Tomasulo Implementation Analysis

COE17B010

Kausik N

Tomasulo Algorithm:

Tomasulo's algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution and enables more efficient use of multiple execution units.

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 other earlier algorithms.

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. This is

a result of the common data bus and reservation station working together to preserve dependencies.

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 stall waiting times.

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.

Problems Analysed:

1. **Image Convolution - Integer Adder, Integer Multiplier**

2. **Floating Point Convolution - FP Adder, FP Multiplier**

3. **Bitwise XOR (String Comparison) - Shifter (Left), Logical operators (AND, OR, XOR)**

PROBLEM 1

Image Convolution:
(In an image, all the pixel values are *integers* ranging from 0 to 255)

Input:

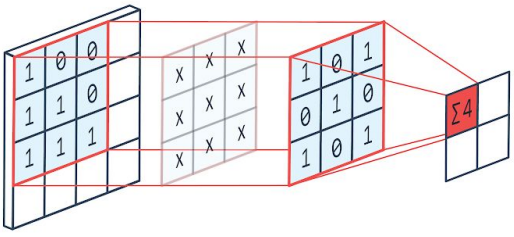
1. Image size - 5x5
2. Window/Kernel size - 3x3

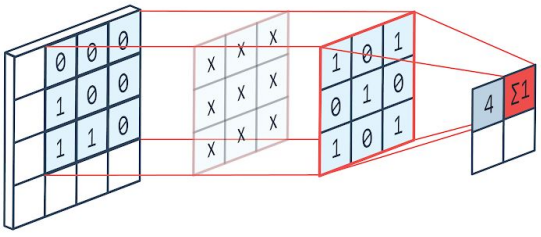
Output:

1. Convolved Image size - 5x5

FUNCTIONAL BLOCKS USED IN PROBLEM 1:

Functional diagram/formula:





$$y[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} h[m,n] \cdot x[i-m, j-n]$$

$$y[i,j] = \sum_{m=-\infty}^{\infty} h[m,-1] \cdot x[i-m, j+1] + h[m,0] \cdot x[i-m, j-0] + h[m,1] \cdot x[i-m, j-1]$$

$$y[i,j] = h[-1,-1] \cdot x[i+1, j+1] + h[-1,0] \cdot x[i+1, j] + h[-1,1] \cdot x[i+1, j-1] + h[0,-1] \cdot x[i, j+1] + h[0,0] \cdot x[i, j] + h[0,1] \cdot x[i, j-1] + h[1,-1] \cdot x[i-1, j+1] + h[1,0] \cdot x[i-1, j] + h[1,1] \cdot x[i-1, j-1]$$

S.No.	Functional blocks	Number of blocks
1	Integer addition	(5*5)*8 = 200
2	Integer multiplication	(5*5)*9 = 225

ASSEMBLY CODE:

S.No.	Instructions	Execution Time
1	Load	2 cc (only for exec)
2	Store	2 cc (only for exec)
3	16-Bit Addition	6 cc
4	16-Bit Multiplication	11 cc

Load/Store Memory Access = 5 cc (Assumed Cache Hit)

IMPLEMENTATION:

Assembly Code : ***ImageConv.s***

No. of instructions = 925

Without Tomasulo Algorithm:

Operations	**Total Time (cc)** No of occur * (Issue+Exec+Mem+WB+Commit)
Load	**18*25*(1+2+5+1+1) = 4500**
Add	**9*25*(1+6+0+1+1) = 2025**
Mul	**9*25*(1+11+0+1+1) = 3150**
Store	**1*25*(1+2+5+0+1) = 225**

In-order issue, In-order execution, In-order commit

Total Time **without Tomasulo** Algorithm = 4500 + 2025 + 3150 + 225

= 9900 clock cycles

CPI = 9900 / 925 = 10.7 cc per ins

With Tomasulo Algorithm:

In-order issue, Out of order execution, In-order commit

Integer Registers = 64

R0 initialised as 0

Integer Adder Reservation Stations = 9

Integer Multiplier Reservation Stations = 9
From Execution of Assembly Code using Tomasulo Algorithm, we obtained output : ***ImageConv_Output.txt***
Total Time **with Tomasulo** Algorithm = **2400 clock cycles**

CPI = 2400 / 925 = 2.59 cc per ins

Speedup = 9900 / 2400 = 4.125

Hazards in Problem 1:

Structural Hazard:

In the assembly code, 9 multiplications are independent of each other at every step and can be done parallelly. But, as only 1 Multiplier Unit is available, only 1 mul can be done at a time. This leads to a structural hazard.
This can be eliminated by adding more functional units (here, 9 mul units), so that independent multiplications can happen parallelly.

Data Hazard:

In the assembly code, the addition at every step requires the output of multiplication. This leads to data dependency/true dependency (RAW) and hence data hazard. This hazard can be eliminated by adding more reservation stations. In our case, we have 9 reservation stations for both integer addition and integer

multiplication, so stalling won't happen as the reservation stations never get full.
Moreover, adding the reservation stations along with more functional units help us in performing out-of-order execution of instructions.

Stalling in Problem 1:

Stalling happens here as at every step, addition accumulation depends on output of multiplication and hence addition is stalled till multiplication completes.

Any WAR hazard (name dependency/false dependency) is eliminated by using the register renaming technique.

PROBLEM 2

Floating-Point Convolution:
(Float values ranges from 0.00 to 16.00)

Input:

1. Matrix size - 5x5
2. Window/Kernel size - 3x3

Output:

1. Convolved Matrix size - 5x5

FUNCTIONAL BLOCKS USED IN PROBLEM 1:

Functional diagram/formula:

$$y[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} h[m,n] \cdot x[i-m, j-n]$$

S.No.	Functional blocks	Number of blocks
1	FP addition	(5*5)*8 = 200
2	FP multiplication	(5*5)*9 = 225

ASSEMBLY CODE:

S.No.	Instructions	Execution Time
1	Load	2 cc (only for exec)
2	Store	2 cc (only for exec)
3	Half-Precision FP Addition	21 cc
4	Half-Precision FP Multiplication	24 cc

Load/Store Memory Access = 5 cc (Assumed Cache Hit)

IMPLEMENTATION:

Assembly Code : ***FloatingConv.s***

No. of instructions = 925

Without Tomasulo Algorithm:

Operations	**Total Time (cc)** No of occur * (Issue+Exec+Mem+WB+Commit)
Load	**18*25*(1+2+5+1+1) = 4500**
FAdd	**9*25*(1+21+0+1+1) = 5400**
FMul	**9*25*(1+24+0+1+1) = 6075**
Store	**1*25*(1+2+5+0+1) = 225**

In-order issue, In-order execution, In-order commit

Total Time **without Tomasulo** Algorithm = 4500 + 5400 + 6075 + 225

= 16200 clock cycles

CPI = 16200 / 925 = 17.51 cc per ins

With Tomasulo Algorithm:

In-order issue, Out of order execution, In-order commit

FP Registers = 64

F0 initialised as 0.0

FP Adder Reservation Stations = 18

FP Multiplier Reservation Stations = 18

From Execution of Assembly Code using Tomasulo Algorithm, we obtained output : ***FloatingConv_Output.txt***

Total Time **with Tomasulo** Algorithm = **2856 clock cycles**

CPI = 2856 / 925 = 3.09 cc per ins

Speedup = 16200 / 2856 = 5.672

Hazards in Problem 2:

Structural Hazard:

In the assembly code, 9 fp multiplications are independent of each other at every step and can be done parallelly. But, as only 1 FPMultiplier Unit is available, only 1 fmul can be done at a time. This leads to a structural hazard.

This can be eliminated by adding more functional units (here, 9 mul units), so that independent fp multiplications can happen parallelly.

Data Hazard:

In the assembly code, the addition at every step requires the output of fp multiplication. This leads to data dependency/true dependency (RAW) and hence data hazard. This hazard can be eliminated by adding more reservation stations. In our case, we have 18 reservation stations for both fp addition and fp multiplication, so stalling won't happen as the reservation stations never get full.
Moreover, adding the reservation stations along with more functional units help us in performing out-of-order execution of instructions.

Stalling in Problem 2:

Stalling happens here as at every step, fp addition accumulation depends on output of fp multiplication and hence fp addition is stalled till fp multiplication completes.

Any WAR hazard (name dependency/false dependency) is eliminated by using the register renaming technique.

PROBLEM 3

String Comparison using Bitwise XOR
(Used in network error checking - Hamming Distance)

Input:

1. Two 16-Bit Binary Values

Output:

1. Bitwise XOR of 2 inputs

FUNCTIONAL BLOCKS USED IN PROBLEM 3:

S.No.	Functional blocks	Number of blocks
1	AND	2*16 = 32
2	OR	16
3	XOR	16
4	Left Shifter	16

ASSEMBLY CODE:

S.No.	Instructions	Execution Time
1	Load	2 cc (only for exec)
2	Store	2 cc (only for exec)

3	AND	1 cc
4	OR	1 cc
5	XOR	1 cc
6	Left Shift	4 cc

Load/Store Memory Access = 5 cc (Assumed Cache Hit)

IMPLEMENTATION:

Assembly Code : ***BitwiseXOR.s***

No. of instructions = 82

Without Tomasulo Algorithm:

Operations	**Total Time (cc)** No of occur * (Issue+Exec+Mem+WB+Commit)
Load	**2*(1+2+5+1+1) = 20**
AND	**32*(1+1+0+1+1) = 128**
OR	**16*(1+1+0+1+1) = 64**
XOR	**16*(1+1+0+1+1) = 64**
Left Shift	**16*(1+4+0+1+1) = 112**
Store	**1*(1+2+5+0+1) = 9**

In-order issue, In-order execution, In-order commit

Total Time **without Tomasulo** Algorithm = 20 + 128 + 64 + 64 + 112 + 9

= 397 clock cycles

CPI = 397 / 82 = 4.84 cc per ins

With Tomasulo Algorithm:

In-order issue, Out of order execution, In-order commit

16-Bit Registers = 16

Logical Reservation Stations = 8

Shifter Reservation Stations = 2

From Execution of Assembly Code using Tomasulo Algorithm, we obtained output : ***BitwiseXOR_Output.txt***

Total Time **with Tomasulo** Algorithm = **105 clock cycles**

CPI = 105 / 82 = 1.28 cc per ins

Speedup = 397 / 105 = 3.781

Hazards in Problem 3:

Structural Hazard:

In the assembly code, 2 AND operations are independent of each other at every step and can be done parallelly. But, as only 1 Logical Unit is available, only 1 AND operation can be done at a time. This leads to a structural hazard.

This can be eliminated by adding more functional units (here, 2 logical units), so that independent AND operations can happen parallelly.

Data Hazard:

In the assembly code, the XOR and OR operations at every step require the output of both ANDs. This leads to data dependency/true dependency (RAW) and hence data hazard. This hazard can be eliminated by adding more reservation stations. In our case, we have 8 reservation stations for logical operation and 2 reservation stations for shifting operation, so stalling won't happen as the reservation stations never get full. Moreover, adding the reservation stations along with more functional units help us in performing out-of-order execution of instructions.

Stalling in Problem 3:

Stalling happens here as at every step, XOR depends on output of both ANDs and hence XOR is stalled till AND completes. Also, OR depends on output of XOR and hence OR is stalled till XOR completes.
Any WAR hazard (name dependency/false dependency) is eliminated by using the register renaming technique.