P4-CPU设计文档

#Verilog #CPU

---

1 Notation

> 命名规则参考:
> CO教程要求
> Digital Design and Computer Architecture

1.1 DataBus

- Instr: 32 位指令信号，将被分为
  - opcode
  - rs
  - rt
  - rd
  - shamt
  - funct
  - imm16
  - imm26
- pc: 32 位程序计数器信号
- npc: 32位下一时钟刻程序计数器信号
- pc4：始终为PC + 4

- ExtImm：EXT输出完成扩展后的立即数
- RegAddr: GRF 5 位写入地址
- RegData: GRF 32 位写入数据
- srcA：ALU写入数据源A
- srcB：ALU写入数据源B
- outC：ALU输出数据源C
- ALUsig：ALU特殊计算标志位，处理Zero或其余标志信号
- MemAddr: DM 32 位写入地址
- MemData: DM 32 位写入数据
- ReadData：DM 32位输出数据

1.2 ControlBus

- RegWrite: GRF 写入控制信号
- MemWrite: DM 写入控制信号
- ALUOp：ALU操作信号
- NPCOp：NPC操作信号
- EXTOp：EXT操作信号
- RegDst：GRF.A3选择数据源
- RegSrc：GRF选择数据源
- ALUSrc：ALU选择数据源

---

2 Requirement

2.1 课程组

- 处理器为 32 位单周期处理器，**不考虑延迟槽**，应支持的指令集为：`add, sub, ori, lw, sw, beq, lui, jal, jr, nop`，其中：
- `nop` 为空指令，机器码 `0x00000000`，不进行任何有效行为（修改寄存器等）。
- `add, sub` 按无符号加减法处理（不考虑溢出）
- `IM` = 4096 ∗ 32 bits，4098 = 2^{12}
- `DM` = 3072 ∗ 32 bits，2^{11} < 3072 < 2^{12}
- 顶层文件为 **mips.v**，有效的驱动信号要求包括且仅包括**同步复位信号 reset** 和**时钟信号 clk**，接口定义如下：

```
module mips(
        input clk,
        input reset
);
```

- 评测要求：使用`$readmemh`来读入code.txt进行IM导入，code.txt要求使用hex指令 + 每条指令一行
- 对于GRF or DM写入数据，使用`$display`

2.2 Keyword

1. **Instruction**:`add, sub, ori, lw, sw, beq, lui, jal, jr, nop`

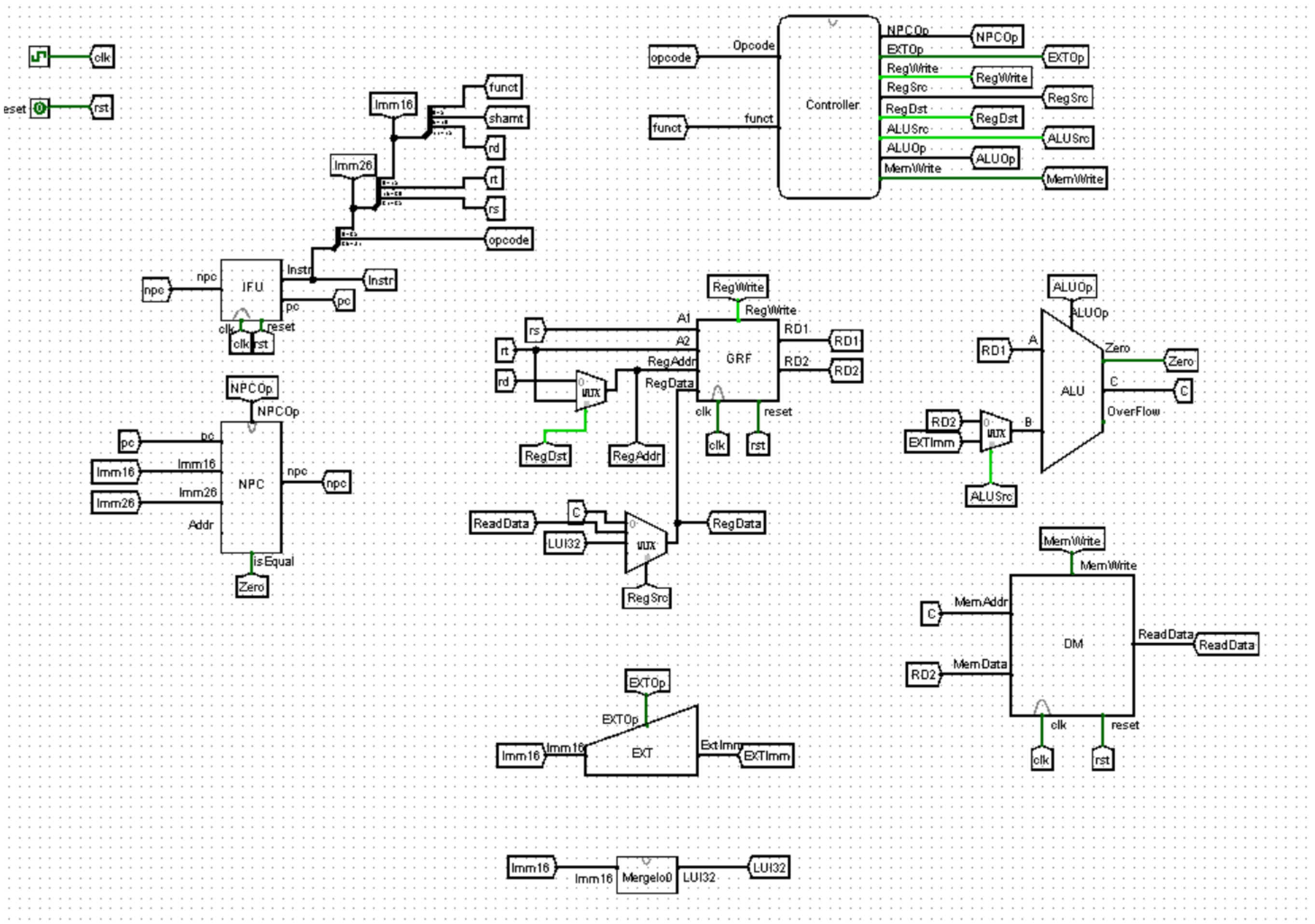
2. no delay branch
3. no **Overflow**
4. **synchronize** reset
5. Top module: `mips.v`
6. use `$readmemh` in IFU(IM)
7. use `$display` in GRF & DM

---

3 Heirarchy Designation

3.1 Buses

整体DataBus + ControlBus参考P3搭建的Logisim电路图



> 在顶层模块`mips.v`中，我需要对于上图中做的翻译仅仅为：将所有的模块进行实例化，使用`wire`变量表示图中的tunnel

3.2 Modification

1. 决定删除Mergelo模块，直接接入到ALU中，单独设置ALUOp进行处理

    > 当时觉得与或门阵列内加入ALU相关控制信号，会造成ALU编码混乱+接线麻烦的问题，在Verilog中依赖代码描述 + 宏感觉实现起来会非常方便

2. P3课上经验吸取：
   对于ALU设置一个4-bit特殊判断位，负责输出如Zero的信号，参考之前pre-challenge，**使用One-heat形式作为标记，每一位作为一个标志符**，方便进行指令扩展；并将该位回传到Controller，在Controller内部结合标志符实现控制信号生成

3.3 WorkFlow

3.3.1 CPU establishment

1. 确定所有需要建立的模块与其**端口信号/功能表**
2. 根据MIPS指令集的RTL描述填写**工程化设计数据通路表**
3. 根据数据通路表 + RTL描述，写出**Controller控制信号表**

4. 根据控制信号表编写**宏定义**`head.v`
5. 在ISE中完成**一个模块**的`.v`文件编写与实现
6. 在ISE中先完成**模块的单元测试**，返回步骤5，直至全部模块搭建测试完成
7. 在`mips.v`中完成模块实例化与组装，组装要求：
    - 顶层模块**导线名称必须与模块接口同名**
    - 按照一条指令的五执行阶段完成代码编写
8. 用自动化测试程序**测试cpu**

3.3.2 Instruction extension

1. **分析并列出**扩展指令执行涉及到的**模块**与**数据通路**
2. 扩展设计模块，并确定其控制信号
3. 对该条指令填写其对应的**Controller控制信号表**
4. 对`head.v`中补充该指令的内容
5. 在**选择一个**需要更改的模块 + 顶层模块数据通路部分执行扩展修改
6. 每次进行模块修改后，进行**扩展后模块的单元测试**，**再返回步骤5**，直至修改完全部模块
7. 整体测试cpu

---

4 Module Designation

4.1 Module List

1. IFU（PC + IM）
2. EXT

3. GRF
4. ALU
5. DM
6. NPC
7. Controller

4.2 IFU

4.2.1 端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
npc[31:0]	I	下一个指令的地址
pc[31:0]	O	当前指令地址
Instr[31:0]	O	当前指令

4.2.2 特别说明

要求数据范围约束：

1. pc，npc 为 **0x0000_3000 ~ 0×0000_6FFF**
2. IM规格：16KiB（4096 × 32bit） → 2^{12}
3. 初始&复位地址：**0x0000_3000**
4. 指令按照**字为单位**进行fetch，取出addr[11:2]
5. 要求："ROM 内部的起始地址是从 0 开始的，即 ROM 的 0 位置存储的是 PC 为 0x00003000 的指令，每条指令是一

个 32bit 常数。”
6. 要求自行实现指令读入：要求使用`$readmemh`读入`code.txt`(说明见系统任务)

4.2.3 实现思路

本质上就是一个寄存器 + 大寄存器数组
依旧实现地址映射：进入的-0x0000_3000，对输出的+0x0000_3000
注意一下：

1. 同步复位的复位值；
2. 取出地址的逻辑；
3. 需要一个`initial`块来执行`$readmemh`

4.3 EXT

4.3.1 端口信号/功能表

信号	方向	描述
EXTOp	I	0则为ZeroEXT，1则为SignEXT
Imm16[15:0]	I	16位立即数
ExtImm[31:0]	O	扩展后的

4.3.2 实现方法

先做两种扩展，之后直接使用assign语句配合三目运算符即可

4.4 GRF

4.4.1 端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位信号
RegWrite	I	写入使能信号
A1[4:0]	I	Src of RD1
A2[4:0]	I	Src of RD2
RegAddr[4:0]	I	写入寄存器地址
RegData[31:0]	I	写入寄存器堆的数据
pc[31:0]	*I*	*当前程序计数器值(辅助评测)*
RD1[31:0]	O	输出1
RD2[31:0]	O	输出2

序号	功能	描述
1	写入	当**RegWrite为1**且时钟**上升沿**时，更新GRF[A3]的数据为WD
2	输出	持续输出本上升沿记录的A1/A2数据，从RD1/RD2输出

4.4.2 特别说明

1. 0号寄存器无法被更改值，始终为0
2. 对于GRF而言，如果成功写入了数据（posedge clk，reset = 0，RegWrite = 1）

需要按照格式（请注意空格）输出：

```
$display("@%h: $%d <= %h", WPC, Waddr,
WData);
```

`WPC`为指令存储地址(pc)，`Waddr`为写入寄存器地址(RegWrite)，`WData`为写入寄存器值(RegData)

4.4.3 实现思路

1. 使用`always @(posedge clk)`进行时序逻辑建模即可
2. **同步复位模式**
3. 注意下复合逻辑关系 + 辅助评测输出
4. 关注寄存器堆的定义：`reg [31:0] GRF[0:31]`

4.5 ALU

4.5.1 端口信号/功能表

信号	方向	描述
srcA[31:0]	I	数据源SrcA
srcB[31:0]	I	数据源SrcB
ALUOp[2:0]	I	选取ALU操作
outC[31:0]	O	计算结果
ALUsig[3:0]	O	标识位

ALUOp	功能	描述
3'b000	ADD加法	C = A + B
3'b001	SUB减法	C = A - B
3'b010	OR或	C = A | B
3'b011	LUI移动imm16至高位	C = (B << 16)

ALUsig	功能	描述
4'bxxx1	出现Zero	beq指令执行跳转
4'bxx1x	待扩展	
4'bx1xx	待扩展	
4'b1xxx	待扩展	

4.5.2 实现思路

没什么太需要注意的，翻译即可
ALUOp设置成3-bit的为之后扩展指令做准备
主要是关注一下ALUsig的描述方法吧，用几个单独的`assign`语句判断是否符合条件，然后按照顺序做**位拼接**`{}`。

4.6 DM

4.6.1 端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
MemWrite	I	写入/读取信号

信号	方向	描述
MemAddr[31:0]	I	写入地址
MemData[31:0]	I	写入数据
pc[31:0]	*I*	*程序计数器，辅助输出评测*
ReadData[31:0]	O	输出数据

序号	功能	描述
1	载入	当MemWrite=1时且在时钟上升沿时，载入在MemAddr处载入MemData
2	读取	持续进行数据输出，按写入地址寻址

4.6.2 特别说明

1. 起始地址：**0x0000_0000**
2. 模块规格：12KiB（3072 × 32bit）→ 2^{12}
3. 地址数据范围：**0x0000_0000 ~ 0×0000_2FFF**
4. 如果需要写入数据：每个**时钟上升沿**到来时若**要写入数据**（即写使能信号为 1 且非 reset 时）则输出写入的位置及写入的值，格式（请注意空格）为：

```
$display("@%h: *%h <= %h", pc, addr, din);
```

符号映射关系：`pc`(pc)，`addr`(MemAddr)，`din`(MemData)

4.6.3 实现方法

注意：

1. **同步**复位
2. 利用`$display`输出，关注输出条件

4.7 NPC

4.7.1 端口信号/功能表

信号	方向	描述
NPCOp[1:0]	I	操作控制码
Imm16[15:0]	I	16位立即数
Imm26[25:0]	I	j指令26位伪地址
Addr[31:0]	I	寄存器相关直接寻址
pc[31:0]	I	当前pc值
npc[31:0]	O	下一个pc
pc4[31:0]	O	常态输出PC+4的值

NPCOp	功能	描述
2'b00	PC+4	npc = pc + 4
2'b01	PC相对寻址	npc = pc + 4 + SignExt(Imm16 | 00)
2'b10	伪直接寻址	npc = pc[31:28] | Imm26 | 00
2'b11	寄存器寻址	npc = Addr

4.7.2 实现方法

需要根据Controller给出的控制信号，作为一个MUX选择四路中一路进行输出

4.8 Controller

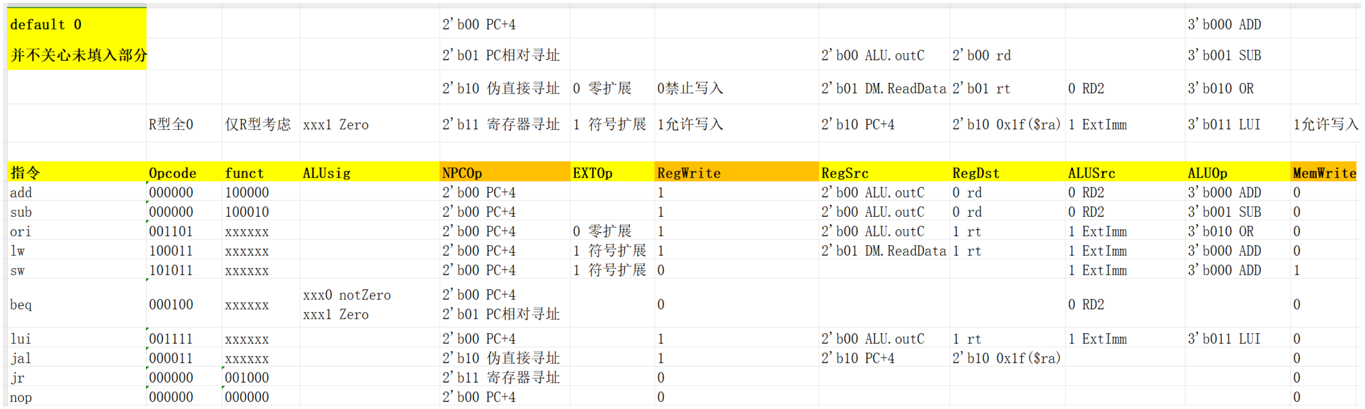
4.8.1 端口信号/功能表

信号	方向	描述
Opcode[5:0]	I	操作码
funct[5:0]	I	R型指令辅助码
ALUsig[3:0]	I	ALU特判标志信号，以独热码形式，每一位对应一种特殊情况是否发生
NPCOp[1:0]	O	NPC操作信号
EXTOp	O	EXT操作信号
RegWrite	O	GRF写入控制信号
RegDst[1:0]	O	GRF选择写入寄存器
RegSrc[1:0]	O	GRF选择RegData数据源
ALUSrc	O	ALU选择数据源
ALUOp[2:0]	O	ALU操作信号
MemWrite	O	DM 写入控制信号

序号	信号	功能
1	RegWrite	0禁止，1允许写入
2	MemWrite	0禁止，1允许写入
3	ALUOp[2:0]	ALU操作信号
4	NPCOp[1:0]	NPC操作信号
5	EXTOp	0零扩展，1符号扩展
6	RegDst[1:0]	2'b00选择rd，2'b01选择rt，2'b10选择0x1f($ra)
7	RegSrc[1:0]	2'b00选择ALU.C，2'b01选择DM.ReadData，2'b10选择PC+4
8	ALUSrc	0选择RD2，1选择ExtImm

4.8.2 实现方法

填表做可视化:



default 0				2’b00 PC+4						3’b000 ADD	
并不关心未填入部分				2’b01 PC相对寻址			2’b00 ALU.outC	2’b00 rd		3’b001 SUB	
				2’b10 伪直接寻址	0 零扩展	0禁止写入	2’b01 DM.ReadData	2’b01 rt	0 RD2	3’b010 OR	
	R型全0	仅R型考虑	xxx1 Zero	2’b11 寄存器寻址	1 符号扩展	1允许写入	2’b10 PC+4	2’b10 0x1f($ra)	1 ExtImm	3’b011 LUI	1允许写入
指令	Opcode	funct	ALUsig	NPCOp	EXTOp	RegWrite	RegSrc	RegDst	ALUSrc	ALUOp	MemWrite
add	000000	100000		2’b00 PC+4		1	2’b00 ALU.outC	0 rd	0 RD2	3’b000 ADD	0
sub	000000	100010		2’b00 PC+4		1	2’b00 ALU.outC	0 rd	0 RD2	3’b001 SUB	0
ori	001101	xxxxxx		2’b00 PC+4	0 零扩展	1	2’b00 ALU.outC	1 rt	1 ExtImm	3’b010 OR	0
lw	100011	xxxxxx		2’b00 PC+4	1 符号扩展	1	2’b01 DM.ReadData	1 rt	1 ExtImm	3’b000 ADD	0
sw	101011	xxxxxx		2’b00 PC+4	1 符号扩展	0			1 ExtImm	3’b000 ADD	1
beq	000100	xxxxxx	xxx0 notZero xxx1 Zero	2’b00 PC+4 2’b01 PC相对寻址		0			0 RD2		0
lui	001111	xxxxxx		2’b00 PC+4		1	2’b00 ALU.outC	1 rt	1 ExtImm	3’b011 LUI	0
jal	000011	xxxxxx		2’b10 伪直接寻址		1	2’b10 PC+4	2’b10 0x1f($ra)			0
jr	000000	001000		2’b11 寄存器寻址		0					0
nop	000000	000000		2’b00 PC+4		0					0

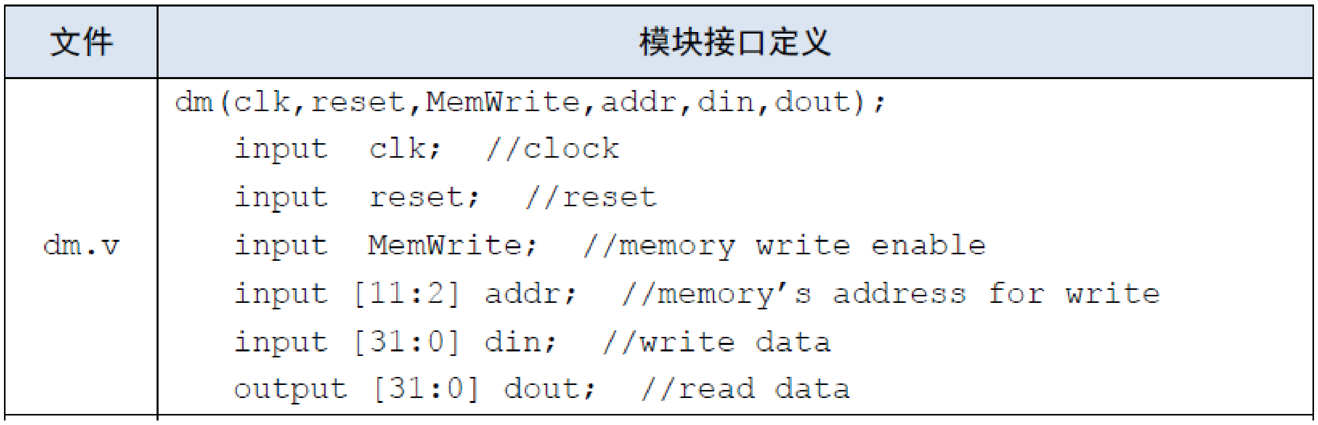
> Controller的设计模式选择：对于完成转译的每条指令，书写其对应的控制信号。
> 因为在实现指令扩展的时候，我们更多以该指令为主体，关心指令对应的控制信号取值，如此**以指令为单位进行控制信号集成**更不容易出现遗漏的bug
> 同时，考虑到其实我希望使用`always @(*)`来描述对于opcode和funct的解码与控制信号生成，如上的写法一定保证了所有输出对应的reg都被赋值了，**不会造成Latch的生成**，我认为针对我的代码描述方式，以上方法会更加安全可靠

---

5 Q & A

5.1

阅读下面给出的 DM 的输入示例中（示例 DM 容量为 4KB，即 32bit × 1024字），根据你的理解回答，这个 addr 信号又是从哪里来的？地址信号 addr 位数为什么是 [11:2] 而不是 [9:0] ？



文件	模块接口定义

dm.v	`dm(clk,reset,MemWrite,addr,din,dout);`
	`    input   clk;   //clock`
	`    input   reset;   //reset`
	`    input   MemWrite;   //memory write enable`
	`    input  [11:2]  addr;   //memory's address for write`
	`    input  [31:0]  din;   //write data`
	`    output  [31:0]  dout;   //read data`

> 对于DM而言，addr信号应该是由ALU的计算结果给出其中ALU.outC信号是32-bit的，而对于容量为4KB的DM而言，因为是按照字节寻址，所以对于DM内部连续32-bit存储单元的地址差为4，因此，ALU.outC的最低两位并无意义，通过取[11:2]与外部ALU.outC接入忽略最低两位。

5.2

思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。
实例：

```verilog
`define ADD 6'b100000
`define SW 6'b101011
// type-a
//...
if (Opcode == 6'h00 && funct == `ADD) begin
        ALUOp = 3'b000;
        RegWrite = 1'b1;
        MemWrite = 1'b0;
end
//...

// type-b
//...
wire add = (Opcode == 6'h00 && funct ==
`ADD);
wire sw = (Opcode == `SW)

ALUOp = (add || sw) ? 3'b000 : //...
//...
```

> **我采用了第一种**的方式，其实就是以指令为单位对控制信号进行集成，我认为这样的方式**更加适合对指令进行扩展**，当新增一条指令的时候，我需要进行的操作只是单独建立一个else-if逻辑块，并将对应的控制信号值填充进去，不用在整个文件中到处去找对应的控制信号代码块的位置，对于一条指令产生的控制信号都是什么更加明确。
> 第二种方式，相对而言更加符合从Logisim直接翻译过来的写法，可能主要的优势在于整个文件是简介的，因为是对于控制信号进行指令的集成，所以对于某一种控制信号对应的指令会更加明确，但我认为没有第一种更加适合扩展。

5.3

在相应的部件中，复位信号的设计都是**同步复位**，这与 P3 中的设计要求不同。请对比**同步复位**与**异步复位**这两种方式的 reset 信号与 clk 信号优先级的关系

> 这个其实很明显，
> 同步复位模式：clk 优先于 reset
> 异步复位模式：reset 优先于 clk
> 看敏感信号列表即可

```
// asynchronize
always @(posedge clk, reset) begin
        if(reset == 1'b1)
        // reset
        else
        // do sth.
end

// synchronize
always @(posedge clk) begin
        if(reset == 1'b1)
        // reset
        else
        // do sth.
end
```

5.4

C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II:

The MIPS32® Instruction Set》中相关指令的 Operation 部分。

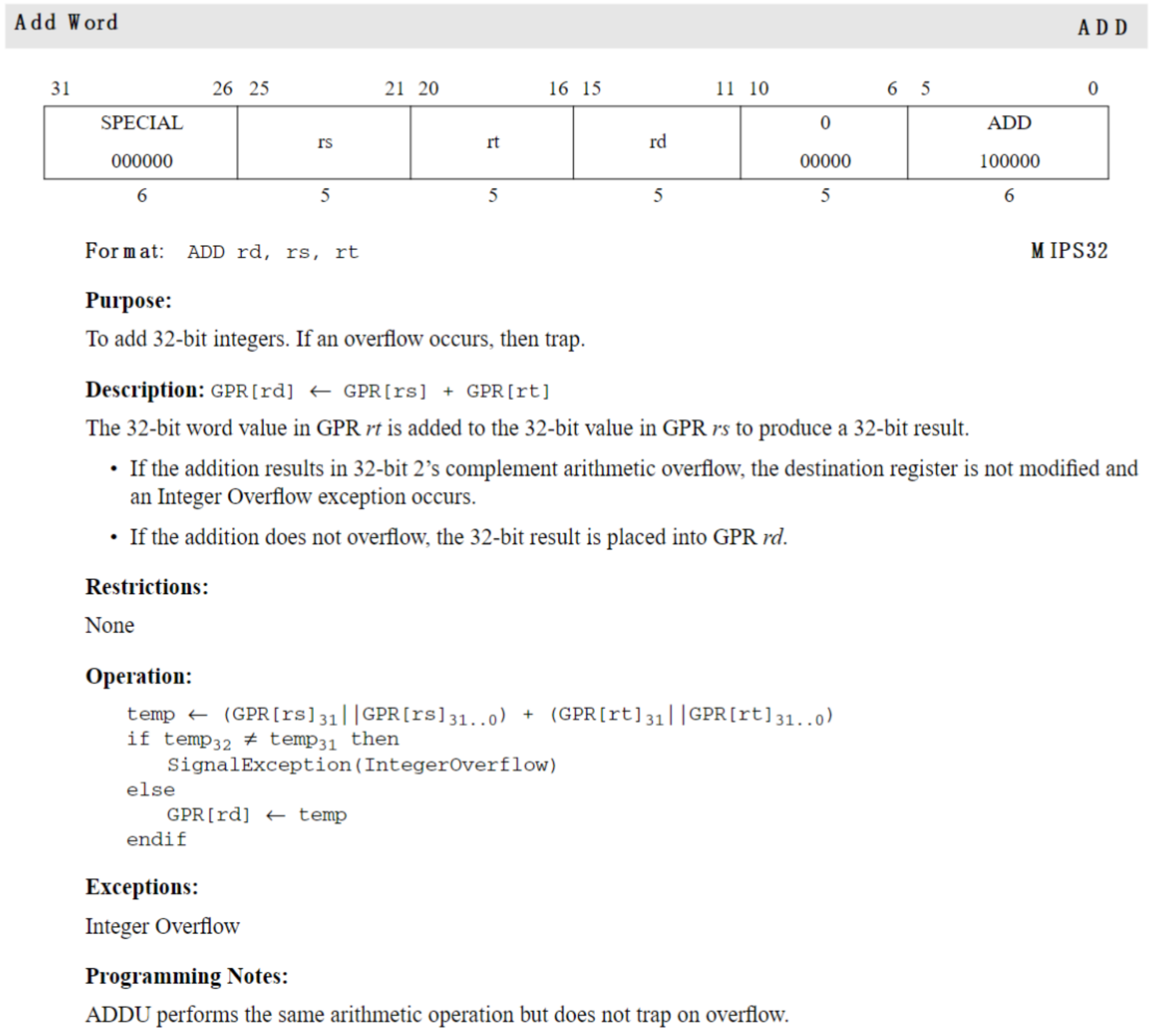
> 从RTL描述来看，其实本质上addi和addiu在行为层面上的唯一区别是：当addi和add发现出现Overflow的时候，会抛出异常，而addu和addiu不会，那么当我们忽略溢出只有，不存在异常的监测，那么自然就等价了。

Debug记录

1. 做单元测试
2. 利用ISE综合功能查看接线是否完备
3. 保证每个数字都有位宽
4. 检查所有模块定义的位宽是否存在且匹配

Reference

add



Add Word **ADD**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format: `ADD rd, rs, rt` **MIPS32**

Purpose:

To add 32-bit integers. If an overflow occurs, then trap.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← $(GPR[rs]_{31}||GPR[rs]_{31..0}) + (GPR[rt]_{31}||GPR[rt]_{31..0})$
if $temp_{32} \neq temp_{31}$ then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

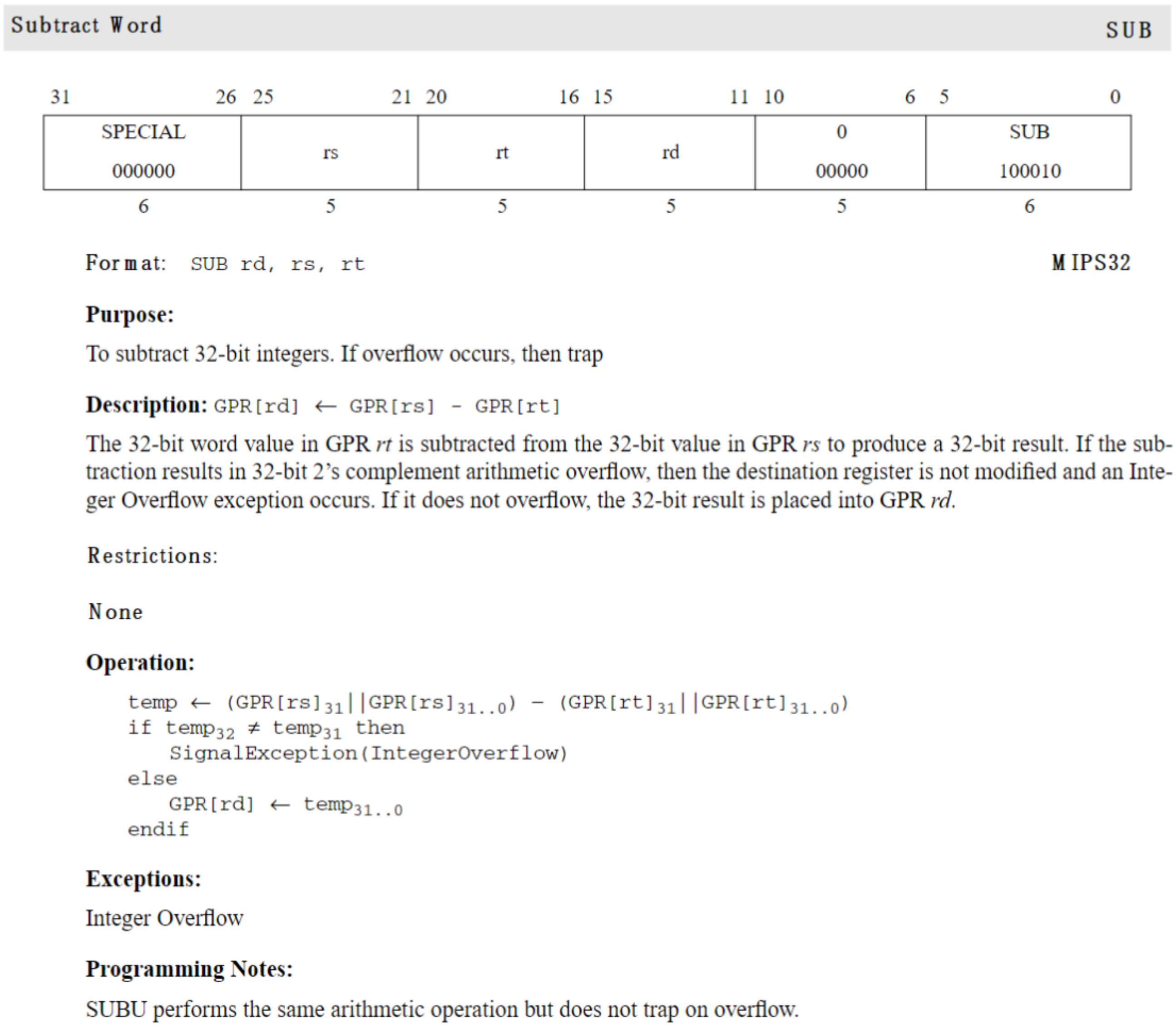
Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

sub



Subtract Word SUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6

Format: `SUB rd, rs, rt` **MIPS32**

Purpose:

To subtract 32-bit integers. If overflow occurs, then trap

Description: `GPR[rd] ← GPR[rs] - GPR[rt]`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]$_{31}$||GPR[rs]$_{31..0}$) − (GPR[rt]$_{31}$||GPR[rt]$_{31..0}$)
if temp$_{32}$ ≠ temp$_{31}$ then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp$_{31..0}$
endif
```

Exceptions:

Integer Overflow

Programming Notes:

SUBU performs the same arithmetic operation but does not trap on overflow.

ori

Or Immediate **ORI**

31 26	25 21	20 16	15 0
ORI 001101	rs	rt	immediate
6	5	5	16

Format: `ORI rt, rs, immediate` **MIPS32**

Purpose:

To do a bitwise logical OR with a constant

Description: `GPR[rt] ← GPR[rs] or immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

Restrictions:

None

Operation:

```
    GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

Exceptions:

None

---

lui

Load Upper Immediate **LUI**

31 26	25 21	20 16	15 0
LUI 001111	0 00000	rt	immediate
6	5	5	16

Format: `LUI rt, immediate` **MIPS32**

Purpose:

To load a constant into the upper half of a word

Description: `GPR[rt] ← immediate ||` 0^{16}

The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into GPR *rt*.

Restrictions:

None

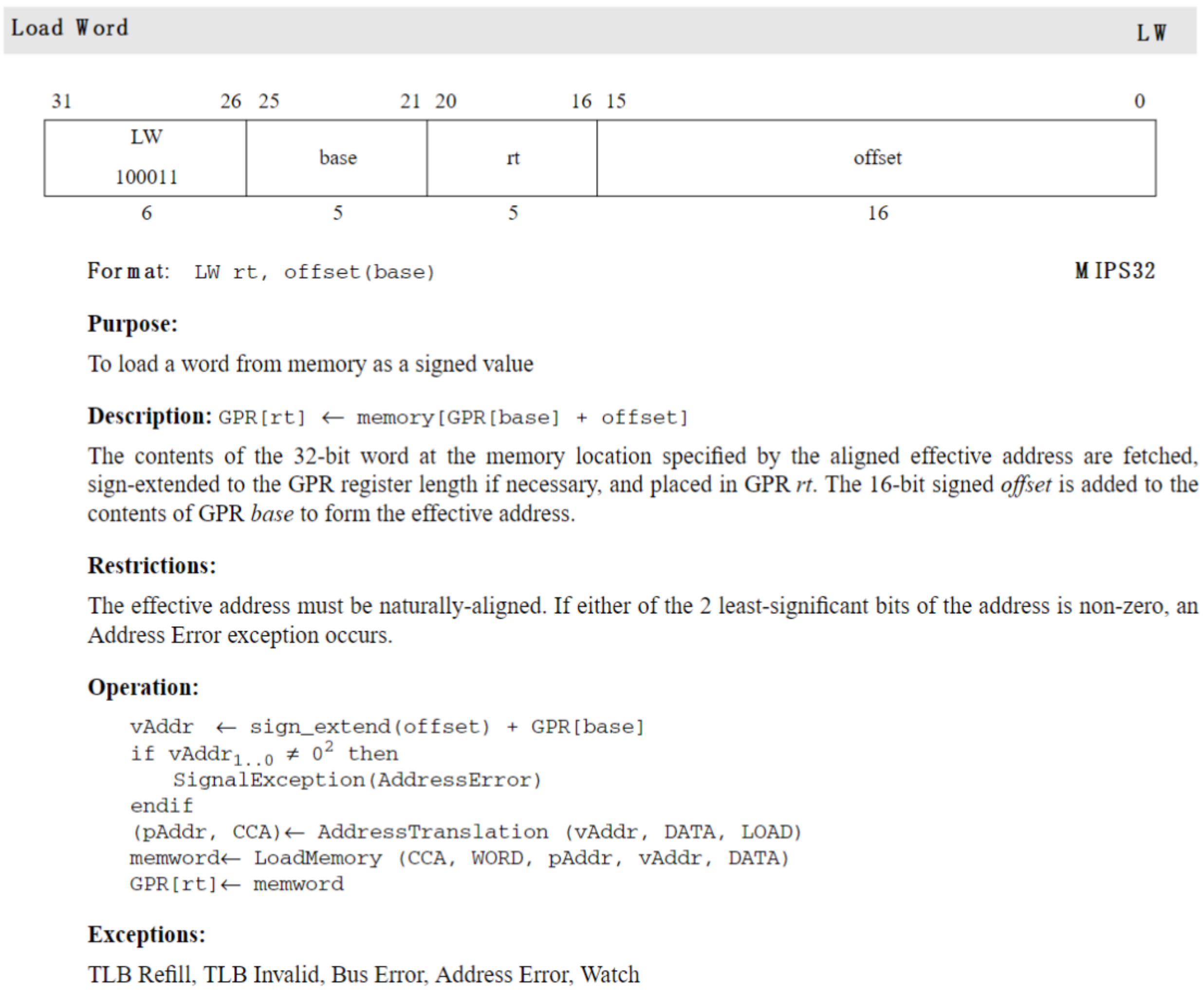
Operation:

GPR[rt] ← immediate || 0^{16}

Exceptions:

None

lw



Load Word **LW**

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

Format: `LW rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory as a signed value

Description: `GPR[rt] ← memory[GPR[base] + offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

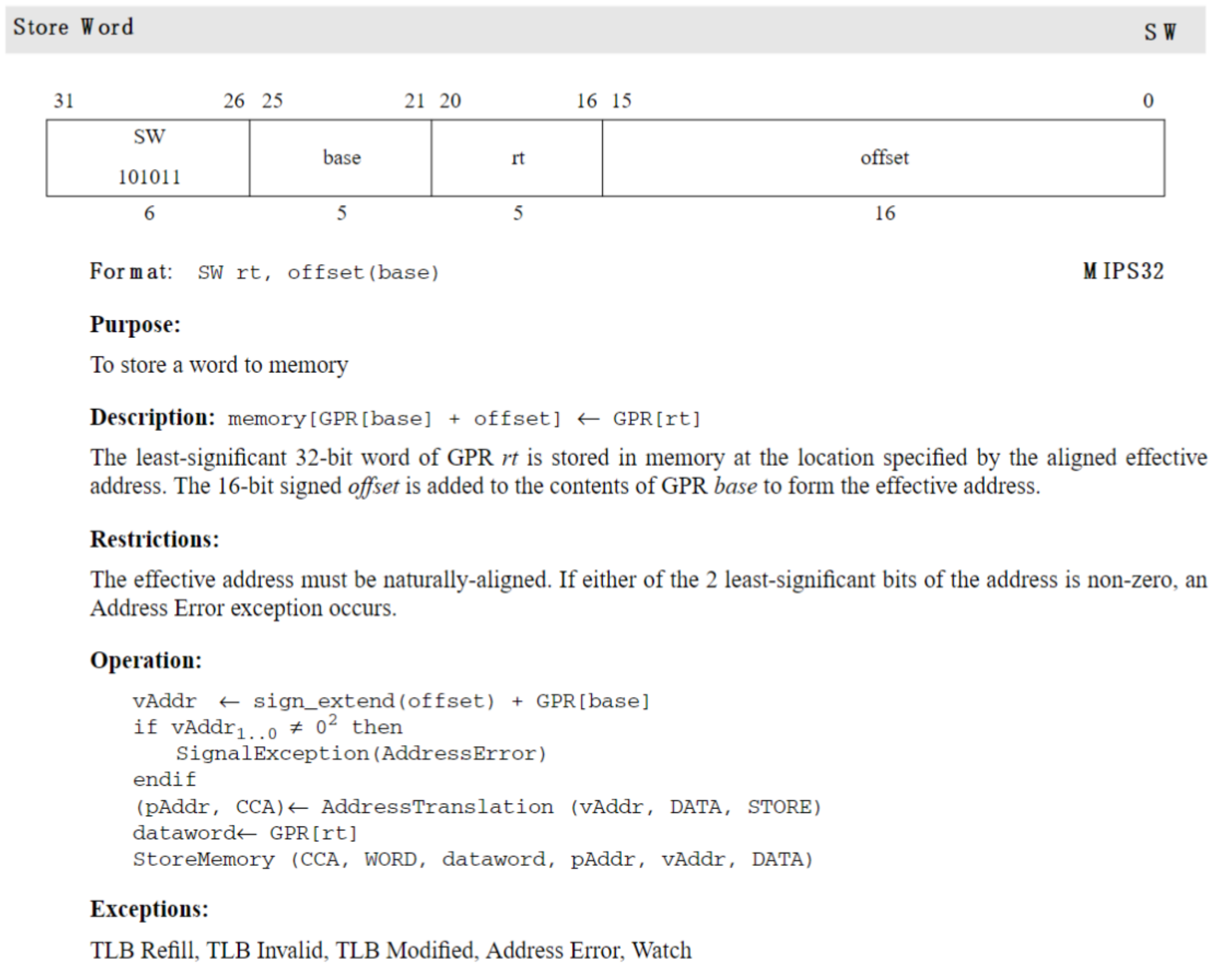
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if $vAddr_{1..0} \neq 0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

sw



Store Word SW

31–26	25–21	20–16	15–0
SW 101011	base	rt	offset
6	5	5	16

Format: `SW rt, offset(base)` MIPS32

Purpose:

To store a word to memory

Description: `memory[GPR[base] + offset] ← GPR[rt]`

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

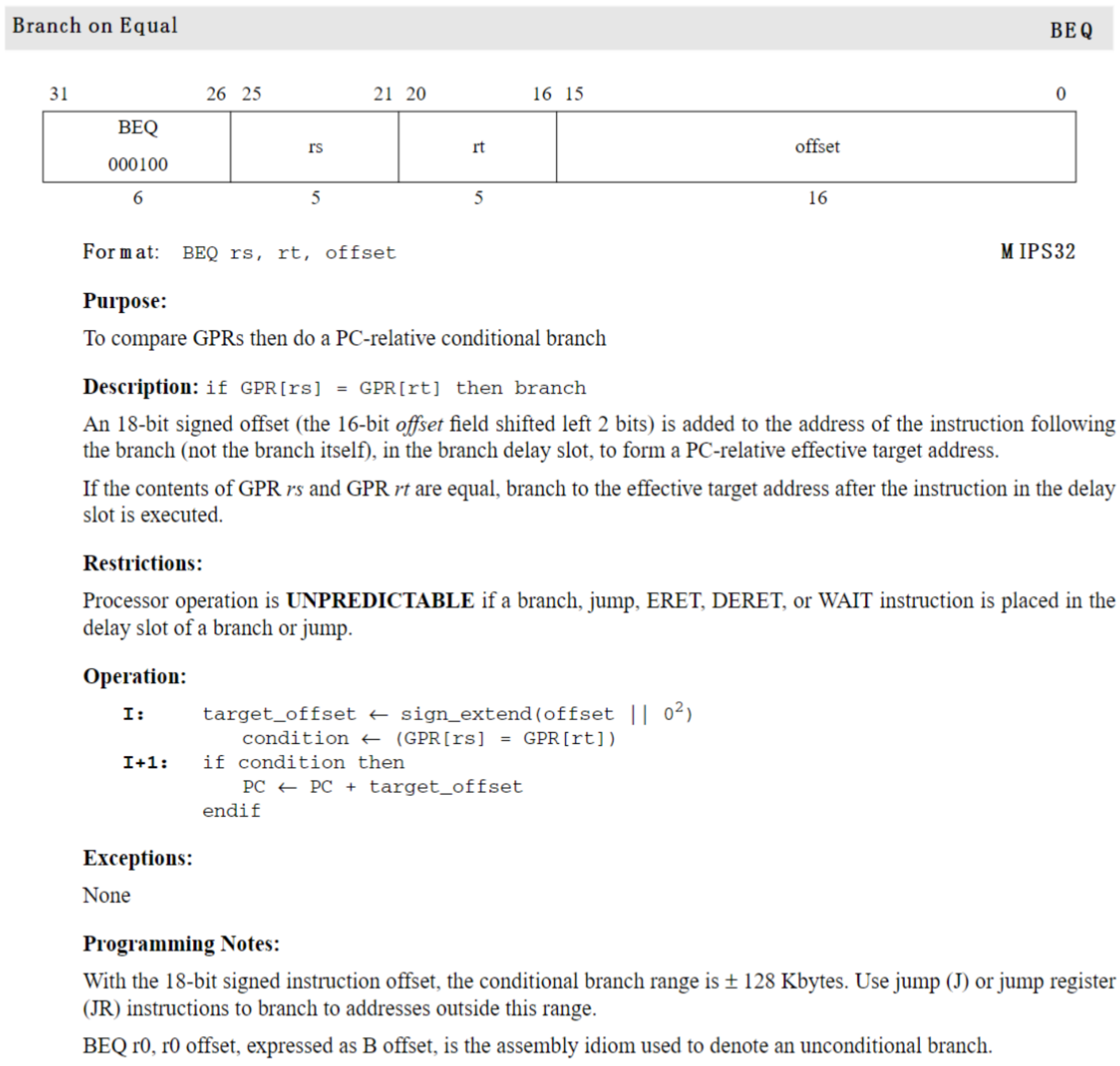
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if $vAddr_{1..0} \neq 0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

beq



Branch on Equal **BEQ**

31 ... 26	25 ... 21	20 ... 16	15 ... 0
BEQ 000100	rs	rt	offset
6	5	5	16

Format: `BEQ rs, rt, offset` **MIPS32**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if GPR[rs] = GPR[rt] then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:      target_offset ← sign_extend(offset || 0$^2$)
            condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

Exceptions:

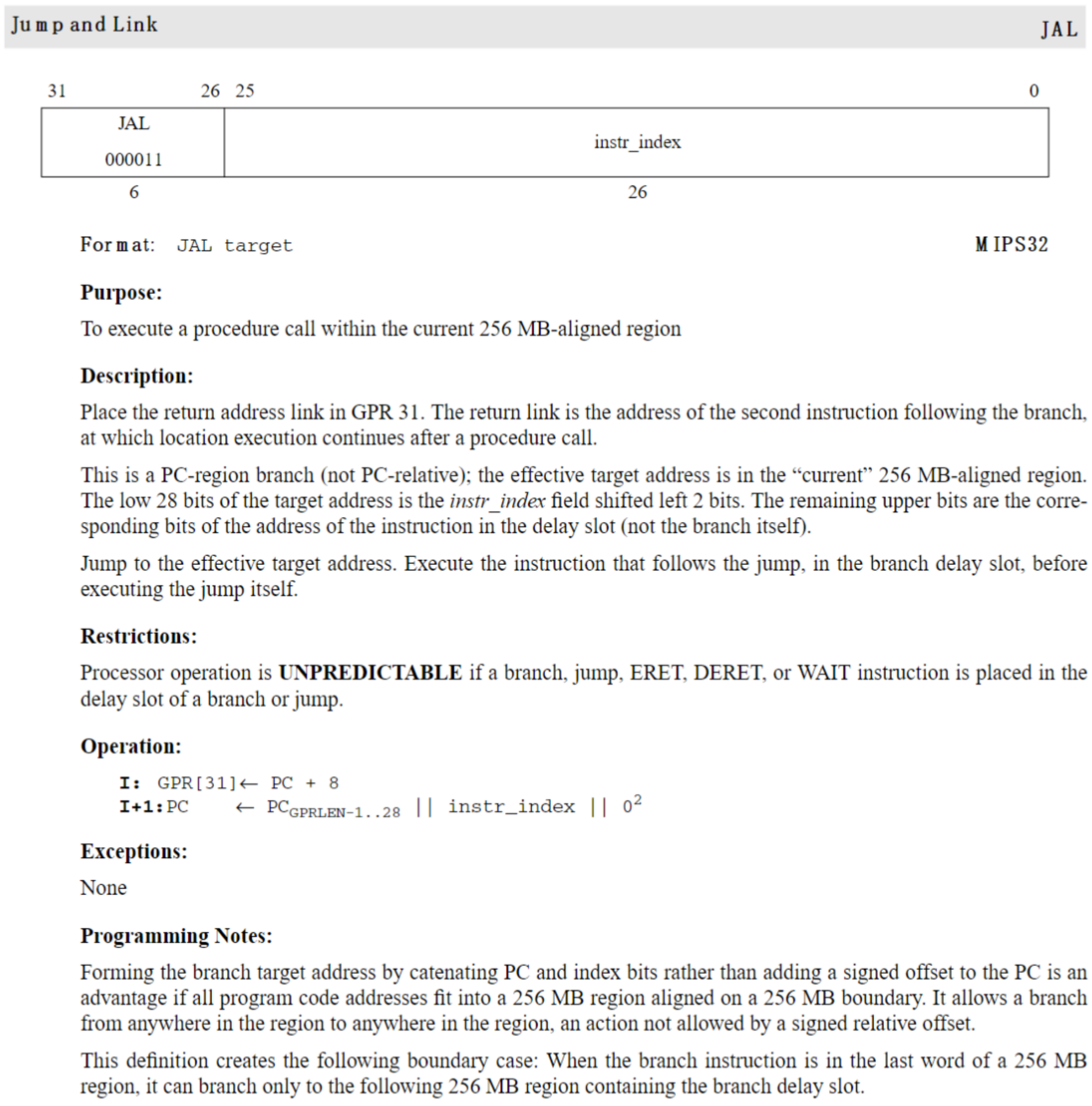
None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.

jal



Jump and Link JAL

31 26	25 0
JAL 000011	instr_index
6	26

Format: `JAL target` MIPS32

Purpose:

To execute a procedure call within the current 256 MB-aligned region

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:   GPR[31]← PC + 8
I+1: PC     ← PC$_{GPRLEN-1..28}$ || instr_index || 0$^2$
```

Exceptions:

None

Programming Notes:

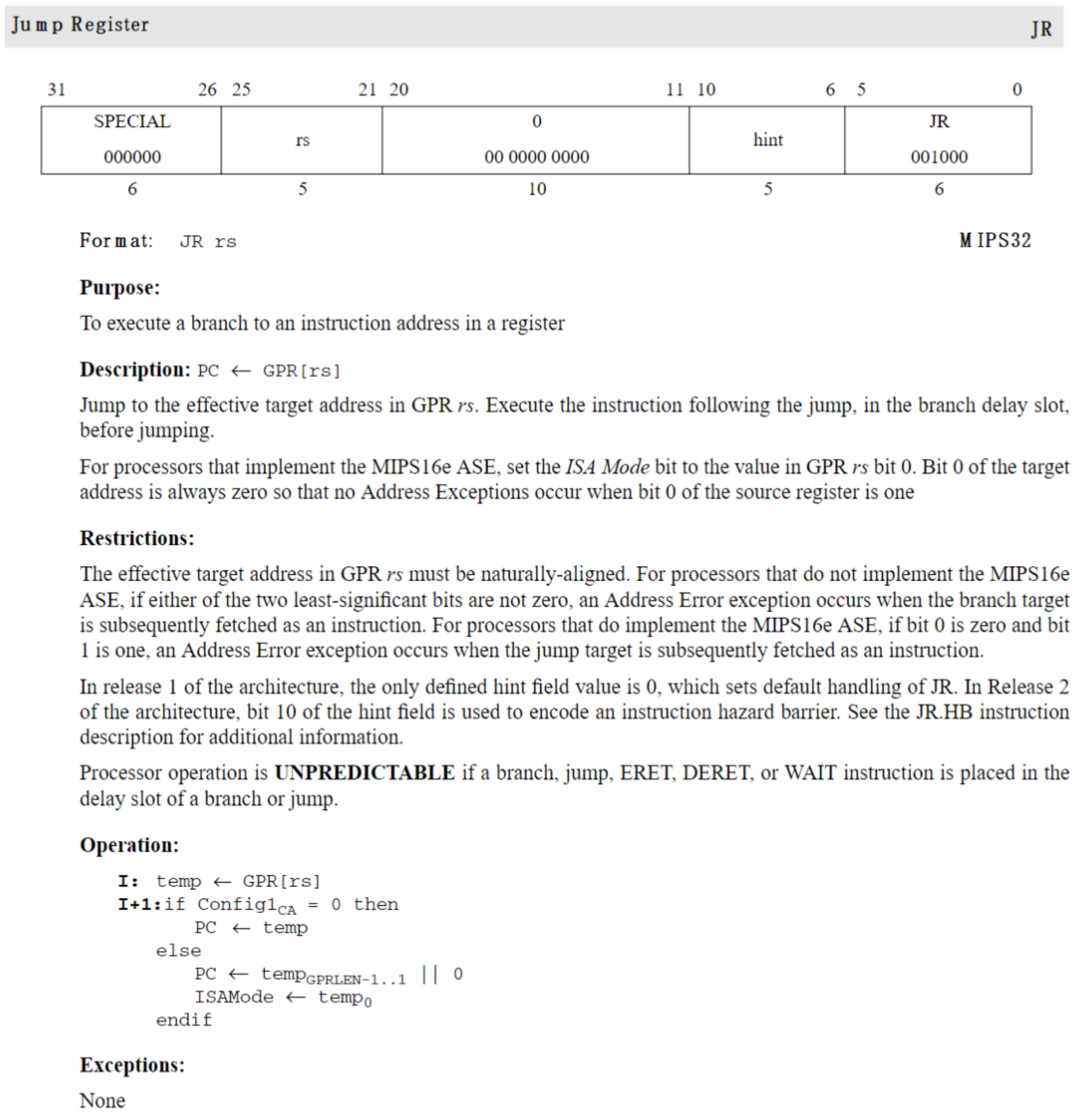
Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

> 这里使用PC + 8是因为考虑到了延时槽吧...

---

jr



Jump Register **JR**

31 26	25 21	20 11	10 6	5 0
SPECIAL 000000	rs	0 00 0000 0000	hint	JR 001000
6	5	10	5	6

Format: `JR rs` **MIPS32**

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← GPR[rs]`

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16e ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

Restrictions:

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16e ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the JR.HB instruction description for additional information.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:   temp ← GPR[rs]
I+1: if $Config1_{CA}$ = 0 then
        PC ← temp
     else
        PC ← $temp_{GPRLEN-1..1}$ || 0
        ISAMode ← $temp_0$
     endif
```

Exceptions:

None

nop

No Operation NOP

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL	0	0	0	0	SLL
000000	00000	00000	00000	00000	000000
6	5	5	5	5	6

Format: `NOP` **Assembly Idiom**

Purpose:

To perform no operation.

Description:

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

Restrictions:

None

Operation:

None

Exceptions:

None

Programming Notes:

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.