PARC Instruction Set Architecture

Author : Christopher Batten, Ji Kim, Berkin Ilbeyi
Date   : August 26, 2014

The PARC ISA is a subset of the MIPS32 ISA with some modifications to match the PARC architecture. It is categorized into several versions, each of which builds off of the previous version as it increases in complexity. It has several differences from MIPS32, in addition to having a different coprocessor 0 (cp0) register space. A system that implements the full PARC ISA will be able to run parallel C++ programs on a multicore architecture.

Table of Contents

1. Differences from MIPS32

The PARC ISA has several important differences from the MIPS32 ISA.

* Little-Endian

Although MIPS32 supports both big- and little-endian architectures, the PARC ISA is strictly little-endian. This means that the least significant bytes in a word are stored in the lower order addresses in memory.

* No branch delay slot

This means that link address for jal/jalr instructions needs to be PC + 4 not PC + 8. Technically, without a branch delay slot there is no reason to keep using PC + 4 for the PC relative branch and jump targets, but it simplifies the compiler so for now the following instructions all use PC + 4 as tbe base address for determining their target: jal, jalr, bne, beq, blez, bgtz, bltz, and bgez.

* No HI/LO registers

MIPS32 uses HI and LO registers to store the 64-bit results of mult, multu, div, and divu instructions. The PARC ISA has its own set of multiply/divide instructions: mul, div, divu, rem, and remu which all target a general purpose register. The PARC ISA does not have HI and LO registers. The multiply instruction only has a signed variant, whereas the divide and remainder instructions have both a signed and unsigned variant. Notice that the names for div and divu are the same but the functionality is different. The PARC multiply/divide/remainder instructions always return a 32-bit result into a general purpose register -- this means that mul will only return the lower half of the 64-bit product as the result.

* Atomic instructions

PARC support atomic instructions in PARCv4. Atomic instructions embody multiple operations that complete atomically with respect to other memory operations. Atomic instructions are important in multicore systems for efficient synchronization.

```
* Address translation

PARC does not yet have a virtual memory space, thus does not use any
address translation to access memory. Memory addresses used by processor
requests are essentially direct mappings to the physical memory, except
that the higher order bits are truncated to the length of the physical
memory address.

* Other features not included from MIPS32

 - Branch likely instructions (b*l)
 - Branch and link instructions (b*al)
 - Test and trap instructions (teq, tge, tlt, ...)
 - Unaligned loads and stores (lwl, lwr, swl, swr)
 - Merged multiply accumulates (madd, maddu, msub, msubu)
 - Rotate instructions (rotr, rotrv)
 - Bit manipulation instructions (clz, clo, ext, ins, seb, seh)
 - Load-link and store-conditional instructions (ll, sc)

--------------------------------------------------------------------------
2. Architectural State
--------------------------------------------------------------------------

* General Purpose Registers

 - 32 GPRs: PARC uses the same symbolic register names as MIPS32.

    + r0  : $zero   the constant value 0
    + r1  : $at     assembler temporary register
    + r2  : $v0     function return value
    + r3  : $v1     "
    + r4  : $a0     function argument register
    + r5  : $a1     "
    + r6  : $a2     "
    + r7  : $a3     "
    + r8  : $a4     "
    + r9  : $a5     "
    + r10 : $a6     "
    + r11 : $a7     "
    + r12 : $t4     temporary registers (callee saved)
    + r13 : $t5     "
    + r14 : $t6     "
    + r15 : $t7     "
    + r16 : $s0     saved registers (caller saved)
    + r17 : $s1     "
    + r18 : $s2     "
    + r19 : $s3     "
    + r20 : $s4     "
    + r21 : $s5     "
    + r22 : $s6     "
    + r23 : $s7     "
    + r24 : $t8     temporary registers (callee saved)
    + r25 : $t9     "
    + r26 : $k0     kernel registers
    + r27 : $k1     "
    + r28 : $gp     global pointer
    + r29 : $sp     stack pointer
    + r30 : $fp     stack frame pointer
    + r31 : $ra     return address

 - epc: exception PC (PARCv4 and higher)
    + Stores return address from exception
```

* Coprocessor 0 Registers

 - mngr2proc: cpr1 (PARCv1 and higher)

    Used to communicate data from the manager to the processor. This
    register has register-mapped FIFO-dequeue semantics meaning reading
    the register essentially dequeues the data from the head of a FIFO.
    Reading the register will stall if the FIFO has no valid data.
    Writing the register is undefined.

 - proc2mngr: cpr2 (PARCv1 and higher)

    Used to communicate data from the processor to the manager. This
    register has register-mapped FIFO-enqueue semantics meaning writing
    the register essentially enqueues the data on the tail of a FIFO.
    Writing the register will stall if the FIFO is not ready. Reading the
    register is undefined.

 - stats_en: cpr21 (PARCv2 and higher)

    Used to enable or disable the statistics tracking feature of the
    processor (i.e. counting cycles and instructions)

 - numcores: cpr16 (PARCv3 and higher)

    Used to store the number of cores present in a multi-core system.
    Writing the register is undefined.

 - coreid: cpr17 (PARCv3 and higher)

    Used to communicate the core id in a multi-core system. Writing the
    register is undefined.

* Reset Vector

 - The reset vector for PARC points to the memory address 0x00001000,
   which is where assembly tests should reside, as well as user code in
   PARCv2, and the kernel bootstrap code for PARCv4.

---
3. PARC ISA Overview

Here is a brief list of the instructions which make up each version of
the PARC ISA.

* PARCv1

PARCv1 contains a subset of instructions of the full PARC ISA that can
run simple assembly benchmarks. We communicate with the processor through
two coprocessor registers (mngr2proc and proc2mngr) which implement a
simple register-mapped two-register FIFO interface.

 - mfc0, mtc0   : read/write coprocessor registers
 - addu, subu, and, or, slt   : register-register arithmetic
 - mul   : multiply operations
 - addiu, lui, ori, sra, sll   : register-immediate arithmetic
 - lw, sw   : word memory operations
 - j, jal, jr   : unconditional jumps
 - bne, beq   : conditional equality branches

* PARCv2

PARCv2 contains a subset of instructions of the PARC ISA that can be used to run raw C programs that do not use system calls. The mngr2proc and proc2mngr coprocessor registers are used here as well to communicate data between the manager and the processor. Statistics can be enabled by setting the coprocessor 0 stats_en register.

```
 - xor, nor, sltu               : more register-register arithmetic
 - srav, srlv, sllv             : more register-register arithmetic
 - div, divu, rem, remu         : division operations
 - andi, xori, slti, sltiu, srl : more register-immediate arithmetic
 - lb, lbu, lh, lhu, sb, sh     : sub-word memory operations
 - bgtz, bltz, bgez, blez       : conditional comparison branches
```

* PARCv3

PARCv3 adds support for a coherent multicore system with atomic instructions and a memory fence instruction.

```
 - amo.add, amo.and, amo.or     : atomic memory operations
 - sync                         : memory fence
```

* PARCv4

PARCv4 adds exception and system call support, allowing it to run full C++ programs. Currently PARC only supports system call exceptions.

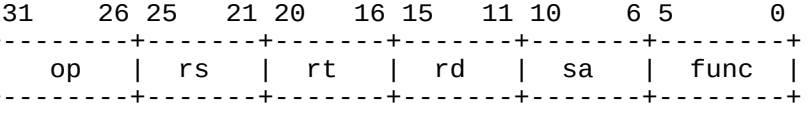
```
 - syscall                      : causes a syscall exception
 - eret                         : return from exception
```

---

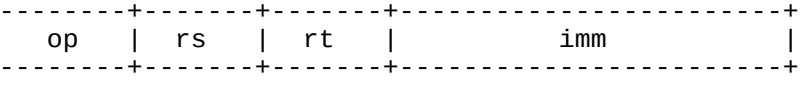
4. PARC Instruction Encoding

The 32-bit PARC instructions have different fields depending on the format of the instruction used. The following are the various instruction encoding formats used in the PARC ISA.

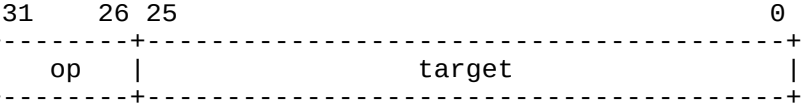
* R-Type:



* I-Type:



* J-Type:



---

5. PARC Instruction Details

For each instruction we include a brief summary, assembly syntax,

instruction semantics, encoding format, and the actual encoding for the instruction. We use the following conventions when specifying the instruction semantics:

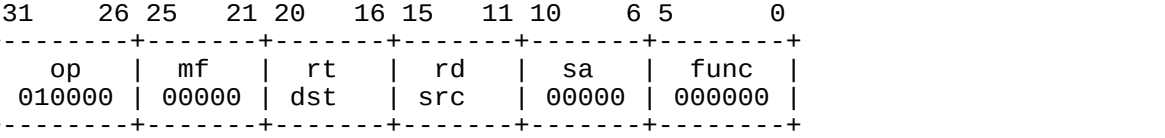
 - R[r_a]   : general-purpose register value for register specifier r_a
 - CP0[r_a]   : coprocessor0 register value for register specifier r_a
 - zext   : zero extend to 32 bits
 - sext   : sign extend to 32 bits
 - M_4B[addr] : 4-byte memory value at address addr
 - M_2B[addr] : 2-byte memory value at address addr
 - M_1B[addr] : 1-byte memory value at address addr
 - PC   : current program counter
 - PC_next   : next program counter
 - atomic {} : atomic with respect to memory
 - <s   : signed less-than comparison
 - >s   : signed greater-than comparison
 - <u   : unsigned less-than comparison
 - >u   : unsigned greater-than comparison

Unless otherwise specified assume instruction updates PC_next with PC+4.

5.1. Read-Write Coprocessor Register Instructions
---

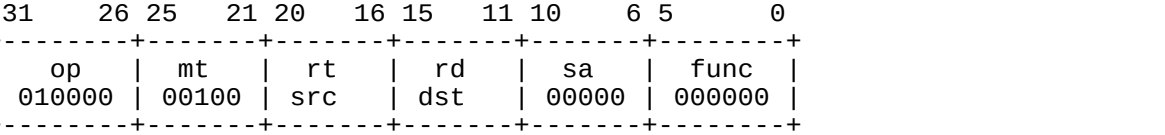
* mfc0

 - Summary   : Move value in coprocessor 0 register to GPR
 - Assembly : mfc0 r_dst, r_src
 - Semantics : R[r_dst] = CP0[r_src]
 - Format   : R-Type



* mtc0

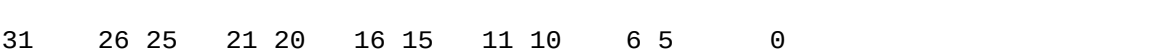
 - Summary   : Move value in GPR to coprocessor 0 register
 - Assembly : mtc0 r_src, r_dst
 - Semantics : CP0[r_dst] = R[r_src]
 - Format   : R-Type



5.2. Register-Register Arithmetic Instructions
---

* addu

 - Summary   : Signed addition with 3 GPRs, no overflow exception
 - Assembly : addu r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] + R[r_src1]
 - Format   : R-Type



```
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 100001 |
 +--------+-------+-------+-------+-------+--------+
```

The 'unsigned' keyword in the instruction name is a misnomer in most cases. The 'unsigned' variant of an instruction simply means that the operation will not trap on an overflow and does *not* imply that operands will be treated as unsigned values. The exceptions to this are the mul/div instructions, included in PARCv2. The PARC ISA, in general, does not support any instructions that use traps.

* subu

 - Summary   : Signed subtraction with 3 GPRs, no overflow exception
 - Assembly : subu r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] - R[r_src1]
 - Format   : R-Type

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 100011 |
 +--------+-------+-------+-------+-------+--------+
```

The 'unsigned' keyword in the instruction name is a misnomer in most cases. The 'unsigned' variant of an instruction simply means that the operation will not trap on an overflow and does *not* imply that operands will be treated as unsigned values. The exceptions to this are the mul/div instructions, included in PARCv2. The PARC ISA, in general, does not support any instructions that use traps.

* and

 - Summary   : Bitwise logical AND with 3 GPRs
 - Assembly : and r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] & R[r_src1]
 - Format   : R-Type

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 100100 |
 +--------+-------+-------+-------+-------+--------+
```

* or

 - Summary   : Bitwise logical OR with 3 GPRs
 - Assembly : or r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] | R[r_src1]
 - Format   : R-Type

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 100101 |
 +--------+-------+-------+-------+-------+--------+
```

* xor

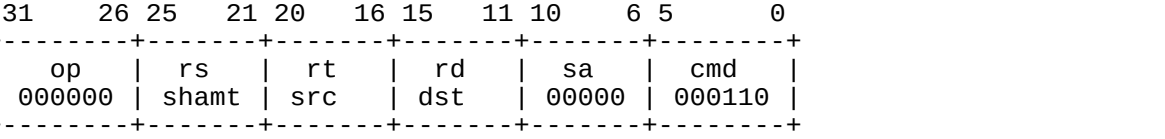
 - Summary   : Bitwise logical XOR with 3 GPRs
 - Assembly : xor r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] ^ R[r_src1]

```
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 100110 |
 +--------+-------+-------+-------+-------+--------+

* nor

 - Summary   : Bitwise logical NOR with 3 GPRs
 - Assembly  : nor r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = !( R[r_src0] | R[r_src1] )
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 100111 |
 +--------+-------+-------+-------+-------+--------+

* slt

 - Summary   : Record result of signed less-than comparison with 2 GPRs
 - Assembly  : slt r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = ( R[r_src0] <s R[r_src1] )
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 101010 |
 +--------+-------+-------+-------+-------+--------+

This instruction uses a signed comparison.

* sltu

 - Summary   : Record result of unsigned less-than comparison with 2 GPRs
 - Assembly  : sltu r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = ( R[r_src0] <u R[r_src1] )
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 101011 |
 +--------+-------+-------+-------+-------+--------+

This instruction uses an unsigned comparison.

* srav

 - Summary   : Shift right arithmetic by register value (sign-extend)
 - Assembly  : srav r_dst, r_src, r_shamt
 - Semantics : R[r_dst] = R[r_src] >>> R[r_shamt]
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | shamt | src   | dst   | 00000 | 000111 |
 +--------+-------+-------+-------+-------+--------+
```

Note that we should ensure that the sign-bit of the source is extended to the right as we do the right shift.

* srlv

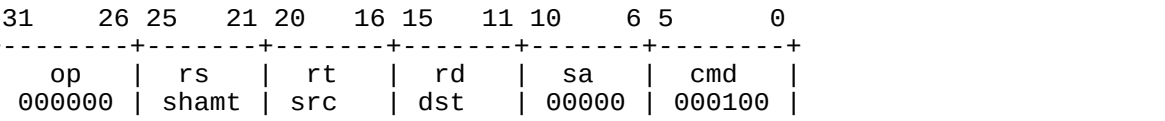
 - Summary   : Shift right logical by register value (append zeroes)
 - Assembly : srlv r_dst, r_src, r_shamt
 - Semantics : R[r_dst] = R[r_src] >> R[r_shamt]
 - Format   : R-Type



Append zeros to the left as we do the right shift.

* sllv

 - Summary   : Shift left logical by register value (append zeroes)
 - Assembly : sllv r_dst, r_src, r_shamt
 - Semantics : R[r_dst] = R[r_src] << R[r_shamt]
 - Format   : R-Type

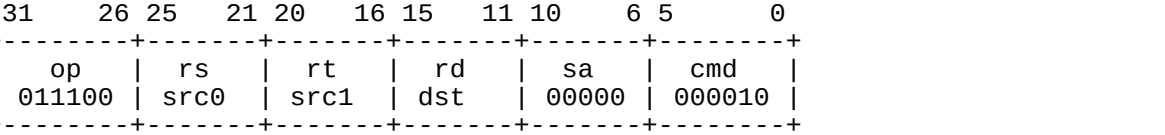


Append zeros to the left as we do the left shift.

5.3. Multiply/Divide Instructions
---

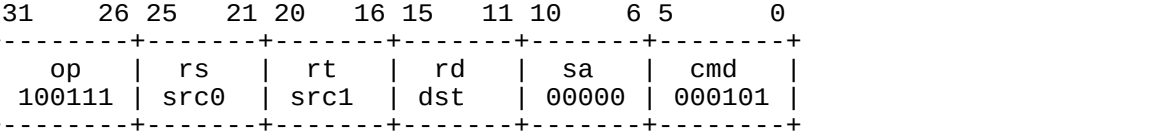
* mul

 - Summary   : Signed multiplication with 3 GPRs
 - Assembly : mul r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] * R[r_src1]
 - Format   : R-Type



* div

 - Summary   : Signed division with 3 GPRs
 - Assembly : div r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] / R[r_src1]
 - Format   : R-Type



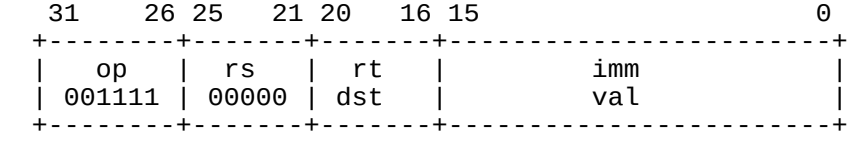
```
* divu

 - Summary   : Unsigned division with 3 GPRs
 - Assembly  : divu r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] / R[r_src1]
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 100111 | src0  | src1  | dst   | 00000 | 000111 |
 +--------+-------+-------+-------+-------+--------+

* rem

 - Summary   : Signed remainder with 3 GPRs
 - Assembly  : rem r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] % R[r_src1]
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 100111 | src0  | src1  | dst   | 00000 | 000110 |
 +--------+-------+-------+-------+-------+--------+

* remu

 - Summary   : Unsigned remainder with 3 GPRs
 - Assembly  : remu r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] % R[r_src1]
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 100111 | src0  | src1  | dst   | 00000 | 001000 |
 +--------+-------+-------+-------+-------+--------+

-------------------------------------------------------------------------
5.4. Register-Immediate Arithmetic Instructions
-------------------------------------------------------------------------

* addiu

 - Summary   : Add constant with no overflow exception
 - Assembly  : addiu r_dst, r_src, i_val
 - Semantics : R[r_dst] = R[r_src] + sext(i_val)
 - Format    : I-Type

  31    26 25   21 20   16 15                      0
 +--------+-------+-------+------------------------+
 |   op   |  rs   |  rt   |         imm            |
 | 001001 | src   | dst   |         val            |
 +--------+-------+-------+------------------------+
```

The ’unsigned’ keyword in the instruction name is a misnomer in most cases. The ’unsigned’ variant of an instruction simply means that the operation will not trap on an overflow and does *not* imply that operands will be treated as unsigned values. The exceptions to this are the mul/div instructions, included in PARCv2. The PARC ISA, in general, does not support any instructions that use traps.

Note that the 16-bit immediate value is sign-extended before being used
in the unsigned comparison.

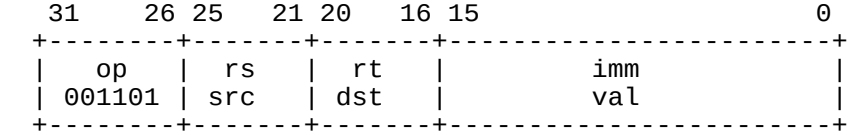
* lui

 - Summary   : Load constant into upper half of word
 - Assembly : lui r_dst, i_val
 - Semantics : R[r_dst] = i_val << 16
 - Format   : I-Type



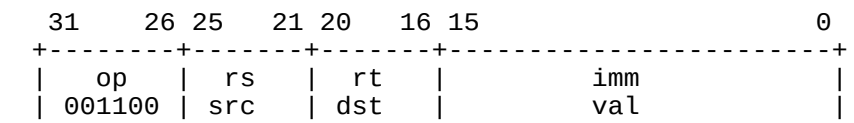
* ori

 - Summary   : Bitwise logical OR with constant
 - Assembly : ori r_dst, r_src, i_val
 - Semantics : R[r_dst] = R[r_src] | zext(i_val)
 - Format   : I-Type



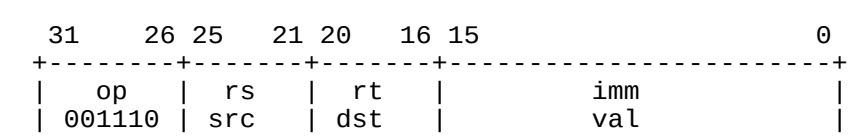
* andi

 - Summary   : Bitwise logical AND with constant
 - Assembly : andi r_dst, r_src, i_val
 - Semantics : R[r_dst] = R[r_src] & zext(i_val)
 - Format   : I-Type



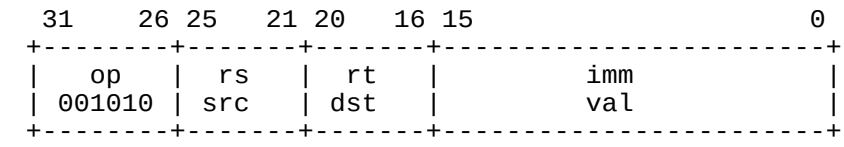
* xori

 - Summary   : Bitwise logical XOR with constant
 - Assembly : xori r_dst, r_src, i_val
 - Semantics : R[r_dst] = R[r_src] ^ zext(i_val)
 - Format   : I-Type



* slti

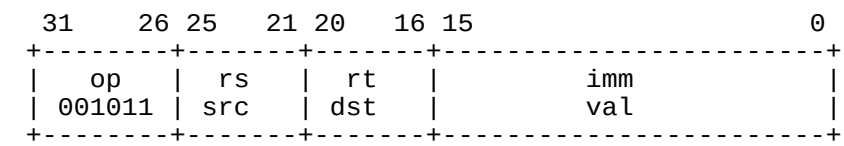
 - Summary   : Set GPR if source GPR < constant, signed comparison
 - Assembly : slti r_dst, r_src, i_val
 - Semantics : R[r_dst] = (R[r_src] <s sext(i_val))
 - Format   : I-Type



The 16-bit immediate value is sign-extended before being used in the signed comparison.

* sltiu

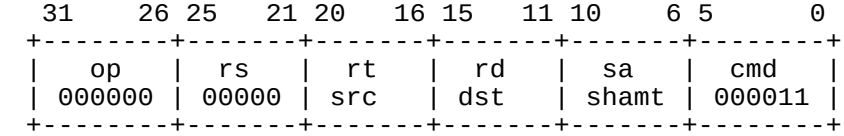
 - Summary : Set GPR if source GPR is < constant, unsigned comparison
 - Assembly : sltiu r_dst, r_src, i_val
 - Semantics : R[r_dst] = (R[r_src] <u sext(i_val))
 - Format : I-Type



The 16-bit immediate value is sign-extended before being used in the unsigned comparison.

* sra

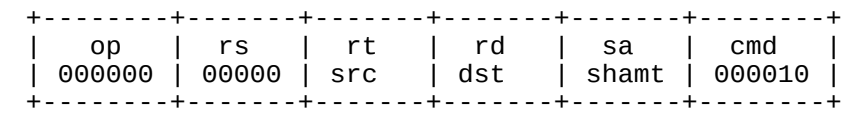
 - Summary : Shift right arithmetic by constant (sign-extend)
 - Assembly : sra r_dst, r_src, i_shamt
 - Semantics : R[r_dst] = R[r_src] >>> i_shamt
 - Format : R-Type



Note that we should ensure that the sign-bit of the source is extended to the right as we do the right shift.

* srl

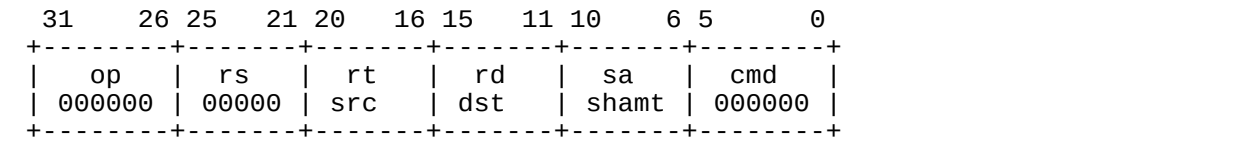
 - Summary : Shift right logical by constant (append zeroes)
 - Assembly : srl r_dst, r_src, i_shamt
 - Semantics : R[r_dst] = R[r_src] >> i_shamt
 - Format : R-Type



Append zeros to the left as we do the right shift.

* sll

 - Summary : Shift left logical constant (append zeroes)
 - Assembly : sll r_dst, r_src, i_shamt
 - Semantics : R[r_dst] = R[r_src] << i_shamt
 - Format : R-Type



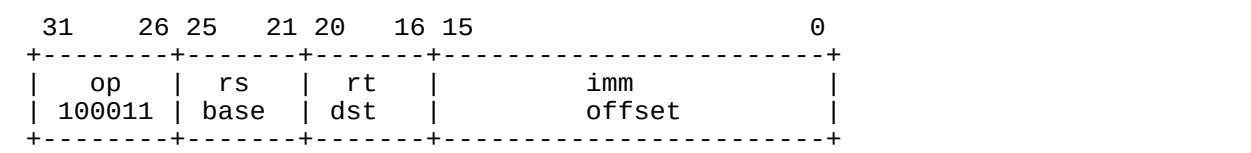
Append zeros to the right as we do the left shift.

5.5. Memory Instructions

---

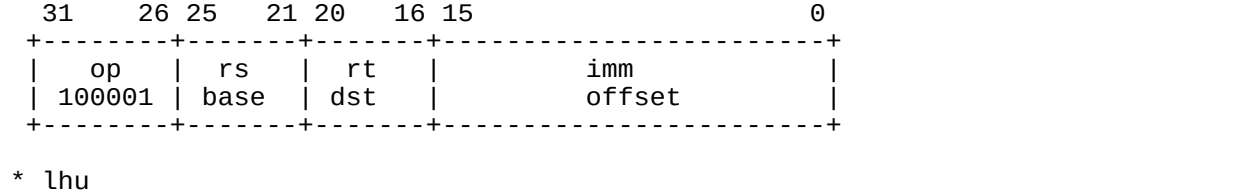
* lw

- Summary : Load word from memory as signed value
- Assembly : lw r_dst, i_offset(r_base)
- Semantics : R[r_dst] = M_4B[R[r_base] + sext(i_offset)]
- Format : I-Type



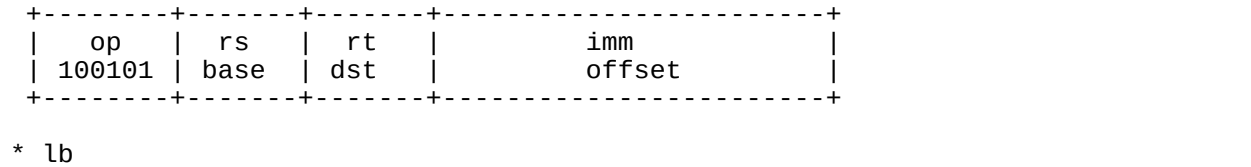
* lh

- Summary : Load a halfword from memory as signed value
- Assembly : lh r_dst, i_offset(r_base)
- Semantics : R[r_dst] = sext(M_2B[R[r_base] + sext(i_offset)])
- Format : I-Type



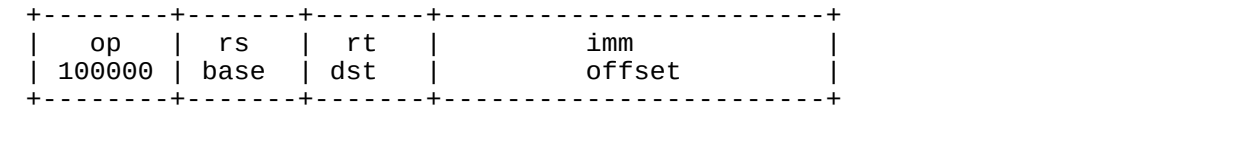
* lhu

- Summary : Load a halfword from memory as unsigned value
- Assembly : lhu r_dst, i_offset(r_base)
- Semantics : R[r_dst] = zext(M_2B[R[r_base] + sext(i_offset)])
- Format : I-Type



* lb

- Summary : Load a byte from memory as signed value
- Assembly : lb r_dst, i_offset(r_base)
- Semantics : R[r_dst] = sext(M_1B[R[r_base] + sext(i_offset)])
- Format : I-Type



```
* lbu

 - Summary   : Load a byte from memory as unsigned value
 - Assembly  : lbu r_dst, i_offset(r_base)
 - Semantics : R[r_dst] = zext( M_1B[ R[r_base] + sext(i_offset) ] )
 - Format    : I-Type

  31    26 25   21 20   16 15                       0
 +--------+-------+-------+-------------------------+
 |   op   |  rs   |  rt   |         imm             |
 | 100100 | base  | dst   |         offset          |
 +--------+-------+-------+-------------------------+

* sw

 - Summary   : Store word into memory
 - Assembly  : sw r_src, i_offset(r_base)
 - Semantics : M_4B[ R[r_base] + sext(i_offset) ] = R[r_src]
 - Format    : I-Type

  31    26 25   21 20   16 15                       0
 +--------+-------+-------+-------------------------+
 |   op   |  rs   |  rt   |         imm             |
 | 101011 | base  | src   |         offset          |
 +--------+-------+-------+-------------------------+

* sh

 - Summary   : Store a halfword to memory
 - Assembly  : sh r_src, i_offset(r_base)
 - Semantics : M_2B[ R[r_base] + sext(i_offset) ] = R[r_src]
 - Format    : I-Type

  31    26 25   21 20   16 15                       0
 +--------+-------+-------+-------------------------+
 |   op   |  rs   |  rt   |         imm             |
 | 101001 | base  | src   |         offset          |
 +--------+-------+-------+-------------------------+

* sb

 - Summary   : Store a byte to memory
 - Assembly  : sb r_src, i_offset(r_base)
 - Semantics : M_1B[ R[r_base] + sext(i_offset) ] = R[r_src]
 - Format    : I-Type

  31    26 25   21 20   16 15                       0
 +--------+-------+-------+-------------------------+
 |   op   |  rs   |  rt   |         imm             |
 | 101000 | base  | src   |         offset          |
 +--------+-------+-------+-------------------------+

------------------------------------------------------------------------------
5.6. Unconditional Jump Instructions
------------------------------------------------------------------------------

* j

 - Summary   : Jump to address
 - Assembly  : j i_targ
 - Semantics : PC_plus4 = PC + 4;
                 PC_next = { PC_plus4[31:28], i_targ << 2 }
```

```
 - Format    : J-Type

  31    26 25                                          0
 +--------+-----------------------------------------+
 |   op   |                 imm                     |
 | 000010 |                 targ                    |
 +--------+-----------------------------------------+
```

i_targ is shifted to the left by 2 bits and the resulting 28 bits are combined with the 4 msb of PC+4 to generate the effective target address.

* jr

```
 - Summary   : Jump to address in register
 - Assembly  : jr r_src
 - Semantics : PC_next = R[r_src]
 - Format    : J-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |       |  cmd   |
 | 000000 | src   | 00000 | 00000 | 00000 | 001000 |
 +--------+-------+-------+-------+-------+--------+
```

The target address in r_src must be naturally aligned.

* jalr

```
 - Summary   : Jump to address and place return address in GPR
 - Assembly  : jalr r_ret, r_targ
 - Semantics : R[r_ret] = PC + 4; PC_next = R[r_targ]
 - Format    : J-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | targ  | 00000 | ret   | 00000 | 001001 |
 +--------+-------+-------+-------+-------+--------+
```

The return address should be the instruction immediately following the branch instruction. Keep in mind that this is different from the MIPS ISA in which the return address is 2 instructions after the branch instruction to account for the branch delay slot.

If r_ret is not defined in the assembly, the return address will be stored in GPR 31 by default. The target address in r_targ must be naturally aligned. r_targ and r_ret should not be equal, as it will cause behavior that is non-idempotent.

* jal

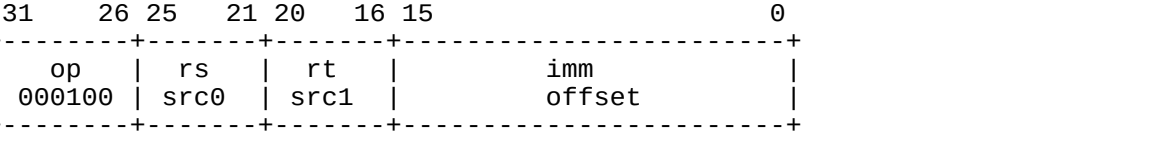
```
 - Summary   : Jump to address and place return address in GPR 31
 - Assembly  : jal i_targ
 - Semantics : R[31] = PC + 4; PC_plus4 = PC + 4;
                 PC_next = { PC_plus4[31:28], i_targ << 2 }
 - Format    : J-Type

  31    26 25                                          0
 +--------+-----------------------------------------+
 |   op   |                 imm                     |
 | 000011 |                 targ                    |
 +--------+-----------------------------------------+
```

i_targ is shifted to the left by 2 bits and the resulting 28 bits are
combined with the 4 msb of PC+4 to generate the effective target address.

5.7. Conditional Branch Instructions
---

* beq

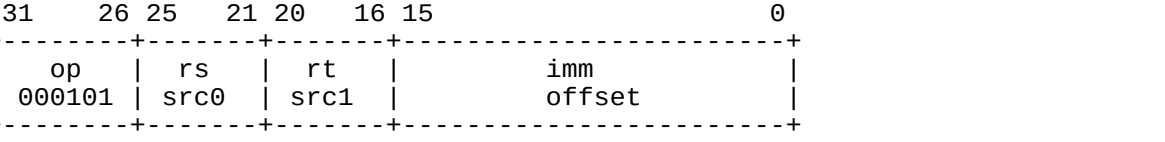
 - Summary   : Branch if 2 GPRs are equal
 - Assembly : beq r_src0, r_src1, i_offset
 - Semantics : if (R[r_src0] == R[r_src1])
                 PC_next = PC + 4 + (sext(i_offset) << 2)
 - Format   : I-Type



The target address offset is relative to the PC of the instruction
after the actual branch.

* bne

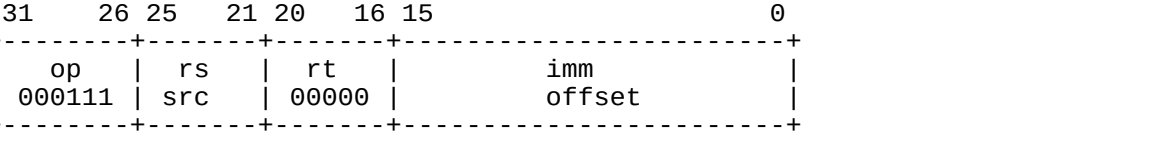
 - Summary   : Branch if 2 GPRs are not equal
 - Assembly : bne r_src0, r_src1, i_offset
 - Semantics : if (R[r_src0] != R[r_src1])
                 PC_next = PC + 4 + (sext(i_offset) << 2)
 - Format   : I-Type



The target address offset is relative to the PC of the instruction
after the actual branch.

* bgtz

 - Summary   : Branch if GPR is greater than zero
 - Assembly : bgtz r_src, i_offset
 - Semantics : if (R[r_src] >s 0)
                 PC_next = PC + 4 + (sext(i_offset) << 2)
 - Format   : I-Type



The target address offset is relative to the PC of the instruction
after the actual branch.

* bltz

 - Summary   : Branch if GPR is less than zero
 - Assembly : bltz r_src, i_offset
 - Semantics : if (R[r_src] <s 0)

```
                PC_next = PC + 4 + ( sext(i_offset) << 2 )
 - Format    : I-Type

  31    26 25   21 20   16 15                      0
 +--------+-------+-------+------------------------+
 |   op   |  rs   |  rt   |         imm            |
 | 000001 | src   | 00000 |         offset         |
 +--------+-------+-------+------------------------+
```

The target address offset is relative to the PC of the instruction *after* the actual branch.

* bgez

```
 - Summary   : Branch if GPR is greater than or equal to zero
 - Assembly  : bgez r_src, i_offset
 - Semantics : if ( R[r_src] >s 0 ) || ( R[r_src] == 0 ) )
                 PC_next = PC + 4 + ( sext(i_offset) << 2 )
 - Format    : I-Type

  31    26 25   21 20   16 15                      0
 +--------+-------+-------+------------------------+
 |   op   |  rs   |  rt   |         imm            |
 | 000001 | src   | 00001 |         offset         |
 +--------+-------+-------+------------------------+
```

The target address offset is relative to the PC of the instruction *after* the actual branch.

* blez

```
 - Summary   : Branch if GPR is less than or equal to zero
 - Assembly  : blez r_src, i_offset
 - Semantics : if ( R[r_src] <s 0 ) || ( R[r_src] == 0 ) )
                 PC_next = PC + 4 + ( sext(i_offset) << 2 )
 - Format    : I-Type

  31    26 25   21 20   16 15                      0
 +--------+-------+-------+------------------------+
 |   op   |  rs   |  rt   |         imm            |
 | 000110 | src   | 00000 |         offset         |
 +--------+-------+-------+------------------------+
```

The target address offset is relative to the PC of the instruction *after* the actual branch.

5.8. Concurrency Instructions

---

* amo.add

```
 - Summary   : Atomic fetch & add
 - Assembly  : amo.add r_dst, r_addr, r_src
 - Semantics : atomic {
                 temp = M_4B[ R[r_addr] ]
                 M_4B[ R[r_addr] ] = temp + R[r_src]
                 R[r_dst] = temp
               }
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
```

```
 |   op   |  rs   |  rt   |  rd   |  sa   |   cmd  |
 | 100111 | addr  | src   | dst   | 00000 | 000010 |
 +--------+-------+-------+-------+-------+--------+
```

Atomic instructions are a series of operations that all perform atomically with respect to other memory operations. The amo.add instruction will perform a fetch and an ADD operation which looks like they both happened at once to other memory operations.

* amo.and

```
 - Summary   : Atomic fetch & and
 - Assembly  : amo.and r_dst, r_addr, r_src
 - Semantics : atomic {
                 temp = M_4B[ R[r_addr] ]
                 M_4B[ R[r_addr] ] = temp & R[r_src]
                 R[r_dst] = temp
               }
 - Format    : R-Type
```

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |   cmd  |
 | 100111 | addr  | src   | dst   | 00000 | 000011 |
 +--------+-------+-------+-------+-------+--------+
```

Atomic instructions are a series of operations that all perform atomically with respect to other memory operations. The amo.and instruction will perform a fetch and an AND operation which looks like they both happened at once to other memory operations.

* amo.or

```
 - Summary   : Atomic fetch & or
 - Assembly  : amo.or r_dst, r_addr, r_src
 - Semantics : atomic {
                 temp = M_4B[ R[r_addr] ]
                 M_4B[ R[r_addr] ] = temp | R[r_src]
                 R[r_dst] = temp
               }
 - Format    : R-Type
```

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |   cmd  |
 | 100111 | addr  | src   | dst   | 00000 | 000100 |
 +--------+-------+-------+-------+-------+--------+
```

Atomic instructions are a series of operations that all perform atomically with respect to other memory operations. The amo.and instruction will perform a fetch and an OR operation which looks like they both happened at once to other memory operations.

* sync

```
 - Summary   : Order loads and stores
 - Assembly  : sync
 - Semantics : memory fence
 - Format    : R-Type
```

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |   cmd  |
```

```
 | 000000 | 00000 | 00000 | 00000 | 00000 | 001111 |
 +--------+-------+-------+-------+-------+--------+
```

All loads and stores that occur before a sync must complete before any loads and stores after the sync can start. A load is complete when the destination register is written and a store is complete when the stored value is visible to all cores in the system.

5.9. Exception Instructions

---

* syscall

 - Summary   : Trap into system call exception
 - Assembly : syscall
 - Semantics : PC_next = 0x00000004; EPC = PC; change to supervisor mode
 - Format   : R-Type

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |   cmd  |
 | 000000 | 00000 | 00000 | 00000 | 00000 | 001100 |
 +--------+-------+-------+-------+-------+--------+
```

* eret

 - Summary   : Return from exception
 - Assembly : eret
 - Semantics : PC_next = EPC; change to user mode
 - Format   : R-Type

```
  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |   cmd  |
 | 000000 | 00000 | 00000 | 00000 | 00000 | 011000 |
 +--------+-------+-------+-------+-------+--------+
```

Uses the return address stored in EPC to return from an exception.