APPLICATION NOTE

AN708
Translating 8051 assembly code to XA

1996 Dec 30

Philips
Semiconductors

PHILIPS

Translating 8051 assembly code to XA AN708

CONTENTS

1. Introduction 3
1.1 The questions you are probably asking right now . . 3
1.2 Overview of the translation process 4
1.3 What you'll need 4
1.4 Preparation 5
1.5 About the original application 5
1.6 Translation decisions you must make 5

2. The translation process 7
2.1 Starting translation:
Producing tentative XA source 7
2.1.1 Systematic changes 7
2.1.2 Translating 7
2.1.3 How the translator works 8
2.1.4 Interpreting translator results 8
2.1.5 Systematic changes revisited 9
2.1.6 What you've got: "tentative XA source" . . . 9
2.2 Continuing translation:
Producing structurally correct XA code 10
2.2.1 XA startup vector (simple case) 11
2.2.2 XA startup and interrupt vectors (more complex case) 11
2.2.3 The XA stack 13
Basics 13
The System Stack 14
The System Stack and Interrupts 14
The User Stack 14
Advanced Stack Issues 14
2.2.4 Feeding the Watchdog 15
2.2.5 Obligatory Hardware Initialization 15
2.2.6 Disposing of Warning Messages 16
Details of Compatibility Instructions 16
2.3 The final step:
Generating debugged XA code 17

3. Special Topics 18
3.1 Trouble-making items 18
3.2 Translating indirect references 18
3.2.1 Indirect reference failure 18
3.2.2 Recommendations for Indirect References 18
3.2.3 Picking a register for indirect references . . 18
3.3 Translating "degenerate" 8051 source code 19
3.3.1 "Here" relative addressing 19
3.3.2 Branch to "Here" 19
3.3.3 Branch to "Here+offset" or "here-offset" . . 19
3.3.4 In-line strings 20
3.3.5 General In-line args 20
3.3.6 Program Resets 21
3.3.7 Compare trees 21
3.3.8 Code Table Fetches 21
3.3.9 Using the stack for vectored execution . . . 21
3.4 Translating "untranslatable" 8051 source 22
3.4.1 PSW bit addressing 22
3.4.2 P2 addressing 22
3.4.3 R7 use 22
3.5 Wrap-up 22
3.6 Trouble checklist 22
3.7 Final Comments 22

Appendix 1: Startup+Interrupt Prototypes 23

Appendix 2: Compare/Branch Summary 27

Example 1 28
8051 CJNE 28
XA CJNE 28
XA CMP/Bxx 28

Example 2 29
8051 CJNE 29
XA CJNE 29
XA CMP/Bxx 29

Example 3 30
8051 CJNE 30
XA CJNE 30
XA CMP/Bxx 30

Example 4 31
8051 31
XA CJNE 31
XA CMP/Bxx 31

Translating 8051 assembly code to XA AN708

1. INTRODUCTION

What if...

- You've been given the task of translating your company's flagship 80C51 application to XA; your manager gives you an XA data manual and some diskettes—and until tomorrow to get it done?
- You're going to be translating lots and lots of 8051 code to XA over the next few months, and you want to set up the most effective system for doing so?
- You want to learn the XA architecture by translating an example 8051 application?

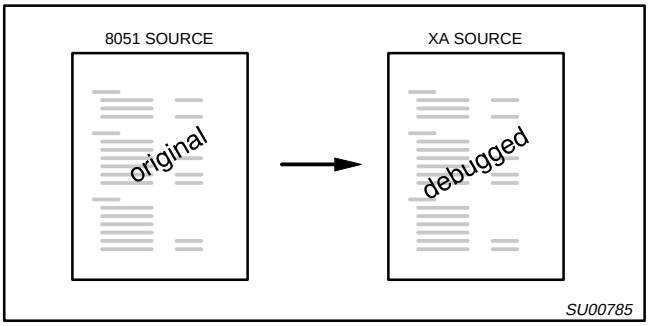

Figure 1.

Relax! Translating production 80C51 source code to debugged XA code is very practical—although we cannot guarantee you'll make your deadline—and this application note brings together the information you'll need.

We're going to generally assume that you're dealing with 80C51 source code you've never seen before.

We will also assume you're a reasonably skilled 80C51 programmer and that you've already studied the basics of XA architecture. We're also going to assume you're able to use Windows™ in general, and you've familiarized yourself with the XA development tools. Although using the simulator or emulator is a key part of verifying your translations, teaching you to use these is beyond the scope of this note.

Source code for the examples may be found on the Philips ftp site.

1.1 The questions you are probably asking right now

How is this application note organized?
We're going to describe a stepwise procedure for translating 80C51 code to working XA code, concentrating mostly on hardware-independent issues. Later on, we'll deal with some very specific issues in detail (section 3: Special Topics), so we recommend you read through all this material before starting to do any translation work.

How long will it take to translate an 8051 application?
That depends on the application and the code. We've seen code from experienced 8051 programmers who've used every conceivable 8051 coding trick to squeeze out the last drop of functionality. We expect this kind of 8051 code will be the most difficult, especially when the code uses target hardware-specific tricks. On the other hand, some 8051 code, especially small applications designed to be easily maintainable, aren't particularly difficult and you may be able to do a complete translation in a matter of a few hours.

Is translation automatic?
Not completely. While the XA does have an equivalent for every 8051 instruction, some code cannot be automatically translated. You'll have to intervene manually.

Is translation a single-pass process?
If you are really expert and you spend enough time, you can probably translate simple to moderately complex applications in a single pass. We don't generally recommend this approach, however, especially when you are starting out. It's probably more productive to iterate several times, using all the development tools available to produce the most robust translation.

Will XA code be bigger?
In all likelyhood, it will be somewhat to significantly larger than the original 8051 code. However, you'll be well-compensated by a significant increase in generality and functionality –the XA instructions are bigger, but they do more– so your XA code will be much easier to maintain and expand.

Will XA code be faster?
Based on the same clock rate, XA code will execute significantly faster than the 80C51 code.

What about 8051 and XA derivatives?
This application note is about translating "general" 8051 code to a "generalized" XA. We'll look briefly at the essential peripheral devices, like the UART and timers, but we're not going to look at other subsystem-specific programs.

What about memory and I/O expansion?
We won't deal with specific memory and I/O issues, but we will comment that the XA is flexible enough to deal with almost any kind of memory and I/O interfaces. The only exception to this rule is some very 8051-specific external interfacing which you'll very likely need to re-engineer anyway.

What's the standard for 8051 code syntax?
The code translater expects the MetaLink ASM51 assembler syntax. This assembler is available for free on the Philips BBS, and it has become the de-facto standard for 8051 and derivatives. If your 8051 source code is based on a different standard, we'll have some suggestions.

What's special about translating your own 8051 code?
As you'll see below, we generally recommend that you deal with mechanical translation issues first and worry about structural changes later. If you're translating 8051 code that's very familiar to you, it is very easy to get distracted by making structural changes too early: you know the constraints of the original design and you'll likely be eager to overcome them by using the significantly greater freedom of the XA architecture.

What is the biggest difference between 8051 and XA afffecting the translation process?
Most 8051 programmers building complex applications spend a significant amount of time reconciling application requirements to 8051 architectural capabilities. When you translate this code to the XA, you'll find many familiar architectural concepts but far fewer constraints on them.

.Windows is a tradmard of Microsoft, Inc.

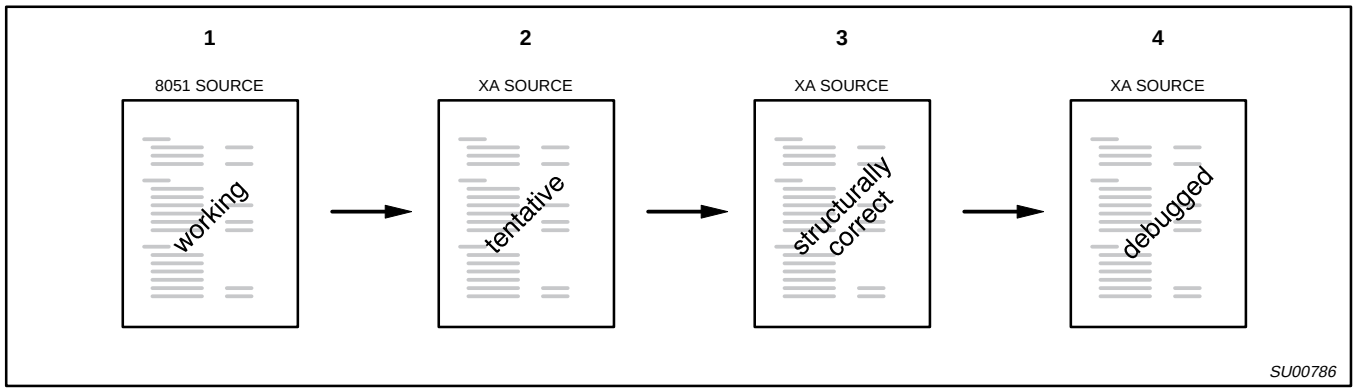

Figure 2. Translation Process

1.2 Overview of the translation process

Let's take a more detailed look at the overall translation process (Figure 2). There are four distinct source versions:

1. The working 8051 source (a copy, not your original source, which you must preserve).
2. A tentative translation of the original 8051 source into XA assembly.
3. The structurally correct XA source.
4. The debugged XA source.

The method of producing each of these source versions is similar to the standard program development cycle, with a single additional step, as shown in Figure 3:

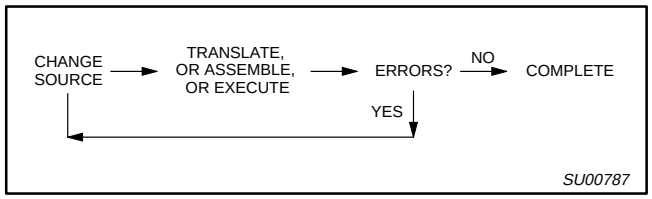

Figure 3.

We recommend that you first make a copy of your original 8051 source. We'll call this the "working" 8051 source. In some cases we've found it convenient to make changes to this source, in particular, when we've found ourselves re-running the translator multiple times for particularly difficult source programs. Be sure to protect your original 8051 source carefully; this is so important we'll remind you to do so several times.

You can see that the edit-assemble cycle and the edit-assemble-execute is preceded by an edit-translate cycle. As we will show below, this extra step is easy. After you complete your work with the translator, you'll have tentative XA source. You'll edit and assemble the tentative XA source, you'll have structurally correct XA source code. You'll run the simulator or emulator and edit and reassemble until your application is proven.

In a few cases, especially with complex applications, you may find it useful to return to an earlier source version. For example, you may encounter some translation issues while debugging your XA source that are best handled by returning to the working 8051 source and repeating the intervening steps. Because of the number of source versions in this process and the potential for returning to an earlier step, we've found that careful organization of the translation process can be very helpful.

1.3 What you'll need

Here's a checklist of what you need to get started translating your 8051 application to XA:

- ☐ The XA Data Handbook (IC25 or its successor); especially Section 2, Chapter 9 and AN704.
- ☐ An 8051 reference, preferably one that's familiar to you.
- ☐ A Windows-based computer
- ☐ The Macraigor XA Development Environment, which includes a translator, an assembler, a simulator, and an optional single-chip XA emulator.
- ☐ The MetaLink ASM51 8051 assembler and its manual, both available on the Philips BBS
- ☐ Macraigor Development hardware is helpful, but optional
- ☐ Your favorite word processor and text utilities.
- ☐ Any and all documentation about the original application. An as-implemented memory map is invaluable. (If you don't have one, we recommend you immediately prepare one by examining the original source code).
- ☐ Design specs for your XA target, particularly the memory map.

1.4 Preparation

Read this application note thoroughly. Section 3, “Special Topics” describes some specific problems you may encounter, and you should have these in mind from the start. Then scan through your original 80C51 source to see how many potential translation issues you can identify.

We recommend that you prepare by choosing a file labeling system to distinguish among the original 8051 source, intermediate translations, and the final XA source file. We have found the following scheme to be useful:

app.asm the original 80C51source file (Not to be modified!)

tapp.asm the working 80C51 source, which may be modified

tapp.1 an intermediate translation

tapp.2 ...

tapp.xa the current XA assembler source

Note that the translator identifies 80C51 source files and activates the 8051 to XA Translator option in the LANGUAGES menu whenever you use a file with a “.asm” extension. The default extension for translated files is “.xa”. You may want to keep copies of intermediate translations or originals.

Choose a method that is comfortable for you and that’s consistent with the size of the job. This seems like a trivial matter, but we’ve found that a little extra bookkeeping care can make the translating job much easier.

Second, we recommend that you make sure that all your tools are installed and are functional before attempting translation:

1. Assemble your original source file with the Metalink assembler; note its assembled size.
2. Assemble and simulate one of the examples that accompany the XA tools.
3. Translate one of the example 8051 files.

In other words, we suggest you are comfortable with, and sure of, your tools before you start actual translation work.

1.5 About the original application

Note what we haven’t suggested: a detailed examination of the original application.

We’re not sure this is necessary!

Intense study of someone else’s code can be so incredibly boring or discouraging that you might balk at going through with the translation. You’ll have to determine in each case exactly how much detailed knowledge of the application is actually necessary to do a translation. As we’ve already mentioned, it almost all cases you should start with a memory map of the original application, but it isn’t always necessary to have everything documented.

For example, we’ve been successful translating a moderately complex application, TinyBASIC, without really attempting to understand how the code works, rather, by just attending to the mechanical translation details.

Ultimately, of course, it is up to you how much you need to know about the original application before doing the translation. In the ideal case, you already have full and clear documentation of your application’s algorithm (what it does) and the implementation (how it does it). In practice, unfortunately, you may have to re-develop this documentation or translate without it.

1.6 Translation decisions you must make

Although the translator does a good job, there will be some grey areas where you will have to decide about specific issues, and possibly make manual changes to the translated code. Here are the issues:

Retaining or Replacing 80C51-like instructions

The XA instruction set includes a number of instructions for compatibility with 80C51, even though the XA architecture provides improved alternatives.

Instructions like “JMP [A+DPTR]”, for example, are supported in the XA, but the full functionality may not translate directly. The translator leaves a warning to mark each use of one of these instructions. You’ll either have to check each case carefully and make adjustments to make sure the translated code will function correctly, or replace the entire construction with one or more native XA instructions.

In general, we recommend you replace these 80C51-style instructions with native XA instructions whenever it is practical. We’ll give you more information about this issue in following sections.

Recoding obscure but functional 80C51-style coding

There is an additional class of 80C51 instructions and constructions that translate directly into XA instructions. These instructions generate no warning messages from the translator because the resulting XA code, while often obscure, will function correctly. One example is a common 80C51 compare-tree construction.

We can’t make a general recommendation in this case, but we’ll admit our bias towards recoding into native XA code whenever possible. The following sections will give you more information on this issue.

Using Native versus Compatibility Mode on the XA

The XA System Configuration Register (SCR) CM bit controls the 8051 Compatibility Mode. At reset, SCR.CM is set to zero and the XA operates in “native” XA mode. Setting SCR.CM to “1” makes the XA register model and indirect register addressing mirrors the 80C51 model.

The effects of the CM bit setting are given in Table 1.

Table 1.

FEATURE	IN NATIVE MODE: CM=0	IN COMPATIBILITY MODE: CM=1
registers (R0, R1 ...)	Accessible only as registers	Accessible as registers or as the first 32 bytes of data memory.
R0, R1 indirect addressing	uses 16-bit pointers: R0, R1	Uses 8-bit pointers: R0L, R0H

In other words, if you do nothing, the XA will operate in native mode. The first 32 bytes of data memory will be accessible only as such and won't be overlaid by registers. You'll be able to use r0 and r1 (and any other register) as a 16-bit pointer.

If you insert an instruction that sets SCR.CM=1, translated code that depends on the overlaid memory and register mapping shown in Table 2 will continue to work, and both r0 and r1 will function as 8 bit indirect pointers. This maintains "pure" code compatibility for translation but effectively wastes 32 bytes of internal RAM data memory.

Table 2.

ORIGINAL 8051 REFERENCE	TRANSLATED XA REFERENCE	OVERLAID DATA MEMORY ADDRESS
R0 (RB0)	R0l (RB0)	0
R1 (RB0)	R0h (RB0)	1
R2 (RB0)	R1l (RB0)	2
R3 (RB0)	R1h (RB0)	3
...		
R7 (RB0)	R3h (RB0)	7
R0 (RB1)	R0l (RB1)	8
...		
R5 (RB3)	R2h (RB3)	29
R6 (RB3)	R3l (RB3)	30
R7 (RB3)	R3h (RB3)	31

* RB = Register Bank.

NOTE: The setting of SCR.CM has no effect on instructions labeled "...included for 80C51 compatibility", e.g., "JMP [A+DPTR]" These instructions function as described in the *XA User Guide* no matter what the setting of SCR.CM.

What are the pros and cons of compatability mode versus native mode?
By design, enabling compatability mode produces the greatest chance of translated code working with the least necessity of manual intervention.

In practice, the deciding factor for choosing is, in our experience, the degree of memory map rearrangement you do and the resulting effects on indirect addressing in the application. Specifically: if your XA memory map requires the use of 16-bit pointers, you'll need to use native mode.

Here's a list of factors to consider:

- Using compatability mode....
  - translated code is more likely to run correctly with minimal manual intervention
  - preserves register assignments
  - preserves direct addressing to registers commonly used in 8051 programs
- Using native mode...
  - is sometimes necessary due to changes in the memory map
  - supports cleaner, more efficient XA code
  - generally requires more manual changes to the translated code
  - frees up 32 bytes of on-chip RAM

Using 24-bit versus 16-bit ("Page 0") addressing
You can save some data and hardware resources if you choose 16-bit addressing; clearly this option is available only to applications with addressing requirements below 64K bytes.

The System Configuration Register (SCR) PZ bit controls the XA's Page 0 mode. At reset, SCR.PZ is set to "0" and the XA uses standard 24-bit XA addressing. If SCR.PZ is set to "1", the XA maintains only 16 bits of address data throughout.

For XA targets implementing a memory space of 64K or less, using Page 0 mode can result in some resource savings because the XA will only PUSH and POP 16-bit values for subroutine calls and returns, respectively, insread of 32-bits in standard operation. See the *XA User Guide* sections 4.3.1 and 4.3.2 for more details

Your choice of 24-bit versus 16-bit addressing has no direct effect on the translation process, but you should be generally aware of the implications of each alternative. We've chosen to set Page 0 mode in our start-up code example (Appendix 1).

2. THE TRANSLATION PROCESS

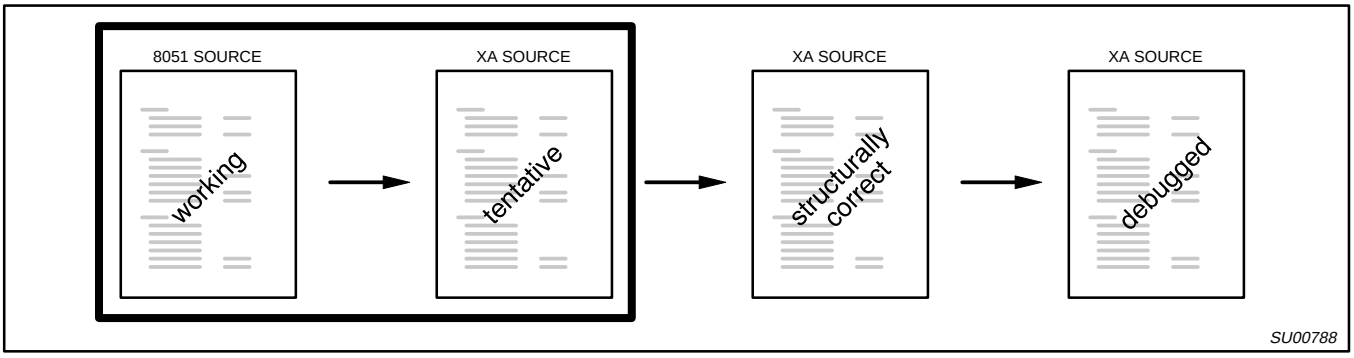

Figure 4.

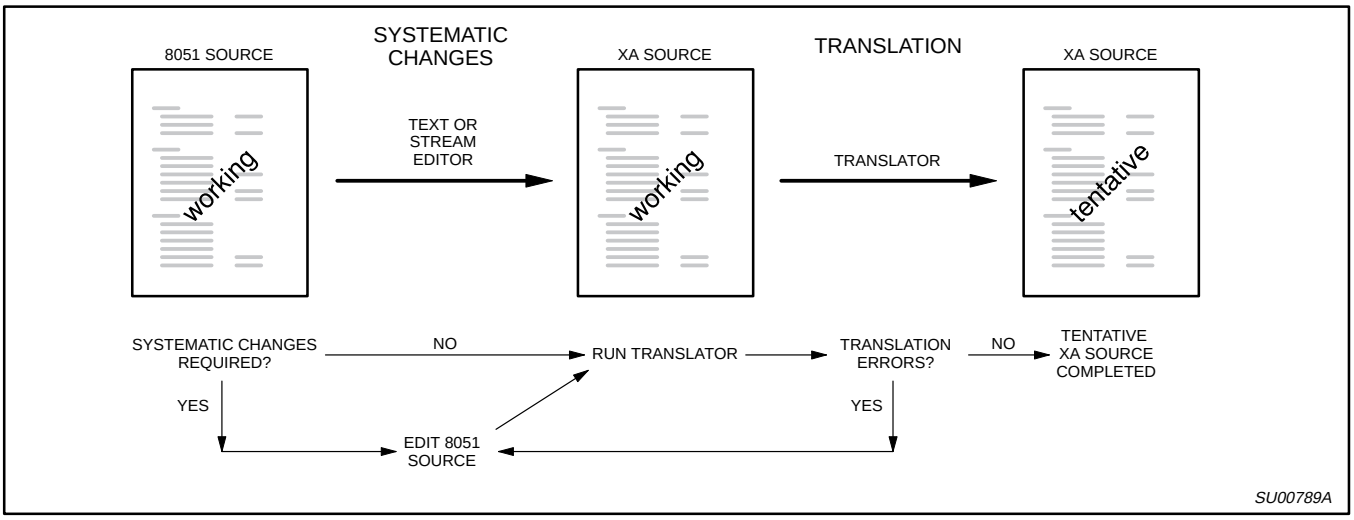

Figure 5.

2.1 Starting translation: Producing tentative XA source

Let's get started by looking at the process of turning 8051 source into tentative XA source code. Figure 5 tells the story. You'll edit, translate, and repeat if necessary until the translator gives no error messages. Don't worry about warnings placed in the translated source code just yet.

2.1.1 Systematic changes

We've chosen the term "systematic changes" to describe anything you might do to change the 8051 source overall.

Do you need to make systematic changes at this stage? The answer depends on the assembler you've been using, the complexity of the code, and the degree to which special directives and other assembler features are present in the source code.

At this point, we'll give you an easy answer that will serve for most purposes: "no". Just go ahead and translate. You'll find out in subsequent steps if this answer was right.

We might as well remind you right now: **Whatever you do, make sure you keep a protected copy of your original 8051 source code somewhere**. It is all too easy to make modifications to the original source file—as you will see, the working 8051 source file may change during this process—and you'll lose valuable information.

2.1.2 Translating

Translating is the soul of simplicity: the XA Development Environment recognizes any file with a ".asm" file as being possibly 8051 assembly source.

1. Open the source file (for example, "`try.asm`")
2. Select "Languages --> 8051 to XA Translator."
3. Respond "Yes" or "No" as you prefer to the query "Include 8051 Code in Output?"
4. The XA translator...
   a. translates your file, leaving it unchanged,
   b. makes a new XA source file with the same name and an "`.xa`" extension ("`try.xa`"), and
   c. automatically saves a copy of the XA source file.

If you choose to include 8051 code in the translator output you'll see the original 8051 instruction, commented out, adjacent to the corresponding XA instruction. This can be very helpful in some cases but the output file can be difficult to read.

We recommend standardizing on the ".xa" extension for all XA assembly source files to distinguish them clearly.

2.1.3 How the translator works

The translator looks at each input source line individually. The translator looks at each source line and attempts to identify it as either a) translatable 8051 code or b) anything else.

If the translator sees 80C51 code, it translates it according to the rules described in Chapter 9 of the *XA User Guide*. The XA was designed to have direct correspondences with 80C51 whenever possible, and most of the translations are probably what you'd expect. The register translations are so critical that we want to restate them here (Figure 6).

All other source lines—that is, anything other than translatable 8051 code—are passed through the translator and appear in the output file unchanged.

The translator doesn't attempt to "understand" what the code is doing; in specific, it doesn't look at more than one source file line or more than one actual instruction at a time. However, the translator warns you when some specific multi-line constructs might go wrong after translation.

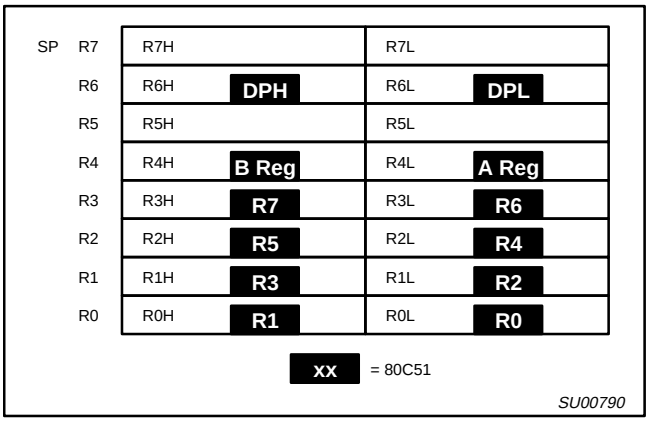

Figure 6.

2.1.4 Interpreting translator results

In most cases, the translator will run without error, and you'll see an open window with the translated source file. This file will contain inserted lines and warnings.

The translator always inserts a line that controls the pagewidth, in order to produce generally convenient displays within source and listing windows:

```
$pagewidth 132t
```

If necessary, the translator inserts an "include" reference, as follows,

```
$include XA-G3.EQU
```

so that any XA register or other reference it produced in translated code is correctly resolved. (If no such references are made, this "include" directive is omitted.) The default location of this file is the PROGRAM subdirectory of the XA Development Tools. You may want to use a different file—for example, for a different XA derivative—or use your own copy of this file.

The translator inserts warnings adjacent to source lines that are translated correctly, but which might not produce the desired results without further intervention. We will take a look at resolving these warnings later in this document.

The translator displays any serious errors in a standard dialog box.

What produces translator errors? In general, when the translator encounters a source line containing what looks like translatable 8051 code, it attempts to do the translation. If the translator finds something about the code that's not expected, it will flag an error for this line.

Fortunately, translator errors are very rare and occur only if the translator finds something entirely unexpected, like source code for an entirely different processor. You'll have to take a look at each error message and decide what to do about them on a case-by-case basis. A large number of errors probably means something very basic is wrong.

2.1.5 Systematic changes revisited

Let's revisit the issue of systematic changes.

You'll need to make systematic changes to your original 8051 source file whenever there are consistent differences between your source and the expectations of the XA Development Tools, especially the translator.

We have found that the easiest way to discover the need for systematic changes is to proceed as if they are not necessary.

If you see significant numbers of similar translator warning messages or XA assembler errors, consider changing your 8051 working source to avoid them.

NOTE: We are suggesting you change the 8051 source file to avoid problems downstream in the translation process. It is for this reason that we use the label "working 8051 source" and we remind you to carefully preserve your original 8051 source file.

If systematic changes are necessary, you may want to work outside the XA Development Environment in order to use the familar, and likely more advanced, features of your favorite text editor. We've found using a UNIX™-style stream editor such as **sed** or **awk** can be very useful, especially in the case of large-scale syntax changes to big source files. It is well beyond the scope of this application note to describe such programs.

We've found the following to require attention:

- 8051 Source code intended for some assembler other than MetaLink

  If you discover large numbers of translation warnings due to unrecognized directives, consider scanning your 8051 working source for directives. Compare them to the XA Development Environment Assembler (see "Help" → "XA Assembler"→ "Directives") and make indicated changes. Some directives don't have equivalents in the XA tools; comment these out or remove them.

  If you see large number of XA assembler errors due to syntax errors, you may have to make significant changes to your working source. Compare the syntax of your source to that described in the XA Development environment ("Help" → "XA Assembler).

- Include files

  The translator doesn't handle all forms of "include" and may or may not pass through a given include file directive. It may be practical in some cases to remove "include" instances and insert the "include" file contents. Otherwise, you may change the existing syntax in your working 8051 source file to one of the alternatives accepted by the XA tools:

  ```
  #include file.ext
  ```

  or

  ```
  $include file.ext
  ```

  and translate the files individually.

Once more we will remind you to keep a protected copy of the 80C51 original source!

2.1.6 What you've got: "tentative XA source"

After you've taken care of translator warnings and any errors, the output file you've got is "tentative" because:

- It doesn't have the proper XA startup vector or any other XA interrupt vector.
- It uses an 8051-style stack.
- It may contain warning messages.
- It might contain illegal or unrecognizable syntax with respect to the XA assembler.
- No comments contain anything about XA implementation.
- If you chose to have 8051 instructions included in the output, there are lots of lines you'll probably want to remove.

We'll see how to resolve these problems in the next section.

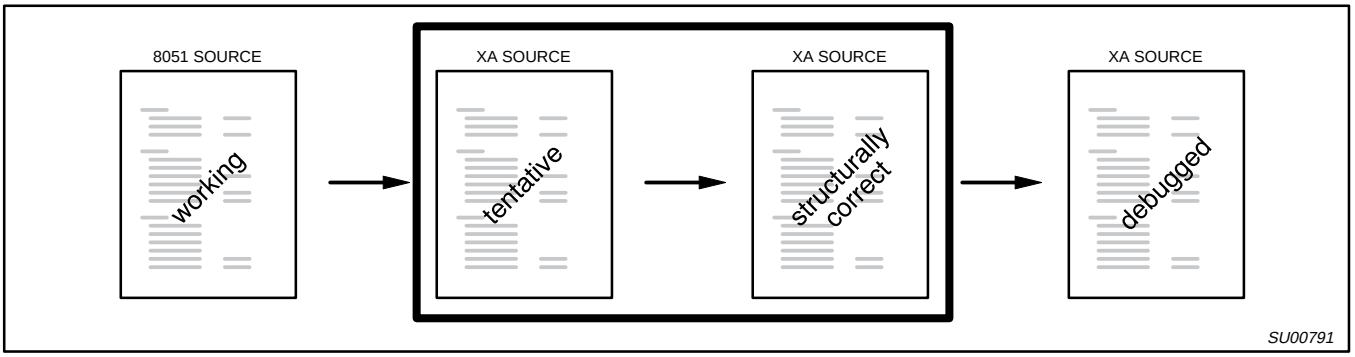

Figure 7.

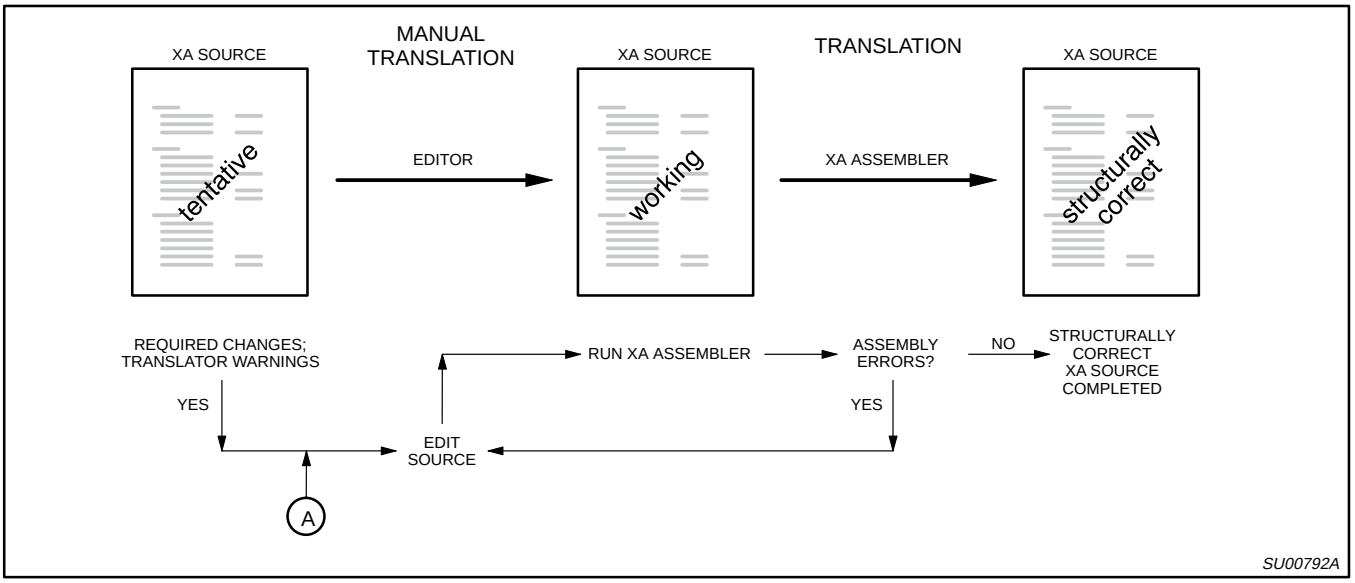

Figure 8.

2.2 Continuing translation: Producing structurally correct XA code

The next step of translation, producing structurally correct XA code from tentative XA code, is summarized in Figure 8.

The diagram item “required changes; translator warnings” denotes the following:

- Installing an XA startup vector and any other required XA interrupt vectors.
- Implementing an XA stack.
- Removing or resolving warning messages.

Later on in this section we will offer a few comments on the remaining issues of tentative XA source: resolving illegal or unrecognizable XA syntax, adding comments about the XA implementation, and removing 8051 instruction comments optionally included by the translator.

Although this application note is targeted to dealing with software translation issues, we’ll also cover the key areas of hardware preparation for real XA hardware. If you are just simulating for now, please feel free to maintain code for real hardware; it won’t have any effect on the simulator, and you’ll be prepared for eventual use on a real XA.

2.2.1 XA startup vector (simple case)

Practiced 8051 programmers usually have a standard program template readily available. In its essence—ignoring interrupts—it looks something like this:

```
     org 0
     ljmp    start                   ; first executed instruction

     org 040h
   start:
     mov     SP,#stack_bottom        ; initialization code
     ...
```

If you've studied the XA, you know that the equivalent minimum startup looks like this:

```
     org 0
     dw      8F00H                   ; initial PSW
     dw      start                   ; reset interrupt vector
     ...

     org 0120h
   start:
     mov     SP,#stack_top           ; initialization code
     ...
```

(See Appendix 2 for a complete listing of a standard XA initialization template.)

As you can see, the XA startup is based on a pair of word vectors at the beginning of code memory, the first taken as the initial Program Status Word (PSW) value, and the second as the initial Program Counter (PC). There is no way for the XA Development Environment translator to automatically translate the 80C51 startup to the XA form, so you'll have to do it yourself: Replace the initial jump with two "dw" define words, the first setting the initial PSW (8f00H is the recommended default) and the second containing the address of the first instruction to execute.

Use this code as a guide for simple cases, such as evaluating code using the XA Development Environment XA simulator. The next section discusses more complex cases. We'll take up the differences in stack initialization further below.

NOTE: It may be useful to make these changes to your working 8051 source file, especially if you return to the translation step more than once.

2.2.2 XA startup and interrupt vectors (more complex case)

Practiced 8051 programmers usually have a vector template for simple applications that looks like the following:

```
     CSEG

     org 0
     ljmp start                      ; Reset Vector

     org 03H
     reti                            ; EXT 0 interrupt: not used

     org 0BH
     reti                            ; Timer 0 interrupt: not used

     org 13H
     reti                            ; EXT1 interrupt: not used

     org 1BH
     reti                            ; Timer 1 interrupt: not used

     org 23H
     ljmp SerialISR                  ; Serial port interrupt

     org 40H
   start:
     mov     SP,#stack_bottom        ; initialization code
```

(This example includes all the interrupt vectors on the core 80C51; derivates differ.)

Drawing again on the complete startup template given in Appendix 2, here's an expanded template that includes the key interrupt vectors for the XA:

```
     dw      08f00H, Start            ; Reset PSW, vector
     dw      08f00H, BreakVec         ; breakpoint PSW, vector
     dw      08f00H, TraceVec         ; trace PSW, vector
     dw      08f00H, StkOvfVec        ; stack overflow PSW, vector
     dw      08f00H, Div0Vec          ; divide by 0 PSW, vector
     dw      08f00H, URetiVec         ; user reti PSW, vector

     org     040H                     ; (TRAP 0–15 exceptions omitted)

     org     080H                     ; Event interrupts:
     dw      08900H, ExtInt0Vec       ; external interrupt 0
     dw      08900H, Timer0Vec        ; timer 0 interrupt
     dw      08900, ExtInt1Vec        ; external interrupt 1
     dw      08900H, Timer1Vec        ; timer 1 interrupt
     dw      08900H, Timer2Vec        ; timer 2 interrupt

     org     090H
     dw      08900H, Rxd0Vec          ; Serial port 0 receive
     dw      08900H, Txd0Vec          ; Serial port 0 transmit
     dw      08900H, Rxd1Vec          ; Serial port 1 receive
     dw      08900H, Txd1Vec          ; Serial port 1 transmit

     org     0100H                    ; (Software interrupts omitted)


     org     00120H                   ; Start of executable code area.

   BreakVec:
   TraceVec:
   StkOvfVec:
   Div0Vec:
   URetiVec:
   ExtInt0Vec:
   Timer0Vec:
   ExtInt1Vec:
   Timer1Vec:
   Timer2Vec:
   Rxd0Vec:
   Txd0Vec:
   Rxd1Vec:
   Txd1Vec:

     reti    ; Location to route interrupts/exceptions with no specific
             ; handler code. This could prevent lockup particularly due
             ; to an unexpected exception such as stack overflow if
             ; there was no vector or handler whatsoever.
             ; (labels for traps and software interrupts omitted)


   ;=====================================================================


     org     0                        ; System exception interrupts:
   ; Beginning of initialization code.

   Start:
     mov     R7, #0100H               ; initialize stack pointer top
     ...
```

For brevity, we've omitted the traps and software interrupts and their corresponding labels for the "reti tie-off" found in the original, but we'd encourage you to include them in all code destined for execution on an actual XA target.

The purpose of this comparison is to highlight the differences in interrupt setup when translating an application between the 80C51 and the XA.

We want you to notice that the XA provides extensive support of exception, trap, and event interrupt mechanisms. Tying off unused interrupts takes a lot more "bookkeeping", but you'll recognize that this cost is well worthwhile when you start using these.

2.2.3 The XA stack

We recommend that the first instruction executed in any XA application be a stack initialization; even if this does no more than duplicate the default initialization (completely sufficient for small test programs).

Odds are that the translator will find an 80C51 stack initialization and translate it, but you'll have to manually locate any such code and replace it with XA-specific initialization, if for no other reason that the byte-sized 80C51 stack initialization will be translated to a byte operation; the translator doesn't handle SP as a special case.

Converting from an 8051-style stack to an XA stack is fairly easy if you understand all the issues. Let's take a look at them, in order of increasing complexity:

Basics

In all but the most trivial 8051 applications you'll most certainly see an instruction like this

```
mov             SP, #stackbottom
```

You should already know the differences between the 8051 and the XA stacks:

- The 8051 stack grows upward, while the XA stack grows downward.
- The 8051 stack contains bytes, while the XA stack contains words.
- The 8051 has only one stack, the XA has a System and a User Stack.
- The 8051 stack must be located in on-board memory, while the XA System Stack must be in the first 64K of RAM, on-chip or off.
- The XA user stack may be located in any memory region.

Because 8051 stack space is generally at a premium, stack allocation in 8051 applications is usually done with great care. With any luck, your original 8051 application documentation will describe the stack allocation in detail, including the expected "high-water mark" of the maximum expected usage. If not, we encourage you to spend a little time to study and characterize how the stack is implemented.

Fortunately, stack allocation is much easier on the XA because stack space and placement is much less restricted.

In simple cases—with simple subroutine calls/returns and no interrupts—you'll be able to transform the 8051 stack allocation to a first approximation by using the following steps:

1. Use only the System Stack.
2. Declare the top of the stack in the XA at a specific RAM address.
3. Allocate as many words in the XA as the original application allocates bytes.

In other words, it may be entirely sufficient to use code like this:

```
   ENDRAM       EQU 0FFFFh                        ; last cell
STACKSIZE       EQU 200h                          ; in bytes
ISTACKPTR       EQU ENDRAM–STACKSIZE – 1          ; initial value

...

   mov          SP,ISTACKPTR                      ; recommended 1st initialization
```

As the XA initializes the System Stack Pointer to 100H, that is, within on-chip memory space for all XA devices, you may be able to use the default setting for small applications. Even so, we recommend that you explicitly document the stack allocation for the translated application and explicitly set the stack pointer. Please also note that the XA provides a stack overflow exception if the stack reaches 80H. It's often a good idea to add an interrupt handler for this overflow exception so that your application can recover from stack overflow.

That said, we'll warn you that more complex 8051 applications may require considerably more special handling with respect to stack operation translation. We'll take a look at the next more complex case—handling interrupts—in a following section and then take a look at special topics later on.

The System Stack

We're going to assume that you're translating a single-threaded 8051 application to the XA for the purposes of this application note; translating a multitasked 8051 application is beyond the scope of this application note.

In most cases, Philips recommends you operate the XA initially in System Mode (see the *XA User Guide*, section 4.2.4, for example) and we'll extend that recommendation here to suggest that most translated applications be operated entirely in System Mode.

You set System Mode by setting the SM bit in the initial PSW. If you do this, your application will have full access to all XA registers, instructions, and memory spaces. Further, you'll have only one stack and one stack pointer, and you can ignore any discussion of the User Stack Pointer. R7 will always contain the System Stack Pointer, which you can simply call "the stack pointer" or SP. Throughout your code, you need to take care that no operation you perform sets PSW.SM to zero.

Interrupts add another level of complexity to the picture. When handling interrupts you must "confirm" the System Mode setting of PSW.SM by making sure that the new PSW portion of each interrupt vector contains a value that will result in PSW.SM = 1. In other words, to preserve System Mode, make sure that all values of PSW specified in interrupt vectors leave PSW.SM=0.

The System Stack and Interrupts

Let's review how interrupts are serviced: As you can see in the XA User Manual, the address of the next instruction and PSW in force at the instant of the interrupt are saved on the System Stack. (**All** interrupts—exception interrupts, event interrupts, software interrupts, and trap interrupts—use the System Stack exclusively.) The new value of the Program Counter (PC) and the new PSW are taken from the code-memory vector associated with that interrupt. Normally, you'll use the RETI instruction at the conclusion of an interrupt service routine to restore the interrupted program flow and Program Status.

Unlike the 8051, the XA PSW value is automatically saved any time an interrupt occurs, so it is unnecessary to save the PSW explicitly. (This means you can save a few bytes in translated programs where the original 8051 code did the save and restore by manually removing the explicit PUSH PSW and POP PSW instructions.)

As described in the *XA User Guide*, section 4.3.2, the amount of information processed on the stack during interrupts varies between the default 24-bit XA operation mode and the optional 16-bit Page 0 mode.

The User Stack

What about the User Stack Pointer? We recommend that you don't worry about it for the purposes of the vast majority of translated applications. Simply make sure that PSW.SM is always set. R7 will always contain the System Stack Pointer.

Of course, the XA offers support for multitasking and, for this purpose, you'll need to know how to use User and Supervisor Mode functionality. See Chapter 5 of the *XA User Manual*, and look for forthcoming applications notes on general multiasking support and running multiple virtual 8051's on a single XA.

Advanced Stack Issues

Clever uses of the 8051 stack have been a key feature of that architecture. We can think of a number of schemes you may have to recognize and handle, as well as some XA-specific issues:

- Multiple 8051 Stacks

  Some applications demand multiple stacks, and we've seen some translation problems handling these. Fortunately, these generally involve schemes in which registers other than the 8051 SP is used for stack pointers—we have generally seen the "SP stack" used only for interrupts in some applications—and so these can be handled routinely.

- 16-bit 8051 Stacks

  Likewise, some 8051 applications require 16 bit stacks, and the ones we've seen also handle these outside the scope of the hardware SP.

- Special PSW handling within interrupt service routines (ISRs)

  When you are translating an 8051 ISR, it is almost certain you'll see a "PUSH PSW" somewhere near the beginning and a matching "POP PSW"—or some equivalents—near the end. Unlike the XA, 8051 interrupt processing doesn't automatically save and restore the PSW value, and the vast majority of applications will require that the foreground's PSW be unchanged through interrupt service.

  However, it is remotely possible that an 8051 application breaks this general rule and depends on an altered value of PSW on return from an ISR.

  Here's a somewhat contrived example: an application with no arithmetic processing whatsoever and extreme storage requirements might use the CY flag to indicate the completion of some ISR-processed event to a foreground processing loop.

  The translator can't detect this special method and certainly can't defeat the standard interrupt processing peformed by the XA hardware, saving and restoring the PSW, so you'll have to modify the code manually to make this algorithm work. Fortunately, XA storage is likely to be much more plentiful and you'll probably be able to use a much more conventional technique.

  By the way, you can manually remove the PUSH/POP pair from the translated code if you want.

- Stacks extending over physical address boundaries

  80C51 stacks necessarily reside within the IRAM memory space. XA stacks may be in IRAM, in external RAM, or --as the stack grows downward across the boundary-- in both. There is no special consideration for the placement of the stack with respect to the internal/external boundary, but you should be aware that access speed may be different for the two types of memory.

2.2.4 Feeding the Watchdog

There is one critical mechanism provided on the XA that has no equivalent in the basic 80C51 architecture (but is implemented in some 80C51 derivatives): the watchdog timer. If you fail to pay attention to the watchdog timer—which is activated automatically by a hardware reset—your translated applications will crash mysteriously!

Fortunately, it is very easy to take care of the watchdog. For most developmental purposes, we recommend simply deactivating the entire subsystem early in your XA initialization code. (Right after stack pointer initialization is a good place.) We will draw again from the complete startup initialization template in Appendix 1. The watchdog timer uses a special “feeding” sequence to enable any changes to its configuration:

```
wdoff:    equ $00                ; WDCON value to turn off WD
...

   mov.b   wdcon,#wdoff          ; Turn off watchdog timer.
   mov.b   wfeed1,#$a5           ; Feed watchdog: use new config
   mov.b   wfeed2,#$5a
```

This code sequence deactivates the watchdog subsystem and places it in a known state.

The watchdog mechanism is not implemented in the XA Tools simulator, and these instructions will have no effect.

2.2.5 Obligatory Hardware Initialization

The XA contains a number of hardware initialization registers. (Namely, BCR, BTRH, BTRL, P0CFGA, P0CFGB, P1CFGA, P1CFGB, P2CFGA, P2CFGB, P3CFGA, and P3CFGB.) The default power-up values of these are often sufficient to get your application running, but the defaults won’t work in every case, and we recommend that you consider initializating these registers as obligatory among the first instructions after reset.

You’ll definitely need to use non-default values to speed up external memory access and to enable external RAM access when executing code in internal code space.

Explaining how to set all these registers is well beyond the scope of this document, but we’ll give a common example, taken from Appendix 1:

```
   mov.b   bcr,#waitd+bus16+adr20   ; Set up bus configuration.
   mov.b   btrh,#dw5+dwa5+dr4+dra5  ; Config bus timing to longest
                                    ; bus cycles
   mov.b   btrl,#wrpuls2+holdmin+ale05+cr4+cra5 ; short ALE, min data
                                    ; hold.

   mov.b   p0cfga,#pcfga_pp         ; Configure port0 types for bus.
   mov.b   p0cfgb,#pcfgb_pp
   mov.b   p1cfga,#p1cfga_bus       ; Configure P1 for quasi–bidirec
   mov.b   p1cfgb,#p1cfgb_bus       ; except A3 – A0 are push–pull.
   mov.b   p2cfga,#pcfgb_pp         ; Configure P2 types for bus.
   mov.b   p2cfgb,#pcfgb_pp
   mov.b   p3cfga,#p3cfga_bus       ; Configure P3 for quasi–bidirect
   mov.b   p3cfgb,#p3cfgb_bus       ; except WR, RD are push–pull.
```

This won’t have any effect on the XA simulator. See section 7.3 in the *XA User Guide* for more details.

2.2.6 Disposing of Warning Messages

Warnings are written as comments to the output file. We recommend resolving all warning messages before proceeding with the translation process.

The majority of warnings are usually due to 8051 instructions that translate directly into XA “compatibility” instructions:

```
JZ/JNZ
JMP[A+DPTR]
MOVC A,[A+PC]
MOV A,[A+DPTR]
MOVC A,[A+DPTR]
```

The resulting XA code may be obscure or non-functional, and the translator flags this usage to warn you to check each instance. You’ll have to understand what the original code does to do this.

A second class of warning messages are generated for bit references, since identities of bits—even those with identical functions—are different on the 8051 and the XA. Some of these problems will be identified in a later stage by the XA assembler, but you’ll have to review all the bits referenced in the translated program sooner or later. No matter what, we can’t over-emphasize the rule that’s equally valid for 80C51 and XA coding:

“All SFR and bit references should be symbolic.”

The first reason for this rule is that identically named SFRs and bits may appear at different places in different 80C51 and XA variants. Secondly, symbolic references receive some error-checking by the assembler that can’t be done with explicit numeric references.

Details of Compatibility Instructions

The XA “compatibility” instructions offer you the option of continuing to use a number of highly-functional 80C51 instructions in your XA code. To do so requires that you understand, in detail, any differences that exist between the 80C51 and XA implementation of the instructions, as well as the specifics of each implementation. You may choose to adjust the code to continue to use these instructions, or recode into native XA instructions. (We generally recommend recoding whenever practical.)

Here are the details, along with alternatives for recoding into native XA instructions:

- JZ/JNZ

  JZ/JNZ in the 80C51 uses the current contents of ACC –not the result of a prior ALU operation–to determine whether the jump is taken or not. This function is duplicated in the XA using the translated ACC, R4L. Check the program logic to make sure that the the value in R4L is valid at the time the JZ/JNZ is executed.

  XA native alternative: recode to use CMP followed by Bxx.

- JMP[]

  The JMP[A+DPTR] instruction, used for implementing jump or call tables, depends on the length of each entry in the jump table. Because the lengths of translated instructions in the tables are likely to differ, the translated algorithm probably won’t work without further manual intervention.

  XA native alternative: implement a vector table instead, do an indexed fetch followed by a jump-thru-register.

- MOVC A,[A+PC]

  The MOVC A,[A+PC] instruction fetches a constant byte out of code memory. Since the algorithm depends on the length of this instruction –one byte on the 80C51, and two bytes on the XA– you’ll have to make an adjustment to make the translated code work correctly.

  XA native alternative: Use standard register-indirection or MOVC to get bytes or words from data or code memory, respectively.

- MOV A,[A+DPTR] and MOVC A,[A+DPTR]

  These instructions are used to fetch one of 256 bytes in a data- or code-memory table, respectively. As long as the translated A, R4L, and the translated DPTR, R6, contain the correct values, algorithms implemented with this instruction should continue to work in the XA.

  XA native alternative: Use standard register-indirection or MOVC to get bytes or words from data or code memory, respectively.

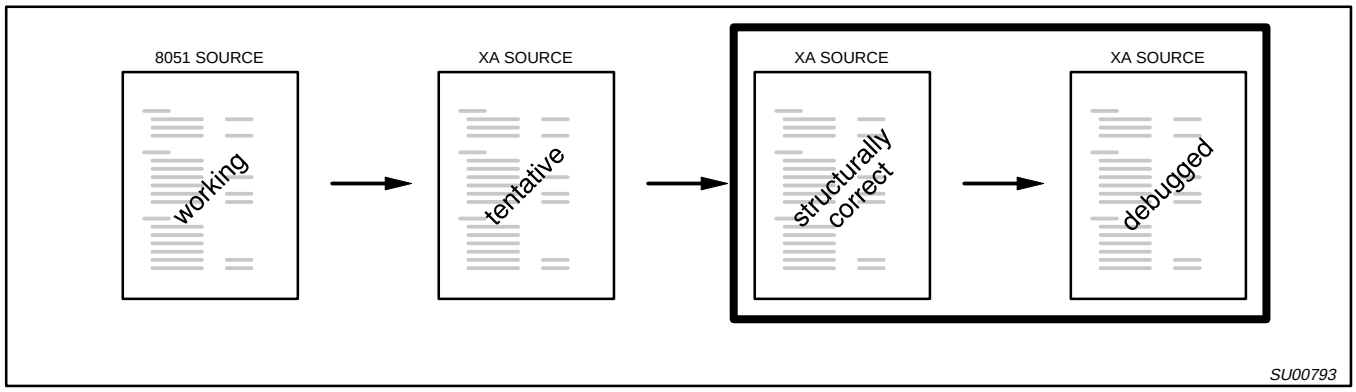

Figure 9.

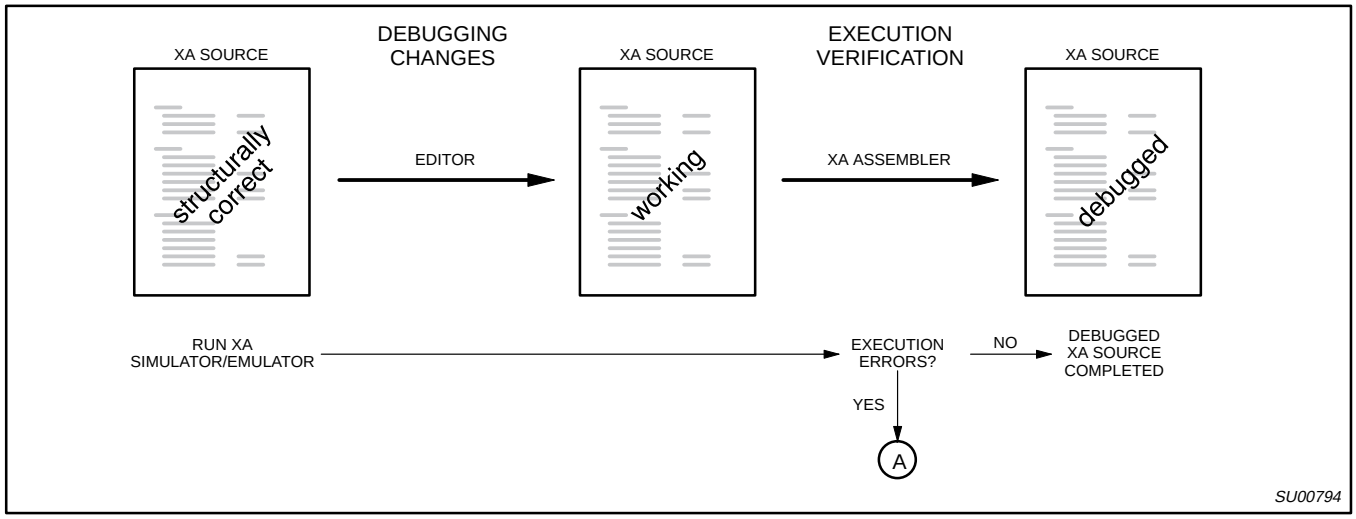

Figure 10.

2.3 The final step: Generating debugged XA code

There's good news at the final step: Transforming structurally correct XA code into debugged XA is almost identical to the standard native code-development cycle. You'll run the XA simulator or XA emulator to verify your code, make corrections, re-assemble, then re-verify, continuing until your program is proven. If you've been careful during previous steps, you shouldn't have any translation-specific problems here.

Figure 10 describes the details.

The only difference for translated code is the very slight chance you'll need to return to an early translation step to correct an unforeseen systematic error in your code. We don't expect this to happen except in extreme cases of specialized code.

As standard XA development techniques are covered by other materials, we'll cut this section short and turn to special topics.

3. SPECIAL TOPICS

We've translated a good deal of 80C51 code, ranging from short functional snippets to applications of considerable size. In the process, we're reminded of the wealth of programming tricks developed over the years by 80C51 programmers to extract all possible functionality from the architecture and to overcome limitations of early tools.

We've also seen a few cases where the 80C51 to XA Translator needs a little extra help.

We've kept notes of these, and we'll share them with you—along with our recommendations for fixes—in this section.

3.1 Trouble-making items

Here's a list from our notes of some potential troublespots. You may want to scan your 80C51 and translated source code for these. Most of these issues are covered in detail in this Application Note.

- 80C51-specific comments in translated code.
- 80C51-style increments in XA that don't affect condition codes, but possibly should.
- Any occurence of the [...+DPTR] operand.
- Any operation on the 80C51 PSW register.
- Any explicit push or pop.
- Over-terse console messages (due to extremely limited ROM size in many 8051 applications).
- Multiply/Divide algorithms, which may translate and run correctly, but very slowly if not re-written for the XA.
- Indirect references (@R0, @R1) in 80C51 code.

3.2 Translating indirect references

Indirect addressing doesn't always translate well from the 80C51 to the XA since the 80C51 implements 8-bit indirection through R0 or R1, while the XA can do 16-bit indirection through any of R0 thru R7.

For programs that are heavily dependent on the 80C51 implementation, you may consider setting SCR.CM to put the XA into 80C51 "compatibility mode" in which XA R0 and R1 function as 8 bit index registers.

In general, we recommend that you examine the issues carefully and recode if necessary to use native 16-bit addressing whenever possible. The increased generality of your code will almost always pay back the effort.

3.2.1 Indirect reference failure

In a few cases, the translator can produce XA code which won't function correctly.

For example, the XA translator translates the 8051 code sequence

```
MOV R1,PTR      ; get a pointer from IRAM
MOV A,@R1       ; load @ptr to ACC
```

to

```
MOV.B R0H,PTR   ; get a pointer from IRAM
MOV.B R4L,[R1]  ; load @ptr to ACC
```

In the 80C51 version, the IRAM location "PTR" contains an 8-bit pointer, which is fetched to R1, one of the two 80C51 index registers. (The source of the pointer is not important; this example would be equally valid if the value of R1 is loaded from any other 8-bit location.) The code uses the byte-size pointer in R1 to load another 8-bit quantity to the accumulator.

In the XA version, an 8-bit pointer is fetched to the XA register corresponding to 80C51 R1, namely R0H. For native mode, the translation has already gone astray as no byte-length index registers are implemented in the XA, and R0H isn't a proper index register anyway. The next instruct, an indirect load, fetches through R1, which is completely uninitialized by this sequence, so the translation fails.

In XA compatibility mode, byte-length pointers are available, but they are defined as the least-significant byte of either R1 or R0, so the 8-bit pointer should be loaded to either R0L or R1L. This translated sequence effectively uses R1L, which is likewise uninitialized, and the translation fails.

3.2.2 Recommendations for Indirect References

The most general solution is to search for all "@R0" and "@R1" addressing in the 8051 code or all "[R0]" and "[R1]" references in translated code. Note that translated code won't contain any references like "[R2]" through "[R7]" since no 8051 code generates these. Although there are no useful changes you can make to the 8051 code prior to translating, you may be able to see better how the code works in the original source, and it is a good idea to consider how pointers might be converted to words from bytes at the most convenient level. Make sure to distinguish between true byte-sized pointers and word-size pointers used to access external memory. Some 8051 programs keep word pointers in adjacent registers or IRAM cells, but there's no guarantee, since we've seem some that don't.

Don't forget that pointers are sometimes checked against limits, and you'll need to convert the limit checking code as well. 8051 CJNE's are translated to CJNE.B's. You may have to manually convert these to CJNE.W's. It's probably a good idea to replace the CJNE tests completely with CMP instruction followed by a conditional branch. In the 8051, the CJNE was the only way to do a non-destructive test, so it was overused—and possibly misused—in situations where a limit might be reached **or** exceeded.

3.2.3 Picking a register for indirect references

If you are manually modifying translated code—following our recommendation to recode certain 80C51-specific constructions—you may need to pick a new register for indirect addressing. Here are some pointers:

In the most general case, you can't use XA registers R0 through R3 for indirect addressing because it is possible that 8051 registers R0 through R7, which translate into byte registers in R0 through R3, namely: R0L, R0H, R1L, ...R3H, might contain useful data.

Of the group R4 through R7, R4 is best preserved for translated code that assigns R4L as "ACC" and R7 is the Stack Pointer. The chances are good that DPTR in the 8051 code was doing something useful, so it's XA twin, R6, is busy.

That leaves R5 as a good candidate. (If R5 is also busy, then you'll have to save a register temporarily; unless stack space is very limited, the stack is as good as any place. Alternatively, you may want to switch register banks if there are free registers in a different bank.)

Remember to consider that it is a standard practice in 8051 coding to assign symbolic names to registers, so you may need to take extra care to track register usage.

3.3 Translating “degenerate” 8051 source code

This section discusses some “degenerate” 8051 programming practices in this section. We use this term for some constructs because they are very 8051-specific, developed over time by programmers to make the most of the available tools and the 8051 architecture.

Most of these don’t translate mechanically very well. You can manually modify the code used in around this kind of code and get it to work, but our experience indicates that it is often better to it outright with maintainable, clear XA code. We’ll show you how.

3.3.1 “Here” relative addressing

Some typical 8051 source code includes branch instructions resembling the following:

```
    JB      testbit1,$+3
    JNB     testbit2,somewhere
    RET                             ; or whatever
```

Early 8051 assemblers had limited symbolic capacity and encouraged such usage. This code should have been written

```
    JB      testbit1,xyz
    JNB     testbit2,somewhere
xyz:
    RET
```

The translator cannot perform this replacement mechanically, and the XA assembler will not accept “here” relative constructs except in specific cases, so we recommend that you correct the original 8051 source code by adding labels. It is a good idea to search 8051 source code for “$-” and “$+” and resolve all these before attempting translation.

Note: The translator will not translate operands like “.+3” and “.-6”. The period operator does not mean “here” in the XA assembly language, where it is reserved for bit addressing constructions, e.g., “SCR.PZ”. The translator will emit an error message and leave dot-based operands alone.

3.3.2 Branch to “Here”

A very typical 8051 source code construction looks like this:

```
    JB      eventbit1,$
```

Of course, at the cost of inventing a label name, this code could have been written

```
wait:
    JB      eventbit1,wait
```

Unlike the 8051, the XA cannot branch (jump) to an odd address. If an labeled instruction prospectively occurs at an odd address, the assembler inserts a NOP before any jump target to place it at the next even address. Without a label, however, the assembler can’t distinguish this instruction as a jump target, so the translator generates a label when this construction is found. The generated label is not very informative, appearing, for example, as follows:

```
XA_ADJUST_0005:
    JB      eventbit1,$
```

This guarantees that the assembler will even-align the target instruction. We recommend that you substitute a better label, at minimum; even better, consider replacing the “$” reference with the full label.

3.3.3 Branch to “here+offset” or “here-offset”

It’s a common 8051 practice to use knowledge of the the length of instructions and the exact movement of the program counter during instructions to use specify branches in conditionals to a place “so many bytes from here”. Without careful documentation, the code is often incomprehensible. For example, this 80C51 fragment maps alpha characters in the accumulator to upper case:

```
    CJNE    A,#’a’,$+3
    JC      C_IN_1
    CJNE    A,#’z’+1,$+3
    JNC     C_IN_1
    ANL     A,#11011111B
C_IN_1:
    RET
```

This code won’t translate. Even though it is obvious what this segment does, it is not immediately obvious how it does it. (See also “Compare Trees” in section 3.3.7.) To which instruction does the first CJNE branch if the branch is not taken? If you keep this construction, you’ll have to figure this out.

We recommended you recode this kind of construction in clear XA code, for example as follows:

```
    CMP.B   R4L,#’a’
    BL      EXIT
    CMP.B   R4L,#’z’
    BG      C_IN_EXIT
    AND.B   R4L,#11011111B
EXIT:
    RET
```

3.3.4 In-line strings

Some 8051 assembly programs follow the practice of embedding strings within code sequences that use them. Often this is done where a canned string is to be output to the console, but it may occur in other contexts, for example in a Tiny BASIC interpreter, where the BASIC commands are sought by successive code routines and the compare strings are embedded in the code.

The print string case might look like the following:

```
    CALL    String_Out
    DB      “This string goes to the console”,0
    NOP                          ; or whatever
```

The subroutine “String_Out” obtains the string address from the stack and outputs the string, in the process adjusting the stack value to that of the following instruction –here a RET. A trailing null is often used as a string delimiter. (Other schemes that delimit the string are possible, for example, a leading byte count, or setting the 80h bit of the final character, but these have no effect on the basic method.) The code at String_Out usually starts out as follows:

```
String_Out:
   POP     DPH
   POP     DPL
   MOV     a,#0
   MOVC    A,@A+DPTR
   ...
```

This method works on the 8051, and very efficiently, because it take advantage of the fact that the CALL places a full 16-bit address on the stack in a single instruction. Unfortunately, the 8051 has no corresponding 16-bit pop to anywhere else but the Program Counter (i.e., via the RET instruction). So the string pointer must be retrieved and placed in DPTR, the most useful place for it, one byte at a time.

The translator transforms this code to

```
    CALL    String_Out
    DB      “This string goes to the console”,0
    NOP
....

String_Out:
   POP.B   DPH
   POP.B   DPL
   MOV.B   R4L,#0
   MOVC.B  A,[A+dptr]
   ...
```

which looks almost the same, but operates very differently. First, the XA CALL places a 16-bit return address on the stack in page 0 mode. In native XA mode, CALL generates a 24-bit return address in two 16-bit stack cells.

When it comes to retrieving the string address, there are no 8-bit stack operations on the XA, so an entire word is popped for each byte POP. If the XA is in page 0 mode, this code will remove two 16-bit words from the stack, one more than was pushed by the CALL, causing a likely fatal stack imbalance as well as incorrect data in DPTR. In native mode, this code will remove the two words pushed by the CALL, leaving the stack in balance, but DPTR will contain an incorrect value. In other words, this code won’t work when translated.

What to do? The answer depends on your application.

If you are certain to use page 0 mode, the two byte POPs can be replaced by a single word POP. This will keep the stack balanced and retrieve the pointer value correctly. The return will always be to an even address, so assure that it is the correct one by placing a label on the first instruction following the string. Eliminate the special 8051-equivalent “A+DPTR” form in favor of a straight indirect code memory fetch, in the process eliminating the need to clear the accumulator, and gaining an implicit autoincrement:

```
    CALL    String_Out
    DB      “This string goes to the console”,0
new_label:
    NOP
....

String_Out:
    POP.W   R6
    MOVC.B  R4L,[R6+]
...
```

Don’t forget to remove the code that increments DPTR further below in the output subroutine.

In XA native mode, on entry to String_Out, address bits 16 thru 23 will be in a word on the top of the stack. In some simple cases you might be able to simply POP these and discard them, but resist the temptation, since this won’t always work if your code moves.

In general, it is best to remove this construction entirely so there will be no mode or address dependence:

```
msg1:
   DB “This string goes to the console”,0
...
   MOV     R6,#msg1
   CALL    String_Out
...
String_Out:
   MOVC.B  R4L,[R6+]
...
```

When you make this conversion, you can forget about the alignment issue, i.e. the need for a dummy label on the code following the embedded string goes away. Of course, you’ll have to invent some labels for the strings and place them somewhere out of the code flow.

3.3.5 General In-line args

We’ve seen strings placed in-line in 80C51 code in the previous section and these at least have the somewhat redeeming value of self-documentation. Imagine finding a code sequence like the following:

```
STK_ER:
          CALL AES_ER
          DB 0FH
```

This is a bit mysterious, especially if the code following is of no particular relevance. To make a long story short, the value ‘0FH’ is an error number, and this routine handles stack errors in a Tiny Basic interpreter. The routine AES_ER could retrieve the error number by popping the presumptive return address into DPTR and doing a MOVC.

This is a good example of the lengths to which 8051 programmers have gone to squeeze optimum performance from the architecture.

Since there are many more registers available in the XA, your translated code could simply move the error code into a free register before calling AES_ER. The error code could even be pushed on the stack:

```
    ADDC    R7,#-2
    MOV.B [R7],#0FH
```

without involving any registers. Whatever you do, we advise you to eliminate in-line arguments.

3.3.6 Program Resets

It is not uncommon to see an 8051 program instruction:

```
JMP     0000H
```

This will translate, but the result won't be correct since there's not an executable instruction at 0.

Better to use:

```
RESET
```

3.3.7 Compare trees

The 8051 construction

```
    CJNE    A,#'a',$+3
    JC      C_IN_1
    CJNE    A,#'z'+1,$+3
    JNC     C_IN_1
...
```

works because the CJNE instruction performs a subtract and sets the carry flag. This construction is the only way to "bin" numbers in the 8051 using the accumulator only. But it is often very difficult to understand.

We recommend recoding to use successive XA compares and branches.

Note that XA CJNE's don't include a form that allows comparing two register arguments, and in the XA you'll likely have more working values in registers.

3.3.8 Code Table Fetches

To do table look ups returning bytes, it's common to see 8051 code that does the following:

```
MOV   A,#index          ; or calculated
MOV   DPTR,#TABLE_BASE
MOVC  A,[A+DPTR]        ; A <-- indexed byte value
```

This code sequence will work equally well on the XA, and may often simply be retained after translation, assuming you can live with the limitations:

1. The table is in the current code page
2. You must use the simulated DPTR (R6) and the simulated accumulator, R4L
3. The table can only be 256 bytes long
4. You can only conveniently fetch bytes this way
5. This minimal form uses three register bytes
6. It's not easy to expand to use multiple indexes

Using native XA code is the alternative. Let's see what that looks like in a similar case:

```
MOV.W   Rn,#index     ; or calculated
ADD.W   Rn,#TABLE_BASE
MOVC.B  RnL,[Rn+]     ; RnL <-- indexed byte value
```

Note: we're re-using the same register for the sake of illustration, which you may not want to do in all cases. Using native code has the following advantages:

- The table is in the current code page or through ES, which can point anywhere in code space.
- You can use any registers for the index and table base.
- You can create tables up to 64K bytes long, maybe longer in some situations.
- You can fetch bytes or words, and the autoincrement makes longer fetches easy.
- This minimal form uses only two register bytes.
- It's easy to expand to use multiple indexes.

3.3.9 Using the stack for vectored execution

A sequence of 2 pushes followed by a "ret" instruction in 80C51 source code means that a new execution address has been calculated by some means; the only way to get it into the 80C51 PC is through the stack, for example:

```
..
push DPL
push DPH
ret
```

This kind of construct is used when "JMP @A+DPTR" is insufficient, for example, in the case of large or very sparse jump tables. Although we've seen cases where the translated code works correctly, it is both inefficient and obscure. We recommend recoding to use jumps through an XA register.

Threaded code requiring double-indirect jumps demands use of the single XA instruction to replace the mass of 80C51 code required to do this function.

3.4 Translating “untranslatable” 8051 source

Before you use the translator to mechanically translate 8051 code, you should be aware of a number of 8051 constructions that may translate without error, but will simply not work.

There are also some limitations on source code that the translator can handle, and it is best to manually or automatically scan your source code for these and eliminate them before attempting to use the translator.

3.4.1 PSW bit addressing

Some 8051 code takes advantage of the fact that PSW bits are bit-addressable.

None of these bits are addressable on the XA. This fact should be mechanically demonstrated by the lack of BIT definitions in “XA.EQU” (or its equivalent in your environment), but it is probably better to take care of this kind of problem explicitly, since you’ll have to make some code changes right off the bat if you’ve got PSW bit dependence.

3.4.2 P2 addressing

On the 80C51 it was common for applications with external RAM and ROM to use P2 addressing to address external RAM data memory. This trick provides a second 16-bit memory pointer, albeit an awkward one, for accessing external memory. Obviously, this extremity is not necessary for the XA, which provides plenty of memory pointer registers. We’ll describe how to recognize this trick and what to do about it.

Here’s what you’ll see in the 80C51 source code:

A sixteen-bit pointer is divided into two bytes. The high-order byte is written to port P2, while the low-order byte is written to or maintained in R0 or R1. This is followed by a MOVX to read or write a data byte through the 16-bit pointer.

(The trick: writes to the P2 SFR are gated to the external bus only when this doesn’t interfere with P2 addessing external program memory. When the 8051 fetches an instruction from external program memory, the contents of SFR P2 are ignored –but retained– and the most significant byte of the program counter is written to the external P2 physical pins. At all other times, the SFR contents are output to the physical pins.)

To translate these references into XA external data memory references:

1. Chose a word pointer register, XA-Rn.
2. Write the value formerly written into P2 into RnH
3. Write the value formerly written into R0 or R1 into RnL
4. Perform a MOVX indexed by XA-Rn

To translate these references into plain old XA internal data references—assuming there’s enough internal RAM available—substitute a MOV for the MOVX. Note that the XA will do an external memory access if the address exceeds the on-chip space.

3.4.3 R7 use

Just a reminder: the XA Stack Pointer is maintained in the XA register R7; avoid the temptation to use R7 as a general register, except in the unusual case that your application doesn’t use the XA stack.

3.5 Wrap-up

We’ve designed the entire suite of XA Development Environment tools—translator, assembler, simulator/debugger—to make the process of translating 80C51 code to the XA to be as smooth as possible.

With the exception in a few cases of the requirement to return to the original source code and re-translate—which occurs only when there’s some systematic differences between the original source and the requirements of the tools—the job is little different from any other native development and debugging job.

3.6 Trouble checklist

In trouble? Sometimes it is easy to lose perspective when you’re immersed in the details of a translation. Here’s a checkist to help you:

- ☐ Have you read the entire application note and studied the special topics?
- ☐ Have you organized your files and file-naming system to the necessary degree?
- ☐ Are you spending too much time understanding obscure 8051 constructs that would be better recoded into native XA code?
- ☐ Should your choice of Page 0 or compatability mode settings be reconsidered?
- ☐ Are you using the simulator effectively?
- ☐ Are you using the correct (up-to-date, variant-specific) include file (e.g., “XA–G3.equ”)?
- ☐ Are you attempting to access external memory on an development tool that only supports single-chip operations?
- ☐ Are you using the symbolic register-naming capabilities of the assembler to the best advantage to translate code, by translating functionality, then assigning a specific register?
- ☐ Are you re-editing translated code too often?
- ☐ Are you making piecemeal changes when systematic changes are required?
- ☐ Would you be better off re-writing the application from scratch?
- ☐ Do have a case so special you need help from Philips Applications?

3.7 Final Comments

We’ve seen that 80C51 code varies enormously with respect to quality, amount of documentation, dependence on architectural “tricks”, and so on. Each translation will have its own flavor. Ultimately, there’s no substitute for experience, so we encourage to you start with simple cases, take them through the process to completion, and then start again with more difficult translation problems.

APPENDIX 1: STARTUP+INTERRUPT PROTOTYPES

The following code is a complete template for software startup, interrupt vectors, and hardware initialization. Extracts from this code are used in several places in the text. You may obtain this code from the Philips BBS as file “XA-SKEL.ASM”.

```
$pagewidth 132t
$listing_min

;=========================================================================

; General purpose skeleton assembly file for the XA–G3

;=========================================================================

; This file may be used as a starting point to develop XA assembly
; language applications. Many definitions are included to simplify XA
; system initialization and to make the code more readable. The user
; will need to adjust the initialization values to suit a specific
; application. Some things to look at are: stack starting location,
; system configuration (8051 compatibility mode; page 0 mode, and
; peripheral timing), watchdog timer setup, bus configuration
; and timing, and port configuration (especially as regards bus
; operation).

; The default setup shown gives an XA with 8051 compatibility turned
; off, page 0 mode on, peripheral timing set to clk/4, watchdog timer
; turned off, the external bus configured to a 16–bit data width with 20
; address lines, bus timing set to the longest cycles but a short ALE
; and minimum data hold, ports; configured for quasi–bidirectional
; mode except for the pins related to bus operation, which are set to
; push–pull.

;=========================================================================

$include xa–g3.equ                    ; Model file for the XA–G3
                                      ; which defines all of the
                                      ; Special Function Registers
                                      ; and addressable bits.


$nolist
;=========================================================================
; Equates to for XA initialization and make setup code self–documenting:

; System Configuration register (SCR):
cmoff:      equ   $00   ; SCR value to turn off 8051 compatibility mode.
cmon:       equ   $02   ; SCR value to turn on 8051 compatibility mode.
page0off:   equ   $00   ; SCR value to turn off Page Zero mode.
page0on:    equ   $01   ; SCR value to turn on Page Zero mode.
time4:      equ   $00   ; SCR value for timer rate = clk / 4.
time16:     equ   $04   ; SCR value for timer rate = clk / 16.
time64:     equ   $08   ; SCR value for timer rate = clk / 64.

; Watchdog timer configuration register (WDCON):
wdoff:      equ   $00   ; WDCON value to turn off watchdog timer.
wdon:       equ   $04   ; WDCON value to turn on watchdog timer.
wdpre64:    equ   $00   ; WDCON value for prescale = 64 * TCLK.
wdpre128:   equ   $20   ; WDCON value for prescale = 128 * TCLK.
wdpre256:   equ   $40   ; WDCON value for prescale = 256 * TCLK.
wdpre512:   equ   $60   ; WDCON value for prescale = 512 * TCLK.
wdpre1K:    equ   $80   ; WDCON value for prescale = 1024 * TCLK.
wdpre2K:    equ   $a0   ; WDCON value for prescale = 2048 * TCLK.
wdpre4K:    equ   $c0   ; WDCON value for prescale = 4096 * TCLK.
wdpre8K:    equ   $e0   ; WDCON value for prescale = 8192 * TCLK.
```

```
; Bus Configuration Register (BCR)
waitd:      equ   $10   ; bcr value to activate Wait Disable.
busd:       equ   $08   ; bcr value to activate Bus Disable.
bus8:       equ   $00   ; bcr value to set 8–bit bus.
bus16:      equ   $04   ; bcr value to set 16–bit bus.
adr12:      equ   $00   ; bcr value to set 12 address lines.
adr16:      equ   $01   ; bcr value to set 16 address lines.
adr20:      equ   $02   ; bcr value to set 20 address lines.

; Bus Timing Register High (BTRH):
dw2:        equ   $00   ; data write cycle without ALE is 2 clocks.
dw3:        equ   $40   ; data write cycle without ALE is 3 clocks.
dw4:        equ   $80   ; data write cycle without ALE is 4 clocks.
dw5:        equ   $C0   ; data write cycle without ALE is 5 clocks.
dwa2:       equ   $00   ; data write cycle with ALE is 2 clocks.
dwa3:       equ   $10   ; data write cycle with ALE is 3 clocks.
dwa4:       equ   $20   ; data write cycle with ALE is 4 clocks.
dwa5:       equ   $30   ; data write cycle with ALE is 5 clocks.
dr1:        equ   $00   ; data read cycle without ALE is 1 clock.
dr2:        equ   $04   ; data read cycle without ALE is 2 clocks.
dr3:        equ   $08   ; data read cycle without ALE is 3 clocks.
dr4:        equ   $0C   ; data read cycle without ALE is 4 clocks.
dra2:       equ   $00   ; data read cycle with ALE is 2 clocks.
dra3:       equ   $01   ; data read cycle with ALE is 3 clocks.
dra4:       equ   $02   ; data read cycle with ALE is 4 clocks.
dra5:       equ   $03   ; data read cycle with ALE is 5 clocks.

; Bus Timing Register Low (BTRL):
wrpuls1:    equ   $00   ; write pulse width is 1 clock.
wrpuls2:    equ   $80   ; write pulse width is 2 clocks.
holdmin:    equ   $00   ; data hold time is minimum.
holdlong:   equ   $40   ; data hold time is 1 clock.
ale05:      equ   $0    ; ALE width is 0.5 clocks.
ale15:      equ   $1    ; ALE width is 1.5 clocks.
cr1:        equ   $00   ; data read cycle without ALE is 1 clock.
cr2:        equ   $04   ; data read cycle without ALE is 2 clocks.
cr3:        equ   $08   ; data read cycle without ALE is 3 clocks.
cr4:        equ   $0C   ; data read cycle without ALE is 4 clocks.
cra2:       equ   $00   ; data read cycle with ALE is 2 clocks.
cra3:       equ   $01   ; data read cycle with ALE is 3 clocks.
cra4:       equ   $02   ; data read cycle with ALE is 4 clocks.
cra5:       equ   $03   ; data read cycle with ALE is 5 clocks.

; Port configuration registers (PnCFGA, PnCFGB):
pcfga_pp:   equ   $ff   ; port config reg a value for push–pull output.
pcfgb_pp:   equ   $ff   ; port config reg b value for push–pull output.
pcfga_qb:   equ   $ff   ; port config reg a value for quasi–bidirect
pcfgb_qb:   equ   $00   ; port config reg b value for quasi–bidirect
pcfga_od:   equ   $00   ; port config reg a value for open drain output.
pcfgb_od:   equ   $00   ; port config reg b value for open drain output.
pcfga_z:    equ   $00   ; port config reg a value for output off.
pcfgb_z:    equ   $ff   ; port config reg b value for output off.
p1cfga_bus: equ   $ff   ; port1 config reg a=quasi, but A3–A0=push–pull.
p1cfgb_bus: equ   $0f   ; port1 config reg b=quasi, but A3–A0=push–pull.
p3cfga_bus: equ   $ff   ; port3 config reg a=quasi, but RD,WR=push–pull.
p3cfgb_bus: equ   $c0   ; port3 config reg b=quasi, but RD,WR=push–pull.

$list
```

```
;=====================================================================
; Start code space:

   org      0                          ; System exception interrupts:
   dw       $8f00, Start               ; Reset PSW, vector.
   dw       $8f00, BreakVec            ; breakpoint PSW, vector.
   dw       $8f00, TraceVec            ; trace PSW, vector.
   dw       $8f00, StkOvfVec           ; stack overflow PSW, vector.
   dw       $8f00, Div0Vec             ; divide by 0 PSW, vector.
   dw       $8f00, URetiVec            ; user reti PSW, vector.

   org      $40                        ; TRAP 0 – 15 exceptions:
   dw       $8800, Trap0Vec            ; trap 0 PSW, vector.
   dw       $8800, Trap1Vec            ; trap 1 PSW, vector.
   dw       $8800, Trap2Vec            ; trap 2 PSW, vector.
   dw       $8800, Trap3Vec            ; trap 3 PSW, vector.
   dw       $8800, Trap4Vec            ; trap 4 PSW, vector.
   dw       $8800, Trap5Vec            ; trap 5 PSW, vector.
   dw       $8800, Trap6Vec            ; trap 6 PSW, vector.
   dw       $8800, Trap7Vec            ; trap 7 PSW, vector.
   dw       $8800, Trap8Vec            ; trap 8 PSW, vector.
   dw       $8800, Trap9Vec            ; trap 9 PSW, vector.
   dw       $8800, Trap10Vec           ; trap 10 PSW, vector.
   dw       $8800, Trap11Vec           ; trap 11 PSW, vector.
   dw       $8800, Trap12Vec           ; trap 12 PSW, vector.
   dw       $8800, Trap13Vec           ; trap 13 PSW, vector.
   dw       $8800, Trap14Vec           ; trap 14 PSW, vector.
   dw       $8800, Trap15Vec           ; trap 15 PSW, vector.

   org      $80                        ; Event interrupts:
   dw       $8900, ExtInt0Vec          ; external interrupt 0.
   dw       $8900, Timer0Vec           ; timer 0 interrupt.
   dw       $8900, ExtInt1Vec          ; external interrupt 1.
   dw       $8900, Timer1Vec           ; timer 1 interrupt.
   dw       $8900, Timer2Vec           ; timer 2 interrupt.
   org      $90
   dw       $8900, Rxd0Vec             ; Serial port 0 receive.
   dw       $8900, Txd0Vec             ; Serial port 0 transmit.
   dw       $8900, Rxd1Vec             ; Serial port 1 receive.
   dw       $8900, Txd1Vec             ; Serial port 1 transmit.

   org      $100                       ; Software interrupts:
   dw       $8100, SWI1Vec             ; SWI1
   dw       $8200, SWI2Vec             ; SWI2
   dw       $8300, SWI3Vec             ; SWI3
   dw       $8400, SWI4Vec             ; SWI4
   dw       $8500, SWI5Vec             ; SWI5
   dw       $8600, SWI6Vec             ; SWI6
   dw       $8700, SWI7Vec             ; SWI7


   org      $0120                      ; Start of executable code area.
BreakVec:
TraceVec:
StkOvfVec:
Div0Vec:
URetiVec:
Trap0Vec:
Trap1Vec:
Trap2Vec:
Trap3Vec:
Trap4Vec:
Trap5Vec:
Trap6Vec:
Trap7Vec:
Trap8Vec:
```

```
Trap9Vec:
Trap10Vec:
Trap11Vec:
Trap12Vec:
Trap13Vec:
Trap14Vec:
Trap15Vec:
ExtInt0Vec:
Timer0Vec:
ExtInt1Vec:
Timer1Vec:
Timer2Vec:
Rxd0Vec:
Txd0Vec:
Rxd1Vec:
Txd1Vec:
SWI1Vec:
SWI2Vec:
SWI3Vec:
SWI4Vec:
SWI5Vec:
SWI6Vec:
SWI7Vec:

   reti     ; Location to route interrupts/exceptions with no specific
            ; handler code. This could prevent lockup particularly due
            ; to an unexpected exception such as stack overflow if
            ; there was no vector or handler whatsoever.


;=====================================================================
; Beginning of initialization code.

Start:
   mov      R7, #$100                  ; initialize stack pointer.

   mov.b    scr,#cmoff+page0on+time4   ; Set up chip configuration.

   mov.b    wdcon,#wdoff               ; Turn off watchdog timer.
   mov.b    wfeed1,#$a5                ; Feed watchdog: use new config
   mov.b    wfeed2,#$5a

   mov.b    bcr,#waitd+bus16+adr20     ; Set up bus configuration.
   mov.b    btrh,#dw5+dwa5+dr4+dra5    ; Config bus timing to longest
                                       ; bus cycles
   mov.b    btrl,#wrpuls2+holdmin+ale05+cr4+cra5 ; short ALE, min data
                                       ; hold.

   mov.b    p0cfga,#pcfga_pp           ; Configure port0 types for bus.
   mov.b    p0cfgb,#pcfgb_pp
   mov.b    p1cfga,#p1cfga_bus         ; Configure P1 for quasi-bidirec
   mov.b    p1cfgb,#p1cfgb_bus         ; except A3 - A0 are push-pull.
   mov.b    p2cfga,#pcfgb_pp           ; Configure P2 types for bus.
   mov.b    p2cfgb,#pcfgb_pp
   mov.b    p3cfga,#p3cfga_bus         ; Configure P3 for quasi-bidirect
   mov.b    p3cfgb,#p3cfgb_bus         ; except WR, RD are push-pull.


; End of initialization, begin user code.


end
```

APPENDIX 2: COMPARE/BRANCH SUMMARY

XA Compare/Branch Operations are summarized in the following tables:

Compare op: (destination – source)

Comparison: destination to source

op	CONDITION(S)	JUMP IF...	ALTERNATE: JUMP IF...
BG	(C OR Z) = 0	above, *unsigned*	not below nor equal, *unsigned*
BL	(C OR Z) = 1	below or equal, *unsigned*	not above, *unsigned*
BCC	C = 0	above or equal, *unsigned*	not below, *unsigned*
BCS	C = 1	below, *unsigned*	not above nor equal, *unsigned*
BGT	((N XOR V) OR Z) = 0	greater, *signed*	not less or equal, *signed*
BGE	(N XOR V) = 0)	greater or equal, *signed*	not less, *signed*
BLT	(N XOR V) = 1	less, *signed*	not greater nor equal, *signed*
BLE	((N XOR V) OR Z) = 1	less or equal, *signed*	not greater, *signed*
BCS	C = 1	carry	
BCC	C = 0	not carry	
BEQ	Z = 1	zero	equal
BNE	Z = 0	not zero	not equal
BOV	V = 1	overflow	
BNV	V= 0	not overflow	
BPL	N = 0	not sign	positive
BMI	N = 1	sign	negative

XA FLAG	MEANING
“Z”	Zero Flag
”C”	Carry Flag
”V”	Overflow Flag
”N”	Sign Flag

NOTE: XA PSW51.P is a bit-testable flag that indicates parity (=1 indicates even parity).

Following are some parallel examples of compare trees done first with 8051 CJNE operations, then with XA CJNE operations (directly translated), and XA CMP instructions followed by branches. These examples are illustrative and not intended to be exhaustive

EXAMPLE 1

8051 CJNE

```
       CJNE       dest,src,NOT_SAME              ; branch if dest ≠ src
       JMP        SAME                           ; dest = source
NOT_SAME:
       JC         DEST_SMALLER                   ; dest ≠ src
SRC_SMALLER:
       ...                                       ; dest > src
       JMP DONE
DEST_SMALLER:
       ...                                       ; dest < src
       JMP DONE
SAME:
       ...                                       ; dest = source
DONE:
       ...
```

XA CJNE

```
       CJNE       dest,src,NOT_SAME              ; branch if dest ≠ src
       JMP        SAME                           ; dest = source
NOT_SAME:
       BCS        DEST_SMALLER                   ; dest ≠ src
SRC_SMALLER:
       ...                                       ; dest > src
       BR         DONE

DEST_SMALLER:                                    ; dest < src
       ...
       BR         DONE
SAME:
       ...                                       ; dest = source
```

XA CMP/Bxx

```
       CMP        dest, src                      ; compare dest to src
       BEQ        SAME                           ; branch if dest = source
NOT_SAME:
       BCS        DEST_SMALLER                   ; branch if dest < src
SRC_SMALLER:
       ...                                       ; dest > src
       BR         DONE

DEST_SMALLER:                                    ; dest < src
       ...
       BR         DONE
SAME:
       ...                                       ; dest = source
DONE:
       ...
```

EXAMPLE 2

8051 CJNE

```
       CJNE       dest,src,NOT_SAME              ; branch if dest ≠ src
       JMP        SAME                           ; dest = source
NOT_SAME:
       JNC        SRC_SMALLER                    ; dest ≠ src
DEST_SMALLER:
       ...                                       ; dest < src
       JMP        DONE
SRC_SMALLER:
       ...                                       ; dest > src
       JMP        DONE
SAME:
       ...                                       ; dest = source
DONE:
       ...
```

XA CJNE

```
       CJNE       dest,src,NOT_SAME              ; branch if dest ≠ src
       JMP        SAME                           ; dest = source
NOT_SAME:
       BCC        SRC_SMALLER                    ; dest ≠ src
DEST_SMALLER:
       ...                                       ; dest < src
       BR         DONE
SRC_SMALLER:
       ...                                       ; dest > src
       BR         DONE
SAME:
       ...                                       ; dest = source
DONE:
       ...
```

XA CMP/Bxx

```
       CMP        dest, src                      ; compare dest to src
       BEQ        SAME                           ; branch if dest = source
NOT_SAME:
       BCC        SRC_SMALLER                    ; branch if dest < src
DEST_SMALLER:
       ...                                       ; dest < src
       BR         DONE
SRC_SMALLER:
       ...                                       ; dest > src
       BR DONE
SAME:
       ...                                       ; dest = source
DONE:
       ...
```

EXAMPLE 3

8051 CJNE

```
       CJNE       dest,src,GTE                ; branch if dest ≠ src
       JC         DEST_SMALLER                ; dest≠src; branch if dest < src
GTE:
       ...                                    ; dest >= src
       JMP        DONE
DEST_SMALLER:
       ...                                    ; dest < src

DONE:
       ...
```

XA CJNE

```
       CJNE       dest,src,NOT_SAME           ; branch if dest ≠ src
       BCS        DEST_SMALLER                ; dest≠src; branch if dest < src
GTE:
       ...                                    ; dest >= src
       BR         DONE
DEST_SMALLER:
       ...                                    ; dest < src
DONE:
       ...
```

XA CMP/Bxx

```
       CMP        dest, src                   ; compare dest to src
       BCS        DEST_SMALLER                ; branch if dest < src
GTE:
       ...                                    ; dest >= src
       BR         DONE
DEST_SMALLER:
       ...                                    ; dest < src
DONE:
       ...
```

EXAMPLE 4

8051

```
       CJNE       dest,src,LTE                ; branch if dest ≠ src
       JNC        SRC_SMALLER                 ; dest≠src; branch if dest > src
LTE:
       ...                                    ; dest <= src
       JM         DONE
SRC_SMALLER:
       ...                                    ; dest > src
DONE:
       ...
```

XA CJNE

```
       CJNE       dest,src,LTE                ; branch if dest ≠ src
       BG         SRC_SMALLER                 ; dest≠src; branch if dest > src
LTE:
       ...                                    ; dest <= src
       BR         DONE
SRC_SMALLER:
       ...                                    ; dest >src
DONE:
       ...
```

XA CMP/Bxx

```
       CMP        dest, src                   ; compare dest to src
       BG         SRC_SMALLER                 ; branch if dest > src
LTE:
       ...                                    ; dest <= src
       BR         DONE
SRC_SMALLER:
       ...                                    ; dest > src
DONE:
       ...
```

Translating 8051 assembly code to XA

AN708

DEFINITIONS

Data Sheet Identification	Product Status	Definition
Objective Specification	**Formative or in Design**	This data sheet contains the design target or goal specifications for product development. Specifications may change in any manner without notice.
Preliminary Specification	**Preproduction Product**	This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Product Specification	**Full Production**	This data sheet contains Final Specifications. Philips Semiconductors reserves the right to make changes at any time without notice, in order to improve design and supply the best possible product.

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

Let's make things better.

Philips Semiconductors

PHILIPS