Tomasulo Hardware Implementation

CED18I015 - DARA SANJANA

COE18B019-GADI JAYASATHWIKA

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. It was developed by Robert Tomasulo at IBM in 1967 and was first implemented in the IBM System/360 Model 91's floating point unit.

Goal: To get a high performance without special compilers.

Motivation to develop this algorithm:

- long FP delays
- only 4 FP registers
- wanted common compiler for all implementations

Key features & hardware structures:

- reservation stations
- distributed hazard detection & execution control
    1. forwarding to eliminate RAW hazards
    2. register renaming to eliminate WAR & WAW hazards
    3. deciding which instruction to execute next
- common data bus

Reservation stations: They are the buffers for Functional Units which store the stalled instructions in order to eliminate RAW hazards. Each reservation station decides when to dispatch instructions to its function unit.

Components of a reservation station:

- Op: Operation to perform in the Functional Unit.
- Vj, Vk: Value of Source operands. Source operands can be values or names of other reservation station entries or load buffer entries that will produce the value.
  (A: used to hold the memory address information for a load or store.)
- Qj, Qk: the reservation station that will produce the relevant source operand (0 indicates the value is in Vj, Vk).
  (Qi - (Only one register unit) the reservation station whose result should be stored in the

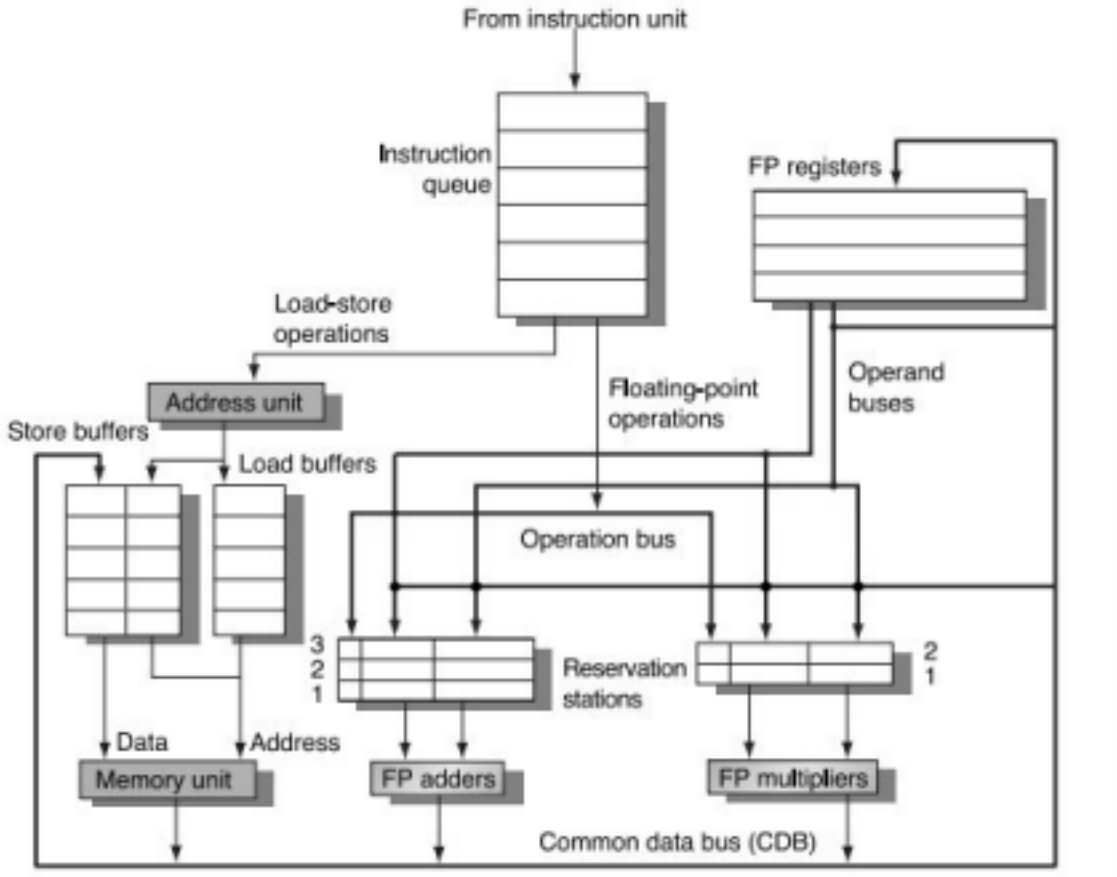
store register (if blank or 0, no values are destined for this register.)

- Busy: Indicates if the reservation station is busy. 1 if occupied, 0 if not occupied.

Register renaming: Tomasulo's Algorithm uses register renaming to correctly perform out-of-order execution. All general-purpose and reservation station registers hold either a real value 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.

Common Data Bus (CDB): It connects functional units and load buffer to reservations stations, registers, and store buffer. Each hardware data structure entry that needs a value from the common data bus grabs the value itself, which is referred to as snooping.

Hardware for Tomasulo's algorithm



Stages of Tomasulo Algorithm:

1. ***Issue***: get instruction from Instruction queue

   If a reservation station is free (no structural hazard), control issues instruction & sends operands. Renaming registers is done eliminating WAR and WAW hazards.

2. ***Execution***: operate on operands (EX)

   When both operands are ready then execute. If not ready, watch the Common Data Bus for result to eliminate RAW hazards.

3. ***Write result***: finish execution (WB)

   Write on Common Data Bus to all awaiting units and mark the reservation station available.

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).

OUR EXECUTION:

```
1 LDR F0 2 + R2
2 AND F1 F3 F4
3 ADD F2 F0 F1
4 MUL F3 F4 F3
5 STR 3 + R1 F3
6 MUL F6 F1 F2
7 SUB F0 F4 F0
8 NAND F0 F7 F8
9 end
```

THEORETICAL EXECUTION

		instruction status					execution	write
		instruction		j	k	issue	complete	result
	1	LDR	F0	2	R2	1	3	4
	2	AND	F1	F3	F4	3	5	6
data hazard(RAW)	3	ADDD	F2	F1	F0	5	11	12
	4	MUL	F3	F4	F3	7	17	18
datahazard(WAW)	5	STR	F3	3	R1	9	19	20
	6	MUL	F6	F1	F2	11	21	22
	7	SUB	F0	F4	F0	13	20	21
datahazard	8	NAND	F0	F7	F8	15	22	23
(WAW),(WAR)								

Due to code complexity and length, instruction difference between 2 fetches is high, so we took two clock cycles difference between fetch of two instructions**.**

ARCHITECTURE AT DIFFERENT CLOCK CYCLES:

Reservation stations

				S1	S2	RS for j	RS for k		clock
Time	Name	busy	op	Vj	Vk	Qj	Qk		7
0	Add1	YES	ADD			INS1	INS2		
0	Add2	no							
	Add3	no							
0	Mult1	YES	MUL	LDF4	LDF3				
0	Mult2	no							
	LGCAL								

Register result status

	F0	F1	F2	F3	F5	F6	F7	F8	F9
FU	1	1	3	3					

Reservation stations

			S1	S2	RS for j	RS for k		clock
Name	busy	op	Vj	Vk	Qj	Qk		16
Add1	YES	SUB	LDF0	LDF4				
Add2	no							
Add3	no							
Mult1	YES	MUL	LDF4	LDF3				
Mult2	YES	MUL	LDF1		ADDD(INS3)			
LGCAL	YES	NAND	LDF7	LDF8				

Register result status

	F0	F1	F2	F3	F5	F6	F7	F8	F9
FU	7			4		6			

IMPLEMENTATION OF TOMASULO CODE:

CHALLENGES:

1-> CHECK OPCODE AND CHECK IF RESOURCES ARE AVAILABLE FOR THAT OPERATION.
2-> IF RESERVATION STATIONS ARE BUSY, STALL
3-> IF RESOURCES(REGISTERS) ARE BEING USED BY OTHER INSTRUCTIONS, STALL/WAIT IN RS
4-> INDICATE THE STALLING INSTRUCTION, IF RS OR REGISTERS ARE IDLE/AVAILABLE.
5-> NO ERROR IN COMPUTING ANSWERS FOR ANY OPERATION

SOLUTION CODE:

-> We created functional units of particular sizes for ADD, SUB , MULT, DIV, LOAD/STORE, LOGICAL.
->We invoked verilog adder modules and multiplication modules during execution.
-> We invoked time function before exec starts and after it ends.
-> We marked them as start and end time, we also invoked time function after writeback and marked as writeback time.
->For other load, store and logical functions, we manually wrote code for execution stages.

-> BUSY ARRAY:

->We maintained a busy list to indicate which registers are busy.

-> If an instruction starts execution, it will append the write registers to the list.

-> Before the start of execution, all instructions check if their operands/ source registers are in busy List.

-> IF yes, they will wait.

-> IF no, they start execution by sending their write registers into busy list.

-> After execution completes, the process's registers are removed from the busy list.

-> So in next cycle., when stalled instructions check, the registers will be available.

-> We increment the count of instructions entering the RS, if it is more than number of RS available, we dont allow instructions to enter.

CLOCK CYCLES:

```
{'fetch': 1, 'execute': 3, 'writeback': 4}      LDR F0 2 + R2
{'fetch': 2, 'execute': 4, 'writeback': 5}      AND F1 F3 F4
{'fetch': 3, 'execute': 10, 'writeback': 11}    ADD F2 F0 F1
{'fetch': 4, 'execute': 14, 'writeback': 15}    MUL F3 F4 F3
{'fetch': 5, 'execute': 16, 'writeback': 17}    STR 3 + R1 F3
{'fetch': 6, 'execute': 20, 'writeback': 21}    MUL F6 F1 F2
{'fetch': 7, 'execute': 14, 'writeback': 15}    SUB F0 F4 F0
{'fetch': 8, 'execute': 44, 'writeback': 45}    DIV F0 F7 F8
```

CODE IS ATTACHED IN GOOGLE CLASSROOM ALONG WITH THIS REPORT.

OUTPUT SCREENSHOT IS GIVEN BELOW

```
pardhu@pardhu-VirtualBox:~/codes$ python3 project.py
Enter the File name: ins2
MEMORY: [1, 2, 4, 2, 6, 5]
FP REGS: {'F0': 23, 'F1': 6, 'F2': 12, 'F3': 10, 'F4': 12, 'F5': 6, 'F6': 5, 'F7': 0, 'F8': 2, 'F9': 1}
ADRESS REGS: {'R0': -1, 'R1': -5, 'R2': 25, 'R3': -10, 'R4': 40}
----------------------
LDR F0 2 + R2
AND F1 F3 F4
ADD F2 F0 F1
MUL F3 F4 F3
STR 3 + R1 F3
MUL F6 F1 F2
SUB F0 F4 F0
DIV F8 F7 F8
----------------------


 1   LDR
Read address: ('2', '+', 'R2')
Mem value in read address: 2
Loaded value in F0 : 2

 2   AND
READ REG1 F3 : 10
READ REG2 F4 : 12
WRITE REG F1 : 8


 3   ADD

READ REG1 F0 : 2
READ REG2 F1 : 8
 4   MUL

READ REG1 F4 : 12
READ REG2 F3 : 10
 ADD WRITE REG F2 : 10
MUL WRITE REG F3 : 120

 5   STR

 7   SUB

READ REG1 F4 : 12
READ REG2 F0 : 2
Address in register R1 : -5
Write address: -2
Read REG F3 : 120
Mem value in write address: 120

 6   MUL

READ REG1 F1 : 8
READ REG2 F2 : 10
SUB WRITE REG F0 : 10
MUL WRITE REG F6 : 80

 8   DIV

READ REG1 F7 : 0
READ REG2 F8 : 2
WRITE REG F0 : 0.0
----------------------

{'fetch': 1, 'execute': 3, 'writeback': 4}      LDR F0 2 + R2
{'fetch': 2, 'execute': 4, 'writeback': 5}      AND F1 F3 F4
{'fetch': 3, 'execute': 10, 'writeback': 11}    ADD F2 F0 F1
{'fetch': 4, 'execute': 14, 'writeback': 15}    MUL F3 F4 F3
{'fetch': 5, 'execute': 16, 'writeback': 17}    STR 3 + R1 F3
{'fetch': 6, 'execute': 20, 'writeback': 21}    MUL F6 F1 F2
{'fetch': 7, 'execute': 14, 'writeback': 15}    SUB F0 F4 F0
{'fetch': 8, 'execute': 44, 'writeback': 45}    DIV F8 F7 F8

MEM: [1, 2, 4, 2, 120, 5]
FP REGS: {'F0': 0.0, 'F1': 8, 'F2': 10, 'F3': 120, 'F4': 12, 'F5': 6, 'F6': 80, 'F7': 0, 'F8': 2, 'F9': 1}
ADDRESS REGS: {'R0': -1, 'R1': -5, 'R2': 25, 'R3': -10, 'R4': 40}


#ins    Start-time              End-time                Exec-time               Write-back time
1 :     1619862096.5431793      1619862096.5488544      0.005675077438354492    1619862096.5488615
2 :     1619862096.582018       1619862096.5875845      0.005566590984863281    1619862096.5875971
3 :     1619862096.6044024      1619862096.6742368      0.06977438926696777     1619862096.6742654
4 :     1619862096.6845985      1619862096.7194006      0.11480212211608887     1619862096.7194118
5 :     1619862096.7195094      1619862096.7200747      0.0005652089691162109   1619862096.7200832
6 :     1619862096.719756       1619862096.8149664      0.09521055221557617     1619862096.8149765
7 :     1619862096.7198622      1619862096.7474375      0.027575254440307617    1619862096.7474506
8 :     1619862097.0040712      1619862097.0048075      0.000736236572265625    1619862097.004816
pardhu@pardhu-VirtualBox:~/codes$
```

S