Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Català
Deutsch
Español
Français
Italiano
日本語
Српски / srpski
中文
Edit links

Tomasulo algorithm

From Wikipedia, the free encyclopedia

Tomasulo's algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution, designed to efficiently utilize multiple execution units. It was developed by Robert Tomasulo at IBM in 1967, and first implemented in the IBM System/360 Model 91's floating point unit.

The major innovations of Tomasulo's algorithm include register renaming in hardware, reservation stations for all execution units, and a common data bus (CDB) on which computed values broadcast to all reservation stations that may need them. These developments allow for improved parallel execution of instructions that would otherwise stall under the use of scoreboarding or other earlier algorithms.

Robert Tomasulo received the Eckert-Mauchly Award in 1997 for his work on the algorithm.[1]

Contents [hide]

Implementation concepts [edit]

The following are the concepts necessary to the implementation of Tomasulo's Algorithm.

Common data bus [edit]

The Common Data Bus (CDB) connects reservation stations directly to functional units. According to Tomasulo it "preserves precedence while encouraging concurrency".[2]:33 This has two important effects:

1. Functional units can access the result of any operation without involving a floating-point-register, allowing multiple units waiting on a result to load in a single clock cycle.[2]:30
2. Hazard Detection and control execution are distributed. The reservation stations control when an instruction can execute, rather than a single dedicated hazard unit.

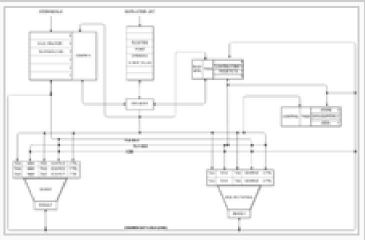
Instruction order [edit]

Instructions are issued sequentially so that the effects of a sequence of instructions, such as exceptions raised by these instructions, occur in the same order as they would on an in-order processor, regardless of the fact that they are being executed out-of-order (i.e. non-sequentially).

Register renaming [edit]

Tomasulo's Algorithm uses register renaming to correctly perform out-of-order execution. All general-purpose and reservation station registers hold either a real values or a placeholder value. If a real value is unavailable to a destination register during the issue stage, a placeholder value is initially used. The placeholder value is a tag indicating which reservation station will produce the real value. When the unit finishes and broadcasts the result on the CDB, the placeholder will be replaced with the real value.

Each functional unit has a single reservation station. Reservation stations hold information needed to execute a single instruction, including the operation and the operands. The functional unit begins processing when it is free and when all source operands needed for an instruction are real.


Tomasulo's floating point unit

Exceptions [edit]

Practically speaking, there may be exceptions for which not enough status information about an exception is available, in which case the processor may raise a special exception, an "imprecise" exception ("imprecise" exceptions cannot occur in non-OoOE implementations)[*why?*].

Programs that experience "precise" exceptions, where the specific instruction that took the exception can be determined, can restart or re-execute at the point of the exception. However, those that experience "imprecise" exceptions generally cannot restart or re-execute, as the system cannot determine the specific instruction that took the exception.

Instruction lifecycle [edit]

The three stages listed below are the stages through which each instruction passes from the time it is issued to the time its execution is complete.

Register legend [edit]

- Op - represents the operation being performed on operands
- Qj, Qk - the reservation station that will produce the relevant source operand (0 indicates the value is in Vj, Vk)
- Vj, Vk - the value of the source operands
- A - used to hold the memory address information for a load or store
- Busy - 1 if occupied, 0 if not occupied
- Qi - the reservation station whose result should be stored in this register (if blank or 0, no values are destined for this register)

Stage 1: issue [edit]

In the issue stage, instructions are issued for execution if all operands and reservation stations are ready or else they are stalled. Registers are renamed in this step, eliminating WAR and WAW hazards.

- Retrieve the next instruction from the head of the instruction queue. If the instruction operands are currently in the registers, then
  - If a matching functional unit is available, issue the instruction.
  - Else, as there is no available functional unit, stall the instruction until a station or buffer is free.
- Otherwise, we can assume the operands are not in the registers, and so use virtual values. The functional unit must calculate the real value to keep track of the functional units that produce the operand.

Pseudocode [edit]

Instruction state	Wait until	Action or bookkeeping

Issue	Station r empty	`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].Q ← r;`

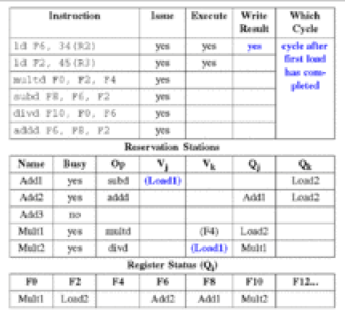
Load or Store	Buffer r empty	if (RegisterStat[rs].Qi¦0) {
		 RS[r].Qj ← RegisterStat[rs].Qi;
		}
		else {
		 RS[r].Vj ← Regs[rs];
		 RS[r].Qj ← 0;
		}
		RS[r].A ← imm;
		RS[r].Busy ← yes;

Load only		RegisterStat[rt].Qi ← r;

Store only		if (RegisterStat[rt].Qi¦0) {
		 RS[r].Qk ← RegisterStat[rs].Qi;
		}
		else {
		 RS[r].Vk ← Regs[rt];
		 RS[r].Qk ← 0
		};

[3]:180

Stage 2: execute [edit]



Instruction	Issue	Execute	Write Result	Which Cycle
ld F6, 34(R2)	yes	yes	yes	cycle after first load has completed
ld F2, 45(R3)	yes	yes		
multd F0, F2, F4	yes			
subd F8, F6, F2	yes			
divd F10, F0, F6	yes			
addd F6, F8, F2	yes			

Reservation Stations

Name	Busy	Op	V_j	V_k	Q_j	Q_k
Add1	yes	subd	(Load1)			Load2
Add2	yes	addd			Add1	Load2
Add3	no					
Mult1	yes	multd		(F4)	Load2	
Mult2	yes	divd		(Load1)	Mult1	

Register Status (Q_i)

F0	F2	F4	F6	F8	F10	F12...
Mult1	Load2		Add2	Add1	Mult2	

Example of Tomasulo's Algorithm[4]

In the execute stage, the instruction operations are carried out. Instructions are delayed in this step until all of their operands are available, eliminating RAW hazards. Program correctness is maintained through effective address calculation to prevent hazards through memory.

- If one or more of the operands is not yet available then: wait for operand to become available on the CDB.
- When all operands are available, then: if the instruction is a load or store
  - Compute the effective address when the base register is available, and place it in the load/store buffer
    - If the instruction is a load then: execute as soon as the memory unit is available
    - Else, if the instruction is a store then: wait for the value to be stored before sending it to the memory unit
- Else, the instruction is an arithmetic logic unit (ALU) operation then: execute the instruction at the corresponding functional unit

Pseudocode [edit]

Instruction state	Wait until	Action or bookkeeping
FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load/store step 1	RS[r].Qj = 0 & r is head of load-store queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]

[3] :180

Stage 3: write result [edit]

In the write Result stage, ALU operations results are written back to registers and store operations are written back to memory.

- If the instruction was an ALU operation
  - If the result is available, then: write it on the CDB and from there into the registers and any reservation stations waiting for this result
- Else, if the instruction was a store then: write the data to memory during this step

Pseudocode [edit]

Instruction state	Wait until	Action or bookkeeping
FP operation or load	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi = r) {` `regs[x] ← result;` `RegisterStat[x].Qi = r` `});` `∀x(if (RS[x].Qj = r) {` `RS[x].Vj ← result;` `RS[x].Qj ← 0;` `});` `∀x(if (RS[x].Qk = r) {` `RS[x].Vk ← result;` `RS[x].Qk ← 0;` `});` `RS[r].Busy ← no;`
Store	Execution complete at r & RS[r].Qk = 0	`Mem[RS[r].A] ← RS[r].Vk;` `RS[r].Busy ← no;`

[3] :180

Algorithm improvements [edit]

The concepts of reservation stations, register renaming, and the common data bus in Tomasulo's algorithm presents significant advancements in the design of high-performance computers.

Reservation stations take on the responsibility of waiting for operands in the presence of data dependencies and other inconsistencies such as varying storage access time and circuit speeds, thus freeing up the functional units. This improvement overcomes long floating point delays and memory accesses. In particular the algorithm is more tolerant of cache misses. Additionally, programmers are freed from implementing optimized code. This is a result of the common data bus and reservation station working together to preserve dependencies as well as encouraging concurrency.[2]:33

By tracking operands for instructions in the reservation stations and register renaming in hardware the algorithm minimizes read-after-write (RAW) and eliminates write-after-write (WAW) and Write-after-Read (WAR) computer architecture hazards. This improves performance by reducing wasted time that would otherwise be required for stalls.[2]:33

An equally important improvement in the algorithm is the design is not limited to a specific pipeline structure. This improvement allows the algorithm to be more widely adopted by multiple-issue processors. Additionally, the algorithm is easily extended to enable branch speculation.[3] :182

Applications and legacy [edit]

Tomasulo's algorithm was unused for several years after its implementation in the System/360 Model 91 architecture. However, it saw a vast increase in utilization during the 1990s for 3 reasons:

1. Once caches became commonplace, Tomasulo's algorithm's ability to maintain concurrency during

unpredictable load times caused by cache misses became valuable in processors.
2. Dynamic scheduling and the branch speculation that the algorithm enables helped performances as processors issued more and more instructions.
3. Proliferation of mass-market software meant that programmers would not want to compile for a specific pipeline structure. The algorithm can function with any pipeline architecture and thus software requires few architecture-specific modifications.[3] :183

Many modern processors implement dynamic scheduling schemes that are derivative of the original Tomasulo’s algorithm, including popular Intel-64 chips.[5]

See also [edit]

- Re-order buffer
- Instruction level parallelism
- Out-of-order execution

External links [edit]

- Dynamic Scheduling - Tomasulo's Algorithm
- HASE Java applet simulation of the Tomasulo's Algorithm

References [edit]

1. ^ "Robert Tomasulo – Award Winner". *ACM Awards*. ACM. Retrieved 8 December 2014.
2. ^ *a b c d* Tomasulo, Robert M. (Jan 1967). "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". *IBM Journal of Research and Development* (IBM) **11** (1): 25–33. doi:10.1147/rd.111.0025. ISSN 0018-8646.
3. ^ *a b c d e* Hennessy, John L.; Patterson, David A. (2012). *Computer Architecture: A Quantitative Approach*. Waltham, MA: Elsevier. ISBN 978-0123838728.
4. ^ "CSE P548 - Tomasulo" (PDF). *washington.edu*. Washington University. 2006. Retrieved 8 December 2014.
5. ^ Intel® 64 and IA-32 Architectures Software Developer’s Manual (Report). Intel. September 2014. Retrieved 8 December 2014.

Categories: Algorithms | Instruction processing