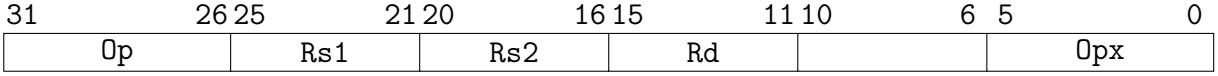
Advanced Computer Architecture (60001)

Chapter 1

Pipelines

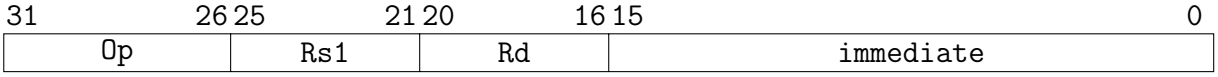
For the sake of example, we're using MIPS, which is a reduced instruction set design (every instruction is 32-bits wide, making it easy to decode);

- **register-register** two source registers and a destination register



  For example, `ADD R8, R6, R4` sets the value of `R8` to be the sum of `R6` and `R4`.

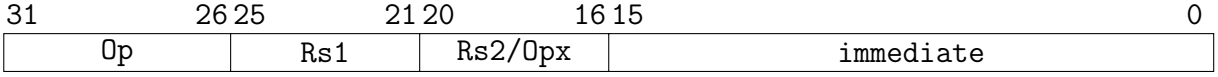
- **register-immediate** one source and one destination register, immediate operand (e.g. adding)



  Examples of this include;

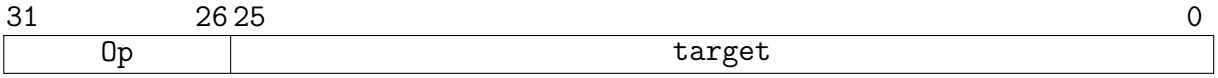
  - `LW R2, 100(R3)` `R2 <- Memory[R3+100]`
    source register is to be added to an immediate constant, could be useful for accessing fields of a struct (field offset is the constant) by using the base of the object
  - `SW R5, 100(R6)` `Memory[R6+100] <- R5`
    same as above, but for storing
  - `ADDI R4, R5, 50` `R4 <- R5 + signExtend(50)`
    note the sign extend; since we have a 16-bit immediate value, if it's a negative number, it needs to be padded to 32-bit

- **branch**



  Two registers, for example if the contents are equal, then branch. One challenge is that the address would have to fit in the immediate field, therefore it's not an **absolute** address, but one that's **relative** to the current program counter. Note that it's multiplied by 4, as it's a fixed size. For example, `BEQ R5, R7, 25`; if `R5 = R7`, then update `PC <- PC + 4 + 25*4`, otherwise `PC <- PC + 4` (we update it regardless).

- **jump / call**



  Unconditional branch, allowing for a jump to anywhere in the full address space of the machine.

The top bits are occupied by the opcode. Register specifier fields also occupy fixed positions in the instruction allowing immediate access to the registers before the instruction finishes decoding. In MIPS, we can have up to 32 registers in the machine as the width of the register specifier fields is 5 ($2^5 = 32$). The largest signed immediate operand is determined by the width of the immediate field, similarly the range of addresses of a conditional jump is the same (albeit scaled by 4). A general machine would execute this in a loop (the slides contain a diagram of this with the components);

```
Instr = Mem[PC]; PC += 4;
rs1 = Reg[Instr.rs1];
rs2 = Reg[Instr.rs2];
imm = signExtend(Instr.imm);
Operand1 = if (Instr.op == BRANCH) then PC else rs1
Operand2 = if (immediateOperand(Instr.op)) then imm else rs2
res = ALU(Instr.op, Operand1, Operand2)
switch (Instr.op) {
  case BRANCH:
    if (rs1 == 0) then PC = PC + imm*4; continue;
  case STORE:
    Mem[res] = rs1; continue
  case LOAD:
    Imd = Mem[res]
}
Reg[Instr.rd] = if (Instr.op == LOAD) then Imd else res
```

The 5 steps of the MIPS datapath are as follows - the pipelined design introduces pipeline buffers between each stage;

1. instruction fetch
2. instruction decode, register fetch
3. execute, address calculation
4. memory access
5. write back

The pipeline buffers allow the result of a stage to be latched in the buffer, to be used in the next clock cycle. This allows each instruction to progressively move down the pipeline on each successive clock tick. Without pipelining, the entire pipeline would operate on one clock cycle, however the number of gates along the path would determine the clock cycle time / rate - a non-pipelined processor would run slowly as there is a long path. In a pipelined design, the cycle time is determined by the slowest of the stages, but the length of each stage is controlled to allow for the highest possible rate.

In addition to this, there is also a decode which controls what is being done (which sources to read from, for example a branch would use PC rather than a register). The control signals configure the multiplexers (MUX), ALU, and read / write to memory. The signal is carried with the corresponding instruction along the pipeline.

Initially the pipeline is empty. With each successive cycle after the first, more of the pipeline will be used, with a new instruction being fetched at each cycle, and the preceding instruction being passed to the next pipeline stage.

Pipeline doesn't make each individual instruction complete quicker; it doesn't help the **latency** of an instruction, but rather the **throughput** of the entire workload. The pipelining doesn't add much cost as we're using the same hardware, other than latches which add some transistor count and energy consumption. Adding more stages to a pipeline could lead to the clock rate being dominated by the time time the signal is spent in the latches.

However, there are a number of hazards to pipelining;

- **structural hazards** - particular hardware units may not be able to be used by two different pipeline stages in the same cycle

  The *PS3* was based on a highly parallel multi-core processor chip. The processor had a conventional unit to run the Linux OS, but also 16 parallel accelerator units (fast but simple units, no cache, just a block of SRAM for instructions and data). In cycle 4, for the first instruction, it would be accessing memory, however in the same cycle, a later instruction would be trying to fetch the instruction from the **same** memory. A structural hazard occurs on simultaneous access; leading to load instructions causing an instruction fetch to stall. This causes a stall (delays the instruction until the next cycle), which introduces pipeline bubbles - a missed opportunity for an instruction to be processed; once a fetch is missed, subsequent steps can't do anything in next cycles. This was solved with a prefetch buffer, where a block of instructions was fetched at once. Instructions were reorganised at compile time.

- **data hazards** - an instruction depends on the result of an incomplete (still in pipeline) instruction (causes bubbles / stalls, can be overcome with forwarding)

  Consider the following example;

  ```
  1 ADD R1,R2,R3
  2 SUB R4,R1,R3
  3 AND R6,R1,R7
  4 OR R8,R1,R9
  5 XOR R10,R1,R11
  ```

  After the instruction is fetched in cycle one (for the `ADD`), `R2` and `R3` are read in in cycle 2, available in cycle 4, but only written back in cycle 5. For the `SUB` and `AND` instructions, the data would need to be sent back in time in the current idea of the pipeline, which obviously isn't feasible. `XOR` is possible, as the register read is in cycle 6. For `OR`, this is fine **if** the register write happens in the first half of the clock cycle, and the register read happens in the second half.

  The previous assumption is that the value had to be written to the register before it could be used; however, at the end of cycle 3 the result of the `ADD` instruction is present in the latch after the execution stage, and could be fed directly into the ALU (same for the next instruction, but rather from the latch after the memory stage). The value needs to be delayed by one clock cycle, before forwarding.

  The changes to the hardware to support this are as follows (see lecture slides for diagrams);

  - add forwarding / bypass paths (to the MUX, mentioned next)
    - result of the ALU from previous cycle (latch)
    - the latch after memory, for forwarding the value to the next instruction but one
    - final wire takes value from memory
  - expand multiplexers before ALU to select where the operands should come from (choose one of the bypass wires if forwarding is needed)
  - decode needs to control bigger multiplexers to select values from bypass paths for forwarding; decode will now need to track which registers are going to be updated by incomplete instructions (the decode stage knows where the operands are going to come from, as well as what operands are still in-flight)

  Data hazards can still exist even with forwarding, for example with loads, as the memory access comes later in the pipeline;

  ```
  1 LW R1,0(R2)
  2 SUB R4,R1,R6
  3 AND R6,R1,R7
  4 OR R8,R1,R9
  ```

Recall that arithmetic may be involved to access memory, hence the stage has to come later. For the value that is required in cycle 4 (execution of `SUB`), the value is only available at the end of the cycle. There is nothing that can be done here, leading to a bubble (also known as a load-use stall). Stalls will also need to be implemented to support this.

Software scheduling can be performed to avoid load hazards (recall that bubbles are missed opportunities for execution). Consider the following code;

```
a = b + c
d = e - f
```

Slow code, without the optimisation takes 10 cycles, with 2 stalls;

```
LW    Rb,b
LW    Rc,c
STALL
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
STALL
SUB   Rd,Re,Rf
SW    d,Rd
```

However, the faster code swaps the order of execution by moving `LW Re,e` in place of the first stall and `SW a,Ra` in place of the second stall. This takes 8 cycles and has no stalls;

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

- **control hazards** - we assume that we already know the next instruction to fetch, however this may not be the case as we haven't decoded the previous instruction yet (may be a jump / branch)

  Consider the following example, where we may risk stalling for three cycles;

  ```
  BEQ R1,R3,36
  AND R2,R3,R5
  OR R6,R1,R7
  ADD R8,R1,R9
  XOR R10,R1,R11 # instruction 36
  ```

  After discovering the branch outcome at cycle 3, we may suffer a bad stall. This can be overcome by adding early branch determination. Add an adder to add the current PC to the immediate operand (in decode stage), add check with with register file, and if it passes we can use the computed next PC value All the logic for determining the branch outcome is moved as early as possible in the pipeline. This still introduces a delay for one cycle, as we have to fetch the next instruction regardless (while we're computing whether we should branch or not). If the branch is taken, the memory access and write back stages are blocked.

Simultaneous multi-threading can eliminate hazards. Without stalls, an instruction could be finished each cycle. Two program counters are maintained and the processing alternates between the two

counters. Each thread has its own program counter and own registers, thus eliminating issues with data hazards (can still occur with memory).

A simple pipeline with 5 stages can run at 5 - 9 GHz, limited by the **critical** path (slowest pipeline stage). The main tradeoff is to do more per cycle or to increase the clock rate.

Introduction to Caches

In *Intel Skylake*, there are L3 caches between the cores, an L2 cache associated with each individual core, as well as L1 data and instruction caches deeply embedded in the processor. In the past, RAM access time was close to the CPU's cycle time, hence there was little need for cache. However, the gap between the processor and the memory grows by around 50% each year, leading to a growing gap (despite DRAM performance still increasing) - access times can be above 100 cycles. The lecture then goes into the memory hierarchy.

Caches works due to the principal of locality, of which there are two types - most (if not all) modern architectures rely heavily on locality;

- **temporal locality** - if an item is referenced, it will be used again soon (loops, reuse)
- **spatial locality** - if an item is reference, items (addresses) close by tend to be used soon (straight-line code, array access)

Consider a direct mapped cache, of size 1KB and blocks that are 32 bytes wide (32 blocks, as $1024 = 32 \times 32$). For a cache of size 2^N bytes, the uppermost $32 - N$ bits are the cache tag, with the lowest M bits being the byte select (the block size is 2^M);

The example would select the second row (bytes 32 to 63), and take byte 32.

byte 31	···	byte 1	byte 0	0
byte 63	···	byte 33	byte 32	1
		⋮		
byte 1023	···	byte 993	byte 992	1

The role of the cache tag is to add metadata, including comparing the tag in the table with the tag in the cache. If it matches, and the valid bit is set, then we have a cache hit, otherwise a cache miss.

This has a problem; cache location 0 can be occupied by data from main memory locations 0, 32, 64, and so on, similarly for location 1 being occupied by memory locations 1, 33, 65. Consider the following program, in C;

```
int A[256];
int B[256];
int r = 0;
for (int i = 0; i < 10; ++i) {
  for (int j = 0; j < 64; ++j) {
    r += A[j] + B[j];
  }
}
```

The memory will be allocated contiguously, each being 256 bytes (or 8 32 byte cache lines). The 1KB direct-map cache should be able to hold all the integers (as it's only a total of 512B). However; `A[0]` and `B[0]` would have identical address bits (for the cache index) - since they are exactly 1024 bytes apart, meaning they will map to the same place, as well as all subsequent values. This means they will continuously displace each other, and only use a subset of the cache.

This is solved with an associative cache - for an N-way set associative cache, there are N entries for each cache index (typically 2 to 4); N direct mapped caches operated in parallel. The cache index selects a set from the cache, the two tags are compared, and the data is selected based on the matching tag result (if either leads to a hit, it's a hit, otherwise a miss).

However, this leads to adding extra hardware, which is on the **critical path** for determining whether it's a cache hit or miss (adding an additional MUX introduces another gate delay, which we already have few of).

An example is the *Intel Pentium 4 Level-1 Cache*. This had a capacity of 8KB (total data it can store), blocks of size 64 bytes (128 blocks in the cache), $N = 4$ (4-way set associative cache), leading to 32 sets. The index was 5 bits, to select one of the 32 sets, leading to a tag of 21 bits ($32 - 5 - 6$, subtracting the index and the 6 to address the byte within the block). The access time of this was 2 cycles.

There are 4 questions for memory hierarchy;

1. where can a block be placed in the upper level? **block placement**

   In a direct-mapped cache, there is only one cache location it can be placed in, determined by its low-order address bits. In an 2-way set-associative cache, it can be placed in either of the two cache locations corresponding to the set determined by low-order address bits. A fully-associative cache, it can be placed in any location - more associativity leads to the issues;

   - more comparisons (more transistors, more energy, larger)
   - better hit rate (a lot of associativity leads to diminishing returns)
   - more predictable; reduced storage layout sensitivity (cache hits are independent of storage layout)

2. how is a block found if it's in the upper level **block identification**

   In addition to the memory required to store the cached data, we need metadata (valid, cache tag) to tell us whether we have a hit or not. The tag doesn't need to include the block offset nor the cache index. As associativity increases, the index shrinks (less indices) and expands the tag.

3. which blocks should be replaced on a miss? **block replacement**

   In a direct-mapped cache, there is no choice to be made, and it can only replace one block. However, with an associative cache we ideally want to choose to replace the block that will be reused least-soon. LRU (least recently used) is a good approximation, random is very good for large caches (LRU only beats random for small caches). A program that periodically sweeps over an array that doesn't fully fit into cache would be bad in LRU.

4. what happens on a write? **write strategy**

   On a store instruction, we need to check if that address is cached. If it is, then we must update it. However, we need to decide whether to write to the next level of the memory hierarchy (write through), main memory, or not (write back). The latter only updates the cache and not the next level of the hierarchy **until** the block is evicted (replaced) - this requires an additional status bit to indicate whether the block is clean or dirty. The write back strategy has an advantage with repeated writes to the same location as the writes are absorbed without propagating to main memory - this may include writing to successive elements in an array. Write through ensures the rest of the system gets updated more promptly. This helps with cache coherency, however it does not solve it entirely, as other processes may have their own cached copies of this data.

The lecture then continues about the bottom of the memory hierarchy (large machines with robot arms for tapes), as well as architectures that don't have caches and therefore don't depend on locality in access patterns. The latter has the idea that with enough threads, memory latency can be hidden.

Turing Tax Discussion

A stored-program computer is a machine that fetches instructions from memory and executes them. *John Backus* dubbed the limitation of the stored-program mode being serial (one instruction at a time) the *von Neumann bottleneck*, and suggested writing our programs in a way that expresses the data dependencies without prematurely committing to the exact order the instructions should be executed in (functional programming).

Alan Turing proposed the idea of a universal / single device which can implement any computable function without further configuration, given the right program. The tax refers to the overhead for execution time, manufacturing cost of the machine, or the energy required for computation. The aim is to characterise the cost difference between programming a general purpose machine to do a job, versus designing a application-specific machine for it.

H.264 encoding is dominated by 5 stages, applied to a stream of macroblocks (blocks of the video) - the remaining steps correspond to finding a compact encoding;

(i) **IME** Integer Motion Estimation — motion relative to previous frame

(ii) **FME** Fractional Motion Estimation — refines the previous estimate

(iii) **IP** Intra Prediction

(iv) **DCT/Quant** Transform and Quantisation

(v) **CABAC** Context Adaptive Binary Arithmetic Coding

An ASIC, fully customised for this task, was able to outperform *Intel Pentium* at a significantly lower energy consumption and smaller size.

Turing Tax is present in our pipeline in many stages. The instruction fetch, maintaining PC, handling and predicting branches are all tasks that could be avoided in a specialised machine. Routing paths be replaced by colocating the producer and consumer together, shortening wires. Registers are also used to pass instructions from one instruction to the next; in a special-purpose machine, this can be a single piece of wire connecting units. The ALU can be configured to do many different things in our pipeline, but in a specialised machine, we may have several ALUs, each specific to a particular operation.

Chapter 2

Dynamic Scheduling, Out-of-order Execution, Register Renaming

Consider the following instructions;

```
1 DIVD F0,F2,F4
2 ADDD F10,F0,F8
3 SUBD F12,F8,F14
```

Note that division can take many cycles (and is slow) and that the `SUBD` instruction doesn't use anything in either of the preceding instructions. The key idea is to allow an instruction that is behind a stall to proceed.

There are a number of constraints on execution order (note that anything in `monospace` denotes an instruction, `I` refers to instruction `I`);

1. `J` is **data dependent** on `I` if `J` tries to read an operand before `I` writes it;

   ```
   1 I: ADD R1,R2,R3
   2 J: SUB R4,R1,R3
   ```

   It can also be the case that `J` is data dependent on `K` which is dependent on `I`. A Read After Write hazard is when a true dependence causes a hazard in the pipeline.

2. There are two types of **name dependence**, both of which have instructions that use the same register / memory location (called a name)

```
1 I: SUB R4,R1,R3
2 J: ADD R1,R2,R3
3 K: MUL R6,R1,R7
```

   The first kind is anti-dependence / Write After Read, where `J` writes an operand before `I` reads it, caused by the reuse of name `R1`.

3. The second type is called an **output dependence**, where `J` writes the operand before `I` does;

```
1 I: SUB R1,R4,R3
2 J: ADD R1,R2,R3
3 K: MUL R6,R1,R7
```

   `K` needs to get the correct `R1`, also called a Write After Write hazard.

The decode stage can be renamed to issue; which decodes the instruction and checks for any structural hazards. There will also be a later read operands stage, where the operands are actually collected. In the issue stage, the register `F0` is checked and we may see that it's marked with some bit that indicates the value isn't ready yet and is being computed by an instruction that's in-flight. This stage may collect the operands if they are already available, otherwise discover where they will come from and collect it later in the read operands step.

The lecture goes into the *IBM 360/91*, which had a small number of FP registers in the instruction set. This prevented compiler scheduling for avoiding load-use stalls.

Consider the following series of instructions, at the issue stage (multiply `F1` and `F2` into `F0`, store into address `X`, and similar for the next two instructions);

```
1 MUL F0,F1,F2
2 SD F0,X
3 MUL F0,F2,F3
4 SD F0,Y
```

The issue stage consults the registers and allocates the instruction being issued to a reservation station. The reservation station for each functional unit holds the opcode and the two operands; both of which need to be present before the instruction can progress to the functional unit. The registers can hold a value and tags (different to the ones we saw in cache); if the tag is null, then the value is valid (for example, the tag for `F0` would be null if no instruction in the machine is producing a value for `F0`). If it isn't null, then it will indicate the ID of the functional unit that will produce the result for that register.

In our example, there are 4 registers `F0` to `F3` and 4 reservation stations (`MUL1`, `MUL2`, `Store1`, `Store2`). Going through the instructions;

1. issue unit looks at instruction 1 and collects the operands from the source registers `F1` and `F2` (assume no instructions in the machine currently; the tag is null and the value is present)
2. allocate the multiply operation to the first unit `MUL1` by copying the values from `F1` and `F2`, and update `F0` with the ID of the multiply unit (`MUL1`)
3. issue looks at instruction 2 and sees `F0` is waiting for `MUL1`
4. allocate the operation to `Store1` by copying in the tag `MUL1`
5. issue looks at instruction 3 and collects the results from the source registers, which are present
6. allocate the operation to `MUL2`, and overwrite `F0` with the ID of the unit `MUL2`
7. similar for the second store, but keeps the tag `MUL2` in the reservation station

There is a common data bus that connects the results of the functional units to **both** the registers as well as the reservation stations. When `MUL1` finishes, a signal is sent and anything listening for that tag will be updated - since `Store1` had the tag from the registers and is waiting on `MUL1`, it would get the value. On the other hand, if `MUL2` were to somehow finish before, it would update the value in the register (as well as the `Store2` reservation station), but not the `Store1` reservation station.

We typically think of registers as a location where data is stored, but in the *Tomasulo* design, the register becomes a tag that connects the production of the value to where it needs to be delivered to. At issue time, we are decoding the dependent structure of the program dynamically, and arranging for the forwarding wiring to deliver resulting operands to the functional units at the right time.

Another walkthrough of the same instructions is as follows. Assume that the value fields of `F1` and `F2` are already populated;

1. instruction 1 issued; **values** of `F1` and `F2` are routed to `MUL1`'s operands 1 and 2 respectively, as both tags are null, the tag of `F0` is updated with ID of `MUL1`, stating that the value will come from there
2. instruction 2 issued; **tag** of `F0` (`MUL1`) is routed to `Store1` as well as address `X`
3. instruction 3 issued; **values** of `F2` and `F3` routed to `MUL2`'s operands (since both tags are null again), and tag of `F0` is **overwritten** with ID of `MUL2`, updating where the value will come from
4. instruction 4 issued; **tag** of `F0` (`MUL2`) is routed to `Store2` as well as address `Y`
5. at this point, nothing has completed yet; both store units are waiting for a value matching the tag to be broadcasted on the common data bus, `F0` is also waiting
6. `MUL1` finishes; broadcasts the result on CDB with the `MUL1` tag - `Store1` picks up the value and stores to memory, `F0` doesn't do anything as it's now waiting for `MUL2`
7. `MUL2` finishes; also does the broadcast, picked up by `Store2` which stores to memory, `F0` also updates its value

There are three stages of this algorithm;

1. **issue**

   Gets the instruction from FP operation queue. If the reservation station is free (hence no structural hazard), the instruction is issued and operands are sent (renaming registers).

2. **execute**

   Actually operands on operands when **both** operands are ready. If they aren't both ready, then watch the CDB for a result.

3. **write result**

   Write to the CDB for all units that are waiting, and also mark the RS as available.

This relies on two busses, the first data bus which goes from issue to the registers, as well as the common data bus which goes from the functional units to the reservation stations as well as the registers.

However, this design is complex, leading to many comparators on the common data bus. The CDB also limits performance as it needs to connect multiple functional units to multiple destinations - the number of functional units that can complete per cycle is dependent on the number of CDBs (one in our case). Finally, the trickiest issue is non-precise interrupts.

As a loop is essentially the same instructions, a compiler wold have to introduce new registers to be able to find a static schedule. The scheme effectively implements register renaming. Consider the following example with a loop (load using `R1` as a pointer, multiply with that result with some constant into `F4` and store it into the same place, then decrement the pointer down to zero);

```
Loop:
    LD      F0 0(R1)
    MULTD   F4 F0 F2
    SD      F4 0(R1)
    SUBI    R1 R1 #8
    BNEZ R1 Loop
```

Assume multiply takes 4 cycles, loads take 8 cycles (misses the L1 cache, hits L2 cache), assume integer instructions and store instructions don't use the CDB. See the slides for the actual pipeline; but it essentially allows for four iterations of the loop to be running (in-flight) in parallel with each other. A more realistic scenario is that the second iteration has a hit on the L1 cache, which dynamically adapts the schedule based on cache hits / misses, something that is very valuable in dynamic scheduling.

Speculation in Dynamically-scheduled Processors

On a conditional jump, we don't have to wait, but rather take a guess which gives us much higher performance if correct and handle the case if not. With a misprediction, we need to roll-back the state to the state before the prediction. This is done by adding a stage to the Tomasulo execution pipeline to commit the state of the machine. The speculative Tomasulo algorithm is as follows (a branch misprediction would flush this buffer);

1. **issue**

   If the reservation station (and the reorder buffer slot) is free, issue instruction, send operands and reorder buffer number for destination. One ROB entry is allocated for each instruction, and this holds the destination register with either its result value (or the tag from where it will come from).

2. **execute** same as before

3. **write result**

   Write to all awaiting units and reorder buffer (also marks RS as free). A ROB entry with the matching tag is also updated.

4. **commit**

   This updates the register with the reorder result. When an instruction is at the head of the buffer and has a present result; the commit-side register (or stored to memory) is updated with result, and instruction is removed from the reorder buffer.

The existing machine is extended with the buffer, the commit unit, and commit-side registers. The issue stage pushes instructions onto the buffer, which listens to results on the CDB, and the commit unit commits the results in **program order**. The only externally visible part of this machine is the commit-side registers (updated when the speculation is resolved); if we want to reset to a previous stage, we only look at what's on the committed side of the machine. Previously, we only had the issue-side registers (which were updated speculatively).

Consider an example as follows (assume that we predict the branch isn't taken);

```
MUL F0,F1,F2
SD F0,X
BEQ R10,Lab
MUL F0,F2,F3
SD F0,Y
```

Note that the ROB would be in the same order as the instructions, and we would also store the prediction in the ROB (for `BEQ`). By the time the first two are committed, and `BEQ` is at the head of the buffer, assume we also have the result for `BEQ`. This is compared to our prediction - consider a misprediction; all subsequent entries in the ROB are trashed, the issue-side registers are reset using values from the commit-side registers (as they are correct even before the misprediction), and finally

the correct target instruction is fetched and issued. As such, we reinstate the register state at the point the conditional branch was executed. Due to this, we can see a penalty as we are discarding work that has been done and we also need to refill the pipeline. Assume that the FUs are still working - once they complete and write back, nothing will be waiting (ROB is flushed, as well as issue-side registers).

One subtlety that could be considered is the case where we know the branch is mispredicted before it's at the head of the buffer. We could start to fetch the correct instruction, however we need to take care to ensure that we are in a consistent state. Another case is when there are two predicted branches; the current approach will handle this fine as the entire ROB is flushed once the first prediction is at the head, and turns out to be a misprediction.

Another subtlety lies in stores buffered in the ROB; it's important to buffer them before writing to the memory system. However if something attempts to load from that memory location, the correct data may not be in memory yet. A conservative strategy for this could be to stall loads until preceding stores have all been completed. Another approach would be to check all preceding store instructions for the address in memory (even if it's one that is waiting to be computed) - if none of them match the load instruction's target, then the load can take place. We would need to stall loads until all addresses are known for preceding instructions. If it matches the address of an uncommitted store, the data can simply be forwarded to the load - care should be taken to ensure we take from the most recent store (in the case multiple instructions are writing to the same address).

Stores and loads use computed addresses, which may not be known at issue time (unlike registers). Consider the following;

```
1 SD F0,0(R3) ; store F0 at address R3
2 LD R2,0(R1) ; load address from memory
3 SD F1,0(R2) ; store F1 to that address
4 LD F2,0(R3) ; load F1 from address R3
```

However, we don't know until instruction 2 has finished executing whether `R1 = R3`. We could predict that it doesn't match, and directly forward `F0` to `F2` from instruction 1 to 4 - if this is correct, we obtain a significant performance improvement (store-to-load forwarding).

There are alternatives for out-of-order processor architectures, we will look at the register update unit (RUU) versus ROB. The RUU essentially unifies the ROB and reservation stations. When the instructions are ready to run (operands populated), the dispatcher selects the next ready entry and dispatches it to a functional unit. The decision is deferred to the dispatch unit, rather than going directly to the reservation station for a functional unit. Note that the tags correspond to the indices in the RUU, rather than to each functional unit as before. As such, the ROB entry acts as renamed register for the instruction's result.

Pentium III attempted to reduce the amount of copying. The register alias table tracks the latest alias for logical registers (pointing to ROB or RRF (retired register file) - would've been commit-side registers in our previous example). Once retired, the pointer is switched to one in the RRF. In *Pentium 4 / NetBurst*, this is more explicit as there are two RATs (frontend and retirement) - the committed state of the machine is represented by the physical registers in the RF (register file, entirely dynamic) pointed to by the retirement RAT.

Watch the end of the lecture for details on *Pentium 4*.

In conclusion, dynamic scheduling reduces the dependence on the compiler scheduling instructions (and the compiler's knowledge of the hardware - compiler can generate code without knowing which processor the code is running on). It also handles dynamic stalls (for example cache misses, that the compiler would not know about in advance). Register renaming maps the limited number of registers in the instruction set to a much larger set of physical registers in the machine. However, this adds a considerable increase in the pipeline depth. Branch misprediction has higher latency. The complexity (and risk of error) is increased, as well as power consumption. Performance can also be difficult to understand.

October 19 - Live Lecture

Consider a program for no temporal locality, or one without spatial locality. An example of the former is one that never revisits data, such as something that deals with a stream of data (once it's dealt with, it's never seen again); a simple example is summing an array (assuming we've never seen the data before so it isn't already in cache). A specific example of the latter is an array of structs where the entire struct fills the cache, but only a single field is used from each struct. Another example would be a large (larger than the cache) hash table - despite the 'constant' access times. If the memory accesses are deliberately random (with little reuse), a tree structure may even be more beneficial.

In a fully associative cache, the cache index bits are unused. Instead we basically scan the tag bits for every cache location, for every access. These are expensive as many comparators are required for **every** access. The MUX on the critical path could potentially become a clock rate limiting problem. The memory stage (and instruction fetch) could likely be the clock rate limiting stage due to slower memory accesses.

Similar structures and issues found in caches will be found in branch prediction, prefetching, and so on.

Consider the following code, which adds a scalar to a vector;

```
for (i=1000; i >= 0; i=i-1)
  x[i] = x[i] + s;
```

Translated into MIPS, we have the following (assuming 8 is the lowest address). Note that it takes 9 clock cycles per iteration;

```
Loop: L.D    F0,0(R1) ; 1 stall
      ADD.D  F4,F0,F2 ; 2 stalls
      S.D    0(R1),F4
      DSUBUI R1,R1,8
      BNEZ   R1,Loop
      NOP             ; delayed branch slot / stall
```

This could be rescheduled to the following, to have 6 cycles (albeit 3 of these are used for the loop overhead);

```
Loop: L.D    F0,0(R1) ; 1 stall
      ADD.D  F4,F0,F2
      DSUBUI R1,R1,8
      BNEZ   R1,Loop
      S.D    8(R1),F4 ; this is altered since it's after DSUBUI
```

The program can be modified by unrolling the loop (each iteration does 4 steps);

```
Loop: L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F0,-8(R1)
      ADD.D  F4,F0,F2
      S.D    -8(R1),F4
      L.D    F0,-16(R1)
      ADD.D  F4,F0,F2
      S.D    -16(R1),F4
      L.D    F0,-24(R1)
      ADD.D  F4,F0,F2
      S.D    -24(R1),F4
      DSUBUI R1,R1,#32
      BNEZ   R1,Loop
      NOP
```

Our opportunities for rescheduling is limited by register reuse. This can be resolved by giving each copy of the loop a fresh set of registers (each copy of load, add, store uses a new set of registers). This moves all loads to the start, where enough time would have elapsed by the time we want to perform the additions (note that the expected `-24(R1)` has changed as it's after the `DSUBUI`, as `8 - 32 = -24`) - this becomes 3.5 cycles per iteration;

```
1  Loop: L.D    F0,0(R1)
2        L.D    F6,-8(R1)
3        L.D    F10,-16(R1)
4        L.D    F14,-24(R1)
5        ADD.D  F4,F0,F2
6        ADD.D  F8,F6,F2
7        ADD.D  F12,F10,F2
8        ADD.D  F16,F14,F2
9        S.D    0(R1),F4
10       S.D    -8(R1),F8
11       S.D    -16(R1),F12
12       DSUBUI R1,R1,#32
13       BNEZ   R1,Loop
14       S.D    8(R1),F16
```

Suppose all reservation stations are occupied. The issue is that we have nowhere to now put the instructions, leading to a stall at the issue stage. As the issue stage is an in-order stage, this prevents overtaking (for example, if later instructions don't require the 'full' reservation stations they won't be seen). These reservation stations are a factor for performance.

The difference between `MM1` and `MM2` is the order of the matrix multiply; the former uses `i,j,k` whereas `MM2` uses `i,k,j`. The latter is better as the compiler lays out the array row-by-row (in the former, the `B[k][j]` isn't walking down a row, even though `C[i][j]` is). This is important as we're able to use spatial locality in `MM2` (we will use all the elements in the cache block before going to the next one). On the other hand `MM2` and `MM3` splits the `j` and `k` loops, where `jj` and `kk` jump by `blocksize`. The inner part of `MM3` still remains `i,k,j` (albeit doing the multiplication in smaller blocks) - known as loop tiling. By using tiles, we get around the issue of displacing tiles that we will use again in matrix multiplication.

Registers are not places to store data, but rather indicate data flow (a special-purpose machine would have a wire connecting units). The ALU is controlled by a signal derived from decoding the instruction (it is multipurpose). However, in a specialised design, we'd likely have exactly the right number of required functional units. In a specialised machine, you'd know what's in cache locations (you'd have cache data, but not the tags) - the existence of cache isn't Turing Tax, but specific choices are. Architects avoid Turing Tax in a number of ways, including SIMD (which amortises the fetch-execute over a vector of operands), more complex instructions (macro-instructions), long cache lines, direct memory access, etc. Compilers also help by vectorising instructions, and so on.

Chapter 3

Control hazards are present in any pipelined processor; we will likely need to know what instruction to fetch even before the current instruction has been evaluated, or even decoded. Branches are typically quite frequent, but can vary depending on the type of program. Amdahl's law suggests that whatever can't be accelerated becomes dominant, when everything else has been accelerated. Speculative dynamic instruction scheduling, along with register renaming, allows us to speculate many instructions, even through several branches. Not only is it important for branches, but other parts of the processor architecture can also benefit from hardware support for dynamic prediction. Virtual function calls (in a polymorphic language) can also depend on the context and may vary from invocation to invocation - they are also indirect jumps.

Alternatives for branch prediction include;

- **multithreading**

  Suppose we have a simple 5 stage pipeline and 4 threads per core. As such there are 4 PCs and 4 sets of registers. By the time thread 0 is executing again in the fetch stage, we would've already determined the outcome of the first instruction - it should've completed the ALU stage. With enough threads, the entire control problem disappears - however we care about the performance of individual threads.

- **predication**

  Predication extends the instruction set by loading condition values into predicate registers (single bit). For example, the simple code `if (x == 10) c = c + 1;` could become the following;

```
:
      LDR r5, X
      p1 <- r5 eq 10
<p1> LDR r1 <- C
<p1> ADD r1, r1, 1
<p1> STR r1 -> C
:
```

  The conditional body consists of instructions guarded by these conditional predicate registers. The control hazard is converted into a data hazard. By doing this, we avoid conditional branches, however this may be unattractive for large bodies of code - it would be better to jump over the entire body rather than fetching and decoding, and then realising the predicate is false. It also eliminates the risk of a misprediction.

- **branch delays**

  The branch instruction is redefined to take place after a following instruction.

```
# source code
if (R1 == 0)
  X = 100
else
  X = 200
R5 = X
```

```
# assembly code
  LW R3,#100
  LW R4,#200
  BEQZ R1,L1
  SW R3,X
  SW R4,X
L1:
  LW R5,X
```

```
# what it does
if (R1 == 0)
  X = 100
else
  X = 100
  X = 200
R5 = X
```

  The instruction after the branch `SW R3,X` is executed regardless; it's already been fetched. If we are branching, then we go to `L1` after it's executed, otherwise we proceed to `SW R4,X`. For MIPS, it's sufficient to have a single delay slot.

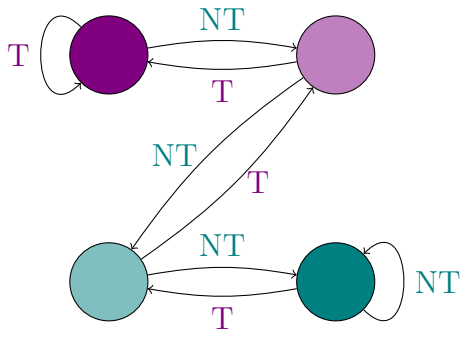
  Ideally, we take an instruction from before the branch instruction. Alternatively, we can take an instruction from either the target address, or from the fall through (although this is only valuable when it's correct) - we also want to ensure that it's the correct thing to do (hence we shouldn't store to memory, as we may store something that shouldn't be stored). The performance of this can be improved with 'cancelling' branches, with variants for whether the branch is likely taken or not. With these instructions, the write-back is disabled if it wasn't supposed to be executed.

Ideally, with a branch predictor, we want to fetch the correct next instruction without stalling. Therefore, we need a prediction before the preceding instruction has even been decoded. There are two types of branches (conditional branches) which is direction prediction (whether it's taken or not), and indirect branches which is target prediction (definitely taken, an example of which is a branch to an address stored in a register, commonly present as virtual function calls or return addresses which is taken from the stack).

Branch Direction Prediction (Takenness)

The simplest idea is to keep a table for each branch containing a bit representing whether it was taken or not last time (branch history table - BHT). If we take the k lowest bits from the program counter to index a table of 2^k size. Once an instruction is completed, we update the corresponding entry with a 1 or 0 depending on whether it was supposed to be taken or not. One limitation of this is that the table is of limited size, hence multiple branch instructions may map to the same entry - same issue as associativity conflicts. A simple example where this may have issues is a nested loop; the first iteration of the inner loop will likely miss, as well as the last iteration of the inner loop (would've had taken in the BHT for the previous iterations, but this will fall through and update to not taken). The next iteration of the outer loop will also cause a miss in the inner loop (since the branch is taken, but the BHT suggests it won't be taken, as it fell through last time).

A solution to the previous idea is to add hysteresis by having a second bit. This scheme only changes the prediction of there are two mispredictions (violet represents predict taken and teal represents predict not taken);



This idea also generalises to higher bit BHTs. Generally, the 2-bit predictor is often very good but can sometimes be quite awful. 1-bit is usually worse, and 3-bit isn't usefully better. This is based on a saturating counter. Most branches are biased; they are either almost always taken or almost always not; this doesn't work for branches which aren't biased. The predictor is often called the bimodal predictor (two clusters).

The bimodal predictor uses local history; the prediction is determined only by a memory address. Consider the following code, with three conditions;

```
1 if (C1) then
2   S1;
3 endif
4 if (C2) then
5   S2;
6 endif
7 if (C3) then
8   S3;
9 endif
```

It's likely that the conditions are correlated; for example, we may not know `C1`, but once we do, we may know `C2` and `C3`. We define the **global history** as the taken / not-taken history for all previous branches. The idea is to keep an m-bit branch history register (BHR), which is a shift register (push a zero or one) recording the taken / not-taken direction of the most recent m branches.

Consider an (m, n) "gselect" predictor. There are m global bits (BHR) recording the behaviour of the last m branches which is used to select which of the 2^m n-bit BHTs to use. A popular choice is $m = n = 2$, hence there are 4 tables of 2×2^k bits. The update is also done based on the same BHR, so a different history would lead us to read a different table.

Variations of this include the global scheme, which is an extreme case where only the global history is used to index the BHT, ignoring the PC. XORing the lower PC bits with the BHR is called "gshare". The use of per-address pattern history is quite popular and effective.

The lecture goes over a few examples of benchmarks, and importantly how it can skew decisions. It's important to evaluate the appropriateness of the benchmarks used and the conclusions drawn, especially where there are outliers.

The idea of a tournament predictor involves having two predictors; one based on global information and one based on local information. A selector is put in place, which decides which predictor to use (the selector is also driven by another predictor, which ideally selects the right predictor for the right branch). We still want to update both predictors, even if one gave us a value we didn't use, and also update the predictor for the selector (to tell it whether the chosen predictor gave the correct result). The profile based prediction is static in the sense that the prediction is encoded in the instructions, but is still derived from a real execution. Note that the tournament predictor beats the profile-based predictor, which beats the 2-bit counter. A good dynamic predictor can outperform a static prediction; by profiling we lose context information.

Branch Target Prediction

The key idea is a branch target buffer. This is essentially a table at the instruction fetch stage, which given the current PC should tell you what the next PC should be; while we are fetching an instruction, we also want to guess what the next instruction to fetch would be. This is indexed by the low-order PC bits and tagged with the high-order bits. If there is a miss (the entry doesn't exist), the branch isn't predicted and proceeds normally (increment PC by 4). However, if we do have a hit, then the predicted PC is used as the next PC. For a miss, if we discover the PC is a branch **and also** taken, then we update the BTB. Similarly, it is also updated when it's an indirect branch such as a switch statement or indirect function call.

This is accessed in parallel with the instruction cache in the fetch stage. In the next stage, at decode, we also find out whether it was a misprediction or not. If it was a misprediction, we squash the instruction, allowing it to progress through the pipeline but disabling the memory and write back stages, and then go back to fetch the correct instruction.

This is combined with the direction predictor from earlier this chapter. The direction predictor is checked simultaneously with the BTB. If there is a hit in the BTB, it gives us the destination of a conditional branch at this PC if the branch is taken. If the direction predictor predicts that the branch is taken, then we can use the destination from the BTB. Note that a good predictor can be larger and slower. One approach for this is to use a fast direction predictor with a BTB predictor, and also access a slower predictor in parallel. This is more valuable in larger pipelines, or a dynamically scheduled processor, as we can re-steer by squashing the initial prediction and fetching with the improved prediction (losing 1 cycle, rather than full penalty).

In a dynamically scheduled processor, we need to consider when to update the branch predictor. The branch outcome may be known before the branch is committed - the branch predictor can be updated either when the outcome is known, or the branch is committed. It's tempting to think that we should update the branch prediction early, as we may be executing in a loop which may have correlation with the branch outcome. On the other hand, there may have been a misprediction, which should not have been updated to the predictor - leading to future mispredictions.

Return addresses are simply indirect jumps, but should be more predictable. As such, many modern processors have special units to handle return addresses. Consider the following code - with the BTB, the second time `F` is invoked, it will get the return wrong as it would've learnt the return to the first call;

```
1 F:
2   body of F
3   ...
```

```
  RET

JSR F
next instruction
...
JSR F
next instruction
```

In x86, `JSR` pushes the return address to the stack and `RET` jumps to the address at the top of the stack. On MIPS, `JAL` (jump and link) jumps and stashes the current PC to a special register `$ra` (no memory accesses associated with performing a function call, unless it's a nested call which would require `$ra` to be pushed on to the stack, or in another register). Consider nested function calls, such that `F` calls `G` which calls `H`; the return addresses form a stack (even if they are actually on registers), and should be easy to predict. Another branch target predictor needs to be added, which maintains a hardware stack of return addresses. The return address predictor (RAP) has the most recently pushed address at the top of the stack. The value at the top of the RAP stack is used as the predicted next PC when the BTB predicts the current instruction to be a `RET`.

When an instruction is successfully decoded, we need to pop the value off the RAP stack when it's confirmed to be a `RET` or push the current PC to the stack if it's confirmed to be a `JSR`. This is done **after** the misprediction check, as we don't want to update the RAP when it's incorrect.

If the call stack is deeper than the RAP stack size (which is fixed and typically quite small), then it will be empty on a return. It's also possible for the RAP to be wrong in the case that the top of the stack has been overwritten (to a new address we should jump to). Another case is that the stack pointer changes, which could happen when a context-switch takes place and switches to another thread. If the return address stack (RAS) is updated before committing (from a speculative execution), it's possible that a `JSR` is mispredicted (leading to the RAS being updated even though no jump has taken place).

October 26 - Live Lecture

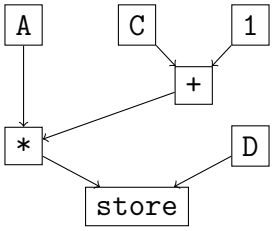
Blocking can be thought of as a transformation on loop nests. The first transformation is referred to as strip mining; the order of execution hasn't changed at all (an outer loop incrementing by S). Strip mining a single loop is as follows;

```
for (k = 0; k < N; k++) {
  ...
}
// becomes
for (kk = 0; kk < N; kk += S) {
  for (k = kk; k < min(kk + S, N); kk++) {
    ...
  }
}
```

Interchanges are applied to move the mined loops to be the outermost loops (safe to interchange loops). The inner `i,k,j` performs a multiplication on a pair of partial matrices, S is chosen such that an $S \times S$ submatrix of B and a row of length S of C can fit into the cache (the reused data, a submatrix, fits into the cache). Note that in `MM3`, S was chosen to specifically not be a power of 2. In `MM4`, each row of the submatrix is mapped to the same cache line.

The benefit of predicted execution is to avoid control hazards. By performing predictions, we are able to fill the machine with work, rather than idling waiting for the result of an execution.

Consider different machines, which all perform the same task. Note that code essentially encodes a graph;



- **register machine**

```
1 load  r1 A
2 load  r2 C
3 add   r3 r2 #1 // r3 = C + 1
4 mul   r4 r1 r3 // r4 = r3 * A
5 store r4 D     // D = r4
```

- **stack machine**

```
1 push  A        // load from A, push to stack
2 push  C
3 add   #1       // add 1 to the value on the top of stack
4 mul            // multiply the top two values on stack
5 store D        // store value on top of stack to memory
```

  This can be difficult to trace. It also has difficulty with copied operands (if an operand needs to be used twice).

- **dataflow machine**

```
1 load  +3 A     // load from A, send to [mul]
2 load  +1 C     // load from C, send to [add] (next)
3 add   +1 #1    // add 1 to value received, send to [mul] (next)
4 mul   +1       // multiply values recieved, send to next
5 store D        // store value received
```

  Recent dataflow instruction sets, such as *EDGE*, *TRIPS*, and *Wavescalar*, bundle instructions into statically scheduled dataflow fragments. Dynamically scheduling then occurs on the fragment level.

- **belt machine**

```
1 load  A        // load A onto belt
2 load  C        // load C onto belt
3 add   -1 #1    // add 1 to the result of the previous instruction
4 mul   -1 -3    // multiply the results from positions -1 and -3 on the belt
5 store -1 D     // store the previous result to memory
```

  The idea is that every instruction will write to a fresh register, with a finite (rolling) window of registers. Operands are collected by an offset backwards to the instruction producing the operand. There is very little energy consumption at the rename logic stage of the pipeline (almost none), compared to classical processors.

  The compiler needs to know the size of the belt. For example, if we know an instruction will need something that will fall off the belt, it can be moved to the front of the belt. Another approach is to spill it.

  Branches can have issues with this (and the dataflow machine). If two branches both generate a value, the compiler will need to think about what values will be used downstream and have an additional copy instruction that moves it to a known location (forces the value to be completed).

The lecture then discusses examples of indirect jumps with a `switch`.

Chapter 4

The objective is to reduce the average memory access time, which is equal to HitTime + MissRate × MissPenalty. This can be improved in three ways;

1. reduce miss rate (hardware or software)
2. reduce miss penalty
3. reduce the time to hit in the cache

Cache Miss Rate Reduction in Hardware

Misses can be classified into one of four classes (the last one isn't always included);

- **compulsory** - experienced the first time the data has been seen in the machine (thought of as misses in an infinite cache)
- **capacity** - cache cannot contain all the blocks needed (thought of as misses in any cache of that given capacity, misses in a fully associative cache)
- **conflict** - caused by a compromise to the associativity of the cache
- **coherence** - other processor / device has invalidated the data

When the cache size is small, the proportion of miss rates is dominated by capacity misses, however as it gets larger, compulsory misses and conflicts take up a larger proportion. Note that CPU benchmarks can often have low compulsory misses as they load some data and perform calculations on it.

The lecture goes into the CPU benchmark suite by SPEC. Note that the main difference between integer and floating point benchmarks is not the type of arithmetic they perform. The former tends to intensively use pointers and hard-to-predict branches - they are difficult to parallelise either manually or by an automatic compiler. On the other hand, the latter tends to be more structured (control flow), operate on regular array data (rather than complex data structures), and benefit more from automatic parallelisation. There are also two types of benchmarks, speed (where we look at the execution time for one run), and rate (where we look at the maximum throughput, for example with a multicore machine). The SPEC benchmarks use the geometric mean of speedups (ratios) relative to the baseline.

Comparing two different processors in the benchmarks show that architectural techniques benefits some workloads more than others - we should consider what the code is doing in the context of the architectural techniques in use.

Consider the first 3 miss classes, assuming the total cache size isn't changed, and what happens if each of the following is changed;

1. **block size**

   Recall that the cache block / line is a unit of allocation, the tag is a memory address of a cached block, a set is a set of cache blocks indexed by a cache index, and a way is a set of alternative locations for a stored block in a given set. There are also comparators and selectors, with the former verifying the tag matches an address, and the selector picking the data from the way that has the matching tag. We use the index to select a set and then use the comparator to check each way for a hit.

   For a random benchmark, consider a (fixed) cache capacity of 16KB. On the graph, there's a minimum miss rate at a block size of 64B; after which point as the block size increases, the miss rate increases. Consider the extreme case, of a 1KB capacity and 256B block size. In this scenario, we only have 4 distinct regions, and if the program accesses more than that, we will end up with misses. The problem with very large cache lines is that we may speculatively fetch data into the large line and never use it (poor utilisation).

   Note that the graph in the slides also doesn't include the time; a larger block will take longer to load (assuming the bus width isn't changed) - this leads to a higher miss penalty.

2. **associativity**

   A set-associative cache has more complex circuitry than a direct mapped cache (there's a MUX in the former). While we may run into associativity conflicts, it may still have a faster speed due to the simple circuitry. We assume that the cycle time gets worse as associativity increases; the miss rate is improved by increasing associativity, but the cache hit time is also increased slightly. This could be improved with way-prediction (similar to squashing the hit in a direct-mapped cache).

   We ideally want the access time for a direct mapped cache and the miss rate of an associative cache. An approach to doing this is to have two caches; one that's large and direct-mapped and another that's very small but fully associative. Both caches are accessed in parallel. When we index the direct mapped cache, we check if we have a hit - if it misses, we check the victim cache. When we allocate into direct mapped cache, we displace the entry (if any) into the victim cache (recall the displacement causes the associativity conflicts). On a victim cache hit, the data is allocated back into the direct-mapped cache.

   This is a competitive algorithm. Given two strategies, each good for some but poor for others (direct-mapped and fully-associative). We want to combine the two to create a composite strategy which cover each other, so we never suffer the worst case behaviour. Consider the ski rental problem.

   In a skewed-associative cache, we use different indices in each cache way (for example a simple hash function, such as `XOR` some index bits with the tag bits and reorder index bits). This is in contrast to the set-associative cache where the same index is used. Consider a 2-way cache with three addresses, for the first function there's a chance that they may still conflict, but it's unlikely that the second hash function will cause another conflict (hence $f_0(\texttt{A}) = f_0(\texttt{B}) = f_0(\texttt{C})$ but $f_1(\texttt{A}) \neq f_1(\texttt{B}) \neq f_1(\texttt{C})$).

   The same idea is present for arrays; if we are really unlikely and get $f_0(\texttt{A[i]}) = f_0(\texttt{B[i]}) = f_0(\texttt{C[i]})$ and $f_1(\texttt{A[i]}) = f_1(\texttt{B[i]}) = f_1(\texttt{C[i]})$, it's very unlikely we end up with $f_0(\texttt{A[i + 1]}) = f_0(\texttt{B[i + 1]}) = f_0(\texttt{C[i + 1]})$ and $f_1(\texttt{A[i + 1]}) = f_1(\texttt{B[i + 1]}) = f_1(\texttt{C[i + 1]})$ (since the hash functions are pseudo-random). In accessing arrays, if the addresses are exactly far enough apart, it could be in the case where all accesses conflict (worst case scenario) - the skewed-associative cache gets around this.

   The paper on the statistical model of this concludes that we get the miss rate of a more associative cache with fewer ways. This also gives us more predictable average performance. However, it requires a decoder per way, some latency is added for the hash function, and LRU is harder to implement.

3. **compiler**

Misses can also be reduced in hardware by performing prefetching. After a cache miss, the stream buffer initiates a fetch for th next block (not allocated in the cache). On an access, the stream buffer is checked in parallel with the cache. The stream buffer is a FIFO queue. Similar idea to a victim cache. Modern processors do a similar technique, but allocate into the cache - with more associativity, cache 'pollution' is less of a concern.

This idea could be extended to track multiple access streams (one stream is good for instruction misses, multiple streams can be important for data, such as multiple arrays).

Decoupled access-execute uses the idea that most programs have almost no dependence between floating point calculations and the sequence of addresses issued into the memory system. The address calculation issue instructions can be separated from the floating point arithmetic instructions that operate on the fetched data. As such, there are two processors, one for access and another for execute. The address-generation side can run ahead of the execution, allowing for values to be streamed into the execute processor (exploiting memory parallelism / pipelines). Loss of decoupling events can occur when the control flow depends on the execution results.

Cache Miss Rate Reduction in Software

The prefetching mentioned in the previous lecture could be left entirely to the programmer and the compiler; many processors have software prefetch instructions. However, since hardware prefetching is generally quite good, there is significant overhead that may outweigh the benefit (slowing down the overall process) - it may be useful for simpler processors. The MIPS `memcpy` library code also prefetches the destination as every store implicitly has a cache load (hence both the source and destination need to be prefetched).

If the prefetch would've generated a fault (accessing something illegal), we don't want the prefetch to fail but we also don't want the prefetch to fail either (it should be silently squashed).

Instruction caches are also quite important, but are left out by benchmark suites as they tend to be quite small. *McFarling* reduced instruction cache misses by choosing an instruction memory layout based on the call graph. Procedures are reordered in memory to reduce conflict misses.

Data optimisations can also take place;

- **storage layout transformations** spatial locality

  Don't change the code, just how the variables are declared and the addresses in which they are stored.

  An example of this is array merging; an array of structs versus a struct of arrays.

  ```
  // before (2 sequential arrays)
  int val[SIZE];
  int key[SIZE];

  // after (1 array of structs)
  struct merge {
    int val;
    int key;
  };
  struct merge merged_array[SIZE];
  ```

  The spatial locality can be improved, depending on the use case. A counter-example is if we were to traverse the `key` array and get the `value` out, then the former should be stored contiguously. However, if this is being used as a hash table for example, it would beneficial for the key and the value to reside in the same cache line (more likely with the array of structs).

  Row-major mapping maps indices horizontally, and similar for column-major (but vertically). The Morton / quadtree layout gives spatial locality in two dimensions, giving a compromise between the two previous layouts. A variant of this is used for texture caching by some GPUs.

- **iteration space transformations** can also improve temporal locality

  Change the order in which loops are executed.

  An example of this is loop fusion;

  ```
  // before
  for (i = 0; i < N; i = i+1) {
    for (j = 0; j < N; j = j+1) {
      a[i][j] = 1 / b[i][j] * c[i][j]
    }
  }
  for (i = 0; i < N; i = i+1) {
    for (j = 0; j < N; j = j+1) {
      d[i][j] = a[i][j] + c[i][j]
  ```

```
  }
}

// after fusion
for (i = 0; i < N; i = i+1) {
  for (j = 0; j < N; j = j+1) {
    a[i][j] = 1 / b[i][j] * c[i][j]
    d[i][j] = a[i][j] + c[i][j]
  }
}

// after array contraction
for (i = 0; i < N; i = i+1) {
  for (j = 0; j < N; j = j+1) {
    cv = c[i][j]
    a = 1 / b[i][j] * cv
    d[i][j] = a + cv
  }
}
```

After fusion, it should be faster due to less loop overhead, but also we avoid loading `a` from memory, as well as reloading `c`. The second transformation is safe if we don't need the array `a` after the loops have been executed; values can be transferred in scalar values rather than an array (contraction). However, this (fusion) isn't always simple - for example a one-dimensional convolution / stencil (dependencies don't align nicely);

```
// before
for (i = 1; i < N; i++) {
  V[i] = (U[i - 1] + U[i + 1]) / 2
}
for (i = 1; i < N; i++) {
  W[i] = (V[i - 1] + V[i + 1]) / 2
}

// after
V[1] = (U[0] + U[2]) / 2
for (i = 2; i < N; i++) {
  V[i] = (U[i - 1] + U[i + 1]) / 2
  W[i - 1] = (V[i - 2] + V[i]) / 2
}
W[N - 1] = (V[N - 2] + V[N]) / 2

// contraction is harder, need last two Vs (3 locations, use 4)
V[1] = (U[0] + U[2]) / 2
for (i = 2; i < N; i++) {
  V[i % 4] = (U[i - 1] + U[i + 1]) / 2
  W[i - 1] = (V[(i - 2) % 4] + V[i % 4]) / 2
}
W[N - 1] = (V[(N - 2) % 4] + V[N % 4]) / 2
```

This can be made fusable by shifting; the middle part of the loop is fusable, but there's some edges to manually perform.

Miss Penalty Reduction

A write-through policy involves writing to underlying layers; cached data can therefore always be discarded and the control bit only includes a valid bit. On the other hand, a write-back policy only updates the cache; cached data may have to be written back if discarded and control bits include both the valid bit as well as a dirty bit. The former has the advantage that the memory has the latest data (note that processors may have their own cache) and is simpler. On the other hand, the latter has lower requirements on the write bandwidth as the data is often overwritten multiple times - it also has better tolerance for slower memory.

There are also policies in place for the case where it's a write-miss. Write allocate allocates a new cache line on a miss, whereas write non-allocate / write-around simply writes the data to the underlying layers (no new cache line is allocated). The former might make more sense for spatial locality, as something that's written could be read again soon.

Consider a write buffer, sitting between the cache and lower memory. The purpose of the buffer is that we don't need to wait for the slower memory to write the data - however, this introduces yet another potential copy of this data. This applies regardless of write-back or write-through. Consider a program that's writing to a vector; ideally it's merged into a single entry in the buffer and written to the next level of the memory hierarchy only once.

Typically on a miss, the entire cache block isn't loaded in parallel but rather word-by-word. Consider the case where the requested word is the first word in the block; ideally the processor can continue as soon as the requested word arrives (early restart). A more general / powerful idea is to have the critical word first; request the required word from the next level of the memory hierarchy first and then continue to populate the rest of the block. This is only really worthwhile in large blocks.

This can also cause issues; especially in the case where the processor restarts and the next instruction also loads into another word in the **same** cache line. Each sector (or word) needs a validity bit; the cache block is the unit of allocation and deliver / transfer data in sectors. This validity bit tells us whether the sector has been updated already.

Another strategy is to have a non-blocking / lockup-free cache. This allows the cache to continue to supply hits during a miss - it requires full / empty bits on registers or out-of-order execution and multi-bank memories. The idea of hit under miss reduces the miss penalty but allowing work to continue during while a miss is being handled. Hit under multiple miss or miss under miss lowers the effective miss penalty further my overlapping multiple misses.

For an in-order pipeline, there are two options on a cache miss. The first is to freeze the pipeline in the memory access stage, stalling any other subsequent instructions. The miss status / handler registers (MSHR) keeps track of outstanding memory accesses and the registers waiting for these results, and only stalls the pipeline when an instruction references something that's needed. This provides some limited out-of-order execution (for loads).

Hit under miss creates the opportunity for load instructions to execute out-of-order. From the thread running on the CPU, care will be taken to ensure the results from the load are the most recent data for that location written by this thread. However, in a multicore system, or a system with a DMA / memory mapped IO device, care should be taken knowing that load instructions can be reordered by this mechanism. Special instructions to be used to enforce the relative ordering of memory accesses, if we care.

Another approach would be to add a second level cache. Let AMAT be the average memory access time, HT be the hit time, MR be the miss rate, and MP be the miss penalty;

$$\begin{aligned} \mathrm{MP_{L1}} &= \mathrm{HT_{L2}} + \mathrm{MR_{L2}} \times \mathrm{MP_{L2}} \\ \mathrm{AMAT} &= \mathrm{HT_{L1}} + \mathrm{MR_{L1}} \times \mathrm{MP_{L1}} \\ &= \mathrm{HT_{L1}} + \mathrm{MR_{L1}} \times (\mathrm{HT_{L2}} + \mathrm{MR_{L2}} \times \mathrm{MP_{L2}}) \end{aligned}$$

The local miss rate is the misses in the cache divided by the total memory access to a given cache. On the other hand, the global miss rate is the misses in this cache divided by the total memory accesses from the CPU (this is what matters).

The lecture continues for a bit discussing *Haswell*'s cache hierarchy.

Multi-level inclusion states that the L_{n+1} cache contains everything in L_n. We may sometimes allocate into L1 but not L2, or the other way around. We may also allocate into both L1 and L2, but not LLC. LLC (or L3) is sometimes used as a victim cache, once data is displaced from L2. If we don't assume multi-level inclusion, it's perfectly valid for L2 to displace something that's frequently being accessed in L1, as it sees no hits for that particular line. However, if we do rely on inclusion, then we need to track which lines cannot be displaced.

Cache coherency is another argument for MLI; consider two cores each with its own L1 and L2 cache. Suppose the first core writes to an address that may be held in one of the caches of the second core. The second core needs to monitor all the bus traffic (loads, stores, accesses) of the other processor - if it sees an invalidation of a particular address, the second core's cache controller needs to check whether it has a cache copy of that address; if it does, then it needs to be invalidated. However, the L1 cache is typically very busy with processor traffic and shouldn't be touched. By relying on MLI, the L2 cache can be used to filter those invalidations.

Hit Time Reduction and Address Translation

Consider the 8 stage pipeline in the *MIPS R4000* (extended from the 5 stage pipeline of the *MIPS R3000*);

- **IF** - first half of instruction fetch, PC selection happens here, also initiation of instruction cache access
- **IS** - second half of access
- **RF** - instruction decode, register fetch, hazard checking, and instruction cache hit detection
- **EX** - execution (effective address calculation, ALU operation, and branch target computation and condition evaluation)
- **DF** - data fetch (first half of access to data cache)
- **DS** - second half of access
- **TC** - tag check, determine whether the data cache access was a hit
- **WB** - write back for loads / register-register operations

Higher clock rate required deepening the pipelining, by splitting the fetch stages into two clock cycles (pipelining the cache itself). The DS stage could forward the instruction to a succeeding instruction, before the check is complete. If the check fails, then an additional cycle penalty is introduced.

Consider the case where the first instruction is a load and the fourth instruction is an arithmetic operation that uses the result. The result can be passed into the execute stage of the fourth instruction before the tag check is complete. By designing a deeper pipeline, the clock cycle can be increased, but can potentially result in two stalls (e.g. if the instruction using the result was right after, whereas the 5 stage pipeline only had a single stall cycle). Similarly, this happens with the branch latency, if the conditions are evaluated during the EX phase.

With the *R4000*, stalls caused by waiting for results from FP units / access to FP hardware, account for a large amount of the lost performance, especially for floating point workloads. The next architecture from MIPS was an out-of-order processor.

Currently, we're discussing reducing the hit time by pipelining, but it doesn't reduce the access latency (only throughput). With dynamic scheduling, this can still help. However, a bottleneck may arise from the bandwidth of the cache, especially in a processor that issues multiple instructions per cycle. A number of things can be done to support parallel access to the cache;

- divide cache into several banks

  Consider how each address can be mapped to a bank (low order bits of the cache index, etc.). For example, a bank for even or odd addresses. If addresses fall into odd/even pairs, then those accesses can be overlapped.

- duplicate the cache

  Stores are copied into both duplicates, but loads can be sent to whichever cache is idle.

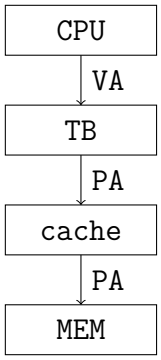
- multi-ported RAM (support two reads to different addresses very cycle)

  A RAM array has a wordline per row, and a bitline per column. In a multi-ported design, there would be multiple wordlines per row and multiple bitlines per column - allowing different rows and different columns to be activated simultaneously.

Simple processors can access memory directly (also what we've assumed so far) - the addresses are generated by the processor and directly used to access memory. However, we may want to run code in an isolated environment, preventing a crash in the code from bringing down the entire system, or preventing malicious code from having full access. Another use is to allow more than one application to be run at a time, preventing interference with each other, or even knowing about each other (each occupying a separate virtual machine). A major use is to use more memory than DRAM (for example, extending the memory hierarchy to HDD). Virtualising the memory can be done by adding address translation.

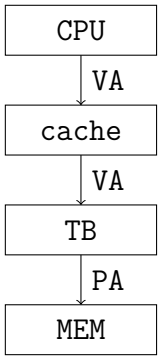
Some approaches of this are as follows (note that the translation is done at `TB`);

- **physically-indexed, physically-tagged**



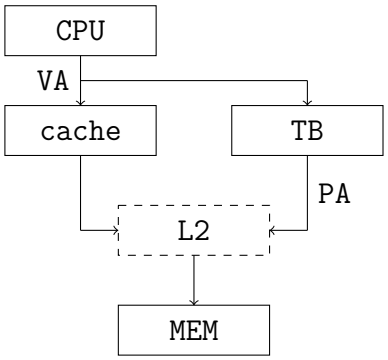
The processor operates on virtual addresses in its protected virtual environment. After translation, we get physical addresses to access main memory. The entire memory hierarchy operates on physical addresses after translation.

- **virtually-indexed, virtually-tagged**



Here, we only do a memory access on a cache miss. As such, the cache ins virtually indexed. On a cache miss, the addresses is then translated to access main memory. The advantage is that it has a better hit time, as it doesn't always have to go through translation (we can get an L1 cache hit without translation). This can run into issues when the processes switch (cache contains data from process being interrupted) - causes correctness (synonym / homonym) problems.

- **virtually-indexed, physically-tagged**



We want the hit time performance of VIVT with the semantic cleanliness of PIPT. The key idea is that the translation is done in parallel with the L1 cache access. The L1 cache accesses uses the low order bits to generate data as well as a tag. Meanwhile, the translation translates the page number part of the address, and produces a physical translation of the page number. The physical address can be compared with the tag. If we have a hit, we have a low latency hit, otherwise, on a miss we have a physical address to access the L2 cache.

The original motivation for address translation was virtual memory - creating a virtual address space for a process, larger than the physical memory. Translations map virtual addresses either to physical pages in the RAM of the machine, or inactive pages stored on the swapping disk. Ideally, the process runs quickly the majority of the time reading from RAM, but may require a page that is inactive, causing a fault and raising an interrupt. The OS then deals with it and hopefully finds the data on the drive, and allocates it to a vacant page in the memory (otherwise fail). The allocation to a page in memory corrects the mapping.

The processor produces a virtual address, that consists of the page number `P` and `W` (word number) - an offset in that page. Note that `B` denotes the page frame address. The processor also has a pointer to the **current page table** (process page table), which contains a mapping from page numbers to physical locations (long with some metadata).

This can be sped up by adding caches (including caching the address translation). This is done with a TLB (translation lookaside buffer), containing recently accessed page table values; this table (also `TB` before) is typically highly associative and closely integrated with the L1 cache.

We want one computer to be able to run two processes in isolation from each other, without knowing about each other. We don't want to care about what machine we have, nor do we care about what other processes are running. Each process has a virtual address space (begins at zero), which maps to different physical addresses. Another use case is that two processes may share the same code (such as shared libraries which are dynamically linked); but have different data. Ideally, we don't want the shared code (which is read-only) to be in different locations, but rather share the same physical location.

When memory is allocated, the process asks the OS to allocate a virtual address space, which needs to be zeroed (cleared). The OS commonly allocates a single **shared** frame that is zeroed out, and points the virtual addresses to it. However, when the process begins to write, a fault occurs and a new frame is allocated, with the mapping fixed. Permissions need to be associated with these pages; a bit allowing writing and a bit to allow reading (hence all initial zeroed pages need to be read-only, after the mapping is updated, the bit is flipped). This trick is known as copy-on-write.

Another use is to have two processes sharing the same memory-mapped file, with both processes having both read and write permissions.

In address translation, there can be issues;

- **homonyms** (same sound, different meaning)

  Same virtual address pointing to two different physical addresses in different processes. If we have a virtually-indexed cache, it must be flushed between context switches (expensive, especially for a large cache), or a process ID needs to be included in the cache tag.

- **synonyms** (different sound, same meaning)

  Different virtual addresses (different processes, or even the same process) pointing to the same physical address. In a virtually-indexed cache, a virtual address could be cached multiple times under different physical addresses, and updates in one copy may not be reflected in the other. A

solution is to prevent synonyms from co-existing in the cache (if they are in the cache, then it must map to the same location); the OS could force synonyms to have the same index bits in a direct mapped cache (page colouring).

A 32-bit address consists of a 20-bit page number, followed by a 12-bit page offset (not translated). In the idea of a VIPT system, only the page offset (untranslated) is used to index the cache. In parallel, the page number is used to index the TLB, to obtain a translated page number (physical address). The translated physical address can be checked against the tag from the L1 cache - if these match, then we have a hit (low hit penalty). However, if it's a miss, we have a translated address that can be sent to the L2 cache. However, this limits the cache to the page size, since we can only use untranslated cache index bits - there are two options to use bigger caches;

- **higher associativity**

  Multiple ways could be indexed with the same bits (not requiring more bits), but having multiple ways gives us a bigger cache. This is quite commonly done, and as a consequence, L1 data and instruction caches are often highly associative. The associativity and the size matches can also be seen to match the aforementioned constraint.

- **page colouring**

  This requires the OS to help. A (cache synonym) conflict occurs when two cache blocks have the same tag (physical address) are mapped to two different virtual addresses.

  Consider a 14-bit cache index, which is more bits than the 12-bit page offset (untranslated) from the virtual address. The two bits are from the untranslated page number, which will then be translated. As such, we need to ensure that when we choose how data is allocated into memory that we don't have the consistency problem as a result.

  A case for this is when two processes map the same file into memory; ideally we'd want their virtual address spaces to be the same. This is a bigger constraint than required; we just need to ensure that the extra bits used match (if this is satisfied, we don't have the synonym conflict problem).

  The `mmap` system call in Linux has the operating system choose an implementation dependent mapping constraint.

Going back to the VIPT structure, the virtual addresses index the cache using the untranslated bits. Whether we get a conflict in the L1 cache is independent of any translations. However, associativity conflicts in the L2 cache depend on how the addresses are translated; they may be translated in such a way that means the cache index bits in the L2 cache are the same. This means that running the same program on the same data could lead to different associativity conflicts, due to different virtual-to-physical mappings.

The OS typically has heuristics to avoid this - choosing non-conflicting pages for frequently-accessed data (page colouring for conflict avoidance).

TLBs may need to support mappings for different size pages, for example the *Haswell* TLB needs to support 128 mappings for 4KB pages and 8 2MB page mappings in the ITLB, as well as 64 mappings for 4KB pages, 8 2MB page mappings, and 4 mappings for 1GB pages in the DTLB.

DRAM and Memory Parallelism

The DRAM device is a large array of cells holding data. DRAM emphasis capacity, leading us to have a single transistor design where data is stored on a capacitor. It's a two dimensional array, where horizontally we have wordlines and vertically we have bitlines. Data is stored as a charge on the capacitor - when we activate a wordline, an entire row of cells is activated, and charge flows along all of the bitlines. The row address goes into a DEMUX, which selects the row, and the charges flow

into a sense amplifier (comparator), which goes into a latch for all the bits in an entire row. The column address bits go into a data selector (a MUX), which routes the data from the latch into the output. However, note that once the row is read, the data is lost from the capacitors. Once the data is latched, the switch is enabled on RAS and data is **written back** into the memory cell - the transistor is reactivated and the charge is allowed to flow in the opposite direction to recharge the capacitors. Data is written to RAM in the same way.

However, the charge may leak out in a number of ways. As such, the DRAM needs to be refreshed (otherwise the charges may become indistinguishable). The data needs to be written back even when not accessing the rows with some refresh mechanism. As a consequence, DRAM is unavailable during refreshes, which happens every 64 milliseconds, or more frequently if the device is hot (self managed). This is typically managed by a microcontroller on the chip, and is typically done a few wordlines at a time in a cyclic fashion. This spreads out the interruption to general service over time.

Once a row is selected, the entire row is latched, allowing elements in the row (different columns) to be accessed with lower latency. For example, with a 60ns (t_{RAC}) DRAM;

- can only perform a row access every 110ns (t_{RC})
- perform column access in 15ns (t_{CAC}) but time between column access is at least 35ns (t_{PC}) - in practice, additional delays make it between 40ns and 50ns

The row access cycle time is longer than the row access time, as data needs to be written back after reading.

The architecture of a SDRAM chip may have multiple banks of DRAM arrays, each with its own sense amplifier array and individual row access latch and decoders. This allows different rows from different banks to be accessed in parallel with one another. Each bank has its own read latch and column decoder, to select data and read it out.

We want to run right up to the limit of reliability, to make the device as small as possible. As such, there's a risk that errors creep in, possibly even from interference from neighbouring cells, or radiation (high-energy cosmic rays, single-event upsets - an individual bit being flipped). The basic idea is to add redundancy through parity (for example, an even or odd number of 1s) / ECC bits. Parity doesn't allow for recovery. An example for a Hamming code is as follows; 64 data bits and 8 parity bits. Consider the data bits as forming a 64 element vector, each with one bit per element. Each individual potential 64 bit value occupies a point in 64-dimensional space. More dimensions are added into this space to separate the valid points. A SEU moves that vector from where it should be into an adjacent location in that space; we want to ensure that the adjacent locations don't correspond to a valid data value. A single error pushes us slightly away from a valid position, and can therefore be pushed back. Ideally, we want to be able to handle failures of entire chips; the idea of RAID with hard drives works here too.

We can test DRAM by trying to cause memory upsets. Imagine an array, with one row that we care about, and two adjacent rows. A program could repeatedly write into the adjacent rows, trying to flip bits in the row we care about. As the row gets closer to the refresh deadline, it becomes more vulnerable to an upset. The *Rowhammer* attack does this. This can be mitigated with ECC, or adaptive refresh which counts accesses and refreshes potential victim rows earlier.

The lecture then goes into further topics in DRAM, including energy optimisations, on-memory processing, stacking, bulk copying and zeroing, and non-volatile memory.

The simple organisation of main memory has the CPU, cache, bus, and memory all at the same widths. A wider organisation of main memory could have parallel data transfer, with the same address in all banks (with a wider bus). Another approach, an interleaved organisation, could allow multiple banks of memory to be addressed individually; parallel addressing as well as parallel data transfer. Consideration should be taken to spread accesses across the banks.

November 2 - Live Lecture

The intuition for the skewed two-way set-associative cache is that there is no restriction on what the functions are, as long as they are different. However, it also needs more than just the low-order k bits, to get more 'randomness'.

When executing a store, there is a chance the address is in the L1 cache. Consider the case where it isn't in the L1 cache. We can allocate in on a read but not a store (write non-allocate) - this may not be ideal as when we write into memory we tend to want to read it back, and the principals of locality tell us that we might want to do it soon. Consider a cache line, and storing only affecting a word at a time. A common access pattern is writing successive words; which would require multiple writes to L2 (for non-allocate).

The stream buffer assumes a fully parallel lookup.

We have several structures mechanisms that result in copies of data being in different places. We need to look at all these places in parallel;

- L1D
- write buffer
- stream buffer (prefetching mechanism)
- victim cache
- store queue

All of these structures are similar and are tag / comparator lookup checks. The store queue should be checked first (not in parallel); we should check if the data we want is generated by an uncommitted (earlier) store instruction.

Chapter 5

Microarchitectural Sidechannels - Spectre and Meltdown

A side-channel asks what we can infer about another thread by observing its effect on the system state. The instruction set definition tells us the effects an instruction is **supposed** to have, but there are effects that code can have on the system state beyond what is expected.

Suppose that thread `A` is running on the first core, with its own L1D, and there's a victim thread `B` running on a different core, with its own L1D, and there's a shared L2 cache. We control the attacker thread `A`. Assume `B` is executing some code (for example, encrypting a message) we know but don't control. Suppose `B` is executing the following, where we have the message in the `P` array, and some replacement code in the `code` array;

```
1 for (i = 0; i < N; ++i) {
2   C[i] = code[P[i]];
3 }
```

The prime and probe technique involves the following (detecting the eviction of the attacker's working set by the victim);

- attacker fills the cache by filling a set with its own lines (ideally the entire cache)
- once victim has executed, attacker could probe by timing accesses to see what was evicted
- if any evictions have taken place, the victim might have touched an address that maps to the same set

Another approach, evict and time, uses the targeted eviction of lines with the overall execution time;

- attacker causes victim to run, preloading its working set and establishing a baseline time

- attacker evicts a line of interest, such as a character we want to check if is in the message, and reruns the victim
- a variation indicates a line was accessed

Another approach, the inverse of prime and probe, is flush and reload, which relies on shared virtual memory (such as shared libraries) and the ability to flush by virtual address;

- the attacker first flushes a shared line of interest (such as the code array) by dedicated instructions or eviction through contention
- run the victim, and then attacker reloads the evicted line by touching it, measuring the time taken
- a fast reload indicates the victim touched a line whereas a slow reload indicates that it didn't

However, it may be slow to wait for the victim process to run. Instead, the attacker may want to make a system call to execute code in the OS, possibly accessing privileged data. Another example is the victim waiting on a lock; the attacker could give the victim the message and release the lock, and observe the access for encryption keys. It's also possible to call the victim code as a function. The latter is to do with language-based security, where the runtime of the programming language protects threads belonging to different security domains. For example, the victim may be an object with secret state but a public access method.

Consider a web browser (that has a JS interpreter), and two pages that require JS for rendering. The JS engine prevents page `A` from accessing page `B`'s data. This is achieved by bounds checking; for example, the call `r = A[i]` is bound checked as follows;

```
1 if (i > 0 && i < A.length()) {
2     p = &A+4*i;
3     r = *p;
4 }
```

Suppose the bounds check is **predicted** to be satisfied, but `i` is out of bounds. This leads to `s = *p`, where `s` is a secret. However, we can use `s` as an index into an array we can access, and then use timing to determine whether the cache line on which `B[s]` falls has been allocated, as a side effect of speculative execution, for example `r = B[A[i]]`;

```
1 if (i > 0 && i < A.length()) {
2     p = &A+4*i;
3     s = *p;
4     r = (B[16 * (s & 1)])
5     // check if the lowest order bit of s is 0 or 1
6     // some cache line in B is allocated into the cache
7 }
```

The flush and reload technique can be used. We need to ensure the branch predictor to expect things to be satisfied, by training it with in-bound subscripts, to trick it into performing a misprediction.

In the C code for this, `array1` is the array that will be accessed out of bounds (`A` in the previous example). There's also another `array2` that acts as a canary, where the cached-ness will be probed (in this previous example, would be `B`). After this process, some statistics are calculated to find outliers.

Different browser windows / tabs should obviously not be in the same address space, since bounds checking is insufficient.

Attacking Other Processes and the OS

Recall the bounds check code (as instructions that would go to the RUU);

```
if p is in bounds
LOAD s = *p
addr = 16*s & 1
r = B[addr]
```

The commit unit checks the head of the queue, once it is ready to commit, it updates the commit side registers (persistent state). This is also where mispredictions are detected. Assume the conditional branch isn't ready; the processor will attempt to execute the other instructions. Once the load instruction is executed, it will broadcast on the CDB and allow the address to be calculated, which then allows the memory allocation to begin. As the cache was initially flushed, this will lead to a miss; the data is then loaded and stored in the cache. Once the conditional check has failed, it will flush the results (not commit them), but it will not reset the cache allocation.

Commonly the operating system benefited from an optimisation where the virtual address space of a process would also include the OS kernel, but marked in **supervisor-mode**. The entire virtual address space is present, but accessing a supervisor-mode mapping in user-mode would lead to an exception. This allows the supervisor bit to be flipped on an interrupt and do whatever is required (accessing kernel data, etc.), and return back by flipping the bit again. It's also common for the kernel's virtual address space to include all of physical memory; this means that it's possible to capture secrets from all other user processes.

The TLB entries don't only just contain the translated physical addresses, but also include metadata; including whether a translation is valid for the current processor state - if we're in user mode, we might not be able to access a particular address translation. This is checked at the access validity check, which goes to the RUU entry of the corresponding load instruction. The designers assumed the microarchitectural state isn't observable; they assumed ti was safe to defer checking at the commit stage.

Modern operation systems have a number of mitigations which complicate the attack. This includes user-mode address-space layout randomisation (ASLR) and later kernel address-space layout randomisation (KASLR). This randomises the placement of code and data in the machines address space. This means that the attacker would have to guess where the OS kernel mappings are in the processes' address space. The distribution of code and data of the OS is randomly distributed in physical memory as well.

Kernel address space isolation (KPTI) mitigates this. The optimisation previously mentioned is given up; each process properly runs in its own virtual address space, and the OS has its own virtual address space. On a context switch, the current TLB state needs to be discarded and reload the TLB state corresponding to the address space that the processor is switching into. This gives a significant performance loss for some specific applications.

Suppose the kernel includes the following bit of code (called a **gadget**) - note that `s` is secret;

```
label:
  s = *p;
  r = B[(s & 1) * 16];
```

To perform the attack, we need the code to be executed, arrange for `p` to point to the secret, and know the address of `B` to observe the cache state. The branch predictor could be trained to mispredict a branch target to be `label`. However, we're not able to read `B` (as it's part of the victim's address space), but we can have data of our own that would conflict in the cache.

We cannot jump to any arbitrary address in the OS; instead we jump into the OS through a table, indexed by the system call number. For example, the user code may have something like the following;

```
__asm__(
    movl $20, %eax // getpid system call
    call *sys      // vsyscall
    movl %eax, pid // get result
)
```

In the kernel;

```
Sysentry:
  syscallid = %eax
  handler = handlers[syscallid];
  *handler();
  sysexit
```

However, this is an indirect function call, which is predicted by the branch target buffer, which gives a prediction. The BTB could be primed to jump to the gadget.

Spectre variant two involves the following;

1. find gadget in victim's code space (or put it there)
2. train branch predictor to cause a speculative branch to the gadget when system call is executed
3. observe microarchitectural or cache sidechannel from the speculatively executed gadget
4. steal secret

To mitigate this, there are a number of approaches. This includes blocking sidechannels, which may be difficult to do completely with certainty. Another approach is to add noise to timers, which adds delays - however the attacker could just repeat the experiments. It's possible to prevent the attacker from poisoning the branch predictor, such as by adding instructions to prevent the use of the predictor; however this requires finding all the uses, as well as paying a penalty of a branch misprediction on every branch. Another approach is to maintain separate predictions for each thread in each protection domain; a per-process branch predictor.

Retpolines are a way of mitigating Spectre; implementing an indirect branch with a return instruction. It also fixes the return address stack to ensure a benign prediction target;

```
RP0: call RP2                 ; push address of RP1 onto stack and jump to RP2
RP1: int 3                    ; breakpoint to capture speculation
RP2: mov [rsp], <jump target> ; overwrite return address to desired target
RP3: ret                      ; return
```

The speculative jump from the return address stack will go back to `RP0`, and then execute a breakpoint instruction to prevent further damage. A branch misprediction will occur and a jump will go to the correct intended target. The goal is to ensure we can jump somewhere without an attacker influencing the prediction.

This lead to more sidechannel vulnerabilities being disclosed. The lecture then goes into a history of the vulnerability.

November 9 - Live Lecture

VIPT does translation in parallel with the L1 data cache access (indexed with untranslated address bits). The physical address from the cache is compared with the translation. Not all of the 12 bits of the page offset are used to index the L1 cache (the low order bits are omitted).

Page colouring for synonym consistency isn't typically done. In page colouring for conflicting conflict avoidance, a simple policy the OS could attempt is to ensure contiguous virtual pages are mapped to contiguous physical pages.

With *Rowhammer*, we may not know the actual mappings. However, since there's no security in obscurity, it's to show that it's possible, either by figuring out how the mappings work or by trying different combinations.

Two threads sharing the same core, such as with hyperthreading or SMT, can allow one thread to influence the branch predictor of another thread. The branch predictor can also be probed to reveal secrets, such as when the branches are influenced by the bits of a cryptographic key.

One possible mitigation of Spectre is to mark cache lines loaded by speculative execution as invalid if the branch isn't taken.

Chapter 6

This lecture looks at how to exploit instruction level parallelism (how to issue and execute multiple instructions per clock cycle), with static scheduling. Dynamic scheduling has significant complexity in hardware - static scheduling aims to shift this burden onto the compiler. The instruction set architecture could be changed to help with this.

Software Pipelining

A simple example (once again, counting down);

```
for (i=1000; i >= 0; i=i-1)
  x[i] = x[i] + s;
```

The original generated assembly has a number of stalls;

```
Loop:
  L.D    F0,0(R1) ; F0=vector element
  stall
  ADD.D  F4,F0,F2 ; add scalar from F2
  stall
  stall
  S.D    0(R1),F4 ; store result
  DSUBUI R1,R1,8  ; decrement pointer
  BNEZ   R1,Loop  ; branch if R1!=0
  stall           ; delayed
```

As seen before, this can be improved with instruction scheduling;

```
Loop:
  L.D    F0,0(R1)
  stall
  ADD.D  F4,F0,F2
  DSUBUI R1,R1,8
  BNEZ   R1,Loop  ; delayed branch
  S.D    8(R1),F4 ; altered to 8
```

This can be improved further with loop unrolling and adjusting the offsets. Only one copy of the loop control overhead exists now (assuming the number of loops is divisible by 3). This can be improved even further by renaming registers.

A further improvement is as follows;

```
S.D    0(R1),F4   ; stores M[i]
ADD.D  F4,F0,F2   ; adds to M[i-1]
L.D    F0,-16(R1) ; loads M[i-2]
DSUBUI R1,R1,#8
BNEZ   R1,LOOP
```

The store uses the result generated by the previous iteration of the loop. The add uses what was loaded in the previous iteration of the loop. This attempts to maximise the distance between when the value is generated and when it is used around the loop. This trick is known as **software pipelining**. If the loop was unrolled 3 times, the software pipelined loop uses the store from the first block, the add from the second, and the load from the third.

Note that this also includes the `FILL` and `DRAIN` phases; the fully pipelined phase is the `LOOP`;

```
FILL:
  L.D    F1,-0(R1)  ; loads M[N]
  L.D    F0,-8(R1)  ; loads M[N-1]
  ADD.D  F4,F1,F2   ; adds to M[N]
LOOP:               ; on entry, i=R1=N
  S.D    0(R1),F4   ; stores M[i]
  ADD.D  F4,F0,F2   ; adds to M[i-1]
  L.D    F0,-16(R1) ; loads M[i-2]
  DSUBUI R1,R1,#8
  BNEZ   R1,LOOP
DRAIN:
```

```
12    S.D    0(R1),F4   ; stores M[i-1]
13    ADD.D  F4,F0,F2   ; adds to M[i-2]
14    S.D    -16(R1),F4 ; stores M[i-2]
```

In Tomasulo, the hardware finds a similar schedule to what we've just done.

Very Large Instruction Word (VLIW)

Multiple instructions issued per cycle can be expensive, as we need to fetch a packet of instructions and check dependencies between all n pairs of instructions to see if they can all be issued in the same cycle. The hardware complexity of this is $O(n^2)$; this is doable for a small number but leads to multiple pipeline stages between fetch and issue. However, the compiler should be able to format these instructions into a packet that the hardware doesn't need to check for dependencies.

Transmeta's processor has four functional units in the machine (`FADD`, `ADD`, `LD`, `BRCC`), and each instruction tells the processor what each unit does in a cycle. A grouping is 128-bits and called a molecule. This can be done dynamically - but the interesting approach is doing the scheduling statically.

A *Texas Instruments* signal processor has 8 functional units, with two separate register files (four units on each). Moving a register from one register file to the other takes an entire clock cycle.

Recall the following unrolled loop (minimising stalls for scalar), also note that `L.D` to `ADD.D` takes 1 cycle and `ADD.D` to `S.D` takes 2;

```
1  Loop: L.D    F0,0(R1)
2        L.D    F6,-8(R1)
3        L.D    F10,-16(R1)
4        L.D    F14,-24(R1)
5        ADD.D  F4,F0,F2
6        ADD.D  F8,F6,F2
7        ADD.D  F12,F10,F2
8        ADD.D  F16,F14,F2
9        S.D    0(R1),F4
10       S.D    -8(R1),F8
11       S.D    -16(R1),F12
12       DSUBUI R1,R1,#32
13       BNEZ   R1,Loop
14       S.D    8(R1),F16
```

This took 14 clock cycles (3.5 per iteration).

In a VLIW machine (for brevity, `L.D` will be written as `L`, and similar for `S` and `ADD`);

clk	mem ref 1	mem ref 2	fp op 1	fp op 2	int op / branch
1	`L F0,0(R1)`	`L F6,-8(R1)`			
2	`L F10,-16(R1)`	`L F14,-24(R1)`			
3	`L F18,-32(R1)`	`L F22,-40(R1)`	`ADD F4,F0,F2`	`ADD F8,F6,F2`	
4	`L F26,-48(R1)`		`ADD F12,F10,F2`	`ADD F16,F14,F2`	
5			`ADD F20,F18,F2`	`ADD F24,F22,F2`	
6	`S 0(R1),F4`	`S -8(R1),F8`	`ADD F28,F26,F2`		
7	`S -16(R1),F12`	`S -24(R1),F16`			
8	`S -32(R1),F20`	`S -40(R1),F24`			`DSUBUI R1,R1,#48`
9	`S 0(R1),F28`				`BNEZ R1,LOOP`

This is now unrolled 7 times; resulting in ≈ 1.3 cycles per iteration; however this needs 15 registers, compared to the 6 before.

In software pipelining, we get 1 cycle per iteration, only using 7 registers;

clk	mem ref 1	mem ref 2	fp op 1	fp op 2	int op / branch
1	`L.D F0,-48(R1)`	`S.D 0(R1),F4`	`ADD.D F4,F0,F2`		
2	`L.D F6,-56(R1)`	`S.D -8(R1),F8`	`ADD.D F8,F6,F2`		`DSUBUI R1,R1,#24`
3	`L.D F10,-40(R1)`	`S.D 8(R1),F12`	`ADD.D F12,F10,F2`		`BNEZ R1,LOOP`

Explicitly Parallel Instruction Computer (EPIC)

This is a joint effort by *Intel* and *HP* for a future architecture; *Itanium* - IA-64.

The IA-64 instruction set issues instructions in instruction groups (a sequence of consecutive instructions with no register data dependencies or load / store dependencies). This means that all the instructions in the group, given enough hardware resources, can be executed in parallel. A group can be arbitrarily long, but compiler must explicit indicate the boundary between groups by placing a `stop` between 2 instructions that belong to different groups.

Instructions are encoded in bundles of 3, which are 128 bits wide; each instruction is 41 bits in length and the remaining 5 bits are taken up with a template field. The template field marks where instructions in the bundle are dependent / independent, and whether they can be issued in parallel with the **next** bundle.

This gives the benefits of a VLIW machine but with more flexibility and more binary compatibility across different implementations. Having a 41-bit long instruction allows for a very large register file.

Instructions can be explicitly sequential (note that the `;;` syntax denotes a stop bit);

```
1 add r1=r2,r3 ;;
2 sub r4=r1,r2 ;;
3 shl r2=r4,r8
```

Or in parallel (all three of these instructions can be issued at the same time);

```
1 add r1=r2,r3
2 sub r4=r11,r21
3 shl r12=r14,r8 ;;
```

The *Itanium* instruction set includes the following mechanisms;

- **register stack**

  This is an attempt to optimise what happens when a function is called. This maintains a mapping from the logical registers in the instructions to the physical registers in the machine. The logical mapping can be explicitly changed by a function call. See the slides for an example of the layout.

  This introduces the idea of explicitly manipulating the logical to physical register mapping. The general-purpose registers are configured to accelerate procedure calls. Registers 32 to 128 are used as a register stack (each procedure is allocated a set of registers from 0 to 96). The registers 0 to 31 are always accessible and addressed as the same.

- **predicated execution**

  The idea is as follows, with 64 1-bit predicate registers;

  ```
  1 (p1) add r1=r2,r3    // executed if p1
  2 (p2) sub r1=r2,r3 ;; // executed if p2
  3      shl r12=r1,r8   // executed always
  ```

  A condition can be stashed in a predicate registers and then used to control whether an instruction executes or not. Predication helps in branch prediction as it allowed us to cheaply jump over a **small** bit of code and avoid polluting the branch predictor state. In *Itanium*, there are additional benefits; if a branch would break a parallel issue packet, the instructions can be moved and predicated.

- **speculative, non-faulting load instructions**

  The instruction `LD.S r1=[a]` speculatively fetches from memory; any exception that may arise is suppressed. Another instruction, which would now be in place of `LD r1=[a]` is `CHK.S r1`; if `LD.S` didn't cause an exception, then nothing happens, otherwise an exception is thrown (at the correct point in the program).

  We may want to also use the speculatively loaded data in an instruction, prior to a check. The speculation status is propagated via NaT; any instruction that consumes a speculative result is also now speculative (suppresses exceptions). `chk.s` checks the entire dataflow sequence for exceptions.

  Original code;

```
1 instruction 1
2 instruction 2
3 ...
4 branch:
5   ...
6 or:
7   ld r1=[a]
8   use=r1
```

  This can be moved as follows;

```
1 ld.s r1=[a]
2 instruction 1
3 instruction 2
4 use=r1
5 ...
6 branch:
7   ...
8 or:
9   chk.s use
```

  Every register in the *Itanium* machine has a 65^{th} bit to indicate whether the value is valid or not (not a thing bit). If the original load to `r1` was an exception, the register is set to not a thing; any further use propagates this bit. `chk.s use` then looks at the bit; if it's not a thing, then the instructions are executed again.

  A speculative advanced load is as follows;

  Before

```
1 inst 1
2 inst 2
3 ...
4 st[?]
5 ...
6 ld r1=[x]
7 use=r1
```

  After

```
1 ld.a r1=[x]
2 inst 1
3 inst 2
4 ...
5 st[?]
6 ...
7 ld.c r1=[x]
8 use=r1
```

  When `ld.a` is executed, it stashes the address `x` into the Advance Load Address Table (ALAT), as well as initiating a load. When a store is executed (`st[?]`), any matching entry is removed from the ALAT. When the `ld.c` instruction is executed, it checks if `x` is still in the ALAT; if it is (no aliasing has occurred since `ld.a`) then it does nothing, otherwise (if aliasing has occurred) it reloads from memory. If the ALAT runs out of slots, then something will be overwritten (leading to a reload regardless of aliasing).

- **rotating register frame**

  This is another trick for explicitly managing the logical to physical mapping for registers. When combined with predication, it's possible to avoid unrolling (and separate fill and drain phases) for a software pipelined loop.

  Consider the loop for copying data (note that `ld4` and `st4` both cause a post-increment by 4);

```
L1: ld4     r35=[r4],4
    st4     [r5]=r37,4
    br.ctop L1 ;;
```

Note that it loads into `r35` but stores from `r37`; the branch instruction `br.ctop` rotates the logical to physical register mapping. When it reads from `r37`; it's actually reading what was assigned into `r35` two iterations prior. The example is as follows;

```
start:
        mov pr.rot    = 0       // clear all rotating predicate registers
        cmp.eq p16,p0 = r0,r0   // set p16=1
        mov ar.lc     = 4       // set loop counter to n-1
        mov ar.ec     = 3       // set epilogue (drain) counter to 3
        ...
loop:
  (p16) ldl r32       = [r12],1 // stage 1: load x
  (p17) add r34       = 1,r33   // stage 2: y=x+1
  (p18) stl [r13]     = r35,1   // stage 3: store y
        br.ctop loop            // branch back
```

Software pipeline stages are activated by predicates. When the pipeline is filling, the stages are progressively activated. Once the pipeline is full, all registers are activated, and once it starts to drain, the registers are progressively deactivated. Once an iteration is done, `r33` is remapped to `r34` and `r34` is remapped to `r35`. See the slides for an animation.

In the first execution, only the first instruction is executed (only one that is active), on the next iteration, the first two instructions are executed (note that all stages are issued in one packet).

The pipeline for this still isn't very simple. However, compared to *IBM*'s *Power 4*, which has a number of stages after the fetch for "instruction crack and group formation" (before even getting to out-of-order processing), it's simpler. *Itanium* has a much shallower pipeline, leading to a lower branch misprediction penalty. *Itanium* performs well for floating point programs (predictable memory access behaviour and branch behaviour) - the compiler knows how to pack. However, dynamically scheduled processors dominated integer performance; *Itanium* does well when static analysis by the compiler is easy.

Hardware-based speculation works better when control flow is unpredictable and when hardware-based branch prediction is superior to software-based branch prediction done at compile time. Compiler-based approaches may benefit from having a more global view of the code sequence (better code scheduling). Hardware based scheduling doesn't require different sequences to achieve good performance for different implementations of an architecture.

November 16 - Live Lecture

When we train the branch target predictor, we're trying to influence the target prediction for a branch in another address space. However, the branch predictor is typically indexed on virtual addresses, which means that training it in the attacker's virtual address space means it can be used in the victim's virtual address space. The branch target needs to be a valid address in the attacker's victim address space. The branch predictor is being trained to a homonym; we are training the predictor to an arbitrary address, as long as that address maps to the gadget's virtual address in the victim.

The point of a retpoline is to execute an indirect branch in an unpoisonable way (not influenced by the prior training of the branch predictor). The trick is to use the return stack predictor (predicts where an instruction will jump to; typically the call location). All modern processors have hardware to provide a branch prediction for a return instruction (uses a model of what the stack holds). Return instructions are just indirect branches, but with a different branch predictor.

The worst case of VLIW is when we fail to pack multiple operations into a molecule. This leads to underutilisation.

A case where unrolling is worse than software pipelining is a high number of iterations. The unrolled part of the loop uses operations well, however the fill phase and drain phase are sub-optimal.

When there's no more space in the IA-64 physical register stack, the processor will have to copy the contents of a register window to a main memory stack. There is typically locality in the stack (calls and returns, rather than multiple calls).

Chapter 7

Superscalar describes what was seen in the *SimpleScalar* simulator; where we have a dynamically scheduled processor. This achieves higher performance, but still misses issue opportunities and wastes resources. See the slides for a visual example.

The difference between fine grained multithreading and simultaneous multithreading is that the former fetches from each of the threads in a round robin fashion, and do not mix (on the same cycle). On the other hand, SMT allows instructions to be fetched from multiple threads, and be packed for simultaneous issue and dispatch.

In a basic out-of-order, we have a fetch stage (with instruction cache), issue-side register alias table (register map) and renaming, and a queue of instructions (RUU or similar). When instructions are ready to be dispatched, the operands are read from registers and sent into the execution units (with possible memory access). Results may then be distributed to other waiting instructions, and finally retired to the commit-side of the machine. An SMT pipeline does the exact same thing, but fetches from 4 PCs in parallel (or rather, dynamically scheduling the fetches). The instruction cache would have allocations from all four different threads. A register map would be required for each thread to tell us what the software visible registers are for each of the threads. After renaming, the instructions are passed into a common queue, in a FCFS fashion. The register renaming phase then allocates physical registers to them, dynamically from a common pool. Instructions are then executed, without regards to the thread they belong to. When the instructions are retired, the thread ID will determine what TLB entries are valid for a particular memory access (to keep within protection domains). Some considerations are the larger working set, as well as possible contention for resources. A case where this may be symbiotic is two threads, one being heavily arithmetic dominated and the other being memory dominated, where they can work without interference.

In *Intel*'s *Pentium 4* (one of the first commercial processors with hyperthreading), the performance of two cores was compared with two threads running on one core. When it does work, it doesn't improve performance by much and commonly made performance worse (hence it was commonly disabled).

One of the most important considerations in the microarchitecture of a SMT core is where the 'price' is being paid; what needs to be replicated (one per thread), and where the contention is likely to be. Both the instruction and data caches are going to be under contention. Some aspects of the branch predictor benefit from being shared, such as branch target buffer (as the two threads may operate on a common piece of code). We do need a register file for each core (both issue and commit side).

SMT raises issues, including each thread risks running slowly; the user experience is dominated by the slowest thread (hence we want the slowest thread to run as fast as possible). SMT may be preferred over FGMT; if only one thread is present, it's able to monopolise of all the processor's resources. There can also be destructive contention, such as thrashing the cache or other shared resources. SMT threads need to be fair; if a thread could monopolise the whole CPU, it could lead to a denial of service. An example is a slow thread that suffers many caches misses could fill the RUU and block issue (a bug in *Pentium 4*); this failed to yield the process and blocked other threads.

One key benefit of multithreading is memory system parallelism; different parts of the hierarchy can be active at the same time (for example, a hit in L1 while servicing an L2 access). Multithreading is another way of getting memory accesses in flight at the same time (similar to hit under miss and

prefetching). This comes naturally as multiple threads tend to have multiple memory accesses - this naturally fills the access latency (hides it) by overlapping computation with data access. The limit to the number of threads comes from the requirement of a lot of logical registers.

Rather than performing dynamic register renaming (as would happen in a dynamically scheduled processor), the register file can be statically partitioned. The goal is to have as many threads as possible running simultaneously on the machine to have memory system parallelism, enabling latency hiding - maximum occupancy. The register file can be partitioned based on the number of registers each thread requires. A program could be compiled (with fewer registers) and observed whether it results in significant spilling to memory, or if the particular program successfully fits into fewer registers. This is done by many GPUs. This leads to a tradeoff where there can be lots of lightweight threads to maximise latency hiding, or fewer heavyweight threads that benefits from many registers (this is called occupancy).

See the slides for an example of this division, as well as an example with CUDA.

Chapter 8

This lecture concerns vector instructions and SIMD (single instruction, multiple data), a trick for reducing the Turing Tax. SIMD is the idea that we have registers that hold entire vectors of operands (and we have instructions that work on entire vectors), which helps to amortise the fetch execute overhead.

The Roofline model is a tool for thinking about what is limiting the performance of a program (and therefore, what should be changed in the code to improve performance). A program with low arithmetic intensity is one that requires a large amount of data to me moved to do a small amount of floating point / arithmetic operations (for example, sparse matrices). On the other hand, a program with high amount of arithmetic intensity is one that can do a huge amount of computation with only a bit of data moved. Arithmetic intensity is the ratio of operations to bytes of DRAM traffic.

The idea for the Roofline model is to plot the arithmetic intensity against the achieved FLOPs per second (both on a log scale). The theoretical peak performance is a ceiling on the achievable performance defined at some horizontal line.

A machine with significantly higher memory bandwidth allows the theoretical peak performance to be reached by a larger class of applications. The theoretical performance can be reached by applications with a lower arithmetic intensity.

The AVX512 instruction set extends the scalar processor with vector registers, called `ZMM0` to `ZMM31`, which are 512 bits wide; these can be used to store 8 doubles, 16 floats, 32 shorts, or 64 bytes. This means that instructions are executed in parallel on 'lanes' (can be thought of as what happens to an individual element-side slice through the sequence of vector registers). The instruction set also provides predicate registers `k0` (always true) to `k7`. Each register holds a predicate per operand (lane), hence each register holds up to 64 bits.

An example of vector addition is as follows (note that in C, this can be written with vector intrinsics as `res = _mm512_maskz_add_ps(k, a, b)`);

```
1 VADDPS zmm1 {k1}{z}, zmm2, zmm3
```

Only the lanes with the corresponding bit set in `k1` are active. In the example above, with zero masking (note `z`), inactive lanes produce zero. On the other hand, if masking only (omit `z`), inactive lanes do not overwrite prior register contents.

Note the following program (note that the arrays are global variables, the loop trip count is known and divisible by 4, and sizes are powers of two);

```
1 float c[1024];
2 float a[1024];
3 float b[1024];
```

```
4 void add() {
5   for (int i = 0; i < 1024; i++) {
6     c[i] = a[i] + b[i];
7   }
8 }
```

Note that the generated instructions use SSE (a simpler version, present on most modern *Intel* processors). The `p` in `movaps` refers to a packed instruction (one that is actually a vector instruction). Once one 'constraint' is relaxed, when the loop trip count is set to `N` (a variable, and not known to be divisible by 4), the code becomes more complex. Part of the code remains the same to handle the majority of cases, and three copies of the loop body are also produced, to handle the cases when it's less than 4.

The next relaxation is to set the arrays to be parameters of the function;

```
1 void add(float *__restrict__ c, float *__restrict__ a, float *__restrict__ b, int
     N) {
2   for (int i = 0; i <= N; i++) {
3     c[i] = a[i] + b[i];
4   }
5 }
```

However, as these pointers are not known (not global), the compiler doesn't know if they are aligned on boundaries (vector operations work much faster on aligned operations). Three **more** copies of the loop body are now added before the main loop, in order to align the start address of the vectorised code on a 32-byte boundary. The removal of the `__restrict__` keyword means that the compiler doesn't know that the pointers refer to disjoint storage regions (hence the pointers can overlap in some way). More code is added at the start to check for overlaps; if overlaps do occur then a non-vectorised version of the loop is used, otherwise the vectorised version is used.

If the compiler refuses to vectorise the loop, there are a number of options;

- ignore vector dependencies compiler hint; telling the compiler that there are no loop-carried dependencies - this only tells the compiler that it is safe, it may still not vectorise

  ```
  1 void add(float *c, float *a, float *b, int N) {
  2   #pragma ivdep
  3   for (int i = 0; i <= N; i++) {
  4     c[i] = a[i] + b[i];
  5   }
  6 }
  ```

- OpenMP pragmas can be used;

  - loopwise (loop can be transformed into a SIMD loop, executed concurrently with SIMD instructions)

    ```
    1 void add(float *c, float *a, float *b, int N) {
    2   #pragma omp simd
    3   for (int i = 0; i <= N; i++) {
    4     c[i] = a[i] + b[i];
    5   }
    6 }
    ```

  - functionwise (generate a variant of this function that operates on a vector of operands, allowing it to be called from a vectorised loop like above)

    ```
    1 #pragma omp declare simd
    2 void add(float *c, float *a, float *b) {
    ```

```
    *c = *a + *b;
}
```

- SIMD intrinsics (tells the compiler which specific vector instructions to generate, guarantees vectorisation)

```
void add(float *c, float *a, float *b) {
  __m128* pSrc1 = (__m128*) a;
  __m128* pSrc2 = (__m128*) b;
  __m128* pDest = (__m128*) c;
  for (int i = 0; i <= N/4; i++) {
    *pDest++ = __mm_add_ps(*pSrc1++, *pSrc2++);
  }
}
```

- SIMT (single instruction, multiple thread) - think of each lane as a thread (predication can be used to handle nested if-then-else, while / for loops, and function calls)

```
#pragma omp simd
for (int i = 0; i < N; i++) {
  if (...) {...} else {...}
  for (int j = ...) {...}
  while (...) {...}
  f(...)
}
```

Consider the following example, which has an additional level of indirection (this spreads out the vector, which could end up existing on different pages);

```
float ALIGN c[1024];
float ALIGN a[1024];
float ALIGN b[1024];
int ALIGN ind[1024];

void add() {
  for (int i = 0; i < 1024; i++) {
    c[i] = a[i] + b[ind[i]];
  }
}
```

This introduces the gather instruction, which is a vector load that loads from a different address in each lane.

Vector instructions may execute n-wide vector operations with an n-wide ALU, or possibly in smaller m-wide blocks. Two execution strategies involve;

- **vector pipelining**

  Consider an extreme case where we have 64 doubles, and only one execution unit. The vector instructions are executed serially, element-by-element with a pipelined functional unit. This works well with vector chaining (each word forwarded to the next instruction as soon as it's available), with the FUs forming a long pipelined chain. Chaining is essentially a vector version of forwarding.

- **uop decomposition**

  Consider a processor that only has a 128-bit vector unit, but we want to execute a 256-bit vector instruction. At decode time, we could break the 256-bit instruction into two uops, and issue them

separately into the out-of-order execution engine. Eventually, the two instructions would retire and meet together in the retire unit, creating the impression that they completed at the same time.

An example of this is in the *AMD Jaguar* processor, where there are two 128-bit ALUs. When the split instructions (split at the decode stage) complete, the retire unit waits for both halves to arrive before it allows for the combined 256-bit instruction to commit. If we know that half of the words are zero (not used), it can be used to avoid producing two instructions at decode time. This can be tracked with a zero-bit.

SIMD reduces Turing tax by performing more work with fewer instructions. However, this depends on the compiler (for simple loops, etc.) or the programmer. Lane-by-lane predication helps to vectorise conditionals, but branch divergence can lead to poor utilisation. Spatial locality for gather instructions can be difficult to achieve unless the accesses fall on a small number of distinct cache lines - a gather could lead to sequential execution if it hits multiple cache lines.

ARM's SVE (scalable vector extension) is an ISA design that is independent to the actual vector length supported by the hardware / registers.

Chapter 9

When the workload consists of thousands of threads, speculation should never be done, nor should branch prediction - there is always another thread waiting with work to do. Multithreading can therefore be used on a large scale, since there is significant parallelism - this helps to hide pipeline hazards as well as memory access latency. In graphics, the parallelism tends to be proportional to the number of pixels on the screen; we shouldn't need a 'loop' to explicitly launch all of these threads.

In a CPU, a small area of the actual chip area is actually dedicated to computation (ALUs), with a large amount of the area being used for control and cache. On the other hand, most of the area on a GPU is used for computation (with much smaller area for control overhead). The GPU has significantly less cache (compared to a CPU) as FGMT (fine grained multithreading) can be used to cover memory access latency.

This example goes over the *NVIDIA G80*, from 2006. There are 16 cores (SMs - streaming multiprocessors, a fetch execute engine) packaged in pairs called TPCs, each with 8 SP, shared memory / scratchpad SRAM associated with each core, and some cache. There's also a high performance interconnection network connecting these processors to an array of memory interfaces (DRAM, off-chip). In front of each DRAM is an L2 cache, as well as a ROP (raster operation processor, for computations like alpha blending and z-buffer, (see graphics module)). This is connected to the CPU through some interface, such as PCIe, and has units for distributing pixel and vertex rendering tasks among the resources, as well as distributing computational work (where the launching of parallel threads is done).

A TPC (texture / processor cluster) groups SMs based on shared resources. Some newer models exclude this. The components include the following;

- **SMC** - streaming multiprocessor controller - (shared)
- **geometry controller** (shared) - directs all primitive and vertex attribute and topology flow in the TPC
- **texture cache** - performs interpolation, indexed by location in a texture with floating point numbers; this has locality in 2D (shared)
- **MT issue** - multithreaded instruction fetch and issue unit
- **C cache** - constant read-only cache
- **I cache** - instruction cache

- **SFU** - special-function unit, for computing transcendental functions such as sin, cos, $\log x$, $\frac{1}{x}$ (2 in an SM)
- **SP** - streaming processor (8 in an SM)

  This is where the computation is actually done. SPs work on 32-wide vector instructions.
- **shared memory** - scratchpad memory (such as a user managed cache), shared by SPs / threads in an SM

Note the following 'translations' between *NVIDIA*'s terminology; warps on a GPU are like threads on a CPU, and threads on a GPU are like lanes on a SIMD CPU. The multithread issue stage selects which warp to issue from in each cycle (FGMT) - each warp consists of control flow executing a sequence of 32-wide SIMD vector instructions. In this product, they're executed in 4 steps, using 8 SPs (vector pipelining with lanewise predication). Each SM has local, explicitly programmed scratchpad memory which is shared between different warps on the same SM. There is a level of parallelism hierarchy; there are threads that run on an SP, warps that run on an SM, and a block of warps that run on an SM together. An SM has an L1 data cache, but no cache-coherency protocol as they do not exchange data between SMs through writes to memory, in the execution of an individual kernel.

The chip has multiple DRAM channels each includes an L2 cache. However each data value can only be in a single L2 location, hence there are no cache coherency issues in L2. Data that lives in one DRAM can only be in that specific cache.

The unified architecture uses the same hardware for both vertex-shader and pixel-shader stages, and instead dynamically schedules the different shading stages of the rendering pipeline. CUDA is a C extension that allows for serial CPU code and parallel GPU code (kernels). A GPU kernel is a C function that operates on a GPU thread (a lane for SIMD CPUs). A group of threads forms a thread block (the unit of scheduling / allocation of threads to an SM) - threads in the same block have access to a common shared memory. Thread blocks are then organised into a grid (hierarchy of parallelism). Consider the following C code, for 'double precision $ax + y$';

```
daxpy(n, 2.0, x, y);

void daxpy(int n, double a, double *x, double *y) {
  for (int i = 0; i < n; ++i) {
    y[i] = a*x[i] + y[i];
  }
}
```

In CUDA, this needs to be split into two parts, the CUDA kernel and the CPU code to launch the kernel on the GPU;

```
// the keyword tells us that it runs on the GPU, and the function is what one
    thread does (one thread does one iteration of the loop)
__global__ void daxpy(int N, double a, double *x, double *y) {
  // figure out where it is in the iteration space, using registers set by the
     thread launch mechanism
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < N) {
    y[i] = a*x[i] + y[i];
  }
}

int main() {
  // kernel setup
  int N = 1024;
```

```
14    int blockDim = 256;                          // number of threads per block
15    int gridDim = 256;                           // number of blocks
16    daxpy<<<gridDim, blockDim>>>(N, 2.0, x, y); // invocation
17  }
```

See the slides for an example of the generated PTX (proprietary pseudo-assembly).

The goal is to fill the machine with enough warps so FGMT can fully occupy the machine, and keep enough memory accesses in flight to hide memory access latency. However, due to the dynamic scheduling, we may encounter a load balancing problem, hence the work might be broken up into a smaller number of blocks in order to reduce the load imbalance issue.

Similar to multithreaded CPUs, there are a limited number of registers; if each warp can use a small number of registers then many can run at the same time. On the other hand, if warps use as many registers as they need, then not as many can run simultaneously.

As we execute successive instructions from a given warp, we progress the execution of all of the threads (which are individual 'lanes' in the warp). Each warp can be interleaved with other warps. The idea is to program what happens in a single CUDA thread; a single lane; also called SIMT (single instruction, multiple threads). Threads can branch, typically by predication. However, if all of the threads agree on a particular branch (or branch not being taken), then a conditional jump can be executed and code that isn't needed can be skipped. Therefore, it becomes important to recognise that threads that form a 32-wide warp have coupled execution progress (and therefore threads in other warps are decoupled).

Threads can either take the same path in a warp or diverge. A warp serially executes each path, disabling some threads; when all paths complete the threads reconverge. Note that divergence only occurs within the same warp. This can be thought of as control-flow coherence; adjacent threads are likely to be adjacent in a warp; and therefore will run in a lock step way. When these threads branch, it'd be ideal if they branched in the same way; spatial branch locality (spatial in terms of adjacent threads in the thread index space).

SIMT

- one thread per lane
- adjacent threads (warp / wavefront) execute in lockstep
- SMT: multiple warps run on the same core to hide memory latency (multithreading on much larger scale)

SIMD

- each thread may include SIMD vector instructions
- SMT: a small number of threads run on the same core to hide memory latency

In the SIMD world, the concept of adjacent threads doesn't exist. Threads are scheduled onto processors in some fashion, whereas the SIMT context, these are very fine grained threads that are being within the lanes of a vector instruction.

On an CPU, we care about spatial locality (adjacent loop iterations access adjacent data) - when we vectorise a loop, adjacent iterations are packed into a single vector instruction. In SIMT, spatial locality concerns the behaviour of adjacent threads. Adjacent threads should access adjacent data.

Consider the two examples; On CPU, this has good spatial locality as it traverses the data in layout order;

```
1  void add(float *c, float *a, float *b) {
2    for (int i = 0; i <= N; i++) {
3      __m128 *pa = (__m128*) &a[i][0];
4      __m128 *pb = (__m128*) &b[i][0];
5      __m128 *pc = (__m128*) &c[i][0];
6      for (int j = 0; j <= N/4; j++) {
```

```
7         *pc++ = _mm_add_ps(*pa++, *pb++)
8       }
9     }
10  }
```

On GPU, this has poor spatial locality since adjacent threads access different columns (for example, thread `i=0` accesses the first element of row 0, `i=1` accesses the first element of row 1, and so on - adjacent thread IDs access data in different cache lines);

```
1  __global__ void add(int N, double *a, double *b, double *c) {
2    int i = blockIdx.x * blockDim.x + threadIdx.x;
3    for (int j = 0; j <= N; j++) {
4      c[i][j] = a[i][j] + b[i][j];
5    }
6  }
```

SIMT requires us to reconsider the spatial locality of our code; in GPU terms, this is known as coalescing.

Another consideration is spatial control locality. In SIMT, branch coherence states that adjacent threads in a warp all usually branch the same way (spatial locality for branches, across threads). On the other hand in SIMD, branch predictability states that each individual branch is mostly taken / not-taken (or is generally predicted by global history).

November 23 - Live Lecture

In FGMT (different threads in different cycles), in each clock cycle, all the operations belong to one thread. As such, they are interleaved in a clock-by-clock basis, hence there is no context switching overhead (instead, we are context switching on every clock cycle). On the other hand, with SMT (dynamic scheduling of operations from a pool of threads), threads may be selected in a round robin fashion, but the out-of-order processor may reordered without prejudice as to which thread they belong to.

With a GPU, we can actually choose how many threads run on a core at once, whereas with typical CPUs, this is limited by the hardware (typically two threads). On a CPU, the two threads may belong to completely different processes, whereas with the GPU, the threads are optimised to be essentially doing the same thing.

Suppose we have very wide vectors (such as 32 elements). Assume that we have a load instruction, which is also 32 wide, then a multiply which consumes the result from the load, and an add that consumes the result from the multiply (and so on). However, the machine may only have 8 wide vector functional units (despite supporting 32 wide) - as such, it does 8 operands from the 32 in each clock cycle. The idea of chaining concerns when we can forward from one vector instruction to another. Since the vector instruction operates over 4 clock cycles, the next instruction can be started after the first block of 8 is done. Essentially; we don't have to wait for all 32 to be done before passing values to the next instruction.

Each level of the parallelism hierarchy can be determined by what resources are shared, as well as capabilities. For example (on the latter), different vector lanes can't execute arbitrarily different code (they must execute the same sequence of vector instructions). In the CUDA execution model; a grid runs on the whole GPU, a set of warps runs on an SM, and this is further subdivided into threads (each warp is a thread of vector instructions) which run on SPs. Once a thread block is mapped onto an SM, it can share resources associated with the SM.

A thread in *NVIDIA*'s terms is a dataflow slice associated with a sequence of lanes (what happens in a lane of a warp).

Chapter 10

Motivation for Multicore, Parallel Programming, and Shared-memory versus Distributed-memory

During the 'escalator'-phase, programmers didn't need to know anything complex to benefit from the performance gains. However, the single threaded performance is no longer growing with Moore's law; the clock rate at which processors can run at has hit a ceiling (somewhere in the early 2000s) - it's rare for a processor to go beyond 4.5 - 5GHz. Similarly, chips have been hitting an power ceiling. A consequence of this is an increase in the number of logical cores.

This is due to power being the critical constraint. Consider the following;

- **dynamic power**

  Power is consumed when signals change (when a wire is charged up to deliver current to the other end of the wire). Power is consumed in resistance. Dennard Scaling is the reason Moore's law has been linked to single thread performance; dynamic power gets smaller as we make transistors smaller. As such, we can increase the clock rate as transistors get smaller (within the same power budget).

- **static leakage**

  Power is consumed when the gates are powered-up, due to leakage through the circuit structure (through the transistors). The end of Dennard Scaling could be part of the reason of the clock rate ceiling. As transistors get smaller, static leakage becomes the most important factor (whereas previously, dynamic power was the only consideration). This leakage is exaggerated at a high voltage, which is required for high clock rates.

We can achieve a given level of performance by increasing the clock rate or exploiting parallelism. The former has an issue where the power increases sharply with clock rate as there is not only high dynamic switching, but also high static leakage (from the high voltage). The latter is more efficient, as it allows us to run at a lower clock rate and use less energy per operation.

One approach to minimise power usage is to compute quickly and then turn it off; which could be useful in a data centre where there are other tenants that could use the machine. Another solution is to go extremely slowly, to minimise energy (as long as it can meet the deadline). Other strategies at a lower level are clock / power gating, where units are turned off when not in use, either functional units / entire cores. Another popular solution is dynamic voltage / clock regulation, where the clock rate / supply voltage are dynamically reduced. An example use case for this is when the battery is low or when the CPU isn't a bottleneck (for example, trying to match the framerate of the screen, or dominated by memory accesses). Another solution is to run multiple cores, each at a slower clock rate. If only one core is active, then the clock rate could be boosted for the active core.

Consider the following example, highlighting why shared memory makes parallel programming easier;

```
1  for (i = 0; i < N; ++i) {
2    par_for (j = 0; j < M; ++j) {
3      A[i, j] = (A[i - 1, j] + A[i, j]) * 0.5;
4    }
5  }
6
7  par_for (i = 0; i < N; ++i) {
8    for (j = 0; j < M; ++j) {
9      A[i, j] = (A[i, j - 1] + A[i, j]) * 0.5;
10   }
11 }
```

Note that the parallelism is slightly different in both parts due to the dependence. Also note that `i` indexes a row and `j` indexes a column. The simplest synchronisation for the first inner loop is a simple barrier, where we wait for all of the threads in the parallel loop to finish before continuing to the next `i` iteration. As such, by the time we enter the second outer loop, we know all the elements have been computed.

The first loop operates on rows in parallel, and the second loop operates on columns in parallel. With a distributed memory system, we'd need to pass messages to transpose the array between the two outer loops; however this isn't an issue for a shared memory system. Shared memory is convenient and fast; communication is done with just loads and stores. However, shared memory encourages programmers to ignore where communication occurs.

OpenMP is a standard design for language extensions for shared-memory parallel programming. The implementation requires some compiler support.

With the simple example above, each iteration should take a similar amount of time. However, if the loop body was more complex, then there may be a load imbalance, hence allocating a fixed amount of iterations to each thread isn't ideal. A self scheduling loop can be seen on the slides. This shows that there is some overhead in managing the loop; when the actual loop code is quite simple, some considerations should be made whether this could be done in chunks, or similar. Another optimisation that could be made is to exploit cache affinity. In the case we have a sequence of parallel loops, ideally each core would work on a corresponding chunk; a core should be assigned iterations of a loop that accesses common data with what was recently accessed.

Using locks for the atomic `FetchAndAdd` is quite expensive. Many processors have built in hardware support for atomic increments, such as GCC's built in `__sync_fetch_and_add(p, inc)`. In a large system, any atomic iteration requires all processors queuing up to be able to execute an atomic operation on the shared variable (even in the single instruction case). When two `FetchAndAdd` messages meet, they could be combined into a single `FetchAndAdd` that increments by 2 and returns two values (`n` and `n + 1`).

The slide goes into an example explaining the meaning of;

```
1 #pragma omp parallel for default(shared) private(i) schedule(static, chunk)
     reduction(+:result)
```

MPI, message-passing interface, is a popular alternative for large scale systems. This is standard API using message passing. MPI allows complex programs to be written with arbitrary graphs of communicating processes. Most of the time, we are concerned with programs that are dominated by loops. The advice is to keep the program simple, with SPMD (single-program, multiple data); where each process has a share of the data and shares the same control-flow (as far as it affects any communications that the program does). This allows us to reason about the program by thinking about a small number of processors.

An example of MPI is the stencil problem;

```
1 DO j = 1, m
2   DO i = 1, n
3     B(i, j) = 0.25 * (A(i - 1, j) + A(i + 1, j) + A(i, j - 1) + A(i, j + 1))
4   END DO
5 END DO
```

In OpenMP, this can be expressed as the following (note the convergence condition has been omitted, additionally `collapse(2)` allows for the two loops be executed in parallel);

```
1 while (!converged) {
2   #pragma omp parallel for private(j) collapse(2)
3   for (i = 0; i < N; ++i)
4     for (j = 0; j < M; ++j)
5       B[i][j] = 0.25 * (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]);
```

```
6    #pragma omp parallel for private(j) collapse(2)
7      for (i = 0; i < N; ++i)
8        for (j = 0; j < M; ++j)
9          A[i][j] = B[i][j];
10 }
```

See the slides for an example of a larger allocation. Each slice needs to have a 'halo' region, which bounds it on the left and right, for items that are being read by a processor but not assigned to. The halo of one region is in the region of the adjacent processor, and similarly that adjacent processor will have a halo in the previous processor. This has to be made explicit in MPI.

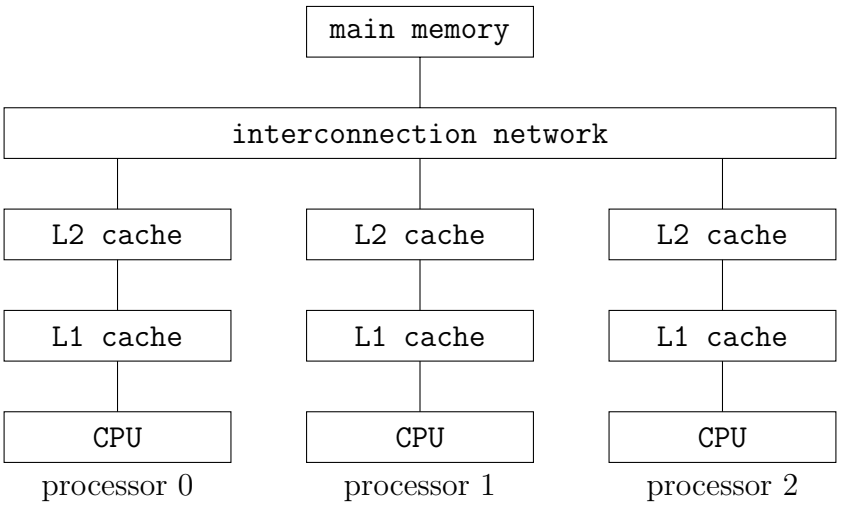
See the example on the slides for this. Note that the allocated space is the data that we (the processor) compute, as well as everything that the data depends on (the halo regions).

OpenMP is easy, but hides the communications (we don't need to consider allocating memory to receive from neighbouring processors) which can lead to bugs if unintended sharing occurs. MPI requires more copying of data (namely in sending data), or at least appears to (this may be done when accessing across boundaries in OpenMP). MPI allows us to see how to reduce communication and synchronisation.

Cache Coherency Protocols

This mainly looks at the 'snooping' family of cache coherency protocols, which works for a modest number of cores. This is due to the reliance on some interconnect between the processors, allowing every processor to see all the traffic on the network.

Consider the following example;



Suppose there is some memory location `x` that sits in main memory. First processor 0 loads from memory location `x`; this is allocated in both the `L2` and `L1` cache for processor 0. Then processor 1 loads from the same location, which becomes allocated in both caches as well (now there are a total of 5 copies). Now assume that processor 0 stores to location `x`; thus updating the `L2` and `L1` caches (assuming there's some writing back); note that the main memory isn't updated. Whether we actually write back or not doesn't make much difference to the fundamental problem; processor 1 doesn't know what's happened. Suppose processor 2 also reads from `x` - it needs to know whether to get `x` from main memory (occasionally, this can be the only source), processor 0, or processor 1. This raises two issues;

- where do we find the latest version of the cache line
- when can we use the cached copy, and when do we have to look for a more updated version

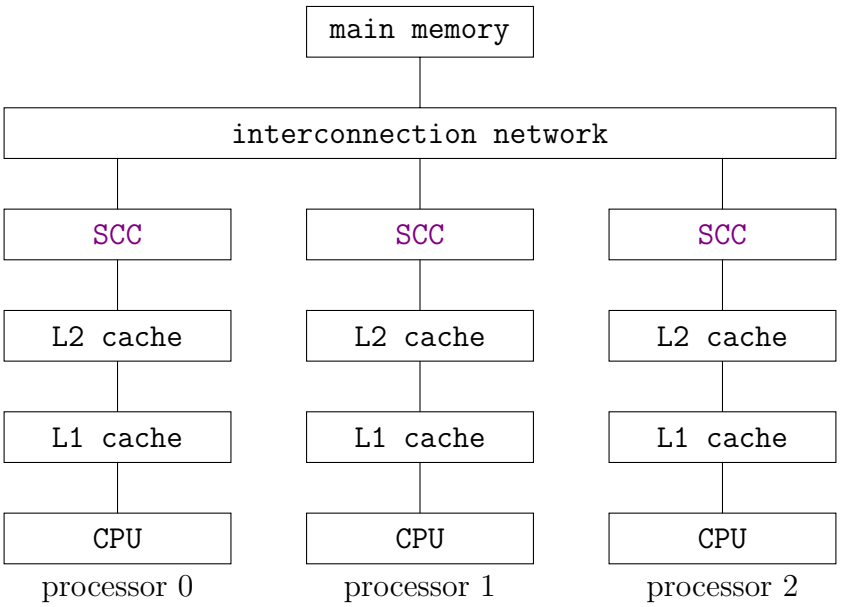
Lamport's idea of sequential consistency states the following - the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specific by its program.

When we observe the behaviour of another core by watching its issued loads and stores, we should see something consistent with its program order. This may not happen as writes / reads may be reordered in some way.

We don't want remote copies to be indefinitely out of date; one approach is to update whoever has a copy of the address once a store is executed (with an update message). To do this; we'd either need to broadcast every store to every remote cache, keep a list of which remote caches may hold the cache line, or at note that if there are any remote copies of this line (a shared bit). There are a number of problems with updates. One issue is that if the cache line is several words long, we may need to do a broadcast for each word in the line. We may also update processors that don't care about the new writes; it would be nice to have exclusive access to the cache line, to avoid broadcasting updates.

Invalidation is a more popular plan; instead of updating remote cache lines, they are invalidated when a store occurs. After the first write to a cache line, we can be sure there are no remote copies (since they are all invalidated), meaning subsequent writes don't need communication. However, if another processor requests a copy, we will know more copies exist and require invalidation if we do another update. This is often better than updates, however if the processors really need the new data quickly, it forces them to suffer a read miss. Each cache line will need some additional state maintained, to exploit this.

In snooping, an additional cache controller (`SCC` in the diagram) is added between the L2 cache and the interconnection network / bus;



The snooping cache controller has to monitor all bus transactions and check them against its caches' tags. For example, if processor 0 was to invalidate address `x`, processor 1's SCC would notice the presence of address `x` in its caches. Similarly, when processor 2 requests address `x`, a read request is broadcasted on the bus. If another processor has a dirty copy of the data, it is important for it to supply this data (in this example, processor 0's copy is more recent than the one in main memory). The SCC therefore needs to monitor for read requests as well.

The Berkeley protocol has 2 bits (4 possible states) for each cache line in each core (note that ownership refers to a processor's job; if it is the owner, its the processor's job to supply the data);

- invalid
- valid clean, potentially shared, unowned

  The processor does not have the right to write to that data, nor is it the processor's responsibility to supply it on a request
- shared-dirty modified, possibly shared, owned

  From the dirty state, if someone else tries to read it, it supplies it and becomes shared-dirty

- dirty modified, only copy, owned

  This has been written to and an invalidation has been broadcast (hence only copy and exclusive access) - subsequent writes don't need another invalidation

The cases are as follows (see the slides for a state diagram);

- **read hit**

  The easiest case; if the cache line is not in an invalid state, the data can be read directly.

- **read miss**

  On a read miss, we broadcast the request on the bus. If another cache line has the line in either a shared-dirty or dirty state, it supplies it and sets its own state to shared-dirty, our copy is set to valid. Otherwise, the line comes from memory, set to valid.

- **write hit**

  On a write hit, nothing happens if the line is in a dirty state (already invalidated all other copies, no further invalidation is required). However, if the state is valid or shared-dirty, an invalidation is performed and the local state is set to dirty. This is done to gain exclusive access.

- **write miss**

  The line comes from the owner (either main memory or another processor, as with the read miss case). All other copies are invalidated (set to invalid) and the line in the requesting (local) cache is set to dirty.

The cache controller will see all the memory traffic of all of the cores. Transitions between states are triggered either by the bus (invalidate, write miss, read miss) or the CPU (read hit, read miss, write hit, write miss). For every transaction on the bus, the cache controller needs to look up the directory (cache line state) information for a specified address. The states don't handle what happens in a cache displacement (see textbook).

The dirty bit records whether remote copies exist (invalidation is done by broadcasting messages on the bus). It also handles where to get updated data with a read miss request on the bus.

Since every bus transaction checks cache tags, there could be contention between what the SCC wants to do with tags and the CPU. One solution is to duplicate the tags, to give the SCC parallel access to the cache state. Another solution is to use multi-level inclusion (if everything in L1 is in L2, then we can use L2 to filter invalidations - only have to check L1 if L2 tag matches). Alternatives for cache inclusivity is a snoop filter.