APPLICATION NOTE

AN702
High level language support in XA

Author: Santanu Roy, Philips Semiconductors,
MCO Applications Group, Sunnyvale, California

1995 Jul 28

Philips Semiconductors

PHILIPS

High level language support in XA AN702

Author: Santanu Roy, Philips Semiconductors, MCO Applications Group, Sunnyvale, California

Introduction

High Level Language (HLL) support is becoming a key feature in modern day microcontroller architecture. The reason is highly visible. It is easier to code a processor in a high-level platform than in conventional assembly because it is portable, i.e., it is not tied to any one machine. Also, the advantage of coding in a high-level language is because it is modular and re-usable which speeds up any code development process considerably.

In recent years, C has been "the language" of choice for all engineers. Thus almost all modern day microcontrollers are designed with C-language support in mind. This article highlights some of the architectural features of Philips XA microcontroller that has been designed to support such languages specifically C.

Supporting HLL

One of the tasks that an architect has to confront is the determination of exactly what instructions should form the functional instruction of a microcontroller to meet high-level language support. An answer to this is to provide an operation code for each functional operation in a high-level programming language. Thus operation codes will exist for +, –, *, /, and so on. Special provision is made for operation on arrays, and all operations that can be applied to data types in a high-level language are directly supported in the architecture. An instruction set ideally should contain only instructions that are used in a HLL, and not implement any non-functional instructions, i.e., instruction that is not expressed as a verb or operator in a high-level language. Thus "LOAD", "STORE", and so on which are not statements made in high-level languages are redundant and only adds to architectural overheads.

An instruction word consists of a single op-code and an operand address for each HLL variable involved in the operation. *Op-codes are symmetric in that they are applicable to any type of addressing and any data type.*

Some general criteria for an ideal architecture could be:

1. Only one instruction should be executed for most common HLL operators.
2. There should be only one memory reference for each referenced operand.
3. There should be explicit addressing only for operands whose location cannot be inferred by recent processing activity, and address should be short.
4. Instructions should be compact, and densely coded.

The XA Architecture

The XA is a register based machine. Hence most variables could be stored in these fast storage registers for high code density and fast execution. However, the beauty of the XA architecture is that, it is optimized for internal memory as well for high throughput and code density, e.g., a register-register ALU operation takes 2 bytes and 3 clocks and the same ALU operation between register-memory (indirectly addressed) is 2 bytes and 4 clocks. So, a large set of variables could be stored in memory with very little loss in performance. Additionally, hooks like "burst mode", etc., are provided to speed up external memory access as well.

Data Types and Sizes

XA directly supports the following basic data types as used in C:

character (char) – signed and unsigned bytes
integer (int) – signed and unsigned words

Constants – Supported as byte/word (char/int) immediate data in the instructions, e.g., ADD R0, #1234 etc. The range is +32,767 to –32,768 for signed and 0 to 65,536 for unsigned word/integer constants, +127 to –128 for signed or 0 to 255 for unsigned bytes/char.

For "short" qualifier, the range is +7 to –8 as used with instructions MOVS and ADDS.

A "long" qualifier to integer is implemented by the compiler by extending (signed/unsigned) the word to the next higher address(+1). In addition to the above,

Bit – This special data type is also supported to access the different bit addressable space in the machine.

Note: All signed data are represented in 2's complement form in the XA.

Type conversion

All operations are performed under natural data sizes, e.g., MULU.b does a 8x8 unsigned multiply of 2 bytes, MULU.w does the same but with 2 word-size operands. So when operands of different types appear in an expression, they are converted to a common type by the compiler, e.g., operation between a *char* (byte) and an *integer* (word) is promoted to *integer-integer*, etc.

Arrays

XA supports addressing byte and word arrays in memory as required by C or any HLL. Offset and auto-increment addressing modes in XA allow easy access and manipulation of array elements. Offsets are signed values of 8 or 16 bits and are used depending on the size of the array.

Static Variables

Static variables unlike automatic provide permanent storage in a function. This means these variables are stored in memory rather than being a part of run-time stack. A wide variety of memory addressing modes are supported in the XA to provide easy access to static variables in memory. In addition to several indirect addressing modes (auto-increment offset) the XA supports direct access to the first 1K of the memory space in each segment. This is ideal for addressing static variables, and has found to generate extremely dense code. A listing of operations to access static variables is given below for reference:

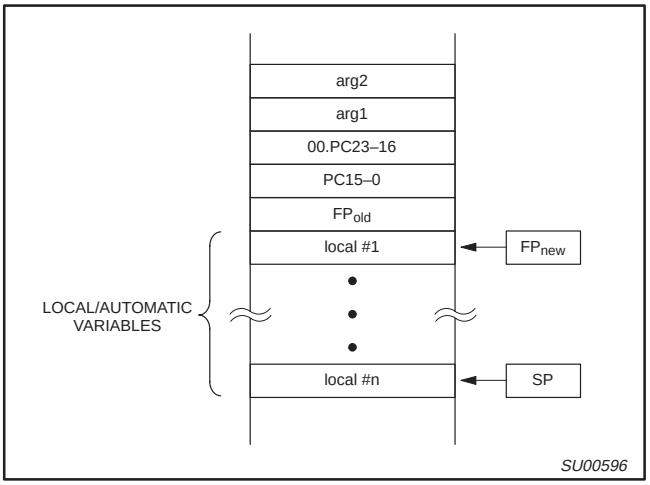
Table 1. Access to Static Variables

ADDRESSING MODES
Rd, direct
direct, Rd
Rd, [Rs+]
[Rs+], Rd
Rd, [Rs+]
Rd, [Rs+Offset8/16]
[Rs+Offset8/16], Rd
direct, direct
[Rd], #immediate
direct, #immediate
[Rd+], #immediate
[Rd+], [Rs+]

Automatic/Dynamic Variables

Within a function, a typical compiler maintains a Frame Pointer (FP), which is used to access function arguments and local automatic variables. To call a function, a compiler pushes arguments onto the stack in reverse order, (the PUSH instruction decrements the SP by 2 each time it is executed, calls the function, then increments the SP by the number of bytes pushed. For instance, to call a function with two one-word arguments, the XA-compiler generates code to do the following:

```
PUSH arg2          ; (SP –=2)
PUSH arg1          ; (SP –=2)
CALL (subroutine)  ;
ADDS SP,4          ; (SP +=4)
```



The CALL instruction pushes the current PC onto the stack. Because all stack pushes are 16-bits in XA, any 8-bit function argument is automatically promoted to word.

Upon function entry, the compiler creates new stack and frame pointers by computing:

```
PUSH FP (old)
FP (new) = SP
SP = SP – Framesize;
```

where “Framesize” is the space required for all local automatic variables. If the frame size is odd, the compiler always rounds it up to the next even number. If there are 2 arguments and 2 local variables, then the frame size is 4 and the stack looks like this:

```
FP+8 second argument
FP+6 first argument
FP+4 return address
FP+2 old FP
FP–0 first local variables
FP–2 second local variable
FP–4 next free stack location (same as SP)
```

If a function argument is defined to be an 8-bit type, then only the lower 8-bits of the value pushed by the caller are to inside the called function.

Upon function exit, the compiler restores the SP and FP to their original value by executing the following:

```
SP = FP
POP FP
RET
```

The return instruction RET sets the new PC by popping the saved PC off the stack.

Because there are so many registers in XA (unlike 8051), any of them could be assigned to hold the FP. Access to variables in the stack space is easily achieved through the indirect-offset addressing modes (signed 8 or 16) with respect to the stack pointer. In almost all the cases the variables pushed onto the stack could be accessed using only a signed 8-bit offset present in XA. The function arguments and variables could be moved in and out of the stack in a single PUSH/POP multiple instructions permitted in XA. In fact up to 8 words or 16 bytes of such information could be moved in and out of the XA stack with one instruction, which increases code density to a large extent during procedure calls and context switching. For example, if register variables are in R1,R2,R3, and R4, a single “PUSH R1,R2,R3,R4” instruction will be generated by the XA-compiler. A corresponding function exit will have a “POP R1,R2,R3,R4” for restoring the variables.

All automatic class of variables will be allocated on run-time stack. The XA has full complement of addressing modes on SP to handle dynamic variables in the stack. Table 2 shows some of the XA addressing modes that could be used for such access.

Table 2.

ANSI-C	XA	Comments
SP–>Offset	R+Offset8/16	
*SP	[R]	
SP+	[R+]	Pop

Operators for HLL support

The structure for op-codes of an ideal architecture should be stated in terms of number of operands required and the relationship between the operands. The structure should be oriented toward efficient coding of an instruction that will support programs written in a HLL with minimum compilation. The XA instruction set is designed to handle such efficiency as reflected in Table 3. The set of instructions that supports the general/basic addressing modes are used to describe HLL support in this table.

Table 3. Mapping of XA ALU Operations to C Operators

ANSI C Operator (op)	XA Op-codes
+= , += + C()	ADD, ADDC
–= , –= – C()	SUB, SUBB
< , <= , == , >= , > , != (s/u)	CMP
&= , |= , ^=	AND , OR , XOR

Data movement in C is given by “=” which is the “MOV” instruction in XA. The MOV instruction not only has the general/basic addressing modes, it also has some additional addressing modes for C-code optimization for memory transfer operations like direct-direct, direct-indirect, indirect-autoincrement – indirect-autoincrement.

Table 4 of two operand case **A = A op B** or **B = A op B** is shown below.

Table 4.

ANSI C	XA
C-operations	**Equivalent XA-operations**
R op = R	R, R
R op= *R	R, [R]
R op= *R++	R, [R+]
R op= direct	R, direct
R op= R–>offset	R, [R+offset]
*R op= R	[R], R
*R++ op= R	[R++], R
direct op= R	direct, R
R–>offset op= R	[R+offset], R
R op= constant	R, #constant
*R op= constant	[R], #constant
*R++ op= constant	[R++], #constant
direct op= constant	direct, #constant
R–>offset op= constant	[R+offset], #constant

The three operand cases A = B op C may regularly be translated as:

```
A = B;
A op= C;
```

exception to above is

```
*R++ = B op C is equivalent to
*R = B;
*R++ op = C;
```

Typical/Frequently used C-code **A = B op C** involves operations that will fetch operands from memory, register, and as immediate data which is embedded in the instruction. The XA has the following choices for operand placements for such three operand operations.

Case 1:
If A = register,

then B and C in A = B and A op= C could have the following choices

(i) Register i.e., R = R and R op= R

(ii) Memory i.e., R = Memory and R op= Memory

where Memory = [R] , direct, [R+], [R+Offset]

(iii) Immediate i.e., R = Immediate and R op= Immediate

Case 2:
If A = Memory

where Memory = [R], direct, [R+], [R+Offset]

then B and C in A = B and A op= C could have the following choices:

(i) Register i.e., Memory = Register and Memory op= Register

(ii) Immediate i.e., Memory = Immediate and Memory op= Immediate.

(iii) Memory i.e., Memory = Memory ([R+], and direct modes only) *for B*

The above indicates that virtually all C operations involving two and three operands could be very efficiently translated in XA assembly code (in two operand cases, it is one-to-one) using a cross-compiler.

NULL DETECT/STRING TERMINATOR

Checking for “0” at the end of a string is natural in XA with the MOV instruction. The Z flag is set whenever such a condition occurs. This is especially important in string copy operations where the loop ends whenever a end of string or ‘\0’ occurs which is reflected in the status flag “Z” in XA. The following lists such C-code and equivalent XA instructions.

```
while ((c=getch()) != '\0')      Label:   MOV [R+], memory
buffer[i++] = c;                          BNE      Label
```

Coding Relational Operations

Performing relational evaluation between two operands A and B in C-language involves fetching operands (a) in memory (b) in register or (c) an immediate value, evaluating the condition and then taking appropriate actions which typically involves a branch-if-true or branch-if-false operations

The operand(s) in memory again could be addressed as direct, indirect, indirect-autoincrement, indirect-offset, etc. The XA provides one-to-one translations of such operations.

Typically such C-statements are as follows:

```
if (A cmp_op B)                  CMP A, B
{ body} /* true */               Bxx LABEL; branch if false
                                 body
                                 LABEL:

if (A cmp_op B)                  CMP A, B
{ body 1 }                       Bxx L1 ; branch if false
else                             body 1
{ body 2 }                       JMP L2
                                 L1: body 2
                                 L2:

while (A cmp_op B)               L1: CMP A, B
{ body }                         Bxx L2 ; branch if false
                                 body
                                 JMP L1
                                 L2:
```

Coding Bitwise Operations

C provides 6 operators for bit manipulation. These are & (Logical AND), | (Logical OR), ^ (Logical-XOR), << (Logical Shift-Left), >> (Logical Shift-right), and ~ (one's complement). There is one-to-one equivalence in XA for such operation class:

(a) & – AND,
(b) | – OR, ^ – XOR,
(c) << – ASL,
(d) >> – LSR, and
(e) ~ – CPL.

Compiler Optimization

Some special cases of Multiply and Divide where the multiplier and divisor could be assumed to a power of 2, following translation could be expected from the compiler during optimization which speeds up code execution and make code denser.

Language extensions to XA could be written as the pre-processor macros of the XA C-compiler as shown in Table 5.

Table 5.

C-code	XA code
R *= R	R <<= R
R *= Constant	R <<= Constant
R /= R	R >>= R
R /= Constant	R >>= Constant

ROLC(R,R) – for rotate left through carry, ROL (R,R) and ROL (R, constant) – for rotate lefts, etc. Same holds for ADDC and SUBB also.

Reentrancy

In a multi-tasking or nested interrupt environment, some system or library subroutines may be activated dynamically. These subroutines require duplication of the variable area of the subroutine per each active copy, utilizing essentially *dynamic memory allocation.*

The allocation of the dynamic area is done by a system service call. The dynamic area is allocated either out of the reserved system memory, when large memory exists in the system, or on the stack, when memory is very limited. In the latter case, the stack pointer is adjusted, to reflect the extra bytes reserved. It will be readjusted just prior to returning from the subroutine.

The subroutine code accesses variables using [R+offset] addressing mode. The register is referred to as a Static Base Register or Frame Pointer.

Since the application stack is separate from the interrupt stack, there's no problem with interrupting the dynamic allocation/de-allocation and application stack pointer adjustments.

Floating Point Support

Although the XA does not have a floating point unit, it has special instructions to provide an extensive support for floating point operations. Instructions like NORM (normalize), SEXT (sign extend), ASL, ASR (Arithmetic shifts) and status flag like "N" (sign), all aid in floating point support. Floating point library routines implementing (IEEE or ANSI) floating point provided with compilers could extensively use such instructions for increased code density and throughput in XA.

Dynamic Code Link/Relocability

The XA allows for dynamic code linking through extensive use of FCALL (Far Call 24-bit addressing). This makes code developed for XA highly portable/relocatable in memory.

Simple relocatable code however could use CALL rel16 and CALL [R] addressing modes which is limited to 64K address.

System Interface

When used for RTOS, system mode with its protected features could be extensively used for system management routines/operating System service e.g., *printf etc* and application task switching. This could be easily done in XA through a TRAP # instruction set up by the compiler requesting system service by the application task. In the event of task switching, a system service call sets up the environment for the new task via the resource access privileges of the task, application stack etc.

Author's Acknowledgement

The author recognizes the following Philips Semiconductor XA team members for their review and inputs on this article:

Ata Khan, Ori Mizrahi-Shalom, and Frank Lee

References:

XA User Guide – Philips Semiconductors

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381