

# ***Arca***

# **Instruction Set Architecture**

## **Reference Manual**

V2



**方舟科技有限公司**

ARCA Technology Corporation



# Arca

## Instruction Set Architecture Reference Manual – V2

Copyright © ARCA Technology Corporation. 2003.

Third party brands, logos and names are the property of those respective third parties.

### Release history

Date	Revision	Change
Jul 2001	V1	ISA Version 1
Jan 2003	V2	ISA Version 2

### Disclaimer

This documentation is provided for use with ARCA Technology Corporation products. No license to ARCA Technology Corporation property rights is granted. ARCA Technology Corporation assumes no liability, provides no warranty either expressed or implied relating to the usage, or intellectual property right infringement, except as provided for by the ARCA Technology Corporation Terms and Conditions of Sale.

ARCA Technology Corporation products are not designed for and should not be used in any medical or life sustaining or supporting equipment.

All information in this document should be treated as preliminary. ARCA Technology Corporation may make changes to this document without notice. Anyone relying on this documentation should contact ARCA Technology Corporation for the current documentation and errata.

### ARCA Technology Corporation

5th Floor, Jade Palace Building,  
76 Zhichun Rd, Haidian,  
Beijing, P. R. China  
Tel: 86-10-62638192  
Fax:86-10-62638348  
Http: [www.arca.com.cn](http://www.arca.com.cn)



# Table of Contents

<b>1 OVERVIEW .....</b>	<b>1</b>
1.1 INTRODUCTION .....	1
1.2 ARCA REGISTERS .....	2
1.3 ARCA EXCEPTIONS .....	2
1.4 ARCA CORE CONFIGURATION .....	2
1.5 ARCA INSTRUCTIONS .....	3
<b>2 PROGRAMMING MODEL .....</b>	<b>4</b>
2.1 DATA TYPES AND ORGANIZATION .....	4
2.1.1 <i>Data Organization In Register</i> .....	4
2.1.2 <i>Data Organization in Memory</i> .....	4
2.2 PROCESSOR MODES .....	6
2.3 REGISTERS .....	7
2.3.1 <i>General Purpose Registers</i> .....	7
2.3.2 <i>Program Counter (PC)</i> .....	8
2.3.3 <i>Control Registers</i> .....	8
2.4 EXCEPTIONS .....	10
2.4.1 <i>Exception Types</i> .....	10
2.4.2 <i>Exception Priorities</i> .....	10
2.4.3 <i>Exception Vectors</i> .....	11
2.4.4 <i>Base address for Exception Vector Table</i> .....	11
<b>3 INSTRUCTION SET .....</b>	<b>12</b>
3.1 INSTRUCTION FORMAT .....	12
3.1.1 <i>Operator field</i> .....	12
3.1.2 <i>Operand field</i> .....	12
3.1.3 <i>Instructions and format summary</i> .....	14
3.2 INSTRUCTION DESCRIPTION .....	15
3.2.1 <i>Immediate load instruction</i> .....	15
3.2.2 <i>Jump Instructions</i> .....	17
3.2.3 <i>Branch Instructions</i> .....	20
3.2.4 <i>Arithmetic Instructions</i> .....	26
3.3 COMPARISON INSTRUCTIONS .....	31
3.3.1 <i>Bitwise Instructions</i> .....	34
3.3.2 <i>Shift Instructions</i> .....	38
3.3.3 <i>Load and Store Instructions</i> .....	41
3.3.4 <i>Miscellaneous Instructions</i> .....	63
3.3.5 <i>System Control Instructions</i> .....	69
3.4 INSTRUCTION ENCODING .....	82
<b>4 APPLICATION BINARY INTERFACE .....</b>	<b>12</b>
4.1 REGISTER USAGE CONVENTIONS .....	12
4.2 STACK LAYOUT .....	13
4.3 FRAME LAYOUT .....	14
4.4 TYPE MAPPING .....	15
4.4.1 <i>Scalar Types</i> .....	15
4.4.2 <i>Aggregate Types</i> .....	15
4.5 BIT-FIELDS .....	16
4.6 RETURN VALUES (FUNCTION RESULTS) .....	17
4.7 ARGUMENT PASSING AND MAPPING .....	18

<b>5 ASSEMBLER MACROS AND OPERATORS .....</b>	<b>93</b>
<b>LIST OF FIGURES .....</b>	<b><a href="#">24</a></b>
<b>LIST OF TABLES .....</b>	<b><a href="#">35</a></b>

# 1 Overview

## 1.1 Introduction

Arca is a new RISC architecture designed from scratch to address the application requirements and challenges for next generation embedded and information appliance markets. The instruction set was designed to allow a very small with very low power consumption, yet high-performance implementation.

As a RISC architecture, Arca incorporates the typical RISC architecture features:

- A large uniform register file.
- A load and store architecture, where data processing only operate on register contents, not directly on memory contents.
- Simple address mode, with all load/store addressing being determined from register contents and instruction field only.
- Uniform and fix-length instruction field, to simplify instruction decoding.

Table 1-1 lists the key features of this architecture.

**Table 1-1 Arca Features**

Feature	Description
Architecture	32-bit load/store RISC architecture
General Register	32 x 32-bits general purpose register file
Control Registers	5 control registers and 1 PC register provided
Instruction Set	32-bit length
Address Space	4 Gbytes address space available
Exception	There are 6 types: reset; illegal instruction; memory access fault; trap; interrupt and debug

## 1.2 Arca registers

Arca has a very simple register model. Only 37 registers including system control registers are provided.

- General purpose registers: 32 in total, each 32 bits wide, no register banking
- Program Counter: 1, 32 bits wide
- Control registers: 5 in total, each 32 bits wide. Only accessible in supervisor mode

## 1.3 Arca exceptions

Arca has two processing modes: user mode and supervisor mode. The system resources that can be accessed in user mode are restricted, while in supervisor mode, all the system resources can be accessed. Only exception can make the processor switches from user mode to supervisor mode. And only RTE (return from exception) instruction can make the processor switches from supervisor mode to user mode.

Arca supports 6 types of exception:

- Reset: includes power-on reset, manual reset and debug bootstrap
- Illegal Instruction
- Memory Access Fault: includes Data Access Fault and Instruction Access Fault
- Trap
- Interrupt
- Debug Break: includes instruction break, data break, software break and debug interrupt

When an exception occurs, Arca switches to supervisor mode. All the 32 general-purpose registers and control registers and system resources can be accessed in supervisor mode.

When an exception occurs, the current processor state and PC are saved to the related control registers. The processor halts execution of the current program flow and begins execution at the address that store in one of a number of fixed addresses in memory, known as the exception vectors. Each exception type was assigned to one exception vector.

## 1.4 Arca core configuration

Besides the Integer Unit which implements Arca instruction set, a typical Arca CPU Core includes modules such as MMU (Memory Management Unit), data and instruction cache, debug and performance monitoring module, etc. Arca provides a convenient and extensible way to exchange values between these modules and GRF, and to expand module specific operations.



## 1.5 Arca instructions

The Arca instruction set can be divided into 9 classes of instruction:

- 1) **Immediate-load instruction:** load an immediate operand to a general purpose register.
- 2) **Jump instruction:** unconditional branch to the target address: PC-relative address or absolute address, with recording a return link address in a general register.
- 3) **Branch instruction:** PC-relative conditional branch. The branch condition include: equal, not equal, less than, greater than or equal, unsigned less than, unsigned greater than or equal.
- 4) **Arithmetic instruction:** include addition, subtraction, multiplication and comparison arithmetic instruction.
- 5) **Shift instruction:** there are three kinds of shift operations: logical left, logical right and arithmetical right, support immediate operand.
- 6) **Bitwise instruction:** performance bitwise operation that include: **AND, ANDN, OR, and XOR**, support immediate operand except **ANDN**.
- 7) **Load and store instruction:** transfer data between the memory system and the general purpose registers in the CPU. There are separate instructions for different purposes: transferring various sized fields, treating loaded data as signed or unsigned integers.
- 8) **Miscellaneous instruction:** include conditional move instructions, count sign bits instructions and swap the register contents with memory contents instruction.
- 9) **System control instruction:** these instructions are used for system management such as read/write control register, return from exception handler and etc.

## 2 Programming Model

This section describes the organization of data in registers and in memory, the available user mode and supervisor mode registers and the exception model.

### 2.1 Data Types and Organization

Table 2-1 lists Arca supported data types and operations performed on these types.

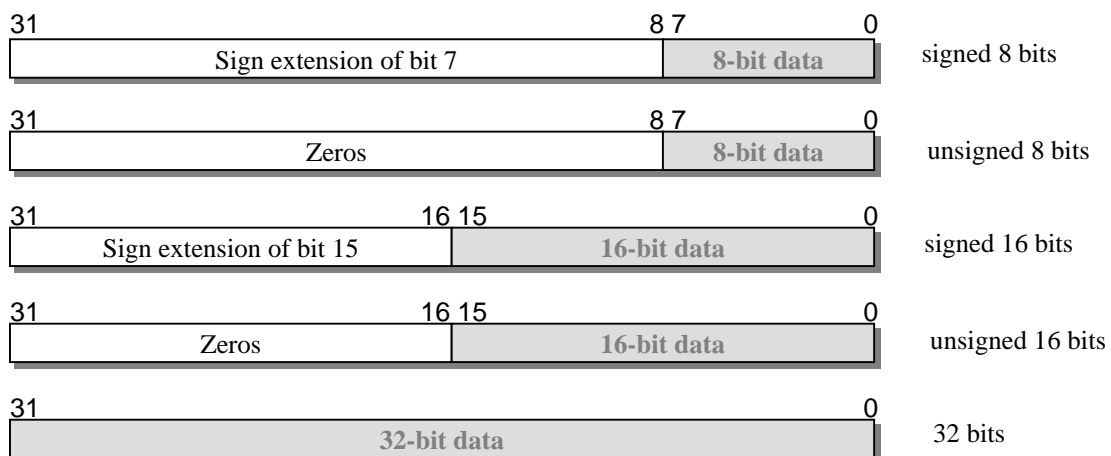
**Table 2-1 Data Type and Operation**

	Data Type	Supported Operations	Note
Byte	8-bits integer, signed	Load, Store	
	8-bits integer, unsigned	Load, Store	
Halfword	16-bits integer, signed	Load, Store	Must be aligned to two-byte boundaries
	16-bits integer, unsigned	Load, Store	
Word	32-bits integer	Load, Store, Arithmetic	Must be aligned to four-byte boundaries

All data types are standard 2's complement representation. Only load/store instructions operate on 8-bits and 16-bits data type, automatically zero-extending or sign-extending as they are loaded. All the other instructions operate on all the 32 bits of the data.

#### 2.1.1 Data Organization In Register

Figure 2-1 summarizes the data organization in registers. Register bit 0 contains the least significant bit of the data, while register bit 31, 15 and 7 contains the most significant bit for 32 bits, 16 bits and 8bits data respectively.



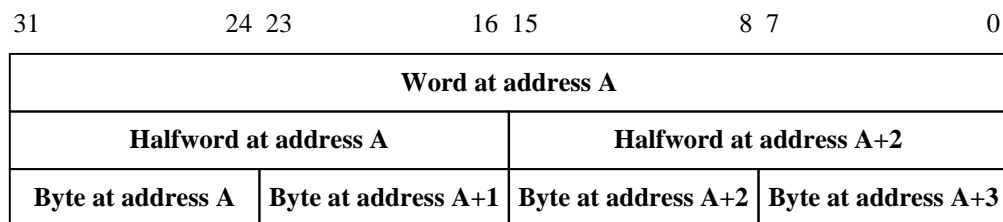
**Figure 2-1 Data Organization in Register**

#### 2.1.2 Data Organization in Memory

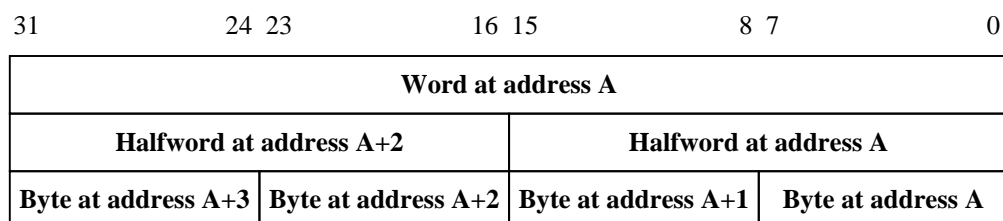
Data organized in memory is either in big-endian format or in little-endian format:

- When Arca is configured as big-endian memory:
  - byte 0 or halfword 0 at a word-aligned address is the most significant byte or halfword within the word at that address.
  - byte 0 at a halfword-aligned address is the most significant byte within the halfword at that address.
- When Arca is configured as little-endian memory:
  - byte 0 or halfword 0 at a word-aligned address is the least significant byte or halfword within the word at that address..
  - byte 0 at a halfword-aligned address is the least significant byte within the halfword at that address..

Figure 2-2 and Figure 2-3 describes the data organization in big-endian and little-endian memory system. For a word-aligned address A, the figures show how the word at address A, the halfword at address A and A+2, and the byte at address A, A+1, A+2 and A+3 map on to each other for each endianness.



**Figure 2-2 Big-endian Memory System**



**Figure 2-3 Little-endian Memory System**

**Note:** it is IMPLEMENTATION DEFINED whether an Arca implementation supports little-endian memory system, big-endian memory system, or both.

If an Arca implementation is configured for a memory system of one endianness but is actually attached to a memory system of the opposite endianness, only word-sized instruction fetches, data loads and data stores can be relied upon. Other memory accesses have unpredictable result.

## 2.2 Processor Modes

The Arca architecture supports two processor modes:

- User mode
- Supervisor mode

Mode changes can be made under software control, or can be caused by external interrupts or exception processing.

Most application programs execute in user mode. While the processor is in user mode, the program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur. This allows a suitably written operation system to control the use of system resources.

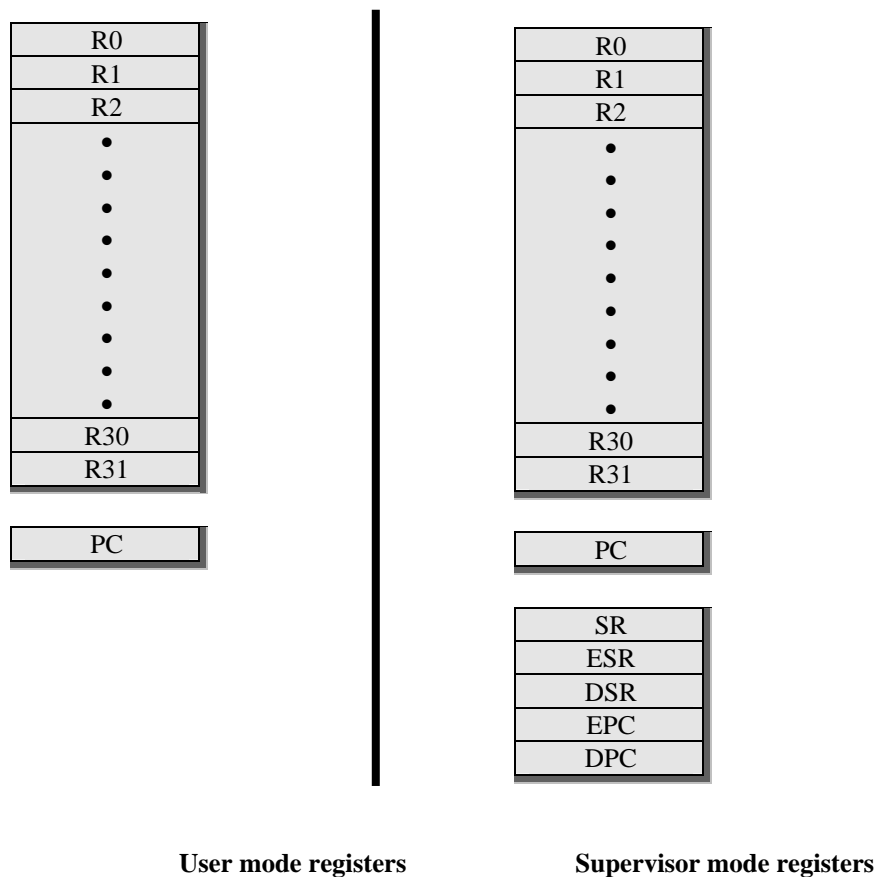
When the processor is in supervisor mode, the programs have full access to system resources and can change mode freely. And the system control instructions can only be executed in supervisor mode.

## 2.3 Registers

Arca has the following registers:

- 32 general purpose registers
- 5 control registers (SR, ESR, DSR, EPC, DPC)
- 1 PC (Program Counter)

In supervisor mode, all the registers can be accessed, while in user mode, only a subset of the registers can be accessed, as shown in Figure 2-4.



**Figure 2-4 Arca Registers**

### 2.3.1 General Purpose Registers

Arca provides 32 general purpose registers (R0~R31) which contain instruction operands and address information.

In these registers, R0 is always reading as 0 and writing to it is ignored. This can be used to make any register operand take a zero value. It is very useful for zero is a particularly common constant. R0 can also be used as the destination register to discard the result of an instruction.

## 2.3.2 Program Counter (PC)

PC contains the address of the instruction in execution. PC is updated automatically when instruction execution. When an exception occurred, PC is replaced with a new value that points to the exception handler.

PC is invisible to software. However, the instruction of J could be used to trace the value of PC.

## 2.3.3 Control Registers

Arca provides 5 control registers for tasks like processor control. All of these registers are only visible in supervisor mode (SR.SM=1). Two instructions RCR and WCR are used for exchanging data between control registers and general purpose registers.

RCR and WCR are supervisor instructions, executing these two instructions in user mode will cause an illegal instruction exception.

Table 2-2 lists the initial value of these registers after power on reset.

**Table 2-2 Control Registers Initial Value After Power On Reset**

Register	Power Reset
R1~R31	0
SR	H'00000008
ESR	Undefined
EPC	Undefined
DSR	Undefined
DPC	Undefined

### 2.3.3.1 SR: Status Register

This is the main control register of Arca. It contains the current processor mode, interrupt enable bit, and other status and control information. The value of this register will be saved to DSR when debug exception or debug bootstrap occurs, or saved to ESR when other exception occurs.

Bit:	31	7	6	5	4	3	2	1	0				
Read:							VB	SM	DS	DE	IE		
Write:													
Reset:	0						0	0	0	1	0	0	0

Bits 31~7 reserved, these bits are always read as 0 and written are ignored.

- **IE (Interrupt Enable):** When it is cleared, interrupt is disabled.
- **DE (Debug Enable):** When it is cleared, no debug exceptions are to be accepted. It doesn't influence Debug bootstrap.
- **DS (Debug State):** 1 indicates an debug exception or debug bootstrap occurs, 0 indicates a non-debug exception occurs. RTE restore PC/SR register from DPC/DSR when DS is 1, and from EPC/ESR when this bit is 0.

- **SM (Supervisor Mode):** 1 for Supervisor mode, 0 for User mode. Write to this bit by WCR instruction is ignored. Program can use RTE to switch from supervise mode to user mode by first clear corresponding bit in ESR or DSR register.
- **VB (Vector Base):** form bit 28~26 of the starting address of Vector Table (the highest 3 bits are 100, pointing to P1 area).

#### **2.3.3.2 EPC: Saved PC for Non-Debug Exception**

EPC is used for saving the current PC when a non-debug exception and non-debug bootstrap occurs.

#### **2.3.3.3 ESR: Saved SR for Non-Debug Exception**

ESR is used for saving the current SR when a non-debug exception and non-debug bootstrap occurs.

#### **2.3.3.4 DPC: Saved PC for Debug Exception**

DPC is used for saving the current PC when a debug exception or debug bootstrap occurs.

#### **2.3.3.5 DSR: Saved SR for Debug Exception**

DSR is used for saving the current SR when a debug exception or debug bootstrap occurs.

## 2.4 Exceptions

Exceptional events such as TLB miss, Interrupt and etc. may arise during normal execution of the current program flow. When Arca CPU verifies an exception request, it halts the normal execution temporarily, preserves the current processor status and program counter and then switches to the exception routine. After the exception is serviced, Arca CPU restores the normal execution.

### 2.4.1 Exception Types

Arca processor can handle exceptions such as **RESET**, **DFAULT**, **ILLINS**, **TRAP**, **IFAUULT**, **INT** and debug-related types as **IBRK**, **DBRK**, **SBRK** and **DBOOT**.

- **RESET**: Reset exception was caused by power-on reset, manual reset or debug bootstrap (**DBOOT**). Power-on reset, manual reset can be induced by an external input pin or watchdog time-out. **DBOOT** is induced by debug module.
- **Memory Access Fault**: Memory access fault exception occurs when a memory access issued from CPU couldn't be satisfied because of events TLB miss, address error and etc. Memory access fault exceptions include data access fault (**DFAULT**) and instruction fetch fault (**IFAUULT**).
- **ILLINS**: Illegal instruction exception, caused by executing a reserved instruction or executing a privileged instruction in user mode.
- **TRAP**: Trap exception was caused by a trap instruction.
- **Debug Break**: Debug break exception includes **IBRK**, **DBRK** and **SBRK**. **IBRK** is caused by instruction breakpoint match or debug interrupt. **DBRK** is caused by data breakpoint match. **SBRK** is caused by SBRK instruction when SR.DE = 1.
- **INT**: Interrupt exception.

### 2.4.2 Exception Priorities

When two or more exceptions occur simultaneously, the highest-priority one will be accepted. The exception priorities among those exception types are fixed as illustrated by the table below:

**Table 2-3 Arca Exception Priorities**

Exception Types	Exception Events	Exception Priorities
RESET	Debug Bootstrap	0 (highest)
	Power-on/Manual Reset	1
DFAULT	Data Access Fault	2
DBRK	Debug Data Breakpoint	3
ILLINS	Reserved Instruction or Privilege Violation	4
TRAP/SBRK	TRAP or SBRK Instruction	5
IBRK	Debug Instruction Breakpoint	6
IFAUULT	Instruction Fetch Fault	7
INT	Interrupt	8 (lowest)



### 2.4.3 Exception Vectors

By providing an 8-entry vector table with each exception type corresponding to one vector entry, Arca processor can switch to the top exception routine conveniently and efficiently. The top exception routine may consult the exception cause register to further determine the specific exception service.

**Table 2-4 Arca Exception Vector Table**

Vector Number	Vector Offset	Exception Type
0	H'00	RESET, DBOOT
1	H'04	ILLINS
2	H'08	IBRK, DBRK, SBRK
3	H'0c	Reserved
4	H'10	INT
5	H'14	TRAP
6	H'18	DFAULT, IFAULT
7	H'1c	Reserved

Since the processing for DFAULT and IFAULT is similar, only one vector entry is assigned for them.

### 2.4.4 Base address for Exception Vector Table

When Arca is in debugging and the debugger is set to host mode by host machine, exception vector table base is fixed at H'EC000000. In addition, vector table base for DBOOT exception is always at H'EC000000.

For other cases, exception vector table is placed on the boundary of 64M memory page of P1 area, which is decided by SR.VB bits (refer to Status Register description).

$$\text{Base address} = \{3B'100, \text{SR.VB}, 26B'0\}$$

Since SR.VB is initialized to 3B'000, so the base address for power-on reset without DBOOT is always H'80000000.

## 3 Instruction set

Arca architecture has 78 instructions, and they are divided into 6 types of major instruction format. This section specifies the instruction set architecture, includes instruction format, detailed description of each instruction, and instruction encoding.

### 3.1 Instruction Format

	31	26	25	21	20	16	15	10	9	5	4	0
Format 1	OP1	R		R		OP2		R		R		
Format 2	OP1	R		R		OP2		---		R		
Format 3	OP1	R		R		OP2		0		I5		
Format 4	OP1	R		R/I5		OP2		I10				
Format 5	OP1	R		R		I16						
Format 6	OP1	R		I21								

**Figure 3-1 Instruction Format**

There are 6 major format types as show in Figure 3-1. The instruction format was composed through operator fields and operand fields.

#### 3.1.1 Operator field

Operator is defined as major operator OP1 and minor operator OP2. Bits [31:26] of a instruction defines OP1 and bits [15:10] of a instruction defines OP2. Every instruction has it major OP field and most of them also have a minor OP field. Some instructions have no minor OP field, its major OP field uniquely specify its operation. In this case, minor OP is generally taken as part of an operand.

#### 3.1.2 Operand field

There are two kinds of operands, register operand and immediate number operand. For register operand, the instruction encode a 5 bits register number; for immediate operand, there are 4 kinds of immediate number according to the encoding length.

- **R:** there are 4 register field in the instruction encoding, bits [25..21], bits [20..16], bits [9..5] and bits [4..0]. These fields contains the register numbers which index into GRF.
- **I5:** 5 bits length immediate number, encoded in bits[4..0] and bits[20..16]. Some instructions interpret the 5 bits immediate number as a signed number, denoted as S5, while some instructions interpret the 5 bits immediate number as an unsigned number, denoted as U5;
- **I10:** 10 bits length immediate number, encoded in bits[9..0]. Some instructions interpret the 10 bits immediate number as a signed number, denoted as S10, while some instructions interpret the 10 bits immediate number as an unsigned number, denoted as U10.
- **I16:** 16 bits length immediate number, encoded in bits[15..0]. Only logical instructions use this field and it is interpreted as signed number in this case, denoted as S16.

- **I21:** 21 bits length immediate number, encoded in bits[20..0]. Some instructions interpret the 21 bits immediate number as a signed number, denoted as S21, while some instructions interpret the 21 bits immediate number as an unsigned number, denoted as U21.

The formats of some system control instructions don't strictly follow the above conventions. For example, RTE/SLEEP/SBRK have no operand; ITLB/DTLB/ICACHE/DCACHE instructions take bits[25..21] as an immediate operands.

### 3.1.3 Instructions and format summary

Table 3-1 gives a summary of all Arca instructions. OP1 and OP2 are merged to a instruction mnemonic such as ADD. Operands follow the notation described in the above section.

**Table 3-1 Arca instructions and format**

Class	Instruction format				Class	Instruction format			
Imm Load	LHI	R,	U21		Load Store	LD8	R,	R,	S10
Jump	J	R,	S21			LR8	R,	R,	R
	JA	R,	R,	U10		LD8U	R,	R,	S10
Branch	BEQ	R,	R,	S10		LR8U	R,	R,	R
	BNE	R,	R,	S10		LD16	R,	R,	S10
	BLT	R,	R,	S10		LR16	R,	R,	R
	BGE	R,	R,	S10		LX16	R,	R,	R
	BLTU	R,	R,	S10		LD16U	R,	R,	S10
	BGEU	R,	R,	S10		LR16U	R,	R,	R
	BEQI	R,	S5,	S10		LX16U	R,	R,	R
	BNEI	R,	S5,	S10		LD32	R,	R,	S10
	BLTI	R,	S5,	S10		LR32	R,	R,	R
	BGEI	R,	S5,	S10		LX32	R,	R,	R
	BEQIU	R,	S5,	S10		SD8	R,	R,	S10
	BNEIU	R,	S5,	S10		SR8	R,	R,	R
	BLTIU	R,	S5,	S10		SD16	R,	R,	S10
	BGEIU	R,	S5,	S10		SR16	R,	R,	R
Arithmetic	MULU	R,	R,	R,		SX16	R,	R,	R
		R				SD32	R,	R,	S10
	ADD	R,	R,	R		SR32	R,	R,	R
	SUB	R,	R,	R		SX32	R,	R,	R
	ADDI	R,	R,	S10					
Compare	SEQ	R,	R,	R	Misc	MVZ	R,	R,	R
	SNE	R,	R,	R		MVNZ	R,	R,	R
	SLT	R,	R,	R		CSB		R,	R
	SGE	R,	R,	R		BREV	R,	R	
	SLTU	R,	R,	R		SWAP	R,	R,	S10
	SGEU	R,	R,	R					
Logical	AND	R,	R,	R	System	SLEEP			
	OR	R,	R,	R		SBRK			
	XOR	R,	R,	R		RTE			
	ANDN	R,	R,	R		TRAP	R,	U10	
	ANDI	R,	R,	S16		RCR	R,	U5	
	ORI	R,	R,	S16		WCR	R,	U5	
	XORI	R,	R,	S16					
Shift	SLL	R,	R,	R		CLD	R,	U5, U10	
	SLR	R,	R,	R		CST	R,	U5, U10	
	SAR	R,	R,	R		ITLB	U5, R		
	SLLI	R,	R,	U5		DTLB	U5, R		
	SLRI	R,	R,	U5		Icache	U5, R, S10		
	SARI	R,	R,	U5		Dcache	U5, R, S10		

## 3.2 Instruction Description

### 3.2.1 Immediate load instruction

Immediate operands are very common in typical programs. Many instructions encode the immediate operand as one source operand to allow a range of constant values to be operated directly. But if the required constant value does not fit the range, then the immediate operand must be loaded separately. Some instructions do not admit an immediate operand, and the immediate operand also must be loaded into register before the operation.

Only one instruction LHI is provided for loading a large immediate operand, because loading the little one can be implemented by **ORI** instruction. **ORI** permit a 16-bit immediate operand as its source operand, or the immediate with register R0 to the destination register, just means loading the 16-bit immediate to the destination:

```
ORI    Ra, R0, 128        ! load 16 bits immediate to Ra
```

If the immediate has 32 bits, LHI is used to load the high 21 bits of the immediate to the destination register, and other bits can be loaded by ORI instruction:

```
LHI     Ra, 1000          ! load high 21 bits to Ra
ORI     Ra, Ra, 126       ! load low 11 bits to Ra
```

R0 is used to denote a register that always reads as zero and ignores writes. This can be used to make any register operand take a zero value. It is very useful for zero is a particularly common constant. R0 can also be used as the destination register to discard the result of an instruction.

## LHI



### Syntax:

**LHI**      *Ra*, *U21*

### Operation:

$Ra = U21 \ll 11$

### Description:

Load the 21 bits unsigned immediate into the high 21 bits of register Ra. The low 11 bits of Ra are filled with Zero.

### Example:

**LHI**      R1, 0x123456

Before execution: R1= 0xffffffff

After execution:    R1= 0x91a2b000

### 3.2.2 Jump Instructions

There are two jump instructions J and JA provided. J jumps to PC relative to its immediate operand as the target address, allows a jump forwards or backwards up to 4MB. And JA absolutely jumps to the target address generated by adding the contents held in the source register operand with the immediately constant operand, then provides a way to jump anywhere in the 4GB address space.

Both of them save the following instruction address to their destination register, this is a link mechanism, since it allows the target instruction sequence to return control back to the instruction sequence that invoked it. It is typically used to implement standard call and return mechanisms. The choice of link register is not fixed by the 32-bit instruction set.

The write to the destination register can be defeated using R0:

```
J    R0, 0x123      ! jump to PC + 4 + 0x123 without link
JA   R0, Ra, 0      ! jump to Ra without link
```

These can be used to achieve an unconditional branch and a return without a link.

## J



### Syntax:

J            Ra, S21

### Operation:

$Ra = PC + 4$   
 $temp = PC + 4 + (S21 \ll 2)$   
 $PC = temp$

### Description:

PC is the address of the jump instruction. The address of the instruction following it is saved in register Ra. The immediate S21 is left shifted two bits, sign extended to 32 bits then added to the address of the instruction following it, and the result is set to PC.

### Example:

1):

J            R1, 0x1234

Before execution: PC= 0x00001234, R1= 0xffffffff

After execution:    PC= 0x00005b08, R1= 0x00001238

2):

J            R1, 0x1fedcc

Before execution: PC= 0x00008a90, R1= 0xffffffff

After execution:    PC= 0x000041c4, R1= 0x00008a94



## JA

Op: 000000	Ra	Rb	Ext: 000010	U10
------------	----	----	-------------	-----

**Syntax:**

*JA Ra, Rb, U10*

**Operation:**

temp = PC + 4  
PC = Rb | (U10 << 2)  
Ra = temp

**Description:**

PC is the address of the jump instruction. The address of the instruction following it is saved in register Ra. The immediate U10 is left shifted two bits, unsigned extended to 32 bits then or to the address held in Rb, and the result is set to PC.

**Example:**

JA R0, R1, 0xf0

Before execution: PC= 0x00001000, R1= 0x00056ac8

After execution: PC= 0x00056af8, R1= 0x00056ac8

### 3.2.3 Branch Instructions

Branch instructions perform the operation of two operands and decide whether to replace PC according to the result. If the result is true, PC will be assigned by the target address that is computed with their immediate operand, else the control flow just falls through to the following instruction. They are classified into two classes as presented in the following.

#### 3.2.3.1 Reg-Reg-Compare-Branch instructions

This class Branch instructions perform the comparison of two source register operands and decide whether to replace PC according to the comparison result. If the result is true, PC will be assigned by the target address that is computed with their immediate operand, else the control flow just falls through to the following instruction.

There are six instructions BEQ, BNE, BLT, BGE, BLTU and BGEU in this instruction class that has the same instruction formats. They perform different comparisons separately: equal, not equal, less than, greater than or equal, unsigned less than, unsigned greater than or equal. Other kinds of comparison can be implemented by swapping the two source register operands.

greater than:	$i > j$ is same as $j < i$
less than or equal:	$i \leq j$ is same as $j \geq i$

## BEQ BNE BLT BGE BLTU BGEU

<b>BEQ:</b>				
Op: 000110	Ra	Rb	Ext: 000110	S10
<b>BNE:</b>				
Op: 000110	Ra	Rb	Ext: 000010	S10
<b>BLT:</b>				
Op: 000110	Ra	Rb	Ext: 000100	S10
<b>BLTU:</b>				
Op: 000110	Ra	Rb	Ext: 001100	S10
<b>BGE:</b>				
Op: 000110	Ra	Rb	Ext: 000000	S10
<b>BGEU:</b>				
Op: 000110	Ra	Rb	Ext: 001000	S10

### Syntax:

```

BEQ      Rb, Ra, S10
BNE      Rb, Ra, S10
BLT      Rb, Ra, S10
BLTU     Rb, Ra, S10
BGE      Rb, Ra, S10
BGEU     Rb, Ra, S10

```

### Operation:

```

target = PC + 4 + (S10 << 2)
following = PC + 4
If (Rb op Ra) {
    PC = target
}
else {
    PC = following
}

```

### Description:

PC is the address of this branch instruction. The immediate S10 is left shifted by 2, sign extended to 32 bits and added the address of the instruction following it, and the result is the target address of the branch. If registers Ra and Rb satisfy the compare condition, the new PC is the address of the branch target, else the new PC is the address of the following instruction.

**Example:**

1):

BEQ R1, R2, 0x40

Before execution: PC= 0x00001234, R1= 0x0000a213, R2= 0x0000a213

After execution: PC= 0x00001338, R1= 0x0000a213, R2= 0x0000a213

2):

BEQ R1, R2, 0x40

Before execution: PC= 0x00001234, R1= 0xffff458b, R2= 0x0000a213

After execution: PC= 0x00001238, R1= 0xffff458b, R2= 0x0000a213

3):

BLT R1, R2, 0x80

Before execution: PC= 0x00001234, R1= 0x0000a213, R2= 0xffff45b80

After execution: PC= 0x00001238, R1= 0x0000a213, R2= 0xffff45b8

4):

BLTU R1, R2, 0x80

Before execution: PC= 0x00001234, R1= 0x0000a213, R2= 0xffff45b80

After execution: PC= 0x00001438, R1= 0x0000a213, R2= 0xffff45b80

### 3.2.3.2 Reg-Imm-Compare-Branch instructions

This class Branch instructions perform the comparison of the register operand and the first immediate operand, then decide whether to replace PC according to the comparison result. If the result is true, PC will be assigned by the target address that is computed with the second immediate operand, else the control flow just falls through to the following instruction.

There are such instructions as BEQL, BNEI, BEQUI, BNEUI, BLTI, BGEI, BLTUI, BGEUI in this instruction class that has the same instruction formats. They perform different comparisons separately: equal, not equal, less than, greater than or equal, unsigned less than, unsigned greater than or equal. Other kinds of comparison can be implemented by decrease and increase the first immediate operand by 1.

greater than:	$i > \text{imm}$ is same as $i \geq \text{imm} + 1$
less than or equal:	$i \leq \text{imm}$ is same as $i < \text{imm} + 1$

Note unsigned flag 'U' is meaningful for EQ/NQ comparison: when U=1, the unsigned number {0 ~ 31} is compared with Rb; when U=0, signed number {-16, 15} is compared.

## BEQI BNEI BEQUI BNEUI BLTI BGEI BLTUI BGEUI

BEQI :				
Op: 000100	S5	Rb	Ext: 000110	S10
BNEI :				
Op: 000100	S5	Rb	Ext: 000010	S10
BEQUI :				
Op: 000100	U5	Rb	Ext: 001110	S10
BNEUI :				
Op: 000100	U5	Rb	Ext: 001010	S10
BLTI :				
Op: 000100	S5	Rb	Ext: 000100	S10
BLTUI :				
Op: 000100	U5	Rb	Ext: 001100	S10
BGEI :				
Op: 000100	S5	Rb	Ext: 000000	S10
BGEUI :				
Op: 000100	U5	Rb	Ext: 001000	S10

### Syntax:

```
BEQI      Rb, S5, S10
BNEI      Rb, S5, S10
BEQUI     Rb, U5, S10
BNEUI     Rb, U5, S10
BLTI      Rb, S5, S10
BLTUI     Rb, U5, S10
BGEI      Rb, S5, S10
BGEUI     Rb, U5, S10
```

### Operation:

```
If (U == 1)
    imm = U5;
else
    imm = S5
target = PC + 4 + (S10 << 2)
following = PC + 4
If (Rb op imm) {
    PC = target
}
else {
    PC = following
}
```

### Description:

PC is the address of this branch instruction. The immediate S10 is left shifted by 2, sign extended to 32 bits and added the address of following instruction, and the result is the target address of the branch. If Rb and Imm satisfy the compare condition, the new PC is the address of the branch target, else the new PC is the address of the following instruction.

**Example:**

1):

```
BEQI R1, 0x3, 0x40
```

Before execution: PC= 0x00001234, R1= 0x00000003

After execution: PC= 0x00001338, R1= 0x00000003

2):

```
BEQI R1, 0x1, 0x40
```

Before execution: PC= 0x00001234, R1= 0x00000003

After execution: PC= 0x00001238, R1= 0x00000003

3):

```
BLTI R1, -1, 0x80
```

Before execution: PC= 0x00001234, R1= 0x00000003

After execution: PC= 0x00001238, R1= 0x00000003

4):

```
BLTUI R1, 31, 0x80
```

Before execution: PC= 0x00001234, R1= 0x00000003

After execution: PC= 0x00001438, R1= 0x00000003

### 3.2.4 Arithmetic Instructions

Arithmetic operation addition, subtraction and multiplication are supported in arithmetic instructions. Division, remainder and other arithmetic operations should be implemented by software.

Instruction **ADD** performs an addition of two register operands, and **ADDI** admit an immediate operand as its source operand.

Only one instruction **SUB** is supported for subtraction between two registers, because subtract an immediate operand can be achieved by adding the negation of this immediate. If R0 is used as the minuend, SUB can achieve the register negation:

```
SUB    Ra, R0, Rb        ! negation of Rb into Ra
```

Instruction **MULU** is provided for unsigned 32-bit to 64-bit multiplication. It is 4-register-operand instruction, accomplishing multiplication in one or more cycles with the 64-bit result putting to two destination registers.





## ADD

Op: 000010	Rc	Rb	Ext: 000000	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**ADD**      *Ra*, *Rb*, *Rc*

**Operation:**

$Ra = Rb + Rc$

**Description:**

Register Rc is added to register Rb, and store the result in the destination register Ra.

**Example:**

**ADD**    R3, R1, R2

Before execution: R1= 0x0000a213, R2= 0xff451bc0, R3= 0xffffffff

After execution:    R1= 0x0000a213, R2= 0xff451bc0, R3= 0xff45bdd3

## ADDI

Op: 000000	Ra	Rb	Ext: 000000	S10
------------	----	----	-------------	-----

**Syntax:**

**ADDI**     *Ra, Rb, S10*

**Operation:**

$Ra = Rb + \text{SignExtend}(S10)$

**Description:**

Immediate S10 is sign-extended and added to register Rb, the result is written into register Ra.

**Example:**

1):

**ADDI**   *R3, R1, 0x34*

Before execution: R1= 0x0000a213, R3= 0xffffffff

After execution:    R1= 0x0000a213, R3= 0x0000a247

2):

**ADDI**   *R3, R1, 0x334*

Before execution: R1= 0x0000a213, R3= 0xffffffff

After execution:    R1= 0x0000a213, R3= 0x0000a147

## SUB

Op: 000010	Rc	Rb	Ext: 100000	Don't care	Ra
------------	----	----	-------------	------------	----

### Syntax:

`SUB      Ra, Rb, Rc`

### Operation:

$Ra = Rb - Rc$

### Description:

Register Rc is subtracted from register Rb, and store the result in the destination register Ra.

### Example:

1):

`SUB    R3, R1, R2`

Before execution: R1= 0x0000a213, R2= 0xff451bc0, R3= 0xffffffff

After execution:    R1= 0x0000a213, R2= 0xff451bc0, R3= 0x00bb8653

2):

`SUB    R1, R0, R2            ! negation of R2 into R1`

Before execution: R1= 0x0000a213, R2= 0xff451bc0

After execution:    R1= 0x000bae440, R2= 0xff451bc0

## MULU

Op: 000010	Rc	Rb	Ext: 010001	Rh	Ra
------------	----	----	-------------	----	----

**Syntax:**

**MULU**      *Rh, Ra, Rb, Rc*

**Operation:**

$Rh:Ra = (\text{Unsigned})Rb * (\text{Unsigned})Rc$

**Description:**

Multiply the two 32-bit unsigned data in register Rb and Rc, and put the 64-bit result into Rh:Ra. Especially, if R0 acts as Rh, the higher 32-bit result will be discarded, the lower 32-bit result is put to Ra.

**Example:**

1):

**MULU**   *R3, R4, R1, R2*

Before execution: R1= 0x0000a213, R2= 0xff451bc0,  
R3= 0x00000000, R4= 0x00000000

After execution:    R1= 0x0000a213, R2= 0xff451bc0,  
R3= 0x0000a19c, R4= 0xadb08f40

### 3.3 Comparison Instructions

Comparison instructions perform the comparison of their two source register operands, and store the comparison result to the destination register. Here the comparison result is a boolean value: 1 indicates the condition of comparison is satisfied, and 0 means the condition is not satisfied.

There are six instructions **SEQ**, **SNE**, **SLT**, **SGE**, **SLTU** and **SGEU** in this instruction class that has the same instruction formats. They perform different comparisons separately: *equal*, *not equal*, *less than*, *greater than or equal*, *unsigned less than*, *unsigned greater than or equal*. Other kinds of comparison can be implemented by swapping the two source register operands.

greater than:	$i > j$ is same as $j < i$
less than or equal:	$i \leq j$ is same as $j \geq i$

Immediate comparison is not supported, so the immediate operand must be loaded into register separately. But for the comparison with zero, R0 can be used as one source register operand and only one single comparison instruction is needed.

## SEQ SNE SLT SGE SLTU SGEU

<b>SEQ:</b>					
Op: 000010	Rc	Rb	Ext: 010110	Don't care	Ra
<b>SNE:</b>					
Op: 000010	Rc	Rb	Ext: 010010	Don't care	Ra
<b>SLT:</b>					
Op: 000010	Rc	Rb	Ext: 010100	Don't care	Ra
<b>SLTU:</b>					
Op: 000010	Rc	Rb	Ext: 011100	Don't care	Ra
<b>SGE:</b>					
Op: 000010	Rc	Rb	Ext: 010000	Don't care	Ra
<b>SGEU:</b>					
Op: 000010	Rc	Rb	Ext: 011000	Don't care	Ra

**Syntax:**

<b>SEQ</b>	<i>Ra, Rb, Rc</i>
<b>SNE</b>	<i>Ra, Rb, Rc</i>
<b>SLT</b>	<i>Ra, Rb, Rc</i>
<b>SLTU</b>	<i>Ra, Rb, Rc</i>
<b>SGE</b>	<i>Ra, Rb, Rc</i>
<b>SGEU</b>	<i>Ra, Rb, Rc</i>

**Operation:**

```
If (Rb op Rc) {  
    Ra = 1  
}  
else {  
    Ra = 0  
}
```

**Description:**

Register Rb is compared with Rc. 1 is written to register Ra if the compare condition is satisfied. Otherwise 0 is written to register Ra.

**Example:**

1):

`SEQ R3, R1, R2`

Before execution: R1= 0x0000a213, R2= 0x0000a213, R3= 0xffffffff

After execution: R1= 0x0000a213, R2= 0x0000a213, R3= 0x00000001

2):

`SNE R3, R1, R2`

Before execution: R1= 0xffff458b, R2= 0x00000000, R3= 0x0000a213

After execution: R1= 0xffff458b, R2= 0x00000000, R3= 0x00000001

3):

`SLT R1, R3, R0``SLTU R5, R6, R7`

Before execution: R1= 0xffffffff, R3= 0xffff4a80,

R5= 0x00001234, R6= 0xffff45b8, R7= 0x00004000

After execution: R1= 0x00000001, R3= 0xffff4a80,

R5= 0x00000000, R6= 0xffff45b8, R7= 0x00004000

**3.3.1**

### 3.3.2 Bitwise Instructions

Bitwise instructions **AND**, **OR**, **XOR**, **ANDN** perform the bitwise operation *and*, *or*, *exclusive-or*, and *not* with two register source operands. The corresponding instructions **ANDI**, **ORI**, **XORI** have an immediate operand as one source operand, they are also supported because this kind of bitwise operations often occur in the programs. Bitwise operation *not* is not provided directly, but it can be achieved by using immediate -1 as one source operand of XORI instruction:

```
XORI    Ra, Rb, -1           ! bitwise not of Rb into Ra
```

Other bitwise instructions can be implemented by combining the above bitwise instructions.

To copy a register's content to another register, **OR** instruction with R0 as its source operand can be used:

```
OR      Ra, R0, Rb           ! copy Rb to Ra
```

Instruction **ORI** can perform a constant loading when uses R0 as its source operand.



## AND OR XOR ANDN

### AND:

Op: 000010	Rc	Rb	Ext: 000001	Don't care	Ra
------------	----	----	-------------	------------	----

### OR:

Op: 000010	Rc	Rb	Ext: 000010	Don't care	Ra
------------	----	----	-------------	------------	----

### XOR:

Op: 000010	Rc	Rb	Ext: 000011	Don't care	Ra
------------	----	----	-------------	------------	----

### ANDN:

Op: 000010	Rc	Rb	Ext: 100001	Don't care	Ra
------------	----	----	-------------	------------	----

### Syntax:

```

AND      Ra, Rb, Rc
OR       Ra, Rb, Rc
XOR      Ra, Rb, Rc
ANDN     Ra, Rb, Rc

```

### Operation:

```

Ra = Rb & Rc           ! AND Ra, Rb, Rc
Ra = Rb | Rc           ! OR  Ra, Rb, Rc
Ra = Rb ^ Rc           ! XOR Ra, Rb, Rc
Ra = Rb & ~Rc          ! ANDN Ra, Rb, Rc

```

### Description:

Register Rb is {AND, OR, XOR, AND NOT} with Register Rc, the result is written to the destination register Ra.

**Example:**

1):

AND R3, R1, R2

Before execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0xffffffff

After execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0x12041240

2):

OR R3, R1, R2

Before execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0xffffffff

After execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0xff755ff8

3):

OR R1, R0, R2 ! copy R2 to R1

Before execution: R1= 0x0000a213, R2= 0xff451bc0

After execution: R1= 0xff451bc0, R2= 0xff451bc0

4):

XOR R3, R1, R2

Before execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0xffffffff

After execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0xed714db8

5):

ANDN R3, R1, R2

Before execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0xffffffff

After execution: R1= 0x12345678, R2= 0xff451bc0, R3= 0x304438

## ANDI ORI XORI

**ANDI:**



**ORI:**



**XORI:**



### Syntax:

```

ANDI    Ra, Rb, S16
ORI     Ra, Rb, S16
XORI    Ra, Rb, S16
  
```

### Operation:

```

Ra = Rb & SignExtend (S16)      ! ANDI Ra, Rb, S16
Ra = Rb | SignExtend (S16)      ! ORI  Ra, Rb, S16
Ra = Rb ^ SignExtend (S16)      ! XORI Ra, Rb, S16
  
```

### Description:

Immediate S16 is sign-extended and then {AND, OR, XOR} with register Rb, the result is written to the destination register Ra.

### Example:

1):

```
ANDI R1, R2, 0xf3
```

Before execution: R1= 0x12345678, R2= 0xff451bc0

After execution: R1= 0x000000c0, R2= 0xff451bc0

2):

```
ORI R1, R2, 0xf4
```

Before execution: R1= 0x12345678, R2= 0xff451bc0

After execution: R1= 0xff451bf4, R2= 0xff451bc0

3):

```
ORI R3, R0, 0xf234 ! load immediate to R3
```

Before execution: R3= 0xffffffff

After execution: R3= 0xfffff234

4):

```
XORI R1, R2, 0xff6a
```

Before execution: R1= 0x12345678, R2= 0xff451bc0

After execution: R1= 0x12345678, R2= 0x00bae4aa

### 3.3.3 Shift Instructions

There are three types of shift operations: *logical left*, *logical right* and *arithmetical right*. Arithmetical left operation is not necessary because it does the same operation with logical left shift. The instruction set supports these three operations directly:

- **SLL/SLLI**: perform logical left shift operation, shift the source operand left and insert zero bits to the least significant bits.
- **SLR/SLRI**: perform logical right shift operation, shift the source operand right and insert zero bits to the most significant bits.
- **SAR/SARI**: perform arithmetic right shift operation, shift the source operand right and insert sign bits to the most significant bits, so that the sign of the destination is the same as the source.

For **SLLI/SLRI/SARI**, the shift amount is specified by the immediate operand. And for **SLL/SLR/SAR**, the shift amount is specified by another source register.

Because the shifted operand is a register, it only has 32 bits, the shift amount should be less than or equal to this number, and the negative shift amount is meaningless. For this reason, the immediate operand is a 5 bits unsigned number to specify the shift amount in instructions **SLLI/SLRI/SARI**. And only the low 5 bits of shift amount register operand are recognized in instructions **SLL/SLR/SAR**. The high 27 bits are simply ignored.

## SLL SLR SAR

<b>SLL:</b>					
Op: 000010	Rc	Rb	Ext: 000100	Don't care	Ra
<b>SLR:</b>					
Op: 000010	Rc	Rb	Ext: 001000	Don't care	Ra
<b>SAR:</b>					
Op: 000010	Rc	Rb	Ext: 001100	Don't care	Ra

### Syntax:

**SLL**      *Ra, Rb, Rc*  
**SLR**      *Ra, Rb, Rc*  
**SAR**      *Ra, Rb, Rc*

### Operation:

$Ra = Rb \ll (Rc \& 0x1F)$                       ! SLL *Ra, Rb, Rc*  
 $Ra = Rb \gg (Rc \& 0x1F)$  (logical)            ! SLR *Ra, Rb, Rc*  
 $Ra = Rb \gg (Rc \& 0x1F)$  (arithmetic)        ! SAR *Ra, Rb, Rc*

### Description:

Register *Rb* is shifted {*logical left, logical right, arithmetical right*} by register *Rc* (the value of 0-4 bits), the result is stored in the destination register *Ra*.

### Example:

1):

SLL    *R3, R1, R2*

Before execution: *R1*= 0x12345678, *R2*= 0x00000004, *R3*= 0xffffffff

After execution:    *R1*= 0x12345678, *R2*= 0x00000004, *R3*= 0x23456780

2):

SLR    *R3, R1, R2*

Before execution: *R1*= 0x87654321, *R2*= 0x00000008, *R3*= 0xffffffff

After execution:    *R1*= 0x87654321, *R2*= 0x00000008, *R3*= 0x00876543

3):

SAR    *R3, R1, R2*

Before execution: *R1*= 0x87654321, *R2*= 0x00000008, *R3*= 0xffffffff

After execution:    *R1*= 0x87654321, *R2*= 0x00000008, *R3*= 0xff876543

## SLLI SLRI SARI

<b>SLLI :</b>					
Op: 000000	Ra	Rb	Ext: 000100	0	U5
<b>SLRI :</b>					
Op: 000000	Ra	Rb	Ext: 001000	0	U5
<b>SARI :</b>					
Op: 000000	Ra	Rb	Ext: 001100	0	U5

### Syntax

**SLLI**     *Ra, Rb, U5*  
**SLRI**     *Ra, Rb, U5*  
**SARI**     *Ra, Rb, U5*

### Operation:

Ra = Rb << U5                                ! SLLI Ra, Rb, U5  
Ra = Rb >> U5 (logical)                    ! SLRI Ra, Rb, U5  
Ra = Rb >> U5 (arithmetical)            ! SARI Ra, Rb, U5

### Description:

Register Rb is shifted {*logical left, logical right, arithmetical right*} by unsigned immediate U5, the result is stored in the destination register Ra.

### Example:

1):  
    SLLI R3, R1, 5

Before execution: R1= 0x12345678, R3= 0xffffffff  
After execution:    R1= 0x12345678, R3= 0x468acf00

2):  
    SLRI R3, R1, 8

Before execution: R1= 0x87654321, R3= 0xffffffff  
After execution:    R1= 0x87654321, R3= 0x00876543

3):  
    SARI R3, R1, 7

Before execution: R1= 0x87654321, R3= 0x00000000  
After execution:    R1= 0x87654321, R3= 0xff0eca86

### 3.3.4 Load and Store Instructions

Load and store instructions transfer data between register and memory. This class of instruction has 21 instructions.

Table 3-2 lists all the load and store instructions with three address modes:

- $R + R$ : the effective address is formed by adding two source registers. This register offsets are useful for accessing arrays or blocks of data.
- $R + R \ll (1, 2)$ : The effective address is formed by adding one register with the value of another register shifting by 1 or 2. This register offsets are useful for accessing arrays.
- $R + Imm$ : the effective address is formed by adding one source register with a 16 bits signed value. This immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and input/output registers.

**Table 3-2 Load and Store Instructions**

	Address Mode	Signed Byte	Unsigned Byte	Signed Halfword	Unsigned Halfword	Word
Load	$R+R$	LR8	LR8U	LR16	LR16U	LR32
	$R+(R \ll (1,2))$			LX16	LX16U	LX32
	$R+Imm$	LD8	LD8U	LD16	LD16U	LD32
Store	$R+R$	SR8		SR16		SR32
	$R+(R \ll (1,2))$			SX16		SX32
	$R+Imm$	SD8		SD16		SD32

For loading a byte or a halfword from memory, we need to distinguish the sign of the value to perform the correct extension into the destination register. When the loaded value is a signed value, the value is loaded from the effective address and sign extended to the destination register. And when the loaded value is an unsigned value, the value is loaded from the effective address and zero extended to the destination register.

Word and halfword addresses must be aligned on word (two LSBs are 0) and halfword (LSB is 0) boundaries, respectively.

## LR8

Op: 001010	Rc	Rb	Ext: 000110	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

LR8      *Ra*, *Rb*, *Rc*

**Operation:**

$Ra = \text{SignExtend}(\text{Mem}[Rb + Rc])$

**Description:**

Load a byte from the effective address formed by adding registers Rb and Rc. The byte is sign-extended into the destination register Ra.

**Example:**

1):

LR8      R3, R1, R2

Before execution: R1= 0x20005678, R2= 0x00000010, R3= 0xffffffff  
Mem[20005688]= 0x78

After execution: R1= 0x20005678, R2= 0x00000010, R3= 0x00000078

2):

LR8      R3, R1, R2

Before execution: R1= 0x20005679, R2= 0x00000010, R3= 0xffffffff  
Mem[20005689]= 0x87

After execution: R1= 0x20005679, R2= 0x00000010, R3= 0xffffffff87



## LR8U

Op: 001010	Rc	Rb	Ext: 001110	Don't care	Ra
------------	----	----	-------------	------------	----

### Syntax:

**LR8U**     *Ra*, *Rb*, *Rc*

### Operation:

$Ra = \text{ZeroExtend}(\text{Mem}[Rb + Rc])$

### Description:

Load a byte from the effective address formed by adding registers Rb and Rc. The byte is zero-extended into the destination register Ra.

### Example:

1):

**LR8U**   *R3*, *R1*, *R2*

Before execution:  $R1 = 0x20005678$ ,  $R2 = 0x00000010$ ,  $R3 = 0xffffffff$   
                              $\text{Mem}[20005688] = 0x78$

After execution:     $R1 = 0x20005678$ ,  $R2 = 0x00000010$ ,  $R3 = 0x00000078$

2):

**LR8U**   *R3*, *R1*, *R2*

Before execution:  $R1 = 0x20005679$ ,  $R2 = 0x00000010$ ,  $R3 = 0xffffffff$   
                              $\text{Mem}[20005689] = 0x87$

After execution:     $R1 = 0x20005679$ ,  $R2 = 0x00000010$ ,  $R3 = 0x00000087$

## LR16

Op: 001010	Rc	Rb	Ext: 000010	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**LR16**      *Ra*, *Rb*, *Rc*

**Operation:**

$Ra = \text{SignExtend}(\text{Mem}[Rb + Rc])$

**Description:**

Load a halfword from the effective address formed by adding registers *Rb* and *Rc*. The halfword is sign-extended into the destination register *Ra*.

**Example:**

1):

**LR16**   *R3*, *R1*, *R2*

Before execution: *R1*= 0x2000567a, *R2*= 0x00000010, *R3*= 0xffffffff  
                    Mem[2000568a]= 0xabcd

After execution:    *R1*= 0x2000567a, *R2*= 0x00000010, *R3*= 0xffffabcd

2):

**LR16**   *R3*, *R1*, *R2*

Before execution: *R1*= 0x2000567c, *R2*= 0x00000010, *R3*= 0xffffffff  
                    Mem[2000568c]= 0x1234

After execution:    *R1*= 0x2000567c, *R2*= 0x00000010, *R3*= 0x00001234

## LR16U

Op: 001010	Rc	Rb	Ext: 001010	Don't care	Ra
------------	----	----	-------------	------------	----

### Syntax:

**LR16U**     *Ra*, *Rb*, *Rc*

### Operation:

$Ra = \text{ZeroExtend}(\text{Mem}[Rb + Rc])$

### Description:

Load a halfword from the effective address formed by adding registers Rb and Rc. The halfword is zero-extended into the result register Ra.

### Example:

1):

**LR16U**     R3, R1, R2

Before execution: R1= 0x2000567a, R2= 0x00000010, R3= 0xffffffff  
                     Mem[2000568a]= 0xabcd

After execution:    R1= 0x2000567a, R2= 0x00000010, R3= 0x0000abcd

2):

**LR16U**     R3, R1, R2

Before execution: R1= 0x2000567c, R2= 0x00000010, R3= 0xffffffff  
                     Mem[2000568c]= 0x1234

After execution:    R1= 0x2000567c, R2= 0x00000010, R3= 0x00001234

## LR32

Op: 001010	Rc	Rb	Ext: 000000	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**LR32**     *Ra*, *Rb*, *Rc*

**Operation:**

$Ra = \text{Mem}[Rb + Rc]$

**Description:**

Load a word from the effective address formed by adding registers Rb and Rc, and store the word into the destination register Ra.

**Example:**

**LR8**     R3, R1, R2

Before execution: R1= 0x20005678, R2= 0x00000010, R3= 0xffffffff  
                    Mem[20005688]= 0x7887abcd

After execution:    R1= 0x20005678, R2= 0x00000010, R3= 0x7887abcd

## LX16

Op: 001010	Rc	Rb	Ext: 010010	Don't care	Ra
------------	----	----	-------------	------------	----

### Syntax:

**LX16**      *Ra*, *Rb*, *Rc*

### Operation:

$Ra = \text{SignExtend}(\text{Mem}[Rb + (Rc \ll 1)])$

### Description:

Load a halfword from the effective address formed by adding registers Rb and the value of register Rc shifting by 1. The halfword is sign-extended into the destination register Ra.

### Example:

1):

**LX16**   *R3*, *R1*, *R2*

Before execution: R1= 0x2000567a, R2= 0x00000008, R3= 0xffffffff  
                     Mem[2000568a]= 0xabcd

After execution:    R1= 0x2000567a, R2= 0x00000008, R3= 0xffffabcd

2):

**LX16**   *R3*, *R1*, *R2*

Before execution: R1= 0x2000567c, R2= 0x00000008, R3= 0xffffffff  
                     Mem[2000568c]= 0x1234

After execution:    R1= 0x2000567c, R2= 0x00000008, R3= 0x00001234

## LX16U

Op: 001010	Rc	Rb	Ext: 011010	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**LX16U**     *Ra*, *Rb*, *Rc*

**Operation:**

$Ra = \text{ZeroExtend}(\text{Mem}[Rb + (Rc \ll 1)])$

**Description:**

Load a halfword from the effective address formed by adding registers Rb and the value of Rc shifting by 1. The halfword is zero-extended into the result register Ra.

**Example:**

1):

**LX16U**     R3, R1, R2

Before execution: R1= 0x2000567a, R2= 0x00000008, R3= 0xffffffff  
Mem[2000568a]= 0xabcd

After execution: R1= 0x2000567a, R2= 0x00000008, R3= 0x0000abcd

2):

**LX16U**     R3, R1, R2

Before execution: R1= 0x2000567c, R2= 0x00000008, R3= 0xffffffff  
Mem[2000568c]= 0x1234

After execution: R1= 0x2000567c, R2= 0x00000008, R3= 0x00001234

## LX32

Op: 001010	Rc	Rb	Ext: 010000	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**LX32**      *Ra*, *Rb*, *Rc*

**Operation:**

$Ra = \text{Mem}[Rb + (Rc \ll 2)]$

**Description:**

Load a word from the effective address formed by adding registers *Rb* and the value of *Rc* shifting by 2, and store the word into the destination register *Ra*.

**Example:**

**LX32**   *R3*, *R1*, *R2*

Before execution: *R1*= 0x20005678, *R2*= 0x00000004, *R3*= 0xffffffff  
                    Mem[20005688]= 0x7887abcd

After execution:    *R1*= 0x20005678, *R2*= 0x00000004, *R3*= 0x7887abcd

## LD8

**Syntax:**

**LD8**                    *Ra, Rb, S10*

**Operation:**

$Ra = \text{SignExtend}(\text{Mem}[Rb + \text{SignExtend}(S10)])$

**Description:**

Load a byte from the effective address formed by adding register Rb to the 10-bit signed immediate S10. The byte is sign-extended into the register Ra.

**Example:**

1):

**LD8**    *R3, R1, 0x3f0*

Before execution: R1= 0x20005698, R3= 0xffffffff

Mem[20005688]= 0x78

After execution: R1= 0x20005698, R3= 0x00000078

2):

**LD8**    *R3, R1, 0x3f0*

Before execution: R1= 0x20005699, R3= 0x00000000

Mem[20005689]= 0x87

After execution: R1= 0x20005698, R3= 0xffffffff87



## LD8U

Op: 001000	Ra	Rb	Ext: 001110	S10
------------	----	----	-------------	-----

**Syntax:**

**LD8U**      *Ra, Rb, S10*

**Operation:**

$Ra = \text{ZeroExtend}(\text{Mem}[Rb + \text{SignExtend}(S10)])$

**Description:**

Load a byte from the effective address formed by adding register Rb to the 10-bit signed immediate S10. The byte is zero-extended into the register Ra.

**Example:**

1):

**LD8U**   *R3, R1, 0x3f0*

Before execution: R1= 0x20005698, R3= 0xffffffff

Mem[20005688]= 0x78

After execution:    R1= 0x20005698, R3= 0x00000078

2):

**LD8U**   *R3, R1, 0x3f0*

Before execution: R1= 0x20005699, R3= 0x00000000

Mem[20005689]= 0x87

After execution:    R1= 0x20005699, R3= 0x00000087

## LD16

Op: 001000	Ra	Rb	Ext: 000010	S10
------------	----	----	-------------	-----

**Syntax:**

**LD16**      *Ra, Rb, S10*

**Operation:**

$Ra = \text{SignExtend}(\text{Mem}[Rb + \text{SignExtend}(S10 \ll 1)])$

**Description:**

Load a halfword from the effective address formed by adding register Rb to the 10-bit signed immediate S10 after shifting it left by 1. The halfword is sign-extended into the destination register Ra.

**Example:**

1):

**LD16**   *R3, R1, 0x8*

Before execution: R1= 0x2000567a, R3= 0xffffffff

Mem[2000568a]= 0xabcd

After execution: R1= 0x2000567a, R3= 0xffffabcd

2):

**LD16**   *R3, R1, 0x8*

Before execution: R1= 0x2000567c, R3= 0xffffffff

Mem[2000568c]= 0x1234

After execution: R1= 0x2000567c, R3= 0x00001234

## LD16U



### Syntax:

**LD16U**     *Ra, Rb, S10*

### Operation:

$Ra = \text{ZeroExtend}(\text{Mem}[Rb + \text{SignExtend}(S10 \ll 1)])$

### Description:

Load a halfword from the effective address formed by adding register Rb to the 10-bit signed immediate S10 after shifting it left by 1. The halfword is zero-extended into the destination register Ra.

### Note:

The LSB of immediate S16 must be 1, else the result is unpredictable.

### Example:

1):

**LD16U**     *R3, R1, 0xffe0*

Before execution: R1= 0x200056aa, R3= 0xffffffff

Mem[2000568a]= 0xabcd

After execution: R1= 0x200056aa, R3= 0x0000abcd

2):

**LD16U**     *R3, R1, 0xffe0*

Before execution: R1= 0x200056ac, R2= 0x00000010, R3= 0xffffffff

Mem[2000568c]= 0x1234

After execution: R1= 0x200056ac, R2= 0x00000010, R3= 0x00001234

## LD32

Op: 001000	Ra	Rb	Ext: 000000	S10
------------	----	----	-------------	-----

**Syntax:**

**LD32**      *Ra, Rb, S10*

**Operation:**

$Ra = \text{Mem}[Rb + \text{SignExtend}(S10 \ll 2)]$

**Description:**

Load a word from the effective address formed by adding register Rb to the 10-bit signed immediate S10 after shifting it by 2. Put the word into the destination register Ra.

**Example:**

**LD32**   *R3, R1, 0x04*

Before execution: R1= 0x20005678, R3= 0xffffffff

Mem[20005688]= 0x7887abcd

After execution:    R1= 0x20005678, R3= 0x7887abcd

## SR8

Op: 001010	Rc	Rb	Ext: 000111	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**SR8**      *Ra*, *Rb*, *Rc*

**Operation:**

Mem [*Rb* + *Rc*] = *Ra*

**Description:**

Store a byte from register *Ra* to the effective address formed by adding registers *Rb* and *Rc*.

**Example:**

**SR8**    *R3*, *R1*, *R2*

Before execution: *R1*= 0x20005678, *R2*= 0x00000010, *R3*= 0x1234abcd  
Mem[20005688]= 0x78

After execution:    *R1*= 0x20005678, *R2*= 0x00000010, *R3*= 0xffffffff  
Mem[20005688]= 0xcd

## SR16

Op: 001010	Rc	Rb	Ext: 000011	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**SR16**     *Ra*, *Rb*, *Rc*

**Operation:**

Mem [*Rb* + *Rc*] = *Ra*

**Description:**

Store a halfword from register *Ra* to the effective address formed by adding registers *Rb* and *Rc*.

**Example:**

**SR16**   *R3*, *R1*, *R2*

Before execution: *R1*= 0x20005678, *R2*= 0x00000010, *R3*= 0x1234abcd  
Mem[20005688]= 0x0000

After execution:    *R1*= 0x20005678, *R2*= 0x00000010, *R3*= 0x1234abcd  
Mem[20005688]= 0xabcd

## SR32

**Syntax:**

**SR32**     *Ra*, *Rb*, *Rc*

**Operation:**

Mem [*Rb* + *Rc*] = *Ra*

**Description:**

Store a word from register *Ra* to the effective address formed by adding registers *Rb* and *Rc*.

**Example:**

**SR32**   *R3*, *R1*, *R2*

Before execution: *R1*= 0x20005678, *R2*= 0x00000010, *R3*= 0x1234abcd  
Mem[20005688]= 0x0000ffff

After execution:    *R1*= 0x20005678, *R2*= 0x00000010, *R3*= 0x1234abcd  
Mem[20005688]= 0x1234abcd

## SX16

Op: 001010	Rc	Rb	Ext: 010011	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**SX16**      *Ra*, *Rb*, *Rc*

**Operation:**

Mem [*Rb* + (*Rc* << 1)] = *Ra*

**Description:**

Store a halfword from register *Ra* to the effective address formed by adding registers *Rb* and the value of *Rc* shifting by 1.

**Example:**

**SX16**   *R3*, *R1*, *R2*

Before execution: *R1*= 0x20005678, *R2*= 0x00000008, *R3*= 0x1234abcd  
Mem[20005688]= 0x0000

After execution:    *R1*= 0x20005678, *R2*= 0x00000008, *R3*= 0x1234abcd  
Mem[20005688]= 0xabcd



## SX32

Op: 001010	Rc	Rb	Ext: 010001	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**SX32**      *Ra*, *Rb*, *Rc*

**Operation:**

Mem [*Rb* + (*Rc* << 2)] = *Ra*

**Description:**

Store a word from register *Ra* to the effective address formed by adding registers *Rb* and the value of *Rc* shifting by 2.

**Example:**

**SX32**   *R3*, *R1*, *R2*

Before execution: *R1*= 0x20005678, *R2*= 0x00000004, *R3*= 0x1234abcd  
Mem[20005688]= 0x0000ffff

After execution:    *R1*= 0x20005678, *R2*= 0x00000004, *R3*= 0x1234abcd  
Mem[20005688]= 0x1234abcd

## SD8

**Syntax:**

*SD8                      Ra, Rb, S10*

**Operation:**

Mem [Rb + SignExtend (S10)] = Ra

**Description:**

Store a byte from register Ra to the effective address formed by adding register Rb to the 10 bits signed immediate S10.

**Example:**

*SD8    R3, R1, 0x3f0*

Before execution: R1= 0x20005698, R3= 0x1234abcd  
Mem[20005688]= 0x78

After execution:    R1= 0x20005698, R3= 0x1234abcd  
Mem[20005688]= 0xcd

## SD16

**Syntax:**

**SD16**      *Ra, Rb, S10*

**Operation:**

Mem [Rb + SignExtend (S10 << 1)] = Ra

**Description:**

Store a halfword from register Ra to the effective address formed by adding register Rb to the 10 bits signed immediate S10 after shifting it left by 1.

**Example:**

**SD16**   R3, R1, 0x3e0

Before execution: R1= 0x200056a8, R3= 0x1234abcd  
                    Mem[20005688]= 0x0000  
After execution:  R1= 0x200056a8, R3= 0x1234abcd  
                    Mem[20005688]= 0xabcd

## SD32

Op: 001000	Ra	Rb	Ext: 000001	S10
------------	----	----	-------------	-----

**Syntax:**

**SD32**      *Ra*, *Rb*, *S10*

**Operation:**

Mem [*Rb* + SignExtend (*S10* << 2)] = *Ra*

**Description:**

Store a word from register *Ra* to the effective address formed by adding register *Rb* to the 10 bits signed immediate *S10* after shifting it left by 2.

**Example:**

**SD32**   *R3*, *R1*, 0x3e0

Before execution: *R1*= 0x200056a8, *R3*= 0x1234abcd  
                    Mem[20005688]= 0x00000000  
After execution:   *R1*= 0x200056a8, *R3*= 0x1234abcd  
                    Mem[20005688]= 0x1234abcd

### 3.3.5 Miscellaneous Instructions

This part includes conditional move instructions **MVZ** and **MVNZ**, count sign bits instruction **CSB**, byte reverse instruction **BREV** and swap the register contents with memory contents instruction **SWAP**.

Conditional move instruction **MVZ** copies the source register to the destination register if the value of another source operand is zero, and **MVNZ** copies the source register to the destination register if the value of another source operand is not zero. They are useful to eliminate some branch instructions. For example, the following instruction sequence evaluate the minimum of two registers:

```
SLT   Rt, Ra, Rb           ! Rt = Ra < Rb
MVZ   Ra, Rt, Rb           ! Ra = Ra < Rb? Ra : Rb
```

As the same reason, to evaluate the maximum of two registers, use:

```
SLT   Rt, Ra, Rb           ! Rt = Ra < Rb
MVNZ  Ra, Rt, Rb           ! Ra = Rb > Ra? Rb : Ra
```

## MVZ

Op: 000010	Rc	Rb	Ext: 001001	Don't care	Ra
------------	----	----	-------------	------------	----

**Syntax:**

**MVZ**      *Ra*, *Rb*, *Rc*

**Operation:**

```
If (Rb == 0) {  
    Ra = Rc  
}  
else {  
    Ra = Ra  
}
```

**Description:**

Copy the content of register Rc to Ra if Rb is zero, otherwise leave Ra unchanged.

**Example:**

1):  
    **MVZ**    R3, R2, R1

Before execution: R1= 0x2000567a, R2= 0x00000010, R3= 0xffffffff

After execution:  R1= 0x2000567a, R2= 0x00000010, R3= 0xffffffff

2):  
    **MVZ**    R3, R2, R1

Before execution: R1= 0x2000567c, R2= 0x00000000, R3= 0xffffffff

After execution:  R1= 0x2000567c, R2= 0x00000000, R3= 0x2000567c

## MVNZ

Op: 000010	Rc	Rb	Ext: 001101	Don't care	Ra
------------	----	----	-------------	------------	----

### Syntax:

**MVNZ**      *Ra*, *Rb*, *Rc*

### Operation:

```

If (Rb != 0) {
    Ra = Rc
}
else {
    Ra = Ra
}

```

### Description:

Copy the content of register Rc to Ra if Rb is not zero, otherwise leave Ra unchanged.

### Example:

1):

**MVNZ**    R3, R2, R1

Before execution: R1= 0x2000567a, R2= 0x00000010, R3= 0xffffffff

After execution:    R1= 0x2000567a, R2= 0x00000010, R3= 0x2000567a

2):

**MVNZ**    R3, R2, R1

Before execution: R1= 0x2000567c, R2= 0x00000000, R3= 0xffffffff

After execution:    R1= 0x2000567c, R2= 0x00000000, R3= 0xffffffff

## CSB

Op: 000000	Ra	Rb	Ext: 010100	Don't care
------------	----	----	-------------	------------

**Syntax:**

CSB      *Ra*, *Rb*

**Operation:**

```
result=0; sign=Rb[31];
for (i = 30; i != 0; i--)
{if (Rb[i] !=sign) break; else result++;}
Ra = result;
```

**Description:**

Count the number of sign bits in Rb then minus 1, put the result (0 ~ 31) into Ra.

**Example:**

1):  
CSB      R3, R1

Before execution: R1= 0x2000567a, R3= 0x00000000

After execution:    R1= 0x2000567a, R3= 0x00000001

2):  
CSB      R3, R0

Before execution: R3= 0x00000000

After execution:    R3= 0x0000001f

3):  
CSB      R3, R1

Before execution: R1= 0x80000000, R3= 0xffffffff

After execution:    R1= 0x80000000, R3= 0x00000000



## BREV

Op: 000000	Ra	Rb	Ext: 011000	Don't care
------------	----	----	-------------	------------

### Syntax:

**BREV**      *Ra*, *Rb*

### Operation:

```
tmp0 = Rb[7:0];
tmp1 = Rb[15:8];
tmp2 = Rb[23:16];
tmp3 = Rb[31:24];
Ra[7:0] = tmp3;
Ra[15:8] = tmp2;
Ra[23:16] = tmp1;
Ra[31:24] = tmp0;
```

### Description:

Reverse the 4 bytes of Rb, put the result into Ra.

### Example:

1):

**BREV**    *R3*, *R1*

Before execution: R1= 0x04030201, R3= 0x00000000

After execution:    R1= 0x04030201, R3= 0x01020304

2):

**BREV**    *R1*, *R1*

Before execution: R1= 0x0a0b0c0d

After execution:    R1= 0xd0c0b0a

## SWAP

Op: 001000	Ra	Rb	Ext: 001111	S10
------------	----	----	-------------	-----

**Syntax:**

**SWAP**      *Ra, Rb, S10*

**Operation:**

```
tmp = ZeroExtend(Mem[Rb + SignExtend(S10 << 2)])  
Mem[Rb + SignExtend(S10 << 2)] = Ra  
Ra = tmp
```

**Description:**

Swap the 32-bits contents of register Ra with the 32-bits contents loading from the effective address formed by adding register Rb to the 10 bits signed immediate S10 after shifting it left by 2.

**Example:**

**SWAP**    R3, R1, 1

Before execution: R1= 0x00001000, R3= 1  
                    Mem[00001004]= 0

After execution:  R1= 0x00001000, R3= 0  
                    Mem[00001004]= 1

### 3.3.6 System Control Instructions

This class includes 12 instructions: **CLD**, **CST**, **RCR**, **WCR**, **TRAP**, **SBRK**, **RTE**, **SLEEP**, **ITLB**, **DTLB**, **ICACHE** and **DCACHE**. Some of them are privilege instructions.

**CLD** and **CST** are used to exchange data between module control register and general register.

**RCR** and **WCR** are used to move data between GPR (General Purpose Register) and CRs (Control Registers include SR, ESR, DSR, EPC, DPC)

**TRAP** is a software interrupt instruction.

**SBRK** is a software debug interrupt instruction.

**RTE** are used to return from exception handlers.

**SLEEP** is used to put machine into sleep, standby or pause mode for power saving.

**ITLB**, **DTLB**, **ICACHE** and **DCACHE**: these instructions are used for special operations applied on Arca embedded RAM.

Instructions **RCR** and **WCR** read and write the control registers of Arca, in their instruction encoding the filed **CR** represents the index number of these control registers that lists in.

Control Register	Index Number
SR	0
ESR	1
EPC	2
DSR	5
DPC	6

## CLD

Op: 011001	Ra	00ID	Ext: 001000	S10
------------	----	------	-------------	-----

**Syntax:**

`CLD        Ra, ID, S10`

**Operation:**

Ra = Module\_Control\_Register (S10)

**Description:**

Put the content of the module control register into general register Ra. The module is specified by ID and the control register number is specified by S10, which must be in the range of 0 ~ 511.

**Example:**

`CLD R1, 0, 0                      ! R1 = MCR`

## CST

Op: 011001	Ra	00ID	Ext: 001001	S10
------------	----	------	-------------	-----

**Syntax:**

`CST        Ra, ID, S10`

**Operation:**

Module\_Control\_Register = Ra

**Description:**

Put the content of general register Ra into the module control register. The module is specified by ID and the control register number is specified by S10, which must be in the range of 0 ~ 511.

**Example:**

`CST R1, 0, 0                      ! MCR = R1`

## RCR

Op: 011001	Ra	Don't care	Ext: 000100	Don't care	CR
------------	----	------------	-------------	------------	----

**Syntax:**

`RCR       Ra, CR`

**Operation:**

`Ra = CR`

**Description:**

Put the content of CPU control register CR into general register Ra. This is a privilege instruction.

**Example:**

1):

`RCR   R1, 0`

Before execution: R1= 0x2000567a, SR= 0x00000008

After execution:  R1= 0x00000008, SR= 0x00000008

2):

`RCR   R2, 1`

Before execution: R2= 0x0000567a, ESR= 0x0000000f

After execution:  R2= 0x0000000f, ESR= 0x0000000f

## WCR

Op: 011001	Ra	Don't care	Ext: 000101	Don't care	CR
------------	----	------------	-------------	------------	----

**Syntax:**

**WCR**      *Ra*, *CR*

**Operation:**

CR = Ra

**Description:**

Put the content of general register Ra into CPU control register CR. This is a privilege instruction.

**Example:**

1):

**WCR**    R1, 0

Before execution: R1= 0x00000003, SR= 0x00000008

After execution: R1= 0x00000003, SR= 0x0000000b (write to SR.MD is ignored)

2):

**WCR**    R2, 2

Before execution: R2= 0x80000000, EPC= 0x00000000

After execution: R2= 0x80000000, EPC= 0x80000000

## TRAP

Op: 011001	Ra	Don't care	Ext: 000000	S10
------------	----	------------	-------------	-----

**Syntax:**

**TRAP**      *Ra, S10*

**Operation:**

Ra = S10  
ESR = SR  
EPC = PC + 4  
SR.IE = 0  
SR.DS = 0  
SR.SM = 1  
vector\_addr = vector\_table\_base + H'14  
PC = Mem[vector\_addr]

**Description:**

TRAP instruction causes a software trap, CPU puts S10 immediate operand into Ra, which can be the entry parameter of trap subroutine, then transfers the control to trap handler whose address is stored in vector table. vector\_table\_base is formed according to the rule described in 2.4.4. PC is the address of this TRAP.



## SBRK

Op: 011001	Don't care	Ext: 010011	Don't care
------------	------------	-------------	------------

### Syntax:

**SBRK**

### Operation:

```
if (SR.DE == 1)
{
    DSR = SR
    DPC = PC + 4
    SR.DE = 0
    SR.DS = 1
    vector_addr = vector_table_base + H'08
}
else
{
    ESR = SR
    EPC = PC
    SR.DS = 0
    vector_addr = vector_table_base + H'04
}
SR.IE = 0
SR.SM = 1
PC = Mem[vector_addr]
```

### Description:

When SR.DE = 1, SBRK instruction causes a debug exception. When SR.DE = 0, SBRK is take as an illegal instruction which causes an illegal instruction exception. vector\_table\_base is formed according to the rule described in 2.4.4. PC is the address of this SBRK.

## RTE

Op: 011001	Don't care	Ext: 000001	Don't care
------------	------------	-------------	------------

### Syntax:

RTE

### Operation:

```

if (SR.DS == 1)
{
    SR = DSR
    PC = DPC
}
else
{
    SR = ESR
    PC = EPC
}

```

### Description:

Return from Exception Routine, i.e., restore Status Register (SR) and jump back to the break point to continue execution of previous program flow. This is a privilege instruction. This is the only way to change CPU from supervisor mode to user mode.

## SLEEP

Op: 011001	Don't care	Ext: 000011	Don't care
------------	------------	-------------	------------

**Syntax:**

**SLEEP**

**Operation:**

Set CPU to power down mode

**Description:**

SLEEP is used to put machine into sleep, standby or pause mode for power saving. This is a privilege instruction.

If the machine is put into sleep or standby mode, it is waken up by an interrupt and acknowledge this interrupt. If the machine is put into pause mode, after the pause time expires the instruction following SLEEP is to be executed. In pause mode, interrupt is ignored. Some implementation may not support SLEEP pause mode.

## ITLB

Op: 000000	0CMD	Rb	Ext: 001101	Don't care
------------	------	----	-------------	------------

### Syntax:

**ITLB**      *CMD, Rb*

### Operation:

Send CMD and virtual address Rb to ITLB

### Description:

Cause special operation on ITLB, the operation type is specified by CMD and the operation object is specified by Rb. This is a privilege instruction.

## DTLB

Op: 000000	0CMD	Rb	Ext: 001001	Don't care
------------	------	----	-------------	------------

**Syntax:**

**DTLB**      *CMD, Rb*

**Operation:**

Send CMD and virtual address Rb to DTLB

**Description:**

Cause special operation on DTLB, the operation type is specified by CMD and the operation object is specified by Rb. This is a privilege instruction.

## ICACHE

**Syntax:**

**ICACHE** *CMD, Rb, S10*

**Operation:**

Send CMD and virtual address Rb + (S10 << 2) to ICACHE

**Description:**

Cause special operation on ICACHE, the operation type is specified by CMD and the operation object is specified by (Rb + (S10 << 2)). This is a privilege instruction for some of CMD.

## DCACHE

**Syntax:**

**DCACHE**    *CMD, Rb, S10*

**Operation:**

Send CMD and virtual address Rb + (S10 << 2) to DCACHE

**Description:**

Cause special operation on DCACHE, the operation type is specified by CMD and the operation object is specified by (Rb + (S10 << 2)). This is a privilege instruction for same of the CMD.

### 3.4 Instruction Encoding

This section gives the instruction encoding. Table 3-3 presents the encoding map of major opcode of Arca instruction set.

**Table 3-3 Instructions Encoding Map**

Op	000	001	010	011	100	101	110	111
000	ADDI, SLLI, SLRI, SARI, JA, BREV, CSB, ITLB, DTLB, ICACHE, DCACHE		ADD, SUB, SLL, SLR, SAR, MVZ, MVNZ, AND, OR, XOR, ANDN, SCC, MULU		BCCI	J	BCC	
001	Load/Store Swap (R+Imm)	LHI	Load/Store (R+R; R+(R<<BN))		ANDI			
010					ORI			
011	XORI	RTE, SLEEP, TRAP, SBRK, CLD, CST, RCR, WCR						
100								
101								
110								
111								



Table 3-4 lists the encoding for each instruction.

**Table 3-4 Instructions Code**

Classify	Instruction	OPcode	EXTcode
Load and store	LD8	001000	000110
	LD8U	001000	001110
	LD16	001000	000010
	LD16U	001000	001010
	LD32	001000	000000
	SD8	001000	000111
	SD16	001000	000011
	SD32	001000	000001
	LR8	001010	000110
	LR8U	001010	001110
	LR16	001010	000010
	LR16U	001010	001010
	LR32	001010	000000
	LX16	001010	010010
	LX16U	001010	011010
	LX32	001010	010000
	SR8	001010	000111
	SR16	001010	000011
	SR32	001010	000001
	SX16	001010	010011
	SX32	001010	010001
Compare	SEQ	000010	010110
	SNE	000010	010010
	SLT	000010	010100
	SLTU	000010	011100
	SGE	000010	010000
	SGEU	000010	011000
Arithmetic	ADDI	000000	000000
	ADD	000010	000000
	SUB	000010	100000
	MULU	000010	010001
	MUL	000010	110001
Bitwise	ANDI	001100	
	ORI	010100	
	XORI	011000	
	AND	000010	000001
	OR	000010	000010
	XOR	000010	000011
	ANDN	000010	100001
Shift	SLLI	000000	000100
	SLRI	000000	001000
	SARI	000000	001100
	SLL	000010	000100
	SLR	000010	001000
	SAR	000010	001100

**Table 3-5 Instructions Code (continue)**

Classify	Instruction	OPcode	EXTcode
Jump	J	000101	
	JA	000000	000010
Branch	BEQ	000110	000110
	BNE	000110	000010
	BLT	000110	000100
	BLTU	000110	001100
	BGE	000110	000000
	BGEU	000110	001000
	BEQI	000100	000110
	BNEI	000100	000010
	BEQUI	000100	001110
	BNEUI	000100	001010
	BLTI	000100	000100
	BLTUI	000100	001100
	BGEI	000100	000000
	BGEUI	000100	001000
Constant Load	LHI	001001	
Miscellaneous	MVZ	000010	001001
	MVNZ	000010	001101
	CSB	000000	010100
	BREV	000000	011000
	SWAP	001000	001111
System control	CLD	011001	001000
	CST	011001	001001
	RCR	011001	000100
	WCR	011001	000101
	TRAP	011001	000000
	SBRK	011001	010011
	RTE	011001	000001
	SLEEP	011001	000011
	ITLB	000000	001101
	DTLB	000000	001001
	ICACHE	000000	000101
	DCACHE	000000	000001



## 4 Application Binary Interface

ABI(Application Binary Interface) is a set of rules to specify the conventions to use system resources like registers and stack. Object codes developed conforming a same ABI can interoperate with each other. This section presents a generic ABI convention developed by ARCA Technology Corporation. Arca Linux is developed with this ABI.

### 4.1 Register Usage Conventions mount

Arca architecture provides 32 general purpose registers (GPR) for application programs, R0 ~ R31, each 32 bits wide. Their usage is given in Table 4-1. In the talbe:

- **Caller save register:** A register is caller save if its value is not guaranteed to be preserved across function calls. Such register is also called scratch since the caller will have to save and restore the register around function calls.
- **Callee save register:** A register is callee save if its value is guaranteed to be preserved across function calls.
- R19 is the reserve scratch register. In assembler, it is used to split on branch instruction to more when the target address is too long. In the compiler, It used to a scratch register in the basic blocks.

**Table 4-1 Registers Usage**

Register Name	Usage
r0	Read as ZERO, write is ignored
r1	Stack pointer, SP, Callee save
r2	Return value, caller save
r2 ~ r7	Parameter passing, caller save
r8 ~ r15	Callee save
r16	Frame pointer, FP, callee save
r17	PIC register, callee save
r18	Linkage register, caller save
r19	Reserved for assembler handling long branch, used for linux system call (trap insn)
r20	Static chain register, caller save, syscall number for linux system calls
r20 ~ r21	Caller save
r22 ~ r27	Callee save
r28 ~ r31	Caller save (or reserved for linux )

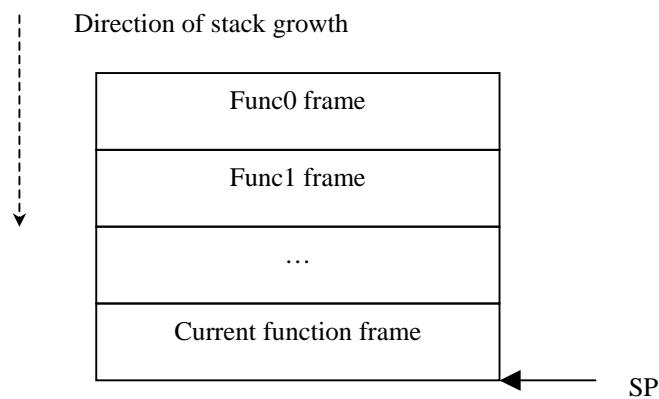
## 4.2 Stack Layout

Stack has the follow features:

- **Direction:** the stack grows for high address to low address.
- **Stack Top:** the top of stack is always reference by **SP** (r1) register and is the address of the last used word on the stack. That is to say,  $SP + 0$  is a valid address.
- **Alignment:** the stack pointer must be aligned to a 4 byte boundary on entry to a function.

Compilers use the stack by pushing and popping frame to represent the local data of a function, usually referred to as a frame. Each called function creates and deletes its own frame.

The topmost frame is the frame of the currently executing function. The stack growth is shown in Figure 4-1.

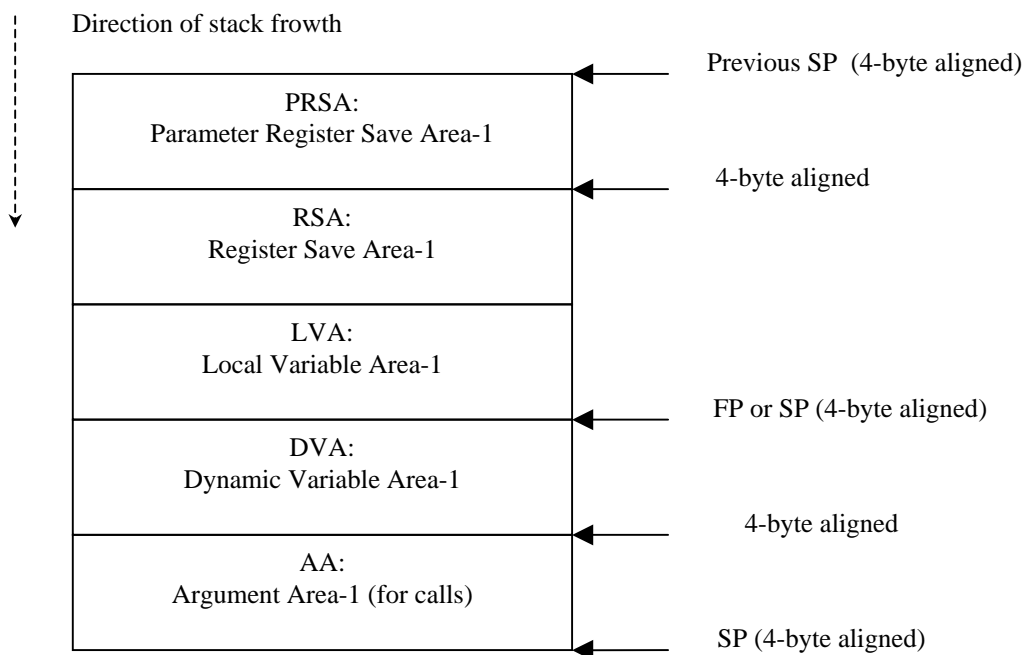


**Figure 4-1 Overview of Stack**

### 4.3 Frame Layout

The frame layout is shown in Figure 4-2. In the frame:

- **PRSA (Parameter Register Save Area):** This area is needed only when the called routine needs a memory copy of its parameters, which are otherwise passed in registers, such as variable length arguments.
- **RSA (Register Save Area):** This area is used to save and restore the callee save registers for this function.
- **LVA (Local Variable Area):** This area is used for the local variable needing memory location and for any compiler temporary. The objects in this area are accessed by positive offsets from SP or FP.
- **DVA (Dynamic Variable Area):** This area is used for any objects that are allocated by extending the stack frame of current routine.
- **AA (Argument Area):** The area contains the remaining elements of the argument list after all the parameter registers have been used.



**Figure 4-2 Frame Layout**

An additional frame pointer register (FP) may be allocated for a frame that addresses the local variable area of the stack frame. An FP is only required when a frame has a dynamic variable area which is dynamically allocated since SP can no longer be used to address the local variable area.

## 4.4 Type Mapping

### 4.4.1 Scalar Types

The scalar types definition is as the following:

**Table 4-2 Mapping of ANSI C Data Types**

TYPES	BYTE ALIGNMENT	Size
Char	1	1 bytes
signed char		1 bytes
unsigned char		1 bytes
short int (signed) unsigned Short int	2	2 bytes 2 bytes
int (signed) unsigned int enum	4	4 bytes 4 bytes 4 bytes
long int (signed) unsigned long int	4	4 bytes 4 bytes
long long int (signed) unsigned long long int	8	8 bytes 8 bytes
Float	4	4 bytes
Double long double	8	8 bytes 8 bytes
Pointer	4	4 bytes

### 4.4.2 Aggregate Types

- **Array type:** Array will have the alignment of the components. The size of an array is always a multiple of the element alignment.
- **Structures and Unions:** Structures and Unions have the alignment of the most strictly aligned component, and to maintain the alignment of internal components, padding is inserted by the compiler. The contents of any padding is undefined.

The size of a structure is always a multiple of its alignment and this may require tail padding. The address of a structure or union is its lowest (smallest) address and the structure fields are allocated in declarative order from lowest address to highest address.

## 4.5 Bit-fields

Bit-fields are associated with an underlying integral type (char, short, int, long or long long). The associated type is the type used in the bit-field definition. Bit-fields obey the same size and alignment rules as other structure members, with the following additions:

- A bit-field never crosses a storage boundary whose alignment is same as the alignment of the underlying type of the bit-field.
- A bit-field shares a storage unit with the previous structure member if and only if the size of the type of the previous member is same as that of the bit-field and there is sufficient space within the storage unit.
- Bit-fields are allocated from right to left (least significant to most significant) on little-endian implementations and from left to right (most significant to least significant) on big-endian implementations.
- The effect of a zero-length bit-field is to force the alignment of the next bit-field to the alignment of the underlying type of the bit-field.



## 4.6 Return Values (Function Results)

Following table shows how function values that are scale type are returned from called functions.

**Table 4-3 Return Values**

Return Values Type	
char, signed char, unsigned char, short int (signed), unsigned short int, int (signed), unsigned int, enum long int (signed), unsigned long int	Return in general register R2
long long int (signed), unsigned long long int	Return in general registers R2, R3 Big endian: R2 (high part), R3 (low part) Little endian: R2 (low part), R3 (high part)
Float	Return in general register R2
double, long double	Return in general registers R2, R3 Big endian: R2 (high part), R3 (low part) Little endian: R2 (low part), R3 (high part)
Pointer	Return in general register R2

Function results of structure types are returned by address. The caller function passes the address of the result destination as an implicit extra parameter in register R2. The called function stores the result in this area and return the address of the area as its result in register R2.

## 4.7 Argument Passing and Mapping

- **Passing Argument**

The Arca uses six general registers (r2-r7) to pass the first six words of arguments from the caller to the called routine. If additional argument space is required, the caller is responsible for allocating this space on the stack.

- **Scalar Arguments**

Arguments are passed using register r2 through r7, with no more than one argument assigned per register. Argument values that are smaller than a 32-bit register occupy a full register. Small signed arguments are sign extended; small unsigned arguments are zero extended.

Arguments larger than a register must be assigned to multiple argument registers as long as there are argument registers available.

Once all the argument register are used, or if there are not enough register left to hold a large argument, the argument was put on the stack. (first word in `sp + 0`, second word in `sp + 4`, and so on)

- **Structure Argument**

Structure arguments that are smaller than 32 bits have their value right justified within the argument register. The unused upper bits within the register are undefined.

Structure arguments that are larger than 32 bits are packed into consecutive registers. When the size of structure parameter is not a multiple of 4 bytes, the value of the padding is undefined. The padding depends on target endianness: for little-endian targets the element is padded at the most significant end, for big-endian targets the element is padded at the least significant end.

If there are not enough register left, the rest part of the structure will be put on the stack (first word in `sp + 0`, second word in `sp + 4`, and so on).

- **Handling Of Variable-arguments Function By The Callee**

When the call is to a function with a variable number of arguments, the caller will pass the arguments in accordance with the rules outlined above. The called routine will allocate space and copy the registers R2-R7 to its own stack space (PRSA). Because the function cannot tell in advance how many of the machine registers may be in use, it must save all the potential parameter registers to the stack.

The implementation of the variable argument manipulation macros will use the memory copies of the parameters. The **va\_arg** macro will address the parameters as an array indexed by the implicit counter to **va\_arg**. The **va\_start** macro is implemented by initializing the variable argument pointer with the address of the argument in the argument list that is the first variant argument. This is the address of the first parameter available through **va\_arg**. Consequently, **va\_arg** is implemented as returning the value at this pointer followed by an increase to address the next parameter in the variable argument list.

**va\_arg** must take into account padding inserted when the parameter is not an exact multiple of 4 bytes in length. All arguments smaller than 4 bytes are padded at the most significant end, and arguments larger than 4 bytes are padded in the last argument element.

**va\_end** serves no purpose except to make the variable argument unusable as a legal pointer



## 5 ASSEMBLER MACROS AND OPERATORS

To simplify assembly coding, this section defines some assembly language macro instructions and operators. Macro instructions are synthesized version from machine instructions. Operators are used to extract a specific number from a constant operand.

operator:

@h21	means: Get the high 21 bits of the 32-bit symbol or the integer value. Instruction: lhi Example: lhi R2, <a href="#">sym1@h21</a> (means: lhi R2, (sym1 >> 11)) lhi R2, <a href="#">0xFFFFF800@h21</a> (means: lhi R2, 0xFFFFF)
@l11	means: Get the low 11 bits of the 32-bit symbol or the integer value. Instructions: ori, andi, xori Example: ori R2, <a href="#">sym1@l11</a> (means: ori R2, (sym1 & 0x7ff)) ori R2, <a href="#">0xFFFF9FFF@l11</a> (means: ori R2, 0x1FF)
@eh21	means: Get the high 21 bits of the 32-bit symbol or the integer value by added the 10 <sup>th</sup> bit. Instruction: lhi Example: lhi R2, <a href="#">sym1@eh21</a> (means: lhi R2, ((sym1 + sym1 & 0x400) >> 11)) lhi R2, <a href="#">0xFFFFFC00@l11</a> (means: lhi R2, 0x0)
@o11	means: Get the low 11 bits of the 32-bit symbol or the integer value and then logic right shifted by 2. Instructions: ld32, sd32, swap Example: ld32 R2, [R3, <a href="#">sym1@o11</a> ] (means: ld32 R2, [R3, (sym1 & 0x7FF) >> 2]) ld32 R2, [R3, <a href="#">0xFFFFFC00@o11</a> ] (means: ld32 R2, [R3, 0x100])
@ho11	means: Get the low 11 bits of the 32-bit symbol or the integer value and then arith right shifted by 1. Instructions: ld16, sd16 Example: ld16 R2, [R3, <a href="#">sym1@ho11</a> ] (means: ld16 R2, [R3, ((int)sym1 << 21) >> 22]) ld16 R2, [R3, <a href="#">0xFFFFFC00@o11</a> ] (means: ld16 R2, [R3, -512])

macro instructions:

mov Ra, Rb	means: or	Ra, Rb, R0
movi Ra, imm	means: ori	Ra, Rb, imm
mova Ra, sym	means: lhi	Ra, <a href="#">sym@h21</a>
	ori	Ra, Ra, <a href="#">sym@l11</a>
movhi Ra, sym	means: lhi	Ra, <a href="#">sym@h21</a>
movlo Ra, sym	means: ori	Ra, <a href="#">sym@l11</a>
nop	means: or	R0, R0, R0

## List of Figures

FIGURE 2-1 DATA ORGANIZATION IN REGISTER .....	4
FIGURE 2-2 BIG-ENDIAN MEMORY SYSTEM .....	5
FIGURE 2-3 LITTLE-ENDIAN MEMORY SYSTEM .....	5
FIGURE 2-4 ARCA REGISTERS.....	7
FIGURE 3-1 INSTRUCTION FORMAT.....	12
FIGURE 4-1 OVERVIEW OF STACK .....	13
FIGURE 4-2 FRAME LAYOUT.....	14

## List of Tables

TABLE 1-1 ARCA FEATURES .....	1
TABLE 2-1 DATA TYPE AND OPERATION .....	4
TABLE 2-2 CONTROL REGISTERS INITIAL VALUE AFTER POWER ON RESET .....	8
TABLE 2-3 ARCA EXCEPTION PRIORITIES .....	10
TABLE 2-4 ARCA EXCEPTION VECTOR TABLE.....	11
TABLE 3-1 ARCA INSTRUCTIONS AND FORMAT .....	14
TABLE 3-2 LOAD AND STORE INSTRUCTIONS .....	41
TABLE 3-3 INSTRUCTIONS ENCODING MAP .....	82
TABLE 3-4 INSTRUCTIONS CODE .....	83
TABLE 3-5 INSTRUCTIONS CODE (CONTINUE).....	84
TABLE 4-1 REGISTERS USAGE .....	12
TABLE 4-2 MAPPING OF ANSI C DATA TYPES.....	15
TABLE 4-3 RETURN VALUES .....	17