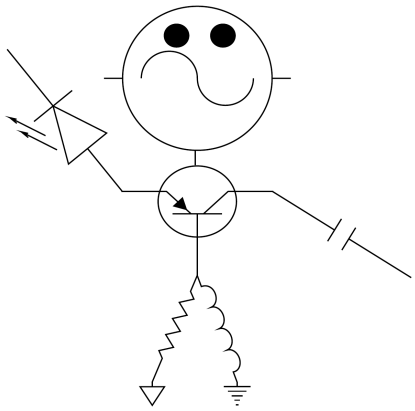
Smallpond Architecture Reference Manual

Zachary Salim, Dominique Hickson,
Andrew Balys, Justin Charlong

Fall Semester 2017



Contents

1 Introduction **4**

2 Purpose **4**

3 Naming **4**

4 Continuation **5**

5 Potential Future Features **5**

6 Smallpond Programming Model **6**
- 6.1 Data Types . . . 6
- 6.2 Condition Codes . . . 6
- 6.3 Registers and Register Conventions . . . 7
  - 6.3.1 Register Zero . . . 7
  - 6.3.2 Argument Registers . . . 7
  - 6.3.3 Temporary Registers . . . 7
  - 6.3.4 Save Registers . . . 7
  - 6.3.5 2's Complement Pair Registers . . . 8
  - 6.3.6 Counter Registers . . . 8
  - 6.3.7 Heap Pointer Register . . . 8
  - 6.3.8 Frame Pointer Register . . . 8
  - 6.3.9 Stack Pointer Register . . . 8
  - 6.3.10 Link Registers . . . 8
  - 6.3.11 Program Counter Register . . . 8
  - 6.3.12 Current Program Status Register . . . 8
- 6.4 Instruction Types . . . 9

7 Smallpond Instruction Set **11**
- 7.1 A Type . . . 11
  - 7.1.1 Addition (ADD) . . . 11
  - 7.1.2 And (AND) . . . 12
  - 7.1.3 Arithmetic Shift Right (ASR) . . . 13
  - 7.1.4 Logical Shift Left (LSL) . . . 14
  - 7.1.5 Logical Shift Right (LSR) . . . 15
  - 7.1.6 Not And (NAND) . . . 16
  - 7.1.7 Not Or (NOR) . . . 17
  - 7.1.8 Not (NOT) . . . 18
  - 7.1.9 Or (OR) . . . 19
  - 7.1.10 Subtraction (SUB) . . . 20
  - 7.1.11 Exclusive Not Or (XNOR) . . . 21
  - 7.1.12 Exclusive Or (XOR) . . . 22
- 7.2 I Type . . . 23
  - 7.2.1 Immediate Addition (ADDI) . . . 23

7.3 Immediate AND (ANDI) . . . 24
7.3.1 Arithmetic Shift Right Immediate (ASRI) . . . 25
7.3.2 Load Word (LDR) . . . 26
7.3.3 Load Byte (LDRB) . . . 27
7.3.4 Load Byte Unsigned (LDRBU) . . . 28
7.3.5 Load Halfword (LDRH) . . . 29
7.3.6 Load Halfword Unsigned (LDRHU) . . . 30
7.3.7 Logical Shift Left Immediate (LSLI) . . . 31
7.3.8 Logical Shift Right Immediate (LSRI) . . . 32
7.3.9 Load Upper Immediate (LUI) . . . 33
7.3.10 Not AND Immediate (NANDI) . . . 34
7.3.11 Not OR Immediate (NORI) . . . 35
7.3.12 OR Immediate (ORI) . . . 36
7.3.13 Store Word (STR) . . . 37
7.3.14 Store Byte (STRB) . . . 38
7.3.15 Store Byte (STRH) . . . 39
7.3.16 Subtract Immediate (SUBI) . . . 40
7.3.17 Exclusive Not OR Immediate (XNORI) . . . 41
7.3.18 Exclusive OR Immediate (XORI) . . . 42
7.4 J Type . . . 43
7.4.1 Jump (J) . . . 43
7.5 B Type . . . 44
7.5.1 Branch (B) . . . 44
7.5.2 Branch and Link (BL) . . . 45
7.5.3 Branch and Return (BR) . . . 46

1 Introduction

Smallpond is a fully custom 32 bit RISC CPU designed by Zachary Salim, Dominique Hickson, Andrew Balys and Justin Charlong under the guidance of Dr. Kris Schindler as a part of CSE 490 Computer Architecture in the Fall of 2017. This architecture contains our favorite features from the MIPS 2000/3000 and ARM 7 TDMI processors as well as some new features including, but not limited to, multiple instruction types, 32 general purpose registers, condition codes for most instructions, a CPSR and many more.

2 Purpose

Smallpond was developed as a way to provide the Computer Architecture (CSE 490) and the Compilers (CSE 443) students a way to work together. This architecture will be the target for the compiler's students work. The hope is for this architecture to be used in many more courses. Eventually seeing a custom operating system (developed by the OS students), Computer Organization (CSE 341) students studying and writing assembly for Smallpond, Intro to Microprocessors (CSE 379) students writing large assembly programs and studying/discussing the features and possible improvements of Smallpond, and finally Computer Architecture (CSE 490) students implementing improvements via new revisions of the architecture.

3 Naming

You are probably wondering "Why Smallpond? That doesn't make any sense at all for an architecture name." There is a meaning behind the name and it will all make sense in a moment. To start, credit for the name goes to Jeremy Pettrot and Andrew Balys. They each came up with half of the name itself. The idea came from Intel's naming scheme for their Skylake, Kabylake and Coffelake CPU lines. All they did was change the first half of the word for each new generation. We wanted a name that could do the same thing. This is when Jeremy suggested "pond" because this is a RISC architecture (smaller than CISC) and a pond is smaller than a lake. When Andrew hear about Jeremy's suggestion he suggested "small" because it was our first revision and it could be expanded upon later i.e "Mediumpond" etc.

4 Continuation

Since I (Zack) own the repository on GitHub I will not be able to just pass it on to the next round of students. I would like to keep the working first revision of Smallpond protected in it's final state in that repository. My idea for future expansion and addition to the repository is to fork of the existing one as a new revision e.g "Smallpond Rev B". Small feature additions would only merit a revision increase (A to B, B to C), whereas a large (pipe-lining the architecture) could be a new generation ("Mediumpond").

5 Potential Future Features

Here are some of the potential features (in no particular order) we thought might be good additions to Smallpond in the future.

- Pipe-lining
- UART communication
- Interrupts
- Exceptions
- Timers
- Hardware debug peripheral (hardware breakpoints)
- Basic Cache using fast-block RAM on FPGA
- GPU co-processor (another board or on-chip)
- Superscalar
- Multi-Core CPU

6 Smallpond Programming Model

6.1 Data Types

Byte 8 bits
Halfword 16 Bits
Word 32 Bits

Note

- Unless specified as *unsigned* integer in the range 0 to 2^N-1, the N-bit value will be considered a signed integer in the range, -2^{N-1} to $+2^{N-1}$, using two's complement format.
- Load and store operations can transfer bytes, halfwords and words to and from memory, automatically zero-extending or sign-extending bytes or halfwords as they are loaded.
- Smallpond instructions are exactly one word (and are aligned on a four-byte boundary).
- All data operations , A type, are performed in word quantities

6.2 Condition Codes

Cond.	**Code**	**Meaning**	**cond. Flag State**
AL	0000	Always	Unconditional
EQ	0001	Equal	Z set
NE	0010	Not equal	Z clear
CA	0011	Carry set	C set
CC	0100	Carry clear	C clear
NG	0101	Negative	N set
PZ	0110	Positive or zero	N clear
VS	0111	Overflow set	V set
VC	1000	Overflow clear	V clear
HI	1001	Unsigned higher	C set and Z clear
LS	1010	Unsigned lower or same	C clear or Z set
GE	1011	Signed greater than or equal	N set and V set, or N clear and V clear (N==V)
LT	1100	Signed less than	N set and V clear, or N clear and V set (N!=V)
GT	1101	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z==0,N==V)
LE	1110	Signed less than or equal	Z set, or either N set and V clear, or N clear and V set (Z==1 or N!=V)

6.3 Registers and Register Conventions

There are a total of 32 registers in the Smallpond processor:

- 32 general purpose registers, with a program counter.
- All registers are 32 bits wide.

Register	**Use**	**Register**	**Use**
R0	0	R16	Saved
R1	Arg_0	R17	Saved
R2	Arg_1	R18	Saved
R3	Arg_2	R19	Pos/Neg
R4	Arg_3	R20	Neg/Pos
R5	Temp	R21	+ Counter
R6	Temp.	R22	- Counter
R7	Temp.	R23	HP
R8	Temp.	R24	FP
R9	Temp.	R25	SP
R10	Temp.	R26	LR_0
R11	Temp.	R27	LR_1
R12	Temp.	R28	LR_2
R13	Saved	R29	LR_3
R14	Saved	R30	PC
R15	Saved	R31	CPSR

6.3.1 Register Zero

Register zero (R0) is hardwired to zero (0x00000000). Writing to register zero will have no effect on its contents.

6.3.2 Argument Registers

Registers 1-4 (R1, R2, R3, R4) should be used to pass arguments into and out of routines.

6.3.3 Temporary Registers

Registers 5-12 (R5, R6, R7, R8, R9, R10, R11, R12) are temporary registers (scratch registers). They should be used for general calculations and for storing temporary results. Their contents need not be preserved across routine calls.

6.3.4 Save Registers

Registers 13-18 (R13, R14, R15, R16, R17, R18) are save registers which should be used to save results. These registers' values should be preserved across routine calls. If the values are modified in a routine their values should be returned before exiting the routine.

6.3.5 2's Complement Pair Registers

Registers 19 and 20 (R19, R20) are a positive/negative pair. Any value stored into register 19 will update register 20 with the 2's complement of register 20. Storing to register 20 will update register 20 with the 2's complement of register 20.

6.3.6 Counter Registers

Registers 21 and 22 (R21, R22) are counter registers. Writing a value to either register will write to both registers. Activating the counter through the 'C' flag in a command will increment register 21 by 1, and decrement register 22 by 1.

6.3.7 Heap Pointer Register

Register 23 (R23) is the heap pointer register. It should be maintained as a constant reference to the heap.

6.3.8 Frame Pointer Register

Register 24 (R24) is the frame pointer register. It should be maintained as a constant reference to the stack.

6.3.9 Stack Pointer Register

Register 25 (R25) is the stack pointer register. It should be maintained as a reference to the stack.

6.3.10 Link Registers

Link registers are meant to be used in the order 0 to 3 when storing and from 3 to 0 when restoring. This is only a software convention

6.3.11 Program Counter Register

Register 30 (R30) is the program counter. It contains a reference to the next instruction's address in memory.

6.3.12 Current Program Status Register

Unused	**N**	**Z**	**C**	**V**
31:4	3	2	1	0

Register 31 (R31) is the current program status register (CPSR). The CPSR contains information as to the status of the processor. It can be used to make decisions and is referred to by programs which make use of conditions. The 'S' flag in an instruction is used to set the CPSR and update its values based on the instruction's results..

6.4 Instruction Types

The Smallpond architecture has 4 instruction types: A type, I type, J type, and B type. The A type instructions utilize up to 2 input registers and output the result into a third register. The I type instructions perform on 1 input register with immediate data and output the result into a second register. The J type instruction is used in conjunction with an immediate to change the program counter. The B type instructions also use immediate data to change the program counter however they can do so conditionally. This section outlines the Smallpond architecture's instructions by instruction type. The four types are outlined below:

A type:

Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
31:26	25:21	20:16	15:11	10	9	8:4	3:0

Immediate type:

Inst.	**RD**	**RS1**	**Immediate**
31:26	25:21	20:16	15:0

Jump type:

Inst.	**Immediate**
31:26	25:0

Branch type:

Inst.	**C**	unused	**Immediate**	**Cond.**
31:26	25	24:20	19:4	3:0

SmallPond Reference sheet

Instruction Set

NAME	MNEMONIC	OP	$INST_{hex}$

A type

Add	*ADD*	RD = RS1 + RS2	0x01
And	*AND*	RD = RS1 & RS2	0x02
Airth Shift R	*ASR*	RD = RS1 >> RS2	0x03
Logical Shift L	*LSL*	RD = RS1 << RS2	0x04
Logical Shift R	*LSR*	RD = RS1 >> RS2	0x05
Not And	*NAND*	RD = ~(RS1 & RS2)	0x06
Not Or	*NOR*	RD = ~(RS1 | RS2)	0x07
Not	*NOT*	RD = ~ RS	0x08
Or	*OR*	RD = RS1 | RS2	0x09
Subtraction	*SUB*	RD = RS1 - RS2	0x0A
X Not Or	*XNOR*	RD = RS1 ⊙ RS2	0x0B
X Or	*XOR*	RD = RS1 ⊕ RS2	0x0C

I type

Addition w/ Imm	*ADDI*	RD = RS + #Imm	0x18
And w/ Imm	*ANDI*	RD = RS & #Imm	0x19
Airth Shift R Imm	*ASRI*	RD = RS << #Imm	0x1A
Load Word	*LDR*	RD = [RS + #Imm]	0x1B
Load Byte	*LDRB*	$RD_{7:0}$ = [RS + #Imm]	0x1C
Load B Unsigned	*LDRBU*	$RD_{7:0}$ = [RS + #UImm]	0x1D
Load Halfword	*LDRH*	$RD_{15:0}$ = [RS + #Imm]	0x1E
Load Halfword Unsigned	*LDRHU*	$RD_{15:0}$ = [RS + #UImm]	0x1F
Logical Shift L Imm	*LSLI*	RD = RS >> #Imm	0x20
Logical Shift R Imm	*LSRI*	RD = RS << #Imm	0x21
Load Upper Imm	*LUI*	$RD_{31:16}$ = [RS + #Imm]	0x22
Not And Imm	*NANDI*	RD = ~(RS & #Imm)	0x23
Not Or Imm	*NORI*	RD = ~(RS | #Imm)	0x24
Or Imm	*ORI*	RD = RS | #Imm	0x25
Store Word	*STR*	[RD] = RS + #Imm	0x26
Store Byte	*STRB*	[RD] = $(RS + \#Imm)_{7:0}$	0x27
Store Halfword	*STRH*	[RD] = $(RS + \#Imm)_{15:0}$	0x28
Subtract Imm	*SUBI*	RD = RS - #Imm	0x29
X Not Or Imm	*XNORI*	RD = RS ⊙ #Imm	0x2A
X Or Imm	*XORI*	RD = RS1 ⊕ #Imm	0x2B

J type

Jump	J	PC = PC + #Imm	0x3F

B type

Branch	*B*	PC = PC + #Imm	0x3C
Branch and Link	*BL*	PC = PC + #Imm	0x3D
Branch and Return	*BR*	PC = PC + #Imm	0x3E

INSTRUCTION FORMAT

A type:

Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	**Unused**	**Cond**
31:26	25:21	20:16	15:11	10	9	8:4	3:0

I type:

Inst.	**RD**	**RS1**	**Immediate**
31:26	25:21	20:16	15:0

J type:

Inst.	**Immediate**
31:26	25:0

B type:

Inst.	**C**	**Unused**	**Immediate**	**Cond.**
31:26	25	24:22	21:4	3:0

REGISTER NAME, NUMBER, USE

NAME	NUMB	USE
Zero	0	The constant value 0
Arg_#(0-3)	1 - 4	Argument
Tmp_#(0-7)	5 - 12	Temporary use registers
Svd_#(0-5)	13 - 18	Saved temporaries
Pos/Neg	19	Pair of 2's comp registers. Value is reflected across regs.
Neg/Pos	20	
+ Counter	21	Pair of reflected count registers.
- Counter	22	
HP	23	Heap pointer
FP	24	Frame pointer
LR_#(0-3)	25 – 29	Link registers used for routine addresses
PC	30	Program counter
CPSR	31	Current program status register

DATA ALIGNMENT

WORD			
HALFWORD		HALFWORD	
BYTE	BYTE	BYTE	BYTE
31 25	24 17	16 8	7 0

7 Smallpond Instruction Set

7.1 A Type

7.1.1 Addition (ADD)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
000001	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The ADD instruction adds the values stored in the two source registers (RS1 and RS2) and stores the result in the destination register RD.

Syntax

ADD{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

Adding the values in Rx and Ry and storing the result in Rz.

ADD Rz, Rx, Ry

7.1.2 And (AND)

31 26	25 21	20 16	15 11	10	9	8 4	3 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
000010	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The AND instruction executes a bitwise AND of the values stored in the two source registers (RS1 and RS2) and stores the result in the destination register RD.

Syntax

AND{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

Anding the values in Rx and Ry and storing the result in Rz.

AND Rz, Rx, Ry

7.1.3 Arithmetic Shift Right (ASR)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
000011	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The ASR instruction arithmetically right shifts the value stored in RS1 by the value stored in RS2 and stores the result into register RD.

Syntax

ASR{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To ASR a value in register Rx by a value in register Ry and store the result into register Rz.

ASR Rz, Rx, Ry

7.1.4 Logical Shift Left (LSL)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
000100	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The LSL instruction logically left shifts the value stored in RS1 by the value stored in RS2 and stores the result into register RD.

Syntax

LSL{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To LSL a value in register Rx by a value in register Ry and store the result into register Rz.

LSL Rz, Rx, Ry

7.1.5 Logical Shift Right (LSR)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
000101	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The LSR instruction logically right shifts the value stored in RS1 by the value stored in RS2 and stores the result into register RD.

Syntax

LSR{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To LSR a value in register Rx by a value in register Ry and store the result into register Rz.

LSR Rz, Rx, Ry

7.1.6 Not And (NAND)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
000110	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The NAND instruction executes a bitwise NAND of the value in RS1 and the value in RS2 and stores the result in register RD.

Syntax

NAND{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To NAND a value in register Rx with a value in register Ry and store the result into Rx.

NAND Rx, Rx, Ry

7.1.7 Not Or (NOR)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
000111	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The NOR instruction executes a bitwise NOR of the value in RS1 and the value in RS2 and stores the result in register RD.

Syntax

NOR{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To NOR a value in register Rx with a value in register Ry and store the result into Rx.

NOR Rx, Rx, Ry

7.1.8 Not (NOT)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
001000	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The NOT instruction executes a bitwise NOT of the value in RS1 and stores the result in register RD.

Syntax

NOT{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To NOT a value in register Rx and store the result into Rx.

NOT Rx, Rx

7.1.9 Or (OR)

31 26	25 21	20 16	15 11	10	9	8 4	3 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
001001	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The OR instruction executes a bitwise OR of the value in RS1 and the value in RS2 and stores the result in register RD.

Syntax

OR{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To OR a value in register Rx with a value in register Ry and store the result into Rx.

OR Rx, Rx, Ry

7.1.10 Subtraction (SUB)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
001010	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The instruction subtracts the value of RS2 from the value of RS1 and stores the result in register RD.

Syntax

{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To SUB a value in register Rx from register Ry and store the result into Rz.

SUB Rz, Ry, Rx

7.1.11 Exclusive Not Or (XNOR)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
001011	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The XNOR instruction executes a bitwise XNOR of the value in RS1 and the value in RS2 and stores the result in register RD.

Syntax

XNOR{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To XNOR a value in register Rx with a value in register Ry and store the result into Rx.

XNOR Rx, Rx, Ry

7.1.12 Exclusive Or (XOR)

31 – 26	25 – 21	20 – 16	15 – 11	10	9	8 – 4	3 – 0
Inst.	**RD**	**RS1**	**RS2**	**S**	**C**	Unused	**Cond.**
001100	25:21	20:16	15:11	10	9	8:4	3:0

Usage

The XOR instruction executes a bitwise XOR of the value in RS1 and the value in RS2 and stores the result in register RD.

Syntax

XOR{cond}{S}{C} RD, RS1, RS2

where:

{cond} Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

S Causes the S bit (bit[25]) to be set to a 1. This specified that the instruction will update the CPSR. If S is omitted then the S bit (bit[25]) is set to 0 and the CPSR is not updated.

C Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

RD Destination register of the instruction.

RS1 Source register 1 of the instruction.

RS2 Source register 2 of the instruction.

Example

To XOR a value in register Rx with a value in register Ry and store the result into Rx.

XOR Rx, Rx, Ry

7.2 I Type

7.2.1 Immediate Addition (ADDI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The ADDI instruction adds the value of RS to the value of a 16 bit immediate and stores the result in register RD.

Syntax

ADDI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To increment a value in register Rx by an immediate of 1.

ADDI Rx, Rx, #1

7.3 Immediate AND (ANDI)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The ANDI instruction is used to execute a bitwise AND with the immediate value and the lower 16 bits of register RS and stores the result into register RD.

Syntax

ANDI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To AND the value in register Rx with 0xF0 and store the result into Ry.

ANDI Ry, Rx, #0xF0

7.3.1 Arithmetic Shift Right Immediate (ASRI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The ASRI instruction is used to arithmetically right shift a register by an immediate value.

Syntax

ASRI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To ASR the value in register Rx by 2 and store the result into Ry.

ASRI Ry, Rx, #2

7.3.2 Load Word (LDR)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The LDR instruction is used to load a word from memory into a register RS. The address of the word that is fetched is calculated by adding the immediate value to the value of register RS

Syntax

LDR RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To LDR the word at the memory location of the value in register Rx with an offset of 4 and store the result into Ry.

LDR Ry, Rx, #4

7.3.3 Load Byte (LDRB)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The LDRB instruction is used to load a sign extended byte from memory into a register RS. The address of the byte that is fetched is calculated by adding the immediate value to the value of register RS.

Syntax

LDRB RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To LDRB the signed byte at the memory location of the value in register Rx with an offset of 4 and store the result into Ry.

LDRB Ry, Rx, #4

7.3.4 Load Byte Unsigned (LDRBU)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The LDRBU instruction is used to load an unsigned byte from memory into a register RS. The address of the byte that is fetched is calculated by adding the immediate value to the value of register RS.

Syntax

LDRBU RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To LDRBU the unsigned byte at the memory location of the value in register Rx with an offset of 4 and store the result into Ry.

LDRBU Ry, Rx, #4

7.3.5 Load Halfword (LDRH)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The LDRH instruction is used to load a sign extended halfword from memory into a register RS. The address of the halfword that is fetched is calculated by adding the immediate value to the value of register RS.

Syntax

LDRH RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To LDRH the signed halfword at the memory location of the value in register Rx with an offset of 4 and store the result into Ry.

LDRH Ry, Rx, #4

7.3.6 Load Halfword Unsigned (LDRHU)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The LDRHU instruction is used to load an unsigned halfword from memory into a register RS. The address of the halfword that is fetched is calculated by adding the immediate value to the value of register RS.

Syntax

LDRHU RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To LDRHU the unsigned halfword at the memory location of the value in register Rx with an offset of 4 and store the result into Ry.

LDRHU Ry, Rx, #4

7.3.7 Logical Shift Left Immediate (LSLI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The LSLI instruction is used to logically left shift a register by an immediate value.

Syntax

LSLI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To LSL the value in register Rx by 2 and store the result into Ry.

LSLI Ry, Rx, #2

7.3.8 Logical Shift Right Immediate (LSRI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The LSRI instruction is used to logically right shift a register by an immediate value.

Syntax

LSRI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To LSR the value in register Rx by 2 and store the result into Ry.

LSRI Ry, Rx, #2

7.3.9 Load Upper Immediate (LUI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**Unused**	**Immediate**
100010	25:21	20:16	15:0

Usage

The LUI instruction is used to put a 16 bit immediate into RD.

Syntax

LUI RD, #imm.

where:

RD — Destination register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

Put the unsigned 16 bit immediate in the upper half of Ry

LUI Ry, #4

7.3.10 Not AND Immediate (NANDI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The NANDI instruction is used to execute a bitwise NAND with the immediate value and the lower 16 bits of register RS and stores the result into register RD.

Syntax

NANDI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To NAND the value in register Rx with 0xF0 and store the result into Ry.

NANDI Ry, Rx, #0xF0

7.3.11 Not OR Immediate (NORI)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The NORI instruction is used to execute a bitwise NOR with the immediate value and the lower 16 bits of register RS and stores the result into register RD.

Syntax

NORI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To NOR the value in register Rx with 0xF0 and store the result into Ry.

NORI Ry, Rx, #0xF0

7.3.12 OR Immediate (ORI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The ORI instruction is used to execute a bitwise OR with the immediate value and the lower 16 bits of register RS and stores the result into register RD.

Syntax

ORI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To OR the value in register Rx with 0xF0 and store the result into Ry.

ORI Ry, Rx, #0xF0

7.3.13 Store Word (STR)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The STR instruction is used to store a word from a register RD into an address in memory. The address of the word in memory is calculated by adding the immediate value to the value of register RS.

Syntax

STR RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To STR the word in register Ry at the location of the value in register Rx with an offset of 4.

STR Ry, Rx, #4

7.3.14 Store Byte (STRB)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The STRB instruction is used to store the lowest byte from register RD into an address in memory. The address of the byte in memory is calculated by adding the immediate value to the value of register RS.

Syntax

STRB RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To STRB the least significant byte in register Ry at the location of the value in register Rx with an offset of 4.

STRB Ry, Rx, #4

7.3.15 Store Byte (STRH)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The STRH instruction is used to store the lowest byte from register RD into an address in memory. The address of the byte in memory is calculated by adding the immediate value to the value of register RS.

Syntax

STRH RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To STRH the least significant half word in register Ry at the location of the value in register Rx with an offset of 4.

STRH Ry, Rx, #4

7.3.16 Subtract Immediate (SUBI)

31 – 26	25 – 21	20 – 16	15 – 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The I instruction is used to perform traction with an immediate. The immediate value is subtracted from register RS and stored into register RD.

Syntax

SUBI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To tract register Rx by 4 and store the result back into Rx (decrement Rx by 4).

SUBI Ry, Rx, #4

7.3.17 Exclusive Not OR Immediate (XNORI)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The XNORI instruction is used to execute a bitwise XNOR with the immediate value and the lower 16 bits of register RS and stores the result into register RD.

Syntax

XNORI RD, RS1, #imm.

where:

RD Destination register of the instruction.

RS1 Source register of the instruction.

#imm. The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To XNOR the value in register Rx with 0xF0 and store the result into Ry.

XNORI Ry, Rx, #0xF0

7.3.18 Exclusive OR Immediate (XORI)

31 26	25 21	20 16	15 0
Inst.	**RD**	**RS**	**Immediate**
ifStrEq	25:21	20:16	15:0

Usage

The XORI instruction is used to execute a bitwise XOR with the immediate value and the lower 16 bits of register RS and stores the result into register RD.

Syntax

XORI RD, RS1, #imm.

where:

RD — Destination register of the instruction.

RS1 — Source register of the instruction.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

To XOR the value in register Rx with 0xF0 and store the result into Ry

XNORI Ry, Rx, #0xF0

7.4 J Type

7.4.1 Jump (J)

31 26 25 0

Inst.	Immediate
111111	25:0

Usage

The J instruction allows the program to relocate the program counter to a location above or below the instruction. It does so by adding the immediate value supplied to the program counter + 4. The immediate value is sign extended.

Syntax

J target_address

where:

#imm. The immediate value of the instruction, must be capable of being represented in 26 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

Example

J *label*

7.5 B Type

7.5.1 Branch (B)

31 26	25	24 22	19 4	3 0
Inst.	**C**	unused	**Immediate**	**Cond.**
111100	25	24:21	19:4	3:0

The branch instruction allows to conditionally relocate the program counter to a location above or below the instruction.

Syntax

B{cond}{C} target_address

where:

C — Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

{cond} — Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

Usage

To conditionally change the PC

Example

B *label*

7.5.2 Branch and Link (BL)

31 26	25	24 22	19 4	3 0
Inst.	**C**	unused	**Immediate**	**Cond.**
111101	25	24:21	19:4	3:0

The branch and link instruction causes a branch to a target address while updating the appropriate link register with the returning address + 4.

Syntax

BL{cond}{C} target_address

where:

C — Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

{cond} — Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

Usage

To conditionally change the PC

Example

BL

7.5.3 Branch and Return (BR)

31 26	25	24 22	19 4	3 0
Inst.	**C**	unused	**Immediate**	**Cond.**
111110	25	24:21	19:4	3:0

The branch and return instruction. This will return the PC to the PC + 4 from the branch location.

Syntax

BR{cond}{C} target_address

where:

C — Causes the C bit (bit[24]) to be set to a 1. This specified that the instruction will increment the counter. If C is omitted then the C bit (bit[24]) is set to 0 and the counter will not be incremented.

#imm. — The immediate value of the instruction, must be capable of being represented in 16 bits. The syntax to represent a decimal number is as follows: #decimal_number, i.e. #1 to represent a one. The syntax to represent a hexadecimal number is as follows: #0xhexadecimal_number, i.e. #0x3F2A to represent a hexadecimal 0x3F2A or #0xA to represent 0x000A.

{cond} — Is the condition under which the instruction is executed. To see a list of conditions see page . If {cond} is omitted the AL (always) condition is used.

Usage

To conditionally change the PC

Example

BR