ASIP Designer

The Compiler Header File of the Tmicro Core

Version L-2016.09

SYNOPSYS®

Copyright Notice and Proprietary Information

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Changes

Version	Date	Change
11R1.0	Apr 2011	Renamed BASE into TMICRO.
		Optimisation of long multiplication 6.5.
		Initialisation code 8.3.
12R1.1	Feb 2012	Removed PM-specific pointer representation.
		Added select property.
12R1.2	May 2012	Use divide step instruction for implementing 16 bit division 3.9.
13R1.1	Oct 2012	Added `duplicate_at_using_opn2` in 6.4.
J-2014.09	Oct 2014	First release under Synopsys.
J-2015.03	Mar 2015	No change.
L-2016.08	Aug 2016	New implementation of `macl` instruction.

Abstract

This document describes the compiler header file of the TMICRO core. This file specifies the mapping of the C built-in integer types and operators to the primitive processor types and primitive processor functions. TMICRO is a 16 bit processor, so we chose to use a 16 bit representation for the C types `int` and `unsigned`. Operators on these types can then be mapped onto ALU instructions is a fairly straightforward way. Pointers are also represented with 16 bits, pointer expressions are mapped onto the ALU or AGU.

The TMICRO processor also supports the C types `long` and `unsigned long`. These are represented as a tuple of two 16 bit values. Operators on the long type are implemented using double-precision arithmetic. Note that the TMICRO processor does not support floating point types.

Contents

1

Introduction

The CHESS compiler uses a compiler processor header file to specify the mapping of the C built-in types and operators to the primitive processor types and primitive processor functions. This document describes the compiler header file `base_chess.h` of the TMICRO core. Actually, `base_chess.h` itself defines only some aspects of the mapping, and then includes some other header files.

TMICRO is a 16 bit processor, so we chose to use a 16 bit representation for the C types `int` and `unsigned`. Operators on these types can then be mapped onto ALU instructions in a fairly straightforward way. This mapping is explained in chapter 3.

Chapter 4 explains how pointers and pointer expressions are mapped on the TMICRO core. Chapter 5 explains how short types such as `char` and `short` are mapped on the TMICRO core.

The TMICRO processor also supports the C types `long` and `unsigned long`. These are represented as a tuple of two 16 bit values. Operators on the long type are implemented using double-precision arithmetic. This mapping is explained in chapter 6.

2

The tmicro_chess.h header file

2.1 Software stack

The C language requires that certain (large) variables are allocated to the memory. On the TMICRO processor, we will allocate variables to the DM memory. To do this, we define the following CHESS property.

```
default_memory       : DM;
```

The C language requires that automatic variables (locals) are allocated on the software stack. In order to organise a stack, a stack pointer register is used to point to the base address of the stack frame of the function that is being executed. The stack pointer is stored in a reserved register. On TMICRO, this is the SP register.

```
stack_pointer      : SP;
```

We will also adopt the convention that the stack grows from small addresses to large addresses, and that the stack pointer points to the first free memory location, one beyond the last occupied location.

```
sp_location        : free;
```

Example The following function has an automatic variable A that is stored in the stack. When the function is entered, a stack frame of size 16 is allocated. The function then loads A[4], which is located at address SP − 12. Finally, the stack frame is deallocated. Note that the access to A[4] and the deallocation are executed in the delay slot of the return instruction.

C function	*Assembly subroutine*
`int stack_alloc()`	`addb sp, 16      ; allocate stack frame`
`{`	`rtd`
`    int A[16];`	`ld r0,dm(sp-12)  ; access automatic A[4]`
`    return A[4];`	`addb sp, -16     ; deallocate stack frame`
`}`	

2.2 Spilling of registers

The CHESS compiler requires that stack load and stack store instructions with a specific addressing mode are identified. CHESS will use these instructions to load or store registers on the stack frame. This is called spilling of registers. On the TMICRO processor, we will use the stack pointer indexed immediate addressing mode. In the nML model, this addressing mode is modelled by means of the load_store_wreg_sp_-indexed rule.

In addition to this nML rule, CHESS uses two properties to identify the spill instructions. The first property indicates for which data types spilling is supported. In general, the types are identified by their memory record alias name. On the TMICRO core, spilling is only supported for the word type, so we must specify the DM memory.

```
spill_memory       : DM;
```

The second property specifies the type of the stack pointer relative index.

```
sp_offset_type     : nint9;
```

Example The following example shows a function `foo` that calls another function `bar`. Before the call to `bar`, two registers are spilled:

- the link register `lr`, which contains the return address of `foo`,
- register `r2`, which contains the argument `b`. `foo`.

After the call, the `b` arguments is loaded into `r1`, and also the link register is reloaded. Also note that a stack frame of size 2 is allocated and deallocated.

C function

```
int bar(int);

int foo(int a, int b)
{
    return bar(a) + b;
}
```

Assembly subroutine

```
addb sp, 2          ; allocate stack frame
st r2,dm(sp-2)      ; save argument b
st lr,dm(sp-1)      ; save link address
cl __sint_bar___sint
ld r1,dm(sp-2)      ; restore argument b
ld lr,dm(sp-1)      ; restore link address
rtd
add r0,r1,r0
addb sp, -2         ; deallocate stack frame
```

2.3 Subroutine linkage

The CHESS compiler relies on a processor specific call and return instruction to implement function calls. These instructions are identified by tagging their primitive functions with the properties `call` and `ret`. On the TMICRO core, the primitive functions are declared as follows.

```
void  ret(addr)   property(ret);
addr  bsr(addr)   property(absolute call);
```

The `bsr` function has one argument, which is the target address of the call. The return value models the link address. This is the address to which the called function must return. The call instruction saves the link address in a designated register called the *link register*. The following property identifies the link register.

```
link_register      : LR;
```

The `ret` function has one argument, which is the address to which must be returned.

2.4 Argument passing

On the TMICRO core, arguments are passed via the first 7 fields of the register file `R`. All scalar input and output arguments, including pointers, are passed via `R`. This is specified by enumerating the 7 fields in the following properties.

```
argument_registers : R0, R1, R2, R3, R4, R5, R6;
```

Note that it is possible to use register field names because of the `syntax` attribute of the register declaration in the nML model.

```
reg R[8]<word,uint3> syntax ("R") ... ;
```

The reason why only 7 out of 8 fields can be used to pass arguments is that indirect function calls are mapped on the `clid` instruction. This instruction needs one `R`-field to specify the target address. So in case a function is called indirectly, only 7 arguments can be passed in `R`. When a function has more than 7 arguments, these are passed via the software stack.

Example The following function has 9 arguments and a return value. The return value is passed in r0. Arguments a to f are passed in r1 to r6. Arguments g, h and i are passed on the stack frame, on locations sp-1, sp-2 and sp-3.

C function

```
int f10(int a, int b, int c,
        int d, int e, int f,
        int g, int h, int i)
{
    return a + b - c
         + d + e - f
         + g + h - i;
}
```

Assembly subroutine

```
add r0,r2,r1        ; return = b + a
sub r0,r0,r3        ;              - c
add r4,r4,r0        ;              + d
add r4,r5,r4        ;              + e
sub r6,r4,r6        ;              - f
ld r5,dm(sp-1)      ; load g
add r6,r5,r6        ;              + g
ld r4,dm(sp-2)      ; load h
add r6,r4,r6        ;              + h
ld r5,dm(sp-3)      ; load i
sub r0,r6,r5        ;              - i
rt
```

2.5 Hardware loops

TMICRO supports zero overhead loops. The CHESS compiler will infer hardware loops based on the doloop property annotation of the hwdo primitive function.

```
void  hwdo(word,addr)  property(absolute doloop);
```

TMICRO supports three hardware loop levels. Two levels are to be used for general C code, as indicated by the loop_levels property, and the third level can be used in interrupt routines.

```
loop_levels        : 2;
```

Furthermore, the registers that keep track of the loop parameters must not be used to store other data, so they are reserved.

```
reserved           : LF, LS, LE, LC;
```

2.6 Register properties

There are a number of registers that must not be used by the compiler for allocating variables. These are specified as reserved registers.

```
reserved           : SRa, SRb;
reserved           : IE, IM;
reserved           : LF, LS, LE, LC;
reserved           : ISR, ILR;
```

Note that the program counter and stack pointer are automatically reserved due to the program_counter and stack_pointer properties.

The single bit condition and carry/borrow registers cannot be spilled, and are therefore identified as status_registers.

```
status_register    : CND, CB;
```

2.7 Complements declarations

TMICRO has ten primitive compare functions. Among these, some complementary functions can be identified. CHESS uses this information to optimise the code.

```
    chess_properties {
        complements : bool lts(word,word), bool ges (word,word);
        complements : bool ltu(word,word), bool geu (word,word);
        complements : bool gts(word,word), bool les (word,word);
        complements : bool gtu(word,word), bool leu (word,word);
        complements : bool eq (word,word), bool ne  (word,word);
    }
```

2.8 Memory copy function

When a C struct variable is assigned to another variable of the same type, the CHESS compiler must make a copy of the memory region in which the struct is stored. To do this, it calls a memory copy function. This is a function with the name `chess_memory_copy()`. It is defined as an inline function in the compiler header file.

```
inline void chess_memory_copy(volatile void* dst,
                              const volatile void* src,
                              const int sz,
                              const int algn)
{
    int* pd = (int*)dst;
    int* ps = (int*)src;
    int ss = sz / sizeof(int);
    if  (ss < 5) {
        if (ss >= 1) *pd++ = *ps++;
        if (ss >= 2) *pd++ = *ps++;
        if (ss >= 3) *pd++ = *ps++;
        if (ss >= 4) *pd++ = *ps++;
    }
    else
        for (int ii = 0; ii < ss; ii++) chess_loop_range(1,) pd[ii] = ps[ii];
}
```

3

Support for `int` and `unsigned`

The mapping of the `int` and `unsigned` data types and operators is modeled in the file `tmicro_int.h`. In this chapter, we will see how the integer types are represented on TMICRO, how conversions are implemented and how the C basic operators are mapped for these types.

3.1 Type representation

The `int` type, and to a lesser extent `unsigned`, are the most naturally used C types. TMICRO is a pure 16 bit processor, so it is most efficient to represent `int` and `unsigned` using the 16 bit primitive type `word`.

```
chess_properties {
    representation int, unsigned  : word;
}
```

3.2 Type conversions

Because `int` and `unsigned` have the same representation, conversions between these types are nil conversions.

```
promotion operator unsigned(int) = nil;
promotion operator int(unsigned) = nil;
```

There is also a conversion from `int` to the primitive type `word`. This conversion is needed for converting loop counts. Because the destination type is primitive, the conversion is added to the primitive name space.

```
namespace tmicro_primitive { promotion word(int) = nil; }
```

3.3 Bitwise logical operators

The Bitwise logical operators are defined by promotion to a primitive function. Note that the same primitive functions are used to define the signed and the unsigned operators.

```
promotion  int operator&(int,int) = word andw(word,word);
promotion  int operator|(int,int) = word orw (word,word);
promotion  int operator^(int,int) = word xorw(word,word);
promotion  int operator~(int)     = word complement(word);

promotion  unsigned operator&(unsigned,unsigned) = word andw(word,word);
promotion  unsigned operator|(unsigned,unsigned) = word orw (word,word);
promotion  unsigned operator^(unsigned,unsigned) = word xorw(word,word);
promotion  unsigned operator~(unsigned)          = word complement(word);
```

Example When the & and | operators are used in expressions, the and and or instructions are generated, as shown in the following example.

C function	*Assembly subroutine*

```
int i_and_or(int a, int b, int c)        and r0,r2,r1
{                                        or r0,r3,r0
    return a & b | c;                    rt
}
```

3.4 Addition and subtraction

Addition and subtraction are also defined by promotion to primitive functions. Note that there are two addition primitives: the ALU addition with carry output, and the AGU addition. The add operator is promoted to both.

```
promotion  int operator+(int,int) = { word add(word,word,uint1&),
                                      word add(word,word) };
promotion  int operator-(int,int) = word sub (word,word,uint1&);
```

The unsigned add and subtract operators are defined is a similar way.

Example When the + and - operators are used in expressions, the add and sub instructions are generated, as shown in the following example.

C function	*Assembly subroutine*

```
int i_and_or(int a, int b, int c)        add r0,r2,r1
{                                        sub r0,r0,r3
    return a + b - c;                    rt
}
```

3.5 Multiplication

The multiply operators are mapped on the signed multiply instruction. The mulss() primitive takes two 16 bit arguments and produces a 32 bit product as two 16 bit word reference arguments. To implement a 16 bit multiplication, we must retain the 16 least significant bits of the product. First, and inline function is defined to extract the LSBs (to avoid polluting the global namespace, the inline function is added to the primitive namespace). The multiply operators are then promoted to this inline function.

```
namespace tmicro_primitive {
    inline word mul(word a, word b) { word x,y; mulss(a,b,x,y); return x; }
}
promotion      int operator*(int,int)           = word mul(word,word);
promotion unsigned operator*(unsigned,unsigned) = word mul(word,word);
```

Example The following example shows the code that results when the * operator is used in a expression.

C function	*Assembly subroutine*

```
int i_mul(int a, int b)                  mulss r1,r2
{                                        mv r0,pl
    return a * b;                        rt
}
```

3.6 Shift operators

The shift operators are defined by promotion to a primitive function. For the signed left and right shift, the logical shift left `lsl` and arithmetic shift right `asr` primitives are used. For the unsigned left and right shift, the logical shift left `lsl` and logical shift right `lsr` primitives are used.

```
promotion int operator<<(int,int)           = word lsl(word,word);
promotion int operator>>(int,int)           = word asr(word,word);
promotion unsigned operator<<(unsigned,int) = word lsl(word,word);
promotion unsigned operator>>(unsigned,int) = word lsr(word,word);
```

3.7 Compare operators

The compare operators are defined by promotion to a primitive function. A distinction is made for signed and unsigned compares.

```
promotion bool operator< (int,int) = bool lts(word,word);
promotion bool operator<=(int,int) = bool les(word,word);
promotion bool operator> (int,int) = bool gts(word,word);
promotion bool operator>=(int,int) = bool ges(word,word);

promotion bool operator< (unsigned,unsigned) = bool ltu(word,word);
promotion bool operator<=(unsigned,unsigned) = bool leu(word,word);
promotion bool operator> (unsigned,unsigned) = bool gtu(word,word);
promotion bool operator>=(unsigned,unsigned) = bool geu(word,word);
```

3.8 Equal operators

The equal and unequal operators are defined by promotion to a primitive function.

```
promotion bool operator==(int,int) = bool eq(word,word);
promotion bool operator!=(int,int) = bool ne(word,word);

promotion bool operator==(unsigned,unsigned) = bool eq(word,word);
promotion bool operator!=(unsigned,unsigned) = bool ne(word,word);
```

3.9 Division and modulo

The division and modulo operators are mapped onto the `ds` instruction. This is done as follows. First, a version of the `divstep` function with `unsiged` arguments is created.

```
promotion void divstep(unsigned, unsigned, unsigned, unsigned&, unsigned&)
        = void divstep(word,word,word,word&,word&);
```

The step primitive is then called 16 times, in order to compute the quotient and remainder.

```
inline unsigned div_remainder(unsigned a, unsigned b, unsigned& rem)
    property(functional)
{
    unsigned q = a;
    unsigned r = 0;
    for (int i = 0; i < 16; i++)
        divstep(b,q,r,q,r);
    rem = r;
    return q;
}
```

The division and modulo operators for `unsigned` can be directly mapped onto `div_remainder`.

```
inline unsigned operator/(unsigned a, unsigned b)
{
    unsigned r;
    return div_remainder(a,b,r);
}

inline unsigned operator%(unsigned a, unsigned b)
{
    unsigned r;
    div_remainder(a,b,r);
    return r;
}
```

The division and modulo operators for `int` require that the operands are first converted into positive values. mapped onto `div_remainder`.

```
inline int operator/(int a, int b)
{
    unsigned abs_a = a < 0 ? -a : a;
    unsigned abs_b = b < 0 ? -b : b;
    unsigned q = abs_a / abs_b;
    return (a^b) < 0 ? -q : q;
}

inline int operator%(int a, int b)
{
    unsigned abs_a = a < 0 ? -a : a;
    unsigned abs_b = b < 0 ? -b : b;
    unsigned r = abs_a % abs_b;
    return a < 0 ? -r : r;
}
```

3.10 Bit fields

In C it is possible to define structs with bit field members. On the TMICRO core, bit fields are packed into variables of type `int`. This is specified with the following property.

```
bitfield_underlying_type : int;
```

The CHESS compiler also needs functions to extract and update bit fields of a specific width located at specific bit position in the containing `int`. These are the functions

```
inline signed int chess_bitfield_extract_signed(int W, int width, int lsb);
inline unsigned int chess_bitfield_extract_unsigned(int W, int width, int lsb);
inline int chess_bitfield_update(int W, int f, int width, int lsb);
```

that are defined in `tmicro_bitfield.h`. They use shifts and bitwise operations to extract and update the bit fields.

4

Support for pointers

CHESS requires that the user defines a pointer type for each memory. Also the built-in pointer operators must be defined. The definitions are located in the `tmicro_int.h` file.

4.1 Type representation

The TMICRO processor has two memories: `DM` and `PM`. Pointers to both memories are represented on the `addr` type. It therefore suffices to have the following pointer representation.

```
chess_properties {
    representation void* : addr;
}
```

4.2 Pointer conversions

The conversions between `signed int` and `unsigned int`, and pointers are supported. As the conversions between the representing types are nil conversions, also the conversions between integers and pointers are nil.

```
promotion operator void*(int)   = nil; // addr(word);
promotion operator int  (void*) = nil; // word(addr);
promotion operator void*   (unsigned) = nil; // addr(word);
promotion operator unsigned(void*)    = nil; // word(addr);
```

4.3 Pointer addition and subtraction

Three additive operators need to be defined: pointer plus offset, pointer minus offset and difference of two pointers. These are defined through promotion to primitive functions.

```
promotion void* operator+(void*,int) = { word add(word,word,uint1&),
                                         word add(word,word) };

promotion void* operator-(void*,int) = word sub(word,word,uint1&);

promotion int operator-(void*,void*) = word sub(word,word,uint1&);
```

According to the C language, when an offset is added to or subtracted from a pointer, the offset is first multiplied by the size of the object to which the pointer points.

Example

C function

```
struct T3i {int i, j, k; };
// sizeof(T3i)==3
T3i* p_add(T3i* p)
{
    return p+2;
}
```

Assembly subroutine

```
mvib r0,6
add r0,r0,r1
rt
```

When the difference of two pointers (to the same type) is taken, the addresses are first subtracted, and the difference is then divided by the size of the object to which the pointers point.

Example

C function

```
struct T3i {int i, j, k; };
// sizeof(T3i)==3
int p_diff(T3i* p, T3i* q)
{
    return p - q;
}
```

Assembly subroutine

```
addb sp, 1
sub r2,r1,r2
st lr,dm(sp-1)
mvib r3,3
cl int_div_QR_div_called___sint___sint
ld lr,dm(sp-1)
addb sp, -1
rt
```

When the divisor is a power of two, the call to the division routine is replaced by a shift instruction.

Example

C function

```
struct T2i {int i, j; };
// sizeof(T2i)==2
int p_diff(T2i* p, T2i* q)
{
    return p - q;
}
```

Assembly subroutine

```
mvib r0,1
sub r1,r1,r2
asr r0,r1,r0
rt
```

4.4 Pointer comparison

The pointer compare operators are defined by promotion to a primitive function. Note that the unsigned relational primitives must be used to compare addresses.

```
promotion bool operator< (void*,void*) = bool ltu(word,word);
promotion bool operator<=(void*,void*) = bool leu(word,word);
promotion bool operator> (void*,void*) = bool gtu(word,word);
promotion bool operator>=(void*,void*) = bool geu(word,word);
promotion bool operator==(void*,void*) = bool eq(word,word);
promotion bool operator!=(void*,void*) = bool ne(word,word);
```

5

Support for short types

In this chapter, we will address the mapping of the short and character types.

5.1 Representation of short types

The TMICRO processor does not support data types that are smaller than 16 bit. The built-in types `signed short` and `unsigned short` are therefore represented in the same way as the `signed int` and `unsigned int` types.

```
chess_properties {
    representation signed short     : int;
    representation unsigned short   : unsigned;
}
```

According to the C language rules, when an operand of a short types is involved in an expression, it is converted to the type `int` through a process known as *integral promotion*. It is therefore not allowed to define operators for the short types.

5.2 Representation of character types

Character types are typically 8 bit wide. As the TMICRO processor does not support 8 bit data types, we choose to represent the character types in the same way as the `int` types.

```
chess_properties {
    representation char, signed char : int;
    representation unsigned char     : unsigned;
}
```

As for short types, it is not allowed to define operators on character types.

6

Support for signed long and unsigned long

The mapping of the `signed long` and `unsigned long` data types and operators is modelled in the file `tmicro_long.h`. In this chapter, we will see how the long types are represented on TMICRO, how conversions are implemented and how the C basic operators are mapped for these types.

6.1 Type representation

We want to represent the long types using 32 bit precision. TMICRO is a pure 16 bit processor, there are no storages that can directly store 32 bit values. We therefore need to represent the long types by means of a structure (this principle is explained in [5]).

```
namespace tmicro_primitive {
   struct dint property(keep_in_registers) {
      unsigned lo;
      unsigned hi;
      dint(unsigned l, unsigned h) : lo(l), hi(h) { }
   };
}

chess_properties {
    representation signed long   : dint;
    representation unsigned long : dint;
}
```

Note that we use a `struct` with two fields of type `unsigned`, we also could have used two `word` elements. The advantages of using a built-in C type are:

- The operators for the long types are defined using double precision algorithms that operate on the `lo` and `hi` elements. When these elements are of type `unsigned`, the C built-in operators can be used to define the double precision routines.
- The C front-end can preform more arithmetic optimisations (such as constant folding, strength reductions, etc...) on a built-in type like `int` and `unsigned` compared to a primitive type like `word`.

The `keep_in_registers` property specifies that the struct must be split into into its members, which are then two individual 16 bit values that can be stored in registers. Also note that TMICRO is a little endian processor, so the first element of the struct must hold the least significant part of the long value. Because `unsigned` operations have clearly defined overflow behavior (compared to `int`), we prefer `unsigned` for the type of both `lo` and `hi`.

6.2 Type conversions

Because `signed long` and `unsigned long` have the same representation, conversions between these types are nil conversions.

```
promotion operator unsigned long(signed long) = nil;
promotion operator signed long(unsigned long) = nil;
```

On TMICRO pointers are 16 bit, we therefore chose not to support the conversions between pointers and the long types.

```
promotion operator void*(signed long)   = undefined;
promotion operator void*(unsigned long) = undefined;
promotion operator signed long(void*)   = undefined;
promotion operator unsigned long(void*) = undefined;
```

A conversion from a 16 bit integer type to a 32 bit integer type must perform sign or zero extension, depending on the sign encoding of the source type. Sign extension is mapped into the `extend_sign()` primitive, so for this conversion the `xs` instruction will be generated. Note that a version of `extend_sign()` with `int` arguments is defined by means of a promotion. Zero extension is implemented by loading zero in the `hi` field of a `dint` struct. These intermediate functions are defined in the primitive namespace.

```
namespace tmicro_primitive {
    promotion int extend_sign(int) = word extend_sign(word);

    inline dint to_dint_se(int i) { return dint(i,extend_sign(i)); }
    inline dint to_dint_ze(int i) { return dint(i,0); }
    inline int to_int(dint w)     { return w.lo; }
}
```

In a second step, promotion to an inline function is used to define these conversions.

```
promotion operator signed long(int)        = dint to_dint_se(word);
promotion operator signed long(unsigned)   = dint to_dint_ze(word);
promotion operator unsigned long(int)      = dint to_dint_se(word);
promotion operator unsigned long(unsigned) = dint to_dint_ze(word);
```

The opposite conversions from a 32 bit integer type to a 16 bit integer type must simply return the `lo` part of the `dint` struct. Again promotion to an inline function is used to define these conversions.

```
promotion operator int(signed long)        = int to_int(dint);
promotion operator int(unsigned long)      = int to_int(dint);
promotion operator unsigned(signed long)   = int to_int(dint);
promotion operator unsigned(unsigned long) = int to_int(dint);
```

6.3 Bitwise logical operators

The Bitwise logical operators are easy to implement. Let us take bitwise AND as example. First, a double-precision routine is defined at the primitive level, in which the `andw()` primitive is called for the `lo` and `hi` parts of the `dint`.

```
namespace tmicro_primitive {
    inline dint l_and(dint a, dint b) {
        return dint(a.lo & b.lo, a.hi & b.hi);
    }
}
```

Next, the `&` operators for `signed long` and `unsigned long` are promoted to this inline function.

```
promotion signed long operator&(signed long,signed long) = dint l_and(dint,dint);
promotion unsigned long operator&(unsigned long,unsigned long) = dint l_and(dint,dint);
```

The same principle is used to define the operators `|`, `^` and `~`. The following table shows a C function that uses the `&` operator, and the generated assembly code.

C function	*Assembly subroutine*
`long test_l_and(long a, long b)`	`rtd`
`{`	`and r1,r3,r5`
`    return a & b;`	`and r0,r2,r4`
`}`	

6.4 Addition and subtraction

Addition and subtraction are a bit more complicated as we need to take care of carry propagation.

First, versions with `int` arguments are defined for the additive primitives. Next, a double-precision routine `l_add` is defined.

```
namespace tmicro_primitive {

    promotion unsigned add (unsigned,unsigned,uint1&)
        property(duplicate_at_using_opn2)
                                 = word add (word,word,uint1&);
    promotion unsigned addc(unsigned,unsigned,uint1,uint1&)
                                 = word addc(word,word,uint1,uint1&);

    inline dint l_add(dint a, dint b) {
        dint r;
        uint1 carry, carry2;
        r.lo = add(a.lo, b.lo,carry);
        r.hi = addc(a.hi, b.hi,carry,carry2);
        return r;
    }
}
```

As can be seen, the long addition is implemented in two steps, using the `add()` and `addc()` primitive functions. The carry output of `add()` is saved in the variable `carry`, which is used as input of `addc()` (see also figure 6.1).

Since there is only one `CB` register, it is not possible to execute two long additions in an overlapping way, where first the two `add` operations are executed followed by the two `addc` operations. This would require that two carry values are stored in `CB`. To prevent this situation, the `duplicate_at_using_opn2` property is given to the `add` operation.

Note that `addc()` also produces a carry out, which is assigned to `carry2`. In this double-precision routine, `carry2` is discarded. In higher precision routines, `carry2` can be used as input to the next `addc()`. As a

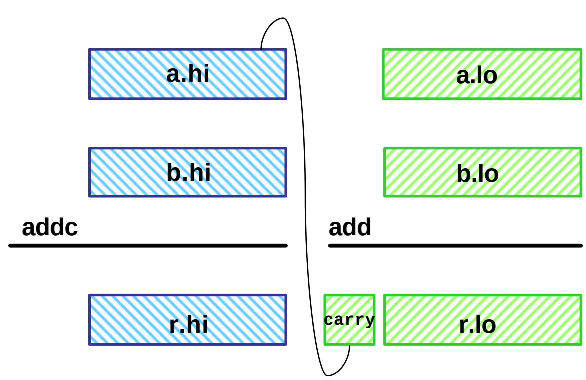

Figure 6.1: The double-precision addition.

final step, the + operators for `signed long` and `unsigned long` are promoted to this inline function.

```
promotion signed long operator+(signed long,signed long)
                                          = dint l_add(dint,dint);
promotion unsigned long operator+(unsigned long,unsigned long)
                                          = dint l_add(dint,dint);
```

The same principle is used to define the - operator, but then using the `sub()` and `subb()` primitives. The following table shows a C function that uses the + operator, and the generated assembly code.

C function	*Assembly subroutine*
`long test_l_add(long a, long b)`	`rtd`
`{`	`add r0,r2,r4`
`    return a + b;`	`addc r1,r3,r5`
`}`	

6.5 Multiplication

The principle of a signed double-precision multiplication is shown in figure 6.2. Three partial products must be computed and added to obtain the result.

- The first partial product is the multiplication of the unsigned values `a.lo` and `b.lo`.
- The second partial product is the multiplication of the signed value `a.hi` and the unsigned value `b.lo`.
- The third partial product is the multiplication of the unsigned value `a.lo` and the signed value `b.hi`.

Note that the 32×32 bit multiplication yields a 64 bit result, but to implement the C multiplication operators we must retain the lower 32 bits only. Therefore the high part of the second and third partial products does not contribute to the end result. Therefore also, it is irrelevant if the operands of the second and third multiplications are signed or unsigned. In principle there is a fourth partial product (the multiplication of the signed value `a.hi` and the signed value `b.hi`), but also this partial product does not contribute to the end result.

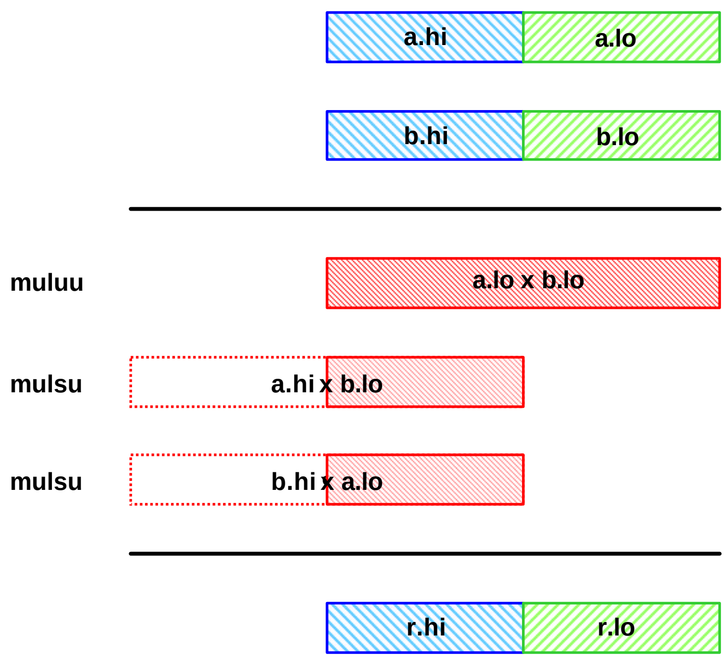

Figure 6.2: The double-precision signed multiplication.

The `muluu` instruction performs an unsigned $\times$ unsigned multiplication and is used to compute the first partial product.

muluu rr,rs: $\quad$ (PH,PL) $\leftarrow$ R[r]$_{\text{unsigned}}$ $\times$ R[s]$_{\text{unsigned}}$

The macl instruction performs a signed × signed multiplication and accumulates the lower 16 bits of the product to the PH register (mapped to the r7 register). It is used to compute and add the second and third partial products.

macl rr,rs: $\quad$ (PH) $\leftarrow$ (PH) + (word)(R[r]$_{\text{signed}}$ $\times$ R[s]$_{\text{signed}}$)

The multiplier primitive muluu() produces the 32 bit product as two 16 bit word reference arguments. First, promotion is used to define an integer version of muluu():

```
promotion void muluu(int,int,int&,int&)
        = void muluu(word,word,word&,word&);
```

The double-precision multiplication algorithm can then be programmed as follows:

```
namespace tmicro_primitive {
    inline dint l_mul(dint a, dint b) {
        int x, y;
        muluu(a.lo,b.lo,x,y);
        y += a.hi * b.lo;
        y += b.hi * a.lo;
        return dint(x,y);
    }
}
```

It can be that one of the parts of the long value, a.lo or bo.lo is zero. When that can be determined at compile time, the muluu() intrinsic for the first product need not be generated.

```
namespace tmicro_primitive {
    inline dint l_mul(dint a, dint b) {
        int x=0, y=0;
        if (!chess_manifest(a.lo==0) &&
            !chess_manifest(b.lo==0))
            muluu(a.lo,b.lo,x,y);
        y += a.hi * b.lo;
        y += b.hi * a.lo;
        return dint(x,y);
    }
}
```

The second and third products do not need any chess_manifest() test, as they are expressed in terms of built-in operators, which are cleaned up automatically.

Finally, the * operators for signed long and unsigned long are promoted to this inline function.

```
promotion   signed long operator*(  signed long,  signed long)
                                          = dint l_mul(dint,dint);
promotion unsigned long operator*(unsigned long,unsigned long)
                                          = dint l_mul(dint,dint);
```

Additionally, an intrinsic long multiplication function lmul is defined. This function takes two short (16-bit) int operands and returns the full precision (32-bit) long product.

```
namespace tmicro_primitive {
    inline dint di_mul(int a, int b)
    {
        int x, y;
        mulss(a,b,x,y);
        return dint(x,y);
    }
}

promotion long lmul(int,int) = dint di_mul(int,int);
```

The following table shows a C function that uses the * operator, and the generated assembly code.

C function	*Assembly subroutine*

`long test_l_mul(long a, long b)`
`{`
`    return a * b;`
`}`
	`muluu (r6,r7),r2,r4`
	`macl (r7),r3,r4`
	`macl (r7),r5,r2`
	`rtd`
	`mv r0,r6`
	`mv r1,r7`

6.6 Compare operators

In a double-precision compare, first the most significant parts are compared. If the result cannot be determined then the least significant parts are also compared. Consider the case of a double-precision signed less-than compare.

```
namespace tmicro_primitive {
    inline bool lts(dint a, dint b)
    {
        if ((signed)a.hi < (signed)b.hi)
            return true;
        else
            if (a.hi == b.hi)
                return a.lo < b.lo;
            else
                return false;
    }
}
```

First, a signed compare is done on the `hi` fields. This may already decide the outcome of the long compare. In case the `hi` fields are equal, an unsigned less-than compare of the `lo` fields will produce the outcome. In a similar way, an `les()` function is defined to implement the less-than-or-equal signed compare.

As for the previous operators, < and <= operators for `signed long` is promoted to these inline functions.

```
promotion bool operator< (signed long a, signed long b) = bool lts(dint,dint);
promotion bool operator<=(signed long a, signed long b) = bool les(dint,dint);
```

The complementary operators > and >= are defined by reversing the operands.

```
inline bool operator> (signed long a, signed long b) { return b < a; }
inline bool operator>=(signed long a, signed long b) { return b <= a; }
```

The unsigned compare operators are implemented in a similar way. The following table shows a C function that uses the < operator, and the generated assembly code.

C function	*Assembly subroutine*

`int test_l_lts(long a, long b)`
`{`
`    return a < b ? 1 : 0;`
`}`
	`lt r2,r4`
	`mvib r0,1`
	`jcr 6`
	`ne r2,r4`
	`jcr 1`
	`ltu r1,r3`
	`jcr 2`
	`rtd`
	`mvib r0,0`
	`nop`
	`rt`

6.7 Equal operators

The double-precision equal and in-equal routines are defined as follows.

```
namespace tmicro_primitive {
    inline bool eq(dint a, dint b)
    {
        return a.hi == b.hi && a.lo == b.lo;
    }

    inline bool ne(dint a, dint b)
    {
        return a.hi != b.hi || a.lo != b.lo;
    }
}
```

They can be used to implement both the signed and unsigned versions of the == and != operators. functions.

```
promotion bool operator==(signed long a, signed long b) = bool eq(dint,dint);
promotion bool operator!=(signed long a, signed long b) = bool ne(dint,dint);

promotion bool operator==(unsigned long a, unsigned long b) = bool eq(dint,dint);
promotion bool operator!=(unsigned long a, unsigned long b) = bool ne(dint,dint);
```

6.8 Shift operators

The principle of a double-precision (logical) left shift is shown in figures 6.3 and 6.4. Figure 6.3 shows the case where the shift factor f is less than 16. Both the `lo` and `hi` fields are shifted to the left over f bits. A group of f bits carries over from the `lo` field to the `hi` field.

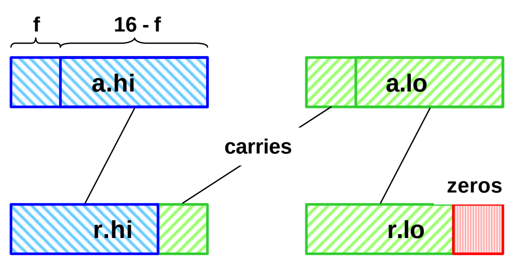

Figure 6.3: The double-precision left shift with shift factor less than 16.

Figure 6.4 shows the case where the shift factor f is at least 16. A group of $32 - f$ bits carries over from the `lo` field to the `hi` field of the result. The `lo` part of the result can be set to zero. The inline

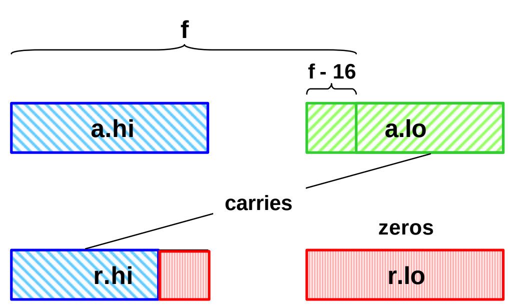

Figure 6.4: The double-precision left shift with a shift factor of 16 or more.

function `l_lsl()` that models this shift has a `dint` operand and return type; but that internally, the `hi` and `lo` fields are converted to `uint`, thus selecting the unsigned shift operators << and >>. Notice that `chess_dont_warn_range()` pragmas are used to disable the warnings the C front end would normally generate when a manifest shift factor that is greater or equal to 16 is applied to an 16 bit `int`.

```
namespace tmicro_primitive {
    inline dint l_lsl(dint a, int f)
    {
        unsigned carries;
        dint r;
        if (f == 0) return a;
        if (f < 16) {
            carries = a.lo >> (16 -f);
            r.lo = chess_dont_warn_range(a.lo << f);
            r.hi = chess_dont_warn_range(a.hi << f) | carries;
        }
        else {  // f >= 16
            carries = a.lo << (f - 16);
            r.lo = 0;
            r.hi = carries;
        }
        return r;
    }
}
```

The `lsl()` function can be used to implement signed and unsigned left shift.

```
promotion signed long operator<<(signed long,int) = dint l_lsl(dint,int);
promotion unsigned long operator<<(unsigned long,int) = dint l_lsl(dint,int);
```

In a similar fashion, an arithmetic shift right `l_asr()` function and a logical shift right `l_lsr()` function can be defined, and used to implement the signed and unsigned right shifts.

```
promotion signed long operator>>(signed long,int) = dint l_asr(dint,int);
promotion unsigned long operator>>(unsigned long,int) = dint l_lsr(dint,int);
```

The following table shows a C function that uses the << operator, and the generated assembly code.

C function	*Assembly subroutine*
`long test_l_lsl(long a, int b)`	`mvib r1,0`
`{`	`eq r4,r1`
`    return a << b;`	`mv r0,r2`
`}`	`mv r1,r3`
	`jcr 12`
	`mvib r3,16`
	`lt r4,r3`
	`jcr 4`
	`mvib r1,-16`
	`add r1,r4,r1`
	`rtd`
	`lsl r1,r0,r1`
	`mvib r0,0`
	`sub r3,r3,r4`
	`lsr r3,r0,r3`
	`lsl r1,r1,r4`
	`lsl r0,r0,r4`
	`or r1,r1,r3`
	`rt`

6.9 Division and modulo

The division and modulo operators are mapped onto called functions. This case is explained in the CHESS modelling manual [5].

7

Intrinsic functions and rewrite rules for conditional assignments

The TMICRO core supports instructions to compute the minimum and the maximum of two values. TMICRO also has a conditional register move instruction that can be used to implement the conditional operator `? :`. We will first introduce intrinsic functions that make these instructions accessible in the C code. Next, we will introduce rewrite rules that transform conditional assignments into these instructions.

7.1 Intrinsic min, max and select functions

The intrinsic functions `min_()`, `max_()` and `select()` are defined for the signed integer type by promotion to the corresponding primitive function. For `select()`, also an unsigned version is defined.

```
promotion int min_(int,int) property(min) = word minw(word,word);
promotion int max_(int,int) property(max) = word maxw(word,word);
promotion int select(bool,int,int) property(select)
                               = word select(bool,word,word);
promotion unsigned select(bool,unsigned,unsigned) property(select)
= word select(bool,word,word);
```

Note that an underscore is appended in the function names `min_()` and `max_()`. This is to avoid conflicts with the preprocessor macros `min` and `max` on certain systems.

With the `property(min)` and `max` annotations, the behaviour of these intrinsics is specified. The C front end uses this information to apply constant folding.

When the `property(select)` is specified the compiler front end will implement conditional assignments by means of the `select()` operation. There is a CHESS property `small_select_threshold` (see [5]), which is set to 3 for TMICRO.

When the `min_()`, `max_()` and `select()` functions are used in expressions, the `min`, `max` and `sel` instructions are generated, as shown in the following examples.

C function	*Assembly subroutine*

`int i_max(int a, int b)`
`{`
`    return max_(a,b);`
`}`
	`max r0,r1,r2`
	`rt`

C function	*Assembly subroutine*

`int i_select(int a, int b, int c, int d)`
`{`
`    return select(a<b,c,d);`
`}`
	`lt r1,r2`
	`sel r0,r3,r4`
	`rt`

C function	*Assembly subroutine*
`int i_ite(int a, int b, int c)`	`mvib r4,0`
`{`	`sub r0,r1,r2`
`    if (c)`	`ne r3,r4`
`        return a + b;`	`rtd`
`    else`	`add r1,r1,r2`
`        return a - b;`	`sel`
`}`	`r0,r1,r0`

7.2 Rewrite rules for min and max

The concept of CHESS rewrite rules can be applied to automatically search for opportunities in the application program where the conditional assignment instructions can be used.

As an example, the following rewrite rules specify that a conditional selection of the smallest of two values maps onto the `min` instruction.

```
chess_rewrite int min_lt_rule(int a, int b) {
    return a < b ? a : b;
} -> {
    return min_(a,b);
}
chess_rewrite int min_le_rule(int a, int b) {
    return a <= b ? a : b;
} -> {
    return min_(a,b);
}
```

The example below shows that this rewrite rule not only applies to the C `? :` operator, but also to conditional assignments that are programmed using an `if` statement.

C function	*Assembly subroutine*
`int r_min(int a, int b)`	`min r0,r1,r2`
`{`	`rt`
`    int x = 0;`	
`    if (a < b)`	
`        x = a;`	
`    else`	
`        x = b;`	
`    return x;`	
`}`	

7.3 Rewrite rules for multiplication

In numerical applications, it is often so that `short` operands are multiplied, and then accumulated using `long` precision. In order to achieve the full precision product, the operands are first converted to `long`.

```
short a, b;
long l;
l = (long)a * (long)b;
```

In this case, it is not needed to execute a full three instruction long multiplication (as defined in `l_mul()`, it suffices to execute the `lmul` intrinsic. Following rewrite rule will automatically convert the above expression into a call to `lmul()`.

```
chess_rewrite long mul_rule(short a, short b) {
    return (long)a * (long)b;
} -> {
    return lmul(a,b);
}
```

8

The tmicro software libraries

8.1 The processor library

The processor library contains the code for the called functions that are used in the compiler header file. The following source files are used to build this library:

- `tmicro_long_div.c`: This file contains the definition of the `div_called()` functions for types `long` and `unsigned long` (see section 3.9).
- `tmicro_init.s`: This file contains the initialization code that is executed before the `main()` function. (see section 8.3).

The library `libtmicro.a` can be built using the `tmicro/lib/libtmicro.prx` project file.

8.2 The runtime C library

A runtime C Library is provided that is compliant with the ISO/IEC C99 standard. This library has been ported to the TMICRO processor. Following header files are supported (in compliance with the requirements of a freestanding implementation of the ISO/IEC C99 standard).

- `<float.h>`

  Limits and parameters of the standard floating-point types. This file is only relevant for processors that support the float and/or double types. It is not relevant on the TMICRO core.
- `<iso646.h>`

  Alternative spellings for some tokens.
- `<limits.h>`

  Limits and parameters of the standard integer types.
- `<stdarg.h>`

  Macros for advancing through variable length function argument lists.
- `<stdbool.h>`

  chess treats the type `bool` and the values `true` and `false` as built-in types. Therefore, the `stdbool.h` file is empty.
- `<stddef.h>`

  This file defines the `ptrdiff_t`, `size_t` and `wchar_t` types, the `NULL` macro, and the `offsetof()` macro.

- `<stdint.h>`

  Specific width integer types and the limits and parameters of these types.

- `<ctype.h>`

  Functions for the classifying and mapping of characters.

- `<errno.h>`

  Macros related to the reporting of error conditions.

- `<stdio.h>`

  Types, macros, and functions for performing input and output. The IO functions are implemented using the hosted IO functionality of Checkers. The following functions are supported:

  ```
  fopen(), fclose(), freopen(), fflush(), fseek(), ftell(), feof(),
  ferror(), clearerr(), perror(), vfprintf(), fprintf(), printf(),
  vfscanf(),  fscanf(), scanf(), fputc(), fputs(), fgetc((), fgets(),
  gets(), ungetc(), fwrite(), fread(), fwrite_word(), fread_word(),
  remove(), rename(), tmpfile(), tmpnam(), sprintf(), snprintf(),
  vsprintf(), vsnprintf(), sscanf(), vsscanf()
  ```

- `<stdlib.h>`

  General utility functions. The following functions are supported:

  ```
  strtol(), strtoul(), atoi(), atol(), rand(), srand(), exit(), qsort()
  ```

- `<string.h>`

  Functions for the manipulation of character arrays.

8.3 Startup code

It is common that, before the `main()` function is started, some initialization code is executed. This code is typically programmed in assembly. It resides in the file `tmicro_init.s` and is archived in `libtmicro.a`. The code coexists with linker directives in `tmicro.bcf`.

On TMICRO the initialization code handles the initialization of the stack pointer, the allocated memory space for the arguments of the `main()` function, and starting the `main()` function.

`tmicro_init.s`

```
; initialize SP, then jump to main
.undef global text _main
.text global 0 tmicro_init
        mvi sp, _sp_start_value_DM
        j _main

; area to load main(argc,char* argv[]) arguments
.bss global 0 _main_argv_area DM 256
```

`tmicro.bcf`

```
_symbol tmicro_init 0             // Start with tmicro_init.s code
_entry_point tmicro_init

_stack DM 1 0x0fff                // Avoid address zero for C locals.

_always_include _main_argv_area   // Reserve space for main() arguments
```

Initialization of the stack pointer

- The stack region is allocated at the lower end of DM. This is done in the linker configuration file, with the _stack directive. Note that address zero is not used in order to avoid that an automatic variable is allocated at the null pointer address.
- The linker automatically defines a symbol to the start address of the stack. For the DM memory on TMICRO, this symbol is called _sp_start_value_DM. In the initialization code, it is used to set the SP register.

Allocation of argv area

- The symbol _main_argv_area is defined as a global symbol of size 256 in the assembly code.
- Since this symbol is not referenced by the application code, an _always_include linker directive is needed

Jump to the main() function

- The linker configuration file contains an _entry_point directive to designate the initialization code as the start of the program.
- To transition from the initialization code to the main() function, a jump instruction is programmed at the end of the initialization code.

Another application of the initialisation code, is the definition of the interrupt vector table. On TMICRO, the interrupt vector addresses are the first 8 even addresses in the program memory. This is explained in the document *Implementing Interrupts on the Tmicro Core* (see [6]).

8.4 Specifying default libraries

The TMICRO processor library libtmicro.a and the runtime C library libc.a are added to the default list of libraries that are added to new software projects. This list can be defined in the CHESSDE *Settings*, by selecting *Compilation* in the left pane, and then selecting *Linking* in the center pane. There the following settings are specified:

- Library path (-L) : *<PROCDIR>/runtime/lib <PROCDIR>*
- Libraries in library path (-l) : *c tmicro*

Bibliography

[1] *Chess Compiler User manual*. Synopsys, September 2016. Version L-2016.09.

[2] *The nML Processor Description Language*. Synopsys, September 2016. Version L-2016.09.

[3] *Checkers ISS Interface manual*. Synopsys, September 2016. Version L-2016.09.

[4] *Go User manual, nML to HDL translation*. Synopsys, September 2016. Version L-2016.09.

[5] *Chess Compiler Processor Modeling manual*. Synopsys, September 2016. Version L-2016.09.

[6] *Implementing Interrupts on the Tmicro Core*. Synopsys, March 2015. Version J-2015.03.