Australian National University

You know how aunt Dorothy knew something was going to happen whenever her truss got warm? Well she had nothing on

Embedded Processors

ENGN8537

Embedded Systems

Overview

- CISC, RISC
- Pipelining
- Branch Prediction
- Memory and Cache
- Interrupts and Exception

CISC vs RISC

Complex Instruction Set Computer vs Reduced Instruction Set Computer.

These two terms reflect two different philosophies in the design of a processor's instruction set and, in turn, the processor architecture.

Before the 1970s, all computers were of CISC type. Memory was the supremely limited resource of the time, hardware wasn't so much of a problem, so the focus was on packing the maximum amount of complexity in to the minimum number of instructions.

Memory is now relatively cheap and the focus has moved to performance. This favours the RISC style as simpler instructions are faster to execute

CISC

CISC style is characterized by:

- Complex Addressing modes
- Instructions that span multiple (and variable numbers) of words
- A large instruction set where specialized instructions implement particular functions
- Arithmetic and Logic operations that can be performed directly on memory locations as well as registers
- Memory-to-memory transfers with a single instruction
- Small program sizes

RISC

RISC style is characterized by:

- Simple addressing modes
- Single word, fixed length instructions
- Small instruction set of general purpose instructions
- Arithmetic and logic operations can only be performed on registers
- All memory accesses though a Load/Store architecture; i.e. no instruction has both input and output in memory
- Instructions designed to be implemented in (the same set of) discrete phases, lends themselves to Pipelining (to come).
- Programs larger in size because more instructions are required to implement comparable functionality

Hybrid Designs

Since the quantum shift from CISC to RISC styles in the 1970’s, the industry has been slowly incorporating more and more CISC-style functionality in order to reduce the number of instructions executed. This only works if the newly introduced instructions are still fast to execute. A simple example of this is single-instruction floating point operations which, by any definition, are complex operations but may be supported by dedicated FPU hardware to keep the speed up and make them fit the RISC paradigm.

Hybrid Designs

Another option for hybrid designs are processors that never really moved out of the CISC era.

x86 instructions are CISC in nature, however they're executed in a RISC core. This is done through "microinstructions" (aka microoperations, uops). The processor fetches several instructions from memory per clock cycle and dynamically translates these in to a sequence of uops. The uops are executed, potentially in parallel, then their results are re-assembled on the other side. This introduces significant extra complexity to a fully-RISC solution but maintains backwards compatibility with existing code and can actually provide the best of both worlds: Minimum program memory footprint combined with fast, pipelined execution.

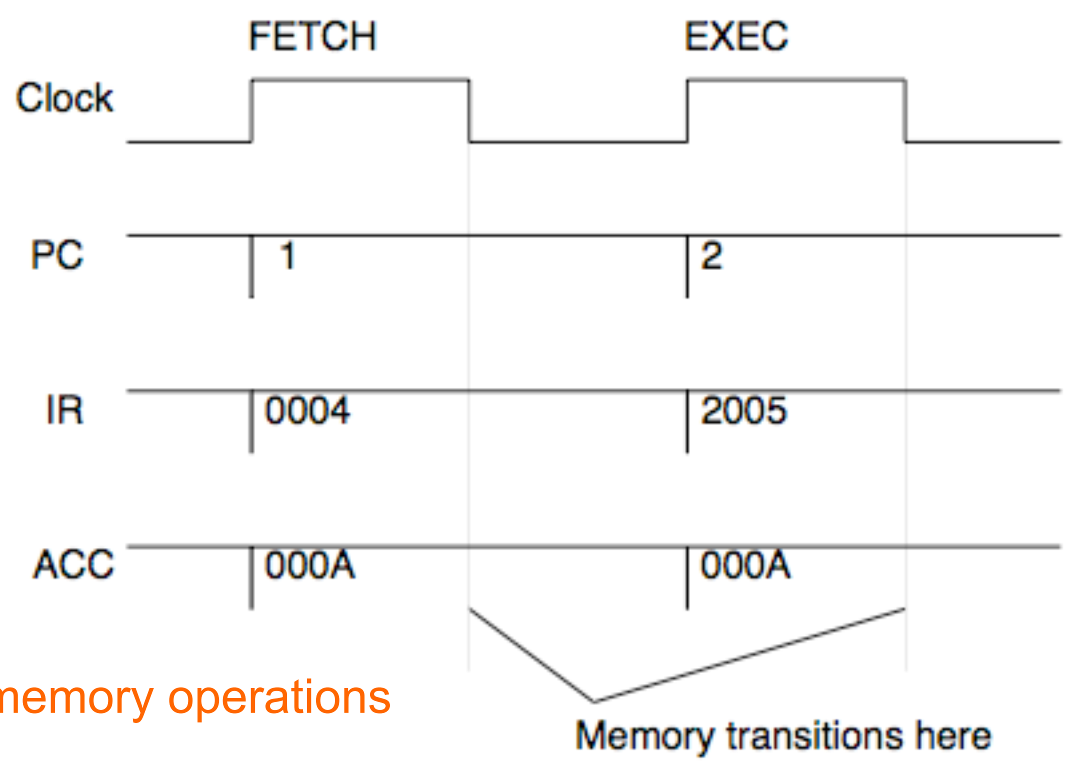
Pipelines

In the preceding slides, we have asserted that RISC style processors are better for pipelining, but what is pipelining and why do we want it?

Here we see the MU0 execution flow. It completes in two clock cycles but implements four operations:

- Fetch
- Decode
- Execute
- Write back

Data memory operations

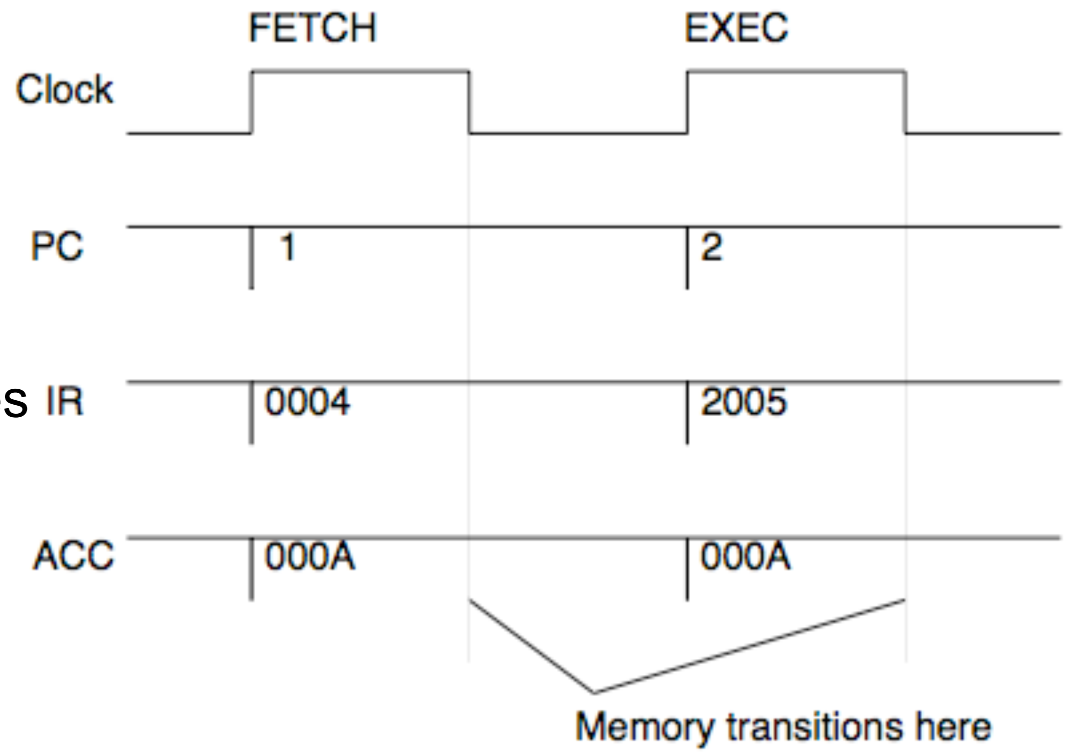


Pipelines

This is a variation on the canonical RISC operation set:

- Fetch
- Decode
- Execute
- Memory
- Write

Canonical RISC machines take 5 clock cycles to execute an instruction.



Pipelines

- Fetch
  - Get the instruction out of memory
- Decode
  - Break down the fetched instruction and read source registers
- Execute
  - Fairly self-explanatory! Do the work.
- Memory
  - Do a memory access, either read or write depending on the operation
- Write
  - Write the result back to a destination register

Pipelines

Example:

ADD r2, r2, r3 r2 ← r2 + r3

FETCH:	Load instruction from memory
DECODE:	Read R2, R3
EXECUTE:	Perform addition
MEMORY:	Do nothing
WRITE:	Put ALU output in to R2

Pipelines

Example:

LD r2, r3	r2 ← [r3]

FETCH:	Load instruction from memory
DECODE:	Read R3
EXECUTE:	Do nothing
MEMORY:	Load data from memory
WRITE:	Put memory value in R2

Pipelines

Example:

ST r2, r3	[r2] ← r3
FETCH:	Load instruction from memory
DECODE:	Read R2, R3
EXECUTE:	Do nothing
MEMORY:	Store data in to memory
WRITE:	Do nothing

Pipelines

Many instructions don’t use all five stages, so why do they do them?
Why doesn’t an add jump straight from the computation to write back?
Because all modern processors PIPELINE.

Instruction \ Clock Cycle	1	2	3	4	5	6	7	8	9	10
1	F	DC	EX	ME	WB					
2		F	DC	EX	ME	WB				
3			F	DC	EX	ME	WB			
4				F	DC	EX	ME	WB		
5					F	DC	EX	ME	WB	
6						F	DC	EX	ME	WB

Pipelines

The core observation of pipelining is this: Each instruction must take five clock cycles to complete, but you may execute more than one instruction at a time!

By breaking each instruction in to *exactly* the same number of pieces like this, each section of the processor can be fully utilized at every point in time.

We can answer our question from the last slide: Why keep the ME or WB stages for instructions that don't need them? Because keeping instructions the same as each each other, they can be pipelined and the *overall* utilization increases significantly.

F	DC	EX	ME	WB					
	F	DC	EX	ME	WB				
		F	DC	EX	ME	WB			
			F	DC	EX	ME	WB		
				F	DC	EX	ME	WB	
					F	DC	EX	ME	WB

Pipelines

Catch 1: Data Hazards

ADD R2, R3, #100	F	DC	EX	ME	WB				
SUB R9, R2, #30		F	DC	DC	DC	DC	EX	ME	WB

The Decode stage is where data from the register file is loaded in to the ALU, but what if the output of one instruction is the input for the subsequent one?

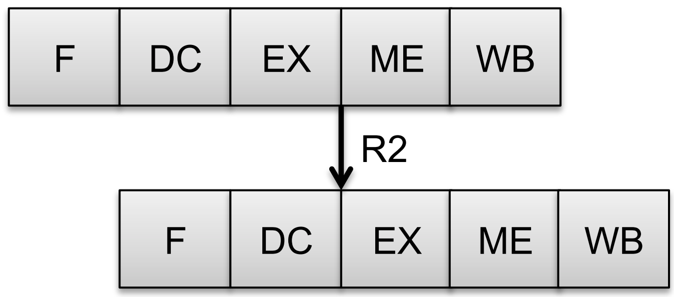
In the above example, the SUB instruction has to stall at the decode stage for three extra clock cycles before the write back of the ADD has updated the register.

Pipelines

Catch 1: Data Hazards

ADD R2, R3, #100

SUB R9, R2, #30



The solution is Operand Forwarding, where the control path logic recognizes the potential for a stall and “short circuits” the result from the output of the EX stage back to its input ready for the SUB, without waiting for the whole write back/decode read cycle.

This is a hardware solution, but if the hardware budget it tight, there’s a software solution: Get the compiler to re-order instructions and make sure the dependent instructions are never in the pipeline together. You don’t get the result faster, but at least you don’t waste the intermediate cycles.

Pipelines

Catch 2: Memory Delays

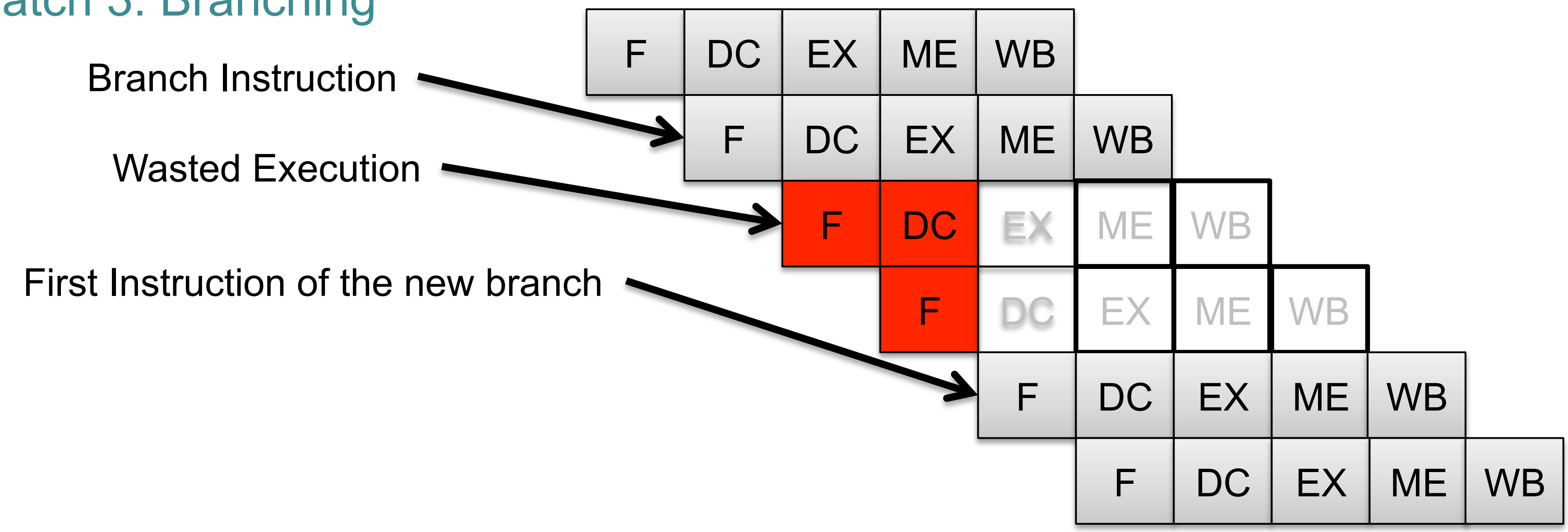
LDI R2, R3	F	DC	EX	ME	ME	ME	ME	WB	
SUB R9, R2, #30		F	DC	EX	EX	EX	EX	ME	WB

Access to cache memory can typically be achieved in a single clock cycle. If the memory access has to go out to main memory, the pipeline may have to stall for 10s of clock cycles.

We will talk about this more when we talk about caches later in the lecture.

Pipelines

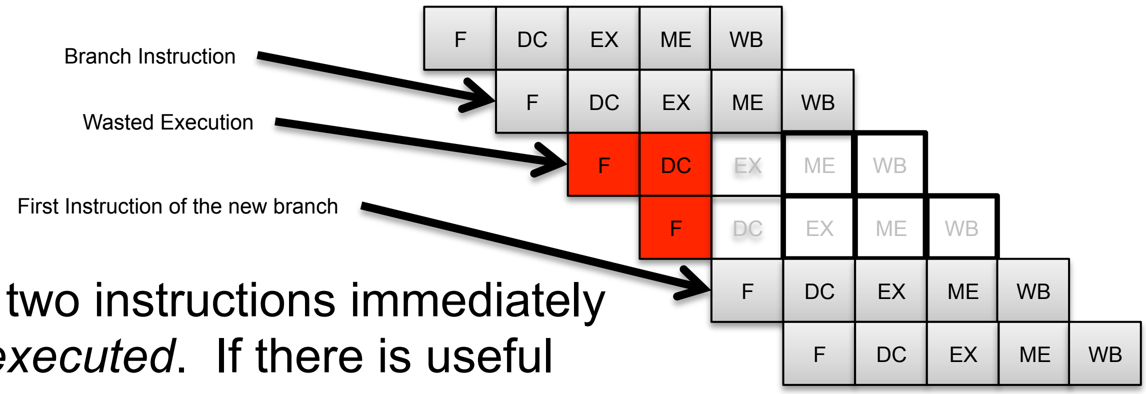
Catch 3: Branching



If one of the instructions is a branch instruction, it will render any instructions that have already been fetched and/or partially executed irrelevant. The penalty of this depends on how late in the pipeline the CPU can determine whether or not the branch is taken. In the example above, the branch condition is tested at Execute time so the decoded instruction after it, and the fetched instruction after that, have to be removed from the pipeline.

Pipelines

Catch 3: Branching



Three common solutions:

Branch Delay Slot. The one or two instructions immediately following a branch are *always executed*. If there is useful work that can be done that doesn't affect the outcome of the branch, it can be inserted in to this slot. If there isn't any work, the compiler must insert NOP instructions.

The second solution (well, attempt at mitigation), is Branch Prediction. BP is implemented on almost all modern medium and large processors and the effectiveness of the BP algorithm is often a key determinant of overall performance.

Lastly, most compilers will try and work out which branch is going to be taken *most often* and arrange the code such that this is the fast path through the pipeline.

Pipelines

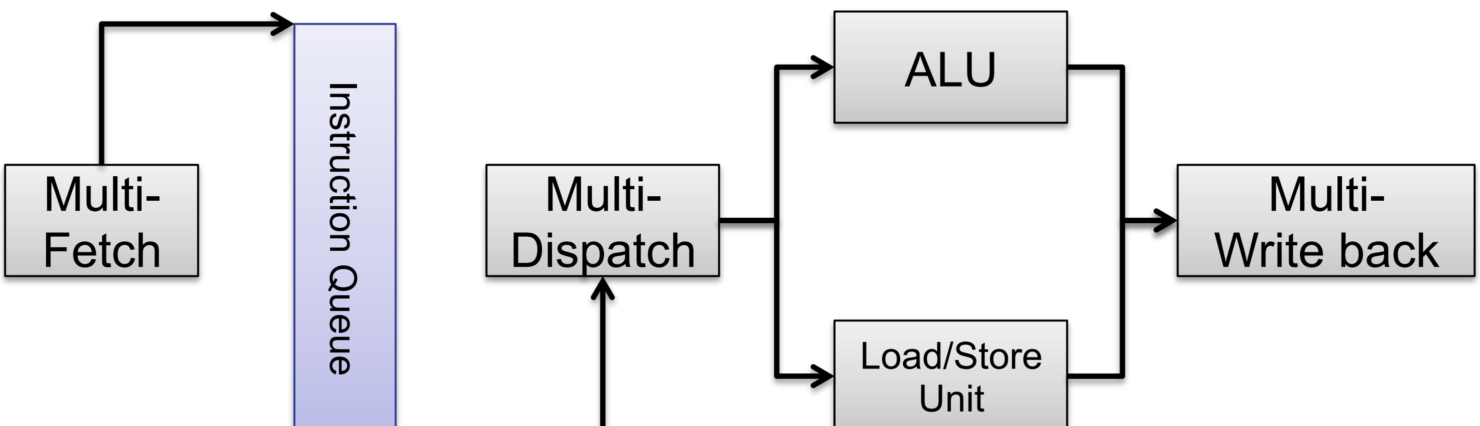
The five-stage model for instruction execution, or something close, is often found implemented in RISC processors however it isn't the only type of pipeline.

An N-stage pipeline can theoretically improve throughput N-times, so deeper pipelines should have better performance. In practice, long pipelines with many instructions executing simultaneously introduce more possibility of data interdependence and higher likelihood of stalls. Moreover, the branch penalty increases the further in to the pipeline the branch outcome is decided.

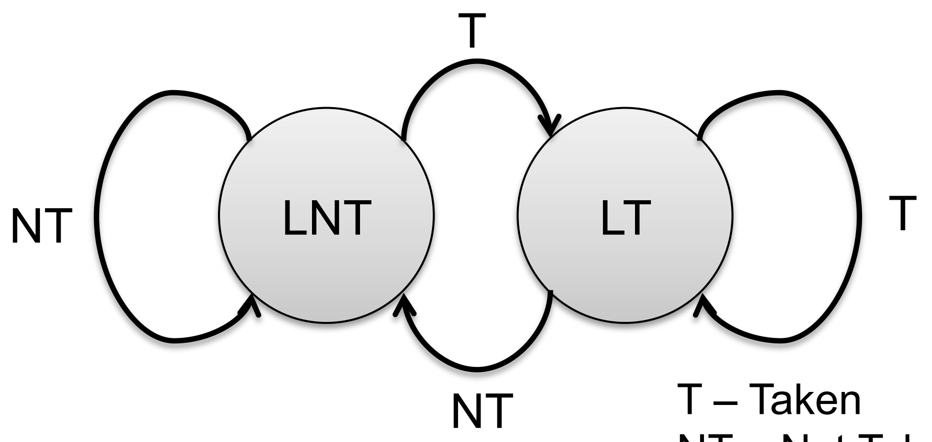
Despite this, some later Pentium 4 processors had pipelines 31 stages deep. In highly constrained devices, such as network packet processors, the pipelines may be several thousand stages deep.

Superscalar Execution

We have previously shown that many instructions either require the ALU or memory access but not both. For example, a direct-addressed load requires no computation and an add involving only registers requires no memory. Processors can exploit this in order to process more than one instruction per clock cycle. This is termed Superscalar Operation. Below is shown a section of a Superscalar pipeline with two execution units, however there may be other units to cover FPU operation, single-cycle multiply or other such operations.



Branch Prediction



T – Taken
NT – Not Taken
LT – Likely Taken
LNT – Likely Not Taken

Two Stage branch prediction.

This assumes that the current branch will have the same outcome as the last branch. This works fairly well for loops, they may be mispredicted on the first pass and *will* be mispredicted on the last pass but all intermediate branches will be correct. If the same loop is re-entered, and assuming the loop executes more than once, its first pass will be mispredicted as well.

Branch Prediction

```
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        do_stuff(i,j);
    }
}
```

There are branches at the end of loops making the decision whether or not the loop is terminated. In this case, the ☐ < 5 checks are done here

Branch Prediction

```
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        do_stuff(i,j);
    }
}
```

The first time this point is reached, the branch will be taken (the loop will loop), but whether or not this is predicted depends on what happened before the loop.

The second time will be correctly predicted as the branch is taken, and was taken last time

The last time will always be mispredicted as the branch is not taken, but was taken last time

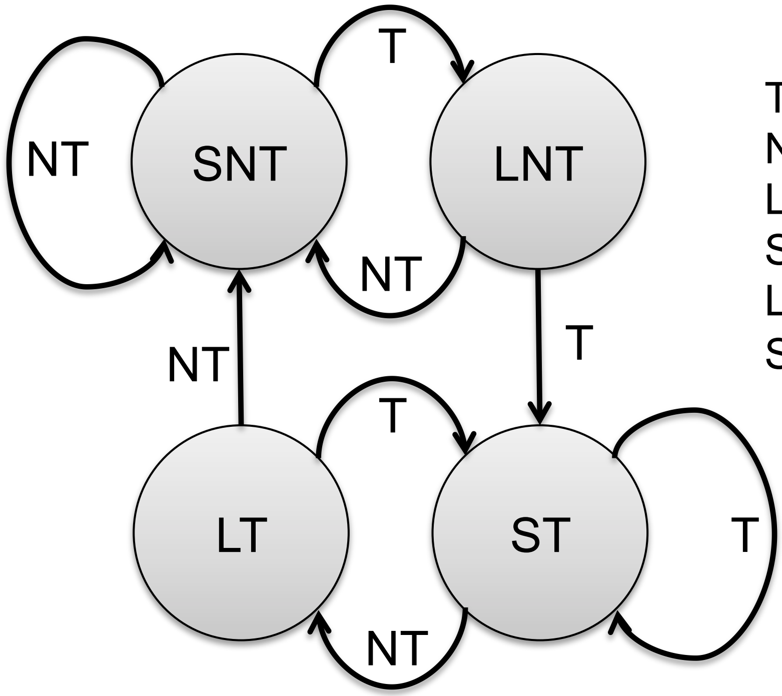
Branch Prediction

```
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        do_stuff(i,j);
    }
}
```

This will be mis-predicted the first four times as it is taken when the last branch, the one at the end of the inner loop, is always not taken (otherwise we wouldn't be here!)

The last iteration through the outer loop will be correctly predicted as both will be not taken.

Branch Prediction



T – Taken
NT – Not Taken
LT – Likely Taken
ST – Strongly Likely Taken
LNT – Likely Not Taken
SNT – Strongly Not Taken

Four Stage branch prediction.

Adding a second bit of state requires two consecutive mispredictions if the predictor has previously been ‘strongly’ conditioned. This correctly handles the outer loop misprediction. Note that nothing fixes the last pass misprediction on either loop.

Branch Prediction

A more complex scheme: Dynamic Branch Prediction Branch Target Buffer.

The CPU keeps a table which contains branch prediction state against the address of the branch instruction itself. Each time an instruction is fetched, its address is compared to addresses in the BTB and the next instruction to be fetched is determined from this state.

This changes the algorithm from predicting the future based on past branch results, to predicting the future of a particular branch based on the history of *that particular branch*.

The table is of course a finite size, typically around 1024 entries, which is sufficient for most repetitive tasks.

Memory and Cache

L1 CPU Cache

L2 CPU Cache

L3…

RAM

Flash

Hard Drive

Network/Cloud

Recall the memory hierarchy presented in the last lecture.

Each level may be used to cache the level above it. To cache means to store for future use. Confusingly, RAM is cached in a cache called Cache!

Hard drive contents can also be cached, this time in normal RAM. This is why if you have lots of programs open your computer it suddenly slows down: You've run out of real RAM and you're now using 'fake RAM' on the hard drive which is much slower.

This is also called 'swap'.

Memory and Cache

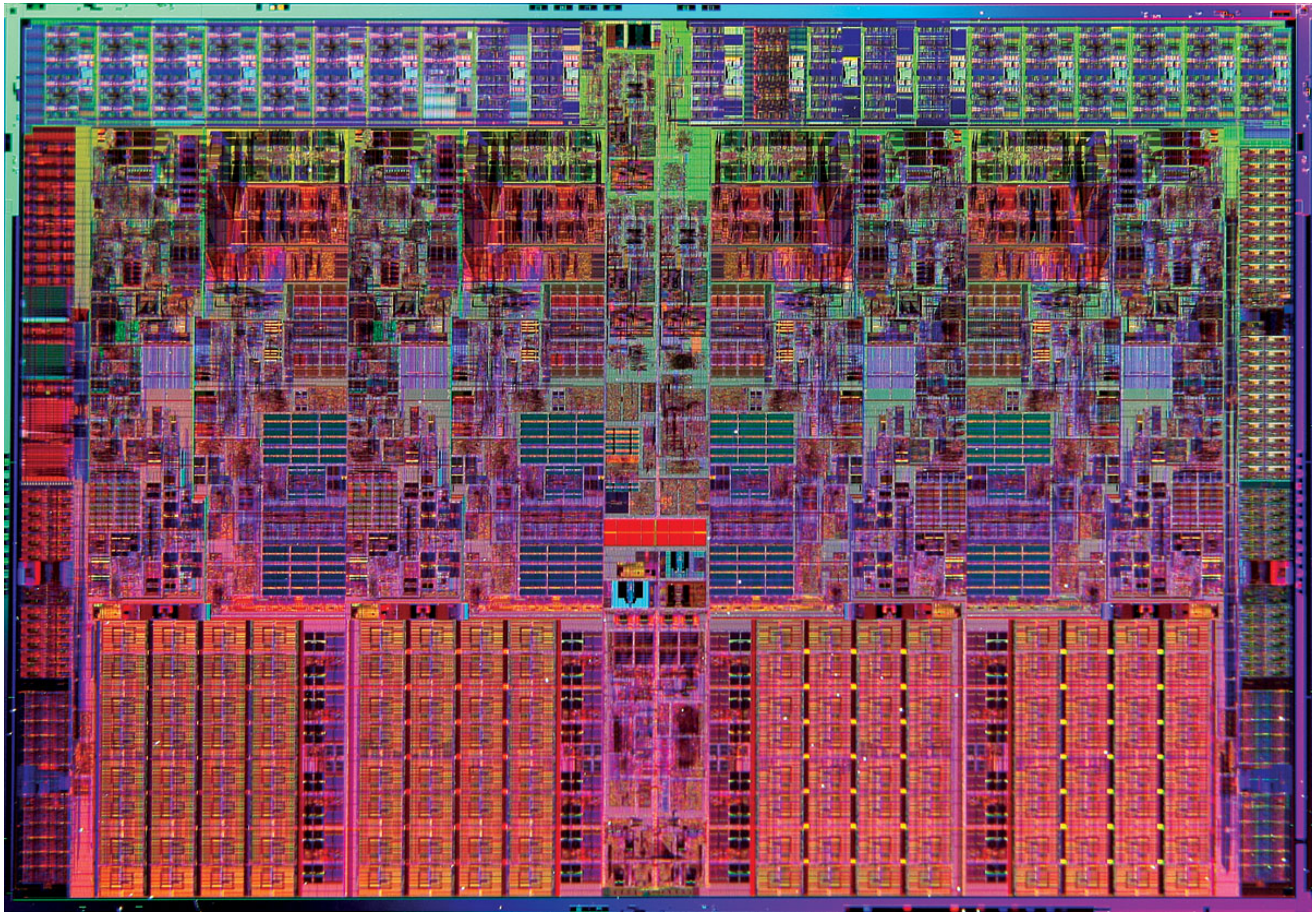
- L1 CPU Cache
- L2 CPU Cache
- L3…
- RAM
- Flash
- Hard Drive
- Network/Cloud

The fastest type of memory is the Cache memory inside the processor itself. We will discuss memories lower in the hierarchy later in the course, but today we concern ourselves only with Cache.

It is almost unimaginably small by today's standards: A few kB in L1 through to 8-24MB in L2/3

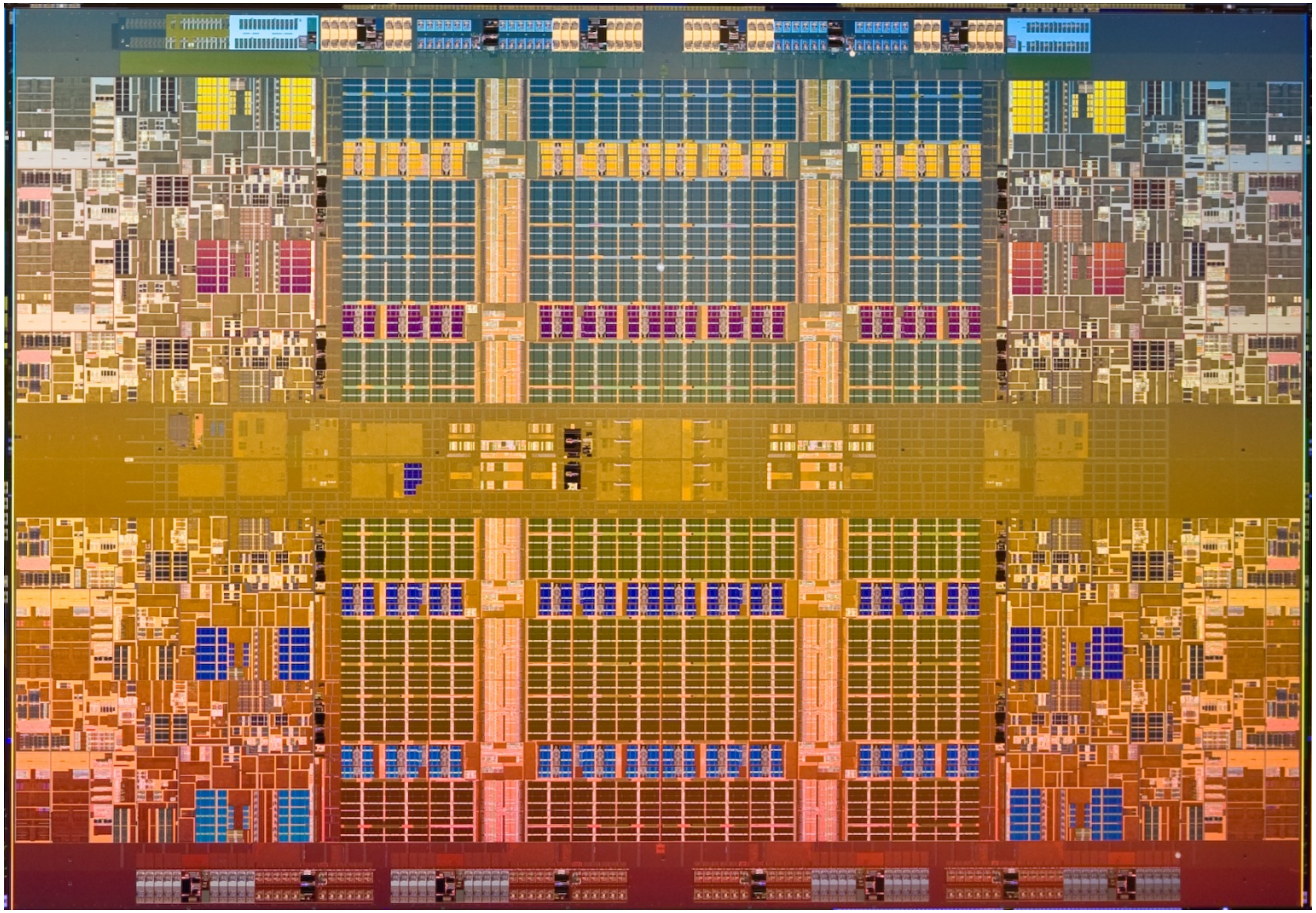
Cache

Why so little? It's physically big! Quad-core i7, 32kB L1 x 2, 256kB L2, 8MB L3



Cache

Xeon octo-core 32kB L1 x 2, 256kB L2, 24MB L3



Cache

Cache works on a principle called locality of reference. Analysis of computer programs shows that most of their time is spent accessing a small subset of their overall memory. This might be due to, for example, loops, small handlers for common events, a core set of functions that call each other etc. The actual mechanism that causes locality of reference isn’t important, however the effects are.

Instructions typically live in main memory, which can take several clock cycles to access. Pipelined processors require that their instructions arrive in a single clock cycle, and Superscalar processors may look to fetch several instructions per cycle.

The only way to keep modern processors fed with instructions is to have a tightly coupled, small and fast set of memory in which the instructions most likely to be executed next are found. Such memory is called the Instruction Cache. There is also a data cache that works on a similar principle, however we will focus on the Instruction Cache in examples.

Cache

Locality of Reference has two parts. First, Temporal Locality suggests that an instruction that has just been used is likely to be needed again soon. Spatial Locality suggests that when fetching one instruction it makes sense to fetch many nearby, as they are likely to be executed next.

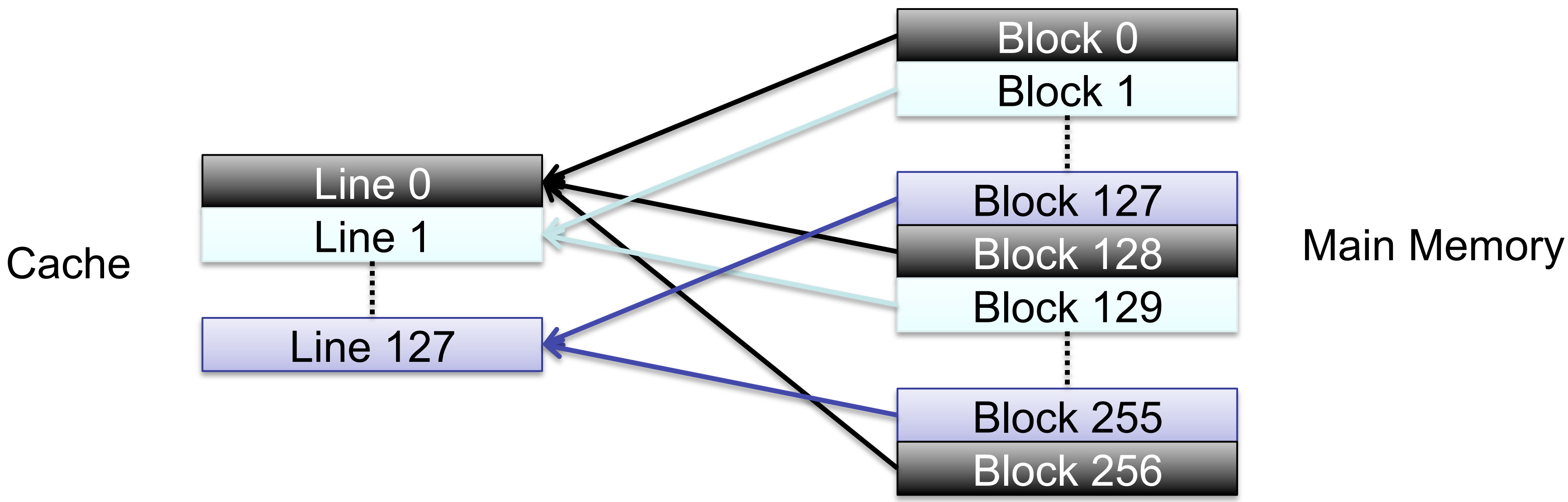
On these principles, a cache may be built in which chunks of main memory are cached based on how recently they were accessed or how near they were to some other piece of memory that was accessed.

These chunks are called Cache Blocks or Cache Lines. They are at least as big as the processor word size (e.g. 32-bit, 64-bit) but are more likely 32- or 64-*byte* in order to take advantage of the spatial locality property.

If a memory access finds its target is already in the cache, this is called a Cache Hit, if not it is a Cache Miss.

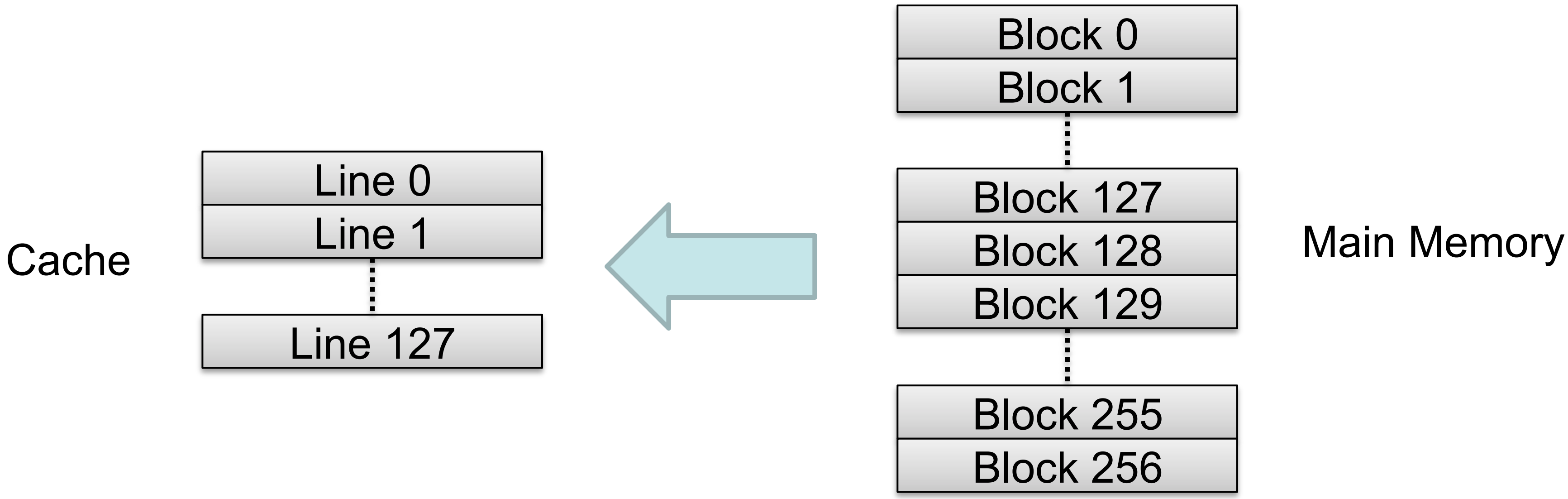
Cache

The cache is, of course, smaller than main memory so some algorithm has to be used to map a main memory location to a cache line. The simplest algorithm is called Direct Mapping. In this scheme, each memory block is stored in a cache line of the same address modulo the cache size. For example, a cache of 128 blocks will store memory block 134 in cache line number 6. DM Caches are simple but there may be contention even when the cache isn't full.



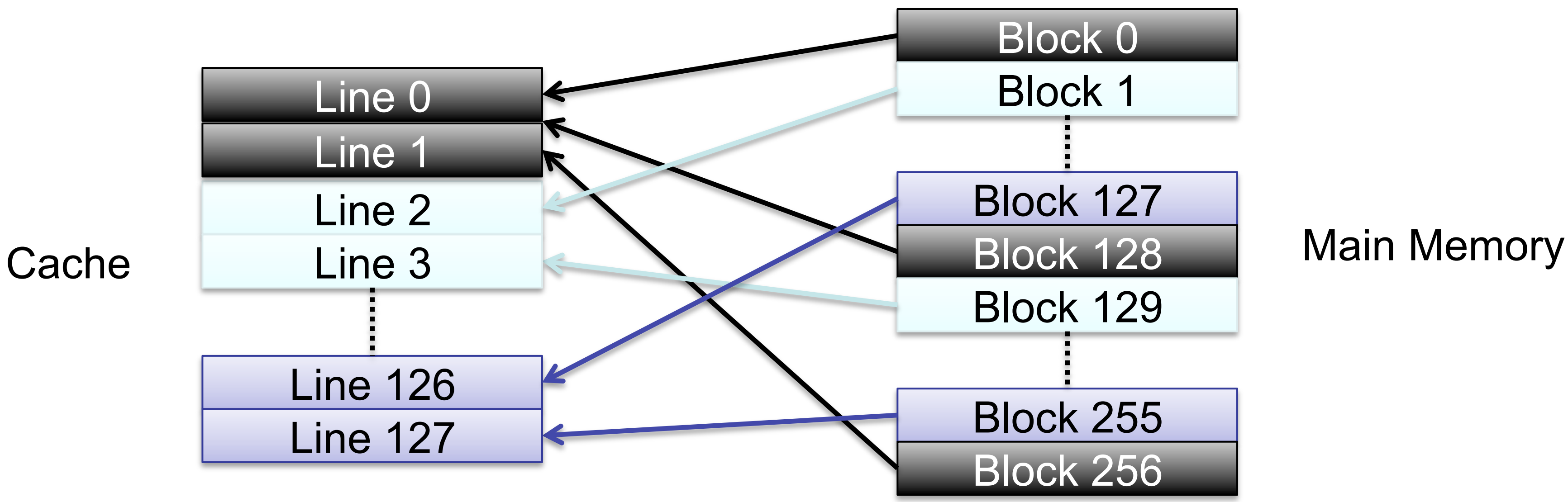
Cache

An Associative Mapping is the most flexible cache arrangement in which any memory location can be stored in any cache location. If the cache isn't full, each new memory block can just take an empty slot. If the cache is full, the cache line to evict is chosen typically using a Least Recently Used algorithm. Complexity comes from finding *where* in the cache a particular memory block has been put when it is next needed.



Cache

Almost all modern caches are Set Associative. These combine the best features of each of the previous methods, as each memory location "short lists" a set of cache lines based upon its address modulo the number of sets, then the entry to evict from the set is chosen by an LRU algorithm as with an Associative cache.



Cache

Cache strategies discussed previously assume a read access, but what happens for a write?

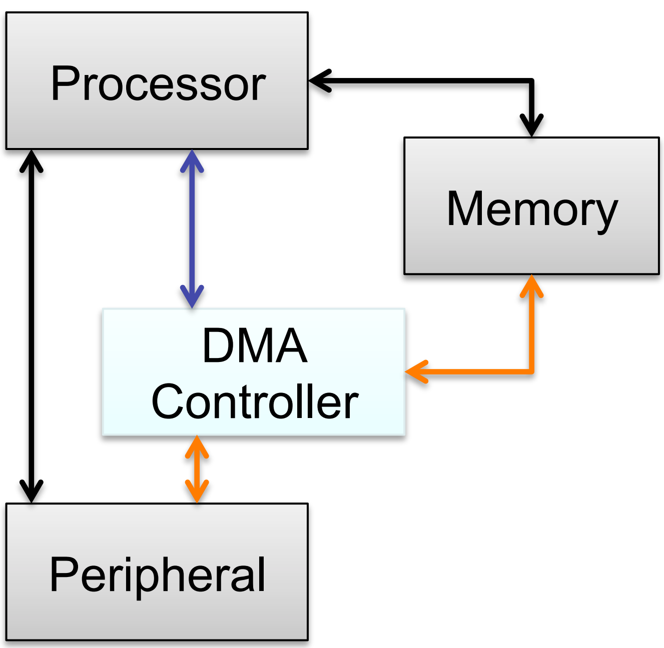
If the memory is in Cache, what should a write do: Write-back or Write-through? In a write-through cache, a write to a memory location in Cache will write the new value to the cache line and the memory location. Write-back writes the new value to cache and marks it dirty; it is then written to main memory only when the line is evicted.

Write-through generates unnecessary writes if the memory location is written several times in sequence. Write-back generates unnecessary writes if only one location in the cache is changed, as the whole line must be written back.

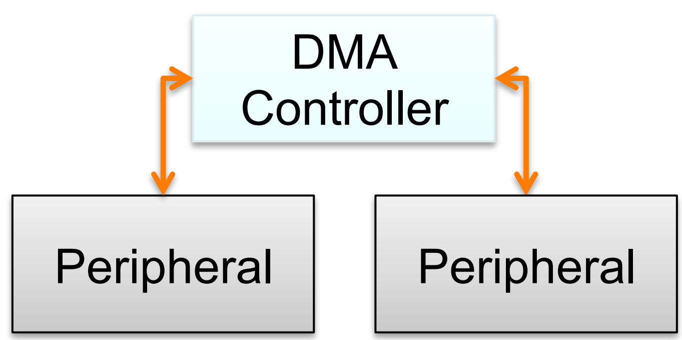
Direct Memory Access

Direct Memory Access (DMA) is a technique used when large amounts of data need to be moved between peripherals and memory without any intermediate processing. Instead of requiring a processor load/store cycle, the data may be moved around without processor intervention.

Traditional DMA is between a peripheral and memory, however Peripheral DMA can move data directly between different peripherals



DMA Access



Peripheral DMA

Direct Memory Access

What’s the interaction between DMA and caches?

Because the memory has multiple masters (writers), information kept in cache can’t easily be the source or destination of a DMA transfer.

Typically memory reserved for DMA transfers will be marked as un-cached and will be much slower to access from the CPU.

This issue of “Cache Coherency” is valid when ever there are multiple memory masters, such as if you have a multi-core CPU. When doing multi-CPU programming, any data that is accessed by more than one CPU can’t feel the full benefit of being cached. We will look in to this next lecture.

Interrupts and Exceptions

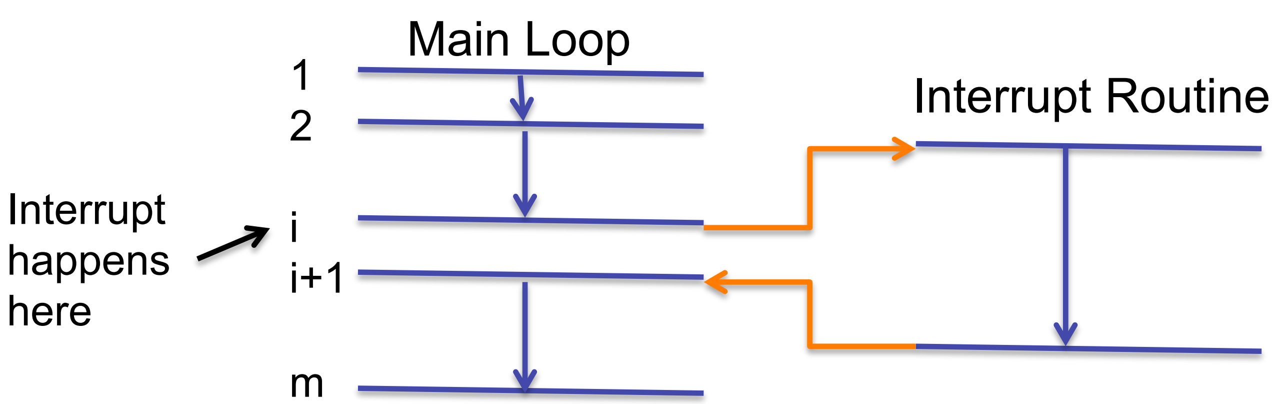
Both Interrupts and Exceptions are events that cause the normal execution of code to be stopped and a new branch taken. They are effectively instantaneously-injected jumps.

The term Interrupt refers to such a jump that's taken as the result of external stimuli, typically a pin on the processor undergoing a transition (high to low, low to high). An Exception has the same effect except as a response to internal conditions such as Divide by Zero (FPU Exception).

Since Exceptions can be triggered from software they are often used to break a 'normal' flow of code. For example, a Cache Miss as described previously might actually be implemented as an exception that executes code to load the new cache entry in.

Interrupts and Exceptions

The most common use of interrupts is as a mechanism for external devices to signal to the processor that they're in a particular state (e.g. Full of Data, Empty of Data, Too Hot, Too Cold, Too Hungry). This stops the processor "wasting" CPU cycles asking the device for this information.



Interrupts and Exceptions

Interrupts are a simple concept but interact with different pieces of the CPU in often-unexpected ways.

Pipelining: Interrupts act like a branch so cause waste of partially-executed instructions. If the interrupt is injected in to the pipeline in the same way as other instructions then it may also be stalled by currently-executing commands, leading to variable interrupt latency. This is often unacceptable.

Caches and Branch Prediction: If the Interrupt Handler uses memory or branches (and it almost always will) then it will leave a legacy in the form of conditioned branch predictors and populated caches. It will change the timing of the code it interrupted (by more than just the time it took to execute itself!).

Tutorial Question 1:

Referring to the memory access latency benchmark on the next slide (and ignoring the last point at 0 which is a bug in the plotter!), answer the following:

1. How many levels of cache does the processor have and (roughly) how big is each one? How big is Main Memory (RAM)?
2. One of the caches is 6MB, is it most likely to be Direct or Associative?

The plot is of steady-state sequential access latency. That is, a loop is run that accesses each memory location in turn, from address 0 up to the length under test. This loop is run several thousand times in succession and latency of each access timed and averaged. This means that any different timing on the first run through can be ignored.

Tutorial Question 1: