AAP: An Altruistic Processor

A reference Harvard architecture for embedded compiler development

Simon Cook
Jeremy Bennett
Edward Jones
Application Note 13. Issue 2.1
Publication date December 2015

Legal Notice

Embecosm is the business name of Embecosm Limited, a private limited company registered in England and Wales. Registration number 6577021.

EMBECOSM®

Table of Contents

List of Figures

List of Tables

EMBECOSM®

Chapter 1. Introduction

AAP is a Harvard architecture specification designed for experimenting with various features in compiler back ends. In particular it has features that are common within small deeply embedded systems, such as a dearth of registers, word address code memory and pointers that will not fit in an integer.

It is also designed to be easy to use in demonstrations and education/training. This includes hardware and simulator implementation as well as the tool chain and library implementation.

The design is based on no processor in particular, although as an open hardware design, it is inspired by the OpenRISC and RISC-V projects. There are features drawn from a wide range of processors developed over the past 30 years. Indeed the branch-and-link operation goes back even further, to the IBM 360.

1.1. Revision History

Revision History
Revision 2.1 10 December 2015 Edward Jones
The Postincrement and Predecrement store instructions defined R_a rather than R_d as the operand to which the update was applied.

Postincrement, Predecement instructions now increment or decrement by the amount of the provided offset (Previously it was a fixed offset of one or two bytes for byte and word operations respectively).

BGTS, **BGTU**, **JGTS**, **JGTU**, **JGTSL**, **JGTUL**. These instructions have been replaced with branches with a 'Less Than or Equal To' condition. The new instructions are **BLES**, **BLEU**, **JLES**, **JLEU**, **JLESL**, **JLEUL**.
Revision 2.0 9 October 2015 Jeremy Bennett
Issue 2.0, which covers the entire ISA.
Revision 1.9 8 October 2015 Jeremy Bennett
Final draft before release 2.0. Adds some notation description and a chapter for the architecture description, which incorporates some of the old intro and the old chapter on **NOP** side effects.
Revision 1.8 8 October 2015 Jeremy Bennett
All 32-bit instructions described.
Revision 1.7 8 September 2015 Jeremy Bennett
First batch of 32-bit ALU instruction described. Various typos fixed.
Revision 1.6 8 September 2015 Jeremy Bennett
All 16-bit instructions described. Encoding of JAL corrected in the summary. R_b used to store the PC for all **BAL** and **JAL** instructions.
Revision 1.5 8 September 2015 Jeremy Bennett
16-bit ALU instruction details complete. Change opcode mnemonics for ALU instructions with constant arguments. Off-by-one encoding for immediate shift values described.
Revision 1.4 4 September 2015 Jeremy Bennett
Structure of detailed instruction descriptions refined. Most 16-bit ALU instructions now documented.
Revision 1.3 4 September 2015 Jeremy Bennett
All instruction formats now shown. All summaries in new format.
Revision 1.2 3 September 2015 Jeremy Bennett
First stage of improved formatting, using LibreOffice Impress to as the basis of the instruction format diagrams for 32-bit instructions (generating SVG and PNG). Clearer summary of instructions used for 32-bit ALU instructions.

Revision 1.1	18 July 2015	Jeremy Bennett

Start of revision process. Remove load/store double instructions. Use second opcode field of 32-bit load/store as extra constant field. Make all load/store offsets signed. Make **BAL** use R_b rather than R_a to keep constant field contiguous.

Revision 1.0	14 April 2015	Jeremy Bennett

Bump release number to 1.0 for issue.

Revision 0.9	14 April 2015	Jeremy Bennett

First public release outlining the architecture.

Revision N/A	11 April 2015	Jeremy Bennett

Correct encoding of 32-bit branches (4 more bits of offset). Correct **NOP** constant meanings. Matches server/simulator commit **b179463**.

Revision N/A	8 April 2015	Jeremy Bennett

Full summary of all 16-bit and 32-bit instructions.

Revision N/A	8 April 2015	Jeremy Bennett

Updated preface in preparation for revised architecture.

Revision N/A	6 April 2015	Simon Cook

Initial concept

Chapter 2. Architecture Description

Figure 2.1 shows the overall structure of AAP.

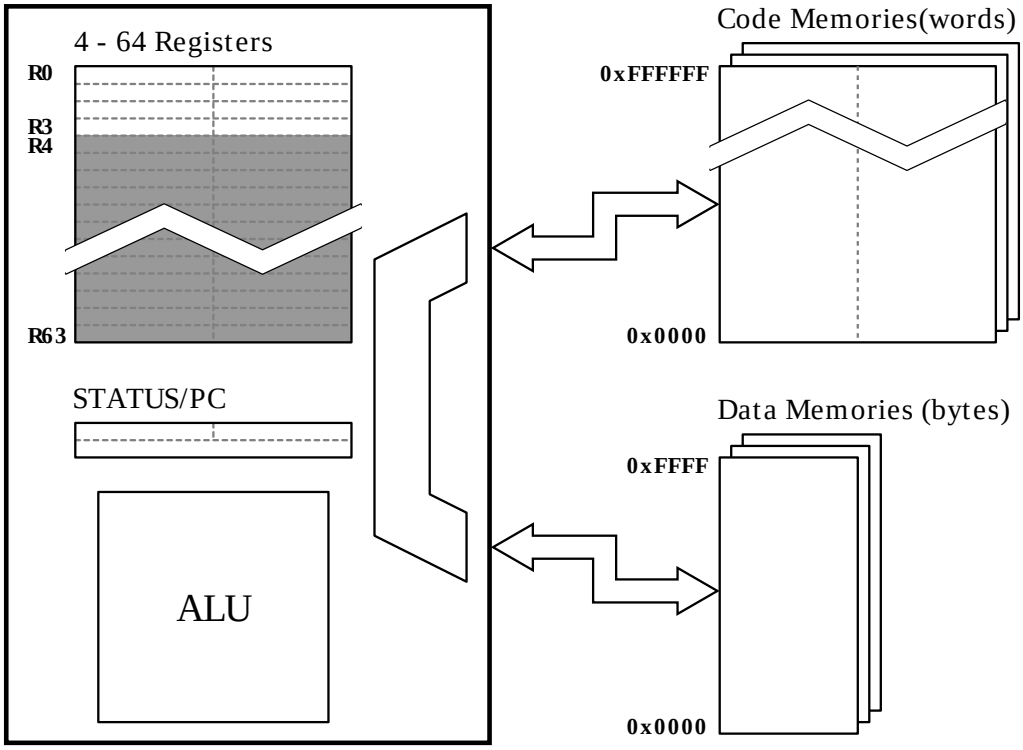


Figure 2.1. AAP architecture

2.1. Basic architectural features

These are the key features of the AAP design.

16-bit RISC architecture

The core design sticks to the RISC principles of 3-address register-to-register operation, a small number of operations and a simple to implement data path. The fundamental data type is the 16-bit integer.

Configurable number of registers

Although 32/64-bit RISC architectures typically have 16 or more general purpose registers, small deeply embedded processors often have far fewer. This represents a significant compiler implementation challenge. To allow exploration of this area, AAP can be configured with between 4 and 64 16-bit registers.

Harvard memory layout

The basic architecture provides a 64k byte addressed data memory and a separate 16M word instruction memory. By requiring more than 16-bits to address the instruction memory, the compiler writer can explore the challenge of pointers which are larger than the native integer type.

Deeply embedded systems often have very small memories, particularly for data, so the size of memories can be configured.

Many architectures also provide more than two address spaces, often for special purposes. For example a small EEPROM alongside Flash memory, or the *Special Purpose Register* block of OpenRISC. AAP can support additional address spaces, allowing support for multiple address spaces throughout the tool chain to be explored.

24-bit program counter with 8-bit status register

AAP requires a 24-bit program counter, which is held in a 32-bit register. The top bits of the program counter then form a status register. Jump instructions ignore these top 8 bits.

At present only one status bit is defined, a carry flag to allow multiple precision arithmetic.

16/32-bit instruction encoding

A frequent feature of many architectures is to provide a subset of the most commonly used parts of the Instruction Set Architecture (ISA) in a short encoding of 16-bits. Less common instructions are then encoded in 32-bits.

Optimizing to use these shorter instructions, is particularly important for compilers for embedded targets, where memory is at a premium. AAP provides such a 16-bit subset with a 32-bit encoding of the full ISA. However it follows the instruction chaining of RISC-V, so even longer instructions could be created in the future.

The fields within each 16-bit instruction are fixed. A 32-bit instruction pairs up those fields to increase the number of instructions.

3-address code

AAP has stuck rigidly to the RISC principle of 3-address instructions throughout. Almost all instructions come in two variants, one where the third argument is a register, and one where the third argument is a constant.

No flags for flow of control

There are no flag registers indicating the results of operations for use in conditional jumps. Instead the operation is encoded within the jump instruction itself.

There is an 8-bit status register as part of the program counter, which includes a carry flag. However this is not used for flow-of-control, but to enable multiple precision arithmetic.

Little endian

The architecture is little-endian—the least significant byte of a word or double word is at the lowest address.

The behavior for instruction memory is that one word is fetched, since it may be a 16-bit instruction. If a second word is needed, then its fields are paired with the first instructions to give larger values for each field. This is done in little-endian fashion, i.e. the field from the second instruction forms the most significant bits of the combined field.

No delay slots

Early RISC designs introduced the concept of a delay slot after branches. This avoided pipeline delays in branch processing. Implementations can now avoid such pipeline delay, so like most modern architectures, AAP does not have delay slots.

`NOP` with argument for simulator control

This idea is taken from OpenRISC. The `NOP` opcode includes fields to specify a register and a constant. These can be used in both hardware and simulation to trigger side-effects.

2.2. Event Handling

Events indirect through instructions in the first 256 (0x100) words of instruction memory. In general these should be 32-bit branch instructions, which means event handlers should reside in first or last 2^{21} words of instruction memory.

At present the following event vector locations (word addresses) are defined

`0x00` Power-on reset.

`0x02` Bus error

The event handling mechanism is still in development. In particular no location is yet defined for the return address to be used by the **`RTE`** instruction (see Section 3.7.1).

2.3. `NOP` Behavior

The **`NOP`** instruction takes an immediate argument which can be used to trigger certain behavior in a simulator.

- 0 : Breakpoint
- 1 : Do nothing
- 2 : Exit with return code in R_d
- 3 : Write char in R_d to standard output.
- 4 : Write char in R_d to standard error.
- All other values: do nothing, but future behavior not guaranteed.

Chapter 3. Instructions

3.1. Notation

In the instruction descriptions below, the following notation is used.

R_d	Destination register number "d" in the general registers.
R_a	First source register number "a" in the general registers.
R_b	Second source register number "b" in the general registers.
PC	The program counter
I	Unsigned immediate value
S	Signed immediate value
dmem[i]	Byte offset "i" in the data memory.
imem[i]	Word offset "i" in the code memory.
carry	The carry flag.
SignExt(x)	The value "x" (which may be one of the above) sign extended as necessary.

Individual bits in the encodings are used as follows.

`0` A zero bit.

`1` A one bit.

d_n Bit "n" of the destination register field.

a_n Bit "n" of the the first source register field.

b_n Bit "n" of the the second source register field.

i_n Bit "n" of the the unsigned constant field.

s_n Bit "n" of the the signed constant field.

3.1.1. Assembler Notation

The assembler generally follows standard GNU assembler conventions. Instructions take the following form:

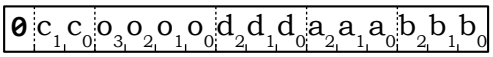
[label**:**] opcode [arguments]

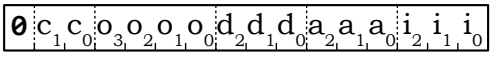
There may be up to 3 arguments, separated by commas. Registers are indicted by **`R`** followed by a number. Constants and constant expressions may be preceded by **`#`** for clarity, but this is not required. C style notation to indicate the base of constants, which defaults to decimal.

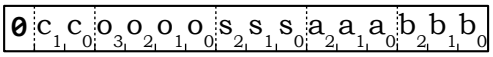
3.2. Instruction Format

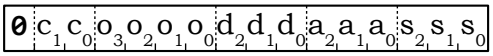
The 16-bit instruction formats are shown in Figure 3.1 and the 32-bit instruction formats in Figure 3.2.

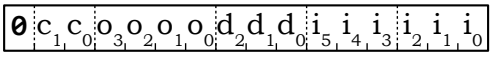
Format

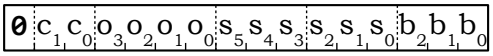
1 

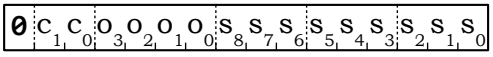
2 

3 

4 

5 

6 

7 

c_1c_0	Opcode class	$b_2...b_0$	Second source register
$o_3...o_0$	Opcode	$i_n...i_0$	Unsigned immediate
$d_2...d_0$	Destination register	$s_n...s_0$	Unsigned immediate
$a_2...a_0$	First source register		

Figure 3.1. AAP 16-bit instruction formats.

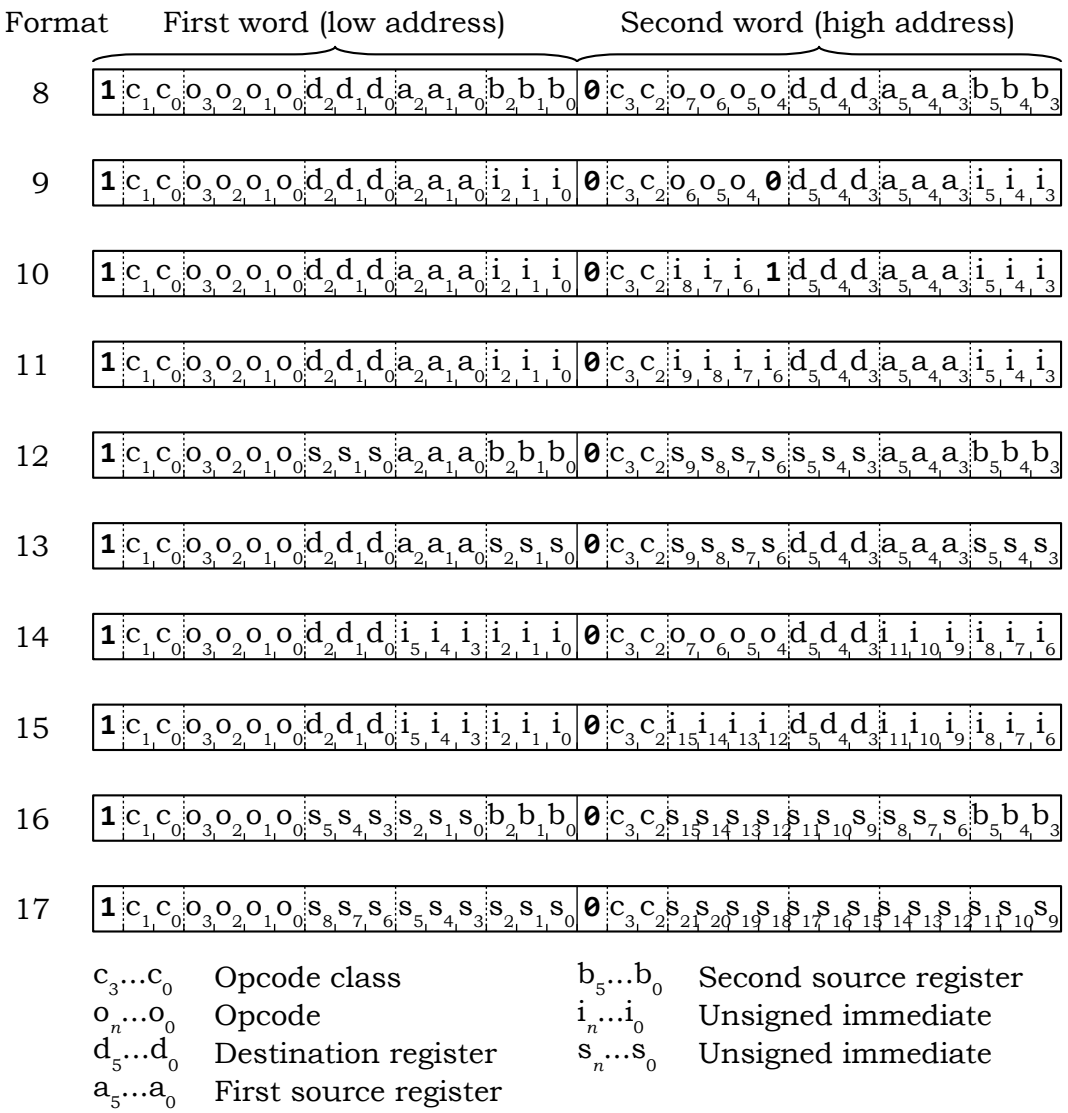


Figure 3.2. AAP 32-bit instruction formats.

Longer instruction formats are possible by setting the top bit of the second word to 1. By repeating this, instructions of arbitrary length are possible.

3.3. Summary of Instructions

3.3.1. 16-bit Instructions of AAP

- Table 3.1 lists all the 16-bit ALU instructions, which have class **00**;
- Table 3.2 lists all the 16-bit load/store instructions, which have class **01**;
- Table 3.3 lists all the 16-bit branch/jump instructions, which have class **10**; and
- Table 3.4 lists all the 16-bit miscellaneous instructions, which have class **11**.

Opcode	Format	Encoding	Description
NOP R_d,I	5	**0000000**dddiiiiii	No operation

Opcode		Format	Encoding	Description
ADD	R_d,R_a,R_b	1	0000001dddaaabbb	Unsigned add
SUB	R_d,R_a,R_b	1	0000010dddaaabbb	Unsigned subtract
AND	R_d,R_a,R_b	1	0000011dddaaabbb	Bitwise AND
OR	R_d,R_a,R_b	1	0000100dddaaabbb	Bitwise OR
XOR	R_d,R_a,R_b	1	0000101dddaaabbb	Bitwise exclusive OR
ASR	R_d,R_a,R_b	1	0000110dddaaabbb	Arithmetic shift right
LSL	R_d,R_a,R_b	1	0000111dddaaabbb	Logical shift left
LSR	R_d,R_a,R_b	1	0001000dddaaabbb	Logical shift right
MOV	R_d,R_a	1	0001001dddaaa000	Move register to register
ADDI	R_d,R_a,#I	2	0001010dddaaaiii	Unsigned add immediate
SUBI	R_d,R_a,#I	2	0001011dddaaaiii	Unsigned subtract immediate
ASRI	R_d,R_a,#I	2	0001100dddaaaiii	Arithmetic shift right immediate
LSLI	R_d,R_a,#I	2	0001101dddaaaiii	Logical shift left immediate
LSRI	R_d,R_a,#I	2	0001110dddaaaiii	Logical shift right immediate
MOVI	R_d,#I	5	0001111dddiiiiii	Move immediate to register

Table 3.1. 16-bit ALU instructions

Opcode		Format	Encoding	Description
LDB	R_d,(R_a,S)	4	0010000dddaaasss	Indexed load byte
LDW	R_d,(R_a,S)	4	0010100dddaaasss	Indexed load word
LDB	R_d,(R_a+,S)	4	0010001dddaaasss	Indexed load byte with postincrement
LDW	R_d,(R_a+,S)	4	0010101dddaaasss	Indexed load word with postincrement
LDB	R_d,(-R_a,S)	4	0010010dddaaasss	Indexed load byte with predecrement
LDW	R_d,(-R_a,S)	4	0010110dddaaasss	Indexed load word with predecrement
STB	(R_d,S),R_a	4	0011000dddaaasss	Indexed store byte
STW	(R_d,S),R_a	4	0011100dddaaasss	Indexed store word
STB	(R_d+,S),R_a	4	0011001dddaaasss	Indexed store byte with postincrement
STW	(R_d+,S),R_a	4	0011101dddaaasss	Indexed store word with postincrement
STB	(-R_d,S),R_a	4	0011010dddaaasss	Indexed store byte with predecrement
STW	(-R_d,S),R_a	4	0011110dddaaasss	Indexed store word with predecrement

Table 3.2. 16-bit load/store instructions

Opcode		Format	Encoding	Description
BRA	S	7	0100000sssssssss	Relative branch
BAL	S,R_b	6	0100001ssssssbbb	Relative branch and link
BEQ	S,R_a,R_b	3	0100010sssaaabbb	Relative branch if equal
BNE	S,R_a,R_b	3	0100011sssaaabbb	Relative branch if not equal
BLTS	S,R_a,R_b	3	0100100sssaaabbb	Relative branch if signed less than

Opcode	Format	Encoding	Description
BLES S,R_a,R_b	3	**0100101sssaaabbb**	Relative branch if signed less than or equal to
BLTU S,R_a,R_b	3	**0100110sssaaabbb**	Relative branch if unsigned less than
BLEU S,R_a,R_b	3	**0100111sssaaabbb**	Relative branch if unsigned less than or equal to
JMP R_d	1	**0101000ddd000000**	Absolute jump
JAL R_d,R_b	1	**0101001ddd000bbb**	Absolute jump and link
JEQ R_d,R_a,R_b	1	**0101010dddaaabbb**	Absolute jump if equal
JNE R_d,R_a,R_b	1	**0101011dddaaabbb**	Absolute jump if not equal
JLTS R_d,R_a,R_b	1	**0101100dddaaabbb**	Absolute jump if signed less than
JLES R_d,R_a,R_b	1	**0101101dddaaabbb**	Absolute jump if signed less than or equal to
JLTU R_d,R_a,R_b	1	**0101110dddaaabbb**	Absolute jump if unsigned less than
JLEU R_d,R_a,R_b	1	**0101111dddaaabbb**	Absolute jump if unsigned less than or equal to

Table 3.3. 16-bit branch/jump instructions

Opcode	Format	Encoding	Description
RTE R_d	1	**0110000ddd000000**	Return from exception

Table 3.4. Miscellaneous 16-bit instructions

3.3.2. 32-bit Instructions of AAP

In the following list, the encoding is shown with the word at the lower address first.

- Table 3.5 lists all the 32-bit ALU instructions, which have class **00xx**;
- Table 3.6 lists all the 32-bit load/store instructions, which have class **01xx**;
- Table 3.7 lists all the 32-bit branch/jump instructions, which have class **10xx**; and
- There are no 32-bit instructions in the miscellaneous class, but if there were, they would have have class **11xx**.

Opcode	Format	Encoding	Description
NOP R_d,I	14	**1000000dddiiiiii** **0000000dddiiiiii**	No operation
ADD R_d,R_a,R_b	8	**1000001dddaaabbb** **0000000dddaaabbb**	Unsigned add
SUB R_d,R_a,R_b	8	**1000010dddaaabbb** **0000000dddaaabbb**	Unsigned subtract
AND R_d,R_a,R_b	8	**1000011dddaaabbb** **0000000dddaaabbb**	Bitwise AND
OR R_d,R_a,R_b	8	**1000100dddaaabbb** **0000000dddaaabbb**	Bitwise OR

Opcode	Format	Encoding	Description
XOR R_d,R_a,R_b	8	**1000101dddaaabbb** **0000000dddaaabbb**	Bitwise exclusive OR
ASR R_d,R_a,R_b	8	**1000110dddaaabbb** **0000000dddaaabbb**	Arithmetic shift right
LSL R_d,R_a,R_b	8	**1000111dddaaabbb** **0000000dddaaabbb**	Logical shift left
LSR R_d,R_a,R_b	8	**1001000dddaaabbb** **0000000dddaaabbb**	Logical shift right
MOV R_d,R_a	8	**1001001dddaaa000** **0000000dddaaa000**	Move register to register
ADDI R_d,R_a,I	11	**1001010dddaaaiii** **000iiiidddaaaiii**	Unsigned add immediate
SUBI R_d,R_a,I	11	**1001011dddaaaiii** **000iiiidddaaaiii**	Unsigned subtract immediate
ASRI R_d,R_a,I	9	**1001100dddaaaiii** **0000000dddaaaiii**	Arithmetic shift right immediate
LSLI R_d,R_a,I	9	**1001101dddaaaiii** **0000000dddaaaiii**	Logical shift left immediate
LSRI R_d,R_a,I	9	**1001110dddaaaiii** **0000000dddaaaiii**	Logical shift right immediate
MOVI R_d,I	15	**1001111dddiiiiii** **000iiiidddiiiiii**	Move immediate to register
ADDC R_d,R_a,R_b	8	**1000001dddaaabbb** **0000001dddaaabbb**	Add with carry
SUBC R_d,R_a,R_b	8	**1000010dddaaabbb** **0000001dddaaabbb**	Subtract with carry
ANDI R_d,R_a,I	10	**1000011dddaaaiii** **000iii1dddaaaiii**	Bitwise AND immediate
ORI R_d,R_a,I	10	**1000100dddaaaiii** **000iii1dddaaaiii**	Bitwise OR immediate
XORI R_d,R_a,I	10	**1000101dddaaaiii** **000iii1dddaaaiii**	Bitwise exclusive OR immediate

Table 3.5. 32-bit ALU instructions

Opcode	Format	Encoding	Description
LDB $R_d,(R_a,S)$	13	**1010000dddaaasss** **000ssssdddaaasss**	Indexed load byte

Opcode	Format	Encoding	Description
LDW R_d,(R_a,S)	13	**1010100dddaaasss** **000ssssdddaaasss**	Indexed load word
LDB R_d,(R_a+,S)	13	**1010001dddaaasss** **000ssssdddaaasss**	Indexed load byte with postincrement
LDW R_d,(R_a+,S)	13	**1010101dddaaasss** **000ssssdddaaasss**	Indexed load word with postincrement
LDB R_d,(-R_a,S)	13	**1010010dddaaasss** **000ssssdddaaasss**	Indexed load byte with predecrement
LDW R_d,(-R_a,S)	13	**1010110dddaaasss** **000ssssdddaaasss**	Indexed load word with predecrement
STB (R_d,S),R_a	13	**1011000dddaaasss** **000ssssdddaaasss**	Indexed store byte
STW (R_d,S),R_a	13	**1011100dddaaasss** **000ssssdddaaasss**	Indexed store word
STB (R_d+,S),R_a	13	**1011001dddaaasss** **000ssssdddaaasss**	Indexed store byte with postincrement
STW (R_d+,S),R_a	13	**1011101dddaaasss** **000ssssdddaaasss**	Indexed store word with postincrement
STB (-R_d,S),R_a	13	**1011010dddaaasss** **000ssssdddaaasss**	Indexed store byte with predecrement
STW (-R_d,S),R_a	13	**1011111dddaaasss** **000ssssdddaaasss**	Indexed store word with predecrement

Table 3.6. 32-bit load/store instructions

Opcode	Format	Encoding	Description
BRA S	17	**1100000ssssssssss** **000ssssssssssss**	Relative branch
BAL S,R_b	16	**1100001ssssssbbb** **000ssssssssssbbb**	Relative branch and link
BEQ S,R_a,R_b	12	**1100010sssaaabbb** **000ssssssaaabbb**	Relative branch if equal
BNE S,R_a,R_b	12	**1100011sssaaabbb** **000ssssssaaabbb**	Relative branch if not equal
BLTS S,R_a,R_b	12	**1100100sssaaabbb** **000ssssssaaabbb**	Relative branch if signed less than
BLES S,R_a,R_b	12	**1100101sssaaabbb** **000ssssssaaabbb**	Relative branch if signed less than or equal to

Opcode	Format	Encoding	Description
BLTU S,R_a,R_b	12	**1100110sssaaabbb** **000ssssssaaabbb**	Relative branch if unsigned less than
BLEU S,R_a,R_b	12	**1100111sssaaabbb** **000ssssssaaabbb**	Relative branch if unsigned less than or equal to
JMP R_d	8	**1101000ddd000000** **0000000ddd000000**	Absolute jump
JAL R_d,R_b	8	**1101001ddd000bbb** **0000000ddd000bbb**	Absolute jump and link
JEQ R_d,R_a,R_b	8	**1101010dddaaabbb** **0000000dddaaabbb**	Absolute jump if equal
JNE R_d,R_a,R_b	8	**1101011dddaaabbb** **0000000dddaaabbb**	Absolute jump if not equal
JLTS R_d,R_a,R_b	8	**1101100dddaaabbb** **0000000dddaaabbb**	Absolute jump if signed less than
JLES R_d,R_a,R_b	8	**1101101dddaaabbb** **0000000dddaaabbb**	Absolute jump if signed less than or equal to
JLTU R_d,R_a,R_b	8	**1101110dddaaabbb** **0000000dddaaabbb**	Absolute jump if unsigned less than
JLEU R_d,R_a,R_b	8	**1101111dddaaabbb** **0000000dddaaabbb**	Absolute jump if unsigned less than or equal to
JMPL R_d	8	**1101000ddd000000** **0000001ddd000000**	Absolute jump long
JALL R_d,R_b	8	**1101001ddd000bbb** **0000001ddd000bbb**	Absolute jump long and link
JEQL R_d,R_a,R_b	8	**1101010dddaaabbb** **0000001dddaaabbb**	Absolute jump long if equal
JNEL R_d,R_a,R_b	8	**1101011dddaaabbb** **0000001dddaaabbb**	Absolute jump long if not equal
JLTSL R_d,R_a,R_b	8	**1101100dddaaabbb** **0000001dddaaabbb**	Absolute jump long if signed less than
JLESL R_d,R_a,R_b	8	**1101101dddaaabbb** **0000001dddaaabbb**	Absolute jump long if signed less than or equal to
JLTUL R_d,R_a,R_b	8	**1101110dddaaabbb** **0000001dddaaabbb**	Absolute jump long if unsigned less than
JLEUL R_d,R_a,R_b	8	**1101111dddaaabbb** **0000001dddaaabbb**	Absolute jump long if unsigned less than or equal to

Table 3.7. 32-bit branch/jump instructions

3.4. Detailed Descriptions of 16-bit ALU Instructions

3.4.1. NOP: No Operation

Encoding (format 5):

0	0	0	0	0	0	0	d_2	d_1	d_0	i_5	i_4	i_3	i_2	i_1	i_0

Syntax:

NOP R_d,I

Constraints:

d ≤ 7

I ≤ 63

Outcome:

PC ← PC + 1

Notes:

This opcode may trigger side-effects in implementations, depending on the value of I, particularly when simulating (see Section 2.3).
All implementations should use d = 0, I = 0 as the break instruction for debugging, which should halt the processor.

All implementations should use d = 0, I = 1 as a true no-operation instruction.

The rationale behind this decision is that in an erroneous program, the most likely value to be encountered as a random instruction is zero, which will stop the processor.

3.4.2. ADD: Unsigned Add

Encoding (format 1):

0	0	0	0	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

ADD R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

$R_d \leftarrow R_a + R_b$

carry ← (($R_a + R_b$) ≥ 2^{16}) ? 1 : 0

PC ← PC + 1

3.4.3. SUB: Unsigned Subtract

Encoding (format 1):

0	0	0	0	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

SUB R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

$R_d \leftarrow R_a - R_b$

carry ← ($R_b > R_a$) ? 1 : 0

PC ← PC + 1

3.4.4. AND: Bitwise AND

Encoding (format 1):

0	0	0	0	0	1	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

AND R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

$R_d \leftarrow R_a$ & R_b

PC ← PC + 1

3.4.5. OR: Bitwise OR

Encoding (format 1):

0	0	0	0	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

OR R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

$R_d \leftarrow R_a$ | R_b

PC ← PC + 1

3.4.6. XOR: Bitwise Exclusive OR

Encoding (format 1):

0	0	0	0	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

XOR R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

$R_d \leftarrow R_a \wedge R_b$

PC ← PC + 1

3.4.7. ASR: Arithmetic Shift Right

Encoding (format 1):

0	0	0	0	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

ASR R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

$R_d \leftarrow (R_a \mid (carry << 16)) >> R_b)$

carry ← 0

PC ← PC + 1

Notes:

If $R_b \geq 17$ the result in R_d will be zero.

The carry flag is always cleared, even if a shift of zero is specified.

3.4.8. LSL: Logical Shift Left

Encoding (format 1):

0	0	0	0	1	1	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

LSL R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

$d \leq 7$

Outcome:

$R_d \leftarrow R_a << R_b$

$PC \leftarrow PC + 1$

Notes:

If $R_b \geq 16$ the result in R_d will be zero.

3.4.9. LSR: Logical Shift Right

Encoding (format 1):

0	0	0	1	0	0	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

LSR R_d,R_a,R_b

Constraints:

$a \leq 7$

$b \leq 7$

$d \leq 7$

Outcome:

$R_d \leftarrow R_a >> R_b$

$PC \leftarrow PC + 1$

Notes:

If $R_b \geq 16$ the result in R_d will be zero.

3.4.10. MOV: Move Register to Register

Encoding (format 1):

0	0	0	1	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	0	0	0

Syntax:

MOV R_d,R_a

Constraints:

$a \leq 7$

$d \leq 7$

Outcome:

$R_d \leftarrow R_a$

$PC \leftarrow PC + 1$

3.4.11. ADDI: Unsigned Add Immediate

Encoding (format 2):

0	0	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0

Syntax:

ADDI R_d,R_a,I

Constraints:

a ≤ 7

d ≤ 7

I ≤ 7

Outcome:

$R_d \leftarrow R_a + I$

carry ← ((R_a + I) ≥ 2^{16}) ? 1 : 0

PC ← PC + 1

Notes:

Adding constant zero can be used to clear the carry flag.

3.4.12. SUBI: Unsigned Subtract Immediate

Encoding (format 2):

0	**0**	**0**	**1**	**0**	**1**	**1**	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0

Syntax:

SUBI R_d,R_a,I

Constraints:

a ≤ 7

d ≤ 7

I ≤ 7

Outcome:

$R_d \leftarrow R_a - I$

carry ← (I > R_a) ? 1 : 0

PC ← PC + 1

3.4.13. ASRI: Arithmetic Shift Right Immediate

Encoding (format 2):

0	**0**	**0**	**1**	**1**	**0**	**0**	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0

Syntax:

ASRI R_d,R_a,I

Constraints:

a ≤ 7

d ≤ 7

1 ≤ I ≤ 8

Outcome:

$R_d \leftarrow (R_a \mid (carry << 16)) >> I)$

carry ← 0

PC ← PC + 1

Notes:

The shift is encoded with a value 1 less than specified (i.e. a shift of 1 is encoded as 000_2. The rationale is that shifting by zero is pointless. It is not needed to clear the carry flag, since there are other ways of clearing the it (for example adding constant zero).

3.4.14. LSLI: Logical Shift Left Immediate

Encoding (format 2):

0	**0**	**0**	**1**	**1**	**0**	**1**	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0

Syntax:

LSLI R_d,R_a,I

Constraints:

a ≤ 7

d ≤ 7

1 ≤ I ≤ 8

Outcome:

$R_d \leftarrow R_a << I$

PC ← PC + 1

Notes:

The shift is encoded with a value 1 less than specified (i.e. a shift of 1 is encoded as 000_2. The rationale is that shifting by zero is pointless. It is not needed to clear the carry flag, since there are other ways of clearing the it (for example adding constant zero).

3.4.15. LSRI: Logical Shift Right Immediate

Encoding (format 2):

0	**0**	**0**	**1**	**1**	**1**	**0**	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0

Syntax:

LSRI R_d,R_a,I

Constraints:

a ≤ 7

d ≤ 7

1 ≤ I ≤ 8

Outcome:

$R_d \leftarrow R_a >> I$

PC ← PC + 1

Notes:

The shift is encoded with a value 1 less than specified (i.e. a shift of 1 is encoded as 000_2. The rationale is that shifting by zero is pointless. It is not needed to clear the carry flag, since there are other ways of clearing the it (for example adding constant zero).

3.4.16. MOVI: Move Immediate to Register

Encoding (format 5):

0	0	0	1	1	1	1	d_2	d_1	d_0	i_5	i_4	i_3	i_2	i_1	i_0

Syntax:

MOVI R_d,I

Constraints:

$d \leq 7$

$I \leq 63$

Outcome:

$R_d \leftarrow I$

$PC \leftarrow PC + 1$

3.5. Detailed Descriptions of 16-bit Load/Store Instructions

3.5.1. LDB: Indexed Load Byte

Encoding (format 4):

0	0	1	0	0	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

LDB R_d,(R_a,S)

Constraints:

$d \leq 7$

$-4 \leq S \leq 3$

Outcome:

$R_d \leftarrow dmem[R_a + SignExt(S)]$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.2. LDW: Indexed Load Word

Encoding (format 4):

0	0	1	0	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

LDW R_d,(R_a,S)

Constraints:

$d \leq 7$

$-4 \leq S \leq 3$

Outcome:

$R_d \leftarrow dmem\ [R_a + SignExt(S)]\ |\ (dmem[R_a + SignExt(S) + 1] << 8)$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.3. LDB: Indexed Load Byte with Postincrement

Encoding (format 4):

0	**0**	**1**	**0**	**0**	**0**	**1**	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

LDB R_d**,(**R_a**+,**S**)**

Constraints:

$d \leq 7$

$-4 \leq S \leq 3$

Outcome:

$R_d \leftarrow dmem[R_a + SignExt(S)]$

$R_a \leftarrow R_a + SignExt(S)$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.4. LDW: Indexed Load Word with Postincrement

Encoding (format 4):

0	**0**	**1**	**0**	**1**	**0**	**1**	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

LDW R_d**,(**R_a**+,**S**)**

Constraints:

$d \leq 7$

$-4 \leq S \leq 3$

Outcome:

$R_d \leftarrow dmem[R_a + SignExt(S)]\ |\ (dmem[R_a + SignExt(S) + 1] << 8)$

$R_a \leftarrow R_a + SignExt(S)$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.5. LDB: Indexed Load Byte with Predecrement

Encoding (format 4):

0	0	1	0	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

LDB R_d**,(-**R_a**,**S**)**

Constraints:

$d \le 7$

$-4 \le S \le 3$

Outcome:

$R_a \leftarrow R_a$ - SignExt(S)

$R_d \leftarrow$ dmem[R_a]

PC ← PC + 1

Notes:

For the avoidance of doubt, the decrement of R_a is carried out *before* R_a is used to compute the address for loading.

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.6. LDW: Indexed Load Word with Predecrement

Encoding (format 4):

0	0	1	0	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

LDW R_d**,(-**R_a**,**S**)**

Constraints:

$d \le 7$

$-4 \le S \le 3$

Outcome:

$R_a \leftarrow R_a$ - SignExt(S)

$R_d \leftarrow$ dmem[R_a] | (dmem[R_a + 1] << 8)

PC ← PC + 1

Notes:

For the avoidance of doubt, the decrement of R_a is carried out *before* R_a is used to compute the address for loading.

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.7. STB: Indexed Store Byte

Encoding (format 4):

0	0	1	1	0	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

STB (**(**R_d**,**S**)**,R_a

Constraints:

$d \leq 7$

$-4 \leq S \leq 3$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \,\&\, 255)$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.8. STW: Indexed Store Word

Encoding (format 4):

0	0	1	1	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

STW (**(**R_d**,**S**)**,R_a

Constraints:

$d \leq 7$

$-4 \leq S \leq 3$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \,\&\, 255)$

$dmem[R_d + SignExt(S) + 1] \leftarrow (R_a >> 8)$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.9. STB: Indexed Store Byte with Postincrement

Encoding (format 4):

0	0	1	1	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

STB (**(**R_d**+,**S**)**,R_a

Constraints:

$d \le 7$

$-4 \le S \le 3$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \& 255)$

$R_d \leftarrow R_d + SignExt(S)$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.10. STW: Indexed Store Word with Postincrement

Encoding (format 4):

0	0	1	1	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

STW (R_d**+,**S**),**R_a

Constraints:

$d \le 7$

$-4 \le S \le 3$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \& 255)$

$dmem[R_d + SignExt(S) + 1] \leftarrow (R_a >> 8)$

$R_d \leftarrow R_d + SignExt(S)$

$PC \leftarrow PC + 1$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.11. STB: Indexed Store Byte with Predecrement

Encoding (format 4):

0	0	1	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

STB (-R_d**,**S**),**R_a

Constraints:

$d \le 7$

$-4 \le S \le 3$

Outcome:

$R_d \leftarrow R_d - SignExt(S)$

dmem[R_d] ← (R_a & 255)

PC ← PC + 1

Notes:

For the avoidance of doubt, the decrement of R_a is carried out *before* R_a is used to compute the address for loading.

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.5.12. STW: Indexed Store Word with Predecrement

Encoding (format 4):

0	0	1	1	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

Syntax:

STW (-R_d**,S),**R_a

Constraints:

d ≤ 7

-4 ≤ S ≤ 3

Outcome:

R_d ← R_d - SignExt(S)

dmem[R_d] ← (R_a & 255)

dmem[R_d + 1] ← (R_a >> 8)

PC ← PC + 1

Notes:

For the avoidance of doubt, the decrement of R_a is carried out *before* R_a is used to compute the address for loading.

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.6. Detailed Descriptions of 16-bit Branch/Jump Instructions

Note

The only branch/jump comparisons provided are for "equal", "not equal", "less than" and "greater than". Branch/jump comparisons for "less than or equal" and "greater than or equal" can be provided by using "greater than" and "less than" respectively in the opposite direction."

Purists will point out that this reduces the opportunity for branch prediction and pipeline preservation. However the limited instruction space means not all opcodes can be provided.

3.6.1. BRA: Relative Branch

Encoding (format 7):

0	1	0	0	0	0	0	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0

Syntax:

BRA S

Constraints:

-256 ≤ S ≤ 255

Outcome:

PC ← PC + SignExt(S)

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.2. BAL: Relative Branch and Link

Encoding (format 6):

0	**1**	**0**	**0**	**0**	**0**	**1**	s_5	s_4	s_3	s_2	s_1	s_0	b_2	b_1	b_0

Syntax:

BAL S,R_b

Constraints:

b ≤ 7

-32 ≤ S ≤ 31

Outcome:

R_b ← PC + 1

PC ← PC + SignExt(S)

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.3. BEQ: Relative Branch if Equal

Encoding (format 3):

0	**1**	**0**	**0**	**0**	**1**	**0**	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

BEQ S,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

-4 ≤ S ≤ 3

Outcome:

PC ← (R_a = R_b) ? PC + SignExt(S) : PC + 1

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.4. BNE: Relative Branch if Not Equal

Encoding (format 3):

0	1	0	0	0	1	1	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

BNE S,R_a,R_b

Constraints:

$a \leq 7$

$b \leq 7$

$-4 \leq S \leq 3$

Outcome:

$PC \leftarrow (R_a \neq R_b)$? PC + SignExt(S) : PC + 1

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.5. BLTS: Relative Branch if Signed Less Than

Encoding (format 3):

0	1	0	0	1	0	0	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

BLTS S,R_a,R_b

Constraints:

$a \leq 7$

$b \leq 7$

$-4 \leq S \leq 3$

Outcome:

$PC \leftarrow (R_a < R_b)$? PC + SignExt(S) : PC + 1

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.6. BLES: Relative Branch if Signed Less Than or Equal To

Encoding (format 3):

0	1	0	0	1	0	1	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

BLES S,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

-4 ≤ S ≤ 3

Outcome:

PC ← (R_a ≤ R_b) ? PC + SignExt(S) : PC + 1

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.7. BLTU: Relative Branch if Unsigned Less Than

Encoding (format 3):

0	1	0	0	1	1	0	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

BLTU S,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

-4 ≤ S ≤ 3

Outcome:

PC ← (R_a < R_b) ? PC + SignExt(S) : PC + 1

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.8. BLEU: Relative Branch if Unsigned Less Than or Equal To

Encoding (format 3):

0	1	0	0	1	1	1	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

BLEU S,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

-4 ≤ S ≤ 3

Outcome:

PC ← (R_a ≤ R_b) ? PC + SignExt(S) : PC + 1

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.6.9. JMP: Absolute Jump

Encoding (format 1):

0	1	0	1	0	0	0	d_2	d_1	d_0	0	0	0	0	0	0

Syntax:

JMP R_d

Constraints:

d ≤ 7

Outcome:

PC ← R_d

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jumping to a non-existent location will trigger a bus error exception.

3.6.10. JAL: Absolute Jump and Link

Encoding (format 1):

0	1	0	1	0	0	1	d_2	d_1	d_0	0	0	0	b_2	b_1	b_0

Syntax:

JAL R_d,R_b

Constraints:

b ≤ 7

d ≤ 7

Outcome:

$R_b \leftarrow PC + 1$

$PC \leftarrow R_d$

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jumping to a non-existent location will trigger a bus error exception.

3.6.11. JEQ: Absolute Jump if Equal

Encoding (format 1):

0	1	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

JEQ R_d,R_a,R_b

Constraints:

$a \leq 7$

$b \leq 7$

$d \leq 7$

Outcome:

$PC \leftarrow (R_a = R_b)\ ?\ R_d : PC + 1$

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.6.12. JNE: Absolute Jump if Not Equal

Encoding (format 1):

0	1	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

JNE R_d,R_a,R_b

Constraints:

$a \leq 7$

$b \leq 7$

$d \leq 7$

Outcome:

$PC \leftarrow (R_a \neq R_b)\ ?\ R_d : PC + 1$

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.6.13. JLTS: Absolute Jump if Signed Less Than

Encoding (format 1):

0	1	0	1	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

JLTS R_d,R_a,R_b

Constraints:

$a \leq 7$

$b \leq 7$

$d \leq 7$

Outcome:

$PC \leftarrow (R_a < R_b)\ ?\ R_d : PC + 1$

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.6.14. JLES: Absolute Jump if Signed Less Than or Equal To

Encoding (format 1):

0	1	0	1	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

JLES R_d,R_a,R_b

Constraints:

$a \leq 7$

$b \leq 7$

$d \leq 7$

Outcome:

$PC \leftarrow (R_a \leq R_b)\ ?\ R_d : PC + 1$

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.6.15. JLTU: Absolute Jump if Unsigned Less Than

Encoding (format 1):

0	1	0	1	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

JLTU R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

PC ← (R_a < R_b) ? R_d : PC + 1

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.6.16. JLEU: Absolute Jump if Unsigned Less Than or Equal To

Encoding (format 1):

0	1	0	1	1	1	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

Syntax:

JLEU R_d,R_a,R_b

Constraints:

a ≤ 7

b ≤ 7

d ≤ 7

Outcome:

PC ← (R_a ≤ R_b) ? R_d : PC + 1

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.7. Detailed Descriptions of 16-bit Miscellaneous Instructions

3.7.1. RTE: Return from Exception

Encoding (format 1):

0	1	1	0	0	0	0	d_2	d_1	d_0	0	0	0	0	0	0

Syntax:

RTE R_d

Constraints:

$d \leq 7$

Outcome:

$PC \leftarrow R_d$

Notes:

This opcode is not fully defined, pending agreement on the exception mechanism for AAP.

3.8. Detailed Descriptions of 32-bit ALU Instructions

At this time, this section is incomplete.

3.8.1. NOP: No Operation

Encoding (format 14, first word at lower address):

1	0	0	0	0	0	0	d_2	d_1	d_0	i_5	i_4	i_3	i_2	i_1	i_0
0	0	0	0	0	0	0	d_5	d_4	d_3	i_{11}	i_{10}	i_9	i_8	i_7	i_6

Syntax:

NOP R_d,I

Constraints:

$d \leq 63$

$I \leq 4095$

Outcome:

$PC \leftarrow PC + 1$

Notes:

This opcode may trigger side-effects in implementations, depending on the value of I, particularly when simulating (see Section 2.3).

There are no conventions for any values of d or I for the 32-bit version of NOP.

3.8.2. ADD: Unsigned Add

Encoding (format 8, first word at lower address):

1	0	0	0	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

ADD R_d,R_a,R_b

Constraints:

$a \leq 63$

$b \leq 63$

$d \leq 63$

Outcome:

$R_d \leftarrow R_a + R_b$

carry $\leftarrow$ (($R_a + R_b$) $\geq 2^{16}$) ? 1 : 0

PC $\leftarrow$ PC + 2

3.8.3. SUB: Unsigned Subtract

Encoding (format 8, first word at lower address):

1	0	0	0	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

SUB R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a - R_b$

carry $\leftarrow$ ($R_b > R_a$) ? 1 : 0

PC $\leftarrow$ PC + 2

3.8.4. AND: Bitwise AND

Encoding (format 8, first word at lower address):

1	0	0	0	0	1	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

AND R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a \,\&\, R_b$

PC $\leftarrow$ PC + 2

3.8.5. OR: Bitwise OR

Encoding (format 8, first word at lower address):

1	0	0	0	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

OR R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a \mid R_b$

PC ← PC + 2

3.8.6. XOR: Bitwise Exclusive OR

Encoding (format 8, first word at lower address):

1	0	0	0	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

XOR R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a \wedge R_b$

PC ← PC + 2

3.8.7. ASR: Arithmetic Shift Right

Encoding (format 8, first word at lower address):

1	0	0	0	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

ASR R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow (R_a \mid (carry << 16)) >> R_b)$

carry ← 0

PC ← PC + 2

Notes:

If $R_b \geq 17$ the result in R_d will be zero.

The carry flag is always cleared, even if a shift of zero is specified.

3.8.8. LSL: Logical Shift Left

Encoding (format 8, first word at lower address):

1	**0**	**0**	**0**	**1**	**1**	**1**	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	**0**	**0**	**0**	**0**	**0**	**0**	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

LSL R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a << R_b$

PC ← PC + 2

Notes:

If $R_b \geq 16$ the result in R_d will be zero.

3.8.9. LSR: Logical Shift Right

Encoding (format 8, first word at lower address):

1	**0**	**0**	**1**	**0**	**0**	**0**	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	**0**	**0**	**0**	**0**	**0**	**0**	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

LSR R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a >> R_b$

PC ← PC + 2

Notes:

If $R_b \geq 16$ the result in R_d will be zero.

3.8.10. MOV: Move Register to Register

Encoding (format 8, first word at lower address):

1	0	0	1	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	0	0	0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	0	0	0

Syntax:

MOV R_d,R_a

Constraints:

$a \leq 63$

$d \leq 63$

Outcome:

$R_d \leftarrow R_a$

$PC \leftarrow PC + 2$

3.8.11. ADDI: Unsigned Add Immediate

Encoding (format 11, first word at lower address):

1	0	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	0	0	i_9	i_8	i_7	i_6	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

ADDI R_d,R_a,I

Constraints:

$a \leq 63$

$d \leq 63$

$I \leq 63$

Outcome:

$R_d \leftarrow R_a + I$

$carry \leftarrow ((R_a + I) \geq 2^{16}) \; ? \; 1 : 0$

$PC \leftarrow PC + 2$

Notes:

Adding constant zero can be used to clear the carry flag.

3.8.12. SUBI: Unsigned Subtract Immediate

Encoding (format 11, first word at lower address):

1	0	0	1	0	1	1	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	0	0	i_9	i_8	i_7	i_6	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

SUBI R_d,R_a,I

Constraints:

a ≤ 63

d ≤ 63

I ≤ 63

Outcome:

$R_d \leftarrow R_a$ - I

carry ← (I > R_a) ? 1 : 0

PC ← PC + 2

3.8.13. ASRI: Arithmetic Shift Right Immediate

Encoding (format 9, first word at lower address):

1	**0**	**0**	**1**	**1**	**0**	**0**	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	**0**	**0**	**0**	**0**	**0**	**0**	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

ASRI R_d,R_a,I

Constraints:

a ≤ 63

d ≤ 63

1 ≤ I ≤ 64

Outcome:

$R_d \leftarrow$ (R_a | (carry << 16)) >> I)

carry ← 0

PC ← PC + 2

Notes:

If I ≥ 17 the result in R_d will be zero.

The shift is encoded with a value 1 less than specified (i.e. a shift of 1 is encoded as 000000_2. The rationale is that shifting by zero is pointless. It is not needed to clear the carry flag, since there are other ways of clearing the it (for example adding constant zero).

3.8.14. LSLI: Logical Shift Left Immediate

Encoding (format 9, first word at lower address):

1	**0**	**0**	**1**	**1**	**0**	**1**	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	**0**	**0**	**0**	**0**	**0**	**0**	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

LSLI R_d,R_a,I

Constraints:

a ≤ 63

d ≤ 63

1 ≤ I ≤ 64

Outcome:

$R_d \leftarrow R_a << I$

PC ← PC + 2

Notes:

If I ≥ 16 the result in R_d will be zero.

The shift is encoded with a value 1 less than specified (i.e. a shift of 1 is encoded as 000000_2. The rationale is that shifting by zero is pointless. It is not needed to clear the carry flag, since there are other ways of clearing the it (for example adding constant zero).

3.8.15. LSRI: Logical Shift Right Immediate

Encoding (format 9, first word at lower address):

1	**0**	**0**	**1**	**1**	**1**	**0**	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	**0**	**0**	**0**	**0**	**0**	**0**	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

LSRI R_d,R_a,I

Constraints:

a ≤ 63

d ≤ 63

1 ≤ I ≤ 64

Outcome:

$R_d \leftarrow R_a >> I$

PC ← PC + 2

Notes:

If I ≥ 16 the result in R_d will be zero.

The shift is encoded with a value 1 less than specified (i.e. a shift of 1 is encoded as 000000_2. The rationale is that shifting by zero is pointless. It is not needed to clear the carry flag, since there are other ways of clearing the it (for example adding constant zero).

3.8.16. MOVI: Move Immediate to Register

Encoding (format 15, first word at lower address):

1	**0**	**0**	**1**	**1**	**1**	**1**	d_2	d_1	d_0	i_5	i_4	i_3	i_2	i_1	i_0
0	**0**	**0**	i_{15}	i_{14}	i_{13}	i_{12}	d_5	d_4	d_3	i_{11}	i_{10}	i_9	i_8	i_7	i_6

Syntax:

MOVI R_d,I

Constraints:

d ≤ 63

I ≤ 65535

Outcome:

$R_d \leftarrow I$

PC ← PC + 2

3.8.17. ADDC: Unsigned Add with Carry

Encoding (format 8, first word at lower address):

1	0	0	0	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	1	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

ADDC R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a + R_b + carry$

$carry \leftarrow ((R_a + R_b + carry) \geq 2^{16}) \ ? \ 1 : 0$

PC ← PC + 2

3.8.18. SUBC: Unsigned Subtract with Carry

Encoding (format 8, first word at lower address):

1	0	0	0	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	1	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

SUBC R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

$R_d \leftarrow R_a - R_b - carry$

$carry \leftarrow ((R_b + carry) > R_a) \ ? \ 1 : 0$

PC ← PC + 2

3.8.19. ANDI: Bitwise AND Immediate

Encoding (format 10, first word at lower address):

1	0	0	0	0	1	1	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	0	0	i_8	i_7	i_6	1	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

ANDI R_d,R_a,I

Constraints:

a ≤ 63

d ≤ 63

I ≤ 511

Outcome:

$R_d \leftarrow R_a$ & I

PC ← PC + 2

3.8.20. ORI: Bitwise OR immediate

Encoding (format 10, first word at lower address):

1	0	0	0	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	0	0	i_8	i_7	i_6	1	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

ORI R_d,R_a,I

Constraints:

a ≤ 63

d ≤ 63

I ≤ 511

Outcome:

$R_d \leftarrow R_a$ | I

PC ← PC + 2

3.8.21. XORI: Bitwise Exclusive OR Immediate

Encoding (format 10, first word at lower address):

1	0	0	0	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	i_2	i_1	i_0
0	0	0	i_8	i_7	i_6	1	d_5	d_4	d_3	a_5	a_4	a_3	i_5	i_4	i_3

Syntax:

XORI R_d,R_a,I

Constraints:

a ≤ 63

d ≤ 63

I ≤ 511

Outcome:

$R_d \leftarrow R_a$ ^ I

PC ← PC + 2

3.9. Detailed Descriptions of 32-bit Load/Store Instructions

3.9.1. LDB: Indexed Load Byte

Encoding (format 13, first word at lower address):

1	0	1	0	0	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

LDB R_d**,(**R_a**,**S**)**

Constraints:

d ≤ 63

-512 ≤ S ≤ 511

Outcome:

$R_d \leftarrow$ dmem[R_a + SignExt(S)]

PC ← PC + 2

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.2. LDW: Indexed Load Word

Encoding (format 13, first word at lower address):

1	0	1	0	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

LDW R_d**,(**R_a**,**S**)**

Constraints:

d ≤ 63

-512 ≤ S ≤ 511

Outcome:

$R_d \leftarrow$ dmem [R_a + SignExt(S)] | (dmem[R_a + SignExt(S) + 1] << 8)

PC ← PC + 2

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.3. LDB: Indexed Load Byte with Postincrement

Encoding (format 13, first word at lower address):

1	0	1	0	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

LDB R_d**,(**R_a**+,**S**)**

Constraints:

$d \leq 63$

$-512 \leq S \leq 511$

Outcome:

$R_d \leftarrow dmem[R_a + SignExt(S)]$

$R_a \leftarrow R_a + SignExt(S)$

$PC \leftarrow PC + 2$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.4. LDW: Indexed Load Word with Postincrement

Encoding (format 13, first word at lower address):

1	0	1	0	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

LDW R_d**,(**R_a**+,**S**)**

Constraints:

$d \leq 63$

$-512 \leq S \leq 511$

Outcome:

$R_d \leftarrow dmem [R_a + SignExt(S)] \mid (dmem[R_a + SignExt(S) + 1] << 8)$

$R_a \leftarrow R_a + SignExt(S)$

$PC \leftarrow PC + 2$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.5. LDB: Indexed Load Byte with Predecrement

Encoding (format 13, first word at lower address):

1	0	1	0	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

LDB R_d**,(-**R_a**,**S**)**

Constraints:

$d \leq 63$

$-512 \leq S \leq 511$

Outcome:

$R_a \leftarrow R_a$ - SignExt(S)

$R_d \leftarrow$ dmem[R_a]

PC ← PC + 2

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.6. LDW: Indexed Load Word with Predecrement

Encoding (format 13, first word at lower address):

1	0	1	0	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

LDW R_d**,(-**R_a**,**S**)**

Constraints:

$d \leq 63$

$-512 \leq S \leq 511$

Outcome:

$R_a \leftarrow R_a$ - SignExt(S)

$R_d \leftarrow$ dmem [R_a] | (dmem[R_a + 1] << 8)

PC ← PC + 2

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.7. STB: Indexed Store Byte

Encoding (format 13, first word at lower address):

1	0	1	1	0	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

STB (**R_d**,S),**R_a**

Constraints:

$d \leq 63$

$-512 \leq S \leq 511$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \& 255)$

$PC \leftarrow PC + 2$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.8. STW: Indexed Store Word

Encoding (format 13, first word at lower address):

0	0	1	1	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

STW (**R_d**,S),**R_a**

Constraints:

$d \leq 63$

$-512 \leq S \leq 511$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \& 255)$

$dmem[R_d + SignExt(S) + 1] \leftarrow (R_a >> 8)$

$PC \leftarrow PC + 2$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.9. STB: Indexed Store Byte with Postincrement

Encoding (format 13, first word at lower address):

1	0	1	1	0	0	1	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0

0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

STB (**R_d+**,S),**R_a**

Constraints:

$d \le 63$

$-512 \le S \le 511$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \& 255)$

$R_d \leftarrow R_d + SignExt(S)$

$PC \leftarrow PC + 2$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.10. STW: Indexed Store Word with Postincrement

Encoding (format 13, first word at lower address):

1	0	1	1	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

STW (**R_d+,S),R_a**

Constraints:

$d \le 63$

$-512 \le S \le 511$

Outcome:

$dmem[R_d + SignExt(S)] \leftarrow (R_a \& 255)$

$dmem[R_d + SignExt(S) + 1] \leftarrow (R_a >> 8)$

$R_d \leftarrow R_d + SignExt(S)$

$PC \leftarrow PC + 2$

Notes:

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.11. STB: Indexed Store Byte with Predecrement

Encoding (format 13, first word at lower address):

1	0	1	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

STB (**-R_d**,S),R_a

Constraints:

$d \le 63$

$-512 \le S \le 511$

Outcome:

$R_d \leftarrow R_d$ - SignExt(S)

dmem[R_d] ← (R_a & 255)

PC ← PC + 2

Notes:

For the avoidance of doubt, the decrement of R_a is carried out *before* R_a is used to compute the address for loading.

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.9.12. STW: Indexed Store Word with Predecrement

Encoding (format 13, first word at lower address):

1	0	1	1	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	s_2	s_1	s_0
0	0	0	s_9	s_8	s_7	s_6	d_5	d_4	d_3	a_5	a_4	a_3	s_5	s_4	s_3

Syntax:

STW (-R_d**,**S**),**R_a

Constraints:

d ≤ 63

-512 ≤ S ≤ 511

Outcome:

$R_d \leftarrow R_d$ - SignExt(S)

dmem[R_d] ← (R_a & 255)

dmem[R_d + 1] ← (R_a >> 8)

PC ← PC + 2

Notes:

For the avoidance of doubt, the decrement of R_a is carried out *before* R_a is used to compute the address for loading.

This opcode accesses data memory, and the computed address is therefore a byte address. Accessing a non-existent memory location will trigger a bus error exception.

3.10. Detailed Descriptions of 32-bit Branch/Jump Instructions

Note
As with the 16-bit instructions, only a limited range of comparisons is provided. See Section 3.6 for an explanation.

3.10.1. BRA: Relative Branch

Encoding (format 17, first word at lower address):

1	1	0	0	0	0	0	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0
0	0	0	s_{21}	s_{20}	s_{19}	s_{18}	s_{17}	s_{16}	s_{15}	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9

Syntax:

BRA S

Constraints:

-2,097,152 ≤ S ≤ 2,097,151

Outcome:

PC ← PC + SignExt(S)

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.2. BAL: Relative Branch and Link

Encoding (format 16, first word at lower address):

1	1	0	0	0	0	1	s_5	s_4	s_3	s_2	s_1	s_0	b_2	b_1	b_0
0	0	0	s_{18}	s_{17}	s_{16}	s_{15}	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	b_5	b_4	b_3

Syntax:

BAL S,R_b

Constraints:

b ≤ 63

-262,144 ≤ S ≤ 262,141

Outcome:

R_b ← PC + 2

PC ← PC + SignExt(S)

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.3. BEQ: Relative Branch if Equal

Encoding (format 12, first word at lower address):

1	1	0	0	0	1	0	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	s_9	s_8	s_7	s_6	s_5	s_4	s_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

BEQ S,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

-512 ≤ S ≤ 511

Outcome:

PC ← (R_a = R_b) ? PC + SignExt(S) : PC + 2

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.4. BNE: Relative Branch if Not Equal

Encoding (format 12, first word at lower address):

1	1	0	0	0	1	1	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	s_9	s_8	s_7	s_6	s_5	s_4	s_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

BNE S,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

-512 ≤ S ≤ 511

Outcome:

PC ← (R_a ≠ R_b) ? PC + SignExt(S) : PC + 2

Notes:

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.5. BLTS: Relative Branch if Signed Less Than

Encoding (format 12, first word at lower address):

1	1	0	0	1	0	0	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	s_9	s_8	s_7	s_6	s_5	s_4	s_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

BLTS S,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

-512 ≤ S ≤ 511

Outcome:

PC ← (R_a < R_b) ? PC + SignExt(S) : PC + 2

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.6. BLES: Relative Branch if Signed Less Than or Equal To

Encoding (format 12, first word at lower address):

1	1	0	0	1	0	1	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	s_9	s_8	s_7	s_6	s_5	s_4	s_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

BLES S,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

-512 ≤ S ≤ 511

Outcome:

PC ← (R_a ≤ R_b) ? PC + SignExt(S) : PC + 2

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.7. BLTU: Relative Branch if Unsigned Less Than

Encoding (format 12, first word at lower address):

1	1	0	0	1	1	0	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	s_9	s_8	s_7	s_6	s_5	s_4	s_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

BLTU S,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

-512 ≤ S ≤ 511

Outcome:

PC ← (R_a < R_b) ? PC + SignExt(S) : PC + 2

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.8. BLEU: Relative Branch if Unsigned Less Than or Equal To

Encoding (format 12, first word at lower address):

1	1	0	0	1	1	1	s_2	s_1	s_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	s_9	s_8	s_7	s_6	s_5	s_4	s_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

BLEU S,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

-512 ≤ S ≤ 511

Outcome:

PC ← (R_a ≤ R_b) ? PC + SignExt(S) : PC + 2

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the offset is the number of words by which to adjust the PC.

Branching to a non-existent location will trigger a bus error exception.

3.10.9. JMP: Absolute Jump

Encoding (format 8, first word at lower address):

1	1	0	1	0	0	0	d_2	d_1	d_0	0	0	0	0	0	0
0	0	0	1	0	0	0	d_5	d_4	d_3	0	0	0	0	0	0

Syntax:

JMP R_d

Constraints:

d ≤ 63

Outcome:

PC ← R_d

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jumping to a non-existent location will trigger a bus error exception.

3.10.10. JAL: Absolute Jump and Link

Encoding (format 8, first word at lower address):

1	1	0	1	0	0	1	d_2	d_1	d_0	0	0	0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	0	0	0	b_5	b_4	b_3

Syntax:

JAL R_d,R_b

Constraints:

$b \leq 63$

$d \leq 63$

Outcome:

$R_b \leftarrow PC + 2$

$PC \leftarrow R_d$

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jumping to a non-existent location will trigger a bus error exception.

3.10.11. JEQ: Absolute Jump if Equal

Encoding (format 8, first word at lower address):

1	1	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JEQ R_d,R_a,R_b

Constraints:

$a \leq 63$

$b \leq 63$

$d \leq 63$

Outcome:

$PC \leftarrow (R_a = R_b)\ ?\ R_d : PC + 2$

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.12. JNE: Absolute Jump if Not Equal

Encoding (format 8, first word at lower address):

1	1	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JNE R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

PC ← (R_a ≠ R_b) ? R_d : PC + 2

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.13. JLTS: Absolute Jump if Signed Less Than

Encoding (format 8, first word at lower address):

1	1	0	1	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JLTS R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

PC ← (R_a < R_b) ? R_d : PC + 2

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.14. JLES: Absolute Jump if Signed Less Than or Equal To

Encoding (format 8, first word at lower address):

1	1	0	1	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JLES R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

PC ← (R_a ≤ R_b) ? R_d : PC + 2

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.15. JLTU: Absolute Jump if Unsigned Less Than

Encoding (format 8, first word at lower address):

1	1	0	1	1	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JLTU R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

PC ← (R_a < R_b) ? R_d : PC + 2

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.16. JLEU: Absolute Jump if Unsigned Less Than or Equal To

Encoding (format 8, first word at lower address):

1	1	0	1	1	1	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

0	0	0	0	0	0	0	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JLEU R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 63

Outcome:

PC ← (R_a ≤ R_b) ? R_d : PC + 2

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.17. JMPL: Absolute Jump Long

Encoding (format 8, first word at lower address):

1	1	0	1	0	0	0	d_2	d_1	d_0	0	0	0	0	0	0

0	0	0	0	0	0	1	d_5	d_4	d_3	0	0	0	0	0	0

Syntax:

JMPL R_d

Constraints:

d ≤ 62

(d % 2) = 2

Outcome:

PC ← (R_{d+1} << 16) | R_d

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jumping to a non-existent location will trigger a bus error exception.

3.10.18. JALL: Absolute Jump Long and Link

Encoding (format 8, first word at lower address):

1	1	0	1	0	0	1	d_2	d_1	d_0	0	0	0	b_2	b_1	b_0

0	0	0	0	0	0	1	d_5	d_4	d_3	0	0	0	b_5	b_4	b_3

Syntax:

JALL R_d,R_b

Constraints:

b ≤ 63

d ≤ 62

(d % 2) = 2

Outcome:

R_b ← PC + 2

PC ← (R_{d+1} << 16) | R_d

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jumping to a non-existent location will trigger a bus error exception.

3.10.19. JEQL: Absolute Jump Long if Equal

Encoding (format 8, first word at lower address):

1	1	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	1	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JEQL R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 62

(d % 2) = 2

Outcome:

PC ← (R_a = R_b) ? ((R_{d+1} << 16) | R_d) : PC + 2

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.20. JNEL: Absolute Jump Long if Not Equal

Encoding (format 8, first word at lower address):

1	1	0	1	0	1	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	1	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JNEL R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 62

(d % 2) = 2

Outcome:

PC ← ($R_a \neq R_b$) ? ((R_{d+1} << 16) | R_d) : PC + 2

Notes:

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.21. JLTSL: Absolute Jump Long if Signed Less Than

Encoding (format 8, first word at lower address):

1	1	0	1	1	0	0	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	1	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

`JLTSL` R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 62

(d % 2) = 2

Outcome:

PC ← ($R_a < R_b$) ? ((R_{d+1} << 16) | R_d) : PC + 2

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.22. JLESL: Absolute Jump Long if Signed Less Than or Equal To

Encoding (format 8, first word at lower address):

1	1	0	1	1	0	1	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	1	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JLESL R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 62

(d % 2) = 2

Outcome:

PC ← (R_a ≤ R_b) ? ((R_{d+1} << 16) | R_d) : PC + 2

Notes:

The comparison between R_a and R_b is a *signed* comparison, where the contents of each register is treated as a 2's-complement signed number.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.23. JLTUL: Absolute Jump Long if Unsigned Less Than

Encoding (format 8, first word at lower address):

1	**1**	**0**	**1**	**1**	**1**	**0**	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0
0	**0**	**0**	**0**	**0**	**0**	**1**	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JLTUL R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 62

(d % 2) = 2

Outcome:

PC ← (R_a < R_b) ? ((R_{d+1} << 16) | R_d) : PC + 2

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.10.24. JLEUL: Absolute Jump Long if Unsigned Less Than or Equal To

Encoding (format 8, first word at lower address):

1	**1**	**0**	**1**	**1**	**1**	**1**	d_2	d_1	d_0	a_2	a_1	a_0	b_2	b_1	b_0

0	0	0	0	0	0	1	d_5	d_4	d_3	a_5	a_4	a_3	b_5	b_4	b_3

Syntax:

JLEUL R_d,R_a,R_b

Constraints:

a ≤ 63

b ≤ 63

d ≤ 62

(d % 2) = 2

Outcome:

PC ← (R_a ≤ R_b) ? ((R_{d+1} << 16) | R_d) : PC + 2

Notes:

The comparison between R_a and R_b is an *unsigned* comparison.

Remember that the program counter is a word address, so the value in R_d should be a word address.

Jump to a non-existent location will trigger a bus error exception.

3.11. Detailed Descriptions of 32-bit Miscellaneous Instructions

There are currently no 32-bit instructions defined in this class.

Chapter 4. ABI

4.1. Defined Registers

Because of the variability in the architecture it is difficult to be too rigid on the ABI. In any case part of the purpose of this architecture to allow exploration of different ABI's. Within this section, the identifier $\mathtt{R}_{max}$ is used to indicate the highest numbered register in the architecture.

The meanings of the following registers are defined.

- **R0**: Link Register
- **R1**: Stack Pointer

Note in particular no frame pointer is defined. It is up to the implementer to decide policy with regard to use of a frame pointer.

4.2. Calling Convention

Again this is flexible, particularly where there can be very few registers. These are the general guidelines.

- All byte arguments are promoted to 16-bits.
- Arguments are passed in **R2**–**R7** (or **R2**–$\mathtt{R}_{max}$ if there are fewer than 8 registers).
- Results are returned on the same registers used to pass arguments.
- *Varargs* are always passed on the stack.
- A good guideline is that approximately one third of unallocated registers should be caller saved, although that can increase to one half where there are plenty of registers. The following registers (if present) are caller saved: **R10**, **R13**, **R16**, **R19**, **R22**, **R25**, **R28**, **R31**, **R33**, **R35**, **R37**, **R39**, **R41**, **R43**, **R45**, **R47**, **R49**, **R51**, **R53**, **R55**, **R57**, **R59**, **R61** and **R63**.