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

31 26	25 21	20 16	15 11	10 6	5 0
Op	Rs1	Rs2	Rd		Opx

  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)

31 26	25 21	20 16	15 0
Op	Rs1	Rd	immediate

  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**

31 26	25 21	20 16	15 0
Op	Rs1	Rs2/Opx	immediate

  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**

31 26	25 0
Op	target

  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;

  ```
  ADD R1,R2,R3
  SUB R4,R1,R3
  AND R6,R1,R7
  OR R8,R1,R9
  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;

  ```
  LW R1,0(R2)
  SUB R4,R1,R6
  AND R6,R1,R7
  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);

31 ... 10	9 ... 5	4 ... 0
cache tag	cache index	byte select
e.g. `0x50`	e.g. `0x01`	e.g. `0x00`

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;

```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;

```
DIVD F0,F2,F4
ADDD F10,F0,F8
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;

   ```
   I: ADD R1,R2,R3
   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)

   ```
   I: SUB R4,R1,R3
   J: ADD R1,R2,R3
   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;

   ```
   I: SUB R1,R4,R3
   J: ADD R1,R2,R3
   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;

```
Loop: L.D    F0,0(R1)
      L.D    F6,-8(R1)
      L.D    F10,-16(R1)
      L.D    F14,-24(R1)
      ADD.D  F4,F0,F2
      ADD.D  F8,F6,F2
      ADD.D  F12,F10,F2
      ADD.D  F16,F14,F2
      S.D    0(R1),F4
      S.D    -8(R1),F8
      S.D    -16(R1),F12
      DSUBUI R1,R1,#32
      BNEZ   R1,Loop
      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.

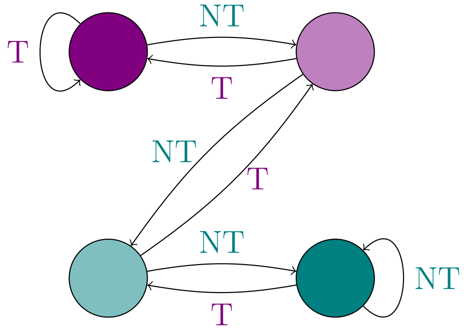
  Ideally, we take an instruction from before the branch instruction. Alternatively, we can take an instruction from either the target address, or from the fall through (although this is only valuable when it's correct) - we also want to ensure that it's the correct thing to do (hence we shouldn't store 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;

```
if (C1) then
  S1;
endif
if (C2) then
  S2;
endif
if (C3) then
  S3;
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;

```
F:
  body of F
  ...
```

```
  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

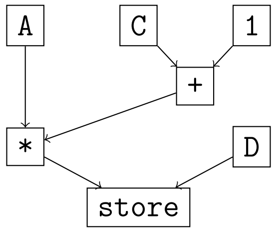
Blocking can be thought of as a transformation on loop nests. The first transformation is referred to as strip mining; the order of execution hasn't changed at all (an outer loop incrementing by S). Strip mining a single loop is as follows;

```
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**

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

- **stack machine**

```
push  A        // load from A, push to stack
push  C
add   #1       // add 1 to the value on the top of stack
mul            // multiply the top two values on stack
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**

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

```
load  A        // load A onto belt
load  C        // load C onto belt
add   -1 #1    // add 1 to the result of the previous instruction
mul   -1 -3    // multiply the results from positions -1 and -3 on the belt
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.