UT.6.01x Courseware (https://www.edx.org) https://courses.edx.org/courses/UTAustinX/UT...

UTAustinX: UT.6.01x Embedded Systems - Shape the World

KarenWest (/dashboard) ▼

Courseware (/courses/UTAustinX/UT.6.01x/1T2014/courseware) **Course Info (/courses/UTAustinX/UT.6.01x/1T2014/info)**

Discussion (/courses/UTAustinX/UT.6.01x/1T2014/discussion/forum) **Wiki (/courses/UTAustinX/UT.6.01x/1T2014/course_wiki)**

Progress (/courses/UTAustinX/UT.6.01x/1T2014/progress)

Questions (/courses/UTAustinX/UT.6.01x/1T2014/a3da417940af4ec49a9c02b3eae3460b/)

Syllabus (/courses/UTAustinX/UT.6.01x/1T2014/a827a8b3cc204927b6efaa49580170d1/)

For this online class, Section 2.6 can be skipped. It is included for the advanced learner wishing to know some of the details inside the processor. This section focuses on the ARM® Cortex™-M assembly language. There are many ARM® processors, and this class focuses on Cortex-M microcontrollers, which executes Thumb® instructions extended with Thumb-2 technology. This class will not describe in detail all the Thumb instructions. Rather, we focus on only a subset of the Thumb® instructions. This subset will be functionally complete without regard to minimizing code size or optimizing for execution speed. Furthermore, we will show general forms of instructions, but in many cases there are specific restrictions on which registers can be used and the sizes of the constants. For further details, please refer to the ARM® Cortex™-M Technical Reference Manual.

Help

Assembly language instructions have four fields separated by spaces or tabs. The **label field** is optional and starts in the first column and is used to identify the position in memory of the current instruction. You must choose a unique name for each label. The **opcode field** specifies the processor command to execute. The **operand field** specifies where to find the data to execute the instruction. Thumb instructions have 0, 1, 2, 3, or 4 operands, separated by commas. The **comment field** is also optional and is ignored by the assembler, but it allows you to describe the software making it easier to understand. You can add optional spaces between operands in the operand field. However, a semicolon must separate the operand and comment fields. Good programmers add comments to explain the software.

```
Label  Opcode Operands       Comment
Func   MOV    R0, #100    ; this sets R0 to 100
              BX   LR     ; this is a function return
```

Observation: A good comment explains why an operation is being performed, how it is used, how it can be changed, or how it was debugged. A bad comment explains what the operation does. The comments in the above two assembly lines are examples of bad comments.

When describing assembly instructions we will use the following list of symbols

Ra Rd Rm Rn **`Rt`** and **`Rt2`** represent registers
`{Rd,}` represents an optional destination register
`#imm12` represents a 12-bit constant, 0 to 4095
`#imm16` represents a 16-bit constant, 0 to 65535
`operand2` represents the flexible second operand as described in Section 3.4.2
`{cond}` represents an optional logical condition as listed in Table 2.10
`{type}` encloses an optional data type

{S} is an optional specification that this instruction sets the condition code bits

Rm {, shift} specifies an optional shift on **Rm**

Rn {, #offset} specifies an optional offset to **Rn**

For example, the general description of the addition instruction

```
ADD{cond} {Rd,} Rn, #imm12
```

could refer to either of the following examples.

```
ADD   R0,#1        ; R0=R0+1
ADD   R0,R1,#10    ; R0=R1+10
```

Table 2.10 shows the conditions **{cond}** that we will use for conditional branching.

Suffix	*Flags*	*Meaning*
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or **HS**	C = 1	Higher or same, unsigned ≥
CC or **LO**	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N ≠ V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N ≠ V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix specified

Table 2.10. Condition code suffixes used to optionally execution instruction.

It is much better to add comments to explain how or even better why we do the action. Good comments also describe how the code was tested and identify limitations. But for now we are learning what the instruction is doing, so in this chapter comments will describe what the instruction does. The assembly **source code** is a text file (with Windows file extension **.s**) containing a list of instructions. If register R0 is an input parameter, the following is a function that will return in register R0 the value (100*input+10).

```
Func MOV   R1,#100     ; R1=100
     MUL   R0,R0,R1    ; R0=100*input
     ADD   R0,#10      ; R0=100*input+10
     BX    LR          ; return 100*input+10
```

The **assembler** translates assembly source code into **object code**, which are the machine instructions executed by the processor. All object code is halfword-aligned. This means instructions can be 16 or 32 bits wide, and the program counter bit 0 will always be 0. The **listing** is a text file containing a mixture of the object code generated by the assembler together with our original source code.

```
Address        Object code     Label  Opcode  Operand          comment
0x000005E2 F04F0164  Func MOV   R1,#0x64     ; R1=100
0x000005E6 FB00F001       MUL   R0,R0,R1     ; R0=100*input
0x000005EA F100000A       ADD   R0,R0,#0x0A  ; R0=100*input+10
0x000005EE 4770           BX    LR           ; return 100*input+10
```

When we **build** a project all files are assembled or compiled then linked together. The address values shown in the listing are relative to the particular file being assembled. When the entire project is built, the files are linked together, and the **linker** decides exactly where in memory everything will be. After building the project, it can be downloaded, which programs the object code into flash ROM. You are allowed to load and execute software out of RAM. But for an embedded system, we typically place executable instructions into nonvolatile flash ROM. The listing you see in the debugger will specify the absolute address showing you exactly where in memory your variables and instructions exist.

VIDEO 2.8.5 INSTRUCTION FORMAT

Help

0:00 / 0:00 1.0x

DR. RAMESH YERRABALLI: OK, so now that we understand what a register is, and

the fact that we have instructions that will manipulate these registers,

let's understand what an instruction looks like.

That is, the format of an instruction.

An instruction--

we'll take a simple example--

Help