Structureandarrays

Module 0374: Structures and arrays in TTPASM
Arrays
Structures
TTP implementation
Structure definition
Accessing members of a structure
1. Arrays
Most compilers allocate space for arrays in a straightforward way. Let us consider the following
declaration where TYPEX is the name of a type, and BUFLEN is a natural number.
TYPEX buffer[BUFLEN]; The total number number of bytes used by variable buffer is sizeof(TYPEX)*BUFLEN.
In terms of address (for byte-addressable architectures), buffer[i] has an address of
buffer + sizeof(TYPEX) * i.
General indexing is difficult in TTPASM because of the lack of a multiplication instruction.
However, indexing into an array of byte-size elements is easy. The following is an example:
// assume the address of an array is in register c // assume the index is in register b // assume an element is one byte wide add b,c // b is now the address of the element at the index If the size of an element is a power of 2, the code is still relatively simple:
// assume the address of an array is in register c // assume the index is in register b // assume an element is four-byte wide add b,b // b=2*b add b,b // b=2*b, now it is 4x the initial value add b,c // b is now the address of the element at the index However, there is no easy way to generate a template of code given the size of each array
element.
Pointer arithmetic is generally as difficult. However, pointer increment can be done easily.
Consider the following C code:
TYPEX *ptr; //... ptr+ 1 // compute the value of this expression The equivalent TTPASM code is as follows:
// assume the value of ptr is in register a // assume the size of TYPEX is defined by the label TYPEX$_sizeldib,TYPEX_sizeadda,b//$
$Manyalgorithmsrelatedtoarraysonlyneedtoaccesselementssequentially.Asaresult,the$
code can be converted to use pointer-arithmetic operations. For example, the following C code
computes the sum of an array:
for (i = 0 , sum = 0 ; i ¡ N; ++i) sum += a[I]; It can be changed to use pointer arithmetic as follows:
for (i= 0 , sum= 0 , ptr=a; i¡N; ++i) sum += *(ptr++); Note that the expression sum += *(ptr++) is the same as first performing sum +=* ptr, then
performing ptr++.
1. Structures
A general structure definition in C/C++ looks like this:
struct STRUCTX TYPE1 m1; // first member TYPE2 m2; // ... TYPEn mn; // last member ; The word width of an architecture is the width of the data bus in the processor core. Currently,
most production processors have a word-width of 64 bits, also known as 8 bytes. If a member
is of a scalar type, then a compiler attempts to make sure the entire member can be accessed
in a single memory operation based on the word width of the processor.
Unless otherwise instructed using a pragma, a compiler sequentially allocates storage for
members within a structure. The offset of a member from the beginning of a structure is based
on the offset of a previous member, but aligned to ensure each elemental type can be
accessed in a single memory cycle.
The offset to the first member is easy to compute because it is at the beginning of the entire
structure, the offset is 0 (zero).
The offsets to the rest of the members are a little more complicated. Let offset(m) refer to
the byte offset to member m in a structure, and alignment(m) refers to the alignment width
of member m (to be defined). Then
. The symbol is the ceiling
function that returns the smallest integer greater than or equal to.
The alignment width of a structure is the maximum of the alignments of its members but is also
restricted by the architecture's word width (in bytes). In this example, alignment(STRUCTX) =
min(wordWidth, max(alignment(m1), alignment(m2),... alignment(mn))). For a 64-bit
architecture, wordWidth is 8.

alignment(t)=min(wordWidth, sizeof(t)).

1. 1 TTP implementation

2. 1. 1 Structure definition

TTP has a wordWidth of 1. As a result, there are no alignment issues!

The concept of a struct is merely a matter of tracking the offset from the beginning of a structure to the members, please an overall size of a struct.

For example, let us consider the following C struct definition:

struct X uint8$_tx; structX*ptr; uint8_ty; ; ThistranslatestothefollowinglabelsinTTPASM:$

X$_x : 0X$*ptr:* $Xx1 + offset(mi+1) = offset$

(mi)+sizeof(m i) alignment(m i + 1) ×alignment(m i +$^1)x$

x

X*y:* Xptr 1 + X*size:* Xy 1 +

1. 1. 2 Accessing members of a structure

Because labels are used to track the offset of a member from the beginning of a structure, given the address of a structure is in a register already, the address of a member is the sum of the address of the structure and the offset to the member. Using the example from the previous section, the following is an example:

// assume the address of a struct X is in register b ldi a,X$_y addb, a//bisnowtheaddressofmemberyofthestructX$