Porting Inline Assembly from Legacy Code Generation Tools to ETEC

Overview

The ETEC compiler supports inline assembly, and in most cases the supported syntax matches that of the existing tools. However, there are some differences, and that is the focus of this document. The information in this document may also prove useful to someone writing new eTPU code in which they have a need to write portions in assembly. Assembler and assembly syntax details can be found in the ETEC Assembler Reference Manual.

The document begins with an overview of the inline assembly processing framework, followed by a section that focuses on C / assembly interfaces, and a section on assembly syntax differences in ETEC versus that found in legacy code.

References

ETEC Reference Manual
ETEC Assembler Reference Manual
ETEC Porting Guide

Inline Assembly Framework

When source code is processed through the ETEC C Compiler, assembly code within must be denoted one of two ways:

- a single line can be encapsulated with `#asm(` <assembly source> `)`.

```
#asm(  alu nil = mach - a, ccs. )
```

- multiple lines of assembly source can be denoted with a starting `#asm` (by itself on a line), followed by assembly source, and ending with a `#endasm` directive (by itself on a line).

```
#asm
PMSMVC_SL4S_NEG:
  alu nil = p23_16 low^ 0xE0, ccs8.
  if Z == 1 then goto PMSMVC_SL4S_OK, flush.
#endasm
```

The assembly source is pre-processed just like other portions of the source file – C comments (// and /* */) are stripped and macros replacement is performed. In general, it is best to use the #asm / #endasm syntax when more than a few consecutive lines of assembly are required. In all cases - #asm, #endasm, #asm() – the text of these can be placed anywhere in the source text as long as it is inside a C function scope – these directives do not need to be the first non-white space text on a source line.

When ETEC encounters a section of source code demarcated by `#asm` / `#endasm`, or one or more consecutive `#asm( )` statements, it combines as many of such statements as possible (if any C tokens intervene, combining ends) and has the ETEC Assembler process the segment of assembly.

ETEC does support use of #asm, etc. directly in function-like macros, such as:

```
#define SUB_48( a, b )\
{\
#asm\
    alu macl = macl - a, ccs.\
    alu mach = mach - b - C.\
#endasm\
}
```

However, this is only because when ETEC runs the C preprocessor, it does so in ETPUC mode, which allows the non-ANSI C construct above to work (#asm, #endasm are each treated as one token). In function-like macros, the '#' character is a special "stringification" operator, and must be followed by a parameter, so the ANSI-compatible way to write the above is:

```
#define hash_asm #asm
#define hash_endasm #endasm

#define SUB_48( a, b )\
{\
hash_asm\
    alu macl = macl - a, ccs.\
    alu mach = mach - b - C.\
hash_endasm\
}
```

One final note: ETEC supports preprocessor directives such as `#if`, `#ifdef`, etc. inside `#asm` / `#endasm` regions.

Optimization

Unless explicitly disabled, the ETEC optimizer will attempt to optimize inline assembly just like any other code. If this is not desired, optimization can be disabled by TBD. NOTE: At the current time, inline assembly is not optimized.

Assembly and C Code Interfaces

This section focuses on the interfaces between assembly code and C code. In particular, it focuses on differences between ETEC handling of such, versus other eTPU code generation tools.

Code Labels

ETEC treats inline assembly and C code labels interchangeably. C code labels have function scope, therefore the exact same label name can exist in two different C functions without conflict. Inline assembly labels work the same way. One side-effect of this is that whether a jump/goto is written in C or assembly, or whether the destination label is a C label or inline

assembly label, in all cases the destination must be within the current function scope. No jump/gotos can be made from one function to another.

To deal with this ETEC mangles C and inline assembly label names to ensure they are unique for linking purposes. Note: Pure assembly labels outside of an assembly scoping mechanism are treated as global.

Calling Assembly From C Code

Assembly routines that are called from C must have a "C function" wrapper, for example

```
fract24 mc_ctrl_pid( /*fract24 error,
                     mc_ctrl_pid_t *p_pid*/)
{
#asm
MC_CTRL_PID:
  /* Inputs:                                              */
  /* register a ...... error */
  /* register diob ... p_pid */

  /* Limit error to range <MIN24, MAX24>                  */
  if V == 0 then goto MC_CTRL_PID_I, flush.
  if N == 0 then goto MC_CTRL_PID_I, no_flush.
  alu a = 0x800000.

// ... more assembly code ...

#endasm
}
```

With such wrappers, it is typically important that no parameters are passed as that can result in the unintended consequence of the compiler generating a function prologue and epilogue. Exceptions to this rule may be made if a scratchpad programming model is enabled, or if parameters are being passed by named register. See the compiler reference manual for further details on these subjects.

These assembly functions can either be called from inline assembly:

```
#asm(call mc_ctrl_pid, no_flush.)
```

or called from C code (assuming in this case that the proper registers have been initialized):

```
Result = mc_ctrl_pid();
```

It is important to note that the above call references the function name, not the label placed at the beginning of the function – that label is not visible outside its function scope.

Below is an example of an assembly routine with a parameter passed via named register:

```
fract24 mc_saturate(registera x)
{
#asm
  /* Inputs: */
  /* register a  ...... x */
  if V == 0 then goto MC_SATUR_END, flush.
       /*---------*/
```



```
  alu a = max.
  return, no_flush.
  alu_if N == 1 then a = a - 1.
       /*---------*/
MC_SATUR_END:
  /* Outputs: */
  /* register a  ..... saturate(x) */
#endasm
}
```

and an example call to it (given the specialized assembly code, the argument must be the result of an operation that updates the V and N flags):

```
this.omega_field = mc_saturate(tmp + this.omega_actual);
```

Calling C Code From Assembly

Since C function names have global scope, assembly code can directly reference a C function from a call instruction. A user using such an operation would of course need to be careful that the assembly properly handled any EABI (call convention) issues.

Using "goto" Between C and Assembly Code

Jumps between C and inline assembly and C/assembly labels work interchangeably, although as mentioned earlier, such jumps must stay within the source function scope. As an example, C code "gotoing" an inline assembly label:

```
goto ASM_LABEL_1;
```

To jump from assembly to C, the same:

```
#asm( seq goto C_LABEL, flush. )
```

"Calling" Threads

When C code calls a function that has _eTPU_thread return type, and no parameters, it issues a jump opcode rather than a call opcode. This is useful for jumping to error handling code that will exit when it completes. For example, the standard error handling entry points are exposed in the standard library header file (eTpu_Lib.h):

```
_eTPU_thread _Error_handler_entry();
_eTPU_thread _Error_handler_scm_off_weeds();
_eTPU_thread _Error_handler_fill_weeds();
```

Condition Code Assumptions / Dependencies

ETEC does not enable condition code sampling on a simple assignment, only on arithmetic expressions. Thus inline assembly that is assuming condition codes are set may require a careful look when porting to ETEC. There may be cases where explicit code to sample the flags needs to be added (note that optimization may be able to remove this later).

```
#ifdef __ETEC__
   // need to test x (sample flags) - can't assume it is set
   #asm(  alu a = a, ccs.)
#endif
```


Variable Access

C code global, channel frame, and engine-relative variables (eTPU2) can be referenced symbolically from within inline assembly that is part of the proper scope. Additionally, an offset to the symbol can be specified, which is useful for accessing a member of a structure; see example below.

```
#asm(  ram diob <- inputs+5.          )  /* diob = inputs.B */
```

ETEC carefully checks that the byte address specified by the "symbol + offset" expression is legal given the type of memory access. For 8-bit and 32-bit memory accesses, the byte address specified must be double even, i.e. 0, 4, 8, etc. For a 24-bit access, the byte address must come out as double even + 1, i.e. 1, 5, 9, etc. If the address is not correct, the assembler/compiler will issue an error. Thus, in the example show above – for the inline assembly to be correct, the symbol `inputs` must be located at a double even address. Then, adding 5 to it nets a double even + 1 address which would be valid for the 24-bit access specified. If `inputs` is actually located at an address like 13, then the offset specified would have to be a multiple of 4; 5 would result in an error.

Complex Symbolic References

Symbolic references of the form <symbol name> + offset are allowed in both RAM instructions, and in ALU instructions that place the specified symbol's address (plus offset if any) into a register. In both cases the symbol name reference must point to a static symbol. The symbol can be static in global address space, channel frame address space, or engine address space (eTPU2). However, besides just <symbol name>, ETEC supports more complex expressions when the referenced symbol is of struct, union or array type. As in the simple case, the expression must result in a statically computable address. The supported operators are "." and "[]". Some examples are:

```
int24 g_arr[4];
int24 g_arr2d[2][2];
struct S1
{
   int x, y, z;
   int8 a, b, c;
};
struct S2
{
   struct S1 s1_arr[2];
   int24 m, n;
};
struct S1 g_s1;
struct S2 g_s2;
…
#asm
        // global var tests - array
        alu p = #8.
        ram p -> g_arr[2] + 4.
        alu p = #9.
        ram p -> g_arr2d[1][0].
        // global var tests - struct
```



```
        alu p31_24 = #13.
        ram p31_24 -> g_s1.c.
        alu p = #14.
        ram p -> g_s2.s1_arr[1].z.
#endasm
```

Use of these complex symbolic references are preferred whenever they replace "+offset" expressions, as it allows the ETEC toolset to properly and consistently calculate the symbol addresses.

Variable/Symbol Name Conflicts

When inline assembly is encountered, the compiler passes variable/symbolic information to the assembler so that it can properly deal with any symbolic accesses it encounters. However, this means that the C code must not contain symbol names that conflict with assembler keywords. For example, variable names matching register names (a, p, diob, etc.) or keywords (ram, seq, etc.) will trigger errors in the assembler and thus will not compile. Assembler reserved names must be avoided when any inline assembly is in use.

Raw Opcode Injection

ETEC and legacy tools support injection of raw opcodes via the `%hex <8-digit hex number>` syntax. However, this is not recommended as it reduces the amount of error checking that the assembler/compiler can do, and can result in link or run-time errors further down the pipe. Since ETEC fully supports the eTPU architecture with the assembly syntax, `%hex` source code should be converted to the proper assembly when processing through ETEC. For example:

```
#ifdef __ETEC__
  #asm( alu a =<< mach; ram p <- asm_omega_actual. )
  #asm( mdu a mults -p. )
#else
  #asm( /*alu a =<< mach; ram p <- asm_omega_actual.*/ %hex
B3C70BAB. )
  #asm( /*mdu a mults -p.*/ %hex 3C19FFE8. )
#endif
```

Another reason for converting raw opcodes to assembly is that it avoids hard-coded addresses. In the above example, if the address of asm_omega_actual changes the opcode %hex B3C70BAB must be carefully changed. This makes such code a challenge to maintain.

Assembly Syntax

The ETEC Assembler syntax is compatible with the existing format used by legacy tools in all but a few cases. Most of these cases are situations where in the past multiple syntaxes have existed for the same function, and ETEC has settled on one consistent solution. In a few cases no syntax compatible between ETEC and legacy tools exist. All the known incompatibilities will be listed below.

Reference the ASH WARE assembler reference manual to determine the supported syntax and note that although ASH WARE adopted a more minimal supported syntax, in almost all cases the

documented syntax will assemble under both ASH WARE and Byte Craft Limited tools, and therefore it is possible to develop inline assembly that is supported by both tools.

Memory Access Size Qualifier

Legacy tools support size qualifiers `p_access_msb` and `p_access_32` on memory accesses. However, this information is superfluous when the source / destination register is properly designated, and is in fact confusing when it conflicts with the source or destination register specified. ETEC requires that the memory access size be controlled by the source or destination register.

Memory Access Size	Source / Destination Registers
8 bits	p31_24
24 bits	p, diob
32 bits	p31_0

Code such as

```
ram p -> by diob, p_access_msb.  // direction = p_msb
```

must be converted to

```
ram p31_24 -> by diob.     // direction = p_msb
```

Indirect Memory Access

ETEC only supports the `by diob` or `by_diob` syntax for indirect memory accesses. The `(diob)` syntax is not supported. Assembly code of the form:

```
ram p31_0 = (diob).
```

must be modified to:

```
ram p31_0 <- by diob.
```

MDU Condition Code Sampling

Specifying that condition codes be sampled on an MDU operation has no effect. The MDU condition codes are always sampled on an MDU operation, and the ALU condition code flags are unaffected. Thus ETEC does not accept specifying condition code sampling on an MDU operation, thereby ensuring that the assembly is clear and understandable. An instruction such as:

```
#asm(  mdu d mults a, ccs. )   /* u_ab * inv_mod_index */
```

must be changed to:

```
#asm(  mdu d mults a. )        /* u_ab * inv_mod_index */
```

No Flush Syntax

ETEC supports the keyword `no_flush`. Legacy tools support another variation – `noflush` – and perhaps others. Such code must be changed from:

```
if N == 0 then goto MC_ELIM_2, noflush.
```

to:

```
if N == 0 then goto MC_ELIM_2, no_flush.
```

P Register Naming Convention

The p register, and portions of it accessible to the ALU, have had two set sets of names in the past. ETEC supports one set of names – those that contain the accessed bits as part of the name, which makes the meaning unambiguous.

ETEC Supported Name	Previous Names
p (default, bits 23_0)	p
p7_0	p_low, p7_0
p15_8	p_mid, p15_8
p23_16	p_high, p23_16
p31_24	p_msb, p31_24
p15_0	p_l16, p15_0
p31_16	p_h16, p31_16
p31_0	p, p_access_32, p31_0

Thus, code such as:

```
#asm(  alu sr =<< p_high + 0x00. ) /* sr = address offset (8:2) */
#asm(  alu a = p_l16.            ) /* a = interpolation argument */
#asm(  alu p = p_h16 - sr.       ) /* p = table step */
```

would need to be changed to:

```
#asm
  alu sr =<< p23_16 + 0x00.       /* sr = address offset (8:2) */
  alu a = p15_0.                  /* a = interpolation argument */
  alu p = p31_16 - sr.            /* p = table step */
#endasm
```

The updated version of code is compatible with both ETEC and legacy tools.

24-bit Immediate with Return

The eTPU instruction set includes a format – A1 – that allows for the loading of a 24-bit immediate. It also allows for a return, but only with an assumed flush. To make this clear, ETEC requires the user to include the `flush` keyword with the `return` keyword. Thus existing code like:

```
#asm(  alu a = #0x7FFFFF; return. )
```

must be changed to:


```
#asm(  alu a = #0x7FFFF; return, flush. )
```

Note that including the `flush` keyword fails to compile with some versions of legacy tools.

Multi-bit Shift

The previous syntax for multi-bit shift sub-instructions took the form of:

```
alu a = diob << sr.   // Shift left
alu a = diob >> sr.   // Shift right
alu a = diob >>R sr.  // Rotate right
```

The describing syntax is in conflict with how the hardware actually works, and is confusing. ETEC does not support this incorrect syntax. The multi-bit shift is actually by $2^{<\text{reg val}>+1}$ bits, bot by <reg val> bits as the legacy syntax seems to indicate. The new syntax makes this clear:

```
alu a = diob <<   (2^^(sr+1));;  // Shift left
alu a = diob >>   (2^^(sr+1));;  // Shift right
alu a = diob >>R  (2^^(sr+1));;  // Rotate right
```

When porting code, multi-bit sub-instructions must be changed to the new syntax.

