Programmer’s Reference Manual

IP2022 Internet Processor™

Revision History

Revision	Release Date	Summary of Changes
1.0	March 15, 2001	Original issue.
1.1	April 5, 2001	Merged with demo board User's Guide.
1.2	June 7, 2001	Rewritten for the Unity IDE.
1.3	October 17,2001	Rewritten for the IP2022 REV2.0 & new look and feel

Part #:: IP2K-DRM-2022PRM-13

Ubicom, Inc.
635 Clyde Avenue
Mountain View, CA 94043
tel 650 210 1500
fax 650 210 8715
www.ubicom.com

Preface

This manual describes the architecture and instruction set of the IP2022 Internet Processor. Much of the information in this manual overlaps with the material in the data sheet, however this document presents a more detailed description of the instruction set architecture for the benefit of programmers. Refer to the data sheet for the electrical specifications, package mechanical drawing, and ordering information.

Related Documentation

Documentation for the IP2022:

- *IP2022 Data Sheet*, available from Ubicom (#IP2K-DDS-IP2022DS).
- *IP2022 Programmer's Reference Manual*, (this book, #IP2K-DRM-2022PRM).

Documentation for the IP2022 Connectivity Kit:

- *Connectivity Kit User's Guide*, available from Ubicom (#IP2K-DUG-CK2UG).

Reference manuals for the tool chain:

- *GNUPro Toolkit—GNUPro Utilities*, available from Red Hat.
- *GNUPro Toolkit—GNUPro Compiler Tools*, available from Red Hat.
- *GNUPro Toolkit—Debugging with GDB*, available from Red Hat.

1.0 Overview

The Ubicom IP2022 Internet Processor ™ combines support for communication physical layer, Internet protocol stack, device-specific application and device-specific peripheral software modules in a single chip, and is reconfigurable over the Internet.

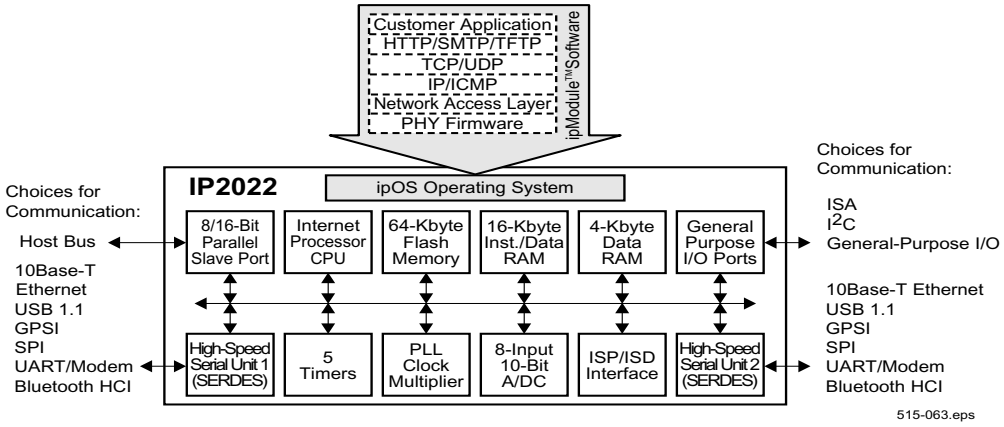


Figure 1-1 IP2022 Block Diagram

It can be programmed, and reprogrammed, using pre-built software modules and configuration tools to create true single-chip solutions for a wide range of device-to-device and device-to-human communication applications. Fabricated in an advanced 0.25-micron process, its RISC-based deterministic architecture

provides high-speed computation, flexible I/O control, efficient data manipulation, in-system programming, and in-system debugging.

Two hardware serializer/deserializer (SERDES) units give the IP2022 the ability to directly connect to a variety of common network interfaces. This function provides the ability to implement on-chip 10Base-T Ethernet (MAC and PHY), USB, and a variety of other fast serial protocols. The inclusion of two SERDES units facilitate translation from one format to another, allowing the IP2022 to be used as a protocol converter. The 100 MHz operating frequency, with most instructions executing in a single cycle, delivers the throughput needed for emerging network connectivity applications, and a flash-based program memory allows both in-system and on-the-fly reprogramming. The IP2022 implements peripheral, communications and control functions as software modules (ipModule™ software), replacing traditional hardware for maximum system design flexibility. This approach allows rapid, inexpensive product design and, when needed, quick and easy reconfiguration to accommodate changes in market needs or industry standards.

On-chip dedicated hardware also includes a PLL, an 8-channel 10-bit ADC, general-purpose timers, single-cycle multiplier, analog comparator, LFSR units, external memory interface, brown-out power voltage detector, watchdog timer, low-power support, multi-source wakeup capability, user-selectable clock modes, high-current outputs, and 52 general-purpose I/O pins.

A TCP/IP network protocol stack is available, and a variety of additional software that is necessary to form a complete end-to-end connectivity solution is being developed. Tools for developing with and using the IP2022, including the complete Red Hat GNUPro tools, are available from leading suppliers.

1.1 Key Features

CPU Features

- RISC engine core with DC to 100 MHz operation
- 10 ns instruction cycle
- Compact 16-bit fixed-length instructions
- Single-cycle instruction execution on most instructions (3 cycles for jumps and calls)
- 16-level hardware stack for high-performance subroutine linkage
- 8 × 8 signed/unsigned single-cycle multiply
- Pointers and stack operation optimized for C compiler
- Uniform, linear address space (no register banks)

On-Chip Memory

- 64 Kbyte (32K × 16) program flash memory
- 16 Kbyte (8K × 16) program/data RAM
- 4 Kbyte linear-addressed data RAM
- Self-programming with built-in charge pump: instructions to read, write, and erase flash memory
- Ability to address up to 128K bytes of external memory

Fast and Deterministic Program Execution and Interrupt Response

- Predictable execution rate for real-time applications
- Fast and deterministic 3-cycle interrupt response
- 30 ns internal interrupt response at 100 MHz including context save
- Hardware save/restore of register context (PC, W, STATUS, MULH, SPDREG, IPH, IPL, DPH, DPL, SPH, SPL, ADDRSEL, DATAH, DATAL)

Multiple Networking Protocols and Physical Layer Support Hardware

- Two full-duplex serializer/deserializer (SERDES) channels for
- 10Base-T (MAC/PHY), USB, and other fast serial protocols
  - Embedded connectivity nodes
  - Two channels for protocol bridging
  - 10Base-T, GPSI, SPI, UART, USB protocols
  - Squelch function for 10Base-T Ethernet
- Four hardware LFSR units
  - CRC generation/checking
  - Data-whitening
  - Encryption

General-Purpose Hardware Peripherals

- Two 16-bit timers with 8-bit prescalers supporting:
  - Timer mode
  - PWM mode
  - Capture/Compare mode
- Parallel host interface, 8/16-bit selectable for use as a communications coprocessor

- One 8-bit timer with programmable 8-bit prescaler
- One 8-bit real-time clock/counter with programmable 15-bit prescaler and 32 kHz crystal input
- Watchdog timer with prescaler
- On-chip PLL clock multiplier with pre- and post-divider
  - 100 MHz on-chip clock from 4 MHz external crystal
- 10-bit, 8-channel ADC with 1/2 LSB accuracy
- Analog comparator with hysteresis enable/disable
- Brown-out minimum supply voltage detector
- External interrupt inputs on 8 pins (Port B)

Sophisticated Power and Frequency/Clock Management Support

- Operating voltage of 2.3V to 2.7V
- Switching the system clock frequencies between different clock sources
- Changing the core clock using a selectable divider
- Shutting down the PLL and/or the OSC input
- Dynamic CPU speed control with **speed** instruction
- Power-On-Reset (POR) logic

Flexible I/O

- 52 I/O Pins
- 2.3V to 3.3V symmetric CMOS output drive
- 5V-tolerant inputs
- Port A pins capable of sourcing/sinking 24 mA
- Optional I/O synchronization to CPU core clock

Programming and Debugging Support

- Updateable application program
  - Run-time self programming
- On-chip in-system programming interface
- On-chip in-system debugging support interface
- Debugging at full IP2022 operating speed
- Programming at device supply voltage level
- Real-time emulation, program debugging, and integrated software development environment

Complete Software Development Environment Ubicom's Software Development Kit (SDK)

- IpOS™ operating system
- IpStack™ Software
  - TCP/IP protocol stack
  - NE2000 Ethernet drivers
- ipWeb Software - HTTP 1.1 Server
- ipFile Flash virtual file system
- ipIO Software - Device I/O driven interfaces
  - MII, I^2C, SPI, GPSI, UART
- ipModule™ Software - Pre-built Connectivity Software modules
  - ipEthernet - 10Base-T Ethernet
  - ipHomePlug - HomePlug power line networking
  - ipUSB - USB 1.1 Host or Device
  - ipBlue - Bluetooth
  - ipWLANstation - 802.11b station (node or bridge)
  - ipWLANaccesspoint - 802.11b access point
- Configuration tool
  - Integrated tool to support rapid development efforts

1.2 Architecture

1.2.1 CPU

The IP2022 implements an enhanced Harvard architecture (i.e. separate instruction and data memories) with independent address and data buses. The 16-bit program memory and 8-bit dual-port data memory allow instruction fetch and data operations to occur in parallel. The advantage of this architecture is that instruction fetch and memory transfers can be overlapped by a multistage pipeline, so that the next instruction can be fetched from program memory while the current instruction is executed with data from the data memory.

Ubicom has developed a revolutionary RISC-based architecture that is deterministic, jitter free, and completely reprogrammable. The IP2022 implements a four-stage pipeline (fetch, decode, execute, and write back). At the maximum operating frequency of 100 MHz, instructions are executed at the rate of one per 10 ns clock cycle.

1.2.2 Serializer/Deserializer Units (SERDES)

One of the key elements in optimizing the IP2022 for device-to-device and device-to-human communication is the inclusion of two on-chip serializer/deserializer (SERDES) units. These units support popular communication protocols such as 10Base-T Ethernet, GPSI, SPI, UART, and USB, allowing the IP2022 to be used as a protocol converter in bridge and gateway applications.

By performing data serialization and deserialization in hardware, the CPU bandwidth needed to support serial communications is greatly reduced, especially at high baud rates. Providing two units allows easy implementation of protocol conversion or bridging functions, such as a USB-to-Ethernet bridge.

1.2.3 Low-Power Support

Particular attention has been paid to minimizing power consumption. For example, an on-chip PLL allows use of a lower-frequency external source (e.g., an inexpensive 4 MHz crystal can be used to produce a 100 MHz on-chip clock), which reduces both power consumption and EMI. In addition, software can change the execution speed of the CPU to reduce power consumption, and a mechanism is provided for automatically changing the speed on entry and return from an interrupt service routine. The **speed** instruction specifies power-saving modes that include a clock divisor between 1 and 128. This divisor only affects the clock to the CPU core, not the timers. The **speed** instruction also specifies the clock source (OSC1 clock, RTCLK oscillator, or PLL clock multiplier), and whether to disable the OSC1 clock oscillator or the PLL. The **speed** instruction executes using the current clock divisor.

1.2.4 Memory

The IP2022 CPU executes from a 32K × 16 flash program memory, 16K × 8 RAM program/data memory and 4K × 8 RAM

data memory. In addition, the ability to write into the program flash memory allows flexible non-volatile data storage. An interface is available for up to 128K bytes of external memory, which can be expanded to 2M bytes by using additional address bits on general purpose I/O. The maximum execution rate is 30 MIPS from flash memory and 100 MIPS from RAM. Speed-critical routines can be copied from the flash memory to the RAM for faster execution. The IP2022 has a mechanism for in-system programming of its flash and RAM program memories through a four-wire SPI interface, and software has the ability to reprogram the program memories at run time. This allows the functionality of a device to be changed in the field over the Internet.

1.2.5 Instruction Set

The IP2022 instruction set, using 16-bit words, implements a rich set of arithmetic and logical operations, including signed and unsigned 8-bit × 8-bit integer multiply with a 16-bit product. See Chapter 4.5 "Instruction Descriptions" on page 42 for detailed description of the instruction set.

2.0 Writing Assembly Code

2.1 Comments, Constants, and Symbols

Comments can occur at the end of a line, after a pound sign (#) or semicolon. The semicolon is recommended, as in:

```
clrb status,c ;this is a comment
;this whole line is a comment
```

Comments can also be enclosed in C-style comment delimiters, such as:

```
/* this is a comment */
```

and:

```
/* this is
a multi-line
comment */
```

As with C, comments may not be nested.

Constants may be character constants, string constants, or numeric constants. A character constant is a single quote followed by a character, such as **'f** which is a byte with the value 102 (decimal) corresponding to its ASCII code. A string constant consists of one or more characters enclosed in double quotes, such as **"Ubicom"**. To use a character with special meaning or a

character outside of the standard ASCII printing characters, a backslash (****) is used to indicate a representation for the character, as shown in Table 2-1.

Table 2-1 Special Characters

Representation	Value	Character
\b	0x08	Backspace (control-H)
\f	0x0C	Form Feed (control-L)
\n	0x0A	New Line (control-J)
\r	0x0D	Carriage Return (control-M)
\t	0x09	Horizontal Tab (control-I)
\xNN	0xNN	NN is the ASCII code for the character, in hex. E.g. **\x09** is equivalent to **\t**.
****	0x5C	Backslash (\)
\"	0x22	Double Quote (")

A numeric constant is an integer. By default, it is interpreted as a decimal number. To express it in binary, prefix the value with **0b**, e.g. **0b01101001**. To express it in hex, prefix the value with **0x**, as in **0x9F**. Hex digits may be either upper or lower case.

A symbol is a name for any nameable object, such as labels and constants. A symbol consists of one or more characters from the set of letters, digits, period (**.**), and underscore (_). A symbol may not begin with a digit. Symbols are case-sensitive, so **abc** is

distinct from **aBc**. Symbols may not be reserved words (see Section 2.8).

The assembler has two special constructs for recovering the address of a symbol in data memory. **%lo8data(symbol)** returns the low byte of the address of **symbol**, and **%hi8data(symbol)** returns the high byte. These are useful for initializing pointers used in the IPH/IPL, DPH/DPL, and SPH/SPL registers. Another set of constructs, **%lo8insn(symbol)** and **%hi8insn(symbol)**, are used to recover addresses in program memory. These are useful for initializing pointers used in the ADDRH/ADDRL register.

To make a symbol visible outside the file it requires a global directive.

Global Symbol Example:

```
    .global _isr
_isr
    page 1f
    jmp1f
1:
    page1b
    jmp1b
```

.func emits debugging information to denote function name, and is ignored unless the file is assembled with debugging enabled. Only `--gstabs' is currently supported. label is the entry point of the function and if omitted name prepended with the `leading char' is used. `leading char' is usually _ or nothing, depending on the target. All functions are currently defined to have void return type. The function must be terminated with .endfunc.

```
.func   reset_vector,reset_vector
reset_vector:
    movw, #FCFG_FRDTS_VALUE(SYSTEM_FREQ) |
    .endfunc
```

2.2 Directives

Please consult the *GNUPro Toolkit—GNUPro Auxiliary Development Tools* manual for a complete list of all assembler directives.

The IP2022 assembler has four directives in addition to the standard list:

.word—four bytes

.long—four bytes

.half—two bytes

.short—two bytes

2.3 Operators

Operators are arithmetic functions, like + or %. Prefix operators are followed by an argument (see Prefix Operators). Infix operators appear between their arguments (see Infix Operators). Operators may be preceded and/or followed by whitespace.

2.3.1 Prefix Operators

The IP2022 assembler (as) has the following *prefix operator*s, each taking one argument, an absolute.

- (Negation) - Two's complement negation.
~ (Complementation) - Bitwise not.

2.3.2 Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from + or -, both arguments must be absolute, and the result is absolute.

- Highest Precedence
  * (Multiplication)
  / (Division) Truncation is the same as the / C operator.
  % (Remainder)
  <
  << (Shift Left) Same as the << C operator.
  >
  >> (Shift Right) Same as the >> C operator.
- Intermediate Precedence
  | (Bitwise Inclusive Or)
  & (Bitwise And)
  ^ (Bitwise Exclusive Or)
  ! (Bitwise Or Not)
- Lowest Precedence

+ (Addition) If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.

- (Subtraction*)* If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections. In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

2.4 Syntax for Numeric Notation

Table 2-2 Notation Syntax

Notation	Example
dec	65
bin	0b01000001
hex	0x41 or 0X41
octal	O101
ascii	'A

2.5 Special Instructions

The assembler has two special constructs for recovering the address of a symbol in data memory. %lo8data(symbol) returns the low byte of the address of symbol, and %hi8data(symbol) returns the high byte. These are useful for initializing pointers used

in the IPH/IPL, DPH/DPL, and SPH/SPL registers. Another set of constructs, %lo8insn(symbol) and %hi8insn(symbol), are used to recover addresses in program memory. These are useful for initializing pointers used in the ADDRH/ADDRL register.

2.6 Memory

The linker script file ip2kelf.ld defines the following sections:

.gpr - general-purpose registers (addresses 0x80 to 0xFF)
.data - preallocated, preinitialized data memory
.text - flash memory
.pram - program space allocated in program RAM
.pram_data - data space allocated in program RAM
.strings - strings stored in flash memory
.reset - reset vector
.config - configuration block
.bss - storage for globals. Initialized to zero.

If there is no memory assignment, data will be placed in Flash memory.

2.7 Assembly to C Calling Conventions

Calling a C function from assembly

Call the C function as below:

```
lcall _main ;precede the C function label
            ;with an _
```

2.8 IP2022-Specific Reserved Words

The following list shows all of the instruction mnemonic names and special-purpose register names. Both the uppercase and lowercase versions of these names are reserved words. Reserved words may not be used as symbolic names.

adccfg	fcfg	page	rfout	s2rcnt	t1cfg2h
adch	ferase	pch	rgdir	s2rsync	t1cfg2l
adcl	fread	pcl	rgout	s2smask	t1cmp1h
adctmr	fwrite	pop	rl	s2tbufh	t1cmp1l
add	inc	pspcfg	rr	s2tbufl	t1cmp2h
addc	incsnz	push	rtcfg	s2tcfg	t1cmp2l
addrh	incsz	radir	s1inte	s2tmrh	t1cnth
addrl	int	rain	s1intf	s2tmrl	t1cntl
and	intspd	raout	s1mode	sb	t2cap1h
break	intvech	rbdir	rttmr	setb	t2cap1l
call	intvecl	rbin	s1rbufh	snb	t2cap2h
callh	ipch	rbinte	s1rbufl	spdreg	t2cap2l
calll	ipcl	rbinted	s1rcfg	speed	t2cfg1h
clr	iph	rbintf	s1rcnt	sph	t2cfg1l
clrb	ipl	rbout	s1rsync	spl	t2cfg2h
cmp	iread	rcdir	s1smask	status	t2cfg2l
cmpcfg	iwrite	rcin	s1tbufh	sub	t2cmp1h
cse	jmp	rcout	s1tbufl	subc	t2cmp1l
csne	loadh	rddir	s1tcfg	swap	t2cmp2h
cwdt	loadl	rdin	s1tmrh	t0cfg	t2cmp2l
datah	mov	rdout	s1tmrl	t0tmr	t2cnth
datal	mulh	redir	s2inte	t1cap1h	t2cntl
dec	muls	rein	s2intf	t1cap1l	tctrl
decsnz	mulu	reout	s2mode	t1cap2h	test
decsz	nop	ret	s2rbufh	t1cap2l	wreg
dph	not	rfdir	s2rbufl	t1cfg1h	xcfg
dpl	or	rfin	s2rcfg	t1cfg1l	xor

2.9 Other Resources

This chapter only briefly describes the assembly-language syntax and the IP2022-specific features of the GNU assembler. For a complete description of the non-IP2022-specific features of the assembly-language syntax, see the *GNUPro Toolkit—GNUPro Utilities* manual.

3.0 Writing C Code

3.1 Data Types

Please consult the *GNUPro Toolkit—GNUPro Compiler Tools* manual for a complete list of all data types.

3.1.1 IP2022 specific Data Types

These are the primitive types defined by ipOS:

Basic Types

```
typedef char s8_t;
typedef unsigned char u8_t;
typedef short int s16_t;
typedef unsigned short int u16_t;
typedef long int s32_t;
typedef unsigned long int u32_t;
typedef long long int s64_t;
typedef unsigned long long int u64_t;
```

Boolean Type

```
typedef unsigned char bool_t;
```

Address Type

```
typedef unsigned short int addr_t;
```

Program RAM (PRAM) Address Type

typedef unsigned short int pram_addr_t;

Task Priority Type

typedef unsigned char priority_t;

Reference Count Type

typedef unsigned char ref_t;

3.2 Writing In-Line Assembly in C

Although it is possible to include inline assembly with in a C function, Ubicom discourages the use of inline assembly and recommends using separate stand-alone assembly functions.

3.2.1 Methods of defining assembly constants in C

1. Defining a constant that can be used in assembly:

```
asm("constant1=8");
```

2. Using a C-defined constant:

```
asm("constant1=%b0" : : "i"(UART_RX_FIFO_SIZE))
```

3.2.2 Methods of defining assembly variables in C

1. With this prototype:

```
volatile u8_t var1 __attribute__((section(".gpr")));
```

_var1 can be used in assembly and var1 can be used in C.

3.2.3 Methods of using C-defined constants in assembly

1. Where constant2 is a C-defined constant:

```
mov w,D(constant2)
```

3.2.4 Methods of using C-defined variables in assembly

1. Where var2 is a C-defined variable:

```
asm("mov w,%0" : : "d"(var2));
```

2. Where %0, %1, and %2 equal to output1, input1, and input2, respectively and output1, input1, and input2 are variables that are defined in C:

```
asm("mov w,%1 and w,%2 mov %0,w" : "d"(output1):
"d"(input1), "d"(input2));
```

3. Where input16 is a 16 bit variable; its low byte is used as %L0, and high byte is used as %H0:

```
asm("mov w,%L0 push wreg mov w,%H0" : : "i"(input16));
```

3.2.5 Methods of reaching SPR or GPR memory locations in C

1. Where S1RCFG is a SPR memory location (=0x0068):

```
*(u8_t *) S1RCFG = b01001000;
```

3.2.6 D() macro

D() is a macro which is defined in the ipOS.h header file to allow the use of C defined constants in inline assembly. Take for example the following code:

```
/*
* rtl8019_read()
* Perform an I/O read on the RTL8019.
*/ u8_t rtl8019_read8(u8_t address) __attribute__
((section (".pram")));
u8_t rtl8019_read8(u8_t address) {
    u8_t result;
    asm("
        mov w, #~$1F
        and "D(RTL8019_ADDR_PORT + RxOUT)", w
        mov w, %1 or "D(RTL8019_ADDR_PORT + RxOUT)", w
        nops "D(RTL8019_ADDR_SETUP_TIME)"
        clrb "D(RTL8019_IORD_PORT + RxOUT)",
"(RTL8019_IORD_PIN)"
        nops "D(RTL8019_READ_SETUP_TIME)"
        mov w, "D(RTL8019_DATAL_PORT + RxIN)"
        setb "D(RTL8019_IORD_PORT + RxOUT)",
"D(RTL8019_IORD_PIN)"
        mov %0, w
        "
        : "=r" (result)
        : "r" (address)
    );
    return result;
}
```

Each of the uppercase indentifiers are defines created by the configuration tool. It is not possible to simply use the defines within the inline assembly because the C preprocessor will not expand defines within strings. We need to force the define to be evaluated,

and then for it to be added to the string. The two macros D() and S() achieve this:

```
/*
 * Inline assembly define
 */
#define S(arg) #arg
#define D(arg) S(arg)
```

The D() macro forces the expression to be fully expanded by the preprocessor. The S() macro then turns its arguments into a string. In C, two adjacent strings are concatenated into a single string.

3.2.7 In-Line Assembly in C Source Files

In-line assembly code is embedded in a C source file using the **asm** statement. To prevent the compiler from re-ordering instructions during its optimization phase, keep blocks of assembly language instructions together in a single **asm** statement with the **volatile** qualifier, as shown in the following example:

```
asm volatile ("
  1:  sb     S2INTF,4
      page   1b
      jmp    1b
      clrb   S2INTF,4
      clrb   STATUS,0
      rl     w,%0
      mov    S2TBUFL,w
      mov    w,#1
```

```
        rl     wreg
        mov    S2TBUFH,w"
        :/* No output */
        :"rS" (data)
    );
```

For more information about the interface between C and in-line assembly language, see the *GNUPro Toolkit—GNUPro Compiler Tools* manual, pages 253 to 261.

3.3 C to Assembly Calling Conventions

Calling an assembly function from C

The function is written in assembly:

```
    .section .text,"ax"
    .global_my_func ;The assembly routine
                    ;label must be declared
                    ;global
    .func _my_func,_my_func
    .section .text._my_func

_my_func:

    MOV    W,#01
    XOR    RBOUT,W
    RET

    .endfunc
```

A prototype must be provided for the assembly function callmust be made, such as:

```
void my_func(void) __attribute__((naked));
```

Then the function is simply invoked by calling the function:

```
my_func();
```

The assembly routine (callee) should do the same steps a C function.

- Preserve the Frame Pointer
- Restore the Frame Pointer

For example:

```
_MyCFuncCall:
   push $fe;Preserve Frame Pointer
   mov W,SPL
    mov $fe,W
    mov W,SPH
    push $fd
   mov $fd,W
;
    mov W,$3(SP);Access passed Parameters
    mulu W,$4(SP);Multiply two Parameters
    mov $81,W;Return Low Byte
    mov W,MULH;Return High Byte
    mov $80,W
;
    pop $fd;Restore Frame Pointer
    pop $fe
    mov W,#$2
    add SPL,W

   ret;
```

4.0 Instruction Set

4.1 Instruction Format

4.2 Addressing Modes

The addressing modes are shown in Table 4-1. For a more detailed explanation of these modes, see the IP2022 Data Sheet.

Table 4-1 Addressing Modes

Addressing Mode	Assembly Language Examples	Description
Immediate	`mov w,#0xff`	Immediate operand is the literal value 0xFF (i.e. 255 decimal).
Direct	`mov w,0xff` `mov 0xff,w`	Direct operand is the global register at address 0xFF. Direct addressing can only be used for addresses between 0x01 and 0xFF (i.e. special-purpose registers and global registers).

Table 4-1 Addressing Modes (continued)

Addressing Mode	Assembly Language Examples	Description
Indirect	`mov w,(ip)` `mov (ip),w`	Indirect operand is the location in data memory addressed by the contents of the IPH/IPL register.
Indirect with Offset, Data Pointer	`mov w,8(dp)` `mov 8(dp),w`	Indirect operand is the location in data memory addressed by the contents of the DPH/DPL register plus an offset of 8. The offset is restricted to a range of 0 to 255. The operand address must be $\geq$ 0x20.
Indirect with Offset, Stack Pointer	`mov w,8(sp)` `mov 8(sp),w`	Indirect operand is the location in data memory addressed by the contents of the SPH/SPL register plus an offset of 8. The offset is restricted to a range of 0 to 255. The operand address must be $\geq$ 0x20.

4.3 Abbreviations Used

Table 4-2 Key to Abbreviations and Symbols

Symbol	Description
W	Working register
fr	File register field (an operand specified using direct addressing, indirect addressing, or indirect-with-offset addressing)
PCL	Virtual register for direct PC modification (direct address 0x09)
STATUS	STATUS register (direct address 0x0B)
IPH	Indirect Pointer High - Upper half of pointer for indirect addressing (direct address 0x04)
IPL	Indirect Pointer Low - Lower half of pointer for indirect addressing (direct address 0x05)
DPH	Upper half of data pointer for indirect-with-offset addressing (direct address 0x0C)
DPL	Lower half of data pointer for indirect-with-offset addressing (direct address 0x0D)
SPH	Upper half of stack pointer for indirect-with-offset addressing (direct address 0x06)
SPL	Lower half of stack pointer for indirect-with-offset addressing (direct address 0x07)
C	Carry bit in the STATUS register (bit 0)
DC	Digit Carry bit in the STATUS register (bit 1)
Z	Zero bit in the STATUS register (bit 2
BO	Brown-out bit in the STATUS register (bit 3)
WD	Watchdog Timeout bit in the STATUS register (bit 4)
PA2:PA0	Page bits in the STATUS register (bits 7:5)

Table 4-2 Key to Abbreviations and Symbols

Symbol	Description
WDT	Watchdog Timer counter and prescaler
,	File register/bit selector separator (e.g. **`clrb status,z`**)
lit8	8-bit immediate operand (i.e. literal) in assembly language instruction
addr13	13-bit address in assembly language instruction
(address)	Contents of memory referenced by address
|	Logical OR
||	Concatenation
^	Logical exclusive OR
&	Logical AND
!=	inequality

4.4 Summary of CPU Instructions

Table 4-3 Key to Abbreviations and Symbols

Symbol	Description
W	Working register
fr	File register field (an operand specified using direct addressing, indirect addressing, or indirect-with-offset addressing)
PCL	Virtual register for direct PC modification (direct address 0x09)
STATUS	STATUS register (direct address 0x0B)
IPH	Indirect Pointer High - Upper half of pointer for indirect addressing (direct address 0x04)

Table 4-3 Key to Abbreviations and Symbols

Symbol	Description
IPL	Indirect Pointer Low - Lower half of pointer for indirect addressing (direct address 0x05)
DPH	Upper half of data pointer for indirect-with-offset addressing (direct address 0x0C)
DPL	Lower half of data pointer for indirect-with-offset addressing (direct address 0x0D)
SPH	Upper half of stack pointer for indirect-with-offset addressing (direct address 0x06)
SPL	Lower half of stack pointer for indirect-with-offset addressing (direct address 0x07)
C	Carry bit in the STATUS register (bit 0)
DC	Digit Carry bit in the STATUS register (bit 1)
Z	Zero bit in the STATUS register (bit 2
BO	Brown-out bit in the STATUS register (bit 3)
WD	Watchdog Timeout bit in the STATUS register (bit 4)
PA2:PA0	Page bits in the STATUS register (bits 7:5)
WDT	Watchdog Timer counter and prescaler
,	File register/bit selector separator (e.g. **`clrb status,z`**)
lit8	8-bit immediate operand (i.e. literal) in assembly language instruction
addr13	13-bit address in assembly language instruction
(address)	Contents of memory referenced by address
|	Logical OR
||	Concatenation

Table 4-3 Key to Abbreviations and Symbols

Symbol	Description
^	Logical exclusive OR
&	Logical AND
!=	inequality

4.4.1 Logical Instructions

Table 4-4 Logical Instructions

Assembler Syntax	Pseudocode Definition	Description	Flags Affected
`and fr,w`	fr = fr & W	AND fr,W into fr	Z
`and w,fr`	W = W & fr	AND W,fr into W	Z
`and w,#lit8`	W = W & lit8	AND W,literal into W	Z
`not fr`	fr = $\overline{fr}$	Complement fr into fr	Z
`not w,fr`	W = $\overline{fr}$	Complement fr into W	Z
`or fr,w`	fr = fr | W	OR fr,W into fr	Z
`or w,fr`	W = W | fr	OR W,fr into W	Z
`or w,#lit8`	W = W | lit8	OR W,literal into W	Z
`xor fr,w`	fr = fr ^ W	XOR fr,W into fr	Z
`xor w,fr`	W = W ^ fr	XOR W,fr into W	Z
`xor w,#lit8`	W = W ^ lit8	XOR W,literal into W	Z

4.4.2 Arithmetic and Shift Instructions

Table 4-5 Arithmetic and Shift Instructions

Assembler Syntax	Pseudocode Definition	Description	Flags Affected
`add fr,w`	fr = fr + W	Add fr,W into fr	C, DC, Z
`add w,fr`	W = W + fr	Add W,fr into W	C, DC, Z
`add w,#lit8`	W = W + lit8	Add W,literal into W	C, DC, Z
`addc fr,w`	fr = C + fr + W	Add carry,fr,W into fr	C, DC, Z
`addc w,fr`	W = C + W + fr	Add carry,W,fr into W	C, DC, Z
`clr fr`	fr = 0	Clear fr	Z
`cmp w,fr`	fr - W	Compare W,fr then update STATUS	C, DC, Z
`cmp w,#lit8`	lit8 - W	Compare W,literal then update STATUS	C, DC, Z
`cse w,fr`	if (fr - W) = 0 then skip	Compare W,fr then skip if equal	None
`cse w,#lit8`	if (lit8 - W) = 0 then skip	Compare W,literal then skip if equal	None
`csne w,fr`	if (fr - W) != 0 then skip	Compare W,fr then skip if not equal	None
`csne w,#lit8`	if (lit8 - W) != 0 then skip	Compare W,literal then skip if not equal	None
`cwdt`	WDT = 0	Clear Watchdog Timer	None
`dec fr`	fr = fr - 1	Decrement fr into fr	Z
`dec w,fr`	W = fr -1	Decrement fr into W	Z

Table 4-5 Arithmetic and Shift Instructions (continued)

Assembler Syntax	Pseudocode Definition	Description	Flags Affected
`decsnz fr`	fr = fr - 1 if fr != 0 then skip	Decrement fr into fr then skip if not zero (STATUS not updated)	None
`decsnz w,fr`	W = fr - 1 if fr != 0 then skip	Decrement fr into W then skip if not zero (STATUS not updated)	None
`decsz fr`	fr = fr - 1 if fr = 0 then skip	Decrement fr into fr then skip if zero (STATUS not updated)	None
`decsz w,fr`	W = fr - 1 if fr = 0 then skip	Decrement fr into W then skip if zero (STATUS not updated)	None
`inc fr`	fr = fr + 1	Increment fr into fr	Z
`inc w,fr`	W = fr + 1	Increment fr into W	Z
`incsnz fr`	fr = fr + 1 if fr != 0 then skip	Increment fr into fr then skip if not zero (STATUS not updated)	None
`incsnz w,fr`	W = fr + 1 if fr != 0 then skip	Increment fr into W then skip if not zero (STATUS not updated)	None
`incsz fr`	fr = fr + 1 if fr = 0 then skip	Increment fr into fr then skip if zero (STATUS not updated)	None
`incsz w,fr`	W = fr + 1 if fr = 0 then skip	Increment fr into W then skip if zero (STATUS not updated)	None

Table 4-5 Arithmetic and Shift Instructions (continued)

Assembler Syntax	Pseudocode Definition	Description	Flags Affected
`muls w,fr`	MULH || W = W × fr	Signed 8 × 8 multiply (bit 7 = sign) W,fr into MULH || W	None
`muls w,#lit8`	MULH || W = W × lit8	Signed 8 × 8 multiply (bit 7 = sign) W,literal into MULH || W	None
`mulu w,fr`	MULH || W = W × fr	Unsigned 8 × 8 multiply W,fr into MULH || W	None
`mulu w,#lit8`	MULH || W = W × lit8	Unsigned 8 × 8 multiply W,literal into MULH || W	None
`rl fr`	fr || C = C || fr	Rotate fr left through carry into fr	C
`rl w,fr`	W || C = C || fr	Rotate fr left through carry into W	C
`rr fr`	C || fr = fr || C	Rotate fr right through carry into fr	C
`rr w,fr`	C || W = fr || C	Rotate fr right through carry into W	C
`sub fr,w`	fr = fr - W	Subtract W from fr into fr	C, DC, Z
`sub w,fr`	W = fr - W	Subtract W from fr into W	C, DC, Z
`sub w,#lit8`	W = lit8 - W	Subtract W from literal into W	C, DC, Z
`subc fr,w`	fr = fr - $\overline{C}$ - W	Subtract $\overline{carry}$,W from fr into fr	C, DC, Z
`subc w,fr`	W = fr - $\overline{C}$ - W	Subtract $\overline{carry}$,W from fr into W	C, DC, Z

Table 4-5 Arithmetic and Shift Instructions (continued)

Assembler Syntax	Pseudocode Definition	Description	Flags Affected
`swap fr`	fr = fr3:0 || fr7:4	Swap high and low nibbles of fr into fr	None
`swap w,fr`	W = fr3:0 || fr7:4	Swap high and low nibbles of fr into W	None
`test fr`	if fr = 0 then Z = 1 else Z = 0	Test fr for zero and update Z	Z

4.4.3 Bit Operation Instructions

Table 4-6 Bit Operation Instructions

Assembler Syntax	Pseudocode Definition	Description	Flags Affected
`clrb fr,bit`	fr,bit = 0	Clear bit in fr	None
`sb fr,bit`	if fr,bit = 1 then skip	Test bit in fr then skip if set	None
`setb fr,bit`	fr,bit = 1	Set bit in fr	None
`snb fr,bit`	if fr,bit = 0 then skip	Test bit in fr then skip if clear	None

4.4.4 Data Movement Instructions

Table 4-7 Data Movement Instructions

Assembler Syntax	Pseudocode Definition	Description	Flags Affected
`mov fr,w`	fr = W	Move W into fr	None
`mov w,fr`	W = fr	Move fr into W	Z
`mov w,#lit8`	W = lit8	Move literal into W	None
`push fr`	(SP) = fr, then SP = SP - 1	Move fr onto top of stack	None
`push #lit8`	(SP) = lit8, then SP = SP - 1	Move literal onto top of stack	None
`pop fr`	fr = SP + 1, then SP = SP + 1	Move top of stack + 1into fr	None

4.4.5 Program Control Instructions

Table 4-8 Program Control Instructions

Assembler Syntax	Description	Flags Affected
`call addr13`	Call subroutine	None
`jmp addr13`	Jump	None
`int`	Software interrupt	None
`nop`	No operation	None
`ret`	Return from subroutine	PA2:0
`retnp`	Return from subroutine, without updating page bits	None
`reti #lit3`	Return from interrupt	All
`retw #lit8`	Return from subroutine with literal into W	PA2:0

4.4.6 System Control Instructions

Table 4-9 System Control Instructions

Assembler Syntax	Description	Flags Affected
`break`	Software breakpoint	None
`breakx`	Software breakpoint, extending the skip	None
`ferase`	Erase a 256 word flash block	None
`fread`	Read from flash memory	None
`fwrite`	Write into flash memory	None
`iread`	Read from external/program memory	None
`ireadi`	Read from external/program memory and increment ADDRL by 2	None
`iwrite`	Write into program RAM	None
`iwritei`	Write into external/program memory and increment ADDRL by 2	None
`loadh addr8`	Load high data (byte) address into DPH	None
`loadl addr8`	Load low data (byte) address into DPL	None
`page addr3`	Load page bits from program address into PA2:0 of the STATUS	PA2:0
`speed #lit8`	Change CPU speed by writing into the SPDREG register	None

4.5 Instruction Descriptions

This section contains detailed information about the instructions for the IP2022 Processor. Each instruction is described in detail and the instruction descriptions are arranged in alphabetical order by instruction mnemonic.

ADD fr,W

Operation:	fr = fr + W
Bits affected:	C, DC, Z
Opcode:	`0001 111f ffff ffff`
Description:	This instruction adds the contents of W to the contents of the specified data memory location and writes the 8-bit result into the same data memory location. W is left unchanged. The register contents are treated as unsigned values. If the result of addition exceeds 0xFF, the C bit is set and the lower eight bits of the result are written to the data memory location. Otherwise, the C bit is cleared. If there is a carry from bit 3 to bit 4, the DC (digit carry) bit is set. Otherwise, the bit is cleared. If the result of addition is zero, the Z bit is set. Otherwise, the Z bit is cleared. A sum of 0x100 is considered zero and therefore sets the Z bit.
Cycles:	1 (3, if jumping by using PCL as the destination)

Example:

```
add 0x099,w
```

This example adds the contents of W to data memory location 0x099. For example, if the data memory location holds 0x7F and W holds 0x02, this instruction adds 0x02 to 0x7F, writes the result 0x81 into the data memory location, and clears the C and Z bits. It sets the DC bit because of the carry from bit 3 to bit 4.

ADD W,fr

Operation:	W = W + fr
Bits affected:	C, DC, Z
Opcode:	`0001 110f ffff ffff`
Description:	This instruction adds the contents of the specified data memory location to the contents of W and writes the 8-bit result into W. The data memory location is left unchanged. The register contents are treated as unsigned values. If the result of addition exceeds 0xFF, the C bit is set and the lower eight bits of the result are written to W. Otherwise, the C bit is cleared. If there is a carry from bit 3 to bit 4, the DC (digit carry) bit is set. Otherwise, the DC bit is cleared. If the result of addition is zero, the Z bit is set. Otherwise, the Z bit is cleared. A sum of 0x100 is considered zero and therefore sets the Z bit.
Cycles:	1

Example:

```
add w,0x099
```

This example adds the contents of data memory location 0x099 to W. For example, if the data memory location holds 0x81 and W holds 0x82, this instruction adds 0x81 to 0x82 and writes the lower eight bits of the result, 0x03, into W. It sets the C bit because of the carry out of bit 7, and clears the DC bit because there is no carry from bit 3 to bit 4. The Z bit is cleared because the result is nonzero.

ADD W,#lit8

Operation: W = W + lit8

Bits affected: C, DC, Z

Opcode: `0111 1011 kkkk kkkk`

Description: This instruction adds an 8-bit literal value (a value specified within the instruction) to the contents of W and writes the 8-bit result into W. The operands are treated as unsigned values.

If the result of addition exceeds 0xFF, the C bit is set and the lower eight bits of the result are written to W. Otherwise, the C bit is cleared.

If there is a carry from bit 3 to bit 4, the DC (digit carry) bit is set. Otherwise, the DC bit is cleared.

If the result of addition is zero, the Z bit is set. Otherwise, the Z bit is cleared. A sum of 0x100 is considered zero and therefore sets the Z bit.

Cycles: 1

Example:

```
add w,#0x12
```

This example adds 0x12 to the contents of W. For example, if W holds 0x82, this instruction adds 0x12 to 0x82 and writes the lower eight bits of the result, 0x94, into W. It clears the C bit because there is no carry out of bit 7, and clears the DC bit because there is no carry from bit 3 to bit 4. The Z bit is cleared because the result is nonzero.

ADDC fr,W

Operation:	fr = C + fr + W
Bits affected:	C, DC, Z
Opcode:	`0101 111f ffff ffff`
Description:	This instruction adds the contents of W and the C bit to the contents of the specified data memory location and writes the 8-bit result into the same data memory location. W is left unchanged. The register contents are treated as unsigned values. If the result of addition exceeds 0xFF, the C bit is set and the lower eight bits of the result are written to the data memory location. Otherwise, the C bit is cleared. If there is a carry from bit 3 to bit 4, the DC (digit carry) bit is set. Otherwise, the bit is cleared. If the result of addition is zero, the Z bit is set. Otherwise, the Z bit is cleared. A sum of 0x100 is considered zero and therefore sets the Z bit.
Cycles:	1

Example:

```
addc 0x099,w
```

This example adds the contents of W and the C bit to data memory location 0x099. For example, if the data memory location holds 0x7F, W holds 0x02, and the carry bit is set, this instruction adds 0x03 to 0x7F, writes the result 0x82 into the data memory location, and clears the C and Z bits. It sets the DC bit because of the carry from bit 3 to bit 4.

ADDC W,fr

Operation: W = C + W + fr

Bits affected: C, DC, Z

Opcode: `0101 110f ffff ffff`

Description: This instruction adds the contents of the specified data memory location and the C bit to the contents of W and writes the 8-bit result into W. The data memory location is left unchanged. The register contents are treated as unsigned values.

If the result of addition exceeds 0xFF, the C bit is set and the lower eight bits of the result are written to W. Otherwise, the C bit is cleared.

If there is a carry from bit 3 to bit 4, the DC (digit carry) bit is set. Otherwise, the DC bit is cleared.

If the result of addition is zero, the Z bit is set. Otherwise, the Z bit is cleared. A sum of 0x100 is considered zero and therefore sets the Z bit.

Cycles: 1

Example:

```
addc w,0x099
```

This example adds the contents of data memory location 0x099 and the C bit to W. If the data memory location holds 0x71, W holds 0x92, and the C bit is set, this instruction adds 0x72 to 0x92 and writes the lower eight bits of the result 0x04 into W. It sets the C bit because of the carry out of bit 7, clears the DC bit because there is no carry from bit 3 to bit 4, and clears the Z bit because the result is nonzero.

AND fr,W

Operation: fr = fr & W

Bits affected: Z

Opcode: `0001 011f ffff ffff`

Description: This instruction performs a bitwise logical AND of the contents of the specified data memory location and W, and writes the 8-bit result into the same data memory location. W is left unchanged. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
and 0x099,W
```

This example performs a bitwise logical AND of the working register W with a value stored in data memory location 0x099. The result is written back to the data memory location.

For example, suppose that the data memory location 0x099 holds the value 0x0F and W holds the value 0x13. The instruction takes the logical AND of 0x0F and 0x13 and writes the result 0x03 back to the data memory location. The result is nonzero, so the Z bit is cleared.

AND W,fr

Operation: W = W & fr

Bits affected: Z

Opcode: `0001 010f ffff ffff`

Description: This instruction performs a bitwise logical AND of the contents of W and the specified data memory location, and writes the 8-bit result into W. The data memory location is left unchanged. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
and w,0x099
```

This example performs a bitwise logical AND of the value stored in data memory location 0x099 with W. The result is written back to W.

For example, suppose that the data memory location 0x099 holds the value 0x0F and W holds the value 0x13. The instruction takes the logical AND of 0x0F and 0x13 and writes the result 0x03 into W. The result is nonzero, so the Z bit is cleared.

AND W,#lit8

Operation: W = W & lit8

Bits affected: Z

Opcode: `0111 1110 kkkk kkkk`

Description: This instruction performs a bitwise logical AND of the contents of W and an 8-bit literal value, and writes the 8-bit result into W. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
and w,#0x0F
```

This example performs a bitwise logical AND of W with the literal value 0x0F. The result is written back to W.

For example, suppose that W holds the value 0x50. The instruction takes the logical AND of this value with 0x0F and writes the result 0x00 into W. The result is zero, so the Z bit is set.

BREAK

Operation:	See text below.
Bits affected:	None
Opcode:	`0000 0000 0000 0001`
Description:	This instruction enters the Break mode used for software debugging. On entry into Break mode, the CPU pipeline is flushed, and program execution stops with the CPU executing a continuous series of **nop** instructions. The **break** instruction is used to suspend program execution at a specified point, so that the contents of registers can be examined through the debugging interface. Break mode can be exited only by reset or debugging commands issued through the ISD/ISP interface.
Cycles:	1
Example:	`break` Enter break mode. Not useful except when using a debugger.

BREAK

Operation: See text below.

Bits affected: None

Opcode: `0000 0000 0000 0101`

Description: This instruction enters the Break mode (and extends a skip) used for software debugging. On entry into Break mode, the CPU pipeline is flushed, and program execution stops with the CPU executing a continuous series of **nop** instructions. The **breakx** instruction is used to suspend program execution at a specified point, so that the contents of registers can be examined through the debugging interface, and then provide a skip. Break mode can be exited only by reset or debugging commands issued through the ISD/ISP interface.

The **breakx** instruction would be used only to provide a break immediately following a skip-on-condition instruction (**sb**, **snb**).

Cycles: 2

Example:

```
breakx
```

Enter break mode, extending a skip. Not useful except when using a debugger.

CALL addr13

Operation:	top-of-stack = PC15:0 + 1 PC12:0 = addr13 PC15:13 = PA2:0
Bits affected:	None
Opcode:	`110k kkkk kkkk kkkk`
Description:	This instruction calls a subroutine. The full 16-bit address of the next program instruction is saved on the stack and the program counter is loaded with a new address, which causes a jump to that program address. Bits 12:0 come from the 13-bit constant value in the instruction, and bits 15:13 come from the PA2:0 bits in the STATUS register. The subroutine is terminated by a **ret** instruction, which restores the saved address to the program counter. Execution proceeds from the instruction following the **call** instruction.
Cycles:	3

Example:

```
        page 0x600 ;set page bits
        call addxy ;call subroutine addxy
        nop ;addxy results
        ;available here
        ...
addxy: ;subroutine address label
        mov w,#0F ;subroutine begins
        add w,0x099
        ...
        ret ;return from subroutine
```

The **call** instruction in this example calls a subroutine called **addxy**. When the **call** instruction is executed, the address of the following instruction (the **nop** instruction) is pushed onto the stack and the program jumps to the **addxy** routine. When the **ret** instruction is executed, the 16-bit program address saved on the stack is popped and restored to the program counter, which causes the program to continue with the instruction immediately following the **call** instruction.

CLR fr

Operation:	fr = 0
Bits affected:	Z
Opcode:	`0000 011f ffff ffff`
Description:	This instruction clears the specified data memory location. It also sets the Z bit unconditionally.
Cycles:	1
Example:	`clr 0x099` This example clears data memory location 0x099 and sets the Z bit.

CLRB fr,bit

Operation:	fr,bit = 0
Bits affected:	None
Opcode:	`1000 bbbf ffff ffff`
Description:	This instruction clears a bit in the specified data memory location without changing the other bits in the register. The data memory location address and the bit number (0 through 7) are the instruction operands.
Cycles:	1
Example:	**`clrb 0x099,7`** This example clears the most significant bit of data memory location 0x099.

CMP W,fr

Operation: fr - W; result is discarded, STATUS is updated

Bits affected: C, DC, Z

Opcode: `0000 010f ffff ffff`

Description: This instruction subtracts the contents of W from the contents of the specified data memory location and discards the result. The data memory location and W are left unchanged. Only the STATUS register bits are updated.

If the result of subtraction is negative (W is larger than fr), the C bit is cleared and the lower eight bits of the result are written to W. Otherwise, the C bit is set.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) bit is cleared. Otherwise, the bit is set.

If the result of subtraction is zero, the Z bit is set. Otherwise, the Z bit is cleared.

Cycles: 1

Example:

```
cmp w,0x099
```

This example subtracts the contents of W from data memory location 0x099. For example, if the data memory location holds 0x35 and W holds 0x06, this instruction subtracts 0x06 from 0x35. It then sets the C bit, clears the DC bit, and clears the Z bit. The contents of the data memory location and W are left unchanged.

CMP W,#lit8

Operation: lit8 - W; result is discarded, STATUS is updated

Bits affected: C, DC, Z

Opcode: `0111 1001 kkkk kkkk`

Description: This instruction subtracts the contents of W from an 8-bit literal and discards the result. W is left unchanged. Only the STATUS register bits are updated.

If the result of subtraction is negative (W is larger than lit8), the C bit is cleared and the lower eight bits of the result are written to W. Otherwise, the C bit is set.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) bit is cleared. Otherwise, the bit is set.

If the result of subtraction is zero, the Z bit is set. Otherwise, the Z bit is cleared.

Cycles: 1

Example:

```
cmp w,#0x0F
```

This example subtracts the contents of W from 0x0F. For example, if W holds 0x06, this instruction subtracts 0x06 from 0x0F. It then clears the C bit, clears the DC bit, and clears the Z bit. The contents of W are left unchanged.

CSE W,#lit8

Operation: fr - W; if result is zero, skip next instruction

Bits affected: None

Opcode: `0100 001f ffff ffff`

Description: This instruction subtracts the contents of W from the specified data memory location and discards the result. The data memory location and W are left unchanged. If the result is zero (i.e. W and the literal are equal), the following instruction is skipped. The STATUS register bits are not updated.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
cse w,0x0F0 ;compare W with 0x0F0
ret ;if not equal, return
nop ;else, skip to here
```

This example compares the contents of W with the contents of data memory location 0x0F0. If W and the data memory location hold the same contents, then the **`ret`** instruction is executed, otherwise the **`nop`** instruction is executed.

CSE W,fr

Operation: lit8 - W; if result is zero, skip next instruction

Bits affected: None

Opcode: `0111 0111 kkkk kkkk`

Description: This instruction subtracts the contents of W from an 8-bit literal and discards the result. W is left unchanged. If the result is zero (i.e. W and the literal are equal), the following instruction is skipped. The STATUS register bits are not updated.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
cse w,#0x1B ;compare W against
            ;escape char.
jmp normal_char ;if equal, jump
            ;to normal_char
jmp escape_char ;else, jump
            ;to escape_char
```

This example compares the contents of W with the ASCII code for the escape character. If W holds 0x1B, then the **escape_char** routine is called, otherwise the **normal_char** routine is called.

CSNE W,fr

Operation: fr - W; if result is nonzero, skip next instruction

Bits affected: None

Opcode: `0100 000f ffff ffff`

Description: This instruction subtracts the contents of W from the specified data memory location and discards the result. The data memory location and W are left unchanged. If the result is nonzero (i.e. W and the literal are not equal), the following instruction is skipped. The STATUS register bits are not updated.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
csne w,0x0F0 ;compare W against
             ;register 0x0F0
ret ;if equal, return
nop ;else, skip to here
```

This example compares the contents of W with the contents of data memory location 0x0F0. If W and the data memory location hold the same contents, then the **`nop`** instruction is executed, otherwise the **`ret`** instruction is executed.

CSNE W,#lit8

Operation: lit8 - W; if result is nonzero, skip next instruction

Bits affected: None

Opcode: `0111 0110 kkkk kkkk`

Description: This instruction subtracts the contents of W from an 8-bit literal and discards the result. W is left unchanged. If the result is nonzero (i.e. W and the literal are not equal), the following instruction is skipped. The STATUS register bits are not updated.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
csne w,#0x1B ;compare W against
             ;escape char.
jmp escape_char ;if equal, jump
             ;to escape_char
jmp normal_char;else, jump to
             ;normal_char
```

This example compares the contents of W with the ASCII code for the escape character. If W holds 0x1B, then the **escape_char** routine is called, otherwise the **normal_char** routine is called.

CWDT

Operation:	Clears Watchdog timer counter and prescaler counter
Bits affected:	Z
Opcode:	`0000 0000 0000 0100`
Description:	This instruction clears the Watchdog Timer counter to zero. It also clears the Watchdog prescaler. If the Watchdog Timer is enabled, the application software must execute this instruction periodically in order to prevent a Watchdog reset.
Cycles:	1
Example:	`cwdt` This example clears the Watchdog Timer counter and the Watchdog Timer prescaler.

DEC fr

Operation:	fr = fr - 1
Bits affected:	Z
Opcode:	`0000 111f ffff ffff`
Description:	This instruction decrements the specified register file by one. If the data memory location holds 0x01, it is decremented to 0x00 and the Z bit is set. Otherwise, the bit is cleared. If the data memory location holds 0x00, it is decremented to 0xFF.
Cycles:	1
Example:	`dec 0x099` This example decrements data memory location 0x099.

DEC W,fr

Operation: W = fr - 1

Bits affected: Z

Opcode: `0000 110f ffff ffff`

Description: This instruction decrements the value in the specified register file by one and moves the 8-bit result into W. The data memory location is left unchanged.

If the data memory location holds 0x01, the value moved into W is 0x00 and the Z bit is set. Otherwise, the Z bit is cleared.

Cycles: 1

Example:

```
dec w,0x099
```

This example decrements the value in the data memory location at 0x099 and moves the result into W. For example, if the data memory location holds 0x75, the value 0x74 is loaded into W, and the Z bit is cleared. The data memory location still holds 0x75 after execution of the instruction.

DECSNZ fr

Operation:	fr = fr - 1; if nonzero result, then skip next instruction
Bits affected:	None
Opcode:	`0100 111f ffff ffff`
Description:	This instruction decrements the specified register file by one and tests the new register value. If that value is not zero, the next program instruction is skipped. Otherwise, execution proceeds normally with the next instruction.
Cycles:	1 if tested condition is false, 2 if tested condition is true
Example:	**`back1:`** **`decsnz 0x18`** **`jmp back1`** **`mov 0x19,w`** The **`decsnz`** instruction decrements data memory location 0x18. If the result is zero, execution proceeds normally with the **`jmp`** instruction to **`back1`**. If the result is nonzero, the **`jmp`** instruction is skipped, and the **`mov`** instruction is executed.

DECSNZ W,fr

Operation: W = fr - 1; if nonzero result, then skip next instruction

Bits affected: None

Opcode: `0100 110f ffff ffff`

Description: This instruction decrements the value in the specified data memory location and moves the result to W. The data memory location is left unchanged.

If the result is zero, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
decsnz w,0x099;decrement 0x099 and
              ;load result into W
ret ;return if result is 0
nop ;otherwise continue here
```

This example takes the contents of data memory location 0x099, decrements that value, and moves the result to W. If the result is nonzero, the **`ret`** instruction is skipped and the **`nop`** instruction is executed. If the result is zero, the **`ret`** instruction is executed.

DECSZ fr

Operation:	fr = fr - 1; if result is 0, then skip next instruction
Bits affected:	None
Opcode:	`0010 111f ffff ffff`
Description:	This instruction decrements the specified register file by one and tests the new register value. If that value is zero, the next program instruction is skipped. Otherwise, execution proceeds normally with the next instruction.
Cycles:	1 if tested condition is false, 2 if tested condition is true
Example:	**`back1:`** **`decsz 0x18`** **`jmp back1`** **`mov 0x19,w`** The **`decsz`** instruction decrements data memory location 0x18. If the result is nonzero, execution proceeds normally with the **`jmp`** instruction. If the result is zero, the **`jmp`** instruction is skipped and the `mov` instruction is executed.

DECSZ W,fr

Operation: W = fr - 1; if result is 0, then skip next instruction

Bits affected: None

Opcode: `0010 110f ffff ffff`

Description: This instruction decrements the value in the specified data memory location and moves the result to W. The data memory location is left unchanged.

If the result is zero, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
decsz w,0x099 ;decrement 0x099 and
              ;load result into W
ret ;return if result is 0
nop ;otherwise continue
    ;from here
```

This example takes the contents of data memory location 0x099, decrements that value, and moves the result to W. If the result is zero, the **`ret`** instruction is skipped and the **`nop`** instruction is executed. If the result is nonzero, the **`ret`** instruction is executed.

FERASE

Operation: See text below.

Bits affected: FBUSY bit in the XCFG register

Opcode: `0000 0000 0000 0011`

Description: This instruction erases a 512-byte (256-word) block of program flash memory. The ADDRH register specifies bits 15:8 of the byte address of the block. If the block is not in program flash memory, no operation is performed. The instruction is non-blocking (i.e. other instructions may execute before the erase operation is complete).

After executing this instruction, the FBUSY bit in the XCFG register is set. When the erase operation is complete, the FBUSY bit goes clear. This instruction must not be executed if the FBUSY bit is already set from a previous **`iread`**, **`ireadi`**, **`iwrite`**, **`iwritei`**, **`fread`**, **`fwrite`**, or **`ferase`** instruction. Program execution out of flash memory is not possible while the FBUSY bit is set, therefore this instruction can only be executed from program RAM.

Cycles: 1

Example:

```
mov w,#0x01 ;load W with 0x01
mov addrx,w ;copy W to ADDRX
mov w,#0x83 ;load W with 0x83
mov addrh,w ;copy W to ADDRH
ferase ;erase block
```

This example erases the 512-byte (256-word) block from byte address 0x18300 to 0x183FF.

The erase operation does not complete and the flash memory is not accessible until the FBUSY bit goes clear after the **`ferase`** instruction is executed. Therefore, any subsequent code that uses the flash memory must either check the state of the FBUSY bit or wait a sufficient number of delay cycles before proceeding.

FREAD

Operation: DATAH || DATAL = (ADDRX || ADDRH || ADDRL)

Bits affected: FBUSY bit in the XCFG register

Opcode: `0000 0000 0001 1011`

Description: This instruction transfers data from program flash memory to data memory. The 24-bit ADDRX/ADDRH/ADDRL register specifies the address of a 16-bit word in program memory which is loaded into the DATAH/DATAL register. If the address is not in program flash memory, no operation is performed. The instruction is non-blocking (i.e. other instructions may execute before the read operation is complete).

After executing this instruction, the FBUSY bit in the XCFG register is set. When the read operation is complete, the FBUSY bit goes clear. This instruction must not be executed if the FBUSY bit is already set from a previous **`iread`**, **`ireadi`**, **`iwrite`**, **`iwritei`**, **`fread`**, **`fwrite`**, or **`ferase`** instruction. Program execution out of flash memory is not possible while the FBUSY bit is set, therefore this instruction can only be executed from program RAM.

Cycles: 1

Example:

```
mov w,#0x01 ;load W with 0x01
mov addrx,w ;copy W to ADDRX
mov w,#0x83 ;load W with 0x83
mov addrh,w ;copy W to ADDRH
mov w,#0x80 ;load W with 0x80
mov addrl,w ;copy W to ADDRL
fread ;read flash memory
... ;wait for FBUSY =0 or
... ;delay for longer than
... ;flash access time
mov w,datal ;move low byte to W
mov 0xFE,w ;copy W to 0xFE
mov w,datah ;move high byte to W
mov 0xFF,w ;copy W to 0xFF
```

This example reads the 16-bit data stored at byte address 0x18380 in program flash memory. First, ADDRH and ADDRL are loaded with 0x018380. After the **`fread`** instruction, the DATAH/DATAL register holds the data stored in program memory at byte address 0x18380. Then, the lower byte is loaded into data memory location 0xFE and the upper byte is loaded into data memory location 0xFF.

FWRITE

Operation: (ADDRX || ADDRH || ADDRL) = DATAH || DATAL

Bits affected: FBUSY bit in the XCFG register

Opcode: `0000 0000 0001 1010`

Description: This instruction writes a word of data to program flash memory from data memory. The 24-bit ADDRX/ADDRH/ADDRL register specifies the address of a 16-bit word in program memory which is loaded with the contents of the DATAH/DATAL register. If the address is not in program flash memory, no operation is performed. The instruction is non-blocking (i.e. other instructions may execute before the write operation is complete).

After executing this instruction, the FBUSY bit in the XCFG register is set. When the write operation is complete, the FBUSY bit goes clear. This instruction must not be executed if the FBUSY bit is already set from a previous **`iread`**, **`ireadi`**, **`iwrite`**, **`iwritei`**, **`fread`**, **`fwrite`**, or **`ferase`** instruction. Program execution out of flash memory is not possible while the FBUSY bit is set, therefore this instruction can only be executed from program RAM.

Cycles: 1

Example:

```
mov w,#0x01 ;load W with 0x01
mov addrx,w ;copy W to ADDRX
mov w,#0x83 ;load W with 0x83
mov addrh,w ;copy W to ADDRH
mov w,#0x80 ;load W with 0x80
mov addrl,w ;copy W to ADDRL
mov w,0xFE ;copy 0xFE to W
mov datal,w ;copy W to DATAL
mov w,0xFF ;copy 0xFF to W
mov datah,w ;copy W to DATAH
fwrite ;write flash memory
```

This example writes the 16-bit data held in the DATAH/DATAL register to address 0x18380 in program flash memory. First, ADDRX/ADDRH/ADDRL is loaded with 0x018380. Then, DATAH/DATAL is loaded from 0xFE and 0xFF. Finally, the **`fwrite`** instruction loads program flash memory from DATAH/DATAL with the data that came from registers 0xFE and 0xFF.

The write operation does not complete and the flash memory is not accessible until the FBUSY bit goes clear after the **`fwrite`** instruction is executed. Therefore, any subsequent code that uses the flash memory must either check the state of the FBUSY bit or wait a sufficient number of delay cycles before proceeding.

INC fr

Operation:	fr = fr + 1
Bits affected:	Z
Opcode:	`0010 101f ffff ffff`
Description:	This instruction increments the specified register file by one. If the data memory location holds 0xFF and is incremented to 0x00, the Z bit is set. Otherwise, the Z bit is cleared.
Cycles:	1
Example:	`inc 0x099` This example increments data memory location 0x099.

INC W,fr

Operation: W = fr + 1

Bits affected: Z

Opcode: `0010 100f ffff ffff`

Description: This instruction increments the value in the specified register file by one and moves the 8-bit result into W. The data memory location is left unchanged.

If the data memory location holds 0xFF, the value moved into W is 0x00 and the Z bit is set. Otherwise, the Z bit is cleared.

Cycles: 1

Example:

```
inc w,0x099
```

This example increments the value at data memory location 0x099 and moves the result into W. For example, if the data memory location holds 0x75, the value 0x76 is loaded into W, and the Z bit is cleared. The data memory location still holds 0x75 after execution of the instruction.

INCSNZ fr

Operation:	fr = fr + 1; if result is nonzero, then skip next instruction
Bits affected:	None
Opcode:	`0101 101f ffff ffff`
Description:	This instruction increments the specified register file by one and tests the new register value. If that value is nonzero, the next program instruction is skipped. Otherwise, execution proceeds normally with the next instruction.
Cycles:	1 if tested condition is false, 2 if tested condition is true
Example:	**`back1:`** **`incsnz 0x099`** **`jmp back1`** **`mov 0x017,w`** The **`incsnz`** instruction increments data memory location 0x099. If the result is zero, execution proceeds normally with the **`jmp`** instruction to **`back1`**. If the result is nonzero, the **`jmp`** instruction is skipped and the **`mov`** instruction is executed.

INCSNZ W,fr

Operation: W = fr + 1; if result is nonzero, then skip next instruction

Bits affected: None

Opcode: `0101 100f ffff ffff`

Description: This instruction increments the value in the specified data memory location and moves the result to W. The data memory location is left unchanged.

If the result is nonzero, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
incsnz w,0x099 ;load 0x099 + 1 to W
ret ;return if 0x099 + 1 is 0
nop ;otherwise continue here
```

This example takes the contents of data memory location 0x099, increments that value, and moves the result to W. If the result is nonzero, the **`ret`** instruction is skipped and the **`nop`** instruction is executed. If the result is zero, the **`ret`** instruction is executed.

INCSZ fr

Operation:	fr = fr + 1; if result is 0, then skip next instruction
Bits affected:	None
Opcode:	`0011 111f ffff ffff`
Description:	This instruction increments the specified data memory location by one and tests the new register value. If that value is zero, the next program instruction is skipped. Otherwise, execution proceeds normally with the next instruction.
Cycles:	1 if tested condition is false, 2 if tested condition is true
Example:	**`back1:`** **`incsz 0x099`** **`jmp back1`** **`nop`** The **`incsz`** instruction increments data memory location 0x099. If the result is nonzero, execution proceeds normally with the **`jmp`** instruction to **`back1`**. If the result is zero, the **`jmp`** instruction is skipped and the **`nop`** instruction is executed.

INCSZ W,fr

Operation: W = fr + 1; if result is 0, then skip next instruction

Bits affected: None

Opcode: `0011 110f ffff ffff`

Description: This instruction increments the value in the specified data memory location and moves the result to W. The data memory location is left unchanged.

If the result is zero, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
incsz w,0x099 ;load 0x099 + 1 to W
ret ;return if 0x099 + 1 is 0
nop ;otherwise continue here
```

This example takes the contents of data memory location 0x099, increments that value, and moves the result to W. If the result is zero, the **`ret`** instruction is skipped and the **`nop`** instruction is executed. If the result is nonzero, the **`ret`** instruction is executed.

INT

Operation:	See text below
Bits affected:	None
Opcode:	`0000 0000 0000 0110`
Description:	This instruction raises an interrupt if the GIE bit is set and the INT_EN bit is clear. If the GIE bit is clear or the INT_EN bit is set, no operation is performed.
Cycles:	3
Example:	`int ;software interrupt`

IREAD

Operation:	DATAH || DATAL = (ADDRX || ADDRH || ADDRL)
Bits affected:	None
Opcode:	`0000 0000 0001 1001`
Description:	This instruction transfers data from external memory, program flash memory, or program RAM to data memory. The 24-bit ADDRX/ADDRH/ADDRL register specifies the address of a 16-bit word which is loaded into the DATAH/DATAL register. The instruction is blocking (i.e. no other instructions may execute until it completes) when it is used to read program RAM or when it is executed from program flash memory to read program flash memory. The instruction is non-blocking when it is used to read external memory or when it is executed from program RAM to read program flash memory.
Cycles:	4 (blocking)/1 (non-blocking)

Example:

```
mov w,#0x00 ;load W with 0x00
mov addrx,w ;copy W to ADDRX
mov w,#0x03 ;load W with 0x03
mov addrh,w ;copy W to ADDRH
mov w,#0x80 ;load W with 0x80
mov addrl,w ;copy W to ADDRL
iread ;read program memory
... ;(nop read access timing)
mov w,datal ;move low byte to W
mov 0xFE,w ;copy W to 0xFE
mov w,datah ;move high byte to W
mov 0xFF,w ;copy W to 0xFF
```

This example reads a word stored at byte address 0x000380 in program RAM. First, ADDRX/ADDRH/ADDRL is loaded with 0x000380. After the **iread** instruction, the DATAH/DATAL register holds the word read from program RAM. Then, the lower byte of the word is copied to data memory location 0xFE and the upper byte is copied to data memory location 0xFF.

IREADI

Operation:	DATAH || DATAL = (ADDRX || ADDRH || ADDRL) ADDRL = ADDRL + 2
Bits affected:	None
Opcode:	`0000 0000 0001 1101`
Description:	This instruction transfers data from program flash memory or program RAM to data memory. The 24-bit ADDRX/ADDRH/ADDRL register specifies the address of a 16-bit word which is loaded into the DATAH/DATAL register, then the address is incremented by 2. The instruction is blocking (i.e. no other instructions may execute until it completes) when it is used to read program RAM or when it is executed from program flash memory to read program flash memory. The instruction is non-blocking when it is executed from program RAM to read program flash memory.
Cycles:	4 (blocking)/1 (non-blocking)

Example:

```
mov w,#0x00 ;load W with 0x00
mov addrx,w ;copy W to ADDRX
mov w,#0x03 ;load W with 0x03
mov addrh,w ;copy W to ADDRH
mov w,#0x80 ;load W with 0x80
mov addrl,w ;copy W to ADDRL
ireadi ;read program memory
... ;(nop read access timing)
mov w,datal ;move low byte to W
mov 0xFE,w ;copy W to 0xFE
mov w,datah ;move high byte to W
mov 0xFF,w ;copy W to 0xFF
```

This example reads a word stored at byte address 0x000380 in program RAM. First, ADDRX/ADDRH/ADDRL is loaded with 0x000380. After the **ireadi** instruction, the DATAH/DATAL register holds the word read from program RAM. Then, the lower byte of the word is copied to data memory location 0xFE and the upper byte is copied to data memory location 0xFF.

IWRITE

Operation:	(ADDRX || ADDRH || ADDRL) = DATAH || DATAL
Bits affected:	None
Opcode:	`0000 0000 0001 1000`
Description:	This instruction transfers data from data memory to external RAM or program RAM. The 24-bit ADDRX/ADDRH/ADDRL register specifies the address of a 16-bit word which is loaded into the contents of the DATAH/DATAL register. If the address is not in external RAM or program RAM, no operation is performed. The instruction is blocking (i.e. no other instructions may execute until it completes) when it is used to write program RAM. The instruction is non-blocking when it is used to write external RAM.
Cycles:	4 (blocking)/1 (non-blocking)

Example:

```
mov w,#0x00 ;load W with 0x00
mov addrx,w ;copy W to ADDRX
mov w,#0x03 ;load W with 0x03
mov addrh,w ;copy W to ADDRH
mov w,#0x80 ;load W with 0x80
mov addrl,w ;copy W to ADDRL
mov w,0xFE ;copy 0xFE to W
mov datal,w ;copy W to DATAL
mov w,0xFF ;copy 0xFF to W
mov datah,w ;copy W to DATAH
iwrite ;write program RAM
```

This example writes the contents of the DATAH/DATAL register to address 0x000380 in program RAM. First, the ADDRX/ADDRH/ADDRL register is loaded with 0x000380. Then, the DATAH/DATAL register is loaded from 0xFE and 0xFF. Finally, the **`iwrite`** instruction loads program RAM from DATAH/DATAL.

IWRITEI

Operation:	(ADDRX || ADDRH || ADDRL) = DATAH || DATAL ADDRL = ADDRL + 2
Bits affected:	None
Opcode:	`0000 0000 0001 1100`
Description:	This instruction transfers data from data memory to program RAM. The 24-bit ADDRX/ADDRH/ADDRL register specifies the address of a 16-bit word which is loaded with the contents of the DATAH/DATAL register, then the address is incremented by 2. If the address is not in program RAM, no operation is performed. The instruction is blocking (i.e. no other instructions may execute until it completes) when it is used to write program RAM.
Cycles:	4 (blocking)/1 (non-blocking)

Example:

```
mov w,#0x00 ;load W with 0x00
mov addrx,w ;copy W to ADDRX
mov w,#0x03 ;load W with 0x03
mov addrh,w ;copy W to ADDRH
mov w,#0x80 ;load W with 0x80
mov addrl,w ;copy W to ADDRL
mov w,0xFE; copy 0xFE to W
mov datal,w ;copy W to DATAL
mov w,0xFF ;copy 0xFF to W
mov datah,w ;copy W to DATAH
iwritei ;write program RAM and
        ;increment ADDRL
```

This example writes the contents of the DATAH/DATAL register to address 0x000380 in program RAM. First, the ADDRX/ADDRH/ADDRL register is loaded with 0x000380. Then, the DATAH/DATAL register is loaded from 0xFE and 0xFF. Finally, the **`iwritei`** instruction loads program RAM from DATAH/DATAL.

JMP addr13

Operation:	PC12:0 = addr13 PC15:13 = PA2:PA0
Bits affected:	None
Opcode:	`111k kkkk kkkk kkkk`
Description:	This instruction causes the program to jump to a specified address. It loads the program counter with the new address. Bits 12:0 come from the 13-bit constant value in the instruction, and bits 15:13 come from the PA2:PA0 bits in the STATUS register. The STATUS register must hold the appropriate value prior to the jump instruction.
Cycles:	3

Example:

```
snb STATUS,0 ;skip if carry clear
page 0x6000 ;set page bits
jmp overflo ;jump to overflo
nop ;continue here if no jump
overflo: ;jump destination
nop ;continue here if jump taken
...
```

This example shows one way to implement a conditional jump. The **jmp** instruction, if executed, causes a jump to **overflo**. The **snb** instruction (test bit and skip if clear) causes the **jmp** instruction to be either executed or skipped, depending on the state of the carry bit (bit 0 of the STATUS register). Because the skip instruction is immediately followed by a **page** instruction, two instructions (**page** and **jmp**) will be skipped if the carry bit is clear. The PA2:PA0 bits of the STATUS register must hold the three high-order bits (bits 15:13) of the word address of the **overflo** entry point prior to the **jmp** instruction. This is the purpose of the **page** instruction.

LOADH addr8

Operation: DPH = addr8(15:8)

Bits affected: None

Opcode: `0111 0000 kkkk kkkk`

Description: This instruction loads the high 8 bits of a data address into DPH. A skip over a **`loadh`** instruction results in an extended skip (following instruction also skipped).

From assembly language, a 16-bit address is specified. The assembler takes the high 8 bits of the address and encodes it in the instruction. The low 8 bits are ignored.

Cycles: 1

Example:

```
MyRegisters = 0x048A ;define symbolic
                     ;address
      loadh MyRegisters ;load 0x02 to DPH
      loadl MyRegisters ;load 0x45 to DPL
```

The first line of this example is a declaration that the symbolic address **`MyRegisters`** is the byte address 0x048A (word address 0x0245). The second line loads DPH with the high 8 bits of the word address for **`MyRegisters`**. The third line loads DPL with the low 8 bits of the word address for **`MyRegisters`**.

LOADL addr8

Operation: DPL = addr8(7:0)

Bits affected: None

Opcode: `0111 0001 kkkk kkkk`

Description: This instruction loads the low 8 bits of a data address into DPH. A skip over a **`loadl`** instruction results in an extended skip (following instruction also skipped).

From assembly language, a 16-bit address is specified. The assembler takes the low 8 bits of the address and encodes it in the instruction. The high 8 bits are ignored.

Cycles: 1

Example:

```
MyRegisters = 0x048A ;define symbolic
                     ;address
      loadh MyRegisters ;load 0x02 to DPH
      loadl MyRegisters ;load 0x45 to DPL
```

The first line of this example is a declaration that the symbolic address **`MyRegisters`** is the byte address 0x048A (word address 0x0245). The second line loads DPH with the high 8 bits of the word address for **`MyRegisters`**. The third line loads DPL with the low 8 bits of the word address for **`MyRegisters`**.

MOV fr,W

Operation:	fr = W
Bits affected:	None
Opcode:	`0000 001f ffff ffff`
Description:	This instruction moves the contents of W into the specified data memory location. W is left unchanged.
Cycles:	1
Example:	`mov 0x099,w ;move W to 0x099`
	This example moves the contents of W into data memory location 0x099.

MOV W,fr

Operation:	W = fr
Bits affected:	None
Opcode:	`0010 000f ffff ffff`
Description:	This instruction moves the contents of the specified data memory location into W. The data memory location is left unchanged. If the data is zero, the Z bit is set. Otherwise, the Z bit is cleared.
Cycles:	1
Example:	`mov w,0x099 ;move register to W` This example moves the contents of the data memory location at address 0x099 into W. The Z bit is set if the value is zero or cleared if the value is nonzero.

MOV W,#lit8

Operation: W = lit8

Bits affected: None

Opcode: `0111 1100 kkkk kkkk`

Description: This instruction loads an 8-bit literal into W.

Cycles: 1

Example:

```
mov w,#0x75
```

This example loads the literal 0x75 into W.

MULS W,fr

Operation:	MULH || W = W x fr
Bits affected:	None
Opcode:	`00101 010f ffff ffff`
Description:	This instruction performs a signed multiply of the contents of the specified data memory location by the contents of W. The low 8 bits of the product are written to W, and the high 8 bits are written to the MULH register. The data memory location is left unchanged. Both operands are interpreted as two's-complement numbers, and the result loaded into MULH || W is also in two's complement format.
Cycles:	1

Example:

```
muls w,0x099 ;signed multiply,
;0x099 by W into MULH and W
```

This example multiplies the contents of the data memory location at address 0x099 by the contents of W into W. The high byte of the result is loaded into the MULH register, and the low byte is loaded into W.

If W holds 0x07 and 0x099 holds 0x06, the result loaded into MULH || W will be 0x002A (42 decimal).

If W holds 0xF9 (-7 decimal) and 0x099 holds 0x06, the result loaded into MULH || W will be 0xFFD6 (-42 decimal).

If W holds 0xF9 (-7 decimal) and 0x099 holds 0xFA (-6 decimal), the result loaded into MULH || W will be 0x002A (42 decimal).

MULS W,#lit8

Operation: MULH || W = W × lit8

Bits affected: None

Opcode: `0111 0011 kkkk kkkk`

Description: This instruction performs a signed multiply of an 8-bit literal by the contents of W. The low 8 bits of the product are written to W, and the high 8 bits are written to the MULH register. The data memory location is left unchanged.

Both operands are interpreted as two's-complement numbers, and the result loaded into MULH || W is also in two's complement format.

Cycles: 1

Example:

```
muls w,#0x75
```

This example multiplies the contents of W by the literal 0x75 into W. The high byte of the result is loaded into the MULH register, and the low byte is loaded into W.

If W holds 0x07 and lit8 is 0x06, the result loaded into MULH || W will be 0x002A (42 decimal).

If W holds 0xF9 (-7 decimal) and lit8 is 0x06, the result loaded into MULH || W will be 0xFFD6 (-42 decimal).

If W holds 0xF9 (-7 decimal) and lit8 is 0xFA (-6 decimal), the result loaded into MULH || W will be 0x002A (42 decimal).

MULU W,fr

Operation: MULH || W = W × fr; high 8 bits of unsigned product

Bits affected: None

Opcode: `0101 000f ffff ffff`

Description: This instruction performs an unsigned multiply of the contents of the specified data memory location by the contents of W. The low 8 bits of the product are written to W, and the high 8 bits are written to the MULH register. The data memory location is left unchanged.

Cycles: 1

Example:

```
mulu w,0x099 ;unsigned multiply,
;0x099 by W into W
```

This example multiplies the contents of the data memory location at address 0x099 by the contents of W into W. The high byte of the result is loaded into the MULH register, and the low byte is loaded into W.

MULU W,#lit8

Operation:	MULH || W = W × lit8
Bits affected:	None
Opcode:	`0111 0010 kkkk kkkk`
Description:	This instruction performs an unsigned multiply of an 8-bit literal by the contents of W. The low 8 bits of the product are written to W, and the high 8 bits are written to the MULH register. The data memory location is left unchanged.
Cycles:	1
Example:	**`mulu w,#0x75`** This example multiplies the contents of W by the literal 0x75 into W. The high byte of the result is loaded into the MULH register, and the low byte is loaded into W.

NOP

Operation: None

Bits affected: None

Opcode: `0000 0000 0000 0000`

Description: This instruction does nothing except to cause a one-cycle delay in program execution.

Cycles: 1

Example:

```
sb 0x05,4 ;set bit 4 in Port A
nop ;no operation, 1-cycle delay
sb 0x05,6 ;set bit 5 in Port A
```

This example shows how a **nop** instruction can be used as a one-cycle delay between two successive read-modify-write instructions that modify the same I/O port. This delay ensures reliable results at high clock rates.

NOT fr

Operation: fr = $\overline{fr}$

Bits affected: Z

Opcode: `0010 011f ffff ffff`

Description: This instruction complements each bit of the specified data memory location and writes the result back into the same register. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
not 0x099 ;complement 0x099
```

Suppose that data memory location 0x099 holds the value 0x1C. This instruction takes the complement of 0x1C and writes the result 0xE3 back to location 0x099. The result is nonzero, so the Z bit is cleared.

NOT W,fr

Operation: W = $\overline{fr}$

Bits affected: Z

Opcode: `0010 010f ffff ffff`

Description: This instruction loads the one's complement of the specified data memory location into W. The data memory location is left unchanged.

If the value loaded into W is zero, the Z bit is set. Otherwise, the bit is cleared.

Cycles: 1

Example:

```
mov w,0x099
```

This example moves the one's complement of data memory location 0x099 into W. For example, if the data memory location holds 0x75, the complement of this value 0x8A is loaded into W, and the Z bit is cleared. The data memory location is left unchanged.

OR fr,W

Operation: fr = fr | W

Bits affected: Z

Opcode: `0001 001f ffff ffff`

Description: This instruction performs a bitwise logical OR of the contents of the specified data memory location and W, and writes the 8-bit result into the same data memory location. W is left unchanged. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
or 0x099,w ;move fr OR W to fr
```

This example performs a bitwise logical OR of W with a value stored in data memory location 0x099. The result is written back to the data memory location 0x099.

For example, suppose that the data memory location 0x099 holds the value 0x0F and W holds the value 0x13. The instruction takes the logical OR of 0x0F and 0x13 and writes the result 0x1F back to data memory location 0x099. The result is nonzero, so the Z bit is cleared.

OR W,fr

Operation: W = W | fr

Bits affected: Z

Opcode: `0001 000f ffff ffff`

Description: This instruction performs a bitwise logical OR of the contents of W and the specified data memory location, and writes the 8-bit result into W. The data memory location is left unchanged. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
or w,0x099 ;move W OR fr to W
```

This example performs a bitwise logical OR of the value stored in data memory location 0x099 with W. The result is written back to W.

For example, suppose that the data memory location 0x099 holds the value 0x0F and W holds the value 0x13. The instruction takes the logical OR of 0x0F and 0x13 and writes the result 0x1F into W. The result is nonzero, so the Z bit is cleared.

OR W,#lit8

Operation: W = W | lit8

Bits affected: Z

Opcode: `0111 1101 kkkk kkkk`

Description: This instruction performs a bitwise logical OR of the contents of W and an 8-bit literal value, and writes the 8-bit result into W. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
or w,#0x0F ;set low four W bits
```

This example performs a bitwise logical OR of W with the literal value 0x0F. The result is written back to W.

For example, suppose that W holds the value 0x50. The instruction takes the logical OR of this value with 0x0F and writes the result 0x5F into W. The result is nonzero, so the Z bit is cleared.

PAGE addr3

Operation: PA2:0 = addr(16:14)

Bits affected: PA2:0

Opcode: `0000 0000 0001 0nnn`

Description: This instruction writes a three-bit value into the PA2:0 bits of the STATUS register (bits 7:5). These bits select the program memory page for subsequent jump and subroutine call instructions.

In assembly language, the full program memory address is specified. The assembler encodes the three high-order bits of this address into the instruction opcode and ignores the fourteen low-order bits.

If a skip instruction is immediately followed by a `page` instruction and the tested condition is true, then two instructions are skipped and the operation consumes three cycles. This is useful for conditional branching to another page in which a **`page`** instruction precedes a **`jmp`** instruction. If several **`page`** instructions immediately follow a skip instruction then they are all skipped plus the next instruction, and a cycle is consumed for each.

Cycles: 1

Example:

```
page 0x8000 ;load PA2:0 with 010
call home1 ;jump to page 2
```

This example sets the PA2:0 bits in the STATUS register to 010. This means that the subsequent **`call`** instruction calls a subroutine that starts in the address range of byte address 0x8000 to 0x9FFE (word address 0x4000 to 0x4FFF) .

POP fr

Operation: fr = (SP + 1), then SP = SP + 1

Bits affected: None

Opcode: `0100 011f ffff ffff`

Description: This instruction increments the SPH/SPL register, then copies the register addressed by the SPH/SPL register to the specified data memory location. This stack is independent of the hardware stack used for subroutine calls and returns.

Cycles: 1

Example:

```
pop 0x0F0 ;pop stack to 0x0F0
```

The **pop** instruction in this example pops a byte off the stack and loads it into the data memory location at address 0x0F0.

PUSH fr

Operation:	(SP) = fr, then SP = SP - 1
Bits affected:	None
Opcode:	`0100 010f ffff ffff`
Description:	This instruction copies the specified data memory location to the register addressed by the SPH/SPL register, then decrements the SPH/SPL register. This stack is independent of the hardware stack used for subroutine calls and returns.
Cycles:	1
Example:	**`push 0x0F0 ;push 0x0F0 onto stack`** The **`push`** instruction in this example pushes the contents of the data memory location at address 0x1F0 onto the stack.

PUSH #lit8

Operation: (SP) = lit8, then SP = SP - 1

Bits affected: None

Opcode: `0111 0100 kkkk kkkk`

Description: This instruction copies an 8-bit literal to the register addressed by the SPH/SPL register, then decrements the SPH/SPL register. This stack is independent of the stack used for subroutine calls and returns.

Cycles: 1

Example:

```
push 0x#FF ;push #0xFF onto stack
```

The **`push`** instruction in this example pushes 0xFF onto the stack.

RET

Operation:	program counter = top-of-stack
Bits affected:	PA2:0
Opcode:	`0000 0000 0000 0111`
Description:	This instruction causes a return from a subroutine. It pops the 16-bit value previously stored on the stack and restores that value to the program counter. This causes the program to jump to the instruction immediately following the **`call`** instruction that called the subroutine. The hardware stack used for subroutine call and return is independent from the stack used with the **`push`** and **`pop`** instructions. The **ret** instruction modifies the PA2:0 bits in the STATUS register to return to the correct place in the program.
Cycles:	3

Example:

```
        page 0x0000 ;set page bits
        calladdxy ;call subroutine addxy
        nop ;addxy results available here
        ...
addxy: ;subroutine entry point
        mov w,0x099 ;subroutine start
        add w,0x00F
        ...
        ret ;return from subroutine
```

The **call** instruction in this example calls a subroutine called **addxy**. When the **call** instruction is executed, the address of the following instruction (the **nop** instruction) is pushed onto the stack and the program jumps to the **addxy** routine. When the **ret** instruction is executed, the saved program address is popped from the stack and restored to the program counter, which causes the program to continue with the instruction immediately following the **call** instruction.

RETI #lit3

Operation: restore CPU registers from shadow registers

Bits affected: STATUS register restored, which affects all bits

Opcode: `0000 0000 0000 1nnn`

Description: This instruction causes a return from an interrupt service routine. It restores the 16-bit program counter value that was saved when the interrupt occurred. In addition, there are three instruction options encoded by the three low-order bits of the instruction, as shown in the table below.

Bit	Function
2	Reinstate the pre-interrupt speed 1 = enable, 0 = disable
1	Store the PC+1 value in the INTVECH and INTVECL registers 1 = enable, 0 = disable
0	Add W to the T0TMR register 1 = enable, 0 = disable

Cycles: 3

Example:

```
org 0 ;set address to 0x000
... ;ISR code goes here
reti #0x4 ;return from interrupt
```

This is an example of an interrupt service routine. When an interrupt occurs, the CPU registers are saved into a set of shadow registers. The program then jumps to the interrupt service routine, which starts at a default address of 0x000 (software can change this address by loading a new interrupt vector into the INTVECH and INTVECL register pair). The interrupt service routine should determine the cause of the interrupt, clear the interrupt flag bit for the event that raised the interrupt, perform any required service for that event, and end with the **`reti`** instruction.

The **`reti`** instruction restores the contents of the program counter from the shadow registers. This causes the IP2022 to continue program execution from the point at which the program was interrupted.

In this example, the immediate operand specifies restoration of the pre-interrupt speed, but no modification is made to the interrupt vector or the T0TMR register.

RETNP

Operation:	program counter = top-of-stack
Bits affected:	None
Opcode:	`0000 0000 0000 0010`
Description:	This instruction causes a return from a subroutine. It pops the 16-bit value previously stored on the stack and restores that value to the program counter. This causes the program to jump to the instruction immediately following the **`call`** instruction that called the subroutine. The hardware stack used for subroutine call and return is independent from the stack used with the **`push`** and **`pop`** instructions. The **retnp** instruction does not use (and does not affect) the PA2:0 bits.
Cycles:	3

Example:

```
        page 0x0000 ;set page bits
        calladdxy ;call subroutine addxy
        nop ;addxy results available here
        ...
addxy: ;subroutine entry point
        mov w,0x099 ;subroutine start
        add w,0x00F
        ...
        ret ;return from subroutine
```

The **call** instruction in this example calls a subroutine called **addxy**. When the **call** instruction is executed, the address of the following instruction (the **nop** instruction) is pushed onto the stack and the program jumps to the **addxy** routine. When the **ret** instruction is executed, the saved program address is popped from the stack and restored to the program counter, which causes the program to continue with the instruction immediately following the **call** instruction.

RETW #lit8

Operation: W = lit8

program counter = top-of-stack

Bits affected: PA2:0

Opcode: `0111 1000 kkkk kkkk`

Description: This instruction loads an 8-bit literal into W and causes a return from a subroutine. The literal can be used to implement lookup tables. The instruction pops the 16-bit value previously stored on the stack and restores that value to the program counter. This causes the program to jump to the instruction immediately following the **`call`** instruction that called the subroutine. The hardware stack used for subroutine call and return is independent from the stack used with the **`push`** and **`pop`** instructions.

It is not necessary to set the PA2:0 bits in the STATUS register to return to the correct place in the program. This is because the full 16-bit program address is restored from the stack. The **`ret`** instruction does not use (and does not affect) the PA2:0 bits.

Cycles: 3

Example:

```
mov w,0x0F0 ;load W with index
add pcl,w ;add index to the pc
retw #0xFF ;if index is 0,
          ;return with 0xFF in W
retw #0xF0 ;if index is 1,
          ;return with 0xF0 in W
retw #0x0F ;if index is 2,
          ;return with 0x0F in W
retw #0x00 ;if index is 3,
          ;return with 0x00 in W
```

This example shows an index being read from data memory location 0x0F0 and being added to the program counter, which causes a jump into an array of **retw** instructions. Depending on what the index is, one of the **retw** instructions will be executed. Each **retw** instruction returns a different value in W.

RL fr

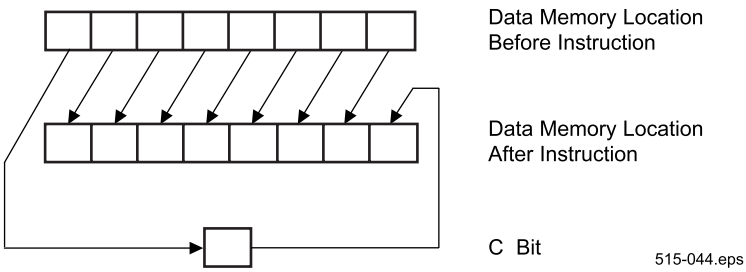
Operation: fr || C = C || fr

Bits affected: C

Opcode: `0011 011f ffff ffff`

Description: This instruction rotates the bits of the specified data memory location left through the C bit and moves the 8-bit result into the data memory location.

The bits obtained from the register are shifted left by one bit position. C is shifted into the least significant bit position and the most significant bit is shifted out into C, as shown in the diagram below.



Cycles: 1

Example:

```
rl 0x099
```

This example rotates the bits of data memory location 0x099 left through the C bit. If the data memory location holds 0x14 and the C bit is set, after this instruction is executed, the data memory location will hold 0x29 and the C bit will be clear.

RL W,fr

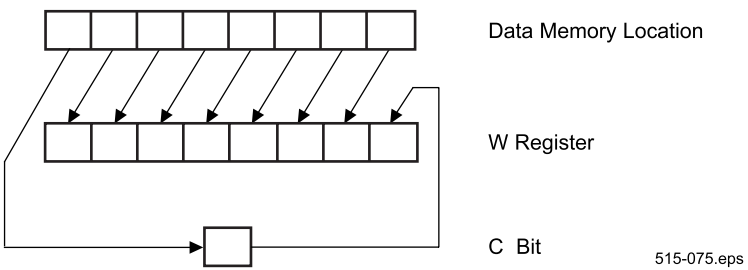
Operation: W || C = C || fr

Bits affected: C

Opcode: `0011 010f ffff ffff`

Description: This instruction rotates the bits of the specified data memory location left through the C bit and moves the 8-bit result into W. The data memory location is left unchanged.

The bits obtained from the register are shifted left by one bit position. C is shifted into the least significant bit position and the most significant bit is shifted out into C, as shown in the diagram below.



Cycles: 1

Example:

```
rl w,0x099
```

This example rotates the bits of data memory location 0x099 left through the C bit and moves the result into W. If the data memory location holds 0x14 and the C bit is set, after this instruction is executed, W will hold 0x29 and the C bit will be clear. The data memory location will still hold 0x14 after execution of the instruction.

RR fr

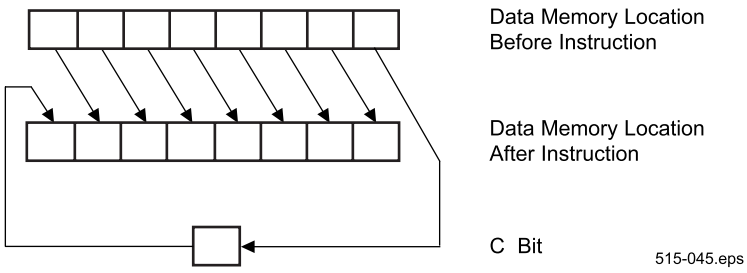
Operation: C || fr = fr || C

Bits affected: C

Opcode: `0011 001f ffff ffff`

Description: This instruction rotates the bits of the specified data memory location right through the C bit and moves the 8-bit result into the data memory location.

The bits obtained from the register are shifted right by one bit position. C is shifted into the most significant bit position and the least significant bit is shifted out into C, as shown in the diagram below.



Cycles: 1

Example:

```
rr 0x099
```

This example rotates the bits of data memory location 0x099 right through the C bit. If the data memory location holds 0x12 and the C bit is set, after this instruction is executed, the data memory location will hold 0x89 and the C bit will be clear.

RR W,fr

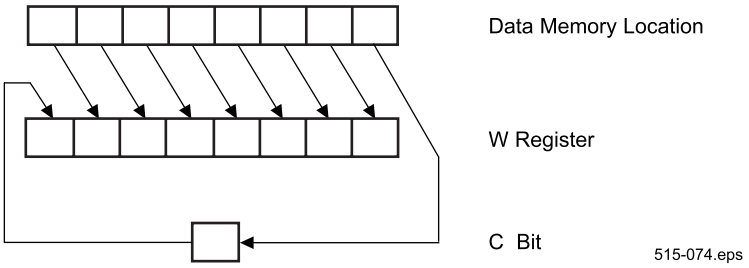
Operation: C || W = fr || C

Bits affected: C

Opcode: `0011 000f ffff ffff`

Description: This instruction rotates the bits of the specified data memory location right through the C bit and moves the 8-bit result into W. The data memory location is left unchanged.

The bits obtained from the register are shifted right by one bit position. C is shifted into the most significant bit position and the least significant bit is shifted out into C, as shown in the diagram below.



Cycles: 1

Example:

```
rr w,0x099
```

This example rotates the bits of data memory location 0x099 right through the C bit and moves the result into W. If the data memory location holds 0x12 and the C bit is set, after this instruction is executed, W will hold 0x89 and the C bit will be clear. The data memory location will still hold 0x12 after execution of the instruction.

SB fr,bit

Operation:	if fr,bit = 1, skip next instruction
Bits affected:	None
Opcode:	`1011 bbbf ffff ffff`
Description:	This instruction tests a bit in the specified data memory location. The data memory location and the bit number (0 through 7) are the instruction operands. If the bit is 1, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.
Cycles:	1 if tested condition is false, 2 if tested condition is true
Example:	**`sb STATUS,0 ;test carry bit`** **`inc 0x099 ;increment if carry=0`** **`nop`**

This example tests the carry bit (bit 0 of the STATUS register). If the bit is 1, the **`inc`** instruction is skipped and the **`nop`** instruction is executed. Otherwise, program execution proceeds normally with the **`inc`** instruction.

SETB fr,bit

Operation:	fr,bit = 1
Bits affected:	None
Opcode:	`1001 bbbf ffff ffff`
Description:	This instruction sets a bit in the specified data memory location to 1 without changing the other bits in the register. The data memory location and the bit number (0 through 7) are the instruction operands.
Cycles:	1
Example:	**`setb 0x099,7`** This example sets the most significant bit of data memory location 0x099.

SNB fr,bit

Operation: if fr,bit = 0, skip next instruction

Bits affected: None

Opcode: `1010 bbbf ffff ffff`

Description: This instruction tests a bit in the specified data memory location. The data memory location and the bit number (0 through 7) are the instruction operands. If the bit is 0, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
snb STATUS,0 ;test carry bit
inc 0x099 ;increment if carry=1
nop
```

This example tests the carry bit (bit 0 of the STATUS register). If that bit is 0, the **`dec`** instruction is skipped and the **`nop`** instruction is executed. Otherwise, program execution proceeds normally with the **`inc`** instruction.

SPEED #lit8

Operation: SPDREG = lit8

Bits affected: None

Opcode: `0000 0001 nnnn nnnn`

Description: This instruction writes an 8-bit value into the SPDREG register. This register controls the power-down options, system clock source, and clock divisor used to generate the CPU core clock from the system clock. The format of the SPDREG register is shown below.

7 6	5 4	3 0
PWRD1:0	CLK1:0	CDIV3:0

PWRD1:0 - Controls whether the PLL clock multiplier and OSC oscillator are running.

CLK1:0 - Selects the system clock source.

CDIV3:0 - Selects the clock divisor used to generate the system clock.

Cycles: 1 instruction cycle (as opposed to clock cycles)

Example:

```
nop ;assume divisor is 4, so
        ;instruction takes 4 cycles
speed div8 ;change divisor to 8,
         ;instruction takes 4 cycles
nop ;instruction takes 8 cycles
speed div1 ;change divisor to 1,
         ;instruction takes 8 cycles
nop ;instruction takes 1 cycle
```

In this example, **div1** and **div8** are assumed to be constants defined with appropriate bit settings for the SPDREG register encoding.

If the clock divisor prior to the first **speed** instruction is 4, the first **nop** and **speed** instructions each take 4 clock cycles.

The first **speed** instruction changes the clock divisor to 8, so the second **nop** and **speed** instructions each take 8 clock cycles.

The second **speed** instruction changes the clock divisor to 1, so the third **nop** instruction takes 1 clock cycle.

SUB fr,W

Operation: fr = fr - W

Bits affected: C, DC, Z

Opcode: `0000 101f ffff ffff`

Description: This instruction subtracts the contents of W from the contents of the specified data memory location and writes the 8-bit result into the same data memory location. W is left unchanged. The register contents are treated as unsigned values.

If the result of subtraction is negative (W is larger than fr), the C bit is cleared and the lower eight bits of the result are written to the data memory location. Otherwise, the C bit is set.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) bit is cleared. Otherwise, the bit is set.

If the result of subtraction is zero, the Z bit is set. Otherwise, the bit is cleared.

Cycles: 1

Example 1:

```
sub 0x099,w
```

This example subtracts the contents of W from data memory location 0x099. For example, if the data memory location holds 0x35 and W holds 0x06, this instruction subtracts 0x06 from 0x35 and writes the result 0x2F into the data memory location. It also sets the C bit, clears the DC bit, and clears the Z bit.

Example 2:

```
mov w,0x095 ;load W from 0x095
sub 0x097,w; subtract low bytes
;C = 0 for borrow out
mov w,0x096 ;load W from 0x096
subc 0x098,w ;subtract high bytes
```

This example performs 16-bit subtraction of data memory locations 0x095 (low byte) and 0x096 (high byte) from data memory locations 0x097 (low byte) and 0x098 (high byte).

The **sub** instruction subtracts the contents of 0x095 from 0x097 and clears the C bit if a borrow occurs out of bit 7, or sets the C bit otherwise. The **subc** instruction subtracts the contents of 0x096 from 0x098 with borrow-in using the C bit.

SUB W,fr

Operation: W = fr - W

Bits affected: C, DC, Z

Opcode: `0000 100f ffff ffff`

Description: This instruction subtracts the contents of W from the contents of the specified data memory location and writes the 8-bit result into W. The data memory location is left unchanged. The register contents are treated as unsigned values.

If the result of subtraction is negative (W is larger than fr), the C bit is cleared and the lower eight bits of the result are written to W. Otherwise, the C bit is set.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) bit is cleared. Otherwise, the bit is set.

If the result of subtraction is zero, the Z bit is set. Otherwise, the Z bit is cleared.

Cycles: 1

Example:

```
sub w,0x099
```

This example subtracts the contents of W from data memory location 0x099 and moves the result into W. For example, if the data memory location holds 0x35 and W holds 0x06, this instruction subtracts 0x06 from 0x35 and writes the result 0x2F into W. It also sets the C bit, clears the DC bit, and clears the Z bit. The data memory location is left unchanged.

SUB W,#lit8

Operation: W = lit8 - W

Bits affected: C, DC, Z

Opcode: `0111 1010 kkkk kkkk`

Description: This instruction subtracts the contents of W from an 8-bit literal and writes the 8-bit result into W. The register contents are treated as unsigned values.

If the result of subtraction is negative (W is larger than lit8), the C bit is cleared and the lower eight bits of the result are written to W. Otherwise, the C bit is set.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) bit is cleared. Otherwise, the bit is set.

If the result of subtraction is zero, the Z bit is set. Otherwise, the bit is cleared.

Cycles: 1

Example:

```
sub w,#0xFF
```

This example subtracts the contents of W from 0xFF. For example, if W holds 0x06, this instruction subtracts 0x06 from 0xFF and writes the result 0xF9 into W. It also sets the C bit, clears the DC bit, and clears the Z bit.

SUBC fr,W

Operation: fr = fr - $\overline{C}$ - W

Bits affected: C, DC, Z

Opcode: `0100 101f ffff ffff`

Description: This instruction subtracts the contents of W and the complement of the C bit (which indicates borrow) from the contents of the specified data memory location and writes the 8-bit result into the same data memory location. W is left unchanged. The register contents are treated as unsigned values.

If the result of subtraction is negative (W + C is larger than fr), the C bit is cleared and the lower eight bits of the result are written to the data memory location. Otherwise, the C bit is set.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) bit is cleared. Otherwise, the bit is set.

If the result of subtraction is zero, the Z bit is set. Otherwise, the bit is cleared.

Cycles: 1

Example 1:

```
subc 0x099,w
```

This example subtracts the contents of W and the complement of the C bit from data memory location 0x099. For example, if the data memory location holds 0x35, W holds 0x06, and the C bit is clear, this instruction subtracts 0x07 from 0x35 and writes the result 0x2E into the data memory location. It also sets the C bit, clears the DC bit, and clears the Z bit.

Example 2:

```
mov w,0x095 ;load W from 0x095
sub 0x097,w ;subtract low bytes
;C = 0 for borrow out
mov w,0x096 ;load W from 0x096
subc 0x098,w ;subtract high bytes
```

This example performs 16-bit subtraction of data memory locations 0x095 (low byte) and 0x096 (high byte) from data memory locations 0x097 (low byte) and 0x098 (high byte).

The **`sub`** instruction subtracts the contents of 0x095 from 0x097 and clears the C bit if a borrow occurs out of bit 7, or sets the C bit otherwise. The **`subc`** instruction subtracts the contents of 0x096 from 0x097 with borrow-in using the C bit.

SUBC W,fr

Operation: W = fr - $\overline{C}$ - W

Bits affected: C, DC, Z

Opcode: `0100 100f ffff ffff`

Description: This instruction subtracts the contents of W and the complement of the C bit (which indicates borrow) from the contents of the specified data memory location and writes the 8-bit result into W. The data memory location is left unchanged. The register contents are treated as unsigned values.

If the result of subtraction is negative (W is larger than fr), the C bit is cleared and the lower eight bits of the result are written to W. Otherwise, the C bit is set.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) bit is cleared. Otherwise, the bit is set.

If the result of subtraction is zero, the Z bit is set. Otherwise, the Z bit is cleared.

Cycles: 1

Example:

```
subc w,0x099
```

This example subtracts the contents of W from data memory location 0x099 and moves the result into W. For example, if the data memory location holds 0x35, W holds 0x06, and the C bit is clear, this instruction subtracts 0x07 from 0x35 and writes the result 0x2E into W. It also sets the C bit, clears the DC bit, and clears the Z bit. The data memory location is left unchanged.

SWAP fr

Operation:	fr = fr3:0 || fr7:4
Bits affected:	None
Opcode:	`0011 101f ffff ffff`
Description:	This instruction exchanges the high-order and low-order nibbles (4-bit fields) of the specified data memory location.
Cycles:	1
Example:	**`swap 0x099`** This example swaps the high-order and low-order nibbles of data memory location 0x099. For example, if the memory location holds 0xA5, after executing this instruction, it will hold 0x5A.

SWAP W,fr

Operation:	W = fr3:0 || fr7:4
Bits affected:	None
Opcode:	`0011 100f ffff ffff`
Description:	This instruction exchanges the high-order and low-order nibbles (4-bit fields) of the value in the specified data memory location and moves the result to W. The data memory location is left unchanged.
Cycles:	1
Example:	`swap W,0x099` This example swaps the high-order and low-order nibbles of the value in data memory location 0x099 and moves the result into W. For example, if the data memory location holds 0xA5, after executing this instruction, W will hold 0x5A.

TEST fr

Operation: if fr = 0, then Z = 1, else Z = 0

Bits affected: Z

Opcode: `0010 001f ffff ffff`

Description: This instruction moves the contents of the specified data memory location into the same register. There is no net effect except to set or clear the Z bit. If the register holds zero, the bit is set. Otherwise, the bit is cleared. If the **`test`** instruction is performed on the T0TMR register, the Timer 0 prescaler is initialized to zero. If the prescaler is about to expire causing Timer 0 to increment and the **`test`** instruction is executed, Timer 0 will not increment.

Cycles: 1

Example:

```
test 0x099 ;test 0x099
sb STATUS,2 ;skip if Z=1
inc w ;increment W
nop
```

This example tests the contents of data memory location 0x099. The **`test`** instruction sets or clears the Z bit based on the contents of the data memory location. The **`sb`** instruction tests the Z bit and skips to the **`nop`** instruction if the Z bit is set. The **`inc`** instruction is executed only if the data memory location is nonzero.

XOR fr,W

Operation: fr = fr ^ W

Bits affected: Z

Opcode: `0001 101f ffff ffff`

Description: This instruction performs a bitwise exclusive OR of the contents of the specified data memory location and W, and writes the 8-bit result into the same data memory location. W is left unchanged. If the result is zero, the Z bit is set.

Cycles: 1

Example:

```
xor 0x099,w ;move fr XOR W to fr
```

This example performs a bitwise logical XOR of W with a value stored in data memory location 0x099. The result is written back to the data memory location 0x099.

For example, suppose that the data memory location 0x099 is holds the value 0x0F and W holds the value 0x13. The instruction takes the logical XOR of 0x0F and 0x13 and writes the result 0x1C back to the data memory location. The result is nonzero, so the Z bit is cleared.

XOR W,fr

Operation: W = W ^ fr

Bits affected: Z

Opcode: `0001 100f ffff ffff`

Description: This instruction performs a bitwise exclusive OR of the contents of W and the specified data memory location, and writes the 8-bit result into W. The data memory location is left unchanged. If the result is 0x00, the Z bit is set.

Cycles: 1

Example:

```
xor w,0x099 ;move W XOR fr to W
```

This example performs a bitwise logical XOR of the value stored in data memory location 0x099 with W. The result is written back to W.

For example, suppose that the data memory location 0x099 holds the value 0x0F and W holds the value 0x13. The instruction takes the logical XOR of 0x0F and 0x13 and writes the result 0x1C into W. The result is nonzero, so the Z bit is cleared.

XOR W,#lit8

Operation: W = W ^ lit8

Bits affected: Z

Opcode: `0111 1111 kkkk kkkk`

Description: This instruction performs a bitwise exclusive OR of the contents of W and an 8-bit literal value, and writes the 8-bit result into W. If the result is 0x00, the Z bit is set.

Cycles: 1

Example:

```
xor w,0x#0F ;complement W3:0
```

This example performs a bitwise logical XOR of W with the literal value 0x0F. The result is written back to W.

For example, suppose that W holds the value 0x51. The instruction takes the logical XOR of this value with 0x0F and writes the result 0x5E into W. The result is nonzero, so the Z bit is cleared.