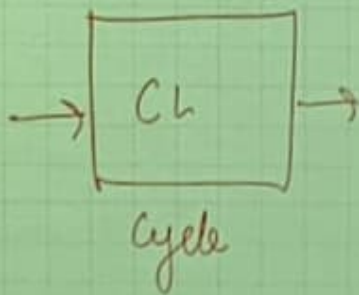




What about frequency?

- What is frequency dependent on ...
- $\text{Frequency} = 1 / \text{clock period}$
- 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

$$P \propto f^3$$

 $P \rightarrow$ power $f \rightarrow$ frequency

2

Thermo-
dynamics

$$\Delta T \propto P$$

 $T \rightarrow$ Temperature

3

We need to increase the number of pipeline stages \rightarrow
more hazards, more forwarding paths

What is ILP = Instruction level parallelism

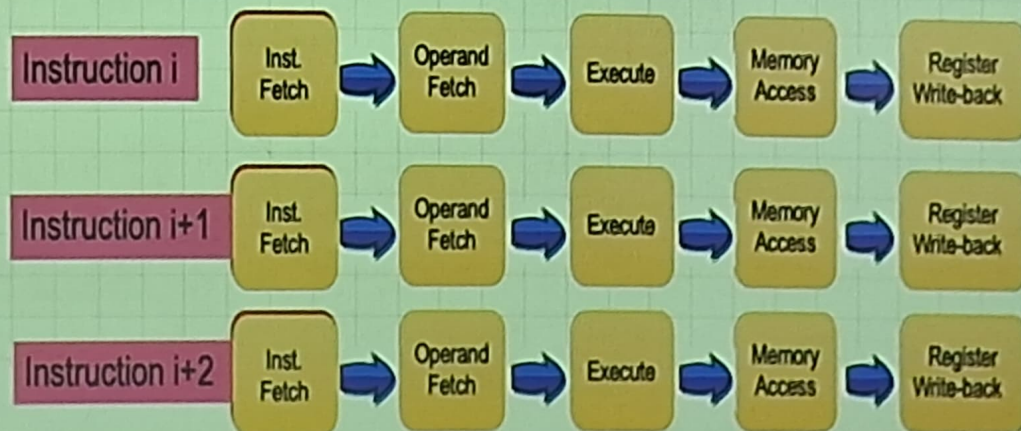
- multiple operations (or instructions) can be executed in parallel, from a single instruction stream
 - so we are **not** yet talking about MIMD, multiple instruction streams

Needed:

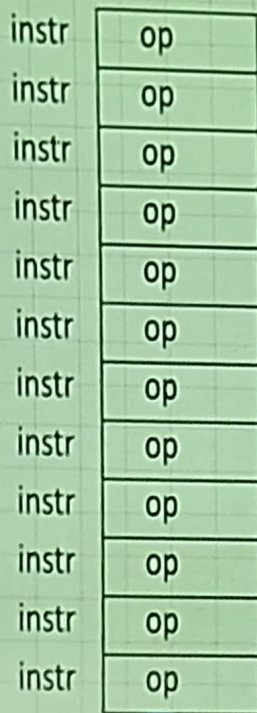
- Sufficient (HW) resources
- Parallel scheduling
 - Hardware solution
 - Software solution
- Application should contain sufficient ILP

Since we cannot increase frequency ...

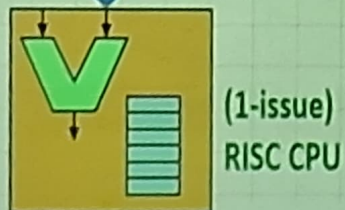
- Increase IPC
 - Issue **more** instructions per cycle
 - 2, 4, or 8 instructions
- Make it a **superscalar processor** → A processor that can execute multiple instructions per cycle
 - Have multiple in-order pipelines



Single Issue RISC vs Superscalar

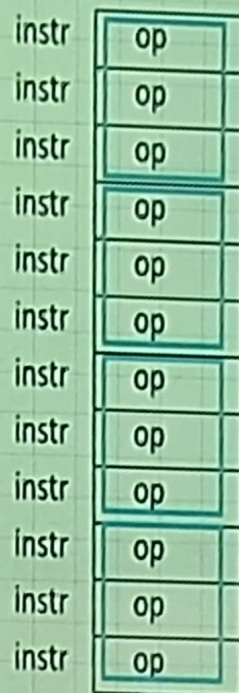


execute
1 instr/cycle



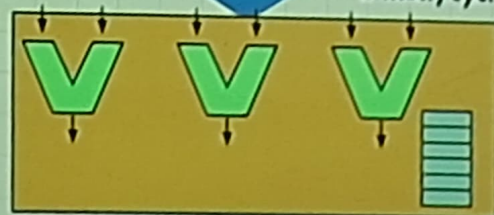
Change HW,

but can use
same code



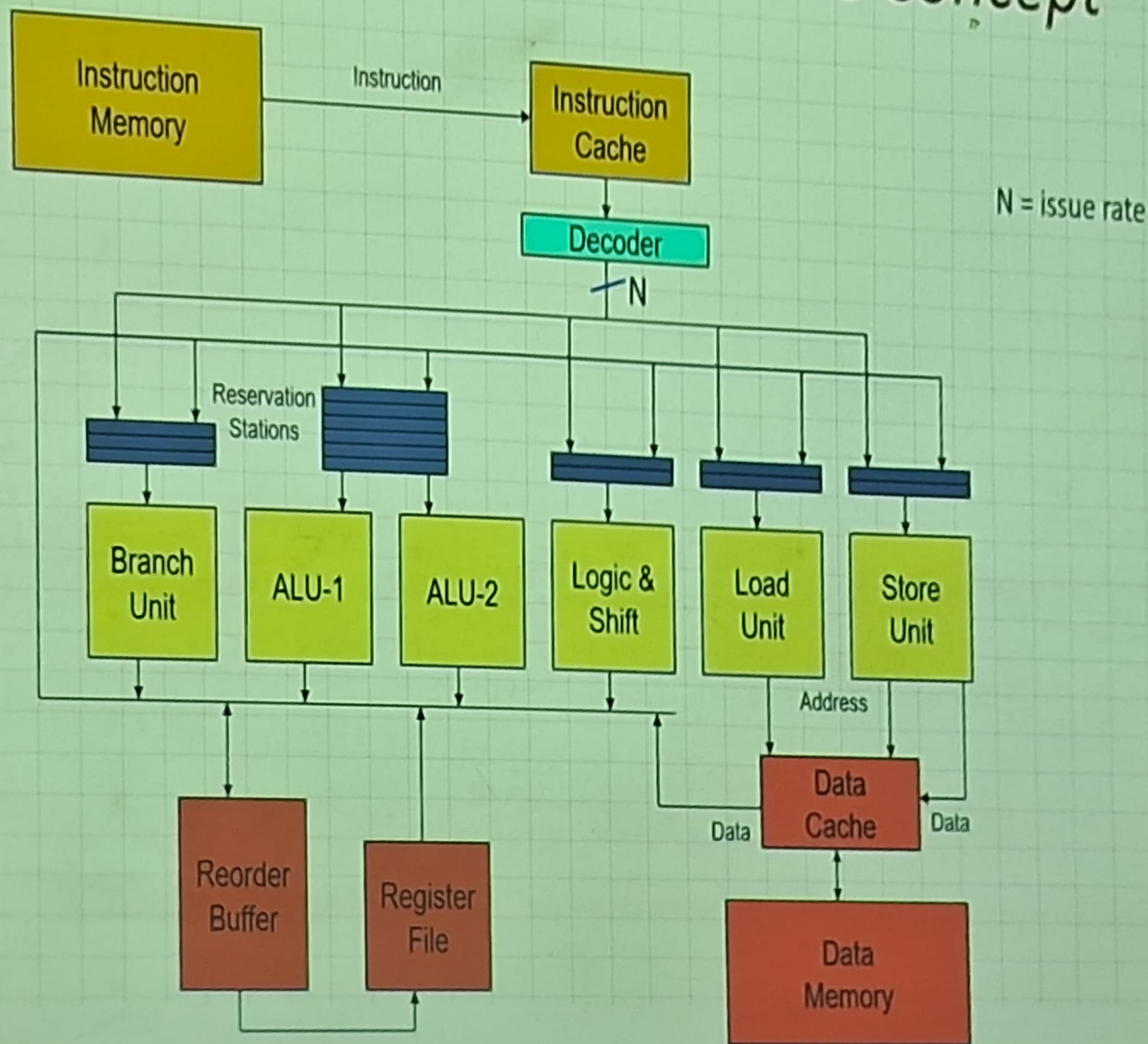
3-issue Superscalar

Issue and (try to) execute
3 instr/cycle





Superscalar: General Architecture Concept





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		?



Register Renaming Technique

- Eliminate name/false dependencies:
 - anti- (WaR) and output (WaW)) dependencies
- Can be implemented
 - by the **compiler**
 - advantage: low cost
 - disadvantage: “old” codes perform poorly
 - by **hardware**
 - advantage: binary compatibility
 - disadvantage: extra hardware needed



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

$c = a + b$
 $d = c + e$

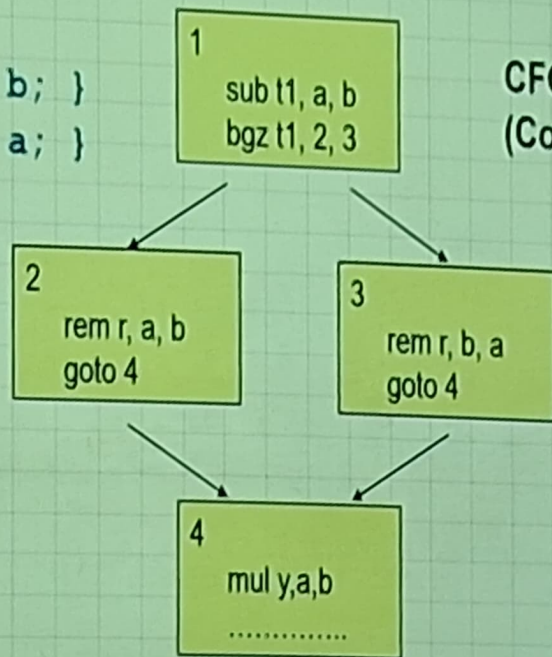
Notes:

1. data dependences can be **both** between **registers** and **memory** data operations
2. data dependencies are shown in the DDG: Data Dependence Graph

Control Dependences: CFG

C input code:

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





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



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

FP Loop: Where are the Hazards?

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

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

```
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
      BNEZ     R1,Loop ;branch R1!=zero
```

```
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
```

```
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
7      BNEZ     R1,Loop
```

FP Loop: Where are the Hazards?

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

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

```

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
      BNEZ   R1,Loop ;branch R1!=zero
    
```

```

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
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
    
```

```

1 Loop: L.D    F0,0(R1)
3      ADD.D  F4,F0,F2
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)

FP Loop: Where are the Hazards?

```
for (i=1000; i>0; i=i-1)
```

```
  x[i] = x[i] + s;
```

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

```
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
      BNEZ   R1,Loop ;branch R1!=zero
```

```
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
7      BNEZ   R1,Loop
```

```
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

```
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
```

```
1 Loop: L.D    F0,0(R1)
3      ADD.D  F4,F0,F2
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)



Register Renaming by HW

- Same example after renaming:

DIV.D R, F2, F4

ADD.D S, R, F8

S.D S, 0(R1)

SUB.D T, F10, F14

MUL.D U, F10, T

Original code:

DIV.D F0, F2, F4


ADD.D F6, F0, F8

S.D F6, 0(R1)

SUB.D F8, F10, F14

MUL.D F6, F10, F8

- Each destination gets a new (physical) register assigned
- Now only RaW hazards remain, which can be strictly ordered
- We will see several HW implementations of Register Renaming
 1. use ReOrder Buffer (ROB) & Reservation Stations, or
 2. use large register file with mapping table



Speculation (Hardware based)

- Execute instructions along predicted execution paths **but only commit the results if the prediction was correct**
- **Instruction commit:** *allowing an instruction to only **update** the register file when instruction is no longer speculative*
- **Need an additional piece of hardware to prevent any irrevocable action until an instruction commits:**
 - Reorder buffer, or Large renaming register file
 - why? think about it?

Speculative OoO

execution with
speculation
using RoB

