Herky 6000 Processor

Quick Reference

Version 1.3

About

The Herky 6000 processor was designed for teaching computer architecture techniques and encoding. This is necessary do to the complex real-world encodings used by the Intel x86/64 processors. Even the simple and eloquent RISC encoding of the ARM series is too complex for beginning students.

As such, the following are the design considerations:

1. The encoding should be as easy to read as possible. This is accomplished by aligning each field to the nibble. As a result, when viewed in hexadecimal, each digit is one field.

2. The processor should mirror the basic instruction set of the Intel x86. Naturally, not all the features will be mirrored, but rather those what appear in CSC 35 assignments.This means the overall design must be use 2-operands with CISC-like memory access.

3. Show students how encodings often make use of bit-patterns to simplify circuity. Instructions and addressing mode codes are aligned as such. For example, the less significant bit of the addressing mode code indicates if a immeidate is stored.

4. Instructions are 24 each. Though, the encoding could allow instructions to be 1,2, or 3 bytes depending on the opcode pattern or addressing mode code.

In addition, the following was considered:

1. The processor should be extendable without requiring additional execution modes or instructions. This is why the LDU (load unsigned) and LDR (load signed) instructions exist. Data is always extended to the word size of the system. The processor should be able to evolve (for example 16-bit to 32-bit) and be able to run 16-bit instructions without any additional hardware.

Registers

The Herky 6000 contains a total of 16 general purpose registers. These are generically named r0 – r15. To retain some syntactic compatibility with the Intel x64 Processor, students may use the name of the equivalent register.

	x64 Name	x86 Name
r0	rax	eax
r1	rbx	ebx
r2	rcx	ecx
r3	rdx	edx

	x64 Name	x86 Name
r4	rsi	esi
r5	rdi	edi
r6		
r7		

	x64 Name
r8	r8
r9	r9
r10	r10
r11	r11

	x64 Name
r12	r12
r13	r13
r14	r14
r15	r15

The size of the registers is purposely not explicitly defined. The system has the ability to load values from memory and automatically sign-extend them to the entire size of the register. This is accomplished through the use of the Load Signed Instruction (LDR). Unsigned values, which will not be sign-extended, can be loaded using Load Unsigned (LDU).

Addressing Modes

The Herky 6000 supports a total of 8 addressing modes. Two of these are used for unary instructions with the remainder used by the standard binary instructions.

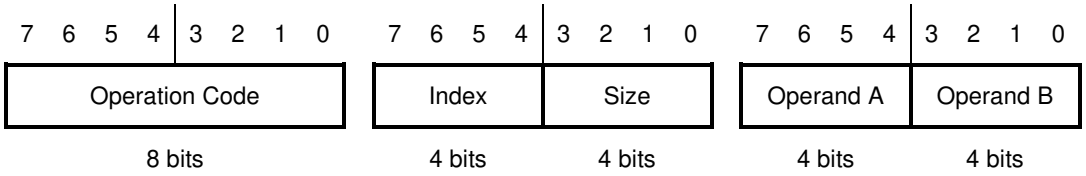
The chart below contains the 3-bit addressing mode codes, names, and a shorthand notation that will be used throughout this document.

Addressing Mode	Shorthand Notation
Unary Register	**reg**
Unary Immediate	**imm**
Register	**reg,reg**
Immediate	**reg,imm**
Register Indirect	**reg,[reg]**
Register Direct	**reg,[imm]**
Register Indirect Indexed	**reg,[reg + idx]**
Register Direct Indexed	**reg,[imm + idx]**

Instruction Format

Overview

Each instruction is 24-bit with most fields aligning to the nibble. Depending on the addressing mode, each instruction can also contain an optional immediate of 1, 2, 4, 8, etc… bytes.



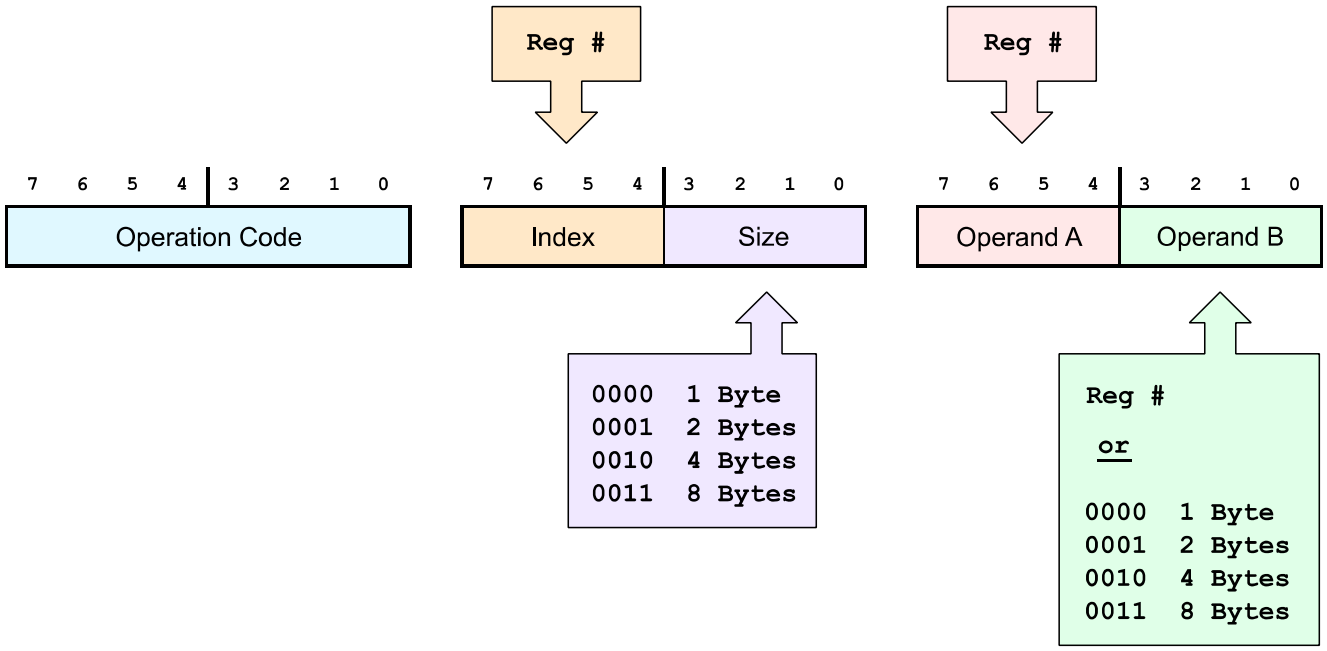
The Herky is a two-operand processor. Operand A will always designate a register number while Operand B can either designate a register number or the byte count of the immediate (depending, of course, on the addressing mode).

Immediates

If the addressing mode requires an immediate, Operand B will contain the size of the immediate rather than the value itself. The following table is used.

	2^n	
0000	1 byte (8-bit)	
0001	2 bytes (16-bit)	
0010	4 bytes (32-bit)	
0011	8 bytes (64-bit)	
.	.	
.	.	
.	.	
1111	32,768 bytes (262,144-bit)	Ludicrously large

Instruction Diagram



Examples

For example, the following is the encoding of Load Register (LDR) with a 1-byte and 2-byte immediate. For contrast, a register-register addressing mode is included.

Assembly	Mode	Encoded	Notes
`LDR r3, 15`	`reg,imm`	`B3 00 30 0F`	Operand B is 2^0 = 1 byte immediate
`LDR r4, 1947`	`reg,imm`	`B3 00 41 07 9B`	Operand B is 2^1 = 2 bytes..
`LDR r5, r7`	`reg,reg`	`B2 00 57`	Register transfer

Opcode Lookup Tables

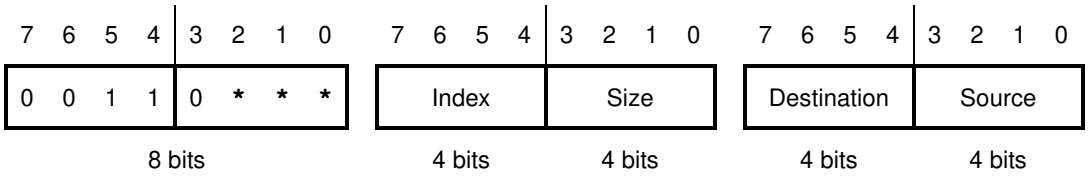
	opcode
NOP	1111 1111
RET	1111 1000
RTI	1111 1010
SYS	1111 1001

	reg	imm
CALL	0111 1000	0111 1001
FREE	0010 1000	
JMP	0110 1000	0110 1001
MARK	0011 1000	
NEG	0001 1000	
NOT	0000 1000	
POP	0100 1000	
PUSH	0101 1000	0101 1001

	reg, reg	reg, imm	reg,[reg]	reg,[imm]	reg, [reg + idx]	reg, [imm + idx]
ADD	0011 0010	0011 0011	0011 0100	0011 0101	0011 0110	0011 0111
AND	0000 0010	0000 0011	0000 0100	0000 0101	0000 0110	0000 0111
CMP	1100 0010	1100 0011	1100 0100	1100 0101	1100 0110	1100 0111
DIV	0110 0010	0110 0011	0110 0100	0110 0101	0110 0110	0110 0111
IN	1011 1010	1011 1011	1011 1100	1011 1101	1011 1110	1011 1111
LDR	1011 0010	1011 0011	1011 0100	1011 0101	1011 0110	1011 0111
LDU	1010 0010	1010 0011	1010 0100	1010 0101	1010 0110	1010 0111
LEA			1110 0100	1110 0101	1110 0110	1110 0111
MUL	0101 0010	0101 0011	0101 0100	0101 0101	0101 0110	0101 0111
OR	0001 0010	0001 0011	0001 0100	0001 0101	0001 0110	0001 0111
OUT	1100 1010	1100 1011	1100 1100	1100 1101	1100 1110	1100 1111
SET	1101 1010	1101 1011	1101 1100	1101 1101	1101 1110	1101 1111
STR			1101 0100	1101 0101	1101 0110	1101 0111
SUB	0100 0010	0100 0011	0100 0100	0100 0101	0100 0110	0100 0111
XOR	0010 0010	0010 0011	0010 0100	0010 0101	0010 0110	0010 0111

Instruction Set Details

ADD Add to Register



Operation

```
Destination ← Destination + Source
```

Syntax

```
ADD Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`0011 0100`
`imm`	invalid	`reg,[imm]`	`0011 0101`
`reg,reg`	`0011 0010`	`reg,[reg + idx]`	`0011 0110`
`reg,imm`	`0011 0011`	`reg,[imm + idx]`	`0011 0111`

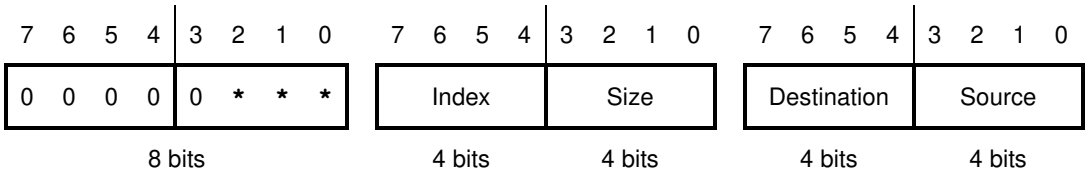
Description

Add the contents of the Source to the Destination register.

Examples

Assembly	Mode	Encoded	Notes
`ADD r4, 15`	`reg,imm`	`33 00 40 0F`	Source = 2^0 = 1 byte immediate
`ADD r3, r7`	`reg,reg`	`32 00 37`	Register transfer
`ADDq r4, [1947]`	`reg,[imm]`	`35 03 41 07 9B`	Source = 2^1 = 2 byte. Size = 2^3 = 8 bytes.
`ADD r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`36 93 45`	Indirect Indexed (size of 8 bytes)

AND Bit-wise And Register



Operation

```
Destination ← Destination AND Source
```

Syntax

```
AND Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`0000 0100`
`imm`	invalid	`reg,[imm]`	`0000 0101`
`reg,reg`	`0000 0010`	`reg,[reg + idx]`	`0000 0110`
`reg,imm`	`0000 0011`	`reg,[imm + idx]`	`0000 0111`

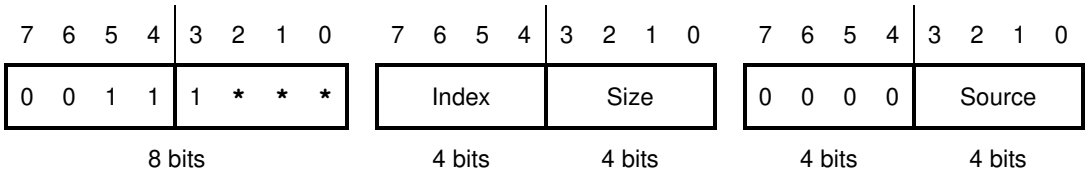
Description

Performs a Bitwise And with contents of the Source and the Destination register. The result is stored in the Destination register.

Examples

Assembly	Mode	Encoded	Notes
`AND r3, 15`	`reg,imm`	`03 00 30 0F`	Source = 2^0 = 1 byte immediate
`AND r3, r7`	`reg,reg`	`02 00 37`	Register transfer
`ANDq r4, [1947]`	`reg,[imm]`	`05 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`ANDq r2, [r6]`	`reg,[reg]`	`04 03 26`	Indirect. Size = 2^3 = 8 bytes.
`AND r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`06 93 45`	Indirect Indexed (size of 8 bytes)

CALL Call Subroutine



Operation

```
Stack.Push(IP)
IP ← Address
```

Syntax

`CALL Address` or `JSR Address`

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	`0111 1000`	`reg,[reg]`	invalid
`imm`	`0111 1001`	`reg,[imm]`	invalid
`reg,reg`	invalid	`reg,[reg + idx]`	invalid
`reg,imm`	invalid	`reg,[imm + idx]`	invalid

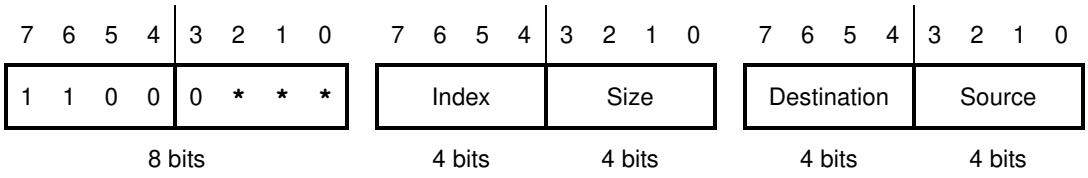
Description

Calls a subroutine. The current IP is pushed on the system stack and is, then, set to the Destination address.

Examples

Assembly	Mode	Encoded	Notes
`CALL 1947`	`imm`	`79 00 01 07 9B`	Bit 3 of the opcode is 1. This makes the lower nibble 1001.
`CALL r5`	`reg`	`78 00 05`	Bit 3 of the opcode is 1. This makes the lower nibble 1000. Source = 2^0 = 1 byte immediate

CMP Compare Register



Operation

```
a - b
```

Syntax

```
CMP a, b
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`1100 0100`
`imm`	invalid	`reg,[imm]`	`1100 0101`
`reg,reg`	`1100 0010`	`reg,[reg + idx]`	`1100 0110`
`reg,imm`	`1100 0011`	`reg,[imm + idx]`	`1100 0111`

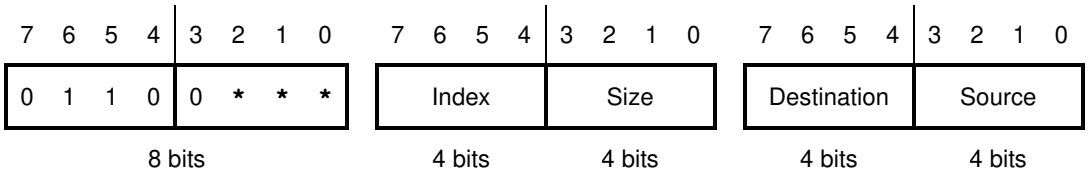
Description

Compares the two operands and sets the system flags. Internally, a subtraction takes place, but neither operand will be modified.

Examples

Assembly	Mode	Encoded	Notes
`CMP r3, 15`	`reg,imm`	`C3 00 30 0F`	Source = 2^0 = 1 byte immediate
`CMP r3, r7`	`reg,reg`	`C2 00 37`	Register transfer
`CMPq r4, [1947]`	`reg,[imm]`	`C5 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`CMPq r2, [r6]`	`reg,[reg]`	`C4 03 26`	Indirect. Size = 2^3 = 8 bytes.
`CMP r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`C6 93 45`	Indirect Indexed (size of 8 bytes)

DIV Divide Register



Operation

```
Destination ← Destination ÷ Source
```

Syntax

```
DIV Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`0110 0100`
`imm`	invalid	`reg,[imm]`	`0110 0101`
`reg,reg`	`0110 0010`	`reg,[reg + idx]`	`0110 0110`
`reg,imm`	`0110 0011`	`reg,[imm + idx]`	`0110 0111`

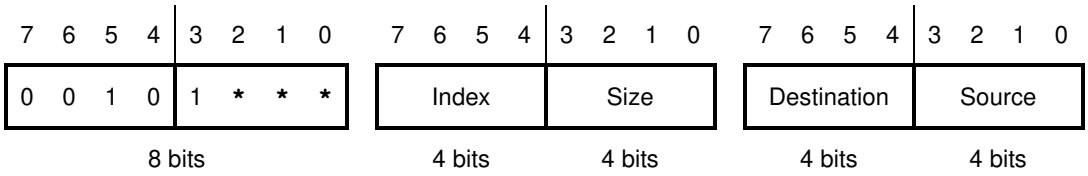
Description

Divides the contents of the Destination register by the Source.

Examples

Assembly	Mode	Encoded	Notes
`DIV r3, 15`	`reg,imm`	`63 00 30 0F`	Source = 2^0 = 1 byte immediate
`DIV r3, r7`	`reg,reg`	`62 00 37`	Register transfer
`DIVq r4, [1947]`	`reg,[imm]`	`65 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`DIVq r2, [r6]`	`reg,[reg]`	`64 03 26`	Indirect. Size = 2^3 = 8 bytes.
`DIV r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`66 93 45`	Indirect Indexed (size of 8 bytes)

FREE Free Stack Frame Mark



Operation

```
SP ← Register
Register ← Stack.Pop()
```

Syntax

```
FREE Register
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	`0010 1000`	`reg,[reg]`	invalid
`imm`	invalid	`reg,[imm]`	invalid
`reg,reg`	invalid	`reg,[reg + idx]`	invalid
`reg,imm`	invalid	`reg,[imm + idx]`	invalid

Description

Restores the stack to the mark stored in the Register. Any FREE instruction should be preceded by a MARK within the same subroutine.

Examples

Assembly	Mode	Encoded	Notes
`FREE r4`	`reg`	`28 00 40`	Bit 3 of the opcode is 1. This makes the lower nibble 1000.
`FREE r7`	`reg`	`28 00 70`	

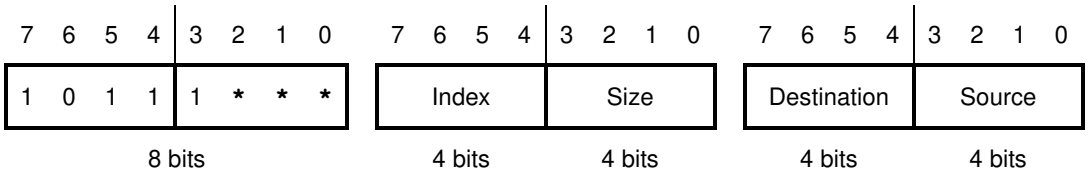
IDIV Divide

Please see DIV.

IMUL Multiply

Please see MUL.

IN Input from Port



Operation

```
Destination ← Port[Source]
```

Syntax

```
IN  Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`1011 1100`
`imm`	invalid	`reg,[imm]`	`1011 1101`
`reg,reg`	`1011 1010`	`reg,[reg + idx]`	`1011 1110`
`reg,imm`	`1011 1011`	`reg,[imm + idx]`	`1011 1111`

Description

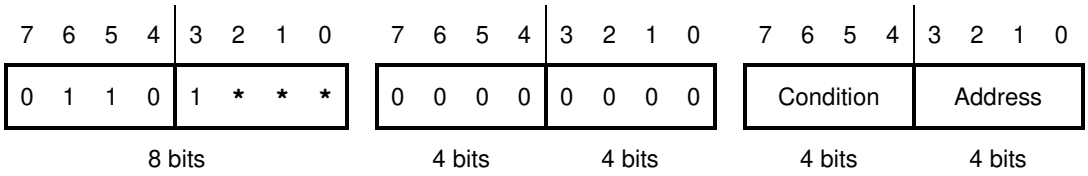
Reads a value for the port specified by Source and stores the result into the Destination register.

This instruction is only usable in supervisor mode.

Examples

Assembly	Mode	Encoded	Notes
`IN r3, 15`	`reg,imm`	`BB 00 30 0F`	Bit 3 of the opcode is 1. This makes the lower nibble 1011.
`IN r3, r7`	`reg,reg`	`BA 00 37`	Bit 3 of the opcode is 1. This makes the lower nibble 1010.
`IN r4, 1947`	`reg,imm`	`BB 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.

Jcc Jump if Condition is Met



Operation

```
IF Condition THEN IP ← Address
```

Syntax

`Jcc Address` Where "Jcc" is one of the mnemonics listed below

Operation Codes

Mode	Opcode
`reg`	`0110 1000`
`imm`	`0110 1001`
`reg,reg`	invalid
`reg,imm`	invalid

Mode	Opcode
`reg,[reg]`	invalid
`reg,[imm]`	invalid
`reg,[reg + idx]`	invalid
`reg,[imm + idx]`	invalid

Description

Transfers execution to the specified address if the conditions are met (please see the table below). Often, when using a conditional jump, a CMP (compare) instruction is used beforehand. This instruction incorporates 9 different assembly mnemonics.

Condition Table

Condition				Hex	Name	Assembly Mnemonic	Flags
0	0	0	0	0	Not Equal	`JNE`	`ZF = 0`
0	0	0	1	1	Equal	`JE`	`ZF = 1`
0	0	1	0	2	Greater Than	`JG`	`SF = 0, ZF = 0`
0	0	1	1	3	Greater Than or Equal	`JGE`	`SF = 0`
0	1	0	0	4	Less Than	`JL`	`SF = 1, ZF = 0`
0	1	0	1	5	Less Than or Equal	`JLE`	`SF = 1`
0	1	1	0	6	Not Overflow	`JNO`	`OF = 0`

0	1	1	1	7	Overflow	JO	OF = 1
1	1	1	1	F	Always	JMP	

Examples

Assembly	Mode	Encoded	Notes
JMP 1947	imm	69 00 F1 07 9B	Bit 3 of the opcode is 1. This makes the lower nibble 1001. 1111 = jump always
JE 1947	imm	69 00 11 07 9B	0001 = jump equal to.
JLE r7	reg	68 00 57	Bit 3 of the opcode is 1. This makes the lower nibble 1000. Source = 2^0 = 1 byte immediate

---

JE Jump If Equal To

Please see Jcc.

JGE Jump If Greater Than Or Equal

Please see Jcc.

JG Jump If Greater Than

Please see Jcc.

JLE Jump If Less Than Or Equal

Please see Jcc.

JL Jump If Less Than

Please see Jcc.

JMP Jump

Please see Jcc.

JNE Jump If Not Equal

Please see Jcc.

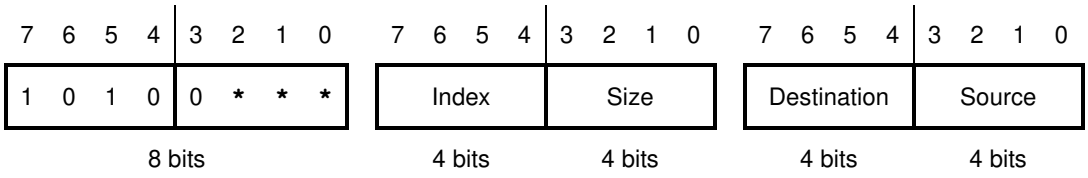
JNO Jump If Not Overflow

Please see Jcc.

JO Jump If Overflow

Please see Jcc.

LDU Load Register Unsigned



Operation

```
Destination ← Source
```

Syntax

```
LDU Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`1010 0100`
`imm`	invalid	`reg,[imm]`	`1010 0101`
`reg,reg`	`1010 0010`	`reg,[reg + idx]`	`1010 0110`
`reg,imm`	`1010 0011`	`reg,[imm + idx]`	`1010 0111`

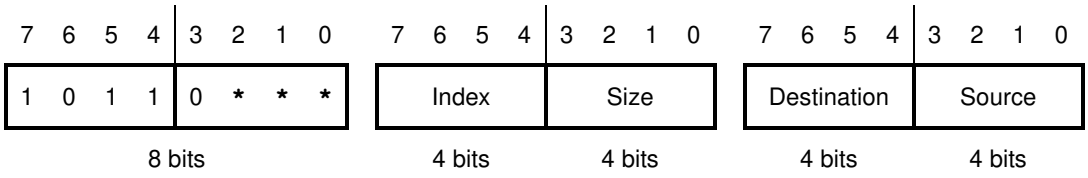
Description

Copy the contents of the Source to the Destination register. In this case, the data read will not be sign-extended into the register. Instead, the remaining bits will be set to zero.

Examples

Assembly	Mode	Encoded	Notes
`LDU r3, 15`	`reg,imm`	`A3 00 30 0F`	Source = 2^0 = 1 byte immediate
`LDU r3, r7`	`reg,reg`	`A2 00 37`	Register transfer
`LDUq r4, [1947]`	`reg,[imm]`	`A5 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`LDUq r2, [r6]`	`reg,[reg]`	`A4 03 26`	Indirect. Size = 2^3 = 8 bytes.
`LDU r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`A6 93 45`	Indirect Indexed (size of 8 bytes)

LDR Load Register Signed



Operation

```
Destination ← Source
```

Syntax

```
LDR Destination, Source
```
or
```
MOV Destination, Source
```

Operation Codes

Mode	Opcode
reg	invalid
imm	invalid
reg,reg	1011 0010
reg,imm	1011 0011

Mode	Opcode
reg,[reg]	1011 0100
reg,[imm]	1011 0101
reg,[reg + idx]	1011 0110
reg,[imm + idx]	1011 0111

Description

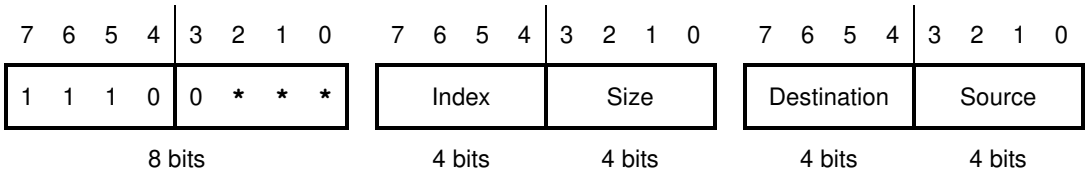
Copy the contents of the Source to the Destination register. In this case, the data read will be sign-extended into the register. This means that the register will contain a valid negative or positive field regardless of the number of bytes read.

The Intel-syntax may be used. Note that the data is *copied* and the Source is unchanged.

Examples

Assembly	Mode	Encoded	Notes
LDR r3, 15	reg,imm	B3 00 30 0F	Source = 2^0 = 1 byte immediate
LDR r3, r7	reg,reg	B2 00 37	Register transfer
LDRq r4, [1947]	reg,[imm]	B5 03 41 07 9B	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
LDRq r2, [r6]	reg,[reg]	B4 03 26	Indirect. Size = 2^3 = 8 bytes.
LDR r4, [r5 + r9 * 8]	reg,[reg + idx]	B6 93 45	Indirect Indexed (Size = 2^3 = 8 bytes)
LDR r4, [1947 + r9 * 8]	reg,[imm + idx]	B7 93 41 07 9B	Direct Indexed (Size = 2^3 = 8 bytes)

LEA Load Effective Address



Operation

```
Destination ← Computed Address of Source
```

Syntax

```
LEA Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`1110 0100`
`imm`	invalid	`reg,[imm]`	`1110 0101`
`reg,reg`	invalid	`reg,[reg + idx]`	`1110 0110`
`reg,imm`	invalid	`reg,[imm + idx]`	`1110 0111`

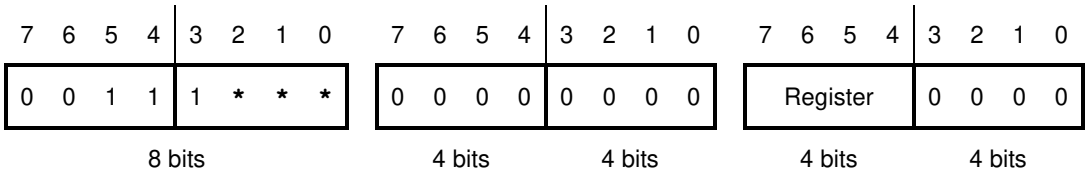
Description

Calculations the effective address as though memory would be accessed. However, rather than read data from memory, the effective addressed is stored into the Destination register.

Examples

Assembly	Mode	Encoded	Notes
`LEA r4, [1947]`	`reg,[imm]`	`E5 00 41 07 9B`	Source = 2^1 = 2 bytes.
`LEA r2, [r6]`	`reg,[reg]`	`E4 00 26`	Indirect. Size = 2^3 = 8 bytes.
`LEA r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`E6 93 45`	Indirect Indexed (2^3 = 8 bytes)

MARK Mark Stack-Frame Position



Operation

```
Stack.Push(Register)
Register ← SP
```

Syntax

```
MARK Register
```

Operation Codes

Mode	Opcode
`reg`	`0011 1000`
`imm`	invalid
`reg,reg`	invalid
`reg,imm`	invalid

Mode	Opcode
`reg,[reg]`	invalid
`reg,[imm]`	invalid
`reg,[reg + idx]`	invalid
`reg,[imm + idx]`	invalid

Description

Saves the current value of the register and then assigns the register the value of stack pointer (SP). This is used primarily in stack frames for both parameters and local variables. Any MARK instruction should be followed by a FREE within the same subroutine.

Examples

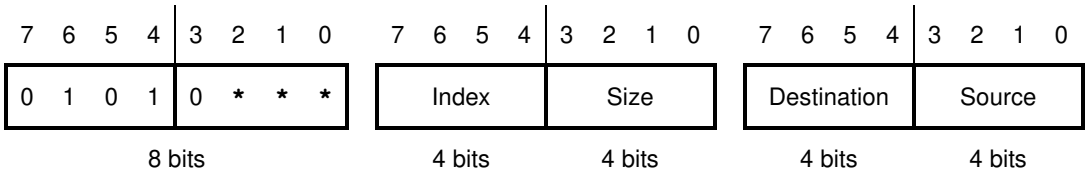
Assembly	Mode	Encoded	Notes
`MARK r4`	`reg`	`38 00 40`	Bit 3 of the opcode is 1. This makes the lower nibble 1000.
`MARK r7`	`reg`	`38 00 70`	

MOV Move

If you are storing a value into memory, please see **STR**.

Otherwise, please see **LDR**.

MUL Multiply Register



Operation

```
Destination ← Destination × Source
```

Syntax

```
MUL Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`0101 0100`
`imm`	invalid	`reg,[imm]`	`0101 0101`
`reg,reg`	`0101 0010`	`reg,[reg + idx]`	`0101 0110`
`reg,imm`	`0101 0011`	`reg,[imm + idx]`	`0101 0111`

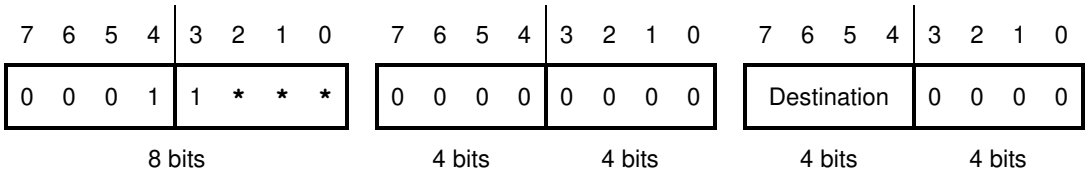
Description

Multiplies the contents of the Destination register by the Source.

Examples

Assembly	Mode	Encoded	Notes
`MUL r3, 15`	`reg,imm`	`53 00 30 0F`	Source = 2^0 = 1 byte immediate
`MUL r3, r7`	`reg,reg`	`52 00 37`	Register transfer
`MULq r4, [1947]`	`reg,[imm]`	`55 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`MULq r2, [r6]`	`reg,[reg]`	`54 03 26`	Indirect. Size = 2^3 = 8 bytes.
`MUL r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`56 93 45`	Indirect Indexed (size of 8 bytes)

NEG 2's Complement Register



Operation

```
Destination ← - Destination
```

Syntax

```
NEG Destination
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	`0001 1000`	`reg,[reg]`	invalid
`imm`	invalid	`reg,[imm]`	invalid
`reg,reg`	invalid	`reg,[reg + idx]`	invalid
`reg,imm`	invalid	`reg,[imm + idx]`	invalid

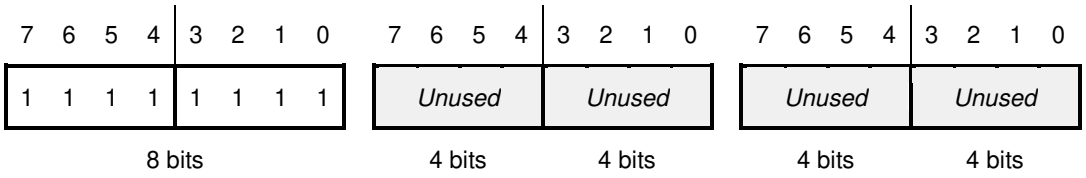
Description

Negates the value source in the Destination Register using 2's complement.

Examples

Assembly	Mode	Encoded	Notes
`NEG r4`	`reg`	`18 00 40`	Bit 3 of the opcode is 1. This makes the lower nibble 1000.
`NEG r7`	`reg`	`18 00 70`	

NOP No Operation



Operation

```
None – ignored by processor
```

Syntax

```
NOP
```

Operation Codes

Opcode

```
1111 1111
```

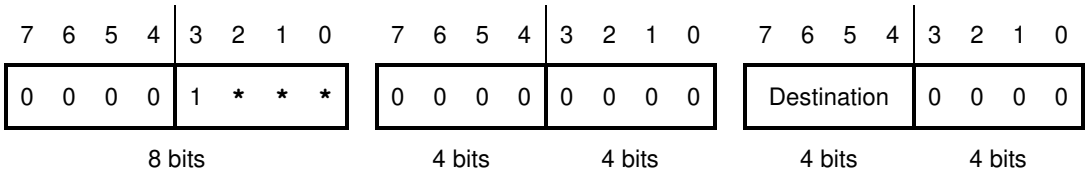
Description

No operation is performed, and the instruction is ignored by the processor.

Examples

Assembly	Mode	Encoded
NOP	None	FF 00 00

NOT Bit-wise Not Register



Operation

```
Destination ← NOT Destination
```

Syntax

```
NOT Destination
```

Operation Codes

Mode	Opcode
`reg`	`0000 1000`
`imm`	invalid
`reg,reg`	invalid
`reg,imm`	invalid

Mode	Opcode
`reg,[reg]`	invalid
`reg,[imm]`	invalid
`reg,[reg + idx]`	invalid
`reg,[imm + idx]`	invalid

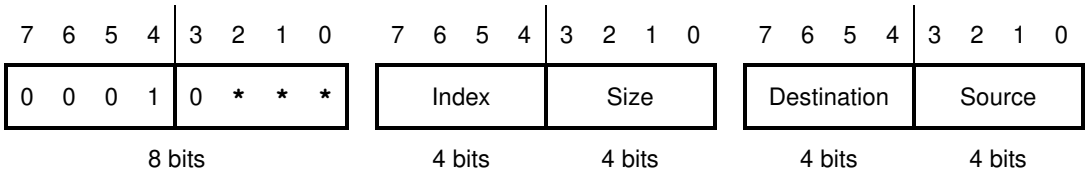
Description

Performs a Bitwise Not with contents of the Destination register.

Examples

Assembly	Mode	Encoded	Notes
`NOT r4`	`reg`	`08 00 40`	Bit 3 of the opcode is 1. This makes the lower nibble 1000.
`NOT r7`	`reg`	`08 00 70`	

OR Bit-wise Or Register



Operation

```
Destination ← Destination OR Source
```

Syntax

```
OR Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`0001 0100`
`imm`	invalid	`reg,[imm]`	`0001 0101`
`reg,reg`	`0001 0010`	`reg,[reg + idx]`	`0001 0110`
`reg,imm`	`0001 0011`	`reg,[imm + idx]`	`0001 0111`

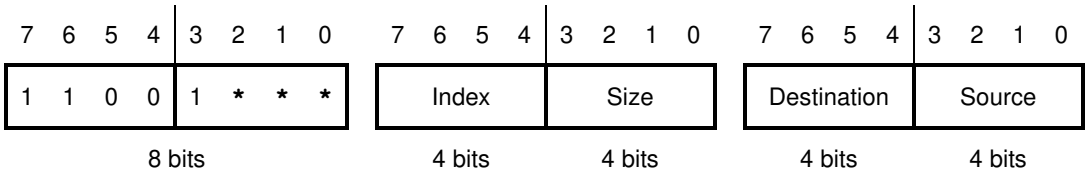
Description

Performs a Bitwise Or with contents of the Source and the Destination register. The result is stored in the Destination register.

Examples

Assembly	Mode	Encoded	Notes
`OR r3, 15`	`reg,imm`	`13 00 30 0F`	Source = 2^0 = 1 byte immediate
`OR r3, r7`	`reg,reg`	`12 00 37`	Register transfer
`ORq r4, [1947]`	`reg,[imm]`	`15 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`ORq r2, [r6]`	`reg,[reg]`	`14 03 26`	Indirect. Size = 2^3 = 8 bytes.
`OR r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`16 93 45`	Indirect Indexed (size of 8 bytes)

OUT Output to Port



Operation

```
Port[Destination] ← Source
```

Syntax

```
OUT Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`1100 1100`
`imm`	invalid	`reg,[imm]`	`1100 1101`
`reg,reg`	`1100 1010`	`reg,[reg + idx]`	`1100 1110`
`reg,imm`	`1100 1011`	`reg,[imm + idx]`	`1100 1111`

Description

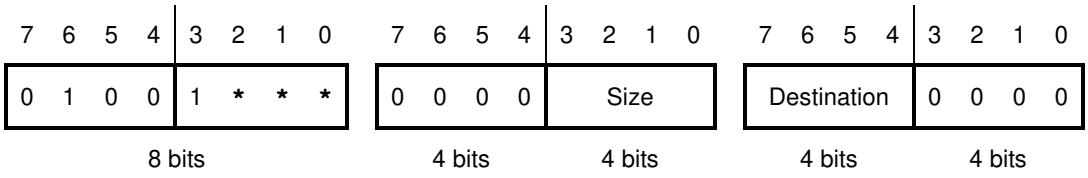
Writes the Source to the port specified by the contents of the Destination register.

This instruction is only usable in supervisor mode.

Examples

Assembly	Mode	Encoded	Notes
`OUT r3, 15`	`reg,imm`	`CB 00 30 0F`	Bit 3 of the opcode is 1. This makes the lower nibble 1011.
`OUT r3, r7`	`reg,reg`	`CA 00 37`	Bit 3 of the opcode is 1. This makes the lower nibble 1010.
`OUT r4, 1947`	`reg,imm`	`CB 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.

POP Pop Stack into Register



Operation

```
Destination ← Stack.Pop()
```

Syntax

```
POP Destination
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	`0100 1000`	`reg,[reg]`	invalid
`imm`	invalid	`reg,[imm]`	invalid
`reg,reg`	invalid	`reg,[reg + idx]`	invalid
`reg,imm`	invalid	`reg,[imm + idx]`	invalid

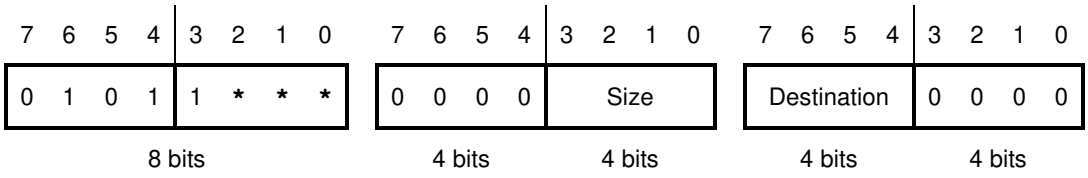
Description

Pops a value from the System Stack and stores it into the Destination Register.

Examples

Assembly	Mode	Encoded	Notes
`POP r4`	`reg`	`48 00 40`	Bit 3 of the opcode is 1. This makes the lower nibble 1000. The Size nibble is 0, so 1 byte is used
`POPq r7`	`reg`	`48 03 70`	Size = 2^3 = 8 bytes.

PUSH Push onto Stack



Operation

```
Stack.Push(Source)
```

Syntax

```
PUSH Source
```

Operation Codes

Mode	Opcode
reg	0101 1000
imm	0101 1001
reg,reg	invalid
reg,imm	invalid

Mode	Opcode
reg,[reg]	invalid
reg,[imm]	invalid
reg,[reg + idx]	invalid
reg,[imm + idx]	invalid

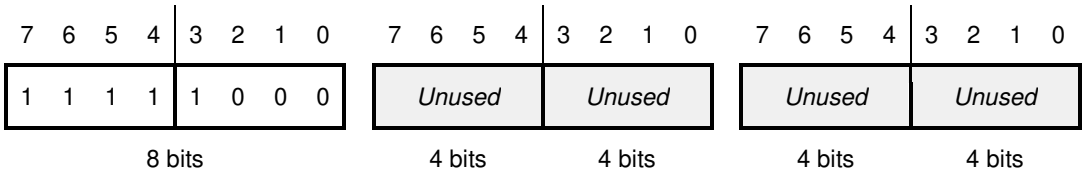
Description

Pushes the contents of the Source register onto the system stack.

Examples

Assembly	Mode	Encoded	Notes
PUSH r4	reg	58 00 04	Bit 3 of the opcode is 1. This makes the lower nibble 1000.
PUSHq r7	reg	58 03 07	Size = 2^3 = 8 bytes. A 64-bit integer is pushed.
PUSH 15	imm	59 00 00 0F	Bit 3 of the opcode is 1. This makes the lower nibble 1001. Source = 2^0 = 1 byte immediate
PUSHq 1947	imm	59 03 01 07 9B	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.

RET Return from Subroutine



Operation

```
IP ← Stack.Pop()
```

Syntax

```
RET
```

Operation Codes

Opcode
1111 1000

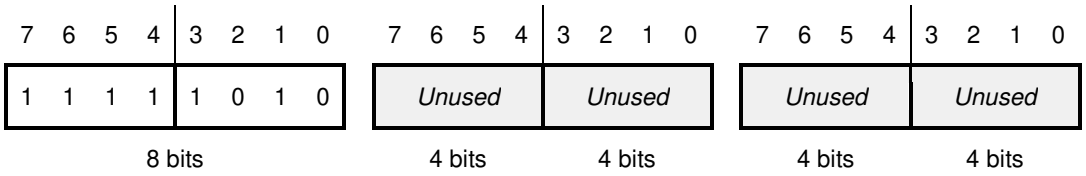
Description

Pops an address off the stack and replaces the IP. This returns the program from a subroutine previously started by CALL.

Examples

Assembly	Mode	Encoded
RET	None	F8 00 00

RTI Return from Interrupt



Operation

```
IP ← Stack.Pop()
AllowInterrupts = true
```

Syntax

```
RTI
```

Operation Codes

Opcode
1111 1010

Description

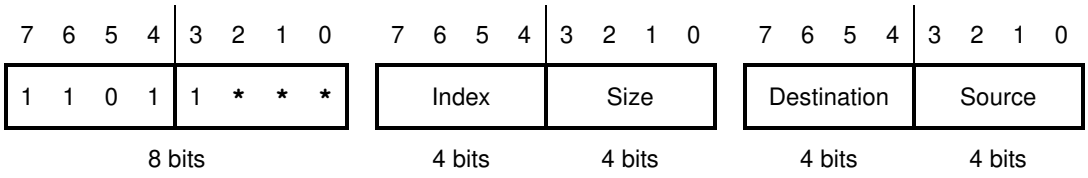
Pops a system word off the stack and replaces the IP. This returns the program from a subroutine previously started by interrupt. The system also reenables interrupts (which are disabled during an interrupt).

This instruction is only usable in supervisor mode.

Examples

Assembly	Mode	Encoded
RTI	None	FA 00 00

SET Set System Value



Operation

```
Setting[Key] ← Value
```

Syntax

```
SET Setting, Value
```

Operation Codes

Mode	Opcode
reg	invalid
imm	invalid
reg,reg	1101 1010
reg,imm	1101 1011

Mode	Opcode
reg,[reg]	1101 1100
reg,[imm]	1101 1101
reg,[reg + idx]	1101 1110
reg,[imm + idx]	1101 1111

Description

Sets a system setting specified in the Key register to the Value. Bit masks will be used to differentiate different system settings. These settings will include:

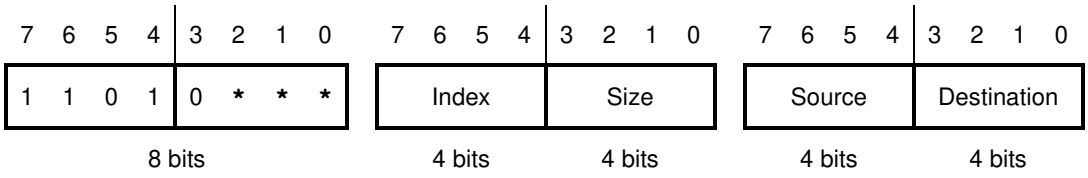
- Special registers
- Addresses in the Vector Table
- etc…

This instruction is only usable in supervisor mode.

Examples

Assembly	Mode	Encoded	Notes
SET r3, 15	reg,imm	D9 00 30 0F	Bit 3 of the opcode is 1. This makes the lower nibble 1001.
SET r3, r7	reg,reg	DA 00 37	Bit 3 of the opcode is 1. This makes the lower nibble 1010.
SETq r4, 1947	reg,imm	D9 03 41 07 9B	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.

STR Store Register



Operation

```
Destination ← Source
```

Syntax

```
STR Destination, Source
```
or
```
MOV Destination, Source
```

Operation Codes

Mode	Opcode
`reg`	invalid
`imm`	invalid
`reg,reg`	invalid
`reg,imm`	invalid

Mode	Opcode
`reg,[reg]`	`1101 0100`
`reg,[imm]`	`1101 0101`
`reg,[reg + idx]`	`1101 0110`
`reg,[imm + idx]`	`1101 0111`

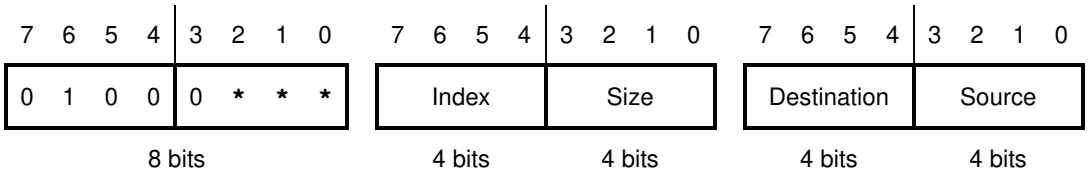
Description

Stores the Source register to the memory specified by the Destination. The Intel-syntax may be used. Note that the data is *copied* and the Source is unchanged.

Examples

Assembly	Mode	Encoded	Notes
`STRq r4, [1947]`	`reg,[imm]`	`D5 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`STRq r2, [r6]`	`reg,[reg]`	`D4 03 26`	Indirect. Size = 2^3 = 8 bytes.
`STR r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`D6 93 45`	Indirect Indexed (size of 8 bytes)
`STR r4, [1947 + r9 * 8]`	`reg,[imm + idx]`	`D7 93 41 07 9B`	Indirect Indexed (size of 8 bytes)

SUB Subtract from Register



Operation

```
Destination ← Destination – Source
```

Syntax

```
SUB Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`0100 0100`
`imm`	invalid	`reg,[imm]`	`0100 0101`
`reg,reg`	`0100 0010`	`reg,[reg + idx]`	`0100 0110`
`reg,imm`	`0100 0011`	`reg,[imm + idx]`	`0100 0111`

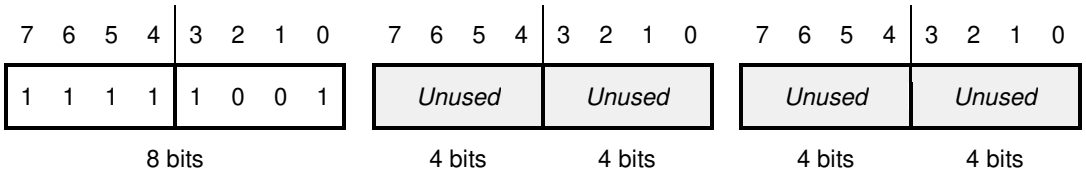
Description

Subtract the contents of the Source from the Destination register.

Examples

Assembly	Mode	Encoded	Notes
`SUB r3, 15`	`reg,imm`	`43 00 30 0F`	Source = 2^0 = 1 byte immediate
`SUB r3, r7`	`reg,reg`	`42 00 37`	Register transfer
`SUBq r4, [1947]`	`reg,[imm]`	`45 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`SUBq r2, [r6]`	`reg,[reg]`	`44 03 26`	Indirect. Size = 2^3 = 8 bytes.
`SUB r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`46 93 45`	Indirect Indexed (size of 8 bytes)

SYS Application System Call



Operation

```
Stack.Push(IP)
IP ← VectorTable[ApplicationIndex]
```

Syntax

```
SYS
```

Operation Codes

Opcode
1111 1001

Description

Calls the Vector Table entry used for applications. This is equivalent to the SYSCALL instruction in the x64 Processor.

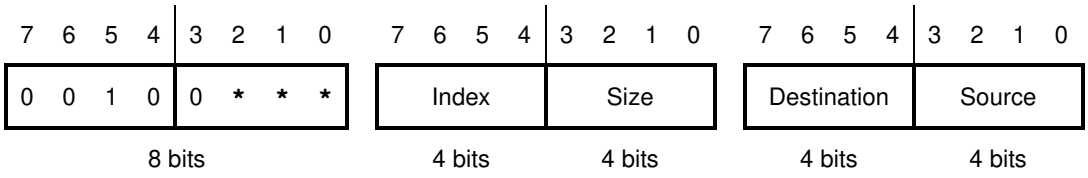
Examples

Assembly	Mode	Encoded
SYS	None	F9 00 00

SYSCALL Application System Call

Please see SYS.

XOR Bit-wise Exclusive-Or Register



Operation

```
Destination ← Destination XOR Source
```

Syntax

```
XOR Destination, Source
```

Operation Codes

Mode	Opcode	Mode	Opcode
`reg`	invalid	`reg,[reg]`	`0010 0100`
`imm`	invalid	`reg,[imm]`	`0010 0101`
`reg,reg`	`0010 0010`	`reg,[reg + idx]`	`0010 0110`
`reg,imm`	`0010 0011`	`reg,[imm + idx]`	`0010 0111`

Description

Performs a Bitwise Exclusive-Or with contents of the Source and the Destination register. The result is stored in the Destination register.

Examples

Assembly	Mode	Encoded	Notes
`XOR r3, 15`	`reg,imm`	`23 00 30 0F`	Source = 2^0 = 1 byte immediate
`XOR r3, r7`	`reg,reg`	`22 00 37`	Register transfer
`XORq r4, [1947]`	`reg,[imm]`	`25 03 41 07 9B`	Source = 2^1 = 2 bytes. Size = 2^3 = 8 bytes.
`XORq r2, [r6]`	`reg,[reg]`	`24 03 26`	Indirect. Size = 2^3 = 8 bytes.
`XOR r4, [r5 + r9 * 8]`	`reg,[reg + idx]`	`26 93 45`	Indirect Indexed (size of 8 bytes)