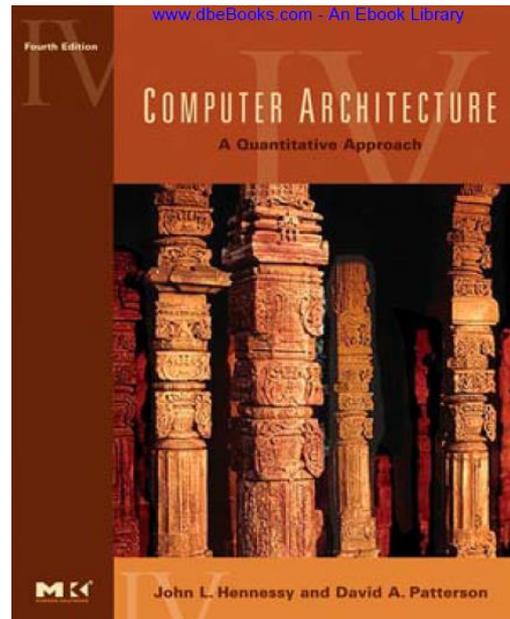




Instruction-Level Parallelism

dynamic scheduling

prepared and instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University





Dynamic Scheduling

Dynamic Scheduling rearranges instruction execution to reduce the stalls while maintaining data flow and exception behavior.

- 😊 Enables handling some cases when dependences are unknown at compile time (e.g. memory reference).
- 😊 Simplifies the compiler.
- 😊 Allows the processor to tolerate cache misses delays by executing other code while waiting for miss resolution.
- 😊 Allows code compiled for one pipeline to run efficiently on a different pipeline.
- 😞 Increases significantly the hardware complexity.



In ordinary pipeline instructions are **in-line** issued and executed.

- If an instruction is stalled in the pipeline, no later instructions can proceed.
- If instruction ***j*** depends on instruction ***i***, all instructions after ***j*** must be stalled until ***i*** is finished and ***j*** can execute.

DIV.D	F0, F2, F4
ADD.D	F10, F0, F8
SUB.D	F12, F8, F14

SUB.D cannot execute because ADD.D dependence on DIV.D causes stall, but is independent of the present pipeline.

To execute SUB.D we separate instruction issue (at ID) into **two parts**: **checking** for **structural hazards** and **waiting** for the absence of a **data hazard**.



Out-Of-Order Execution

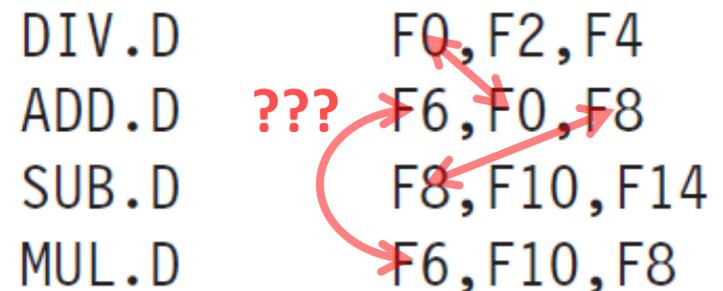
Instructions are still **in-order** issued, but start execution as soon as their data operands are available.

Such a pipeline does **out-of-order** (**OOO**) execution, implying out-of-order completion.

OOO introduces possibilities of **WAR** and **WAW** hazards, not existing in in-order pipeline.

ADD.D and SUB.D are **anti dependent**.

Executing SUB.D before ADD.D (waits for F0) violates the anti dependence, causing a **WAR** hazard.





Likewise, to avoid violating output dependences of F6 by MUL.D, WAW hazards must be handled.

register renaming avoids these hazards.

OOO completion must preserve **exception behavior** to happen **exactly** as by in-order.

- No instruction generates an exception until the processor knows that the instruction raising the exception will be executed.

OOO splits the ID stage into two stages:

1. **Issue**—Decode instructions, check for **structural hazards**.
2. **Read operands**—Wait until no **data hazards**, then read operands.



An IF stage preceding **issue** stage fetches either into an instruction register or a pending instructions queue. Instructions are issued from these.

The EX stage follows the **read operands** stage and may take multiple cycles, depending on the operation.

The pipeline allows simultaneous execution of multiple instructions.

- Without it a major advantage of OOO is lost.
- Requires multiple functional units.

Instructions are issued in-order, but can enter execution out of order. There are two OOO techniques: **scoreboarding** and **Tomasulo's algorithm**.



Tomasulo's Dynamic Scheduling

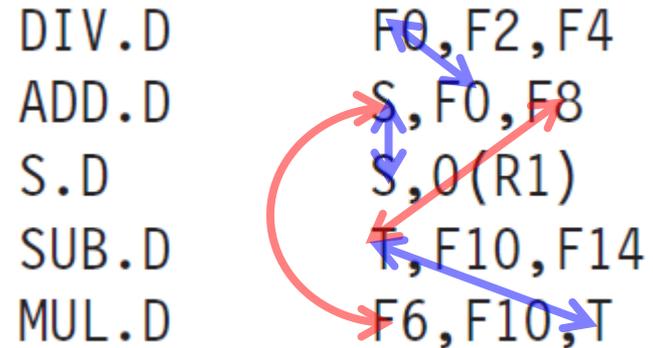
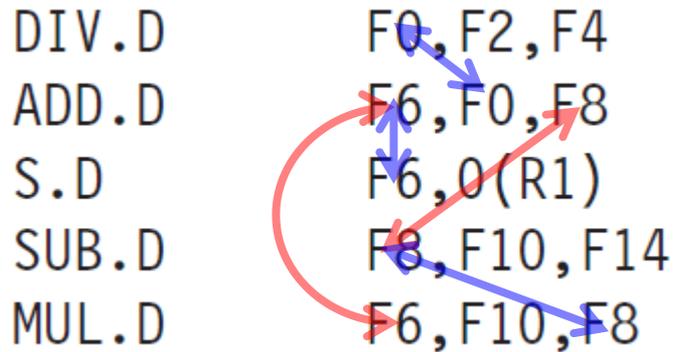
Invented for IBM 360/91 FPU by Robert Tomasulo.

- Minimizes **RAW** hazards by tracking when operands are available.
- Minimizes **WAR** and **WAW** hazards by register renaming.

We assume the existence of **FPU** and **load-store** unit, and use MIPS ISA.

Register renaming eliminates **WAR** and **WAW** hazards.

- Rename all destination registers, including those with pending read and write for earlier instructions.
- OOO writes do not affect instructions depending on earlier value of an operand.



This code includes potential **WAW** and **WAR** hazards. Anti dependence, **WAR** hazard, and **WAW** hazard if MUL.D finishes before ADD.D. It is called **name dependence**.

True data dependencies.

Name dependencies can be eliminated by **register renaming**. Any subsequent usage of F8 must be replaced by T. Very difficult for the compiler (branches may intervene).



Tomasulo's algorithm can handle renaming across branches.

Register renaming is provided by **Reservation Station (RS)**, buffering the operands of instructions waiting to issue.

RS fetches and buffers an operand as soon as it is available, eliminating the need to get it from the **Register File (RF)**.

Pending instruction designate the RS that will provide their operands. At **issue**, pending operands are renamed from RF specifier to RS.

When successive writes to RF (WAW) overlap in execution, only the last one actually updates the RF.



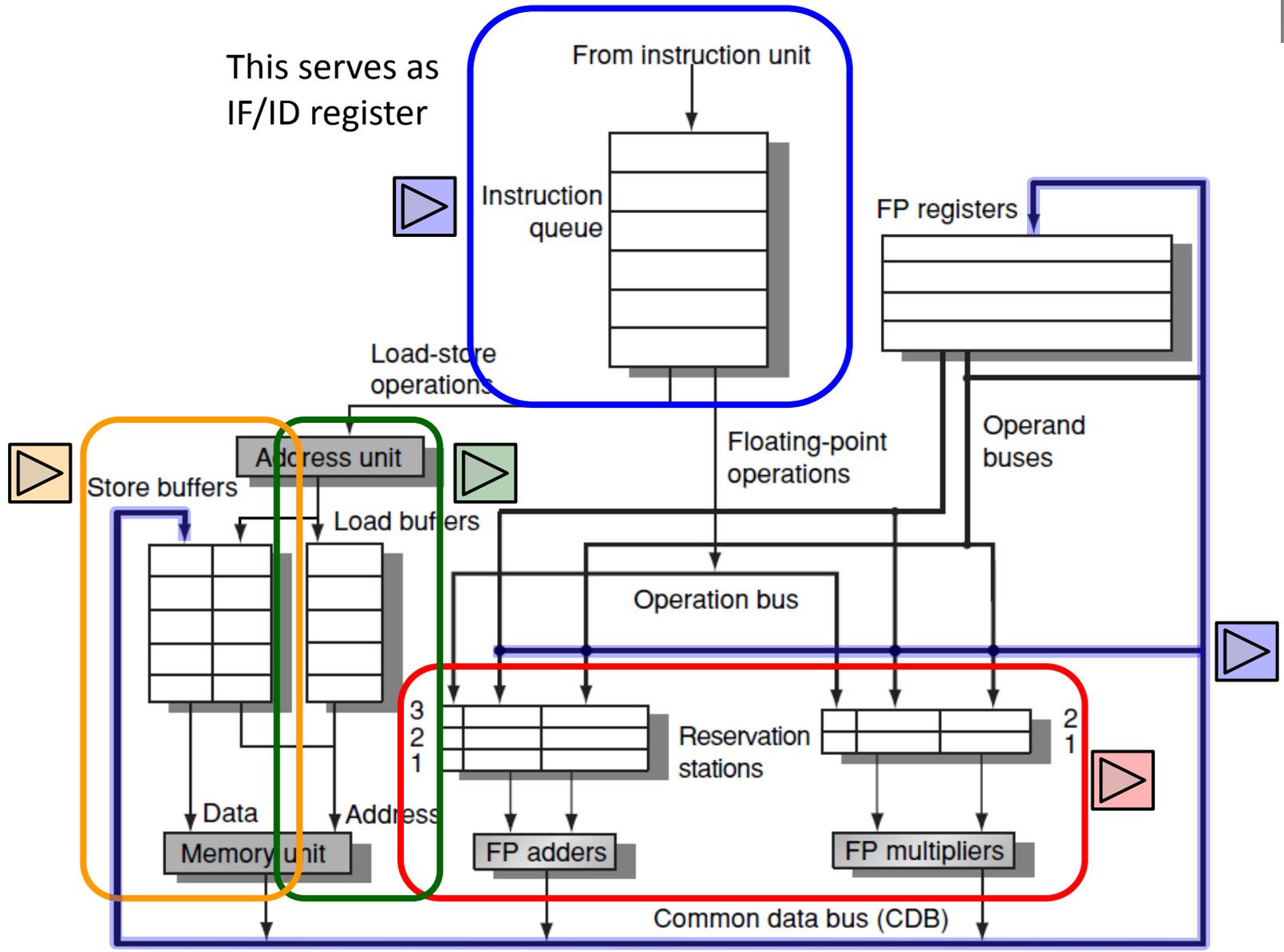
Here can be more RSs than real registers, so it can eliminate hazards that compiler could not.

Unlike the ordinary pipelined processor, where the hazard detection and execution control was **centralized**, it is now **distributed**.

The information held at each RS of a functional unit determines when an instruction can start execution at that unit.

RS passes results directly to the functional units where the results are required through **Common Data Bus (CDB)** rather than going through RF.

Pipeline supporting **multiple** execution units and issuing **multiple** instructions per CLK cycle requires more than one CDB.





Instructions are sent from the instruction unit into a **queue** from where they issue in FIFO order. 

RSs include the **operations** and the actual **operands**, together with information for hazard detection and resolution. 

Load buffers:

1. hold the components of the effective address until it is computed,
2. track outstanding loads waiting on memory, and
3. hold the results of completed loads, waiting for the CDB. 



Store buffers:

1. hold the components of the effective address until it is computed,
2. hold the destination addresses of outstanding stores waiting for the data value to store, and
3. hold the address and data to store until the memory unit is available.



All results of the FPU and load unit are put on the **CDB**, which goes to the FP registers, to the RSs and to the store buffers.



The adder implements also subtraction and the multiplier implements also division.



The Steps of an Instruction

1. Issue

Get next instruction from the head of the queue. Instructions are maintained in FIFO and hence issued in-order.

If there is an empty matched RS, issue the instruction to that RS together with the operands if those are currently in RF.

If there is not an empty matched RS, there is a **structural hazard**. Instruction stalls until RS is freed.



If the operands are not in RF, keep track of the functional unit producing the operands. This step **renames registers**, eliminating **WAR** and **WAW** hazards.

2. Execute

If an operand is not yet available, monitor CDB for its readiness.

When available, the operand is placed at any RS awaiting it. When all the operands are available the operation is executed.

By delaying operations until all their operands are available **RAW** hazards are avoided.



Several instructions could become ready on the same CLK cycle.

Independent units can start execution in the same cycle.

If few instructions are ready for the same FPU, choice can be arbitrary.

Load and stores require **two-step execution** process.

The 1st step computes the effective address when the register is available. The address is placed in the load or store buffer.

Load is executed as soon as the memory unit is available.



Stores wait for the value to be stored before being sent to the memory unit.

Load and stores are maintained in the program order to prevent **hazards through memory**.

To preserve **exception** behavior, no instruction is allowed to initiate execution until all branches preceding that instruction in program order have completed.

This guarantees that only instructions that would really be executed raise an exception. If BP is used, the processor must know that the BP is correct before allowing execution of instruction after BP (in program).



If the processor records the exception, it can allow the execution after BP, but raise it only if it enters to write results.

Speculation will provide more complete solution

3. Write Results

When the result is available, put it on the CDB and from there into the RF and any RSs waiting for the result.

Stores are buffered into the store buffer until both the value to be stored and the store address are available.

The result is then written as soon as the memory unit is free.



The Reservation Station Data Structure

Each RS has seven fields:

- **Op** – The operation to perform on the source operands S1 and S2.
- **Qj** , **Qk** – The RS that will produce S1 and S2. $Q_j = 0$ or $Q_k = 0$ indicates that the source operands are available in V_j or V_k , or operand is unnecessary.
- **Vj** , **Vk** – The values of S1 and S2.
- **A** – Holds information for the memory address calculation for load or store.
- **Busy** – This RS and its functional unit are occupied.



Each RF has the field:

- **Qi** – The number of the RS containing the operation whose result should be stored into the register.

Qi=0 means that no active instruction is computing a result destined for this register and the register contents is a valid value.

Each of the load and store buffers have a field *A*, which holds the result of the effective address once the first execution step (of the two-step) is completed.

Tomasulo's scheme has two major advantages:

1. the distribution of the hazard detection logic, and
2. the elimination of stalls for WAW and WAR hazards.



Example: What is the contents of Tomasulo's data structure when the first load has completed and written its result?

```
L.D      F6,32(R2)
L.D      F2,44(R3)
MUL.D    F0,F2,F4
SUB.D    F8,F2,F6
DIV.D    F10,F0,F6
ADD.D    F6,F8,F2
```

Instruction status

Instruction	Issue	Execute	Write Result
L.D F6,32(R2)	✓	✓	✓
L.D F2,44(R3)	✓	✓	
MUL.D F0,F2,F4	✓		
SUB.D F8,F2,F6	✓		
DIV.D F10,F0,F6	✓		
ADD.D F6,F8,F2	✓		

Instruction status is not a part of the hardware



Reservation station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					44 + Regs[R3]
Add1	yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[32 + Regs[R2]]	Mult1		

L.D	F6, 32 (R2)
L.D	F2, 44 (R3)
MUL.D	F0, F2, F4
SUB.D	F8, F2, F6
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

Register status

F0	F2	F4	F6	F8	F10
Mult1	Load2		Add2	Add1	Mult2



WAR hazard involving R6 is eliminated in one of two ways.

L.D	F6, 32 (R2)
L.D	F2, 44 (R3)
MUL.D	F0, F2, F4
SUB.D	F8, F2, F6
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

If the L.D has been completed, V_k field of DIV.D will store the result and is therefore independent of ADD.D (as shown in instruction status).

If L.D had not completed, Q_k of DIV.D would point to Load1 RS and DIV.D would be independent of ADD.D.

In either case the ADD.D can issue and execute without affecting DIV.D.



Example: Assume the following latencies: load 1 cycle, add 2 cycles, multiply 6 cycles and divide 12 cycles.

What the status tables look like when the MUL.D is ready to write result?

Instruction status					
Latency	Instruction		Issue	Execute	Write Result
1	L.D	F6, 32 (R2)	√	√	√
1	L.D	F2, 44 (R3)	√	√	√
6	MUL.D	F0, F2, F4	√	√	
2	SUB.D	F8, F2, F6	√	√	√
12	DIV.D	F10, F0, F6	√		
2	ADD.D	F6, F8, F2	√	√	√



Reservation station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no				L.D	F6, 32 (R2)	
Load2	no				L.D	F2, 44 (R3)	
Add1	no		Load1		MUL.D	F0, F2, F4	
Add2	no				SUB.D	F8, F2, F6	
Add3	no				DIV.D	F10, F0, F6	
Mult1	yes	MUL	Mem[44+Regs[R3]]	Regs[F4]	ADD.D	F6, F8, F2	
Mult2	yes	DIV		Load1			Mult1

Register	Field	F0	F2	F4	F6	F8	F10
status	Qi	Mult1					Mult2

Add has been completed since the operands of DIV.D were copied, thereby avoiding the WAR hazard in F6.

Even if the **load of F6** was delayed, the add into F6 could be executed without triggering a WAW hazard.



Tomasulo Algorithm Details

Instruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	<pre> if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Qi ← r; </pre>

rs and **rt** are the source registers. **rd** is the destination register. **r** is the reservation station (**RS**) or buffer that the instruction is assigned to. **Regs[·]** is the register file, **RegisterStat[·]** is the register status.



If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the RS that will produce the values needed as source operands.

The instruction waits at the RS until both its operands are available, indicated by zero in the Q fields.

The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back.

When an instruction has finished execution and the CDB is available, it can do its write back.



Instruction state	Wait until	Action or bookkeeping
Issue Load or store	Buffer r empty	<pre>if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;</pre>
Load only		<pre>RegisterStat[rt].Qi ← r;</pre>
Store only		<pre>if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};</pre>

imm is the sign-extended immediate field.



Instruction state Wait until Action or bookkeeping

Execute FP operation	$RS[r]. Q_j=0$ and $RS[r]. Q_k=0$	Compute results. Operands are in V_j and V_k ;
Execute Load-store step 1	$RS[r]. Q_j=0$ and r is head of load-store queue	$RS[r]. A \leftarrow RS[r]. V_j + RS[r]. A$;
Execute Load step 2	Load step 1 complete	Read from $Mem[RS[r]. A]$;

All the buffers, registers, and RSs whose value of Q_j or Q_k is the same as the completing RS, update their values from the CDB and mark their Q fields with zero to indicate that values have been received.



Instruction state	Wait until	Action or bookkeeping
-------------------	------------	-----------------------

Write result of FP operation or load	Execution complete at r and CDB available	$\forall x$ (if (RegisterStat[x].Qi=r) {Regs[x] \leftarrow result ; RegisterStat[x].Qi \leftarrow 0 }) ; $\forall x$ (if (RS[x].Qj=r) {RS[x].Vj \leftarrow result ; RS[x].Qj \leftarrow 0 }) ; $\forall x$ (if (RS[x].Qk=r) {RS[x].Vk \leftarrow result ; RS[x].Qk \leftarrow 0 }) ; RS[r].Busy \leftarrow No ;
Write result of store	Execution complete at r and RS[r].Qk=0	Mem[RS[r].A] \leftarrow RS[r].Vk ; RS[r].Busy \leftarrow No ;

The CDB broadcasts its result to many destinations in a single clock cycle.

If the waiting instructions have their operands, they can all begin execution on the next clock cycle.



Loads go through two steps in Execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store.

To **preserve exception** behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed.

Because program order is not maintained after the issue stage, this restriction is usually implemented by preventing any instruction from leaving the issue step, if there is a pending branch in the pipeline.

We will later remove this restriction.



A Loop Example

The power of Tomasulo's algorithm in handling WAR and WAW hazards is demonstrated in loops.

```
Loop:   L.D           F0,0(R1)
        MUL.D        F4,F0,F2
        S.D           F4,0(R1)
        DADDIU       R1,R1,-8
        BNE          R1,R2,Loop; branches if R1≠R2
```

If branched are predicted **taken**, RS usage allows multiple executions of the loop to proceed at once.

The loop is unrolled dynamically by HW, using the RSs obtained by renaming to act as additional registers.

No need for compiler unrolling.



Let all the instructions in two successive iterations be issued, but assume that none of the operations within the loop has completed.

Instruction status

Instruction	From iteration	Issue	Execute	Write Result
L.D F0,0(R1)	1	√	√	
MUL.D F4,F0,F2	1	√		
S.D F4,0(R1)	1	√		
L.D F0,0(R1)	2	√	√	
MUL.D F4,F0,F2	2	√		
S.D F4,0(R1)	2	√		

The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.



Reservation station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load			0		Regs[R1] + 0
Load2	yes	Load	Loop:	L.D	F0,0(R1)		Regs[R1] - 8
Add1	no			MUL.D	F4,F0,F2		
Add2	no			S.D	F4,0(R1)		
Add3	no			DADDIU	R1,R1,-8		
				BNE	R1,R2,Loop;		
Mult1	yes	MUL			Regs[F2]	Load1	
Mult2	yes	MUL			Regs[F2]	Load2	
Store1	yes	Store	Regs[R1]		0	Mult1	
Store2	yes	Store	Regs[R1] - 8		0	Mult2	

Register status

Field	F0	F2	F4	F6	...	F30
Qi	Load2		Mult2			



Two copies of the loop could be sustained with a CPI close to 1.0, provided MULT completes in 4 clock cycles.

For 6 cycles MULT, additional iteration is needed to be processed before the steady state can be reached, requiring more FP RSs.

Load and **store** can safely be done OOO if they access different addresses.

In case of same address, if **load precedes store**, order interchange results in a **WAR** hazard.

If **store precedes load**, interchanging order results in a **RAW** hazard.



Similarly, interchanging **two stores** to the same address results in a **WAW** hazard.

To determine if a **load** can be executed, the processor can check whether any uncompleted preceding **store** (in code order) shares the same memory address.

Let a load have completed A field calculation. Address conflicts are detected by examining the A field of all active store buffers.

If a conflict is found, the load is not sent to the load buffer until the conflicting store completes.



The processor must have computed the A field associated with any earlier memory operation.

A simple solution is to perform the effective address calculations (A field) in code order.

Stores operate similarly, except that the processor must check for conflicts in both load and store buffers.

A store must wait until there are no unexecuted loads or stores that are earlier in program order and share the same memory address.

Notice that loads can be reordered freely. (why?)



Dynamic scheduling yields very high performance, provided branches are predicted accurately. The major drawback is the HW complexity.

Each RS must contain a high speed associative buffer, and complex control logic.

Single CDB is a bottleneck. More CDBs can be added.

Since each CDB must interact with each RS, the associative tag-matching HW must be duplicated at each RS for each CDB.

Summary: Tomasulo's alg. combines two techniques: renaming of the ISA registers to a larger set, and buffering of source operands from the RF.



Tomasulo's scheme, invented for IBM 360/91, is widely adopted in multiple-issue processors since 1990s.

It can achieve high performance without requiring the compiler to target code to a specific pipeline structure.

Caches, with the inherently unpredictable delays, is one of the major motivations for dynamic scheduling.

OOO execution allows the processor to continue executing instructions while awaiting the completion of a cache miss, hiding some of the cache miss penalty.

Dynamic scheduling is a key component of speculation (discussed next).



Hardware-Based Speculation

Hardware speculation extends the ideas of dynamic scheduling.

Branch prediction (BP) reduces the direct stalls attributable to branches, but is insufficient to generate the desired amount of ILP.

Exploiting more parallelism requires to overcome the limitation of control dependence.

It is done by **speculating** on the outcome of branches and executing the program as if our guesses were correct.



Speculation combines three key ideas: **dynamic BP**, **speculation** and **dynamic scheduling**.

dynamic BP speculatively chooses which instructions to execute, allowing the execution of instructions **before** control dependences are resolved.

Speculation fetches, issues, and **executes** instructions, as if BP were **always correct**, unlike **dynamic scheduling** which only fetches and issues such instructions.

A mechanism to handle the situation where the speculation is incorrect is required (**undo**).



An **undo** capability is required to cancel the effects of an incorrectly speculated sequence.

Dynamic scheduling **without** speculation only partially overlaps basic blocks because it requires that a **branch be resolved** before actually executing any instructions in the successor basic code block.

Dynamic scheduling **with** speculation deals with the scheduling of different combinations of basic code blocks.

HW-based speculation is essentially a **data-flow execution**: Operations execute as soon as their operands are available.



An instruction is executed and bypassing its results to other instructions.

It however does not perform any updates that cannot be undone (writing to RF or MEM), until it is known to be no longer speculative.

This additional step in the execution sequence is called **instruction commit**.

instructions may finish execution considerably before they are ready to commit.

Speculation allows instructions to execute **OOO** but it forces them to commit **in order**.



The commit phase requires special set of buffers holding the results of instructions that have finished execution but have not committed yet.

This buffer is called the **reorder buffer (ROB)**. It is also used to pass results among instructions that may be speculated.

The ROB holds the result of an instruction between completion and commitment.

The **ROB is a source of operands** for instructions in the interval between **completion** and **commitment**, just as the RSs provide operands in Tomasulo's algorithm.



In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find the result in the RF.

With speculation, the RF is not updated until the instruction commits.

The ROB is similar to the store buffer in Tomasulo's algorithm. The function of the store buffer is integrated into the ROB for simplicity.

Each entry in the ROB contains four fields:

The **instruction type** field indicates whether the instruction is a branch, a store, or a register operation (ALU, load).



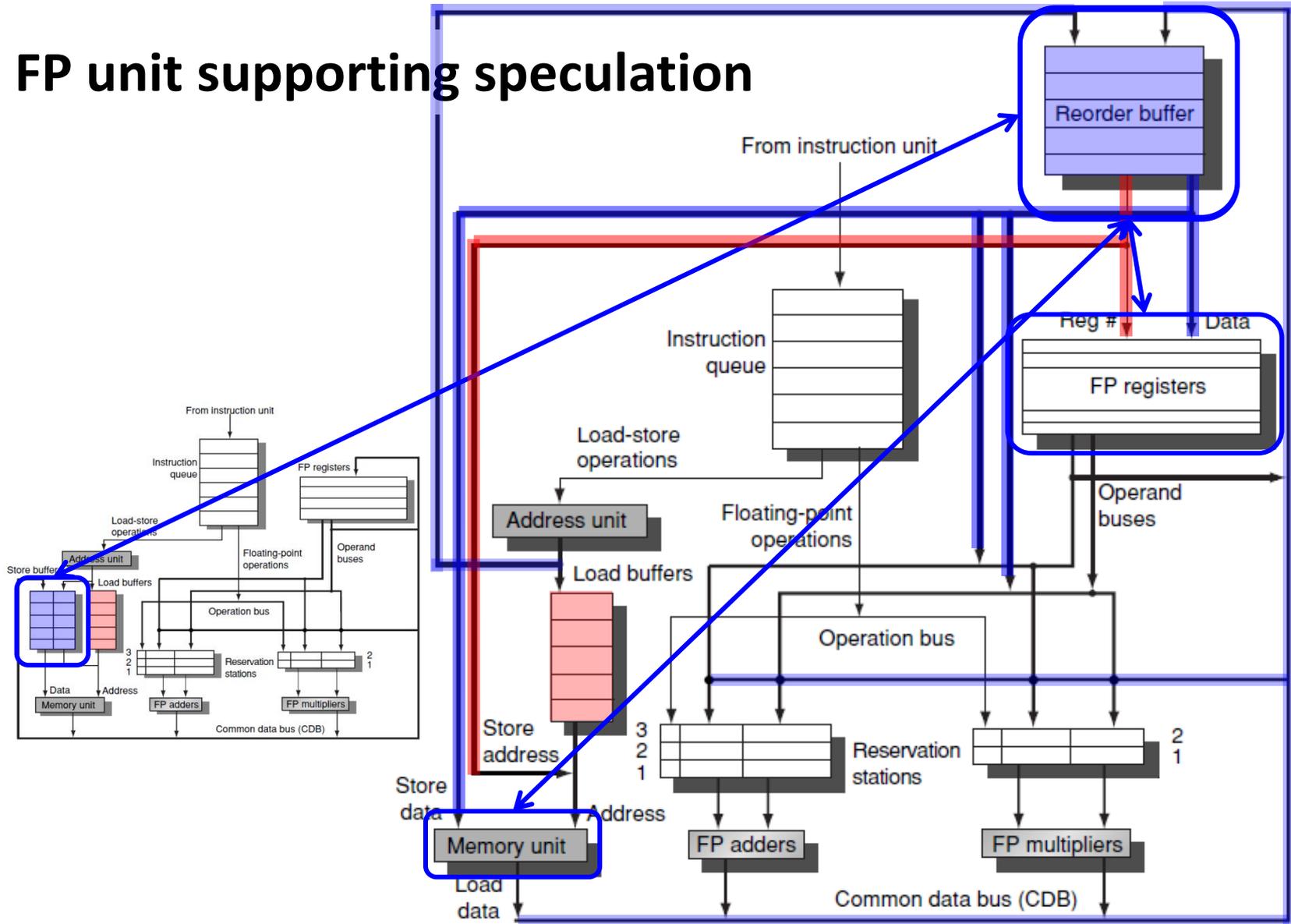
The **destination** field supplies the register number (for loads and ALU operations) or the memory address (for stores) where the instruction result should be written.

The **value** field holds the value of the instruction result until the instruction commits.

The **ready** field indicates that the instruction has completed execution, and the value is ready.



FP unit supporting speculation





The Four Steps of Instruction Execution

Issue. Get an instruction from the instruction queue. Issue it if there is an empty **RS** and an empty slot in the **ROB**, otherwise instruction issue is stalled.

Send the operands to the RS if they are available in either **RF** or the **ROB**. Update the control entries to indicate the buffers are in use.

The number of the **ROB** entry allocated for the result is also sent to the RS, so that it can be used to tag the result when it is placed on the **CDB**.

Notice that the **ROB** is a **queue**. Its update at Issue ensures **in-order commitment**.



Execute. If one or more of the operands is not yet available, monitor the **CDB** while waiting for the register to be computed. This step checks for **RAW** hazards.

When both operands are available at **RS**, execute the operation.

Instructions may take multiple clock cycles in this stage, and **loads** still require two steps in this stage.

Stores need only have the base register available at this step, since execution for a store at this point is only effective address calculation.



Write result. Write it on the **CDB** (with the **ROB** tag sent when the instruction issued) and from the **CDB** into the **ROB** and any **RS** waiting for this result.

Mark the **RS** as available.

Special actions are required for **stores**. If the value to be stored is available, it is written into the Value field of the **ROB** entry for the store.

If not available yet, the **CDB** is monitored until that value is broadcast, at which time the Value field of the **ROB** entry of the store is updated.



Commit. Commitment takes three different sequences.

- 1) A branch** reached the head of the **ROB**. If prediction is **correct** the branch finishes. If **incorrect** (wrong speculation), the **ROB** is **flushed** and execution restarts at the correct successor of the branch.
- 2) Normal** commit occurs when an instruction reaches the head of the **ROB** and its result is present in the buffer. The processor then updates the **RF** with the result and removes the instruction from the **ROB**.
- 3) Store** is similar except that **MEM** is updated rather than **RF**.



Instruction commitment reclaims its entry in the **ROB** and the **RF** or **MEM** destination is updated, eliminating the need for the **ROB** entry.

If the **ROB** fills, issuing instructions stops until an entry is made free, thus enforcing **in-order commitment**.

Example. How tables look when MUL.D is ready to commit? (same example discussed in Tomasulo)

L.D	F6, 32 (R2)
L.D	F2, 44 (R3)
MUL.D	F0, F2, F4
SUB.D	F8, F6, F2
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2



Reorder buffer (status)

Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F6,32(R2)	Commit	F6	Mem[32+ Regs[R2]]
2	no	L.D F2,44(R3)	Commit	F2	Mem[44+ Regs[R3]]
3	yes	MUL.D F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D F8,F2,F6	Write result	F8	#2 – #1
5	yes	DIV.D F10,F0,F6	Execute	F10	
6	yes	ADD.D F6,F8,F2	Write result	F6	#4 + #2

The **ROB** entries are dictated at the issue stage, hence the #1, #2, #4, etc.

Although the SUB.D (#4) has completed execution, it does not commit until the MUL.D (#3) commits.



Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no			L.D F6, 32 (R2)				
Load2	no			L.D F2, 44 (R3)				
Add1	no			MUL.D F0, F2, F4				
				SUB.D F8, F2, F6				
Add2	no			DIV.D F10, F0, F6				
				ADD.D F6, F8, F2				
Add3	no							
Mult1	no	MUL.D	Mem[44+ Regs[R3]]	Regs[F4]				#3
Mult2	yes	DIV.D		Mem[32+ Regs[R2]]	#3			#5

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	...	yes	yes



Issue all instructions

Waits until **RS**[r] and **ROB**[b] both available

Action or bookkeeping

```
if (RegisterStat[rs].Busy) /*in-flight instr. writes rs*/
    {h ← RegisterStat[rs].Reorder;
    if (ROB[h].Ready) /* Instr completed already */
        {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;}
    else {RS[r].Qj ← h;} /* wait for instruction */
} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};
RS[r].Busy ← yes; RS[r].Dest ← b;
ROB[b].Instruction ← opcode;
ROB[b].Ready ← no;
```



Waits until RS[r] and ROB[b] both available

Action or bookkeeping

FP operations and stores

```
if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/  
    {h ← RegisterStat[rt].Reorder;  
    if (ROB[h].Ready) /* Instr completed already */  
        {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;}  
    else {RS[r].Qk ← h;} /* wait for instruction */  
} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};
```

FP	RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;
loads	RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;
stores	RS[r].A ← imm;



Instructions execution	Waits until	Action or bookkeeping
Execute FP op	$(RS[r].Q_j == 0)$ and $(RS[r].Q_k == 0)$	Compute results—operands are in V_j and V_k
Load step 1	$(RS[r].Q_j == 0)$ and there are no stores earlier in the queue	$RS[r].A \leftarrow RS[r].V_j + RS[r].A;$
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from $Mem[RS[r].A]$
Store	$(RS[r].Q_j == 0)$ and store at queue head	$ROB[h].Address \leftarrow RS[r].V_j + RS[r].A;$



Write results all except store

Waits until execution done at **RS** [r] and **CDB** available

```
b ← RS[r].Dest; RS[r].Busy ← no;
∀x(if (RS[x].Qj==b) {RS[x].Vj ← result;
                    RS[x].Qj ← 0});
∀x(if (RS[x].Qk==b) {RS[x].Vk ← result;
                    RS[x].Qk ← 0});
ROB[b].Value ← result;
ROB[b].Ready ← yes;
```

Write results store

Waits until execution done at **RS**[r] and **RS**[r].Qk == 0

```
ROB[h].Value ← RS[r].Vk;
```



Commit

Waits until Instruction is at the head (h) of the **ROB** and **ROB[h].ready == yes**.

Action or bookkeeping

```
d ← ROB[h].Dest; /* register dest, if exists */
if (ROB[h].Instruction==Branch)
    {if (branch is mispredicted)
        {clear ROB[h], RegisterStat; fetch branch dest;};}
else if (ROB[h].Instruction==Store)
    {Mem[ROB[h].Destination] ← ROB[h].Value;}
else /* put the result in the register destination */
    {Regs[d] ← ROB[h].Value;};
ROB[h].Busy ← no; /* free up ROB entry */
/* free up dest register if no one else writing it */
if (RegisterStat[d].Reorder==h)
    {RegisterStat[d].Busy ← no;};
```



A Loop Example

```
Loop:  L.D          F0,0(R1)
        MUL.D       F4,F0,F2
        S.D         F4,0(R1)
        DADDIU      R1,R1,-8
        BNE         R1,R2,Loop; branches if R1≠R2
```

Assume that all the instructions in the loop have been issued twice, and that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution.

Since only the FP pipeline is considered, assume the effective address (R1) for the store is computed by the time the instruction is issued.



Reorder buffer (status)

Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]
2	no	MUL.D F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	yes	S.D F4,0(R1)	Write result	0 + Regs[R1]	#2
4	yes	DADDIU R1,R1,#-8	Write result	R1	Regs[R1] - 8
5	yes	BNE R1,R2,Loop	Write result		
6	yes	L.D F0,0(R1)	Write result	F0	Mem[#4]
7	yes	MUL.D F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	yes	S.D F4,0(R1)	Write result	0 + #4	#7
9	yes	DADDIU R1,R1,#-8	Write result	R1	#4 - 8
10	yes	BNE R1,R2,Loop	Write result		



The register values and any memory values are not written until an instruction commits, enabling **undoing** speculative actions upon miss prediction.

Let the BNE be not taken the first time (the first loop is always performed). The instructions prior to the branch will commit when each reaches the head of the **ROB**.

When the branch reaches the **ROB** head, the **ROB** is cleared and the instructions fetch begins from the other path.

In practice, speculative processors try to recover **as early as possible** after a branch is miss predicted.



Recovery is done by clearing the **ROB** for all entries that appear after the miss predicted branch, allowing those that are in the **ROB** before the branch to continue.

Fetch restarts at the correct branch successor.

Exceptions are handled by not recognizing the exception until it is ready to commit.

If a speculated instruction raises an exception, the exception is recorded in the **ROB**, being flushed along with the instruction when the **ROB** is cleared.



In speculative processor the **RSs** and register status field contain the same basic information as Tomasulo's.

The differences are that **RS** numbers are replaced with ROB entry numbers in the Q_j and Q_k fields, as well as in the register status fields.

A destination field was added to the **RSs**, designating the **ROB** entry destined for the result produced by this **RS** entry.

The key difference from Tomasulo is that no instruction after the earliest uncommitted instruction is allowed to commit (complete).



It implies that the processor with the **ROB** can dynamically execute code while maintaining a **precise exception**.

If MUL.D caused an exception, it waits until it reaches the **ROB's** head and takes the exception, flushing all pending instructions from the **ROB**. Because commitment happens in order, this yields a **precise exception**.

L.D	F6, 32 (R2)
L.D	F2, 44 (R3)
MUL.D	F0, F2, F4
SUB.D	F8, F6, F2
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

Tomasulo's algorithm in contrast completes the SUB.D and ADD.D, and F8 and F6 are overwritten before the MUL.D raised exception, yielding **imprecise exception**.



Multiple Issue

Dynamic scheduling and speculation can achieve an ideal CPI of one.

We would like to decrease the CPI to less than one. But cannot if only one instruction is issued per clock cycle.

A **VLIW (very long instruction word)** multiple-issue processor is issuing a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet.

VLIW processors are inherently statically scheduled by the compiler.



VLIWs use multiple, independent functional units. It issues multiple operations by placing these in one instruction.

Intel Itanium I and II contain six operations per instruction packet.

We consider a VLIW processor with instructions that contain five operations: one integer operation (which could also be a branch), two FP operations, and two memory references.

The instruction have a 16–24 bit field for each unit, yielding an instruction length of 80 - 120 bits.



Example: Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle.

Show an **unrolled** (compiler) version of the loop $x[i] = x[i] + s$ for such a processor. Unroll to minimize stalls. Ignore delayed branches.

```
Loop:   L.D      F0,0(R1)      ;F0=array element
        ADD.D   F4,F0,F2      ;add scalar in F2
        S.D     F4,0(R1)      ;store result
        DADDUI  R1,R1,#-8     ;decrement pointer
                                   ;8 bytes (per DW)
        BNE    R1,R2,Loop     ;branch R1!=R2
```



Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)	Load hazard		
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		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)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)	Addition latency		DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

VLIW instructions that occupy the inner loop replace the unrolled sequence (23 cycles).

The code takes 9 cycles assuming no branch delay. The issue rate is 2.5 operations per cycle.



The efficiency, the percentage of available slots that contained an operation, is about 60%.

To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop.

The VLIW code requires at least eight FP registers, while the same code for the base MIPS can use two FP registers or five when unrolled and scheduled.



+ Dynamic Scheduling + Speculation

Put multiple issue, dynamic scheduling and speculation together. Such microarchitecture is used in modern microprocessors.

Consider an issue rate of two instructions per clock, no different from modern processors that issue more instructions per clock.

Assume a separate integer and FPU, each can initiate an operation on every clock. The pipeline can issue any combination of two instructions in a clock.

Tomasulo's scheme supports integer unit, FPU and speculative execution.



Combining speculative dynamic scheduling with multiple issue requires to be able to complete and commit multiple instructions per clock.

Example: Execute the following code on a two-issue processor, once without speculation and once with speculation.

```
Loop:   LD      R2,0(R1)      ;R2=array element
        DADDIU R2,R2,#1     ;increment R2
        SD      R2,0(R1)    ;store result
        DADDIU R1,R1,#8     ;increment pointer
        BNE    R2,R3,LOOP   ;branch if not last element
```



There are separate integer units for address calculation, ALU operations, and branch condition evaluation.

Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.



No speculation

Loop iteration	Instruction	Issue	Execute	Mem access	Write CDB	Comment
1	LD R2,0(R1)	1	2	3	4	Execute directly
1	DADDIU R2,R2,#1	1	5	6	6	Wait for LW
1	SD R2,0(R1)	2	3	7	7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3	3	4	Execute directly
1	BNE R2,R3,LOOP	3	7	7	7	Wait for DADDIU
2	LD R2,0(R1)	4	8	9	9	Wait for BNE
2	DADDIU R2,R2,#1	4	11	11	11	Wait for LW
2	SD R2,0(R1)	5	9	13	13	Wait for DADDIU
2	DADDIU R1,R1,#8	5	8	8	9	Wait for BNE
2	BNE R2,R3,LOOP	6	13	13	13	Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17	17	18	Wait for LW
3	SD R2,0(R1)	8	15	19	19	Wait for DADDIU
3	DADDIU R1,R1,#8	8	14	14	15	Wait for BNE
3	BNE R2,R3,LOOP	9	19	19	19	Wait for DADDIU

Cannot start EXE until branch resolution

Cannot start EXE until branch resolution

19



With speculation

Loop iteration	Instruction	Issue	Execute	Read access	Write CDB	Commit	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5			6	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6		7	No execute delay
2	DADDIU R2,R2,#1	4	8			9	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Start EXE upon R1's value written to CDB

Start EXE upon R1's value written to CDB





Because the completion rate on the non speculative pipeline is falling behind the issue rate rapidly, the non speculative pipeline will stall when a few more iterations are issued.

The branch can be a critical performance limiter. Speculation helps significantly.

The third branch in the speculative processor executes in clock cycle 13, while it executes in clock cycle 19 on the non speculative pipeline.



Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences
Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Hardware speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls