数字逻辑与处理器基础实验

流水线 MIPS 处理器设计实验报告

2019011008

无 92 刘雪枫

目录

一、实验名称与内容

实验名称：流水线 MIPS 处理器设计

实验内容：将春季学期实验四设计的多周期 MIPS 处理器改进为流水线结构，并利用此处理器完成背包算法。

二、处理器设计

（一）实现的指令集

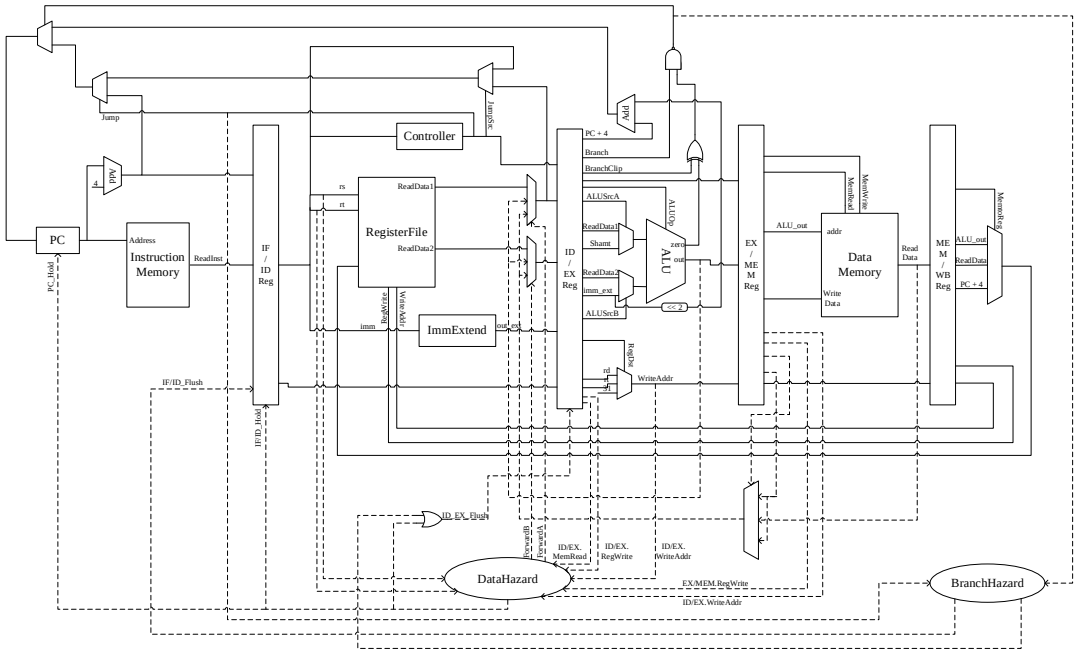
本次实验实现的指令集为：

1. R 型算术指令：add、addu、sub、subu、and、or、xor、nor、slt、sltu、sll、、srl、sra；
2. I 型算术指令：lui、addi、addiu、andi、sltiu；
3. 存取指令：lw、sw；
4. 分支指令：beq、bne、blez、bgtz、bltz；
5. 跳转指令：j、jal、jr、jalr。

上述指令的具体格式与各个指令的作用参见附录一与附录二。

（二）数据通路

数据通路图如下：



（三）处理器模块

根据上述指令集，我设计了如下模块：

1. PipelineCPU：总连线模块；
2. Controller：控制信号模块；
3. ALU：算术逻辑单元模块；
4. RegFile：寄存器堆模块；
5. DataMem：数据存储器模块；
6. InstMem：指令存储器模块；
7. PC：程序计数器模块；
8. ImmExtend：立即数扩展模块；
9. IF_ID_Reg、ID_EX_Reg、EX_MEM_Reg、MEM_WB_Reg：流水线寄存器模块；
10. DataHazard：数据冒险模块；
11. BranchAndJumpHazard：分支冒险模块；

12. TestBench：Test bench 模块

（四）工作过程

I. 工作阶段

IF 阶段负责程序计数器的更新以及取指令，以 PC 寄存器的输出为指令地址得到指令，并储存在 IF/ID 寄存器中。

ID 阶段负责寄存器堆的读取和立即数扩展，并进行 j、jr、jal、jalr 跳转指令的判断。本阶段将寄存器堆中读取的寄存器数据以及立即数扩展结果存入到 ID/EX 寄存器中。

EX 阶段进行 ALU 运算，以及将要写入的寄存器地址 WriteAddr 的选择。

MEM 阶段进行数据存储器的读取或写入。

EX 阶段选择写入寄存器堆的数据，并将数据写入寄存器堆。

II. 分支与跳转

在 ID 阶段进行 j、jr、jal、jalr 跳转指令的跳转；在 EX 阶段进行 beq、bne、blez、bgtz、bltz 分支指令的跳转。由于分支指令的判断晚于跳转指令一个阶段，因此为了避免分支指令后面紧跟的跳转指令产生冒险，本处理器采取分支指令优先的方式，即在电路逻辑上先判断分支指令是否跳转，再判断跳转指令是否跳转。

对于分支指令，使用两个控制信号 Branch 和 BranchClip 进行控制。Branch 控制信号用来控制本指令是否是分支指令。而分支指令是否跳转还取决于 ALU 运算的结果。由于分支指令是否跳转取决于操作数比较大小的结果，因此比较结果只可能是 0 或 1，故根据 ALU 输出的 zero 信号进行判断。其中，beq 和 bne 指令采用两个操作数之差是否为零进行判断，blez 和 bgtz 采用大于运算进行比较，bltz 采用小于运算进行比较。这样，5 个分支指令有些指令是当 zero 为 0 时跳转，有些指令是当 zero 为 1 时跳转。因此引入 BranchClip 信号，代表是否对 zero 信号取反，从而统一为当信号为 1 时跳转。这样只需要 BranchClip 对 zero 做异或运算即可。这样，分支指令跳转的充要条件是：`ID_EX_Reg.Branch &&`

`(ALU_zero ^ ID_EX_Reg.BranchClip)`。鉴于后续书写的简便考虑，代码中记录了不进行跳转的信号：`no_branch = (ID_EX_Reg.Branch == 0 || !(ALU_zero ^ ID_EX_Reg.BranchClip))`，即等价于`ID_EX_Reg.Branch`与`(ALU_zero ^ ID_EX_Reg.BranchClip)`进行与非运算。

对于跳转指令，使用了`Jump`和`JumpSrc`两个控制信号。`Jump`信号用于表示当前指令是否是跳转指令，而`JumpSrc`信号则控制跳转的目标地址是来自于立即数（j、jal）还是寄存器（jr、jalr）。

根据之前所述，分支优先于跳转。因此，最终的逻辑为，当`no_branch`为 0 时，即进行分支指令的跳转，否则当`Jump`为 1 时，进行跳转指令的跳转，否则不进行跳转。

关键代码为：

```
assign J_out = Jump == 0 ? (PC_o + 4) :
JumpSrc == 0 ? {PC_Plus_4[31:28], rs, rt, rd, Shamt, Funct,
2'b00
} : ReadData1Actual;
assign no_branch = (ID_EX_Reg.Branch == 0 || !(ALU_zero ^ ID
_EX_Reg.BranchClip));
assign PC_i = no_branch ? J_out :
ID_EX_Reg.PC_Plus_4 + (ID_EX_Reg.imm_ext << 2);
```

其中，`PC_i`为 PC 模块的输入。

III. 冒险

i) 数据冒险

关于寄存器值的冒险，注意到，由于最早用到寄存器值的阶段是 jr 指令，jr 指令在 ID 阶段就要获取寄存器的值，因此对于算术指令，因此对于数据冒险，需要在 ALU 刚刚计算出结果时就要进行转发，并且需要转发到 ID 阶段，而不能在 ALU 计算结果写入 EX/MEM 寄存器之后将数据转发到 ALU 运算的输入，因为这会导致 jr 指令可能无法得到正确的值。因此将转发提前到 ALU 计算结果后并转发到 ID 阶段是必要的。因此转发的逻辑（关键的转发代码）为：

```
assign ForwardA =
                (ID_EX_RegWrite && ID_EX_WriteAddr != 0 && I
D_EX_WriteAddr == rs) ? 2'b01 :
```

```
                (EX_MEM_RegWrite && EX_MEM_WriteAddr != 0 &&
 EX_MEM_WriteAddr == rs) ? 2'b10 :
                2'b00;

assign ForwardB =
                (ID_EX_RegWrite && ID_EX_WriteAddr != 0 && I
D_EX_WriteAddr == rt) ? 2'b01 :
                (EX_MEM_RegWrite && EX_MEM_WriteAddr != 0 &&
 EX_MEM_WriteAddr == rt) ? 2'b10 :
                2'b00;
```

对于 Load/Use 冒险，采取阻塞一个周期的方式。使用 `LW_Stall` 信号代表需要阻塞一个周期，关键代码为：

```
assign LW_Stall =
                ID_EX_MemRead && (ID_EX_WriteAddr != 0) && (
ID_EX_WriteAddr == rs || ID_EX_WriteAddr == rt);
```

ii) 控制冒险

当需要进行跳转时，会产生控制冒险，需要擦除掉已经执行的指令。由于跳转指令需要擦除一条指令，而分支指令需要擦除两条指令，因此关键代码为：

```
assign IF_ID_Flush = Jump || !no_branch;
assign ID_EX_Flush = !no_branch;
```

iii) 阻塞与擦除

综合上述冒险，当产生阻塞一个周期的请求时，需要擦除 ID/EX 阶段的指令，并让 PC 寄存器和 IF/ID 寄存器不更新。分别使用 `PC_Hold` 和 `IF_ID_Hold` 信号来使其不更新。设控制冒险中要求 ID/EX 寄存器擦除的信号为 `Branch_ID_EX_Flush`，而最终的擦除信号为 `ID_EX_Flush`，则最终的代码为：

```
assign PC_Hold = LW_Stall;
assign IF_ID_Hold = LW_Stall;
assign ID_EX_Flush = Branch_ID_EX_Flush || LW_Stall;
```

此外，`IF_ID_Flush` 信号直接连接到控制冒险的 `IF_ID_Flush` 信号即可。

三、综合与实现情况

（一）时序性能

本次的管脚约束中，将时钟周期设置为 10.00ns，观察综合后的时序裕量为-1.335ns：

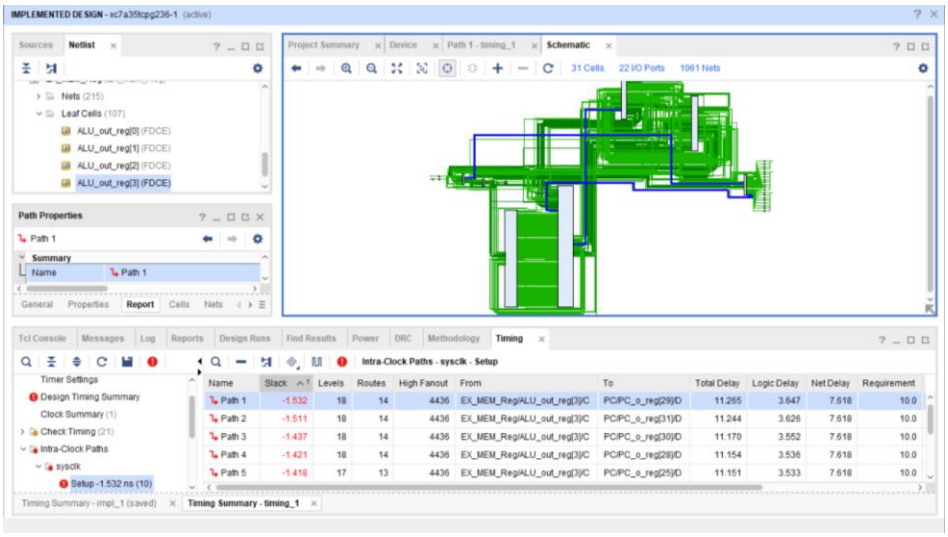
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1.335 ns	Worst Hold Slack (WHS):	0.134 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	-25.941 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	29	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	35354	Total Number of Endpoints:	35354	Total Number of Endpoints:	17862

综合后的时序裕量为-1.532ns：

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1.532 ns	Worst Hold Slack (WHS):	0.046 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	-507.139 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	1479	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	35394	Total Number of Endpoints:	35394	Total Number of Endpoints:	17882

时序裕量为负值，说明时钟周期设置的偏短。预计时钟周期最短应为 11.54ns，因此频率最高大致为 86.7MHz。

观察关键路径：



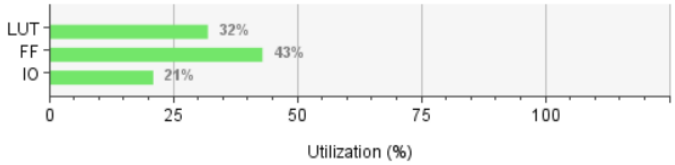
可以看到关键路径是包含数据寄存器 DataMem 和冒险检测单元的路径。因此处理器的频率主要被数据存储器和冒险检测单元所限制。

（二）逻辑资源占用情况

处理器的逻辑资源占用情况如下：

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
PipelineCPU	6577	17881	2374	1053	6817	6577	144	22	1
DataMem (DataMem)	5030	16424	2176	1052	6301	5030	0	0	0
EX_MEM_Reg (EX_ME...	0	106	0	0	46	0	0	0	0
ID_EX_Reg (ID_EX_R...	84	158	1	0	101	84	0	0	0
IF_ID_Reg (IF_ID_Reg)	263	65	0	0	149	263	0	0	0
ImmExtend (ImmExtend)	1	0	0	0	1	1	0	0	0
MEM_WB_Reg (MEM_...	0	104	0	0	38	0	0	0	0
PC (PC)	579	32	5	1	215	579	19	0	0
RegFile (RegFile)	639	992	192	0	447	639	0	0	0

Resource	Utilization	Available	Utilization %
LUT	6577	20800	31.62
FF	17881	41600	42.98
IO	22	106	20.75



四、仿真验证

本次实验经过了多次汇编程序对冒险进行测试，均通过测试。测试所用汇编程序参见 asm_program 中的 hazard_test 文件夹。

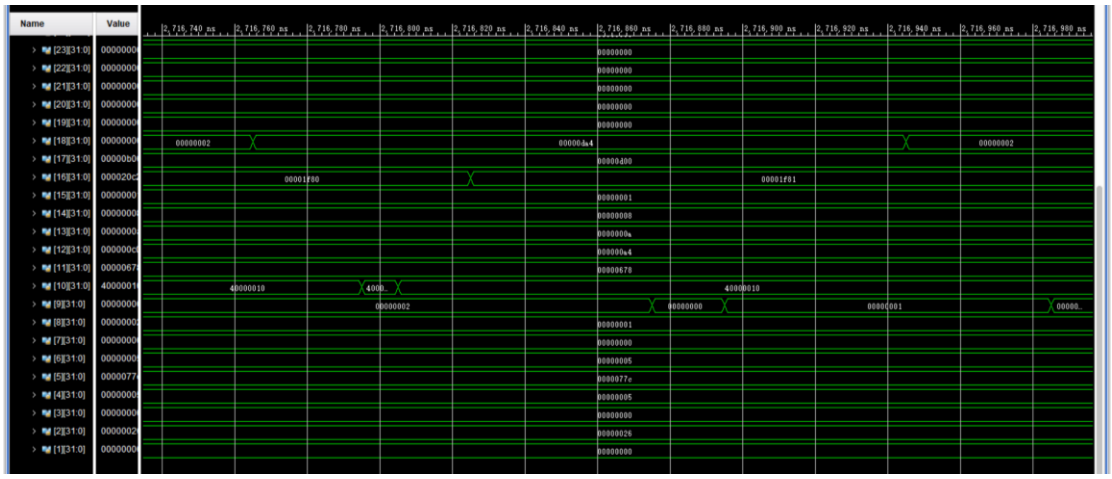
最终，用此 CPU 解决背包问题。背包问题的汇编程序参见 asm_program 中的 pack.asm。

本次背包的测试数据为：

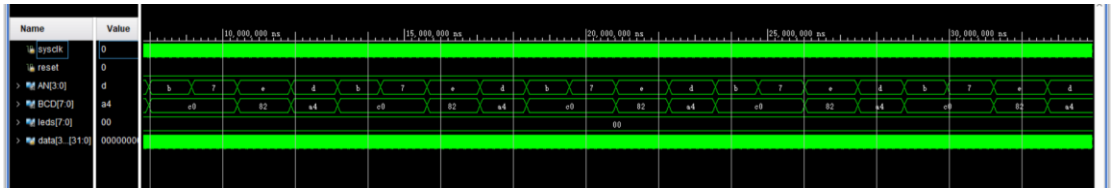
序号	1	2	3	4	5
重量	2	1	3	2	1
价值	12	10	20	15	8

答案应为 38，转为十六进制是 26。

进行仿真结果如下：



可以看到，最终寄存器$v0 的值确实是十六进制的 26。继续仿真，观察七段数码管（即 AN 与 BCD）的值：



可以看到，最终 AN 与 BCD 的值若显示在七段数码管上也确实是 0026，因此仿真结果正确。

五、硬件调试情况

将设计代码使用 Vivado 2020.2 烧录到 FPGA 上之后运行，七段数码管上很快便出现结果“0026”，因此硬件调试成功。

六、程序清单

本次报告的目录结构如下：

```
|  实验报告.pdf
|
├──asm_program
|   |  pack.asm
|   |
|   └──hazard_test
|          branch_hazard.asm
|          data_hazard.asm
|          jump_hazard.asm
|          no_hazard.asm
|          sp.asm
|
└──src
    ├──constrs
    |      PipelineCPU.xdc
    |
    ├──sim
    |      TestBench.v
    |
    └──source
           ALU.v
           BranchAndJumpHazard.v
           Controller.v
           DataHazard.v
           DataMem.v
           EX_MEM_Reg.v
           ID_EX_Reg.v
           IF_ID_Reg.v
           ImmExtend.v
           InstMem.v
           MEM_WB_Reg.v
           PC.v
           PipelineCPU.v
           RegFile.v
```

其中，各个目录及文件的内容如下：

1. asm_program：存放汇编程序；
    a) pack.asm：背包问题汇编语言程序
    b) 测试冒险的汇编语言程序
2. src：存放源代码
    a) constrs：存放管脚约束文件
    b) sim：存放 test bench 文件
    c) source：存放处理器源代码

上述文件均已经与本报告一同上交。

七、心得体会

本次实验总体上看还是比较顺利的。

在这次实验中，我基本上完全独立完成了本次实验。在编写代码的过程中，我也对课上所讲的流水线有了更深的理解。在编写代码的过程中，我也遇到了一些困难，例如 jr 指令对数据的要求时间比较早，要求提前进行转发，这个转发的细节耗费了我一些精力来思考；此外分支和跳转指令由于不在同一阶段进行判断，因此它们的优先级先后问题也是我需要特别思考的。此外，Load/Use 冒险的阻塞在之前的理论课上没有特别思考过如何实现，本次实现阻塞时也耗费了一些时间来思考。

另外，这次试验也有一些不足之处。首先，处理器的频率较低，没有进行更深入的优化；第二，指令存储器没有使用 IP Core，因此在切换汇编程序等操作上耗费了一些时间；第三，一些细节上也编写的稍显繁琐，没有进行优化，等等。

总而言之，本次实验还是让我有很多收获的。最后感谢老师在这一学期的倾情讲授，感谢助教的悉心指导，没有老师和助教的帮助，我也难以得到这么多的收获。

附录

附录一 指令格式表

指令格式表						
	OpCode [5:0]	Rs [4:0]	Rt [4:0]	Rd [4:0]	Shamt [4:0]	Funct [5:0]
	OpCode [5:0]	Rs [4:0]	Rt [4:0]	imm/offset [15:0]		
	OpCode [5:0]	target [25:0]				
R 型算术指令						
add	0	rs	rt	rd	0	0x20
addu	0	rs	rt	rd	1	0x21
sub	0	rs	rt	rd	2	0x22
subu	0	rs	rt	rd	3	0x23
and	0	rs	rt	rd	0	0x24
or	0	rs	rt	rd	1	0x25
xor	0	rs	rt	rd	2	0x26
nor	0	rs	rt	rd	3	0x27
slt	0	rs	rt	rd	0	0x2a
sltu	0	rs	rt	rd	0	0x2b
sll	0	0	rt	rd	shamt	0
srl	0	0	rt	rd	shamt	0x02
sra	0	0	rt	rd	shamt	0x03
I 型算术指令						
lui	0x0f	0	rt	imm		
addi	0x08	rs	rt	imm		
addiu	0x09	rs	rt	imm		
andi	0x0c	rs	rt	imm		
sltiu	0x0b	rs	rt	imm		
存取指令						
lw	0x23	rs	rt	offset		
sw	0x2b	rs	rt	offset		
分支指令						
beq	0x04	rs	rt	offset		
bne	0x05	rs	rt	offset		
blez	0x06	rs	0	offset		
bgtz	0x07	rs	0	offset		
bltz	0x01	rs	0	offset		
跳转指令						
j	0x02	target				
jal	0x03	target				
jr	0	rs	0			0x08
jalr	0	rs	0	rd	0	0x09

附录二 指令说明表

指令说明表	
指令	说明
R 型算术指令	
add	R[rd] = R[rs] + R[rt]
addu	R[rd] = R[rs] + R[rt]
sub	R[rd] = R[rs] - R[rt]
subu	R[rd] = R[rs] - R[rt]
and	R[rd] = R[rs] & R[rt]
or	R[rd] = R[rs] | R[rt]
xor	R[rd] = R[rs] ^ R[rt]
nor	R[rd] = ~(R[rs] | R[rt])
slt	R[rd] = (R[rs]± < R[rt]±) ? 1 : 0
sltu	R[rd] = (R[rs]ø < R[rt]ø) ? 1 : 0
sll	R[rd] = R[rt] << shamt
srl	R[rd] = R[rt]ø >> shamt
sra	R[rd] = R[rt]± >> shamt
I 型算术指令	
lui	R[rt] = {imm, 16'b0}
addi	R[rt] = R[rs] + SignExtImm
addiu	R[rt] = R[rs] + SignExtImm
andi	R[rt] = R[rs] & ZeroExtImm
sltiu	R[rt] = (R[rs]ø < SignExtImmø) ? 1 : 0
存取指令	
lw	R[rt] = M[R[rs] + SignExtImm]
sw	M[R[rs] + SignExtImm] = R[rt]
分支指令	
beq	if (R[rs] == R[rt]) PC += 4 + (offset << 2)
bne	if (R[rs] != R[rt]) PC += 4 + (offset << 3)
blez	if (R[rs] <= 0) PC += 4 + (offset << 4)
bgtz	if (R[rs] > 0) PC += 4 + (offset << 5)
bltz	if (R[rs] < 0) PC += 4 + (offset << 6)
跳转指令	
j	PC = {PC[31:28], target << 2}
jal	R[31] = PC + 4; PC = {PC[31:28], target << 2}
jr	PC = R[rs]
jalr	R[31] = PC + 4; PC = R[rs]