Multi-Cycle MIPS Processor 实验报告

16307130194 陈中钰

16 级 计算机科学技术学院

Contents

1 总体状况

1.1 多周期 MIPS 处理器

单周期处理器在一个周期内执行一条完整的指令，结构易于解释且控制单元简单，不需要其他非体系结构状态，但是在任一时刻，处理器中只有部分元件处于使用中的状态，而大部分的元件在静止中，加上时钟周期是由最慢的指令决定的，因此在执行其他指令的时候，还存在一段时间是所有元件都在静止的，这极大地限制了处理器的效率。此外单周期处理器使用了 3 个加法器、指令存储器和数据存储器分离，过于冗余。

因此有了多周期处理器，通过往处理器中**添加寄存器**，用于储存中间量，以此把处理器划分为多个部分，同时把各个指令分解为多个短步骤，指令运行会更加高效、紧凑。同时还复用**单个 ALU**，**合并指令、数据存储器**。此外，还需要**设计 FSM** 来控制每条指令各部分的运行顺序。

1.2 指令集

1. 书中结构已包含的指令：add，sub，and，or，slt，sw，lw，beq
2. 书中添加的指令：addi，j
3. **要求以外添加指令：bne，nop，andi，ori，slti，xor，xori，nor，sll，srl，sra，jal，lui**

（以上指令的实现完全按照 MIPS 指令集文档中的格式，故不再附上指令格式要求。）

1.3 规格

regfile：32bit*32

RAM：32bit*64

1.4 实现功能（均已展示）（会在仿真中对部分功能进行验证）

1. 可以屏蔽时钟，实现**暂停**功能（stop）
2. 可在暂停的情况下，实现**单条指令运行**功能（next）
3. 可以使**各个寄存器、regfile、memory 同步归零**（reset）
4. 可以通过 16 个 LED 灯、8 个 7 段数码，**直观地查看**电路中各个模块**所有的值**

1.5 总体结构

1. 代码结构

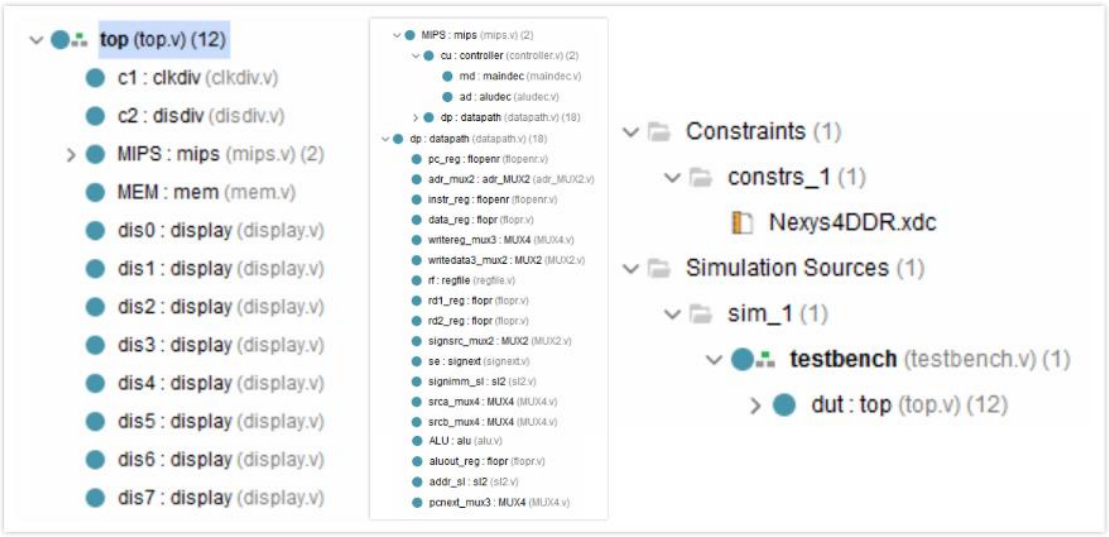


Figure 1 总体代码结构

其中，top 为顶层文件，分为 MIPS、MEMORY、显示模块三个部分。而 testbench 为处理器的仿真文件。

2. 硬件结构

由于额外添加了 j，addi，**bne，nop，andi，ori，slti，xor，xori，nor，sll，srl，sra，jal，lui 共 15 条指令，**在要求的基础上，增加、拓宽了一些控制信号，在后面会详细介绍。

3. FSM 结构

由于**多支持了 15 条指令**，因此在原来的基础上把 **state** 从 11 种增加到 **19 种**。在后面会详细介绍。

1.6 测试代码

位于 MIPS_multi_cycle_32bit\MIPS_ multi _cycle_32bit.srcs\test_files\ 文件夹中，含有 **memfile6（含有所有指令）、sumfile0（1~100 的求和）、sortfile0（插入排序）共 3 份测试代码**。每份测试代码有.dat、.txt 为后缀的两份同名文件。其中，**.dat 为十六进制代码**，用于输入单周期处理器中运行；而**.txt 文件为完整的代码说明**，含有汇编代码、代码描述、地址、十六进制码等内容，方便检验代码、处理器的正确性。

此外，由于合并了指令、数据存储器，因此原来的更改了低位数据内容的指令会覆盖指令代码，于是要**把数据存取的位置改到程序指令的后面**，以免影响指令的存储。

```
1  #       Assembly            Description          Address     Machine     Binary
2  main    addi $t0, $0, 80    # t0=160=0xa0        0           200600a0
3          addi $t1, $0, 8     # t1=8               4           20010008
4
5          addi $8, $0, 8      # $8=8               8           20080008
6          sw   $8, 0($t0)     # [160] = 8          c           acc80000    101011 00110
7          addi $8, $0, 4      # $8=4               10          20080004
8          sw   $8, 4($t0)     # [164] = 4          14          acc80004
9          addi $8, $0, 2      # $8=2               18          20080002
10         sw   $8, 8($t0)     # [168] = 2          1c          acc80008
11         addi $8, $0, 7      # $8=7               20          20080007
12         sw   $8, 12($t0)    # [172] = 7          24          acc8000c
13         addi $8, $0, 3      # $8=3               28          20080003
14         sw   $8, 16($t0)    # [176] = 3          2c          acc80010
```

Figure 2 sortfile0.txt 部分示例

```
200600a0
20010008
20080008
acc80000
20080004
acc80004
20080002
acc80008
```

Figure 3 sortfile.dat 部分示例

1.7 文件读入地址修改（以下文件均可在 test_files 文件夹中可以找到）

在 mem 模块，需把如下两行地址修改为待执行代码文件的实际地址。

```
//initialize instr memory
initial
$readmemh("C:/Users/ECHOES/Desktop/multi_cycle_pj/MIPS_

//read data
assign readdata=RAM[memadr[31:2]];

//reset instr memory
always@(posedge clk)
if(reset)
    $readmemh("C:/Users/ECHOES/Desktop/multi_cycle_pj/M
```

在 regfile 模块，需要把如下地址修改为 emptyreg.dat（在 test_files 文件夹中）的实际地址。

```
always@(posedge clk)
if(reset)
    $readmemh("C:/Users/ECHOES/Desktop/multi_cy
```

2 显示实现

（代码实现请看工程文件，在此不展示）

2.1disdiv

这是用于七段数码管扫描显示用的分频模块。在每次 input clk 的上升沿时，对 reg 向量 q 进行+1 的操作，另外把 q[x]赋值给 output clk。其中，如果 x 越大则 output clk 的频率越小，而 x 越小则频率越大、越接近 input clk 的频率。

由于开发板在任意时刻，只能显示一个 7 段数码，故需要利用人的视觉残留，在适当的频率下使每个 7 段数码轮流显示其对应值，可以产生 8 个 7 段数码同时显示的效果。在这里取 x=17，能稳定显示数字。如果 x 偏大则频率过小，会看到 8 个 7 段数码轮流显示，如果 x 偏小则频率过高，会导致明暗不一、闪烁、部分不显示、显示错误数字等错觉。

2.2display

传入一个 4bit 的数 T。如**控制 s 为 1，则将 T 转化为十六进制（0~F）的 7 段数码显示**，并记录在 7bit 的 C 中；如 **s 为 0，则转化为 - 符号进行显示**。

2.3top（用于显示的部分）及约束文件

2.3.1 输入及调控

1. next-SW[15]：在暂停的情况下，上升沿时，运行到下一条指令；stop-SW[14]：为 1 时，屏蔽时钟，暂停。通过把输入 MEM 和 MIPS 的 clk 信号修饰为 next|((~stop)&clk)来实现。

2. lightsrc-SW[13]：由于 controller 的输出信号很重要，因此用 LED 灯输出。但信号宽度超过了 16，因此加入 1 位的信号来确定显示前半部分/后半部分的控制信号。
3. reset-SW[12]：为 1 时，寄存器、mem、regfile 同步归零
4. 输出模块选择 SW[9:6] + 输出值选择 SW[5:0]：控制 8 个 7 段数码管，按照对应数据所需位数输出对应的十六进制值，多余的位输出 - 符号。其他没有对应的，则输出 8 个 - 符号。
5. clk-CLK100MHZ：输入时钟，用分频模块产生其他所需频率的时钟

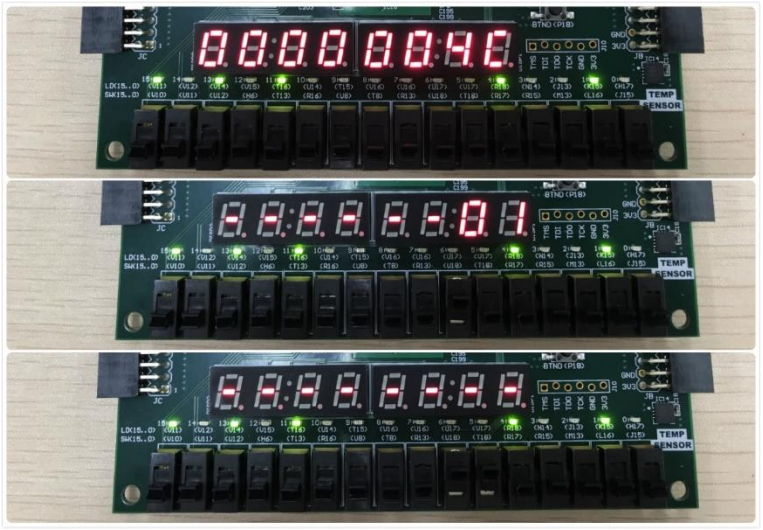


Figure 4 7 段数码管显示示例

2.3.2 输出及显示机制

1. discu[15:0]-LED[15:0]:由于 controller 的输出的调控信号很重要，且大部分位长为 1，所以用 LED 灯显示。根据 lightsrc 信号来决定输出前半/后半部分的信号。

lightsrc	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	alusrca		branch	iord	memtore	regdst		alusrcb		pcsrc		aluop			extend	extsrc
1	next	stop	lightsrc	reset	clk			pcen	alucontrol			nrwrite	memwrite		irwrite	regwrite

Figure 5 LED 显示对应关系

2. C[6:0]:在某一时刻需要输出的 1 个 7 段数码数字
3. AN[7:0]:指定 8 个中需要点亮的 7 段数字
4. 8 个 7 段数码管取值机制：把选择显示模块的 SW[9:6]、选择显示值的 SW[5:0]传入到 MIPS 中的各个模块、mem 模块，同时向上述模块各传入一对输出值 tx[31:0]、disopx[7:0]值，通过 SW 来选择要显示的数据，并加载到 tx 值中，而 disop 则

记录要显示的位数（0~8 位）。最后在 top 模块中，按照 SW 值，选取最终要显示的 tx、disopx 值，并传给 top 模块中的 T、disop。另外，SW 产生的多余的选项，则对 T、disop 传入 0 值，也就是显示 0 位数。**以上内容均通过嵌套的 case 语句来实现。（由于按照模块来划分，因此部分数值是有重复的、有分割的，虽然有些冗余，但是能方便对应某一模块来查看其全部输入输出值，易于 debug）**

5. **8 个 7 段数码管显示机制**：把 32bit 的 T 分开成 8 个十六进制数，分别用 display 模块进行转换，并记录到 C0~C7 中，多余的位会被转化为 – 符号。接着采用合适的频率，扫描显示 8 个数码管，最终会显示出对应的 8 个数字。

SW[9:6]	模块	SW[5:0]	值	十六进制位数（0~8 位）
0000	FSM	000000	state	2
		000001	nextstate	2
		else	0	0
0001	PC	000000	pc	8
		000001	pcnext	8
		000010	pcen	1
		000011	aluresult	8
		000100	aluout	8
		000101	pcjump	8
		000110	pcsrc	1
		else	0	0
0010	adr mux2	000000	pc[7:2]	2
		000001	aluout	8
		000010	iord	1
		000011	adr	8
		else	0	0
0011	instr reg	000000	irwrite	1
		000001	readdate	8
		000010	instr	8
		else	0	0
0100	data reg, op, funct	000000	readdate	8
		000001	data	8
		000010	op	2
		000011	funct	2
		else	0	0
0101	writereg mux3	000000	instr[20:16]	2
		000001	instr[15:11]	2
		000010	5’b11111	2
		000011	regdst	1
		000100	writereg	2
		else	0	0

0110	writedate3 mux2	000000	aluout	8
		000001	data	8
		000010	memtoreg	1
		000011	writedate3	8
		else	0	0
0111	reg	000000	instr[25:21]	2
		000001	instr[20:16]	2
		000010	writereg	2
		000011	writedata3	2
		000100	regwrite	1
		000101	regout1	8
		000110	regout2	8
		000111	regouta	8
		001000	regoutb	8
		001001	writedata	8
		else	0	0
1000	regfile	000000~ 011111	rf[SW[4:0]]	8
		else	0	0
1001	signext	000000	instr[15:0]	4
		000001	instr[10:6]	2
		000010	extsrc	1
		000011	extend	1
		000100	signimm	8
		000101	signimmsh	8
		else	0	0
1010	ALU	000000	pc	8
		000001	regouta	8
		000010	16	8
		000011	signimm	8
		000100	alusrca	1
		000101	srca	8
		000110	regoutb	8
		000111	4	8
		001000	signimmsh	8
		001001	alusrcb	1
		001010	srcb	8
		001011	alucontrol	1
		001100	aluresult	8
		001101	zero	1
		001110	aluout	8
		else	0	0
1011	addr sl	000000	instr[25:0]	8

		000001	addrsl	8
		000010	pc[31:28]	1
		000011	pcjump	1
		else	0	0
1100	mem	000000	memwrite	1
		000001	memadr	8
		000010	writedata	8
		000011	readdata	8
		else	0	0
1101	memfile	000000~ 111111	RAM[swt[5:0]]	8
else			0	0

Figure 6 开关取值对应关系

2.3.3 输入输出描述

通过上述输入输出，可以**直观地、简易地实现以上功能，并能按照模块分类，直观查看处理器中任意模块的所有值**。

3 模块实现

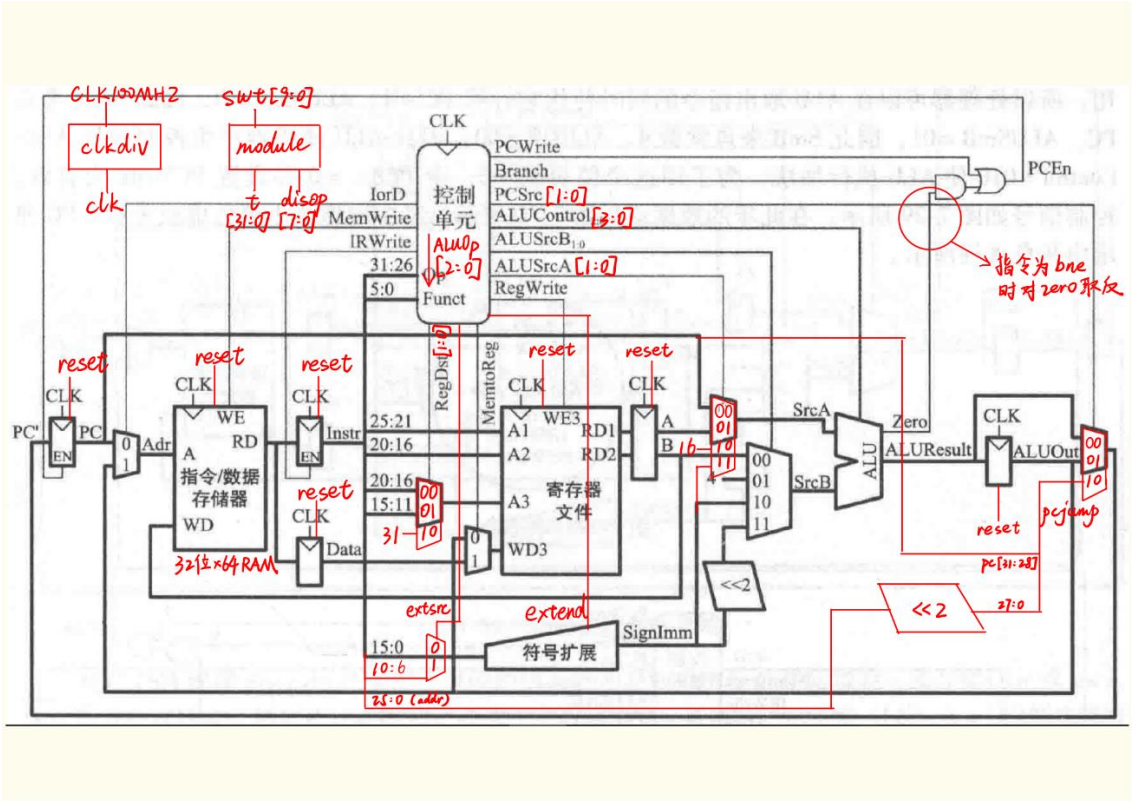


Figure 7 **处理器硬件设计（红色部分：修改了的硬件、增加/拓宽的变量）**
（左上角：分频模块、7 段数码取值模块设计）

（代码请看工程文件，在此不展示）

3.1clkdiv

时钟分频模块，实现原理与上文中的 disdiv 模块一样，但是频率不一样：**用于开发板查看指令运行时，为了能看的清楚，一般频率在 1Hz~1.5Hz，使用 q[26]；在仿真时，一般运行一次有 10ms，为了能在运行 10~20ms 内就能查看所有的指令运行情况，采用 q[0]**，也就是 100MHz。

3.2MIPS

MIPS 是处理器中最重要的模块，**由 controller、datapath 两个模块组成**。其中，datapath 含有处理器中的大部分硬件模块，并连接成数据通路；而 controller 含有 **main_decoder、alu_decoder**，以及 main_decoder 含有的 **FSM**，共同根据 instr 的内部顺序，来生成对应的控制信号，并调控 **datapath** 中的各个模块以及 mem 的运行。

3.2.1 controller

controller 解码 instr，而 FSM 控制每条指令内部的运行顺序并生成对应的控制信号。FSM 按照下图逻辑，通过 state register 更新 state，并通过 next state logic 来获得 next state，并输出当前 state 对应的控制信号，用于调控 datapath 和 mem 模块的运行。

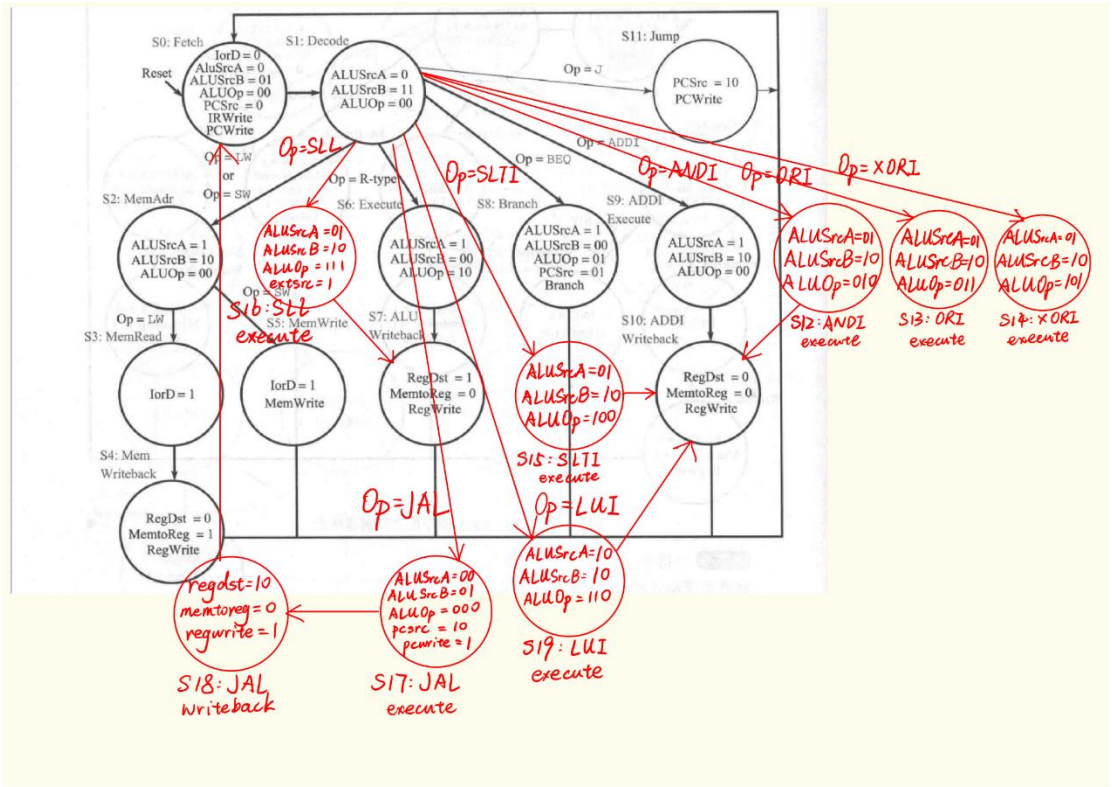


Figure 8 FSM 控制逻辑（**红色部分：增加了的状态**）

在书中设计的的基础上，做了以下**修改：**

1. 由于添加了 15 条指令，**FSM 额外增加了 8 个 state**，此外还有部分新指令的 state 可以与原来的 state 复用，就不再冗余地额外添加。
2. 添加的 jal 指令中有 GPR[31]<-PC+8 的操作，reg 需要取常数 31，于是**拓宽 regdst 到 2 位长**。在原来的 instr[20:16]、instr[15:11]两个选择上，增加了 31 的选择。

3. 添加的 sll、srl、sra 指令中需要用 instr[10:6]作为移位位数，且需要进行位拓展，故**添加 extsrc 信号**，用来选择向符号扩展模块输入 instr[15:0]或者 instr[10:6]。
4. 由于 andi、ori、xori 需要对立即数进行零拓展，而其他 R type、ls、sw 等指令采用符号拓展，因此**添加 extend 信号**，用以判断拓展方式。
5. 由于添加了 andi、ori、xori、slti、lui 等运算，因此算上 default，alu_decoder 共需要 7 种 aluop，故把 **aluop 拓展为 3 位**。
6. 由于添加了 or、slt、nor、xor、sll、srl、sra、lui 等运算，因此控制 ALU 选择运算方式的 **alucontrol 拓宽为 4 位**。
7. 由于有 beq 和 bne 两个指令，一个在 zero 为 1 时有效，另一个在 zero 为 0 时有效，因此**计算 pcen 时要根据 op 来判断取哪种**。
8. 由于加入了 j 指令，在获得 pcnext 的时候要增加 pcjump 这一个选项，于是**拓宽 pcsrc 到 2 位长**，并加入 pcjump 的选择。
9. 由于 lui 指令是左移 16 位，而被左移数 instr[15:0]作为 srcb 的选择，则需要在 srca 的选择中添加常数 16。且由于 sll、sra、srl 指令需要取 instr[20:16]寄存器的值，因此从中取出的值作为 srcb，但同时还需要移位数，就只能作为 srca，因此要把移位结果 signimm 多连一根线到 srca 的选择中。因此 srca 要多两个选择，因此 **alusrca 要拓宽到 2 位**。

3.2.2 datapath

datapath 中各个部分受到 controller 发来的控制信号控制着，并随着时钟上升沿，依次执行指令的一个个部分，并一条条指令地执行下去。

1. enable – resettable flop：PC、instr 寄存器。在上升沿时，若 reset 为 1，则加载 0，若为 0，则看 enable 信号，若为 1 则更新寄存器的值，否则保持不变。
2. resettable flop：MEM 输出数据、寄存器输出数据、ALUresult 寄存器。相当于 enable 信号总是为 1.
3. 符号扩展

   添加了 extend 信号，来判断对立即数采用零拓展（如 andi、ori、xori）或者符号拓展（如 lw、sw）。
4. 移位器

   把拓展为 32 位的 instr[15:0]左移 2 位，即 x4。beq、bne 指令都是指定向前/后跳的指令数目，因此需要把数目 x4 获得 signimmsh，并和(pc+4)相加，形成最终要跳到的 pc 位置 pcbranch 作为 pcnext 的选项之一。
5. regfile

   由 reg 数组实现 32 个 32bit 的 reg，并能通过 regwrite 控制是否写入数据（如 lw 则要写入），通过 regdst 选择要写入的寄存器（如 lui 则要选择常熟 31），通过 memtoreg 来选择要写入的数据（如一般的运算指令则要选择 aluout 作为写入的数据）。从寄存器中读出来的两个数据存储在寄存器中，等待下一个时钟上升沿时取用。此外，**在 clear 为 1 时，会同步读入 emptyreg.dat 文件，实现 regfile 的清零**。
6. wrmux(MUX4)

   选择 writereg，即要写入的 reg。由于 jal 指令中有 GPR[31]<-PC+8 的操作，所以**拓宽了 regdst，添加了 5’b11111 选择，并把原来的 MUX2 改为 MUX4**。

7. srcbmux(MUX4)
   通过 alusrcb 控制 ALU 的 srcb 输入的取值，由于 jal 指令**把 alusrcb 拓宽为 2 位，同时把 MUX2 改为 MUX4**。
8. ALU
   通过 alusrca 来控制取用的 srca 值、通过 alusrcb 来控制取用的 srcb（如 add，则要取寄存器文件读出来两个数值分别作为 srca、srcb）。通过 alucontrol 来控制 ALU 的运算方式，并输出计算结果（aluresult）和 ZF（zero）。
9. 为了获得 pcjump，还需要多**增加一个左移两位的移位模块**，对 j 指令 instr[25:0]处的地址 x4，并拼接上 pc 的前 4 位来生成 pcjump 值。
10. 通过 pcsrc 来选择 pcnext 取用的值（如为 j 指令，则取 pcjump 值）。

3.3mem

mem 用于储存、读取指令、数据，由 reg 型数组实现。传入 adr 作为地址，读出指令或数据，并根据 memwrite 信号决定是否要写入数据。**此外，当 clear 为 1 时，将会同步读入当前的指令文件，实现 mem 的还原。**

3.4top

top 为顶层模块，除了用于在开发板上输出数据的内容外，就是组织 MIPS、mem 的结构。MIPS 进行正常的处理器运行，并从 mem 中读取 instr 指令来执行，从 mem 中读取数据（如 lw 等）或者向 mem 写入数据（如 sw 等）。

4 仿真实现

仿真用于检验处理器是否能按照所执行的代码要求运行、能否实现相关功能，**检验时主要通过观察 pc、regfile、mem 等值来判断是否正确**。注意在仿真前，需要将 clkdiv 中的分频取 outclk=q[0]，使其能在短时间内运行完全部指令，便于观察；并将 mem 模块中待运行的代码地址改好。另外，**testbench.v 为整个处理器的仿真文件**。

4.1仿真 1：memfile6（含有所有指令）

memfile6 含有实现的所有指令，可以根据 pc 来一条条对应检验指令是否正确。在这里简单指出 sra、jal、sw 指令的正确性。

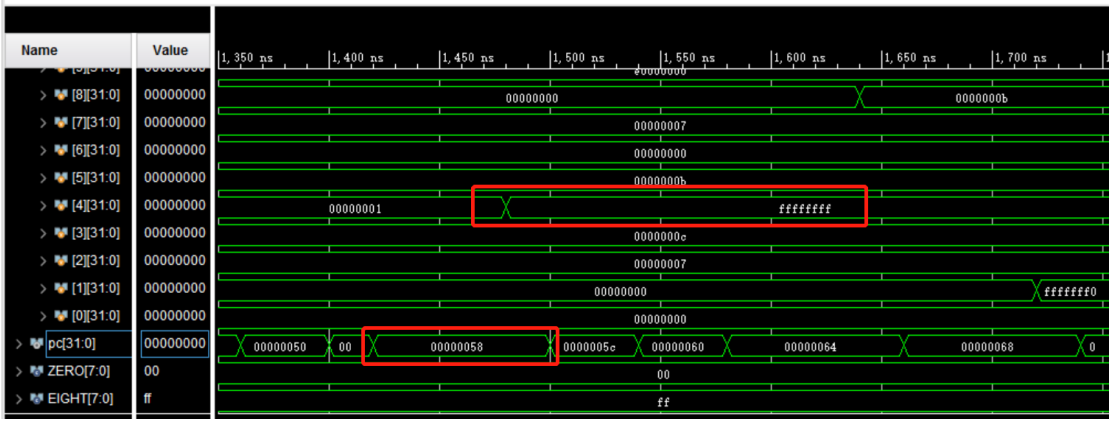


Figure 9 sra: $4 = e0000000 >> 29 = ffffffff

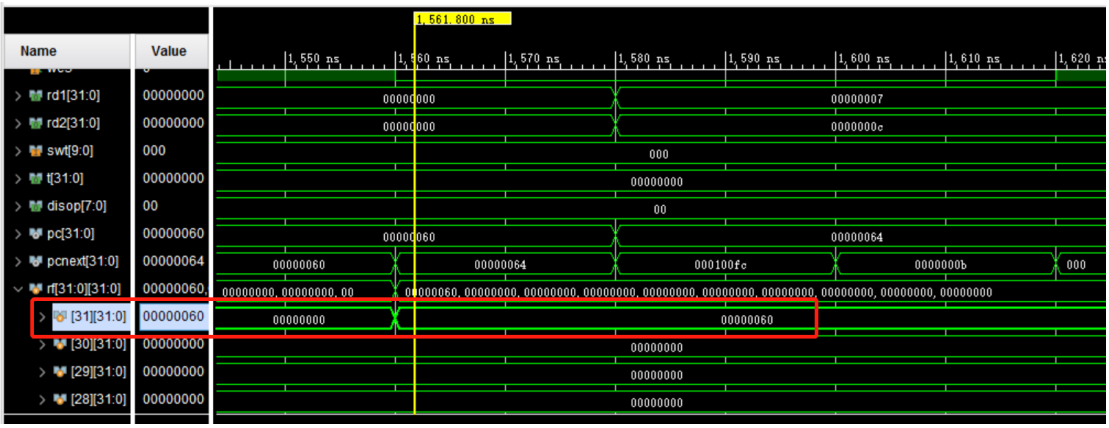


Figure 10 jal: jal 0x60

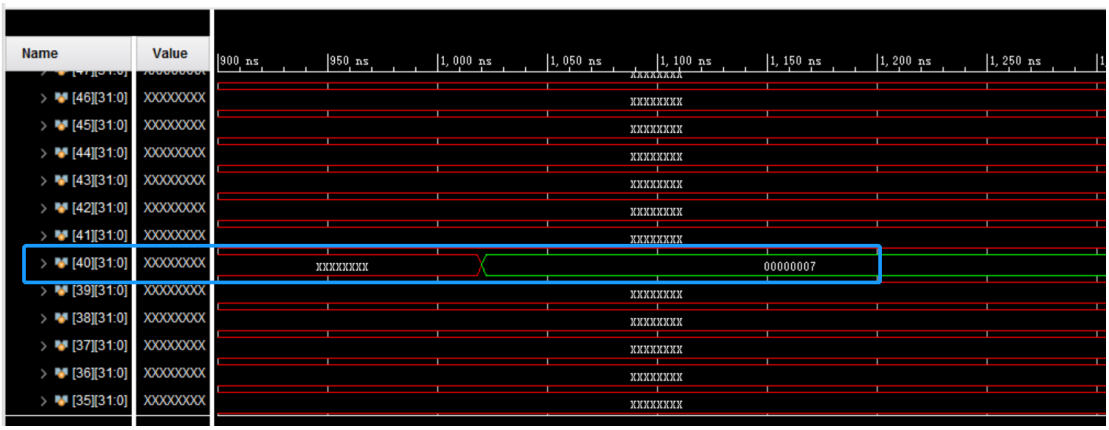


Figure 6 sw: [148 + $3] = $7 = 7

4.2仿真 2：sortfile0（8 个数的插入排序）

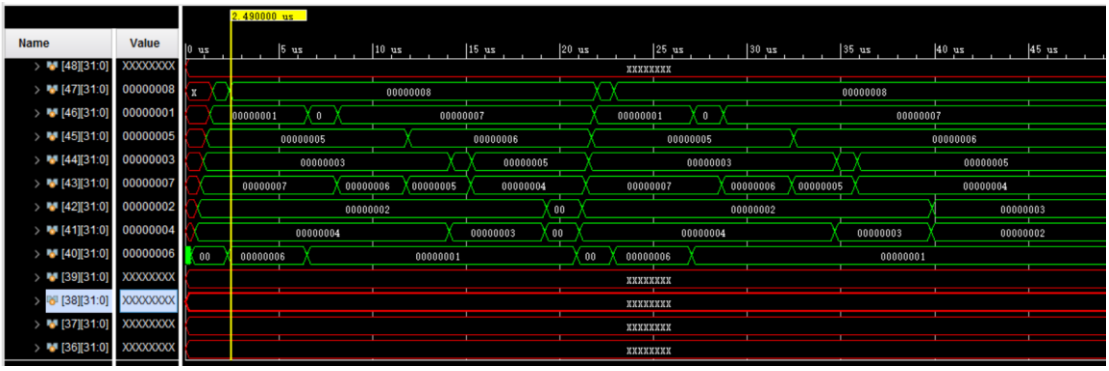


Figure 7 sortfile0 验证

可以看到插入排序的变化过程：

84273516->64273518->14273568->14263578->14253678->13254678->12345678

4.3仿真 3：sumfile0（1~100 求和）

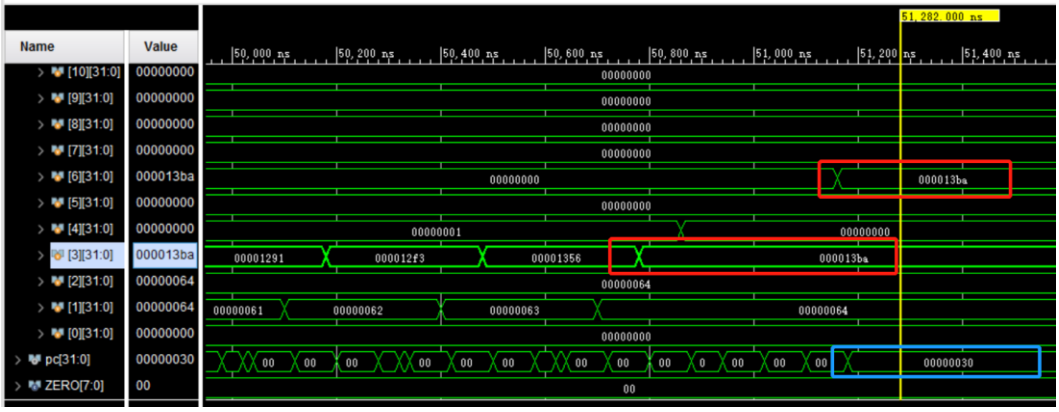


Figure 8 1~100 求和

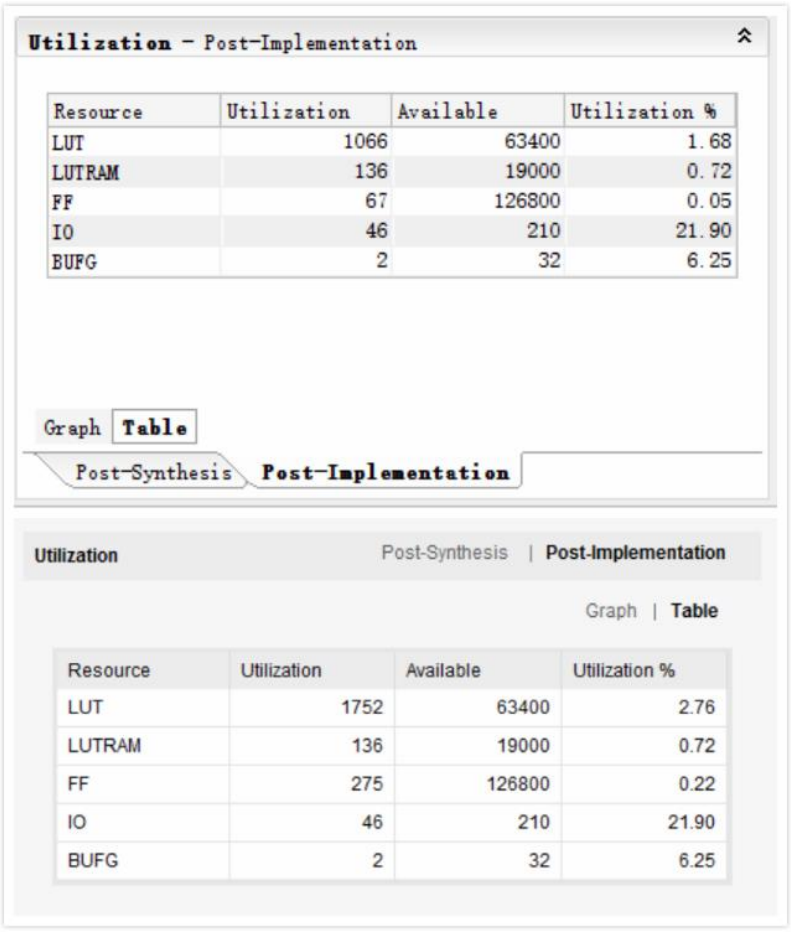
5 性能分析

（以下性能在运行 sortfile0 程序、clkdiv 分频取 q[0]的情况下测定。）

5.1时钟占用

Figure 9 时钟占用（上为单周期，下为多周期）

5.2资源占用



Utilization – Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	1066	63400	1.68
LUTRAM	136	19000	0.72
FF	67	126800	0.05
IO	46	210	21.90
BUFG	2	32	6.25

Graph **Table**

Post-Synthesis **Post-Implementation**

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1752	63400	2.76
LUTRAM	136	19000	0.72
FF	275	126800	0.22
IO	46	210	21.90
BUFG	2	32	6.25

Figure 10 资源占用（上为单周期，下为多周期）

6 实验感想

5.1对多周期 MIPS 处理器的运行机制更加了解，是单周期处理器的升级版，也为之后流水线 MIPS 处理器的实现打下基础；

5.2更加熟悉 FSM 的控制和实现机制，学会设计控制指令的 FSM 状态；

5.3对 MIPS 指令集有了更深的了解，并学会书写 MIPS 指令程序；

5.4对 verilog 语言更加深入；

5.5**能熟练运用开关来实现功能，能使用 LED 灯以及 8 个 7 段显示数码来直观地显示所有数值，能极大地方便 debug 及正确性验证。**