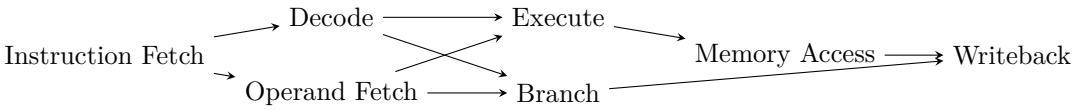
1. Describe a 5-stage pipeline suitable for executing the RISC-V ISA .

   Here are the main parts of executing an instruction and a dependency graph for the RISC-V ISA. As we can see, there are 5 stages which must be done serially. Therefore the 5-stage pipeline will be Instruction Fetch; Decode & Operand Fetch; Execute & Branch; Memory Access and Writeback.



   - Instruction Fetch

     Load the value addressed by the program counter into the "instruction register" and increment the program counter.

   - Decode & Operand Fetch

     Decode the instruction to determine what operation we are performing. In this stage we also determine which bits are immediates and determine the value of any immediates. In parallel we can load two operands (rs1 and rs2) from the registers. Although some instructions only require one operand, it is far faster to load both and discard one if it's not needed. The RISC-V is designed to make this particularly easy – all source registers are stored in the same place.

     If the instruction is a branch instruction by an immediate (such as JAL), then we can immediately write back to the program counter so that the next instructions fetched are correct.

     Decode control signals are initialised in the decode stage.

   - Execute & Branch

     Perform an arithmetic or logical operation and if necessary update the program counter. Although branch is writing back to a register, it makes more sense to write to the program counter as soon as we know it has to be written to. This will reduce the branch-taken penalty. Therefore the ALU will forward the result of arithmetic to the branch unit, which can set the PC to these values if necessary.

   - Memory Access (MA)

     Read from or store to memory if required. Most instructions will not need to access memory. Accessing memory can take substantial amounts of time.

   - Writeback (W)

     We store the result of the instruction in the destination register if there is any result to be stored.

2. What is the significance of some registers being preserved in some circumstances and others not?

   The reason that some registers are preserved is that the calling convention denotes some registers should be nonvolatile in order to retain a consistent ABI (Application Binary Interface) for callers. While others are volatile and have no such constraints. This consistent interface allows good integration.

   If a function preserves a register, then the value of that register after the function call is the same as the value at the start of the function call.

3. What control hazards apply to your pipeline and how could they be resolved?

   When branching, we know the updated value of the PC and update it at the end of the execute stage. However, the two next instructions could have fetched from the wrong

address. The simplest solution is therefore to flush the next two instructions. Note that commits are only made after the execute stage – writing to the PC in execute, storing to memory in memory access or writing to registers in write-back. Therefore the two instructions we would cancel will not have made any commits and so we do not have to "undo" them.

For conditional branches; a more efficient (and complicated) approach involves branch prediction. We "guess" which instructions will be executed next and execute those. If the guess was correct then the processor does not need to issue any bubbles; and if it was incorrect then the pipeline has the same number of bubbles as before. This method can require more hardware, however a good implementation can increase performance significantly.

Modern branch predictors can have 99% accuracy on predicting the next instructions to be issued. This can therefore provide a massive speedup. The expected cost of a branch instruction decreases from 3 cycles to 0.03 cycles.

Branch predictors take many forms:

- Static Branch prediction

  The most basic branch predictors make assumptions such as "we are likely to take backwards branches and unlikely to take forwards branches" as for loops are backwards branches and repeat many times, while if statements are run at most once.

  Static branch predictors usually guess correctly – however can fail to spot complicated access patterns and are therefore not as accurate as dynamic branch predictors. Static branch predictors are quick and easy to implement. They are therefore suitable for implementation on a pipeline such as the RISC-V one described above.

- Dynamic Branch prediction

  More advanced branch predictors store large tables of the history of whether branches were taken and run complex algorithms to determine whether to take branches or not. They make assumptions that past behaviour is an accurate prediction of future behaviour – lots of code is in loops and is therefore very predictable if patterns can be found.

Similarly, we could attempt branch target prediction – which "guesses" where the branch will go to. This is particularly relevant for unconditional branches (jumps) where the target depends on the value of a register (such as JALR), however branch target predictors are also used in conditional branches. Assumptions could include "the value of the register will not change". Many unconditional branches depending on registers are function returns and it is unusual for functions to change their own return addresses.

In the simple 5-level pipeline, branch prediction is not as important as it is in deeply pipelined superscalar architectures – where dozens of instructions can be executed speculatively before the branch is calculated.

4. What data hazards apply to your pipeline and how could they be resolved?

   This pipeline can only experience read-after-write hazards. Of which there are two main types. One can be resolved without inefficiency while the third is more complicated and will decrease performance.

   - Execute-to-Execute

     If we execute an instruction which will change the result in a register r_i and either of the next two instructions have r_i as an operand; then they will fetch the outdated version of r_i. This will lead to a potential wrong result.

The writeback and decode stages are the two shortest pipeline stages. Therefore they can be executed in half a cycle. We can therefore perform register writeback in the first half of a clock cycle and fetch from registers in the second half. This eliminates the data hazard between execute and the instruction two instructions later without the need for any stalls.

We can resolve the hazard between execute and the next instruction by adding a forwarding path from the output of execute to the input of execute. This will update the value of any registers which have been modified and are used in the next instruction. Hence providing the next instruction with the updated value for r_i with no stalls.

- load-use

  Memory accesses take time. Even an L1 cache hit takes ~ 3 clock cycles. Higher level caches take tens of cycles and DRAM takes hundreds. In the worst-case, a memory access could cause a page fault, requiring hundreds of thousands or millions of cycles to load the memory in from the hard disk. Between the time that a memory access is started and returns, we cannot execute any instructions which depend on the fetched instruction. This is known as a "load-use hazard".

  The solution is not to execute instructions which depend on the result of the memory access until the memory access returns. There are four ways to do this.

  The simplest solution is to flush the pipeline after the memory access and issue NOPs until the memory access returns. This could cause the processor to frequently stall, dramatically reducing performance and making any computer almost unusable. This may be a suitable approach on very simple processors such as in embedded systems with very small memories. However, this is not a viable solution in the general case.

  In Dynamic Scheduling, the hardware reorders the order in which instructions are issued while preserving data flow and exception behaviour. If we reorder the instructions to maximise the amount of time after a memory access before the fetched value is needed, then we can give the instruction time to complete before the processor stalls, allowing dozens or even hundreds of extra cycles before the value is needed. This can virtually eliminate the performance hit from cache hits. However, on cache misses or page faults Dynamic Scheduling will have a small impact. Additionally, Dynamic Scheduling cannot *always* reorder well; meaning sometimes even cache hits will cause stalls.

  Multithreading allows the processor to execute code without any data dependencies. For a single-issue processor[1] such as we are talking about, there are two types of multithreading: fine-grained and course-grained multithreading. In fine-grained multithreading, consecutive instructions are issued by different threads – if one stalls then the other will issue on consecutive cycles until the memory access returns. The second type of multithreading is coarse-grained multithreading; when the first thread stalls the others will start issuing instructions. Implementing either type of multithreading will require multiple sets of registers – one per thread.

  The final "solution" is to context switch. If a process can truly execute nothing for hundreds of thousands or millions of cycles, then yielding may be more efficient. For example if a process with a single thread attempts to read from memory a value which all subsequent instructions depend on and causes a page fault then it may yield and allow another process to execute while the memory access takes place. This decision should be based on an estimate of how long until the memory access will return versus how long it will take to context switch.

[1] talking about data dependency resolution in superscalars, VLIW or EPIC would be dramatic overkill – this would add synchronous multithreading

5. What restrictions on memory accesses are imposed by your ISA?

   I will interpret "your ISA" as the ISA the pipeline in previous sections uses (RISC-V) rather than the ISA my computer uses (x86-64).

   The pipeline in previous stages is using the RISC-V ISA .

   RISC-V imposes the following restrictions on memory accesses:

   - RISC-V is a load-store architecture. This means that only load and store instructions can access memory; arithmetic instructions can only operate on CPU registers. Therefore we cannot update a single-use value in memory without first loading it into a register – potentially displacing the value stored in that register, meaning we have to reloading the original value afterwards.

     For example if we wish to increment all values of a contiguous array (ignoring RISC-V vector extensions); then for each element in the array we have to:

     – Load each element into a register

     – Increment the register

     – Write the element back to memory

     This requires two memory accesses, having a free register and will potentially have the side effect of trashing the cache.

   - RISC-V has no instruction which accesses memory addressed by the sum of two registers and an immediate. This is a common operation in object-oriented languages which is provided by both ARM and x86. RISC-V has to perform an additional machine instructions for many operations such as accessing an elements in nested structs or unions inside structs etc. This means RISC-V is less efficient for object-oriented languages.

   - The ISA will define a maximum pointer size. This limits the size of memory which the CPU can access. For example if we use RV32I, then we cannot access more than 4GB of memory.

6. How does the i386 differ from RISC-V in goals and implementation

   - i386 is a family of processors while RISC-V is an ISA .

     The primary difference between the two is that i386 is a physical processor on which code can be run, while RISC-V is a list of instructions which a processor could implement.

   - i386 is proprietary

     RISC-V was designed to be an open source ISA – this means that anyone can use it and build a processor using RISC-V . i386 was designed by Intel to be built and sold by Intel; it is therefore proprietary.

   - i386 was designed to be backwards compatible

     i386 was designed such that it would be able to run all previous x86 code – both for previous 32-bit architectures and 16-bit architectures. Conversely, RISC-V was a "fresh-start" without any backwards compatibility requirements – although many features were inherited from MIPS .

   - i386 ran on a CISC instruction set while RISC-V uses a RISC instruction set.

     i386 ran on the x86-32 instruction set. This is the 32-bit version of x86. x86-32 was built with two optimisations in mind, ease of programming for assembly-writers and optimising machine code size.

Conversely, RISC-V is designed to make the common case fast. It is designed to have a smaller set of instructions which are easy to implement in hardware and can allow the processor to run at a higher clock frequency.

- x86-32 was designed for assembly writers

  x86-32 was designed with the expectation that people would be routinely writing assembly in it. Many x86-32 instructions are very specific and obscure, but help bridge the semantic gap between high-level programs and machine code.

  RISC-V was designed to be a target for compilers with the expectation that people would almost never directly write assembly. This means many RISC instructions are simple but allow for good compiler optimisation. As a result programming in RISC-V requires far more instructions than the equivalent program in x86-32.

- x86-32 has variable length instructions

  x86-32 instructions are designed such that the most common instructions are short and the least common instructions are very long. The result is that x86-32 code is compressed almost to the entropy limit with instruction sizes varying between 1 and 15 bytes. However, this makes decoding very complicated. Determining where an instruction ends is difficult and we cannot fetch from registers until we have done so. This serialises the decoding process.

  RISC-V has constant-length instructions. Each instruction is 4 bytes (in both 32 and 64-bit systems) and the source and destination registers (if applicable) are all stored in the same place. Therefore, decoding is simple – fetch the next 4 bytes and load from the registers specified at bits [6:11] and [19:24]. Therefore different parts of decoding can be done in parallel and this greatly speeds up execution.

- RISC-V processors can have much higher clock frequencies

  RISC-V has such simple instructions that processors can be clocked far faster than those implementing x86 such as i386. The difference is incomparable, however as i386 was designed in the mid 1980s and therefore the hardware available was much lower quality.

  For example common RISC-V processors can be clocked at 5GHz, while the fastest i386 processor only had a clock speed of 40MHz.

- i386 was built using 1980s ideas and technology

  This means that i386 had very few additional features. For example it did was not a superscalar processor, had no vector extensions and did not support either dynamic execution or speculative execution. Instructions were executed only if they were issued and they were executed in the order in which they occurred in the code. An example of the technology available in the 1980s is that the i386 cache used (the much slower) DRAM rather than SRAM .

  RISC-V was designed with the intent of being run on all types of processors and therefore has support for every modern feature and has space left for many future ideas. RISC-V can have vector extensions, can be executed out-of-order (dynamic scheduling) and is amenable to speculative execution.

- i386 had specialised registers

  x86-32 requires specialised registers, with many instructions performing very specialised operations on single registers. Therefore i386 processors have specialised instructions.

  RISC-V instructions are orthogonal – meaning that any instruction can be applied with any register as argument (with the exception of the destination register being x0) – registers x1–x31 are general purpose.

- i386 uses 32-bit addresses

The i386 family of processors used 32-bit addresses, RISC-V is designed to support a range of address sizes. 32 and 64 bit are both supported and there is work to standardise RV128I which would support 128 bit instructions.

- x86-32 is monolithic

  x86-32 was designed to be implemented in full by powerful microprocessors. This means there is no "minimal subset" of x86-32 which could be implemented. Implementing x86-32 requires implementing thousands of instructions. This makes it very unfriendly for researchers to use.

  Conversely, RISC-V was designed to be implementable by many different types of projects and is therefore segmented into modules. A minimal implementation of RISC-V requires only 50 instructions and other sets of instructions can be added as required. There is also a small set of RISC-V which is purposefully designed for use on tiny embedded systems (RV32E) and requires only 16 registers.

- x86-32 is complete

  Many processors of all complexities have been implemented using x86-32, therefore the instruction set is (largely) bug-free and is very well specified. RISC-V is a very new ISA and is largely incomplete – the basic instructions are well-specified however, more advanced privileged instructions, vector extensions and more are not well-specified.