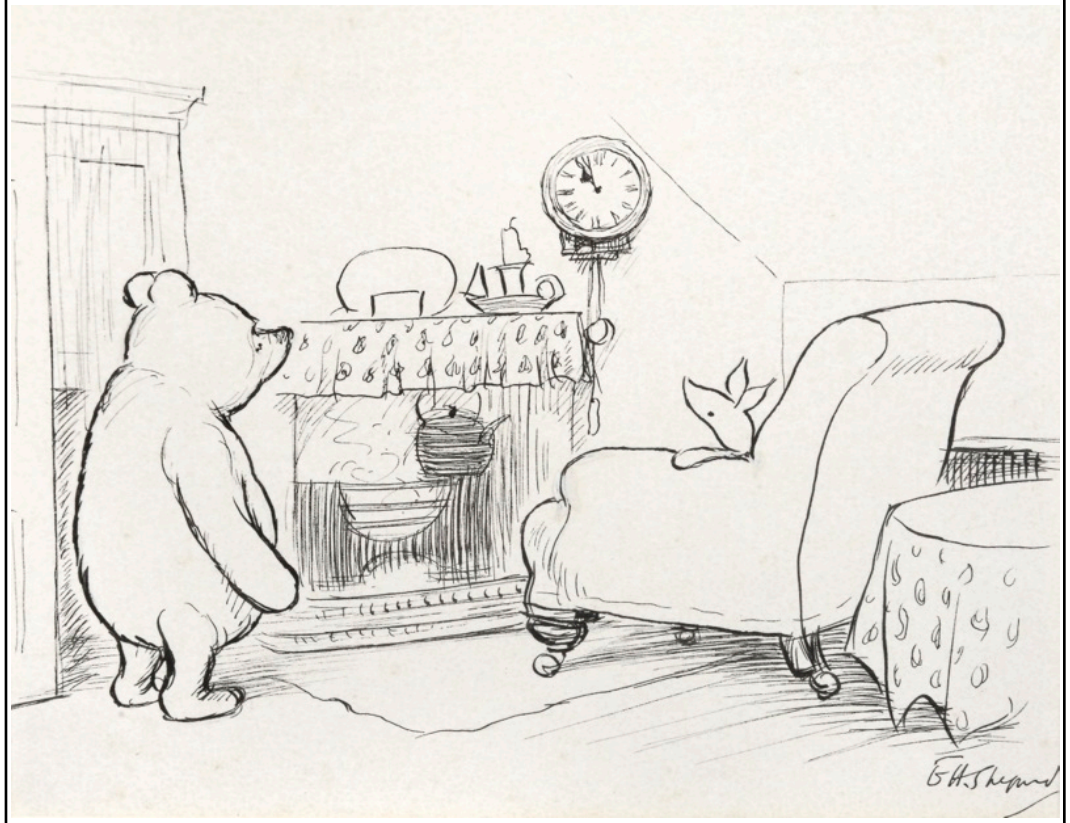
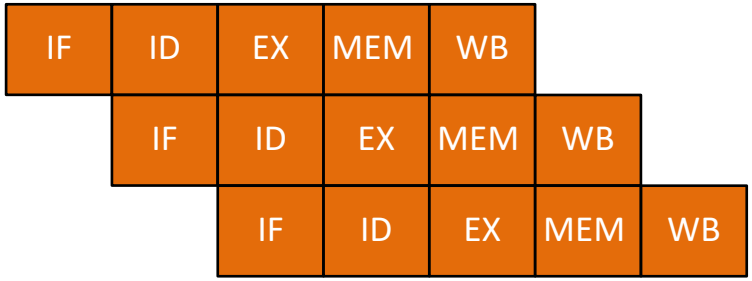
Out of order Execution

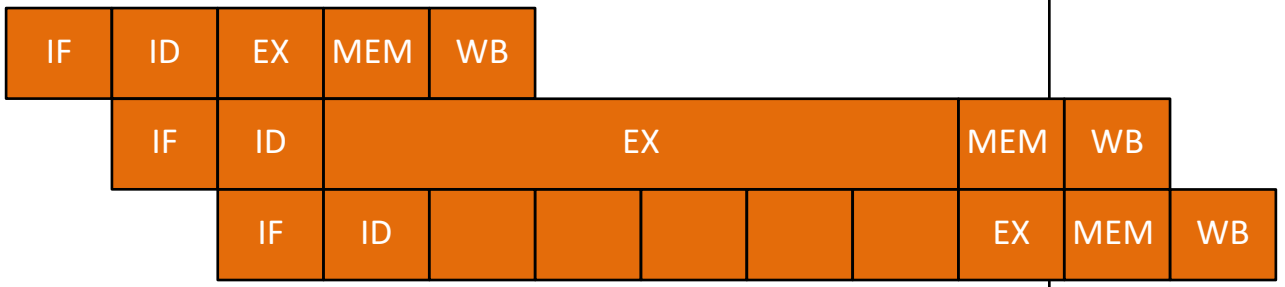


Instruction stages



We have assumed that all stages.

There is a problem with the EX stage multiply (MUL) takes more time than ADD



MUL

ADD

We can clearly delay the execution of the ADD until the MUL is finished.

This stalls the pipeline so is not desirable.

But suppose there is no dependence between the two instructions.

A = B*C & D = E + F

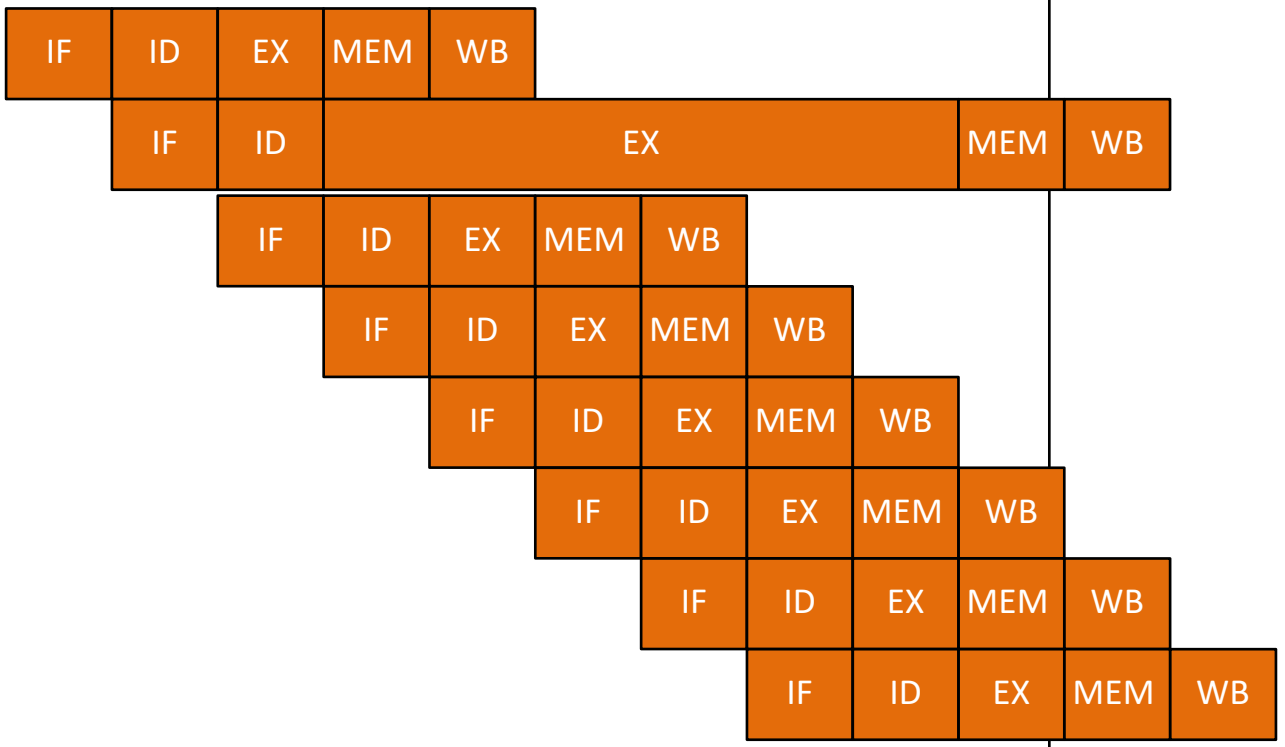
Out of order Execution

Long EX

Start as soon as possible

So start any instructions which are independent of the long latency instruction.

If there are enough we can hide the latency



Seems to work if we have enough instructions.

Even if we don't have enough instructions before the next dependent instruction we may be able to do some out of re-ordering.

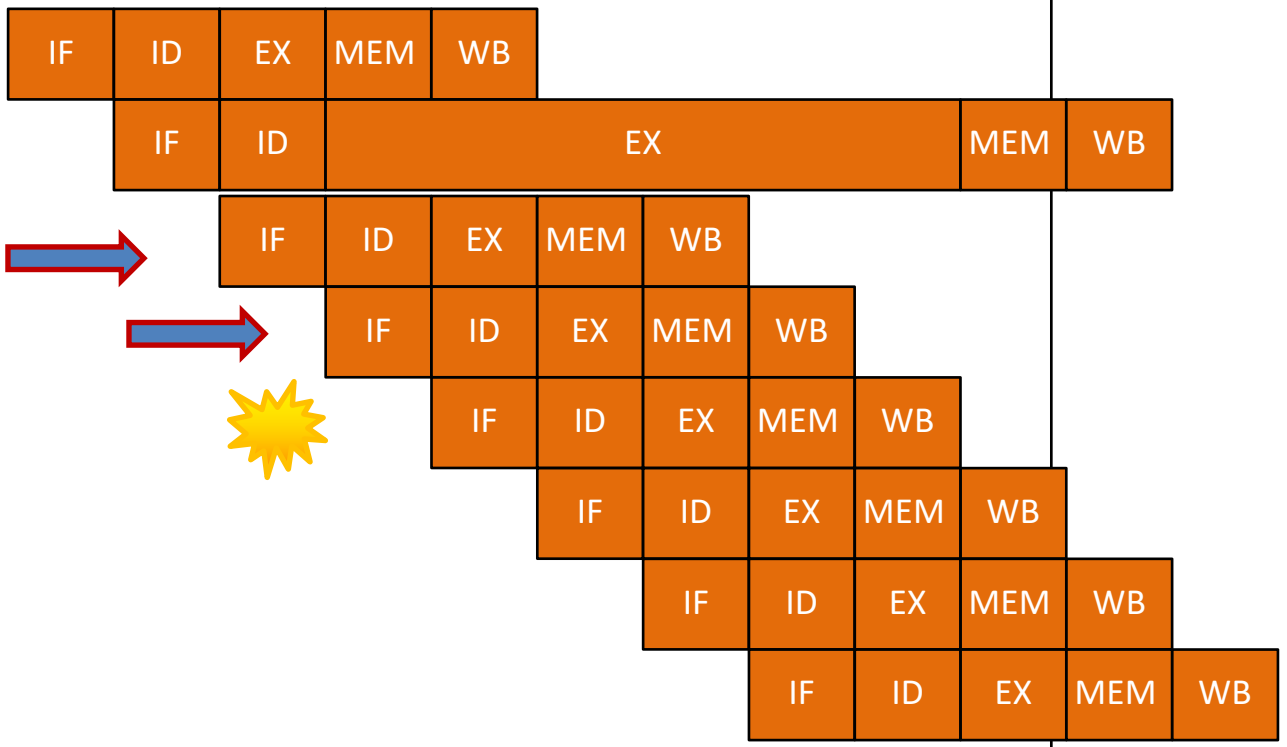
Move the MUL earlier
Look beyond the next dependent instruction

Out of order Execution

Exception

MUL

Exception handling



There is a problem with the MUL instruction (an overflow for instance).

The exception doesn't occur until the next two instructions have completed.

We would like the machine to be in a consistent state when the exception is handled:
•All previous instructions should be retired
•No later instruction should be retired

Or the instruction throws an exception before MUL is complete

Precise exceptions

Retired

This means an instruction is committed, the execution is finished – values are stored and the architectural state of the machine is updated to show the effect of that instruction.

Why

Von Neuman requires it

Enables (easy) recovery from exceptions

Enables (easy) restartable processes

Aids debugging.
[Makes it possible]

Debugging: becomes very hard because for instance – if the statements following the MUL cause a message to be printed out it looks as if the programme should have completed the instruction that is causing the problem.

IBM 360/195

ROB

Reorder buffer – ROB

Complete the instructions out of order – but retire them (update the architectural state of the machine **in order.**

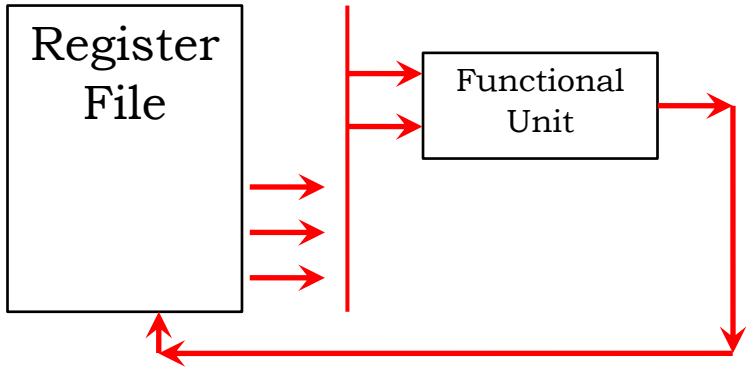
Instructions are decoded and reserve an entry in the Reorder Buffer

When an instruction completes it writes the result into the ROB.

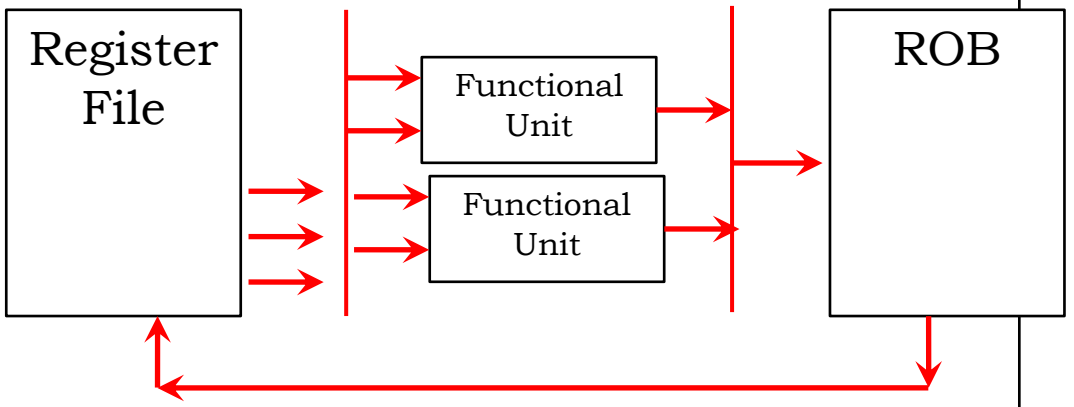
When an instruction is the oldest in the ROB (means it has completed [without throwing an exception]) make it architecturally visible.

Architectural visible: moved to the register file or memory.

ROB Modification



Data in the Register File is part of the ISA.
Visible to Programmer: Architectural state.



Must have more functional units
Plus the ROB – which is not architecturally visible

ROB Entry

Must reserve the space before the op starts

If no space stall

Must contain information which allows it to write the results to register

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exc?

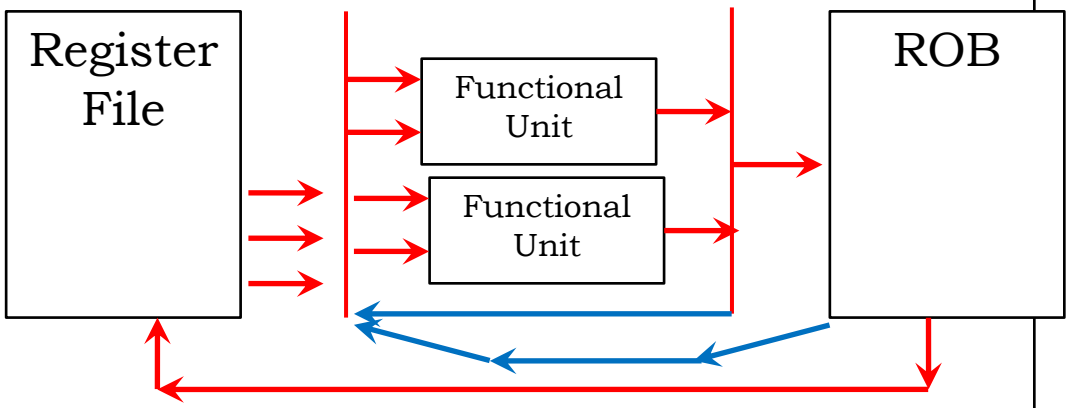
Must have the Destination register.
Must have value for Destination Register.

Must have PC (says which is the oldest instruction)

But suppose a value which is written in the ROB is wanted before it is architecturally visible

Get it directly from the ROB

ROB Modification



An extra data path from the ROB to the functional units (and of course the forwarding path from the output of the Funtional Units to their inputs (By passing)

The ROB is content addressable. Search by register ID

ROB Simplification

A value for the functional unit is assumed to be in the register.

Remember we are already using register renaming. It means the physical location of (say) register 2 depends on context.

Register 2 is part of the architectural state. **But** when you ask for the value of register 2, or ask to store a value in register 2, the physical location of register 2 varies.

So the register had a valid bit and if that is set to false when you access a register then what is in the register is the ROB entry that contains (or will contain the value of the register.

Renaming

The register ID is renamed to the re-order buffer entry that will hold the register's value.

Register ID -> ROB entry ID
Architectural register ID - > Physical register ID

Now the ROB entry ID is the "register ID"

Now there *appear* to be a large number of registers

Operations

IF – fetch the instruction. In order.

D – Access the register file or the ROB. Check if instruction can execute – are all the values present (may be in the ROB – if so not architectural
E – Instructions can complete out of orer

R – completion. Write result to ROB.

W – retirement/commit – if no exceptions write to an **architectural register file** (or memory). If exceptions flush pipeline. Start exception handler

Dispatch/Execution – *in order*
Completion – *out of order*
Retirement – *in order*

Simple principle and supports precise exceptions

Need to access ROB to get results not yet in register file – indirection means increased latency and complexity

Elegant

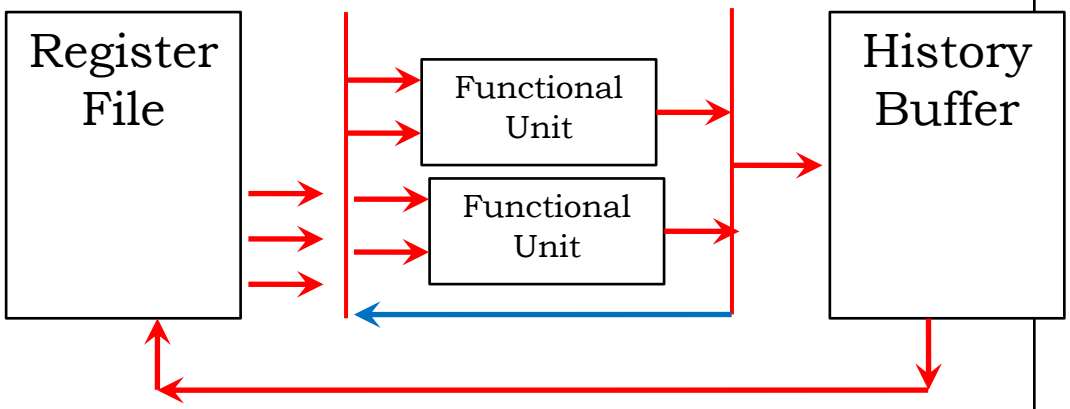
But

We are increasing the complexity of execution in order to take care of something which is not likely to occur *an exception*

Alternative: assume all is well and only take action when something goes wrong.

History Buffer

A similar buffer but with a different purpose



Only used for exceptions

On decode an entry is reserved in the history buffer.
On completion the **old** value of the destination is stored in the HB
When the instruction is the oldest the HB location simply becomes accessible for the system to write a value.

Dynamic Memory Scheduling

The order of execution is determined dynamically (and not by the compiler)

The reorder buffer solves the problems of instructions which take longer to complete and via renaming solves the problem of data dependencies, which are not true data dependencies.

Includes delays not predictable at compile time. Is and operand in cache?

But true data dependency stalls the dispatch of instructions to the functional unit –if you don't have the values you can't execute the instructions

Operation of an in-order pipeline in the presence of a data dependency:

False data dependency (register conflict)
allows the dispatch of younger instructions into execution units – ROB

True data dependency
Stalls the dispatch younger instructions to proceed

Problem of register v memory

MULL R1 <- R2, R3
ADD R4 <- R5, R1 *True dependency stall*

Any following commands cannot dispatch

LD R1 <- R2(0)
ADD R4 <- R5, R1 *True dependency stall*

We know how long we have to wait for the multiply to finish – but we don't know how long the load will take.

Is it from memory; Level 1 cache; Level 2 cache .. ?

Cannot tell ahead of time because of possibility of dynamic memory allocation

Preventing dispatch stalls

Have talked about compiler re-ordering. Which will work with MUL – fixed latency.

But LD has variable latency, so harder to do at compile time.

Can also do fine-grained multi-threading – but again difficult to in the presence of variable latency.

Variable latency is where out of order execution becomes valuable.

Only dispatch the instruction when the operands become available.
Dispatch subsequent instructions

How do you know when the operands become available?

Data flow architecture -
not Von Neuman – but exists on all (nearly all?) current high performance computers.

Move dependent instructions

Any instructions which are truly dependent are moved to an area to wait : *reservation stations*

Monitor the source values of each instruction. When all available dispatch instruction

No longer dispatch in control order, but now in data flow order.

Variable latency is no longer a problem – we just wait until all operands are available.
(*If too long stall will of course occur)*

Non dependent instructions can keep executing.

Requirements for “Out of order execution”

1.Need to link the instruction which is waiting for a value to the instruction which will produce it

No longer communicate through registers

1.Need to hold instructions out of the way of the execution stream until they have all their operands

2.Instructions need to keep track on the status of their source values

3.When all the source values are available the instruction needs to be re-inserted into the execution stream (sent to the functional unit)

Actions implementing the requirements

1.Associate a tag with each data value

register renaming

This is the same as using the ROB – associate the architectural register ID, with a ROB ID.

1.Reservation station. After renaming each instruction is placed in the reservation station.

Moving out of the way

1.When the value of a data item is available broadcast a tag.

The tag is a string which identifies the data value; simplest is to use the reservation station ID. Broadcast because more than one instruction may need the value. Instructions in the reservation station need to compare the tags, with the tag. on their required values

1.When all source values are ready, dispatch instruction(s) to appropriate functional units. If more than one instruction need to arbitrate on functional units.

Review of concepts

1.Register renaming means that only true dependencies are left and also allows a mechanism to link the results of calculations to the operations which require those results

1.The Reservation stations allows the pipeline to keep going bypassing dependent operations

1.The tag broadcast allows the creation of a data flow like architecture. Note that every place in the RS as well as every entry in the RAT needs a comparator to check the tag bit. This means that there is a significant increase in complexity

Imprecise

Imprecise Interrupts/Exceptions

Developed for 360/91.
Used in the 360/195

Computer stopped at this PC
Probably the problem is close to here.

Patterson comments
"*Not so popular with programmers*"
He can say that again!

Basically you do "imprecise debugging"

Taught me early that *debugging – running to a crash* . Is a bad way to develop code

Final stage of parallelism

CPI must be at least one, unless more than one instruction issued per clock cycle.
1.Superscalar processors
    a.Static
    b.Dynamic
2.VLIW – very long instruction Word

Reaching 4 instructions / clock

Out of order Execution