Make assembly programs compatible with the XMOS XS1 ABI

IN THIS DOCUMENT

The XMOS XS1 Application Binary Interface (ABI) defines the linking interface for objects compiled from C/C++, XC and assembly code. This tutorial explains how to write functions in assembly code that can be linked against objects generated by the XMOS compiler.

1 Symbols

As the assembler parses an assembly file, it maintains a current address which it increments every time it allocates storage.

Symbols are used to associate names to addresses. Symbols may be referenced in directives and instructions, and the linker patches the corresponding address onces its value is calculated.

The program below defines a symbol with name `f` that refers to the value of the current address. It also makes the symbol globally visible from other files, which can reference the symbol by its name.

```
# Give the symbol f the value of the current address.
f:
# Mark the symbol f as global.
  .globl f
```

The symbol is defined by writing its name followed by a colon. The `.globl` directive makes the symbol visible from outside of the file.

2 Alignment

The XS1 ABI specifies minimum alignment requirements for code and data. The start of a function must be 2-byte aligned, and data must be word-aligned. An address is aligned by placing the `.align` directive before the definition of a symbol.

Publication Date: 2013/11/11 REV B

The program below defines a symbol `f` that is defined to be the next 2-byte aligned address.

```
# Force 2 byte alignment of the next address.
  .align 2
f:
```

3 Sections

Each object file may contain multiple sections. When combined by the linker, sections with the same name in each object file are placed together at consecutive addresses. This allows different types of code or data to be grouped together in the final executable.

The XS1 ABI requires functions to be placed in the `.text` section, read-only data in the `.cp.rodata` section and writable data in the `.dp.data` section. The default section is the `.text` section, and the current section can be changed using one of the following directives.

Figure 1: Sections supported by the XMOS linker

Section	Used For	Directive
`.text`	Executable code	`.text`
`.dp.data`	Writable data	`.section .dp.data, "awd", @progbits`
`.cp.data`	Read only data	`.section .cp.rodata, "ac", @progbits`

3.1 Data

The example program below defines a 4-byte writeable object, initialized with the value 5, and aligned on a 4-byte boundary.

```
  .section .dp.data, "awd", @progbits
  .align 4
x:
  .word 5
```

You can use the following directives to emit different types of data.

Figure 2: Directives for emitting different types of data

Directive	Description
`.byte`	Emits a 1 byte value
`.short`	Emits a 2 byte value
`.word`	Emits a 4 byte value
`.space`	Emits an *n*-byte array of zero-initialized storage, where *n* is the argument to the directive
`.asciiz`	Emits a null terminated ASCII string
`.ascii`	Emits an ASCII string (no implicit terminating character)

3.2 Arrays

The program below defines a global array that is 42 bytes in size.

```
  .section .dp.data, "awd", @progbits
  .globl a
  .align 4
a:
  .space 42
  .globl a.globound
  .set a.globound, 42
```

The XS1 ABI requires that for each global array `f` there is a corresponding global symbol `f.globound` which is initialized with the number of elements of the first dimension of the array. You can use the `.set` directive to perform the initialization. Note that this value is used for array bounds checking if the variable is used by an XC function.

4 Functions

The XS1 ABI specifies rules for passing parameters and return values between functions. It also defines symbols for specifying the amount of hardware resources required by the function.

4.1 Parameters and return values

Scalar values of up to 32 bits are passed as 32 bit values. The first four parameters are passed in registers *r0*, *r1*, *r2* and *r3*, and any additional parameters are passed on the stack. Similarly, the first four return values are returned in the registers *r0*, *r1*, *r2* and *r3*, and any additional values are returned on the stack.

In the XC function prototype below, the parameters a and b are passed in registers *r0* and *r1*, as are the return values.

```
{int, int} swap(int a, int b);
```

An assembly implementation of this function is shown below.

```
  .globl swap
  .align 2
swap:
  mov r2, r0
  mov r0, r1
  mov r1, r2
  retsp 0
```

4.2 Caller and callee save registers

The XS1 ABI specifices that the registers *r0*, *r1*, *r2*, *r3* and *r11* are *caller-save*, and all other registers are *callee-save*.

Before a function is called, the contents of all caller-save registers whose values are required after the call must be saved. Upon returning from a function, the contents of all callee-save registers must be the same as on entry to the function.

The following example shows the prologue and epilogue for a function that uses the callee-save registers *r4*, *r5* and *r6*. The prologue copies the register values to the stack, and the epilogue restores the values from the stack back to the registers.

```
# Prologue
  entsp 4
  stw r4, sp[1]
  stw r5, sp[2]
  stw r6, sp[3]

# Main body of function goes here
# ...

# Epilogue
  ldw r4, sp[1]
  ldw r5, sp[2]
  ldw r6, sp[3]
  retsp 4
```

4.3 Resource usage

The linker attempts to calculate the amount of resources required by each function, including its memory requirements, and the number of threads, channel ends and timers it uses. This allows the linker to check that the resource usage of the final executable does not exceed that available on the target device.

For a function `f`, the resource usage symbols defined by the XS1 ABI are as follows.

Figure 3: Resource usage symbols defined by the XS1 ABI

Symbol	Description
`f.nstackwords`	Stack size (in words)
`f.maxthreads`	Maximum number of threads allocated, including the current thread
`f.maxchanends`	Maximum number of channel ends allocated
`f.maxtimers`	Maximum number of timer allocated

You can define resource usage symbols using the `.linkset` directive. If a function is global, you should also make the resource usage symbols global.

The example program below defines resource usage symbols for a function `f` that uses 4 words of stack, 2 threads, 0 timers and 2 channel ends.

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 5
.globl f.maxthreads
.linkset f.maxthreads, 2
.globl f.maxtimers
.linkset f.maxtimers, 0
.globl f.maxchanends
.linkset f.maxchanends, 2
```

In more complex cases, you can use the maximum (`$M`) and addition (+) operators to build expressions for the resource usage that are evaluated by the linker. If two functions are called in sequence, you should compute the maximum for the two functions, and if called in parallel you should compute the sum for the two functions.

The example program below defines resource usage symbols for a function `f` that extends the stack by 10 words, allocates two timers and calls functions `g` and `h` in sequence before freeing the timer and returning.

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 10 + (g.nstackwords $M h.nstackwords)
.globl f.maxthreads
.linkset f.maxthreads, 1 + ((g.maxthreads-1) $M (h.maxthreads-1))
.globl f.maxtimers
.linkset f.maxtimers, 2 + (g.maxtimers $M h.maxtimers)
.globl f.maxchanends
.linkset f.maxchanends, 0 + (g. maxchanends $M h.maxchanends)
```

You can omit the definition of a resource usage symbol if its value is unknown, for example if the function makes an indirect call through a function pointer. If the value of the symbol is required to satisfy a relocation in the program, however, the program will fail to link.

4.4 Side effects

The XC language requires that functions used as boolean guards in `select` statements have no side effects. It also specifies that functions called from within a transaction statement do not declare channels. By default, a function `f` is assumed to be side-effecting and to declare channels unless you explicitly set the following symbols to zero.

Figure 4: Symbols for denoting side-effects

Symbol	Description
`f.locnoside`	Specifies whether the function is side effecting
`f.locnochandec`	Specifies whether the function allocates a channel end

5 Elimination blocks

The linker can eliminate unused code and data. Code and data must be placed in elimination blocks for it to be a candidate for elimination. At final link time, if all of the symbols inside an elimination block are unreferenced, the block is removed from the final image.

The example program below declares a symbol within an elimination block.

```
  .cc_top f.function, f
f:
  .cc_bottom f.function
```

The first argument to the `.cc_top` directive and the `.cc_bottom` directive is the name of the elimination block. The `.cc_top` directive takes an additional argument, which is a symbol on which the elimination of the block is predicated on. If the symbol is referenced, the block is not eliminated.

Each elimination block must be given a name which is unique within the assembly file.

6 Typestrings

A typestring is a string used to describe the type of a function or variable. The encoding of type information into a typestring is specified by the XS1 ABI. The following directives are used to associate a typestring with a symbol.

Figure 5: Typestring directives

Binding	Directive
Global	`.globl name, "typestring"`
External	`.extern name, "typestring"`
Local	`.locl name, "typestring"`

When a symbol from one object file is matched with a symbol with the same name in another object, the linker checks whether the typestrings are compatible. If the typestrings are compatible linking continues as normal. If the typestrings are function types which differ only in the presence of array bound parameters the linker generates a thunk and replaces uses of the symbol with this thunk to account for the difference in arguments. The linker errors on all other typestring mismatches. This ensures that programs that are compiled from multiple files are as robust as those compiled from a single file.

If you fail to emit a typestring for a symbol, comparisons against this symbol are assumed to be compatible. If you are implementing a function which takes an array of unknown size, you should emit a typestring to allow it to be called from both C and XC. In other cases, typestrings can be omitted, but error checking is not performed.

7 Example

The program below prints the words “Hello world” to standard output.

```
const char str[] = "Hello world";

int main() {
  printf(str);
  return 0;
}
```

The assembly implementation below complies with the XS1 ABI.

```
  .extern printf, "f{si}(p(c:uc),va)"
  .section .cp.rodata, "ac", @progbits
  .globl str, "a(12:c:uc)"
  .cc_top str.data, str
  .align 4
str :
  .asciiz "Hello world"
.cc_bottom str.data
  .globl str.globound
  .set str.globound, 12

  .text
  .globl main, "f{si}(0)"
.cc_top main.function, main
  .align 2
main:
  entsp 1
  ldaw r11, cp[str]
  mov r0, r11
  bl printf
  ldc r0, 0
  retsp 0
.cc_bottom main.function
  .globl main.nstackwords
  .linkset main .nstackwords, 1 + printf.nstackwords
  .globl main.maxthreads
  .linkset main.maxthreads, printf.maxthreads
  .globl main.maxtimers
  .linkset main.maxtimers, 0 + printf.maxtimers
  .globl main.maxchanends
  .linkset main.maxchanends, 0 + printf.maxchanends
  .linkset main.locnochandec, 1
  .linkset main.locnoside, 1
```

By defining symbols for resource usage, the linker can check whether the program fits on a target device. By providing typestrings, the linker can check type compatibilty when different object files are linked. The linker can eliminate unused code and data since it is placed in elimination blocks.