

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

本 CPU 为 Verilog 实现的流水线 MIPS - CPU, 支持的指令集包含 {LB、LBU、LH、LHU、LW、SB、SH、SW、ADD、ADDU、SUB、SUBU、MULT、MULTU、DIV、DIVU、SLL、SRL、SRA、SLLV、SRLV、SRAV、AND、OR、XOR、NOR、ADDI、ADDIU、ANDI、ORI、XORI、LUI、SLT、SLTI、SLTIU、SLTU、BEQ、BNE、BLEZ、BGTZ、BLTZ、BGEZ、J、JAL、JALR、JR、MFHI、MFLO、MTHI、MTLO、NOP}。为了实现这些功能, CPU 在顶层模块 mips 下并列包含了 DATAPATH、FORWARD_CONTROL、STALL_CONTROL 三个模块。

DATAPATH 模块下分五个流水级 F、D、E、M、W。F 级包含了 PC、IM 部件, D 级包含了 GRF、EXT、NPC、CMP 部件; E 级包含了 ALU 部件; M 级包含了 DM 部件; W 级连接到 D 级寄存器。相邻两流水级之间还各设置了一个流水线寄存器 REG_D、REG_E、REG_M、REG_W, 用于存储流水的信息。此外, 为满足数据冒险的转发需求, 还设置了六个转发多路选择器: D 级 MF_GRFRD1_D、MF_GRFRD2_D 分别用于选择需进入 CMP 部件和 E 级寄存器的两个寄存器值; E 级 MF_ALUA_E 用于选择参与 ALU 运算的第一个数据, MF_ALUB_E 用于选择参与 ALU 运算的第二个来自寄存器的数据, MF_MemData_E 用于选择进入 M 级寄存器的数据; MF_DMWD_M 用于选择写入 DM 的数据。

FORWARD_CONTROL 模块用于生成转发 MUX 的选择控制信号。

STALL_CONTROL 模块用于生成暂停信号。

此外, 工程文件中包含了名为 MACRO 的.v 文件用于宏定义。本流水线 CPU 主要采用分布式译码, 设置一个专门的 DECODER 模块, 在 D、E、M、W 级流水线寄存器分别实例化, 传入各级指令, 同时传出该级所需要的控制信号。

（二）关键模块定义

1. PC（F 级）

当前 PC 值的存储由一个 32 位 reg 变量保存。PC 模块端口定义如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
Pc_en	I	1	Pc 写使能信号
reset	I	1	同步复位信号
NPC	I	32	由 NPC 模块计算的下一个 PC 值
PC_F	O	32	当前的 PC
PC_4F	O	32	当前的 PC + 4
PC_8F	O	32	当前的 PC + 8

端口 Verilog 声明如下：

```
module PC(  
    input clk,  
    input pc_en,  
    input reset,  
    input [31:0] NPC,  
    output [31:0] PC_F,  
    output [31:0] PC4_F,  
    output [31:0] PC8_F  
);
```

2. IM（F 级、外置）

IM 端口声明如下：

信号名	方向	位宽	描述
i_inst_addr	I	32	当前 PC 值
i_inst_rdata	O	32	当前指令序列

3. REG_D (D 级)

REG_D 模块端口定义如下:

信号名	方向	位宽	描述
IR_F	I	32	F 级 32 位指令序列
pc4_F	I	32	F 级的 pc + 4
Pc8_F	I	32	F 级的 pc + 8
D_en	I	1	D 级寄存器写使能信号
Clk	I	1	时钟信号
Reset	I	1	同步复位信号
IR_D	O	32	D 级 32 位指令序列
Pc4_D	O	32	D 级的 pc + 4
Pc8_D	O	32	D 级的 pc + 8

端口 Verilog 声明如下:

```
module REG_D(  
    input [31:0] IR_F,  
    input [31:0] pc4_F,  
    input [31:0] pc8_F,  
    input D_en,  
    input clk,  
    input reset,  
    output reg [31:0] IR_D,  
    output reg [31:0] pc4_D,  
    output reg [31:0] pc8_D  
);
```

4. D_decoder (D 级)

该模块为 DECODER.v 译码器在 D 级的实例化, 专门产生 D 级的控制信号。

D_decoder 模块端口定义如下:

信号名	方向	位宽	描述
IR_D	I	32	D 级 32 位指令序列

A1_D	O	5	D 级指令对应传入 GRF A1 端口的寄存器编号
A2_D	O	32	D 级指令对应传入 GRF A2 端口的寄存器编号
EXTOp	O	1	EXT 控制信号
NPCOp	O	3	NPC 控制信号

端口 Verilog 实例化如下：（DECODER 模块声明见后）

```

DECODER d_decoder (
    .IR(IR_D),
    .A1(A1_D),
    .A2(A2_D),
    .EXTOp(EXTOp),
    .NPCOp(NPCOp)
);

```

5. NPC（D 级）

NPC 模块端口定义如下：

信号名	方向	位宽	描述
IR_D	I	32	D 级 32 位指令序列
NPCOp	I	3	NPC 控制信号
Pc_F	I	32	F 级的 pc
Pc4_D	I	32	D 级的 pc + 4
Pc8_D	I	32	D 级的 pc + 8
GPR_rs	I	32	GRF RD1 端口读出的值
Zero	I	1	CMP 的判零信号
npc	O	32	下一个 pc 值，传入 pc

NPCOp 控制信号真值表如下：

NPCOp[2:0]	功能	NPCOp[2:0]	功能
000	$NPC = pc_F + 4$	100	\

001	B 类指令 NPC 计算	101	\
010	跳转到 26 位立即数	110	\
011	跳转到 GPR_rs 寄存器	111	\

端口 Verilog 声明如下：

```
module NPC(
    input [31:0] IR_D,
    input [2:0] NPCOp,
    input [31:0] pc_F,
    input [31:0] pc4_D,
    input [31:0] pc8_D,
    input [31:0] GPR_rs,
    input zero,
    output reg [31:0] npc
);
```

6. EXT

EXT 模块端口定义如下：

信号名	方向	位宽	描述
EXTOp	I	1	扩展信号选择
IR_D	I	32	D 级 32 位指令序列
EXTResult	O	32	扩展后的 32 位信号

端口 Verilog 声明如下：

```
module EXT(
    input [31:0] IR_D,
    input EXTOp,
    output [31:0] EXTResult
);
```

7. GRF (D 级)

GRF 中包含 32 个 32 位寄存器，分别对应 0~31 号寄存器，其中 0 号寄存器读取的结果恒为 0。具体模块端口定义如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
RegWrite_W	I	1	写使能信号
Pc4_W	I	32	W 级的 pc + 4
Pc4_D	I	32	D 级的 pc + 4
Pc8_D	I	32	D 级的 pc + 8
A1_D	I	5	读寄存器的地址
A2_D	I	5	读寄存器的地址
A3_W	I	5	写寄存器的地址
RegData	I	32	32 位写入数据
RD1	O	32	32 位读出数据
RD2	O	32	32 位读出数据

端口 Verilog 声明如下：

```
module GRF( // support "internal forwarding"
    input clk,
    input reset,
    input RegWrite_W,
    input [31:0] pc4_W,
    input [31:0] pc4_D,
    input [31:0] pc8_D,
    input [4:0] A1_D,
    input [4:0] A2_D,
    input [4:0] A3_W,
    input [31:0] RegData,
    output reg [31:0] RD1,
    output reg [31:0] RD2
);
```

GRF 内部 32 bit * 32 的寄存器的具体实现如下：

```
reg [31:0] register[0:31];
```

8. CMP

CMP 模块端口定义如下：

信号名	方向	位宽	描述
V1	I	32	来自转发 mux MF_GRFRD1_D 的输出信号
V2	I	32	来自转发 mux MF_GRFRD2_D 的输出信号
zero	O	32	判零信号

端口 Verilog 声明如下：

```
module CMP(  
    input [31:0] V1,  
    input [31:0] V2,  
    output zero  
);
```

9. REG_E (E 级)

REG_E 模块端口定义如下：

信号名	方向	位宽	描述
IR_D	I	32	D 级 32 位指令序列
V1	I	32	来自转发 mux MF_GRFRD1_D 的输出信号
V2	I	32	来自转发 mux MF_GRFRD2_D 的输出信号
EXTResult	I	32	扩展后的 32 位信号
pc4_D	I	32	D 级的 pc + 4
Pc8_D	I	32	D 级的 pc + 8
E_clr	I	1	E 级寄存器清除信号
Clk	I	1	时钟信号

Reset	I	1	同步复位信号
IR_E	O	32	E 级 32 位指令序列
V1_E	O	32	来自转发 mux MF_GRFRD1_D 的输出信号流水到 E 级
V2_E	O	32	来自转发 mux MF_GRFRD2_D 的输出信号流水到 E 级
EXTResult_E	O	32	扩展后的 32 位信号流水到 E 级
Pc4_E	O	32	E 级的 pc + 4
Pc8_E	O	32	E 级的 pc + 8

端口 Verilog 声明如下：

```
module REG_E(
    input [31:0] IR_D,
    input [31:0] V1,
    input [31:0] V2,
    input [31:0] EXTResult,
    input [31:0] pc4_D,
    input [31:0] pc8_D,
    input clk,
    input reset,
    input E_clr,
    output reg [31:0] IR_E,
    output reg [31:0] V1_E,
    output reg [31:0] V2_E,
    output reg [31:0] EXTResult_E,
    output reg [31:0] pc4_E,
    output reg [31:0] pc8_E
);
```

10. E_decoder (E 级)

该模块为 DECODER.v 译码器在 E 级的实例化，专门产生 E 级的控制信号。

E_decoder 模块端口定义如下：

信号名	方向	位宽	描述
IR_E	I	32	E 级 32 位指令序列
ALU_Aop	O	1	ALU 中 ALUSrcA 端口选择信号

ALU_Bop	O	2	ALU 中 ALUSrcB 端口选择信号
ALUOp	O	4	ALU 控制信号
Shamt_E	O	5	E 级指令的 shamt 字段
MDUOp	O	4	MDU 模块控制信号
Start_E	O	1	MDU 模块 start 信号

端口 Verilog 实例化如下：（DECODER 模块声明见后）

```
DECODER e_decoder (
    .IR(IR_E),
    .ALU_Aop(ALU_Aop),
    .ALU_Bop(ALU_Bop),
    .ALUOp(ALUOp),
    .shamt(shamt_E),
    .start(start_E),
    .MDUOp(MDUOp)
);
```

11. ALU（E 级）

ALU 支持 AND、OR、ADD、SUB、SLL、SRL、SRA、SLT（有符号）等运算，其中 ADD、SUB 运算不支持溢出检测。

ALU 端口声明如下：

信号名	方向	位宽	描述
ALUSrcA	I	32	ALU 操作数 1
ALUSrcB	I	32	ALU 操作数 2
ALUOp	I	4	ALU 功能选择信号
Shamt_E	I	5	来自 E 级译码器的 shamt_E
ALUResult_E	O	32	ALU 运算结果

端口 Verilog 声明如下：

```

module ALU(
    input [31:0] ALUSrcA,
    input [31:0] ALUSrcB,
    input [3:0] ALUOp,
    input [4:0] shamt_E,
    output reg [31:0] ALUResult_E
);

```

ALU 控制信号真值表如下：

ALUOp[3:0]	功能	ALUOp[3:0]	功能
0000	A AND B	1000	\
0001	A OR B	1001	\
0010	A + B	1010	\
0011	A - B	1011	\
0100	A 逻辑左移 B	1100	\
0101	A 逻辑右移 B	1101	\
0110	A 算数右移 B	1110	\
0111	SLT	1111	\

12. MDU (E 级)

MDU 支持 MULT、MULTU、DIV、DIVU、MFHI、MFLO、MTHI、MTLO 等运算。

MDU 端口声明如下：

信号名	方向	位宽	描述
In1	I	32	MDU 操作数 1
In2	I	32	MDU 操作数 2
MDUOp	I	4	MDU 功能选择信号
start_E	I	1	来自 E 级译码器的 start_E
clk	I	1	时钟信号
Reset	I	1	同步复位信号

Hi	O	32	32 位 hi 寄存器
Lo	O	32	32 位 lo 寄存器
Busy	O	1	Busy 信号（是否正在进行乘除法）

端口 Verilog 声明如下：

```
module MDU( // mutiply-divide unit
    input [31:0] in1,
    input [31:0] in2,
    input [3:0] MDUOp,
    input start_E,
    input clk,
    input reset,
    output reg [31:0] hi,
    output reg [31:0] lo,
    output reg busy
);
```

MDU 控制信号真值表如下：

MDUOp[3:0]	功能	MDUOp[3:0]	功能
0000	A MULT B	1000	\
0001	A MULTU B	1001	\
0010	A DIV B	1010	\
0011	A DIVU B	1011	\
0100	MTHI A → HI	1100	\
0101	MTLO A → LO	1101	\
0110	\	1110	\
0111	\	1111	NONE

13. REG_M（M 级）

REG_M 模块端口定义如下：

信号名	方向	位宽	描述
Hi_E	I	32	E 级输出的 hi 寄存器的值
Lo_E	I	32	E 级输出的 lo 寄存器的值
IR_E	I	32	E 级 32 位指令序列
V2_E	I	32	来自转发 mux MF_GRFRD2_D 的输出信号流水到 E 级的信号
ALUResult_E	I	32	ALU 计算结果
pc4_E	I	32	E 级的 pc + 4
Pc8_E	I	32	E 级的 pc + 8
Clk	I	1	时钟信号
Reset	I	1	同步复位信号
IR_M	O	32	M 级 32 位指令序列
V2_M	O	32	来自转发 mux MF_GRFRD2_D 的输出信号流水到 M 级
ALUResult_M	O	32	扩展后的 32 位信号流水到 M 级
Pc4_M	O	32	M 级的 pc + 4
Pc8_M	O	32	M 级的 pc + 8
Hi_M	O	32	M 级输出的 hi 寄存器的值
Lo_M	O	32	M 级输出的 lo 寄存器的值

端口 Verilog 声明如下：

```

module REG_M(
    input [31:0] hi_E,
    input [31:0] lo_E,
    input [31:0] IR_E,
    input [31:0] V2_E,
    input [31:0] ALUResult_E,
    input [31:0] pc4_E,
    input [31:0] pc8_E,
    input clk,
    input reset,
    output reg [31:0] IR_M,
    output reg [31:0] ALUResult_M,
    output reg [31:0] V2_M,
    output reg [31:0] pc4_M,
    output reg [31:0] pc8_M,
    output reg [31:0] hi_M,
    output reg [31:0] lo_M
);

```

14. M_decoder (M 级)

该模块为 DECODER.v 译码器在 M 级的实例化,专门产生 M 级的控制信号。

M_decoder 模块端口定义如下:

信号名	方向	位宽	描述
IR_M	I	32	M 级 32 位指令序列
MemWrite_M	O	1	DM 写使能信号

端口 Verilog 实例化如下: (DECODER 模块声明见后)

```

DECODER m_decoder (
    .IR(IR_M),
    .MemWrite(MemWrite_M)
);

```

15. BE (M 级)

BE 模块端口定义如下:

信号名	方向	位宽	描述
IR_M	I	32	M 级 32 位指令序列

MemAddr	I	32	写地址信号
M_data_byteen	O	4	DM 字节写使能信号

端口 Verilog 声明如下：

```
module BE(
    input [31:0] IR_M,
    input [31:0] MemAddr,
    output reg [3:0] m_data_byteen
);
```

16. DM（M 级、外置）

DM 端口声明如下：

信号名	方向	位宽	描述
m_data_addr	I	32	写入数据的地址
m_data_wdata	I	32	待写入 DM 的 32 位数据信号
m_data_byteen	I	4	写数据字节使能信号
m_inst_addr	I	32	M 级 pc 值
m_data_rdata	O	1	DM 读出的数据

17. DM_EXT（M 级）

DM_EXT 端口声明如下：

信号名	方向	位宽	描述
IR_M	I	32	M 级 32 位指令序列
DM_RD_in	I	32	DM 读出的数据
MemAddr	I	32	写入数据的地址
DM_RD_out	O	32	DM 读出的数据处理后的数值

端口 Verilog 声明如下：

```

module DM_EXT(
    input [31:0] IR_M,
    input [31:0] DM_RD_in,
    input [31:0] MemAddr,
    output reg [31:0] DM_RD_out
);

```

18. REG_W (W 级)

REG_W 模块端口定义如下：

信号名	方向	位宽	描述
Hi_M	I	32	M 级输出的 hi 寄存器的值
Lo_M	I	32	M 级输出的 lo 寄存器的值
IR_M	I	32	M 级 32 位指令序列
ALUResult_M	I	32	ALU 计算结果流水到 M 级
pc4_M	I	32	M 级的 pc + 4
Pc8_M	I	32	M 级的 pc + 8
Clk	I	1	时钟信号
Reset	I	1	同步复位信号
DM_RD	I	32	DM 读出的 32 位数据
IR_W	O	32	W 级 32 位指令序列
ALUResult_W	O	32	扩展后的 32 位信号流水到 W 级
Pc4_W	O	32	W 级的 pc + 4
Pc8_W	O	32	W 级的 pc + 8
DM_RD_W	O	32	DM 读出的 32 位数据流水到 W 级
Hi_W	O	32	W 级输出的 hi 寄存器的值
Lo_W	O	32	W 级输出的 lo 寄存器的值

端口 Verilog 声明如下：

```

module REG_W(
    input [31:0] hi_M,
    input [31:0] lo_M,
    input [31:0] IR_M,
    input [31:0] ALUResult_M,
    input clk,
    input reset,
    input [31:0] pc4_M,
    input [31:0] pc8_M,
    input [31:0] DM_RD,
    output reg [31:0] IR_W,
    output reg [31:0] ALUResult_W,
    output reg [31:0] pc4_W,
    output reg [31:0] pc8_W,
    output reg [31:0] DM_RD_W,
    output reg [31:0] hi_W,
    output reg [31:0] lo_W
);

```

16. W_decoder (W 级)

该模块为 DECODER.v 译码器在 W 级的实例化,专门产生 W 级的控制信号。

W_decoder 模块端口定义如下:

信号名	方向	位宽	描述
IR_W	I	32	W 级 32 位指令序列
RegWrite_W	O	1	GRF 写使能信号
A3_W	O	5	GRF 写入地址
REGorMEM_W	O	1	回写数据信号的选择 0: ALUResult_W; 1: DM_RD_W;
REGop_W	O	3	GRF 写入数据端口选择信号

端口 Verilog 实例化如下: (DECODER 模块声明见后)


```

DECODER w_decoder (
    .IR(IR_W),
    .A3(A3_W),
    .RegWrite(RegWrite_W),
    .REGorMEM(REGorMEM_W),
    .REGop(REGop_W)
);

```

17. DECODER

译码器 DECODER 端口声明如下：

端口声明	方向	位宽	描述
IR	I	32	指令 Instr
REGorMEM	O	1	grf 回写控制信号之一： 0: ALUResult 回写； 1: DM 读出的值回写
MemWrite	O	1	DM 写使能信号： 0: 不能写入； 1: 可以写入
NPCOp	O	3	NPC 计算方式选择信号： 000: $PC = PC + 4$; 001: $PC = PC + 4 + ALUZero \&\& Sign_ext(16imm \parallel 00)$ (b 类) 010: $PC = PC[31:28] \parallel 26imm \parallel 00$ (j、jal) 011: $PC = GPR[rs]$ (jr \$ra, 注: \$ra 不一定得是\$31) 100: 暂未定义 101: 暂未定义 110: 暂未定义 111: 暂未定义
EXTOp	O	1	EXT 对 16 位立即数的扩展方式： 0: 符号扩展 1: 零扩展

A1op	O	2	GRF 中 A1 端口选择信号： 00: 读取 rs (Instr[25:21]) 01: 暂未定义 10: 暂未定义 11: 暂未定义
A2op	O	1	GRF 中 A2 端口选择信号： 0: 读取 rt (Instr[20:16]) 1: 暂未定义
A3op	O	2	GRF 中 A3 端口选择信号： 00: 读取 rd (Instr[15:11]) 01: 读取 rt (Instr[20:16]) 10: 读取常数 31 (即\$r31 址) 11: 暂未定义
REGop	O	3	GRF 写入数据端口选择信号： 000: 回写 REGorMEM 选择后的信号 001: 回写 PC_W 加 8 的值 010: 回写 hi_W 011: 回写 lo_W 100: 暂未定义 101: 暂未定义 110: 暂未定义 111: 暂未定义
RegWrite	O	1	GRF 写使能信号： 0: 不可写 1: 可写
ALU_Aop	O	1	ALU SrcA 端口选择信号： 0: 读取 RD1 1: 暂未定义
ALU_Bop	O	2	ALU SrcB 端口选择信号：

			00: 读取 RD2 01: 读取 16 位立即数（根据 ALUZero 选择）扩展后的 32 位数据 10: 读取指令 shamt 片段 0 扩展后的结果 11: 暂未定义
ALUOp	O	4	ALU 功能选择信号： 0000: and 0001: or 0010: add 0011: sub 0100: sll 0101: srl 0110: sra 0111: slt 1000: lui 1001: sllv 1010: srlv 1011: srav 1100: xor 1101: nor 1110: sltu
A1	O	5	GRF 的 A1 端口输入地址
A2	O	5	GRF 的 A2 端口输入地址
A3	O	5	GRF 的 A3 端口输入地址
shamt	O	5	IR 中 shamt 字段
Tuse_A1_0	O	1	A1 端口数据需要在 D 级使用
Tuse_A1_1	O	1	A1 端口数据需要在 E 级使用
Tuse_A2_0	O	1	A2 端口数据需要在 D 级使用
Tuse_A2_1	O	1	A2 端口数据需要在 E 级使用

Tuse_A2_2	O	1	A2 端口数据需要在 M 级使用
Tnew	O	3	该指令产生结果所需要的时钟周期
Start	O	1	MDU start 控制信号
MDUOp	O	4	MDU 功能选择信号： 0000: MULT 0001: MULTU 0010: DIV 0011: DIVU 0100: MTHI 0101: MTLO 0110-1110: 暂未定义 1111: NONE
MDU_IR	O	1	是否需要用到乘除法器件

端口 verilog 声明如下：

```
module DECODER(
    input [31:0] IR,
    output reg REGorMEM,
    output reg MemWrite,
    output reg [2:0] NPCOp,
    output reg EXTOp,
    output reg [1:0] A1op,
    output reg A2op,
    output reg [1:0] A3op,
    output reg [2:0] REGOp,
    output reg RegWrite,
    output reg ALU_Aop,
    output reg [1:0] ALU_Bop,
    output reg [3:0] ALUOp,
    output [4:0] A1,
    output [4:0] A2,
    output [4:0] A3,
    output [4:0] shamt,
    //-----
    output reg Tuse_A1_0,
    output reg Tuse_A1_1,
    output reg Tuse_A2_0,
    output reg Tuse_A2_1,
    output reg Tuse_A2_2,
    output reg [2:0] Tnew,
    output reg start,
    output reg [3:0] MDUOp,
    output reg MDU_IR
);
```

译码器真值表如下：（本电路实现中 X 全置 0）

指令	opcode	funct	RT	REGorME M	MemWr ite	NPCop	EXTOp	A1op	A2op	A3op	REGop	RegWrit e	ALU_Ao p	ALU_Bo p	ALUOp
and	000000	100100		0	0	000	0	00	0	00	000	1	0	00	0000
or	000000	100101		0	0	000	0	00	0	00	000	1	0	00	0001
xor	000000	100110		0	0	000	0	00	0	00	000	1	0	00	1100
nor	000000	100111		0	0	000	0	00	0	00	000	1	0	00	1101
add	000000	100000		0	0	000	0	00	0	00	000	1	0	00	0010
addu	000000	100001		0	0	000	0	00	0	00	000	1	0	00	0010
sub	000000	100010		0	0	000	0	00	0	00	000	1	0	00	0011
subu	000000	100011		0	0	000	0	00	0	00	000	1	0	00	0011
addi	001000			0	0	000	0	00	0	01	000	1	0	01	0010
addiu	001001			0	0	000	0	00	0	01	000	1	0	01	0010
andi	001100			0	0	000	1	00	0	01	000	1	0	01	0000
ori	001101			0	0	000	1	00	0	01	000	1	0	01	0001
xori	001110			0	0	000	1	00	0	01	000	1	0	01	1100
slli	001010			0	0	000	0	00	0	01	000	1	0	01	0111
slliu	001011			0	0	000	0	00	0	01	000	1	0	01	1110

指令	opcode	funct	RT	REGorME M	MemWr ite	NPCop	EXTOp	A1op	A2op	A3op	REGop	RegWrit e	ALU_Ao p	ALU_Bo p	ALUOp
lw	100011			1	0	000	0	00	0	01	000	1	0	01	0010
lb	100000			1	0	000	0	00	0	01	000	1	0	01	0010
lbu	100100			1	0	000	0	00	0	01	000	1	0	01	0010
lh	100001			1	0	000	0	00	0	01	000	1	0	01	0010
lhu	101001			1	0	000	0	00	0	01	000	1	0	01	0010
sw	101011			0	1	000	0	00	0	00	000	0	0	01	0010
sb	101000			0	1	000	0	00	0	00	000	0	0	01	0010
sh	101001			0	1	000	0	00	0	00	000	0	0	01	0010
beq	000100			0	0	001	0	00	0	00	000	0	0	00	0000
bne	000101			0	0	001	0	00	0	00	000	0	0	00	0000
blez	000110			0	0	001	0	00	0	00	000	0	0	00	0000
bgtz	000111			0	0	001	0	00	0	00	000	0	0	00	0000
bltz	000001		00000	0	0	001	0	00	0	00	000	0	0	00	0000
bgez	000001		00001	0	0	001	0	00	0	00	000	0	0	00	0000
lui	001111			0	0	000	0	00	0	01	000	1	0	01	1000
sll	000000	000000		0	0	000	0	00	0	00	000	1	0	00	0100
srl	000000	000010		0	0	000	0	00	0	00	000	1	0	00	0101
sra	000000	000011		0	0	000	0	00	0	00	000	1	0	00	0110
sllv	000000	000100		0	0	000	0	00	0	00	000	1	0	00	1001
srlv	000000	000110		0	0	000	0	00	0	00	000	1	0	00	1010
srav	000000	000111		0	0	000	0	00	0	00	000	1	0	00	1011
sllt	000000	101010		0	0	000	0	00	0	00	000	1	0	00	0111
slltu	000000	101011		0	0	000	0	00	0	00	000	1	0	00	1110
j	000010			0	0	010	0	00	0	00	000	0	0	00	0000
jal	000011			0	0	010	0	00	0	10	001	1	0	00	0000
jalr	000000	001001		0	0	011	0	00	0	00	001	1	0	00	0000
jr	000000	001000		0	0	011	0	00	0	00	000	0	0	00	0000

指令	opcode	funct	RT	REGorMEM	MemWrite	NPCOp	EXTOp	ALOp	A2Op	A3Op	REGOp	RegWrite	ALU_AOp	ALU_BOp	ALUOp	MDUOp	start
mult	000000	011000		0	0	000	000	00	0	00	000	0				0000	1
multu	000000	011001		0	0	000	000	00	0	00	000	0				0001	1
div	000000	011010		0	0	000	000	00	0	00	000	0				0010	1
divu	000000	011011		0	0	000	000	00	0	00	000	0				0011	1
mfhi	000000	010000		0	0	000	000	00	0	00	010	1				1111	0
mflo	000000	010010		0	0	000	000	00	0	00	011	1				1111	0
mthi	000000	010001		0	0	000	000	00	0	00	000	0				0100	0
mtlo	000000	010011		0	0	000	000	00	0	00	000	0				0101	0

(三) 重要机制实现方法

1. 跳转

该 CPU 涉及的跳转指令有：b 类指令（beq、bne、bgtz、bgez、bltz、blez）、j、jal、jr、jalr。

B 类指令较为相似，以 beq 为例（ $GPR[rs] == GPR[rt]$ 则 $zero = 1$ ）。

BEQ：相等时转移

编码	31	26	25	21	20	16	15	0
	beq 000100		rs		rt		offset	
	6		5		5		16	
格式	beq rs, rt, offset							
描述	if (GPR[rs] == GPR[rt]) then 转移							
操作	if (GPR[rs] == GPR[rt]) PC ← PC + 4 + sign_extend(offset 0 ²) else PC ← PC + 4							
示例	beq \$s1, \$s2, -2							
其他								

对于 beq 指令，控制器解析 beq 指令，输出的 $NPCOp = 001$ ，同时 ALU 判断两个操作数 $GPR[rs]$ 、 $GPR[rt]$ 是否相等，若相等将 zero 置一。在 NPC 内，当 $ALUZero = 1$ 并且 $NPCOp = 001$ 时，才执行跳转，即 $PC = PC + 4 + zero \&\& \text{Sign_ext}(16imm \parallel 00)$ ，若不满足该条件，则只执行 $PC = PC + 4$ 。

其他 B 类指令如 bgtz，只须将置 zero 为 1 的条件改为 $GPR[rs] > 0$ 即可，在此不赘述。

J 类指令如下：

J: 跳转

编码	31	26	25	0
	j 000010	instr_index		
	6	26		
格式	j target			
描述	j 指令是 PC 相关的转移指令。当把 4GB 划分为 16 个 256MB 区域，j 指令可以在当前 PC 所在的 256MB 区域内任意跳转。			
操作	$PC \leftarrow PC_{31..28} instr_index 0^2$			
示例	j Loop_End			
其他	如果需要跳转范围超出了当前 PC 所在的 256MB 区域内时，可以使用 JR 指令。			

对于 j 指令，控制器解析 j 指令，将 NPCOp 置为 010，同时 NPC 读入 26 位立即数， $PC = \{PC[31:28], 26imm, 2\{0\}\}$ （verilog 位拼接运算符），从而实现跳转。

JAL: 跳转并链接

编码	31	26	25	0
	jal 000011	instr_index		
	6	26		
格式	jal target			
描述	jal 指令是函数指令，PC 转向被调用函数，同时将当前 PC+4 保存在 GPR[31]中。当把 4GB 划分为 16 个 256MB 区域，jal 指令可以在当前 PC 所在的 256MB 区域内任意跳转。			
操作	$PC \leftarrow PC_{31..28} instr_index 0^2$ $GPR[31] \leftarrow PC + 4$			
示例	jal my_function_name			
其他	jal 与 jr 配套使用。jal 用于调用函数，jr 用于函数返回。当所调用的函数地址超出了当前 PC 所在的 256MB 区域内时，可以使用 jalr 指令。			

对于 jal 指令，其 NPC 相关控制信号与上述 j 指令相同。不同的是，对于 jal 指令，还需要通过 REGop 信号选择 10，将 PC + 4 的值存入 \$31 寄存器，为此，在 grf 写数据端口 WD3 放置了一个 4 选 1 多路选择器，其第三个选择端口为当前 PC 加 4 的值。

JR: 跳转至寄存器

编码	31	26	25	21	20	11	10	6	5	0
	special 000000	rs		0 00 0000 0000			0 00000	jr 001000		
	6	5		10			5		6	
格式	jr rs									
描述	PC ← GPR[rs]									
操作	PC ← GPR[rs]									
示例	jr \$31									
其他	jr 与 jal/jalr 配套使用。jal/jalr 用于调用函数，jr 用于函数返回。									

对于 jr 指令，其将 GPR[rs]寄存器存的 32 位值存入 PC，在数据通路上只需要在 NPC 的输入端口增加一个读入寄存器(32 位数据)的端口,同时设置 NPCOp = 011 时，将 pc 设置为输入的 32 位数据的值。

JALR: 跳转并链接

编码	31	26	25	21	20	16	15	11	10	6	5	0
	special 000000	rs			0 00000		rd		0 00000		jalr 001001	
	6		5		5		5		5		6	
格式	jalr rd, rs											
描述	jalr 指令是函数指令，PC 转向被调用函数(函数入口地址保存在 GPR[rs]中)，同时将当前 PC+4 保存在 GPR[rd]中。											
操作	$PC \leftarrow GPR[rs]$ $GPR[rd] \leftarrow PC + 4$											
示例	jalr \$s1, \$31											

对于 jalr 指令，其涉及的 PC 操作与 jr 相同，都是将 GPR[rs]存入 PC。同时，它与 jal 指令相似，都需要将当前 pc 加 4 的值存入寄存器。不同的是，jal 将 pc + 4 存入\$31 寄存器，故 A3op 选择 10；而 jalr 将 pc + 4 存入\$rd 寄存器，故 A3op 选择 00。

2. 转发

转发 mux:

	D		E			M
Forward MUX	MF_GRFRD1_D	MF_GRFRD2_D	MF_ALUA_E	MF_ALUB_E	MF_MemData_E	MF_DMWD_M
sequence_input	RD1	RD2	V1_E_left	V2_E_left	V2_E_left	V2_M
forward_input	ALUResult_M	ALUResult_M	ALUResult_M	ALUResult_M	ALUResult_M	ALUResult_W
	pc4_M	pc4_M	pc4_M	pc4_M	pc4_M	DM_RD_W
	hi_M	hi_M	hi_M	hi_M	hi_M	pc4_W
	lo_M	lo_M	lo_M	lo_M	lo_M	hi_W
	ALUResult_W	ALUResult_W	ALUResult_W	ALUResult_W	ALUResult_W	lo_W
	DM_RD_W	DM_RD_W	DM_RD_W	DM_RD_W	DM_RD_W	
	pc4_W	pc4_W	pc4_W	pc4_W	pc4_W	
	hi_W	hi_W	hi_W	hi_W	hi_W	
	lo_W	lo_W	lo_W	lo_W	lo_W	
select	MF_GRFRD1_D_sel	MF_GRFRD2_D_sel	MF_ALUA_E_sel	MF_ALUB_E_sel	MF_MemData_E_sel	MF_DMWD_M_sel
output	V1	V2	V1_E_right	V2_E_right_ALUB	V2_E_right2_RegM	MemData

3. 暂停

Tuse-Tnew 表格:

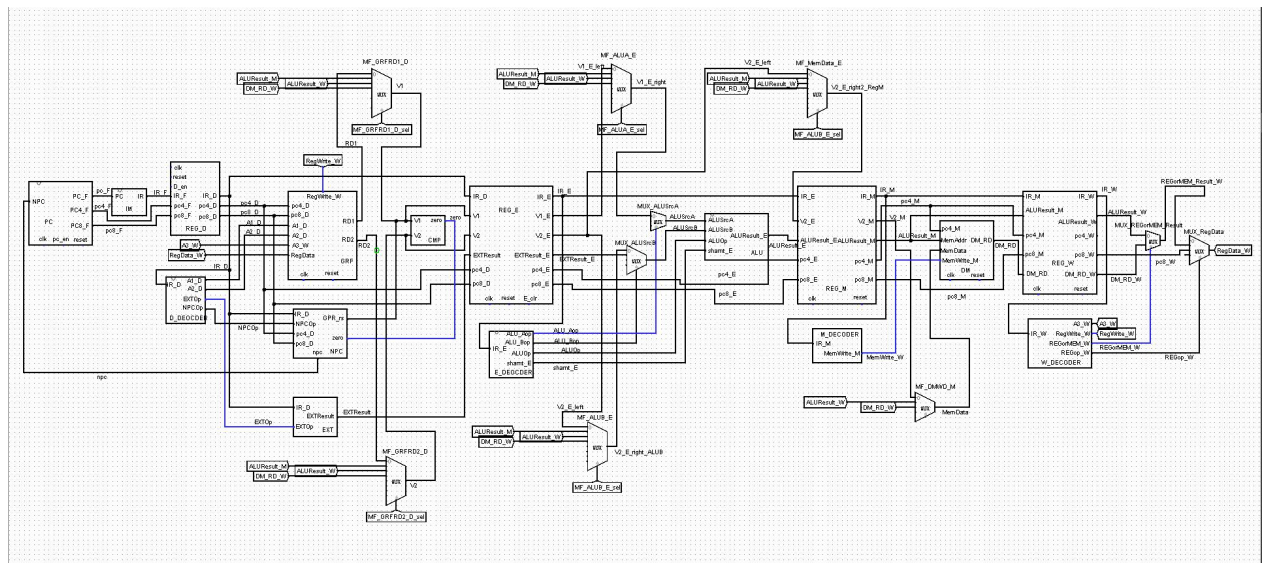
	Tuse			Tnew			
	A1	A2		功能部件	E	M	W
and	1	1		ALU	1	0	0
or	1	1		ALU	1	0	0
xor	1	1		ALU	1	0	0
nor	1	1		ALU	1	0	0
add	1	1		ALU	1	0	0
addu	1	1		ALU	1	0	0
sub	1	1		ALU	1	0	0
subu	1	1		ALU	1	0	0
addi	1			ALU	1	0	0
addiu	1			ALU	1	0	0
andi	1			ALU	1	0	0
ori	1			ALU	1	0	0
xori	1			ALU	1	0	0
slti	1			ALU	1	0	0
sltiu	1			ALU	1	0	0
lw	1			DM	2	1	0
lb	1			DM	2	1	0
lbu	1			DM	2	1	0
lh	1			DM	2	1	0
lhu	1			DM	2	1	0
sw	1	2					
sb	1	2					
sh	1	2					

beq	0	0					
bne	0	0					
blez	0						
bgtz	0						
bltz	0						
bgez	0						
lui				ALU	1	0	0
sll		1		ALU	1	0	0
srl		1		ALU	1	0	0
sra		1		ALU	1	0	0
sllv	1	1		ALU	1	0	0
srlv	1	1		ALU	1	0	0
srav	1	1		ALU	1	0	0
slt	1	1		ALU	1	0	0
sltu	1	1		ALU	1	0	0
j							
jal				PC	0	0	0
jalr	0			PC	0	0	0
jr	0						
nop							

4. 附：无转发数据通路表格（不含 BE、MDU、DM_EXT 模块）

I	J	K	L	M	N	O	P
REG_E	E		REG_M	M		REG_W	
	实例化DECODER	ALU		实例化DECODER	DM		实例化DECODER
IR_D	IR_E	ALUSrcA	IR_E	IR_M	clk	IR_M	IR_W
V1	ALU_Aop	ALUSrcB	V2_E	MemWrite_M	reset	ALUResult_M	A3_W
V2	ALU_Bop	ALUOp	ALUResult_E		MemAddr=ALUResult_E	clk	RegWrite_W
EXTRResult	ALUOp	shamt	pc4_E		MemData=V2_M	reset	
pc4_D	shamt_E	ALUResult_E	pc8_E		MemWrite_M	pc4_M	
pc8_D			clk		pc4_M	pc8_M	
clk			reset		DMRD	DMRD	
reset			IR_M			IR_W	
E_clr			ALUResult_M			ALUResult_W	
IR_E			V2_M			pc4_W	
V1_E			pc4_M			pc8_W	
V2_E			pc8_M			DMRD_W	
EXTRResult_E							
pc4_E							
pc8_E							
	Tnew_D			Tnew_E			Tnew_M
	Tnew_E			Tnew_M			Tnew_W

5. 附：LOGISIM 数据通路（不含 BE、MDU、DM_EXT 模块）



二、测试方案

（一）典型测试样例

1. 顺序执行指令（无流水线冒险）

```
1  # no forwarding, no stall, no jump or b_instr
2  # sequence
3  lui $0, 0x1234
4  lui $1, 0xf234
5  lui $2, 0x2345
6  lui $3, 0xf456
7
8  ori $4, 0x4567
9  ori $5, 0x5678
10 ori $6, 0x6789
11 ori $7, 0x789a
12
13 addu $8, $0, $0
14 addu $9, $1, $1
15 addu $10, $2, $2
16 addu $11, $3, $3
17
18 subu $12, $2, $1
19 subu $13, $3, $2
20 subu $14, $4, $3
21 subu $15, $5, $4
22
23 sll $16, $1, 2
24 sll $17, $2, 3
25 sll $18, $3, 4
26 sll $19, $4, 5
27
28 srl $20, $1, 2
29 srl $21, $2, 3
30 srl $22, $3, 4
31 srl $23, $4, 5
32
33 sra $24, $1, 2
34 sra $25, $2, 3
35 sra $26, $3, 4
36 sra $27, $4, 5
37
38 slt $28, $1, $2
39 slt $29, $2, $3
```

```
40  slt $30, $3, $4
41  slt $31, $4, $5
42
43  sw $1, 0($0)
44  sw $2, 4($0)
45  sw $3, 8($0)
46  sw $4, 12($0)
47  sw $5, 16($0)
48  sw $6, 20($0)
49
50  lw $11, 0($0)
51  lw $12, 4($0)
52  lw $13, 8($0)
53  lw $14, 12($0)
54  lw $15, 16($0)
55  lw $16, 20($0)
56
57  nop
58  nop
59  nop
60  nop
```

2. 纯 ALU 计算类指令（有流水线冒险）

```
1  lui $1, 0xffff
2  ori $2, $1, 0xffff
3  ori $3, $1, 0x1234
4  ori $4, $3, 0x5678
5  ori $5, $4, 0x3333
6  ori $6, $4, 0x1111
7  ori $7, $4, 0x2222
8
9  lui $8, 0x000f
10 addu $9, $8, $7
11 subu $10, $9, $8
12 addu $11, $10, $9
13 subu $12, $11, $0
14 subu $13, $9, $10
15
16 sll $14, $13, 2
17 sll $15, $2, 4
18 ori $15, $15, 0x1111
19 sll $16, $15, 2
20 sll $17, $16, 2
21 srl $18, $17, 2
22 srl $19, $18, 1
23 srl $20, $19, 1
24 srl $21, $19, 2
25 sra $22, $21, 2
26 sra $23, $21, 2
27 sra $24, $22, 1
28 sra $25, $22, 2
29
30 addu $1, $0, $0
31 lui $2, 0x000f
32
33 slt $26, $24, $25
34 slt $27, $26, $25
35 slt $27, $27, $26
36 slt $28, $25, $27
37 lui $1, 0x00ff
38 slt $29, $1, $2
39 slt $30, $0, $29
```


3. load 指令（有流水线冒险）

```
1  # initial
2  lui $1, 0xffff
3  ori $1, $1, 0xffff
4  lui $2, 0x1234
5  ori $2, $2, 0x1234
6  lui $3, 0x2345
7  ori $3, $3, 0x2345
8  lui $4, 0x3456
9  ori $4, $4, 0x3456
10 lui $5, 0x4567
11 ori $5, $5, 0x4567
12
13 sw $1, 0($0)
14 sw $2, 4($0)
15 sw $3, 8($0)
16 sw $4, 12($0)
17 sw $5, 16($0)
18
19 # start
20 lw $6, 0($0)
21 addu $7, $6, $1
22 addu $7, $7, $0
23 addu $7, $7, $0
24 addu $7, $7, $0
25 addu $7, $7, $0
26 subu $8, $6, $2
27 subu $9, $6, $3
28
29 lw $10, 4($0)
30 ori $11, $10, 0x0001
31 ori $12, $10, 0x0010
32 ori $13, $10, 0x0100
33
34 lw $14, 8($0)
35 sll $15, $14, 2
36 srl $16, $14, 1
37 sra $17, $14, 1
38
39 lw $18, 12($0)
40 lw $19, 16($0)
41 slt $20, $18, $19
42 nop
```

4. store 指令（有流水线冒险，无跳转指令相关数据冒险）

```
1  lui $1, 0x1234
2  ori $1, $1, 0x1234
3  sw $1, 0($0)
4
5  lui $2, 0x234
6  ori $2, $2, 0x234
7  sw $2, 4($0)
8
9  lw $3, 0($0)
10 sw $3, 8($0)
11
12 lw $4, 0($0)
13 lui $6, 0x1234
14 subu $5, $4, $6
15 addu $4, $4, $4
16 sw $4, 0($5)
17 #-----test continue-----
18 #initial
19 ori $30, $0, 0x0234
20 ori $29, $0, 0x0324
21 ori $28, $0, 0x0844
22 ori $27, $0, 0x0538
23 ori $26, $0, 0x011c
24
25 sw $30, 0($0)
26 sw $29, 4($0)
27 sw $28, 8($0)
28 sw $27, 12($0)
29 sw $26, 16($0)
30 #start
31 lw $10, 0($0)
32 lw $11, 4($0)
33 sw $10, 0($11)
34
35 lw $11, 8($0)
36 lw $12, 12($0)
37 sw $12, 0($11)
38 lw $12, 16($0)
39 sw $11, 0($12)
```


5. store 指令（有流水线冒险，有跳转指令相关数据冒险（如：jal））

```
1  ori $1, $0, 0x0020
2  lui $2, 0x3562
3  sll $3, $1, 4
4  sw $2, 4($1)
5
6  ori $3, $1, 0x0028
7  lw $4, 4($1)
8  srl $5, $2, 4
9  sw $4, 8($3)
10 slt $10, $3, $4
11 sw $3, 12($3)
12 lw $3, 8($3)
13 ori $1, $1, 0x0030
14
15 jal here
16 lui $12, 0x4566
17 sw $ra, 16($1)
18 here:
19 jal there
20 addu $11, $ra, $1
21 sw $ra, -400($ra)
22 there:
23 nop
```

6.beq 指令（有流水线冒险）

```
1  # initial
2  lui $1, 0xffff
3  ori $1, $1, 0xffff
4  lui $2, 0x1234
5  ori $2, $2, 0x1234
6  lui $3, 0x2345
7  ori $3, $3, 0x2345
8  lui $4, 0x3456
9  ori $4, $4, 0x3456
10 lui $5, 0x4567
11 ori $5, $5, 0x4567
12
13 sw $1, 0($0) # $1 = 0xffff_ffff
14 sw $2, 4($0) # $2 = 0x1234_1234
15 sw $3, 8($0) # $3 = 0x2345_2345
16 sw $4, 12($0) # $4 = 0x3456_3456
17 sw $5, 16($0) # $5 = 0x4567_4567
18
19 # start
20 lw $1, 8($0) # $1 = $3
21 lw $2, 8($0) # $2 = $3
22
23 beq $1, $2, yes1
24 ori $6, $1, 0x1234 # delayed branching
25 yes2:
26 ori $7, $1, 0x1234
27 sll $8, $7, 2
28 yes1:
29 beq $7, $8, yes2
30 sll $7, $7, 2
31 yes3:
32 lw $9, 0($0) # $9 = $1
33 beq $9, $1, yes3 # delayed branching
34 sll $1, $1, 2
35
36 beq $0, $1, no4
37 addu $1, $0, $0 # delayed branching
38
39 beq $1, $0, yes5
40 subu $1, $3, $2 # delayed branching
41 no4:
42 nop
43
44 yes5:
45 addu $9, $1, $2
46 addu $10, $3, $4
47 beq $9, $10, no6
48 slt $11, $9, $10 # delayed branching
49
50 beq $11, $0, no7
51 srl $12, $11, 31 # delayed branching
52
53 beq $12, $0, yes8
54 srl $13, $12, 31 # delayed branching
55
56 yes8:
57 beq $13, $0, no9
58 srl $13, $13, 31 # delayed branching
59
60 no6:
61 no7:
62 nop
63
64 no9:
65 nop
```

7.j 指令（有流水线冒险）

```
1  ori $1, $0, 1  # $1 = 1
2  ori $2, $0, 32  # $2 = 32
3  ori $30, $0, 0x0040  # $30 = 0x0040 (base address)
4
5
6  Loop:
7      beq $1, $2, end_loop
8
9      sll $3, $1, 4
10     addu $4, $3, $30
11     addu $20, $1, $1
12     sw $20, 4($4)
13     sw $1, 0($4)
14     j loop
15     lw $1, 4($4)  # delayed branch
16 end_Loop:
17     ori $6, $1, 0
18     j j1
19     addu $5, $1, $1
20
21 j1:
22     beq $5, $6, j1
23     addu $6, $6, $6
24     j end
25     nop
26     nop
27 end:
```

8.jal 指令（有流水线冒险）

```
1  ori $3, $0, 8
2  ori $1, $0, 0x0034
3  ori $2, $0, 0x2345
4  sw $2, 0($1)
5  lw $ra, 0($1)
6  jal label1  # 0x0000_3014
7  addu $ra, $ra, $3
8  subu $ra, $ra, $3
9  label3:
10 subu $ra, $ra, $3
11 jal label2
12 sra $ra, $ra, 4
13
14 label1:
15     j label3
16     sll $ra, $ra, 4
17 label2:
18     nop
19     addu $ra, $ra, $ra
```

9.jr 指令（有流水线冒险）

```
1  addu $1, $0, 0x3000 # $1 = base_pc
2  ori $2, $1, 0x0020
3  sw $2, 0($0)
4  lw $1, 0($0)
5  jr $1
6  addu $2, $2, $1
7  ori $3, $1, 0x0020
8  sll $4, $1, 2
9  ori $2, $1, 0x002c # here
10 addu $2, $2, $1
11 sw $2, 0($0)
12 lw $3, 0($0)
13 beq $2, $3, end
14 nop
15 nop
16 end:
17 nop
18 subu $3, $2, $3
```

10.MDU 模块相关指令

```
1  ori $2, $0, 6
2  ori $3, $0, 2
3  ori $4, $0, 3
4  ori $31, $0, 1024
5  mult $3, $4
6  mflo $5
7  beq $5, $2, label
8  ori $10, $0, 5
9  ori $11, $0, 6
10 label:
11 ori $11, $0, 0xc0f
12 ori $15, $0, 4
13 mult $11, $15
14 mflo $12
15 jr $12
16 ori $1, $0, 9
17 lui $9, 0x2222
18 ori $12, $0, 326
19 ori $13, $0, 4
20 mfhi $13
21 sw $2, 4($13)
22 mthi $2
23 mtlo $3
24 div $2, $3
25 mfhi $22
26 mflo $23
27 ori $14, $0, 256
28 lui $15, 0xffff
29 mult $15, $14
30 mfhi $26

31 mflo $27
32 multu $15, $14
33 mthi $26
34 mflo $27
35 divu $15, $14
36 mthi $9
37 mfhi $26
38 mflo $27
39 div $15, $14
40 mtlo $31
41 mfhi $26
42 mflo $27
43 sw $31, -4($31)
44 div $15, $14
45 mtlo $31
46 mfhi $26
47 mflo $27
48 lw $30, -4($31)
```

11.DM 模块相关指令（测试新增的字节、半字节操作）

```
1  ori $30, $0, 256
2  ori $1, $0, 0x1234
3  lui $2, 0x5678
4  addu $3, $1, $2
5  ori $4, $0, 0x98a1
6  lui $5, 0xb2c3
7  addu $6, $4, $5
8  ori $15, $0, 4
9  sw $3, 4($15)
10 sw $6, 8($15)
11 lw $21, 4($15)
12 addu $31, $21, $30
13 lbu $21, 4($15)
14 addu $31, $21, $30
15 lbu $21, 5($15)
16 addu $31, $21, $30
17 lbu $21, 6($15)
18 addu $31, $21, $30
19 lbu $21, 8($15)
20 addu $31, $21, $30
21 lbu $21, 9($15)
22 addu $31, $21, $30
23 lbu $21, 6($15)
24 addu $31, $21, $30
25 lbu $21, 8($15)
26 addu $31, $21, $30
27 lbu $21, 10($15)
28 addu $31, $21, $30
29 sb $6, 0($15)
30 sb $6, 1($15)
31 sb $6, 2($15)
32 sb $6, 3($15)
33 sh $6, 12($15)
34 sh $6, 14($15)
35 lw $25, 0($15)
36 lw $25, 4($15)
37 lw $25, 8($15)
38 lw $25, 12($15)
```

三、思考题

（一）题目描述：为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？

第一问：首先，ALU 的工作模式是直接计算并立即将计算结果传出使写入某一寄存器，而乘除法 `mult`、`multu`、`div`、`divu` 并非在计算结束后直接将结果写入寄存器，而是先保存在 `hi` 和 `lo` 寄存器，再通过新指令 `mflo`、`mfhi` 将最近产生的结果写入寄存器，这与 ALU 的工作步骤不符，因此无法直接整合进 ALU。其次，在实际工程中，乘除法并不是 `verilog` 中乘除符号一样的简单计算，其计算过程较长（本实验中为 5 周期或 10 周期），而 ALU 中其他计算均为 1 周期结束，因此二者的时序不符，无法将乘除法部件加入到 ALU 中。

第二问：对于乘法，两 32 位数相乘其结果位数最高可能达到 64 位，因此计算需要两个 32 位变量分别存储高 32 位和低 32 位；对于除法，将产生商和余数两个 32 位数，同样需要两个 32 位变量分别存储。与此同时，根据 MIPS 指令的规则，一条指令最多只能存入一个寄存器，因此在乘除模块中就需要两个独立的寄存器存储计算产生的两个结果，之后再根据需要决定是否、什么时候将这两个独立寄存器中的数据存入到 GRF 中，这样也更加快速，无需再次计算。

（二）题目描述：参照你对延迟槽的理解，试解释“乘除槽”。

乘除法在乘除模块中开始进行后，需要 5 或 10 周期才能产生结果，在这个时间中，后面的指令可以继续进入流水线，此时需要在 D 级对指令进行判断，如果为乘除指令，则因乘除模块的占用而暂停，若为其他指令，其操作与乘除不相干扰（如：在 E 级进入 ALU，与 MDU 模块并行工作），可以继续执行，由此可以提升 CPU 的效率。

（三）题目描述：举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint：考虑 C 语言中字符串的情况）

由 c 语言中的字符串可知，一个英文字母占用一个字节，因此在处理字符串

时需要以字节为单位进行处理,尤其是当处理数量不为四的倍数时按字节访问在性能上更具优势。

(四) 题目描述: 在本实验中你遇到了哪些不同指令类型组合产生的冲突? 你又是如何解决的? 相应的测试样例是什么样的?

与 p5 类似的, 将指令分为如下几类:

load: lw, lb, lbu, lh, lhu

store: sw, sb, sh

branch: beq, bne, bgtz, bgez, bltz, blez

calc_r: R 型 ALU 计算类指令 (包括 sll, sllv 移位指令, 不包括几个特别的, 如: jalr、mult 等)

calc_i: I 类指令, 除了 lui

MDU 指令: mult, multu, div, divu, mtlo, mthi, mflo, mfhi

跳转指令: j, jal, jr, jalr

对乘除法指令添加新的 Tnew: 要使用 hi 寄存器的 TNEW_HI; 要是用 lo 寄存器的 TNEW_LO。这里实际上 TNEW_HI 和 TNEW_LO 都是为指令 mfhi、mflo 准备的。本质上 HI 和 LO 的 Tnew 是相同的, 但是为了便于转发和区分写入 GRF 的数据来源, 在本 cpu 中将 TNEW_HI 和 TNEW_LO 设为不同的值。

其他指令方法与 p5 基本一致, 具体见文档前面 Tuse Tnew 表格。

(五) 题目描述: 为了对抗复杂性你采取了哪些抽象和规范手段? 这些手段在译码和处理数据冲突的时候有什么样的特点与帮助?

首先对于控制信号, 本 cpu 采用分布式译码, 通过流水指令信号到每一级流水线寄存器, 再在每一级流水线寄存器之后实例化 decoder 译码器译码出当前流水级所需要的控制信号, 减少了控制信号的传递。

其次, 在 b 类指令 (beq、bne、bgtz、bgez、blez、bltz) 的 RTL 语言中, 可以发现, 除了判定成功分支的条件有些许不同以外, 其他基本相同。所以只需要在 CMP 模块内对 beq 判断从 grf 读入的两个源操作数 V1、V2 是否相同; 对 bne 判断 V1、V2 是否不相同, 其他指令以此类推。

再次, 在 ALU 的输入端口中增加了指令的 shamt 字段, 使 ALU 同时支持

sll 和 sllv 运算，降低模块间的耦合度。