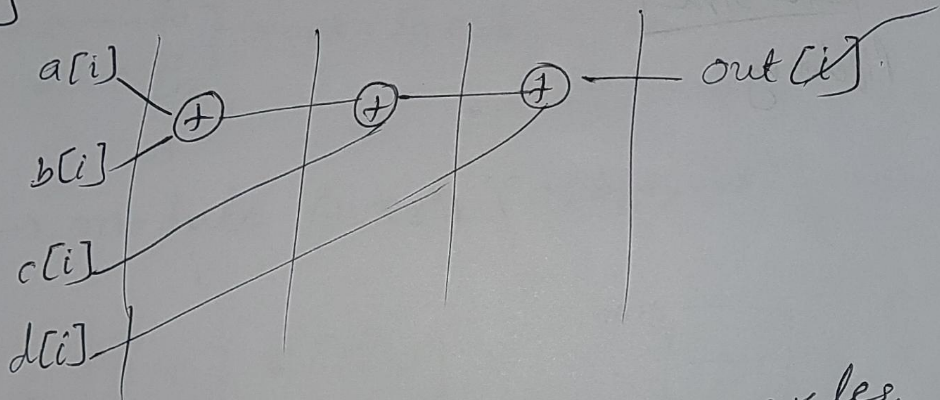
⇒ HLS for loop

```
void fn (a[10], b[10], c[10], d[10])
{
    for (i=0; i<10; i++){ # pragma HLS pipelining
        out[i] = a[i]+b[i]+c[i]+d[i];
    }
    return out;
}
```

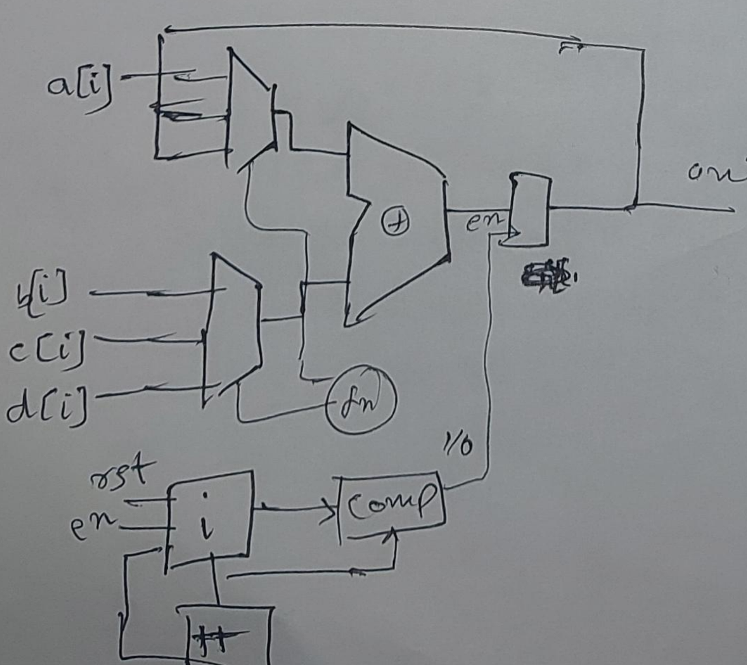
3x10 cycles

clk to execute itr : K

itrations : n



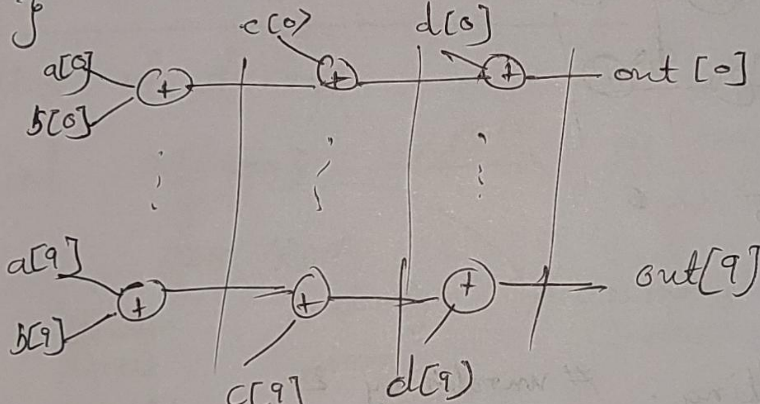
1) iterative implementation : 30 cycles. (K*n)



2) Pipelined implementation: 12 cycles $(k + (n-1) \times 1)$
$= n + k - 1$

3) Unrolling:

```
void fn(a[10], b[10], c[10], d[10]) {
    out[0] = a[0] + b[0] + c[0] + d[0]
    ⋮
    out[9] = a[0] + b[0] + c[0] + d[0]
    return out;
}
```

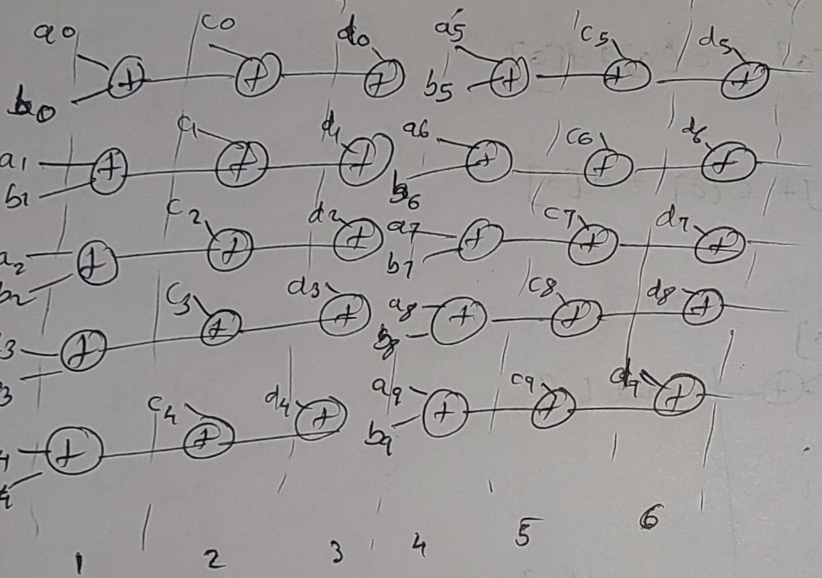


	time	Resources
Iterative	30	1
Pipelined	12	3 → best sol^n w.r.t. resource/perf
Unrolled	3	10
Partial unroll	15	2

3a) Unroll with resource constrained → (6 cycles)

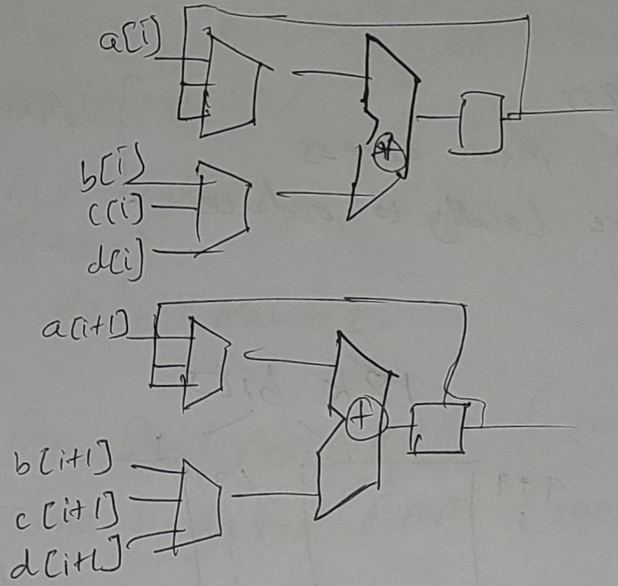
RC = 5 adders

obj. = ~~minimize~~ schedule with min time stamp
w/o violating RC



4) Partial Unrolling: # unroll by 2

```
void fn(a[10], b[10], c[10], d[10]) {
    for (i=0; i<10; i+=2) {
        out[i] = a[i] + b[i] + c[i] + d[i];
        out[i+1] = a[i+1] + b[i+1] + c[i+1] + d[i+1];
    }
    return out;
}
```



15 cycles

HLS of Arrays

int a[1000] $\xleftrightarrow{MAP}$ m[1000][31 : 0]

RAM

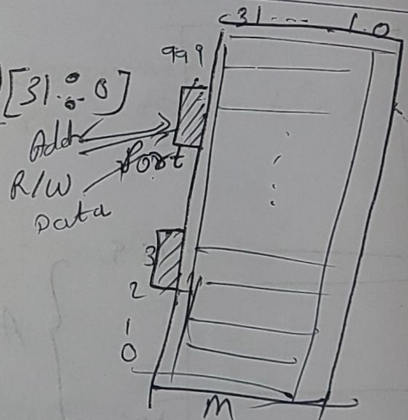
↓

RAM / ROM

Random Access memory

Readonly memory

Access through Index

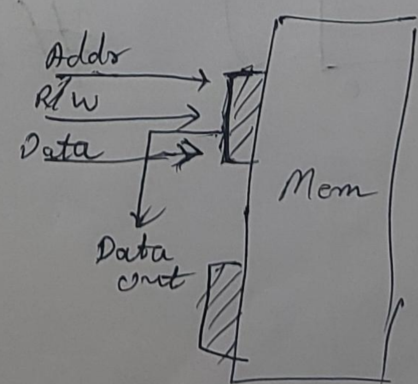


memory: 1/2 ports

```
void fun(int A[20], int* s){
  for(i=3; i<20; i++)
     s[i] = A[i]+A[i-1]+A[i-2]
            +A[i-3]
}
```

mem access = 68

⇓




```
void fun (int A[20], int * s){
   t1 = A[0], t2 = A[1], t3 = A[2]
   for (i=3; i<20; i++){ x = A[i]
     s[i] = x + t1 + t2 + t3;
     t1 = t2
     t2 = t3
}    t3 = x
```

Mem Access = 20

Array

1) Reduce Mem Access
2) Remove Redundant Mem access
3) Read once and store locally to reduce

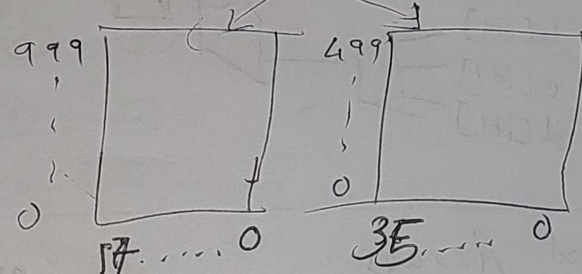
Manually

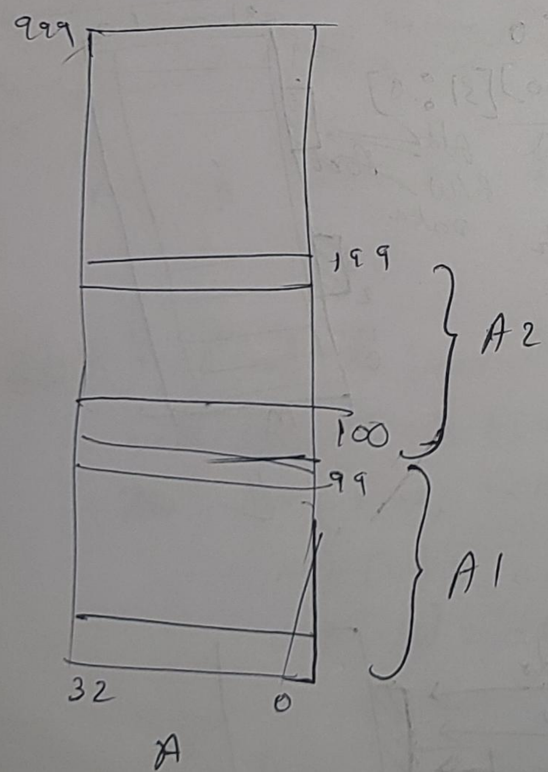
→ **Array Merging:**

int A[100]

int B[100]

18k bits





A1+A2 => A

A2[99:0] <—> A[199:100]

A1[99:0] <—> A[99:0]

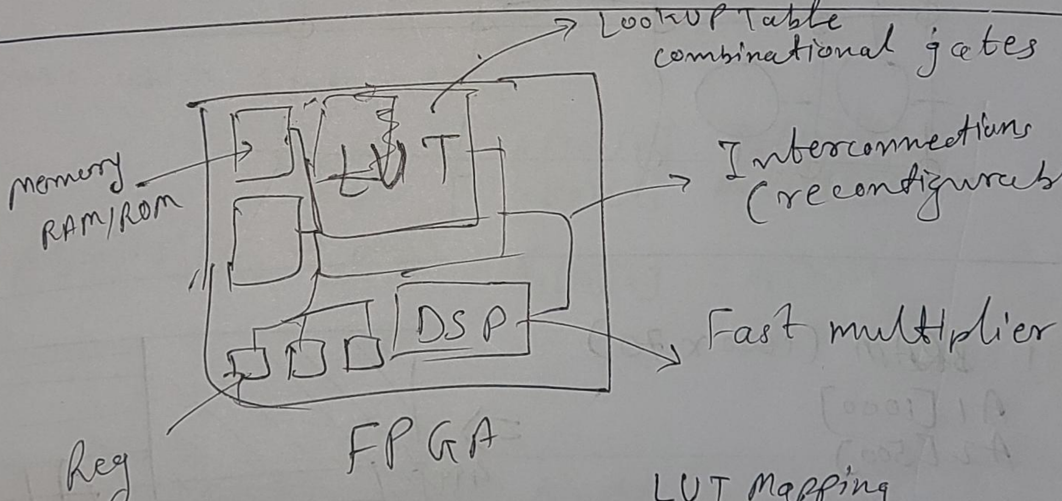
#pragma HLS ARRAY_MAP variable A1 instance A horizontal

#pragma HLS ARRAY_MAP variable A2, instance A horizontal

```
(A) int A1[1000], A2[500]; # Map A1 -> A ; Map A2 -> A
for (i=0; i<1000; i++)
{
    A1[i] = value1;
    if (i<500)
        A2[i] = value2;
}
```

Given: One Dual Port RAM (2000 X 32)

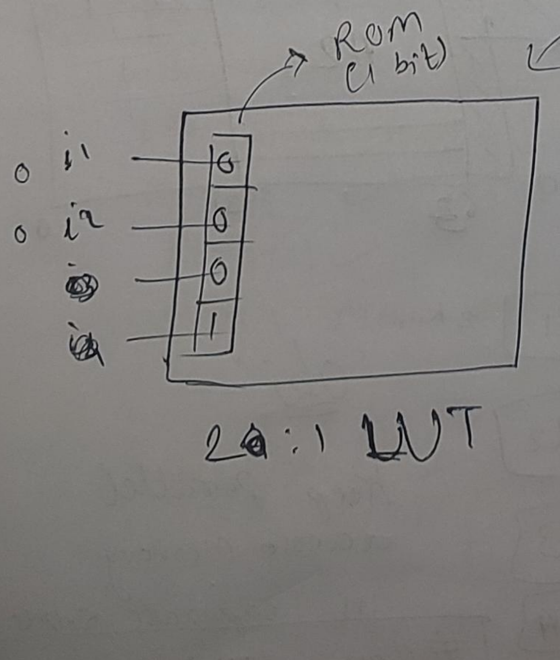
Given: One Single Port RAM (1000 X 70)



LUT Mapping

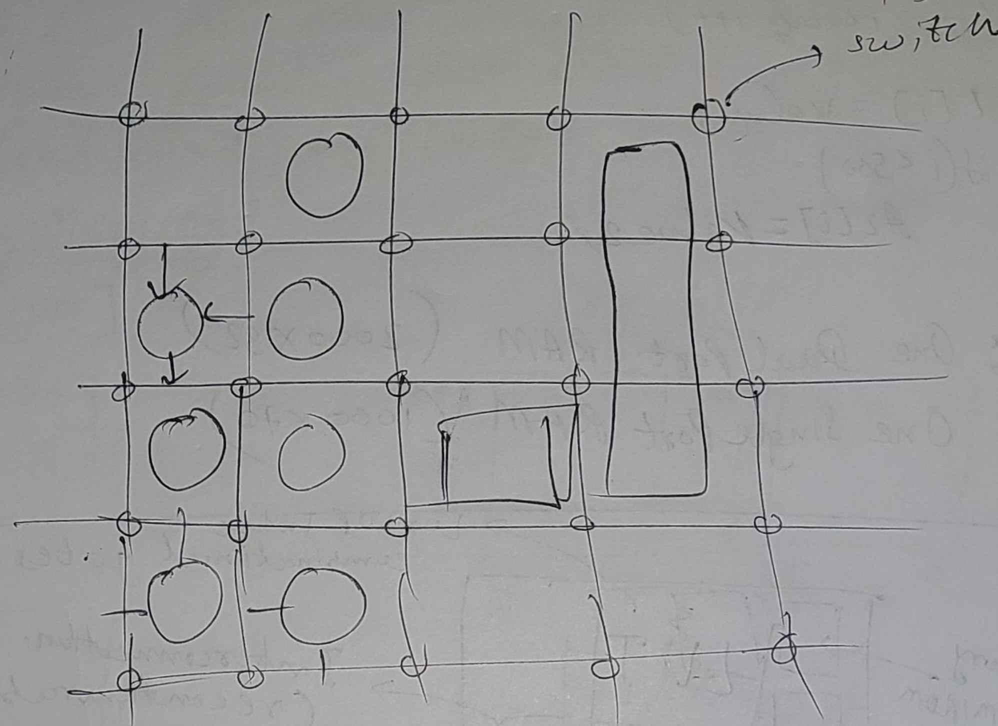
AND

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1



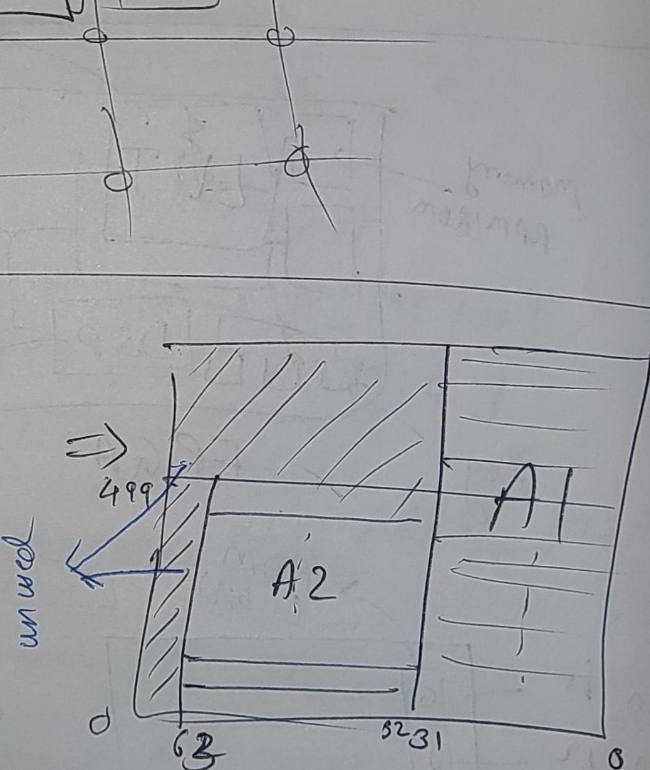
4:1 LUT

Interconnection

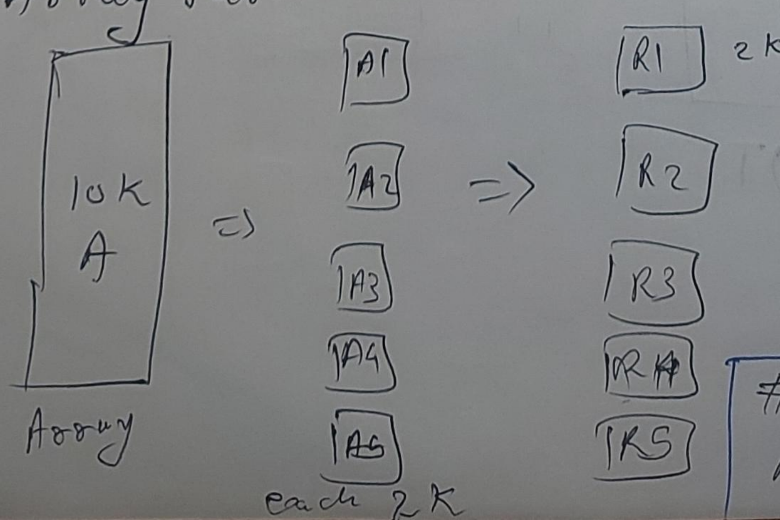


⊛ ⇒ given 1 BRAM (1000 x 70)
A1 [1000]
A2 [500]

⇒



⇒ Array Partition

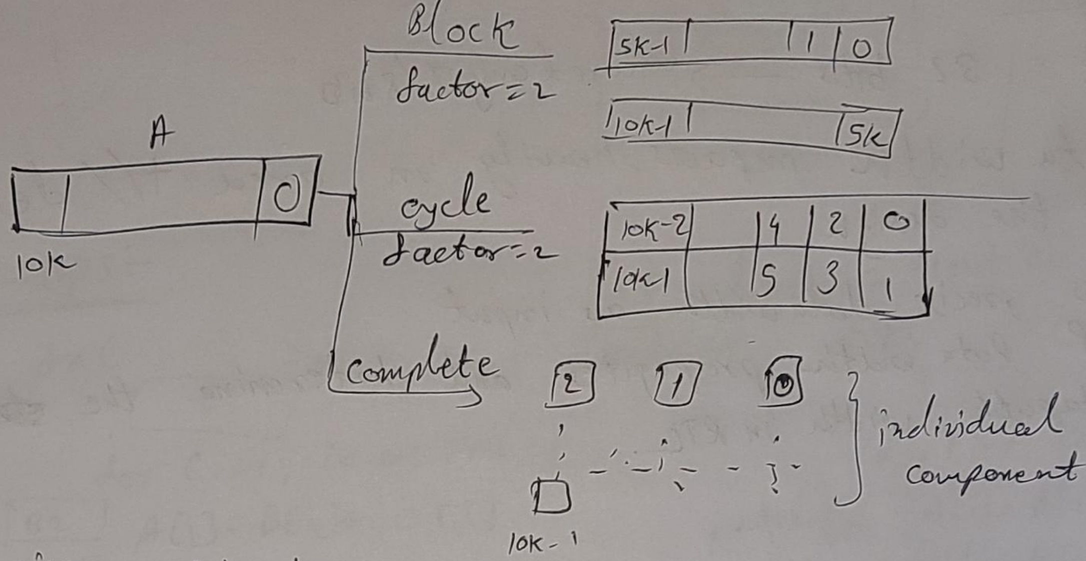


Obj:

Keep parallel accesse memory in saperate RAMs.

#Pragma HLS array-partition
Array name <type> factor = __
Dim = __

type: block/cycle/complete



Dim: which dim to partition

Dim=3: $A_1[10][6]$, $A_2[10][6]$, $A_3[10][6]$, $A_4[10][6]$

A[10][6][4]

Dim=2: $A_1[10][4]$ . . . . $A_6[10][4]$

Unsupported C constructs

Data types: int, char, float ✓

structure, Union: ✓ printf/scanf : ✗ → Remove

pointer: ✓

Array: ✓

Dynamic mem Alloc (malloc/calloc): ✗

[sol: Rewrite program by statistically allocating memory]

Function: ✓

Recursive Function: ✗ ← Rewrite non recursive version as the fn

standard template Library: ✗ → write it yourself.

System calls → ✗ → Remove them

Coding style for HLS

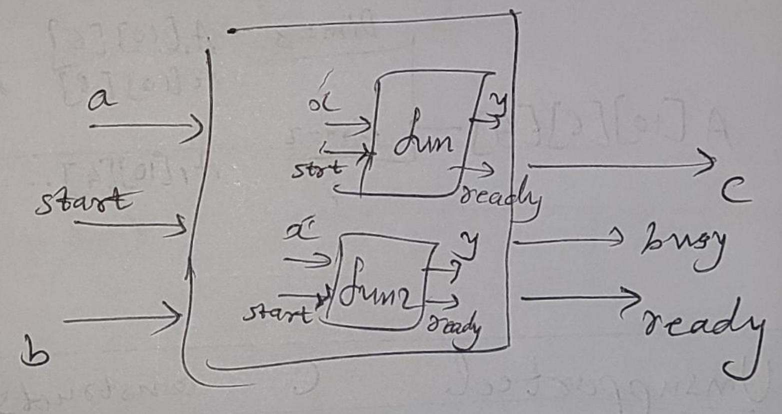
Data types

Int : 32 bits, ⟶ required only 7/8 bits

1) Data width impacts heavily on area +// delay of the design
2) Uses precise datawidth as input
3) I/P Data width propogates and determine the datapath width in RTL.

Function

```
top (a, b, *c) {
    ;
    :
    fun (x, y)
    fun2 (x, y)
    :
}
```

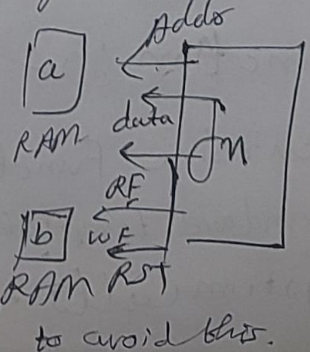


1) small function should be inlined to reduce area overhead.
2) Each fn is allocated a dedicated H/w module
3) Functions can be scheduled in parallel. ⇒ improves performence.
4) Must be careful about the ~~parameters~~ arguments

```
top (...) {
    int a[100], b[100];
    :
    fn (a[], b[]);
}
```

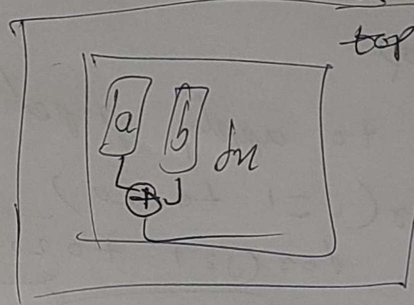
declare inside

5) to avoid



6) Array may be duplicated to avoid this.

```
fn( ) {
   int a[100], b[100]
}   ;
```



Loops:-

```
fn(        ){
   B1
   for(i=1; i<100; i++){
B2    A[i] = B[i] + C[i]
   }
               200
   for(i=1; i<100; i++){
B3    D[i] = E[i] + F[i]
   }
   B4
}
```

Loop parallization: limit1 = limit2

```
for(   ){
   B1
   for(i=1; i<100; i++){
parallel B2   A[i] = B[i] + C[i]
              D[i] = E[i] + F[i]
   }
   B3
}
```

limit1 ≠ limit2

```
for( ){
   ~~for(i=~~ fn1( ); //loop1  } both scheduled
   fn2( ); //loop2             } parallely
   ;
}
```

Nested Loop:

① when to apply pipeline

```
for (i = 1 to 20)  ——→ pipeline.
    for (j = 1 to 20) {  ——→ pipeline
        [ B1 ]
    }
}
```

⇒ outer loop:

→ Inner loop will be unrolled.

→ Run Faster but with lot of area overhead.

→ pipelines for 20 times

* suggested ⇒ Inner loop:

→ Pipeline inner loop

→ Require less resource

→ pipeline for 400 times.