单周期处理器设计文档

一、 模块规格

1. Instruction Fetch Unit（取指令单元）

1) 端口定义

表 1 IFU 的端口定义

名称	方向	端口规格	功能描述
sign_imm32	input	[31:0]	传入符号扩展后的 offset，以适应 beq 指令
addr26	input	[25:0]	传入 j 型指令的 26 位 addr，以适应 j 型指令
ra32	input	[31:0]	传入寄存器读出的 32 位 addr，以适应 jr 指令
PC_choice	input	[1:0]	传入 PC 的四种转移规则选择信号
clk	input	[0:0]	时钟信号
reset	input	[0:0]	同步复位信号，驱动 PC 同步复位成 0x3000
funct	output	[5:0]	当前指令的 funct 段。（[5:0]段）
shamt	output	[4:0]	当前指令的 shamt 段。（[10:6]段）
rd	output	[4:0]	当前指令的 rd 段。（[15:11]段）
rt	output	[4:0]	当前指令的 rt 段。（[20:16]段）
rs	output	[4:0]	当前指令的 rs 段。（[25:21]段）
opcode	output	[5:0]	当前指令的 opcode 段。（[31:26]段）
PC+4	output	[31:0]	下一条指令的地址，以适应 jal 指令

2) 功能说明

IF 中由 PC 计数器、PC 转移器、指令存储器等组成。

PC 的初始值为 0x00003000。当时钟上升沿来临时，将其更新为 PC 转移器所计算出的下一个 PC 值。当 reset 信号有效且上升沿时，将 PC 计数器同步复位为 0x00003000 值。PC 计数器的 32bit 输出经过截取后传给指令存储器，成为读取指令的 10bit 地址。

PC 转移器根据当前的 PC 状态、若干不同种类的立即数分别计算出顺序执行、分支执行、跳转执行、跳转寄存器执行的下一个 PC 值，并由 PC_choice 信

号选择其中一路成为 PC 的下一个状态：

PC_choice = 0 时，对应顺序执行：PC’ = PC + 4；

PC_choice = 1 时，对应 beq 成立：PC’ = PC + 4 + sign_imm32*4；

PC_choice = 2 时，对应 j 型指令：PC’ = PC[31:28]||addr26||00；

PC_choice = 3 时，对应 jr 指令： PC’ = ra32。

指令存储器从 PC 计数器截取 10bit 指令地址，从 1024*32bit 的 ROM 中读取 32 位指令并输出。

总体来说，取指令单元由时钟信号和复位信号控制，在每一个上升沿输出待执行的机器码指令。指令的执行顺序由 PC 计数器控制，PC 的次态由当前态、立即数和指令的类型决定。

2. General Register File（通用寄存器堆）

1) 端口定义

表 2 GRF 的端口定义

名称	方向	端口规格	功能描述
A1	input	[4:0]	读取地址为 A1
A2	input	[4:0]	读取地址为 A2
A3	input	[4:0]	写入 32bit 数据的目的地址 A3
WD	input	[31:0]	写入到 A3 的 32bit 数据
clk	input	[0:0]	时钟信号
reset	input	[0:0]	同步复位信号
WE	input	[0:0]	写使能信号。为 1 时可写，为 0 时忽略 WD
RD1	output	[31:0]	从 A1 读取的 32bit 数据
RD2	output	[31:0]	从 A2 读取的 32bit 数据

2) 功能说明

GRF 由具有写使能的 32 个寄存器组成，每个寄存器均为 32bit，均初始化为 0，其中 0 号寄存器（地址为 5bit 00000）始终为常数 0，忽略对其的任何修改。

其余寄存器由 5 位地址编码，从 00000 到 11111。在每时每刻，A1 传入的地址对应的寄存器的 32bit 值输出到 RD1 中，A2 传入的地址对应的寄存器的 32bit 值输出到 RD2 中。

在时钟 clk 处于上升沿且写使能 WE=1 时，寄存器将当前时刻地址为 A3 的寄存器中写入 32bit 数据 WD。

当同步复位信号 reset=1 时，所有寄存器在 clk 上升沿复位成 0 值。

总体来说，GRF 的两个读取操作（RD1 = reg[A1], RD2 = reg[A2]）在每时每刻都生效，GRF 的一个写入操作（reg[A3] = WD）当且仅当 clk 为上升沿且 WE=1 时生效。

3. Arithmetic Logical Unit（算术逻辑单元）

1) 端口定义

表 3 ALU 的端口定义

名称	方向	端口规格	功能描述
A	input	[31:0]	运算对象 1
B	input	[31:0]	运算对象 2
ALUControl	input	[2:0]	ALU 运算类型控制标记，支持 5 种运算
C	output	[31:0]	由对象 1 和 2 得到的运算结果
Zero	output	[0:0]	结果是否为 0：为 0 则 Zero=1，否则 Zero=0

2) 功能说明

该模块为纯组合逻辑，与时钟无关。

本 ALU 实现暂不检测溢出。

当 ALUControl = 010 时，C = A + B（简单二进制加法，不考虑溢出）。

当 ALUControl = 110 时，C = A - B（简单二进制减法，不考虑溢出）。

当 ALUControl = 000 时，C = A & B（按位与）。

当 ALUControl = 001 时，C = A | B（按位或）。

当 ALUControl = 111 时，C = A < B（小于置位，若 A 小于 B，则 C 置为 1；否则 C 置为 0）。

无论 ALU 进行什么运算操作，在任何时刻 Zero = 1 当且仅当 C = 0；在任何时刻 Zero = 0 当且仅当 C ≠ 0。

4. Data Memory（数据内存）

1) 端口定义

表 4 DM 的端口定义

名称	方向	端口规格	功能描述
Addr	input	[31:0]	读取/写入的目标地址，按字节寻址
WD	input	[31:0]	待写入的 32bit 数据
clk	input	[0:0]	时钟信号
reset	input	[0:0]	同步复位信号，驱动清零内存
WE	input	[0:0]	写使能信号。为 1 时可写，为 0 时忽略 WD
RD	output	[31:0]	从地址 Addr 读取出的 32bit 数据

2) 功能说明

DM 由 1024 个字段组成，每个字段均为 32bit，均初始化为 0。

DM 在同一时刻只能有意义地支持读取和写入其中一种操作。

当 WE = 0 时，截取 Addr 的[11:2]位得到按字寻址的地址，读取该地址，从 RD 输出。此为读取。

当 WE = 1 时，若 clk 处于上升沿，则更新 Addr 对应的字段数据为 WD 中获取的数据。下一时刻 clk 上升沿过去，RD 立即读取 Addr 的数据。此为写入。

当同步复位信号 reset=1 时，所有字段在 clk 上升沿复位成 0 值，回到初始状态。

总体来说，DM 的读取操作（RD1= mem[Addr]）在 WE=0 时生效，DM 的写入操作（mem[Addr] = WD）当且仅当 clk 为上升沿且 WE=1 时生效。

5. Immediate Calculator（立即数计算器）

1) 端口定义

表 5 ImmCalc 的端口定义

名称	方向	端口规格	功能描述
imm16	input	[15:0]	I 指令中的 16 位立即数
sign_imm32	output	[31:0]	符号扩展后的立即数
zero_imm32	output	[31:0]	零扩展后的立即数
lui_imm32	output	[31:0]	将 imm16 加载至高 16 位后的立即数

2) 功能说明

I 指令的 16 位立即数可能作为内存操作指令 lw/sw 的带符号偏移量、ori 的无符号参数或是 lui 的参数等。它们分别需要对立即数进行符号扩展、零扩展、加载至高位的操作。

立即数计算器 ImmCalc 获取一个 16bit 立即数 imm16，并将其符号扩展、零扩展、加载至高位的结果 sign_imm32、zero_imm32、lui_imm32 分别输出。

6. Controller（控制器）

1) 端口定义

表 6 Controller 的端口定义

名称	方向	端口规格	功能描述
opcode	input	[5:0]	32bit 指令中的 opcode 段
funct	input	[5:0]	32bit 指令中的 funct 段
MemtoReg	output	[0:0]	为 0 时，GRF 接收 ALU 运算结果； 为 1 时，GRF 接收 DM 读取结果。
MemWrite	output	[0:0]	DM 的写使能信号
Branch	output	[0:0]	**优先级低**于跳转指令相关信号。非跳转指令时，Branch 为 0 时，PC 的变换规则为 PC+4；Branch 为 1 时，若 beq 的两个源寄存器数值相等，则 PC 的变换规则为 PC+4+Imm*4，否则为 PC+4。
ALUControl	output	[2:0]	ALU 的运算控制信号

ALUSrc	output	[1:0]	ALU 的运算对象 2 的来源。
RegDst	output	[0:0]	GRF 的写入对象。为 0 时为 rt；为 1 时为 rd
RegWrite	output	[0:0]	GRF 的写使能信号
Jump	output	[0:0]	当 JR=0 时，Jump 为 0 时，PC 转移由 Branch 决定；Jump 为 1 时，按照 26 位立即数的拼接法转移 PC。
Link	output	[0:0]	为 0 时，无特殊操作； 为 1 时，将 PC+4 存储到$ra 中。
JR	output	[0:0]	为 0 时，无特殊操作； 为 1 时，令 PC' = ra32。

2) 功能说明

Controller 根据 32bit 指令的 opcode 段和 funct 段，指令特异性地确定各个控制信号。

MemtoReg 控制 GRF 接收的 WD（写入数据）是 ALU 的运算结果 C，或是 DM 的读取结果 RD。

MemWrite 直接传给 DM 的 WE，控制 DM 是否写使能。

Branch 决定当前命令除跳转指令外，是不是分支指令。当不是跳转指令时：若不是分支指令，则 Branch = 0，则 PC 的运算规则一定为 PC' = PC+4。若是，则 Branch = 1，则 PC 的运算规则可能为 PC' = PC+4，也可能为 PC' = PC+4，取决于 beq 条件的成立与否。

ALUControl 控制 ALU 的运算类型，直接传给 ALU 的 ALUControl。

ALUSrc 控制 ALU 的第二个运算对象的来源。即 ALU 的 B 可能接收 GRF 的读出，也可能接收立即数的符号扩展、零扩展、加载高位。

RegDst 控制 GRF 写入的目的地址 A3。可能向 rt 寄存器写入，也可能向 rd 寄存器写入。

RegWrite 直接传给 GRF 的 WE，控制 GRF 是否写使能。

Jump、Link、JR 共同决定了 j、jal、jr 的 PC 变化和存储特性，为 PC 的转移增加了两条规则。

总体来说，Controller 特异性识别指令，从而输出控制信号，控制数据通路中数据的流动和运算。

具体地，如何根据某条指令的机器码和其意义设计控制信号，见下文“控制器设计”中阐述。

二、 数据通路设计

表 7 数据通路设计表

op	NPC	GRF				ALU		DM	
	Imm	A1	A2	**A3**	**WD**	A	**B**	Addr	WD
R	/	rs	rt	rd	ALU.C	GRF.RD1	GRF.RD2	/	/
ori	/	rs	/	rt	ALU.C	GRF.RD1	IMM.Zero	/	/
lw	/	rs	/	rt	DM.RD	GRF.RD1	IMM.Sign	ALU.C	/
sw	/	rs	rt	/	/	GRF.RD1	IMM.Sign	ALU.C	GRF.RD2
lui	/	rs	/	rt	ALU.C	GRF.RD1	IMM.Lui	/	/
jal	INS.26	/	/	$ra	IFU.PCp4	/	/	/	/
j	INS.26	/	/	/	/	/	/	/	/
jr	ALU.C	rs	rt	/	/	GRF.RD1	GRF.RD2	/	/
beq	IMM.Sign	rs	rt	/	/	GRF.RD1	GRF.RD2	/	/

因此，在 GRF.A3、GRF.WD、ALU.B、NPC.Imm 等四处应该设置由控制信号选择的多路选择器。

三、 控制器设计

1. 真值表设计

此处的真值表指的是对于每一个(opcode, funct)的组合，都有一组确定的控制信号的组合。不同指令的不同控制信号组合并起来就成为了一张真值表。

支持指令集{addu, subu, ori, lw, sw, beq, lui, nop, j, jal, jr}的一张可能的真值表如下：

表 8 指令-控制信号真值表

funct	100001	100011	/	/	/	/	000000
opcode	000000	000000	001101	100011	101011	001111	000000
	addu	**subu**	**ori**	**lw**	**sw**	**lui**	**nop**
MemtoReg	0	0	0	1	x	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	0	x
ALUControl	010	110	001	010	010	010	xxx
ALUSrc	00	00	10	01	01	11	xx
RegDst	1	1	0	0	x	0	x
RegWrite	1	1	1	1	0	1	x
Jump	0						
Link	0						
JR	0						

funct	/	/	001000	/
opcode	000011	000010	000000	000100
	jal	**j**	**jr**	**beq**
MemtoReg	0	0	0	x
MemWrite	0	0	0	0
Branch	0	0	0	1
ALUControl	000	000	010	110
ALUSrc	00	00	00	00
RegDst	0	0	0	x
RegWrite	1	0	0	0
Jump	1	1	0	0
Link	1	0	0	0
JR	0	0	1	0

2. 控制信号设计规则

1) MemtoReg

MemtoReg 直接表示了是否将内存中数据存入寄存器中。因此当且仅当指令为 lw 时，才使 MemtoReg=1。

为方便简化真值表，当 RegWrite=0 时，MemtoReg 的值不必须为 0。而当 RegWrite=1 时，除 lw 以外的其他指令的 MemtoReg 的值必须为 0。

2) MemWrite

MemWrite 直接表示了是否将寄存器中数据存入内存中。因此当且仅当指令为 sw 时，才使 MemWrite=1。

其余时刻，由于 DM 的 WD 段直接连接到了寄存器的 RD2 输出，则 MemWrite 必须为 0。

3) Branch

Branch 直接表示了是否为分支指令。因此所有可能按照 PC' = PC+4+Imm*4 规则跳转的指令，Branch 都应为 1。其余指令除 nop 外，Branch 都应该为 0。nop 对应的 Branch 值无所谓，因其 Imm=0。

4) ALUControl

ALUControl 对于运算型指令，直接设为其指令意义对应的运算标志。

对于需要计算地址的 lw/sw 指令，由于有 addr+offset 的计算需求，故设为“+”运算对应的标志。

对于 beq 指令，由于有判断(GRF[rs]-GRF[rt])==0 的需求，故设为“-”运算对应的标志。

5) ALUSrc

ALUSrc 表示 ALU 的运算对象是从寄存器取出（R 型指令和 beq 指令），还是从立即数运算获得（I 型指令）。

对于 I 指令，考察其指令的具体意义，可以判断其立即数需要符号扩展、零扩展还是加载到高位，由此设置 ALUSrc。

6) RegDst

RegDst 表示存入寄存器的地址为 rt 还是 rd。对于 R 型指令，目的寄存器为 rd；对于 I 型指令，目的寄存器为 rt。

为方便简化真值表，当 RegWrite=0 时，RegDst 的值任意。

7) RegWrite

RegWrite 表示是否 GRF 写使能。对于跳转指令、sw 指令、nop 指令等不操作寄存器的指令，RegWrite=0。

8) Jump

Jump 直接表示了是否为跳转指令。因此所有应该按照 PC' = addr32 规则跳转的指令，Jump 都应为 1。其余指令的 Jump 均应该为 0。

9) Link

Link 直接决定了 GRF 参与储存的地址和数据是正常结果，还是$31 和 PC+4。当且仅当 jal 指令 Link 信号为 1。

10) JR

JR 直接决定了 PC'的第四种转移规则，故当且仅当 jr 指令 JR 信号为 1。

3. 控制器设计：从机器码到控制信号

对于某一条指令，其意义和控制信号被其 opcode（和 funct）唯一确定。

故控制器可由指令的识别和信号的生成两部分构成。

1) **指令的识别**

以 lw 指令的 opcode "100011"为例，采用以下的与门进行特异性识别指令：

一个 lw 识别器应当其 opcode=100011 时输出 1，其他时候输出 0。因此构造以下结构：

```
assign lw_Detector = op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];
```

对于 R 指令，其识别器应针对其 funct 码类似构造。

2) **信号的生成**

由真值表我们知道，lw 对应的信号可合并为 1000100101。因此可构造如下信号生成器：

```
assign lw_Signal = SignExtend(lw_Detector) & 10'b1000100101;
```

这样，对于每一个指令就有一个 10 位 Signal 信号。它们其中有且只有一个为目的信号，其余的均为 10 位 0。这样，构造一个或门即可得到最终信号：

```
assign Signal = lw_Signal | sw_Signal | beq_Signal |…;
```

四、 测试 CPU

1. 测试代码

测试指令集：{addu, subu, ori, lw, sw, beq, lui, nop, j, jal, jr}

测试 MIPS 代码：

```
jal loadimm
   save:
   sw $t0, 0($s0)    # m1 = 7       65536    7     65536
   sw $t1, 4($s0)    # m2 = 55      62       55       62
   sw $t2, 8($s0)    # m3 = 62      55       62       55
   sw $t3, 12($s0)   # m4 = 65536   7        65536    7

   lw $t3, 0($s0)    # t3 = 7       65536    7     65536
   lw $t2, 4($s0)    # t2 = 55      62       55       62
   lw $t1, 8($s0)    # t1 = 62      55       62       55
   lw $t0, 12($s0)   # t0 = 65536   7        65536    7

   subu $t4, $t1, $t2   # t4 = 7    -7       7        -7
   beq $t4, $t3, save   # back      down     back  down

   j save               #              back          back

   loadimm:
      ori $t0, $0, 7       # t0 = 7
      ori $t1, $t0, 50     # t1 = 55
      addu $t2, $t0, $t1   # t2 = 62
```

```
nop

lui $t3, 1            # t3 = 65536
subu $t4, $t3, $t2    # t4 = 65474
nop

ori $s0, $0, 4        # s0 = 4
jr $ra
```

2. 预期结果

首先，jal 直接跳转至 `loadimm` 标签后的指令。$t0, $t1, $t2, $t3, $t4, $s0 寄存器分别按时间顺序获得值 7, 55, 62, 65536, 65474, 4。之后 jr 直接跳回 `save` 标签后的指令。

之后，DM 中按字寻址的第 1 位存储 7，第 2 位存储 55，第 3 位存储 62，第 4 位存储 65536。后将 7、55、62、65536 分别读到$t3, $t2, $t1, $t0 中。

之后，$t4 = 62-55 = 7，与$t3 相等。代码重复从标签 `save` 执行。

第二次执行后，$t4 = 55-62 = -7，与$t3 不相等，代码顺序向下执行。

遇到 j 指令，重新跳回 `save` 标签后的指令。完成一个循环周期。

之后代码将无限循环下去，每一个循环周期内 lw/sw 代码段被执行两次。

五、 思考题

1. 数据通路设计

1) input [11:2] addr

可以直接连线到 32 位的 wire [31:0] addr 上，并自动切片成 10 位信号：

```
DM(.addr(addr32))
```

如果使用 input [9:0] addr，则还要在传入信号时手动切片：

```
DM(.addr(addr32[11:2]))
```

然而，它们都不保留完整的 32 位信号。

2) 同步复位 reset

reset 针对 PC 计数器、GRF 寄存器堆、DM 数据内存进行同步复位。PC 计数器设置为 0x00003000，GRF 和 DM 全部清零。

可以认为，reset 的含义为重新运行程序，此时指令应该从第一条指令开始执行，故 PC=0x00003000。此时上次运行的全部数据影响都应该消除，故 GRF 和 DM 恢复初始状态到全零。

2. 控制器设计

1) 使用条件语句

```
always@(*) begin
    case(opcode)
    6'b100011: begin // lw
           {MtoR, MWrite, Br, ALUCtrl, ALUSrc, RDst, RWrite } =
           {1'b1, 1'b0, 1'b0, 3'b010, 2'b01, 1'b0, 1'b1};
    end
    endcase
end
```

优点：拓展指令和拓展信号方便。无需改变 Controller 电路。

缺点：没什么缺点。

2) 使用 assign 语句仿照 Logisim 的与或门阵列

```
assign lw_Detector = op[5] & !op[4] & !op[3] & !op[2] & op[1] &
op[0];
output ALUSrc;
assign ALUSrc = ori_Detector | lw_Detector | sw_Detector | … ;
```

优点：纯组合逻辑，在信号的更新时间上不易出错。

缺点：不够直观，与真值表对应关系弱，不好 debug。

3) 使用 assign 语句传入控制信号拼接的常数

```
assign lw_Detector = op[5] & !op[4] & !op[3] & !op[2] & op[1] &
op[0];
```

```
output [9:0] signals;
assign signals = (10'b10001001010 & SignExt10(lw_Detector)) |
(10'b01001001000 & SignExt10(sw_Detector)) | …;
```

优点：与真值表对应关系强，纯组合逻辑。

缺点：扩展信号位数时较为麻烦。

3. 综合

1) 忽略溢出时的 add(i) 和 add(i)u

add 和 addu：

设被加数为$A = a|x, B = b|y$，其中a, b均为最高一位。

忽略溢出和最高位的 CarryOut 时，add 的结果为$C = a|A + b|B$为 33 位数据。当存入目的寄存器时，只保留了低 32 位。a, b和$A + B$的进位均因为截取被忽略，得到的结果即为$A + B$忽略进位的结果，与 addu 相同。

addi 和 addiu：

设被加数为$A = a|x$ 和 i。其中a为最高一位。

所以将i符号扩展到 33 位，相当于先符号扩展至 32 位y，再令$I = b|y$。其中b为y的最高一位。

忽略溢出和最高位的 CarryOut 时，addi 的结果为$C = a|A + I = a|A + b|y$为 33 位数据。当存入目的寄存器时，只保留了低 32 位。a, b和$A + y$的进位均因为截取被忽略，得到的结果即为$A + y$忽略进位的结果。

而 addiu 也是先符号扩展i至 32 位y。故与 addi 结果相同。

2) 单周期处理器的优缺点

优点：实现简单

缺点：每条指令都跑满了整个数据通路和计算单元，但是可能大部分信息是没有用的，被 RegWrite 和 MemWrite 忽略掉的。然而无论如何一条指令都要等待一个时钟周期才能完成，而一个时钟周期内必须完成全部数据通路，而不是只完成所需要的部分通路。因此计算效率不高，相当于将所有指令都视为像 lw 一样最费时间的指令。这样并不符合 common case fast 原则。

3) jal, jr 与栈的关系

jal 和 jr 必须搭配栈使用。jal 将返回地址存在$ra 中，jr 读取$ra 中的地址。

然而在递归等使用场景中，$ra 只有一个，但是层层调用的返回地址却不止一个。由于在一次 call function 和 return 中，只用到一个返回地址关系，所以可以将该层及其以上所有层的 ra 记录到栈中，再将$ra 设为新的地址。

由于函数调用和返回的路径特性和栈的 push 和 pop 一致，所以使用栈来存储$ra 的历史值。