CPU设计文档

#logisim

---

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
- pc: 32 位程序计数器信号
- RegAddr: GRF 5 位写入地址
- RegData: GRF 32 位写入数据
- MemAddr: DM 32 位写入地址
- MemData: DM 32 位写入数据
- ReadData：DM 32位输出数据
- ExtImm：EXT输出完成扩展后的立即数

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
- RegSrc：GRF选择数据源
- RegDst：GRF.A3选择数据源
- ALUSrc：ALU选择数据源

---

2 Module Designation

要求实现模块：

1. IFU（PC + IM）
2. GRF
3. ALU
4. DM
5. EXT
6. Controller

自行加入设计元件：

1. NPC
2. Mergelo0（为lui指令准备）

2.1 IFU

2.1.1 端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**异步**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

2.1.2 特别说明

要求数据范围约束：

1. pc，npc 为 **0x0000_3000 ~ 0×0000_6FFF**
2. 初始地址：**0x0000_3000**
3. 使用ROM实现
4. 指令按照**字为单位**进行fetch
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。”

2.1.3 实现思路

1. 对于初始地址+地址范围要求，需要进行**地址映射**
   内部实现对pc - 0×0000_3000再使用，对结果进行+

0x0000_3000 后再接入到npc
最后实现0x0000 ~ 0×3FFF(11_1111_1111_1111)

2. 按照字为单位进行fetch，直接将**npc低两位替换为0**

2.2 GRF

> 使用p0实现版本

2.2.1 端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**异步**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
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

2.3 ALU

2.3.1 端口信号/功能表

信号	方向	描述
A[31:0]	I	数据源SrcA
B[31:0]	I	数据源SrcB
ALUOp[2:0]	I	选取ALU操作
C[31:0]	O	计算结果
Zero	O	标识位
Overflow	O	发生溢出问题

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	AND与	C = A & B

2.3.2 特别说明

加减法目前先不考虑溢出问题

2.3.3 实现思路

1. 3-bit编码预留出功能扩展空间
2. 通过编码一定程度上区分功能：算数、逻辑
3. 可以加入一个判断等器判断Zero，或者用32输入的OR

4. Overflow先一直置为0即可

2.4 DM

2.4.1 端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**异步**复位
MemWrite	I	写入/读取信号
MemAddr[31:0]	I	写入地址
MemData[31:0]	I	写入数据
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	当MemWrite=0时，进行数据输出

2.4.2 特别说明

1. 使用RAM实现，容量为 3072 × 32bit
2. 起始地址：**0x0000_0000**
3. 地址数据范围：**0x0000_0000 ~ 0×0000_2FFF**
4. 使用双端口模式：**Separate load and store ports**

2.4.3 实现方法

基本使用Logisim库内置的RAM实现即可，稍微封装一下即可

2.5 EXT

2.5.1 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

2.5.2 实现方法

用俩Bit Extender，用一个MUX选择即可

2.6 NPC

2.6.1 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

信号	方向	描述
npc[31:0]	O	下一个pc

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

2.6.2 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

2.7 MergeIo0

2.7.1 端口信号/功能表

信号	方向	描述
Imm16	I	输入的16位立即数
LUI32	O	完成低16位置0 (Imm16 || 16'h0000)

> 设计思考：这玩意感觉如果放进ALU里面好麻烦，需要单独划出一个操作码来控制它，目前ALU本身就有点复杂了，后续可能还要加入移位元件。感觉在ALU内部用移位元件 + 多余操作码控制完全没必要，而且就`lui`一条指令会用到，干脆直接外部多个元件，RegSrc的MUX多条通路完事了

2.8 Controller

2.8.1 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
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
RegSrc[1:0]	O	GRF选择数据源
RegDst	O	GRF.RegData选择数据源
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

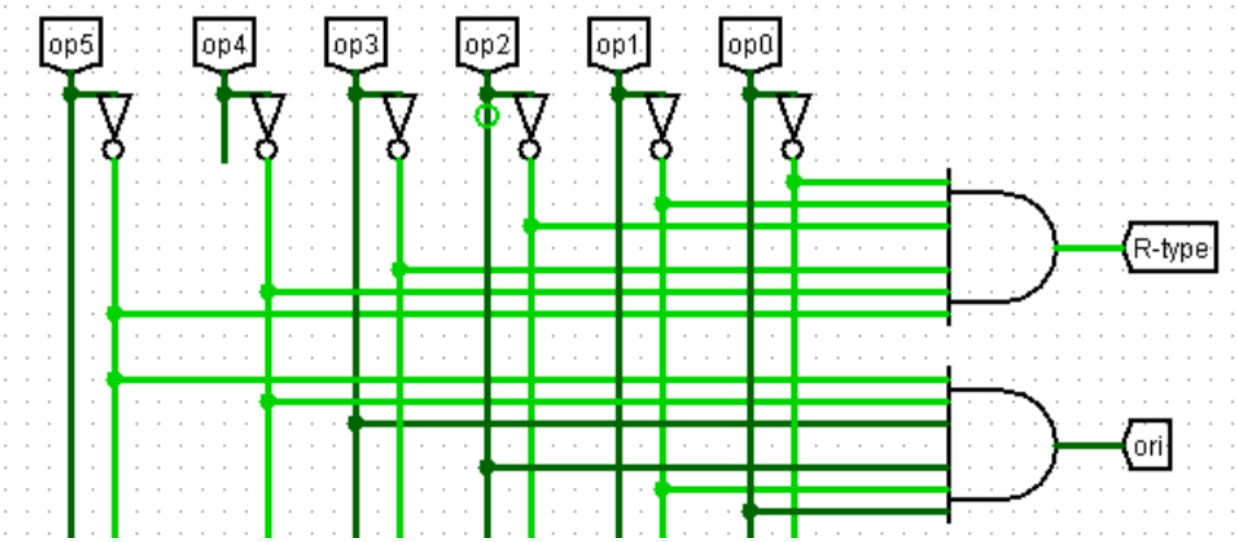
序号	信号	功能
1	RegWrite	1允许写入
2	MemWrite	1允许写入
3	ALUOp[2:0]	ALU操作信号
4	NPCOp[1:0]	NPC操作信号
5	EXTOp	0零扩展，1符号扩展
6	RegSrc[1:0]	2'b00选择ALU.C，2'b01选择DM.ReadData，2'b10选择Mergelo0.LUI32
7	RegDst	0选择rd，1选择rt
8	ALUSrc	0选择RD2，1选择ExtImm

填表做可视化

default 0 并不关心未填入部分			2'b00 PC+4						3'b000 ADD	
			2'b01 PC相对寻址			2'b00 ALU.C			3'b001 SUB	
			2'b10 伪直接寻址	0 零扩展		2'b01 DM.ReadData	0 rd	0 RD2	3'b010 OR	
	R型全0	仅R型考虑	2'b11 寄存器寻址	1 符号扩展	1允许写入	2'b10 Mergelo0.LUI32	1 rt	1 ExtImm	3'b011 AND	1允许写入
指令	Opcode	funct	NPCOp	EXTOp	RegWrite	RegSrc	RegDst	ALUSrc	ALUOp	MemWrite
add	000000	100000	2'b00 PC+4		1	2'b00 ALU.C	0 rd	0 RD2	3'b000 ADD	0
sub	000000	100010	2'b00 PC+4		1	2'b00 ALU.C	0 rd	0 RD2	3'b001 SUB	0
ori	001101	xxxxxx	2'b00 PC+4	0 零扩展	1	2'b00 ALU.C	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	2'b01 PC相对寻址		0			0 RD2		0
lui	001111	xxxxxx	2'b00 PC+4		1	2'b10 Mergelo0.LUI32	1 rt			0
nop	000000	000000	2'b00 PC+4		0					0

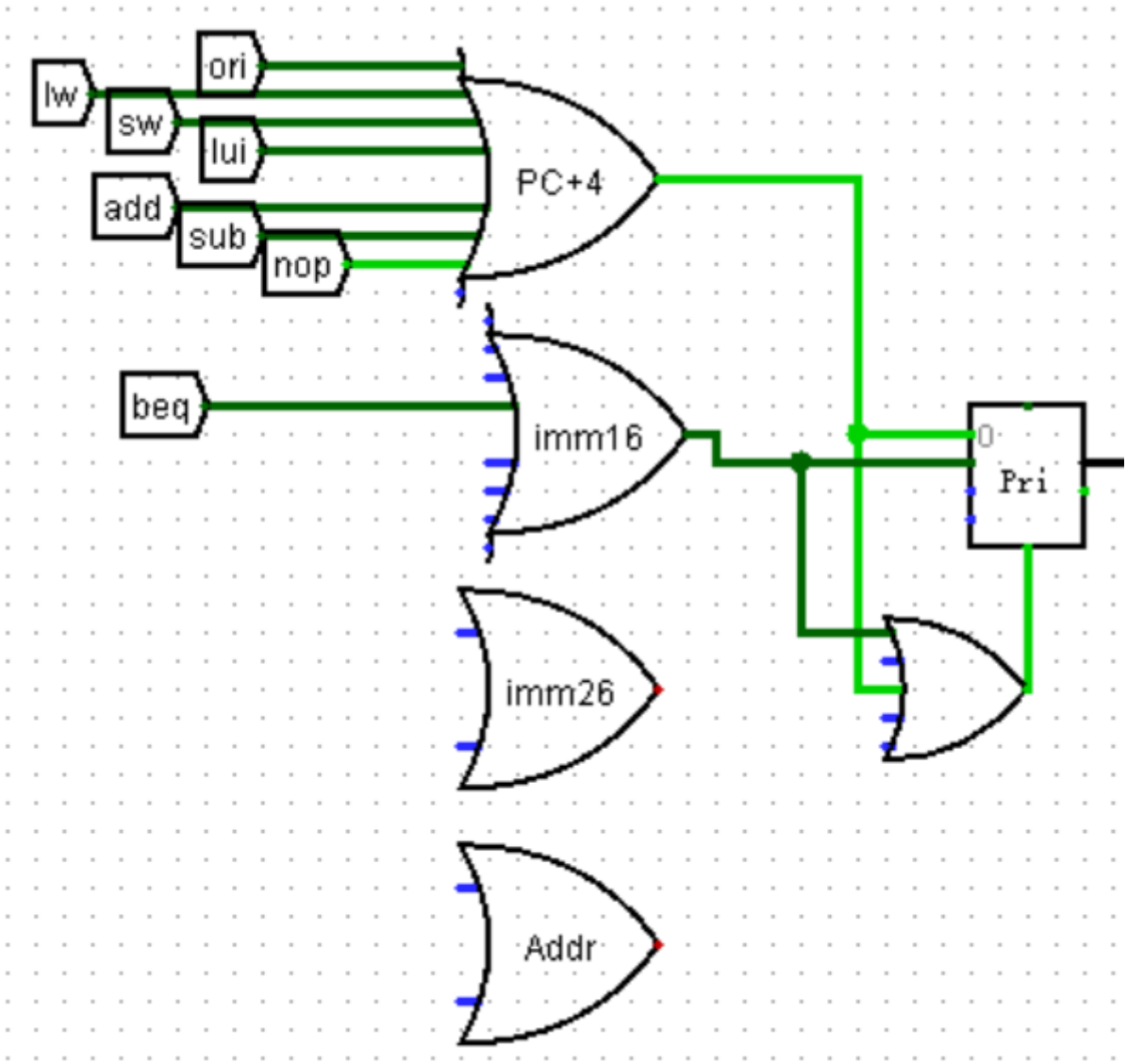
2.8.2 实现方法

1. 用AND门阵列确定指令类型



2. 根据指令类型，通过OR阵列部分控制信号
3. 对于多位需要编码的控制信号，利用coder进行编码

特殊设计，我不希望控制信号出现Floating值（感觉会莫名得有危险），人为置0



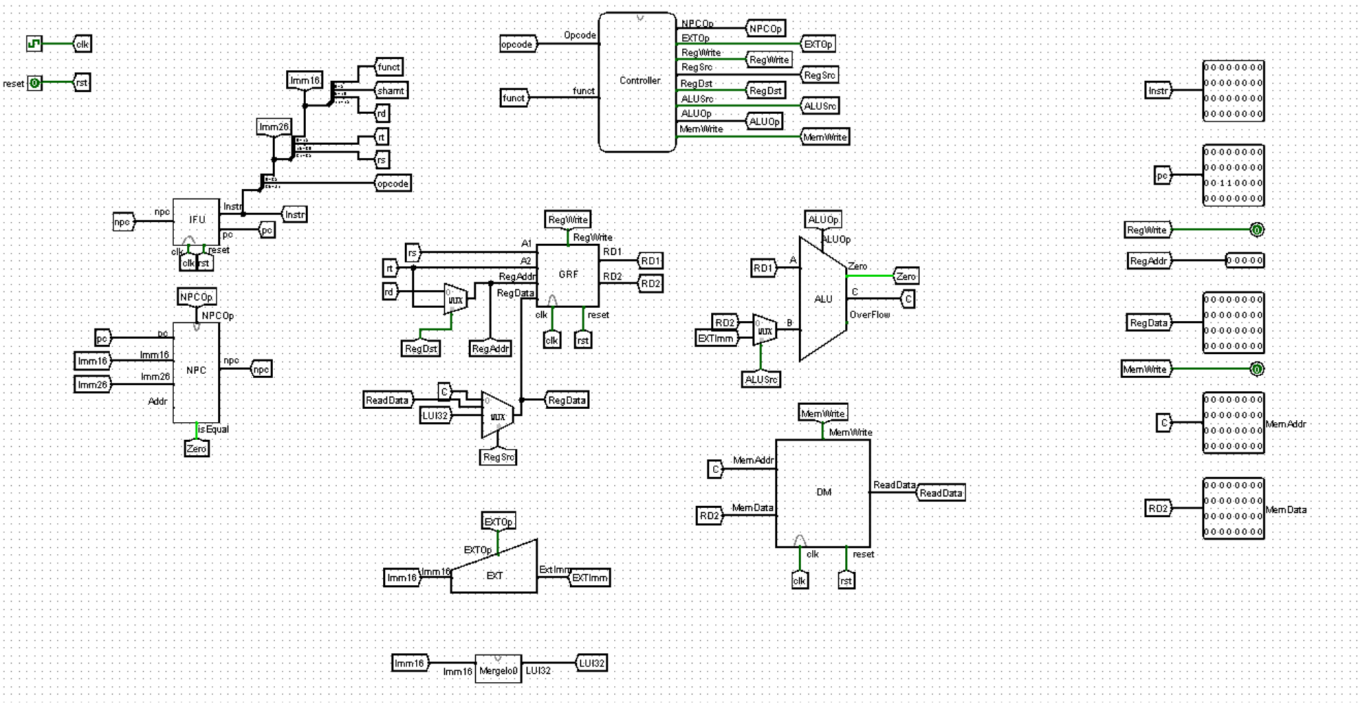
3 Put Everything Together

3.1 工程化设计分析表

指令/模块	NPC		IFU.pc	Splitter	EXT	GRFs				ALU		DM		Mergelo0
模块输入端口	PC	imm16	DI	Instr	immIn	A1	A2	A3	WD	A	B	Addr	WD	DI
add	IFU.pc	IFU.pc	NPC.out	IFU.Instr		Split.rs	Split.rt	Split.rd	ALU.C	GRFs.R1	GRFs.R2			
sub	IFU.pc	IFU.pc	NPC.out	IFU.Instr		Split.rs	Split.rt	Split.rd	ALU.C	GRFs.R1	GRFs.R2			
ori	IFU.pc	IFU.pc	NPC.out	IFU.Instr	Split.imm16	Split.rs		Split.rt	ALU.C	GRFs.R1	EXT.ExtImm			
lui	IFU.pc	IFU.pc	NPC.out	IFU.Instr				Split.rt	Mergelo0.DO					Split.imm16
beq	IFU.pc	IFU.pc	NPC.out	IFU.Instr		Split.rs	Split.rt			GRFs.R1	GRFs.R2			
lw	IFU.pc	IFU.pc	NPC.out	IFU.Instr	Split.imm16	Split.rs		Split.rt	DM.DO	GRFs.R1	EXT.ExtImm	ALU.C		
sw	IFU.pc	IFU.pc	NPC.out	IFU.Instr	Split.imm16	Split.rs	Split.rt			GRFs.R1	EXT.ExtImm	ALU.C	GRFs.R2	
nop	IFU.pc	IFU.pc	NPC.out	IFU.Instr										
整合	IFU.pc	IFU.pc	NPC.out	IFU.Instr	Split.imm16	Split.rs	Split.rt	Split.rd Split.rt	ALU.C DM.DO Mergelo0.DO	GRFs.R1	GRFs.R2 EXT.ExtImm	ALU.C	GRFs.R2	Split.imm16

3.2

基于上表，进行顶层连线



---

4 Q & A

4.1

> 1. 上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法。请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。

状态**存储**功能：

- 上游：IFU（PC）
- 下游：GRF

状态**转移**功能：

- 上游：NPC
- 下游：Controller、ALU、DM

4.2

> 2. 现在我们的模块中 IM 使用 ROM， DM 使用 RAM，GRF 使用 Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

合理：

1. 首先IM的值在目前阶段我们并不存在指令去更改它，使用不可该的ROM符合其功能特性
2. 与ROM相对的，由于我们存在lw和sw指令需要去与DM进行交互，所以必然需要可由指令修改的存储
3. GRF使用register体现的是对于容量、访存速度、价格的trade-off，Register的访存速度高但容量小，符合GRFs使用极其频繁且需要较高访存速度的特性，更加符合工业生产。

4.3

> 3. 在上述提示的模块之外，你是否在实际实现时设计了其他的模块？如果是的话，请给出介绍和设计的思路。

是的，我添加的模块为：

1. NPC

2. Mergelo0（为lui指令准备）

对于NPC而言，我对比了黑书和教程课件内容，认为统一使用NPC进行次态PC值更新的方法更方便我**简化顶层模块布线**，我认为统一处理后可以有效提升顶层模块界面简洁性，易于维护。同时，在利用NPC集成后，我认为这十分适合使用**OOpre的单元测试思想**进行测试，可以更加方便进行debug。

对于Mergelo0模块是**单独为lui指令**进行处理的。我在设计时想到了几种方案。其一为在ALU中单独设立一块shifter服务LUI指令（这个还要与sll等移位指令的shifter分开），其二在ALU内部为与移位指令共用一个shifter，其三为单独设立一个模块进行操作。
前两种方案需要单独为其设计一个ALUOp编码以完成控制，后者需要在顶层模块的RegSrc内加入一个编码来完成控制，我在权衡下选择后者，主要因为不太希望提升ALU内部复杂度，但是其实后来想想几种方法都也可以。

4.4

> 事实上，实现 `nop` 空指令，我们并不需要将它加入控制信号真值表，为什么?

这是因为nop唯一执行的就是PC + 4 → PC，控制信号没什么意义

4.5

1. 对于ori需要测试负数情况，以保证我们的实现中没有错误的选择0扩展
2. 没有对数据范围的边界值进行测试
3. 没有测试lw和sw输入正/负立即数的情况来检查我们的扩展是否存在问题
4. 没有测试对于imm最低两位非0的情况
5. 对于beq的测试没有进行对数据边界的测试

---

5 Testing

5.1 Manual Test

> 边界/特值测试手动做，随机测试由自动化程序做

单元测试 + 整体测试

> 从main模块双击后点击View可将当前子模块**连接**到当前上层模块，共享数据
> 单独点击并用左上角工具调整输入可以进行单元测试，但是如果希望与上层共享数据需要进行上述操作

测试代码编写：

PYTHON

```
# check lui & ori

# 1
lui $1, 0x0000
ori $1, 0x0001

# -1
lui $2, 0xffff
ori $2, 0xffff

# bound +
lui $3, 0x7fff
ori $3, 0xffff

# bound -
lui $4, 0x1000
ori $4, 0x0000

# check arthmetic ins
add $5, $1, $2
add $6, $1, $1
sub $7, $2, $1
sub $8, $1, $2

add $5, $3, $1
```

```
add $6, $4, $2
sub $7, $3, $2
sub $8, $4, $1

# check sw and lw
lui $5, 0x0000
ori $5, 0x2ffc
sw $3, 0($0)
sw $3, 4($0)
sw $4, 8($0)
sw $3, -4($5)
sw $4, -8($5)

lw $0, 0($0)
lw $6, 0($0)
lw $6, 4($0)
lw $6, 8($0)
lw $6, -4($5)
lw $6, -8($5)

# check beq jump
L1:
beq $2, $2, L3 # jump down
L2:
add $t1, $t1, $t1
beq $1, $1, L2
L3:
```

```
    beq $1, $3, L2
    beq $3, $3, L1 #jump up
```

5.2 Auto Test

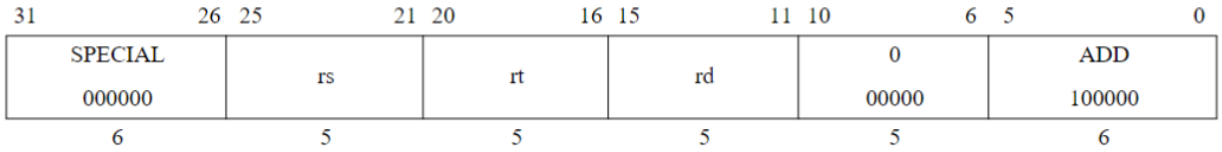
搭建样例生成器`CO-tester`，构造大量自由随机可执行样例
见**自动测试提交窗口**

Reference

---

add

Add Word **ADD**



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 temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow

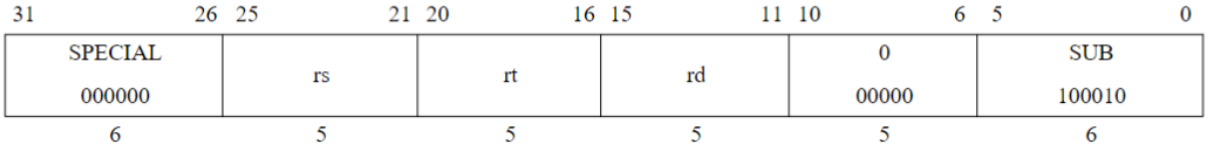
Programming Notes:

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

---

sub

Subtract Word SUB



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 temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp31..0
endif
```

Exceptions:

Integer Overflow

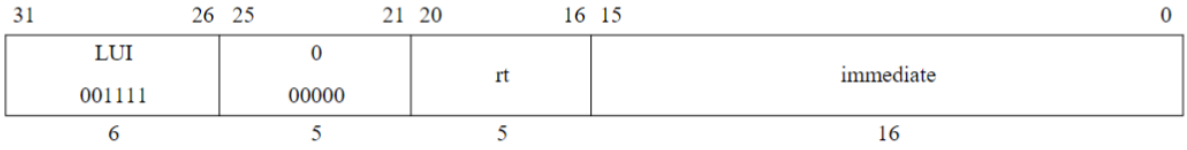
Programming Notes:

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

---

lui

Load Upper Immediate LUI



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
```

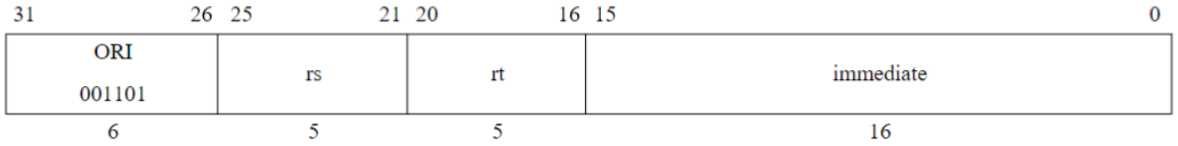
Exceptions:

None

---

ori

Or Immediate ORI



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:

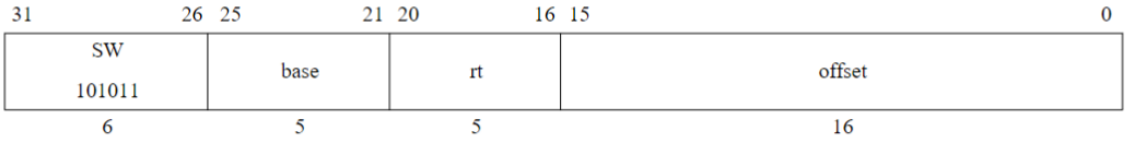
```
GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

Exceptions:

None

sw

Store Word — SW



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:

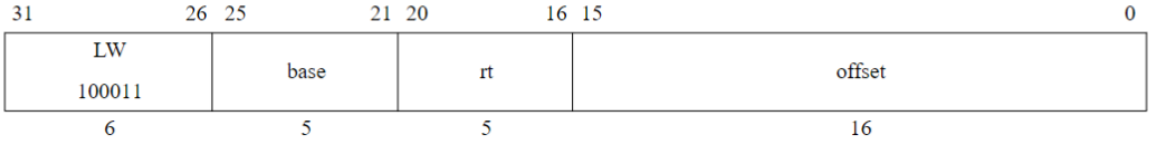
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

lw

Load Word **LW**



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 vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

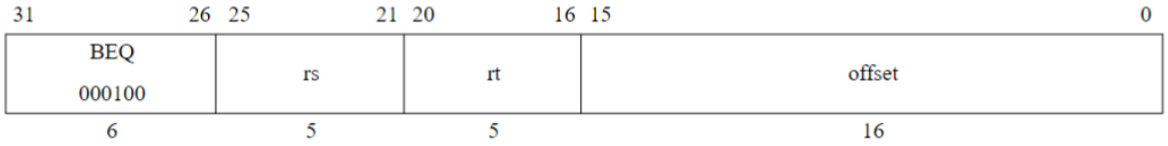
Exceptions:

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

---

beq

Branch on Equal **BEQ**



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:

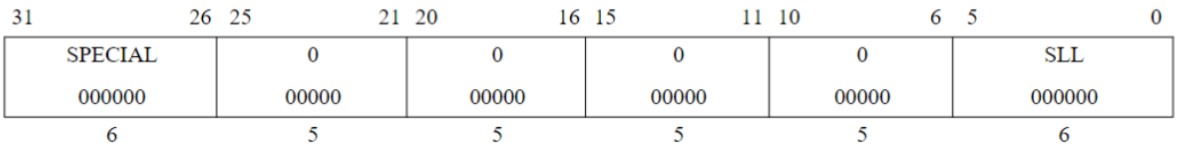
With the 18-bit signed instruction offset, the conditional branch range is ± 128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

---

nop

No Operation — NOP



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.