How to improve performance?

- There are **3** factors:
  - IPC, #instructions, and frequency
  - #instructions is dependent on the compiler → not on the architecture
- Let us look at IPC and frequency

How to improve performance?

- There are **3** factors:
  - IPC, #instructions, and frequency
  - #instructions is dependent on the compiler → not on the architecture
- Let us look at IPC and frequency
- IPC

1 if there are no stalls, otherwise < 1

  - What is the IPC of an in-order pipeline?

Methods to increase IPC

- Forwarding
- Having more not-taken branches in the code
- Faster instruction and data memories

What about frequency?

- What is frequency dependent on …
- **Frequency = 1 / clock period**
- Clock Period:
  - 1 pipeline stage is expected to take 1 clock cycle
  - Clock period = maximum latency of the pipeline stages
- How to reduce the clock period?
  - Make each stage of the pipeline smaller by increasing the number of pipeline stages
  - Use faster transistors

Limits to increasing frequency - II

- What does it mean to have a very high frequency?
- Before answering, keep these facts in mind:

1. Thumb Rule — $P \propto f^3$
2. Thermo-dynamics — $\Delta T \propto P$

P → power
f → frequency
T → Temperature

Limits to increasing frequency - II

- What does it mean to have a very high frequency?
- Before answering, keep these facts in mind:

1. Thumb Rule — $P \propto f^3$
2. Thermo-dynamics — $\Delta T \propto P$

P → power
f → frequency
T → Temperature

3. We need to increase the number of pipeline stages → more hazards, more forwarding paths

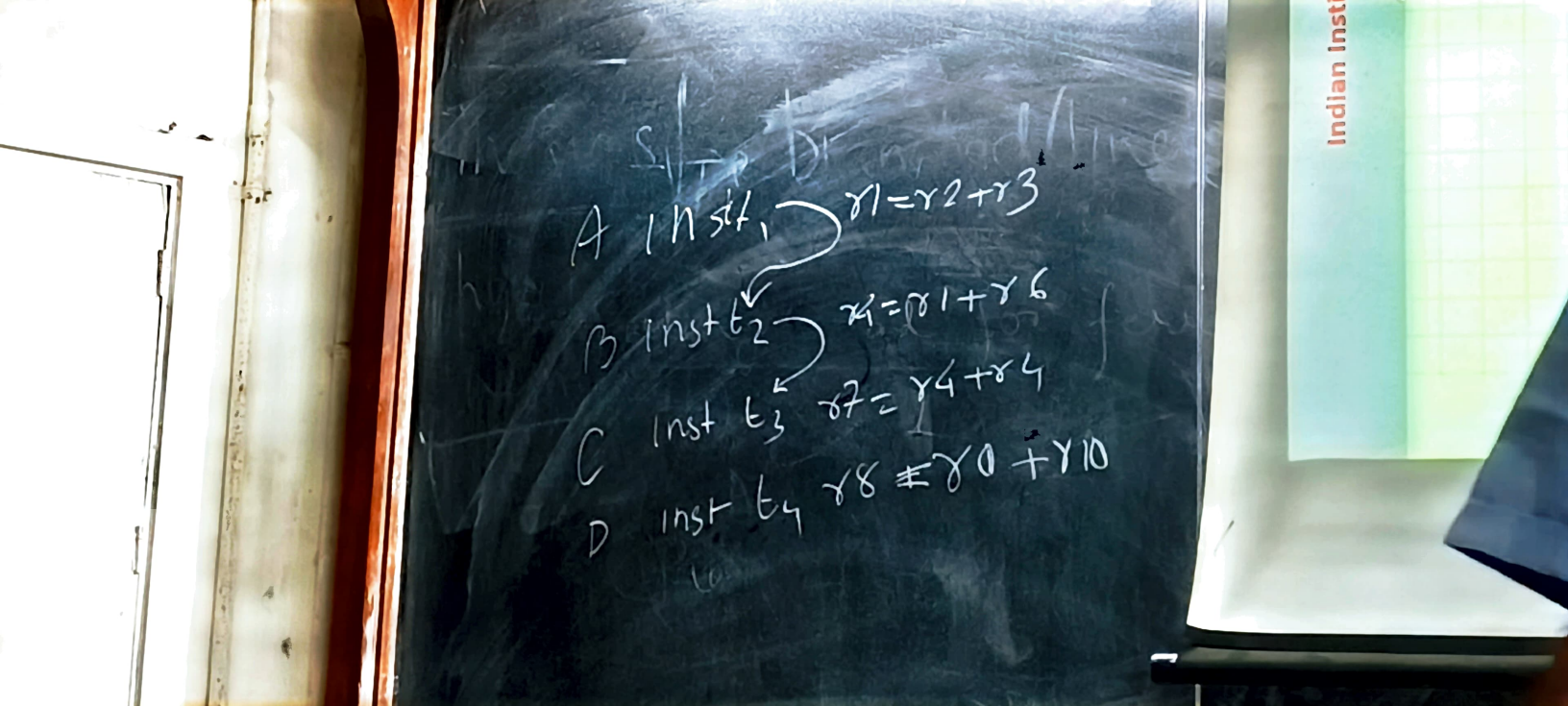
Pipeline Stages vs IPC

- $CPI = \frac{1}{IPC}$

$$CPI = CPI_{ideal} + stall\_rate * stall\_penalty$$

- The stall rate will remain more or less constant per instruction with the number of pipeline stages
- The **stall penalty** (in terms of cycles) will however **increase**
- This will lead to a net increase in CPI and **loss** in IPC

As we increase the number of stages, the IPC goes down.


A Inst1 r1=r2+r3
B Inst t2 r4=r1+r6
C Inst t3 r7=r4+r4
D Inst t4 r8=r0+r10

What is ILP = Instruction level parallelism

- *multiple operations (or instructions) can be executed in parallel, from a single instruction stream*
  - *so we are **not** yet talking about MIMD, multiple instruction streams*

Needed:

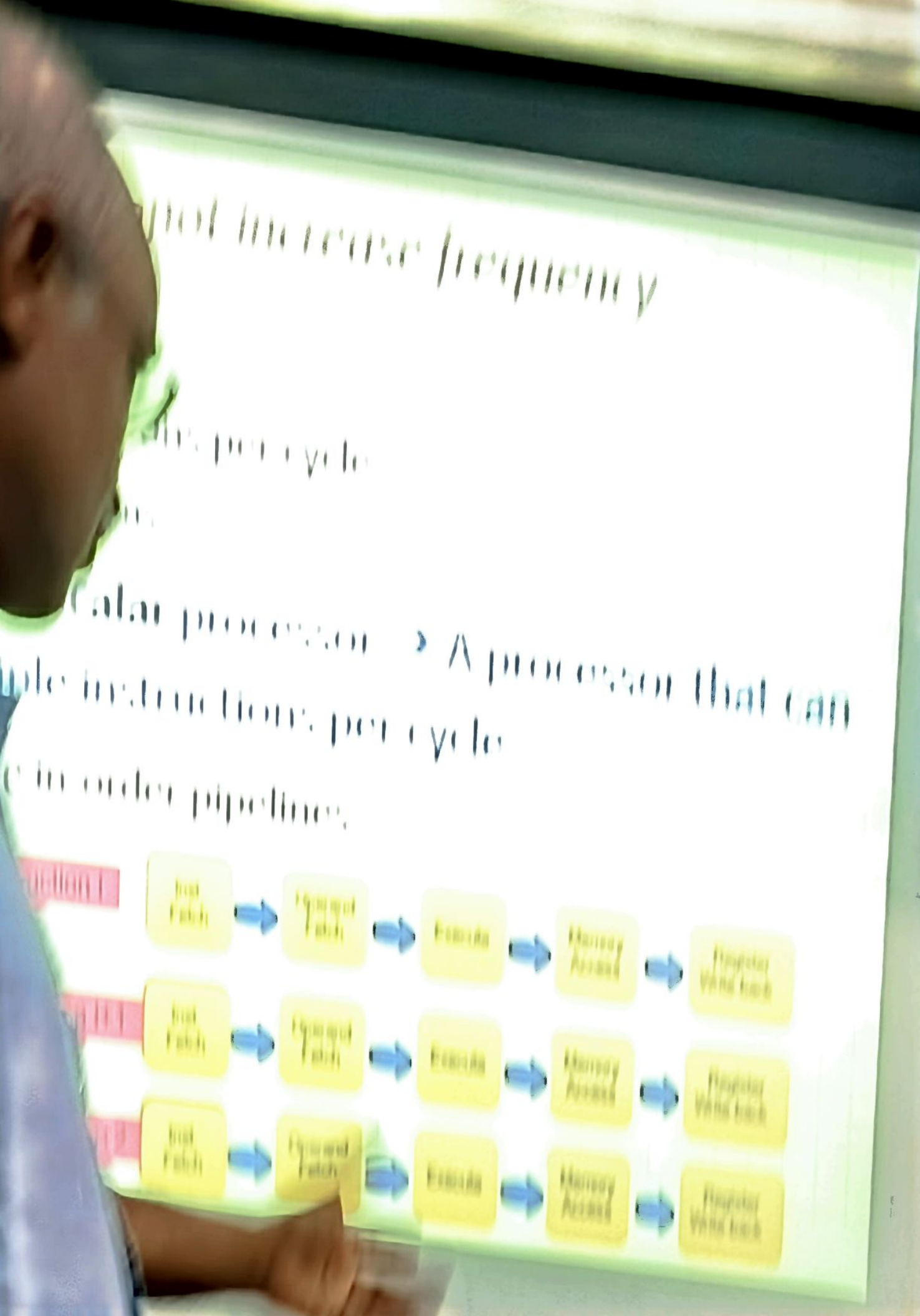
- Sufficient (HW) resources
- Parallel scheduling
  - Hardware solution
  - Software solution
- Application should contain sufficient ILP

What is ILP = Instruction level parallelism

- *multiple operations (or instructions) can be executed in parallel, from a single instruction stream*
  - *so we are **not** yet talking about MIMD, multiple instruction streams*

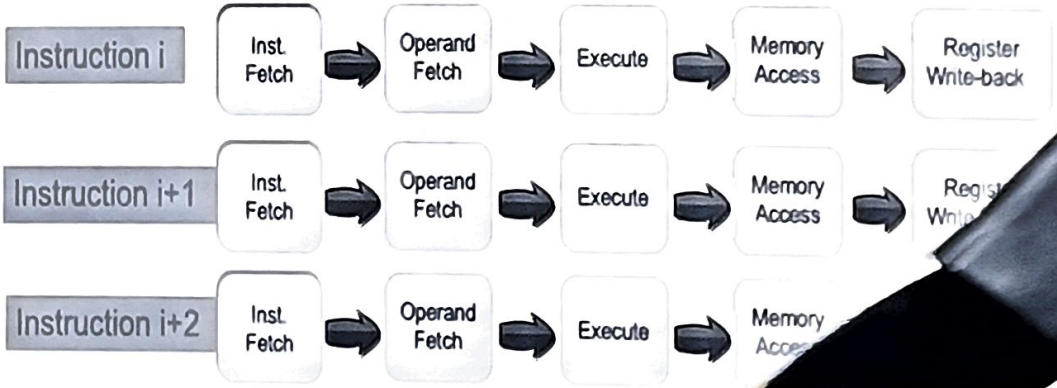
Needed:

- Sufficient (HW) resources
- Parallel scheduling
  - Hardware solution
  - Software solution
- Application should contain sufficient ILP

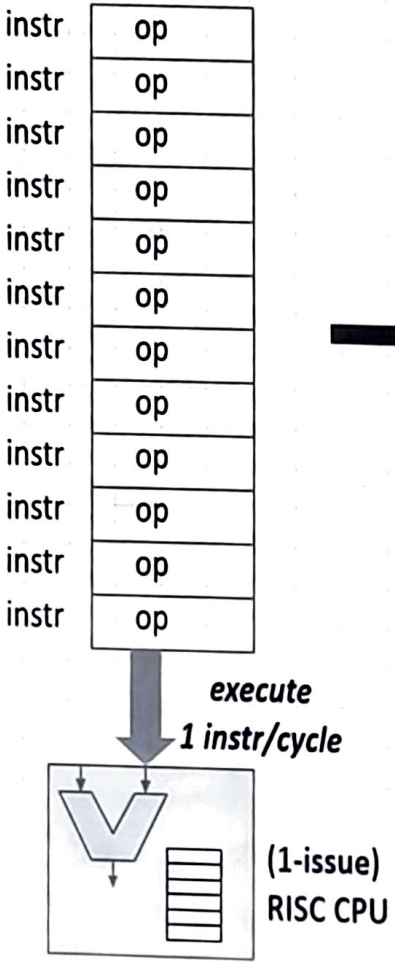


Since we cannot increase frequency ...

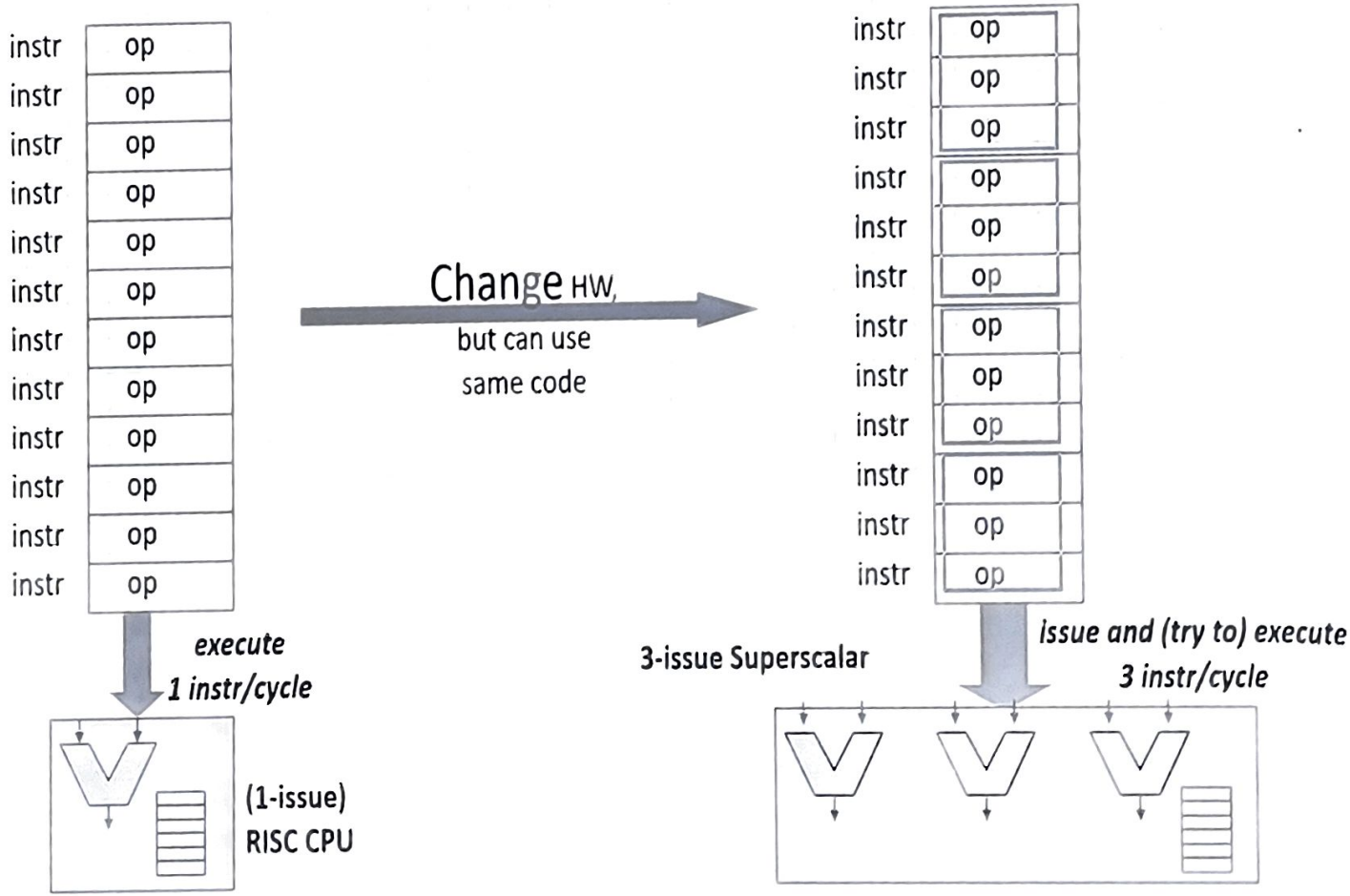
- Increase IPC
  - Issue more instructions per cycle
    - 2, 4, or 8 instructions
- Make it a **superscalar** processor → A processor that can execute multiple instructions per cycle
  - Have multiple in-order pipelines



Single Issue RISC vs Superscalar



Single Issue RISC vs Superscalar



Example of Superscalar Execution

- Superscalar processor organization, example:
  - simple pipeline: IF, EX, WB
  - fetches/issues upto 2 instructions each cycle (= 2-issue)
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)							
L.D	F2,48(R3)							
MUL.D	F0,F2,F4							
SUB.D	F8,F2,F6							
DIV.D	F10,F0,F6							
ADD.D	F6,F8,F2							
MUL.D	F12,F2,F4							

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX				
SUB.D	F8,F2,F6		IF	EX				
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF				
MUL.D	F12,F2,F4							

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX		
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF		EX		
MUL.D	F12,F2,F4					IF		

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX	EX	
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF		EX	EX	
MUL.D	F12,F2,F4					IF		

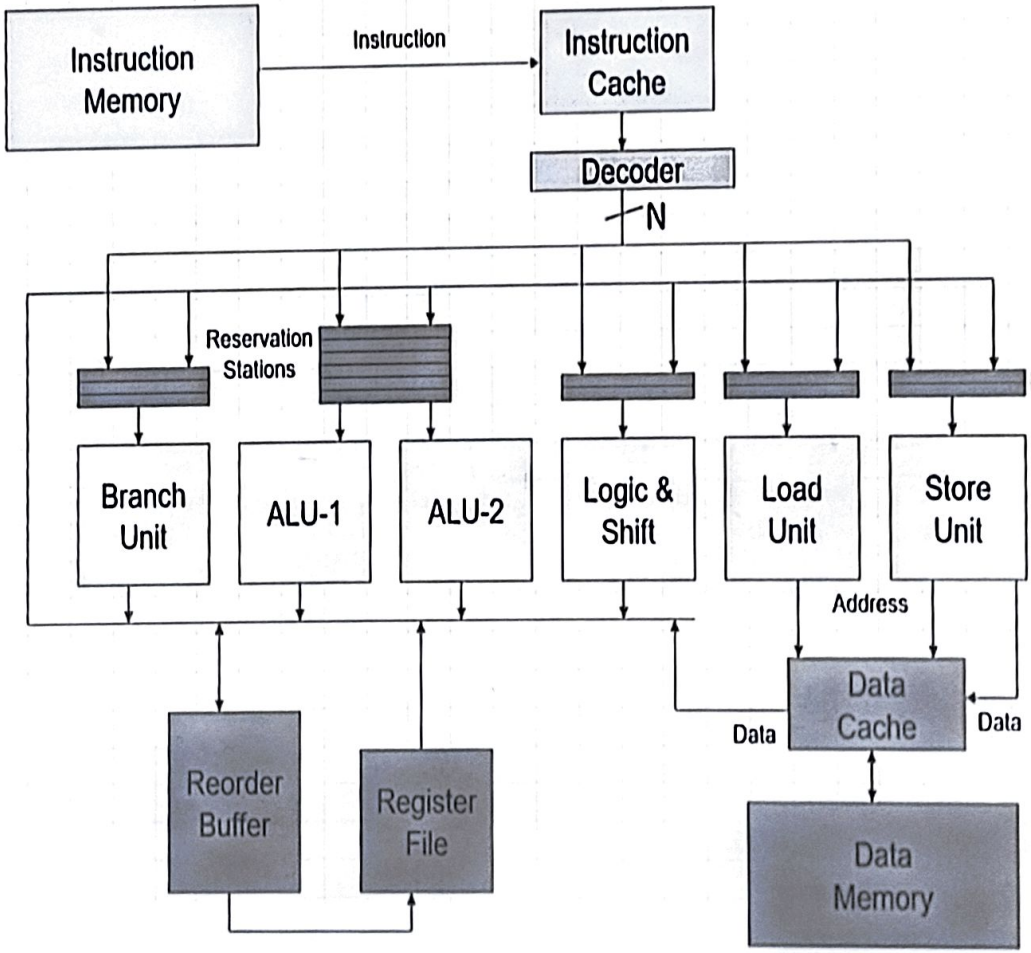
cannot execute structural hazard

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX	EX	WB
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				EX
ADD.D	F6,F8,F2			IF		EX	EX	WB
MUL.D	F12,F2,F4					IF		?

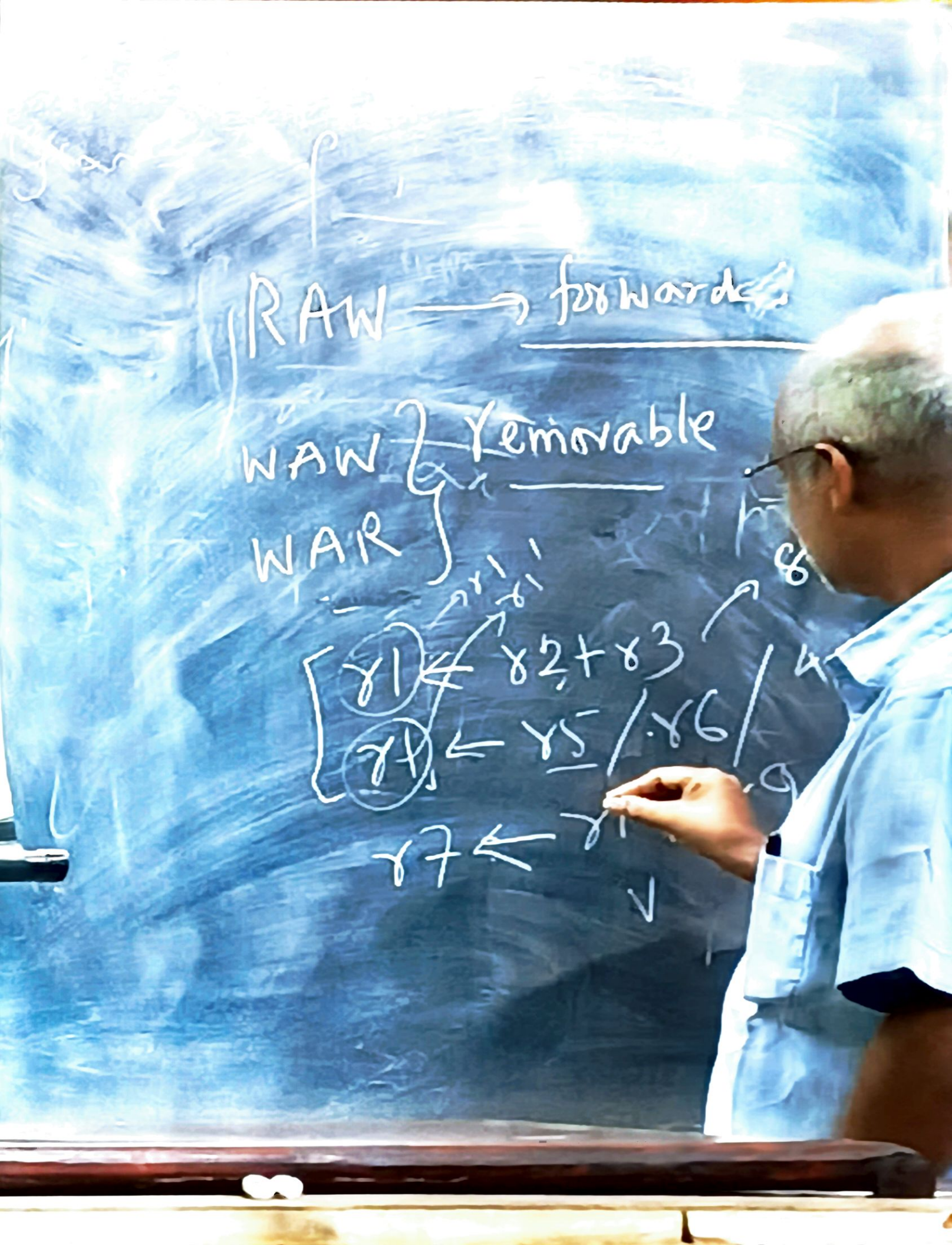
Superscalar: General Architecture Concept

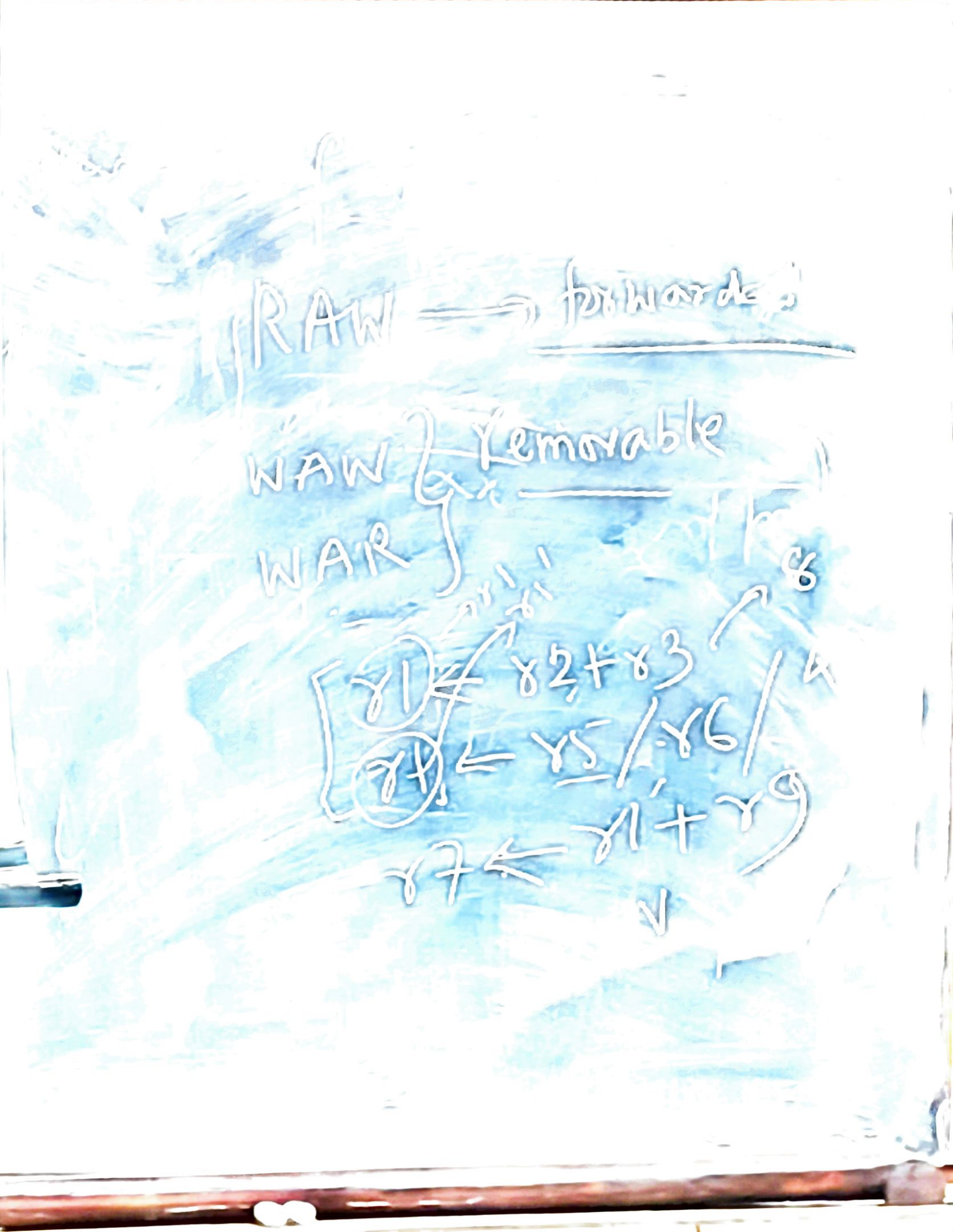


N = issue rate

Hazards

- Three types of hazards
  - **Structural**
    - Multiple instructions need access to the same hardware at the same time
  - **Data dependence**
    - There is a dependence between operands (in register or memory) of successive instructions
  - **Control dependence**
    - Determines the order of the execution of basic blocks
    - When jumping/branching to another address the pipeline has to be (partly) squashed and refilled


RAW → forward
WAW
WAR
removable
r1 ← r2 + r3
r4 ← r5 / r6
r7 ← r1


RAW → forwarded
WAW
WAR
removable

Impact of Hazards

- Hazards cause pipeline 'bubbles'

  Increase of CPI (and therefore execution time)

- $T_{exec} = N_{instr} * CPI * T_{cycle}$

  where

  - $CPI = CPI_{base} + \Sigma_i \langle CPI_{hazard\_i} \rangle$
  - $\langle CPI_{hazard} \rangle = f_{hazard} * \langle Cycle\_penalty_{hazard} \rangle$
  - f_{hazard} = fraction [0..1] of occurrence of this hazard

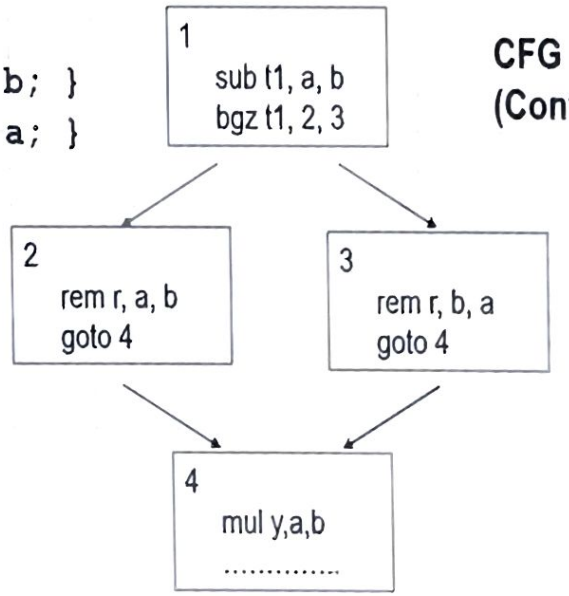
Data dependences

- **RaW** read after write
  - real or flow dependence
  - can only be avoided by value prediction (i.e. speculating on the outcome of a previous operation)
- **WaR** write after read
- **WaW** write after write
  - WaR and WaW are false or name dependencies
  - Could be avoided by **renaming** (if sufficient registers are available); see later slide

Control Dependences: CFG

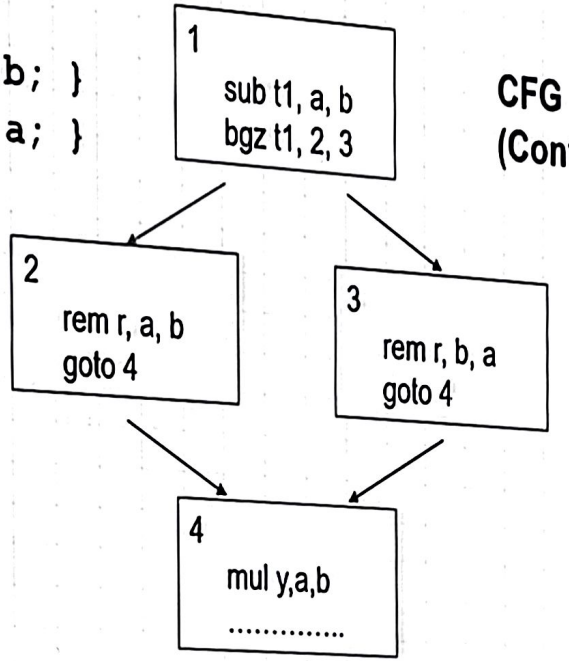
C input code:

```
if (a > b)   { r = a % b; }
   else      { r = b % a; }
y = a*b;
```



CFG (Control Flow Graph):

```
if (a > b)   { r = a % b; }
   else      { r = b % a; }
y = a*b;
```



CFG (Control Flow Graph):

Questions:

- *How real are control dependences?*
- *Can 'mul y, a, b' be moved to block 2, 3 or even block 1?*
- *Can 'rem r, a, b' be moved to block 1 and executed speculatively?*

Avoiding pipeline stalls due to Hazards

- **Structural**
  - Buy more hardware
    - Extra units, pipelined units, more ports on RF and data memory (or banked memories), etc.
  - Note: more HW means bigger chip => could increase cycle time t_{cycle}
- **Data dependence**
  - Real (RaW) dependences: add Forwarding (aka Bypassing) logic
    - Compiler optimizations
  - False (WaR & WaW) dependences: use renaming (either in HW or in SW)

Avoiding pipeline stalls due to Hazards

- **Structural**
  - Buy more hardware
    - Extra units, pipelined units, more ports on RF and data memory (or banked memories), etc.
  - Note: more HW means bigger chip => could increase cycle time t_{cycle}
- **Data dependence**
  - Real (RaW) dependences: add Forwarding (aka Bypassing) logic
    - Compiler optimizations
  - False (WaR & WaW) dependences: use renaming (either in HW or in SW)
- **Control dependence**
  - Adding extra pipeline HW to reduce the number of Branch delay slots
  - Branch prediction
  - Avoiding Branches

Software Techniques - Example

- This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
   x[i] = x[i] + s;
```

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

Instruction producing result	*Instruction using result*	*Latency in cycles*	*stalls between in cycles*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

FP Loop: Where are the Hazards?

- **First translate into MIPS code:**
  - To simplify, assume 8 is lowest address

```
Loop: L.D    F0,0(R1) ;F0=vector element
      ADD.D  F4,F0,F2 ;add scalar from F2
      S.D    0(R1),F4 ;store result
      DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
      BNEZ   R1,Loop  ;branch R1!=zero
```

FP Loop Showing Stalls

- 9 clock cycles: Rewrite code to minimize stalls?

```
1 Loop: L.D    F0,0(R1)  ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2  ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4  ;store result
7       DADDUI R1,R1,-8  ;decrement pointer 8B (DW)
8       stall            ;assumes can't forward to branch
9       BNEZ   R1,Loop   ;branch R1!=zero
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

Revised FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4   ;altered offset when move DSUBUI
7       BNEZ   R1,Loop
```

```
1 Loop: L.D    F0,0(R1) ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2 ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4 ;store result
7       DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
8       stall           ;assumes can't forward to branch
9       BNEZ   R1,Loop  ;branch R1!=zero
```

Swap DADDUI and S.D by changing address of S.D

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?

Unroll Loop Four Times (straightforward way)

Rewrite loop to minimize stalls?

```
1 Loop: L.D    F0,0(R1)
                                  1 cycle stall
3       ADD.D  F4,F0,F2
                                  2 cycles stall
6       S.D    0(R1),F4     ;drop DSUBUI & BNEZ
7       L.D    F6,-8(R1)
9       ADD.D  F8,F6,F2
12      S.D    -8(R1),F8    ;drop DSUBUI & BNEZ
13      L.D    F10,-16(R1)
15      ADD.D  F12,F10,F2
18      S.D    -16(R1),F12  ;drop DSUBUI & BNEZ
19      L.D    F14,-24(R1)
21      ADD.D  F16,F14,F2
24      S.D    -24(R1),F16
25      DADDUI R1,R1,#-32   ;alter to 4*8
26      BNEZ   R1,LOOP
```

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

Unrolled Loop That Minimizes Stalls

```
1 Loop:L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    0(R1),F4
10      S.D    -8(R1),F8
11      S.D    -16(R1),F12
12      DSUBUI R1,R1,#32
13      S.D    8(R1),F16 ; 8-32 = -24
14      BNEZ   R1,LOOP
```

14 clock cycles, or 3.5 per iteration