A "Real-World" Microprocessor: Basic MSP430 Assembly from Roots in LC-3*

Matthew Johnson

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0†

Abstract

In this lab, students apply what they have learned to implement some basic assembly coding principals on real world hardware. They take what they know from the educational LC-3 and apply the basic principals to a new hardware situation.

1 An Intro to the MSP430 from the LC-3

This week you will go over the basic differences between the MSP430's assembly ISA and the LC-3's, and learn how to write a basic assembly program for the MSP-430 using TI's Code Composer Studio. You have two main tasks ahead of you:

1. Following the breadboard assembly instructions, put together a basic I/O package for the MSP430 launchpad. Once you have everything (hopefully) wired correctly, the pre-flashed test program should run correctly. If it doesn't, work with the labbies to troubleshoot your board!
2. Coding in MSP430 Assembly, **implement a Fibonacci sequence calculator**. This should be done with a loop and run infinitely. Step through, explain, and demonstrate the code, using the CCS4 Debugger. Be sure to view the registers while stepping through the program. Observe the amount of CPU cycles each of the instructions takes to complete. Detailed Instructions (Section 3: Part I Assignment Detail)

2 Some Background Information

2.1 Main Differences Between MSP430 and LC-3

- **The MSP430 has a larger assembly instruction set than the LC-3**

  MSP430 assembly includes some task specific instructions (Such as `inc` and `dec`) to simplify reading the language

*Version 1.4: Aug 16, 2011 1:45 pm +0000
†http://creativecommons.org/licenses/by/3.0/

Some MSP430 assembly instructions are interpreted instructions (Such as `pop` and `push`)

Definition 1: Interpreted Instructions

An instruction that is decomposed by the assembler into several smaller/ more basic fundamental instructions.

Example

`pop R3` contains two implicit instructions: `mov @SP, R3` and `add #0x02, SP`

Math and logical instructions are similar, but do not have a specific destination.

- MSP430 instructions come in two flavors, dual operand and single operand. Neither type has an explicit destination register, rather, the last operand serves as the destination too.
  For Example: `add R4, R5` in MSP430 assembly corresponds to `add R5, R4, R5` in LC-3

  WARNING: Be careful to not overwrite data you wish to keep! If you need to preserve the values in both operand registers, you will need to save one of them first using a `mov` instruction.

MSP430 Supports some byte as well as word instructions

- Some MSP430 instructions allow you to address and write/read from a specific 8 bit byte in memory instead of the entire 16 bit word. The MSP430 memory has byte level addressability, but word instructions only operate on even numbered memory addresses (implicitly modifying the next odd numbered memory byte too). In many cases, especially when working with memory mapped I/O registers, you may need to operate on one specific byte only. To do so, just add a `.b` onto the end of the assembly instruction
  For example: `mov.b #0, &P1DIR` sets 8 bit length P1DIR register to zero without accidentally modifying the registers around it.

  ASIDE: MSP430 assembly specifies `.w` for executing word length instructions as well as `.b` for bit length instructions. The assembler by default assumes word length, so you the programmer don't have to explicitly write `mov.w R5, R14` although you should be conscious that `mov R5, R14` means the same thing.

The MSP430 has 16 CPU registers

- The MSP430 has twice as many CPU registers as the LC-3. Like in the LC-3 though, some of the MSP430's registers are reserved for the MSP430 runtime environment. Registers R0-R3 are reserved (Program Counter, Stack Pointer, Status Register, and a Constant Generation Register respectively), leaving registers R4 through R15 available for general purpose use as defined by the programmer.
  In your assembly programs you have 12 general purpose registers at your disposal, but you also must manage and keep track of the additional options.

MSP430 Register Usage Diagram

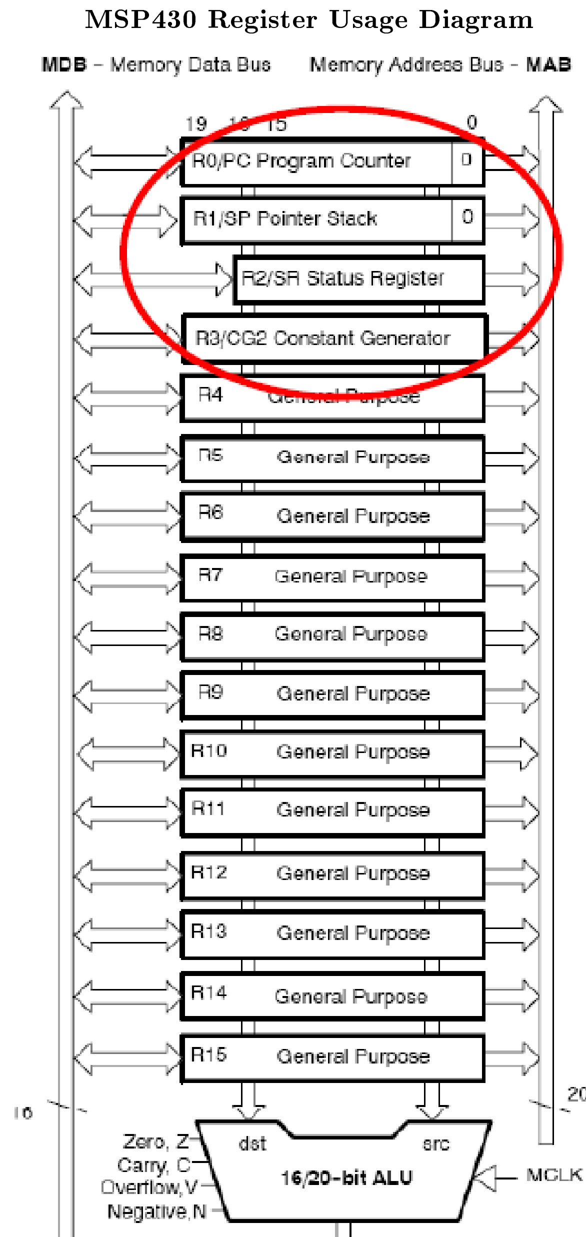

Figure 1

Indirect, relative, and absolute addressing occurs differently

- Instead of different indirect and direct load and store instructions (`LD`, `LEA`, `LDI`, etc...), the MSP430 uses one versatile `mov` instruction with different operand addressing modes.
  `mov` can both read and write from memory– it acts like both a load and store. (`mov R4, &0x0200` corresponds to a `ST` while `mov &0x0200, R4` corresponds to a `LD`) Be careful though, unlike in LC-3, `mov`**does NOT update the condition register.**
  Differentiate between the various direct and indirect modes by using special syntax to specify

the type of operand you want. This allows you to mix addressing types (read indirect and store direct, etc...) even though everything is in one mov instruction.

* Direct register access: Rn (where n is the number of a general purpose register) Example: `R4` refers directly to R4
* Immediate Values: #x (where X is an immediate numerical value or label) Example: `#02h` refers to the literal hex number 2
* Indirect Access From a Register: @Rn (where n is the number of a general purpose register) Example: `@R6` refers indirectly to the data stored in the memory location in R6
* Indirect Offset Access: x(Rn) (where n is the number of a general purpose register and x is either an literal offset or a label) Example: `0(R7)` refers to the data stored in the location in memory pointed to by R7

NOTE: This has the same end result as `@R7`. By TI code convention though, @Rn cannot be used to specify the destination of an operation, so if you wish to store a result indirectly, you must use the 0(Rn) syntax.

TIP: In this example R7 essentially contained the address while the literal offset was a small number. Offset Access can be very powerful when looked at the other way: where the literal contains a starting location in memory (potentially a label) and the register contains a small offset value incremented to access a series of locations in memory.

MSP430 Addressing Modes

Table 3–3. Source/Destination Operand Addressing Modes

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/–	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/–	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/–	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

Figure 2

You can also perform indirect or relative operand addressing with operations other than loads and stores

Example 1

`add @R4, R5` takes the data stored in the address pointed to by R4 and adds it with R5, storing the result in R5.

For more information, see the summary chart Figure 2 (MSP430 Addressing Modes) or the comprehensive MSP430 users guide[1] section 3.3.0 through 3.3.7

The MSP430 has two types of memory

- The MSP430 has both traditional RAM and non-volatile Flash memory. On a power reset, all values in RAM are cleared, so your program will be stored in Flash. The Flash write process is fairly involved, so we won't be writing to it in this class during run time (Code Composer will take care of loading your programs). In a nutshell, your program must store any temporary or changing values to RAM memory, although it can read your instructions and any preset constants from flash

  Important Memory Locations:

  0x0200 : The Lowest Address in RAM
  0x0280 : The Highest Address in RAM
  0xF800 : The Beginning of Flash Memory
  0xFFE0 : The Beginning of the Interrupt Vector Table

[1] See the file at <http://cnx.org/content/m37151/latest/MSP430 User Guide-slau144e.pdf>

MSP430 Memory Map

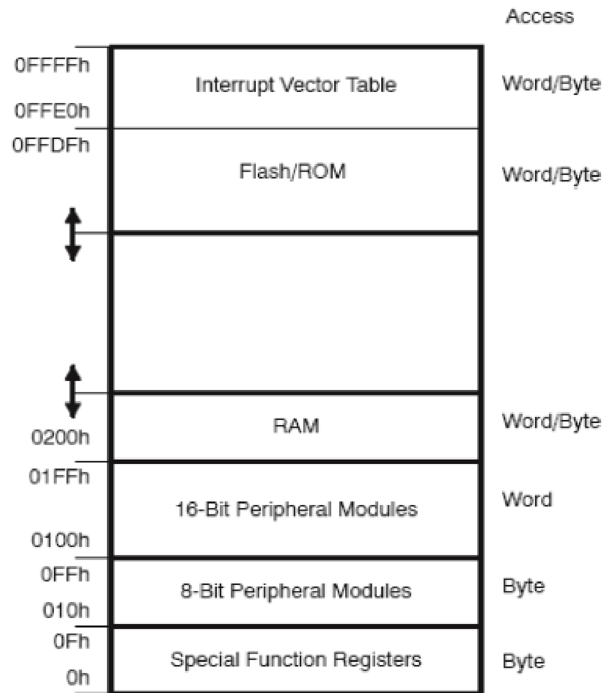

Figure 3

The MSP430 Uses Memory Mapped I/O Peripherals

- These devices function independently of the main processor, and use memory mapped registers to communicate with the program executing on the main CPU.
- Peripherals free up CPU resources and also allow more usage of low power CPU suspend modes. You'll learn more about peripherals in Lab 5

MSP430 Architecture Block Diagram

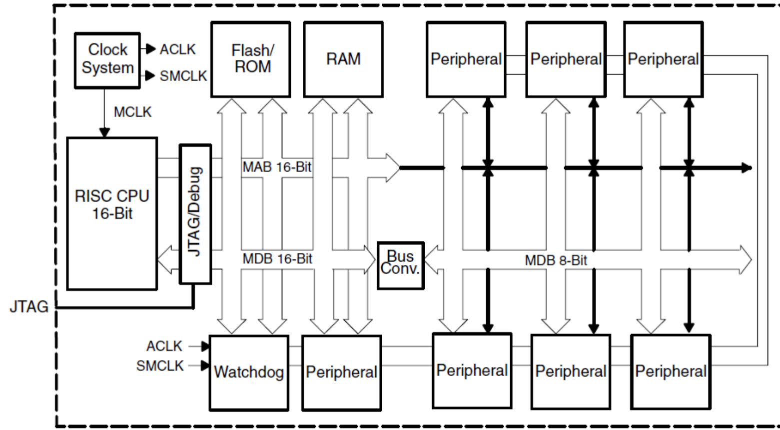

Figure 4

2.2 Example Code Translations

LC-3 Assembly	LC-3 Pseudocode	MSP430 Assembly	MSP430 Pseudocode
`AND R4,R5,R6;`	R4 <- R5 & R6	`mov.w R5,R4;` `and.w R6,R4;`	R4 <- R5 R4 <- R4 & R6
`BRz R4,Loop;`	if R4 == 0, branch to label "Loop"	`tst R4;` `jz Loop;`	load the attributes of R4 into the SR jump to label "Loop" if the zero bit is flagged

Table 1

2.3 Other Useful Information

The code composer debugger actually runs on the real MSP430 hardware through a JTAG interface. To debug code, **you have to have the launchpad board plugged into the computer**.

The debugger controls the CPU's clock (and therefore can monitor it). **To see how many clock cycles something takes, go to Target -> Clock -> Enable, and look in the bottom right corner of the screen for a small counter with a clock next to it.**

3 Part I Assignment Detail

Your task is to create a simple MSP430 assembly program using CCS4 and the MSP430 launchPad to calculate a Fibonacci sequence. You do not need to explicitly display the sequence, but rather use the Code Composer register view tools to watch the sequence progress as you step through your program.

To view the registers in Code Composer Studio v4, first start a debug session. Once you are in the debug perspective, you can go to View–> Registers to open the register dialog. From there, expand the section "Core Registers" to see your CPU registers, or the section "Port_1_2" to see the raw data from the input pins.

Enable the clock cycle monitor (Target–>Clock–>Enable) and you will see a yellow clock icon at the very bottom of your screen. This tells you how many actual CPU clock cycles have passed since you enabled it. Observe the different amounts of time that different instructions take.

Definition 2: The Fibonacci Sequence
The sequence of numbers starting with 0 , 1 in which N= (N-1) + (N-2) 0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

ASIDE: The Fibonacci sequence plays an important role in the natural world. It appears in many biological sequences, and is fundamentally linked to the famed "golden ratio." For more "fun" info about Leonardo Fibonacci, see the ever reliable Wikipedia [2]

Diagrams courtesy of TI document slau144e "MSP430 User's Guide"

The LC-3 was developed by Yale N. Patt (University of Texas at Austin) and Sanjay J. Patel (University of Illinois at Urbana-Champaign) and is used in their book Introduction to Computing Systems.

[2] http://en.wikipedia.org/wiki/Fibonacci