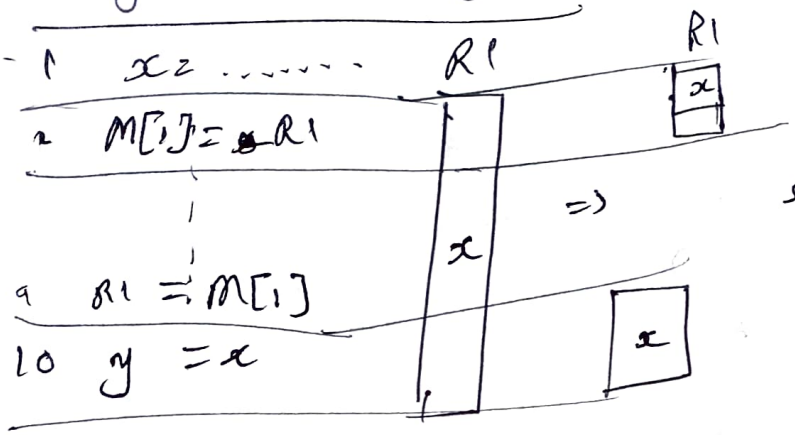
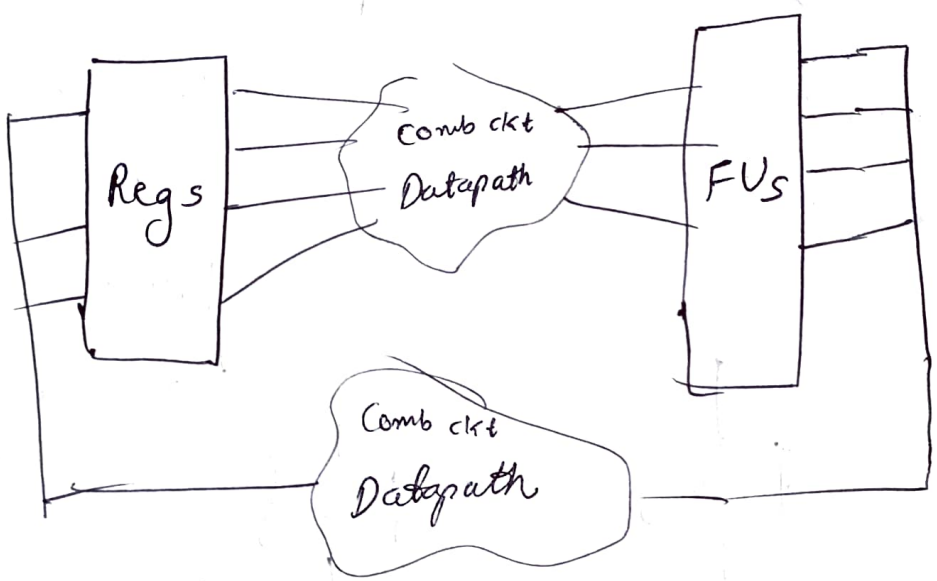
⇒ Register spilling:



⇒ Datapath and Controller generation:



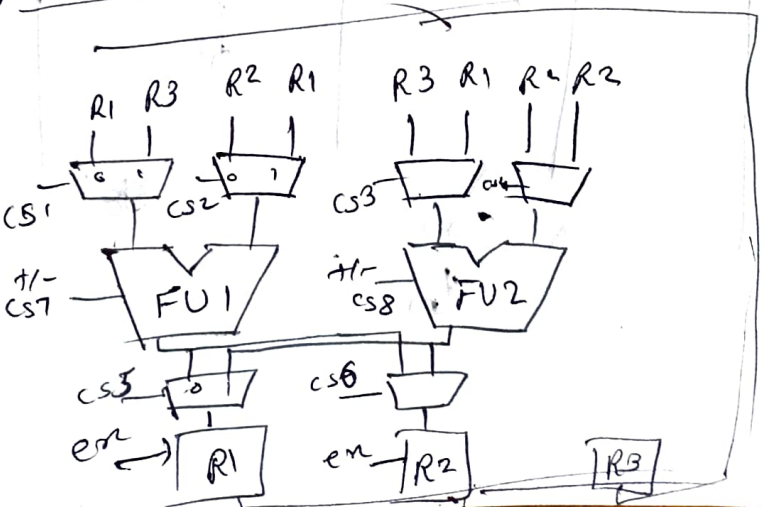
Interconnection
1) MUX based
2) Bus based

obj: reduce Interconnection cost.
MUX, # switch, Mux size

T1. $R1 = FU1_{+}(R1, R2)$
$R2 = FU2_{+}(R3, R2)$

T2. $R2 = FU1_{-}(R3, R1)$
$R1 = FU2_{-}(R1, R2)$

⇒ Mux based datapath



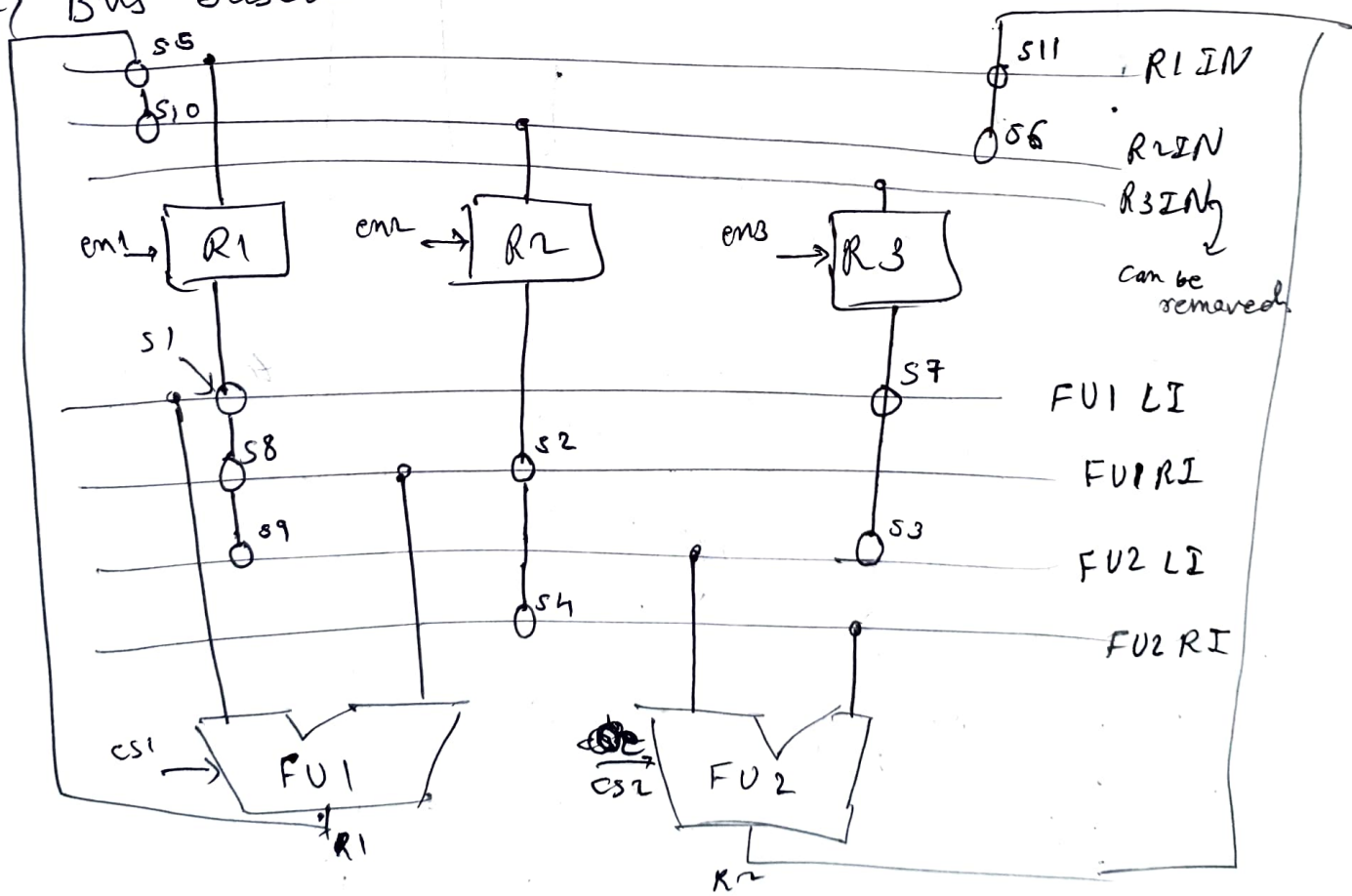
	CS1	CS2	CS3	CS4	CS5	CS6	CS7	CS8	en1	en2	en3
T1	0	0	0	0	0	1	0	0	1	1	0
T2	1	1	1	1	1	0	1	1	1	1	0

CS1 CS7

	CS1	CS5	CS6	CS7	en1	en2	en3
T1	0	0	1	0	1	1	0
T2	1	1	0	1	1	1	0

→ M2 & M4 can be removed. by changing T1: R1 = FU1 + CR2, R1

=> Bus based



	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	en1	en2	en3	cs1	cs2
T1	1	1	1	1	1	1	0	0	0	0	0	1	d	0	0	0
T2	0	0	0	0	0	0	1	1	1	1	1	b	1	0	1	1

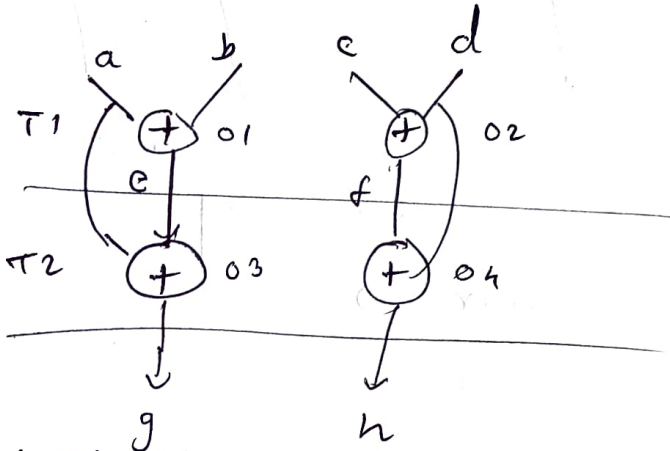
→ Optimization obj:

- Minimum no. of buses to accomodate all data transfers.
- Find max. no. of data transfer that can be accomodated in a fixed number of buses.

=>

T1: e = a+b
f = c+d

T2: g = a+e
h = f+d



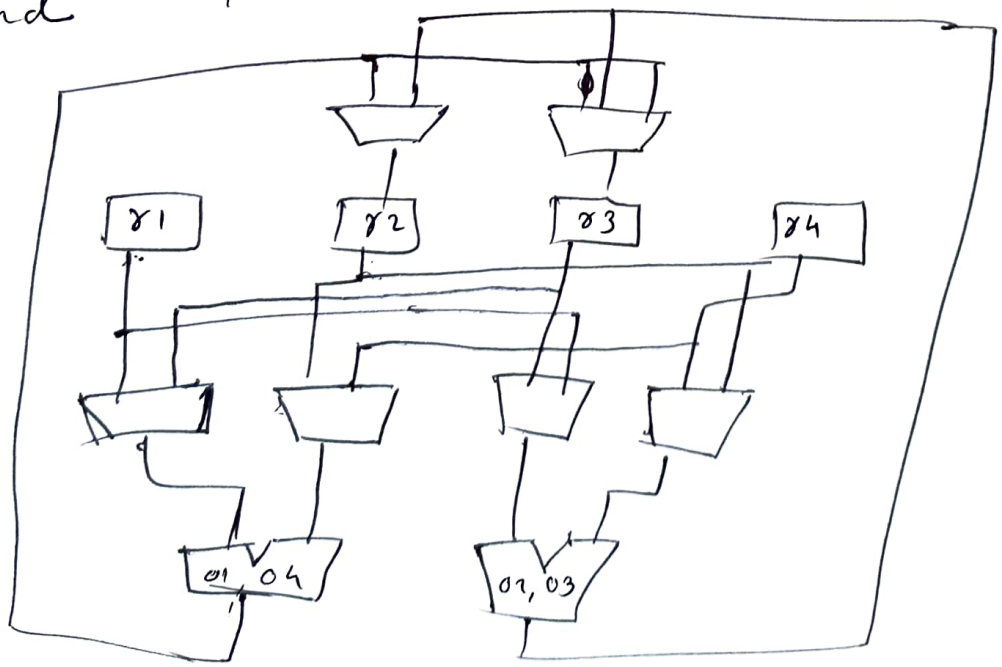
r1 = a
r2 = b, e, g
r3 = c, f, h
r4 = d

Reg bind

ALU 1 : O1, O4

ALU 2 : O2, O3

FU bind

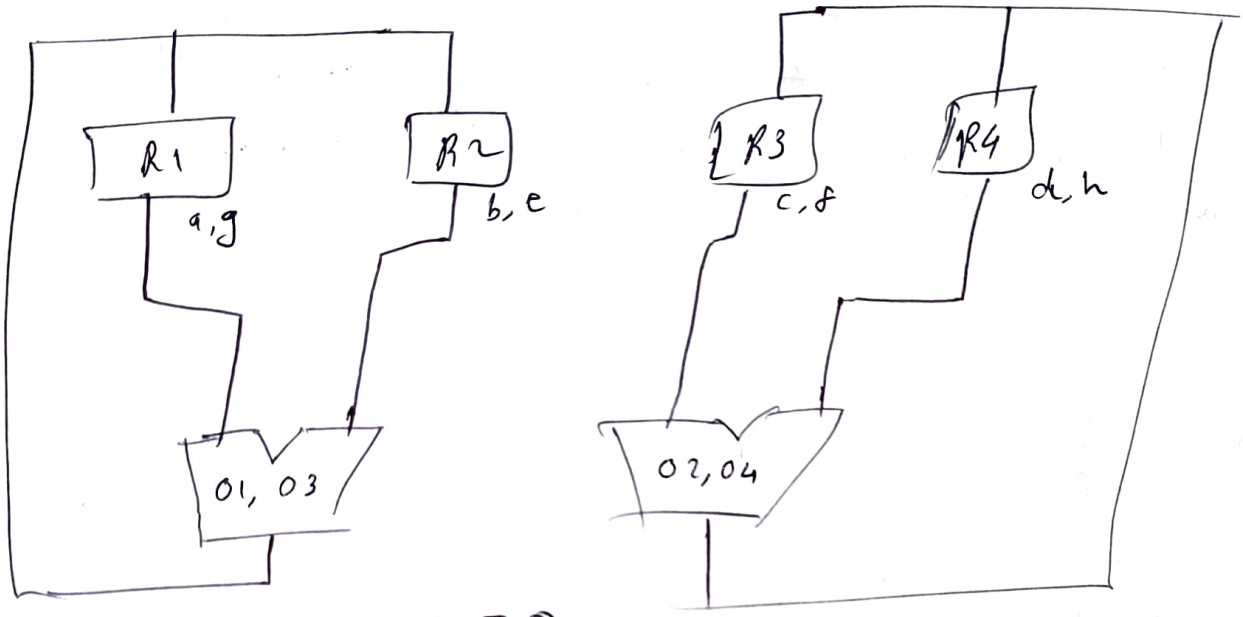


6 2:1 Mux

→ other possible binding

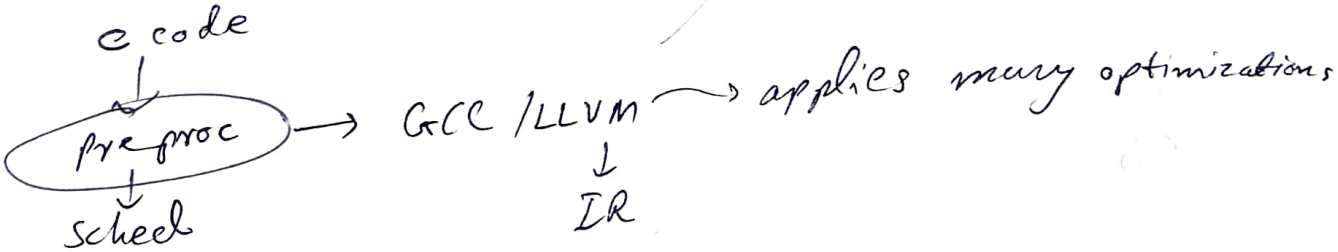
ALU1 : 01, 03
ALU2 : 02, 04

R1 : a, g
R2 : b, e
R3 : c, f
R4 : d, h



2 : 1 mux = 0

Frontend Optimization and its impact

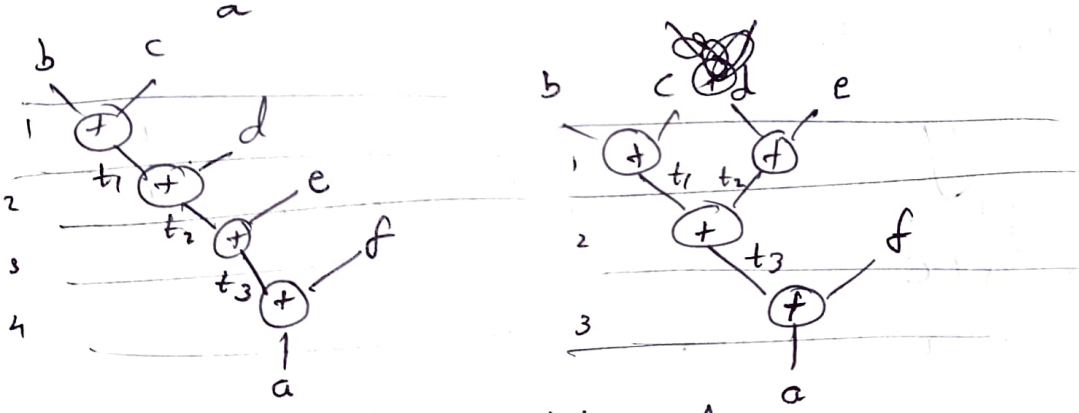


→ $a = b + c + d + e + f$ — 3 adder gen

$t_1 = b + c$
$t_2 = t_1 + d$
$t_3 = t_2 + e$
$a = t_3 + f$

or

$t_1 = b + c$
$t_2 = d + e$
$t_3 = t_1 + t_2$
$a = t_3 + f$



Tree Height Reduction.

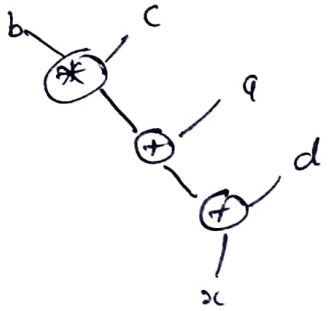
- To force 2nd → add paranthesis

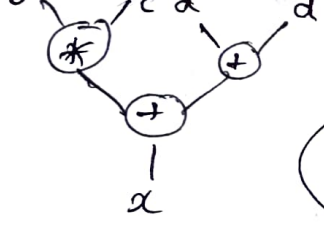
$a = (b + c) + (d + e) + f$

→ $x = a + b * c + d$ ⇒ $x = a + d + b * c$ (commmutativity)

or

Associativity
Distributivity
rewrite expressions to reduce tree height





→ Constant Propogation

```
a = 0                a = 0
b = a + 1     =>     b = 1
c = 2 * b            c = 2
```

→) Reduce latency, resources.

→ Copy propogation.

```
a = x                b = x + 1
b = a + 1     =>     c = 2 * x
c = 2 * x
```

– Reduce the registers

→ Common subexpression elimination

– Identify CSE & remove them
– Reduce latency & resources.

```
a = x + y            a = x + y            a = x + y
c = y + x     =>     c = a         =>     b = a + a   or   a << 1
b = a + c            b = a + c
```

→ Operation strength Reduction :

```
b = 2 * a     =>     b = a << 1
```

1) multiply by const ≡ left shift.

```
b = a * 9     =>     b = (a << 3) + a
```

2) Division by const = Right shift

```
a = b / 2     =>     a = b >> 1
```

→

```
→ c = 7
for (i=0; i<N; i++)
    y[i] += c*i;
```

⇓ Induction variable & loop invariant

```
c = 7, k = 0
for (i=0, i<N; i++) {
    y[i] += k;
    k += c;
}
```

→ Code Motion

- Move instruction within a program w/o changing the functionality.

```
a = b+c                    if(c)
d = d-e                        a = b+c
if(c)          =>              x = x+y
    x = x+y   code       else
else          motion           d = d-e
    x = x-d                    x = x-d
```

```
a = ......                 a = ...
 ; if(c) ....      =>      b = a+c
 ; loop() ....   reduce     ; if c,
b = a+C        life time    ; loops
               of the var
```

- Loop invariant codemotion.

```
for (i=0; i<n; i++){              t = a+b;
    t = a+b; // loop invariant    for (i=0, i<n, i++) {
    :                     =>          :
    x[i] = t+y;                       x[i] = t+y;
}                                 }
```

- reduces latency