The RAT Assembler Manual

Version: 2.00

Table of Contents

Table of Contents ..... - 1 -
The RAT Assembler ..... - 2 -
RAT Assembler Overview ..... - 2 -
RAT Assembler Memory Issues ..... - 2 -
Memory Segmentation ..... - 2 -
Start-up Code ..... - 2 -
RATASM File Generation ..... - 3 -
xxx.asl ..... - 3 -
xxx.err ..... - 3 -
prog_rom.vhd ..... - 3 -
xxx.dbg ..... - 3 -
Assembly Language Overview ..... - 4 -
RAT Assembler Comments ..... - 4 -
RAT Assembler Labels ..... - 4 -
RAT Assembler Directives ..... - 4 -
Directive: .ORG ..... - 5 -
Segment Directives: .CSEG and .DSEG ..... - 6 -
Directive: .CSEG ..... - 6 -
Directive: .DSEG ..... - 6 -
Directive: .DB ..... - 6 -
Directive: .BYTE ..... - 7 -
Directive: .EQU ..... - 7 -
RAT Preferred Program Structure ..... - 9 -
RAT Sample Style File ..... - 11 -
RAT Sample List File Output ..... - 13 -
The RAT Instruction Set ..... - 15 -
RAT Assembly Instructions by “Group” ..... - 15 -
Logical Group ..... - 15 -
Arithmetic Group ..... - 15 -
Flow Control Group ..... - 15 -
Shift and Rotate Group ..... - 16 -
Input/Output Group ..... - 16 -
Data Control Group ..... - 16 -
Memory Group ..... - 16 -
RAT Assembly Instructions by Type ..... - 17 -
Instruction Type: Reg/Reg ..... - 18 -
Instruction Type: Reg/Imm ..... - 20 -
Instruction Type: Imm ..... - 22 -
Instruction Type: Reg ..... - 23 -
Instruction Type: None ..... - 25 -
RAT Assembly Instructions Type Listing ..... - 26 -
Detailed RAT Assembly Instruction Description ..... - 27 -
Revision History: ..... - 63 -

The RAT Assembler

RAT Assembler Overview

The RAT assembler is responsible for generating machine code for the RAT instruction set. Kianoosh Salami and Bryan Mealy initially defined the RAT instruction set architecture in an attempt to generate a meaningful academic experience for CPE 233 students. Jeff Gerfen made subsequent modifications to the RAT assembler.

The RAT assembler, *ratasm*, was written in a CYGWIN environment and is based on standard scanning (FLEX) and parsing (YACC) tools. The ratasm assembler is a two-pass assembler; the first pass is dedicated primarily to locating various labels and assigning them appropriate values; the second pass handles opcode assignment of the assembly instructions and inserts startup code as required.

RAT Assembler Memory Issues

The RAT architecture comprises of the many modules including various memories, Program Counter, I/O, Interrupts, and a control unit. The RAT assembler has direct involvement with two types of memory in the RAT architecture: the program memory and the scratch RAM.

Memory Segmentation

Computer hardware is generally comprised of different memory modules, each serving a distinct purpose. Software items such as assemblers generally have options to configure these memories prior to actual program execution. The RAT assembler contains commands referred as “directives” that allow programmers to configure these memories with a modest set of controls.

The programmer’s view of the RAT microcontroller models the two memory types as program memory and scratch memory. The RAT assembler allows the programmer several configuration options for these memories. The RAT assembler these memory types as one memory that is divided into to segments: the code segment and the data segment. The code segment refers to instruction memory while the data segment refers to scratch memory. As a result, the programmer must explicitly specify which segment a particular instruction or directive belongs to using the appropriate assembler directives. The .DSEG and .CSEG assembler directives specify the data segment and code segment, respectively. Usage details for these directives appears in a later section.

Start-up Code

The .BYTE assembler directive allows programmers to name and initialize references to memory locations in scratch RAM. Use of the .BYTE directive requires programmers to state initial values for the specified memory; these initial values must be written to scratch RAM using RAT assembly instructions.

As a feature of the RAT assembler, the RAT assembler automatically generates the code that initializes the associated locations in scratch RAM. This code is referred to as “start-up” code. Using the .BYTE directive requires that the program arrange program memory such that the program code starts at a location in program memory that leaves adequate space to store the start-up code.

RATASM File Generation

The RAT assembler, *ratasm,* is specifically designed for the beginning assembly language programmer in mind. As a result, ratasm includes in-depth error checking and report generatin. Running ratasm generates several files which can be useful in program understanding and/or debugging. This section lists the files generated by ratasm. For all the files listed below with a "xxx" filename, ratasm replaced the "XXX" with the filename (not including the extension) given to the program that ratasm attempted to assemble.

xxx.asl

This is the output list file associated with the assembly language program that was assembled. This file is generated automatically when program is assembled and is not dependent on whether assembled had errors or not. In the case where the program has errors, the xxx.asl files lists those errors including a helpful comment in most cases.

The output listing file likewise provides all associated instruction opcodes, assembler directives, data memory information, symbol table listing, and other useful information. If your program contains errors, the errors are listed in this file as well as the error file. If there's any information not provided in the output listing file, then you probably don't need it.

xxx.err

If the program you are attempting to compile contains errors, the list of errors and error messages is printed in the xxx.err file. There are basically two types of errors in the ratasm assembler: 1) errors that the assembler can pinpoint, and 2) errors that the assembler knows is an error but cannot specify what the exact error is. In the first case, ratasm prints a relatively helpful error message to help the programmer find and fix the error. In the second case, however, ratasm provides little or no information about the error other than the line in the program where ratasm assumes the error occurred.

prog_rom.vhd

This file is a VHDL model of a ROM object containing the machine code for the successfully assembled program. This file is only generated if the program successfully assembles. If the program does not assemble correctly, ratasm deletes any existing prog_rom.vhd file in the directory where ratasm was executed.

xxx.dbg

This is a specially formatted file containing information used by the debugger/simulator. If you're not the debugger, then you won't be needing to see the information contained in this file.

Assembly Language Overview

Assembly language programs have four distinct parts: 1) comments, 2) labels, 3) assembler directives, and 4) assembly language instructions. The ratasm enforces theses guidelines in an effort to make the resulting assembly code more understandable to the human reader.

RAT Assembler Comments

The RAT assembler uses the semi-colon character (;) in order to indicate a comment. The comment character can appear in any column of source code text. All text that follows the comment character on any source code line is considered to be the comment and is thus ignored by the ratasm.

Comments are used to describe *what* is happening and *how* something is being done if it is not patently obvious from the source code. The primary purpose of comments is to further the readability and understandability of the assembly code. The text associated with comments is ignored by the RAT assembler.

Comments should adhere to the following guidelines.

- Each line of assembly code must contain a comment unless the purpose of the statement on the line is clear. Comments should not reiterate what is patently obvious from the associated source code.
- Comments should never contain right-side delimiters
- Use multi-line comments to make the length of the comment proportional to the complexity of the source code that the comment is describing.

RAT Assembler Labels

The ratasm assembler uses labels to mark locations in both the code and data segments. These locations can then be utilized by various instructions in the code segment. Labels are specified by a string followed immediately (without white space) by a colon. Label definitions can appear in either the code segment or data segment and used accordingly. Label definitions in the code segment can appear as the first field in any valid instruction or can appear on lines by themselves. Label definitions in the data segment can appear with out associated assembler directives. Label definitions can appear on associated with some directives, but not all of them (see the section on assembler directives). Multiple label definitions cannot appear on the same line in the source code.

RAT Assembler Directives

The ratasm assembler has several directives in order to provide the programmer with more versatility in overall program design. The directives enforce a clear and concise style of programming and generate errors and warnings upon misuse. Table 1 shows the current list of ratasm directives; an explanation of these directives follows the table.

Directive	Short Description
.CSEG	Indicates following information is associated with code segment
.DSEG	Indicates following information is associated with data segment
.ORG	Allows to adjust information placement in code and data segments
.EQU	Allows numeric values to be associated with strings
.DEF	Allows register file register to be associate with string
.BYTE	Allows user to reserve uninitialized memory
.DB	Allows user to reserve and initialize memory

Table 1: A list of ratasm assembler directives.

Directive: .ORG

The .ORG directive is shorthand for “origin”. This directive is used to placed data and instructions at known locations in either the data or the code segments, respectively. The .ORG directive takes one argument, which sets the location counter in the given segment. Both the code and data sections maintain their own counters which are incremented according to the data that is declared (the data segment) or the instructions that are listed (the code segment). The .ORG directive effectively sets these respective counters to the value associated with the argument provided with the .ORG directive. The .ORG directive must appear in the first column of the source listing and thus cannot have a label on the same line. Figure 1 shows a programming example of the .ORG directive (as well as some other soon to be discussed directives).

```
;-------------------------------------------------------------------------
;-- .ORG used in code segment
;-------------------------------------------------------------------------
.CSEG
.ORG   0x34    ; sets the code segment counter to 0x34

       LSL    R5     ; instruction at address 0x34
       LSR    R6     ; instruction at address 0x35

;-------------------------------------------------------------------------
;-- .ORG used in data segment
;-------------------------------------------------------------------------
.DSEG
.ORG   0x5A    ; set the data segment counter to 0x5A

var_name1  .BYTE    0    ; associated var_name1 with memory address 0x5A
var_name2  .BYTE    0    ; associated var_name2 with memory address 0x5A
```

Figure 1: Example usage of the .ORG directives in both code and data segments.

Segment Directives: .CSEG and .DSEG

The MCU memory space is divided into a code segment and a data segment. The .CSEG and .DSEG directives provide a means to differentiate between code and data segments The opcodes of all program instructions are placed in program memory and thus forms the code segment. All declared data is associated with data memory and is thus considered the data segment. Executable instructions must appear in the code segment while memory-type directives must appear in the data segment. Further details appear in the following sections.

Directive: .CSEG

The .CSEG directive is used to indicate that all the labels after the .CSEG directive are defined in terms program memory (as opposed to data memory). Instructions can only appear in the code segment while data can only be declared in the data segment. Attempts to declare memory in the code segment will result in an assembler error.

The .CSEG directive has no argument. When the .CSEG directive is used, the code memory address reverts back to the either the code memory position one location after the previously issued instruction or revert back to the previously issued .ORG value that was issued in the code segment. The .CSEG directive must appear in the first column of the source listing and thus cannot have a label on the same line. Figure 1 shows an example of .CSEG usage.

Directive: .DSEG

The .DSEG directive is used to indicate that all the labels after the .DSEG directive are defined in terms program memory (as opposed to data memory). Instructions can only appear in the code segment while data can only be declared in the data segment. Attempts to declare instructions in the data segment will result in an assembler error.

The .DSEG directive has no argument. When the .DSEG directive is used, the data memory address reverts back to the either the data memory position one location after the previously declared memory or revert back to the previously issued .ORG value that was issued in the data segment. The .DSEG directive must appear in the first column of the source listing and thus must not have a label on the same line. Figure 1 shows an example of .DSEG usage.

Directive: .DB

The .DB directive is used to reserve and initialize a given number of bytes in scratch memory starting at the current data memory address. This directive can be used both with and without a label on the same line. When the .DB directive appears with a label on the same line, the label is assigned the current data memory counter and can thus be used in some RAT instructions. When the .DB directive is used without a label, memory is initialized at the current address in data memory. The .DB directive initializes one byte for each decimal or hex number following the .DB directive. The internal counter used the data segment automatically tracks the proper location in data memory as more memory locations are specified. The .DB directive can only appear in the data segment.

```
;-------------------------------------------------------------------------
;-- .DB Directive usage
;-------------------------------------------------------------------------
.DSEG                    ; we're in the data segment
.ORG   0x20              ; set the data segment counter to 0x20

                  ; define ascii equivalent for numerals 0 to 9
                  ; 0x30 = '0', 0x31 = '1', etc.
                  ;
ascii_digits:     .DB 0x30, 0x31, 0x32, 0x33, 0x34
                  .DB 0x35, 0x36, 0x37, 0x38, 0x39
```

Figure 2: Example of the .DB directive.

Directive: .BYTE

The .BYTE directive is used to reserve a given number of uninitialized memory locations starting at the current data memory address. This directive can be used both with and without a label on the same line. When the .BYTE directive appears with a label, the label is assigned the current data memory counter and can appear in appropriate RAT instructions. The one argument to the .BYTE directive specifies the number of bytes to reserve in memory (uninitialized) starting at the current data memory location. The internal counter of the data segment automatically tracks the proper location in data memory as more memory locations are specified. The .BYTE directive can only appear in the data segment. Figure 2 shows the two forms of the .BYTE assembler directive.

```
;-------------------------------------------------------------------------
;-- .BYTE Directive usage
;-------------------------------------------------------------------------
.DSEG                    ; we're in the data segment
.ORG   0x30              ; set the data segment counter to 0x30

btn_cnt1  .BYTE     6    ; declare 6 bytes of data starting at data
                         ;  memory address 0x30; the label can be
                         ;  used for accessing the specified data
          .BYTE     3    ; declare 3 bytes of data starting at data
                         ;  memory address 0x36
```

Figure 3: Example of the .BTYE directive.

Directive: .EQU

The .EQU directive associates a label with an 8-bit value. The value range can either be signed or unsigned and can be specified as either a decimal or hexadecimal value. This directive allows for the replacement of specialized values such as masks with more descriptive alpha-type comments. The .EQU directive can appear in either the code or data segments. It is customary assembly language programming practice to place all .EQU assembler directives somewhere near the beginning of the assembly source code file and before any assembly language instruction.

The .EQU directive requires a label value field and a numeric value field. An equal sign must be placed between these two fields. Figure 4 shows examples of the .EQU directive.

```
;---------------------------------------------------------------------
;- Port Constants
;---------------------------------------------------------------------
.EQU ADC_PORT     = 0x02          ; port for ADC
.EQU ENG_PORT     = 0x0C          ; port for English converter
.EQU RAT_CLR_PORT = 34            ; port for clear all regs port
;---------------------------------------------------------------------

;---------------------------------------------------------------------
;- Bit Mask Constants
;---------------------------------------------------------------------
.EQU BTTN1_MASK = 0x08        ; mask all but BTN5
.EQU BTTN2_MASK = 0x02        ; mask all but BTN5
;---------------------------------------------------------------------
```

Figure 3: Examples of the .EQU assembler directive.

RAT Preferred Program Structure

Because assembly language programs can become exceedingly long, all assembly language program should be written in modular form in order to enhance the overall readability and understandability of the program. Each subroutines and specific areas of code that performs a single function should be delineated from other parts of the code and well commented.

The importance of proper style in assembly language coding is often understated. The reality is that an assembly program coded using a standardized style provides more information than a program that does not follow a standard. Using a standardized style makes the program more readable (and thus understandable) to anyone who may need to understand, debug, modify, or assign a grade to a given program.

Definitions: the word "should" appearing in this section indicates the given guideline is generally considered a *requirement*; in other words, it is not optional.

General Guidelines:

- Every assembly language program should contain three parts: 1) a comment banner, 2) assembler directives, and, 3) the assembly code.

    1) *Comment banner*: Every program file should contain a comment banner that includes information pertinent to the program. This includes names, dates, purpose, revision history etc.

    2) *Assembler Directives*: All .EQU and .BYTE directives should appear in one area which generally follows the comment banner. These directives should include comments that explain the intended purpose and/or usage of each directive. These directives should contain meaningful names as a form of self-commenting.

    3) *Assembly Code*: The assembly code should adhere to the bulleted items listed below:

General Assembly Code Instruction Guidelines:

- All assembly mnemonics should be in uppercase.
- All hexadecimal values should be in listed in uppercase.
- The source code should NEVER contain tabs; use multiple single spaces as an alternative to characters.
- White space and comments should be used liberally to enhance the readability of your code. In this context, white space is defined as empty lines or spaces - *but not tab characters!*
- All assembly code instruction mnemonics should start in the same column.
- The first operands of assembly language instructions should be aligned.
- All assembly source code should use a Courier font to ensure readability.
- All assembly source code lines should be less than 80 columns in length.

Label Guidelines:

- All labels should be in lower case (which delineates them from the upper cased used for instructions).
- Labels should exhibit self-commenting by having a meaningful name relative to the underlying code. Label length is limited to 32 characters but should be kept as short as possible while still retaining a self-commenting quality.
- All labels should start in the left-most column in your assembly source code.

Subroutine Guidelines:

- Subroutine labels (the name of the subroutine) should be self-commenting and give a rough indication as to the function performed by the subroutine. Subroutine labels should not be overly lengthy.
- The first character of a subroutine label should be capitalized in order to make them appear different than normal code labels.
- Subroutine labels should be kept as short as possible and apply self-commenting characteristics.
- Each subroutine definition should contain a comment banner explaining the intended purpose of the subroutine. This banner should also note the values are being passed to it (the information the subroutine expects to find in each register) and the registers that the subroutine modifies. This banner should contain as much information as required to ensure the subsequent understanding of the assembly code contained therein.

RAT Sample Style File

Figure 4 shows an example assembly language program highlighting respectable RAT assembly language source code appearance.

```
;- Programmer: Pat Wankaholic
;- Date: 09-29-10
;-
;- This program does something really cool. Here's the description.
;---------------------------------------------------------------------

;---------------------------------------------------------------------
;- Port Constants
;---------------------------------------------------------------------
.EQU SWITCH_PORT = 0x30          ; port for switches ----- INPUT
.EQU LED_PORT    = 0x0C          ; port for LED output --- OUTOUT
.EQU BTN_PORT    = 0x10          ; port for button input - INPUT
;---------------------------------------------------------------------

;---------------------------------------------------------------------
;- Misc Constants
;---------------------------------------------------------------------
.EQU BTN2_MASK = 0x08            ; mask all but BTN5
.EQU B0_MASK   = 0x01            ; mask all but bit0
.EQU B1_MASK   = 0x02            ; mask all but bit1
;---------------------------------------------------------------------

;---------------------------------------------------------------------
;- Memory Designation Constants
;---------------------------------------------------------------------
.DSEG
.ORG     0x00

COW:   .DB 9,7,6,5
       .DB 4,3,2,1

.ORG     0x34

btn2_counter:  .BYTE    0x05
btn3_counter:  .BYTE    0x04
;---------------------------------------------------------------------

.CSEG
.ORG         0x00

start:       SEI                          ; enable interrupts

main_loop:   IN      R0, BTN_PORT         ; input status of buttons
             IN      R1, SWITCH_PORT      ; input status of switches
             AND     R0, BTN2_MASK        ; clear all but BTN2
             BRN     bit_wank             ; jumps when BTN2 is pressed

             ;-------------------------------------------------------------
             ;- nibble wank portion of code
             ;-------------------------------------------------------------
wank:        ROL     R1                  ; rotate 2 times - msb-->lsb
             ROL     R1
             MOV     R0,R1               ; transfer data register to be read out
bit3:        BRN     fin_out             ; jump unconditionally to led output
             ;-------------------------------------------------------------

             ;-------------------------------------------------------------
             ; bit-wank algo: do something Blah, blah, blah ...
             ;-------------------------------------------------------------
bit_wank:    LD      R0,0x00              ; clear s0 for use as working register

             OR      R0, B0_MASK          ; set bit0
bit1:        LSR     R1                   ; shift msb into carry bit
             BRCC    bit2                 ; jump if carry not set
             OR      R0, B1_MASK          ; set bit1
```

```
bit2:       LSR     R1                  ; shift msb into carry bit
            BRCS    bit3                ; jump if carry not set
            ;-------------------------------------------------------------------

            CALL    My_sub              ; subroutine call
fin_out:    OUT     R0,LED_PORT         ; output data to LEDs
            BRN     main_loop           ; endless loop
            ;-------------------------------------------------------------------

            ;-------------------------------------------------------------------
            ; My_sub: This routines does something useful. It expects to find
            ; some special data in registers s0, s1, and s2. It changes the
            ; contents of registers blah, blah, blah...
            ;-------------------------------------------------------------------
My_sub:     LSR     R1                  ; shift msb into carry bit
            BRCS    bit3                ; jump if carry not set
            RET
```

Figure 4: Example RAT code with glowing with preferred RAT coding style.

RAT Sample List File Output

The following code is a sample source code listing output generated by the ratasm assembler.

```
List FileKey
-------------------------------------------------------------------------
C1      C2      C3      C4    || C5
-------------------------------------------------------------------
C1:  Address (decimal) of instruction in source file.
C2:  Segment (code or data) and address (in code or data segment)
       of inforation associated with current linte. Note that not all
       source lines will contain information in this field.
C3:  Opcode bits (this field only appears for valid instructions.
C4:  Data field; lists data for labels and assorted directives.
C5:  Raw line from source code.
-------------------------------------------------------------------------


(0001)                            || ;- Programmer: Pat Wankaholic
(0002)                            || ;- Date: 09-29-10
(0003)                            || ;-
(0004)                            || ;- This program does something really cool. Here's the description.
(0005)                            || ;-------------------------------------------------------------------------
(0006)                            ||
(0007)                            || ;-------------------------------------------------------------------------
(0008)                            || ;- Port Constants
(0009)                            || ;-------------------------------------------------------------------------
(0010)                       048  || .EQU SWITCH_PORT = 0x30          ; port for switches ----- INPUT
(0011)                       012  || .EQU LED_PORT = 0x0C             ; port for LED output --- OUTOUT
(0012)                       016  || .EQU BTN_PORT = 0x10             ; port for button input - INPUT
(0013)                            || ;-------------------------------------------------------------------------
(0014)                            ||
(0015)                            || ;-------------------------------------------------------------------------
(0016)                            || ;- Misc Constants
(0017)                            || ;-------------------------------------------------------------------------
(0018)                       008  || .EQU BTN2_MASK = 0x08            ; mask all but BTN5
(0019)                       001  || .EQU B0_MASK = 0x01              ; mask all but bit0
(0020)                       002  || .EQU B1_MASK = 0x02              ; mask all but bit1
(0021)                            || ;-------------------------------------------------------------------------
(0022)                            ||
(0023)                            || ;-------------------------------------------------------------------------
(0024)                            || ;- Memory Designation Constants
(0025)                            || ;-------------------------------------------------------------------------
(0026)                            || .DSEG
(0027)                       000  || .ORG     0x00
(0028)                            ||
(0029)  DS-0x000             004  || COW:   .DB 9,7,6,5
(0030)  DS-0x004             004  ||        .DB 4,3,2,1
(0031)                            ||
(0032)                       052  || .ORG     0x34
(0033)                            ||
(0034)  DS-0x034             005  || btn2_counter:  .BYTE    5
(0035)  DS-0x039             004  || btn3_counter:  .BYTE    4
(0036)                            || ;-------------------------------------------------------------------------
(0037)                            ||
(0038)                            || .CSEG
(0039)                       000  || .ORG         0x00
(0040)                            ||
-------------------------------------------------------------------------------------------
-STUP-  CS-0x000  0x36009  0x009  ||           MOV     r0,0x09     ; write dseg data to reg
-STUP-  CS-0x001  0x3A000  0x000  ||           LD      r0,0x00     ; place reg data in mem
-STUP-  CS-0x002  0x36007  0x007  ||           MOV     r0,0x07     ; write dseg data to reg
-STUP-  CS-0x003  0x3A001  0x001  ||           LD      r0,0x01     ; place reg data in mem
-STUP-  CS-0x004  0x36006  0x006  ||           MOV     r0,0x06     ; write dseg data to reg
-STUP-  CS-0x005  0x3A002  0x002  ||           LD      r0,0x02     ; place reg data in mem
-STUP-  CS-0x006  0x36005  0x005  ||           MOV     r0,0x05     ; write dseg data to reg
-STUP-  CS-0x007  0x3A003  0x003  ||           LD      r0,0x03     ; place reg data in mem
-STUP-  CS-0x008  0x36004  0x004  ||           MOV     r0,0x04     ; write dseg data to reg
-STUP-  CS-0x009  0x3A004  0x004  ||           LD      r0,0x04     ; place reg data in mem
-STUP-  CS-0x00A  0x36003  0x003  ||           MOV     r0,0x03     ; write dseg data to reg
-STUP-  CS-0x00B  0x3A005  0x005  ||           LD      r0,0x05     ; place reg data in mem
-STUP-  CS-0x00C  0x36002  0x002  ||           MOV     r0,0x02     ; write dseg data to reg
-STUP-  CS-0x00D  0x3A006  0x006  ||           LD      r0,0x06     ; place reg data in mem
-STUP-  CS-0x00E  0x36001  0x001  ||           MOV     r0,0x01     ; write dseg data to reg
-STUP-  CS-0x00F  0x3A007  0x007  ||           LD      r0,0x07     ; place reg data in mem
-------------------------------------------------------------------------------------------
(0041)  CS-0x010   1A000   0x010  || start:       SEI                         ; enable interrupts
(0042)                            ||
(0043)  CS-0x011  0x32010  0x011  || main_loop:   IN      R0, BTN_PORT        ; input status of buttons
(0044)  CS-0x012  0x32130         ||              IN      R1, SWITCH_PORT     ; input status of switches
(0045)  CS-0x013  0x20008         ||              AND     R0, BTN2_MASK       ; clear all but BTN2
(0046)  CS-0x014  0x08048         ||              BRN     bit_wank            ; jumps when BTN2 is pressed
(0047)                            ||
(0048)                            ||              ;-------------------------------------------------------------------
(0049)                            ||              ;- nibble wank portion of code
(0050)                            ||              ;-------------------------------------------------------------------
(0051)  CS-0x015  0x10102  0x015  || wank:        ROL      R1                 ; rotate 2 times - msb-->lsb
(0052)  CS-0x016  0x10102         ||              ROL      R1
(0053)  CS-0x017  0x04009         ||              MOV      R0,R1              ; transfer data register to be read out
(0054)  CS-0x018  0x08088  0x018  || bit3:        BRN      fin_out            ; jump unconditionally to led output
(0055)                            ||              ;-------------------------------------------------------------------
```

```
(0056)                          ||
(0057)                          ||              ;--------------------------------------------------------------------
(0058)                          ||              ; bit-wank algo: do something Blah, blah, blah ...
(0059)                          ||              ;--------------------------------------------------------------------
(0060)  CS-0x019  0x38000  0x019 || bit_wank:    LD      R0,0x00              ; clear s0 for use as working register
(0061)                          ||
(0062)  CS-0x01A  0x22001        ||              OR      R0, B0_MASK          ; set bit0
(0063)  CS-0x01B  0x10101  0x01B || bit1:        LSR     R1                   ; shift msb into carry bit
(0064)  CS-0x01C  0x0A071        ||              BRCC    bit2                 ; jump if carry not set
(0065)  CS-0x01D  0x22002        ||              OR      R0, B1_MASK          ; set bit1
(0066)  CS-0x01E  0x10101  0x01E || bit2:        LSR     R1                   ; shift msb into carry bit
(0067)  CS-0x01F  0x0A040        ||              BRCS    bit3                 ; jump if carry not set
(0068)                          ||              ;--------------------------------------------------------------------
(0069)                          ||
(0070)  CS-0x020  0x08099        ||              CALL    My_sub              ; subroutine call
(0071)  CS-0x021  0x3400C  0x021 || fin_out:     OUT     R0,LED_PORT          ; output data to LEDs
(0072)  CS-0x022  0x08008        ||              BRN     main_loop            ; endless loop
(0073)                          ||              ;--------------------------------------------------------------------
(0074)                          ||
(0075)                          ||              ;--------------------------------------------------------------------
(0076)                          ||              ; My_sub: This routines does something useful. It expects to find
(0077)                          ||              ; some special data in registers s0, s1, and s2. It changes the
(0078)                          ||              ; contents of registers blah, blah, blah...
(0079)                          ||              ;--------------------------------------------------------------------
(0080)  CS-0x023  0x10101  0x023 || My_sub:      LSR     R1                   ; shift msb into carry bit
(0081)  CS-0x024  0x0A040        ||              BRCS    bit3                 ; jump if carry not set
(0082)  CS-0x025   18002         ||              RET
(0083)                          ||
(0084)                          ||
(0085)                          ||
(0086)                          ||
```

The RAT Instruction Set

RAT Assembly Instructions by "Group"

Anytime you attempt to label any large group of things in a logical manner, you're bound to fail. The groupings listed below can be somewhat helpful in some situations. The listing itself can be useful but the groupings are somewhat arbitrary.

Logical Group

INSTR	TYPE	FORM	EXAMPLE
AND	reg/reg	AND rx,ry	
AND	reg/imm	AND rx,imm	
OR	reg/reg	OR rx,ry	
OR	reg/imm	OR rx,imm	
EXOR	reg/reg	EXOR rx,ry	
EXOR	reg/imm	EXOR rx,imm	
TEST	reg/reg	TEST rx,ry	
TEST	reg/imm	TEST rx,imm	

Arithmetic Group

INSTR	TYPE	FORM	EXAMPLE
ADD	reg/reg	ADD rx,ry	
ADD	reg/imm	ADD rx,imm	
ADDC	reg/reg	ADDC rx,ry	
ADDC	reg/imm	ACCD rx,imm	
SUB	reg/reg	SUB rx,ry	
SUB	reg/imm	SUB rx,imm	
SUBC	reg/reg	SUBC rx,ry	
SUBC	reg/imm	SUBC rx,imm	
CMP	reg/reg	CMP rx,ry	
CMP	reg/imm	CMP rx,imm	

Flow Control Group

INSTR	TYPE	FORM	EXAMPLE
BRN	imm	BRN label	
CALL	imm	CALL label	
BREQ	imm	BREQ label	
BRNE	imm	BRNE label	
BRCS	imm	BRCS label	
BRCC	imm	BRCC label	
RET	none	RET	
RETID	none	RETID	
RETIE	None	RETIE	

Shift and Rotate Group

INSTR	TYPE	FORM	EXAMPLE
LSL	reg	LSL rx	
LSR	reg	LSR rx	
ROL	reg	ROL rx	
ROR	reg	ROR rx	
ASR	reg	ASR rx	

Input/Output Group

INSTR	TYPE	FORM	EXAMPLE
IN	reg/imm	IN rx,imm	
OUT	reg/imm	OUT rx,imm	
SEI	none	SEI	
CLI	none	CLI	

Data Control Group

INSTR	TYPE	FORM	EXAMPLE
PUSH	reg	PUSH rx	
POP	reg	POP rx	
CLC	none	CLC	
SEC	none	SEC	
WSP	reg	WSP rx	

Memory Group

INSTR	TYPE	FORM	EXAMPLE
LD	reg/reg	LOAD rx,(ry)	
LD	reg/imm	LOAD rx,imm	
ST	reg/reg	STORE rx,(ry)	
ST	reg/imm	STORE rx,imm	
MOV	reg/reg	MOV rx,ry	
MOV	reg/imm	MOV rx,imm	

RAT Assembly Instructions by Type

The RAT instruction set has five types of instructions; the number and type of operands determines the instruction type. Each of these instruction types has their own distinct format. Table 2 provides an overview of the five types of instruction formats.

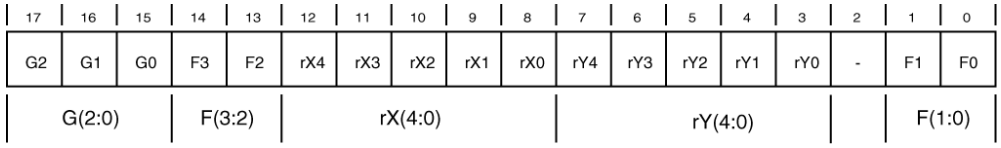
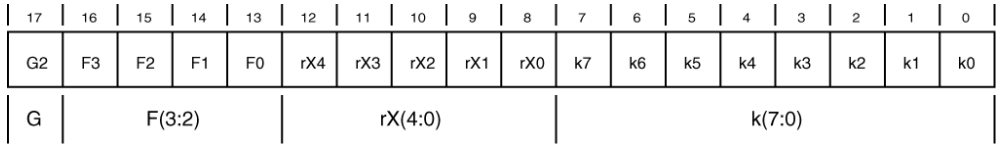
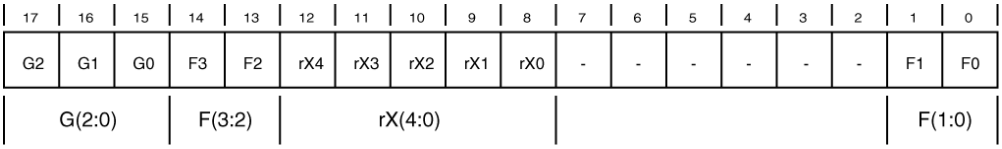
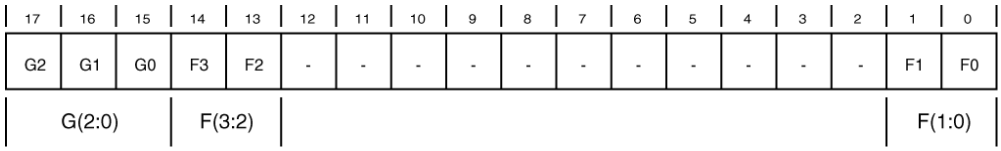
Instr Type	Instruction Format
reg/reg	
reg/imm	
imm	
reg	
none	

Table 2: Instruction types and associated instruction formats.

Instruction Type: Reg/Reg

AND	reg/reg	AND rx,ry	
OR	reg/reg	OR rx,ry	
EXOR	reg/reg	EXOR rx,ry	
TEST	reg/reg	TEST rx,ry	
ADD	reg/reg	ADD rx,ry	
ADDC	reg/reg	ADDC rx,ry	
SUB	reg/reg	SUB rx,ry	
SUBC	reg/reg	SUBC rx,ry	
CMP	reg/reg	CMP rx,ry	
MOV	reg/reg	MOV rx,ry	
LD	reg/reg	LOAD rx,(ry)	
ST	reg/reg	STORE rx,(ry)	

Table 3: Reg/Reg-type instructions.

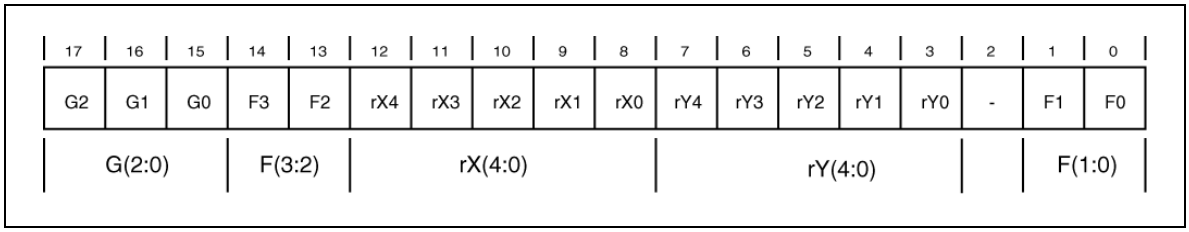


Figure 5: Reg/Reg-type instruction format.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	0	0	0	0	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		0	0
OR	0	0	0	0	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		0	1
EXOR	0	0	0	0	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		1	0
TEST	0	0	0	0	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		1	1
ADD	0	0	0	0	1	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		0	0
ADDC	0	0	0	0	1	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		0	1
SUB	0	0	0	0	1	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		1	0
SUBC	0	0	0	0	1	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		1	1
CMP	0	0	0	1	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		0	0
MOV	0	0	0	1	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		0	1
LD	0	0	0	1	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		1	0
ST	0	0	0	1	0	rX	rX	rX	rX	rX	rY	rY	rY	rY	rY		1	1

Table 4: Reg/Reg-type instructions with opcodes.

Instruction Type: Reg/Imm

AND	reg/imm	AND rx,imm	
OR	reg/imm	OR rx,imm	
EXOR	reg/imm	EXOR rx,imm	
TEST	reg/imm	TEST rx,imm	
ADD	reg/imm	ADD rx,imm	
ADDC	reg/imm	ACCD rx,imm	
SUB	reg/imm	SUB rx,imm	
SUBC	reg/imm	SUBC rx,imm	
CMP	reg/imm	CMP rx,imm	
IN	reg/imm	IN rx,imm	
OUT	reg/imm	OUT rx,imm	
MOV	reg/imm	MOV rx,imm	
LD	reg/imm	LOAD rx,imm	
ST	reg/imm	STORE rx,imm	

Table 5: Reg/Imm-type instructions.

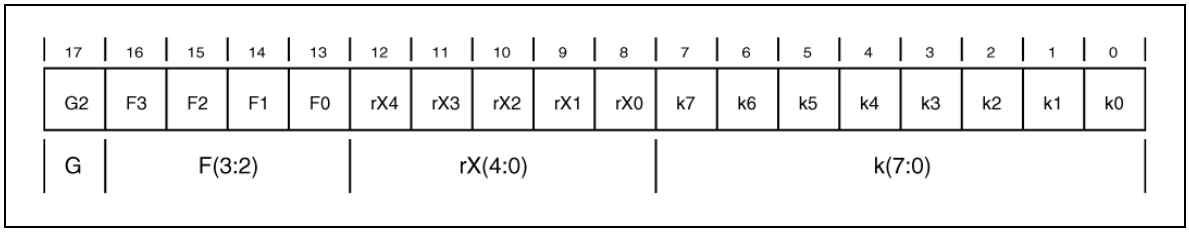


Figure 6: Reg/Imm-type instruction format.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	1	0	0	0	0	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
OR	1	0	0	0	1	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
EXOR	1	0	0	1	0	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
TEST	1	0	0	1	1	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
ADD	1	0	1	0	0	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
ADDC	1	0	1	0	1	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
SUB	1	0	1	1	0	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
SUBC	1	0	1	1	1	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
CMP	1	1	0	0	0	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
IN	1	1	0	0	1	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
OUT	1	1	0	1	0	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
MOV	1	1	0	1	1	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
LD	1	1	1	0	0	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k
ST	1	1	1	0	1	rX	rX	rX	rX	rX	k	k	k	k	k	k	k	k

Table 6: Reg/Imm-type instructions with opcodes.

Instruction Type: Imm

BRN	imm	BRN label	
CALL	imm	CALL label	
BREQ	imm	BREQ label	
BRNE	imm	BRNE label	
BRCS	imm	BRCS label	
BRCC	imm	BRCC label	

Table 7: Imm-type instructions.

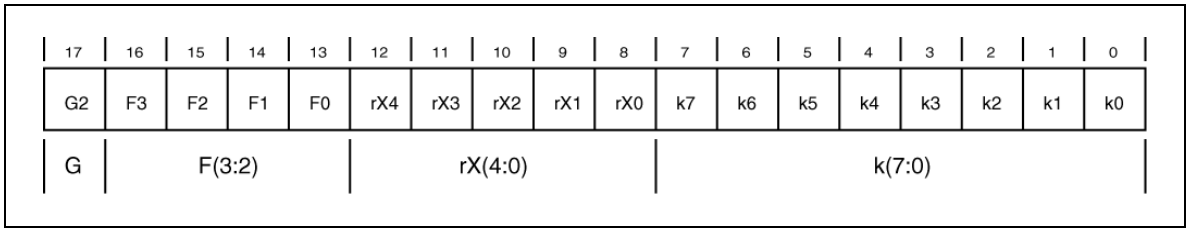


Figure 7: Imm-type instruction format.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BRN	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	0
CALL	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	1
BREQ	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	1	0
BRNE	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	1	1
BRCS	0	0	1	0	1	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	0
BRCC	0	0	1	0	1	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	1

Table 8: Imm-type instructions with opcodes.

Instruction Type: Reg

LSL	reg	LSL rx	
LSR	reg	LSR rx	
ROL	reg	ROL rx	
ROR	reg	ROR rx	
ASR	reg	ASR rx	
PUSH	reg	PUSH rx	
POP	reg	POP rx	
WSP	reg	WSP rx	

Table 9: Reg-type instructions.

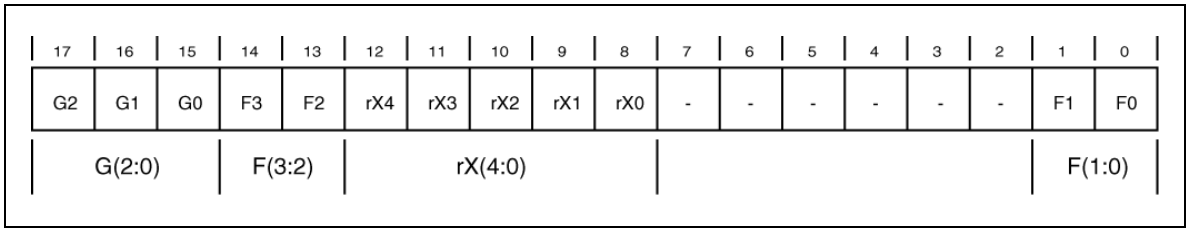


Figure 8: Reg-type instruction format.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSL	0	1	0	0	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	0
LSR	0	1	0	0	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	1
ROL	0	1	0	0	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	1	0
ROR	0	1	0	0	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	1	1
ASR	0	1	0	0	1	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	0
PUSH	0	1	0	0	1	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	1
POP	0	1	0	0	1	rX	rX	rX	rX	rX	-	-	-	-	-	-	1	0
WSP	0	1	0	1	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	0

Table 10: Reg-type instructions with opcodes.

Instruction Type: None

CLC	none	CLC	
SEC	none	SEC	
RET	none	RET	
RETI	none	RETI	
SEI	none	SEI	
CLI	none	CLI	

Table 11: None-type instructions.

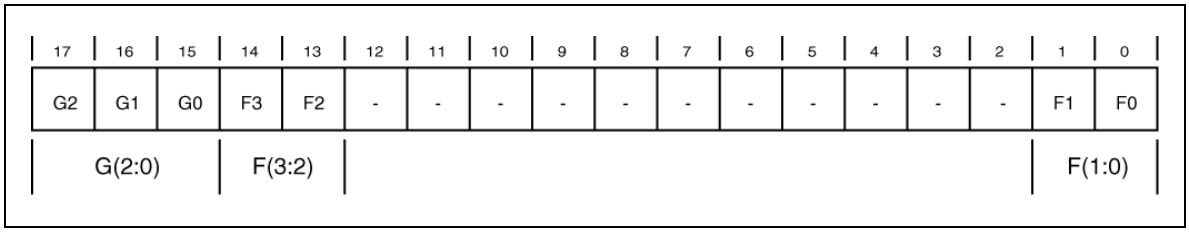


Figure 9: None-type instruction format.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLC	0	1	1	0	0												0	0
SEC	0	1	1	0	0												0	1
RET	0	1	1	0	0												1	0
RETI	0	1	1	0	0												1	1
SEI	0	1	1	0	1												0	0
CLI	0	1	1	0	1												0	1

Table 12: None-type instructions with opcodes.

RAT Assembly Instructions Type Listing

Here is a listing of all RAT instruction by type.

Reg/Reg Type

AND	reg/reg	AND rx,ry	
OR	reg/reg	OR rx,ry	
EXOR	reg/reg	EXOR rx,ry	
TEST	reg/reg	TEST rx,ry	
ADD	reg/reg	ADD rx,ry	
ADDC	reg/reg	ADDC rx,ry	
SUB	reg/reg	SUB rx,ry	
SUBC	reg/reg	SUBC rx,ry	
CMP	reg/reg	CMP rx,ry	
MOV	reg/reg	MOV rx,ry	
LD	reg/reg	LD rx,(ry)	
ST	reg/reg	ST rx,(ry)	

Reg/Imm Type

AND	reg/imm	AND rx,imm	
OR	reg/imm	OR rx,imm	
EXOR	reg/imm	EXOR rx,imm	
TEST	reg/imm	TEST rx,imm	
ADD	reg/imm	ADD rx,imm	
ADDC	reg/imm	ACCD rx,imm	
SUB	reg/imm	SUB rx,imm	
SUBC	reg/imm	SUBC rx,imm	
CMP	reg/imm	CMP rx,imm	
MOV	reg/imm	MOV rx,imm	
LD	reg/imm	LD rx,imm	
ST	Reg/imm	ST rx,imm	
IN	reg/imm	IN rx,imm	
OUT	reg/imm	OUT rx,imm	

Imm Type

BRN	Imm	BRN label	
CALL	Imm	CALL label	
BREQ	Imm	BREQ label	
BRNE	Imm	BRNE label	
BRCS	Imm	BRCS label	
BRCC	Imm	BRCC label	

Reg Type

LSL	Reg	LSL rx	
LSR	Reg	LSR rx	
ROL	Reg	ROL rx	
ROR	Reg	ROR rx	
ASR	Reg	ASR rx	
PUSH	Reg	PUSH rx	
POP	Reg	POP rx	
WSP	Reg	WSP rx	

None Type

CLC	none	CLC	
SEC	none	SEC	
RET	none	RET	
RETI	none	RETI	
SEI	none	SEI	
CLI	none	CLI	

Detailed RAT Assembly Instruction Description

The following section lists each of the RAT instruction in a detailed format. The instruction details include the following:

- Instruction mnemonic
- Short Instruction description
- Associated RTL statements
- Condition flag affects
- Extended instruction description
- Detailed instruction format
- Instruction usage example

ADD *(addition)*

RTL: *Rd ← Rd + Rs (reg – reg form)*
RTL: *Rd ← Rd + immed (reg – imm form)*

Forms:

```
ADD   Rd,Rs
ADD   Rd,imm_val
```

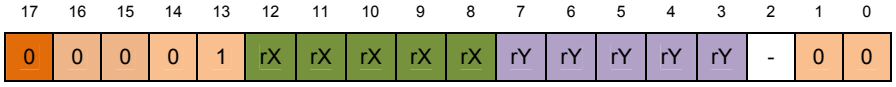
Carry Flag: *set if the addition operation results in a carry out of the MSB position.*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The ADD instruction performs an addition operation on the two operands with the result being stored in the destination register. The value in the destination register Rd is overwritten with the result of this operation. The TEST instruction has two distinct forms which are differentiated by the source operand.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. Instruction execution does not affect the value in the source register.

Opcode:



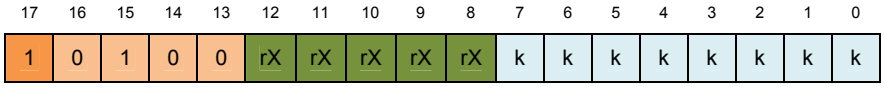
rX: *destination register;* rY: *source register*

Usage:

```
ADD   r1,r4    ; addition of values in registers r1 & r4;
               ;  result is placed in r1; value in r4 is not
               ;  affected.
               ;    r1 = 0xA4   r4 = 0xC7          (before exec)
               ;    r1 = 0x6B   r4 = 0xC7  Z=0 C=1 (after exec)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
ADD   r1,0xDC  ; addition of values in register r1 & 0xDC;
               ;  result is placed in r1
               ;    r1 = 0x24          (before execution)
               ;    r1 = 0x00  Z=1 C=1 (after execution)
```

ADDC *(addition including Carry flag)*

RTL: *Rd ← Rd + Rs + C (reg – reg form)*
RTL: *Rd ← Rd + immed + C (reg – imm form)*

Forms:

```
ADDC    Rd,Rs
ADDC    Rd,imm_val
```

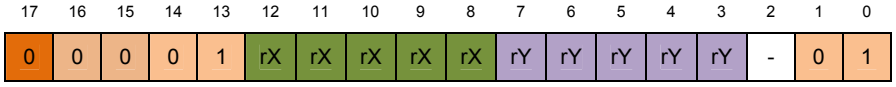
Carry Flag: *set if the addition operation results in a carry out of the MSB position.*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The ADDC instruction performs an addition operation on the two operands and the Carry flag with the result being stored in the destination register. The result of this operation overwrites the value in the destination register Rd. The ADDC instruction has two distinct forms, which are differentiated by the source operand.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. The value in the source register is not affected by instruction execution.

Opcode:



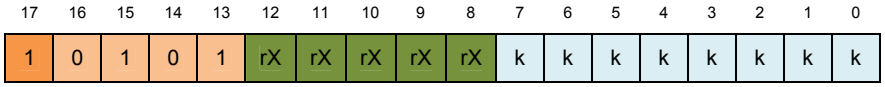
rX: *destination register;* rY: *source register*

Usage:

```
ADDC    r1,r4   ; addition of values in registers r1 & r4;
                ;  result is placed in r1; value in r4 is not
                ;  affected.
                ;    r1 = 0xA4   r4 = 0xC7  C =1       (before exec)
                ;    r1 = 0x6C   r4 = 0xC7  Z=0 C=1    (after exec)
```

Register-Immediate Form: An immediate value specifies the source operand and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
ADDC    r1,0xDC  ; addition of values in register r1 & 0xDC & C flag;
                 ;  result is placed in r1
                 ;    r1 = 0x24  C=1         (before execution)
                 ;    r1 = 0x00  Z=0 C=1     (after execution)
```

AND *(logical bitwise AND)*

RTL: *Rd ← Rd · Rs (reg – reg form)*
RTL: *Rd ← Rd · immed (reg – imm form)*

Forms:
```
AND   Rd,Rs
AND   Rd,imm_val
```

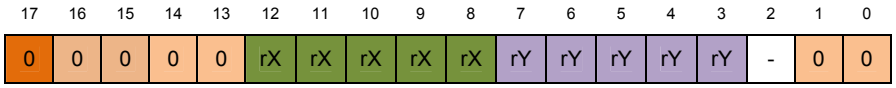
Carry Flag: *not affected*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The AND instruction performs a bit-wise logical AND operation between the source and destination operands and places the result in the register specified by the destination operand. The AND instruction has two distinct forms which are differentiated by the source operand. The result of the AND operation overwrites the value in the destination register.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. Instruction execution does not affect the value in the source register.

Opcode:



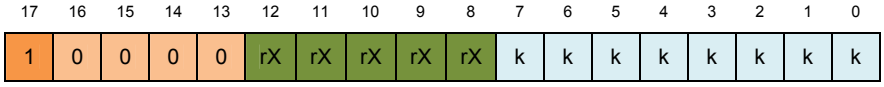
rX: *destination register;* rY: *source register*

Usage:

```
AND    r1,r4    ; bitwise and of values in register r1 & r4;
                ;  result is placed in r1; value in r4 is not
                ;  affected.
                ;    r1 = 0xA4   r4 = 0xC7  (before execution)
                ;    r1 = 0x84   r4 = 0xC7  (after execution)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
AND    r1,0x3C  ; bitwise AND of values in register r1 & 0x4A;
                ;  result is placed in r1
                ;    r1 = 0xA4   (before execution)
                ;    r1 = 0x24   (after execution)
```

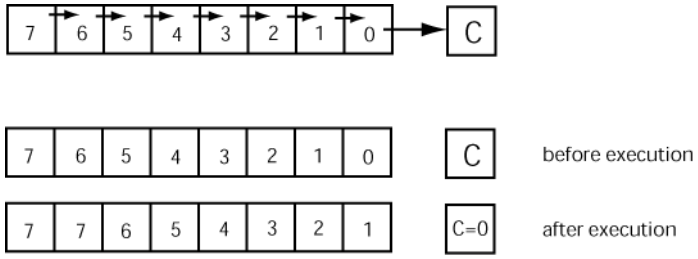
ASR *(arithmetic shift right)*

RTL: *Rd ← Rd(7) & Rd(6) & Rd(6:1), C ← Rd(0)* **Forms:** `ASR Rd`

Carry Flag: *takes value of lsb of destination register*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The ASR instruction performs a shift right operation on the destination register. The current value of the destination register MSB remains unchanged. The MSB is considered to be the sign bit so the MSB is also shifted right as a result of this operation. The LSB of the destination register before the shift operation is shifted into the Carry flag.



Opcode:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	0

rX: *destination register*

Usage:

```
ASR    r1      ; arithmetic shift right of register r1;
               ;  result is placed in r1;
               ;    r1 = 0xE7                (before execution)
               ;    r1 = 0xF3   C=1  Z=0     (after execution)
```

BRCC *(branch if carry cleared)*

RTL: *if (C==0) then PC ← imm_val, else nop*

Forms:

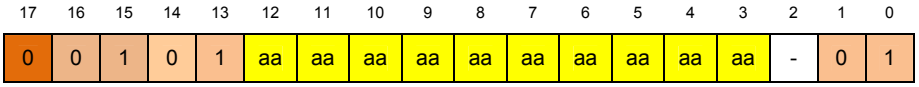
```
BRCC    label
BRCC    imm_val
```

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The BRCC instruction branches to an address value when the Carry flag is cleared (C=0). If the Carry flag is presently set, program flow drops through to the instruction following the BRCC instruction. The immediate value associated with the branch is designated by a program label or a constant value.

Opcode:



aa: program memory address

Usage:

```
DOGBONE:
          ADD     r1,r2      ; add register r2 to register r1
                             ;  Carry flag is affected by this operation.
                             ;
          BRCC    DOGBONE    ; if C=0, program flow will jump to instruction
                             ;  after the label argument of this instruction.
                             ;  If C=1, program execution drops to the
                             ;  instruction following BRCC
```

BRCS *(branch if carry set)*

RTL: *if (C==1) then PC ← imm_val, else nop*

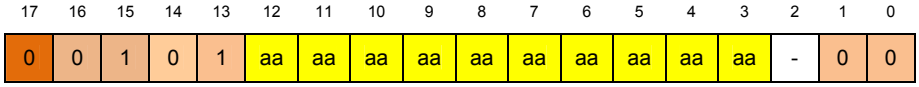
Forms:
```
BRCS    label
BRCS    imm_val
```

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The BRCS instruction branches to an address value when the carry flag is set. If the carry flag is cleared, program flow drops through to the instruction following the BRCS instruction. The immediate value associated with the branch is designated by a program label or a constant value.

Opcode:



aa: program memory address

Usage:

```
WHISKER:
          ADD     r1,r2     ; add register r2 to register r1
                            ;  Carry flag is affected by this operation.
                            ;
          BRCS    WHISKER   ; if C=1, program flow will jump to instruction
                            ;  after the label argument of this instruction.
                            ;  If C=0, program execution drops to the
                            ;  instruction following BRCC
```

BREQ *(branch if equal)*

RTL: *if (Z==1) then PC ← imm_val, else nop*

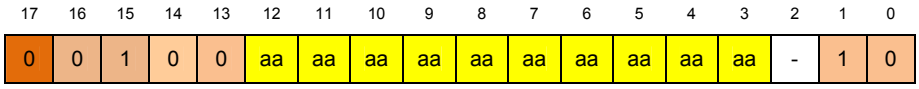
Forms:
```
BREQ   label
BREQ   imm_val
```

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The BREQ instruction branches to an address value in the case that the zero flag is set. If the zero flag is presently cleared, program flow drops through to the instruction following the BREQ instruction. The immediate value associated with the branch is designated by a program label or a constant value. The mnemonic for this instruction is somewhat confusing. The "equal" portion part of the instruction is associated with the fact if two non-equal values are subtracted from each other, the result will set the zero flag.

Opcode:



aa: program memory address

Usage:

```
WHISKER:
          ADD     r1,r2     ; add register r2 to register r1
                            ;  Zero flag is affected by this operation.
                            ;
          BREQ    WHISKER   ; if Z=1, program flow will jump to instruction
                            ;  after the label argument of this instruction.
                            ;  If Z=0, program execution drops to the
                            ;  instruction following BREQ
```

BRN *(unconditional branch)*

RTL: *PC ← imm_val*

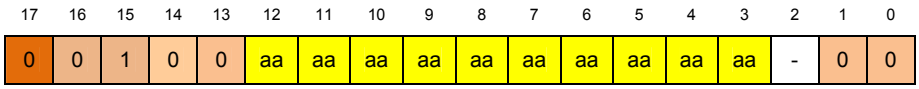
Forms: `BRN immed_val`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The BRN instruction causes an unconditional branch to the immediate address associated with the instruction. The immediate address is specified by use of a program label or a constant value.

Opcode:



aa: program memory address

Usage:

```
NOODLE:             ; typical program label
    ROL    R1       ; rotate register r1 left
    BRN    NOODLE   ; unconditional branch back to ROL instruction
```

BRNE *(branch if not equal)*

RTL: *if (Z==0) then PC ← imm_val, else nop*

Forms:

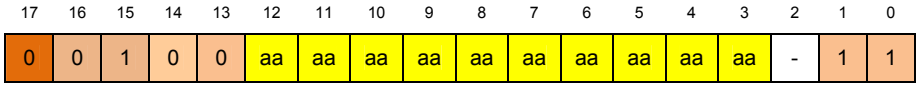
```
BRNE    label
BRNE    imm_val
```

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The BRNE instruction branches to an address value in the case that the zero flag is cleared. If the zero flag is presently set, program flow drops through to the instruction following the BRNE instruction. The immediate value associated with the branch is designated by a program label or a constant value. The mnemonic for this instruction is somewhat confusing. The "not equal" portion part of the instruction is associated with the fact if two non-equal values are subtracted from each other, the result will clear the zero flag.

Opcode:



aa: program memory address

Usage:

```
WHISKER:
          ADD     r1,r2     ; add register r2 to register r1
                            ;  Zero flag is affected by this operation.
                            ;
          BRNE    WHISKER   ; if Z=0, program flow will jump to instruction
                            ;  after the label argument of this instruction.
                            ;  If Z=1, program execution drops to the
                            ;  instruction following BRNE
```

CALL *(branch to subroutine)*

RTL: *PC ← imm_val, SP ← PC* **Forms:** `CALL imm_val`

Carry Flag: *not affected* **Zero Flag:** *not affected*

Description: The CALL instruction is used to direction program flow to a set of instructions that are generally designated to be a subroutine. The address associated with the CALL instruction is the address of the next executed instruction following the CALL instruction. Before the CALL instruction immediate address is loaded in the PC, the old value of the PC is pushed onto the stack. The old value of the PC contains the address of the instruction after the CALL instruction and is the first instruction executed after a return from the subroutine. Each CALL instruction should have an accompanying RET instruction in order to avoid stack underflow problems.

Opcode:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	1

aa: program memory address

Usage:

```
;-----------------------------------------
;- ADD_VALS subroutine: add some registers
;-----------------------------------------
ADD_VALS:
           ADD   r1,r2
           ADD   r1,r3
           ADD   r1,r4
           RET
;-----------------------------------------
           CALL  ADD_VALS     ; branches to ADD_VALS subroutine
```

CLI *(clear interrupt flag)*

RTL: *IF ← 0*

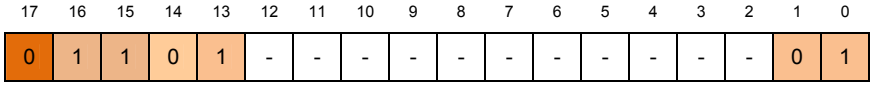
Forms: `CLI`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The CLI instruction disables the MCU from receiving interrupts. The IF bit (interrupt flag) must be set in order to allow the MCU to process interrupts. The CLI instruction clears the IF bit. Interrupts that may appear when the interrupts are disabled are not acted upon by the MCU.

Opcode:



Usage:

```
CLI             ; clear interrupt flag to allow interrupts
                ;    IF=1            (before execution)
                ;    IF=0            (after execution)
```

CLC *(clear Carry flag)*

RTL: *C ← 0* **Forms:** `CLC`

Carry Flag: *cleared (C=0)* **Zero Flag:** *not affected*

Description: The CLC instruction clears the current value of the carry flag. The instruction requires no arguments.

Opcode:

Usage:

```
CLC             ;  clear the Carry flag
                ;    C=1       (before execution)
                ;    C=0       (after execution)
```

CMP *(compare two values)*

RTL: *Rd - Rs (reg – reg form)*
RTL: *Rd - immed (reg – imm form)*

Forms:
```
CMP   Rd,Rs
CMP   Rd,imm_val
```

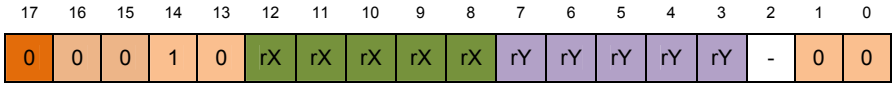
Carry Flag: *set if the addition operation results in a borrow (underflow) into the MSB position.*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The CMP instruction performs a subtraction operation on the two operands; the result is not written back to the destination register but the Z and C flags are altered according to the result of the subtraction operation. Specifically, the value in the source operand is subtracted from the value in the destination register. The Carry flag is set by this operation and indicate the instruction execution resulted in an underflow. The instruction does not modify the destination register.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. The source register is not affected by instruction execution.

Opcode:



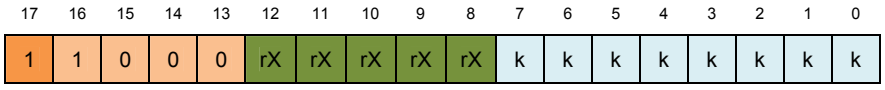
rX: *destination register;* rY: *source register*

Usage:

```
CMP   r1,r4    ; value in register r4 is subtracted from value in
               ;  register r1; C and Z flags are affected but
               ;  values in registers do not change
               ;    r1 = 0xD4   r4 = 0xC7          (before exec)
               ;    r1 = 0xD4   r4 = 0xC7  Z=0 C=0 (after exec)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
CMP   r1,0xC8  ; value 0xC8 is subtracted from value in
               ;  register r1; C and Z flags are affected but
               ;  values in registers do not change
               ;    r1 = 0x88          (before execution)
               ;    r1 = 0x88  Z=0 C=1 (after execution)
```

EXOR *(logical bitwise exclusive OR)*

RTL: *Rd ← Rd xor Rs (reg – reg form)*
RTL: *Rd ← Rd xor immed (reg – imm form)*

Forms:
```
EXOR    Rd,Rs
EXOR    Rd,imm_val
```

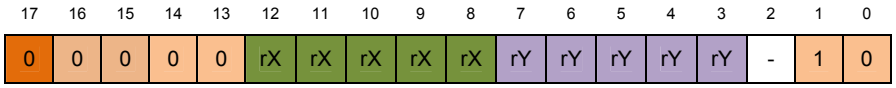
Carry Flag: *not affected*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The EXOR instruction performs a bit-wise logical exclusive OR operation between the source and destination operands and places the result into the register specified by the destination operand. The EXOR instruction has two distinct forms which are differentiated by the source operand. The EXOR instruction overwrites the value in the destination register. .

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. Instruction execution does not affect the value in the source register.

Opcode:



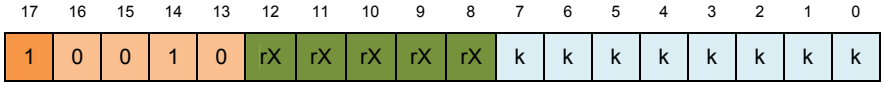
rX: *destination register;* rY: *source register*

Usage:

```
EXOR    r1,r4   ; bitwise exclusive OR of values in register r1 & r4;
                ;  result is placed in r1; value in r4 is not
                ;  affected.
                ;    r1 = 0xA4    r4 = 0xC7       (before execution)
                ;    r1 = 0x63    r4 = 0xC7  Z=0  (after execution)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
EXOR    r1,0x7C  ; bitwise exclusive OR of values in register
                 ; r1 & 0x1C;  result is placed in r1
                 ;    r1 = 0xF0       (before execution)
                 ;    r1 = 0x8C  Z=0  (after execution)
```

IN *(input data from input port)*

RTL: *Rd ← in_port(imm_val)*

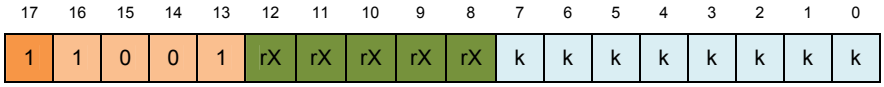
Forms: `IN Rd,imm_val`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The IN instruction inputs the data on the input port specified by the source operand into the register specified by the destination operand. Data read in from the input port overwrites the value in the destination register. The immediate value for the source operand can be any 8-bit value.

Opcode:



rX: *destination register;* k*: immediate value*

Usage:

```
IN   r1,0x23   ; input value on input port number 0x23 and
               ;  place in register r1;
               ;    r1=0xD4   in port 0x23 val=0xC8  (before exec)
               ;    r1=0xC8                          (after exec)
```

LD *(load value from data memory into register)*

RTL: *Rd ← (Rs)* *(reg – reg form)*
RTL: *Rd ← (immed)* *(reg – imm form)*

Forms:
```
LD   Rd,(Rs)
LD   Rd,(imm_val)
```

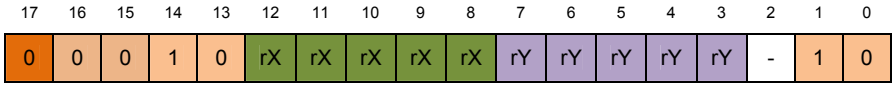
Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The LD instruction copies data from data memory into the destination register. This instruction has two distinct forms which are differentiated by the source operand. The source operand is not affected by the execution of this instruction.

Register-Register Form: The source operand is specified by an indirect register reference; the contents of the source register is used as the address of the data in data memory to be transferred to the destination register. Instruction execution does not affect the value of the specified data memory location.

Opcode:



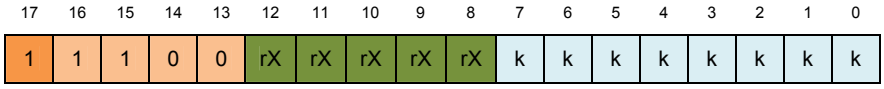
rX: *destination register;* rY: *source register*

Usage:

```
LD   r1,(r4)   ; value of data addressed by the value in
               ;  register r4 is placed into register r1;
               ;  the value in register r4 is not affected.
               ;    r1=0xD4 r4=0xC7 mem loc 0xC7=34 (before exec)
               ;    r1=0x34 r4=0xC7 mem loc 0xC7=34 (after exec)
```

Register-Immediate Form: The source operand specifies the address of the data in data memory to be transferred to the destination register.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
LD   r1,0x45   ; value of data in memory address 0x45 is
               ;  placed into register r1;
               ;    r1=0xD4   mem loc 0x45=CD (before exec)
               ;    r1=0xCD   mem loc 0x45=CD (after exec)
```

LSL *(logical shift left)*

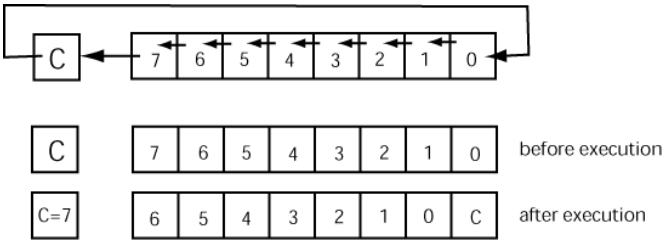
RTL: *Rd ← Rd(6:0) & C, C ← Rd(7)*

Forms: `LSL Rd`

Carry Flag: *takes value of msb of destination register before shift operation*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The LSL instruction performs a left shift operation on the destination register. The MSB of the original destination register, Rd(7), is shifted into the carry flag. The previous value of the carry flag becomes the LSB of the new value in the destination register.



Opcode:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	0

rX: *destination register*

Usage:

```
LSL    r1      ; logical shift left of register r1;
               ;  result is placed in r1;
               ;    r1 = 0x54   C=1      (before execution)
               ;    r1 = 0xA9   C=0      (after execution)
```

LSR *(logical shift right)*

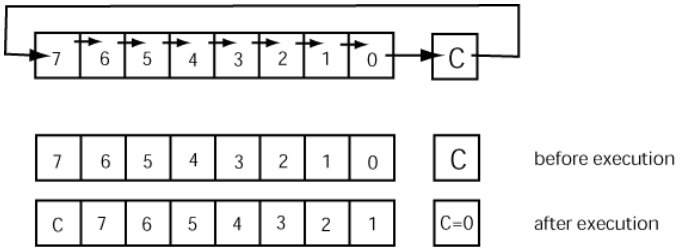
RTL: *Rd ← C & Rd(6:0), C ← Rd(0)*

Forms: `LSR Rd`

Carry Flag: *takes value of lsb of destination register before shift operation*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The LSR instruction performs a right shift operation on the destination register. The LSB of the original destination register is shifted into the carry flag. The previous value of the carry flag becomes the MSB of the new value in the destination register.



Opcode:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	0	1

rX: *destination register*

Usage:

```
LSR    r1      ; logical shift right of register r1;
               ;  result is placed in r1;
               ;    r1 = 0x54   C=1      (before execution)
               ;    r1 = 0xA5   C=0      (after execution)
```

MOV *(move value into register)*

RTL: *Rd ← Rs (reg – reg form)*
RTL: *Rd ← immed (reg – imm form)*

Forms:
```
MOV   Rd,Rs
MOV   Rd,imm_val
```

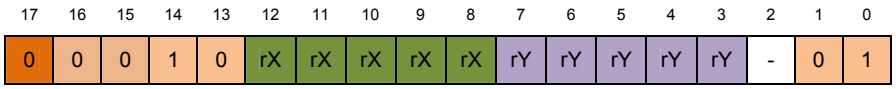
Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The MOV instruction copies the data from the source operand into the destination register.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. Instruction execution does not affect the value in the source register.

Opcode:



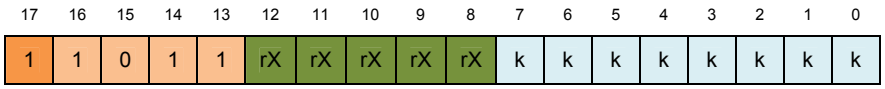
rX: *destination register;* rY: *source register*

Usage:

```
MOV   r1,r4    ; value in register r4 is place in register r1;
               ;  value in register r4 does not change
               ;     r1 = 0xD4   r4 = 0xC7  (before execution)
               ;     r1 = 0xC7   r4 = 0xC7  (after execution)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

OR *(logical bitwise OR)*

RTL: *Rd ← Rd + Rs (reg – reg form)*
RTL: *Rd ← Rd + immed (reg – imm form)*

Forms:
```
OR    Rd,Rs
OR    Rd,imm_val
```

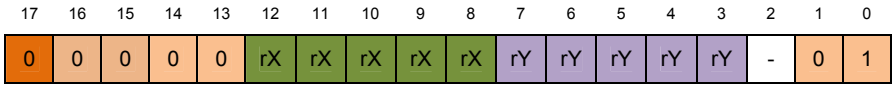
Carry Flag: *not affected*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The OR instruction performs a bit-wise logical OR operation between the source and destination operands and places the result in the register specified by the destination operand. The OR instruction has two distinct forms which are differentiated by the source operand. The value in the destination register is overwritten with the result of the OR operation.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. Instruction execution does not affect the value in the source register.

Opcode:



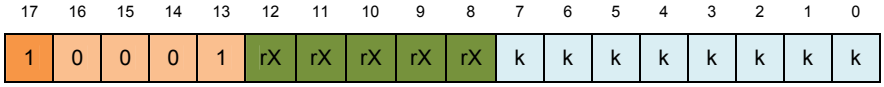
rX: *destination register;* rY: *source register*

Usage:

```
OR    r1,r4    ; bitwise OR of values in register r1 & r4;
               ;  result is placed in r1; value in r4 is not
               ;  affected.
               ;    r1 = 0xA4   r4 = 0xC7      (before execution)
               ;    r1 = 0xE7   r4 = 0xC7  Z=0 (after execution)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
OR    r1,0x1C  ; bitwise OR of values in register r1 & 0x1C;
               ;  result is placed in r1
               ;    r1 = 0x24      (before execution)
               ;    r1 = 0x3C  Z=0 (after execution)
```

OUT *(output data from register to output port)*

RTL: *out_port(imm_val) ← Rd*

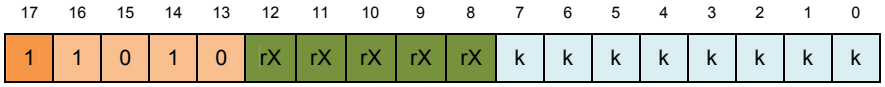
Forms: `OUT Rd,imm_val`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The OUT instruction outputs data in the source register to the output port specified by the source operand. Instruction execution does not affect the value in the destination register. The immediate value can be any 8-bit value.

Opcode:



rX: source *register;* k: *immediate value*

Usage:

```
OUT   r1,0x37  ; output value in register r1 to output port
               ;  designated by 0x37
               ;    r1=0xD4                          (before exec)
               ;    r1=0xD4   out port 0x37 = 0xD4   (after exec)
```

POP *(copy data from stack into register)*

RTL: *Rd ← (SP), SP ← SP +1*

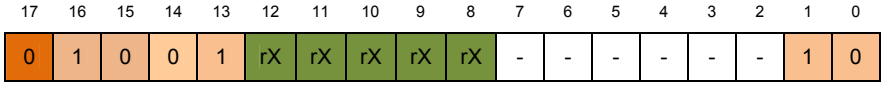
Forms: `POP R1`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The POP instruction copies data from the stack into the destination register. The POP instruction overwrites data in the destination register. The stack pointer (SP) is incremented during the POP operation under hardware control. The POP operation is normally used in conjunction with the PUSH operation to ensure the integrity of the stack.

Opcode:



rX: *destination register*

Usage:

```
POP     r1      ; logical shift right of register r1;
                ;  result is placed in r1;
                ;     r1 = 0x71                (before execution)
                ;     r1 = 0xE2                (after execution)
                ;                  (0xE2 was on the top of stack)
```

PUSH *(store data from register onto stack)*

RTL: *(SP) ← Rd, SP ← SP - 1*

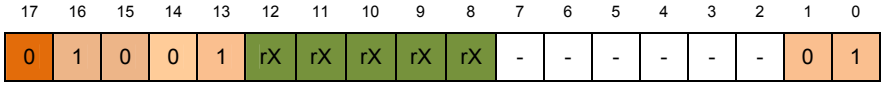
Forms: `PUSH Rd`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The PUSH operation copies data from the destination register into the stack. The PUSH instruction does not later the contents of the destination register. The stack pointer (SP) is decremented during the PUSH operation, which is under hardware control. The PUSH operation is normally used in conjunction with the POP operation to ensure the integrity of the stack.

Opcode:



rX: *destination register*

Usage:

```
PUSH    r1      ; place value from the top of stack into
                ;  register r1; stack pointer is
                ;  result is placed in r1;
                ;    r1 = 0x71                (before execution)
                ;    r1 = 0x71                (after execution)
```

RET *(return from subroutine)*

RTL: *PC ← (SP), SP + 1*

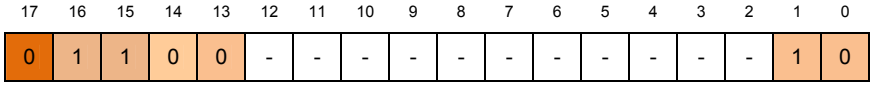
Forms: **`RET`**

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The RET instruction is typically issued on the exit from a subroutine. The RET instruction pops the stack into the PC; details of stack "popping" are purposely not stated as part of this instruction. The return instruction should be used in conjunction with the CALL instruction in order to prevent stack overflow issues.

Opcode:



Usage:

```
RET             ; return from subroutine; stack is popped into
                ;  program counter (PC)
```

RETID *(return from interrupt handler with interrupts disabled)*

RTL: *PC ← (SP),*
SP+1,
Z← shadZ, C← shadC,
IF ← 0

Forms: `RETID`

Carry Flag: *The Carry flag is overwritten with the shadow Carry flag.*

Zero Flag: *The Zero flag is overwritten with the shadow Zero flag.*

Description: The RETID instruction is issued on the exit from an interrupt service routine. The RETID instruction pops the stack into the PC and restores the C and Z flags form their respective shadow registers. The RETID instruction disables the IF (interrupt flag).

Opcode:

Usage:

```
RETID          ; return from interrupt handler; stack is popped into
               ; program counter (PC); shadow C & Z flags
               ; overwrite the real C & Z flags
               ; C=0  Z=1  shadC=1  shadZ=0 IF=1 (before execution)
               ; C=1  Z=0  shadC=1  shadZ=0 IF=0(after execution)
```

RETIE *(return from interrupt handler with interrupts enabled)*

RTL: *PC ← (SP),*
SP+1,
Z← shadZ, C← shadC,
IF ← 1

Forms: `RETIE`

Carry Flag: *The Carry flag is overwritten with the shadow Carry flag.*

Zero Flag: *The Zero flag is overwritten with the shadow Zero flag.*

Description: The RETIE instruction is issued on the exit from an interrupt service routine. The RETIE instruction pops the stack into the PC and restores the C and Z flags form their respective shadow registers. The RETIE instruction enables the IF (interrupt flag).

Opcode:

Usage:

```
RETIE          ; return from interrupt handler; stack is popped into
               ; program counter (PC); shadow C & Z flags
               ; overwrite the real C & Z flags
               ; C=0  Z=1  shadC=1  shadZ=0 IF=1 (before execution)
               ; C=1  Z=0  shadC=1  shadZ=0 IF=1(after execution)
```

ROL *(rotate left)*

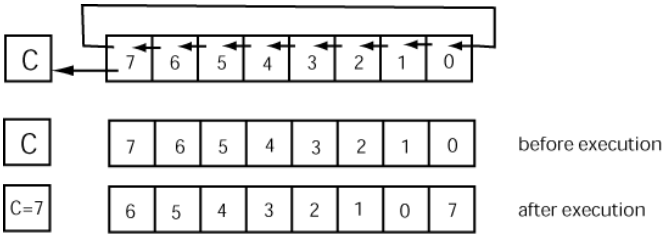
RTL: *Rd ← Rd(6:0) & Rd(7), C ← Rd(7)*

Forms: `ROL Rd`

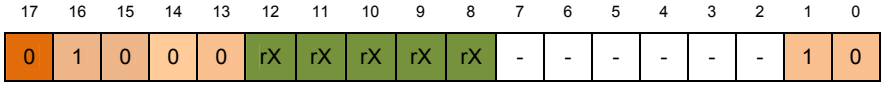
Carry Flag: *takes value of msb of destination register before shift operation*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The ROL instruction performs a shift left operation on the destination register. In the rotate left operation, the MSB of the destination register before the shift becomes the LSB of the destination register after the shift. The carry flag is loaded with the value of the MSB before the shift left operation.



Opcode:



rX: *destination register*

Usage:

```
ROL    r1      ; logical shift right of register r1;
               ;  result is placed in r1;
               ;    r1 = 0x71               (before execution)
               ;    r1 = 0xE2   C=0  Z=0    (after execution)
```

ROR *(rotate right)*

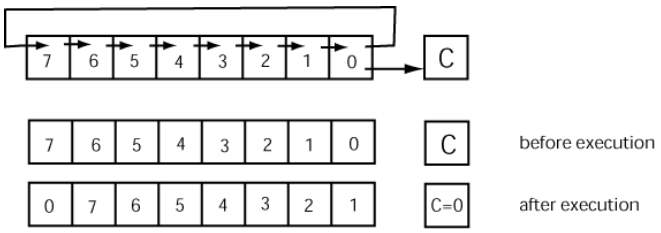
RTL: *Rd ← Rd(0) & Rd(7:1), C ← Rd(0)*

Forms: `ROR Rd`

Carry Flag: *takes value of lsb of destination register before shift operation*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The ROR instruction performs a shift right operation on the destination register. In the rotate right operation, the LSB of the destination register before the shift becomes the MSB of the destination register after the shift. The carry flag is loaded with the value of the LSB before the shift left operation.



Opcode:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	rX	rX	rX	rX	rX	-	-	-	-	-	-	1	1

rX: *destination register*

Usage:

```
ROR    r1      ; logical shift right of register r1;
               ;  result is placed in r1;
               ;    r1 = 0x8B            (before execution)
               ;    r1 = 0xC5   C=1      (after execution)
```

SEC *(set Carry flag)*

RTL: *C ← 1*

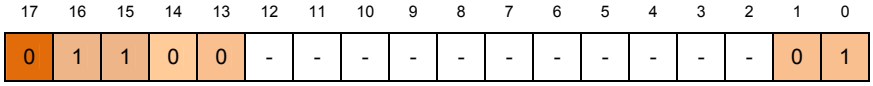
Forms: **SEC**

Carry Flag: *cleared (C=1)*

Zero Flag: *not affected*

Description: The SEC instruction sets the current value of the carry flag. The instruction requires no arguments.

Opcode:



Usage:

```
SEC             ;  set the Carry flag
                ;    C=0       (before execution)
                ;    C=1       (after execution)
```

SEI *(set interrupt flag)*

RTL: *IF ← 1*

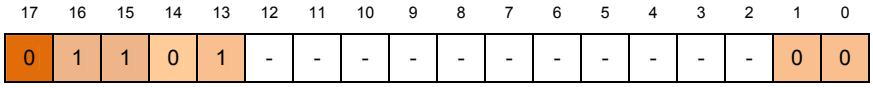
Forms: `SEI`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The SEI instruction enables the MCU to receive interrupts. The IF (interrupt flag) must be set in order for the MCU core to be able to process interrupts. The SEI instruction sets the IF bit. If there is an interrupt pending when the IF bit is set under program control, an interrupt cycle is entered on the next instruction cycle following the SEI instruction. Entry into an interrupt cycle automatically disables any future interrupts until a the program executes an SEI instruction.

Opcode:



Usage:

```
SEI             ; set interrupt flag to allow interrupts
                ;    IF=0            (before execution)
                ;    IF=1            (after execution)
```

ST *(store value from register into data (scratchpad) memory)*

RTL: (*Rd*) ← *Rs* *(reg – reg form)*
RTL: *(immed)* ← *Rd* *(reg – imm form)*

Forms: `ST Rs,(Rd)`
`ST Rs,(imm_val)`

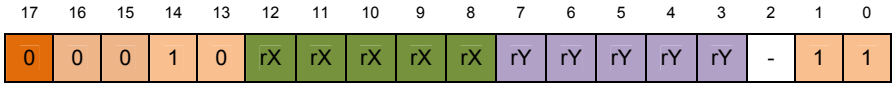
Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The ST instruction copies data from the source register into data memory at the location specified by the destination operand. This instruction has two distinct forms which are differentiated by the destination operand. Instruction execution does not affect either the source or the destination operand.

Register-Register Form: The destination operand is specified by an indirect register reference; the contents of the destination register is used as the address of the data in data memory to be transferred to the destination register. Instruction execution does not affect the data at the specified memory location.

Opcode:



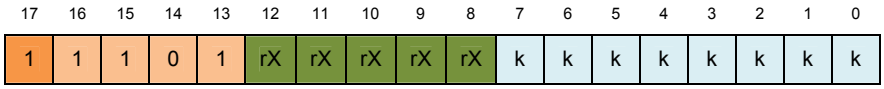
rX: *source register;* rY: *destination register*

Usage:

```
ST   r1,(r4)   ; value of data in register r1 is placed in
               ;  data memory location addressed by the value in
               ;  register r4; value in register r4 is not affected.
               ;    r1=0xD4 r4=0xC7  mem loc 0xC7=34 (before exec)
               ;    r1=0xD4 r4=0xC7  mem loc 0xC7=D4 (after exec)
```

Register-Immediate Form: The destination operand specifies the address in data memory to store the value in the source register.

Opcode:



rX: *source register;* k: *immediate value*

Usage:

```
ST   r1,0x5D   ; value of data in register r1 is placed in
               ;  data memory location 0x5D;
               ;    r1=0x1F   mem loc 0x5D=34 (before exec)
               ;    r1=0x1F   mem loc 0x5D=1F (after exec)
```

SUB *(subtraction)*

RTL: *Rd ← Rd - Rs (reg – reg form)*
RTL: *Rd ← Rd - immed (reg – imm form)*

Forms:
```
SUB   Rd,Rs
SUB   Rd,imm_val
```

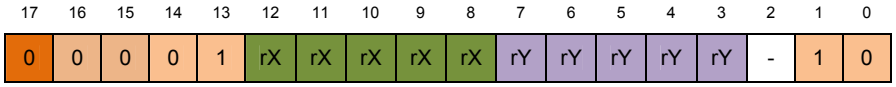
Carry Flag: *set if the addition operation results in a borrow (underflow) into the MSB position.*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The SUB instruction performs a subtraction operation on the two operands with the result being stored in the destination register. Specifically, the value in the source register is subtracted from the value in the destination register. The Carry flag is set by this operation and indicates the instruction execution resulted in an underflow. This instruction overwrites the date in the destination register Rd.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. The value in the source register is not affected by instruction execution.

Opcode:



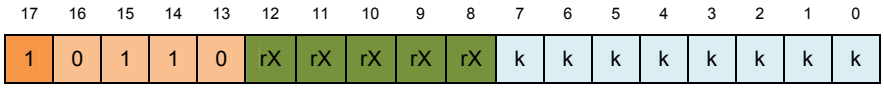
rX: *destination register;* rY: *source register*

Usage:

```
SUB   r1,r4     ; subtraction of values in registers r1 & r4;
                ;  result is placed in r1; value in r4 is not
                ;  affected.
                ;    r1 = 0xD4   r4 = 0xC7          (before exec)
                ;    r1 = 0x0D   r4 = 0xC7  Z=0 C=0 (after exec)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
SUB   r1,0xC8  ; subtraction of immediate value of 0xC8 from value
in r1; ;
               ;  result is placed in r1
               ;    r1 = 0x88          (before execution)
               ;    r1 = 0xC0  Z=0 C=1 (after execution)
```

SUBC *(subtraction including Carry flag)*

RTL: *Rd ← Rd - Rs - C (reg – reg form)*
RTL: *Rd ← Rd - immed - C (reg – imm form)*

Forms:
```
SUBC    Rd,Rs
SUBC    Rd,imm_val
```

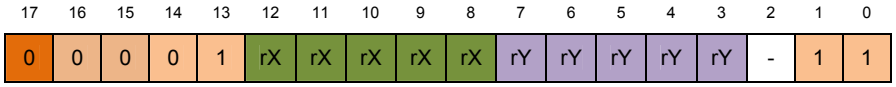
Carry Flag: *set if the addition operation results in a borrow (underflow) into the MSB position.*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The SUB instruction performs a subtraction operation on the two operands and the Carry flag with the result being stored in the destination register. Specifically, the value in the source register and the Carry flag are subtracted from the value in the destination register. The Carry flag is set by this operation and indicate the instruction execution resulted in an underflow. The SUBC instruction has two distinct forms which are differentiated by the source operand. This instruction overwrites the value in the destination register Rd.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. Instruction execution does not affect the value in the source register

Opcode:



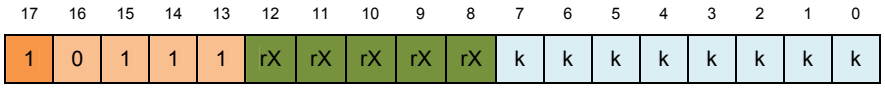
rX: *destination register;* rY: *source register*

Usage:

```
SUBC    r1,r4   ; addition of values in registers r1 & r4;
                ;  result is placed in r1; value in r4 is not
                ;  affected.
                ;    r1 = 0xD4   r4 = 0xC7  C=1      (before exec)
                ;    r1 = 0x0C   r4 = 0xC7  Z=0 C=0  (after exec)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
SUBC    r1,0xC8  ; addition of values in register r1 & 0xDC & C flag;
                 ;  result is placed in r1
                 ;    r1 = 0x89  C=1      (before execution)
                 ;    r1 = 0xC0  Z=0 C=1  (after execution)
```

TEST *(logical bitwise AND; registers do not change)*

RTL:
RTL:

Forms:
```
TEST   Rd,Rs
TEST   Rd,imm_val
```

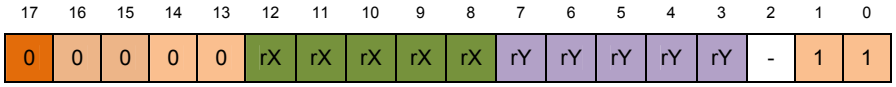
Carry Flag: *not affected*

Zero Flag: *set if all bits in Rd are zero after operation is complete; reset in all other cases.*

Description: The TEST instruction performs a bit-wise logical AND operation between the source and destination operands; the result is not written back to the destination operand but the Z flag is altered according to the result of the AND operation. The TEST instruction has two distinct forms which are differentiated by the source operand. This instruction does not affect the contents of either the source or the destination register.

Register-Register Form: The source operand is specified by a register; the source operand is the value in that register. This instruction has no affect on either the source or destination register.

Opcode:



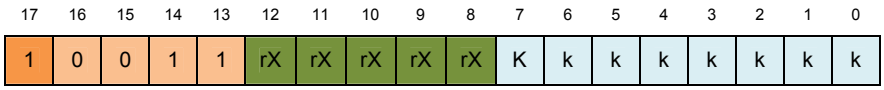
rX: *destination register;* rY: *source register*

Usage:

```
TEST    r1,r4   ; bitwise AND of values in register r1 & r4;
                ;  Z flag is modified; values in r1 & r4 are not
                ;  affected.
                ;    r1 = 0xA4   r4 = 0xC7      (before execution)
                ;    r1 = 0xA4   r4 = 0xC7  Z=0 (after execution)
```

Register-Immediate Form: The source operand is specified as an immediate value and can be any 8-bit value. Instruction execution does not affect the value in the destination register.

Opcode:



rX: *destination register;* k: *immediate value*

Usage:

```
TEST    r1,0x3C  ; bitwise AND of values in register r1 & 0x4A;
                 ;  Z flag is modified; value in r1 is not affected
                 ;    r1 = 0xA4      (before execution)
                 ;    r1 = 0xA4  Z=0 (after execution)
```

WSP *(write stack pointer)*

RTL: *SP ← Rd*

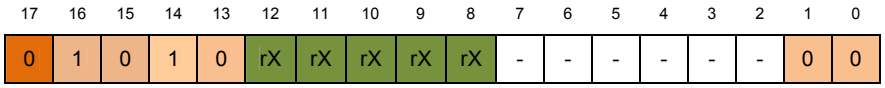
Forms: `WSP Rd`

Carry Flag: *not affected*

Zero Flag: *not affected*

Description: The WSP instruction writes a value from the destination register into the stack pointer (SP). The stack pointer is 8 bits wide so this instruction completely initializes SP. The WSP instruction does not affect the value of the destination register.

Opcode:



rX: *destination register*

Usage:

```
WSP     r1      ; write r1 to the stack pointer (SP)
                ;     r1 = 0x1D   SP=0x30     (before execution)
                ;     r1 = 0x1D   SP=0x1D     (after execution)
```

Revision History:

2011.01.01
I made a tiny mod to a code example, renamed this document, and uploaded it to Google docs. I also added this revision history on the final page of the document.

2011.01.04:
Ordered instructions alphabetically.

2011.03.30
Cleaned up various errors in instruction definitions, added .DB directive description, added WSP, RETID, and RETIE.

2012.10.09
Fixed ST opcode and removed misc. whitespace to eliminate blank pages in the document.
V1.0

2012.12.27
Did general document clean-up. Extended description of various assembler features. Added section on instruction formats. Added headings and table of contents.
v2.00