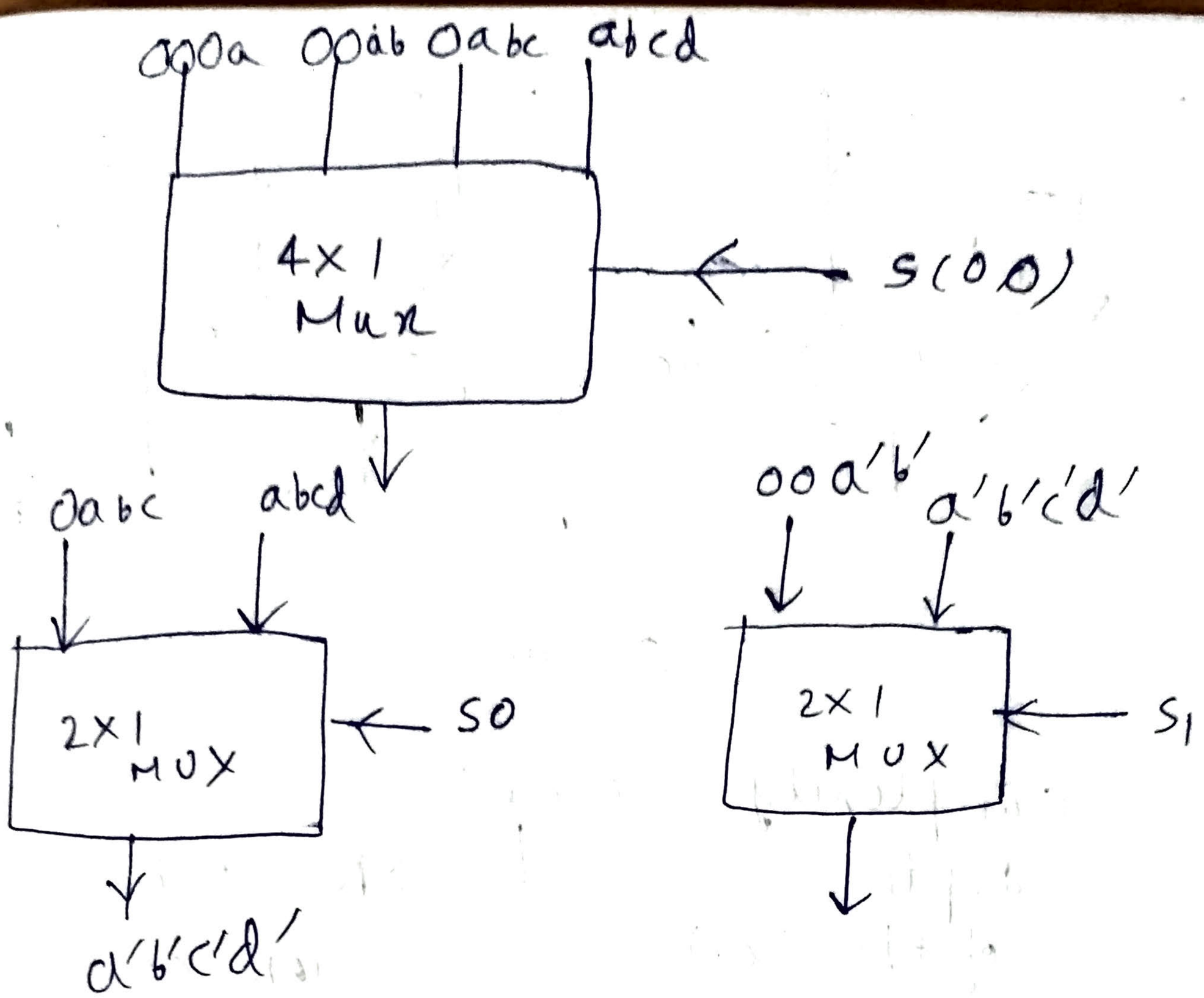
Let value = value of(w);
i < value.

Bluespec compiler says w is a parametric type variable and hence we can't say w+1, or i<w.

When we say Integer i, it has no bound. (not a 32bit or 64 bit no:)

Logical Shift Right Circuit





Code this.

ALU

ADD, SUB, MUL, DIV

AND, OR, XOR

LSR, ASR, LSL, ROR

Shift Operations

define

$S_0(abcde) = abcde$

$S_1(abcde) = 0abcd$

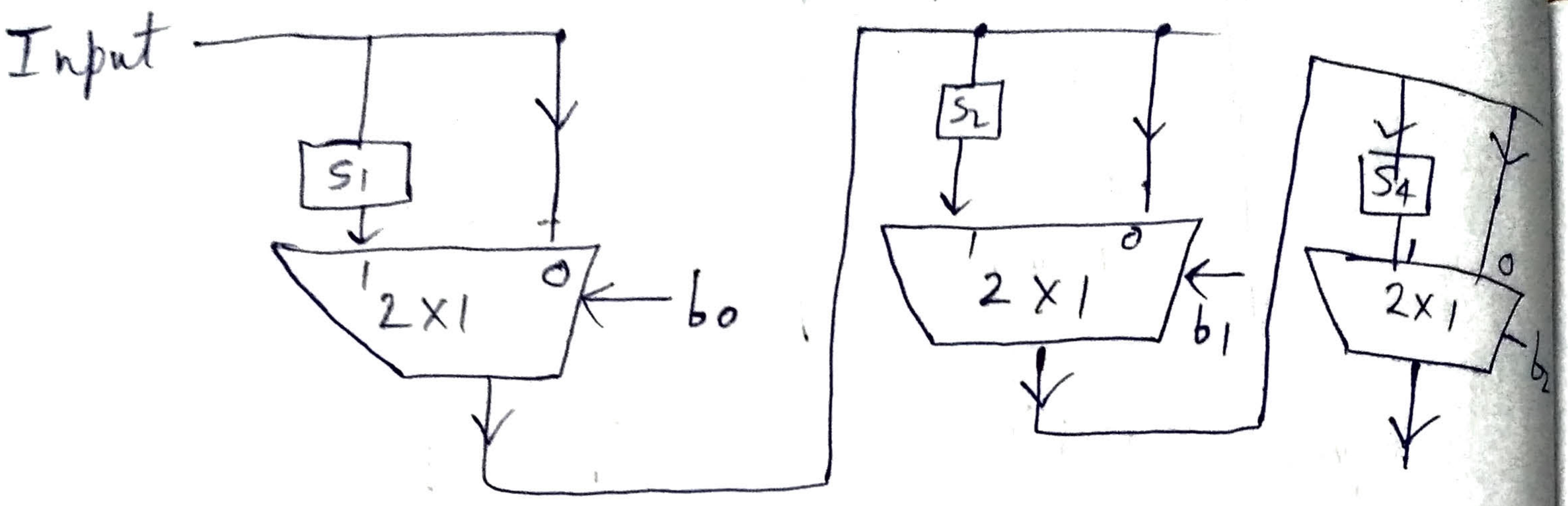
$S_2(a,b,c,d,e) = 00abc$

$S_4(abcde) \Rightarrow 0000a$

S = No. of shifts required.

$S = \underbrace{b_2\ b_1\ b_0}_{\text{bits}}$

given S, abcde design shift circ.



Circuit.

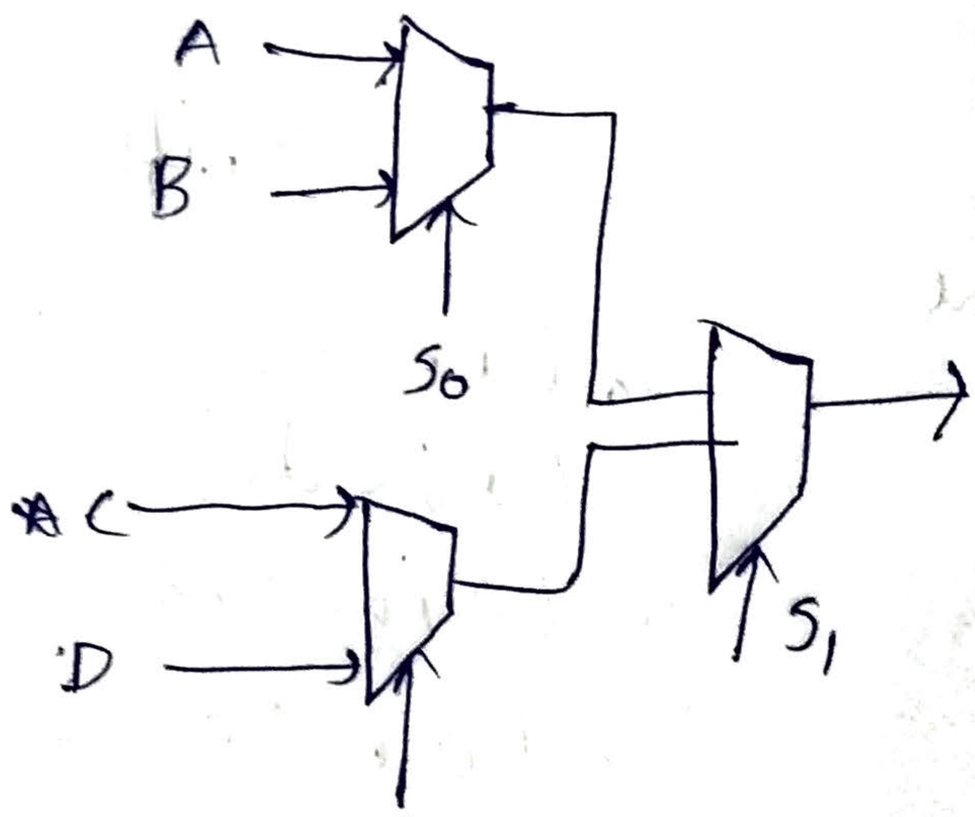
complexity: O(K)
(depth of the circ.)
↳ No. of bits in 'S'

Can we do it in O(1), O(log K), O(√K).

4x1 Multiplexer

case {S1, S0}, matches

0: A;
1: B;
2: C;
3: D;
end-case



The above code in machine language will take atleast 30-40 lines. If we are executing many times, We lose both power, time in executing the same inst. Instead if we design hardware for it, it would be really helpful. Now-a days

we have re-configurable processors in which we can design it ourselves.

There are some typedefs : [slides]

[~~is~~ Did not note down a topic: Colour coding, etc: Refer slides]

Multiplication by repeated Addition

b Multiplicand : 1101

a Multiplier : 1011 (slides)

[(a[i]==0) ? 0 : b]

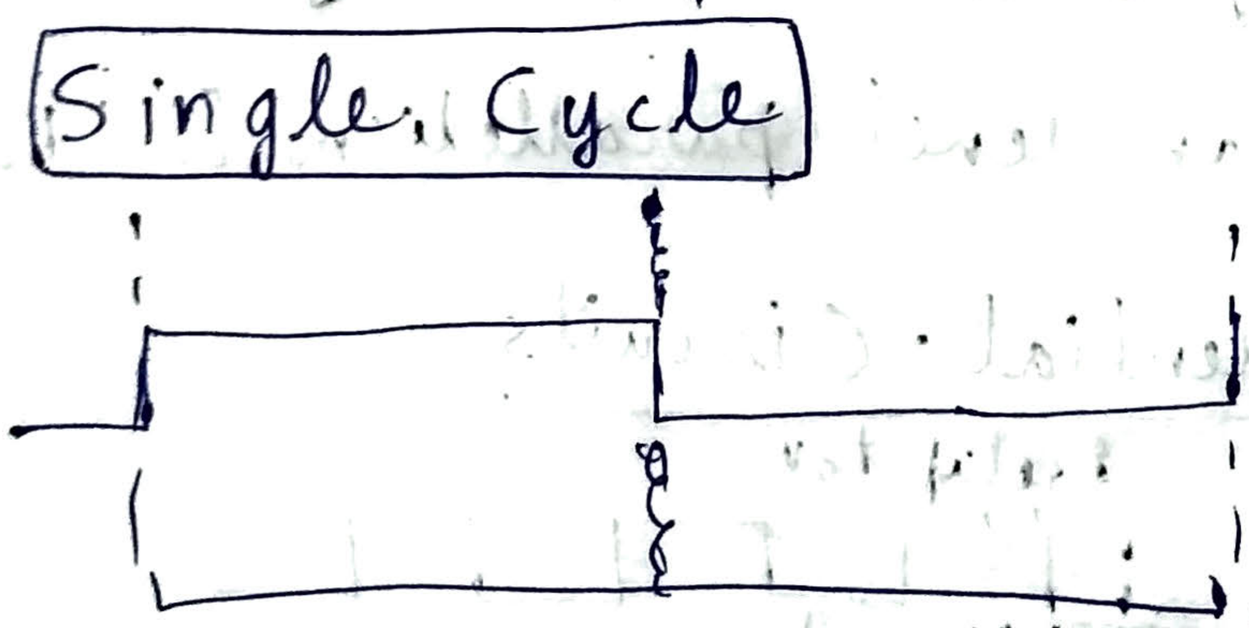
Processors → Single Cycle [All the operations in 1 cycle]
clock length : ↑

→ Multi-cycle [Multiple cycles]
clock time period : lesser

↓ pipelined [model]

(Doing operations in parallel by using multiple registers)

Super-scalar processors use pipelined model.



Clock cycle length is increased.

Clk span > max(~~delay~~ delay_add, delay_sub, ...)

Performance isn't so good.

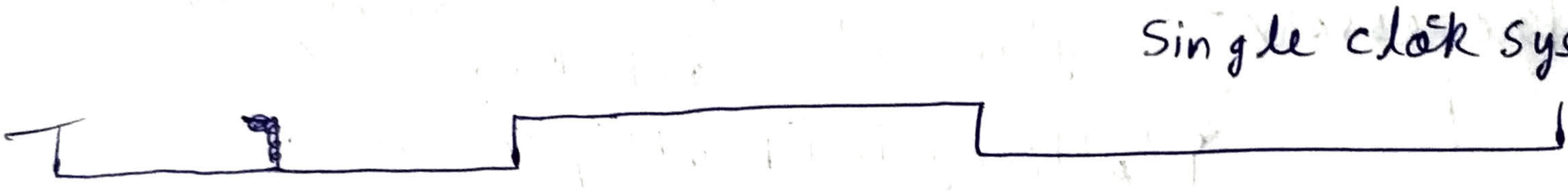
To counter this, are have multiple clock cycle method:

For eg: Load inst. may take 5 steps.
Add: 3 steps.

Clk span > max([l_1-l_5], [a_1-a_3], ...)

here we pay the price of a long ~~inst~~ process only when it is invoked as apposed to single clock system where we pay the price ~~in~~ for all the functions in the ISA.

Single clock system



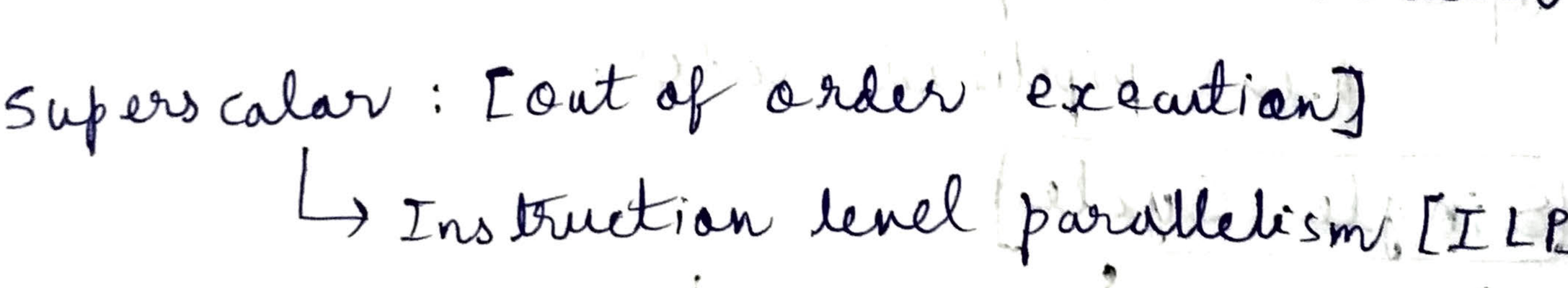
. . . Multicycle.

1 Inst Next Inst.

Inorder - pipeline [Output comes in order]

Superscalar: [out of order execution]
↳ Instruction level parallelism [ILP]

Sequential - Circuits



In b/w (t_{su}^{-}, t_{w}^{+}) input shd not oscillate. Else it enters a meta stability ~~where~~ state in which

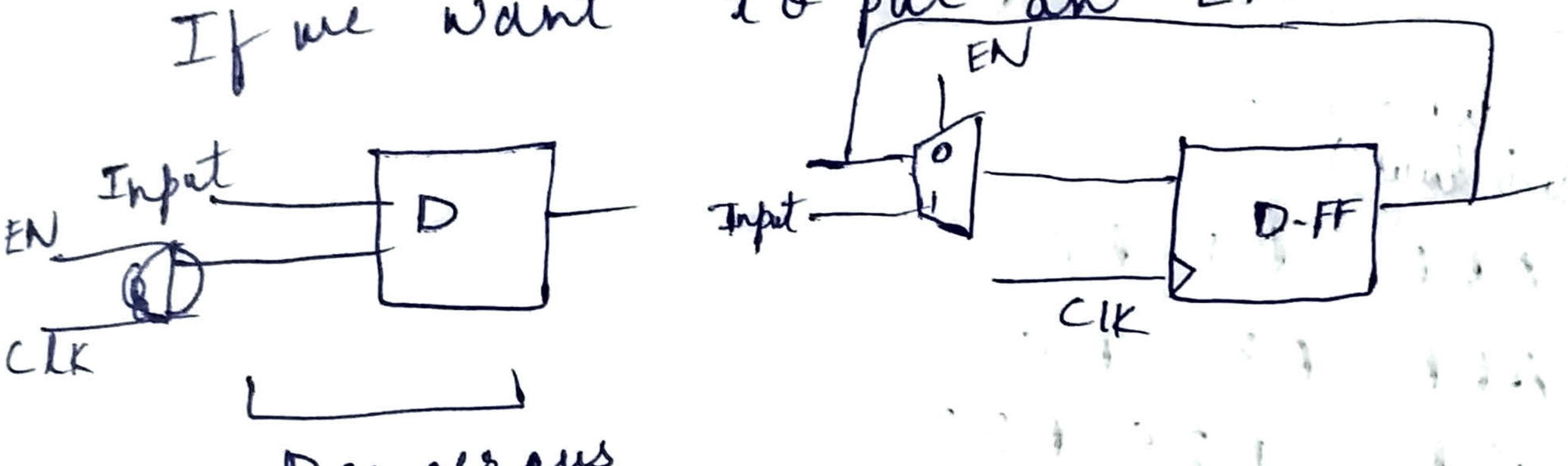
it oscillates b/w 0, 1 : ∞ ly.

$[t - t_{su}, t]$ $[t, t + t_h]$

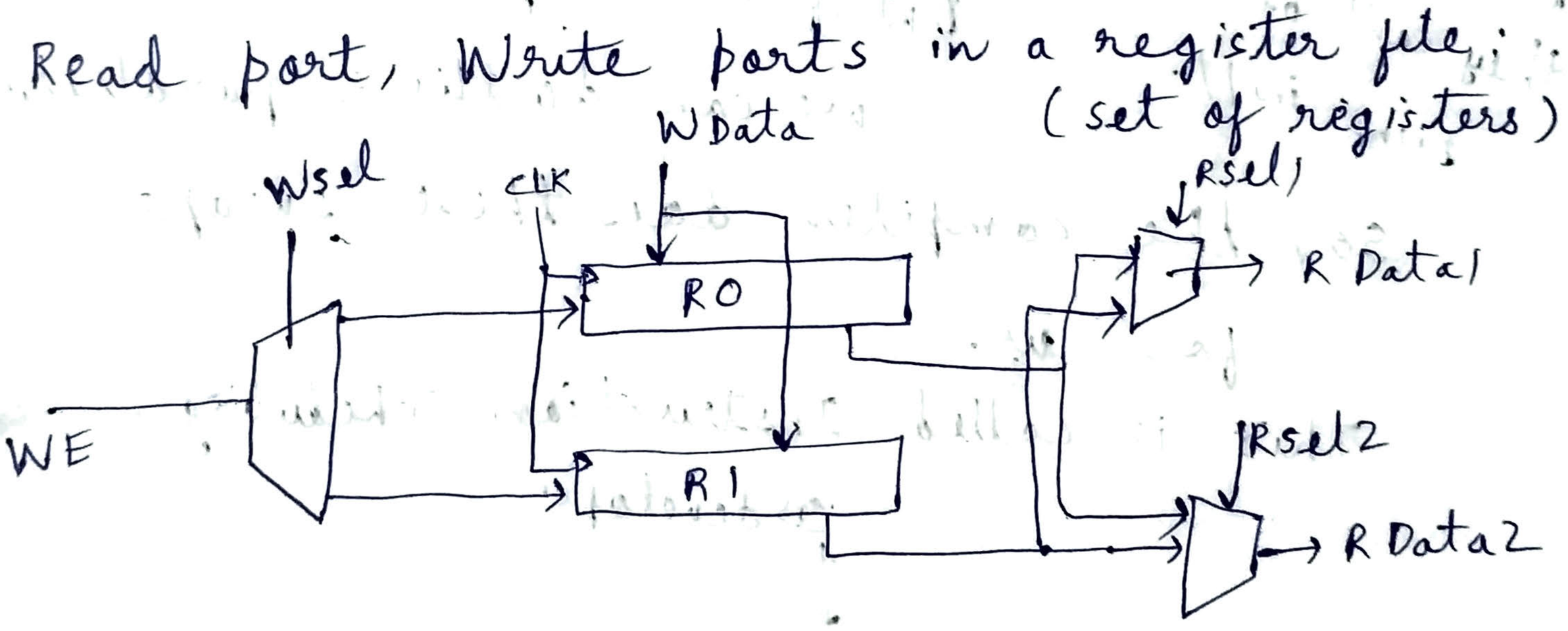
↳ setup violation ↳ hold-time violation

If clock span is so less that it's value is b/w $(t_{su} + t_h)$; meta stability state is inevitable.

If we want to put an EN to the FF:



Read port, Write ports in a register file : (set of registers)



2 read ports as we may want to access both the Registers simultaneously.

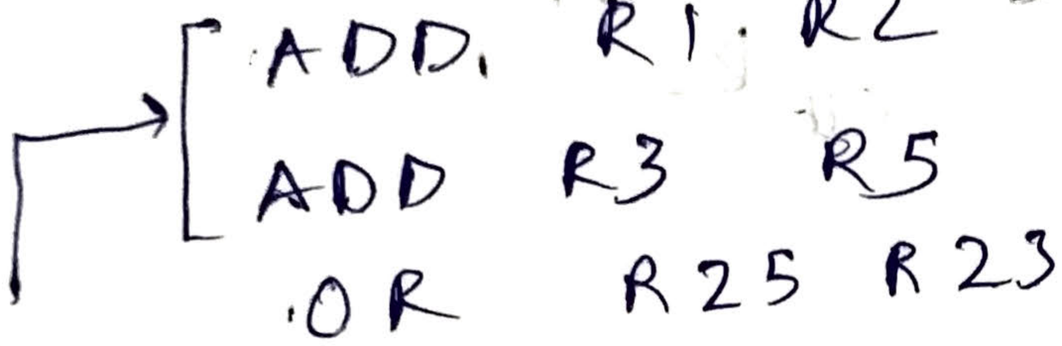
When we are executing inst. in parallel 2 instructions may want to access 5-diff Registers in a file at a time leading to structural hazards. So, increase the read, write ports.

If 2 ADD's are present:
(and only one adder is present)
structural hazard.
so increase the no. of ALU's or use
pipelined model.

To check whether Inst. can be executed
in parallel or not: We put a window
(generally 4-8) set of instructions.

Instruction Scheduling

* Let W = 2.
(window)

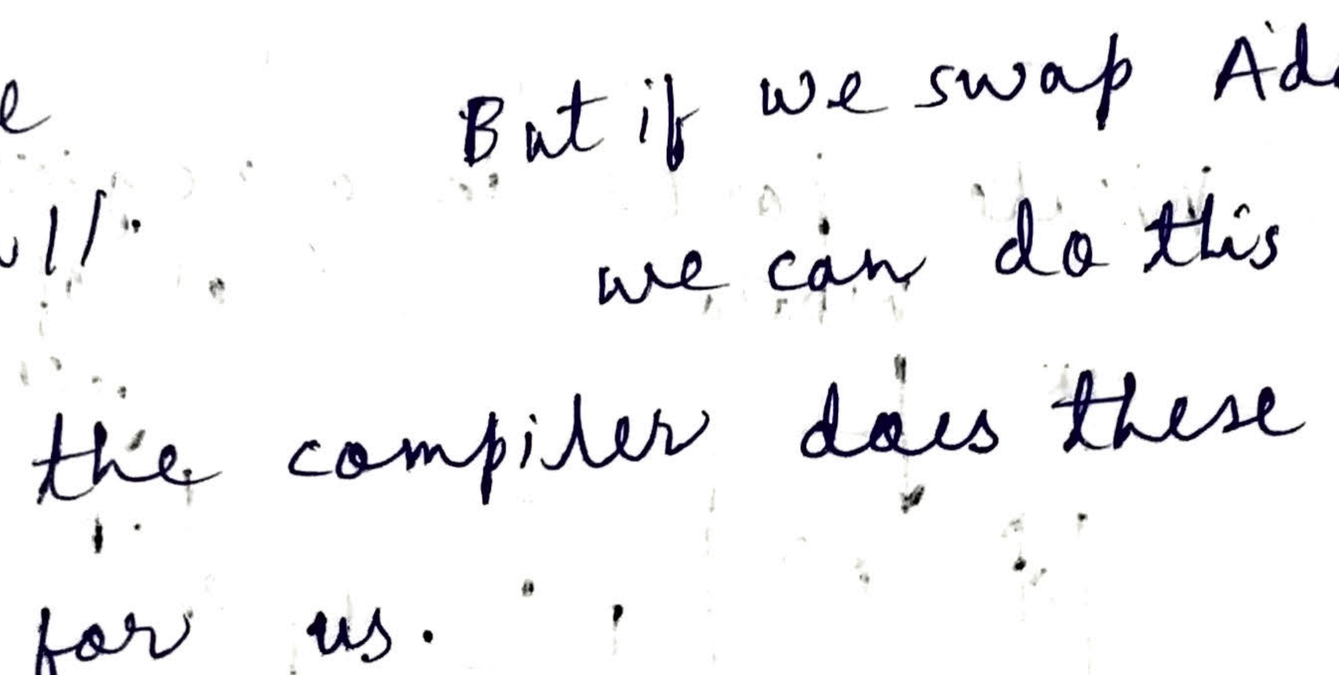


It thinks they can't be run in //.

But if we swap Add, Or
we can do this in parallel.

So, the compiler does these swaps
for us.

This is called Instruction Scheduling.



The efficiency of swapping may be
diff in diff. compilers.

Data hazards

ADD R1 R2 R3

ADD R4 R7 R1

RAW [Read after write] dependency.
flow dependency, True dep.

MUL R1 R2 R3

ADD R3 R7 R10

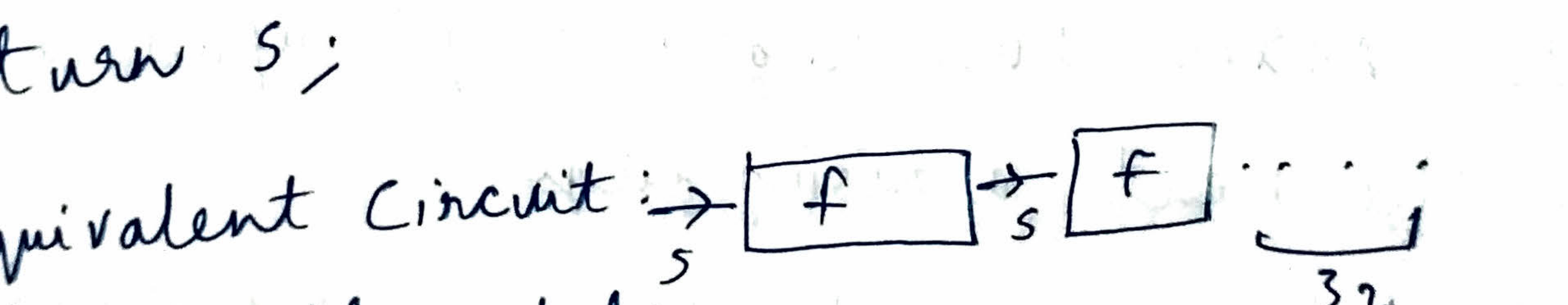
If ADD finishes faster than MUL,
new value of R3 is multiplied
with R2 but we want old value
(WAR)

Loop using registers:

For C Integer i=0; i<32; i++)
begin
let s = f(s);
end
return S;

Equivalent Circuit: → s [f] → s [f] ... (32)

Another implementation:



Sequential elements in Bluespec:

```
Reg#(Bit#(32)) a <- mkReg(0);        // initialisation
                    b, prod, tp;

Reg#(Bit#(6)) i <- mkReg(32);

rule mulstep if (i<32);

    Bit#(32) m = (a[i]==0) ? 0 : b;
                 [a[0]==0]
                                    a <= a>>1;
    Bit#(33) sum = add32(m, tp, 0);

    prod[i] <= sum[0];          // -> prod <=
                                    (sum[0],
    tp. <= sum[32:1];               (prod>>1)
                                     [30:1])
    i <= i+1;

endrule
```

Instructions in the loop can be executed in parallel.

What happens if i++ happens before prod[i] is updated? Although i+1 is available at the Adder bus, registers get updated only in the next a rising edge.

Although we saved space, we couldn't do good with speed. (we added extra time - updating registers).

2-2-17

Combinational IFFT

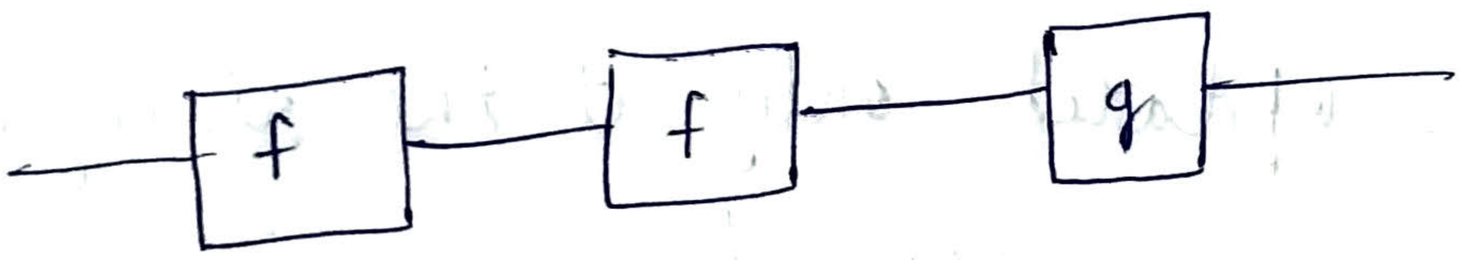
(Inverse fast forward transform).

The inputs are complex nos: represented with 32 bits. (16 for real part, 16 for complex).

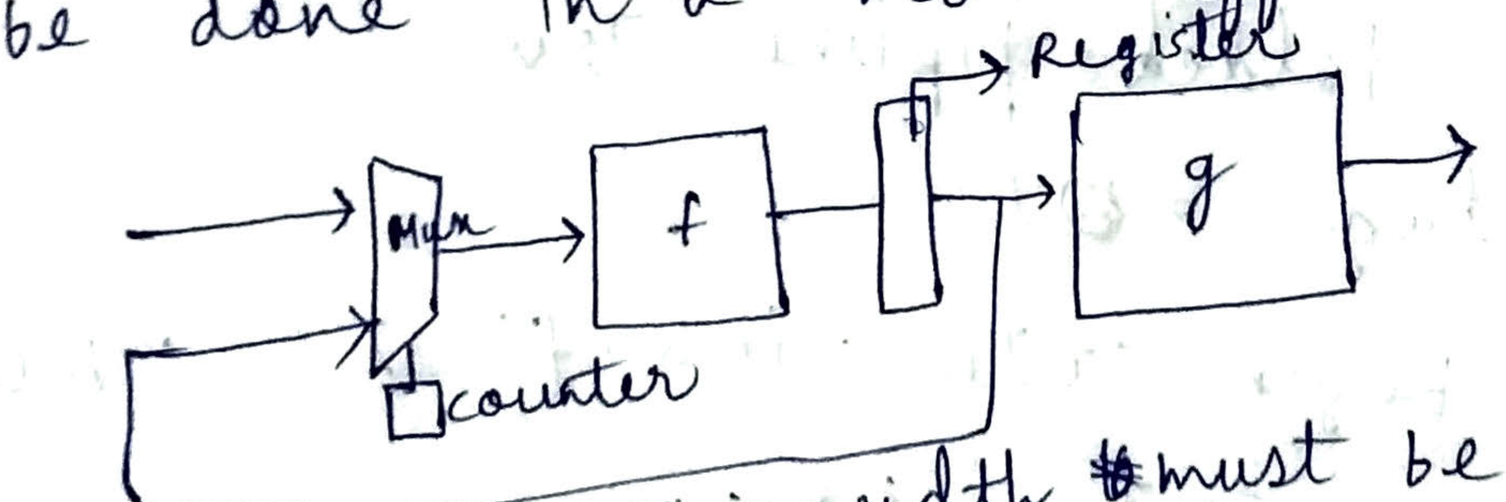
Four inputs each are given to butterfly 4 units which returns 4 complex nos. which are then fed into the permute box which permutes the inputs and this process goes on.

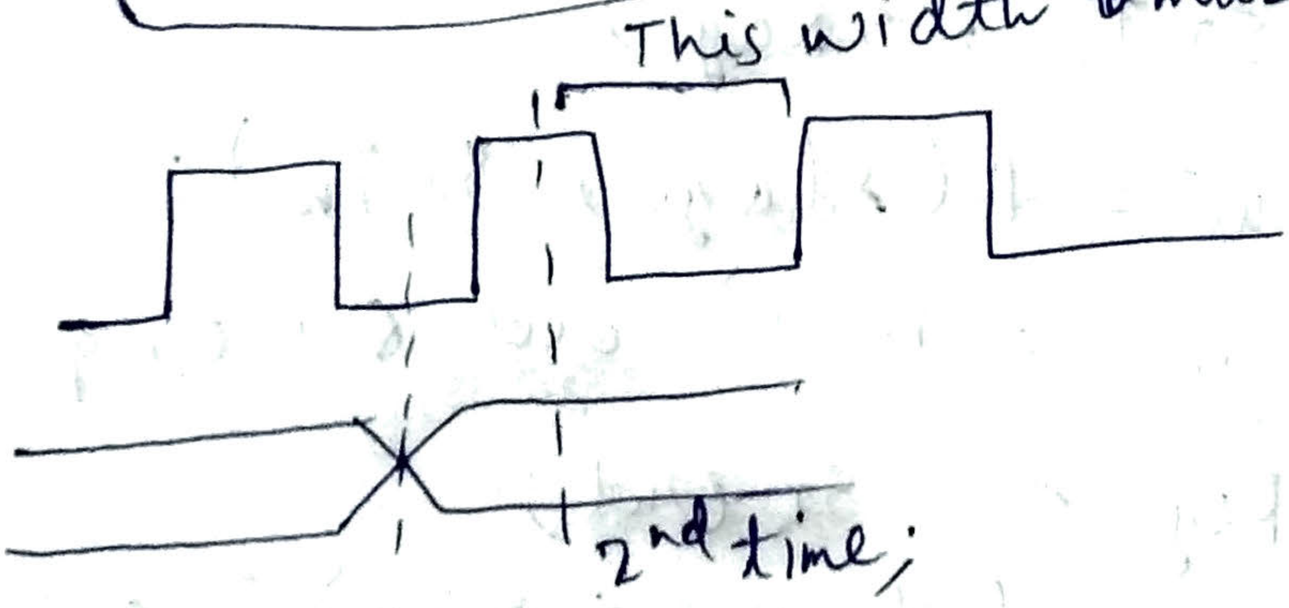
All the bfly 4 units are identical in design but the constant factor (T twidle factor) depends on the position at which the bfly unit is present. We observe in the circuit that the same hardware again and again? Can we optimize this? Yes, with FOLDING.

eg:



can be done in a more efficient way:





1st computation is done

In the first circ:

$$t_{clock} > 2t_f + t_g$$

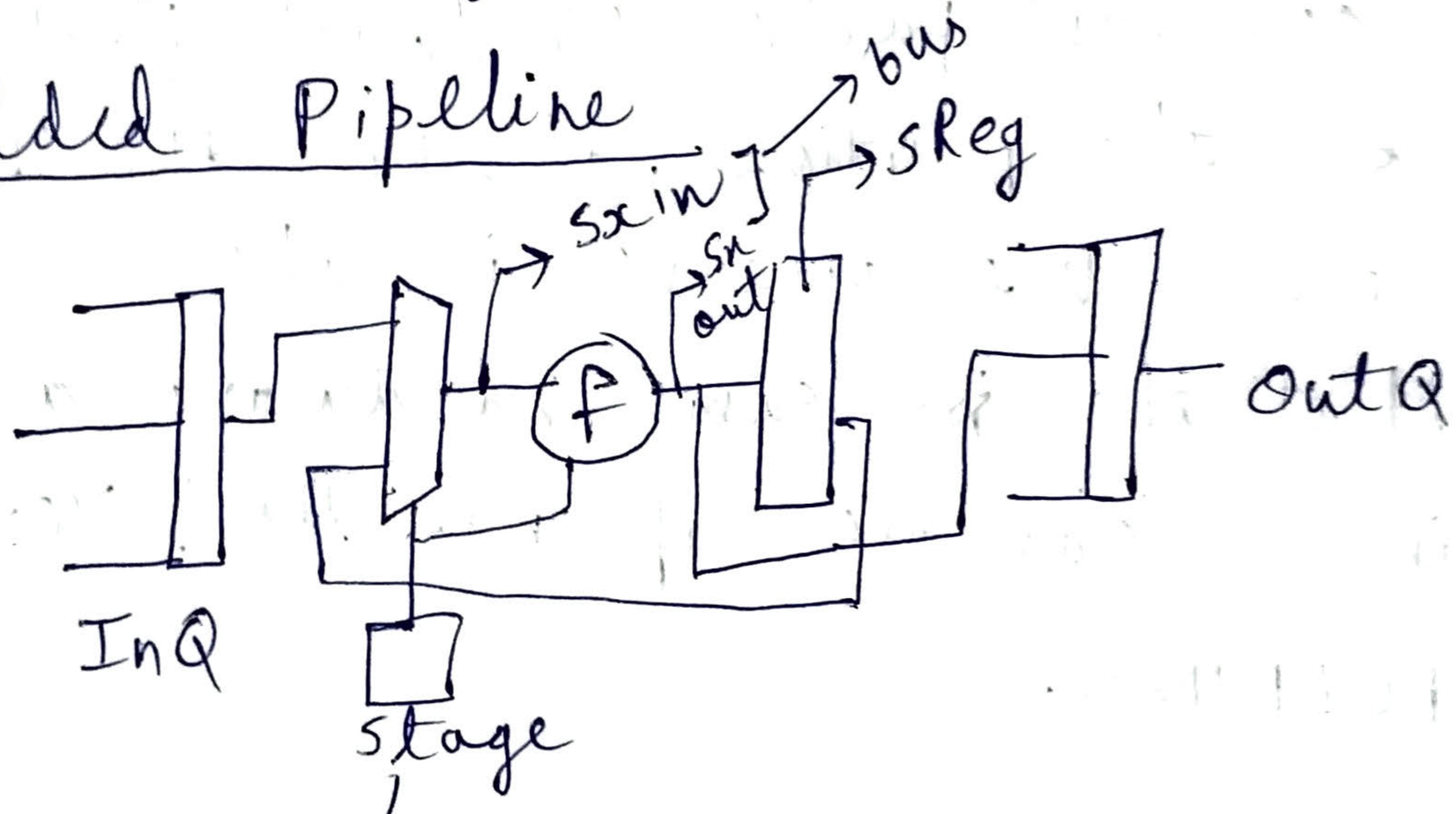
2nd $\Rightarrow$ $t_{clock} > \max(t_{cq} + t_f + t_{su})$

If we try to introduce an extra instruction into the ISA, we may have to pay the price as we may have to increase the clock period. Processor freq. decreases.

[$t_{clock} > \max(t_{add}, \cdots, t_{IFFT})$]

So by folding we have opt. on space.

Folded Pipeline



stage gets updated only at the rising edge as it is a seq. element.

```
rule folded-pipline (True);
  if (stage == 0)
    begin sxIn = inQ.first(); inQ deqed
  else  sxIn = sReg;
  sxOut = f(stage, sxIn);
  if (stage == n-1) outQ.enq(sxOut);
endrule  else sReg <= sxOut;
  stage <= (stage == n-1) ? 0 : stage+1;
```

Having a Queue makes our circ. flexible. We need not wait for ~~that~~ the reqd no. of clock cycles. We can just push it into the Queue.

Even if we change the order of statements in the code, it doesn't affect the output. As update happens only at the next clock cycle.

'=' symbol: naming a bus.

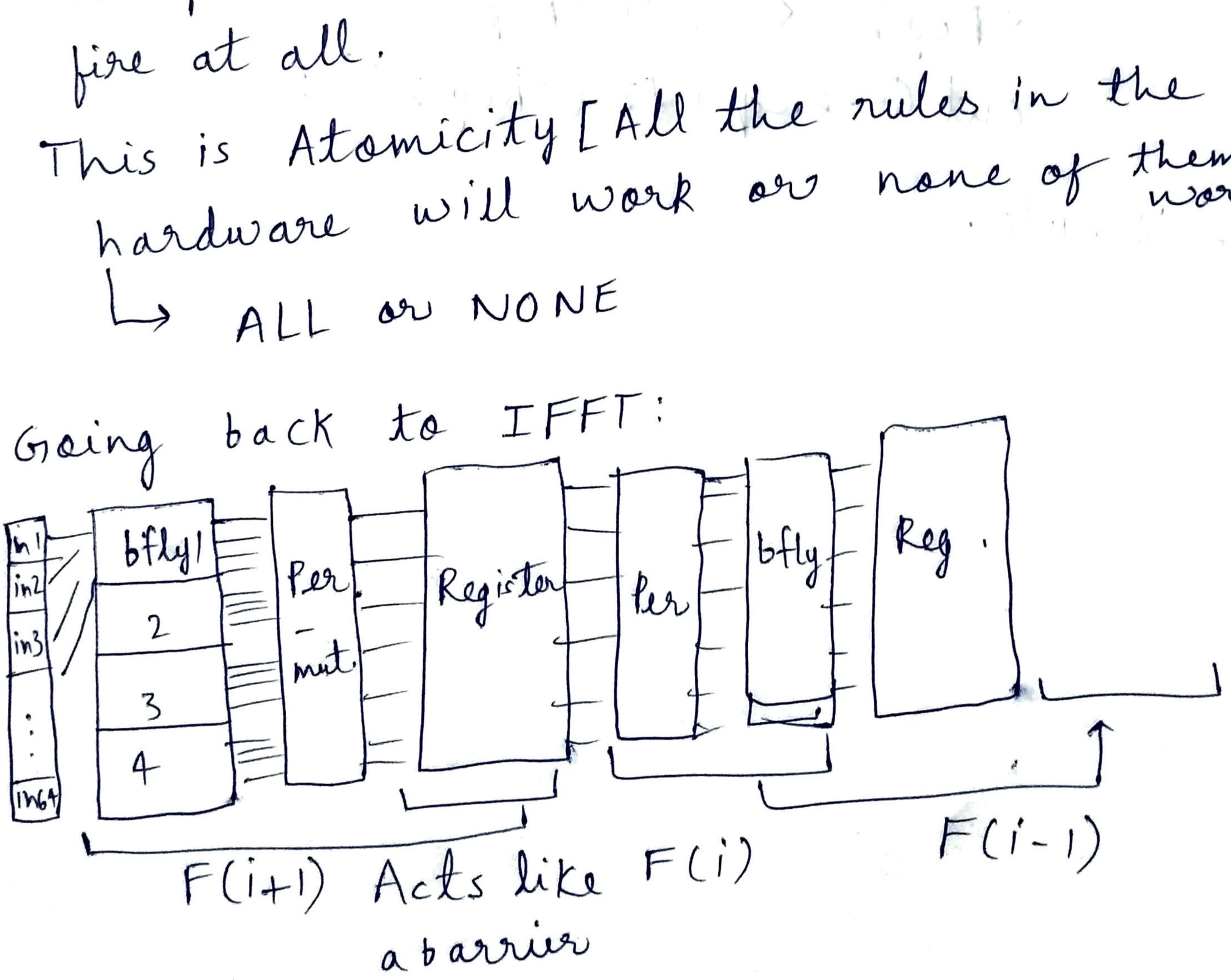
⇐ → state element.

~~B~~ Border Cases: if (in Q is ~~full~~ empty || out Q is full) bluespec sees to it that the rule doesn't fire at all.

This is Atomicity [All the rules in the hardware will work or none of them work]

↳ ALL or NONE

Going back to IFFT:



Although this takes 3 clock cycles, we get one output / every clockcycle.
⇒ Throughput is 1output / clock cycle.
So we gain on throughput.

If we see pipeline and Multicycle:
~~pipeline~~ both of them have similar clock speed but throughput is lost in M. Cycle.

Pipeline Code:

```
rule sync-pipeline (True);
    inQ.deq();
    sReg1 <= f0 (inQ.first());
    sReg2 <= f1(sReg1);
    outQ.enq (f2 (sReg2));
end rule.
```