Assmebly Programming Guide

1 Introduction

In Lab Exercise 3, you are asked to write an assembly program vecsum( ) which can sum up two arrays placing the result into a third array. In this guide, we will give you a brief overview of assembly programming and show you how to write an assembly function veccopy( ) which copies the content of one array into another array. With a little modification you can change it into vecsum( ). It would be helpful to you if you wrote your vecsum( ) prior to coming to lab and tested your code on your home PC using the free evaluation CCS.

In the following sections we will give a brief intro on C5510 registers, assembly instructions, and finally demonstrate the veccopy( ) program.

2 C5510 Registers

Although there are many registers on a C5510 DSP processor, only some of them are commonly used: temporary registers, (extended) auxiliary registers, accumulators, and block repeat registers.

2.1 Temporary Registers: T0, T1, T2, T3

These registers are all of length 16 bits. Usually they are used to store 16-bit data values or constant values temporarily. These registers are very handy in assembly programming. Here are some examples:

```
Ex1:
    MOV #3, T0    ; Move the constant value -3 into T0

Ex2:
    ADD #-1, T0   ; T0 = T0 - 1
```

One thing to note here is that some assembly instructions can change the value of T3 automatically by themselves. If you store a data in T3, the value might be changed by some instructions without you knowing it. So try to avoid using T3 unless you know assembly instructions very well.

2.2 Auxiliary Registers: AR0, AR1, · · ·, AR7

Auxiliary registers are usually used as a pointers to store data addresses. Sometimes they are also used to store data values when we run out of registers to store data values. When we store a data address in ARx, the data value stored in

that address can be accessed using *ARx. Pointer increment can be done to ARx before and after the data value *ARx is accessed depending on the syntax. Consider the memory map in Fig. 1. If we run the following example code (Ex1–Ex7), we will get different results for AR1 and T1 after evaluating the code. For more detailed information, refer to Sec. 6.4 of *TMS320C55x DSP CPU Reference Guide* (spru371f.pdf).

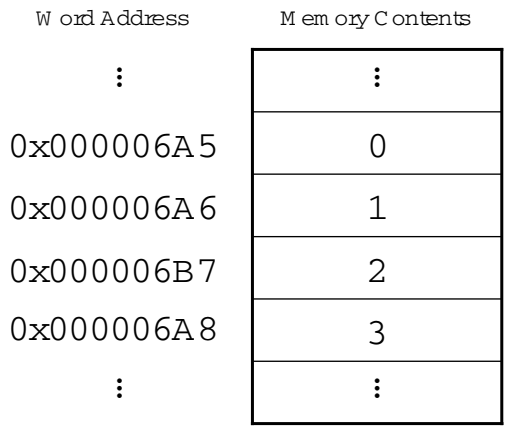

Figure 1: Memory map

```
Ex1:
    MOV #0x000006A5, AR1    ; Move address 0x000006A5 into AR1
    MOV *AR1, T1            ; Move the data *AR1 into T1;

Result: AR1 is UNCHANGED in the second line;AR1=0x000006A5, T1=0.


Ex2:
    MOV #0x000006A5, AR1
    MOV *+AR1, T1

Result: AR1 is incremented by 1 BEFORE we access the memory
(called pre-increment), thus AR1=0x000006A6, T1=1.


Ex3:
    MOV #0x000006A5, AR1
    MOV *AR1+, T1
```

```
Result: AR1 is incremented by 1 AFTER we access the memory (called
post-increment), thus AR1=0x000006A6, T1=0.


Ex4:
    MOV #0x000006A5, AR1
    MOV *AR1(#2), T1

Result: AR1 is not incremented, but we access the memory address
AR1+2. Hence AR1=0x000006A5, T1=2.


Ex5:
    MOV #0x000006A5, AR1
    MOV *+AR1(#2), T1

Result: AR1 is pre-incremented by 2 BEFORE we access the memory,
thus AR1=0x000006A7, T1=2.


Ex6:
    MOV #0x000006A5, AR1
    MOV #3, T0
    MOV *AR1(T0), T1

Result: AR1 is not incremented, but we access the memory address
AR1+T0, thus AR1=0x000006A5, T1=3.


Ex7:
    MOV #0x000006A5, AR1
    MOV #3, T0
    MOV *(AR1+T0), T1

Result: AR1 is post-incremented by T0 AFTER we access the memory,
thus AR1=0x000006A8, T1=0.
```

Note that although we only show the ARx increment example here, the syntax of ARx decrement is very similar. You only have to replace the "+" in the above examples by "−". One last note, auxiliary register ARx is part of extended auxiliary register XARx, i.e. XARn=(ARxH ARx). Since we usually work within one memory page, the higher 7 bits ARxH rarely change. Therefore we usually to work on ARx rather than on the long registers XARx.

2.3 Accumulators: AC0, AC1, AC2, AC3

When we compute the multiplication of two 16-bit numbers, the result is of legnth 32 bits. Ordinary registers are either of 16-bit or 23-bit and thus not long enough to store the multiplication result. Accumulators are registers of length 40 bits, hence they are long enough to store the multiplication result. Moreover, since they are of 40-bit, they are less vulnerable to the overflow problem if we add many numbers together (for instance: convolution). As a result, we usually use accumulators to store the arithmetic results.

```
Ex1:
    MOV *AR0+, AC0  ; Move the data *AR0 into AC0; AR0 = AR0 + 1
    ADD *AR1+, AC0  ; AC0 = AC0 + *AR1; AR1 = AR1 + 1
    MOV AC0, *AR2+  ; *AR2 = AC0; AR2 = AR2 + 1
```

2.4 Block Repeat Registers: BRC0, BRC1

These two registers are used when we have use the looping instructions RPTBLOCAL and RPTB. If the is only one loop, it will be executed BRC0 + 1 times; if there is an inner loop nested inside an outer loop, the outer loop will be executed BRC0 + 1 times while the inner loop BRC1 + 1 times.

```
Ex:
    MOV #4, BRC0          ; Outer loop 4 + 1 = 5 times
    MOV #2, BRC1          ; Inner loop 2 + 1 = 3 times
    MOV #0, T0;
    RPTBLOCAL OUTLOOP-1 ; Outer loop starts
    RPTBLOCAL INLOOP-1  ; Inner loop starts
    MOV *AR0+, *AR1
    ADD T0, *AR1+
INLOOP                  ; End of the inner loop
    ADD #1, T0
OUTLOOP                 ; End of the outer loop
```

3 Assembly Instructions

The syntax of an assembly instruction is: consists of four fields:

```
[label]   mnemonic    operand list    [;comment]
```

where [] indicates the filed is optional. You are not required to give label to each assembly instruction line. If a label is not assigned to the line, you must leave that column blank, otherwise the code will not work.

Assembler is very strict on the syntax. The syntax has to be strictly followed, otherwise it will not work. How do we find the syntax of an assembly instruction? You can find the description of its syntax on *TMS320C55x DSP Mnemonic Instruction Set Reference Guide* (spru374g.pdf). For instance, if we look at the syntax of the branch instruction XCC on page 5-507 of spru374g.pdf, the syntax is:

```
XCC label, cond
```

What is label? It is the label field we mentioned earlier. Now what is cond? To know what cond is, we should read Chapter 1 of spru374g.pdf which defines the terms, symbols, and abbreviations used in the guide. In Sec. 1.2 you can find the list of all of the possible testing conditions for cond field. If you read through it, you will find that all of the testing conditions are comparisons against 0. What happens if your testing condition is *AR1 == 1?? You need to make some changes. For instance, assume that you want to write a assembly program such that **if *AR1 equals to 1, AR2 is incremented by 1**. If you look at Table 1-3 of spru374g.pdf, you will find that we can not use *AR1 directly for comparison. We can only use ACx, ARx, and Tx for comparison. So what should we do? We should move the value of *AR1 into T0 first and then do the comparison. As we mentioned earlier we can only compare against 0 in the cond field. How should we solve the problem? One way would be to subtract 1 from T0 first and then compare it against 0. The resulting code would be

```
    MOV *AR1, T0      ; T0 = *AR1
    ADD #-1, T0       ; T0 = *AR1 - 1
    XCC LINE, T0==#0 ; If T0 = *AR1 - 1 == 0, do the following
                      ; instructions, otherwise jump to LINE
    ADD #1, AR2       ; AR2 = AR2 + 1
LINE
```

Here we only use XCC as our example, but this is generally what is done when people program in assembly: figure out what instruction to use, find the description of the syntax on the Mnemonic Instruction Set Reference Guide (spru374g.pdf), and then do the necessary changes to your program to make sure the syntax is correct. Other than XCC, MOV, ADD, RPTBLOCAL that we see in the examples of this note, there are other instructions that are commonly used: RPTB, SFTS, XCCPART,SUB, PSH, POP, MPY, MAC and RET. We will talk about them in the lectures soon.

4 Assembly Programming Example

Here we will use the instructions we have seen so far to write a assembly function which is called from C to copy the content of an array into another array. The

function has the prototype:

```
void veccopy(int *v1, int *v2, int length, int shift);
```

It does the following:

```
for (i=0;i<length;i++)
{
    v2[i] = v1[i];
    if (shift ==1 ) v2[i] = v2[i] << 1;
}
```

How should we write a assembly program to achieve this? First we need to know where the values of v1, v2, length, and shift are located when the C program hands the control to the assembly function. From what we learned in the lecture, you should know that

- v1 → AR0
- v2 → AR1
- length → T0
- shift → T1

With this information, we can write the assembly program easily as follows:

```
    .global _veccopy  ; Declare the function to be global so C
                      ; can call it

_veccopy              ; Start to define the function
    ADD #-1, T0;      ; T0 = length - 1    <-- Why minus 1?
    ADD #-1, T1;      ; T1 = shift -1      <-- Why minus 1?
    MOV T0,  BRC0;    ; Repeat the following loop T0 + 1 = length times
    RPTBLOCAL LOOP-1  ; Start of the loop
    MOV *AR0+,  T0;   ; T0 = *AR0; AR0 = AR0 + 1 <-- Why increment?
    XCC SHIFT, T1==#0 ; If T1 = shift - 1 == 0, do the following
    SFTS T0, #1;      ; Shift T0 = *AR0 to the left by 1
SHIFT
    MOV T0, *AR1+;    ; *AR1 = T0; AR1 = AR1 + 1 <-- Why increment?
LOOP
    RET
```

If you can answer those "why" questions in the comments, you should know the program pretty well. With just a few changes of this code, you can come up with your own vecsum() program code. You can use the following C code on CCS to test if your code is working correctly. Remember to test your code before coming to Lab 3 to so you can get the credits for the prelab problem.

```
#include <stdio.h>

#define BUFSIZE 4

int vec1[BUFSIZE] = {0, 1, 2, 3};
int vec2[BUFSIZE] = {5, 6, 7, 8};
int sum[BUFSIZE] = {0, 0, 0, 0};

void vecsum(int *v1, int *v2, int *v3, int length, int shift);

void main() {

    int i;

    vecsum(vec1, vec2, sum, BUFSIZE, 1);

    for(i = 0; i < BUFSIZE; i++)
        printf("%d\n", sum[i]);
}
```