Addressing Modes for TI TMS320C54x*

Douglas L. Jones
Swaroop Appadwedula
Matthew Berry
Mark Haun
Dima Moussa
Daniel Sachs

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 1.0†

Abstract

The TI TMS320C54x microprocessor provides a number of ways to specify the location of data to be used in calculations. Immediate addressing, direct addressing, and indirect addressing are the three main types. Knowing the basic addressing modes of a microprocessor is important because they map directly into assembly language syntax and because the need to use a particular addressing mode often dictates which instruction one picks for a given task.

Microprocessors provide a number of ways to specify the location of data to be used in calculations. For example, one of the data values to be used in an `add` instruction may be encoded as part of that instruction's **opcode**, the raw machine language produced by the assembler as it parses your assembly language program. This is known as **immediate addressing**. Alternatively, perhaps the opcode will instead contain a memory address which holds the data (**direct addressing**). More commonly, the instruction will specify that an auxiliary register holds the memory address which in turn holds the data (**indirect addressing**). The processor knows which addressing mode is being used by examining special bit fields in the instruction opcode.

Knowing the basic addressing modes of your microprocessor is important because they map directly into assembly language syntax. Many annoying and sometimes hard-to-find bugs are caused by inadvertently using the wrong addressing mode in an instruction. Also, in any assembly language, the need to use a particular addressing mode often dictates which instruction one picks for a given task.

Chapter five, **Data Addressing**, in the *CPU and Peripherals*[**?**] reference contains extended descriptions of most of the addressing modes described below.

1 Accumulators: src, dst

Whenever the abbreviations `src` or `dst` are used in the assembly language syntax description for an instruction, it means that only the accumulators `A` and `B` may be used for that particular operand. These are seen

*Version 2.7: Feb 25, 2004 11:03 am +0000
†http://creativecommons.org/licenses/by/1.0

everywhere, but two classic examples are `ld`, which always loads data into an accumulator from somewhere else, and `sth/stl`, which always store data from an accumulator to somewhere else.

Examples:

```
ld     *AR5,A     ; sets A = (contents of memory location pointed to by AR5)
sth    B,*AR7+    ; sets (contents of memory location pointed to be AR7) = B,
;    and then increments AR7 by one
```

2 Memory-mapped Registers: MMR, MMRx, MMRy

Many of the TMS320C54x registers are memory-mapped, meaning that they occupy real addresses at the low end of data memory space. The most commonly used of these are the auxiliary registers `AR0` through `AR7`. Whenever the abbreviation `MMR` is used in the assembly language syntax description for an instruction, it means that any memory-mapped register may be used for that particular operand. Only eight instructions use memory-mapped register addressing: `ldm`, `mvdm`, `mvmd`, `mvmm`, `popm`, `pshm`, `stlm`, and `stm`. With `mvmm`, since the instruction accepts two memory-mapped register operands, `MMRx` and `MMRy`, only `AR0-AR7` and `SP` may be used.

Do not use an asterisk in front of `ARx` variables here, since this is not indirect addressing.

Examples:

```
mvmm    AR3,AR5   ; sets AR5 = AR3
stm     #5,AR2    ; sets AR2 = 5
ldm     AR0,A     ; sets A = AR0
```

3 Immediate Addressing: #k3, #k5, K, #k9, #lk

Immediate addressing means that the numerical value of the data is itself provided within the assembly instruction. Various TMS320C54x instructions allow immediate data of 3, 5, 8, 9, or 16 bits in length, which are signified in the assembly language syntax descriptions with one of the above symbols. The 16-bit form is the most common and is signified by `#lk`. 16-bit immediate values always require an extra instruction word and therefore take an extra machine cycle to execute.

An immediate data operand is almost always specified in assembler syntax by prepending a pound sign (#) to the data. Depending on the context, the assembler may assume that you meant immediate addressing anyway.

Examples:

```
ld       #0,A      ; sets A = 0
cmpm     AR1,#1    ; sets flag TC = 1 if AR1 == 1; else TC = 0
```

Labels make this more complicated. Recall that a label in your assembly code is nothing more than shorthand for the memory address where the labeled code or data is stored. So does an instruction like

```
stm     coef,AR2   ; sets AR2 = memory address of label coef
```

mean to store the contents of memory location `coef` in `AR2`, or does it mean to store the memory address `coef` itself in `AR2`? The second interpretation is correct. Because the `stm` instruction has only one form, expecting a `#lk` immediate operand, the assembler does not care whether the label is prefixed with a pound sign or not. Still, it would have been better for us to include the pound sign in the above example for clarity.

Many instructions have several versions allowing the use of different addressing modes (see `ld` for a good example of this). With these instructions, including the pound sign is not optional when specifying immediate addressing. The only safe rule, then, is always to prefix the label with a pound sign if you wish to specify the memory address of the label and not the contents of that address.

If you are not sure how a particular instruction has been assembled, you can always examine the `.lst` file produced by the assembler, and compare the hexadecimal opcodes listed to the left of the assembly instructions with the assembly opcodes given in the assembly language manual (Chapter 4 of the *Mnemonic Instruction Set*[**?**] reference).

4 Direct Addressing: Smem and others

In the modes called **direct addressing** by TI, the instruction opcode contains a memory offset (see the "dma" bits on page 5-8 of the *CPU and Peripherals*[**?**] reference) seven bits long, which is combined with either the `DP` (data pointer) or `SP` (stack pointer) register to obtain a complete 16-bit data-memory address. This divides the data memory into pages of 128 words each.

`SP` is initialized for you in the core file and should not need to be modified. `SP`-referenced direct addressing is used by the `pshd`, `pshm`, `popd`, and `popm` instructions for stack manipulation, as well as by all subroutine calls and returns, which save program addresses on the stack.

`DP`-referenced direct addressing is available wherever you see the `Smem` abbreviation in an assembly syntax description. The advantage of `DP`-referenced addressing over the `*(lk)` form described in the next section is that `DP`-referenced addressing will not add an extra instruction word (and corresponding extra machine cycle). The disadvantage is that it is limited to 128 words of contiguous memory, and you have to make sure that `DP` points to the right 128 words. `DP` may be changed with the `ld` instruction as needed.

Examples:

```
ld      10,A       ; sets A = (contents of memory location DP + 10)
add     6,B        ; sets B = B + (contents of memory location DP + 6)
```

> NOTE: Make sure you understand that the numbers 10 and 6 above are interpreted as memory addresses, **not** data values. To get data values, you would need to use a pound sign in front of the numbers.

5 Absolute Addressing: dmad, pmad, *(lk)/Smem

This seems to be TI's term for all the forms of direct addressing which it does not call direct addressing! It is represented in assembly-instruction syntax-definitions using one of the above abbreviations (`*(lk)` addressing is available when the syntax definition says `Smem`).

5.1 dmad

`dmad` (Data Memory ADdress) operands are used by `mvxx` data move instructions and represent 16-bit memory addresses in data memory whose contents are used in the instruction.

Example:

```
f3ptr   .word    0          ; reserve one word of storage; initialize to 0
. . . .
mvdm    f3ptr,AR4  ; set AR4 = memory address of f3ptr
```

5.2 pmad

`pmad` (Program Memory ADdress) operands are used by the `firs`, `macd`, `macp`, `mvdp`, and `mvpd` instructions, as well as all subroutine calls and branching instructions. They represent 16-bit addresses in program memory whose contents are used in the instruction, or jumped to in the case of branch instructions. Other than subroutine calls and branches, the most common use of a `pmad` is for the `firs` instruction.

Example:

```
firs    *AR3+,*AR4+,coefs
```

NOTE: `coefs` is a label in the program section of the code, **not** the data section.

5.3 *(lk)

`*(lk)` addressing is a syntactic oddity. The asterisk symbol generally means that indirect addressing is being used (see below), but this is actually direct addressing with a 16-bit data memory address encoded in the instruction's last word. The reason for the asterisk is that TI **does** set the "I" bit in the opcode, usually denoting indirect addressing, and this form can only be used when an `Smem` is called for in the assembly syntax. Other bits in the low byte of the first instruction word tell the processor that the "`*(lk)` exception" is to be used, and to fetch the memory address in the next word (see the MOD bits on page 5-10 of the *CPU and Peripherals*[**?**] reference). You can easily recognize this addressing mode in `.lst` files because the low byte of the first instruction word always equals `F8h`.

Examples:

```
hold    .word    1           ; reserve one word of storage and initialize to 1
count   .word    0           ; reserve one word of storage and initialize to 0
```

```
. . . .
ld       *(count),B  ; sets B = 0 (assuming memory was not changed)
st       T,*(hold)   ; sets (storage location at address hold) = T
```

6 Indirect Addressing: Smem, Xmem, Ymem

Indirect addressing on the TMS320C54x always uses the auxiliary registers `AR0` through `AR7` and comes in two basic flavors. These are easily recognized from the assembly language syntax descriptions as either `Smem` or `Xmem/Ymem`.

6.1 Smem

In `Smem` indirect addressing, only one indirect address is used in the instruction and a number of variations is possible (see the table on page 5-13 of the *CPU and Peripherals*[**?**] reference). An asterisk is always used, which signifies indirect addressing. Any of the registers `AR0-AR7` may be used, with optional modifications: automatic post-decrement by one, pre- and post-increment by one, post-increment and post-decrement by `n` (`n` being stored in `AR0`), and more, including many options for circular addressing (which automatically implements circular buffers) and bit-reversed addressing (which is useful for FFTs).

6.2 Xmem/Ymem

`Xmem/Ymem` indirect addressing is generally used in instructions that need two different indirect addresses, although there are a few instances where an `Xmem` by itself is specified in order to save bits in the opcode for other options. In `Xmem/Ymem` indirect addressing, fewer bits are used to encode the option modifiers in the opcode; hence, fewer options are available: post-increment by one, post-decrement by one, and post-increment by `AR0` with circular addressing.

Examples:

```
stl    B,*AR6     ; sets (contents of location pointed to by AR6) = low word of B
stl    B,*AR6+0%  ; sets (contents of location pointed to by AR6) = low word of B,
;      then increments AR6 with circular addressing
mar    *+AR3(-6)  ; decrements AR3 by 6 (increment by -6)
```

NOTE: The `mar` (modify address register) instruction is unusual in the sense that it takes an `Smem` operand but does nothing with the data pointed to by the `ARx` register. Its purpose is to perform any of the allowed register modifications discussed above without having to do anything else. This is often handy when you are using an `Xmem/Ymem`-type instruction but need to do an `ARx` modification that is only allowed with an `Smem`-type operand.

7 Summary

The `ld` instruction is illustrative of the many possible addressing modes which can be selected with the proper choice of assembly language syntax:

```
ld      #0,A         ; immediate data:  sets A = 0
ld      0,A          ; DP-referenced direct:  sets A = (contents of the address DP + 0)
ld      mydata,A     ; DP-referenced direct:  sets A = (contents of the address
;       DP + lower seven bits of mydata)
ld      #mydata,A    ; immediate data:  sets A = 16 bit address mydata
ld      *(mydata),A  ; *(lk) direct:  sets A = (contents of the 16 bit address mydata)
ld      B,A          ; accumulator:  sets A = B
ld      *AR1+,A      ; indirect:  sets A = (contents of address pointed to by AR1),
;       and afterwards increments AR1 by one
ldm     AR2,A        ; memory-mapped register:  sets A = AR2
```