22 - EFI Byte Code Virtual Machine

This section defines an EFI Byte Code (EBC) Virtual Machine that can provide platform- and processor-independent mechanisms for loading and executing EFI device drivers.

22.1 Overview

The current design for option ROMs that are used in personal computer systems has been in place since 1981. Attempts to change the basic design requirements have failed for a variety of reasons. The EBC Virtual Machine described in this chapter is attempting to help achieve the following goals:

- Abstract and extensible design
- Processor independence
- OS independence
- Build upon existing specifications when possible
- Facilitate the removal of legacy infrastructure
- Exclusive use of EFI Services

One way to satisfy many of these goals is to define a pseudo or virtual machine that can interpret a predefined instruction set. This will allow the virtual machine to be ported across processor and system architectures without changing or recompiling the option ROM. This specification defines a set of machine level instructions that can be generated by a C compiler.

The following sections are a detailed description of the requirements placed on future option ROMs.

22.1.1 Processor Architecture Independence

Option ROM images shall be independent of supported 32-bit and supported 64-bit architectures. In order to abstract the architectural differences between processors option ROM images shall be EBC. This model is presented below:

- 64-bit C source code
- The EFI EBC image is the flashed image
- The system BIOS implements the EBC interpreter
- The interpreter handles 32 vs. 64 bit issues

Current Option ROM technology is processor dependent and heavily reliant upon the existence of the PC-AT infrastructure. These dependencies inhibit the evolution of both hardware and software under the veil of “backward compatibility.” A solution that isolates the hardware and support infrastructure through abstraction will facilitate the uninhibited progression of technology.

22.1.2 OS Independent

Option ROMs shall not require or assume the existence of a particular OS.

22.1.3 EFI Compliant

Option ROM compliance with EFI requires (but is not limited to) the following:

- Little endian layout
- Single-threaded model with interrupt polling if needed
- Where EFI provides required services, EFI is used exclusively. These include:
  - Console I/O
  - Memory Management
  - Timer services
  - Global variable access
- When an Option ROM provides EFI services, the EFI specification is strictly followed:
  - Service/protocol installation
  - Calling conventions
  - Data structure layouts
  - Guaranteed return on services

22.1.4 Coexistence of Legacy Option ROMs

The infrastructure shall support coexistent Legacy Option ROM and EBC Option ROM images. This case would occur, for example, when a Plug and Play Card has both Legacy and EBC Option ROM images flashed. The details of the mechanism used to select which image to load is beyond the scope of this document. Basically, a legacy System BIOS would not recognize an EBC Option ROM and therefore would never load it. Conversely, an EFI Firmware Boot Manager would only load images that it supports.

The EBC Option ROM format must utilize a legacy format to the extent that a Legacy System BIOS can:

- Determine the type of the image, in order to ignore the image. The type must be incompatible with currently defined types.
- Determine the size of the image, in order to skip to the next image.

22.1.5 Relocatable Image

An EBC option ROM image shall be eligible for placement in any system memory area large enough to accommodate it.

Current option ROM technology requires images to be shadowed in system memory address range 0xC0000 to 0xEFFFF on a 2048 byte boundary. This dependency not only limits the number of Option ROMs, it results in unused memory fragments up to 2 KiB.

22.1.6 Size Restrictions Based on Memory Available

EBC option ROM images shall not be limited to a predetermined fixed maximum size.

Current option ROM technology limits the size of a preinitialization option ROM image to 128 KiB (126 KiB actual). Additionally, in the DDIM an image is not allowed to grow during initialization. It is inevitable that 64-bit solutions will increase in complexity and size. To avoid revisiting this issue, EBC option ROM size is only limited by available system memory. EFI memory allocation services allow device drivers to claim as much memory as they need, within limits of available system memory.

The PCI specification limits the size of an image stored in an option ROM to 16 MB. If the driver is stored on the hard drive then the 16MB option ROM limit does not apply. In addition, the PE/COFF object format limits the size of images to 2 GB.

22.2 Memory Ordering

The term memory ordering refers to the order in which a processor issues reads (loads) and writes (stores) out onto the bus to system memory. The EBC Virtual Machine enforces strong memory ordering, where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

22.3 Virtual Machine Registers

The EBC virtual machine utilizes a simple register set. There are two categories of VM registers: general purpose registers and dedicated registers. All registers are 64-bits wide. There are eight (8) general-purpose registers (**R0**-**R7**), which are used by most EBC instructions to manipulate or fetch data. Table 149 lists the general-purpose registers in the VM and the conventions for their usage during execution.

Table 149. General Purpose VM Registers

Index	Register	Description
0	R0	Points to the top of the stack
1-3	R1-R3	Preserved across calls
4-7	R4-R7	Scratch, not preserved across calls

Register **R0** is used as a stack pointer and is used by the CALL, RET, PUSH, and POP instructions. The VM initializes this register to point to the incoming arguments when an EBC image is started or entered. This register may be modified like any other general purpose VM register using EBC instructions. Register **R7** is used for function return values.

Unlike the general-purpose registers, the VM dedicated registers have specific purposes. There are two dedicated registers: the instruction pointer (**IP**), and the flags (**Flags**) register. Specialized instructions provide access to the dedicated registers. These instructions reference the particular dedicated register by its assigned index value. Table 150 lists the dedicated registers and their corresponding index values.

Table 150. Dedicated VM Registers

Index	Register	Description		
0	FLAGS			
			Bit	Description
			0	C = Condition code
			1	SS = Single step
			2..63	Reserved
1	IP	Points to current instruction		
2..7	Reserved	Not defined		

The VM **Flags** register contains VM status and context flags. Table 151 lists the descriptions of the bits in the **Flags** register.

Table 151. VM Flags Register

Bit	Flag	Description
0	C	Condition code. Set to 1 if the result of the last compare was true, or set to 0 if the last compare was false. Used by conditional JMP instructions.
1	S	Single-step. If set, causes the VM to generate a single-step exception after executing each instruction. The bit is not cleared by the VM following the exception.
2..63	-	Reserved

The VM **IP** register is used as an instruction pointer and holds the address of the currently executing EBC instruction. The virtual machine will update the **IP** to the address of the next instruction on completion of the current instruction, and will continue execution from the address indicated in **IP**. The **IP** register can be moved into any general-purpose register (**R0**-**R7**). Data manipulation and data movement instructions can then be used to manipulate the value. The only instructions that may modify the **IP** are the `JMP`, `CALL`, and `RET` instructions. Since the instruction set is designed to use words as the minimum instruction entity, the low order bit (bit 0) of **IP** is always cleared to 0. If a JMP, CALL, or RET instruction causes bit 0 of **IP** to be set to 1, then an alignment exception occurs.

22.4 Natural Indexing

The natural indexing mechanism is the critical functionality that enables EBC to be executed unchanged on 32- or 64-bit systems. Natural indexing is used to specify the offset of data relative to a base address. However, rather than specifying the offset as a fixed number of bytes, the offset is encoded in a form that specifies the actual offset in two parts: a constant offset, and an offset specified as a number of natural units (where one natural unit = sizeof (VOID *)). These two values are used to compute the actual offset to data at runtime. When the VM decodes an index during execution, the resultant offset is computed based on the natural processor size. The encoded indexes themselves may be 16, 32, or 64 bits in size. Table 152 describes the fields in a natural index encoding.

Table 152. Index Encoding

Bit #	Description
N	Sign bit (sign), most significant bit
N-3..N-1	Bits assigned to natural units (w)
A..N-4	Constant units (c)
0..A-1	Natural units (n)

As shown in Table 152, for a given encoded index, the most significant bit (bit N) specifies the sign of the resultant offset after it has been calculated. The sign bit is followed by three bits (N-3..N-1) that are used to compute the width of the natural units field (n). The value (w) from this field is multiplied by the index size in bytes to determine the actual width (A) of the natural units field (n). Once the width of the natural units field has been determined, then the natural units (n) and constant units (c) can be extracted. The offset is then calculated at runtime according to the following equation:

Offset = (c + n * (sizeof (VOID *))) * sign

The following sections describe each of these fields in more detail.

22.4.1 Sign Bit

The sign bit determines the sign of the index once the offset calculation has been performed. All index computations using “n” and “c” are done with positive numbers, and the sign bit is only used to set the sign of the final offset computed.

22.4.2 Bits Assigned to Natural Units

This 3-bit field that is used to determine the width of the natural units field. The units vary based on the size of the index according to Table 153. For example, for a 16-bit index, the value contained in this field would be multiplied by 2 to get the actual width of the natural-units field.

Table 153. Index Size in Index Encoding

Index Size	Units
16 bits	2 bits
32 bits	4 bits
64 bits	8 bits

22.4.3 Constant

The constant is the number of bytes in the index that do not scale with processor size. When the index is a 16-bit value, the maximum constant is 4095. This index is achieved when the bits assigned to natural units is 0.

22.4.4 Natural Units

Natural units are used when a structure has fields that can vary with the architecture of the processor. Fields that precipitate the use of natural units include pointers and EFI INTN and UINTN data types. The size of one pointer or INTN/UINTN equals one natural unit. The natural units field in an index encoding is a count of the number of natural fields whose sizes (in bytes) must be added to determine a field offset.

As an example, assume that a given EBC instruction specifies a 16-bit index of 0xA048. This breaks down into:

- Sign bit (bit 15) = 1 (negative offset)
- Bits assigned to natural units (w, bits 14-12) = 2. Multiply by index size in bytes = 2 x 2 = 4 (A)
- c = bits 11-4 = 4
- n = bits 3-0 = 8

On a 32-bit machine, the offset is then calculated to be:

- Offset = (4 + 8 * 4) * -1 = -36
- On a 64-bit machine, the offset is calculated to be:
- Offset = (4 + 8 * 8) * -1 = -68

22.5 EBC Instruction Operands

The VM supports an EBC instruction set that performs data movement, data manipulation, branching, and other miscellaneous operations typical of a simple processor. Most instructions operate on two operands, and have the general form:

```
INSTRUCTION Operand1, Operand2
```

Typically, instruction operands will be one of the following:

- Direct
- Indirect
- Indirect with index
- Immediate

The following subsections explain these operands.

22.5.1 Direct Operands

When a direct operand is specified for an instruction, the data to operate upon is contained in one of the VM general-purpose registers **R0**-**R7**. Syntactically, an example of direct operand mode could be the ADD instruction:

```
ADD64 R1, R2
```

This form of the instruction utilizes two direct operands. For this particular instruction, the VM would take the contents of register **R2**, add it to the contents of register **R1**, and store the result in register **R1**.

22.5.2 Indirect Operands

When an indirect operand is specified, a VM register contains the address of the operand data. This is sometimes referred to as register indirect, and is indicated by prefixing the register operand with "@." Syntactically, an example of an indirect operand mode could be this form of the ADD instruction:

```
ADD32 R1, @R2
```

For this instruction, the VM would take the 32-bit value at the address specified in **R2**, add it to the contents of register **R1**, and store the result in register **R1**.

22.5.3 Indirect with Index Operands

When an indirect with index operand is specified, the address of the operand is computed by adding the contents of a register to a decoded natural index that is included in the instruction. Typically with indexed addressing, the base address will be loaded in the register and an index value will be used to indicate the offset relative to this base address. Indexed addressing takes the form

$@R_1$ (+n,+c)

where:

- $\mathbf{R_1}$ is one of the general-purpose registers (**R0**-**R7**) which contains the base address
- **+n** is a count of the number of "natural" units offset. This portion of the total offset is computed at runtime as **(**n * sizeof (VOID *))

- **+c** is a byte offset to add to the natural offset to resolve the total offset

The values of **n** and **c** can be either positive or negative, though they must both have the same sign. These values get encoded in the indexes associated with EBC instructions as shown in Table 152. Indexes can be 16-, 32-, or 64-bits wide depending on the instruction. An example of indirect with index syntax would be:

```
ADD32 R1, @R2 (+1, +8)
```

This instruction would take the address in register **R2**, add (8 + 1 * sizeof (VOID *)), read the 32-bit value at the address, add the contents of **R1** to the value, and store the result back to **R1**.

22.5.4 Immediate Operands

Some instructions support an immediate operand, which is simply a value included in the instruction encoding. The immediate value may or may not be sign extended, depending on the particular instruction. One instruction that supports an immediate operand is MOVI. An example usage of this instruction is:

```
MOVIww R1, 0x1234
```

This instruction moves the immediate value 0x1234 directly into VM register **R1**. The immediate value is contained directly in the encoding for the MOVI instruction.

22.6 EBC Instruction Syntax

Most EBC instructions have one or more variations that modify the size of the instruction and/or the behavior of the instruction itself. These variations will typically modify an instruction in one or more of the following ways:

- The size of the data being operated upon
- The addressing mode for the operands
- The size of index or immediate data
- To represent these variations syntactically in this specification the following conventions are used:
- Natural indexes are indicated with the “Index” keyword, and may take the form of “Index16,” “Index32,” or “Index64” to indicate the size of the index value supported. Sometimes the form Index16|32|64 is used here, which is simply a shorthand notation for Index16|Index32|Index64. A natural index is encoded per Table 152 and is resolved at runtime.
- Immediate values are indicated with the “Immed” keyword, and may take the form of “Immed16,” “Immed32,” or “Immed64” to indicate the size of the immediate value supported. The shorthand notation Immed16|32|64 is sometimes used when different size immediate values are supported.
- Terms in brackets [ ] are required.
- Terms in braces { } are optional.
- Alternate terms are separated by a vertical bar |.
- The form R_1 and R_2 represent Operand 1 register and Operand 2 register respectfully, and can typically be any VM general-purpose register **R0**-**R7**.

- Within descriptions of the instructions, brackets [ ] enclosing a register and/or index indicate that the contents of the memory pointed to by the enclosed contents are used.

22.7 Instruction Encoding

Most EBC instructions take the form:

```
INSTRUCTION R1, R2 Index|Immed
```

For those instructions that adhere to this form, the binary encoding for the instruction will typically consist of an opcode byte, followed by an operands byte, followed by two or more bytes of immediate or index data. Thus the instruction stream will be:

```
(1 Byte Opcode) + (1 Byte Operands) + (Immediate data|Index data)
```

22.7.1 Instruction Opcode Byte Encoding

The first byte of an instruction is the opcode byte, and an instruction’s actual opcode value consumes 6 bits of this byte. The remaining two bits will typically be used to indicate operand sizes and/or presence or absence of index or immediate data. Table 154 defines the bits in the opcode byte for most instructions, and their usage.

Table 154. Opcode Byte Encoding

Bit	Sym	Description
6..7	Modifiers	One or more of: Index or immediate data present/absent Operand size Index or immediate data size
0..5	Op	Instruction opcode

For those instructions that use bit 7 to indicate the presence of an index or immediate data and bit 6 to indicate the size of the index or immediate data, if bit 7 is 0 (no immediate data), then bit 6 is ignored by the VM. Otherwise, unless otherwise specified for a given instruction, setting unused bits in the opcode byte results in an instruction encoding exception when the instruction is executed. Setting the modifiers field in the opcode byte to reserved values will also result in an instruction encoding exception.

22.7.2 Instruction Operands Byte Encoding

The second byte of most encoded instructions is an operand byte, which encodes the registers for the instruction operands and whether the operands are direct or indirect. Table 155 defines the encoding for the operand byte for these instructions. Unless otherwise specified for a given instruction, setting unused bits in the operand byte results in an instruction encoding exception when the instruction is executed. Setting fields in the operand byte to reserved values will also result in an instruction encoding exception.

Table 155. Operand Byte Encoding

Bit	Description
7	0 = Operand 2 is direct 1 = Operand 2 is indirect

4..6	Operand 2 register
3	0 = Operand 1 is direct 1 = Operand 1 is indirect
0..2	Operand 1 register

22.7.3 Index/Immediate Data Encoding

Following the operand bytes for most instructions is the instruction's immediate data. The immediate data is, depending on the instruction and instruction encoding, either an unsigned or signed literal value, or an index encoded using natural encoding. In either case, the size of the immediate data is specified in the instruction encoding.

For most instructions, the index/immediate value in the instruction stream is interpreted as a signed immediate value if the register operand is direct. This immediate value is then added to the contents of the register to compute the instruction operand. If the register is indirect, then the data is usually interpreted as a natural index (see Section 22.4) and the computed index value is added to the contents of the register to get the address of the operand.

22.8 EBC Instruction Set

The following sections describe each of the EBC instructions in detail. Information includes an assembly-language syntax, a description of the instruction functionality, binary encoding, and any limitations or unique behaviors of the instruction.

ADD

Syntax

```
ADD[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Adds two signed operands and stores the result to Operand 1. The operation can be performed on either 32-bit (ADD32) or 64-bit (ADD64) operands.

Operation

```
Operand 1 <= Operand 1 + Operand 2
```

Table 156. ADD Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0C

1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the R_2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is ADD32 and Operand 1 is direct, then the result is stored back to the Operand 1 register with the upper 32 bits cleared.

AND

Syntax

```
AND[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Performs a logical AND operation on two operands and stores the result to Operand 1. The operation can be performed on either 32-bit (AND32) or 64-bit (AND64) operands.

Operation

```
Operand 1 <= Operand 1 AND Operand 2
```

Table 157. AND Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x14

1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is AND32 and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

ASHR

Syntax

```
ASHR[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Performs an arithmetic right-shift of a signed 32-bit (ASHR32) or 64-bit (ASHR64) operand and stores the result back to Operand 1

Operation

```
Operand 1 <= Operand 1 SHIFT-RIGHT Operand 2
```

Table 158. ASHR Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x19

1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2+ Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is ASHR32, and Operand 1 is direct, then the result is stored back to the Operand 1 register with the upper 32 bits cleared.

BREAK

Syntax

```
BREAK [break code]
```

Description

The BREAK instruction is used to perform special processing by the VM. The break code specifies the functionality to perform.

BREAK 0 – Runaway program break. This indicates that the VM is likely executing code from cleared memory. This results in a bad break exception.

BREAK 1 – Get virtual machine version. This instruction returns the 64-bit virtual machine revision number in VM register **R7**. The encoding is shown in Table 159 and Table 160. A VM that conforms to this version of the specification should return a version number of 0x00010000.

Table 159. VM Version Format

Bits	**Description**
63-32	Reserved = 0
31..16	VM major version
15..0	VM minor version

BREAK 3 – Debug breakpoint. Executing this instruction results in a debug break exception. If a debugger is attached or available, then it may halt execution of the image.

BREAK 4 – System call. There are no system calls supported for use with this break code, so the VM will ignore the instruction and continue execution at the following instruction.

BREAK 5 – Create thunk. This causes the interpreter to create a thunk for the EBC entry point whose 32-bit IP-relative offset is stored at the 64-bit address in VM register **R7**. The interpreter then replaces the contents of the memory location pointed to by **R7** to point to the newly created thunk. Since all EBC IP-relative offsets are relative to the next instruction or data object, the original offset is off by 4, so must be incremented by 4 to get the actual address of the entry point.

BREAK 6 – Set compiler version. An EBC C compiler can insert this break instruction into an executable to set the compiler version used to build an EBC image. When the VM executes this instruction it takes the compiler version from register **R7** and may perform version compatibility checking. The compiler version number follows the same format as the VM version number returned by the BREAK 1 instruction.

Table 160. BREAK Instruction Encoding

Byte	Description
0	Opcode = 0x00
1	0 = Runaway program break 1 = Get virtual machine version 3 = Debug breakpoint 4 = System call 5 = Create thunk 6 = Set compiler version

Behaviors and Restrictions

- Executing an undefined BREAK code results in a bad break exception.
- Executing BREAK 0 results in a bad break exception.

CALL

Syntax

```
CALL32{EX}{a} {@}R1 {Immed32|Index32}
CALL64{EX}{a} Immed64
```

Description

The CALL instruction pushes the address of the following instruction on the stack and jumps to a subroutine. The subroutine may be either EBC or native code, and may be to an absolute or **IP**-relative address. CALL32 is used to jump directly to EBC code within a given application, whereas CALLEX is used to jump to external code (either native or EBC), which requires thunking. Functionally, the CALL does the following:

```
R0 = R0 - 8;
PUSH64 ReturnAddress
if (Opcode.ImmedData64Bit) {
 if (Operands.EbcCall) {
  IP = Immed64;
 } else {
  NativeCall (Immed64);
 }
} else {
 if (Operand1 != R0) {
  Addr = Operand1;
 } else {
  Addr = Immed32;
 }
 if (Operands.EbcCall) {
  if (Operands.RelativeAddress) {
   IP += Addr + SizeOfThisInstruction;
  } else {
   IP = Addr
  }
 } else {
  if (Operands.RelativeAddress) {
   NativeCall (IP + Addr)
  } else {
   NativeCall (Addr)
  }
 }
```

Operation

```
R0 <= R0 – 16
[R0] <= IP + SizeOfThisInstruction
IP <= IP + SizeOfThisInstruction + Operand 1 (relative CALL)
IP <= Operand 1 (absolute CALL)
```

Table 161. CALL Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index data absent 1 = Immediate/index data present
	6	0 = CALL32 with 32-bit immediate data/index if present 1 = CALL64 with 64-bit immediate data
	0..5	Opcode = 0x03

1	Bit	Description
	6..7	Reserved = 0
	5	0 = Call to EBC 1 = Call to native code
	4	0 = Absolute address 1 = Relative address
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..5	Optional 32-bit index/immediate for CALL32	
2..9	Required 64-bit immediate data for CALL64	

BEHAVIOR AND RESTRICTIONS

- For the CALL32 forms, if Operand 1 is indirect, then the immediate data is interpreted as an index, and the Operand 1 value is fetched from memory address [R_1 + Index32].
- For the CALL32 forms, if Operand 1 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 1 register contents such that Operand 1 = R_1 + Immed32.
- For the CALLEX forms, the VM must fix up the stack pointer and execute a call to native code in a manner compatible with the native code such that the callee is able to access arguments passed on the VM stack..
- For the CALLEX forms, the value returned by the callee should be returned in **R7**.
- For the CALL64 forms, the Operand 1 fields are ignored.
- If Byte7:Bit6 = 1 (CALL64), then Byte1:Bit4 is assumed to be 0 (absolute address)
- For CALL32 forms, if Operand 1 register = **R0**, then the register operand is ignored and only the immediate data is used in the calculation of the call address.
- Prior to the call, the VM will decrement the stack pointer **R0** by 16 bytes, and store the 64-bit return address on the stack.
- Offsets for relative calls are relative to the address of the instruction following the CALL instruction.

CMP

Syntax

CMP[32|64][eq|lte|gte|ulte|ugte] R_1, {@}R_2 {Index16|Immed16}

Description

The CMP instruction is used to compare Operand 1 to Operand 2. Supported comparison modes are =, <=, >=, unsigned <=, and unsigned >=. The comparison size can be 32 bits (CMP32) or 64 bits (CMP64). The effect of this instruction is to set or clear the condition code bit in the **Flags** register per the comparison results. The operands are compared as signed values except for the CMPulte and CMPugte forms.

Operation

```
CMPeq: Flags.C <= (Operand 1 == Operand 2)
CMPlte: Flags.C <= (Operand 1 <= Operand 2)
CMPgte: Flags.C <= (Operand 1 >= Operand 2)
CMPulte: Flags.C <= (Operand 1 <= Operand 2) (unsigned)
CMPugte: Flags.C <= (Operand 1>= Operand 2) (unsigned)
```

Table 162. CMP Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index data absent 1 = Immediate/index data present
	6	0 = 32-bit comparison 1 = 64-bit comparison
	0..5	Opcode
		0x05 = CMPeq compare equal 0x06 = CMPlte compare signed less then/equal 0x07 = CMPgte compare signed greater than/equal 0x08 = CMPulte compare unsigned less than/equal 0x09 = CMPugte compare unsigned greater than/equal
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	Reserved = 0
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R_2 + Immed16.
- Only register direct is supported for Operand 1.

CMPI

Syntax

```
CMPI[32|64]{w|d}[eq|lte|gte|ulte|ugte] {@}R1 {Index16}, Immed16|Immed32
```

Description

Compares two operands, one of which is an immediate value, for =, <=, >=, unsigned <=, or unsigned >=, and sets or clears the condition flag bit in the **Flags** register accordingly. Comparisons can be performed on a 32-bit (CMPI32) or 64-bit (CMPI64) basis. The size of the immediate data can be either 16 bits (CMPIw) or 32 bits (CMPId).

Operation

```
CMPIeq: Flags.C <= (Operand 1 == Operand 2)
CMPIlte: Flags.C <= (Operand 1 <= Operand 2)
CMPIgte: Flags.C <= (Operand 1 >= Operand 2)
CMPIulte: Flags.C <= (Operand 1 <= Operand 2)
CMPIugte: Flags.C <= (Operand 1>= Operand 2)
```

Table 163. CMPI Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = 16-bit immediate data 1 = 32-bit immediate data
	6	0 = 32-bit comparison 1 = 64-bit comparison
	0..5	Opcode
		0x2D = CMPIeq compare equal 0x2E = CMPIlte compare signed less then/equal 0x2F = CMPIgte compare signed greater than/equal 0x30 = CMPIulte compare unsigned less than/equal 0x31 = CMPIugte compare unsigned greater than/equal
1	Bit	Description
	5..7	Reserved = 0
	4	0 = Operand 1 index absent 1 = Operand 1 index present
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index	
2..3/4..5	16-bit immediate data	
2..5/4..7	32-bit immediate data	

Behaviors and Restrictions

- The immediate data is fetched as a signed value.
- If the immediate data is smaller than the comparison size, then the immediate data is sign-extended appropriately.
- If Operand 1 is direct, and an Operand 1 index is specified, then an instruction encoding exception is generated.

DIV

Syntax

DIV[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

Description

Performs a divide operation on two signed operands and stores the result to Operand 1. The operation can be performed on either 32-bit (DIV32) or 64-bit (DIV64) operands.

Operation

```
Operand 1 <= Operand 1 / Operand 2
```

Table 164. DIV Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x10
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2+ Index16].
- If Operand 2 is direct, then the immediate data is considered a signed value and is added to the register contents such that Operand 2 = R_2 + Immed16

- If the instruction is DIV32 form, and Operand 1 is direct, then the upper 32 bits of the result are set to 0 before storing to the Operand 1 register.
- A divide-by-0 exception occurs if Operand 2 = 0.

DIVU

Syntax

```
DIVU[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Performs a divide operation on two unsigned operands and stores the result to Operand 1. The operation can be performed on either 32-bit (DIVU32) or 64-bit (DIVU64) operands.

Operation

```
Operand 1 <= Operand 1 / Operand 2
```

Table 165. DIVU Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x11
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the value is fetched from memory as an unsigned value at address [R_2+ Index16].
- If Operand 2 is direct, then the immediate data is considered an unsigned value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16
- For the DIVU32 form, if Operand 1 is direct then the upper 32 bits of the result are set to 0 before storing back to the Operand 1 register.
- A divide-by-0 exception occurs if Operand 2 = 0.

EXTNDB

Syntax

```
EXTNDB[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Sign-extend a byte value and store the result to Operand 1. The byte can be signed extended to 32 bits (EXTNDB32) or 64 bits (EXTNDB64).

Operation

```
Operand 1 <= (sign extended) Operand 2
```

Table 166. EXTNDB Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x1A
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the byte Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value, is added to the signed-extended byte from the Operand 2 register, and the byte result is sign extended to 32 or 64 bits.
- If the instruction is EXTNDB32 and Operand 1 is direct, then the 32-bit result is stored in the Operand 1 register with the upper 32 bits cleared.

EXTNDD

Syntax

`EXTNDD[32|64] {@}R`$_1$`, {@}R`$_2$` {Index16|Immed16}`

Description

Sign-extend a 32-bit Operand 2 value and store the result to Operand 1. The Operand 2 value can be extended to 32 bits (EXTNDD32) or 64 bits (EXTNDD64).

Operation

```
Operand 1 <= (sign extended) Operand 2
```

Table 167. EXTNDD Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x1C
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the 32-bit value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = R_2 + Immed16, and the value is sign extended to 32 or 64 bits accordingly.
- If the instruction is EXTNDD32 and Operand 1 is direct, then the result is stored in the Operand 1 register with the upper 32 bits cleared.

EXTNDW

Syntax

```
EXTNDW[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Sign-extend a 16-bit Operand 2 value and store the result back to Operand 1. The value can be signed extended to 32 bits (EXTNDW32) or 64 bits (EXTNDW64).

Operation

```
Operand 1 <= (sign extended) Operand 2
```

Table 168. EXTNDW Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x1B
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the word value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = R_2 + Immed16, and the value is sign extended to 32 or 64 bits accordingly.
- If the instruction is EXTNDW32 and Operand 1 is direct, then the 32-bit result is stored in the Operand 1 register with the upper 32 bits cleared.

JMP

Syntax

```
JMP32{cs|cc} {@}R1 {Immed32|Index32}
JMP64{cs|cc} Immed64
```

Description

The JMP instruction is used to conditionally or unconditionally jump to a relative or absolute address and continue executing EBC instructions. The condition test is done using the condition bit in the VM **Flags** register. The JMP64 form only supports an immediate value that can be used for either a relative or absolute jump. The JMP32 form adds support for indirect addressing of the JMP offset or address. The JMP is implemented as:

```
if (ConditionMet) {
 if (Operand.RelativeJump) {
  IP += Operand1 + SizeOfThisInstruction;
 } else {
  IP = Operand1;
 }
}
```

Operation

```
IP <= Operand 1 (absolute address)
IP <= IP + SizeOfThisInstruction + Operand 1 (relative address)
```

Table 169. JMP Instruction Encoding

Byte	Description	
0	Bit	Description
	7	0 = Immediate/index data absent 1 = Immediate/index data present
	6	0 = JMP32 1 = JMP64
	0..5	Opcode = 0x01
1	Bit	Description
	7	0 = Unconditional jump 1 = Conditional jump
	6	0 = Jump if **Flags.C** is clear (cc) 1 = Jump if **Flags.C** is set (cs)
	5	Reserved = 0
	4	0 = Absolute address 1 = Relative address
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..5	Optional 32-bit immediate data/index for JMP32	

Byte	Description
2..9	64-bit immediate data for JMP64

Behaviors and Restrictions

- Operand 1 fields are ignored for the JMP64 forms
- If the instruction is JMP32, and Operand 1 register = **R0**, then the register contents are assumed to be 0.
- If the instruction is JMP32, and Operand 1 is indirect, then the immediate data is interpreted as an index, and the jump offset or address is fetched as a 32-bit signed value from address [R_1 + Index32]
- If the instruction is JMP32, and Operand 1 is direct, then the immediate data is considered a signed immediate value such that Operand 1 = R_1 + Immed32
- If the jump is unconditional, then Byte1:Bit6 (condition) is ignored
- If the instruction is JMP64, and Byte0:Bit7 is clear (no immediate data), then an instruction encoding exception is generated.
- If the instruction is JMP32, and Operand 2 is indirect, then the Operand 2 value is read as a natural value from memory address [R_1 + Index32]
- An alignment check exception is generated if the jump is taken and the target address is odd.

JMP8

Syntax

```
JMP8{cs|cc} Immed8
```

Description

Conditionally or unconditionally jump to a relative offset and continue execution. The offset is a signed one-byte offset specified in the number of words. The offset is relative to the start of the following instruction.

Operation

```
IP = IP + SizeOfThisInstruction + (Immed8 * 2)
```

Table 170. JMP8 Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Unconditional jump 1 = Conditional jump
	6	0 = Jump if **Flags.C** is clear (cc) 1 = Jump if **Flags.C** is set (cs)
	0..5	Opcode = 0x02

1	Immediate data (signed word offset)

Behaviors and Restrictions

- If the jump is unconditional, then Byte0:Bit6 (condition) is ignored

LOADSP

Syntax

```
LOADSP [Flags], R₂
```

Description

This instruction loads a VM dedicated register with the contents of a VM general-purpose register **R0**-**R7**. The dedicated register is specified by its index as shown in Table 150.

Operation

```
Operand 1 <= R₂
```

Table 171. LOADSP Instruction Encoding

BYTE	Description	
0	Bit	Description
	6..7	Reserved = 0
	0..5	Opcode = 0x29
1	7	Reserved
	4..6	Operand 2 general purpose register
	3	Reserved
	0..2	Operand 1 dedicated register index

Behaviors and Restrictions

- Attempting to load any register (Operand 1) other than the **Flags** register results in an instruction encoding exception.
- Specifying a reserved dedicated register index results in an instruction encoding exception.
- If Operand 1 is the **Flags** register, then reserved bits in the **Flags** register are not modified by this instruction.

MOD

Syntax

```
MOD[32|64] {@}R₁, {@}R₂ {Index16|Immed16}
```

Description

Perform a modulus on two signed 32-bit (MOD32) or 64-bit (MOD64) operands and store the result to Operand 1.

Operation

```
Operand 1 <= Operand 1 MOD Operand 2
```

Table 172. MOD Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x12
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = R_2 + Immed16, and the value is sign extended to 32 or 64 bits accordingly.
- If Operand 2 = 0, then a divide-by-zero exception is generated.

MODU

Syntax

MODU[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

Description

Perform a modulus on two unsigned 32-bit (MODU32) or 64-bit (MODU64) operands and store the result to Operand 1.

Operation

```
Operand 1 <= Operand 1 MOD Operand 2
```

Table 173. MODU Instruction Encoding

BYTE	Description

0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x13
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered an unsigned immediate value such that Operand 2 = R_2 + Immed16.
- If Operand 2 = 0, then a divide-by-zero exception is generated.

MOV

Syntax

MOV[b|w|d|q]{w|d} {@}R_1 {Index16|32}, {@}R_2 {Index16|32}

MOVqq {@}R_1 {Index64}, {@}R_2 {Index64}

Description

This instruction moves data from Operand 2 to Operand 1. Both operands can be indexed, though both indexes are the same size. In the instruction syntax for the first form, the first variable character indicates the size of the data move, which can be 8 bits (b), 16 bits (w), 32 bits (d), or 64 bits (q). The optional character indicates the presence and size of the index value(s), which may be 16 bits (w) or 32 bits (d). The MOVqq instruction adds support for 64-bit indexes.

Operation

```
Operand 1 <= Operand 2
```

Table 174. MOV Instruction Encoding

Byte	Description	
0	Bit	Description
	7	0 = Operand 1 index absent 1 = Operand 1 index present
	6	0 = Operand 2 index absent 1 = Operand 2 index present
	0..5	0x1D = MOVbw opcode 0x1E = MOVww opcode 0x1F = MOVdw opcode 0x20 = MOVqw opcode 0x21 = MOVbd opcode 0x22 = MOVwd opcode 0x23 = MOVdd opcode 0x24 = MOVqd opcode 0x28 = MOVqq opcode
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional Operand 1 16-bit index	
2..3/4..5	Optional Operand 2 16-bit index	
2..5	Optional Operand 1 32-bit index	
2..5/6..9	Optional Operand 2 32-bit index	
2..9	Optional Operand 1 64-bit index (MOVqq)	
2..9/10..17	Optional Operand 2 64-bit index (MOVqq)	

Behaviors and Restrictions

- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.

MOVI

Syntax

```
MOVI[b|w|d|q][w|d|q] {@}R1 {Index16}, Immed16|32|64
```

Description

This instruction moves a signed immediate value to Operand 1. In the instruction syntax, the first variable character specifies the width of the move, which may be 8 bits (b), 16 bits (w), 32-bits (d), or 64 bits (q). The second variable character specifies the width of the immediate data, which may be 16 bits (w), 32 bits (d), or 64 bits (q).

Operation

```
Operand 1 <= Operand 2
```

Table 175. MOVI Instruction Encoding

BYTE	Description	
0	Bit	Description
	6..7	0 = Reserved 1 = Immediate data is 16 bits (w) 2 = Immediate data is 32 bits (d) 3 = Immediate data is 64 bits (q)
	0..5	Opcode = 0x37
1	Bit	Description
	7	Reserved = 0
	6	0 = Operand 1 index absent 1 = Operand 1 index present
	4..5	0 = 8 bit (b) move 1 = 16 bit (w) move 2 = 32 bit (d) move 3 = 64 bit (q) move
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit index	
2..3/4..5	16-bit immediate data	
2..5/4..7	32-bit immediate data	
2..9/4..11	64-bit immediate data	

Behaviors and Restrictions

- Specifying an index value with Operand 1 direct results in an instruction encoding exception.
- If the immediate data is smaller than the move size, then the value is sign-extended to the width of the move.
- If Operand 1 is a register, then the value is stored to the register with bits beyond the move size cleared.

MOVIn

Syntax

`MOVIn[w|d|q] {@}R`$_1$ `{Index16}, Index16|32|64`

Description

This instruction moves an indexed value of form (+n,+c) to Operand 1. The index value is converted from (+n, +c) format to a signed offset per the encoding described in Table 152. The size of the Operand 2 index data can be 16 (w), 32 (d), or 64 (q) bits.

Operation

```
Operand 1 <= Operand 2 (index value)
```

Table 176. MOVIn Instruction Encoding

BYTE	Description	
0	Bit	Description
	6..7	0 = Reserved 1 = Operand 2 index value is 16 bits (w) 2 = Operand 2 index value is 32 bits (d) 3 = Operand 2 index value is 64 bits (q)
	0..5	Opcode = 0x38
1	Bit	Description
	7	Reserved
	6	0 = Operand 1 index absent 1 = Operand 1 index present
	4..5	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index	
2..3/4..5	16-bit Operand 2 index	
2..5/4..7	32-bit Operand 2 index	
2..9/4..11	64-bit Operand 2 index	

Behaviors and Restrictions

- Specifying an Operand 1 index when Operand 1 is direct results in an instruction encoding exception.
- The Operand 2 index is sign extended to the size of the move if necessary.
- If the Operand 2 index size is smaller than the move size, then the value is truncated.
- If Operand 1 is direct, then the Operand 2 value is sign extended to 64 bits and stored to the Operand 1 register.

MOVn

Syntax

```
MOVn{w|d} {@}R1 {Index16|32}, {@}R2 {Index16|32}
```

Description

This instruction loads an unsigned natural value from Operand 2 and stores the value to Operand 1. Both operands can be indexed, though both operand indexes are the same size. The operand index(s) can be 16 bits (w) or 32 bits (d).

Operation

```
Operand1 <= (UINTN)Operand2
```

Table 177. MOVn Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 1 index absent 1 = Operand 1 index present
	6	0 = Operand 2 index absent 1 = Operand 2 index present
	0..5	0x32 = MOVnw opcode 0x33 = MOVnd opcode
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional Operand 1 16-bit index	
2..3/4..5	Optional Operand 2 16-bit index	
2..5	Optional Operand 1 32-bit index	
2..5/6..9	Optional Operand 2 32-bit index	

Behaviors and Restrictions

- If an index is specified for Operand 2, and Operand 2 register is direct, then the Operand 2 index value is added to the register contents such that Operand 2 = (UINTN)(R_2 + Index).
- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.
- If Operand 1 is direct, then the Operand 2 value will be 0-extended to 64 bits on a 32-bit machine before storing to the Operand 1 register.

MOVREL

Syntax

```
MOVREL[w|d|q] {@}R1 {Index16}, Immed16|32|64
```

Description

This instruction fetches data at an **IP**-relative immediate offset (Operand 2) and stores the result to Operand 1. The offset is a signed offset relative to the following instruction. The fetched data is unsigned and may be 16 (w), 32 (d), or 64 (q) bits in size.

Operation

```
Operand 1 <= [IP + SizeOfThisInstruction + Immed]
```

Table 178. MOVREL Instruction Encoding

BYTE	Description	
0	Bit	Description
	6..7	0 = Reserved 1 = Immediate data is 16 bits (w) 2 = Immediate data is 32 bits (d) 3 = Immediate data is 64 bits (q)
	0..5	Opcode = 0x39
1	Bit	Description
	7	Reserved = 0
	6	0 = Operand 1 index absent 1 = Operand 1 index present
	4..5	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index	
2..3/4..5	16-bit immediate offset	
2..5/4..7	32-bit immediate offset	
2..9/4..11	64-bit immediate offset	

Behaviors and Restrictions

- If an Operand 1 index is specified and Operand 1 is direct, then an instruction encoding exception is generated.

MOVsn

Syntax

MOVsn{w} {@}R_1, {Index16}, {@}R_2 {Index16|Immed16}

MOVsn{d} {@}R_1 {Index32}, {@}R_2 {Index32|Immed32}

Description

Moves a signed natural value from Operand 2 to Operand 1. Both operands can be indexed, though the indexes are the same size. Indexes can be either 16 bits (MOVsnw) or 32 bits (MOVsnd) in size.

Operation

```
Operand 1 <= Operand 2
```

Table 179. MOVsn Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 1 index absent 1 = Operand 1 index present
	6	0 = Operand 2 index/immediate data absent 1 = Operand 2 index/immediate data present
	0..5	0x25 = MOVsnw opcode 0x26 = MOVsnd opcode
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index (MOVsnw)	
2..3/4..5	Optional 16-bit Operand 2 index (MOVsnw)	
2..5	Optional 32-bit Operand 1 index/immediate data (MOVsnd)	
2..5/6..9	Optional 32-bit Operand 2 index/immediate data (MOVsnd)	

Behaviors and Restrictions

- If Operand 2 is direct, and Operand 2 index/immediate data is specified, then the immediate value is read as a signed immediate value and is added to the contents of Operand 2 register such that Operand 2 = R_2 + Immed.
- If Operand 2 is indirect, and Operand 2 index/immediate data is specified, then the immediate data is interpreted as an index and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].

- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.
- If Operand 1 is direct, then the Operand 2 value is sign-extended to 64-bits on 32-bit native machines.

MUL

Syntax

MUL[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

Description

Perform a signed multiply of two operands and store the result back to Operand 1. The operands can be either 32 bits (MUL32) or 64 bits (MUL64).

Operation

```
Operand 1 <= Operand * Operand 2
```

Table 180. MUL Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0E
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 2 immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is MUL32, and Operand 1 is direct, then the result is stored to Operand 1 register with the upper 32 bits cleared.

MULU

Syntax

```
MULU[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Performs an unsigned multiply of two 32-bit (MULU32) or 64-bit (MULU64) operands, and stores the result back to Operand 1.

Operation

```
Operand 1 <= Operand * Operand 2
```

Table 181. MULU Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0F
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is MULU32 and Operand 1 is direct, then the result is written to the Operand 1 register with the upper 32 bits cleared.

NEG

Syntax

NEG[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

Description

Multiply Operand 2 by negative 1, and store the result back to Operand 1. Operand 2 is a signed value and fetched as either a 32-bit (NEG32) or 64-bit (NEG64) value.

Operation

```
Operand 1 <= -1 * Operand 2
```

Table 182. NEG Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0B
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is NEG32 and Operand 1 is direct, then the result is stored in Operand 1 register with the upper 32-bits cleared.

NOT

Syntax

NOT[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

Description

Performs a logical NOT operation on Operand 2, an unsigned 32-bit (NOT32) or 64-bit (NOT64) value, and stores the result back to Operand 1.

Operation

```
Operand 1 <= NOT Operand 2
```

Table 183. NOT Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0A
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is NOT32 and Operand 1 is a register, then the result is stored in the Operand 1 register with the upper 32 bits cleared.

OR

Syntax

`OR[32|64] {@}`R_1`, {@}`R_2` {Index16|Immed16}`

Description

Performs a bit-wise OR of two 32-bit (OR32) or 64-bit (OR64) operands, and stores the result back to Operand 1.

Operation

```
Operand 1 <= Operand 1 OR Operand 2
```

Table 184. OR Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x15
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is OR32 and Operand 1 is direct, then the result is stored to Operand 1 register with the upper 32 bits cleared.

POP

Syntax

```
POP[32|64] {@}R1 {Index16|Immed16}
```

Description

This instruction pops a 32-bit (POP32) or 64-bit (POP64) value from the stack, stores the result to Operand 1, and adjusts the stack pointer **R0** accordingly.

Operation

```
Operand 1 <= [R0]
R0 <= R0 + 4 (POP32)
R0 <= R0 + 8 (POP64)
```

Table 185. POP Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x2C
1	Bit	Description
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is read as a signed value and is added to the value popped from the stack, and the result stored to the Operand 1 register.
- If Operand 1 is indirect, then the immediate data is interpreted as an index, and the value popped from the stack is stored to address [R_1 + Index16].
- If the instruction is POP32, and Operand 1 is direct, then the popped value is sign-extended to 64 bits before storing to the Operand 1 register.

POPn

Syntax

```
POPn {@}R1 {Index16|Immed16}
```

Description

Read an unsigned natural value from memory pointed to by stack pointer **R0**, adjust the stack pointer accordingly, and store the value back to Operand 1.

Operation

```
Operand 1 <= (UINTN)[R0]
R0 <= R0 + sizeof (VOID *)
```

Table 186. POPn Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	Reserved = 0
	0..5	Opcode = 0x36
1	Bit	Description
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is fetched as a signed value and is added to the value popped from the stack and the result is stored back to the Operand 1 register.
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the value popped from the stack is stored at [R_1 + Index16].
- If Operand 1 is direct, and the instruction is executed on a 32-bit machine, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

PUSH

Syntax

```
PUSH[32|64] {@}R1 {Index16|Immed16}
```

Description

Adjust the stack pointer **R0** and store a 32-bit (PUSH32) or 64-bit (PUSH64) Operand 1 value on the stack.

Operation

```
R0 <= R0 - 4 (PUSH32)
R0 <= R0 - 8 (PUSH64)
[R0] <= Operand 1
```

Table 187. PUSH Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x2B
1	Bit	Description
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is read as a signed value and is added to the Operand 1 register contents such that Operand 1 = R_1 + Immed16.
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the pushed value is read from [R_1 + Index16].

PUSHn

Syntax

```
PUSHn {@}R1 {Index16|Immed16}
```

Description

Adjust the stack pointer **R0**, and store a natural value on the stack.

Operation

```
R0 <= R0 - sizeof (VOID *)
[R0] <= Operand 1
```

Table 188. PUSHn Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	Reserved = 0
	0..5	Opcode = 0x35
1	Bit	Description
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is fetched as a signed value and is added to the Operand 1 register contents such that Operand 1 = R_1 + Immed16.
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the Operand 1 value pushed is fetched from [R_1 + Index16].

RET

Syntax

```
RET
```

Description

This instruction fetches the return address from the stack, sets the **IP** to the value, adjusts the stack pointer register **R0**, and continues execution at the return address. If the RET is a final return from the EBC driver, then execution control returns to the caller, which may be EBC or native code.

Operation

```
IP <= [R0]
R0 <= R0 + 16
```

Table 189. RET Instruction Encoding

BYTE	Description

0	Bit	Description
	6..7	Reserved = 0
	0..5	Opcode = 0x04
1	Reserved = 0	

Behaviors and Restrictions

- An alignment exception will be generated if the return address is not aligned on a 16-bit boundary.

SHL

Syntax

```
SHL[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Left-shifts Operand 1 by Operand 2 bit positions and stores the result back to Operand 1. The operand sizes may be either 32-bits (SHL32) or 64 bits (SHL64).

Operation

```
Operand 1 <= Operand 1 << Operand 2
```

Table 190. SHL Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x17
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].

- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is SHL32, and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

SHR

Syntax

`SHR[32|64] {@}R1, {@}R2 {Index16|Immed16}`

Description

Right-shifts unsigned Operand 1 by Operand 2 bit positions and stores the result back to Operand 1. The operand sizes may be either 32-bits (SHR32) or 64 bits (SHR64).

Operation

```
Operand 1 <= Operand 1 >> Operand 2
```

Table 191. SHR Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x18
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is SHR32, and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

STORESP

Syntax

```
STORESP R1, [IP|Flags]
```

Description

This instruction transfers the contents of a dedicated register to a general-purpose register. See Table 150 for the VM dedicated registers and their corresponding index values.

Operation

```
Operand 1 <= Operand 2
```

Table 192. STORESP Instruction Encoding

BYTE	Description	
0	Bit	Description
	6..7	Reserved = 0
	0..5	Opcode = 0x2A
1	7	Reserved = 0
	4..6	Operand 2 dedicated register index
	3	Reserved = 0
	0..2	Operand 1 general purpose register

Behaviors and Restrictions

- Specifying an invalid dedicated register index results in an instruction encoding exception.

SUB

Syntax

```
SUB[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Subtracts a 32-bit (SUB32) or 64-bit (SUB64) signed Operand 2 value from a signed Operand 1 value of the same size, and stores the result to Operand 1.

Operation

```
Operand 1 <= Operand 1 - Operand 2
```

Table 193. SUB Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0D
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is SUB32 and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

XOR

Syntax

XOR[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

Description

Performs a bit-wise exclusive OR of two 32-bit (XOR32) or 64-bit (XOR64) operands, and stores the result back to Operand 1.

Operation

```
Operand 1 <= Operand 1 XOR Operand 2
```

Table 194. XOR Instruction Encoding

BYTE	Description	
0	Bit	Description
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x16
1	Bit	Description
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is XOR32 and Operand1 is direct, then the result is stored to the Operand 1 register with the upper 32-bits cleared.

22.9 Runtime and Software Conventions

22.9.1 Calling Outside VM

Calls can be made to routines in other modules that are native or in another VM. It is the responsibility of the calling VM to prepare the outgoing arguments correctly to make the call outside the VM. It is also the responsibility of the VM to prepare the incoming arguments correctly for the call from outside the VM. Calls outside the VM must use the CALLEX instruction.

22.9.2 Calling Inside VM

Calls inside VM can be made either directly using the CALL or **CALLEX** instructions. Using direct CALL instructions is an optimization.

22.9.3 Parameter Passing

Parameters are pushed on the VM stack per the CDECL calling convention. Per this convention, the last argument in the parameter list is pushed on the stack first, and the first argument in the parameter list is pushed on the stack last.

All parameters are stored or accessed as natural size (using naturally sized instruction) except 64-bit integers, which are pushed as 64-bit values. 32-bit integers are pushed as natural size (since they should be passed as 64-bit parameter values on 64-bit machines).

22.9.4 Return Values

Return values of 8 bytes or less in size are returned in general-purpose register **R7**. Return values larger than 8 bytes are not supported.

22.9.5 Binary Format

PE32+ format will be used for generating binaries for the VM. A VarBss section will be included in the binary image. All global and static variables will be placed in this section. The size of the section will be based on worst-case 64-bit pointers. Initialized data and pointers will also be placed in the VarBss section, with the compiler generating code to initialize the values at runtime.

22.10 Architectural Requirements

This section provides a high level overview of the architectural requirements that are necessary to support execution of EBC on a platform.

22.10.1 EBC Image Requirements

All EBC images will be PE32+ format. Some minor additions to the format will be required to support EBC images. See the *Microsoft Portable Executable and Common Object File Format Specification* pointed to in Appendix Q for details of this image file format.

A given EBC image must be executable on different platforms, independent of whether it is a 32- or 64-bit processor. All EBC images should be driver implementations.

22.10.2 EBC Execution Interfacing Requirements

EBC drivers will typically be designed to execute in an (usually preboot) EFI environment. As such, EBC drivers must be able to invoke protocols and expose protocols for use by other drivers or applications. The following execution transitions must be supported:

- EBC calling EBC
- EBC calling native code
- Native code calling EBC
- Native code calling native code
- Returning from all the above transitions

Obviously native code calling native code is available by default, so is not discussed in this document.

To maintain backward compatibility with existing native code, and minimize the overhead for non-EBC drivers calling EBC protocols, all four transitions must be seamless from the application perspective. Therefore, drivers, whether EBC or native, shall not be required to have any knowledge of whether or not the calling code, or the code being called, is native or EBC compiled code. The onus is put on the tools and interpreter to support this requirement.

22.10.3 Interfacing Function Parameters Requirements

To allow code execution across protocol boundaries, the interpreter must ensure that parameters passed across execution transitions are handled in the same manner as the standard parameter passing convention for the native processor.

22.10.4 Function Return Requirements

The interpreter must support standard function returns to resume execution to the caller of external protocols. The details of this requirement are specific to the native processor. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code.

22.10.5 Function Return Values Requirements

The interpreter must support standard function return values from called protocols. The exact implementation of this functionality is dependent on the native processor. This requirement applies to return values of 64 bits or less. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code. Note that returning of structures is not supported.

22.11 EBC Interpreter Protocol

The EFI EBC protocol provides services to execute EBC images, which will typically be loaded into option ROMs.

EFI_EBC_PROTOCOL

Summary

This protocol provides the services that allow execution of EBC images.

GUID

```
#define EFI_EBC_PROTOCOL_GUID \
 {0x13ac6dd1,0x73d0,0x11d4,\
  {0xb0,0x6b,0x00,0xaa,0x00,0xbd,0x6d,0xe7}}
```

Protocol Interface Structure

```
typedef struct _EFI_EBC_PROTOCOL {
 EFI_EBC_CREATE_THUNK          CreateThunk;
 EFI_EBC_UNLOAD_IMAGE          UnloadImage;
 EFI_EBC_REGISTER_ICACHE_FLUSH RegisterICacheFlush;
 EFI_EBC_GET_VERSION          GetVersion;
} EFI_EBC_PROTOCOL;
```

Parameters

CreateThunk	Creates a thunk for an EBC image entry point or protocol service, and returns a pointer to the thunk. See the `CreateThunk()` function description.
UnloadImage	Called when an EBC image is unloaded to allow the interpreter to perform any cleanup associated with the image's execution. See the `UnloadImage()` function description.
RegisterICacheFlush	Called to register a callback function that the EBC interpreter can call to flush the processor instruction cache after creating thunks. See the `RegisterICacheFlush())` function description.
GetVersion	Called to get the version of the associated EBC interpreter. See the `GetVersion()` function description.

Description

The EFI EBC protocol provides services to load and execute EBC images, which will typically be loaded into option ROMs. The image loader will load the EBC image, perform standard relocations, and invoke the `CreateThunk()` service to create a thunk for the EBC image's entry point. The image can then be run using the standard EFI start image services.

EFI_EBC_PROTOCOL.CreateThunk()

Summary

Creates a thunk for an EBC entry point, returning the address of the thunk.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_EBC_CREATE_THUNK) (
 IN EFI_EBC_PROTOCOL   *This,
 IN EFI_HANDLE       ImageHandle,
 IN VOID          *EbcEntryPoint,
 OUT VOID          **Thunk
 );
```

Parameters

This	A pointer to the `EFI_EBC_PROTOCOL` instance. This protocol is defined in Section 22.11.
ImageHandle	Handle of image for which the thunk is being created.
EbcEntryPoint	Address of the actual EBC entry point or protocol service the thunk should call.
Thunk	Returned pointer to a thunk created.

Description

A PE32+ EBC image, like any other PE32+ image, contains an optional header that specifies the entry point for image execution. However for EBC images this is the entry point of EBC instructions, so is not directly executable by the native processor. Therefore when an EBC image is loaded, the loader must call this service to get a pointer to native code (thunk) that can be executed which will invoke the interpreter to begin execution at the original EBC entry point.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Image entry point is not 2-byte aligned.
EFI_OUT_OF_RESOURCES	Memory could not be allocated for the thunk.

EFI_EBC_PROTOCOL.UnloadImage()

Summary

Called prior to unloading an EBC image from memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_EBC_UNLOAD_IMAGE) (
 IN EFI_EBC_PROTOCOL    *This,
 IN EFI_HANDLE         ImageHandle
 );
```

Parameters

This	A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in Section 22.11.
ImageHandle	Image handle of the EBC image that is being unloaded from memory.

Description

This function is called after an EBC image has exited, but before the image is actually unloaded. It is intended to provide the interpreter with the opportunity to perform any cleanup that may be necessary as a result of loading and executing the image.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Image handle is not recognized as belonging to an EBC image that has been executed.

EFI_EBC_PROTOCOL.RegisterICacheFlush()

Summary

Registers a callback function that the EBC interpreter calls to flush the processor instruction cache following creation of thunks.

Prototype

```
typedef
EFI_STATUS
(* EFI_EBC_REGISTER_ICACHE_FLUSH) (
 IN EFI_EBC_PROTOCOL       *This,
 IN EBC_ICACHE_FLUSH       Flush
 );
```

Parameters

This	A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in Section 22.11.
Flush	Pointer to a function of type EBC_ICACH_FLUSH. See “Related Definitions” below for a detailed description of this type.

Related Definitions

```
typedef
EFI_STATUS
(* EBC_ICACHE_FLUSH) (
 IN EFI_PHYSICAL_ADDRESS   Start,
 IN UINT64          Length
 );
```

Start	The beginning physical address to flush from the processor's instruction cache.
Length	The number of bytes to flush from the processor's instruction cache.

This is the prototype for the *Flush* callback routine. A pointer to a routine of this type is passed to the EBC **EFI_EBC_REGISTER_ICACHE_FLUSH** protocol service.

Description

An EBC image's original PE32+ entry point is not directly executable by the native processor. Therefore to execute an EBC image, a thunk (which invokes the EBC interpreter for the image's original entry point) must be created for the entry point, and the thunk is executed when the EBC image is started. Since the thunks may be created on-the-fly in memory, the processor's instruction cache may require to be flushed after thunks are created. The caller to this EBC service can provide a pointer to a function to flush the instruction cache for any thunks created after the `CreateThunk()` service has been called. If an instruction-cache flush callback is not provided to the interpreter, then the interpreter assumes the system has no instruction cache, or that flushing the cache is not required following creation of thunks.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.

EFI_EBC_PROTOCOL.GetVersion()

Summary

Called to get the version of the interpreter.

Prototype

```
typedef
EFI_STATUS
(* EFI_EBC_GET_VERSION) (
 IN EFI_EBC_PROTOCOL   *This,
 OUT UINT64     *Version
 );
```

Parameters

This	A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in Section 22.11.
Version	Pointer to where to store the returned version of the interpreter.

Description

This function is called to get the version of the loaded EBC interpreter. The value and format of the returned version is identical to that returned by the EBC BREAK 1 instruction.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Version pointer is **NULL**.

22.12 EBC Tools

22.12.1 EBC C Compiler

This section describes the responsibilities of the EBC C compiler. To fully specify these responsibilities requires that the thunking mechanisms between EBC and native code be described.

22.12.2 C Coding Convention

The EBC C compiler supports only the C programming language. There is no support for C++, inline assembly, floating point types/operations, or C calling conventions other than CDECL.

Pointer type in C is supported only as 64-bit pointer. The code should be 64-bit pointer ready (not assign pointers to integers and vice versa).

The compiler does not support user-defined sections through pragmas.

Global variables containing pointers that are initialized will be put in the uninitialized VarBss section and the compiler will generate code to initialize these variables during load time. The code will be placed in an init text section. This compiler-generated code will be executed before the actual image entry point is executed.

22.12.3 EBC Interface Assembly Instructions

The EBC instruction set includes two forms of a CALL instruction that can be used to invoke external protocols. Their assembly language formats are:

```
CALLEX Immed64
CALLEX32 {@}R1 {Immed32}
```

Both forms can be used to invoke external protocols at an absolute address specified by the immediate data and/or register operand. The second form also supports jumping to code at a relative address. When one of these instructions is executed, the interpreter is responsible for thunking arguments and then jumping to the destination address. When the called function returns, code begins execution at the EBC instruction following the CALL instruction. The process by which this happens is called thunking. Later sections describe this operation in detail.

22.12.4 Stack Maintenance and Argument Passing

There are several EBC assembly instructions that directly manipulate the stack contents and stack pointer. These instructions operate on the EBC stack, not the interpreter stack. The instructions include the EBC PUSH, POP, PUSHn, and POPn, and all forms of the MOV instructions.

These instructions must adjust the EBC stack pointer in the same manner as equivalent instructions of the native instruction set. With this implementation, parameters pushed on the stack by an EBC driver can be accessed normally for stack-based native code. If native code expects parameters in registers, then the interpreter thunking process must transfer the arguments from EBC stack to the appropriate processor registers. The process would need to be reversed when native code calls EBC.

22.12.5 Native to EBC Arguments Calling Convention

The calling convention for arguments passed to EBC functions follows the standard CDECL calling convention. The arguments must be pushed as their native size. After the function arguments have been pushed on the stack, execution is passed to the called EBC function. The overhead of thunking the function parameters depends on the standard parameter passing convention for the host processor. The implementation of this functionality is left to the interpreter.

22.12.6 EBC to Native Arguments Calling Convention

When EBC makes function calls via function pointers, the EBC C compiler cannot determine whether the calls are to native code or EBC. It therefore assumes that the calls are to native code, and emits the appropriate EBC CALLEX instructions. To be compatible with calls to native code, the calling convention of EBC calling native code must follow the parameter passing convention of the native processor. The EBC C compiler generates EBC instructions that push all arguments on the stack. The interpreter is then responsible for performing the necessary thunking. The exact implementation of this functionality is left to the interpreter.

22.12.7 EBC to EBC Arguments Calling Convention

If the EBC C compiler is able to determine that a function call is to a local function, it can emit a standard EBC CALL instruction. In this case, the function arguments are passed as described in the other sections of this specification.

22.12.8 Function Returns

When EBC calls an external function, the thunking process includes setting up the host processor stack or registers such that when the called function returns, execution is passed back to the EBC at the instruction following the call. The implementation is left to the interpreter, but it must follow the standard function return process of the host processor. Typically this will require the interpreter to push the return address on the stack or move it to a processor register prior to calling the external function.

22.12.9 Function Return Values

EBC function return values of 8 bytes or less are returned in VM general-purpose register **R7**. Returning values larger than 8 bytes on the stack is not supported. Instead, the caller or callee must allocate memory for the return value, and the caller can pass a pointer to the callee, or the callee can return a pointer to the value in the standard return register **R7**.

If an EBC function returns to native code, then the interpreter thunking process is responsible for transferring the contents of **R7** to an appropriate location such that the caller has access to the value using standard native code. Typically the value will be transferred to a processor register. Conversely, if a native function returns to an EBC function, the interpreter is responsible for transferring the return value from the native return memory or register location into VM register **R7**.

22.12.10 Thunking

Thunking is the process by which transitions between execution of native and EBC are handled. The major issues that must be addressed for thunking are the handling of function arguments, how the external function is invoked, and how return values and function returns are handled. The following sections describe the thunking process for the possible transitions.

22.12.10.1 Thunking EBC to Native Code

By definition, all external calls from within EBC are calls to native code. The EBC CALLEX instructions are used to make these calls. A typical application for EBC calling native code would be a simple "Hello World" driver. For a UEFI driver, the code could be written as shown below.

```
EFI_STATUS EfiMain (
  IN EFI_HANDLE       ImageHandle,
  IN EFI_SYSTEM_TABLE    *ST
  )
{
     ST->ConOut->OutputString(ST->ConOut, L"Hello World!");
     return EFI_SUCCESS;
}
```

This C code, when compiled to EBC assembly, could result in two PUSHn instructions to push the parameters on the stack, some code to get the absolute address of the `OutputString()` function, then a CALLEX instruction to jump to native code. Typical pseudo assembly code for the function call could be something like the following:

```
PUSHn       _HelloString
PUSHn       _ConOut
MOVnw       R1, _OutputString
CALLEX64R1
```

The interpreter is responsible for executing the PUSHn instructions to push the arguments on the EBC stack when interpreting the PUSHn instructions. When the CALLEX instruction is encountered, it must thunk to external native code. The exact thunking mechanism is native processor dependent. For example, a supported 32-bit thunking implementation could simply move the system stack pointer to point to the EBC stack, then perform a CALL to the absolute address specified in VM register **R1**. However, the function calling convention for the Itanium processor family calls for the first 8 function arguments being passed in registers. Therefore, the Itanium processor family thunking mechanism requires the arguments to be copied from the EBC stack into processor registers. Then a CALL can be performed to jump to the absolute address in VM register **R1**. Note that since the interpreter is not aware of the number of arguments to the function being called, the maximum amount of data may be copied from the EBC stack into processor registers.

22.12.10.2 Thunking Native Code to EBC

An EBC driver may install protocols for use by other EBC drivers, or UEFI drivers or applications. These protocols provide the mechanism by which external native code can call EBC. Typical C code to install a generic protocol is shown below.

```
EFI_STATUS Foo(UINT32 Arg1, UINT32 Arg2);

MyProtInterface->Service1= Foo;

Status = LibInstallProtocolInterfaces (&Handle, &MyProtGUID, MyProtInterface,
NULL);
```

To support thunking native code to EBC, the EBC compiler resolves all EBC function pointers using one level of indirection. In this way, the address of an EBC function actually becomes the address of a piece of native (thunk) code that invokes the interpreter to execute the actual EBC function. As a result of this implementation, any time the address of an EBC function is taken, the EBC C compiler must generate the following:

- A 64-bit function pointer data object that contains the actual address of the EBC function
- EBC initialization code that is executed before the image entry point that will execute EBC BREAK 5 instructions to create thunks for each function pointer data object
- Associated relocations for the above

So for the above code sample, the compiler must generate EBC initialization code similar to the following. This code is executed prior to execution of the actual EBC driver's entry point.

```
MOVqq R7, Foo_pointer; get address of Foo pointer
BREAK 5                 ; create a thunk for the function
```

The BREAK instruction causes the interpreter to create native thunk code elsewhere in memory, and then modify the memory location pointed to by R7 to point to the newly created thunk code for EBC function Foo. From within EBC, when the address of Foo is taken, the address of the thunk is actually returned. So for the assignment of the protocol Service1 above, the EBC C compiler will generate something like the following:

```
MOVqq R7, Foo_pointer; get address of Foo function pointer
MOVqq R7, @R7            ; one level of indirection
MOVn  R6, _MyProtInterface->Service1 ; get address of variable
MOVqq @R6, R7            ; address of thunk to ->Service1
```

22.12.10.3 Thunking EBC to EBC

EBC can call EBC via function pointers or protocols. These two mechanisms are treated identically by the EBC C compiler, and are performed using EBC CALLEX instructions. For EBC to call EBC, the EBC being called must have provided the address of the function. As described above, the address is actually the address of native thunk code for the actual EBC function. Therefore, when EBC calls EBC, the interpreter assumes native code is being called so prepares function arguments accordingly, and then makes the call. The native thunk code assumes native code is calling EBC, so will basically “undo” the preparation of function arguments, and then invoke the interpreter to execute the actual EBC function of interest.

22.12.11 EBC Linker

New constants must be defined for use by the linker in processing EBC images. For EBC images, the linker must set the machine type in the PE file header accordingly to indicate that the image contains EBC.

```
#define IMAGE_FILE_MACHINE_EBC 0x0EBC
```

In addition, the linker must support EBC images with of the following subsystem types as set in a PE32+ optional header:

```
#define IMAGE_SUBSYSTEM_EFI_APPLICATION       10
#define IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER    11
#define IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER       12
```

For EFI EBC images and object files, the following relocation types must be supported:

```
// No relocations required
#define IMAGE_REL_EBC_ABSOLUTE  0x0000
// 32-bit address w/o image base
#define IMAGE_REL_EBC_ADDR32NB  0x0001
// 32-bit relative address from byte following relocs
#define IMAGE_REL_EBC_REL32    0x0002
// Section table index
#define IMAGE_REL_EBC_SECTION   0x0003
// Offset within section
#define IMAGE_REL_EBC_SECREL   0x0004
```

The ADDR32NB relocation is used internally to the linker when RVAs are emitted. It also is used for version resources which probably will not be used. The REL32 relocation is for PC relative addressing on code. The SECTION and SECREL relocations are used for debug information.

22.12.12 Image Loader

The EFI image loader is responsible for loading an executable image into memory and applying relocation information so that an image can execute at the address in memory where it has been loaded prior to execution of the image. For EBC images, the image loader must also invoke the interpreter protocol to create a thunk for the image entry point and return the address of this thunk. After loading the image in this manner, the image can be executed in the standard manner. To implement this functionality, only minor changes will be made to EFI service `EFI_BOOT_SERVICES.LoadImage()`, and no changes should be made to `EFI_BOOT_SERVICES.StartImage()`.

After the image is unloaded, the EFI image load service must call the EBC `EFI_BOOT_SERVICES.UnloadImage()` service to perform any cleanup to complete unloading of the image. Typically this will include freeing up any memory allocated for thunks for the image during load and execution.

22.12.13 Debug Support

The interpreter must support debugging in an EFI environment per the EFI debug support protocol.

22.13 VM Exception Handling

This section lists the different types of exceptions that the VM may assert during execution of an EBC image. If a debugger is attached to the EBC driver via the EFI debug support protocol, then the debugger should be able to capture and identify the exception type. If a debugger is not attached, then depending on the severity of the exception, the interpreter may do one of the following:

- Invoke the EFI ASSERT() macro, which will typically display an error message and halt the system
- Sit in a while(1) loop to hang the system
- Ignore the exception and continue execution of the image (minor exceptions only)

It is a platform policy decision as to the action taken in response to EBC exceptions. The following sections describe the exceptions that may be generated by the VM.

22.13.1 Divide By 0 Exception

A divide-by-0 exception can occur for the EBC instructions `DIV`, `DIVU`, `MOD`, and `MODU`.

22.13.2 Debug Break Exception

A debug break exception occurs if the VM encounters a BREAK instruction with a break code of 3.

22.13.3 Invalid Opcode Exception

An invalid opcode exception will occur if the interpreter encounters a reserved opcode during execution.

22.13.4 Stack Fault Exception

A stack fault exception can occur if the interpreter detects that function nesting within the interpreter or system interrupts was sufficient to potentially corrupt the EBC image's stack contents. This exception could also occur if the EBC driver attempts to adjust the stack pointer outside the range allocated to the driver.

22.13.5 Alignment Exception

An alignment exception can occur if the particular implementation of the interpreter does not support unaligned accesses to data or code. It may also occur if the stack pointer or instruction pointer becomes misaligned.

22.13.6 Instruction Encoding Exception

An instruction encoding exception can occur for the following:

- For some instructions, if an Operand 1 index is specified and Operand 1 is direct
- If an instruction encoding has reserved bits set to values other than 0
- If an instruction encoding has a field set to a reserved value.

22.13.7 Bad Break Exception

A bad break exception occurs if the VM encounters a BREAK instruction with a break code of 0, or any other unrecognized or unsupported break code.

22.13.8 Undefined Exception

An undefined exception can occur for other conditions detected by the VM. The cause of such an exception is dependent on the VM implementation, but will most likely include internal VM faults.

22.14 Option ROM Formats

The new option ROM capability is designed to be a departure from the legacy method of formatting an option ROM. PCI local bus add-in cards are the primary targets for this design although support for future bus types will be added as necessary. EFI EBC drivers can be stored in option ROMs or on hard drives in an EFI system partition.

The new format defined for the UEFI specification is intended to coexist with legacy format PCI Expansion ROM images. This provides the ability for IHVs to make a single option ROM binary that contains both legacy and new format images at the same time. This is important for the ability to have single add-in card SKUs that can work in a variety of systems both with and without native support for UEFI. Support for multiple image types in this way provides a smooth migration path during the period before widespread adoption of UEFI drivers as the primary means of support for software needed to accomplish add-in card operation in the pre-OS boot timeframe.

22.14.1 EFI Drivers for PCI Add-in Cards

The location mechanism for UEFI drivers in PCI option ROM containers is described fully in Section 11.3. Readers should refer to this section for complete details of the scheme and associated data structures.

22.14.2 Non-PCI Bus Support

EFI expansion ROMs are not supported on any other bus besides PCI local bus in the current revision of the UEFI specification.

This means that support for UEFI drivers in legacy ISA add-in card ROMs is explicitly excluded.

Support for UEFI drivers to be located on add-in card type devices for future bus designs other than PCI local bus will be added to future revisions of the UEFI specification. This support will depend upon the specifications that govern such new bus designs with respect to the mechanisms defined for support of driver code on devices.

23 - Firmware Update and Reporting

The UEFI Firmware Management Protocol provides an abstraction for device to provide firmware management support. The base requirements for managing device firmware images include identifying firmware image revision level and programming the image into the device.

The protocol for managing firmware provides the following services.

- Get the attributes of the current firmware image. Attributes include revision level.
- Get a copy of the current firmware image. As an example, this service could be used by a management application to facilitate a firmware roll-back.
- Program the device with a firmware image supplied by the user.
- Label all the firmware images within a device with a single version.

When UEFI Firmware Management Protocol (FMP) instance is intended to perform the update of an option ROM loaded from a PCI or PCI Express device, it is recommended that the FMP instance be attached to the handle with **EFI_LOADED_IMAGE_PROTOCOL** for said Option ROM.

When the FMP instance is intended to update internal device firmware, or a combination of device firmware and Option ROM, the FMP instance may instead be attached to the Controller handle of the device. However in the case where multiple devices represented by multiple controller handles are served by the same firmware store, only a single Controller handle should expose FMP. In all cases a specific updatable hardware firmware store must be represented by exactly one FMP instance.

Care should be taken to ensure that the FMP instance reports current version data that accurately represents the actual contents of the firmware store of the device exposing FMP, because in some cases the device driver currently operating the device may have been loaded from another device or media.

23.1 Firmware Management Protocol

EFI_FIRMWARE_MANAGEMENT_PROTOCOL

Summary

Firmware Management application invokes this protocol to manage device firmware.

GUID

```
// {86C77A67-0B97-4633-A187-49104D0685C7}
#define EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GUID \
 { 0x86c77a67, 0xb97, 0x4633, \
  {0xa1, 0x87, 0x49, 0x10, 0x4d, 0x06, 0x85, 0xc7 }}
```