B1 Bus

You've designed the single-cycle processor NPC, and you understand how devices work. But we've let the simulation environment provide the functionality of the device, Now it's time to implement NPC - devices communication in hardware.

In a computer, each module does not work independently; data exchange is required between different modules: between the CPU and the memory controller, between the memory controller and the memory chips, between the instruction fetch unit and the decode unit, etc. Communication between these modules must be conducted through a set of agreed-upon protocols. The same is true for software. In the DiffTest tool that we use, NEMU needs to communicate with Spike, and NPC also needs to communicate with NEMU to implement the corresponding functions.

In a broad sense, a bus is a communication system used to transfer data between different modules. We will now focus on the hardware and introduce the bus in a narrower sense, i.e. the communication protocols between hardware modules.

Bus - Communication Protocols Between Hardware Modules

We will start with communication between modules inside the processor to understand the general organization of bus protocols.

The Simplest Bus

In your NPC, there is an Instruction Fetch Unit (IFU) and an Instruction Decode Unit (IDU), and these two units need to communicate to transfer instructions from the IFU to the IDU.

```
+-----+           +-----+
| IFU | inst ---> | IDU |
+-----+           +-----+
```

In this simple scenario, there is an implied bus protocol: the module that initiates communication is generally called the master, and the module that responds to communication is called the slave. In the simple interaction scenario above, the IFU, as the master, sends information to the IDU, which is the slave, with the information being the current instruction. As a single-cycle processor, obviously, their communication actually implies the following agreement:

- Every cycle, the master sends valid information to the slave.
- Once the master sends valid information, the slave can receive it immediately.

Asynchronous Bus

However, if the IFU cannot guarantee to fetch an instruction every cycle, then the IDU needs to wait for the IFU to complete the instruction fetch. In this case, it is necessary to add a `valid` signal to the communication content to indicate when the IFU sends a valid instruction to the IDU. The communication protocol also needs to be updated as follows:

- Information is considered valid only when the `valid` signal is valid.
- Once the master sends valid information, it is considered received by the slave immediately.

```
+-----+ inst  ---> +-----+
| IFU | valid ---> | IDU |
+-----+            +-----+
```

So, how do we prevent the IDU from executing invalid instructions? Recalling the state machine model of the processor, we just need to keep the processor's state unchanged when the instruction is invalid. At the circuit level, the state is the sequential logic components, therefore, we just need to set the write enable of the sequential logic components to invalid when the instruction is invalid.

Further, suppose some instructions are too complex to decode, and the IDU needs multiple cycles to decode a single instruction. Under this assumption, when the IFU successfully fetches an instruction, the IDU may not have finished decoding the previous instruction, At this time, the IFU should wait for the IDU to complete the current decoding work before sending the next instruction to the IDU. To implement the function of making the IFU wait, it is necessary to add a `ready` signal to the communication content to indicate when the IDU can receive the next instruction. The communication protocol also needs to be updated as follows:

- Information is considered valid only when the `valid` signal is set.
- The information sent by the master is considered received by the slave only when the `ready` signal is set.

```
+-----+ inst  ---> +-----+
| IFU | valid ---> | IDU |
+-----+ <--- ready +-----+
```

This actually refers to an asynchronous bus, where it is unpredictable when communication between the two modules will occur; it only happens when both `valid` and `ready` signals are set. Both signals being set, also known as a "handshake," indicates that the master and slave have reached a consensus on the successful transmission of information. Clearly, this communication protocol is more flexible, allowing the master and slave to decide when to send or receive messages based on their own work situations. When the `valid` signal is set and the `ready` signal is not set, the IFU needs to wait for the IDU to be ready. At this time, the IFU should temporarily store the information to be sent to prevent loss of information.

The RTL (Register Transfer Level) implementation of an asynchronous bus

The asynchronous bus determines when to communicate through a handshake protocol. At the RTL level, it mainly consists of two parts: interface signals and communication logic.

Interface signals are the signals that need to be transmitted between modules, such as the `inst`, `valid`, and `ready` signals in the example above. Chisel provides a Decoupled template, which comes with `valid` and `ready`, and asynchronous bus interfaces can be easily implemented through meta-programming:

```
class Message extends Bundle {
  val inst = Output(UInt(32.W))
}

class IFU extends Module {
  val io = IO(new Bundle { val out = Decoupled(new Message) })
  // ...
}
class IDU extends Module {
  val io = IO(new Bundle { val in = Flipped(Decoupled(new Message))
})
```

```
  // ...
}
```

When more information needs to be transmitted, it is also very convenient to add signals. This is the benefit brought by abstraction!

```
 class Message extends Bundle {
   val inst = Output(UInt(32.W))
+  val pc = Output(UInt(32.W))
 }
```

Next is the communication logic, which is how to implement the bus protocol with circuits. The protocol is an agreement between the master and the slave, and what really needs to be implemented in the circuit is the behavior of both modules while adhering to this agreement. That is, both modules enter different states based on the different conditions of the handshake signals, thereby taking different actions. This is the state machine! For the master, we can easily draw its state transition diagram:

```
   +-+ valid = 0
   | v          valid = 1
1. idle ----------------> 2. wait_ready <-+
   ^                           |       |    | ready = 0
   +--------------------------+      +-----+
             ready = 1
```

Specifically:

1. Initially in the `idle` state, unset `valid` signal
   1. If there is no need to send a message, remain in the `idle` state
   2. If a message needs to be sent, transition to the `wait_ready` state and wait for the slave to be ready
2. In the `wait_ready` state, detect the slave's `ready` signal
   1. If the `ready` signal is set, the handshake is successful, return to the idle state
   2. If the `ready` signal is unset, continue to stay in the `wait_ready` state and wait

With the state transition diagram, we can easily write the corresponding RTL. Here is a simple Chisel code snippet for reference:

```scala
class IFU extends Module {
  val io = IO(new Bundle { val out = Decoupled(new Message) })

  val s_idle :: s_wait_ready :: Nil = Enum(2)
  val state = RegInit(s_idle)
  state := MuxLookup(state, s_idle)(List(
    s_idle       -> Mux(io.out.valid, s_wait_ready, s_idle),
    s_wait_ready -> Mux(io.out.ready, s_idle, s_wait_ready)
  ))

  io.out.valid := instruction need to be send
  // ...
}
```

Implementing IDU Communication in RTL

After understanding the implementation process of IFU communication, try to analyze and draw the state transition diagram of IDU, and write the RTL code for IDU communication. This is just an exercise about bus, you don't need to modify the IDU code in your NPC at the moment.

When we fully implement the communication logic of both IFU and IDU, the above bus protocol will also be effectively implemented.

Processor Design from a Bus Perspective

We can reconsider the processor design itself from a bus perspective. Assuming the processor consists of 4 modules that need to communicate with each other: IFU sends instructions to IDU, IDU sends the decoded results to EXU, EXU sends the computation results to WBU.

```
+-----+ inst  ---> +-----+  ...  ---> +-----+  ...  ---> +-----+
| IFU | valid ---> | IDU | valid ---> | EXU | valid ---> | WBU |
+-----+ <--- ready +-----+ <--- ready +-----+ <--- ready +-----+
```

From this perspective, different microarchitectures of processors essentially represent varying communication protocols between modules:

- For a single-cycle processor, it is bascially every message sent by the upstream being valid each cycle, while the downstream remains ready to receive new messages.
- For a multi-cycle processor, messages are invalid when the upstream module is idle, and new messages are not accepted when the downstream module is busy. The IFU fetches the next instruction only after receiving the completion signal from WBU.
    - Unlike traditional textbooks, this is a message-controlled distributed multi-cycle processor, where the "distributed" aspect lies in: whether two modules can communicate with each other solely depends on their states, independent of other modules.

> **Multi-cycle Processor in Textbooks**
>
> In a multi-cycle processor, an instruction is divided into different stages and executed in different cycles, hence each instruction takes multiple clock cycles to complete:
>
> - In the 1st cycle, IFU fetches the instruction and passes it to IDU
> - In the 2nd cycle, IDU sends the decoded result to EXU
> - In the 3rd cycle, EXU passes the computation result to WBU
> - In the 4th cycle, WBU writes back the result to the register file
> - In the next cycle, IFU fetches the next instruction...

- For a pipelined processor, IFU can continuously fetch instructions, and each module attempts to send messages downstream every cycle as soon as it completes processing any message.
- For an out-of-order execution processor, it can be seen as a minor extension of the pipeline: Each downstream module has a queue, and the upstream module only needs to send messages to the queue without worrying about the state of the downstream queue.

Usually, centralized control requires a global controller responsible for collecting the states of all modules and then using these states to control the next steps of each module. The multi-cycle processor typically introduced in traditional textbooks is often a centralized multi-cycle processor based on a large state machine. It collects the states of each module through a global state machine to control communication between each module and determine what the next state should be. For example, if instruction fetching is completed in the current cycle, then in the next cycle, it should enter the decoding state and control IDU to perform decoding.

```
                   +--------------+
    +------------> |  Controller  | <--------------+
    |              +--------------+               |
    |               ^            ^                |
    v               v            v                v
 +-----+  inst   +-----+   ...   +-----+   ...   +-----+
 | IFU | ------> | IDU | ------> | EXU | ------> | WBU |
 +-----+         +-----+         +-----+         +-----+
```

In textbooks, usually only a few instructions are used to introduce the basic principles of multi-cycle state machines, which is not a big deal. However, the scalability of centralized control is relatively low. As the number of modules and complexity increase, the design of the controller becomes more complex:

- With an increase in the number of instructions, the types of instructions also increase. The centralized state machine needs to consider all stages of execution for each type of instruction.
- Inserting a new stage in this processor would require a redesign of the controller.
- In real processors, the working time of each module may vary:
  - Instruction fetching in IFU may have delays.
  - IDU may complete decoding in just one cycle.
  - In EXU, different instructions may have varying execution times:
    - Integer arithmetic instructions in RVI typically complete calculations in one cycle.
    - Division usually takes a long time.
    - Multiplication might be faster but could still take several cycles.
    - Memory access instructions have unpredictable wait times.
- The execution of some instructions may trigger exceptions, and interrupts can arrive at any time.

In such complex scenarios, considering the combination of different states for each module, making unified decisions becomes very challenging.

In the distributed control mentioned earlier, where each module's behavior depends only on its own state and the state of downstream modules, each module can work independently. For example, in an out-of-order execution processor, the upstream module can continue working until the downstream queue is full. In distributed control, it is very easy to insert a new module; you only need to modify the implementation of the interface of its upstream and downstream modules. Therefore, distributed control offers better scalability.

By adopting this handshake-based distributed control, the design of different microarchitectures of processors can be unified. Furthermore, out-of-order execution processors are inherently distributed control systems because they have many modules and states for each module, and various events can occur at any time (such as interrupts or pipeline stalls), if centralized control were used, it would be nearly impossible to ensure that the controller makes correct decisions for each module when different events occur.

> ### Benefits of Buses in System Design
>
> You should be able to appreciate one of the benefits of buses in system design: By dividing modules and enabling communication between them, the overall complexity of system design and maintenance is reduced. A centralized controller needs to communicate with every module, making its design and maintenance the most challenging; By decomposing global communication into interactions between upstream and downstream modules through bus protocols, the need for a centralized controller is eliminated, thereby reducing the complexity of the entire processor design.
>
> In fact, many examples in the field of computing use message passing to reduce system complexity, such as microkernels in operating systems, MPI programming frameworks in distributed systems, client-server models in software architecture, and even the entire internet communicates through network packets.
>
> While processor design deals with hardware, design patterns are not solely a hardware issue, and we can still draw useful experiences from the software domain to help improve processor design.

Using the function abstraction and meta-programming features in Chisel, we can unify the design patterns of processors with different microarchitecture and easily "upgrade" the microarchitecture of processors.

```
class NPC extends Module {
  val io = // ...

  val ifu = Module(new IFU)
  val idu = Module(new IDU)
  val exu = Module(new EXU)
  val wbu = Module(new WBU)


  StageConnect(ifu.io.out, idu.io.in)
  StageConnect(idu.io.out, exu.io.in)
  StageConnect(exu.io.out, wbu.io.in)
  // ...
}

object StageConnect {
  def apply[T <: Data](left: DecoupledIO[T], right: DecoupledIO[T])
= {
    val arch = "single"
    // To illustrate the concept of abstraction, some details have
been omitted in this code.
    if      (arch == "single")   { right.bits := left.bits }
    else if (arch == "multi")    { right <> left }
    else if (arch == "pipeline") { right <> RegEnable(left,
left.fire) }
    else if (arch == "ooo")      { right <> Queue(left, 16) }
  }
}
```

NPC Refactoring

Attempt to refactor the NPC using the bus concept mentioned above. Although this is not mandatory, if you are using Chisel for development, we strongly recommend refactoring your code. This will also prepare you for future integration with SoC and pipeline implementation.

If you are developing in Verilog, you might find the refactoring work somewhat tedious. However, we want to emphasize that design patterns and RTL implementation are different layers. Despite potential challenges, we encourage you to consider how to achieve the correct design patterns through Verilog.

System bus

Apart from the interconnection of various modules within the processor as mentioned earlier, how the processor connects to memory and devices is also crucial, as real processors cannot operate without being connected to memory and peripherals. The bus that connects the processor to memory and devices is typically referred to as the system bus. Below, we will use the connection between the processor and memory as an example to explain how the system bus should be designed.

Accessing Read-Only Memory

Due to the fundamental need to read data from memory, let's first consider how a processor completes a read operation through the system bus. Assuming a processor specification of Nx32, where memory contains N words, each word being 32 bits. Additionally, assuming that the memory's read latency is fixed at 1 cycle, which essentially represents synchronous memory, where the delay between receiving a read request and returning data is constant, as seen in SRAM's access characteristics. In reality, the `pmem_read()` provided by the NPC simulation environment has no read latency; it can return read data in the current cycle upon receiving a read request. However, this is only used to facilitate the implementation of a single-cycle processor and does not reflect an actual memory device.

If we do not consider write operations, a read-only memory (ROM) is enough. To read data from ROM, the corresponding bus only requires two signals: address and data. The bit widths of these signals are log2(N) and 32, respectively.

```
+-----+ raddr[log2(N)-1:0] ---> +-----+
| CPU | <---        rdata[31:0] | MEM |
+-----+                         +-----+
```

The communication protocol is as follows:

- The master (CPU) sends the read address `raddr` to the slave (MEM).
- In the next cycle, the slave responds to the master with the data `rdata`.
- The above actions occur every cycle.

Evaluate the Clock Frequency and Program Performance of Single-Cycle NPC

Before further modifying the NPC, please try to evaluate the current NPC's clock frequency using the `yosys-sta` project you used in the pre-learning phase. However, before evaluation, you need to perform the following tasks:

1. Run the microbench test with the `train` scale first and record the number of cycles required for its completion.
2. Comment out the code that calls `pmem_read()` and `pmem_write()` through DPI-C in RTL, then instantiate a memory device for instruction fetching and another for

data access. To maintain the single-cycle nature, the instantiated memory needs to return read data in the current cycle, so we can implement it through flip-flops similar to the register file. If you are using Verilog, you can directly instantiate a `RegisterFile` module, ensuring correct port connections. For consistent testing results, we agree to instantiate memories of size 256x32b each, totaling 1KB. Instantiate two such memories for a total size of 2KB.

The reason for this modification is that a single-cycle NPC requires completing a full instruction lifecycle every cycle, making it impossible to connect to any real-world memory. Instead, two registerfile like memory devices are connected for frequency evaluation. After modification, you can evaluate the clock frequency of the single-cycle NPC.

Based on the evaluated clock frequency and the previously recorded number of cycles for microbench execution, you can estimate how long it will take for the NPC to run microbench in the future. Note that this is not simulation time but an estimated program execution time at the mentioned clock frequency. For example, in a certain version of NPC in `yosys-sta` project with nangate45 technology, the clock frequency is 51.491MHz, indicating microbench would take 3.870s to run but simulation took 19.148s.

However, this estimation is not accurate and can be considered very optimistic due to:

- The single-cycle NPC is far from tape out; for instance, when modifying memory device earlier, I/O-related parts were ignored.
- The mentioned clock frequency is post-synthesis; after layout and routing, net delays introduced will further reduce the clock frequency.
- The memory corresponding to instruction fetching was optimized out by yosys due to lack of write operations.
- The memory corresponding to data access cannot accommodate microbench as it requires 1MB of memory for successful execution at training scale. This size far exceeds the number of flip-flops that can be accommodated in actual processor chip design. Set EDA tool processing time aside, just filling up so many flip-flops on a chip would introduce significant area occupation affecting net delays significantly.

Hence, the reference value of this evaluation result is minimal; consider it as practice for subsequent evaluations.

Transform the Memory Accessed by IFU into SRAM

1. Write an SRAM module in RTL for the Instruction Fetch Unit (IFU) with a 32-bit address width and a 32-bit data width.
2. Upon receiving a read request, the SRAM module will use DPI-C to call `pmem_read()` to read the data, then delay returning the read data by 1 cycle to simulate a more realistic memory operation.

However, since this SRAM requires 1 cycle to retrieve the read data, the NPC is no longer strictly a single-cycle processor but rather a simple multi-cycle processor:

1. In the first cycle, the IFU issues an instruction fetch request.
2. In the second cycle, the IFU receives the instruction and passes it to subsequent modules for decoding and execution.

If you have refactored the NPC as suggested earlier, you will find that transforming the NPC into a multi-cycle processor is not difficult to achieve.

Adapt DiffTest for Multi-Cycle Processor

After modifying the NPC into a multi-cycle processor, the NPC no longer executes an instruction every cycle. To ensure the DiffTest functions correctly, you need to adjust the timing of checks. You may need to read some RTL states from the simulation environment to help determine when to perform DiffTest checks.

Accessing Read-Write Memory

To support write operations, new signals need to be added to the bus:

- Firstly, it is necessary to include the write address `waddr` and write data `wdata`.
- Since write operations do not occur every cycle, an additional signal called write enable `wen` is required.
  - While read operations also do not occur every cycle (for example, in a multi-cycle processor like the modified NPC where the second cycle does not require instruction fetching), read enable `ren` is theoretically not essential as read operations do not alter circuit states.

- However, in practice, a read enable `ren` signal is typically included. If there are no read requests, the memory does not need to perform read operations, thereby saving power.
- Write operations may only write into certain bytes within a word (for example, the `sb` instruction writes only 1 byte), hence the need for a write mask signal `wmask` to specify which bytes in the data should be written.

```
+-----+ raddr[log2(N)-1:0] ---> +-----+
|     | ren                ---> |     |
|     | <---        rdata[31:0] |     |
|     | waddr[log2(N)-1:0] ---> |     |
| CPU | wdata[31:0]        ---> | MEM |
|     | wen                ---> |     |
|     | wmask[3:0]         ---> |     |
+-----+                         +-----+
```

Additionally, the communication protocol needs to define write operation behavior. We represent this using pseudocode:

```c
if (wen) {
  // "wmask_full" represents the result of expanding "wmask" by bits.
  M[waddr] = (wdata & wmask_full) | M[waddr] & ~wmask_full;
}
```

Transform the Memory Accessed by LSU (Load-Store Unit) into SRAM

1. Write an SRAM module in RTL for the LSU with a 32-bit address width and a 32-bit data width.
2. Upon receiving read and write requests, use DPI-C to call `pmem_read()` / `pmem_write()` and delay returning the read data by 1 cycle.
   - The device access functionality within `pmem_read()` / `pmem_write()` can be retained at this stage. We will discuss how to access peripherals through the bus in upcoming texts.

After implementation, for a `load` instruction, the NPC will require 3 cycles to complete:

1. In the first cycle, the IFU issues an instruction fetch request.

2. In the second cycle, the IFU receives the instruction and passes it to subsequent modules for decoding. Upon identifying a `load` instruction, it sends a memory access request through the LSU.
3. In the third cycle, the LSU retrieves the data and hands it over to the WBU for register writeback.

Simultaneous Read and Write to the Same Address - RTFM (Read The Friendly Manual)

When simultaneous read and write operations occur on the same address, which data will the memory actually return? The specific behavior in such cases needs to be referred to the manual: the device's manual specifies the behavior of this operation, and in some devices, the read result is undefined. This is a common characteristic in SRAMs and Block RAMs found in FPGAs.

Therefore, it is better for the master side to avoid simultaneous read and write operations to the same address. If unavoidable, detection logic can be implemented on the master side. When detecting simultaneous read and write to the same address, options include delaying the write operation to first read the old data or delaying the read operation to first write the data before reading the newly written data. This approach ensures that the read result is well-defined.

However, in the current NPC design, the IFU only issues read requests, and the LSU does not simultaneously issue read and write requests (depending on whether a `load` or `store` instruction is being executed). Therefore, you do not need to consider this issue within the NPC at this stage.

More General Memory Device

In reality, the characteristic of a 1-cycle read latency is typically only achievable with SRAM, because it can be produced using the same process as the processor. However SRAM is very expensive. To achieve lower-cost memory, other technologies with higher storage density like DRAM are often used. However, due to electrical characteristics, the read latency of these memories is typically greater than 1 cycle of the processor.

In this scenario, the processor cannot continuously send read requests because the processor's request rate exceeds the memory's service rate. This situation leads to the memory device being occupied by useless requests, significantly reducing the efficiency of the entire system. To address this issue, the master needs to inform the slave when to send valid requests and also identify when the slave can receive requests. This can be achieved through a handshake mechanism: by adding a pair of handshake signals in the read protocol between the master and slave. Specifically, adding `rvalid` indicates that the read request `raddr` sent by the processor is valid, and adding `rready` indicates that the memory device can receive read requests. Here, `rvalid` effectively acts as a read enable signal `ren`.

```
+-----+ raddr[log2(N)-1:0] ---> +-----+
|     | rvalid             ---> |     |
|     | <---             rready |     |
|     | <---         rdata[31:0] |     |
| CPU | waddr[log2(N)-1:0] ---> | MEM |
|     | wdata[31:0]        ---> |     |
|     | wen                ---> |     |
|     | wmask[3:0]         ---> |     |
+-----+                         +-----+
```

In fact, when memory completes a read operation is also unpredictable in advance. For example, DRAM periodically refreshes the capacitors of memory cells. If a read request is received during a refresh cycle, the data will only be read out after the refresh cycle is completed. On the other hand, the processor may not be ready to receive data from memory because the previously read data has not been fully utilized. These issues also need to be addressed through handshake signals: adding `rvalid` signal indicates that the memory has read valid data, and adding `rready` signal indicates that the processor is ready to receive data from memory. However, these two signals have the same names as the handshake signals for read requests mentioned earlier. To distinguish, we add the prefix `a` to the handshake signals related to read addresses, resulting in `arvalid` and `arready`.

```
+-----+ araddr[log2(N)-1:0] ---> +-----+
|     | arvalid             ---> |     |
|     | <---             arready |     |
|     | <---         rdata[31:0] |     |
|     | <---              rvalid |     |
| CPU | rready              ---> | MEM |
|     | waddr[log2(N)-1:0]  ---> |     |
|     | wdata[31:0]         ---> |     |
|     | wen                 ---> |     |


|      | wmask[3:0]          ---> |      |
+------+                          +------+
```

Therefore, in the completion process of a read transaction, both the master and slave need to go through two handshakes: The master first waits for `arready` to ensure the slave has received the read address, then waits for `rvalid` to receive the read data. The slave, on the other hand, first waits for `arvalid` to receive the read address, then waits for `rready` to ensure the master receives the read data. Of course, at the RTL implementation level, these are all state machines.

Similarly, write transactions also require handshakes, thus necessitating the addition of `wvalid` and `wready` signals:

```
+------+ araddr[log2(N)-1:0] ---> +------+
|      | arvalid             ---> |      |
|      | <---             arready |      |
|      | <---          rdata[31:0] |      |
|      | <---              rvalid |      |
| CPU  | rready              ---> | MEM  |
|      | waddr[log2(N)-1:0]  ---> |      |
|      | wdata[31:0]         ---> |      |
|      | wmask[3:0]          ---> |      |
|      | wvalid              ---> |      |
+------+ <---              wready +------+
```

The Significance of Microarchitectural Design in Handshake Signals - Decoupling

From the Microarchitectural point of view, the most important significance of the handshake signal is to shield the processing delay between the two communicating devices. For example, the timing of when data is read from DRAM is influenced by many factors, including refresh timing, DRAM controller request scheduling, whether the row buffer in the DRAM chip is hit, and even the electrical characteristics of the chip. Similarly, when the CPU sends requests and receives data, it is also influenced by various factors such as when the program executes memory access instructions, pipeline congestion, and cache status.

If one device has to consider these factors of the other device during communication, such a design would be impossible to achieve because we cannot know the state of the other device.

> Successfully decoupling the states of both devices through handshake signals allows both devices to not worry about the state of the other module and simply wait for the handshake. Therefore, as long as modules adhere to the same bus protocol, they can smoothly operate after connecting to the bus.

Error handling and exceptions

In certain scenarios, errors can occur when memory device processes read and write operations. For example, reading or writing addresses beyond the storage range, or detecting damaged memory cells through checksums. In such cases, the slave typically should inform the master of the error and let the master decide how to proceed. Therefore, when returning read data from memory, an additional `rresp` signal is transmitted to indicate whether the read operation was successful. If unsuccessful, the read data `rdata` is considered invalid. Similarly, after processing a write operation, the memory also needs to send back a write response signal `bresp` (where `b` stands for `backward`). Of course, this signal also requires a handshake.

```
+-----+ araddr[log2(N)-1:0] ---> +-----+
|     | arvalid             ---> |     |
|     | <---            arready |     |
|     | <---         rdata[31:0] |     |
|     | <---          rresp[1:0] |     |
|     | <---              rvalid |     |
|     | rready              ---> |     |
| CPU | waddr[log2(N)-1:0]  ---> | MEM |
|     | wdata[31:0]         ---> |     |
|     | wmask[3:0]          ---> |     |
|     | wvalid              ---> |     |
|     | <---              wready |     |
|     | <---          bresp[1:0] |     |
|     | <---              bvalid |     |
+-----+ bready              ---> +-----+
```

In a processor, if there are errors in read or write operations, exceptions can be further thrown to notify the software for handling. For example, RISC-V has defined 3 types of `Access Fault` exceptions, representing errors when fetching instructions from memory, reading data, and writing data. In this way, errors within the memory can be communicated to the processor through bus protocols and then notified to the software via the processor's exception handling mechanism.

Bus protocol widely used in the industry - the AXI protocol family.

We make a slight modification to the bus protocol mentioned above:

1. Separate the write address and write data, using the `aw` prefix for write address and the `w` prefix for write data.
2. Rename `wmask` to `wstrb`, and categorize the signals into 5 groups based on their functions.

```
araddr  --->                araddr  --->               araddr  ---> -+
arvalid --->                arvalid --->               arvalid --->
AR
<--- arready                <--- arready               <--- arready -+
<--- rdata                  <--- rdata
<--- rresp                  <--- rresp                 <--- rdata   -+
<--- rvalid                 <--- rvalid                <--- rresp    |
rready  --->       1        rready  --->       2       <--- rvalid   R
waddr   --->      ===>      awaddr  --->      ===>     rready  ---> -+
wdata   --->                awvalid ---> *
wmask   --->                <--- awready *             awaddr  ---> -+
wvalid  --->                wdata   --->               awvalid --->
AW
<--- wready                 wmask   --->               <--- awready -+
<--- bresp                  wvalid  --->
<--- bvalid                 <--- wready                wdata   ---> -+
bready  --->                <--- bresp                 wstrb   --->  |
                            <--- bvalid                wvalid  --->  W
                            bready  --->               <--- wready  -+

                                                       <--- bresp   -+
                                                       <--- bvalid   B
                                                       bready  ---> -+
```

This gives us the specification of the AXI4-Lite bus protocol from the AMBA AXI manual! AXI4-Lite consists of 5 transaction channels: Read Address (AR), Read Data (R), Write Address (AW), Write Data (W), and Write Response (B). Their operational states are solely determined by their respective handshake signals, allowing them to work independently. For example, read requests and write requests can be sent simultaneously in the same cycle.

In practical usage, we also need to avoid two scenarios related to handshake signals:

1. Both master and slave are waiting for each other to set the handshake signal to 1:
   - The master waits for the slave to set `ready` to 1 before setting `valid` to 1.
   - The slave waits for the master to set `valid` to 1 before setting `ready` to 1.
   - This results in an endless waiting loop, causing a **deadlock**.
2. Both master and slave are tentatively engaging in handshaking, but when they fail, they cancel the handshake
   - In the first cycle, the master sets `valid` to 1, but `ready` is 0, leading to a failed handshake.
   - In the second cycle, the slave observes that the master set `valid` to 1 in the previous cycle and sets `ready` to 1 in this cycle. However, as the previous handshake failed, the master sets `valid` back to 0 in this cycle, resulting in another failed handshake.
   - In the third cycle, the master notices that the slave set `ready` to 1 in the previous cycle and sets `valid` to 1 in this cycle. But due to the previous failed handshake, the slave sets `ready` back to 0 in this cycle, leading to another failed handshake.
   - This continuous back-and-forth results in both sides endlessly waiting, causing a **livelock**.

```
          +----+    +----+    +----+    +----+
 clk      |    |    |    |    |    |    |    |
          +    +----+    +----+    +----+    +----+
          +---------+         +---------+
valid               |         |         |
                    +---------+         +----------
                    +---------+         +----------
ready               |         |         |
          +---------+         +---------+
```

Avoiding Deadlock and Livelock in Handshaking

To avoid the above problems, the AXI specification added some constraints on the behavior of handshake signals. You need to RTFM (Read The Friendly Manual) to find these constraints and understand them correctly.

Please note that you must consult the official manual. If you rely on sources that are not sufficiently formal, you will find yourself in a painful debugging black hole when integrating into the SoC.

Let NPC support AXI4-Lite

By now, you have understood why each signal is included in the AXI4-Lite bus protocol and grasped the design philosophy of this bus specification from a demand perspective. Now, you can integrate AXI4-Lite into NPC, and let NPC to access memory through AXI4-Lite.

Convert the memory access interfaces of IFU and LSU to AXI4-Lite

1. Transform the memory access interfaces of IFU and LSU into AXI4-Lite.
2. Wrap the SRAM modules accessed by IFU and LSU with AXI4-Lite, while internally using DPI-C for data read/write operations.

Since IFU only performs read operations on memory and does not write to memory, the handshake signals for the `AW`, `W`, and `B` channels of IFU can all be set to 0. However, a better practice is to ensure they remain 0 at the other end of the handshake signals using `assert()`. Additionally, even though the access latency of the SRAM module is currently fixed at 1 cycle, you need to correctly implement the AXI4-Lite bus protocol using handshakes on both the master and slave ends.

Knowledge Gained through Reflection is True Mastery

Traditional textbooks often introduce buses at the protocol level without explicitly detailing how to implement a bus at the RTL level. Therefore, if one solely relies on textbooks, buses remain a relatively abstract concept. "OSOC" is a hands-on learning project that has already introduced how to implement buses at the RTL level in our course materials.

However, if you find that there are still some difficulties that are hard to overcome during the implementation process, it's time to be alarmed: you may lack the ability to progressively translate requirements into code. You may need to reflect on your past learning methods, such as:

- Are you overly reliant on diagrams, leading to a lack of microarchitecture design skills and feeling lost without diagrams?
- Even worse, are you overly dependent on materials and books filled with example code? Resulting in not only a lack of microarchitecture design skills but also an

inability to convert designs into code. We strongly discourage beginners from reading such materials and books until their first version of code is functional!

We hope that you prioritize "independent thinking" in your learning journey rather than just being a code transporter. If you find independent thinking challenging from this point onwards, it's likely because you missed out on crucial aspects during your previous learning, leaving you unequipped to tackle upcoming tasks alone. If this is the case, we recommend restarting your learning journey with the right mindset through "OSOC".

In fact, the AXI4-Lite we are currently implementing is just a simplified version of AXI. The complete AXI bus specification includes more signals and features. If you are interested, you can refer to the official manual to learn more about the details. We will gradually introduce more features that we need to use in the upcoming course materials.

Maintain the Right Mindset, Gradually Grasp All Details through Iterative Development and Debugging

However, as a widely used bus protocol in the industry, AXI has many details, making it challenging for most beginners to fully understand it upon initial exposure. Therefore, it is essential to maintain the right mindset. The idea of "writing all bus code at once and never needing to change it again" is unrealistic.

In reality, everyone gradually comprehends all the details of a bus through iterative development and debugging. Even senior engineers go through this process when learning new concepts. The notion of wanting to master everything instantly does not align with the objective laws of learning.

Test Bus Implementation

Implement random delays in the SRAM module to test whether the bus implementation works correctly with arbitrary delays. Of course, you can gradually add memory access delays in a simple-to-complex order:

1. Modify the access delays of SRAM sequentially to 5, 10, 20, etc.
2. Add an LSFR in the SRAM module to determine the delay for the current request.
3. Add LSFR in IFU and LSU to determine the delay of the `valid` signal in the `AR` / `AW` / `W` channels and the delay of the `ready` signal in the `R` / `B`

channels.

If the NPC can boot RT-Thread correctly despite random delays filled with LFSRs, it gives you a big boost of confidence in your code.

Arbiter in Bus

After the modifications mentioned above, we have instantiated two SRAM modules. However, in real hardware, these two SRAM modules should map to the same memory. Currently, this is achieved by accessing the unique memory in the simulation environment using `pmem_read()` / `pmem_write()`. In real hardware, functions like `pmem_read()` / `pmem_write()` do not exist. Therefore, it is necessary to consider how to enable IFU and LSU to access the same memory in the hardware implementation.

This is actually a multi-master problem. Since a slave can only handle one request at a time, an arbiter is used to make decisions: When multiple masters simultaneously access the same slave, the master that obtains access rights will be granted permission to successfully access the slave. The requests from other masters will be blocked at the arbiter, waiting for the master with access rights to finish its access before they can gain access next.

To summarize, the arbiter needs to implement the following functions:

1. Scheduling: Selecting a master that is currently sending a valid request.
2. Blocking: Blocking access from other masters.
3. Forwarding: Forwarding the request from the master that has obtained access rights to the slave, and upon receiving the response from the slave, forwarding it back to the respective master.

When multiple masters are accessing simultaneously, the specific selection process is essentially a scheduling strategy issue. In complex systems, scheduling strategies need to consider more factors: firstly, avoiding starvation, ensuring that each master can eventually gain access rights after a finite number of arbitrations; secondly, preventing deadlock, where the blocking caused by the arbiter should not lead to a situation of circular waiting in the entire system. However, since NPC is a multi-cycle processor and IFU and LSU masters do not send requests simultaneously, you do not need to consider complex scheduling strategies. Choosing any simple scheduling strategy will suffice.

Implement an AXI4-Lite Arbiter

Keep one AXI4-Lite SRAM module and develop an AXI4-Lite arbiter to select a master from IFU and LSU to communicate with the SRAM module.

Hint: The arbiter is essentially a state machine, and the blocking and forwarding functionalities are fundamentally achieved through manipulating handshake signals.

Evaluate NPC Clock Frequency and Program Performance

With the implementation of AXI4-Lite, NPC can now be connected to actual SRAM. The evaluation will focus on an NPC with an AXI4-Lite interface, including the recently implemented AXI4-Lite arbiter. The AXI4-Lite interface SRAM module implemented through DPI-C is not within the scope of this evaluation.

Following the same evaluation method, another version of the NPC has a clock frequency of 297.711MHz on the nangate45 process provided by default by the `yosys-sta` project. This calculates to microbench needing 1.394s to run, but simulation takes 29.861s. The increase in simulation time is due to the multi-cycle NPC having a lower IPC than the single-cycle NPC, requiring more cycles to execute programs. Although IPC decreases, the significant increase in clock frequency results in faster program execution.

It's important to remember that the evaluation results for the single-cycle NPC mentioned above are overly optimistic, possibly to an impractical extent. However, the evaluation results for this multi-cycle NPC are more realistic, at least that implementing a 1MB SRAM is possible. Nonetheless, there is still a significant difference from our upcoming tape out, as the cost of fabricating a 1MB SRAM remains high. Next steps involve integrating into SoC to make the evaluation results more representative of chip fabrication scenarios.

Why Implement the Bus First?

Many students have been puzzled by this question in the past, with some even believing it to be a trap set by the course material. The primary reason for their confusion stems from the notion that all code will need to be rewritten when transitioning to implementing pipeline. Another perspective suggests that starting with a single-cycle design will also lead to a complete rewrite later on.

The concern about the need for rewriting arises partly from traditional textbooks, which focus on explaining processor design principles clearly but often overlook the smooth transition between different microarchitectures. This aspect falls more into the realm of engineering practice, which traditional textbooks do not typically emphasize as a standalone topic. Additionally, some students overly rely on traditional textbooks and implement solutions without considering appropriate design patterns for such transitions.

For beginners, this is not a significant issue as understanding design patterns requires a deep understanding of various details before abstracting them. We should not expect beginners to come up with excellent design patterns right from the start. However, if you aim to delve deeper into the subject matter, you should not view textbook content as the entirety of processor design or consider rewriting as a waste of time. Embracing the opportunity to rewrite actually presents significant growth potential for beginners. It prompts questions like: Why are there such significant differences between the two versions? Can common characteristics be abstracted from these differences? How can we do things better if we were to start over?

Valuable experiences and innovative ideas are often stem from addressing challenges. As individuals progress into work roles, they will encounter diverse problems that lack standard textbook solutions. Knowledge from textbooks is limited; relying solely on them restricts our growth potential. What truly aids in solving future unknown problems is our thinking habits - the mental frameworks developed through repeated contemplation hold more significance than textbook knowledge.

Reflecting on processor design patterns, we have been contemplating and summarizing related experiences since May 2017. These insights were put into practice during the Longxin Cup competitions at Nanjing University in 2017 and 2018, as well as in the inaugural "OSOC" program initiated by the University of Chinese Academy of Sciences in 2019, without causing excessive rewrites. Many students may not be aware of these circumstances and may make judgments based on their own experiences.

Specifically, leveraging appropriate features of Chisel allows for relatively minor code adjustments: transitioning from single-cycle to multi-cycle designs, only 30 lines of code need to be changed (3.75% of 800 lines of code); transitioning from multi-cycle to pipeline without forwarding, only 50 lines of code need to be changed (5.00% of 1000 lines of code); adding forwarding functionality to the pipeline, only 20 lines of code need to be changed (1.82% of 1100 lines of code). Even if developing in Verilog, with correct design patterns, the proportion of code changes should be roughly similar. From a proportional standpoint, it is far from necessitating complete rewrites.

We want to say: as a beginner, maintain a curious mindset and embrace the principle of "learning by doing." When others suggest that you need to rewrite your work, do not blindly follow such advice; instead, ponder why and delve into answering this question through your own practical exploration. In learning, you are indeed the protagonist.

So Why Implement the Bus First?

This is to practice the design principle of "complete first, perfect later."

By taking a single-cycle processor as the starting point, the pipeline should be considered part of the "perfect later" phase. Therefore, implementing the bus first is a step towards aligning the processor towards being able to tape out, achieving the "complete first" aspect. Removing any functionality would either render the processor unable to run RT-Thread or result in significant differences between memory, peripherals, and a synthesizable design

Building on the "complete first" foundation, we can then use quantitative evaluation methods to understand the performance benefits brought by each optimization measure. This approach aims to enable beginners to quantitatively comprehend how pipeline design enhances system performance, moving beyond traditional textbooks that often treat pipeline design as a task of translating block diagrams into RTL code.

Multi-device system

So far, our system only has memory. Obviously, a real computer system consists of more than just memory; it also includes other devices. Therefore, we need to consider how to allow NPCs to access other devices.

When we previously studied devices, we introduced memory-mapped I/O, a mechanism that uses different memory addresses to indicate the devices accessed by the CPU. In our simulation environment, we used `pmem_read()` and `pmem_write()` to select the device to access based on the memory address, thus implementing memory-mapped I/O functionality. However, real hardware does not have the `pmem_read()` and `pmem_write()` functions found in the simulation environment.

In reality, hardware implements memory-mapped I/O through a crossbar module (sometimes written as Xbar). The Xbar is a multiplexer switch module for buses that can forward requests from input buses to different output buses based on the requested address, thereby passing them to different downstream modules. These downstream modules could be devices or another Xbar. For example, in the diagram below, the Arbiter is used to select a request from IFU and LSU and forward it downstream. The downstream Xbar then forwards the request to the downstream device based on the address in the request.

```
+-----+      +---------+      +------+      +-----+
| IFU | ---> |         |      |      | ---> | UART|  [0x1000_0000, 0x1000_0fff)
+-----+      |         |      |      |      +-----+
             | Arbiter | ---> | Xbar |
+-----+      |         |      |      |      +-----+
| LSU | ---> |         |      |      | ---> | SRAM|  [0x8000_0000, 0x80ff_ffff)
+-----+      +---------+      +------+      +-----+
```

If the requested address falls within the address space range of the UART serial device, the Xbar will forward the request to the UART. If the requested address falls within the address space range of SRAM, then the request will be forwarded to SRAM. If the Xbar finds that the requested address does not belong to any downstream address space, for example `0x0400_0000`, it will return an error `decerr` via the AXI4-Lite `resp` signal, indicating a decoding error. Here, "address decoding" refers to translating the requested address into the ID of the downstream bus channel, and the address decoder in Xbar can be seen as a core module in hardware implementation of memory-mapped I/O.

Arbiter and Xbar can also be combined into a multi-input multi-output Xbar, sometimes referred to as an Interconnect or bus bridge depending on the context. For example, the diagram below shows a 2-input 2-output Xbar. It can connect multiple masters and multiple slaves, first using an Arbiter to determine which master the current request is from, and then based on that request's address deciding which slave to forward it to.

```
+-----+      +------+      +-----+
| IFU | ---> |      | ---> | UART|  [0x1000_0000, 0x1000_0fff)
+-----+      |      |      +-----+
             | Xbar |
+-----+      |      |      +-----+

| LSU | ---> |       | ---> | SRAM|  [0x8000_0000, 0x80ff_ffff)
+-----+      +-------+      +-----+
```

Physical Memory Attributes (PMA) and Bare Metal Programming

The above topology connection method may also cause some special issues. Since the IFU can be connected to the Xbar and UART, it means the CPU can fetch instructions from the UART, but obviously, the UART device cannot store programs and instructions.

Therefore, if the program mistakenly jumps to `0x1000_0000`, the CPU will send a read request to the UART, and the UART will return a data based on the device's behavior, which may represent the encoding of the UART's internal state or a character received by the UART. On one hand, the CPU may mistakenly execute the result returned by UART as an instruction, leading to serious errors; on the other hand, IFU's access to the UART device may change the device's state, causing the UART to enter an unpredictable state.

To avoid triggering such problems, generally some check mechanisms are added in hardware, adding several permission attributes for each address space, such as readable flag, writable flag, executable flag, etc. Before IFU issues an instruction fetch request, it first checks if the address of the request belongs to an executable address space, if not executable, it throws an exception directly. In RISC-V, if the address space of devices in the system is fixed, permission checks can be implemented through the PMA (Physical Memory Attribute) mechanism; while for modern PCI-e devices with dynamically allocated address spaces during operating system initialization, permission checks can be implemented through PMP (Physical Memory Protection) mechanism or virtual memory mechanism. If you are interested in these two mechanisms, you can refer to relevant content in the RISC-V manual.

Of course, if these check mechanisms are not implemented in the CPU, one must be extremely careful when developing bare metal programs: if a bare metal program goes astray, the consequences could be unimaginable.

With Xbar in place, we can now move the peripheral functions previously implemented in the simulation environment to RTL.

Implementing UART Functionality with AXI4-Lite Interface

Write a slave module with an AXI4-Lite interface, containing a device register. When a write request is sent to this device register, the lower 8 bits of the written data will be treated as a character and output using `$write()` or `printf()`. For ease of testing, the address of this device register can be set to be the same as the address used for the UART in the previous simulation environment. Once implemented, you will also need to write an Xbar module to integrate this UART-capable module into the system.

In reality, we have not fully implemented a UART using RTL, as `$write()` or `printf()` still rely on the simulation environment for character output. However, as an exercise in bus implementation, this is sufficient, considering that UART implementation involves many electrical details. Soon, we will integrate into an SoC that includes a real UART controller. By testing bus implementation through this exercise now, integration into the SoC in the future will also be smoother.

Implementing CLINT with AXI4-Lite Interface

CLINT(Core Local INTerrupt controller) is a commonly used interrupt controller in RISC-V systems, responsible for managing clock interrupts and software interrupts. However, since our system currently does not require interrupt functionality, we will focus on clock-related functions for now.

You are required to implement a CLINT module with an AXI4-Lite interface and integrate it into the system. CLINT includes a read-only device register `mtime`, which increments at a certain rate, with the simplest implementation being an increment by 1 per cycle. Similarly, for testing convenience, its address can be set to be the same as the address used for the clock in the previous simulation environment.

However, the progression of `mtime` does not directly reflect the passage of time; there is a coefficient difference that needs to be processed by software after being read out. In real processor chips, this coefficient is generally equal to the clock frequency in the CLINT module, allowing software to measure real time. In the simulation environment where there is no concept of a base frequency, if this coefficient equals the simulation rate, we can calculate real-time progression based on the progression of `mtime` in the simulation environment. Specifically, you will also need to modify the relevant IOE code to make `AM_TIMER_UPTIME` return time close to real time.

Lastly, you need to consider the bit width of the `mtime` register. The `mtime` defined in the manual mentioned above is 64 bits to avoid overflow during actual usage. However, currently NPC is 32 bits; if we only read out the lower 32 bits of `mtime`, after some time, `mtime` will overflow, causing errors in the system's time functionality. Although it may not be easy to reach the point of overflow of `mtime` in the simulation environment, if NPC runs at a frequency of 500MHz in the future, overflow is highly likely to occur. Therefore, software running on a 32-bit NPC needs to sequentially read out the lower 32 bits and upper 32 bits of `mtime`, combine them into a 64-bit value for use by higher-level applications.