Chapter

Pipe Line Hazards

Pipeline Stalls

Return to the central problem.

How to keep the pipeline full and moving.

There are three things internal to the pipeline which cause problems – these are referred to as **hazards**

The other thing is external to the pipeline and is related to fetching both the instructions and the data from the place they are stored. This may be disk – or it may be **main memory**

3 Memory stalls

Getting instructions

The instruction fetch assumes that you can get a fresh instruction every cycle.

That implies that the location of the instruction can be accessed with a latency of less than 1 cycle.

Any data wanted for the instruction must be similarly transferable to the data registers in again 1 cycle.
(Data slightly less crucial – data is not normally wanted every cycle).

Memory does exist which allows access in a single machine cycle and this is known as cache memory.

Cache

Memory which can be accessed in one cycle exists

Why not have all memory fast memory?
It is expensive

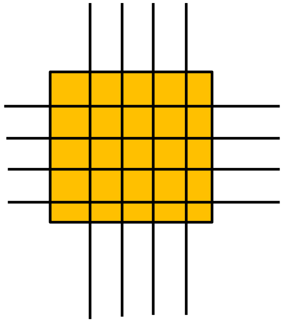
But for high performance machines?
It takes up more space on the chip

In addition
There are other ways of using that space which have a greater impact on performance.

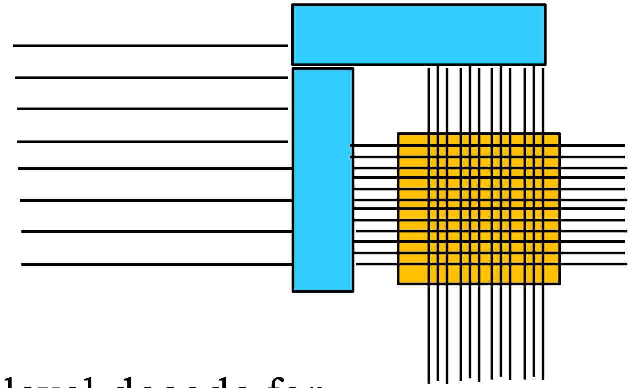
The larger the amount of memory that you have the harder it is to transfer in one cycle.

Direct access

Suppose you have 8 address lines …
you can access 16 locations



Of course with 8 bits you can have 256 addresses, but then you have to decode the address.
Decoding takes time



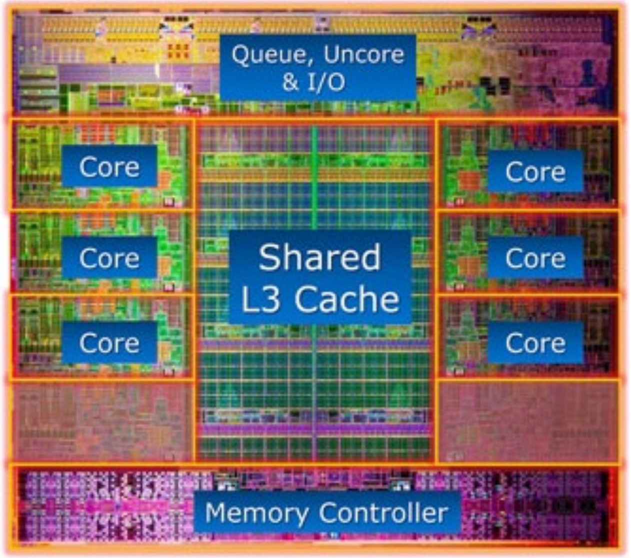
A single level decode for
4 GB is 65000 by 65000 lines

6 Addressing memory

Space

If you look at a modern chip it is already dominated by memory area.

Remember every core will have local L1 and L2 Cache



The performance balance between size and speed is not all at the fastest.

Streaming

Finding the route to correct memory location in more than 4GB takes time.

Once the address is set up it is much quicker to access the next word.

If you design your chip correctly you can even get the chip to do it automatically.

Ask for a word – return a string of words one after another – the transfer only taking a small number of clock cycles, even down to 1.

Another advantaged – suppose we transfer 256 words.

0	0	1	0	1	1	1	0	1	1	0	1	1	0	0	0

Hazards

Data blocks

0	0	1	0	1	1	1	0	1	1	0	1	1	0	0	0

16 bit word – 65536 addresses. (64K memory)

Simple demux means 256 by 256 = 512 lines to access a word.

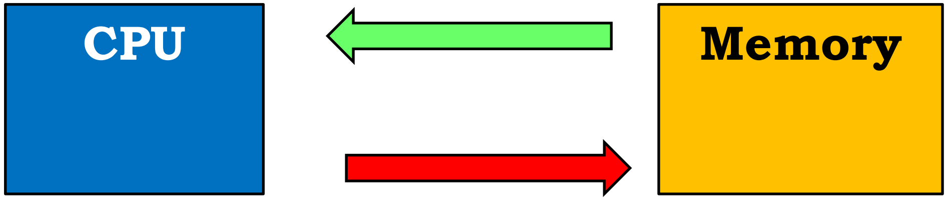
Divide memory into blocks of 256 words.
Now only have 256 blocks to address which fit into a 16 by 16 square array – so need only 32 lines to directly address.
So by blocking initial access is also faster

Finding the block address from the word address is just a bitwise **or** with the address.

0	0	1	0	1	1	1	0								

Pre-fetch

Fetching 256 words takes (256 + setup) cycles
Only useful is those words are required.



Suppose we transfer the word we require.
(Setup + 1) cycle.

Next instruction is not in the block being transferred.
We now have to wait for the block to transfer before starting the next recovery

Might abort the transfer - this will take some time and complicated the wiring.
Might make the memory multi-port – more than one request serviced at a time. Extra complication.

Multi-port memory is actually used.

Extra complication normally means: space; power; and time.

10 Locality

Locality

For the programme we know that the standard action of the Programme Counter **PC** is indeed to add one.

This is called locality. In particular the next instruction is likely to be near the current one.
Spatial Locality

Even better if you are executing any sort of loop you are likely to want to re-use that instruction.
Temporal Locality

But it is also true if you access a data item you are likely to want to access a nearby data item soon after.

Working through an array for instance.
Measurements of real programmes demonstrate that indeed they exhibit a high degree of temporal and spatial locality and the whole idea of caches is based on this observation.

Multi-port memory is actually used.

Hazards

Cache

Caches are a vital part of getting a modern processor to work efficiently.

How much cache
How is it arranged
Why do we have multi-level caches
What happens when a value in cache is changed
When do we eject existing data from the cache

Deal with this later.

Hazards

Hazards

Pipeline Hazards

Three types of problems associated with pipelining.

Structural Hazards
Caused by resource conflicts, where two different instructions (execution overlapped) want to use the same piece of hardware.

Data Hazards
Where an operand is not available when needed.
The result of a previous unfinished instruction.

Control Hazards
Caused by jumps and branches.
A jump means that the subsequent expression will not be executed, but that cannot be told until the instruction has been decoded.
A branch means the subsequent expression may or may not be executed and what happens cannot be determined until some operand has been evaluated

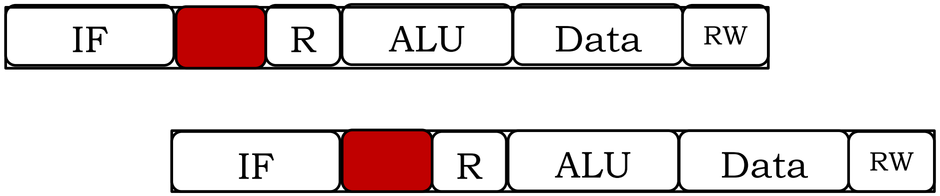
All need to be identified and corrected.

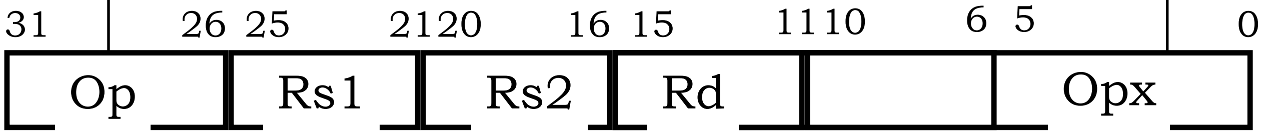
Effective action determines the success of the pipeline

Pipeline Hazards

Pipeline speed requires balanced design.

An instructions must enter the pipeline every cycle.
If only a few instructions are pipelined then the efficiency falls.





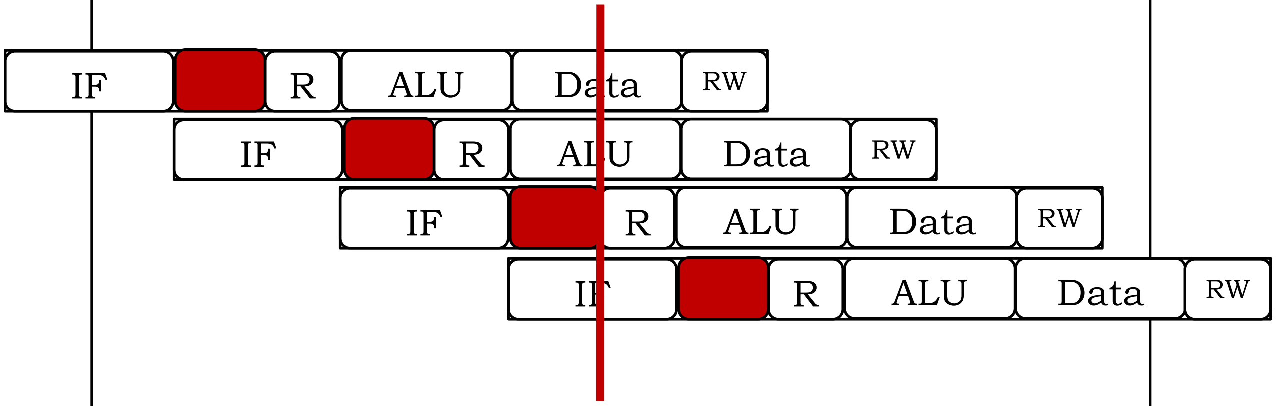
Fixed field decoding means registers can be decoded at the same time as the operation – if the operation does not require registers no harm is done.

A general principle

Guess the outcome if it can be done without penalty

Structural Hazards

If the hardware will not allow a certain set of actions.
Consider the pipeline with 4 instructions



Instruction 1 is (may be) doing a data access (read or write)
Instruction 4 is doing an instruction fetch.

More likely cache

Both require accesses to memory.
If the memory only allows one access (*simplest*)

⇒ Instruction 4 must wait a cycle for instruction 1 to complete the access

This is called **stalling the pipeline** and the speedup will drop from

$$\frac{4*800}{900+4*200} = 1.88 \quad \text{to} \quad \frac{4*800}{900+4*200 + 200} = 1.68$$

Lost 10%

Structural Hazards

Many structural hazards can be solved by increasing or duplicating some hardware.

PC counter needs an adder
The instruction may be to add two registers.
Give the system two adders.

The PC normally has its own dedicated adder, but modern multi-core machines will have multiple functional units to add, multiply and divide.

The use of multiple functional units goes back to the super computers of the 1960’s

The read/write problem can be solved by **multi-port memory** which allows most than one read/write operation to occur together.

Some modern multi-core chips have multiple functional units which are shared by the cores.
Why share Level 3 cache but not Level 1.

In general – add hardware.

Data Dependence

Instructions such as add/multiply need to take values from memory and move them to a register.

Usually high level languages do not allow the programmer to place the variables in a particular register.

The compiler will decide which register to put the source and destination data.
There are many variables and not many registers.

Some values in the registers are intermediate and are never reused.

C= A + B
D = E*C
F = A/C
etc.

C is never used again.

Register reuse

```
C= A + B             $R3 = $R1 + $R2
D = E*C              $R4 = $R2*$R3
F = A/C              $R2 = $R1/$R3
if (F ==13.0)         bnz $R2, dest
   go to <label>
etc.
```

C is never used again.

It would be silly to transfer the new value from register 3 to a place in memory. It would take time **and** bandwidth

It would also be silly to transfer the value in $R1 back to the location A, unless it is updated.

Now $R2 contains the value from B in the first line and the value from E in the second line.

It is certainly a **hazard**.

Is this a problem?

False dependence

Register rename

C= A + B	$R3 = $R1 + **$R2**
D = E*C	$R4 = $R2*$R3
F = A/Z	**$R2** = $R1/$R5
if (F ==13.0)	bnz $R2, dest
go to <label>	
etc.	

What is the connection between C and F.

None they have just been put in the same register. This is a hazard which may be solved by **register renaming.**

In other words putting one of the variables in a different register.

This is a *write after read* hazard. WAR

The hazard is due to the placement of the C and F by the compiler and can be solved by using a different placement.

True dependence

Data Dependence

C= A + B	$R3 = $R1 + **$R2**
D = E*C	$R4 = **$R2***$R3
F = A/Z	**$R2** = $R1/$R5

True dependence
Output 1 is input of 2

Inst2 is said to be **data dependent** on *Inst1* when *2* reads data written by *1*
Without knowing the value of C we cannot calculate D.
Register renaming does **not** solve. Data must be transferred.

Inst2 is said to be **data antidependent** on *Inst1* when *2* writes into a location read from by *1*

Not true dependence


Inst1
Register/ Memory
Inst2

Read after Write or **RAW**
Register rename or store from 1 before writing 2

Hazards

Data Dependence

Inst2 is said to be **output dependent** on *Inst1* when *2* writes data to the same place as *1*


Inst1
Register/ Memory
Inst2

Not true dependence

This is Write after Write **WAW**
Soluble by **register renaming** or by storing the register value in memory before the second write.

Remember only a small number of registers and so renaming may not be possible in every case.
Stalling is always possible

True and False Dependences

WAR, RAR, RAW.

But what about when there is a true dependence. One thing is for the compiler to **reorder** the instructions.

C= A + B	$R3 = $R1 + **$R2**
D = E*C	$R4 = **$R2***$R3
F = A/Z	$R6 = $R1/$R5
G = B*B	

Becomes

C= A + B	$R3 = $R1 + **$R2**
F = A/ Z	$R6 = $R1/$R5
G = B*B	
D = E*C	$R4 = **$R2***$R3

renamed

Compiler re-ordering

In the case where there is a true data dependence the compiler can move instructions around.

As long as it does not affect the calculation

Showed moving instructions back, but obviously also move instructions forward.

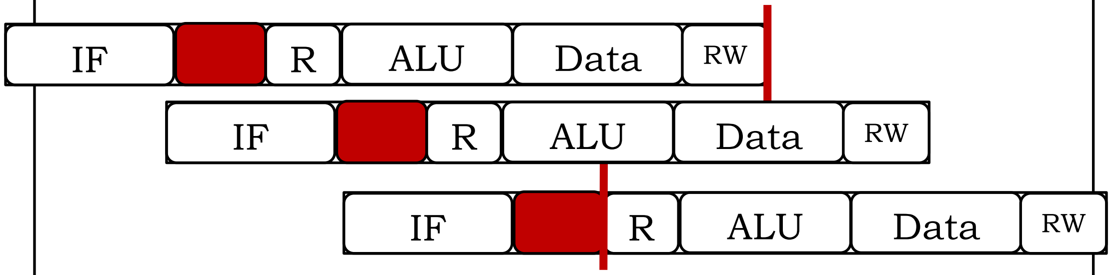
Will also work for false dependencies, if there are not enough registers for renaming.

What happened if Z=0 and F = A/Z therefore throws and exception.
The code says that D = E*C has been executed.
But it hasn't.
This is called an **imprecise exception** and we will come back to it.

Data Hazards

Data hazards exist when the result of instruction is required by an instruction in the pipeline

add **$s0**, $t0, $t1
add $s3, $t3, $t2
add $s2, **$s0**, $s1



f the write occurs in the first 100ps and the read is done at the end of the second stage 200ps.

⇒Need to stall for 2 cycles.
⇒Efficiency is

$$\frac{4*800}{900+4*200} = 1.88 \quad \text{to} \quad \frac{4*800}{900+4*200+2*200} = 1.52/$$

Lost 20%
Actually the result is at the output of the ALU at the end of stage 3.

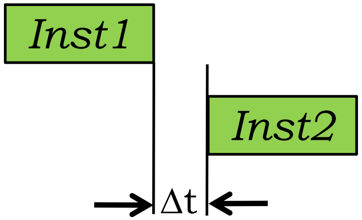
Introduce new hardware to feed the output of the ALU back into the input.
No stall required.

This is called **forwarding** or **bypassing**

So exploiting the lack of balance

Control Dependence

Inst2 is said to be **control dependent** on *Inst1* if 1 must complete before we know if *2* is to be executed.



$\Delta t > 0$

if (F ==13.0)   bnz $R2, dest

We have to know the outcome of the test before we can decide even what the correct value of the PC.

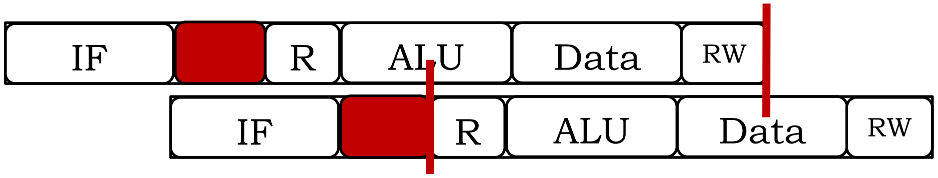
This is the most serious of the hazards so far. Simple re-order will not solve it.

Control Hazards

Make a decision on the basis of an unfinished instruction – **control hazard**

bneq

Next command



The second cycle reads the register. Suppose we read the register on the first half of the cycle and allow a test on the second part.

Branches are 13% of instructions Specint2000

Extra hardware may allow us to calculate addresses and load the PC on the second stage.

Still have to stall for 1 cycle. May not be able to resolve on the second stage even with these complications

When coding branches should normaly be untaken

Predict: assume the branch will not be taken.

If correct **Untaken**
⇒No need to stall.

If incorrect
⇒Stall

No penalty for a wrong guess

Control Hazards

Branch prediction

Decision based on context
branches at the end of loops are normally taken

Dynamic prediction:
look at what is happening and make a prediction.

Need to keep history, can lead to prediction with 90% accuracy

Problem
When the prediction is wrong, partially started instructions must have no long term effect.

The longer the pipeline, the worse the problem.

Delayed Branch

Move a instruction which does not affect the branch until after the branch.
a)the branch is loaded
b)the extra instruction is loaded
c)The PC result of the branch is available.

The delay is hidden.
Only used to defer for a single instruction.
Useful for short branches.

Data Fetch
PC or data – *cache*

Structural Hazard
More hardware

Data Hazard
Rename – reorder. *Out of order execution*

Control Hazard
Delayed Branch – *branch prediction*

Out of order execution and branch prediction and cache – we need to look at in more detail.