计算机组成原理实验报告

一、 CPU 设计方案综述

(一) 总体设计概述

此 CPU 为通过 Verilog 硬件描述语言实现的 FDEMW 全速转发顺序执行 5 级流水线CPU，支持MIPS指令集中的{addi, addu, subu, ori, lw, sw, beq, lui, j, jal, jalr, jr, nop}共 13 条指令。“5 级”所指及相应部件如**表 1** 所示。

表 1 流水阶段及相应部件

阶段	部件
Instruction **F**etch	PC, IM
Instruction **D**ecode	NPC, GRF, EXT, CMP
Execution	ALU
Read Data from **M**emory Write Data to **M**emory	DM
Write Data Back to Register	(GRF)

实现此 CPU 时，将其分为“机制”（mechanism）与“策略”（policy）两大部分，即将数据通路相关模块与核心控制模块分开考虑，并采用模块化和层次化设计方法。顶层有效的驱动信号包括且仅包括：时钟信号 clk 和同步复位信号 reset。

(二) 数据通路相关模块实现方法

1. PC

由起始值为 0x00003000 的具有同步复位功能的 reg [31:0]实现，复位值为起始地址。可接收 enable 信号，以便在 nop 指令时保持原值不变。

2. IM

由容量为$32\ \text{bit} \times 1024 = 4\ \text{B} \times 1024 = 4\ \text{KB}$的 reg [31:0]实现。在这种设计中，每个连续的地址都能取一个 word，故将 PC 中储存的地址右移 2 位用于在 IM 中寻址。此处暂时未将 PC、NPC 和 IM 封装在 IFU 这一大模块中；如有需要，后续修改也并不困难。

3. NPC

由计算下一条指令地址的逻辑实现。

4. GRF

由具有同步复位功能的 reg [31:0]实现，复位值为 0x00000000。可接收 writeable 信号，以便在非写入指令时，GRF 中的数据不能被误写。由于 0 号寄存器的值始终为 0x00000000，故向 0 号寄存器写入值始终无效。

5. EXT

根据控制信号，将输入的 16 位立即数进行 0 扩展、符号扩展或加载到高位为 32 位整数后输出。

6. CMP

直接连接在 GRF 后的 32 位整数比较分支信号生成器，使在该流水线设计中，b 类指令后只需插入一个 nop 指令就能保证正确性。若仍在 ALU 中生成 b 类指令的分支信号，则在该流水线设计中，必须在 b 类指令后插入 3 个 nop 指令才能保证正确性。

7. ALU

支持 32 位整数加、减、与、或、或非、异或、移位、比较等功能，不检测溢出。输出结果由 ALUop 信号控制。

8. DM

由起始地址为 0x00000000 的具有异步复位功能的容量为$32\ \text{bit} \times 1024 = 4\ \text{B} \times 1024 = 4\ \text{KB}$的 reg [31:0]实现，复位值为 0x00000000。可接收 writeable 信号，以便在非写入指令时，DM 中的数据不能被误写。在这种设计中，每个连续的地址都能取一个 word，故将地址右移 2 位来以 word 为单位寻址，辅以第 1 位来得到 half-word、第[1:0]位来得到 byte。

9. 流水线寄存器组

4个流水线寄存器组衔接各流水阶段，接收上一阶段部件新产生的数据，并将储存的上一阶段部件的旧数据传给下一阶段部件。指令将在这些寄存器间流动，并且传给相应阶段的控制器来产生控制信号。均可同步复位，复位值为 0x00000000。其中，IF/ID 级寄存器可接收 writeable 信号，以便在 nop 指令时保持原值不变。

(三) 重要机制实现方法

1. 转发

首先假设所有数据冒险均可通过转发解决。也就是说，当某一指令前进到必须使用某一流水阶段中相应寄存器的值时，这个寄存器的值一定已经产生，并存储于后续某个流水线寄存器中，只是还没来得及写入 GRF。存在这种需求的部件有：

(1) D 级 CMPin1 和 CMPin2；

(2) D 级 NPC 中 retAddr 的输入；

(3) E 级 ALUin1 和 ALUin2；

(4) M 级 DMdataIn。

为了实现转发机制，需要对这些输入前加上一个多选器。这些多选器的默认输入来源是上一级中已经转发过的数据。特别地，由于 GRF 既可被视为在 D 级中，也可以被视为在 W 级中，则需对 GRF 采用内部转发机制，即当前 GRF 被写入的值会即时反馈到读取端上。

由此可确定截至目前选择信号的生成规则：只要当前位点的读取寄存器地址和某转发输入来源的写入寄存器地址相等且不为 0，就选择该转发输入来源；在有多个转发输入来源都满足条件时，最新产生的数据优先级最高。

2. 暂停

事实上，有些数据冒险并不能由转发处理，因为先前指令的新数据还未来得及产生，后续指令就要使用。此时只能暂停流水线，等待先前指令的新数据产生。为了方便处理，需要暂停流水线时，将指令暂停在 D 级。如何判断某条指令的新数据是否已经产生？需要引入**需求时间—供给时间模型**。

定义需求时间 Tuse 为：指令进入 D 级后，后续某个功能部件必须要使用某寄存器值时，需要经过的时钟周期数。

定义供给时间 Tnew 为：位于 E 级及其后续各级的指令，能够产生要写入某寄存器的新值时，需要经过的时钟周期数。在目前的 CPU 中，W 级的指令 Tnew 恒为 0；对于同一条指令，TnewM 为 TnewE − 1 和 0 之间的最大值。

现在就可以得出需要暂停的条件了：D 级指令读取寄存器的地址与 E 级或 M 级的指令写入寄存器的地址相等且不为 0，且 D 级指令的 Tuse 小于对应 E 级或 M 级指令的 Tnew。在其他情况下，数据冒险均可通过转发机制解决。

暂停时需阻止任何新数据写入 PC、阻止任何新数据写入 IF/ID 级寄存器、复位 ID/EX 级流水寄存器。

3. 控制与抽象

控制器将每一条机器指令中的信息，解码为传输给 CPU 各部分的控制信号。实现控制逻辑需要考虑**译码方式**和**译码风格**。

译码方式上，可分为**集中式译码**与**分布式译码**。**集中式译码**指在 `F` 级或 `D` 级前，根据该指令将所有控制信号全部解析出，然后让其随着流水往后逐级传递。使用这种方法，只需要在新指令到来时进行一次译码，减少了后续流水级的逻辑复杂度，但流水级之间需要传递的信号数量很大。**分布式译码**指每一级都部署一个控制器，负责译出当前级指令所需的控制信号。这种方法较为灵活，“现译现用”有效降低了流水级间传递的信号量，但是需要实例化多个控制器，浪费硬件资源。

译码风格上，可分为**指令驱动型**与**控制信号驱动型**。**指令驱动型**指控制逻辑整体在一个 `case` 语句之下，通过判断指令的类型，来对所有的控制信号一一进行赋值。这种方法便于添加指令，不易遗漏控制信号，甚至可以通过控制信号**表 2**，编写 Python 脚本将 Excel 表格自动转化为相应 Verilog 代码。由于代码编辑器多数都具有多行光标功能，手动编写控制器代码也并不繁琐。**控制信号驱动型**指为每个指令定义一个 `wire` 型变量，使用或运算描述组合逻辑，对每个控制信号进行单独处理。这种方法可显著减少代码量，缺点是如错添或漏添了某条指令，很难锁定出现错误的位置。为使修改数据通路和添加指令更加直观、便捷和机械化，采用分布式译码的方式和指令驱动型的风格。

Tim Peters 的 *The Zen of Python*（节选）特别能体现本 CPU 采用的控制逻辑实现原则：

> *Beautiful is better than ugly.*
> *Explicit is better than implicit.*
> *Simple is better than complex.*
> *Complex is better than complicated.*
> *Flat is better than nested.*
> *Sparse is better than dense.*
> *Readability counts.*
> *Special cases aren't special enough to break the rules.*
> *In the face of ambiguity, refuse the temptation to guess.*
> *There should be one - and preferably only one - obvious way to do it.*
> *If the implementation is hard to explain, it's a bad idea.*
> *If the implementation is easy to explain, it may be a good idea.*

具体地，能由控制器进行译码并抽象的，就不用原始指令数据让多选器实现；能通过列表的形式机械地实现的，就不用条件表达式；能将控制信号定义为宏的，就不用具体数值（因此实际控制信号并不完全与**表 2** 中的值对应）。例如，转发和暂停机制需要指令的读取寄存器和写入寄存器的地址信息，而这些信息在控制器解码指令时就可以确定并统一抽象为 `Ause1`、`Ause2` 和 `Anew`（比如`jal`的`Anew`就固定为`31`），因此在生成选择信号时，无需对指令分类，大大简化了控制逻辑。又例如，由于 `jal` 或 `jalr` 指令产生的数据为 `PC + 8` 并非由 ALU 得出，为了保持原有转发选择逻辑不变，将 `PC + 8` 的值与 `ALUoutOriginal` 在 `E` 级由选择信号 `writeRetAddr` 控制，抽象为 `ALUout` 传给 `EX/MEM` 寄存器，体现了“*Special cases aren't special enough to break the rules*”的思想。

实现转发需要寄存器地址（A 信息），实现暂停需要寄存器值供需时间（T 信息），称用 A 信息和 T 信息实现转发与暂停的方法为 AT 法。“*There should be one - and preferably only one - obvious way to do it.*”在此 CPU 设计中，这大概就是指 AT 法吧。

4. 核心逻辑的具体实现

```
wire retAddrFwdOp = (Ause1D == AnewM && Ause1D != 0 && TnewM == 0) ? 1 : 0;
wire CMPin1fwdOp = (Ause1D == AnewM && Ause1D != 0 && TnewM == 0) ? 1 : 0;
wire CMPin2fwdOp = (Ause2D == AnewM && Ause2D != 0 && TnewM == 0) ? 1 : 0;
wire [1:0] ALUin1fwdOp = (Ause1E == AnewM && Ause1E != 0 && TnewM == 0) ? 2 :
                         (Ause1E == AnewW && Ause1E != 0 && TnewW == 0) ? 1 :
                                                                          0;
wire [1:0] ALUin2fwdOp = (Ause2E == AnewM && Ause2E != 0 && TnewM == 0) ? 2 :
                         (Ause2E == AnewW && Ause2E != 0 && TnewW == 0) ? 1 :
                                                                          0;
wire [31:0] ALUresult = writeRetAddr ? PCplus4E + 4 : ALUresultOriginal;
wire stall = (Ause1D == AnewE && Ause1D != 0 && Tuse1D < TnewE) ||
             (Ause2D == AnewE && Ause2D != 0 && Tuse2D < TnewE) ||
             (Ause1D == AnewM && Ause1D != 0 && Tuse1D < TnewM) ||
             (Ause2D == AnewM && Ause2D != 0 && Tuse2D < TnewM);
```

流水线CPU.xlsx

表 2 控制信号

Type	Instruction	opcode	funct	Ause1	Ause2	Anew	Tuse1	Tuse2	Tnew	NPCop	GRFwr	GRFdataOp	EXTop	ALUop	ALUAop	ALUBop	DMwr	DMstoreOp	DMloadOp
B	**beq**	**000100**		**rs**	**rt**	**0**	**0**	**0**	**0**	**0b01**	**0**			**0b0010**	**0**	**0**	**0**		
B	bgez	000001		**rs**	**rt**	**0**	**0**	**0**	**0**	0b01	0			0b0101	0	0	0		
B	bgtz	000111		**rs**	**rt**	**0**	**0**	**0**	**0**	0b01	0			0b0000	0	0	0		
B	blez	000110		**rs**	**rt**	**0**	**0**	**0**	**0**	0b01	0			0b0001	0	0	0		
B	bltz	000001		**rs**	**rt**	**0**	**0**	**0**	**0**	0b01	0			0b0100	0	0	0		
B	bne	000101		**rs**	**rt**	**0**	**0**	**0**	**0**	0b01	0			0b0011	0	0	0		
I	addi	001000		rs	0	rt	1	0	2	0b00	1	0b00	0b01	0b1011	0	1	0		
I	addiu	001001		rs	0	rt	1	0	2	0b00	1	0b00	0b01	0b1011	0	1	0		
I	andi	001100		rs	0	rt	1	0	2	0b00	1	0b00	0b00	0b0111	0	1	0		
I	lb	100000		**rs**	**0**	**rt**	**1**	**0**	**3**	0b00	1	0b01	0b01	0b1011	0	1	0		0b001
I	lbu	100100		**rs**	**0**	**rt**	**1**	**0**	**3**	0b00	1	0b01	0b01	0b1011	0	1	0		0b100
I	lh	100001		**rs**	**0**	**rt**	**1**	**0**	**3**	0b00	1	0b01	0b01	0b1011	0	1	0		0b000
I	lhu	100101		**rs**	**0**	**rt**	**1**	**0**	**3**	0b00	1	0b01	0b01	0b1011	0	1	0		0b011
I	**lui**	**001111**		**0**	**0**	**rt**	**0**	**0**	**2**	**0b00**	**1**	**0b00**	**0b10**	**0b1011**	**0**	**1**	**0**		
I	**lw**	**100011**		**rs**	**0**	**rt**	**1**	**0**	**3**	**0b00**	**1**	**0b01**	**0b01**	**0b1011**	**0**	**1**	**0**		**0b010**
I	**ori**	**001101**		**rs**	**0**	**rt**	**1**	**0**	**2**	**0b00**	**1**	**0b00**	**0b00**	**0b1000**	**0**	**1**	**0**		
I	sb	101000		**rs**	**rt**	**0**	**1**	**2**	**0**	0b00	0		0b01	0b1011	0	1	1	0b10	0b001
I	sh	101001		**rs**	**rt**	**0**	**1**	**2**	**0**	0b00	0		0b01	0b1011	0	1	1	0b01	0b000
I	slti	001010		rs	0	rt	1	0	2	0b00	1	0b00	0b01	0b0100	0	1	0		
I	sltiu	001011		rs	0	rt	1	0	2	0b00	1	0b00	0b01	0b0110	0	1	0		
I	**sw**	**101011**		**rs**	**rt**	**0**	**1**	**2**	**0**	**0b00**	**0**		**0b01**	**0b1011**	**0**	**1**	**1**	**0b00**	**0b010**

I	xori	001110		rs	0	rt	1	0	2	0b00	1	0b00	0b00	0b1010	0	1	0	
J	j	000010		0	0	0	0	0	0	0b11	0						0	
J	**jal**	**000011**		**0**	**0**	**31**	**0**	**0**	**0**	**0b11**	**1**	**0b10**					**0**	
R	add	000000	100000	rs	rt	rd	1	1	2	0b00	1	0b00		0b1011	0	0	0	
R	**addu**	**000000**	**100001**	rs	rt	rd	1	1	2	**0b00**	**1**	**0b00**		**0b1011**	**0**	**0**	**0**	
R	and	000000	100100	rs	rt	rd	1	1	2	0b00	1	0b00		0b0111	0	0	0	
R	jalr	**000000**	001001	rs	0	rd	0	0	0	0b10	1	0b10					0	
R	**jr**	**000000**	**001000**	**rs**	**0**	**0**	**0**	**0**	**0**	**0b10**	**0**						**0**	
R	nor	**000000**	100111	rs	rt	rd	1	1	2	0b00	1	0b00		0b1001	0	0	0	
R	or	000000	100101	rs	rt	rd	1	1	2	0b00	1	0b00		0b1000	0	0	0	
R	sll	**000000**	000000	0	rt	rd	0	1	2	0b00	1	0b00		0b1101	1	0	0	
R	sllv	000000	000100	rs	rt	rd	1	1	2	0b00	1	0b00		0b1101	0	0	0	
R	slt	**000000**	101010	rs	rt	rd	1	1	2	0b00	1	0b00		0b0100	0	0	0	
R	sltu	000000	101011	rs	rt	rd	1	1	2	0b00	1	0b00		0b0110	0	0	0	
R	sra	**000000**	000011	0	rt	rd	0	1	2	0b00	1	0b00		0b1111	1	0	0	
R	srav	000000	000111	rs	rt	rd	1	1	2	0b00	1	0b00		0b1111	0	0	0	
R	srl	**000000**	000010	0	rt	rd	0	1	2	0b00	1	0b00		0b1110	1	0	0	
R	srlv	000000	000110	rs	rt	rd	1	1	2	0b00	1	0b00		0b1110	0	0	0	
R	sub	000000	100010	rs	rt	rd	1	1	2	0b00	1	0b00		0b1100	0	0	0	
R	**subu**	**000000**	**100011**	rs	rt	rd	1	1	2	**0b00**	**1**	**0b00**		**0b1100**	**0**	**0**	**0**	
R	xor	000000	100110	rs	rt	rd	1	1	2	0b00	1	0b00		0b1010	0	0	0	
R	**nop**	**000000**	**000000**	0	rt	rd	0	1	2	**0b00**	**0**						**0**	

二、 测试方案

(一) 典型测试样例

对于普通运算和 `b` 类分支指令，除了测试指令功能，只需将冲突指令排列组合即可，此处暂略。需要注意的是，`jal` 指令允许在延迟槽中改变存入 GRF 中的`$ra` 值，尽管实际很少有程序会这样做。因此特意编写以下死循环程序检验正确性。

```
1   ori     $t0, $t0, 4
2   ori     $t1, $t1, 1
3   jal     label
4   sub     $ra, $ra, $t0
5   ori     $t2, $t2, 1
6   ori     $t3, $t3, 1
7   label:
8   ori     $t4, $t4, 1
9   jr      $ra
10  nop
```

(二) 自动测试程序

使用 Python 3 编写了一个自动测试程序。为了更加方便地编写自动测试程序，需修改 MARS 使其运行程序时能输出寄存器和存储器的写入明细字符串。还需通过 Vivado 导出命令行仿真辅助文件到一个测试专用目录下，其中包括了仿真开始时需要自动加载到 IM 中的机器码文件 `code.txt`。修改机器码时，只需修改测试专用目录下的 `code.txt` 文件即可自动加载到 IM 中，而 Verilog 代码可以在原工程目录中。Verilog 代码也可被导出到该目录下以增加便携性，但考虑到可能还会在原工程目录下修改 Verilog 代码，为避免反复修改反复导出或忘记导出浪费时间，暂时不导出 Verilog 代码进行仿真。自动测试程序与执行步骤大致如下：

(1) 将设定的仿真所需时间写入 `cmd.tcl` 文件。

(2) 随机生成指令程序文件。

(3) 调用 MARS 读取该文件并将汇编生成的机器码写入 `code.txt` 文件。

(4) 调用 MARS 运行该程序并将输出的寄存器和存储器的写入明细字符串写入文件。

(5) 调用 Vivado 相关组件依次进行 Compile、Elaborate 和 Simulate。仿真过程中，要求 GRF 和 DM 调用系统任务`$display` 显示在 TCL Console 中的内容会随自动生成的其它仿真信息记录在文件中。

(6) 比较 MARS 和仿真输出。

```
import os
import subprocess
import random
PAUSE_AT_FAIL    = True
TEST_START_NUM   = 1
TEST_END_NUM     = 708
USELESS_INFO_LEN = 17
SIMULATION_TIME  = "20 us"
ASM_FILE_DIR     = "code/708/"
VIVADO_DIR       = "C:/Xilinx/Vivado/2020.1/bin/"
AK = True
def gen_code(file_name):
    with open(file_name, "w") as f:
        for _ in range(5):
            rd = random.randint(1, 27)
            imm = random.randint(0, 3070) * 4
            f.write(f"ori\t${rd}, {imm}\n")
        for _ in range(5):
            target = random.randint(1, 27)
            source = random.randint(0, 27)
            imm = random.randint(0, 7) * 4
            f.write(f"sw\t${target}, {imm}(${source})\n")
            f.write(f"lw\t${target}, {imm}(${source})\n")
        for _ in range(5):
            target = random.randint(1, 27)
            source = random.randint(0, 27)
            rd = random.randint(0, 27)
            f.write(f"addu\t${target}, ${source}, ${rd}\n")
        for _ in range(5):
            target = random.randint(1, 27)
```

```
            source = random.randint(0, 27)
            rd = random.randint(0, 27)
            f.write(f"subu\t${target}, ${source}, ${rd}\n")
        for _ in range(5):
            rd = random.randint(1, 27)
            imm = random.randint(0, 0xffff)
            f.write(f"lui\t${rd}, {imm}\n")
        f.write("nop\nnop\n")
if __name__ == '__main__':
    print("Autotest started...")
    os.chdir(os.path.dirname(__file__))
    with open("cmd.tcl", "w") as f:
        f.write(f"run {SIMULATION_TIME};\nquit;")
    print("Initialization succeeded!")
    for i in range(TEST_START_NUM, TEST_END_NUM + 1):
        print(f"Testing case {i}...")
        asm_file_name = f"{ASM_FILE_DIR}testpoint{i}.asm"
        master_name = f"master/master_{i}.txt"
        query_name = f"query/query_{i}.txt"
        # gen_code(asm_file_name)
        subprocess.call(f"java -
jar MARS.jar {asm_file_name} nc mc CompactDataAtZero a dump .text He
xText code.txt")
        master = subprocess.Popen(f"java -
jar MARS.jar {asm_file_name} nc mc CompactDataAtZero db", shell=Fals
e, stdout=subprocess.PIPE, text="utf-
8").communicate()[0].splitlines()
        with open(master_name, "w") as f:
            f.writelines(master)
        os.system(VIVADO_DIR + "xvlog -prj vlog.prj --nolog")
        os.system(VIVADO_DIR + "xelab -debug typical -
L xil_defaultlib -L unisims_ver -L unimacro_ver -L secureip --
snapshot mips_tb xil_defaultlib.mips_tb xil_defaultlib.glbl --
nolog")
```

```
57          os.system(VIVADO_DIR + "xsim mips_tb -
    key {Behavioral:sim_1:Functional:mips_tb} -tclbatch cmd.tcl -
    log " + query_name)
58          with open(query_name, "r") as f:
59              query = f.read()
60          line_cnt = 0
61          master_iter = iter(master)
62          for line_query in query.splitlines()[USELESS_INFO_LEN:]:
63              line_master = next(master_iter)
64              line_cnt += 1
65              while "$ 0" in line_master:
66                  line_master = next(master_iter)
67              if line_master == "" or line_master is None:
68                  print(f"Test case {i} is accepted!")
69                  break
70              elif line_query is None:
71                  print(f"Test case {i} failed at line {line_cnt}!")
72                  print("Your outputs are fewer than expected.")
73                  AK = False
74                  break
75              elif line_master != line_query:
76                  print(f"Test case {i} failed at line {line_cnt}!")
77                  print(f"Expected answer is {line_master}.")
78                  print(f"Your answer is {line_query}.")
79                  AK = False
80                  break
81          if PAUSE_AT_FAIL and not AK:
82              os.system("pause")
83      if AK:
84          print("\nCongratulations!\nTest cases are all killed!")
```

三、 思考题

(一) 流水线冒险

1. **在采用本节所述的控制冒险处理方式下，PC 的值应当如何被更新？请从数据通路和控制信号两方面进行说明。**

数据通路方面，PC 的新值由 NPC 提供，NPC 接收 PC 的旧值、b 类指令立即数、j 类指令立即数、r 类指令跳转地址并运算得到 PC 可能的新值。

控制信号方面，有 2 位控制信号和 1 位跳转信号来选择并确认以 PC+4、b 类指令跳转地址、j 类指令跳转地址、r 类指令跳转地址作为 PC 的新值。

2. **对于 jal 等需要将指令地址写入寄存器的指令，为什么需要回写 PC+8？**

Due to jump delay slot, instruction at PC+4 always gets executed before jal jumps to label, so we need to return to PC+8.

(二) 数据冒险的分析

1. **为什么所有的供给者都是存储了已执行的指令所产生的数据的流水级寄存器，而不是由 ALU 或者 DM 等部件来提供数据？**

对于类似以下运算指令序列，在 ALU 执行 add 指令时需要 ALU 已计算出的$t0 的新值。由于 ALU 是纯组合逻辑部件，如果 ALU 的输入端口直接从输出端口转发，组合逻辑中的输入输出会振荡因而是不确定的。

```
1   addi    $t0, $t0, 1
2   add     $t0, $t0, $t0
```

对于含 load 类或 b 类指令的序列，如果直接转发 DM 或 ALU 的输出，功能虽然正确，但 CPU 时钟频率大幅度降低，因为关键路径变长，总延迟增加。

(三) AT 法处理流水线数据冒险

1. **如果不采用已经转发过的数据，而采用上一级中的原始数据，会出现怎样的问题？试列举指令序列说明这个问题。**

数据可能出错。比如 Store 类指令需要 ALU 计算出的地址，而 ALU 的输入值可能是从其它地方转发的。如果直接使用 GRF 的原始数据，就会出错。

2. 我们为什么要对 GPR 采用内部转发机制？如果不采用内部转发机制，我们要怎样才能解决这种情况下的转发需求呢？

因为如果当 W 级写 GPR 而 D 级读 GPR 时，如果没有转发，则虽然 GPR 被更新了，但旧值进入了 E 级。若不用内部转发，可以设置在时钟上升沿写 GPR、下降沿读 GPR。

3. 为什么 0 号寄存器需要特殊处理？

因为 0 号寄存器不可写入，值始终为 0，所以若要转发的是 0 号寄存器的值，需始终为 0。

4. 什么是“最新产生的数据”？

即最近那级流水线寄存器中的相应数据。

5. 在 AT 法判断转发条件的时候，只提到了“供给者需求者的 A 相同，且不为 0”。但 CPU 在判断能否写入 GRF 的时候，是需要 we 信号的。为何在 AT 法中不需要特判 we 呢？为了用且仅用 A 和 T 完成转发，在翻译出 A 的时候，要结合 we 做什么操作呢？

事实上，we 信号为 0 的指令只有 b 类或 j 类指令。在控制器中，这些指令的 Anew 信息被设为 0。又因为在用 AT 法控制转发时，需判断 Ause 或 Anew 不为 0，所以可以用且仅用 A 和 T 完成转发，而无需特判 we。