Totally Logical

CHAPTER 1

ADDRESS SPACE

INTRODUCTION

Two address spaces are available for the Z8PLUS MCU:

- Register file RAM contains addresses for all the control registers and all the general purpose registers.
- Program memory contains addresses for all memory locations where executable code and/or data are stored.

REGISTER FILE SPACE

The on-chip register file RAM is organized into 16 pages, where each page has 256 addressable memory locations. The first page (page 0) contains both control registers and general purpose registers. All the remaining pages (pages 1 through 15) contain only general purpose registers. Figure 1-1 illustrates the complete register file RAM space. As shown, control registers are located in the upper half of page 0. Any specific implementation of the Z8PLUS core may use only a subset of the complete register file RAM space.

Table 1-1 describes the Core Control Registers and Table 1-2 shows the Page 0 Register File organization.

All registers on the Z8PLUS-family products are fully read/writable. Hardware may write lock certain registers or bits under some conditions. The `TCTLHI` register is one such example.

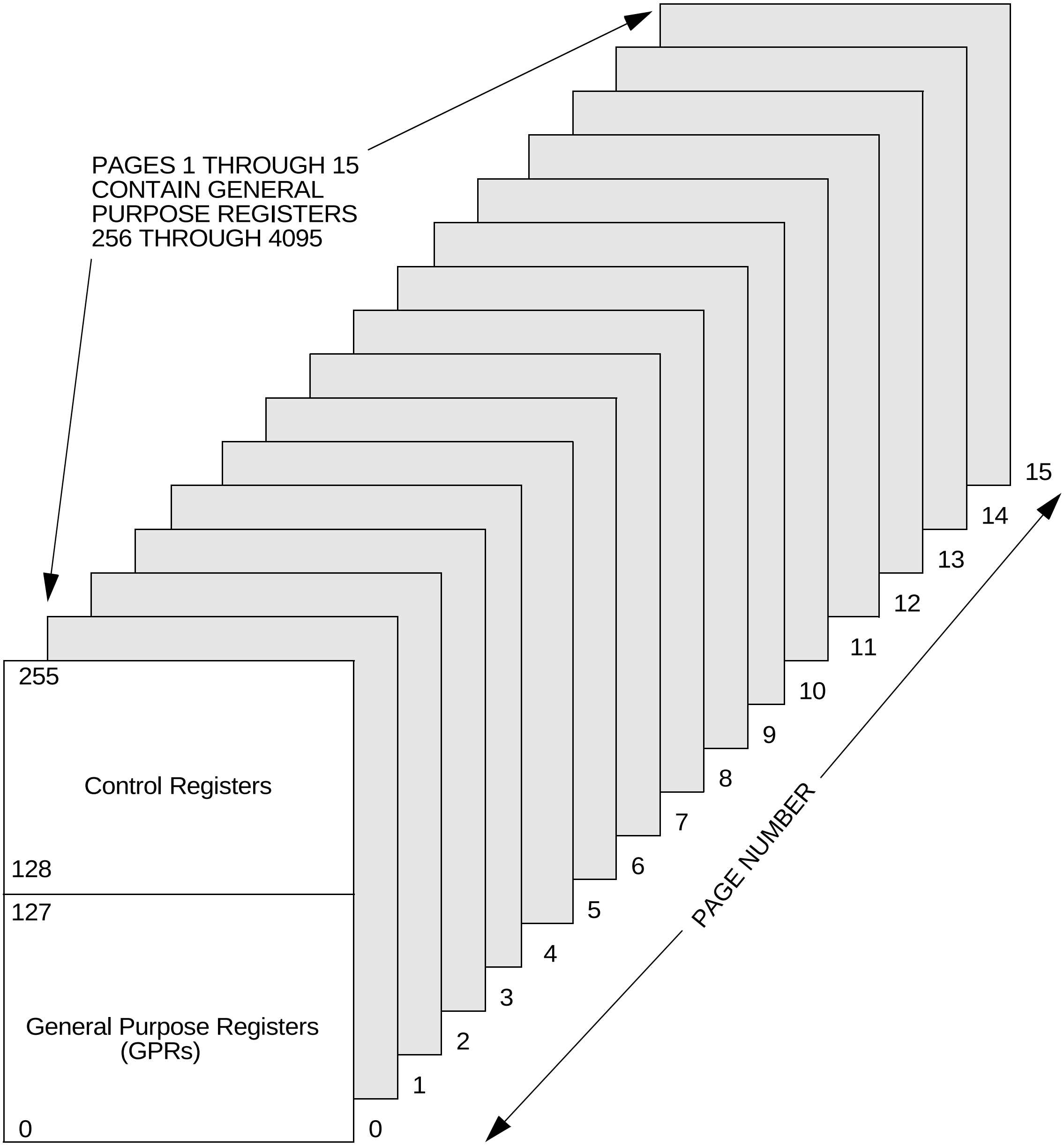

Figure 1-1. Complete Register File RAM Space

Table 1-1. Z8PLUS Core Control Registers

Hex Address	Register Name	Register Description	Comments
0FFH	STKPTR (SPL)	Stack Pointer Low	LSB of Stack Pointer
0FEH	SPH	Stack Pointer High	MSB of Stack Pointer
0FDH	REGPTR(RP)	Register Pointer	
0FCH	FLAGS	Flags	
0FBH	IMASK	Interrupt Mask 1	Ints. 0 - 6
0FAH	IREQ	Interrupt Request 1	Ints. 0 - 6
0F9H	IMASK2	Interrupt Mask 2	Ints. 7 - 14
0F8H	IREQ2	Interrupt Request 2	Ints. 7 - 14
0F7H			Reserved
0F6H			Reserved
0F5H			Reserved
0H4H			Reserved
0F3H			Reserved
0F2H			Reserved
0F1H			Reserved
0F0H			Reserved

The Stack Pointer High register (`0FEH`), the interrupt mask register `2` (`0F9H`), and the interrupt request register 2 (`0F8H`) are optional and are reserved if not implemented.

Table 1-2. Page 0 Register File Organization

Hex Address Range	Register Description
F0 - FF	Core Control Registers
E0 - EF	Virtual Copy of the Current Working Register Set
D0 - DF	Port Logic Control Registers
C0 -CF	Timer Peripherals Control Registers
B0 - BF	Reserved for Future Extensions
A0 - AF	Reserved for Future Extensions
90 - 9F	Reserved for Future Extensions
80 - 8F	Reserved for Future Extensions
70 - 7F	General Purpose Registers
60 - 6F	General Purpose Registers
50 - 5F	General Purpose Registers
40 - 4F	General Purpose Registers
30 - 3F	General Purpose Registers
20 - 2F	General Purpose Registers
10 -1F	General Purpose Registers
00 - 0F	General Purpose Registers

Registers can be accessed as either 8-bit or 16-bit registers using Direct, Indirect, or Indexed Addressing. All general-purpose registers can be referenced or modified by any instruction that accesses an 8-bit register, without the need for special instructions. Registers accessed as 16 bits are treated as even-odd register pairs. In this case, the data's Most Significant Byte (MSB) is stored in the even numbered register, while the Least Significant Byte (LSB) goes into the next higher odd numbered register (Figure 1-2).

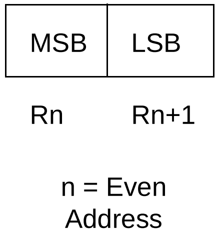

Figure 1-2. 16-Bit Register Addressing

By using a logical instruction and a mask, individual bits within registers can be accessed for bit set, bit clear, bit complement, or bit test operations. For example, the instruction `AND R15, MASK` performs a bit clear operation. Figure 1-3 shows this example.

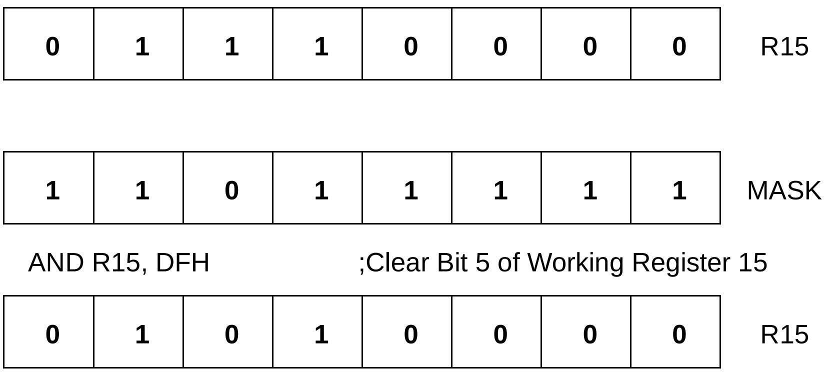

Figure 1-3. Accessing Individual Bits (Example)

When instructions are executed, registers are only read, not written, when defined as sources; and read and/or written when defined as destinations. All General-Purpose Registers function as accumulators, address pointers, index registers, stack areas, or scratch pad memory.

General-Purpose Registers

General-Purpose Registers (GPR) are undefined after the device is powered up. The registers keep their last value after any reset, as long as the reset occurs in the V_{CC} voltage-specified operating range. It does not keep its last state from a V_{LV} reset if V_{CC} drops below `1.8V`.

Working Register Groups

Instructions can access 8-bit registers and register pairs (16-bit words) using either 4-, 8-, or 12-bit address fields. Eight-bit address fields refer to the actual address of the register within the current page. For example, Register 58H is accessed by calling upon its 8-bit address, 01011000 (58H). The lower nibble of the Register Pointer specifies the current RAM page.

With 4-bit addressing, the register file is logically divided into 16 Working Register Groups of 16 registers each, as shown in Table 1-3. These 16 registers are known as Working Registers. A Register Pointer (one of the control registers, FDH) contains the base address of the active Working Register Group. The High nibble of the Register Pointer determines the current Working Register Group.

When accessing one of the Working Registers, the 4-bit address of the Working Register is combined with the upper four bits (High nibble) of the Register Pointer, thus forming the 8-bit actual address. Figure 1-4 illustrates this operation. Since working registers are typically specified by short format instructions, there are fewer bytes of code needed. In addition, when processing interrupts or changing tasks, the Register Pointer (see Figure 1-5) speeds context switching. A special Set Register Pointer (SRP) instruction sets the contents of the Register Pointer.

Data transfer across RAM page boundaries can be accomplished via 12-bit addressing. Using certain instruction modes, data can be moved from the current page and working group into any register on the chip by specifying the absolute 12-bit address, including page. Not all family members support 12-bit addressing. See the applicable product specification for specific information.

Table 1-3. Working Register Groups

Register Pointer (FDH) High Nibble (Binary)	Working Register Group (HEX)	Actual Registers (HEX)
1111	F	F0 - FF
1110	E	E0 - EF
1101	D	D0 - DF
1100	C	C0 - CF
1011	B	B0 - BF
1010	A	A0 - AF
1001	9	90 - 9F
1000	8	80 - 8F
0111	7	70 - 7F

Table 1-3. Working Register Groups (Continued)

Register Pointer (FDH) High Nibble (Binary)	Working Register Group (HEX)	Actual Registers (HEX)
0110	6	60 - 6F
0101	5	50 - 5F
0100	4	40 - 4F
0011	3	30 - 3F
0010	2	20 - 2F
0001	1	10 - 1F
0000	0	00 - 0F

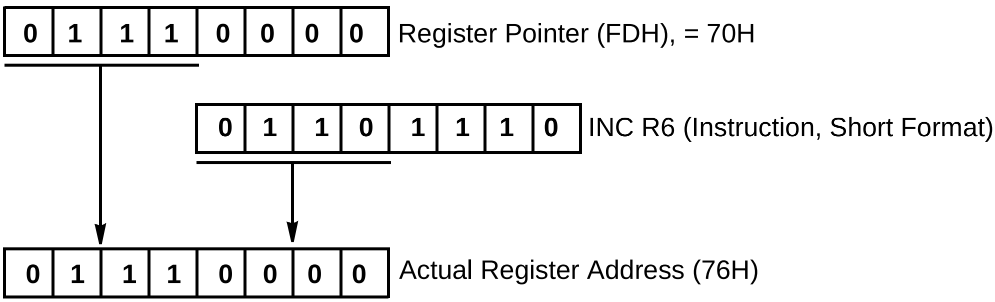

Figure 1-4. Working Register Addressing (Example)

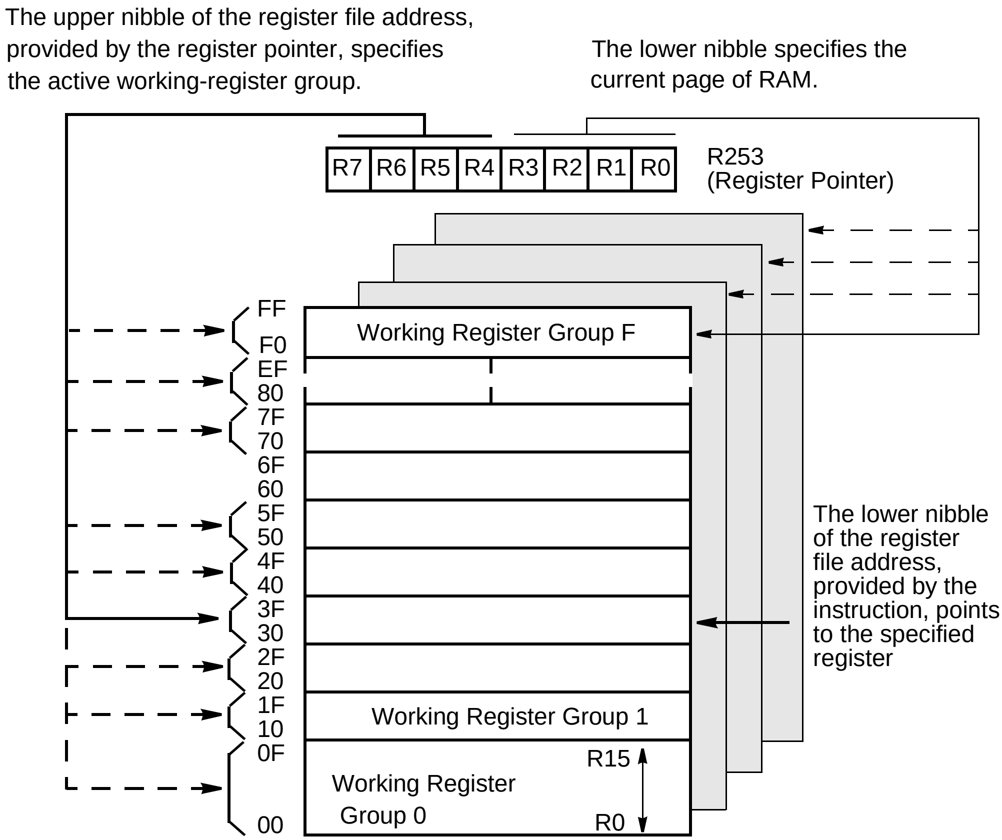

Figure 1-5. Register Pointer

Precautions

Registers in the Standard Register File must be correctly used or certain conditions produce inconsistent results.

- The watch-dog timer can only be disabled via software if the first instruction out of RESET performs this function. During the execution of the first instruction after the Z8PLUS leaves RESET, the upper five bits of the TCTLHI register can be written. After the first instruction, hardware does not allow the upper five bits of this register to be written.
- Some control registers, including the port inputs and timer count registers, may be updated by hardware. Writing these registers from software always overrides the hardware update from the same cycle, but with unpredictable results. For example, writing into the count value register of a running timer can cause

unexpected results if the hardware was in the process of decrementing the timer for the terminal count and generating an interrupt.

- The register space from `0E0H-0EFH` is special. The MCU uses these addresses to flag accesses via 4-bit addressing mode to the current working register group. There are no physical registers at that location. Care must be taken that the Register Pointer never points at `Group E` on the first page (be loaded with `E0H`). This is an undefined case. Also, indirect addressing does *not* redirect a second time and find the working registers. This is also an undefined case. As an example, in the code below, `R0` does *not* find the data in register `08`. It returns garbage. `R2` correctly contains a copy of register `08`.

```
SRP             #%00

LD              R1, #%E8

LD              R0, @R1

LD              R2,%E8
```

CONTROL AND PERIPHERAL REGISTERS

Control Registers

The standard control registers govern the operation of the CPU. Any instruction which references the register file can access these control registers. Available control registers are:

- Stack Pointer Low (SPL or STKPTR)
- Stack Pointer High (SPH)
- Register Pointer (RP or REGPTR)
- Flags (FLAGS)
- Interrupt Mask 1 (IMASK)
- Interrupt Request 1 (IREQ)
- Interrupt Mask 2 (IMASK2)
- Interrupt Request 2 (IREQ2)

A 16-bit Program Counter (PC) to determine the sequence of current program instructions. The PC is not an addressable register.

Peripheral Registers

Peripheral registers are used to transfer data, configure the operating mode, and control the operation of the on-chip peripherals. Any instruction that references the register file can access the peripheral registers. Possible peripheral registers can include:

- Timer Count Value Register for `Timer` *n*
- Auto-Initialization Value Register(s) for `Timer` *n*
- Timer Control Registers (High and Low Byte)
- Watch-Dog Timer Registers (High and Low Byte)

In addition, the port registers are considered to be peripheral registers. Ports generally have at least the following four dedicated registers which are readable and writable by software:

- Port Input Value Register
- Port Output Value Register
- Port Control Register
- Port Special Function Register

PROGRAM MEMORY

The program memory map is shown in Figure 1-6. The first two bytes of program memory are reserved for the PC rollover vector. When the PC wraps around to `0000H`, bytes `0000H` and `0001H` are executed as instructions, enabling a user defined behavior for this occurrence. For example, a `JR` instruction in `0000H` and a corresponding displacement in `0001H` could be defined for the PC rollover vector. The next 30 bytes of Program Memory are reserved for the interrupt vectors. These locations contain 16-bit vectors that correspond to the available interrupts. Address `0020H` through the end of the populated memory (`0FFFFh`, 64 KB maximum) consists of on-chip mask-programmable ROM or EPROM or Flash. The first byte of program memory executed following a `RESET` is located at `0020H`. See the product data sheet for the exact program, data, register memory size, and address range available.

The internal program memory may be one-time programmable (OTP) or mask programmable dependent on the specific device. A ROM protect feature prevents dumping of the ROM contents. The ROM Protect option is mask-programmable and is selected by the customer when the ROM code is submitted. For programmable memory devices, the ROM Protect option is an OTP programming option.

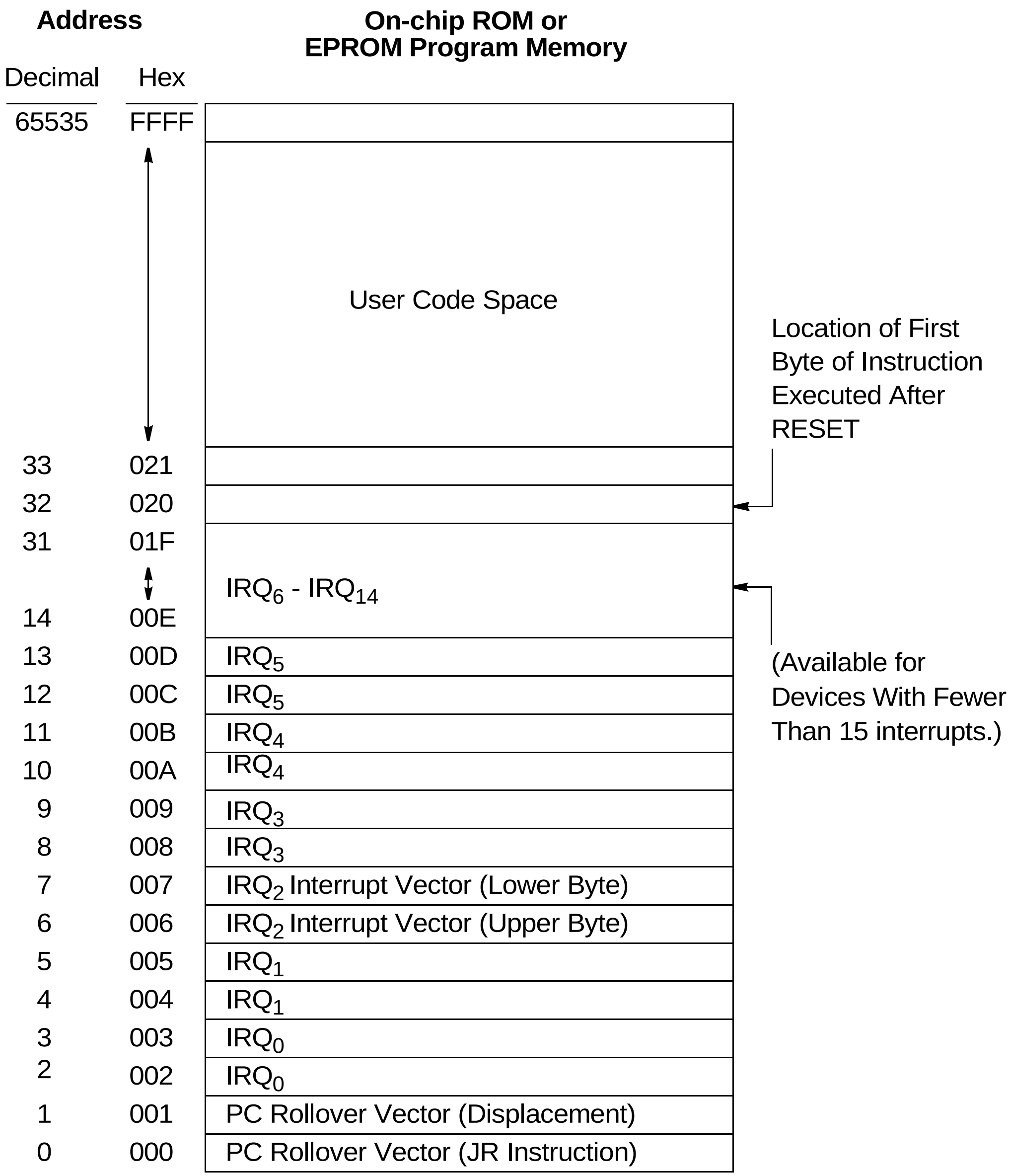

Figure 1-6. Program Memory Map

STACK

The stack always resides in the general purpose registers of the on-chip register file RAM. The stack pointer register (SP) contains an address into the standard register file that is the address of the operand that is currently on the top of the stack. The register 0FFH is the 8-bit stack pointer (SP), that is used for all stack operations (see Figure 1-7).

Some devices prepend the lower nibble of register 0FEH to form a 12-bit stack pointer. Otherwise, register 0FEH is reserved.

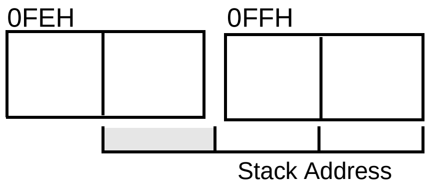

Figure 1-7. Stack Pointer

The stack address is decremented prior to a PUSH operation and incremented after a POP operation. The stack address always points to the data stored on the top of the stack. The stack is a return stack for CALL instructions and interrupts, as well as a data stack.

During a CALL instruction, the contents of the Program Counter are saved on the stack. The PC is restored during a RET instruction. Interrupts cause the contents of the PC and FLAGS registers to be saved on the stack. The IRET instruction restores them (see Figure 1-8).

An overflow or underflow can occur when the stack address is incremented or decremented during normal stack operations. The programmer must prevent this occurrence or unpredictable operation may result. The stack must not encroach into the control registers.

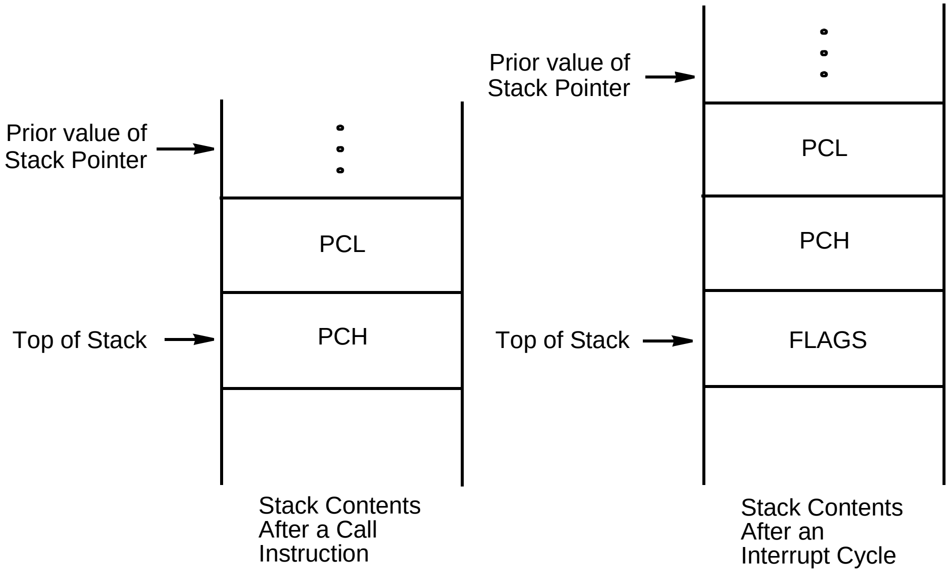

Figure 1-8. Stack Operations