

Tutorial 5: Memory Access

Problem 1: Registers, Memory and the Code Compilation

Consider the C code in Figure 1. Draw a memory map for all the variables in the program.

```

int main (void)
{
int a[ ] = {100, 101, 102, 103, 104, 105,106, 107, 108, 109};
int b[ ] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
int i,temp;
    for(i=0;i<10;i=i+1)
    {
        temp = b[i];
        b[i] = a[i];
        a[i] = temp;
    }
return 0;
}

```

Figure 1: Program Data Structures

In the lectures we have several times discussed that variables refer to memory locations with each location (variable) having an address and a content. We have also seen repeatedly that C variables associate with the processor registers. In this problem, we take a closer look at analysing the C variables and their associations with memory and registers .

If we take the view that variables refer to memory locations, then the memory map and the assembly program corresponding to the loop portion of the C program in Figure 1 are presented in Figure 2.

| | |
|--|--|
| <pre> ; assume v1 contains the address of a[] mov a1, #0 ; str a1, [v1,#120] ; i = 0 loop: ldr a1, [v1,#120] ; i cmp a1, #10 ; test for (i < 10) beq fin ; loops end add v2, v1, #40 ; address of b[0] ldr a2, [v2, a1, lsl #2] ; a1<<2 equivalent to i*4 str a2, [v1, #124] ; temp = b[i] ldr a3, [v1, a1, lsl #2] ; str a3, [v2, a1, lsl #2] ; b[i] = a[i] ldr a2, [v1, #124] ; str a2, [v1, a1, lsl #2] ; a[i] = temp add a1, a1, #1 ; str a1, [v1,#120] ; i++ b loop ; monr more loop fin:word 100, 101, 102, 103, 104, 105, 106, 106, 107, 108, 109 ; a[] .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 ; b[] .space 4 ;i .space 4 ;temp </pre> | |
|--|--|

Figure 2: Memory Map and the Assembly Instructions

In Figure 2 we assume that register V1 contains the address of a[0] and the array elements a[0] .. a[9] and b[0] .. b[19], and variables i and temp are placed in consecutive memory locations. In this code, memory locations corresponding to index i and temp are accessed with an “offset” from a “base” (a[0]). Also note that addresses of a[i], and b[i] are separated by (10 × 4 = 40) bytes. In each iteration of the loop index i is loaded onto register A1. Register A1 is used as an “index” register with its content shifted by two bits (multiplied by 4) and added to the “base registers” V1 for accessing a[i] (ldr a3, [v1, a1, lsl #2]), and V2 for accessing b[i] (ldr a2, [v2, a1, lsl #2]). Also note that in each iteration of the loop index i is incremented, until it reaches 10 and the loop falls through.

We make three important observations about the loops in programs of Figure 1 and Figure 2. First observation is that the scope of loop index i and variable temp does not extend beyond the loop. That means these C variables are only needed during the iterations of the loop. The second observation is that for every iteration of the loop the program needs to access the memory twice to load loop index i and save its incremented version. The same is true for the variable temp. Every save of b[i] in variable temp is followed by its load for save in a[i]. The last observation is that the C assignment (a[i] = b[i]); cannot be done directly in the assembly. In the assembly equivalent code, we need a load from a[i] (ldr a3, [v1, a1, lsl #2]) followed by a store in b[i] (str a3, [v2, a1, lsl #2]). Register A3 is used as a temporary staging location for the storage of a[i].

These three observations provide us with a clue to reflect back on the assembly code of Figure 2. We first note that load and store instructions corresponding to loop index i (highlighted in grey) are superfluous and can be dropped. That is because register A1 always keep track of the loop index i. Furthermore, we note that load and store instructions corresponding to variable temp (highlighted in bold) are superfluous and can be dropped. That is because register A2 always keep track of the variable temp. That means we can simply associate loop index i and variable temp to register A1 and A2 and avoid using memory for them.

The consequence of this simplification is a smaller code and a much faster code, as accessing memory is always many times slower than accessing registers. The revised memory map and the assembly program corresponding to the loop portion of the C program in Figure 1 are presented in Figure 3.

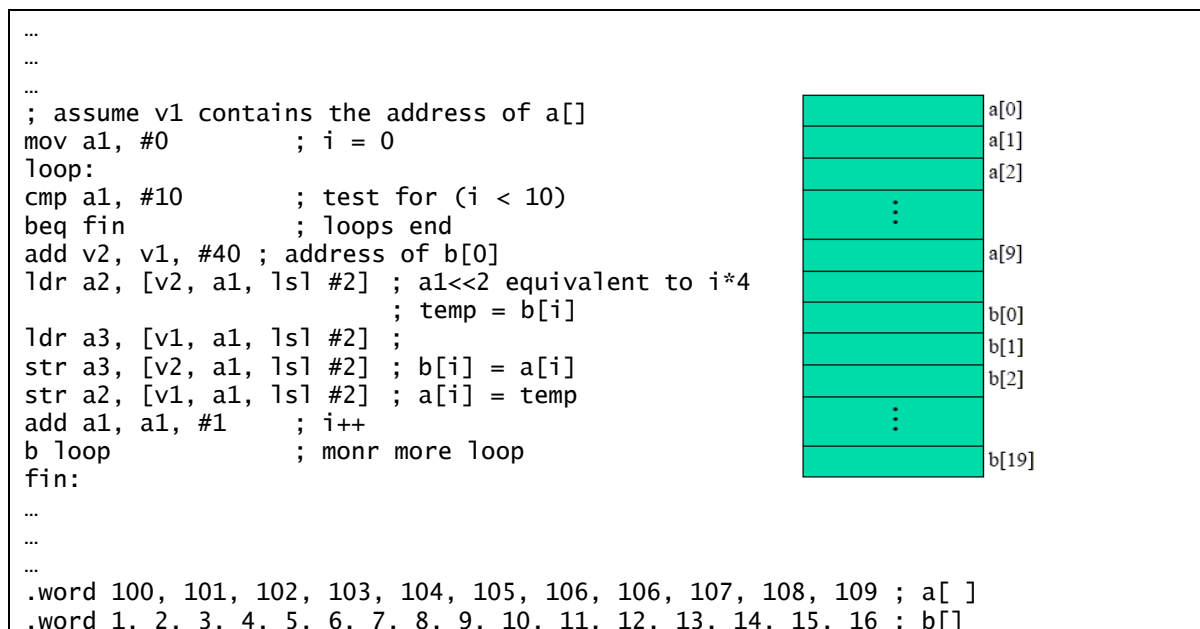


Figure 3: The Revised Memory Map and Assembly Instructions

Problem 2: Little and Big Endian Machines

Consider the C code in Figure 4.

What are the outputs of the printf statements?

Write an equivalent assembly language program for the part corresponding to (c, *(c+1), *(c+2), *(c+3))

```
#include <stdio.h>

int main (void)
{
    int a=287454020;
    char *c;
    c = ( char *) &a;
    printf("Number as integer = \"%x\"\n\n", a);
    printf("Number as 4 characters = \"%x%x%x%x\"\n\n", *c, *(c+1), *(c+2),
*(c+3));
    return 0;
}
```

Figure 4: Assembly Program on Big and Little Endian

The variable (a = 287454020) is represented in a 1-word 4-byte hexadecimal format as (a = 0x11223344). Type casting address of the integer type variable (a) to the character type pointer (c) has the effect of pointing the pointer (c) to the most significant byte of (0x11223344), which is (0x11), if the machine is a “big endian” or least significant byte of (0x11223344), which is (0x44), if the machine is a “little endian”. Figure 5 shows the print out of the printf statements in both cases.

```
//If the Machine is "big endians"
Number as integer = "11223344"
Number as 4 characters = "11223344"

//If the Machine is "little endians"
Number as integer = "11223344"
Number as 4 characters = "44332211"
```

Figure 5: The Outputs of printf Statements

The equivalent segment of the assembly language program for the part corresponding to (*c, *(c+1), *(c+2), *(c+3)) would be something like the code in Figure 6.

```
ldr    r4, =a
ldrb  r0, [r4, #0] ; *c
ldrb  r1, [r4, #1] ; *(c+1)
ldrb  r2, [r4, #2] ; *(c+2)
ldrb  r3, [r4, #3] ; *(c+3)
a:    .word 287454020 ; a
```

Figure 6: Assembly Code Corresponding to (c, *(c+1), *(c+2), *(c+3))

Problem 3: Word Alignment

Consider the C code in Figure 7.

What are the outputs of the printf statements?

Write an equivalent assembly language program for the parts corresponding to (a1, a2, a3, a4, a5) and (*c).

```

#include <stdio.h>

int main (void)
{
    static char  a1=20, a2 =54, a3 = 74, a4 = 28, a5 = 34;
    int *c;
    c = ( int *) &a2;
    printf("Number as 5 characters = \"%x%x%x%x%x\\\"\\n\\n",a1, a2, a3, a4, a5);
    printf("Number as integer = \"%x\\\"\\n\\n", *c);
    return 0;
}

```

Figure 7: The Outputs of printf Statements

Each variable of type character requires one byte. Assuming the first variable (a1) is stored at address 0x80, the mapping in Figure 8 presents the mapping of variables in the memory. The first printf statement simply prints the variables (a1) to (a5) in sequence. The C statement (c = (int *) &a2) type casts the address of the character type variable (a1) (0x81) to an integer pointer type (c). Since (c =0x81) is not a multiple of 4, (*c) in the second printf statement causes memory misalignment and rotation of bytes in the word when it is accessed. The amount of rotation = (c % 4) × 8 = 1 × 8 = 8 bits. That means that what is read by (*c) is 0x141c4a36.

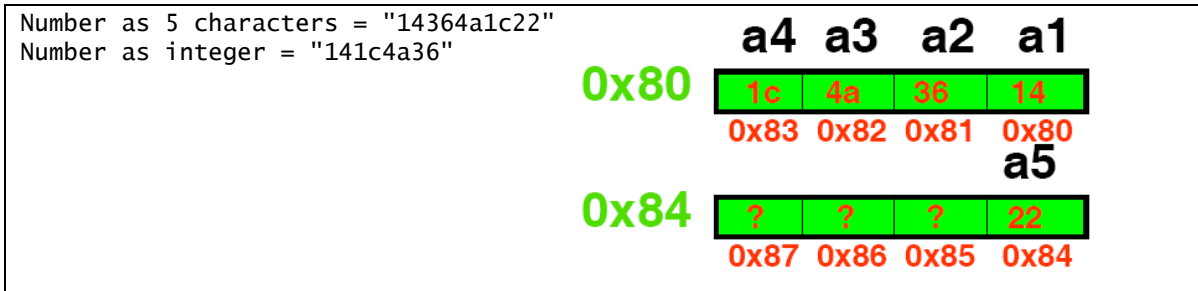


Figure 8: The Outputs of printf Statements and the Memory Map for Memory Alignment

The equivalent segments of the assembly language program for the parts corresponding to (a1, a2, a3, a4, a5) and (*c) are shown in Figure 9.

```

ldr    r7, =a
ldrb  r0, [r7, #0] ;a1
ldrb  r1, [r7, #1] ;a2
ldrb  r2, [r7, #2] ;a3
ldrb  r3, [r7, #3] ;a4
ldrb  r4, [r7, #3] ;a5
...
...
...
ldr    r6, [r7, #1] ;*c

a1:    .byte 20
       .byte 54
       .byte 74
       .byte 28
       .byte 20

```

Figure 9: Assembly Program Corresponding to Memory Alignment