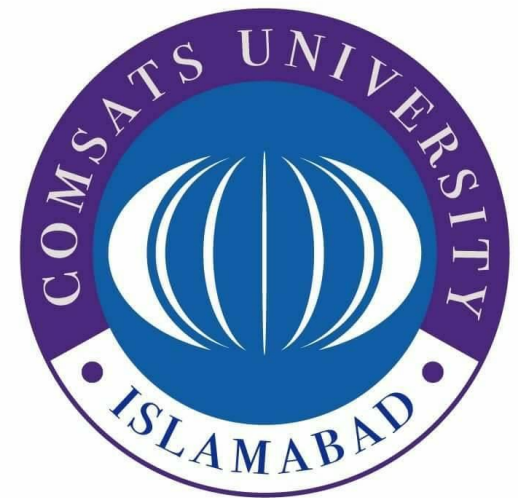




Advance Computer Architecture

Lecture 23,24,25

Control and Data Path for single and multicycle processor



Instruction Level Parallelism

Overlapping execution of instructions improves performance. This potential overlap among instructions is called instruction-level parallelism (ILP)

Introduction

- Pipelining become universal technique in 1985
 - Overlaps execution of instructions
 - Exploits “Instruction Level Parallelism (ILP)”
- Two main approaches:
 - Dynamic → hardware-based
 - Used in server and desktop processors
 - Not used as extensively in Parallel Multiprogrammed Microprocessors (PMP)
 - Static approaches → compiler-based
 - Not as successful outside of scientific applications

Review of basic concepts

- Pipelining → each instruction is split up into a sequence of steps – different steps can be executed concurrently by different circuitry.
- A basic pipeline in a RISC processor
 - IF – Instruction Fetch
 - ID – Instruction Decode
 - EX – Instruction Execution
 - MEM – Memory Access
 - WB – Register Write Back
- Two techniques:
 - Superscalar → A superscalar processor executes more than one instruction during a clock cycle.
 - VLIW → very long instruction word – compiler packs multiple independent operations into an instruction

Basic superscalar 5-stage pipeline

Superscalar → a processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor.

The hardware determines (statically/ dynamically) which one of a block on **n** instructions will be executed next.

	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1	IF	ID	EX	MEM	WB				
i+2		IF	ID	EX	MEM	WB			
i+3		IF	ID	EX	MEM	WB			
i+4			IF	ID	EX	MEM	WB		
i+5			IF	ID	EX	MEM	WB		
i+6				IF	ID	EX	MEM	WB	
i+7				IF	ID	EX	MEM	WB	
i+8					IF	ID	EX	MEM	WB
i+9					IF	ID	EX	MEM	WB

A single-core superscalar processor → SISD

A multi-core superscalar → MIMD.

Hazards → pipelining could lead to incorrect results.

- Data dependence “true dependence” → Read after Write hazard (RAW)

i: sub R1, R2, R3 % sub d,s,t → d = s-t

i+1: add R4, R1, R3 % add d,s t → d = s+t

Instruction (i+1) reads operand (R1) before instruction (i) writes it.

- Name dependence “anti dependence” → two instructions use the same register or memory location, but there is no data dependency between them.

- Write after Read hazard (WAR) → Example:

i: sub R4, R5, R3

i+1: add R5, R2, R3

i+2: mul R6, R5, R7

Instruction (i+1) writes operand (R5) before instruction (i) reads it.

- Write after Write (WAW) (Output dependence) → Example

i: sub R6, R5, R3

i+1: add R6, R2, R3

i+2: mul R1, R2, R7

Instruction (i+1) writes operand (R6) before instruction (i) writes it.

More about hazards

- Data hazards → RAW, WAR, WAW.
- Structural hazard → occurs when a part of the processor's hardware is needed by two or more instructions at the same time. Example: a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory. They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.
- Control hazard (branch hazard) → are due to branches. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the fetch stage).

Instruction-level parallelism (ILP)

- When exploiting instruction-level parallelism, goal is to maximize CPI
 - Pipeline CPI =
 - Ideal pipeline CPI +
 - Structural stalls +
 - Data hazard stalls +
 - Control stalls
- Parallelism with basic block is limited
 - Typical size of basic block = 3-6 instructions
 - Must optimize across branches

Data Hazard and Dependencies

There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences.

- Instruction i produces a result that may be used by instruction j.
- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction l

Data dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it is s causes a “stall.”
- Data dependence conveys:
 - Possibility of a hazard
 - Order in which results must be calculated
 - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect

Name dependence

- Two instructions use the same name but no flow of information
 - Not a true data dependence, *but is a problem when reordering instructions*
 - *Antidependence*: instruction j writes a register or memory location that instruction i reads
 - Initial ordering (i before j) must be preserved
 - *Output dependence*: instruction i and instruction j write the same register or memory location
 - Ordering must be preserved
- To resolve, use renaming techniques

Control dependence

- Every instruction is control dependent on some set of branches and, *in general*, the control dependencies must be preserved to ensure program correctness.
 - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
 - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

- Example

```
if C1 {  
    S1;  
};  
if C2 {  
    S2;  
};
```

S1 is control dependent on C1; S2 is control dependent on C2 but not on C1

Compiler techniques for exposing ILP

- Loop transformation technique to optimize a program's execution speed:
 1. reduce or eliminate instructions that control the loop, e.g., pointer arithmetic and "end of loop" tests on each iteration
 2. hide latencies, e.g., the delay in reading data from memory
 3. re-write loops as a repeated sequence of similar independent statements → space-time tradeoff
 4. reduce branch penalties;
- Methods
 1. pipeline scheduling
 2. loop unrolling
 3. strip mining
 4. branch prediction

Scheduling :

It refers to ordering the execution of instructions in a program to improve the performance.

• **Static Scheduling (Software based approach)** - In static Scheduling, the compiler optimised the code, gives a schedule and hardware just executes it in sequence without reordering the instructions ie. in-order execution. It does not lead to improvement in performance.

If one instruction stalls, succeeding instruction also get stalled.

Eg: LD R1,0(R2)

Add R3,R1,R4

Sub R5,R6,R7

Here there is a RAW dependency in I1 and I2 for R1, hence the second instruction I2 has to be stalled. As a consequence of this, the third instruction I3 which is an independent instruction will also be delayed since we are following in-order execution.

• **Dynamic Scheduling (Hardware based approach)** - In dynamic scheduling, the hardware rearranges the instruction execution to reduce the stalls while maintaining the data flow and exception behaviour.

All instructions pass through the issue stage in order, then they can be stalled or bypass each other depending on the availability of operands. Thus execution and completion happens out-of-order.

Eg: LD R1,0(R2)

Add R3,R1,R4

Sub R5,R6,R7



LD R1,0(R2)

Sub R5,R6,R7

Add R3,R1,R4

The hardware rearranges the code such that the Sub instruction can bypass Add instruction since it's an independent instruction ie. its operands are available to it. Hence it need not to wait for Add instruction.

Thus ready instructions can execute before earlier instructions that are stalled.

Out-of-order execution can lead to other 2 types of data hazard ie. WAR and WAW.

There are 2 schemes of implementing Dynamic Scheduling

- Scoreboarding
- Tomasulo's Algorithm

Dynamic scheduling with scoreboarding

- Scoreboarding is a technique for allowing instructions to execute out of order when there are sufficient resources structural hazard and no data dependence. Goal is to maintain an execution rate of 1 instruction per clock cycle
- WAR hazard does not exist in normal MIPS pipeline, but it can arise in dynamic scheduling where instructions execute out of order

DIV.D	F0, F2, F4
ADD.D	F10, F0, F8
SUB.D	F8, F8, F14

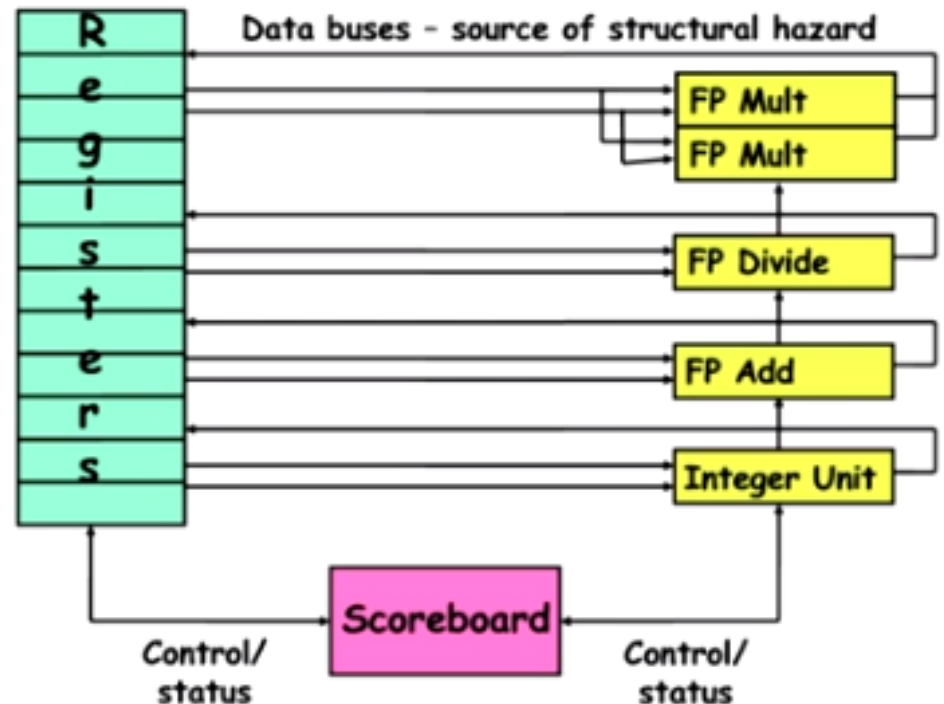
- Above example can cause WAR hazard if SUB.D executes before ADD.D clears RO stage
- A WAW hazard could also occur if F10 was destination of SUB.D and SUB.D was executed before ADD.D
- Scoreboard takes full responsibility of such hazards

A Scoreboard For MIPS :

Every instruction goes through a centralised hardware scoreboard. The scoreboard sends control signal to tell when an instruction can execute and when it can write results into destination register.

Out of order execution requires multiple instruction in the EX stage simultaneously which can be achieved with multiple functional units.

The instructions having a dependence are stalled but those having no dependence are allowed to continue.



3 Parts Of Scoreboard :

1. Instruction Status : In which of the 4 stages(Issue,RO,EX,WB) the instruction is.

2. Functional Unit Status :

- i. Busy :Whether the functional unit is busy or not.
- ii. Op : Operation to be performed.
- iii. F_i : Destination Register.
- iv. F_j, F_k : Source registers.
- v. Q_j, Q_k : Functional unit going to produce source registers F_j and F_k
- vi. R_j, R_k : Flags indicating whether F_j and F_k are ready or not.

3. Register Result Status : Indicates which functional unit will write the registers.

Scoreboard data structure

[illegible][illegible][illegible]

4 Stages of Scoreboard Control :

1.(ID Stage 1) Issue : Here instruction is decoded.If the functional unit is free(ie. no structural hazard),and no previously stalled instruction has the same destination register(ie. no WAW dependency),the scoreboard issues the instruction.

Issue of instruction always takes place in order . If there is structural or WAW hazard then the instruction issue stalls and no further instruction will be issued untill these hazards are cleared.

2.(ID Stage 2)Read Operand : A source operand is available or we can read a source operand if no earlier issued stalled instructions is going to write it(ie. the scoreboard resolves RAW hazard dynamically in this stage).When the source operand is available, the scoreboard tells the functional units to proceed to read operands from the registers and begin execution.The instructions may be sent to execution stage out of order.

3.Execution(EX) : Operate on operands.When result is ready control signal is sent to the scoreboard.

4.Write Back(WB) :Once the scoreboard is aware that the functional unit has completed its execution,it checks for WAR hazard.If none,it writes the result.If WAR hazard exist, then it stalls the instruction.

Eg : Div F0 F2 F4
Add F6 F2 F8
Sub F8 F10 F12

Let EX stage of Sub instruction is completed before Add instruction, but to resolve WAR hazard for F8, the sub instruction will be stalled untill Add instruction does not read the data from F8.

Scoreboard example

- Consider the inst sequence

L.D	F6, 34(R2)
L.D	F2, 45(R3)
MUL.D	F0, F2, F4
SUB.D	F8, F6, F2
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

Execution Stage Length: MUL.D=10 cc, DIV.D=40 cc, ADD.D/SUB.D=2 cc, L.D/S.D=1 cc

Scoreboard example

		Instruction Status				
Instruction		Issue	Read Operand	Execution Complete	Write Result	Comments
L.D	F6, 34(R2)	1				
L.D	F2, 45(R3)					
MUL.D	F0, F2, F4					
SUB.D	F8, F6, F2					
DIV.D	F10, F0, F6					
ADD.D	F6, F8, F2					

[illegible][illegible]

Scoreboard example

Instruction Status					
Instruction	Issue	Read Operand	Execution Complete	Write Result	Comments
L.D F6, 34(R2)	1	2			
L.D F2, 45(R3)	2				Can't Issue, FU busy
MUL.D F0, F2, F4					
SUB.D F8, F6, F2					
DIV.D F10, F0, F6					
ADD.D F6, F8, F2					

[illegible][illegible]

Scoreboard summary

- Tries to maintain one instruction/cycle if no structural hazard
- Scoreboard is responsible for all hazard detection
 - IS stage: Struct/WAW hazard
 - RO stage: RAW hazard
 - WR stage: WAR hazard
- Scoreboard can't eliminate WAW/WAR hazards
- Can't reduce RAW hazards either (no data forwarding)
- FUs in scoreboard enabled processor are not pipelined

Tomasulo's Approach

- Dynamic register renaming
- Register renaming is provided using reservation stations
- Read and keep a copy of available operands at IS stage
 - Avoids WAR, WAW hazards
 - Values are stored in reservation stations

Tomasulo's approach

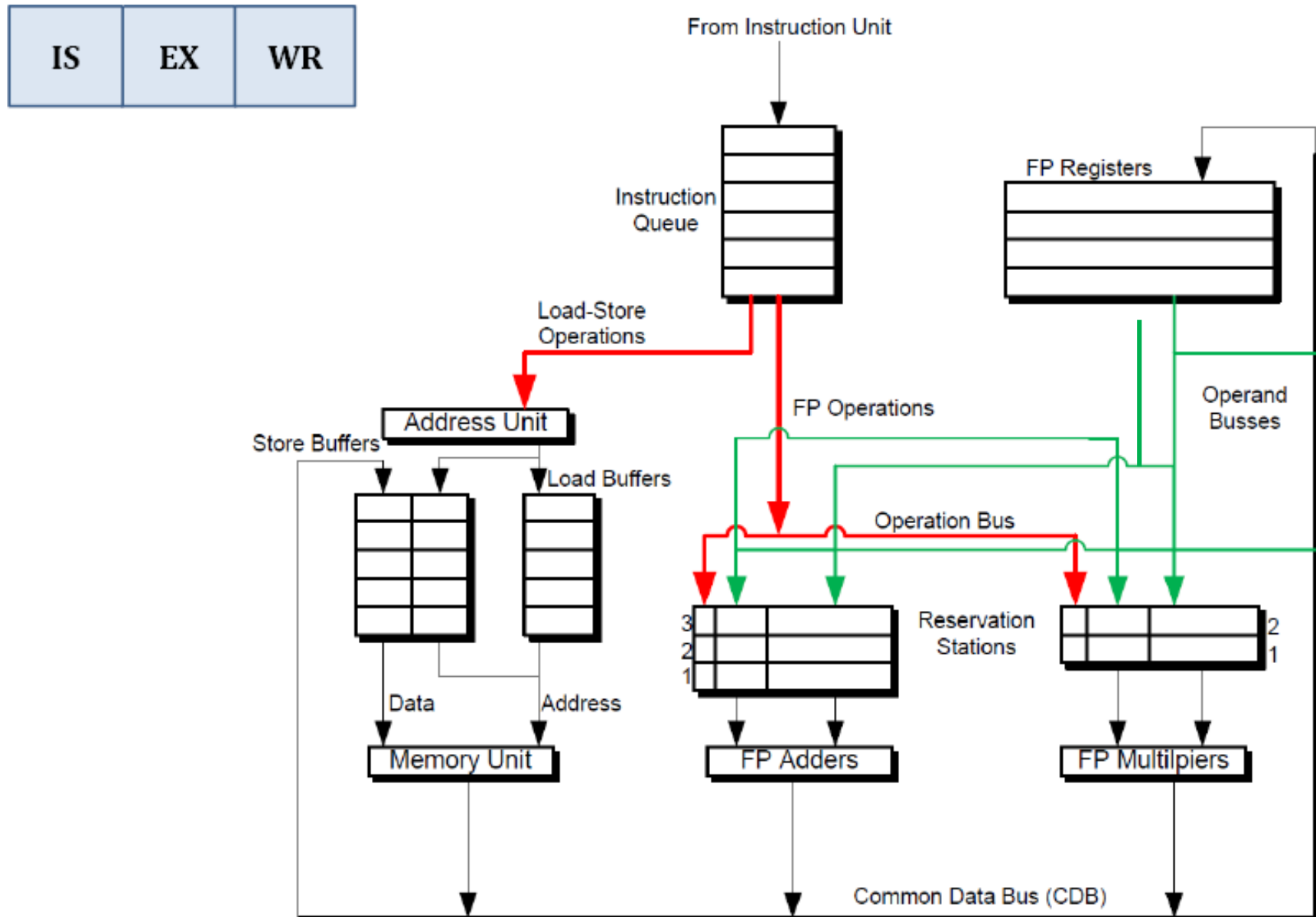
- Used in IBM in 60's
- Tracks when operands are available
 - Satisfies data dependence (RAW)
- Removes name dependence
 - Uses register renaming
- Very similar to what is used today
 - Almost all high performance processors use a derivative of Tomasulo's hardware
 - Much of the terminology survives today

Tomasulo's Algorithm Vs Scoreboarding

- **It shares many ideas with the scoreboarding scheme**
- **It combines the key elements of the scoreboarding scheme with the introduction of register renaming**
- **Register renaming is achieved with the help of reservation stations, which buffer the operands of instructions waiting to issue, and by the issue logic**
- **Hazard detection and execution control functionality are distributed**
- **Results are directly passed to functional units from the reservation stations, rather than going through the registers**
- **A common result bus, known as common data bus (CDB), allows all units waiting for an operand to be loaded simultaneously**



MIPS floating point unit using Tomasulo's algo



Dynamic Scheduling - Tomasulo

❖ Issue

- ❖ Get next instruction from FIFO queue
- ❖ If RS available, issue the instruction to the RS with operand values if available
- ❖ If operand values not available, stall the instruction

Dynamic Scheduling - Tomasulo

❖ Execute

- ❖ When operand becomes available, store it in any reservation stations waiting for it
- ❖ When all operands are ready, execute the instruction
- ❖ Loads and store uses buffers
- ❖ No instruction will initiate execution until all branches that precede it in program order have completed

Dynamic Scheduling - Tomasulo

❖ Write result

- ❖ Write result into CDB (there by it reaches the reservation station, store buffer and registers file) with name of execution unit that generated the result.
- ❖ Stores must wait until address and value are received

Detailed example

Assume

R2 is 100

R3 is 200

F4 is 2.5

Load: 2 cycles
Add: 2 cycles
Mult: 10 cycles
Divide: 40 cycles

Reservation Stations

		Is	Ex	W
1. L.D	F6, 34(R2)	1	2	4
2. L.D	F2, 45(R3)	2	3	5
3. MUL.D	F0, F2, F4	3	6	16
4. SUB.D	F8, F2, F6	4	6	8
5. DIV.D	F10, F0, F6	5	17	57
6. ADD.D	F6, F8, F2	6	9	11

	Busy	Op	Vj	Vk	Qj	Qk	A
LD1	0						
LD2	0						
AD1	0						
AD2	0						
AD3							
ML1	0						
ML2	0						

F0 F2 F4 F6 F8 F10 F12

Cycle:

57

Register Status:

							...
--	--	--	--	--	--	--	-----

Loop based example

Loop:	LD	F0	0	R1
	MUL.D	F4	F0	F2
	SD	F4	0	R1
	SUBI	R1	R1	#8
	BNEZ	R1	Loop	

Assume
Load: 2 cycles
Mul: 4 cycles
But first load takes
8 cycles in EX
(data cache miss)

Dispatching SUBI Instruction

Reservation Stations

Iter	Inst	Is	Ex	W
1.	L.D F0, 0(R1)	1	9	10
1.	MUL.D F4, F0, F2	2	14	15
1.	SD F4, 0(R1)	3	16	17
2.	L.D F0, 0(R1)	6	10	11
2.	MUL.D F4, F0, F2	7	15	16
2.	SD F4, 0(R1)	8	17	18

Cycle: 18

R1: 64

Register Status:

	Busy	Op	Vj	Vk	Qj	Qk	A
LD1	0						
LD2	0						
LD3	1	LD					64
SD1	0						
SD2	0						
SD3	1	SD			ML1		64
AD1	0						
AD2	0						
AD3	0						
ML1	1	MUL.D		R(F2)	LD3		
ML2	0						

F0	F2	F4	F6	F8	F10	F12	
LD3		ML1					...

Home Work

- Complete the remaining given example by calculation its cycles using remaining iterations.

Speculation

- Dynamic scheduling with prediction
 - Fetch and issue predicted instructions and stall execution until branch outcome
- Speculation
 - Fetch, issue and **execute** instructions as if the prediction is always correct
- Hardware based speculation combines three key ideas
 - Dynamic branch prediction to choose which instruction to execute
 - Dynamic scheduling to fetch and decode the instruction
 - Speculation Go with the execution
 - With off course the ability to undo the effects if prediction is wrong

Hardware-based speculation

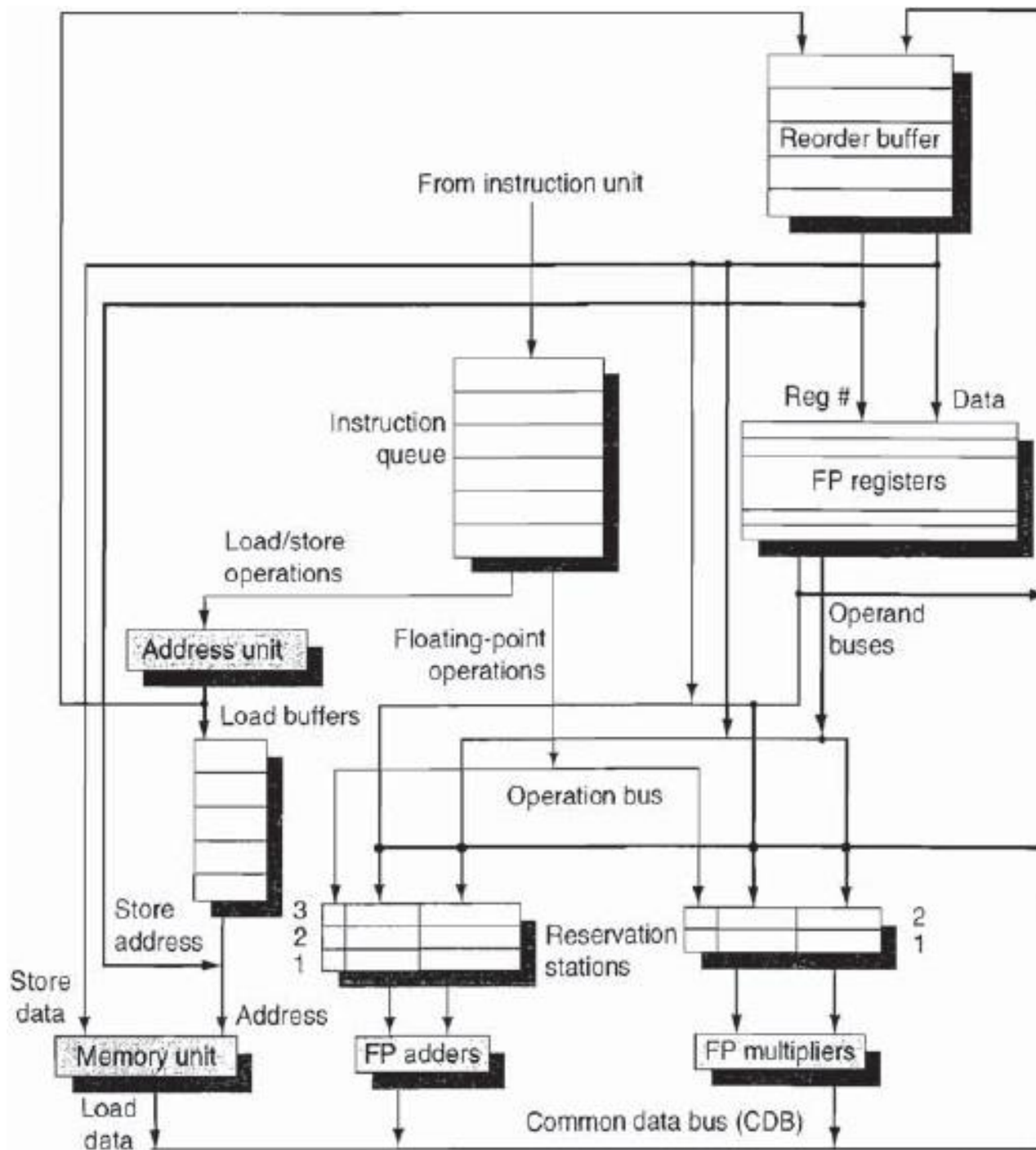
- Separate data forwarding and instruction completion
- Use forwarded value on CDB for speculative execution
- Don't perform updates that can not be undone i.e. don't write register files or memory
- Add **instruction commit** stage to update registers and memory when it no longer remains speculative
- Key idea: allow O-O-O execution but in-order commit

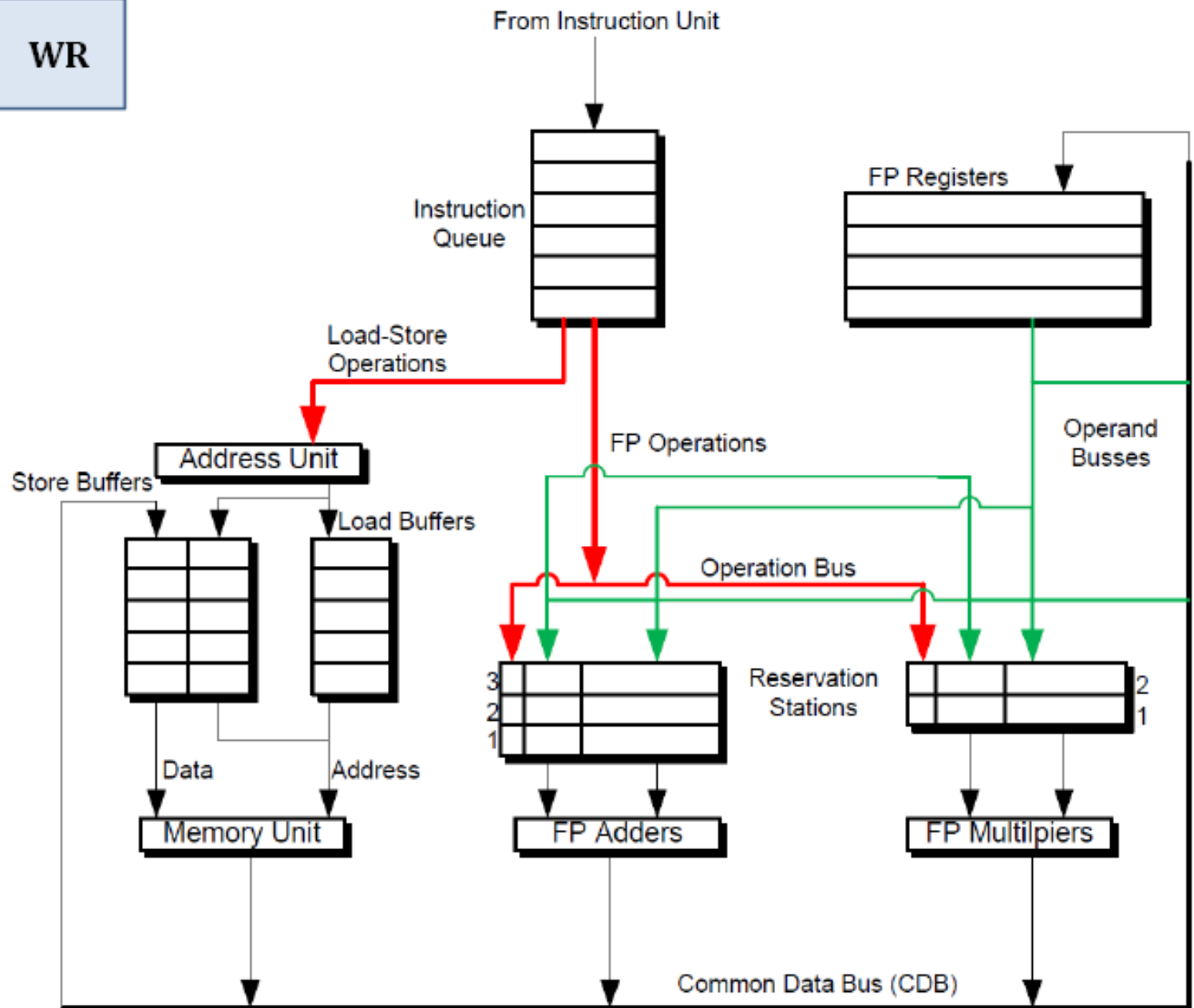
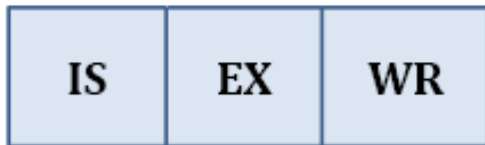
Significance of adding commit phase

- Prevent any irreversible action
 - Because neither register nor memory values are actually written until instruction commits, processor can easily undo its speculative actions when a branch is mispredicted
- Exceptions may become precise again
 - Exception are handled by not recognizing the exception until the instruction is ready to commit
 - If speculated instruction raises an exception and it is after the mispredicted branch, then it is flushed automatically
- Keeps the sequential illusion to outside world

Changes to Tomasulo's hardware

- Adding the commit phase requires new hardware
 - It is called Reorder Buffer (ROB)
 - It holds the result of instructions that have finished execution but not committed
- ROB contains four fields
 - **Instruction type** (register operation, branch, store)
 - **Destination** (register number for loads and ALU operations, memory address for store)
 - **Value** (value of instruction result)
 - **Ready** (indicates that instruction has executed and value is ready)
- Register renaming is performed through ROB's and not reservation stations





Four steps of instruction execution

- Issue
 - Read the instruction from instruction queue
 - Issue instruction if
 - Empty slot in ROB
 - Empty slot in reservation station
 - Stall issue otherwise
 - Number of ROB entry allocated also sent to reservation station
 - Read operands from ROB or register files

Four steps of instruction execution

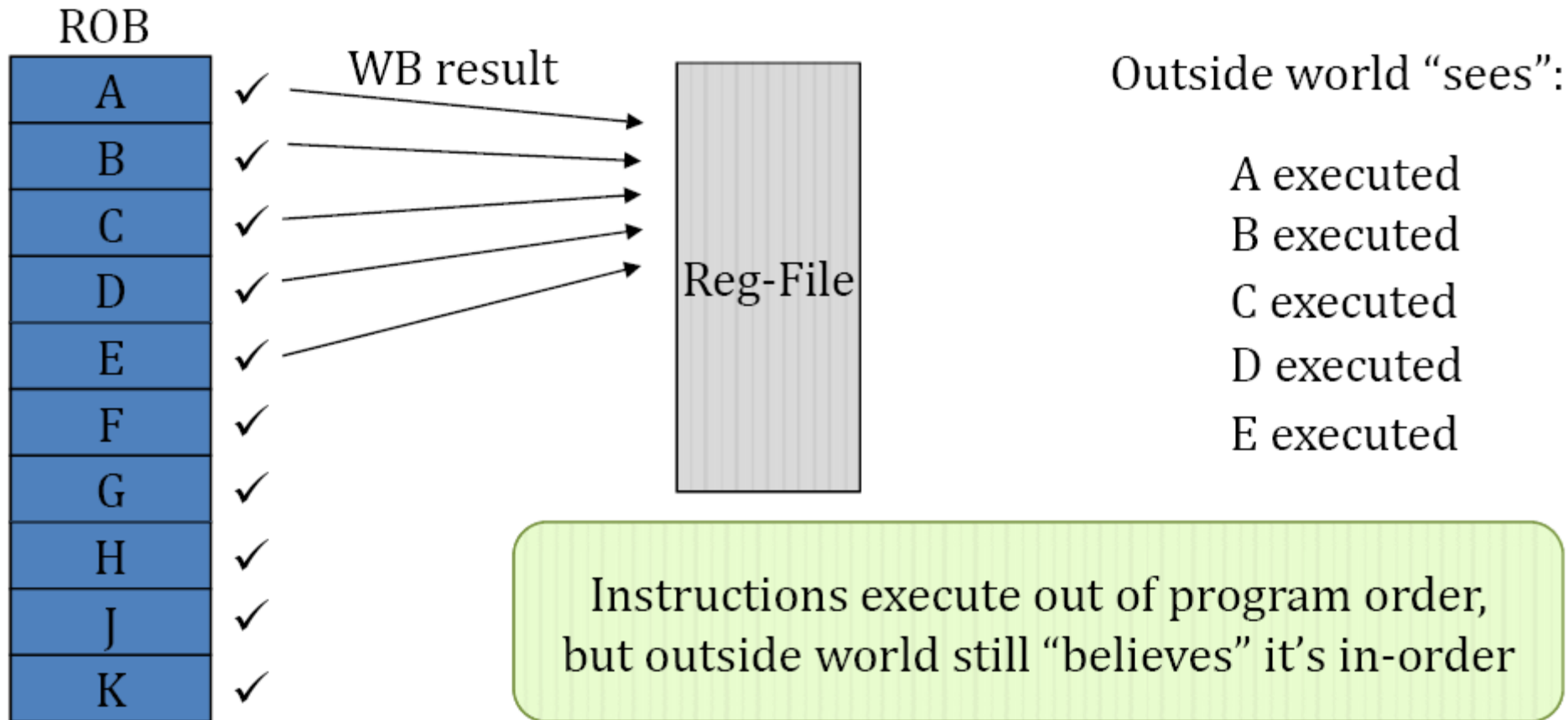
- Execute
 - Wait for the operands.
 - If one or more operands not available, monitor the CDB
 - Checks for RAW hazards
- Write result
 - Broadcast result on CDB
 - Any dependent instruction in RS will grab the value
 - Write result to ROB (not register file)
 - The register file holds official register state
 - We will update it in program order
 - Mark ready bit in ROB

Four steps of instruction execution

- Commit
 - Wait until an instruction is oldest in ROB
 - i.e it is at the head of ROB
 - Write result (if ready bit is set)
 - If register operation: write to register file
 - If store write to memory
 - Advance ROB head to next instruction
 - This is what outside world sees
 - Every thing is in order

Commit illustrated

- Commit changes the processor state



2-Loop unrolling

- Scheme to reduce loop overhead
 - Increases no of useful instructions relative to overhead instruction
- Replicates loop body multiple times and adjusts loop termination code
- Improves scheduling by reducing branches

Loop unrolling: Example

Loop:	L.D	F0, 0(R1)
	ADD.D	F4, F0, F2
	S.D	F4, 0(R1)
	L.D	F6, -8(R1)
	ADD.D	F8, F6, F2
	S.D	F8, -8(R1)
	L.D	F10, -16(R1)
	ADD.D	F12, F10, F2
	S.D	F12, -16(R1)
	L.D	F14, -24(R1)
	ADD.D	F16, F14, F2
	S.D	F16, -24(R1)
	DADDUI	R1, R1, #-32
	BNE	R1, R2, Loop

27 cc, 4 insts => 27/4 = 6.75 cc/ins









Loop:	L.D	F0, 0(R1)
	stall	
	ADD.D	F4, F0, F2
	stall	
	stall	
	S.D	F4, 0(R1)
	L.D	F6, -8(R1)
	stall	
	ADD.D	F8, F6, F2
	stall	
	stall	
	S.D	F8, -8(R1)
	L.D	F10, -16(R1)
	stall	
	ADD.D	F12, F10, F2
	stall	
	stall	
	S.D	F12, -16(R1)
	L.D	F14, -24(R1)
	stall	
	ADD.D	F16, F14, F2
	stall	
	stall	
	S.D	F16, -24(R1)
	DADDUI	R1, R1, #-32
	stall	
	BNE	R1, R2, Loop

Static scheduling + loop unrolling

Loop:	L.D	F0, 0(R1)
	L.D	F6, -8(R1)
	L.D	F10, -16(R1)
	L.D	F14, -24(R1)
	ADD.D	F4, F0, F2
	ADD.D	F8, F6, F2
	ADD.D	F12, F10, F2
	ADD.D	F16, F14, F2
	S.D	F4, 0(R1)
	S.D	F8, -8(R1)
	DADDUI	R1, R1, #-32
	S.D	F12, 16(R1)
	S.D	F16, 8(R1)
	BNE	R1, R2, Loop

14 cc, 4 insts => $14/4 = 3.5$ cc/ins

Summary

No. of cc/iteration	Static Scheduling	Loop Unrolling
9		
7		
6.75		
3.5		

- Loop unrolling
 - Advantage
 - Increased performance due to overhead reduction
 - Disadvantage
 - Increased code size
 - Register pressure

How many times loop can be unrolled

- Minimum
 - Depends on the number of stalls in the original code
 - Try to remove as many stalls as possible (target: zero stalls)
- Maximum
 - Depends on available number of registers
 - Only even FP registers are used
 - Same register can't be used again while unrolling

Minimum unrolling

Unscheduled implementation = 9cc

Loop:	L.D	F0, 0(R1)
	stall	
	ADD.D	F4, F0, F2
	stall	
	stall	
	S.D	F4, 0(R1)
	DADDUI	R1, R1, #-8
	stall	
	BNE	R1, R2, Loop

Loop:	L.D	F0, 0(R1)
	L.D	F6, -8(R1)
	ADD.D	F4, F0, F2
	ADD.D	F8, F6, F2
	DADDUI	R1, R1, #-16
	S.D	F4, 16(R1)
	S.D	F8, 8(R1)
	BNE	R1, R2, Loop

8 cc, 2 insts => $8/2 = 4$ cc/ins

Maximum unrolling

Unscheduled implementation = 9cc

```

Loop:  L.D          F0, 0(R1)
        stall
        ADD.D       F4, F0, F2
        stall
        stall
        S.D         F4, 0(R1)
        DADDUI      R1, R1, #-8
        stall
        BNE         R1, R2, Loop
    
```

- We suppose that we have 26 FP registers (F0-F25) and only even numbered registers can be used
- 20 cc, 6 insts => $20/6 = 3.33$ cc/ins

```

Loop:  L.D          F0, 0(R1)
        L.D          F6, -8(R1)
        L.D          F10, -16(R1)
        L.D          F14, -24(R1)
        L.D          F18, -32(R1)
        L.D          F22, -40(R1)
        ADD.D       F4, F0, F2
        ADD.D       F8, F6, F2
        ADD.D       F12, F10, F2
        ADD.D       F16, F14, F2
        ADD.D       F20, F18, F2
        ADD.D       F24, F22, F2
        S.D         F4, 0(R1)
        S.D         F8, -8(R1)
        DADDUI      R1, R1, #-48
        S.D         F12, 32(R1)
        S.D         F16, 24(R1)
        S.D         F20, 16(R1)
        S.D         F24, 8(R1)
        BNE         R1, R2, Loop
    
```

Unrolled Loop Detail

- In real programs we do not usually know the upper bound on the loop.
- Suppose it is n , and we would like to unroll the loop to make k copies of the body.
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
 - The first executes $(n \bmod k)$ times and has a body that is the original loop.
 - The second is the unrolled body surrounded by an outer loop that iterates (n/k) times.
- We shall see in Chapter 4, this technique is similar to a technique called strip mining, used in compilers for vector processors.
- For large values of n , most of the execution time will be spent in the unrolled loop body.

How to do static scheduling and loop unrolling

- Checklist
 - How many stalls are there in original unscheduled code
 - Are there independent instructions to cover the stalls by reordering
 - If yes then reorder
 - How many stalls are left?
 - How many to unroll to cover remaining stalls
 - What would be the value of final loop counter after unrolling
 - Whether counter was incrementing or decrementing
 - How to adjust displacement values

Loop unrolling example 2

Instruction Producing Result	Instruction Using Result	Latency in Clock Cycles
FP MUL	Another FP ALU op	6
FP ADD	Another FP ALU op or Store Double	4
Load Double	FP ALU op	1
Load Double	Store Double	0

```
For (i=0; i<1000; i = i + 1 )  
    x[i] = x[i] + s
```

- How will the loop be unrolled??

3. Strip mining

- The block size of the outer block loops is determined by some characteristic of the target machine, such as the vector register length or the cache memory size.
 - Number of iterations = n
 - Goal: make k copies of the loop body
 - Generate pair of loops:
 - First executes $n \bmod k$ times
 - Second executes n / k times
 - “Strip mining”

4-Branch prediction

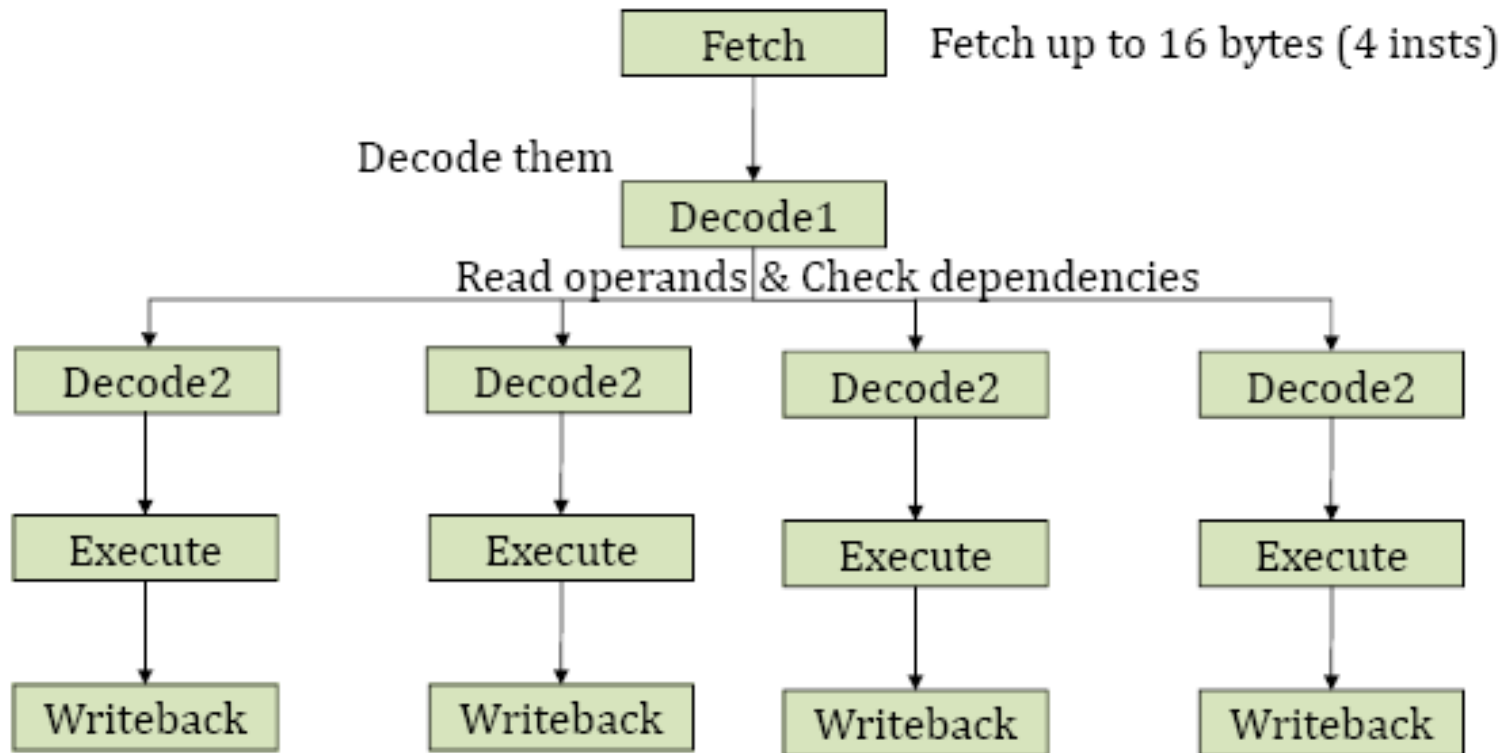
- Branches are very frequent
 - Approximately 15-20% of all the instructions
- Can not wait until we know where it goes
 - Long pipelines
 - Higher penalties
- Inaccurate branch prediction mechanisms can make a fast processor look like a very mediocre processor

Branch prediction

- Predict branches
 - And predict them well
 - Misprediction penalties are high
 - 17 wasted cycles in P4
 - What is the point of pipelining if we have to perform fill & drain all the time
- Fetch and decode on predicted path
 1. Don't execute until branch is resolved (this is what is used in Tomasulo's approach)
 2. Execute anyway
- Recover from mispredictions
 - Restart fetching on the corrected path

Problem with simple branch prediction

- More ILP leads to more burden due to control dependence
- Branch prediction reduces stalls but not enough
- Example: Assume we can execute 4 instructions in parallel



Multiple issue processors

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

VLIW Processors

- Package multiple operations into one instruction
- Must be enough parallelism in code to fill the available slots,
- Disadvantages:
 - Statically finding parallelism
 - Code size
 - No hazard detection hardware
 - Binary code compatibility