Getting higher performance

Greater performance (at a given price) sells computers.

- Technology **scaling**: happening anyway
  - 'Moore's Law' – double the transistor count every ~~two~~ three years
  - More transistors allow more speed-up techniques
  - Smaller transistors switch faster
- Higher clock rates: part technology, part **pipelining**
  - Subject of this lecture
- **Parallelism**: more functional units; more cores
  - **Superscalar** processing offers some single-thread speed up
  - Multicore processors offer multi-thread speed up (when applicable!)
    … and instantiating more processors is relatively easy in hardware

How to make processors faster

Increase the clock rate

- As a consequence of Moore's Law
- Through circuit design
- Through microarchitectural design
  - More parallelism: **pipelining**

Do more in each cycle

- Parallelism within instruction execution

Do more in each instruction

- Somewhat limited by ISA
  - Completely new ISA expensive – possibly unmarketable
  - Instructions added to speed up desirable operations which are slow in software

Do several instructions at once

- E.g. see facing page

Pipelining

Pipelining works by starting executing one instruction before its predecessor(s) have finished. Instructions 'chase' each other through the circuits. The time taken for a circuit to evaluate is a bit uncertain, especially as different signal paths tend to be different lengths, so instructions are deliberately separated into **stages** with pipeline **latches**. The latches prevent one operation from overrunning its predecessor (by delaying it) and are synchronised by the clock.

It is not always sensible (or possible) to insert a latch at exactly the optimum place so a pipeline may not be precisely *balanced*. The clock rate is limited by the slowest pipeline stage (the critical path). To go faster:

- Decrease the logic complexity per stage
- Increase the pipeline depth (number of stages)

Superscalar processors

This module is not going to delve into the details of superscalar issue. As you might imagine, this is another level of complexity. This page is included mostly for interest and to point out the where the problems are similar to those in pipelining.

Superscalar processors can issue more than one instruction at the same time. In principle, if two instructions can be executed simultaneously the processor can go twice as fast. However there are some practical issues.

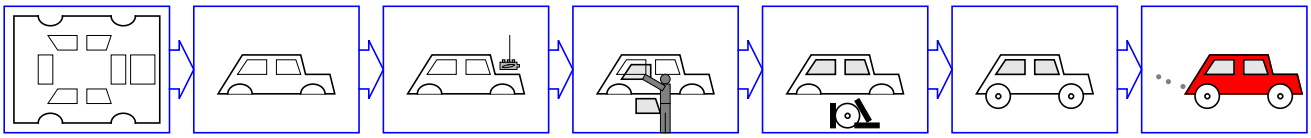
- More functional units are required
  - ALUs can be replicated reasonably easily
  - Ports to memory are hard to replicate
- Higher instruction fetch bandwidth required
  - Wider instruction bus
- Instructions are not independent; hazards (see later) occur
- **Widening** the pipeline increases the speculation depth
  - More to discard when branching
- Out-of-order issue desirable to find non-dependent instructions
  - More complication
- Out-of-order completion desirable if instructions out of order
  - Nightmare with exceptions!

Multicore

- Relatively easy from the hardware perspective: largely copy-and-paste. More complex when worrying about cache coherency
- Independent processors require independent threads which can stay active making useful contributions. This is a software problem. A very difficult software problem.

Pipelines

- A pipeline is a *production line.*
- Each worker/stage does a (simple fast) repetitive job
- Product/data is passed on to next worker/stage



Pipelines introduce **parallelism** at **small cost**

- Operational stages have to be present anyway
- Increase throughput – assuming that there are lots of identical (similar) products to process
- Rely on streaming data
- *Increase* latency – speed limited by slowest stage

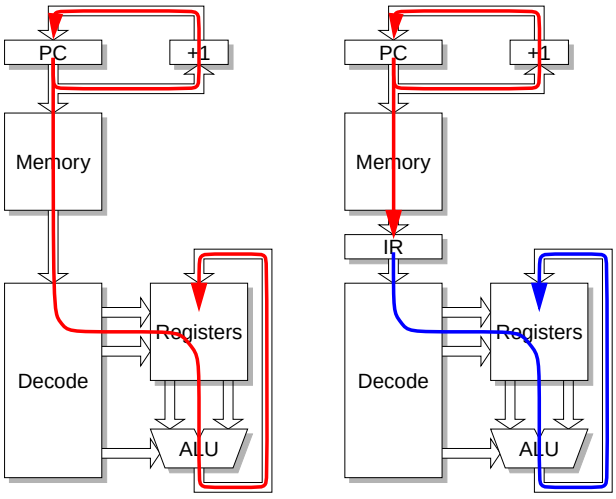
Processor instructions *almost* fit this category

Pipelining

First, design process so that each step only uses each resource once.

E.g. Stump design in lab. uses ALU to increment PC AND operate on data.

Insert another 'ALU' (need only be an incrementer) to avoid duplication (left-hand figure).



Unpipelined Pipelined

All the operation can now happen in a single, if rather long, cycle.

Split the operation (as before) into two cycles (as Stump) with a register in between (right-hand figure). Only the logic in one part would be active at any time, so run the operations *concurrently.* This is now a simple (two-stage) pipeline.

Worked example

This process can be extended quite easily as long as it is dealing with a *stream* of data. The 'obvious' next step is to add a register after the decoder to make a three stage pipeline.

The benefit of a pipeline is that it reduces the critical path so the unit can be clocked faster. As the pipeline can perform (up to) one operation per clock cycle, a faster clock will give a **higher throughput**.

A two-stage pipeline can be clocked at (almost) twice the speed of an unpipelined unit if the pipeline is **balanced**: i.e. the logic is 'cut' half way along the critical path so the two stages take the same amount of time. This is not always practical. In the example on the left, let's imagine the following delays:

Unit	Delay
Register propagation delay (inc. PC, IR)	3 ns
Register setup time (PC, IR, reg. bank)	2 ns
Incrementer delay	5 ns
Memory delay	20 ns
Decoder delay	10 ns
Register bank read time	5 ns
ALU delay	15 ns

PC increment time = 3 + 5 + 2 = 10 ns

PC to register write (unpipelined) = 3 + 20 + 10 + 5 + 15 + 2 = 55 ns

PC to IR = 3 + 20 + 2 = 25 ns

IR to register write = 3 + 10 + 5 + 15 + 2 = 35 ns

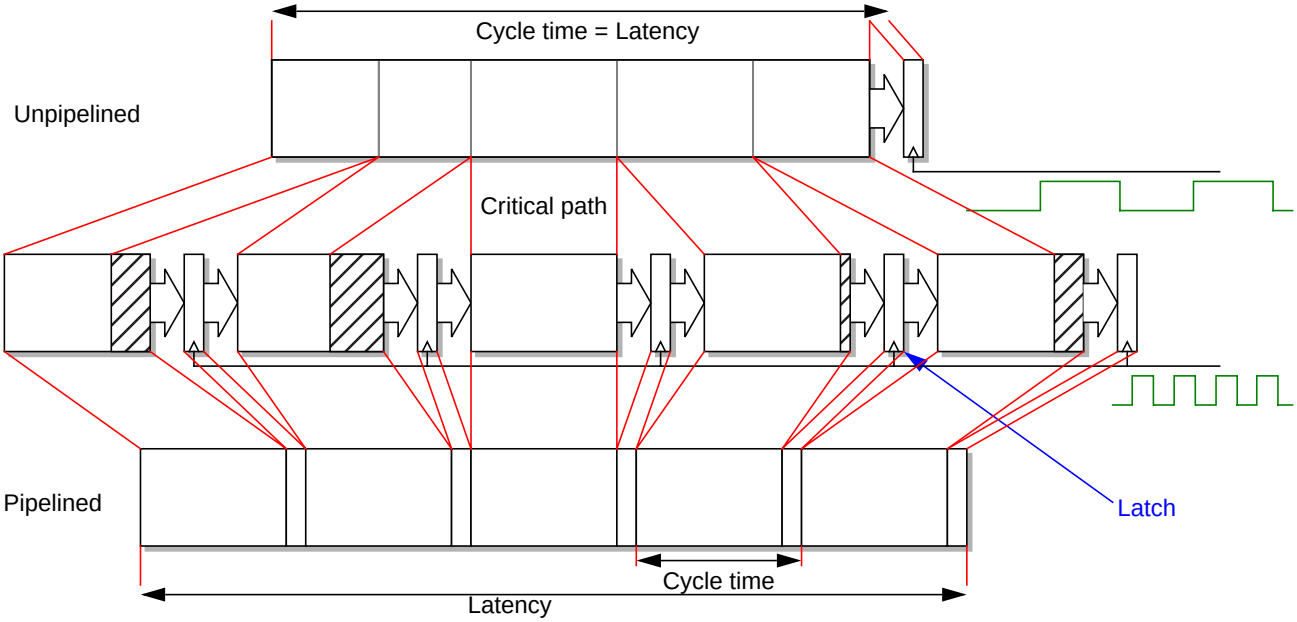
The unpipelined minimum cycle time is max(10 ns, 55 ns) = 55 ns

The pipelined minimum cycle time is max(max(10 ns, 25 ns), 35 ns) = 35 ns

Taking this example further, let's add a register between the decoder and the execution datapath to make a three-stage pipeline.

Potential speed-up

There is, inevitably, some overhead and some ‘dead’ time introduced:



- The speed-up will be less than N× and there is a sensible/practical upper limit to ‘N’.
- Achieving even this assumes that the pipeline flows continuously without problems

Potential speed-up

Partitioning a task into ‘N’ stages gives a maximum speed-up of ‘N’ times.

This assumes:

- All the stages take exactly the same time
  - Impractical to divide exactly evenly … therefore some ‘dead’ time
  - Harder to balance as ‘N’ increases
- The partitioning has no added cost
  - It does have cost: there is a delay imposed by the registers

Disadvantages of pipelining include:

- Increased latency. From the start, the first instruction will complete in:
  Unpipelined: 55 ns
  Two-stage: 2 × 35 ns = 70 ns
  Three-stage: 2 × 25 ns = 75 ns
- Additional latches – fairly minor penalty
- **Dependencies** if the pipeline is not linear

Worked example – continued

Taking this example further, let’s add a register between the decoder and the execution datapath to make a three-stage pipeline.

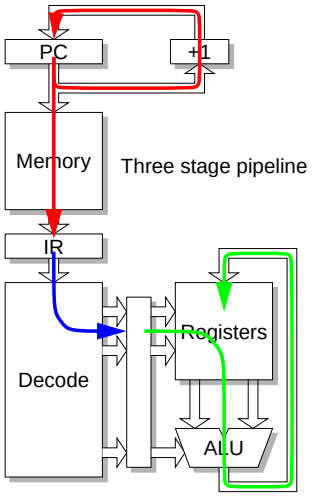
IR to decode reg. = 3 + 10 + 2 = 15 ns

Decode register to register write
= 3 + 5 + 15 + 2 = 25 ns

The three stage pipelined minimum cycle time is:
max(max(10 ns, 25 ns), 15 ns, 25 ns)
= 25 ns

Just over twice the speed of the unpipelined design.

As a pipeline gets deeper it gets harder to find appropriate places to insert registers and keep the balance, i.e. all stages roughly equal in delay.



Dependencies

The free flow of a pipeline depends on one instruction immediately following another on each cycle. This is straightforward if the data are all independent.

Microprocessor instructions are not always independent. One instruction often uses a result from its immediate (or near) predecessor. This **dependency** may require one instruction to be complete before another starts to execute.

The potential problems get worse if instructions may be executed in parallel or in a different order from the way they were written. These dangers are referred to as ‘**hazards**’ and must be alleviated for correct operation.

Memory Timing

In this, and all the examples given here it has been assumed that the memory response is a constant time (usually within one cycle). On a ‘fast’ processor this would be a cache access (in the top-level cache).

In practice there are cache misses (at various levels) which cause pipeline stalls (and, sometimes, more), reducing throughput. These are outside the scope of this module and have been ignored.

Control Hazards

Pipelines work on data streams – flow is predictable

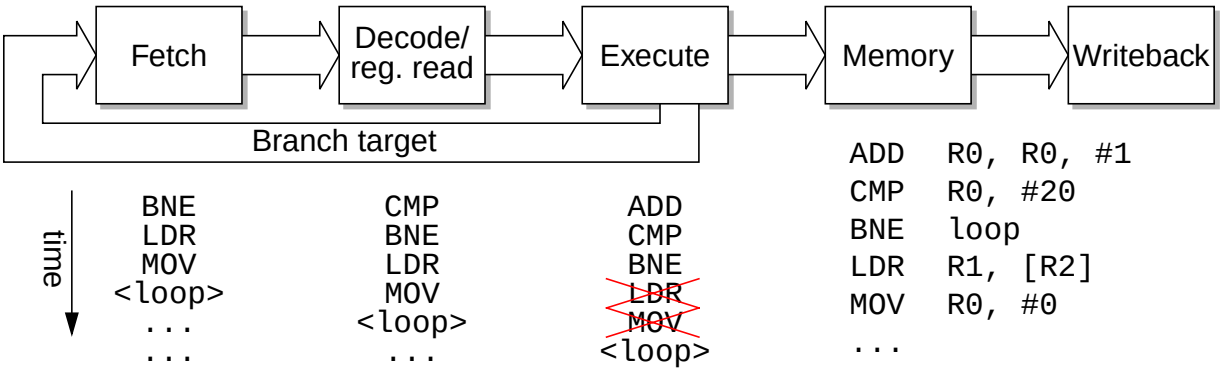
A processor instruction stream is predictable a lot of the time – but sometimes branches

Worse: conditional branch gives a fork in execution and it's not clear to the prefetch unit:

- that this is a branch instruction
- if it is, which way to fork

This is a **control hazard**.

The five-stage pipeline below calculates the branch target (and condition) in 'Execute'.



Latency is reduced by applying the branch as early as is feasible.

Pipeline performance

Measuring performance

What a user wants is maximum performance on a particular application. This depends on both the instruction throughput and the 'power' of the particular instruction set. (If an ISA has only very simple instructions it will require more operations to do the same thing as one with more complex instructions – providing the complex instructions are applicable.)

The instruction throughput is the product of the clock frequency and the average number of **Instructions Per Clock** (IPC); on a processor with one functional unit the latter cannot exceed 1, and will usually be lower due to various disruptions to the flow, such as multi-cycle operations or branches. This figure is quoted as '**MIPS**' (Million Instructions Per Second). The reciprocal – **Clocks Per Instruction** (CPI) is also sometimes quoted. CPI is probably more applicable to processors with a single execution unit where it must be at least 1.

Control hazards: effect on performance

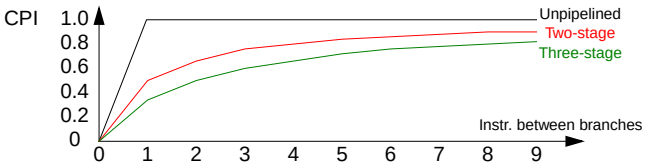
Control hazards means that the processor has fetched instructions which are not wanted. The pipeline must be flushed and refilled from the correct destination.

There are various solutions: the easiest (conceptually) is to let the prefetch unit keep going unless a branch is determined to exist and be taken, then inform the prefetch unit and **discard** the speculative fetches in between.

This is a pipeline **flush**: it carries a significant penalty in a deep pipeline because the *pipeline latency* must be paid for each time.

Cost of pipeline flushes

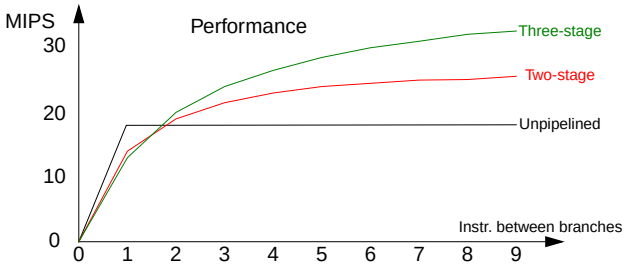
The penalty for pipeline flushes depends on how often they occur. In 'typical' microprocessor code a branch (or other control flow change) occurs roughly every six or seven instructions, on average.



Taking the throughput and latency figures from the earlier examples:

Branch every …	Unpipelined (18.2 MHz)		Two-stage (28.6 MHz)		Three-stage (40 MHz)	
	CPI	MIPS	CPI	MIPS	CPI	MIPS
2	2/2 = 1.00	18.2	2/3 = 0.67	19.0	2/4 = 0.50	20.0
3	3/3 = 1.00	18.2	3/4 = 0.75	21.4	3/5 = 0.60	24.0
4	4/4 = 1.00	18.2	4/5 = 0.80	22.9	4/6 = 0.67	26.7
5	5/5 = 1.00	18.2	5/6 = 0.83	23.8	5/7 = 0.71	28.6
6	6/6 = 1.00	18.2	6/7 = 0.86	24.5	6/8 = 0.75	30.0
7	7/7 = 1.00	18.2	7/8 = 0.88	25.0	7/9 = 0.78	31.1
8	8/8 = 1.00	18.2	8/9 = 0.89	25.4	8/10=0.80	32.0
9	9/9 = 1.00	18.2	9/10=0.90	25.7	9/11=0.82	32.7

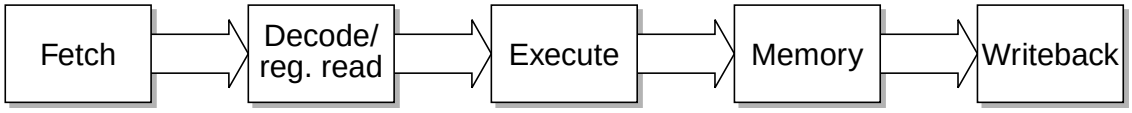
The *illustrative* example shows that, whilst the clock-rate increase from pipelining typically enables a speed up the cost of flushing the pipeline opposes this. Here, a three-stage pipeline is not quite twice as fast as the unpipelined unit.



A deeper pipeline suffers more from flushes as it takes more cycles to refill. The software *needs* to branch every so often. To alleviate the problem it is possible to predict (guess) when a branch may happen and fetch from the target instead. With a deep pipeline – some processors have exceeded 30 stages – it is essential to predict the future flow if the pipeline is ever to be full of wanted instructions.

Data Hazards

Considering a 'classic' five-stage pipeline




```
ADD   R0, R0, #1
CMP   R0, #20
```

This (ARM) code fragment uses only one register but it has two values.

It is vital that the CMP acts on the value *after* the ADD.

If the execution path is pipelined, the ADD may not have completed to the stage of writing back the result before CMP reads the R0 register.

This must not be allowed to happen.

Solutions:

- **Stall** the CMP until the ADD completes — Slows the whole pipeline down
- **Forward** the ADD result to the CMP input — Requires extra hardware/complexity
- **Defer** the CMP but issue a later instruction first — Out-of-order execution; Even more complex to track

The 'classic' five-stage pipeline

This has become a standard pipeline example and serves to illustrate the problems with **hazards** in a pipelined processor. The functions of the stages are:

- **Prefetch**: read an instruction from memory; increment the PC
- **Decode**: classify instruction and set up control; read register operands
- **Execute**: perform the ALU function; possibly an address calculation
- **Memory**: loads and stores to perform transfer; delay for internal ops.
- **Writeback**: 'retire' ALU result or loaded value to register bank

Some points to note:

- This does not follow a von Neumann architecture: there may be separate, parallel memory operations for instructions and data
- The result of each instruction takes several cycles to be written to the register file

Taking the second point, execute:

```
ADD   R0, R0, #1
CMP   R0, #20
```

When the 'ADD' is executing the 'CMP' has reached the decoder. If it reads 'R0' at this point it will get the value before the addition, i.e. the wrong value. Assuming (reasonably) that it's possible to read a register in the same cycle it is written, to make this work the programmer could write:

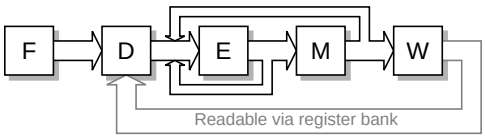
```
ADD   R0, R0, #1    ; When in 'writeback' …
NOP                 ; … this in 'memory' …
NOP                 ; … this in 'execute' …
CMP   R0, #20       ; … and this can now go
```

Some early RISCs did rely on the programmer (compiler) to do this. It is clearly a waste of fetch bandwidth – although non-dependent instructions can execute rather than 'NOP's.

It is not too difficult for the decoder to note the 'ADD' destination at issue and insert the 'NOP's automatically by **stalling** the issue for two clocks. Of course this leads to a significant performance penalty.

Forwarding

If the decoder is tracking the progress of potential results down the pipe it is sometimes possible to get them before they reach the register bank. This is called 'forwarding' and requires extra buses to bring results back up the pipeline. It is commonly employed to alleviate possible stalls.



Forwarding cannot eliminate all stalls. Consider the code:

```
LDR   R0, [R1, #offset]
CMP   R0, #20
```

One cycle after the 'LDR' has been issued it will occupy the 'execute' stage, having just completed the address calculation. The 'R0' value will not be available for another cycle (at the end of the 'memory' stage) so a one cycle stall is needed until the loaded value can be forwarded from there.

Instruction ordering

If a non-dependent instruction can be placed after a load it can be issued instead of stalling, and thus becomes 'free'. It is common for compilers (and enlightened assembler programmers) to try to shuffle instructions together to keep the pipeline busy. E.g.

```
LDR   R0, [R1, #offset] ;
MOV   R4, #0            ; Use 'stall' cycle
CMP   R0, #20           ;
```

Note that this is an *example*; deeper pipelines may, potentially, stall for more cycles. Instruction reordering to interleave operations further may improve performance more. It makes the object code harder to follow by hand, though!

Some (more sophisticated) processors will reorder instructions dynamically, in hardware ('**out-of-order** issue'); this is even more complex/expensive, especially if an **exception** occurs, but it does offer a performance increase.

Data Hazards

There are three named classes of hazard which must be guarded against in pipelined – and, particularly, concurrent and out-of-order – microarchitectures.

The important thing is that instructions *appear* to execute sequentially in the order in which they're written as far as the programmer (compiler) is concerned.

RAW – Read After Write

- An instruction tries to read a value before it has been updated
- Potential problem in any pipelined microarchitecture

WAR – Write After Read

- An instruction is delayed and one further on in the instruction stream which modifies one of its sources is issued
- Can only happen if instructions issued out-of-order

WAW – Write After Write

- Two instructions which write to the same destination are 'in flight'; they must complete in the correct order
- Can only happen if instructions issued out-of-order

Data Hazards

In a load-store architecture (like a RISC) hazards are typically associated with (architectural) registers. Hazards may also appear when dealing with memory – for instance if operations can happen out of order due to (e.g.) **write buffering**; this is effectively out of the scope of this module, however.

RAW – Read After Write

A read occurring after a write should return the value written. This can mean waiting for the value to be evaluated or loaded and, possibly, forwarding the value to make it available sooner than it would otherwise have been. RAW hazards start to appear when *evaluation* is pipelined.

RAW hazards can be mitigated by *stalling* (which is bad for performance) or *forwarding* (which has a hardware/complexity overhead).

WAR – Write After Read

If a write appears after a read in the code the read should return the value before the write. Because register reads happen at the start of evaluation this is guaranteed if the instructions are issued in-order. If instructions are issued out-of-order then the write may need to be delayed until all the 'preceding' reads have taken place. Alternatively, it may be possible to complete the write whilst retaining the 'old' value of the register as well, so the read can pick this up. This principle is employed in, for example, *register renaming*.

WAW – Write After Write

If two writes have the same destination then the eventual result should be that from the second-written instruction. With in-order execution this is guaranteed; if completion an occur out-of-order then some form of interlocking is needed.

Note that there could be a read in between the writes so two closely spaced successive writes are not always 'silly'.

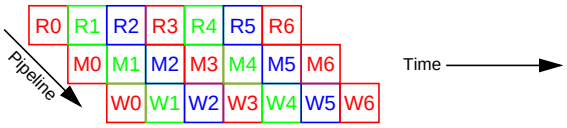
The problem

A microprocessor instruction goes through three basic phases:

- **Read** operands from registers or memory
- **Modify** data by executing some function
- **Write** back the result to a register or memory

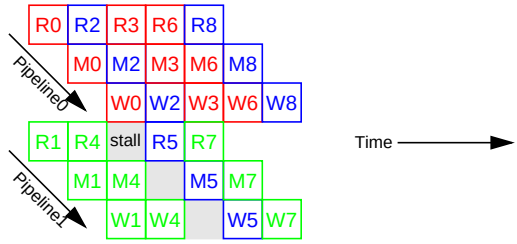
At object code level these three phases appear atomic, which guarantees completion (write) of one instruction before the start (read) of the next.

As instructions begin to be pipelined these automatic guarantees disappear.



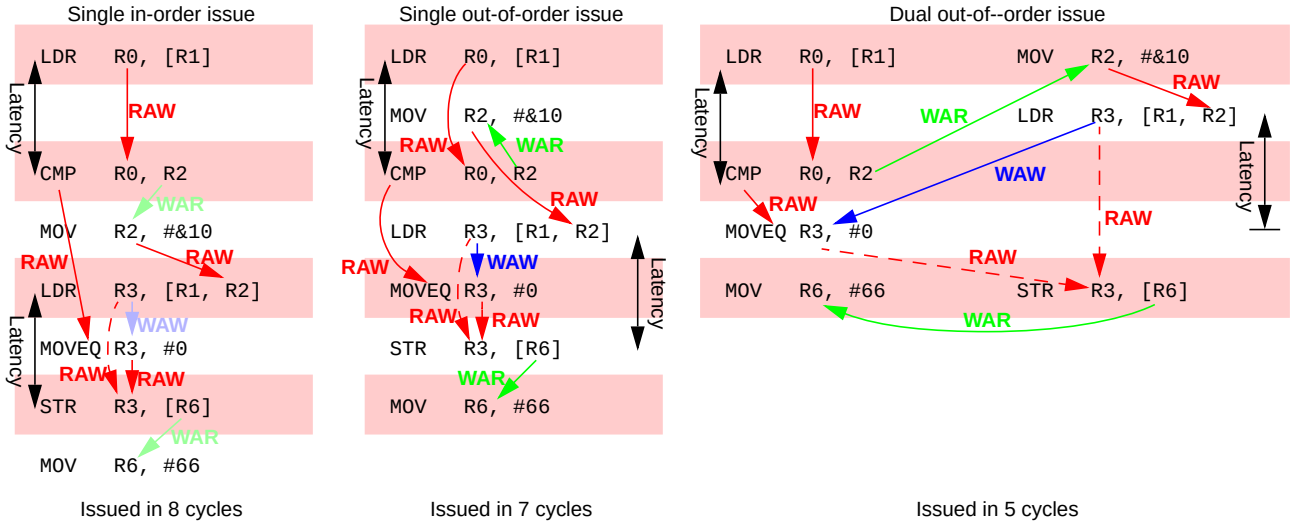
E.g. if R1 precedes W0 but *depends* on it, there will be a problem (RAW).

With instructions out of order and concurrent (in superscalar processors) things get messier: writes can get out of order (WAW) and, potentially happen 'too early' (WAR).



Data Hazards

```
LDR    R0, [R1]        ; Load - has latency
CMP    R0, R2          ; RAW - R0
MOV    R2, #&10        ; WAR - R2
LDR    R3, [R1, R2]    ; Load - has latency
MOVEQ  R3, #0          ; WAW - R3
STR    R3, [R6]        ; Correct value?
MOV    R6, #66         ; WAR - R6
```



Data Hazards

Single in-order issue

Here we are constrained as to what may be issued. The first instruction is a load. The subsequent instruction depends on the load (R0) and there is a cycle before the loaded value is returned and can be forwarded. The next cycle is therefore a stall. The CMP can then be issued and R0 is forwarded to the execute input.

The next MOV is straightforward and its result can be forwarded to the subsequent LDR for its address calculation.

There is no problem with writes to R3 because these happen in-order. There is a dependency on the 'EQ' predicate that the CMP has completed before its condition can be verified.

The STR must wait for the load latency but that is filled by the conditional MOV. The appropriate result can be forwarded to the STR.

The final MOV changes R6 but the STR will already have read that.

Single out-of-order issue

Whilst waiting for the first load, the CMP is stalled but the next MOV can be issued. This changes R2 which is used by the CMP – a WAR hazard. There are several solutions to this: one would be not to issue the CMP yet and stall. However the approach illustrated is both faster and more complicated (somehow) resolving the two different R2s so the CMP uses the original value whilst the newer one is picked up in the later address calculation. (It will still need forwarding!) The exact mechanism used is not important here.

Subsequent instructions can be issued in the written order although interlocks are necessary to protect against the WAW and subsequent WAR hazards in case they were not. Note that the R3 value to be stored is forwarded from one of two places depending on the conditional operation.

One cycle has been saved for considerable extra effort.

Dual out-of-order issue

If two instructions can be issued simultaneously there can be further speed-up at the price of greater complexity and, of course, another functional unit.

The first MOV can run in parallel with the first load, bearing in mind there is a WAR hazard to resolve. If there had been two LDRs here they could *not* have both been issued because there is (typically) only one port to the memory; the chance did not occur in this case.

The second load can now be issued – forwarding its 'new' R2 value whilst the CMP is still stalled waiting for data with the 'old' R2.

Because there could be two outstanding R3 values the WAW still needs resolution.

In this example the enforced sequentiality of the CMP-MOVEQ has left the second functional unit unemployed. The R6 value still offers a potential WAR hazard which (as shown) is not a problem because the read value would be used before the written-back one arrives.

An alternative would have been to issue the MOV R6 before the STR: this would be possible by keeping the 'old' and 'new' registers logically separate.

Register Renaming

[Really a bit advanced for this module, so just a brief mention.]

In register renaming there are typically more physical than logical registers. E.g. an ARM with 15 register names {R0-R14} (the PC is a bit different) could have (say) 25 registers. Logic keeps track of which register has which name at every stage of execution. Each time a register is written to a new (spare) register is given that name for the future. Thus an 'old' and 'new' R2 is possible. When nothing is left waiting for an 'old' register it becomes free for reallocation, thus running out of registers is rarely, if ever, a limitation.

Out-of-order execution – more problems

Two cases:

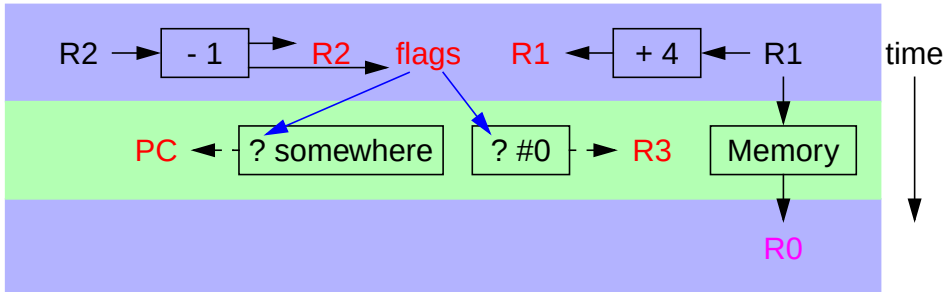
- Issue instructions out of order
- Issue in order but allow *completion* out of order
  - Example: don't stall waiting for loaded data

To illustrate the class of problem:

```
LDR    R0, [R1], #4   ; Load R0 and modify R1
SUBS   R2, R2, #1     ; Modify R2; set flags
MOVNE  R3, #0         ; Maybe overwrite R3
BNE    somewhere      ; Make a control decision
```

With superscalar issue:

flag dependency resolved



Problem: if the load aborts the processor will need to stop, run a recovery routine to find the appropriate page and restart. But what about the preceding state changes?

Exceptions

'Exception' is a blanket term for things which disrupt the code flow, typically, but not always, 'unexpectedly'. Some exceptions are caused by instructions, others by outside influenced such as interrupts.

An interrupt acts like a 'call' ('BL' in ARM terminology), invoking an (operating system) routine and returning to the point of interruption

The most common model – largely because it's easiest to deal with – is to think that, when an exception occurs, all the instructions up to some point in the instruction stream have executed and none has beyond that point. This means that, on return, the code can resume by jumping to a particular instruction.

If instructions are issued out of order then it becomes more difficult to determine what this 'point' is. However if this model is not observed it would be necessary to record a lot more state to ensure that, on return, only the 'missing' instructions were reissued.

All this can be fairly straightforward if the exception (like a 'call') is known about early on as it can be committed to like any other branch. (Note, however, that, by definition, an interrupt is not *predictable*.)

The 'nasty' case is a memory abort (i.e. page fault etc.) because these are only detected late in the pipeline.

```
LDR   R0, [R1], #4    ; Load R0 and modify R1
SUBS  R2, R2, #1      ; Modify R2; set flags
MOVNE R3, #0          ; Maybe overwrite R3
BNE   somewhere       ; Make a control decision
```

This code looks quite straightforward and the loaded `R0` can be deferred because it is not immediately used. However if the load is discovered to have failed at a later time it is important that the `PC` is brought back to this code and `R2` has its original value. There are various structures and techniques which are used to deal with these problems including **reorder buffers**, **history buffers**, **register renaming** …

The details of these is beyond the scope of this module but you might like to investigate out of interest.

Structural Hazards

Another class of 'hazards' that is sometimes discussed, a 'structural hazard' simply means there is no unit capable of doing a particular job (in a given cycle). This generally applies to multiple instruction issue machines.

A typical example would be memory operations in a dual-issue processor: two instructions can be issued in a cycle unless they are both memory {load, store} operations when there is only a single memory bus.

This could cause a stall unless another (out-of-order) instruction can be found which does not need the memory 'structure'.

This may also apply to other, non-duplicated units {multiplier, FPU…}.