B5 Pipelined Processor

We have improved the NPC's instruction supply capacity through icache, although the area budget is very limited, adding icache has significantly improved the overall performance of the NPC. The remaining areas for optimization include improving data supply capabilities and computational efficiency.

Evaluating the ideal benefit of dcache

An effective method for improving data supply capacity is to add dcache. In the previous section, we asked you to estimate the ideal benefit of dcache using performance counters. After optimizing icache, the acceleration ratio provided by dcache under ideal conditions may change. Try reevaluating the ideal benefit of dcache.

Evaluate the expected performance of the dcache using cachesim.

Try to improve your cachesim so that it can read mtrace, then evaluate the expected performance of a dcache of a certain size.

You will likely find that, within the remaining area budget, it is challenging to effectively enhance the NPC's data supply capacity through dcache. Therefore, allocating the remaining area to improving computational efficiency is a more scientifically sound decision. The current NPC is multi-cycle, meaning it can only execute one instruction after several cycles. If we can increase the NPC's instruction throughput, we can improve its computational efficiency. Pipeline technology, as an instruction-level parallelism technique, can effectively enhance the NPC's instruction throughput.

Evaluating the ideal gain in computational efficiency

We have not yet introduced the specific implementation of the pipeline, but you can already estimate the ideal gain of pipeline technology based on performance

counters. Assuming that NPC can execute one instruction per cycle except for memory access instructions, try to estimate the ideal acceleration ratio of NPC under the current conditions of icache deficiency.

📢 Evaluate performance gains from a system-wide perspective

This estimate may surprise you.

Remember the example of a 5,000x speedup from Amdahl's law? The performance gains from a single technology, when viewed in isolation, may be vastly different from the gains achieved when that technology is integrated into a full system context. If you implement out-of-order execution and multiple issue, but the instruction and data supply cannot keep pace, in a real chip, these features may simply be a collection of gates that occupy space and consume power but offer little improvement in program performance.

Pipeline Basics

Factory Pipeline

The concept of pipelines also exists in our lives, the most common example being factory pipeline. For example, a factory produces products that need to go through five processes: assembly, labeling, bagging, boxing, and external inspection. If we use 1 to 5 to label these processes and A, B, C, etc. to label different products, then the space-time diagram without using a pipeline would be as follows:

```
          ----> Time
  | Product
  | +---+---+---+---+---+
  V |A.1|A.2|A.3|A.4|A.5|
    +---+---+---+---+---+
                        +---+---+---+---+---+
                        |B.1|B.2|B.3|B.4|B.5|
                        +---+---+---+---+---+
                                            +---+---+---+---+---+
                                            |C.1|C.2|C.3|C.4|C.5|
                                            +---+---+---+---+---+
 ===========================================================================
```

```
          ----> Time
 | Employee
 | +---+               +---+               +---+
 V |A.1|               |B.1|               |C.1|
   +---+               +---+               +---+
       +---+               +---+               +---+
       |A.2|               |B.2|               |C.2|
       +---+               +---+               +---+
           +---+               +---+               +---+
           |A.3|               |B.3|               |C.3|
           +---+               +---+               +---+
               +---+               +---+               +---+
               |A.4|               |B.4|               |C.4|
               +---+               +---+               +---+
                   +---+               +---+               +---+
                   |A.5|               |B.5|               |C.5|
                   +---+               +---+               +---+
```

The space-time diagram with a pipeline would be as follows:

```
          ----> Time
 | Product
 | +---+---+---+---+---+
 V |A.1|A.2|A.3|A.4|A.5|
   +---+---+---+---+---+
       +---+---+---+---+---+
       |B.1|B.2|B.3|B.4|B.5|
       +---+---+---+---+---+
           +---+---+---+---+---+
           |C.1|C.2|C.3|C.4|C.5|
           +---+---+---+---+---+
               +---+---+---+---+---+
               |D.1|D.2|D.3|D.4|D.5|
               +---+---+---+---+---+
                   +---+---+---+---+---+
                   |E.1|E.2|E.3|E.4|E.5|
                   +---+---+---+---+---+
=============================================================
          ----> Time
 | Employee
 | +---+---+---+---+---+
 V |A.1|B.1|C.1|D.1|E.1|
   +---+---+---+---+---+
       +---+---+---+---+---+
       |A.2|B.2|C.2|D.2|E.2|
```

```
        +---+---+---+---+---+
            +---+---+---+---+---+
            |A.3|B.3|C.3|D.3|E.3|
            +---+---+---+---+---+
                +---+---+---+---+---+
                |A.4|B.4|C.4|D.4|E.4|
                +---+---+---+---+---+
                    +---+---+---+---+---+
                    |A.5|B.5|C.5|D.5|E.5|
                    +---+---+---+---+---+
```

As can be seen, in the pipeline method, although the production time of each product has not been reduced, since each employee remains in a working state, they can continuously process the same process for different products, so that, overall, one product is completed at any given moment, thereby increasing the throughput of the production line.

Instruction Pipeline

Similar to a factory pipeline, processors can also use pipelines to execute instructions. We divide the instruction execution process into several stages, allowing each component to handle one stage, and keeping these components in working state so that they can continuously handle the same stage of different instructions. This means that, overall, one instruction is completed in each cycle, thereby improving the processor's throughput.

When learning bus, we asked everyone to upgrade NPC to a distributed control multi-cycle processor. Multi-cycle processors already have the concept of stages, and their working process is very similar to the non-pipeline method in the factory scenario mentioned above. Therefore, it is not difficult to understand how instruction pipelines work.

We can perform a simple analysis and evaluation of the performance of several processors. Assuming that the processor's work is divided into five stages: instruction fetch (IF), decoding (ID, instruction decode), execution (EX), load-store (LS), and write back (WB), their logical delays are all 1 ns, and we will not consider the delays of instruction fetch and load-store for now.

1. Single-cycle processor: No registers between stages, so the critical path is 5 ns and the frequency is 200 MHz. Each instruction takes 1 cycle to execute, i.e., 5 ns; Each cycle executes 1 instruction, i.e., `IPC = 1`.
2. Multi-cycle processor: There are registers between stages, so the critical path is 1 ns and the frequency is 1000 MHz. Each instruction takes 5 cycles to execute, i.e., 5 ns; One instruction is executed every 5 cycles, i.e., `IPC = 0.2`.

3. Pipelined Processor: There are registers between stages, so the critical path is 1 ns and the frequency is 1000 MHz. Each instruction takes 5 cycles to execute, i.e., 5 ns; Each cycle executes 1 instruction, i.e., `IPC = 1` ..

Processor	Frequency	Instruction Execution Delay	IPC
Single cycle	200MHz	5ns	1
Multiple cycles	1000MHz	5ns	0.2
Pipelined	1000MHz	5ns	1

As can be seen, although the instruction execution delay is still 5ns, the pipeline has the advantages of high frequency and high IPC. These advantages are essentially brought about by instruction-level parallelism technology: each cycle of the pipelined processor processes five different instructions.

Of course, the above data is only obtained from an ideal situation. If we consider the load-store operations in SoC, the IPC will not be as high. In addition, pipelined processors cannot always execute five instructions in each cycle, which will be further analyzed below.

> **Longer Pipelines**
>
> The above example only divides the pipeline into five stages. In fact, we can divide the pipeline into more stages, making the logic of each stage simpler and thereby increasing the overall frequency of the processor. This type of pipeline is called “superpipeline.” For example, if the pipeline can be divided into 30 stages, according to the above estimate, the frequency can theoretically reach 6GHz.
>
> However, current mainstream high-performance processors generally only divide the pipeline into about 15 stages. What factors do you think may make it inappropriate to divide the pipeline into too many stages?

Simple implementation of Pipeline

It is not difficult to implement a pipelined processor based on a multi-cycle processor with a handshake mechanism. For each stage's input `in` and output `out`, we only need to correctly process the following signals (`bits` refers to the load that needs to be transferred between stages):

- `out.bits`, generated by the current stage
- `out.valid`, generated by the current stage, usually related to `in.valid`
- `in.ready`, generated by the current stage, set to invalid when busy, and set to valid after the current instruction is processed
- `out.ready`, identical to the next stage's `in.ready`
- `in.bits`, when both the current stage's `in.ready` and the previous stage's `out.valid` are valid, it is updated to the previous stage's `out.bits`
- `in.valid`, left as an exercise for you

Based on the above content, we can package the processing of the last three signals into a function, and then use it to connect the various stages:

```
def pipelineConnect[T <: Data, T2 <: Data](prevOut: DecoupledIO[T],
  thisIn: DecoupledIO[T], thisOut: DecoupledIO[T2]) = {
    prevOut.ready := thisIn.ready
    thisIn.bits := RegEnable(prevOut.bits, prevOut.valid &&
thisIn.ready)
    thisIn.valid := ???
  }

pipelineConnect(ifu.io.out, idu.io.in, idu.io.out)
pipelineConnect(idu.io.out, exu.io.in, exu.io.out)
pipelineConnect(exu.io.out, lsu.io.in, lsu.io.out)
// ...
```

Specifically, the IFU does not need to wait for the current instruction to finish executing before fetching the next instruction. The `RegEnable` mentioned above serves the role of the “pipeline stage register” described in traditional textbooks, but we can also understand it from the bus perspective as a buffer for downstream modules to receive messages: After successful handshaking between upstream and downstream modules, the upstream module assumes the downstream module has successfully received the message, so it no longer retains the message. Therefore, the downstream module must record the received message in the buffer to prevent message loss. For the first three signals, since their specific logic is related to the behavior of the current stage, they must be implemented in the corresponding module of the current stage.

In particular, there are some situations in pipelined processors where the current instruction cannot be executed, which is called a “hazard.” Hazards are mainly divided into three categories: structural hazards, data hazards, and control hazards. If hazards are ignored and execution is forced, it will cause the CPU state machine transfer results to be inconsistent

with the ISA state machine, resulting in the instruction execution results not matching their semantics. Therefore, in pipeline design, we need to detect hazards and either eliminate them through hardware design or wait for the hazards to no longer occur. For the latter, this can be achieved by adding wait conditions to `in.ready` and `out.valid`.

Structural Hazard

Structural hazards occur when different stages in the pipeline need to access the same component simultaneously, but the component cannot support simultaneous access by multiple stages. For example, in the instruction sequence shown in the figure below, at time T4, I1 is reading data in the LSU, and I4 is fetching instructions in the IFU, both of which require memory access; at time T5, I1 is writing registers in the WBU, and I4 is reading registers in the IDU, both of which require access to the register file.

```
            T1   T2   T3   T4   T5   T6   T7   T8
          +----+----+----+----+----+
I1: lw    | IF | ID | EX | LS | WB |
          +----+----+----+----+----+
               +----+----+----+----+----+
I2: add        | IF | ID | EX | LS | WB |
               +----+----+----+----+----+
                    +----+----+----+----+----+
I3: sub             | IF | ID | EX | LS | WB |
                    +----+----+----+----+----+
                         +----+----+----+----+----+
I4: xor                  | IF | ID | EX | LS | WB |
                         +----+----+----+----+----+
```

Some structural hazards can be completely avoided in hardware design so that they do not occur during CPU execution: We only need to enable these components to support simultaneous access by multiple stages in the hardware design. Specifically:

- For the register file, we only need to implement its read port and write port independently, allowing the IDU to access the register file through the read port and the WBU to access the register file through the write port
- For memory, there are multiple solutions:
  - Separate read and write ports like the register file to implement true dual-port memory
  - Divide memory into instruction memory and data memory, which operate independently
  - Introduce a cache; if the cache hit occurs, there is no need to access memory

Differences between textbooks and real systems

Most solutions in textbooks are based on simplified assumptions, which may not hold true in real processors, so they may not be suitable for use in OSOC scenarios.

For example, SDRAM memory chips cannot separate read ports from write ports. Both READ and WRITE commands are transmitted to SDRAM chips via the SDRAM memory bus. Dividing memory into instruction memory and data memory will cause instructions and data to be located in different address spaces, which violates the ISA specification's memory model of using a unified address space for instructions and data. On the one hand, modern compilers cannot compile programs that adapt to the above scheme; on the other hand, even if assembly language is used to develop programs, programs such as bootloader's loader cannot run correctly.

In fact, OSOC places higher demands on everyone: learning to evaluate a scheme from a system-wide perspective. The simplified assumptions in textbooks can help everyone focus on learning the current knowledge points, but in the future, you will face real projects, and only by learning to connect the various factors in the system, you will be able to make reasonable and effective decisions in your future work.

Some structural hazards cannot be completely avoided, such as:

- When the cache is missing, IFU and LSU still need to access memory at the same time
- The SDRAM controller queue is full and cannot continue to receive requests
- The divider calculation takes dozens of cycles, and another calculation cannot be started before the previous one is completed

To deal with the above situation, a simple approach is to wait: If IFU and LSU perform load-store operations at the same time, let one wait for the other; Wait until the SDRAM controller queue has free space; Wait until the divider completes the current calculation. The good news is that the bus inherently has a wait function, so as long as the slave device, downstream module, or arbitrator sets ready to invalid, the detection and handling of structural hazards can be reduced to the bus state machine, eliminating the need to implement dedicated structural hazard detection and handling logic.

Who waits for whom?

> While waiting, should IFU wait for LSU, or should LSU wait for IFU? Or are both options possible? Why?

Data Hazard

Data hazards occur when instructions at different stages depend on the same register data, and at least one instruction writes to that register. For example, in the instruction sequence shown in the figure below, I1 writes to register a0, but the write operation is not completed until the end of time T5. Before that, I2 reads the old value of a0 at time T3, I3 reads the old value of a0 at time T4, I4 reads the old value of a0 at time T5, and I5 can only read the new value of a0 at time T6.

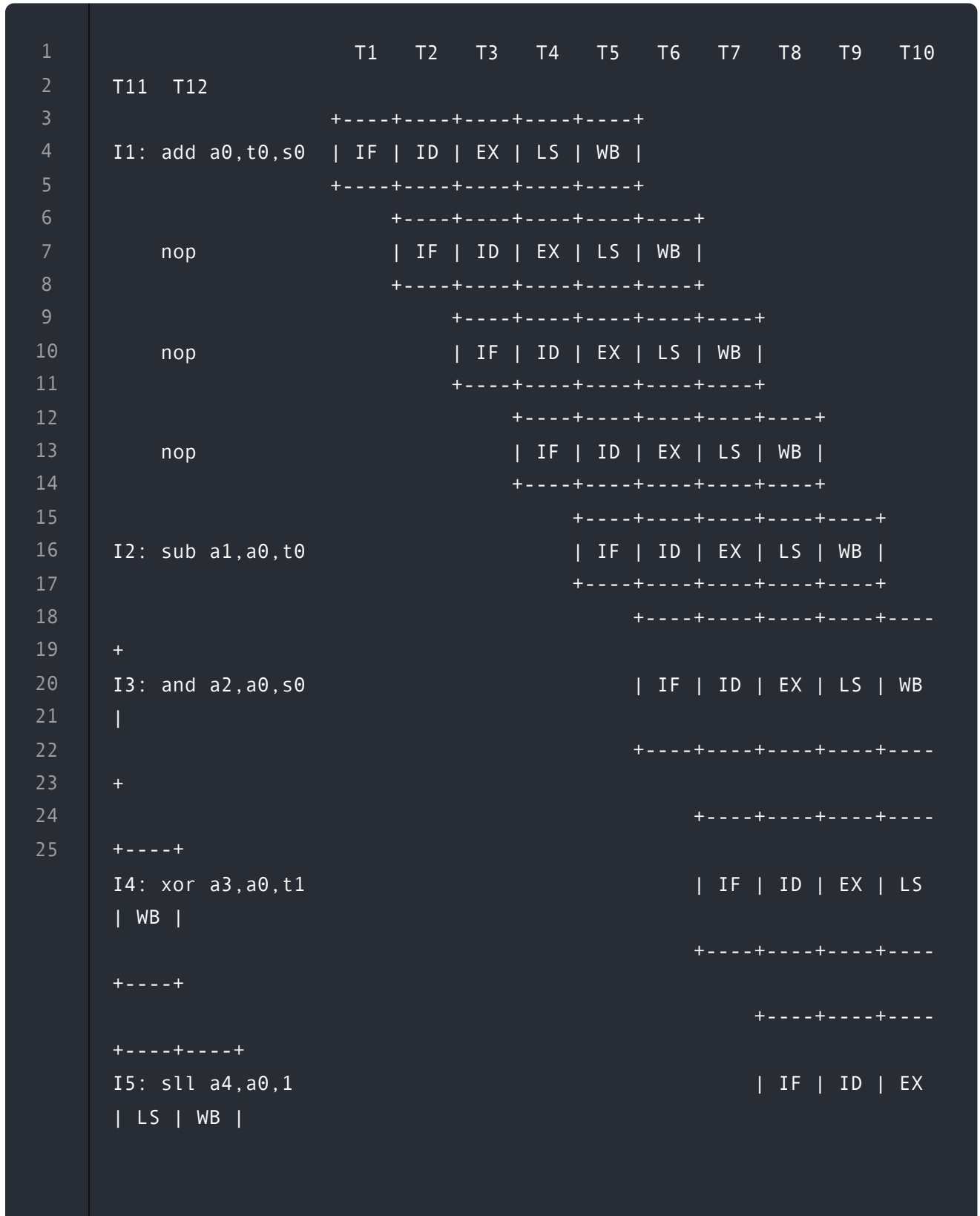
```
                    T1   T2   T3   T4   T5   T6   T7   T8   T9
                  +----+----+----+----+----+
I1: add a0,t0,s0  | IF | ID | EX | LS | WB |
                  +----+----+----+----+----+
                       +----+----+----+----+----+
I2: sub a1,a0,t0       | IF | ID | EX | LS | WB |
                       +----+----+----+----+----+
                            +----+----+----+----+----+
I3: and a2,a0,s0            | IF | ID | EX | LS | WB |
                            +----+----+----+----+----+
                                 +----+----+----+----+----+
I4: xor a3,a0,t1                 | IF | ID | EX | LS | WB |
                                 +----+----+----+----+----+
                                      +----+----+----+----+----+
I5: sll a4,a0,1                       | IF | ID | EX | LS | WB |
                                      +----+----+----+----+----+
```

The above data hazard is called a read after write (RAW) hazard. The characteristic of a RAW hazard is that one instruction needs to write to a certain register, while another younger instruction needs to read that register. Obviously, if this data hazard is not handled, instructions I2, I3, and I4 will calculate the wrong result because they read the old value of a0, violating the semantics of instruction execution.

> **The sequential relationship between instructions**
>
> In dynamic instruction sequences, “older” and “younger” are generally used to describe the sequential relationship between different instructions. For example, “I1 is

older than I2" indicates that I1 is executed before I2 in terms of time; "I2 is younger than I1" indicates that I2 is executed after I1 in terms of time.

There are several ways to solve the RAW hazard. From a software perspective, instructions are generated by the compiler, so one way is to have the compiler detect the RAW hazard and insert a nop instruction to wait for the instruction writing to the register to complete, as shown in the figure below:



```
                                                                +----+----+----
+----+----+
```

The essence of inserting empty instructions is still waiting, but compilers can actually do better: Instead of waiting, it is better to execute some meaningful instructions. This can be achieved by letting the compiler perform instruction scheduling. The compiler can try to find instructions that have no data dependencies and adjust their order without affecting the behavior of the program. An example is shown below, where I6, I7, and I8 have no data dependencies with I1, so they can be scheduled to execute after I1:

```
I1: add a0,t0,s0          I1: add a0,t0,s0
I2: sub a1,a0,t0          I6: add t5,t4,t3
I3: and a2,a0,s0          I7: add s5,s4,s3
I4: xor a3,a0,t1   --->   I8: sub s6,t4,t2
I5: sll a4,a0,1           I2: sub a1,a0,t0
I6: add t5,t4,t3          I3: and a2,a0,s0
I7: add s5,s4,s3          I4: xor a3,a0,t1
I8: sub s6,t4,t2          I5: sll a4,a0,1
```

> **Compilers and Out-of-Order Execution Processors**
>
> If we delegate the compiler's instruction scheduling task to the hardware, we get an out-of-order execution processor. Of course, in order to perform instruction scheduling in hardware, we need to add a number of additional hardware modules. However, fundamentally, both of these technologies are designed to improve the efficiency of the processor's instruction execution.

However, when it comes to instruction scheduling, compilers can only do their best and cannot always find suitable instructions. For example, division instructions require dozens of cycles to execute, and it is usually difficult for compilers to find so many suitable instructions. In such cases, if the compiler is to handle RAW hazards, it can only insert empty instructions.

Worse still, some RAWs cannot be resolved by the compiler alone. Consider a load instruction as the dependent instruction. This type of RAW hazard is called a load-use hazard:

```
                    T1   T2   T3  ....  T?   T?   T?   T?   T?   T?
T?
                  +----+----+----+-------------+----+
I1: lw  a0,t0,s0  | IF | ID | EX |      LS     | WB |
                  +----+----+----+-------------+----+
                       +----+----+----+----+----+
    nop X ?            | IF | ID | EX | LS | WB |
                       +----+----+----+----+----+
                                                 +----+----+----+----
+----+
I2: sub a1,a0,t0                                 | IF | ID | EX | LS
| WB |
                                                 +----+----+----+----
+----+
```

In fact, in real SoCs, it is almost impossible for software to predict the latency of a load-store instruction when it is executed in the future:

- If the cache hit, the data may return after 3 cycles
- If the cache miss, accessing SDRAM may result in the data returning after 30 cycles
- If it happens to coincide with SDRAM charge refresh, the data may return after 30+ cycles
- If the CPU frequency is increased from 500MHz to 600MHz, the number of cycles required for data return increases

For this reason, modern processors almost all use hardware to detect and handle RAW hazards. Since register write operations occur in the WBU, the register numbers to be written are also propagated to the WBU along with the pipeline. In other words, we can find which register the corresponding instruction will write to in each stage. If the register to be read by the instruction located in the IDU is the same as the register to be written in a subsequent stage, a RAW hazard occurs:

```
def conflict(rs: UInt, rd: UInt) = (rs === rd)
def conflictWithStage[T <: Stage](rs1: UInt, rs2: UInt, stage: T) =
{
  conflict(rs1, stage.rd) || conflict(rs2, stage.rd)
}
val isRAW = conflictWithStage(IDU.rs1, IDU.rs2, EXU) ||
            conflictWithStage(IDU.rs1, IDU.rs2, LSU) ||
            conflictWithStage(IDU.rs1, IDU.rs2, WBU)
```

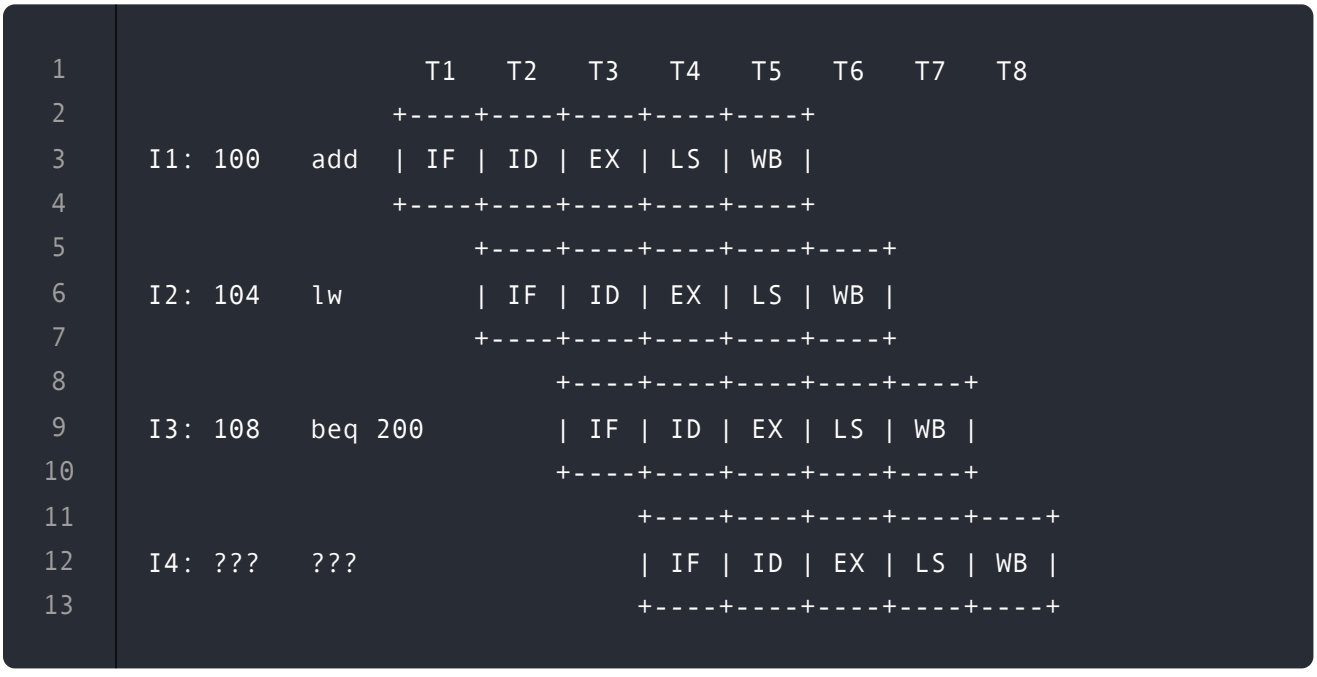
The pseudocode above is just a rough idea. In reality, you need to consider more issues: Not all instructions need to be written to registers, not all stages are executing instructions, not all instructions need to read rs2 (such as U-type instructions), and the value of the zero register is always `0`, etc. How to write the correct RAW detection code is up to you to figure out.

After detecting a RAW hazard, the simplest way to handle it is to wait: just set `in.ready` and `out.valid` to invalid. It can be seen that this hardware detection and handling of RAW hazards does not require advance knowledge of when the instruction execution will end. This is because various waits during instruction execution are passed to the pipeline through the bus handshake signal. Therefore, it is more applicable than the above software solution.

```
                    T1   T2   T3   T4   T5   T6   T7   T8   T9   T10
T11  T12
                  +----+----+----+----+----+
I1: add a0,t0,s0  | IF | ID | EX | LS | WB |
                  +----+----+----+----+----+
                       +----+-------------------+----+----+----+
I2: sub a1,a0,t0       | IF |         ID        | EX | LS | WB |
                       +----+-------------------+----+----+----+
                            +-------------------+----+----+----+----
+
I3: and a2,a0,s0            |         IF        | ID | EX | LS | WB
|
                            +-------------------+----+----+----+----
+
                                                +----+----+----+----
+----+
I4  xor a3,a0,t1                                | IF | ID | EX | LS
| WB |
                                                +----+----+----+----
+----+
                                                     +----+----+----
+----+----+
I5: sll a4,a0,1                                      | IF | ID | EX
| LS | WB |
                                                     +----+----+----
+----+----+
```

Control Hazards

Control hazard refers to a situation where a jump instruction changes the order of instruction execution, causing the IFU to retrieve instructions that should not be executed. For example, in the instruction sequence shown in the figure below, which instruction the IFU of T4 should retrieve can only be known after I3 calculates the jump result at time T5.



In addition to the above branch instructions, `jal` and `jalr` also cause similar problems. Assuming that I3 in the above figure is a jump instruction, we expect to retrieve the instruction at the jump target at time T4. At time T4, the IDU is decoding I3, so logically it should be able to catch up. However, modern processors generally consider that it cannot catch up, and therefore treat it as a control hazard.

> **Why do modern processors handle this in this way?**
>
> In fact, some textbooks do handle control hazards in the manner described above. What factors do you think prevent the textbook approach from being used in actual processor designs?

Even CPU exceptions can cause control hazards. When an exception is thrown, the instruction must be fetched immediately from the memory location pointed to by mtvec, but generally speaking, the processor cannot know at the time of fetching whether the execution of this instruction will cause an exception.

The above problems are all caused by the inability to determine which instruction needs to be fetched next during the instruction fetch phase. If you choose to wait, you have to wait until the previous instruction is almost complete before you can know the actual address of

the next instruction. For example, load-store instructions have to wait until the load-store operation is complete before the `resp` signal on the bus can confirm that no exceptions were thrown during the load-store process. Obviously, this solution will cause the instruction pipeline to stall, greatly reducing the processor's instruction execution throughput. If you choose not to wait, you may fetch some instructions that should not be executed. If no further processing measures are taken, the processor's state transition will be inconsistent with the ISA state machine, resulting in incorrect execution results.

In order to deal with control hazards, modern processors often use speculative execution technology. Speculative execution is essentially a prediction technology. The basic idea is to try to guess a choice while waiting. If the guess is correct, it is equivalent to making the correct choice in advance, thereby saving the overhead of waiting. Speculative execution consists of three parts:

- Selection strategy - Before obtaining the correct result, a choice is speculated based on a certain strategy
- Checking mechanism - When the correct result is obtained, the previously speculated choice is checked to see if it matches the correct result
- Error recovery - If the check reveals a mismatch, the system must roll back to the state at the time of the selection strategy, and make the correct choice based on the obtained correct result

For control hazards, the simplest speculative execution strategy is to “always speculate that the next static instruction will be executed.” Considering the implementation of this strategy from the above three aspects, the details are as follows:

- Selection strategy - Very simple, just let the IFU continuously fetch instructions from `PC + 4`.
- Checking mechanism - According to the semantics of the instruction, only when executing branch and jump instructions, as well as when throwing exceptions, is it possible for the CPU to change the execution flow. In other cases, execution is sequential. Therefore, in other cases, the above speculative selection is always correct, and no additional checks are required. Therefore, it is only necessary to check whether the jump result is consistent with the speculative selection when executing branch and jump instructions, as well as when throwing exceptions. In other words, check whether the jump result is `PC + 4`.
- Error recovery - If the above jump result is not `PC + 4`, it means that the previous speculation was wrong, and the instructions taken based on this speculation should not be executed, but should be eliminated from the pipeline. This action is called “flushing”; at the same time, the IFU needs to take instructions from the correct jump result.

The performance improvement brought about by speculative execution is related to the accuracy of the speculation. If the accuracy of the speculation is high, the IFU can retrieve the correct instruction in advance with a high probability, thereby saving the overhead of waiting. If the accuracy of the speculation is low, the IFU often fetches instructions that should not be executed, and these instructions are subsequently flushed. During this period, the behavior of the pipeline is equivalent to not executing any valid instructions, thereby reducing the throughput of executing instructions. Specifically:

- Since exceptions are low-probability events in the processor execution process, the vast majority of instructions will not throw exceptions when executed. Therefore, for exceptions, the accuracy rate of the above strategy is close to 100%.
- The execution results of branch instructions are only taken and not taken. The above strategy is equivalent to always predicting “not taken.” Therefore, in terms of probability, for branch instructions, the accuracy rate of the above strategy is close to 50%.
- The behavior of a jump instruction is to jump unconditionally to the target address, but there are many possible target addresses. The probability of jumping exactly to `PC + 4` is very low. Therefore, for jump instructions, the accuracy of the above strategy is close to 0%.

Based on the above analysis, speculative execution can correctly handle control hazards on the one hand, and on the other hand, it can also bring certain performance improvements compared to passive waiting. However, for branch instructions and jump instructions, there is still room for improvement in the above speculative execution scheme, which we will discuss further below.

There are some details to note regarding the implementation of speculative execution:

- From a demand perspective, flushing is intended to restore the processor's state to the moment before the control hazard occurred. Therefore, we can deduce how to handle the relevant implementation details from the perspective of the state machine. The state machine perspective tells us that the state of the processor is determined by the sequential logic circuit, and the state update of the processor is controlled by the control signal. Therefore, to achieve the effect of flushing, we only need to consider setting the relevant control signals to invalid. For example, by setting `valid` to invalid, most of the instructions being executed by the components can be flushed directly.
- However, if there are still some states within the components that affect control signals, additional considerations are needed, such as the state machine in the I-cache, especially since issued AXI requests cannot be retracted, so we must wait for the request to complete.

- Speculative execution means that the currently executed operation may not necessarily be needed in the future. If the speculation is wrong, the relevant operation should be canceled. However, some operations are difficult to cancel, including updating the register file, updating the CSR, writing to memory, accessing peripherals, etc. Once the state of these modules changes, it is difficult to restore them to their previous state. Therefore, the state of these modules can only be updated after confirming that the speculation is correct.

Implement a simple pipelined processor

Handle various hazards in the simplest way possible to implement the most basic pipeline structure. After implementation, try running microbench to check if your implementation is correct.

Hint:

- In order for DiffTest to work correctly, you may need to modify the signals passed to the simulation environment.
- For now, you can ignore the implementation related to exception handling. We will implement it later.

Nested speculative execution

Think about it: if you encounter consecutive branch instructions or jump instructions, will your design still work correctly?

For the implementation of exceptions in pipelined processors, we also need to consider the following details:

- When throwing an exception, `mepc` needs to be set to the PC value of the instruction where the exception occurred. This feature is called “precise exception.” If this property is not satisfied, when returning from exception handling through `mret`, it is impossible to return precisely to the instruction where the exception occurred, resulting in inconsistency between the states before and after the exception. This may cause the system software to be unable to use the exception mechanism to implement certain key mechanisms of modern operating systems, such as process switching and page scheduling. However, in a pipelined processor, the PC of the IFU changes constantly, and by the time an exception is thrown during the execution of an instruction, the PC of the IFU no longer matches that

instruction. In order to obtain the PC that matches the instruction, we need to pass the corresponding PC downstream when the IFU fetches the instruction.

- An instruction may throw an exception during speculative execution, but if the speculation is incorrect, then the instruction should not actually be executed, and the thrown exception should not be handled. Therefore, operations that require updating the processor state during exception handling must wait until the speculation is confirmed to be correct before they can actually be performed, including writing to `mepc`, `mcause`, jumping to the location indicated by `mtvec`, etc.
- The update of `mcause` depends on the type of exception. In RISC-V processors, different exception numbers are generated by different components. The generated exception numbers also need to be passed to the downstream of the pipeline, and only after the speculation is confirmed to be correct can `mcause` be actually written.

Exception Number	Exception Description	Component Where Exception First Occurred
0	Instruction address misaligned	IFU
1	Instruction access fault	IFU
2	Illegal Instruction	IDU
3	Breakpoint	IDU
4	Load address misaligned	LSU
5	Load access fault	LSU
6	Store/AMO address misaligned	LSU
7	Store/AMO access fault	LSU
8	Environment call from U-mode	IDU
9	Environment call from S-mode	IDU
11	Environment call from M-mode	IDU
12	Instruction page fault	IFU
13	Load page fault	LSU
15	Store/AMO page fault	LSU

Implement a pipeline that supports exception handling

Based on the above content, implement a pipeline that supports exception handling. After implementation, run some exception handling-related tests to check if your implementation is correct.

Multiple exceptions occur

In a pipelined processor, different instructions are executed at different stages, which means that different exceptions may occur simultaneously at different stages. For example, while the IFU finds that the instruction address is not aligned, the IDU finds an illegal instruction, and the LSU finds a load-store error. How should this be handled?

Although we do not require you to implement all exceptions at this time, you can still think about whether your design can handle these situations correctly.

Intel's Superpipeline Architecture

Prior to 2005, Intel used superpipeline technology to pursue the ultimate clock speed. In February 2004, Intel released a processor with the code name **Prescott**, which had an unprecedented 31-stage pipeline depth and a clock speed of 3.8GHz, even with the 90nm process node at the time. However, actual measurements showed that compared to the previous generation 20-stage pipeline Northwood, the performance of this processor **did not improve much**, but instead became the hottest and most power-consuming single-core processor in x86 history. In fact, if it weren't for cooling and power consumption issues, the clock speed of this processor could have been even higher than 3.8GHz: **A year before its release, Intel claimed that the clock speed of this processor could reach up to 5GHz**.

From a microarchitecture perspective, the execution efficiency of the 31-stage pipeline was not ideal either. On the one hand, the instructions in the pipeline were full of data hazards, and many instructions could only wait in the pipeline due to RAW hazards. On the other hand, the cost of flushing the pipeline due to speculative execution errors in branch instructions is very high. Assuming that the processor calculates whether to take a branch instruction at the 26th level, in the case of a speculative execution error, all instructions taken in the previous 25 cycles need to be

> flushed. In particular, Prescott is a multi-issue processor **that can perform four simple ALU operations per cycle**, estimating it to be a 4-issue processor, in the case of speculative execution errors, it needs to flush 25 * 4 = 100 instructions! Although Prescott uses some advanced technologies to improve the accuracy of speculation, according to actual measurement results, **the performance of many programs still declines due to the excessive length of the pipeline**.
>
> Later, Intel abandoned this aggressive superpipeline technology route, **the pipeline depth of subsequent architectures is generally around 15 levels, with a maximum of 20 levels**.

Testing and Verification of Pipelined Processors

The processing object of the pipeline is instructions, which means that different instruction sequences will have different effects on the behavior of the pipeline. Therefore, the verification process of the pipeline should use instruction sequences as test inputs and cover as many different situations as possible. According to the behavior of instructions in the pipeline, they can be roughly divided into the following 10 types:

- Instructions that can be completed by the ALU in one cycle, such as addition, subtraction, logic, and shift
- Control flow transfer instructions, divided into three types: conditional branch, `jal`, and `jalr`
- Load-store instructions, divided into two types: load and store
- CSR instructions
- `ecall`, `mret`
- `fence.i`

Just considering the combinations of the above 10 types of instructions in a traditional 5-stage pipeline, there are already $10^5 = 100000$ possibilities. In fact, we also need to consider the various hazards mentioned above: load-store may need to wait, there are data dependencies between instructions, control flow transfer instructions may cause pipeline flushes... In short, there are too many instruction sequences formed by different combinations of situations. Even if an instruction generator is developed, it is difficult to ensure that all of the above situations are covered. It is even more difficult to design so many test cases manually.

> **The correctness of the processor cannot be proven by traditional instruction test sets.**
>
> If you are familiar with instruction test sets such as `riscv-tests`, you need to understand that passing `riscv-tests` does not mean that the pipeline NPC is correct. In fact, the number of test cases provided in `riscv-tests` is far from the rough estimate of 100,000 mentioned above. This already indicates that there must be certain instruction sequences that cannot be covered by `riscv-tests`. More fundamentally, `riscv-tests` tests the behavior of a single instruction, while to fully test a pipelined processor, it is necessary to traverse various instruction sequences. Therefore, if your pipelined NPC only passes the traditional instruction test set `riscv-tests`, you should not be 100% confident in your NPC implementation.

Since it is difficult for humans to design test cases, let's find a way to delegate this task to tools! Let's review the formal verification tool we used to verify the cache. It can automatically help us find test cases that violate `assert`. If no such cases are found, it proves that the design is correct. If we can apply formal verification to the pipeline NPC, we can let the tool help us automatically find incorrect instruction sequences!

To use formal verification, we also need a REF and to write appropriate verification conditions (i.e., `assert`). For REF, we need another implementation that can correctly execute the instruction sequence. Since the formal verification tool we use can only perform verification at the RTL level, it is currently not possible to access instruction set emulators such as NEMU and Spike. However, we can consider the single-cycle NPC developed earlier, which is also a microarchitecture implementation of the RISC-V ISA and can necessarily execute the instruction sequence correctly.

As for the verification conditions, one idea is to check whether the states of the GPRs of the DUT and REF are consistent after they have executed the same instructions, as in DiffTest. In C code, checking whether all GPRs are consistent is just a single `memcmp()` call, which is not very costly; but in formal verification, checking whether all GPRs are consistent is treated as a constraint condition for “solving equations,” which can significantly increase the computational cost of the BMC solving process. Therefore, it is necessary to find a simpler comparison method.

Reflecting on the state machine model, the new state depends on the current state and the state transition. Therefore, in addition to directly comparing the new states, we can also compare them from another perspective: If the current states are consistent and the state

transitions are also consistent, then the new states should also be consistent. Generally speaking, describing a state transition is much simpler than describing the state itself (i.e., the state space), so comparing whether the state transitions are consistent is typically a simpler comparison method. Taking the above GPR state as an example, although the GPR state space has $16 \times 32 = 512$ bits, a RISC-V instruction can write to at most one GPR. We only need to record which GPR this instruction updates to which value, and compare whether the DUT and REF records are consistent, without having to directly compare whether the 512-bit GPR state space is consistent, thus greatly simplifying the comparison overhead.

Based on the analysis above, it is easy to write pseudocode for verifying the top-level module. Here, we use Chisel as pseudocode. If you are developing in Verilog, you can still draw on the relevant ideas to write pseudocode for verifying the top-level module.

```
class PipelineTest extends Module {
  val io = IO(new Bundle {
    val inst = Input(UInt(32.W))
    val rdata = Input(UInt(XLEN.W))
  })

  val dut = Module(new PipelineNPC)
  val ref = Module(new SingleCycleNPC)

  dut.io.imem.inst  := io.inst
  dut.io.imem.valid := ...
  dut.io.dmem.rdata := io.rdata
  dut.io.dmem.valid := ...
  // ...

  ref.io.imem.inst := dut.io.wb.inst
  // ...

  when (dut.io.wb.valid) {
    assert(dut.io.wb.rd  === ref.io.wb.rd)
    assert(dut.io.wb.res === ref.io.wb.res)
  }
}
```

The above pseudocode only provides a rough framework. There are still many details that you need to pay attention to and supplement:

- In order to reduce the solution space of BMC, the DUT only includes the pipeline itself, without the cache and various peripheral modules. However, the load-store delay caused by cache misses can be achieved through handshake signals.
- Instructions are one of the inputs for verifying the top-level module, which means that the BMC will traverse various instructions. Under the influence of the bound, the BMC will traverse various combinations of instructions in different cycles, which means that all instruction sequences of a given length will be traversed.
- The generated instruction sequence will be input into the IFU of the DUT in order, but since it takes several cycles to complete the execution of an instruction in a pipeline NPC, while it only takes one cycle in a single-cycle NPC, it is necessary to synchronize between the DUT and REF so that after the DUT completes the execution of an instruction, the instruction is only input into the REF. Therefore, it is necessary to obtain the currently completed instruction in the WBU of the DUT.
- In addition to comparing the written GPR number `rd` and its value `res`, it is also necessary to compare the PC to check whether the control flow transfer is correct. However, we can choose to compare the new value of the PC, which allows us to detect PC inconsistency errors one cycle earlier.
- Some instructions will not update GPR. For these instructions, we do not need to compare `rd` and `res`.
- For load instructions, the load-store results are also used as one of the inputs to the pipeline, so they need to be reflected in the ports of the top-level module. To ensure that the DUT and REF are in the same state after executing the same load instruction, it is also necessary to ensure that they read the same data.
- We do not perform a specific comparison of the execution results of store instructions. There are two main considerations for this:
  1. Both the write data and write address come from GPR. If they are incorrect, then there must be an instruction older than the store instruction that wrote the incorrect value to GPR. This situation can be checked using the above mechanism.
  2. If the write data and write address are correct, but the AXI write channel signal is incorrect, this problem belongs to the scope of AXI bus implementation, not the scope of pipeline verification. In principle, checks for AXI signals can also be added to the pipeline verification framework, but this will increase the constraints on BMC, thereby increasing the cost of solving.
- The handshake signals between IFU and LSU must be handled correctly.
- We do not perform a specific comparison for CSR writing either, because if there is an error in the implementation of a certain CSR writing, the value of that CSR can be read into GPR with an additional instruction, exposing the incorrect value, thus reducing the comparison of CSR writing to the comparison of GPR writing.

- It is also necessary to ensure that GPR and CSR have the same initial state in both DUT and REF. Therefore, GPR and CSR need to be initialized. Note that this is only a requirement for formal verification. During simulation and tape-out, not all GPR and CSR need to be initialized.
- Since the instruction sequence is generated by BMC, if no restrictions are imposed, the generated instruction sequence may contain illegal instructions. If the NPC does not support the handling of illegal instruction exceptions, the behavior of the NPC executing illegal instructions is undefined, which is not beneficial to comparing the execution results of instruction sequences. To solve this problem, one idea is to only allow BMC to generate legal instructions. This can be achieved by using the `assume` statement provided by both Chisel and SystemVerilog, which is used to express the preconditions for verification. For example, if `isIllegal` indicates that "the instruction decoding result is an illegal instruction," then `assume(!isIllegal)` indicates that "the instruction decoding result is not an illegal instruction" as a prerequisite for verification. At this point, BMC will solve under this prerequisite, thereby eliminating illegal instructions in the instruction sequence.
- The CSR register addresses in CSR instructions also need to be considered. Similarly, you can use the `assume` statement to restrict the CSR register addresses in CSR instructions to the range of implemented CSRs.
- Another consideration is unaligned memory access. If no restrictions are imposed, the addresses calculated in the generated memory access instructions may be unaligned. This issue can also be resolved using the `assume` statement.

Switching to a more efficient model checking tool

On 2024/08/20 02:30:00, we modified the way we call the formal verification tool, replacing the call to the Z3 solver with a call to the BtorMC model checker to improve the efficiency of formal verification. If you are developing with Chisel, please refer to the "Simple Example of Formal Verification" section in the caching section.

Implementing Pipeline Testing with Formal Verification

Although it is not mandatory, we strongly recommend that you test your pipeline with formal verification. However, you need to carefully consider how to write the verification conditions for `assert` and `assume`. If they are written incorrectly, it may lead to false positives or false negatives: False positives can be found and fixed during debugging, but false negatives are difficult to find. Therefore, this task is

essentially to test whether everyone has a deep enough understanding of the details of the pipeline.

In addition, regarding the bounds of BMC, you can choose an appropriate parameter so that the formal verification tool can traverse a sufficient number of instruction sequences and cover various hazard combinations. Generally speaking, the bound may need to be greater than 10, but this requires the solution process to take several hours or even tens of hours. However, in terms of the convenience of the tool, it is still very worthwhile, because if you write test cases manually, even if it takes several days, you may not be able to write test cases that cover some extreme cases. However, in order to quickly find some counterexamples, you can start testing with a small bound and then gradually test larger bounds.

Using formal verification to test more complex processors

A research team at the Institute of Software, Chinese Academy of Sciences, has developed a formal verification-based RISC-V processor testing framework. Through this framework, they even found some very hidden bugs in the **NutShell Processor** that can boot Linux. For details, please refer to their **nutshell-fv** project. This case also highlights the advantages of formal verification technology. However, since the current NPC functionality has been significantly simplified, to integrate it into this testing framework, you may need to make some adjustments to your code and the testing framework's code. If you are interested, you can read the README in the project to learn how to use it, and review the relevant code to understand the details of the testing framework.

Make the pipeline flowing

After implementing a simple pipelined processor, we will discuss how to improve the efficiency of the pipeline.

Evaluate the performance of a simple pipeline

Before optimization, you need to evaluate the performance of the simple pipeline described above. Compared to the multiple cycles before implementing the pipeline, how much improvement in performance did you find after implementing the pipeline?

If you find that performance has actually declined, we recommend that you use performance counters to gain a deeper understanding of the reasons why.

The ideal situation for a pipeline is that one instruction can be executed per cycle. However, in general, the throughput of a pipeline cannot reach the ideal situation. The main reasons are as follows:

- Insufficient instruction supply capacity, unable to provide sufficient instructions to the pipeline
- Insufficient data supply capacity, load-store instruction execution is blocked
- Insufficient computing efficiency, due to the existence of the above three hazards, the pipeline needs to be blocked

Identify performance bottlenecks

To improve pipeline efficiency, we must first identify performance bottlenecks. Try adding more performance counters to the pipelined processor and analyze the current performance bottlenecks from various causes of blockages.

It is up to you to decide whether or not to use the following optimizations based on your design.

The following are some optimizations that may be useful, and whether or not to use them in your RTL design will depend on a number of factors, including your previous design, an analysis of your current performance counters, and the amount of area left available. But we do ask that you evaluate the expected performance benefit of the technique before proceeding with the RTL implementation. This is a very important training exercise for architectural design skills: You can choose not to implement an optimization technique, but you need to back up your decision with data from quantitative analysis methods.

However, "reducing the blockage of data hazards" is an important point for optimization, and we make it mandatory.

In short, if you've done a good job of area optimization, you now have more opportunities to optimize.

Enhancing instruction supply capacity

In the previous multi-cycle processor, assuming that each instruction executes for 5 cycles, as long as the instruction supply capacity reaches 0.2 instructions/cycle, the instruction consumption demand of multi-cycle processor can be satisfied. However, in pipelined processors, the instruction consumption requirement is ideally increased to 1 instruction/cycle, if the instruction supply capacity cannot reach the corresponding level, the advantages of pipelining cannot be utilized. However, for an icache, memory needs to be accessed in the event of cache miss, and the above mentioned instruction supply capacity must not be achieved, so we focus on the instruction supply capacity of the icache in the event of cache hit. Depending on the design of the icache, the decision of whether or not to optimize the icache here may vary.

Specifically, if your icache can determine whether a cache hit has occurred within 1 cycle and return an instruction to the IFU in the event of cache hit, then the icache's instruction supply capacity is already close to 1 instruction/cycle, which is basically sufficient for the pipelined processor's instruction consumption demands. In this case, you basically don't need to further increase the instruction supply capacity, but you may need to pay the price of low frequency, since the icache needs to complete more operations in a cycle, which will also lead to the pipeline can't work at a higher frequency.

If your icache also takes multiple cycles to return instructions to the IFU in the event of cache hit, the maximum instruction supply capacity of the icache is 0.5 instructions/cycle, or even less. Under this instruction supply capacity, the performance of the pipeline is significantly constrained. Assuming that it takes 3 cycles for icache to return an instruction to the IFU in the event of cache hit, the following instruction timing diagram is obtained. Since `cache` is pronounced the same as `cash`, some literatures also use `$` to refer to `cache`, The following diagram also uses `I$` to refer to `icache` for simplicity, replacing the previous `IF`, because the waiting time of `IF` is the same as the access time of icache.

```
        T1   T2   T3   T4   T5   T6   T7   T8   T9   T10  T11  T12
T13
     +--------------+----+----+----+----+
 I1  |      I$      | ID | EX | LS | WB |
     +--------------+----+----+----+----+
                    +--------------+----+----+----+----+
```

```
 I2                |      I$      | ID | EX | LS | WB |
                   +--------------+----+----+----+----+
                                  +--------------+----+----+----+--
--+
 I3                               |      I$      | ID | EX | LS |
WB |
                                  +--------------+----+----+----+--
--+
```

To improve the ability of the icache to supply instructions, it is also necessary to improve the throughput of the icache. This requirement is very similar to the "increase throughput of instruction execution" mentioned above, Naturally, we can also try to pipeline the icache accesses!

```
        T1   T2   T3   T4   T5   T6   T7   T8   T9
      +----+----+----+----+----+----+----+
 I1   | I$1| I$2| I$3| ID | EX | LS | WB |
      +----+----+----+----+----+----+----+
           +----+----+----+----+----+----+----+
 I2        | I$1| I$2| I$3| ID | EX | LS | WB |
           +----+----+----+----+----+----+----+
                +----+----+----+----+----+----+----+
 I3             | I$1| I$2| I$3| ID | EX | LS | WB |
                +----+----+----+----+----+----+----+
```

Estimating the performance benefit of pipelined icache

Try to roughly estimate the performance benefit of implementing pipelined icache based on performance counters.

The above figure further divides the icache access into 3 phases, and overlaps these 3 phases in time by pipeline technique, so that the throughput of icache on hit is close to 1 instruction/cycle. In order to implement pipelined icache, you can design the above phases according to the state transfer process when the icache hits. This is very similar to how we changed the processor to a pipelined processor, or even simpler than the processor pipeline, because icache works without happening control hazards. Of course, if the icache misses, it is still necessary to block the icache pipeline, and control memory accesses and icache updates through a state machine. Since a miss causes the icache to be updated, this can

lead to problems similar to data hazard, so you need to think about how to handle the situation correctly, and how to do this depends on your implementation.

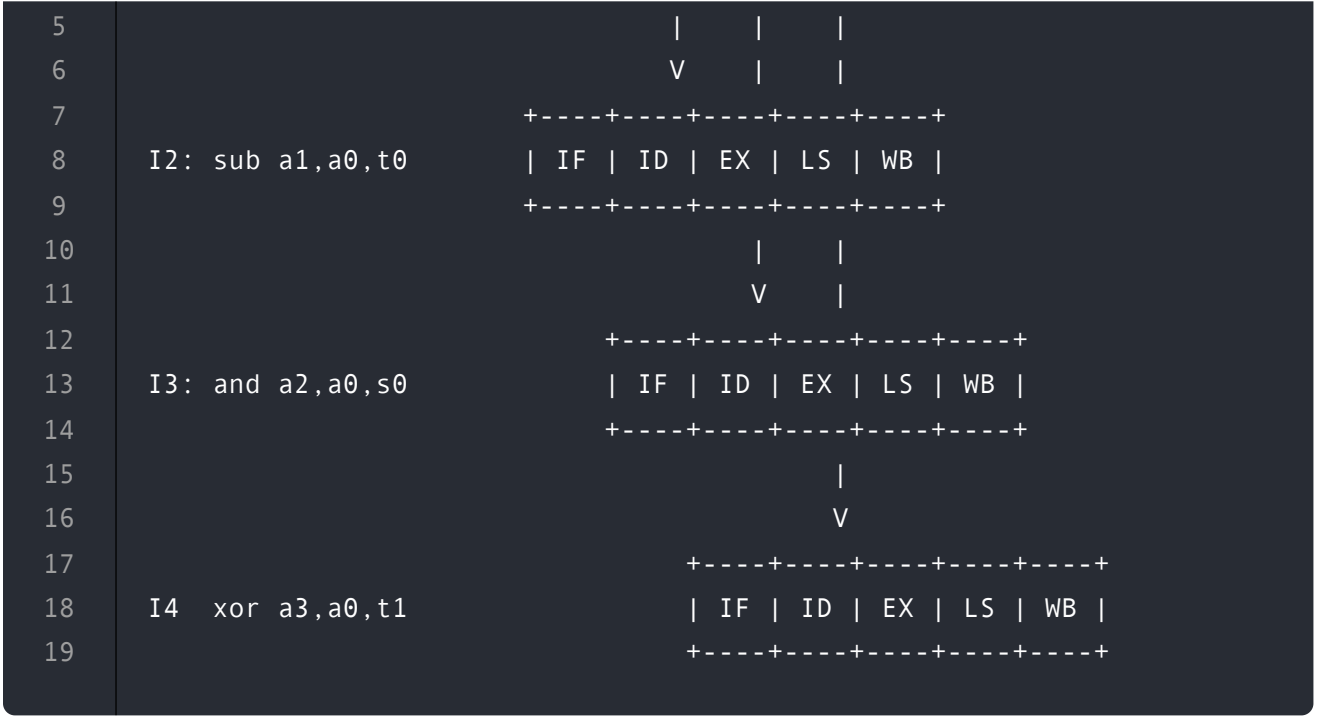
> **Implementing pipelined icache**
>
> If your icache takes multiple cycles to return instructions to the IFU on cache hit, try to borrow the idea of an instruction pipeline and pipeline the accesses to the icache to improve its instruction supply capability. After implementation, try to evaluate whether the increase in instruction supply capability is as expected by using performance counters and benchmarks, respectively.
>
> The division of icache accesses into three phases above is just an example, you should decide how to divide the phases according to your specific design. If you've used something like `PipelineConnect()` to implement a pipeline before, you'll find that pipelining the icache is very easy to implement: You're close to 90% of the way there by just defining the information that needs to be passed between stages.
>
> Once implemented, try comparing it to the previously estimated performance gains to check that your implementation meets expectations.

Reduce data hazard blockings

In the simple pipeline described above, we handle RAW data hazards by blocking and waiting. This approach requires waiting for the dependent registers to be written with new values before the blocked instructions can continue to be executed. Obviously, such waiting will reduce the throughput of the pipeline.

One observation is that the new value of the dependent register is written in the WB stage, but in fact, this value has already been calculated in the EX stage and will be passed to the LS stage and WB stage through the pipeline. Therefore, we can consider taking the calculated new value from these stages in advance for use by subsequent instructions, so that they do not have to wait for the dependent register to complete the update, and can obtain the correct source operand to start execution. This technique is called “forwarding” or “bypass.”

```
                    T1   T2   T3   T4   T5   T6   T7   T8
                  +----+----+----+----+----+
I1: add a0,t0,s0  | IF | ID | EX | LS | WB |
                  +----+----+----+----+----+
```



Estimating the ideal performance improvement of forwarding technology

As can be seen, forwarding technology can effectively eliminate RAW blocking other than load-use hazards. As for load-use hazards, because load instructions need to wait for data to be read before forwarding, the pipeline still needs to be blocked before that.

Try to estimate the ideal performance improvement that forwarding technology can bring based on performance counters.

Let's first discuss how forwarding technology modifies the data path. In the above pipeline, there are three forwarding sources: the EX stage, the LS stage, and the WB stage. All of them may carry data that can be forwarded. However, forwarding cannot be performed unconditionally. The forwarding source must meet the following conditions: The data is to be written to a register, the register number to be written to is consistent with the dependent register number, and the data is ready. The first two conditions are consistent with the RAW hazard detection conditions mentioned above, so the RAW hazard detection logic can be reused. The last condition is related to the behavior of the instruction. Most computational instructions can calculate the result in the EX stage, so the result obtained in the EX stage can be forwarded to the ID stage. However, load instructions can only calculate the load-store address in the EX stage, so it should not be forwarded in the EX stage. Furthermore, in the LS stage, the returned data can only be obtained after the R channel of the bus has been handshaken, so it should not be forwarded before that.

In addition to changes in the data path, we also need to consider modifications to the control path. Previously, in simple pipeline processors, once a RAW hazard was detected, the pipeline would be blocked. With the use of forwarding technology, the conditions for blocking the pipeline will change:

- If no RAW hazard is detected in the ID stage, there is no need to block the pipeline, which is consistent with the simple pipeline described above.
- If the ID stage detects a RAW hazard and there is a stage that meets the forwarding conditions, there is no need to block the pipeline, and the forwarded data is treated as the output of the ID stage.
- If the ID stage detects a RAW hazard but there is no stage that meets the forwarding conditions, the pipeline needs to be blocked.

In particular, if there are multiple instructions that satisfy the forwarding conditions at the same time, careful consideration is required. Consider the following instruction sequence:

```
                      T1   T2   T3   T4   T5   T6   T7   T8
                    +----+----+----+----+----+
I1: add a0, a0, a1  | IF | ID | EX | LS | WB |
                    +----+----+----+----+----+
                         +----+----+----+----+----+
I2: add a0, a0, a1       | IF | ID | EX | LS | WB |
                         +----+----+----+----+----+
                              +----+----+----+----+----+
I3: add a0, a0, a1            | IF | ID | EX | LS | WB |
                              +----+----+----+----+----+
                                   +----+----+----+----+----+
I4: add a0, a0, a1                 | IF | ID | EX | LS | WB |
                                   +----+----+----+----+----+
```

Assuming that I1 does not depend on instructions older than itself, there is no need to block the execution of I1. As for I2, it depends on the result of I1, but through forwarding technology, the result of I1 in the EX stage can be forwarded to I2 in the ID stage at time T3, so there is no need to block the execution of I2. For I3, at time T4, it is found that I2 in the EX stage and I1 in the LS stage both satisfy the forwarding conditions, but from the perspective of the ISA state machine, the instructions are executed serially, so the `a0` read by I3 should be the result of the most recent write to `a0`, so I2 in the EX stage should be selected for forwarding; Similarly, for I4, at time T5, I3 in the EX stage, I2 in the LS stage, and I1 in the WB stage all satisfy the forwarding conditions, but from the perspective of the ISA state machine, I3 in the EX stage should be selected for forwarding. That is, when multiple

instructions satisfy the forwarding conditions at the same time, the youngest instruction should be selected for forwarding.

Implementing forwarding technology

Based on the above, implement forwarding technology in the pipeline to eliminate most RAW hazards. After implementation, compare the results with the previously estimated performance improvement to check whether your implementation meets expectations.

Forwarding solutions in textbooks

The above forwarding solution forwards the calculation results from other stages to the ID stage, then selects the correct operand and sends it to the pipeline segment register. Most forwarding solutions in textbooks forward to the EX and LS stages, then select the correct operand before calculation.

Try comparing these two schemes. If you are unsure, you can implement both schemes separately, then compare them in terms of IPC, frequency, area, and other aspects.

Reduce control hazard blockings

In the simple pipeline described above, we use speculative execution to deal with control hazards. Specifically, we speculate that “the next static instruction will always be executed.” If the speculation is correct, the blocking caused by control hazards can be eliminated. However, in reality, the above speculation is not always correct. In this case, the pipeline needs to be flushed, wasting several cycles. One way to improve the execution efficiency of the pipeline is to reduce the negative impact of flushing the pipeline.

Estimate the performance gains associated with eliminating control hazards.

Attempt to roughly estimate the performance gains associated with completely eliminating control hazards based on performance counters, thereby optimizing the ideal performance gains of the technology.

To reduce the negative impact of flushing the pipeline on processor execution efficiency, the following two approaches can be considered:

1. Reduce the cost of a single pipeline flush. This requires calculating the results of branch instructions as quickly as possible. Some textbooks describe a solution that calculates branch results during the ID phase, which can clearly improve pipeline IPC, but this method needs to be evaluated comprehensively from the perspectives of frequency and area.
2. Reduce the number of pipeline flushes. This requires improving the accuracy of speculation. According to the above analysis, the speculation accuracy of “no exceptions will happen” is already close to 100%, so the main consideration is the accuracy of branch instruction and jump instruction speculation.

Branch prediction technology is commonly used to improve the accuracy of branch instruction speculation. The module that performs branch prediction is called a branch predictor. The execution results of branch instructions are either “taken” or “not taken.” Therefore, branch prediction technology only needs to predict one of the two choices. The specific method of giving a prediction result is called a branch prediction algorithm. Considering whether to refer to runtime information, branch prediction algorithms are divided into static prediction and dynamic prediction. Here, we will first introduce the static prediction algorithm, and we will introduce the dynamic prediction algorithm in A Stage.

Understand the importance of branch prediction in modern processors.

Try to calculate the proportion of branch instructions in the dynamic instruction count. That is, assuming that on average every x instructions contain one branch instruction, find x.

Assume that in an ideal five-stage pipeline processor, the instruction supply capacity is 1 instruction per cycle, there are no structural hazards or data hazards, the load-store latency is 0 cycles, and jump instructions can always be predicted correctly, but branch instructions may be predicted incorrectly, and branch instructions calculate the jump result in the EX stage. Based on the x obtained above, calculate the IPC of the processor under different branch prediction accuracies: 100%, 99.5%, 99%, 95%, 90%, and 80%.

> Modify the above processor to a 15-stage out-of-order single-issue pipeline, in which the branch instruction calculates the jump result at stage 13. Keep the other assumptions unchanged and recalculate the IPC of the processor under different branch prediction accuracy rates.
>
> Continue to improve the processor to a 15-stage out-of-order four-issue pipeline. Keep the other assumptions unchanged and recalculate the IPC of the processor under different branch prediction accuracy rates.

Static prediction is based solely on the branch instruction itself. Since the branch instruction itself does not change during program execution, the prediction result is always the same for a given branch instruction and a given static prediction algorithm. The above “always speculating that the next static instruction will be executed” is, from the perspective of branch prediction technology, “always not jumping,” which is a static prediction algorithm. Another static prediction algorithm is “always jump.”

In fact, depending on the direction of the branch jump, there is a bias in whether or not the branch instruction jumps. This is actually related to the behavior of loops in the program. For example, when the branch jump target is located ahead (young instruction), it may be a loop exit, so there is a bias not to jump; when the branch jump target is located behind (old instruction), it may be a re-execution of the loop body, so there is a bias to jump. A static prediction algorithm that utilizes this characteristic is called BTFN (Backward Taken, Forward Not-taken): if the branch jump target is located behind, then predict a jump; otherwise, predict no jump. When implementing this, you only need to obtain the prediction result based on the sign bit of the B-type instruction offset. In fact, the RISC-V manual also recommends that compilers generate code according to the BTFN pattern:

```
Software should also assume that backward branches will be predicted
taken and
forward branches as not taken, at least the first time they are
encountered.
```

An important indicator for evaluating a branch prediction algorithm is prediction accuracy. Similar to the previous icache, for a given program, the branch instructions that need to be executed and the execution results of each branch instruction are fixed. As long as we obtain the itrace of the program running, we can quickly calculate the accuracy of a branch prediction algorithm without the need for RTL-level simulation. Furthermore, itrace already contains the complete instruction stream, and the trace of instructions other than branches

has no effect on the execution result of branch instructions. Therefore, what we really need is only the trace of branch instructions, which we call btrace (branch trace).

Based on the above analysis, we only need to implement a branch prediction simulator, which we call branchsim. branchsim receives btrace and predicts whether each branch instruction will jump based on the branch prediction algorithm. It then compares the results with the execution results recorded in btrace and calculates the prediction accuracy of the algorithm. As for btrace, we can quickly generate it using NEMU.

Implement branchsim

Based on the above introduction, implement a simple branch prediction simulator branchsim, then evaluate the accuracy of the above static prediction algorithms, and evaluate the performance gains brought by the prediction algorithm based on the prediction accuracy.

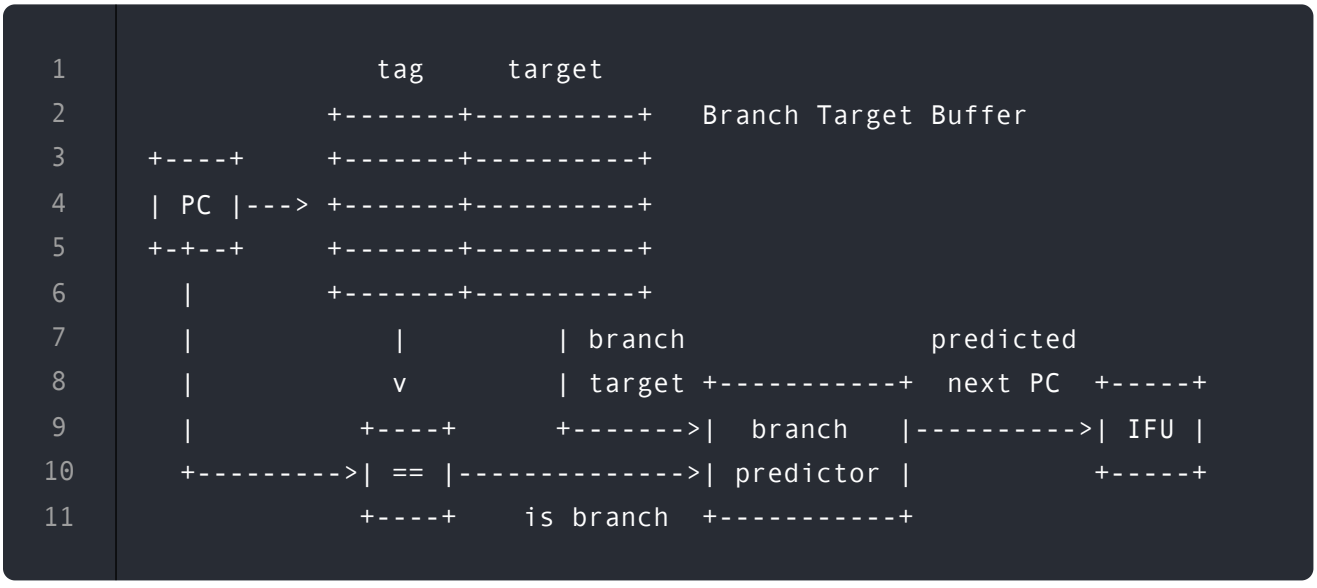
If you are interested in dynamic prediction algorithms, you can first study the relevant materials, then implement the corresponding algorithms in branchsim and evaluate their accuracy.

Similar to cachesim, branchsim can also be used as a REF for branch prediction performance. For example, the simple pipeline above uses a static prediction algorithm that “always jumps,” and the relevant performance counters should be completely consistent with the statistics provided by branchsim.

After selecting a branch prediction algorithm with good performance through branchsim, we can consider how to implement a branch predictor in the processor. Generally speaking, the prediction results of the branch predictor need to be provided to the IFU for use: if a jump is predicted, the IFU fetches the instruction from the jump target of the branch instruction; otherwise, the IFU fetches the instruction from `PC + 4`. But in fact, we can only know whether an instruction is a branch instruction at the ID stage, and if it is a branch instruction, we also need to know its jump target at the ID stage. In the IF stage, we only have the PC value, and it is difficult to obtain the above information for branch prediction.

The solution to the above problem is to maintain a table that maps PC values to branch target addresses. This table is called the Branch Target Buffer (BTB). The BTB can be regarded as a special cache indexed by the PC value. If there is a hit, it means that the PC corresponds to a branch instruction, and the jump target can be read from it. If there is a miss, it means that the instruction corresponding to the PC is not a branch instruction, and

at this point, the IFU can fetch the instruction from PC + 4. In addition, the BTB needs to be filled and updated during processor execution. In principle, the BTB can be updated as early as when the branch instruction is decoded in the ID stage. Usually, the number of items in the BTB is limited. If no free table items are found during the update, according to the principle of locality, the old table items should be overwritten at this time. In particular, when the processor is reset, all entries in the BTB are invalid, and the correct jump target cannot be read at this time. However, since branch prediction is a speculative execution technique, prediction errors do not affect the correctness of the processor's program execution. Once the correct entries are written to the BTB, effective prediction can be performed.



Implementing a branch predictor

Since information from different branches may be replaced in a BTB with a limited number of entries, when performing branch prediction, the jump target of the branch instruction corresponding to the current PC is not always available. This situation will affect the accuracy of branch prediction, so you need to add a BTB to branchsim to adjust its prediction accuracy.

Specifically, first implement a simple BTB in RTL, with no restrictions on its organization. You can choose to use direct mapping, fully associative, or set-associative based on your needs. You can also add new fields to the BTB according to your requirements. After implementation, evaluate its area and select an appropriate number of BTB entries based on the remaining area. then adjust branchsim according to this number of items, and re-evaluate the accuracy of the branch prediction algorithm.

After re-evaluation, consider all factors to decide how to implement the branch predictor. After implementation, compare the branch prediction accuracy of RTL with

the branch prediction accuracy statistics in branchsim.

The above describes branch instruction prediction. Jump instructions also require speculative execution. Jump instructions are unconditional, but there are multiple jump results, so jump instruction prediction mainly involves predicting the jump target. Jump instructions are divided into direct jumps `jal` and indirect jumps `jalr`. For a given `jal` instruction, the jump target is determined. Therefore, as long as the jump target of the `jal` instruction is recorded in the BTB, and as long as the corresponding table entry is not replaced, the next time the `jal` instruction is encountered, the instruction at the correct jump target can be obtained with 100% accuracy. As for the `jalr` instruction, its jump target is determined by the value of the source register when the instruction is executed. It is difficult to predict successfully with only a static prediction algorithm, so a corresponding dynamic prediction algorithm needs to be considered. For simplicity, a prediction algorithm that "always does not jump" can be used for the `jalr` instruction at present.

Estimating the correct performance gain of speculative execution of jump instructions

Try to estimate the ideal performance gain when the `jal` and `jalr` instructions are executed correctly based on the performance counter.

If you want to predict the jump target of the `jal` instruction, you can let the `jal` instruction share the same BTB with the branch instruction, or you can let the `jal` instruction use another BTB separately. The former can save space, but the table entries between instructions may overwrite each other, thereby affecting the prediction accuracy; the latter is the opposite. You can decide how to design based on the evaluation of branchsim.

Implementing jump target prediction for jal instructions

Based on the above plan, implement jump target prediction for `jal` instructions.

Predicting the jump target of the ret instruction

Generally speaking, it is difficult to correctly predict the jump target of the `jalr` instruction. However, as a special type of `jalr` instruction, the `ret` instruction is

relatively easy to predict correctly. If you are interested and have sufficient remaining space, you can refer to the relevant information on the “return address stack” and implement jump target prediction for the `ret` instruction in the processor.

Optimize the pipeline

Based on the analyzed performance bottlenecks, and subject to area constraints, attempt to invest limited area resources in the most worthwhile optimization technologies to maximize processor performance.

Re-evaluating processor performance optimization

We can anticipate that you may not be particularly pleased when completing this section: Either the available area is extremely limited, making it difficult to add optimization techniques; Or adding optimization techniques results in a decrease in processor frequency, offsetting the IPC improvements provided by the technique; Or, after investing a lot of effort, you finally achieve good performance in terms of area and frequency, but find that the improvement in IPC is very small...

More importantly, you should feel a sense of helplessness in your heart: As the peak of technology in textbooks on architecture principles, is this the performance of pipelines? In fact, this illusion of technological peak comes from the significant weakening of instruction supply and data supply in textbooks, which misleads you into thinking that computing efficiency is everything in processor design, and makes you think that out-of-order multiple issue is the ultimate pursuit of processor design.

In fact, your experience reflects the well-known **memory wall** in the field of computer systems: The performance of memory severely limits the performance of the CPU. Theoretically, upgrading a multi-cycle processor to a pipeline processor should result in a performance improvement of nearly five times. However, in a system that includes modern memory, the final performance is closely related to the performance of the memory. Amdahl's law actually predicts your helplessness: If the load-store cannot keep up, no matter how fast the CPU's computing power is, it will be futile.

Therefore, you need to wake up from the illusion that “learning the pipeline will give you the ability to design architecture.” You need to recognize the significance of

performance counters, Amdahl's law, simulators, etc. for architecture design. You need to learn to make reasonable trade-offs between area, clock speed, and IPC, and even find a solution that performs well in all aspects. This is the capability a qualified architect must possess.

However, these skills cannot be acquired simply by translating the architectural diagrams in books into RTL code. If you merely refer to the code in some books, forget about it. On the contrary, the reason we have organized the lecture content in the current order is to help you awaken from the illusion mentioned above and confront the reality of the memory wall in SoCs, and then learn to scientifically explore reasonable processor optimization solutions step by step.

From the results, the performance of the processor you designed may still be weak, but if you have completed the training we set, you have already developed real architectural design capabilities: you have learned to analyze performance bottlenecks, think of new solutions, estimate performance gains, implement these solutions under various constraints, and evaluate whether your implementation meets expectations... These are much more important than just being able to write a pipeline processor using RTL.

Therefore, you can also test whether you have architectural design capabilities using the following methods: If you don't know what to do without referring to reference books, or if you need to ask others to determine the merits of a plan, then you do not have architectural design capabilities.

Handling `fence.i`

Finally, we need additional processing of the execution of the `fence.i` instruction in the pipeline. Recall that the semantics of `fence.i` are meant to allow subsequent fetch operations to see the data modified by the previous store instruction. We have implemented icache with a special treatment to ensure that subsequent fetch operations do not fetch stale instructions through the icache.

In a pipelined processor, stale instructions may exist in the pipeline. Consider the following example: Suppose `fence.i` is in effect during the EX stage, but as instructions younger than `fence.i`, I3 and I4 have already been taken out and are in the pipeline, they may be stale, and continuing to execute them may result in a transfer of the CPU state machine that does not match that of the ISA state machine, thus causing an error.

```
                  T1   T2   T3   T4   T5   T6   T7
               +----+----+----+----+----+
I1: add        | IF | ID | EX | LS | WB |
               +----+----+----+----+----+
                    +----+----+----+----+----+
I2: fence.i         | IF | ID | EX | LS | WB |
                    +----+----+----+----+----+
                         +----+----+
I3: ??? may be stale     | IF | ID |
                         +----+----+
                              +----+
I4: ??? may be stale          | IF |
                              +----+
                                   +----+----+----+----+----+
I5: sub                            | IF | ID | EX | LS | WB |
                                   +----+----+----+----+----+
```

Design a counterexample

Based on the above analysis, design a `fence.i` related test case, so that the test case runs correctly in the previous multi-cycle processor, but runs incorrectly in the pipelined processor.

The solution to the above problem is simple: since the above instruction should not be executed, just flush it. The implementation can reuse the logic of flushing in case of speculative execution errors.

Correctly implementence.i in the pipeline

Flush the pipeline while executing `fence.i`, then re-run the above test case. If your implementation is correct, the test case will run successfully.