Out of Order Execution, Hardware Speculation, and Advanced Branch Prediction

Introduction

In the previous lab, you implemented a pipelined CPU capable of efficiently executing the RV32I instruction set. Now, a customer wants to extend the processor with more complex instructions, such as multiplication and division, which introduce longer-latency operations. Simply increasing the clock period to accommodate these instructions would degrade overall performance, making multi-cycle execution a more practical approach. However, this introduces challenges like stalling and inefficient resource utilization, which can be addressed through dynamic scheduling.

This lab focuses on designing an Out-of-Order (OoO) Execution CPU that employs dynamic scheduling and hardware speculation to improve performance. Your processor will support a subset of the RV32M (multiplication) extension alongside the RV32I base instruction set from the previous lab. To enable OoO execution, you will implement Tomasulo's Algorithm, as described in *Computer Architecture: A Quantitative Approach*. Additionally, to mitigate the increased impact of branch mispredictions caused by longer execution times, you will implement a gshare branch predictor to enhance prediction accuracy and reduce penalties.

This document is designed as an implementation guide to be used in conjunction with the textbook which provides a detailed conceptual background and general algorithm. Please begin by reading Section 3.3 – 3.6 in *Computer Architecture: A Quantitative Approach*.

Tomasulo's Algorithm with Hardware Speculation

Tomasulo's Algorithm is a dynamic scheduling technique used to improve instruction-level parallelism by allowing out-of-order execution while resolving data hazards through register renaming. It minimizes stalls caused by read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) dependencies by using reservation stations to track instruction status and forwarding operands via a common data bus (CDB).

Hardware speculation extends Tomasulo's Algorithm to improve performance by enabling speculative execution while preventing change in the program state. Dynamic branch prediction allows the processor to predict branch outcomes and execute instructions beyond unresolved

control dependencies. Speculative execution enables instructions following a branch to begin execution before the branch outcome is verified. To ensure correctness, a reorder buffer (ROB) holds speculative results until they are validated, committing all instructions *in order*, this also maintain precise exceptions. Dynamic speculation with dynamic branch prediction reduces stalls and maximizes instruction level parallelism.

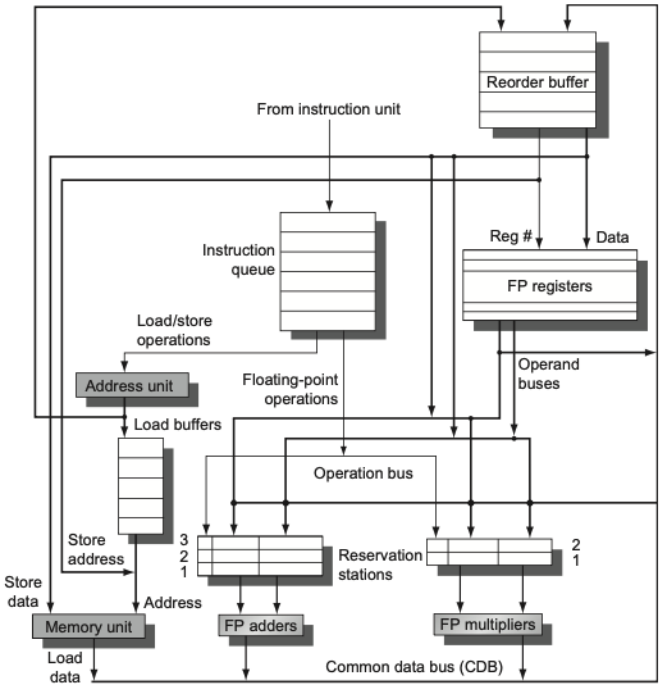

Figure 1: High level architecture of Tomasulo CPU.

Instruction execution can be broken down into the 4 following stages:

Issue: The instruction is fetched and decoded. The packets to send to the reservation station and reorder buffer are assembled. If the operands are available, they are placed in the packet otherwise the source of these operands is written. If the reorder buffer and reservation stations have space, the assembled packets are sent to both.

Execute: The instruction waits until its operands are valid, and then is scheduled to execute on the proper functional unit. The instruction executes, potentially for multiple cycles. Following execution, the functional unit output data is marked as valid.

Write Result: Valid output data is selected between and sent onto the common data bus and a handshake signal is sent to the functional unit and reservation station to make them available. Results from the CDB are forwarded to all dependent instructions in the reservation stations. Data at the instruction's ROB address is also updated.

Commit: If the instruction at the head of the reorder buffer is valid, the instruction type is used to determine how to interpret the data fields. Store instructions write to data memory, register instructions write to the register file, and branches are checked for a branch misprediction. The register status register, preceding load buffer, and branch prediction buffer and global history shift register are updated based on the instruction.

Major Components Implementation

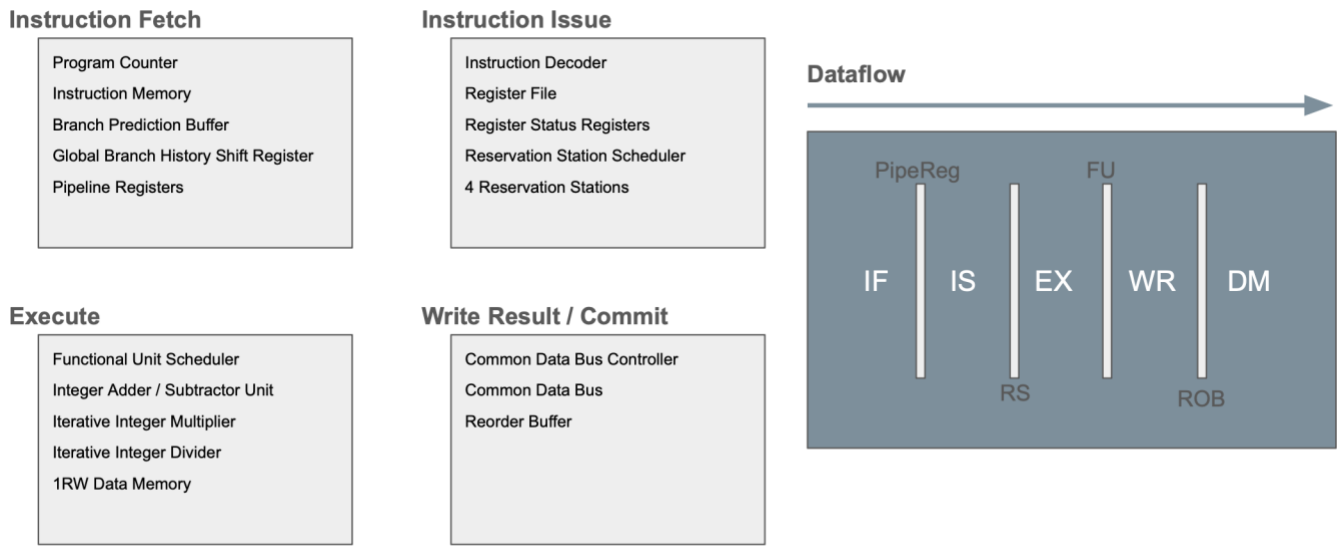

Figure 2: High-level dataflow and major components grouped by instruction stage.

Major components and high-level dataflow of the implementation is shown in Figure 2. A description of the new components with an emphasis on implementation details is provided below:

Branch Prediction Buffer: The BPB is an array of 2-bit FSMs, addressed by the lower bits of the instruction address XOR'd with the global branch history shift register. This scheme ensures accuracy within a wide neighborhood of instruction memory and allows predictions to be overwritten when the program progresses and the lower bits loop around. The 2-bit FSM's diagram is shown in Figure 3.

The FSM requires a branch taken signal and a valid input signal to ensure state updates occur only when a branch instruction is detected. The overall branch prediction buffer will require both a read and write address to allow new predictions of the current instruction and updating prediction results of the instruction at the head of the reorder buffer.

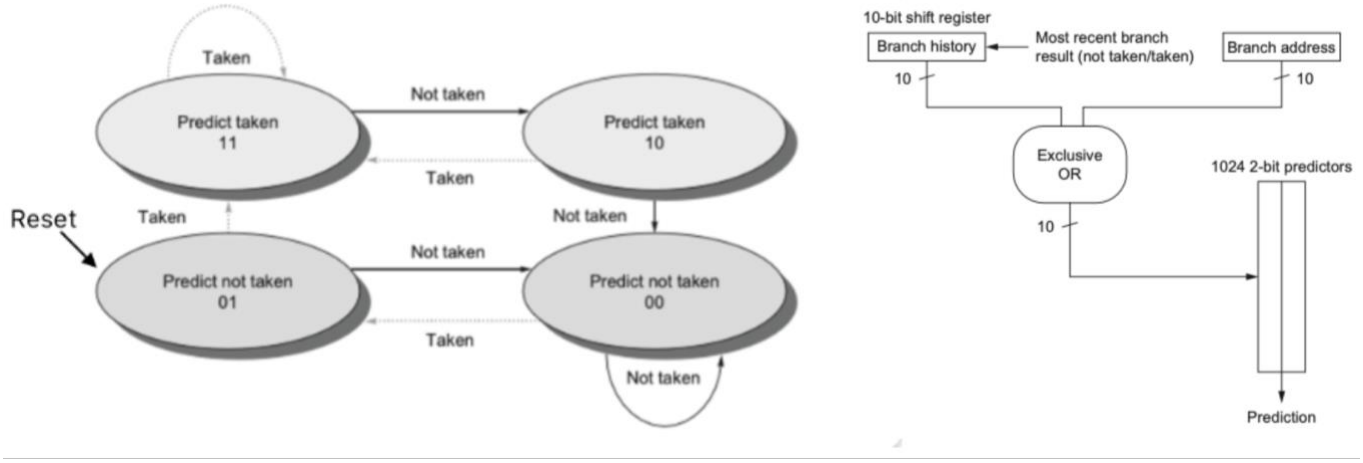

Figure 3: 2-bit branch prediction FSM and gshare block diagram.

Global Branch History Shift Register: This shift register stores the results of the most recent branches and is XOR'd with the instruction address to index the branch prediction buffer. New data should be shifted in on one side and the oldest data should be shifted out of the other side when a new branch instruction has been validated. A block diagram of the gshare branch prediction scheme is also shown in Figure 3.

Pipeline Register: As in the pipelined CPU design, a register should be placed between the instruction fetch and issue stages to reduce the critical path.

Regfile Status Register (RegisterStat): Stores whether the data currently in the register file is valid or if there is data being calculated that will overwrite it.
Must accommodate the following functionality:

- Read from both operands of instruction in issue.
- Conditionally update the status of the destination register from issue stage.
- Conditionally update the destination register status of the commit instruction.

Reservation Station Scheduler: Abstracted in the textbook diagram, this scheduler determines which reservation station to send the instruction being issued to. Many different schemes will work for this – most implementations will not encode a preference between multiple available reservation stations. This unit may also include a queue mechanism to keep track of the order in which instructions are issued for use in the functional unit scheduler.

Reservation Station (RS): Acts as a pipeline register between the issue and execution stages where instructions may wait for multiple cycles if they are not ready for execution. A RS should output valid depending on the the Q fields of the RS packet and the preceding store status register (if a load instruction). Instructions may be valid for multiple cycle without having data accepted if they are need to wait for an earmarked functional unit. This flow of data can be facilitated by interfacing with the functional unit scheduler.
Must accommodate the following functionality:

- Accept data from instruction issue stage.
- Monitor CDB for data to update V and Q fields.
- Handshake with and send data to functional unit and clear itself after data has been accepted or after functional unit output is valid.

When to clear the RS is a key design decision. If cleared following routing to a functional unit additional registers must be included to retain certain RS signals. If cleared upon transmission onto the CDB, the likelihood of stalling due to RS being full increases.

Functional Unit Scheduler: Routes RS operands to the correct functional unit based on the functional unit ready signals and RS valid + control signals. Must account for multiple RS requesting access to the same functional unit. An important consideration is how to handle the store and load instructions which have two parts to their execution. Store instructions require transmission of two separate data packets on the CDB, the memory address and data to store. While we could expand the width of the CDB this would explode our area due to the addition of many additional multiplexers and wires. Load instructions require computing the read memory address using the ALU and reading data memory. Occupying the ALU while waiting for the second stage of instruction execution would be inefficient, particularly for the load instruction which may have to stall due to a preceding store.

Integer Adder / Subtractor Unit: A simplified version of the ALU designed at the start of the quarter this functional unit should perform addition or subtraction of two 32-bit operands depending on an input control signal. It will be beneficial to consider integration with the provided multiplier and divider functional units when designing this unit and the surrounding

logic. The output data should be registered before transmission on the CDB to match the timing of the other functional units.

Common Data Bus (CDB): Serves as a "data highway" for results from the functional units. The CDB is monitored by the reservation stations and reorder buffers to update their data as soon as operations are completed. This unit includes a scheduler that selects between the functional units (adder, multiplier, divider, memory) based on their valid signals, accommodating that multiple units may be valid at the same time. Care should be taken to make this logic as simple logic to send CDB data at the start of the clock cycle (CC).

Reorder Buffer (ROB): Stores the instructions to be committed into processor state – ie. changing register file or data memory – in a FIFO structure to ensure in-order committal. The reorder buffer should have either 8 or 16 addresses, this number places an upper bound on the number of instructions that we can have waiting to be committed at one time. The amount of data stored in each ROB packet should be minimized as each register is replicated for each ROB address creating a large area and power impact especially in adding 32-bit wide data fields.

Must accommodate the following functionality:

- Accept data from the issue stage to.
- Output valid and data signal of issue operands.
- Output preceding store status signals for instructions in RS's to regulate the execution of loads, as loads must stall until there are no preceding loads to avoid RAW through memory.
- Update value and ready fields by monitoring the CDB.
- Commit instructions from the front of the FIFO and clear ROB address.

Provided Components:

Finite state machine diagrams for the provided iterative multiplier and divider are provided in Figures 4 and 5. The multiplier implements the booth multiplication algorithm which performs signed integer multiplication through a series of additions and bit shifts. The divider implements the non-restoring division algorithm which works in a similar manner, with an additional step to deal with negative inputs and cases where the dividend is smaller than the divisor.

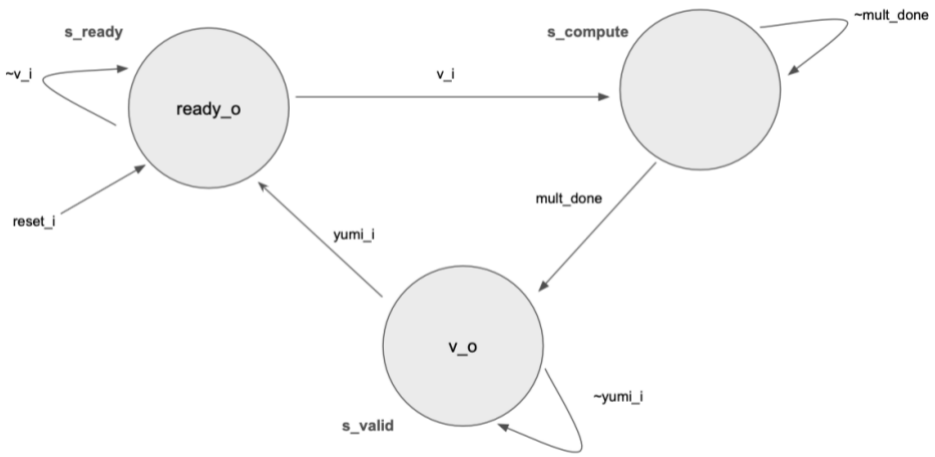

Figure 4: Booth multiplier FSM chart.

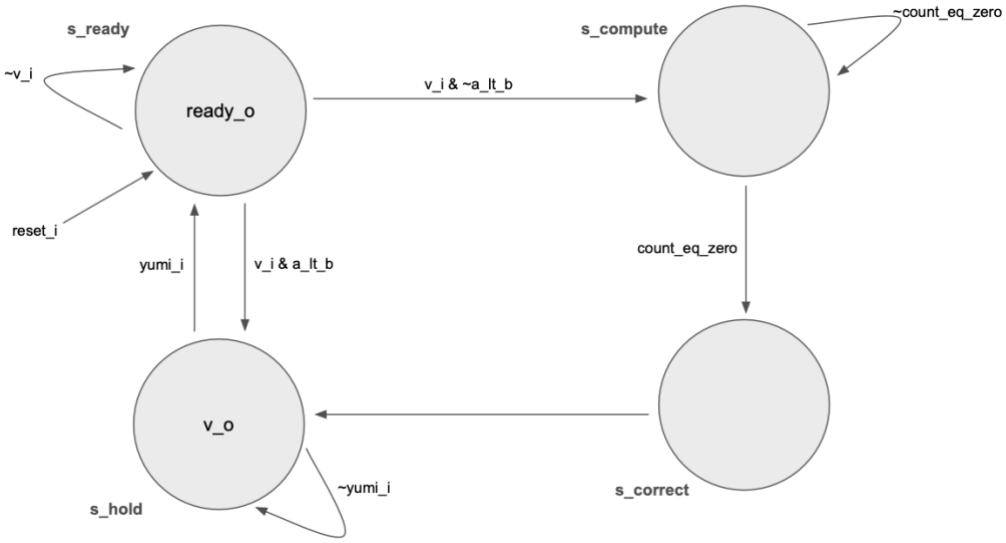

Figure 5: Non-restoring divider FSM chart.

The functional units employ a handshake protocol to facilitate the data transfer to and from the module using the following signals:

v_i (Valid Input): Tells the FU to begin computation with the current inputs. Input data is registered upon assertion, ensuring that subsequent changes do not affect computation.

yumi_i (Yummy): Notifies that the consumer has consumed the FU's output data and the FU can now accept new data.

ready_o (Output Ready): The FU is ready to accept new data. Asserted at start of the CC.

v_o (Output Valid): The output data is valid and will not change until it has been eaten. Asserted at start of the CC.

For those who are interested in the algorithms behind the implementation of the multiplier and divider, C scripts are provided to demonstrate the algorithm in software. Comments have also been left in the source files.

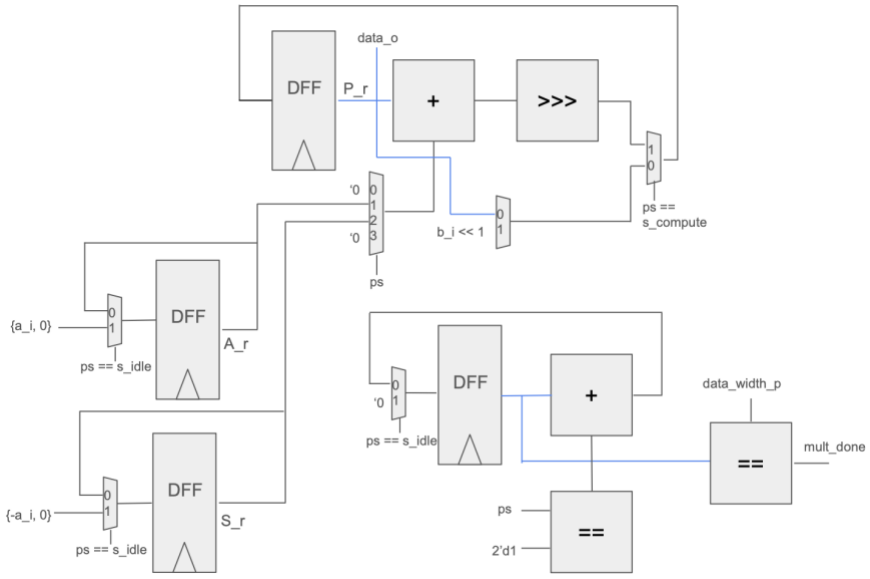

Figure 6: Booth multiplier datapath.

The provided data memory, consistent with previous labs, is 1RW meaning that only one read or write can be performed per cycle in contrast with 1R1W which supports simultaneous reads and writes. The size of an SRAM memory scales with the square root of the number of read/write ports, so this design choice saves on area and power consumption but requires additional logic to control access between load and store instructions which may resolve in the same clock cycle. The block diagram and pseudocode are shown in Figure 7.

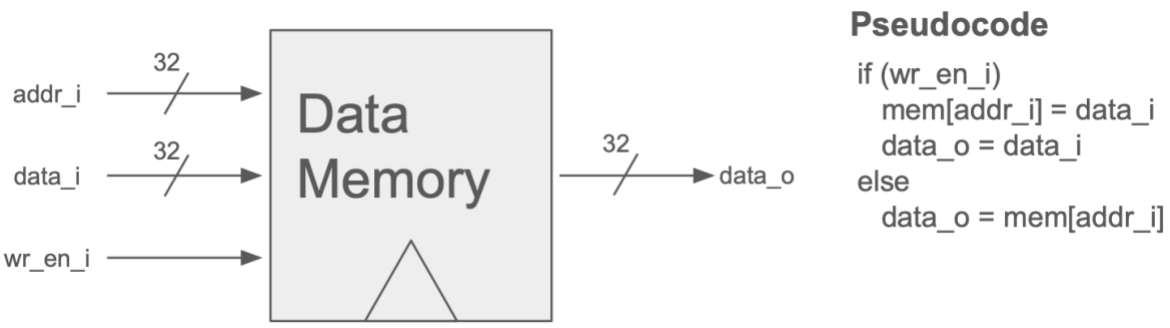

Figure 7: 1RW memory block diagram and pseudocode.

Getting Started

After reading this document and the relevant textbook sections, thoroughly review Figure 3.18 from the textbook, which has been reprinted on this document's last page. The figure outlines Tomasulo's algorithm in pseudocode and will be a crucial reference during implementation.

Next, download the provided scaffolding code and read through the source files. The interface for the register_status module has been provided and utilizes *structs* which are a synthesizable SystemVerilog construct that provide designers a shorthand to name and bundle multiple wires together. Please read this reference to learn more about struct syntax. 'structs.svh' is a provided header file in which you can declare structs to be used in your design. Consider the data that must be stored in the reorder buffer, reservation stations, and transmitted on the CDB and fill in these structs.

After defining the structs, go through the major components and build them piece by piece. A recommended design flow is as follows and demonstrated in 'register_status.sv':

- Review the functional requirements of the module and write corresponding pseudocode.
- Define the module interface (input and output ports).
- Translate the pseudocode into hardware-conscious RTL.

As you develop your solution you should write testbenches for each individual module to ensure their accuracy. Testing code for the provided multiplier, divider, and data memory has also been provided and may be a valuable reference when designing testbenches for the individual components. Benchmarks to test your overall design against have also been provided and will be used for evaluation during the lab demo.

Connecting all modules together may be difficult due to the scale of this project, maintain a focus on clear commenting and naming conventions to help reduce typing errors. Be methodical in your design and think through the hardware you want to generate before writing large chunks of code.

Turn In & Evaluation

The provided benchmarks will be run on your design at the demo time and checked for accuracy. You will also submit your code on the course website and complete a written assessment covering lab concepts at the time of your demo. The following instructions are tested and should be implemented: ADD, ADDI, SUB, SW, LW, MUL, MULH, DIV, REMU, BNE, BEQ, BLT, JAL.

Demo Question Topics

- Hazards created by out of order execution and their solution.
- Comparisons and tradeoffs regarding different CPU implementations.

Resources and Acknowledgements

Computer Architecture: A Quantitative Approach by Hennessy and Patterson

3.2 - 3.6, C.7

Figures 1, 2, and 3 taken from *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson.

Status	Wait until	Action or bookkeeping

Issue all instructions		`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].Dest ← rd; ROB[b].Ready ← no;`

FP operations and stores	Reservation station (r) and ROB (b) both available	`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 operations		`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;`
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	`RS[r].A ← RS[r].Vj + 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].Qj == 0) and store at queue head	`ROB[h].Address ← RS[r].Vj + RS[r].A;`

Write result all but store	Execution done at *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;`

Store	Execution done at *r* and (RS[r].Qk == 0)	`ROB[h].Value ← RS[r].Vk;`

Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	`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;};`

Figure 3.18 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the sources, `r` is the reservation station allocated, `b` is the assigned ROB entry, and `h` is the head entry of the ROB. `RS` is the reservation station data structure. The value returned by a reservation station is called the `result`. `Register-Stat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure.

Figure 8: Steps of Tomasulo's Algorithm with hardware speculation.