What is the difference between an **accumulator instruction set architecture** and a general-purpose register instruction set architecture?

Consider what a CPU compute instruction needs to do. It needs to provide at least the following information:

- What computation to perform (add, subtract, shift, compare)
- What values to perform the computation on (the inputs to the computation)
- Where to put the result (the output of the computation)

For example, to execute `X = Y + Z`, you need to tell the processor to fetch the values held in Y and Z, add them, and write/store the result to X.

In an accumulator architecture, most compute instructions operate on a special register called the accumulator. Most operations, therefore, have the accumulator as an *implicit* argument to the instruction. The accumulator either provides an input to the instruction, receives the output from the instruction, or both. To perform `X = Y + Z` on an accumulator-based machine, the instruction sequence would look roughly like this:

- Load Y into the accumulator
- Add Z to the accumulator
- Store accumulator to X

If I had a more complex expression, such as "I = J + K + L + M + N + O", the sequence might look like this:

- Load J into the accumulator
- Add K to the accumulator
- Add L to the accumulator
- Add M to the accumulator
- Add N to the accumulator
- Add O to the accumulator
- Store accumulator to I

Because most operations involve the accumulator, you don't need to dedicate any opcode bits to specify it. As I mentioned before, the accumulator is *implied.* Also, in the machine itself, the accumulator could be built into the arithmetic unit itself, simplifying the hardware.

In a **general-purpose register** architecture, compute instructions take multiple arguments to specify which registers to read values from, making them more flexible. But, you need more opcode bits to specify which registers to operate on, and you need to provide paths for all of those registers to the arithmetic unit.

For the example expressions above, the code ends up not looking much different:

- Load Y into register R0
- Load Z into register R1
- Add R0 to R1, putting the result in R2
- Store R2 to X

General purpose register machines start showing an advantage when you can keep values in registers across many operations. For example, suppose I wanted to run this slightly more complicated program:

```
1.   X = A + B
2.   Y = A - B
```

In an accumulator machine, I have to reload A and B for both operations. In a general purpose machine, I load A and B only once.

Within general purpose register machines, there are multiple varieties as well: (This list is not exhaustive; also, some architectures blend these concepts.)

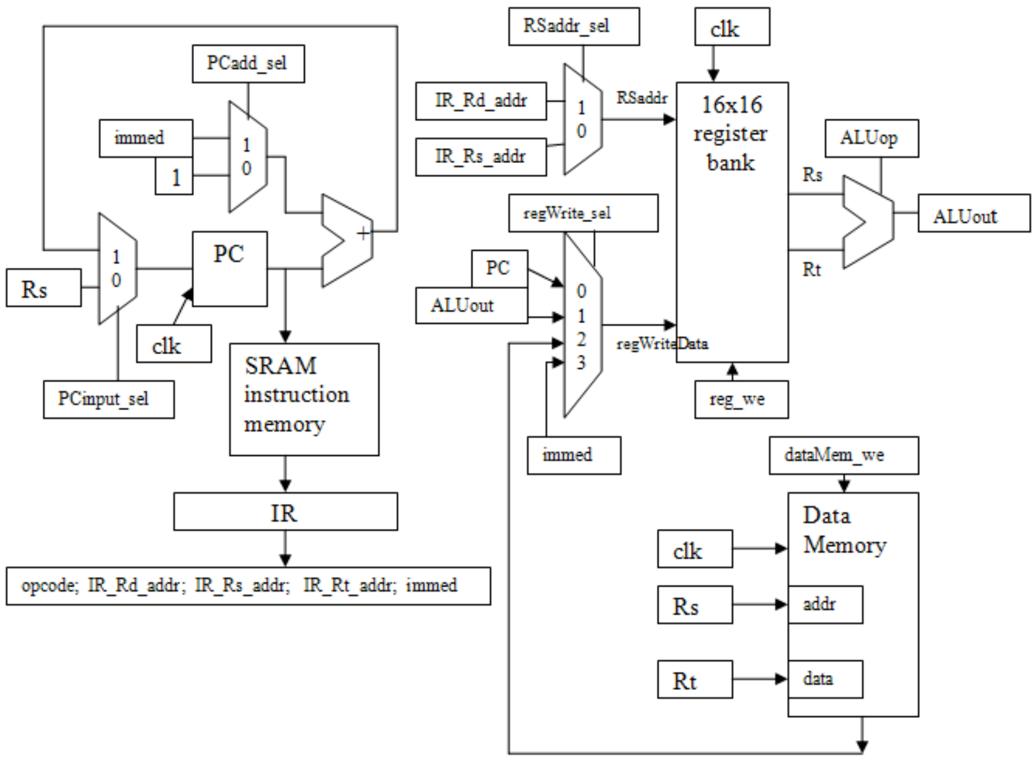
- **Register-Memory**: One operand comes from a register, and one operand comes from memory.
- **Register-Register, 2-address**: Both operands come from registers, but the result must overwrites one of the inputs.
- **Register-Register, 3-address**: Both operands come from registers, and the result can go to its own register.

The x86 processor is a Register-Memory machine that also offers 2-address Register-Register instructions. Most RISC machines are 3-address Register-Register machines, with separate load/store instructions. Both are general-purpose register machines.

Compare those to the 6502, an accumulator machine. Most arithmetic (addition, subtraction, rotates and shifts) operates on the A register. The two other registers, X and Y, support only increment, decrement and compare; they're mainly used for indexing memory and loop counters.

A third architecture type is the *stack architecture*. A stack architecture is similar to an accumulator architecture in that all the of computation is focused on a single point. The difference is that a stack architecture always reads its arguments from a stack, and always puts its results on the stack. Both the inputs *and* the outputs of a compute instruction are implicit. Other instructions then need to manage pushing values onto and popping values off of the stack.

You don't find many stack architectures in actual chips, but they're popular in interpreters (Java bytecode, for example, or FORTH), and in HP's RPN-based calculators.



16-instruction CPU

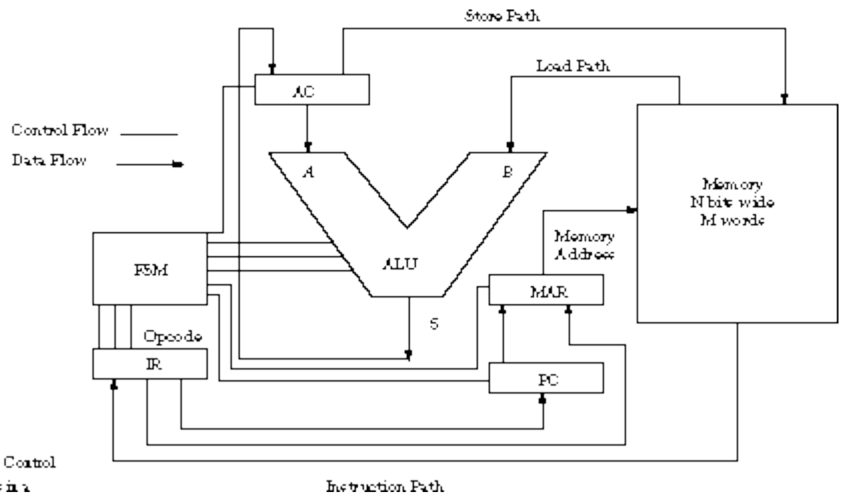


Figure 11.4 Control and data flows in a

single accumulator architecture

ISA Classification Types

CPU instruction set architectures can be classified according to where the operands come from in ALU operations. There are four basic classifications:

- **Stack**
  - OP1 « The Stack
  - OP2 « The Stack
- **Accumulator-Memory**
  - OP1 « Accumulator
  - OP2 « Memory
- **Register-Memory**
  - OP1 « CPU Register
  - OP2 « Memory
- **Register-Register (Load/Store)**
  - OP1 « CPU Register
  - OP2 « CPU Register

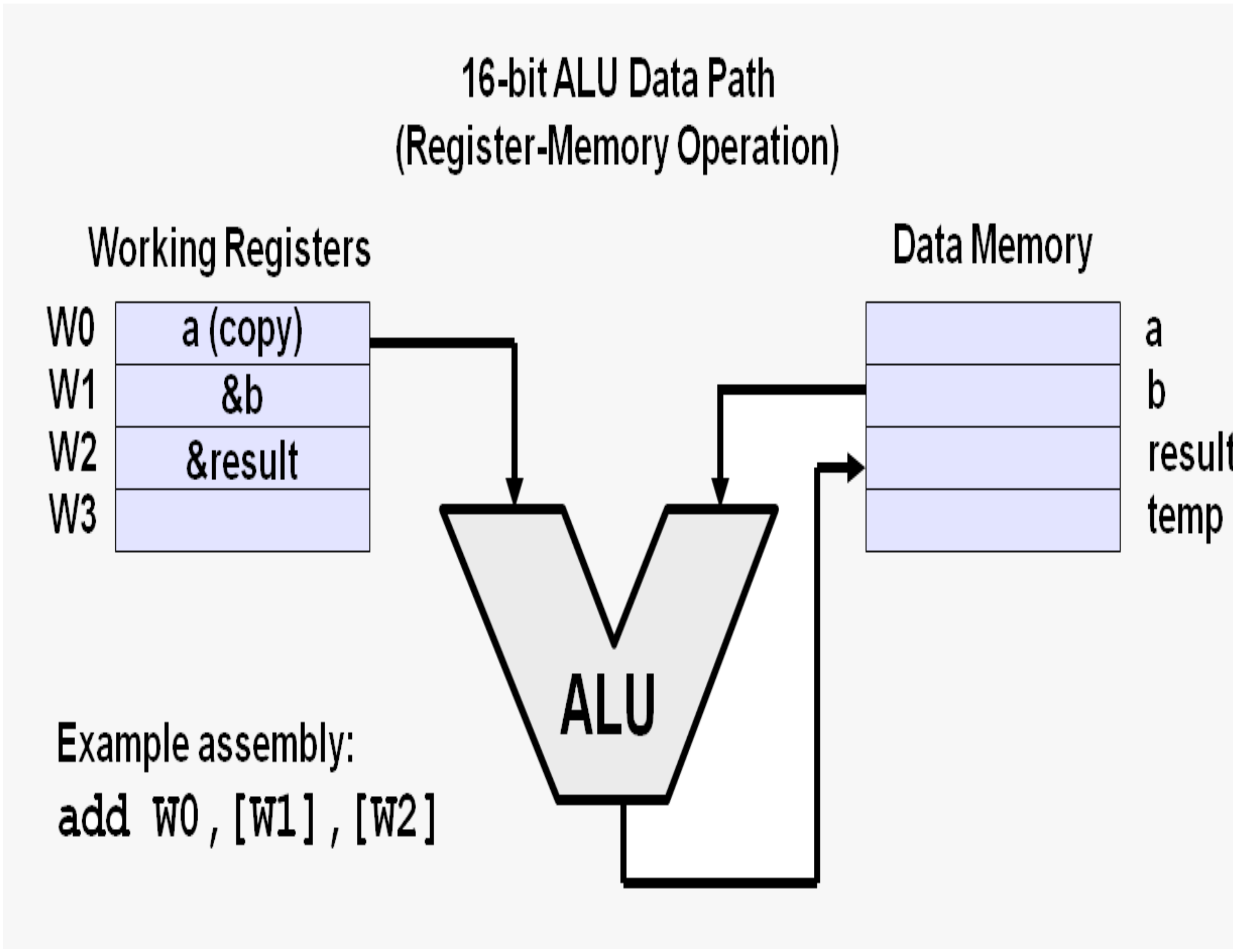
Most MCU CPUs are Register-Memory machines, since the speed of the memory technology often matches the CPU speed. These architectures support efficient bit (read/modify/write) operations.

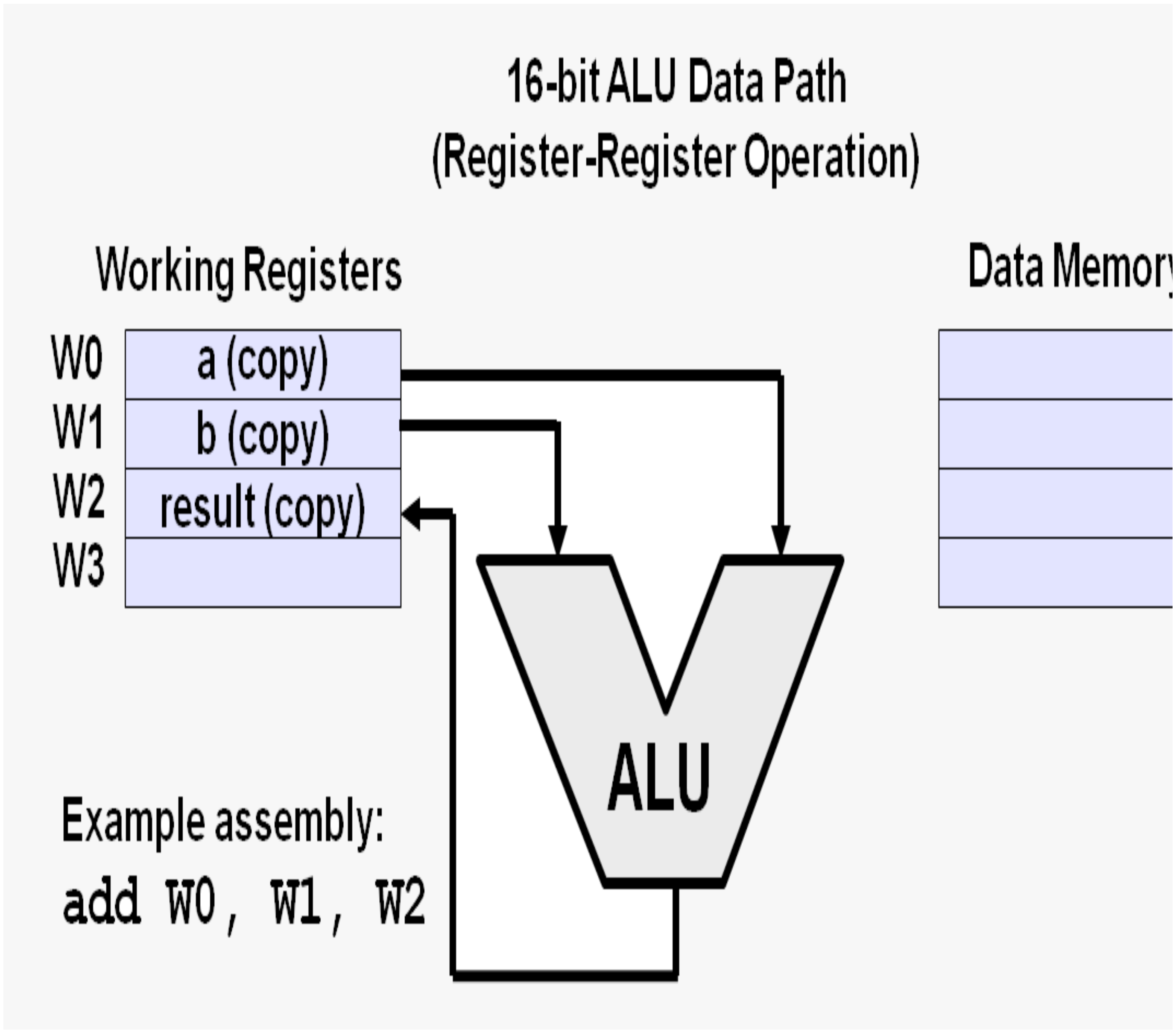
Modern high-performance MCU CPUs are increasingly moving to Load/Store architectures to de-couple the CPU speed from memory speed, and using cache memories to enable sustained high CPU performance.

The ISA classification defines the required addressing modes for the instruction set:

16-Bit ISA Classification

PIC24/dsPIC implement a **hybrid** ISA architecture, supporting both **Register-Memory** and **Register-Register** operations as shown below:



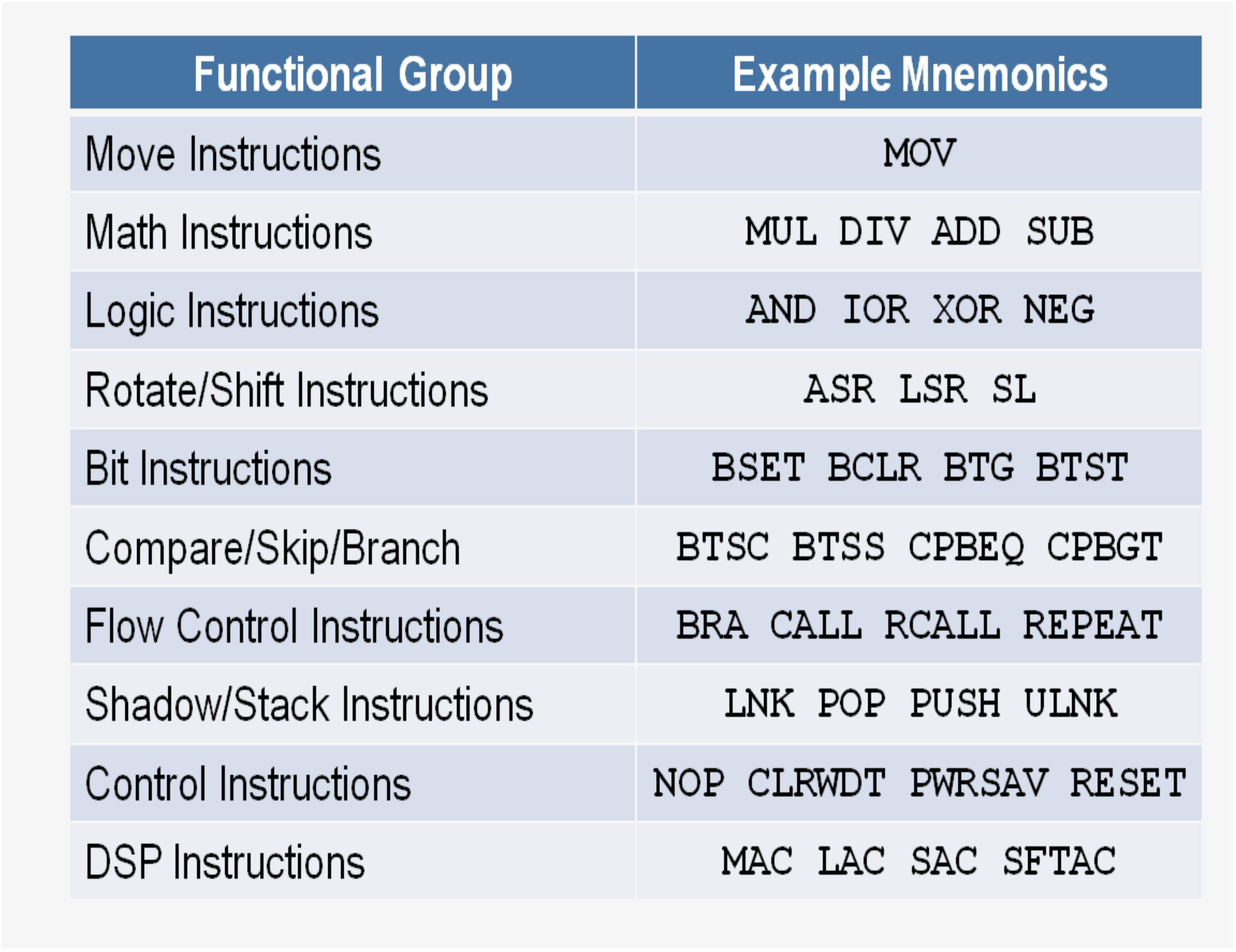


Additionally, the 40-bit DSP ALU implements an **Accumulator-Memory** architecture as described here.

Instruction Categories

The 16-bit MCU and DSC instruction set provides a rich suite of instructions that support traditional microcontroller applications, and a class of instructions that support math intensive applications. The majority of the instructions are encoded in a single 24-bit word, and execute in a single instruction cycle.

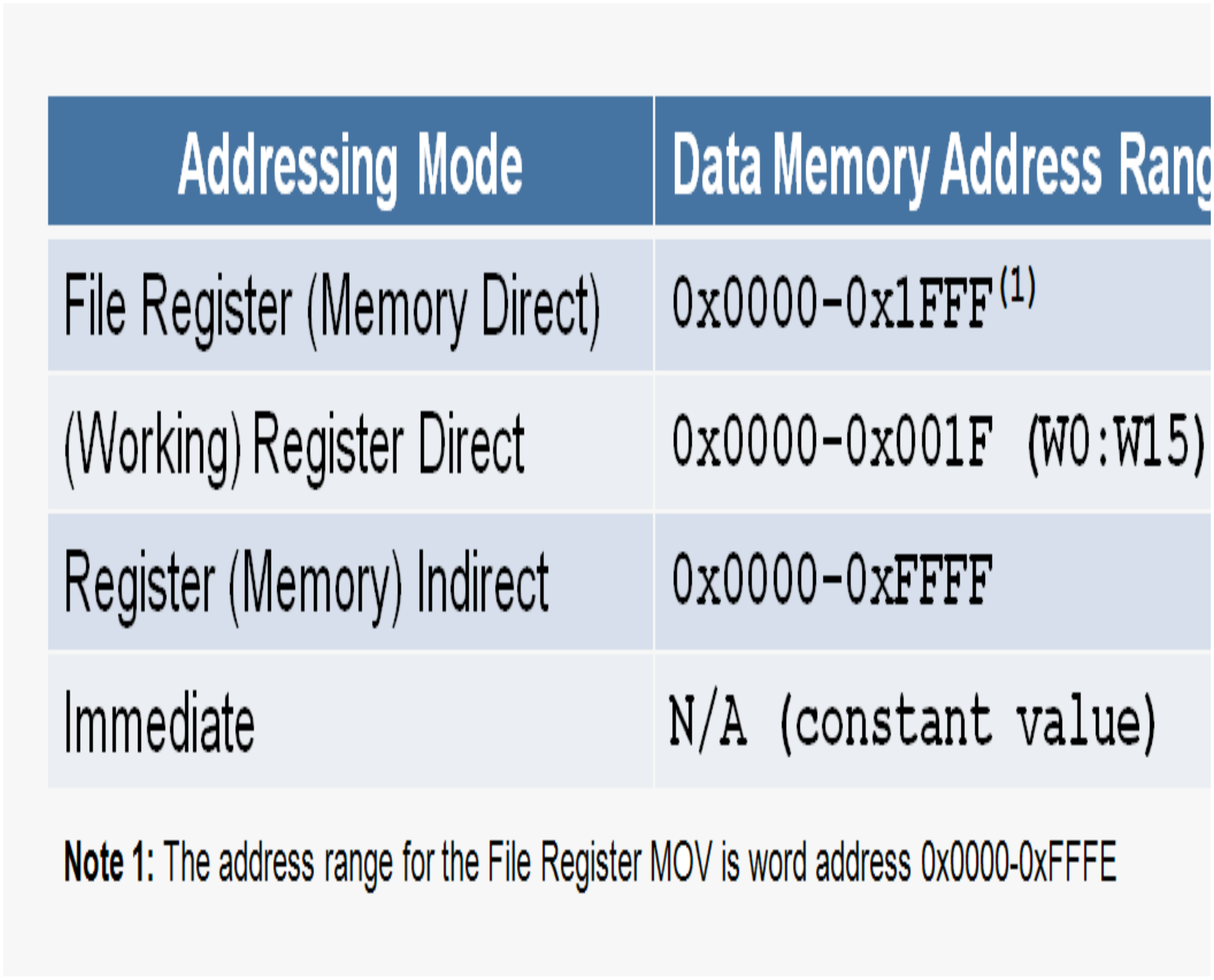
Depending on the device family, the 16-bit MCU and DSC instruction set contains up to 84 instructions, which can be grouped into the functional categories shown in the following table:



Functional Group	Example Mnemonics
Move Instructions	MOV
Math Instructions	MUL DIV ADD SUB
Logic Instructions	AND IOR XOR NEG
Rotate/Shift Instructions	ASR LSR SL
Bit Instructions	BSET BCLR BTG BTST
Compare/Skip/Branch	BTSC BTSS CPBEQ CPBGT
Flow Control Instructions	BRA CALL RCALL REPEAT
Shadow/Stack Instructions	LNK POP PUSH ULNK
Control Instructions	NOP CLRWDT PWRSAV RESET
DSP Instructions	MAC LAC SAC SFTAC

Addressing Modes

The 16-bit MCU and DSC devices support three native Addressing modes for accessing data memory, along with several forms of immediate addressing. Data accesses may be performed using file register addressing, register direct or indirect addressing, and immediate addressing, allowing a fixed value to be used by the instruction. The data memory address range accessed by each addressing mode is summarized below:



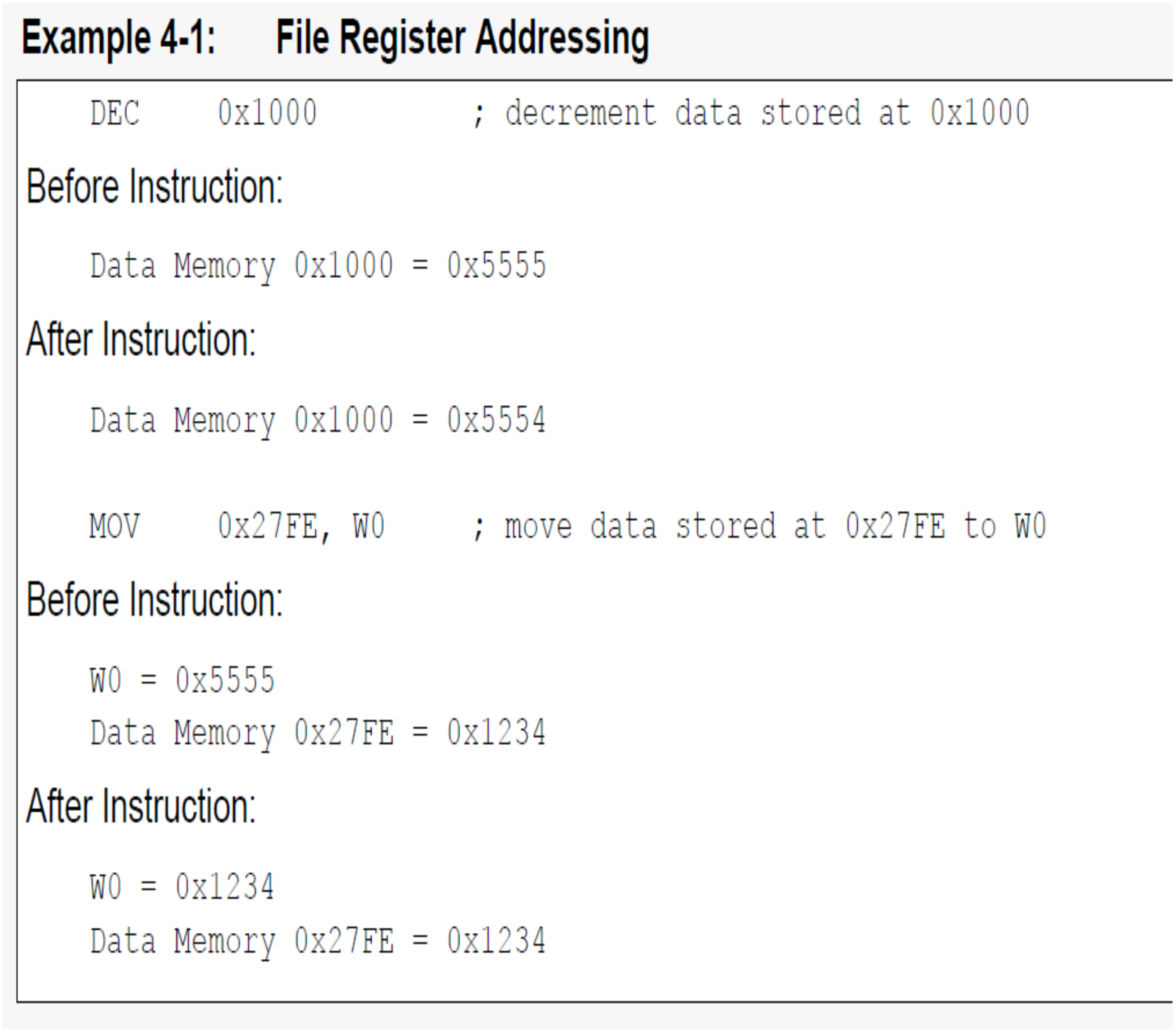
Addressing Mode	Data Memory Address Rang
File Register (Memory Direct)	0x0000-0x1FFF[1]
(Working) Register Direct	0x0000-0x001F (W0:W15)
Register (Memory) Indirect	0x0000-0xFFFF
Immediate	N/A (constant value)

Note 1: The address range for the File Register MOV is word address 0x0000-0xFFFE

File Register (Memory Direct) Addressing

File Register (or Memory Direct) addressing provides the ability to operate on data stored in the lower 8 kBytes of data memory ("NEAR" RAM). Instructions use a predetermined address as an operand.

The majority of instructions that use file register addressing provide **byte/word** access to the lower 8 kBytes data memory, with the exception of the MOV instruction which provides **word** access to all 64 kBytes. This allows the loading of the data from any location in data memory to any working register (W0:W15), and storing the contents of any working register to any location in data memory. Examples of File Register addressing are shown below:



Example 4-1: File Register Addressing

```
    DEC     0x1000          ; decrement data stored at 0x1000
```

Before Instruction:

```
    Data Memory 0x1000 = 0x5555
```

After Instruction:

```
    Data Memory 0x1000 = 0x5554


    MOV     0x27FE, W0      ; move data stored at 0x27FE to W0
```

Before Instruction:

```
    W0 = 0x5555
    Data Memory 0x27FE = 0x1234
```

After Instruction:

```
    W0 = 0x1234
    Data Memory 0x27FE = 0x1234
```

Register Direct Addressing

Register direct addressing is used to access the contents of the 16 working registers (W0:W15). Any working register may be used for any instruction that supports this addressing mode.

Instructions using this addressing mode use the contents of the specified working register as operands for the operation to be performed. This addressing mode supports both **Byte** and **Word** access. Sample instructions which utilize register direct addressing are shown below:

Example 4-3: Register Direct Addressing

```
    EXCH    W2, W3          ; Exchange W2 and W3
```
Before Instruction:
```
    W2 = 0x3499
    W3 = 0x003D
```
After Instruction:
```
    W2 = 0x003D
    W3 = 0x3499
```

```
    IOR     #0x44, W0       ; Inclusive-OR 0x44 and W0
```
Before Instruction:
```
    W0 = 0x9C2E
```
After Instruction:
```
    W0 = 0x9C6E
```

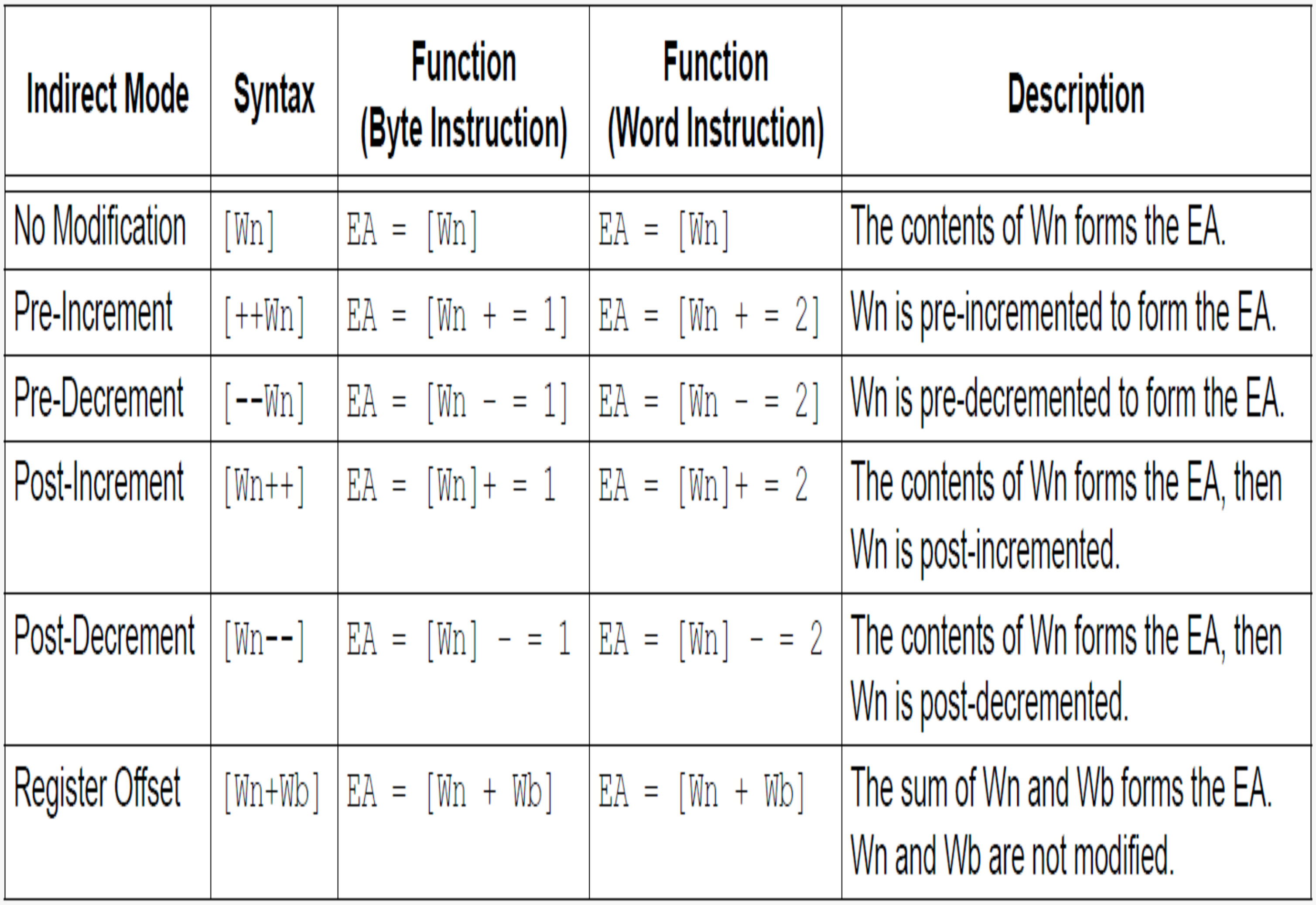
```
    SL      W6, W7, W8      ; Shift left W6 by W7, and store to W8
```
Before Instruction:
```
    W6 = 0x000C
    W7 = 0x0008
    W8 = 0x1234
```
After Instruction:
```
    W6 = 0x000C
    W7 = 0x0008
    W8 = 0x0C00
```

Register Indirect Addressing

Register Indirect addressing is used to indirectly access any location in data memory by treating the contents of a working register as an Effective Address (EA) to data memory. Essentially, the contents of the working register become a pointer to the location in data memory which is to be accessed by the instruction.

Additionally, the contents of the working register may be modified pre or post operation, providing an efficient mechanism for processing data stored sequentially in memory. The modes of indirect addressing supported are shown below:

Table 4-2: Indirect Addressing Modes



Indirect Mode	Syntax	Function (Byte Instruction)	Function (Word Instruction)	Description
No Modification	`[Wn]`	`EA = [Wn]`	`EA = [Wn]`	The contents of Wn forms the EA.
Pre-Increment	`[++Wn]`	`EA = [Wn + = 1]`	`EA = [Wn + = 2]`	Wn is pre-incremented to form the EA.
Pre-Decrement	`[--Wn]`	`EA = [Wn - = 1]`	`EA = [Wn - = 2]`	Wn is pre-decremented to form the EA.
Post-Increment	`[Wn++]`	`EA = [Wn]+ = 1`	`EA = [Wn]+ = 2`	The contents of Wn forms the EA, then Wn is post-incremented.
Post-Decrement	`[Wn--]`	`EA = [Wn] - = 1`	`EA = [Wn] - = 2`	The contents of Wn forms the EA, then Wn is post-decremented.
Register Offset	`[Wn+Wb]`	`EA = [Wn + Wb]`	`EA = [Wn + Wb]`	The sum of Wn and Wb forms the EA. Wn and Wb are not modified.

The following examples illustrate indirect addressing with pre/post increment/decrement EA modification:

xample 4-4: Indirect Addressing with Effective Address Update

```
MOV.B   [W0++], [W13--]          ; byte move [W0] to [W13]
                                 ; post-inc W0, post-dec W13
```

efore Instruction:

```
W0 = 0x2300
W13 = 0x2708
Data Memory 0x2300 = 0x7783
Data Memory 0x2708 = 0x904E
```

fter Instruction:

```
W0 = 0x2301
W13 = 0x2707
Data Memory 0x2300 = 0x7783
Data Memory 0x2708 = 0x9083

ADD     W1, [--W5], [++W8]       ; pre-dec W5, pre-inc W8
                                 ; add W1 to [W5], store in [W8]
```

efore Instruction:

```
W1 = 0x0800
W5 = 0x2200
W8 = 0x2400
Data Memory 0x21FE = 0x7783
Data Memory 0x2402 = 0xAACC
```

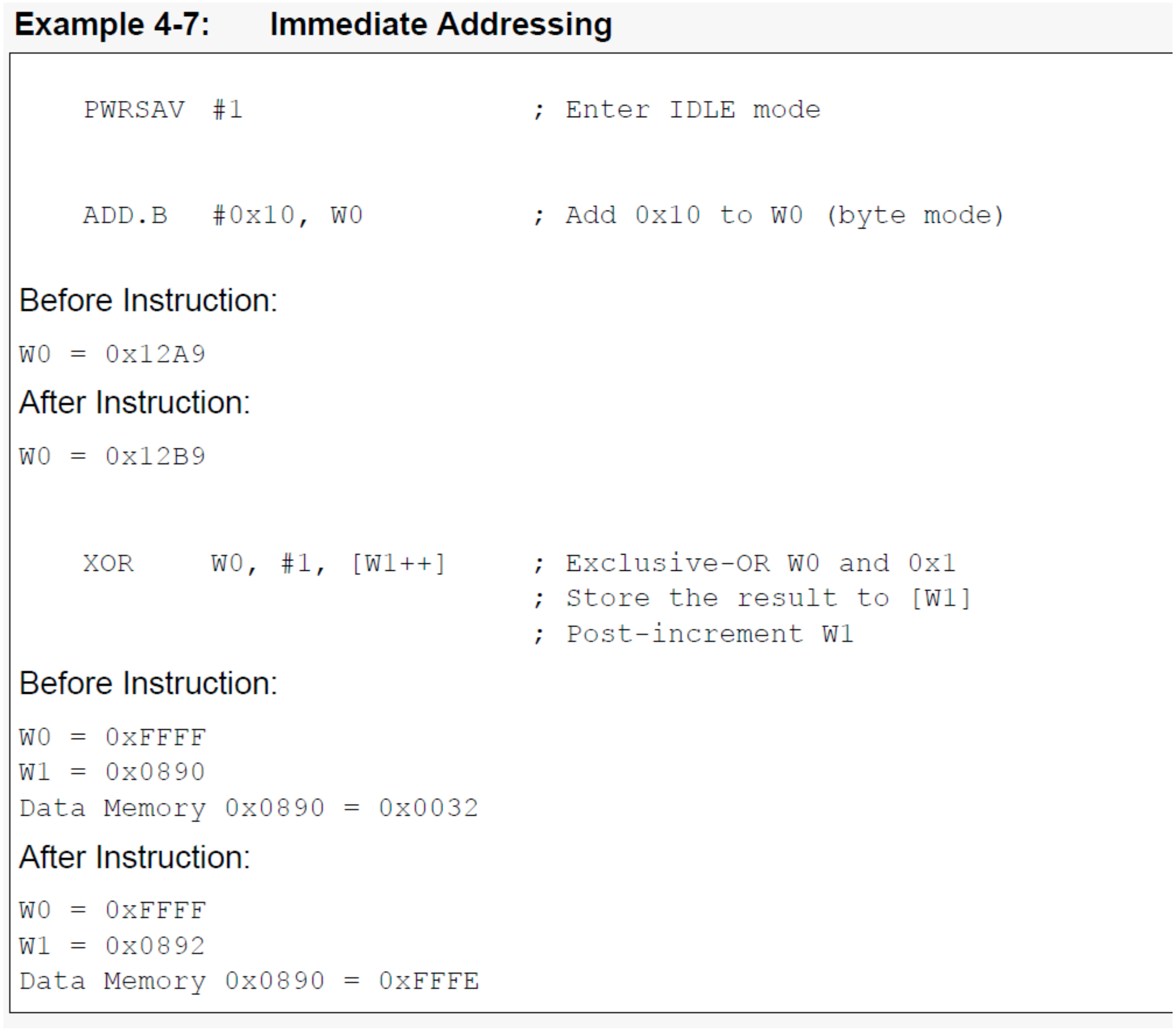
fter Instruction:

```
W1 = 0x0800
W5 = 0x21FE
W8 = 0x2402
Data Memory 0x21FE = 0x7783
Data Memory 0x2402 = 0x7F83
```

Immediate Addressing

In immediate addressing, the instruction encoding contains a predefined constant operand which is used by the instruction. The size of the immediate operand varies with the instruction type. Constants of size 1-bit, 4-bit, 5-bit, 6-bit, 8-bit, 10-bit, 14-bit and 16-bit are allowed, depending on the instruction. Constants may be signed or unsigned.

The following illustrates some examples using Immediate Addressing:



Example 4-7: Immediate Addressing

```
    PWRSAV  #1                  ; Enter IDLE mode


    ADD.B   #0x10, W0           ; Add 0x10 to W0 (byte mode)
```

Before Instruction:

```
W0 = 0x12A9
```

After Instruction:

```
W0 = 0x12B9
```

```
    XOR     W0, #1, [W1++]      ; Exclusive-OR W0 and 0x1
                                ; Store the result to [W1]
                                ; Post-increment W1
```

Before Instruction:

```
W0 = 0xFFFF
W1 = 0x0890
Data Memory 0x0890 = 0x0032
```

After Instruction:

```
W0 = 0xFFFF
W1 = 0x0892
Data Memory 0x0890 = 0xFFFE
```