浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：官泽隆

学　　院：计算机科学与技术学院

系：

专　　业：计算机科学与技术

学　　号：3180103008

指导教师：翁恺

2020 年 11 月 14 日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目 **PCPU with stall**

学生姓名：官泽隆 专业：计算机科学与技术 学号：3180103008

同组学生姓名：/ 指导老师：翁恺

实验地点：曹光彪二期-301 实验日期：2020 年 11 月 14 日

一、 实验目的和要求

Objective:

1. PCPU in practice

Task:

1. Implement a *Pipelined CPU* running basic MIPS ISA
2. In the VGABased-Debugger project, replace module *mips* with your own PCPU
3. Validate the design of your own PCPU on SWORD Board

二、实验内容和原理

原理：

Within the same instruction set, different instructions has different complexities. For example, the *lw* instruction has the longest data path and will be the bottleneck of clock period. The key to this problem is to further decompose one instruction into multiple “phases”, each of which takes one cc. In this way, clock frequency can be improved, and “complex” instructions can be arranged more phases, minimizing the impact to clock period.

MCPU: A *Multi-Cycle CPU* takes variable clock cycles to execute one instruction.

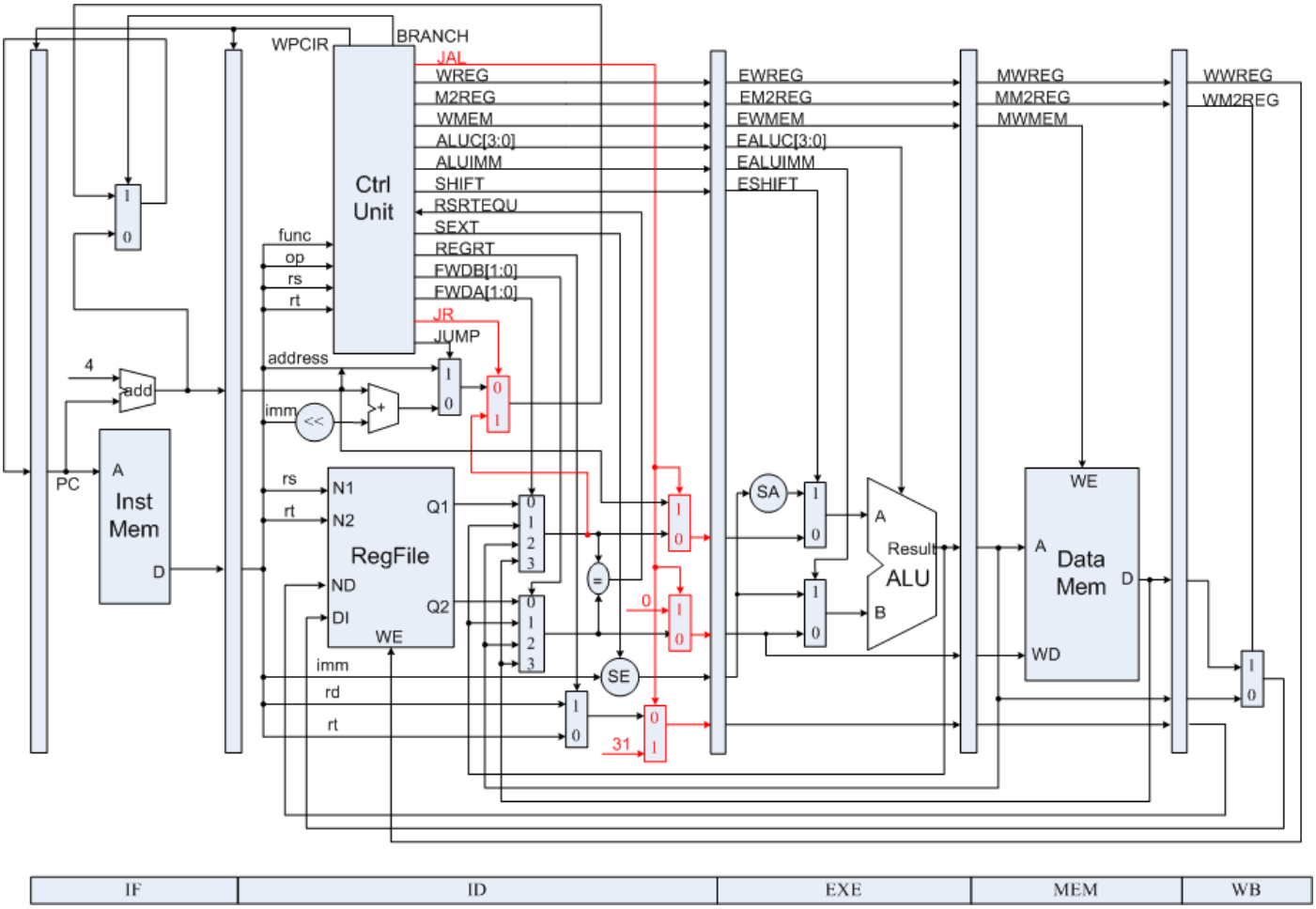
PCPU exploits more parallelism between instructions, allowing larger throughput of instruction or reducing CPI.

1. Overall Approach

We are to implement a *Pipelined CPU* that is functionally-equivalent to previous SCPU. Instructions required are as below.

```
enum Opr_T
{
    dst, // rd, rs, rt
    tsi, // rt, rs, imm
    dta, // rd, rt, sa
    S,   // rs
    tob, // rt, offset(base)
    ti,  // rt, imm
    sto, // rs, rt, offset
    T    //target
};       // oprand contained in one instruction
unordered_map<string, Opr_T> op_opr{
    {"add", dst},{"sub", dst},{"and", dst},{"or", dst}, {"slt", dst},
    {"sll", dta},{"srl", dta},
    {"jr", S}, // R-type
    {"addi", tsi}, {"ori", tsi}, {"slti", tsi},
    {"lw", tob}, {"sw", tob},
    {"lui", ti},
    {"beq", sto},{"bne", sto}, // I-type
    {"j", T}, {"jal", T}, // J-type
};
```

设计基于这张最终效果图：



diff：没有 forwarding 逻辑；流水传递 32 位指令，以满足 lui 指令在 wb 阶段所需数据；由于调试器的存在，级间寄存器的 CE 和 rst 信号变复杂

stall 的检测采用的是简单粗暴的这种形式：

```
wire Q1_hazard = rs && ((ex_WREG & ex_nd == rs) | (mem_WREG & mem_nd == rs));
wire Q2_hazard = rt && ((ex_WREG & ex_nd == rt) | (mem_WREG & mem_nd == rt));
assign stall = Q1_hazard | Q2_hazard;
```

这比实际需求要更严格，比如 I 型指令只 care rs 读出的值。

具体的分情况判断在 forwarding 实验一定会完成（判断转发来源）

跳转在 ID 解析完毕，如果有数据竞争会引起 stall.

有长度为 1 个指令的跳转延迟槽。

2. Building Data path

认识到一个事实：整个PCPU的设计是可以由几个**Stage**的模块确定的。 代码逻辑都在 *Stage.v 中，e.g. Module IfStage(...)，

Parse and Automate I/O of *Stage module

- Special Case: input clk, rst
- Port: 一般不区分，但部分Input port
    - 加 stage_ 前缀 to explicit src, allowing forwarding
    - 如不指明，默认本级，i.e. port i 连接 *_i 线网
- 需要注意Register的D端和Q端：对 reg 信号作为源引用时，是指Q端

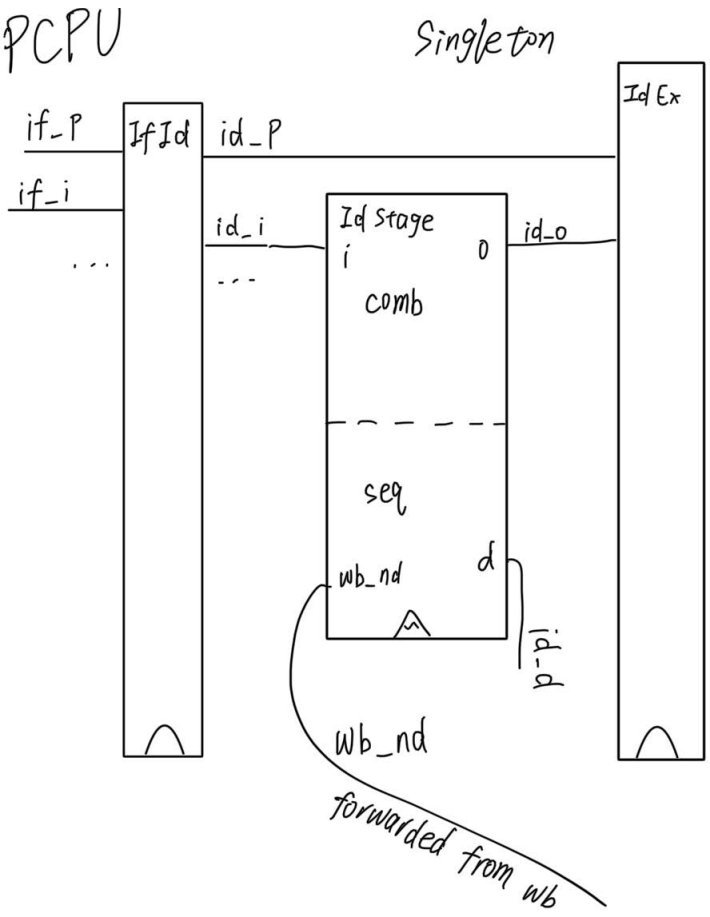
*Registers.v 生成： What should be pipelined

- 面向什么：e.g. IfId 保证 Id 的输入
- 考虑Mem/Wb：如果找不到request的信号，则认为该信号要从前级pipeline过来

Final instantiation: PCPU.v 生成

- I/O interface
- 根据 name，从线网里面拿 注意forwarding
- 已经在Stage代码中暴露了调试output端口
- Special Case: regfile debug; stall

代码基本自动化生成，实现在随附的3个python脚本中。 register_gen有对推断的注释。使用时，先 register_gen 然后PCPU_gen， 最后在生成的PCPU.v末尾，有需要手工完成的TODO



3. Control

可以从 SCPU 的控制器照搬代码。但还是有几个地方不同，起初成为我 PCPU 藏着的 bug.

由于 rs rt 采用了独立的比较器，因此条件分支的判断要写成

```
beq & RSRTEQU | bne & ~RSRTEQU
```

同时图中 BRANCH 的名字起得不是很好，因为从它控制的 PCMUX 来看，它其实也包括了跳转（J 型指令），因此要写

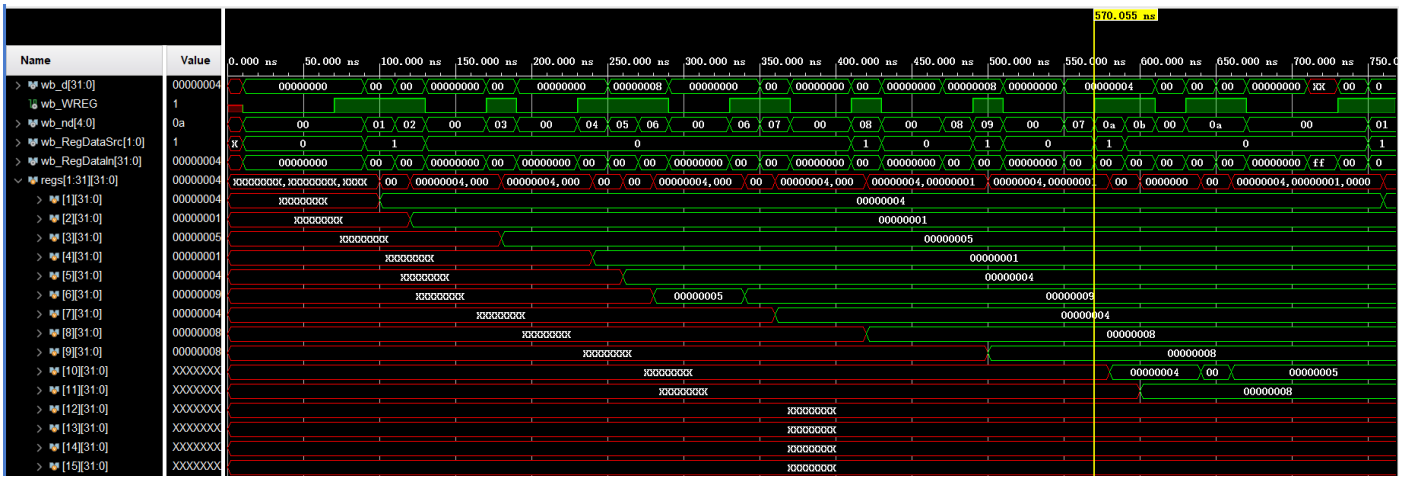
```
assign BRANCH = beq & RSRTEQU | bne & ~RSRTEQU | JR | JUMP;
```

然后 jal 是通过 alu 做 pc_p4+0，把 pc_p4 传到 aluout，因此 ALU_Ctr 做加法操作的情况还要考虑率 jal. 我认为这种设计是意图减少转发时源的数量。

三、实验过程和数据记录

1. 行为仿真——助教给的测试程序

对应 sim_mips.v



2. 行为仿真——黑盒测试 Fibonacci

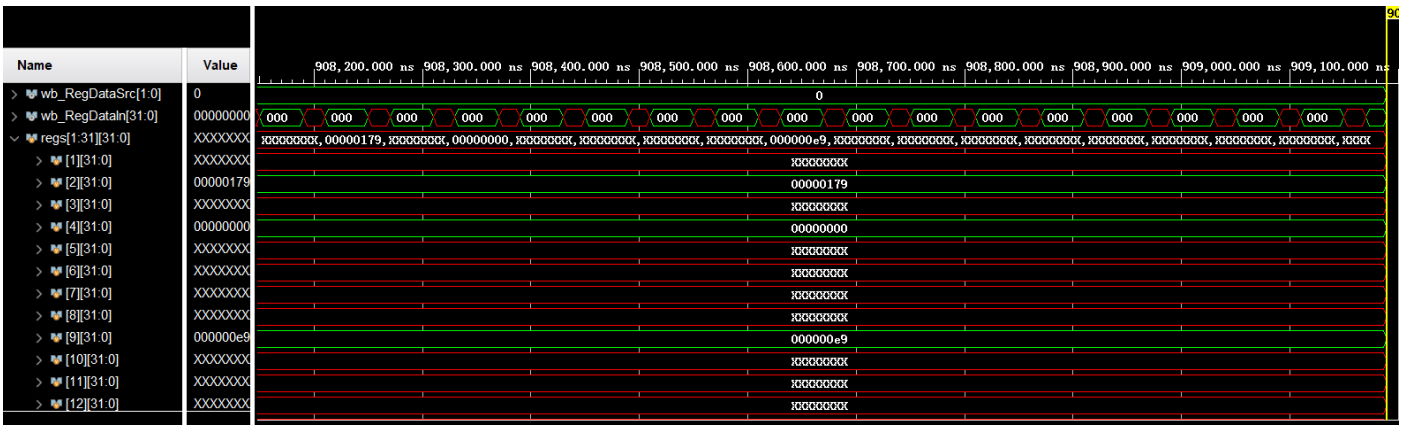
路径：这一测试的程序欲计算 Fib(14)，设置入口环境，设置出口死循环。观察结果。

测试用程序 fib.s 随附。

测试程序 fib.s 验证了最容易出错的一些功能：sw, lw, j, jal, jr, bne.

得出 Fib(14)=377 正确！

	0
1	X
2	377
3	X
4	0
5	X
6	X
7	X
8	X
9	233
10	X
11	X
12	X
13	X
14	X
15	X
16	X
17	X
18	X
19	X
20	X
21	X
22	X
23	X
24	X
25	X
26	X
27	X
28	X
29	2048
30	X
31	12



The Rule

The Fibonacci Sequence can be written as a "Rule" (see Sequences and Series).

First, the terms are numbered from 0 onwards like this:

n =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
x_n =	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	...

3. 下板