So, what does performance depend on ...

- **#instructions in the program**
  - Depends on the compiler
- **Frequency**
  - Depends on the transistor technology and the architecture
    - If we have more pipeline stages, then the time to traverse each stage reduces roughly proportionally
    - Given that each stage needs to be processed in one clock cycle, smaller the stage, higher the frequency
    - To increase the frequency, we simply need to increase the number of pipeline stages
- **IPC**
  - Depends on the architecture and the compiler
  - A large part of this book is devoted to this aspect.

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: $P \propto f^3$

   P → power
   f → frequency

2. Thermo-dynamics: $\Delta T \propto P$

   T → Temperature

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

What is ILP = Instruction level parallelism

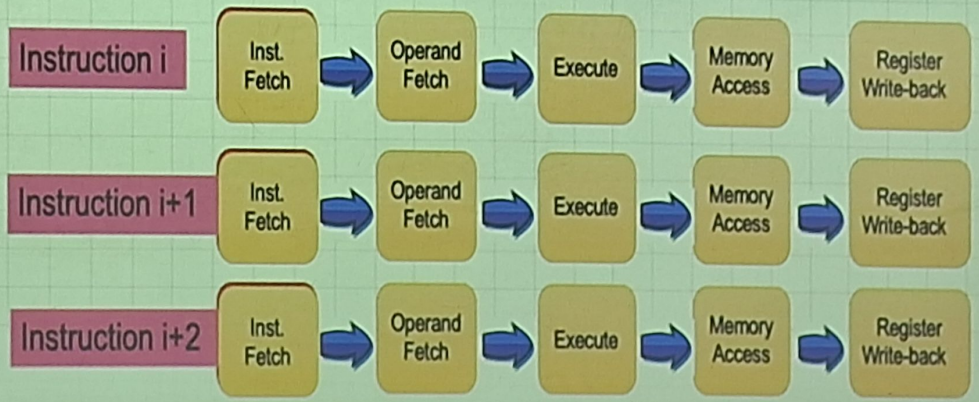
- *multiple operations (or instructions) can be executed in parallel, from a single instruction stream*
  - *so we are **not** yet talking about MIMD, multiple instruction streams*

Needed:

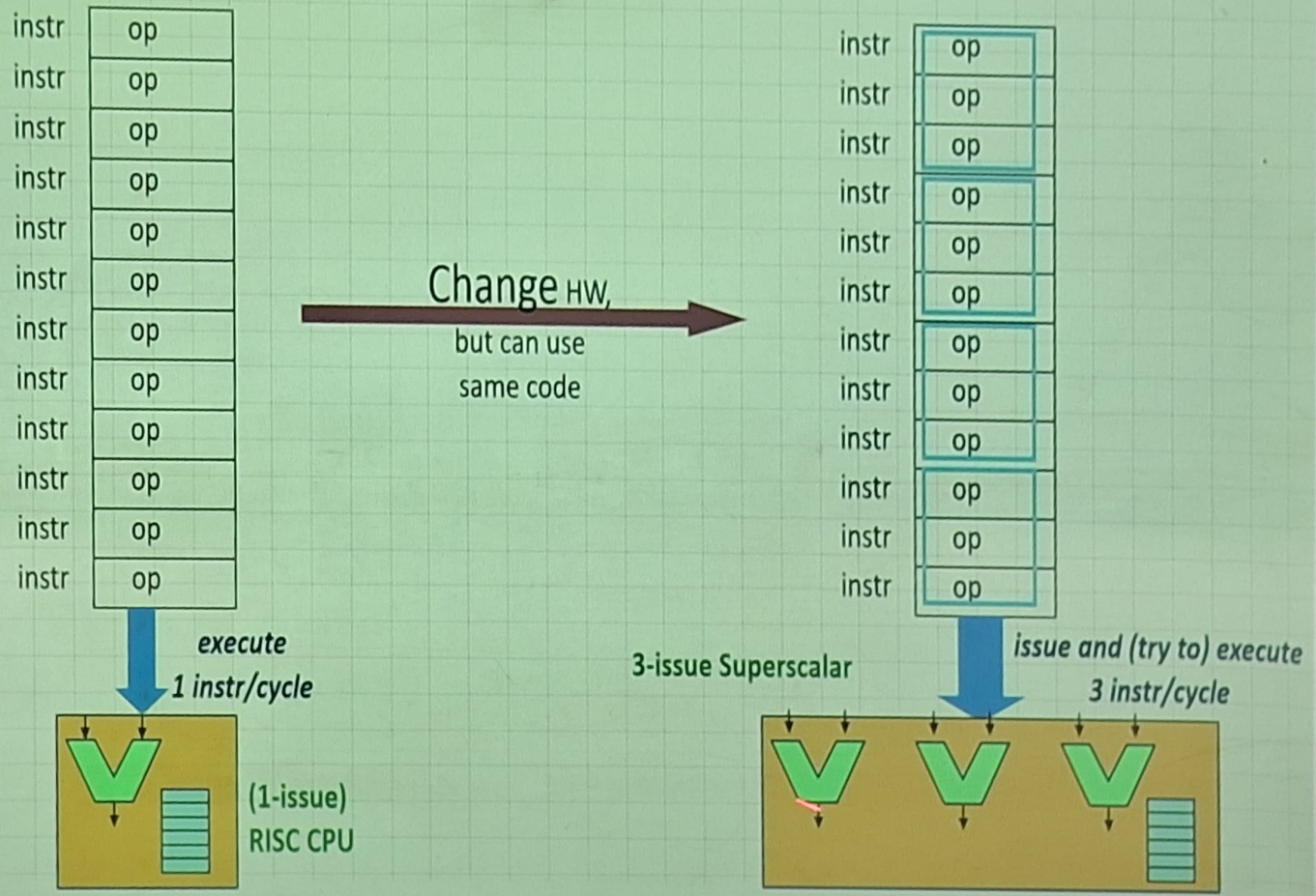
- Sufficient (HW) resources
- Parallel scheduling
  - Hardware solution
  - Software solution
- Application should contain sufficient ILP

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



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			
SUB.D	F8,F2,F6		IF	EX	EX			
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF				
MUL.D	F12,F2,F4							

stall because of data dep.

cannot be fetched because window full

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		

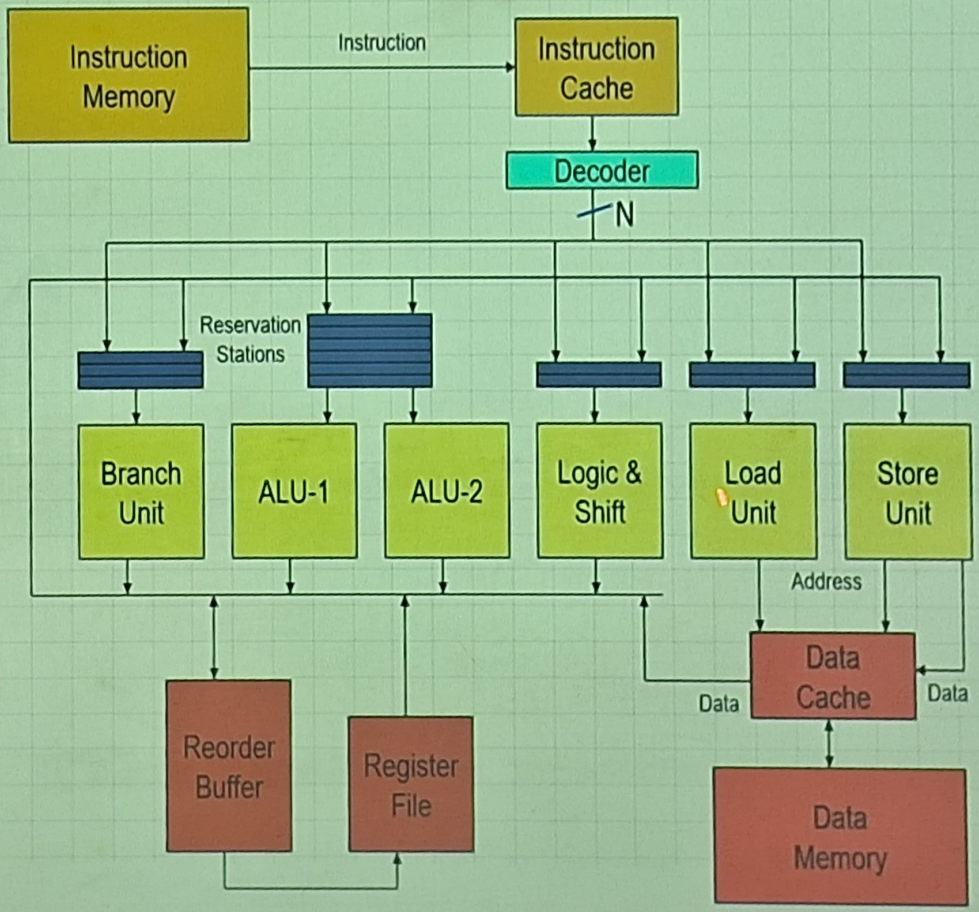
cannot execute
structural hazard

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

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
- Hazards cause scheduling problems and delay the pipeline

Impact of Hazards

- Hazards cause pipeline 'bubbles'

  Increase of CPI (and therefore execution time)

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

  where

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

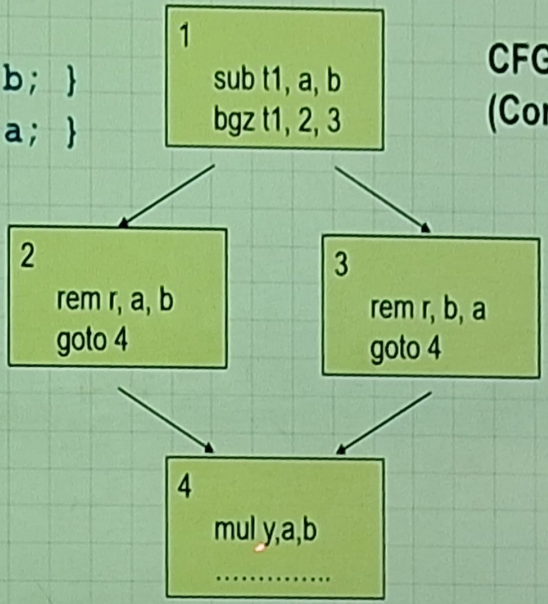
Data dependences

- **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):

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)
- **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
```

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

Dynamic Scheduling Principle

- What we examined so far is *static scheduling*
  - Compiler reorders instructions so as to avoid hazards and reduce stalls
- *Dynamic scheduling*:
  hardware rearranges instruction execution to reduce stalls
- Example:

```
    DIV.D   F0,F2,F4      ; takes 24 cycles and
RaW; real dependence      ; is not pipelined
    ADD.D   F10,F0,F8

    SUB.D   F12,F8,F14
```

This instruction cannot continue even though it does not depend on previous Div and Add

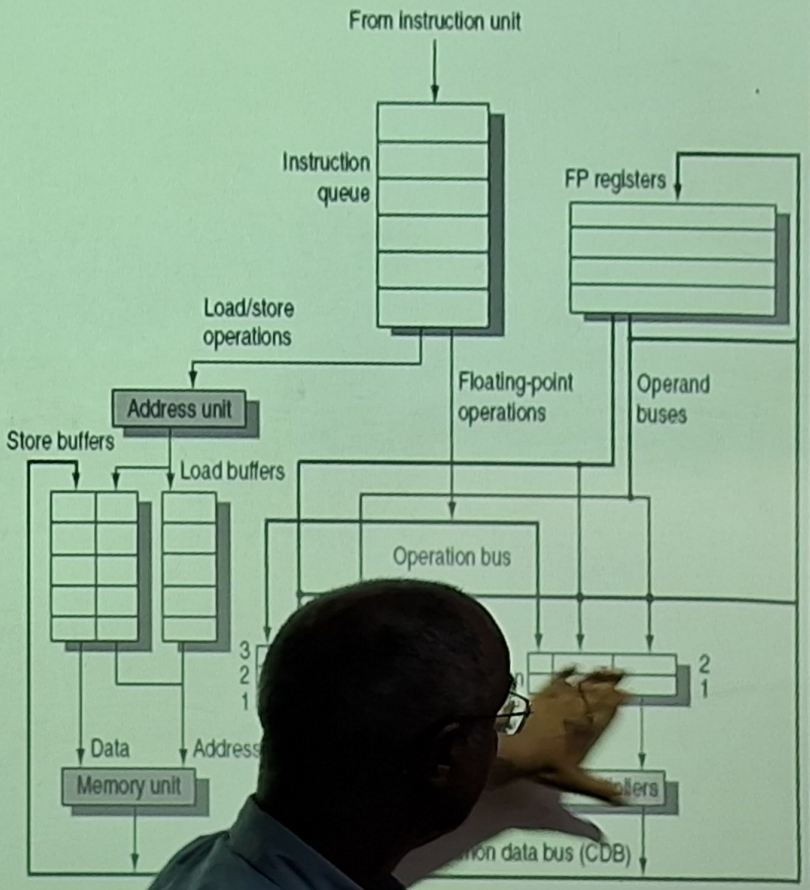
- **Key idea:** Allow instructions behind stall to proceed
- Book describes **Tomasulo** algorithm in detail, but we first describe general idea

Beginning of dynamic scheduling

- 1967 @ IBM
  - Robert Tomasulo
  - Dynamic scheduling of Floating Point operations on multiple execution units
  - Allowing Out-Of-Order execution
  - Register renaming in HW
    - Only a few architectural visible Floating Point registers were available
    - Renaming extends this number (although not architectural visible)
  - Reservation stations in front of Execution Units
  - CDB: Common Data Bus for all result broadcasts
- *see https://en.wikipedia.org/wiki/Tomasulo_algorithm*

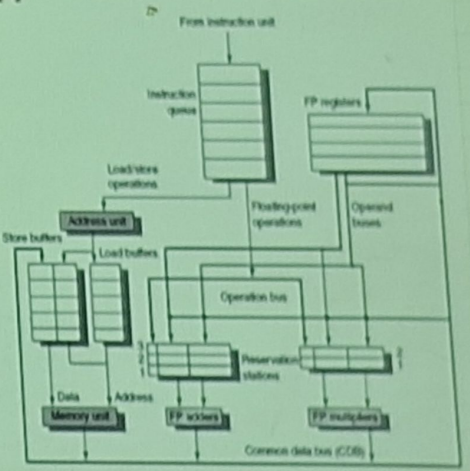
Tomasulo's Algorithm

- Top-level design:



- Note: Load and Store buffers contain data and addresses, act like reservation stations

Tomasulo's Algorithm



3 basic steps:

- Issue
  - Get next instruction from FIFO queue
  - If available RS (reservation station), issue instruction to the RS, with operand values if they are available
  - If operand values not available, stall the instruction
- Execute
  - When operand becomes available, store it in all reservation stations waiting for it
  - When all operands are ready, issue the instruction
  - Loads and stores are maintained in program order
  - No instruction allowed to initiate execution until all earlier branches (that precede it in program order) have completed

Speculation (Hardware based)

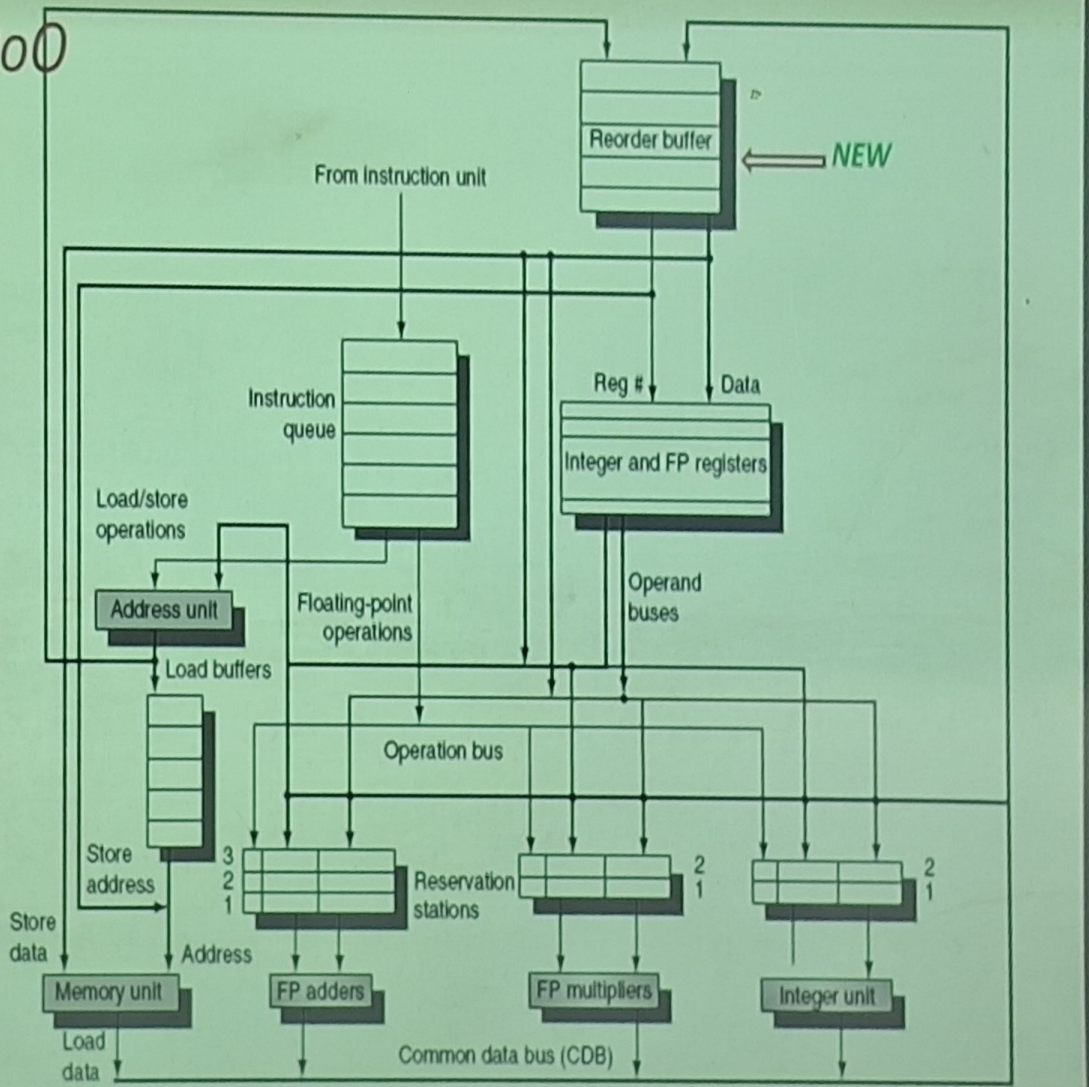
- Execute instructions along predicted execution paths but ***only commit the results if the prediction was correct***

- 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?

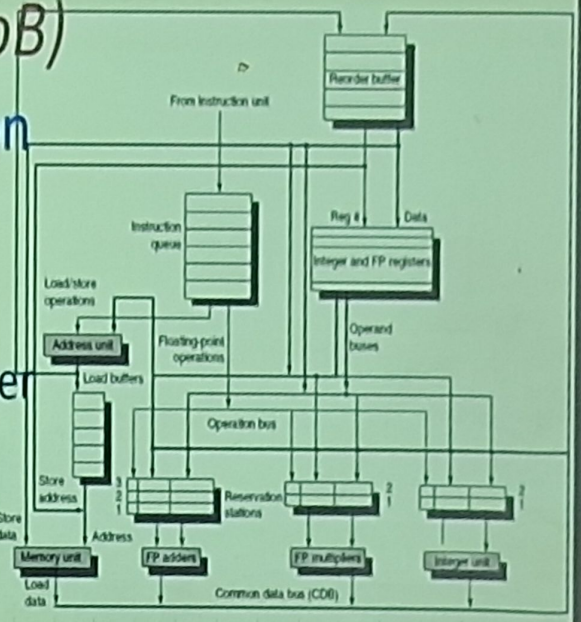
Speculative OoO

execution with speculation using RoB



Reorder Buffer (RoB)

- RoB – holds the result of instruction between completion and commit
  - Four fields per entry:
    1. Instruction type: branch/store/register
    2. Destination field: register number
    3. Value field: output value
    4. Ready field: completed execution?
       - is the data valid



- Modify reservation stations:
  - Operand source-id is now RoB (if its producer is still in the RoB) instead of functional unit (check yourself!)

Reorder Buffer (RoB)

- Register values and memory values are not written until an instruction commits

- RoB effectively renames the destination registers
  - every destination gets a new entry in the RoB
- On misprediction:
  - Speculated entries in RoB are cleared

- Exceptions:
  - Not recognized/taken until it is ready to commit
  - Precise exceptions require that 'later' entries in RoB are cleared

Flynn* Taxonomy, 1966

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)