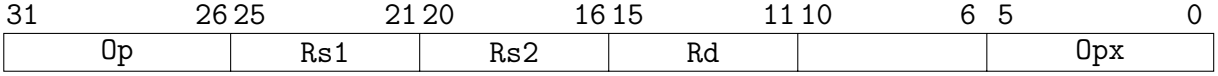
Advanced Computer Architecture (60001)

Chapter 1

Pipelines

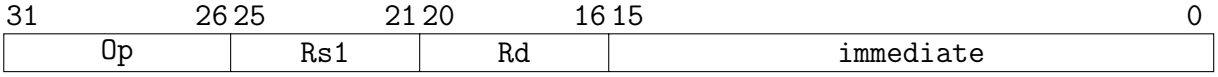
For the sake of example, we're using MIPS, which is a reduced instruction set design (every instruction is 32-bits wide, making it easy to decode);

- **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`.

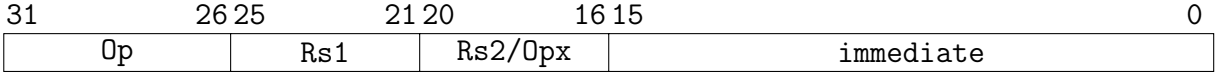
- **register-immediate** one source and one destination register, immediate operand (e.g. adding)



  Examples of this include;

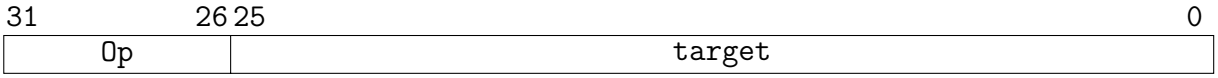
  - `LW R2, 100(R3)` `R2 <- Memory[R3+100]`
    source register is to be added to an immediate constant, could be useful for accessing fields of a struct (field offset is the constant) by using the base of the object
  - `SW R5, 100(R6)` `Memory[R6+100] <- R5`
    same as above, but for storing
  - `ADDI 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), 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 FD F0,X
3 MUL F0,F2,F3
4 FD 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 *Tomosulo* 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.