

Data Dependence — RAW Hazard

takes 20 cycles and is not pipelined

sub.d cannot execute though it does not depend on preceding instrs

LAES
Embedded Systems Architecture

In this lesson I will describe Tomasulo algorithm and how it implements dynamic instruction scheduling. Before I describe the details of Tomasulo's algorithm, I will describe the basic idea behind dynamic scheduling. Here are three MIPS instructions that divide double and add double and subtract double. There is a data dependence between the divide and the ad double. Since the divide writes F0 and the ad reads F0 AS indicated by the Red Arrow. The ad double stalls for 19 cycles in its decode stage because it is data dependent on the divide and because the ad double stalls in its decode state to subtract also needs to stall in its fetch tails. The main point of this example is that the subtract is stalled even though it does not depend on any previous instruction.

Data Dependence — RAW Hazard

takes 20 cycles and is not pipelined

sub.d cannot execute though it does not depend on preceding instrs

LAES
Embedded Systems Architecture

The basic idea of dynamic scheduling is to get rid of such stall cycles by allowing instructions to execute out of order. There are also some stall cycles caused by structural hazard. If there is only one floating point adder, but they are not the main focus of this lesson.

Dynamic Scheduling — The Idea

in-order execution

out-of-order execution

LAES
Embedded Systems Architecture

In an in order processor, it's like a road with one lane.

Dynamic Scheduling — Road Analogy

- One lane: if one car stops, all cars behind it have to stop
 
- Multiple lanes: if one car stops, cars behind it can take over
 

LAES
Embedded Systems Architecture



TU Berlin Logo

If one car stops all cars behind it have to stop also. An out of order processor, on the other hand is like a road with temporarily multiple lanes. If one car stops the cars behind, it can take over. I will now describe Tomasulo's algorithm. This is a picture of Robert Tomasulo, the man who invented the algorithm in 1997.

Tomasulo's Algorithm



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



TU Berlin Logo

LAES
Embedded Systems Architecture

He received the Eckard Mauchly Award, which is an important award in computer science for its ingenious algorithm. If you have an algorithm named after you, you made it in this field. Unfortunately, he passed away in 2008.

Tomasulo's Algorithm

- Used in IBM 360/91 FPU (before caches)



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



TU Berlin Logo

LAES
Embedded Systems Architecture

The compiler developed for the entire 360 family of processes. 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. This motivated Thomas UNU to find a way to get more effective registers, and he did that by implementing register renaming in hardware.

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 implementation of the Tomasulo algorithm



LAES
Embedded Systems Architecture

TU Berlin

In addition, the IBM 360, ninety one had long memory accesses and long floating point delays.

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 implementation of the Tomasulo algorithm



LAES
Embedded Systems Architecture

TU Berlin

Now I describe some key hardware structures used in Tomasulo algorithm. Here's the same example as before. We want to allow the subtract double to proceed when the add double is told. To be able to do so, we need to get the add double out of the way, so to speak. In other words, we need to buffer the add double somewhere and tomasulo's algorithm. Or broadcast on a bus called the common Data bus to all reservation stations.

Key Structures

DIV.D	F0, F2, F4
ADD.D	F10, F0, F8
SUB.D	F12, F8, F14



TU Berlin

- To allow SUB.D to proceed, need to buffer ADD.D somewhere
 - In Tomasulo algorithm these buffers called **Reservation Stations (RSs)**
- To allow ADD.D to proceed when its operands become available, RSs must be informed when result available
 - In Tomasulo algorithm results broadcasted on **Common Data Bus (CDB)**

LAES
Embedded Systems Architecture

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. The first phase is the instruction fetch phase. In this phase, the next instruction is fetched, but not in a single pipeline register, but into a 5 foot queue of pending instructions. The second stage is the issue phase. In this phase, the next instruction is retrieved from the instruction queue, and if there is a free matching reservation station, the instruction is placed in that reservation station.

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



LAES
Embedded Systems Architecture

TU Berlin
Technische Universität Berlin

In this phase, the next instruction is retrieved from the instruction queue, and if there is a free matching reservation station, the instruction is placed in that reservation station.

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



LAES
Embedded Systems Architecture

TU Berlin
Technische Universität Berlin

If there is no free matching reservation station, then we have a structural hazard in the execution has to store. Furthermore, if the values of the opera if the values of the operations of the instruction are currently available in register in register.Then the instruction is issued to the reservation station together with these values. If not, the operand values are currently being produced and the instruction issued together with identifiers of the reservation station that will produce these operands.The third phase in Tomasulo algorithm is the execute phase when all operands are available, meaning there is no raw hazard and a functional unit that can execute the instruction is available.

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



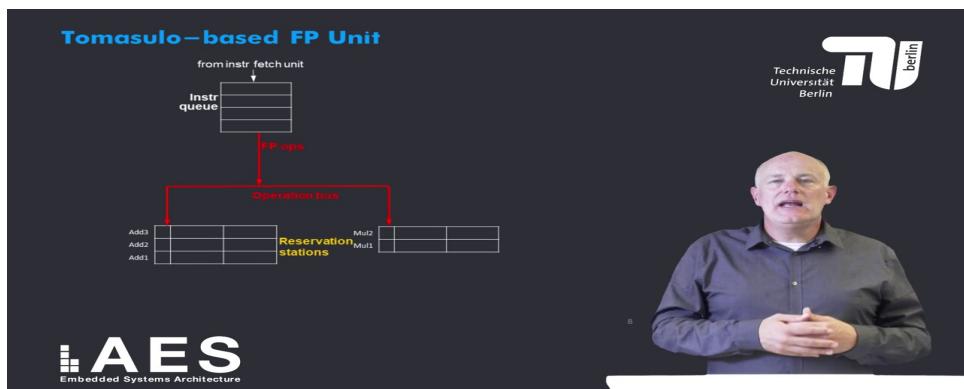
LAES
Embedded Systems Architecture

TU Berlin
Technische Universität Berlin

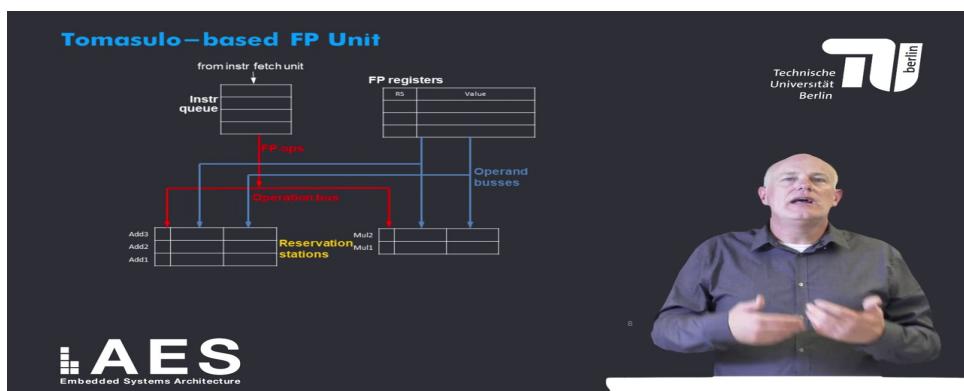
Then the instruction is executed if not, all operands are available, the reservation station must monitor the common data bus for the result to become available. First, the instruction is fetched by the instruction fetch unit into a FIFO queue of pending instructions.



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. As explained before, the instruction is issued together with its operand values if they are available in registers. Otherwise they are issued together with identifiers of the reservation stations that will produce these operant values in the next phase.



The first field is the operation to perform, such as add or subtract. The second field, or as one contains the identifier of the reservation station that will produce the first operand value. A value of 0 indicates that the source operand is already available. Similarly, the RS2 field contains the identifier of the RS that will produce a second operand value. The next two fields are Val, one and Val 2. The value of the 1st and of the second operand. Note that either the errors field or the value field is valid, but not both. The six field contains an immediate or address. 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. An example to make this more concrete, suppose the instruction is an ad. The first operand is being produced by mul two and the 2nd operand value is available in the register.

RS Structure

Each RS has 7 fields:

- **op**: operation to perform
- **RS1**: RS that will produce 1st operand (0: operand available)
- **RS2**: RS that will produce 2nd operand
- **Val1**: Value of 1st operand
- **Val2**: Value of 2nd operand
- **Imm/addr**: holds immediate or effective address
- **Busy**: RS occupied

Example:

- Add
- 1st operand being produced by RS Mul2
- 2nd operand available in register



LAES
Embedded Systems Architecture

Then the content of the reservation station is as shown here.

RS Structure

Each RS has 7 fields:

- **op**: operation to perform
- **RS1**: RS that will produce 1st operand (0: operand available)
- **RS2**: RS that will produce 2nd operand
- **Val1**: Value of 1st operand
- **Val2**: Value of 2nd operand
- **Imm/addr**: holds immediate or effective address
- **Busy**: RS occupied

Example:

- Add
- 1st operand being produced by RS Mul2
- 2nd operand available in register

op	RS1	RS2	Val1	Val2	Imm/addr	busy
add	Mul2	0	n/a	12.55	n/a	1



LAES
Embedded Systems Architecture

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.Operand is available, value one is not applicable since it is being produced by mold two and value 2 is for example 12.

RS Structure

Each RS has 7 fields:

- **op**: operation to perform
- **RS1**: RS that will produce 1st operand (0: operand available)
- **RS2**: RS that will produce 2nd operand
- **Val1**: Value of 1st operand
- **Val2**: Value of 2nd operand
- **Imm/addr**: holds immediate or effective address
- **Busy**: RS occupied

Example:

- Add
- 1st operand being produced by RS Mul2
- 2nd operand available in register

op	RS1	RS2	Val1	Val2	Imm/addr	busy
add	Mul2	0	n/a	12.55	n/a	1



LAES
Embedded Systems Architecture

Register Structure

Each register has field RS:

- RS ID that will produce this value
- Blank / 0 if not applicable



LAES
Embedded Systems Architecture

Register Structure

Each register has field RS:

- RS ID that will produce this value
- Blank / 0 if not applicable

10

LAES
Embedded Systems Architecture

TU Berlin
Technische Universität Berlin

Register Structure

Each register has field RS:

- RS ID that will produce this value
- Blank / 0 if not applicable

10

LAES
Embedded Systems Architecture

TU Berlin
Technische Universität Berlin

Register Structure

Each register has field RS:

- RS ID that will produce this value
- Blank / 0 if not applicable

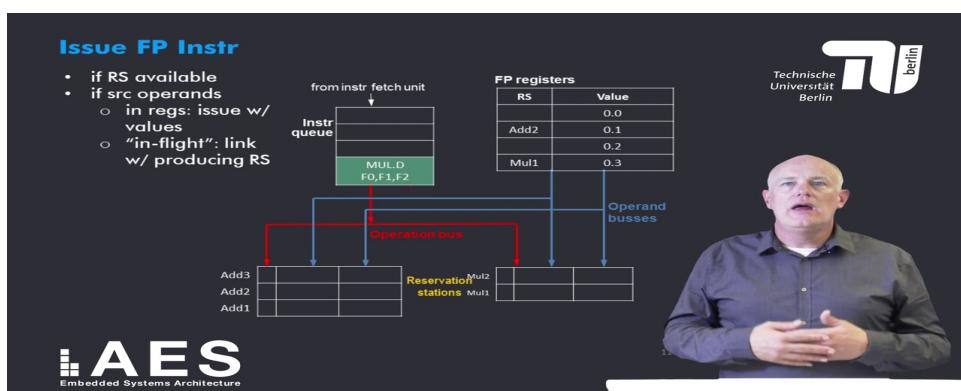
RS	Value
Mul1	
Add2	

11

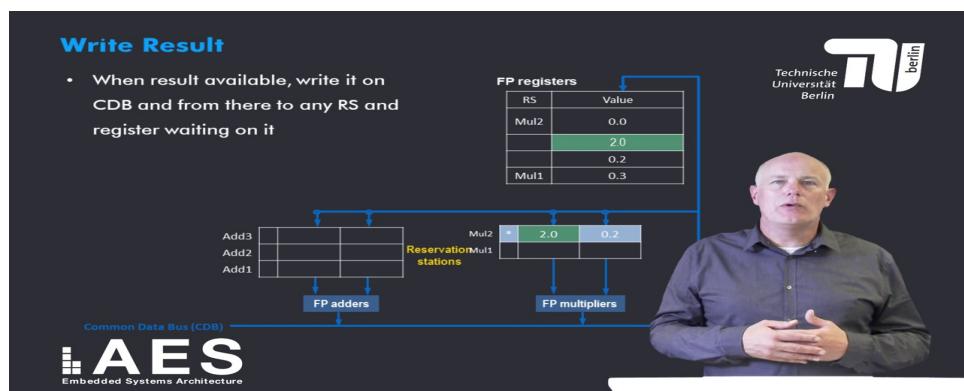
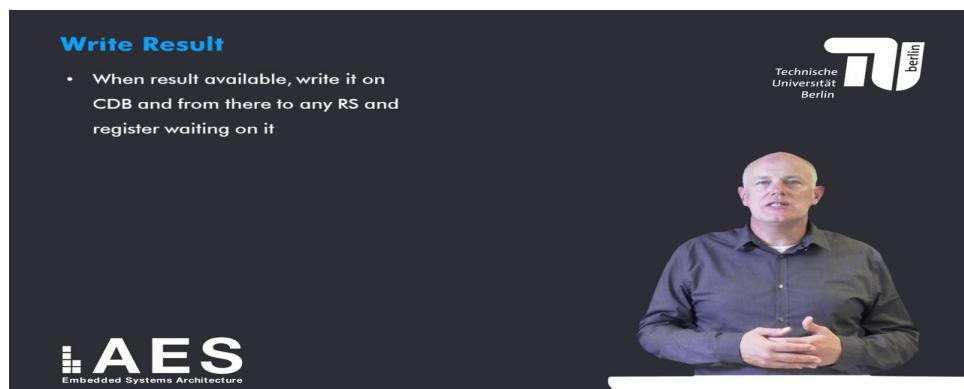
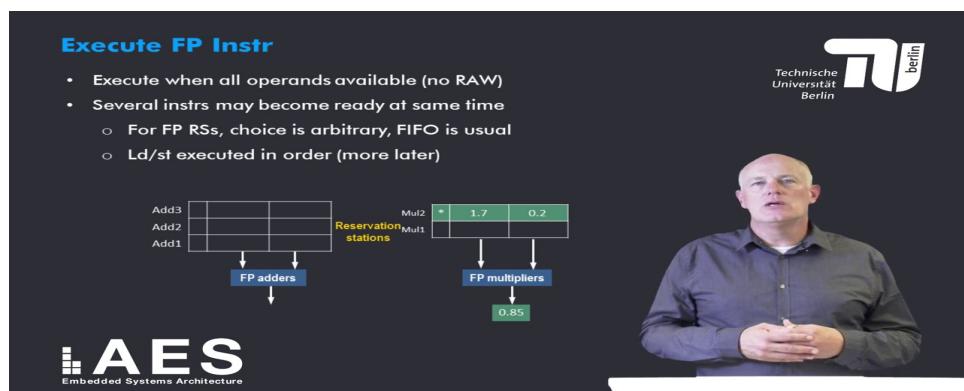
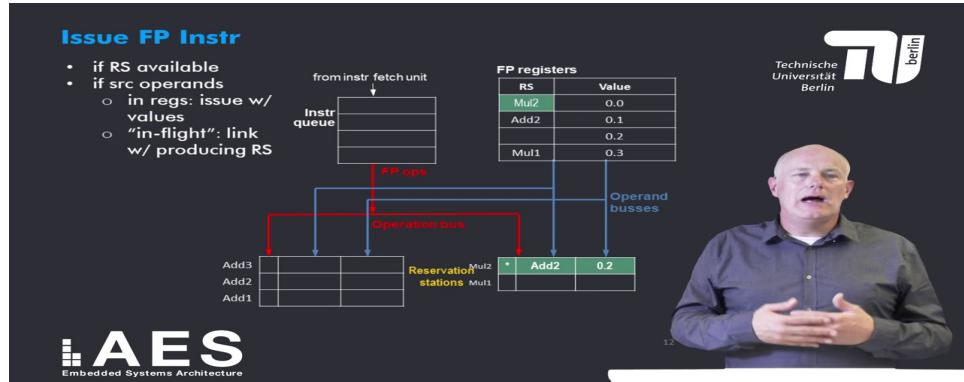
LAES
Embedded Systems Architecture

TU Berlin
Technische Universität Berlin

It is issued together with the operand value. On the other hand, if the source operand is currently in flight being produced, then the instruction is linked with the reservation station that will produce this apparent value. Suppose now that the instruction multiply double F0F1F2 is at the head of the instruction queue and should be issued as highlighted here in green, the register file shows that register F2 is available and contains the value of 0.2 register F1 on the other hand is being produced by reservation station ADD 2.



The multiply instruction will issue 2 reservation station mode 2 for example, highlighted in green.



Write Result

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

The diagram illustrates the Tomasulo's algorithm architecture. On the left, a floating-point register file (FP registers) is shown with four entries:

RS	Value
Mul2	0.0
	2.0 (highlighted in green)
Mul1	0.2

On the right, two reservation stations are shown:

Reservation stations	Mul2	*	2.0	0.2
----------------------	------	---	-----	-----

Arrows indicate data flow from the FP registers to the reservation stations. Below the reservation stations, FP adders and FP multipliers are connected to the Common Data Bus (CDB). The LAES logo (Embedded Systems Architecture) is at the bottom left.

Thanks for watching in the next lesson. I will show how several consecutive instructions are dynamically scheduled using TOMASULO'S algorithm. Hoping that you will fully understand it and never forget about it. Hope to see you back.