

## Data Dependence — RAW Hazard

takes 20 cycles and is not pipelined


  
 Technische Universität Berlin


div.d F0,F1,F2	IF	ID	D1	D2	D3	...	D20													
add.d F3,F0,F4		IF	ID						A1	A2	A3	A4								
sub.d F12,F10,F11			IF							ID				A1	A2	A3				





Before I describe the details of Tomasulo's algorithm, I will describe the basic idea behind dynamic scheduling. There is a data dependence between the divide and the add double. Then this pipeline diagram shows how the instructions will be executed on the pipeline processor with multi cycle operations. The add double stalls for 19 cycles in its decode stage because it is data dependent on the divide and because the add double stalls in its decode state to subtract also needs to stall in its fetch tails. The basic idea of dynamic scheduling is to get rid of such stall cycles by allowing instructions to execute out of order. If there is only one floating point adder, but they are not the main focus of this lesson. The goal was to achieve high floating point for performance without having to develop a special compiler for the three 6091 processor IBM wanted to use. The three 6091 processor had a number of limitations which made which made it difficult to achieve high floating point performance. First, it had only four double precision floating point registers, which limited the effectiveness of compiler scheduling.



## Tomasulo's Algorithm

- Used in IBM 360/91 FPU (before caches)
- Goal: high FP performance without special compilers
- Conditions:
  - Small number of FP registers (4) prevented efficient compiler scheduling
  - Tomasulo tried to get more effective registers → renaming in hardware
  - Long memory accesses and FP delays



Robert Tomasulo, recipient of the 1997 Eckert-Mauchly Award for the ingenious Tomasulo algorithm.




  
 Technische Universität Berlin

To be able to do so, we need to get the add double out of the way, so to speak. These buffers are called reservation stations. For that reason, in Tomasulo's algorithm, the results produced by functional units. This brings me to the pipeline phases of Tomasulo algorithm. I purposely say phases rather than stages because pipeline phases may take several clock cycles, whereas the pipeline stage always takes a single cycle. In this phase, the next instruction is fetched, but not in a single pipeline register, but into a 5 foot queue of pending instructions.




## Tomasulo Pipeline Phases

- **IF:** fetch next instr into FIFO queue of pending instructions
- **Issue:**
  - get next instr from head of instr queue
  - if matching RS free (no structural hazard), issue instr to RS
    - w/ operand values if they are currently in registers
    - otherwise, w/ identifiers of RSs that will produce operands

### Tomasulo Pipeline Phases

- **IF:** fetch next instr into FIFO queue of pending instructions
- **Issue:**
  - get next instr from head of instr queue
  - if matching RS free (no structural hazard), issue instr to RS
    - w/ operand values if they are currently in registers
    - otherwise, w/ identifiers of RSs that will produce operands
- **Execute:**
  - when all operands available (no RAW) and FU free, execute
  - if not, monitor CDB for result

Then in the second phase, the instructions at the head of the queue is issued to a matching reservation stations. Here I have drawn in total 5 reservation stations, 3 floating point addition register reservation stations that can hold floating point add and subtract operations, and three floating point multiply reservation stations that can hold multiply instructions. Otherwise they are issued together with identifiers of the reservation stations that will produce these operand values in the next phase. In this example, the reservation stations add one to add three are connected to one or several floating point errors, and the reservation stations. And mold two are connected to one or more floating point multiplied multipliers. In order to keep track of the state of every instruction, each reservation station consists of seven fields. The second field, or as one contains the identifier of the reservation station that will produce the first operand value.

### RS Structure

Each RS has 7 fields:

- **op:** operation to perform
- **RS1:** RS that will produce 1<sup>st</sup> operand (0: operand available)
- **RS2:** RS that will produce 2<sup>nd</sup> operand





Note that either the errors field or the value field is valid, but not both. If the instruction has an immediate operand, it is stored here and after address calculation loads and store store the effective address in this field. The operation is an ad or as one is mill two since middle 2 produces the first operand or is 2 is 0, indicating that the 2nd.55. To track whether an operand value is available in a register or is currently being produced, each register also has an additional field Rs. This field is blank or zero if no currently active instruction computes a result destined for this register. The second register is currently being produced by reservation station Mill One and the 4th Register is currently being produced by the reservation station at 2. As explained before, a floating point instruction is issued if a matching reservation station is available.

### Issue FP Instr

- if RS available
- if src operands
  - in regs: issue w/ values
  - "in-flight": link w/ producing RS





### Issue FP Instr

- if RS available
- if src operands
  - in regs: issue w/ values
  - "in-flight": link w/ producing RS

from instr fetch unit

Instr queue

MUL.D F0,F1,F2

FP registers

RS	Value
RS	0.0
Add2	0.1
Add1	0.2
Mul1	0.3

Operation bus

Operand busses

Reservation stations

Add3	Add2	Add1

AES Embedded Systems Architecture

Technische Universität Berlin

When all operands are available, the instruction in the reservation station can be executed. As usual, I will make sure that loads and stores are executed in order. Here is a simple example of instruction execution. The instruction in reservation station ADD 2 highlighted in green is ready to execute. The subtracting instruction in reservation station at two has executed. 0 onto the common data bus together with the identifier of the reservation station. The result is broadcasted on the common data bus to all awaiting reservation stations and the register file.

### Write Result

- When result available, write it on CDB and from there to any RS and register waiting on it

FP registers

RS	Value
RS	0.0
Mul2	0.0
Add2	0.1
Add1	0.2
Mul1	0.3

Reservation stations

Add3	Add2	Add1

Common Data Bus (CDB)

FP multipliers

Mul2	Add2	Mul1

AES Embedded Systems Architecture

Technische Universität Berlin

0.