The xr16 Specifications

Version 0.6, December, 1999

(Work-in-progress)

Table of Contents

Editor: Jan Gray

1 Introduction

xr16 is a simple 16-bit reduced instruction set computer designed to run integer-only C programs. The xr16 design is optimized for an area-efficient pipelined implementation in a field-programmable gate array and other gate- and interconnect-constrained environments. It also features compact 16-bit instructions.

This document provides specifications for the xr16:

- instruction set architecture;
- assembly language;
- C language calling conventions;
- systems programming architecture; and
- the first xr16 implementation.

2 Instruction Set Architecture Specification

2.1 State

The architectural (programmer-visible) state consists of the program counter, the register file, and memory, which stores both data and instructions.

2.1.1 Data formats

Data are represented as 8-bit bytes, 16-bit words, or 32-bit long words:

```
typedef unsigned char Byte; /*  8 bits */
typedef unsigned Word;      /* 16 bits */
typedef unsigned long Long; /* 32 bits */
```

2.1.2 Program counter

The 16-bit program counter pc stores the memory address of the next instruction to execute. The least-significant bit of pc, pc_0, is hardwired to 0.

```
Word pc;
```

On reset, pc is initialized to 0. As each instruction is executed, pc is automatically incremented by 2, so instruction execution proceeds sequentially through memory. The branch instructions (br beq bne bc bnc bv bnv blt bge ble bgt bltu bgeu bleu bgtu) may add a *branch displacement* to pc and effect a program transfer. The jump instructions (jal call) load pc with the target effective address and store the return address (the address of the next instruction) in a register.

On interrupt, pc is set to the address of the interrupt handler, and the address of the interrupted instruction is stored in r14.

Therefore a jump instruction, or an interrupt, are the only mechanisms that read pc into a register.

2.1.3 Register file

xr16 has sixteen general purpose 16-bit integer registers, r0 through r15:

```
Word r[16];
```

r0 is hardwired to 0. It always reads as 0, and writes to r0 are ignored.

r13 is also called sp.

r14 is reserved as the interrupt return address and should neither be read nor written by application code.

r15 is a fully general purpose register, but it is also the implicit destination register of the call instruction. Therefore it is typically used as the function return address register.

For more information on register usage conventions, see section 4, Calling Conventions.

Within the register file, a byte or word is stored in one 16-bit register. A long word is stored in a pair of registers, *rx* and *rx'*. The less-significant 16-bits of the long word are stored in *rx* and the more-significant 16-bits in *rx'*. (See 3.4.)

2.1.4 Memory

The application memory address space consists of an array of up to 2^{16} 8-bit bytes. Byte sequences in memory can also be interpreted as *big endian* 16-bit words and 32-bit long words.

```
Byte mem[];

load_byte(addr) := mem[addr];
load_word(addr) := (load_byte(addr) <<  8) | load_byte(addr + 1);
load_long(addr) := (load_word(addr) << 16) | load_word(addr + 2);
```

Word addresses must be even (divisible by two.) Long word addresses must be divisible by four.

2.1.5 Instructions

All xr16 instructions are 16-bit words stored in memory, in big endian format, at even addresses.

xr16 has no condition codes; however, some instruction sequences, such as (`imm`, `addi`, `bne`), that is, (immediate prefix, add immediate, branch not equal), are interlocked (not interruptible).

2.2 Instruction set summary

15 12	11 8	7 4	3 0	Instruction	Effect(s)
0	*rd*	*ra*	*rb*	**add** *rd,ra,rb*	r[rd] = r[ra] + r[rb];
1	*rd*	*ra*	*rb*	**sub** *rd,ra,rb*	r[rd] = r[ra] - r[rb];
2	*rd*	*ra*	*imm*	**addi** *rd,ra,imm*	r[rd] = r[ra] + immed;
3	*rd*	0	*rb*	**and** *rd,rb*	r[rd] = r[rd] ∧ r[rb];
3	*rd*	1	*rb*	**or** *rd,rb*	r[rd] = r[rd] ∨ r[rb];
3	*rd*	2	*rb*	**xor** *rd,rb*	r[rd] = r[rd] ⊕ r[rb];
3	*rd*	3	*rb*	**andn** *rd,rb*	r[rd] = r[rd] ∧ ~r[rb];
3	*rd*	4	*rb*	**adc** *rd,rb*	r[rd] = r[rd] + r[rb] + t;
3	*rd*	5	*rb*	**sbc** *rd,rb*	r[rd] = r[rd] - r[rb] - t;
4	*rd*	0	*imm*	**andi** *rd,imm*	r[rd] = r[rd] ∧ immed;
4	*rd*	1	*imm*	**ori** *rd,imm*	r[rd] = r[rd] ∨ r[rb];
4	*rd*	2	*imm*	**xori** *rd,imm*	r[rd] = r[rd] ⊕ r[rb];
4	*rd*	3	*imm*	**andni** *rd,imm*	r[rd] = r[rd] ∧ ~immed;
4	*rd*	4	*imm*	**adci** *rd,imm*	r[rd] = r[rd] + immed + t;
4	*rd*	5	*imm*	**sbci** *rd,ra,imm*	r[rd] = r[rd] - immed - t;
4	*rd*	6	*imm*	**srli** *rd,imm*	r[rd] = r[rd] >> immed;
4	*rd*	7	*imm*	**srai** *rd,imm*	r[rd] = (signed)r[rd] >> immed;
4	*rd*	8	*imm*	**slli** *rd,imm*	r[rd] = r[rd] << immed;
4	*rd*	9	*imm*	**srxi** *rd,imm*	r[rd] = (t << 16-immed) | (r[rd] >> immed)
4	*rd*	A	*imm*	**slxi** *rd,imm*	r[rd] = (r[rd] << immed) | (t << (immed-1));
5	*rd*	*ra*	*imm*	**lw** *rd,imm(ra)*	r[rd] = load_word(r[ra] + immed);
6	*rd*	*ra*	*imm*	**lb** *rd,imm(ra)*	r[rd] = $0_{15:8}$ || load_byte(r[ra] + immed);
8	*rd*	*ra*	*imm*	**sw** *rd,imm(ra)*	store_word(r[rd], r[ra] + immed);
9	*rd*	*ra*	*imm*	**sb** *rd,imm(ra)*	store_byte((Byte)r[rd], r[ra] + immed);
A	*rd*	*ra*	*imm*	**jal** *rd,imm(ra)*	target = r[ra] + immed; r[rd] = pc; pc = target;
B	0	*disp*		**br** *label*	pc += 2×sign_ext(disp) + 2;
B	2	*disp*		**beq** *label*	if (==) pc += 2×sign_ext(disp) + 2;
B	3	*disp*		**bne** *label*	if (!=) pc += 2×sign_ext(disp) + 2;
B	4	*disp*		**bc** *label*	if (carry_sum(,)) pc += 2×sign_ext(disp) + 2;
B	5	*disp*		**bnc** *label*	if (!carry_sum(,)) pc += 2×sign_ext(disp) + 2;
B	6	*disp*		**bv** *label*	if (overflow_sum(,)) pc += 2×sign_ext(disp) + 2;
B	7	*disp*		**bnv** *label*	if (!overflow_sum(,)) pc += 2×sign_ext(disp) + 2;
B	8	*disp*		**blt** *label*	if ((signed)<) pc += 2×sign_ext(disp) + 2;
B	9	*disp*		**bge** *label*	if ((signed)>=) pc += 2×sign_ext(disp) + 2;
B	A	*disp*		**ble** *label*	if ((signed)<=) pc += 2×sign_ext(disp) + 2;
B	B	*disp*		**bgt** *label*	if ((signed)>) pc += 2×sign_ext(disp) + 2;
B	C	*disp*		**bltu** *label*	if ((unsigned)<) pc += 2×sign_ext(disp) + 2;
B	D	*disp*		**bgeu** *label*	if ((unsigned)>=) pc += 2×sign_ext(disp) + 2;
B	E	*disp*		**bleu** *label*	if ((unsigned)<=) pc += 2×sign_ext(disp) + 2;
B	F	*disp*		**bgtu** *label*	if ((unsigned)>) pc += 2×sign_ext(disp) + 2;
C	*imm12*			**call** *function*	r[15] = pc; pc = $\mathrm{imm12}_{11:0}$ || $0_{3:0}$;
D	*imm12*			**imm** *imm12*	$\mathit{immed(next)}_{15:4}$ = imm12;

2.3 Instruction specification format

Each instruction is specified below. Each specification provides the instruction name, assembly format, description, and pseudo-code of its operation:

Instruction

15 12	11 8	7 4	3 0
op	*rd*	*ra*	*rb*

Format: **instruction** *rd*,*ra*,*rb*

Description: Describes the instruction semantics and defines situations in which the behavior of the instruction or instruction sequence is undefined. If applicable, notes that the instruction is interlocked.

Operation:

```
/* Describes the instruction's effects upon the
 * architected state and invisible state.
 *
 * Operations upon the architected state
 * are in boldface.
 */
```

Comment: Optional comments, if any.

2.3.1 Invisible state

Besides the architected state, the instruction set specification employs the following invisible state to model machine behavior. This state is not necessarily present in any given implementation, but all implementations behave *as if* it were.

```
typedef Word Bit;     /* 1 bit */

Word a;               /* prefix A operand (as if add) */
Word b;               /* prefix B operand (as if add) */
Bit  carry_out;       /* prefix carry out */
Bit  shift_out;       /* prefix shift out */

Bit  has_imm_prefix;  /* pending imm prefix */
Word imm_prefix;      /* 12-bit immediate prefix */

Bit  can_interrupt;   /* interrupt can occur after instruction */

Word t;               /* temporary */
Word immed;           /* temporary immediate value */
```

2.3.2 Operation prefixes

Each instruction specification below uses an *operation prefix* to advance certain aspects of the architected and invisible state.

Operation *pre*:

```
pc = pc + 2;
r[0] = 0;
has_imm_prefix = false;
can_interrupt = true;
```

Operation *pre_sign_ext_imm*:

```
if (has_imm_prefix)
    immed = imm_prefix$_{11:0}$ || imm$_{3:0}$;
else
    immed = sign_ext(imm$_{3:0}$);
pre
```

Operation *pre_byte_offset_imm*:

```
if (has_imm_prefix)
    immed = $imm\_prefix_{11:0}$ || $imm_{3:0}$;
else
    immed = $0_{15:4}$ || $imm_{3:0}$;
*pre*
```

Operation *pre_word_offset_imm*:

```
if (has_imm_prefix)
    immed = $imm\_prefix_{11:0}$ || $imm_{3:0}$;
else
    immed = $0_{15:5}$ || $imm_{0}$ || $imm_{3:1}$ || 0;
*pre*
```

2.4 Instruction specifications

2.4.1 Add

15 12	11 8	7 4	3 0
0	*rd*	*ra*	*rb*

Format: `add` *rd,ra,rb*

Description: Add the two register operands *ra* and *rb* and store the result in *rd*.

Behavior is undefined if *rb* is the result of the previous instruction.

This instruction is interlocked.

Operation:

```
pre
a = r[ra];
b = r[rb];
can_interrupt = false;
r[rd] = r[ra] + r[rb];
```

2.4.2 Add immediate

15 12	11 8	7 4	3 0
2	*rd*	*ra*	*imm*

Format: `addi` *rd,ra,imm*

Description: Add the register operand *ra* and the immediate operand and store the result in *rd*.

This instruction is interlocked.

Operation:

```
pre_sign_ext_immed
a = r[ra];
b = immed;
can_interrupt = false;
r[rd] = r[ra] + immed;
```

2.4.3 Add carry

15 12	11 8	7 4	3 0
3	*rd*	4	*rb*

Format: `adc` *rd,rb*

Description: Add the previous instruction's carry result and the two register operands *rd* and *rb,* and store the result in *rd*.

Behavior is undefined if *rb* is the result of the previous instruction.

This instruction is interlocked.

Operation:

```
(needs work)
pre
a = r[rd];
b = r[rb];
t = carry_out;
carry_out = carry_sum(r[rd], r[rb], t);
can_interrupt = false;
r[rd] = r[rd] + r[rb] + t;
```

2.4.4 Add carry immediate

15 12	11 8	7 4	3 0
4	*rd*	4	*imm*

Format: **adci** *rd*,*imm*

Description: Add the previous instruction's carry result and the register operand *rd* and the immediate operand, and store the result in *rd*.

This instruction is interlocked.

Operation:

```
(needs work)
pre_sign_ext_immed
a = r[rd];
b = immed;
t = carry_out;
carry_out = carry_sum(r[rd], immed, t);
can_interrupt = false;
r[rd] = r[rd] + immed + t;
```

2.4.5 And

15 12	11 8	7 4	3 0
3	*rd*	0	*rb*

Format: **and** *rd*,*rb*

Description: *Bitwise and* the two register operands *rd* and *rb* and store the result in *rd*.

Behavior is undefined if *rb* is the result of the previous instruction.

Operation:

```
pre
r[rd] = r[rd] ∧ r[rb];
```

2.4.6 And immediate

15 12	11 8	7 4	3 0
4	*rd*	0	*imm*

Format: **andi** *rd*,*imm*

Description: *Bitwise and* the register operand *rd* and the immediate operand and store the result in *rd*.

Operation:

```
pre_sign_ext_immed
r[rd] = r[rd] ∧ immed;
```

2.4.7 And not

15 12	11 8	7 4	3 0
3	*rd*	3	*rb*

Format: **andn** *rd*,*rb*

Description: *Bitwise and* the register operand *rd* with the complement of the register operand *rb* and store the result in *rd*.

Behavior is undefined if *rb* is the result of the previous instruction.

Operation:

```
pre
r[rd] = r[rd] ∧ ~r[rb];
```

2.4.8 And not immediate

15 12	11 8	7 4	3 0
4	*rd*	3	*imm*

Format: **andni** *rd*,*imm*

Description: *Bitwise and* the register operand *rd* with the complement of the immediate operand and store the result in *rd*.

Operation:

```
pre_sign_ext_immed
r[rd] = r[rd] ∧ ~immed;
```

2.4.9 Branch

15 12	11 8	7 0
B	0	*disp*

Format: **br** *label*

Description: Unconditional branch to label.

Operation:

```
pre
pc += 2×sign_ext($disp_{7:0}$) + 2;
```

2.4.10 Branch on equal

15 12	11 8	7 0
B	2	*disp*

Format: **beq** *label*

Description: Branch to label if the operands of the branch prefix compare *equal*.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (a == -b)
    pc += 2×sign_ext($disp_{7:0}$) + 2;
```

2.4.11 Branch on not equal

15 12	11 8	7 0
B	3	*disp*

Format: **bne** *label*

Description: Branch to label if the operands of the branch prefix compare *not equal*.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (a != -b)
    pc += 2×sign_ext($disp_{7:0}$) + 2;
```

2.4.12 Branch on carry

15 12	11 8	7 0
B	4	*disp*

Format: **bc** *label*

Description: Branch to label if the sum of the operands of the branch prefix, interpreted as unsigned integers, produces a carry-out.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (carry_sum(a, b))
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.13 Branch on not carry

15 12	11 8	7 0
B	5	*disp*

Format: **bnc** *label*

Description: Branch to label if the sum of the operands of the branch prefix, interpreted as unsigned integers, does not produce a carry-out.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (!carry_sum(a, b))
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.14 Branch on overflow

15 12	11 8	7 0
B	6	*disp*

Format: **bv** *label*

Description: Branch to label if the sum of the operands of the branch prefix, interpreted as signed integers, overflows.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (overflow_sum(a, b))
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.15 Branch on not overflow

15 12	11 8	7 0
B	7	*disp*

Format: **bnv** *label*

Description: Branch to label if the sum of the operands of the branch prefix, interpreted as signed integers, does not overflow.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (!overflow_sum(a, b))
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.16 Branch on less than

15 12	11 8	7 0
B	8	*disp*

Format: **`blt`** *`label`*

Description: Branch to label if the first operand of the branch prefix is *less than* the second operand, treating both operands as *signed* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if ((signed)a < (signed)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.17 Branch on greater than or equal

15 12	11 8	7 0
B	9	*disp*

Format: **`bge`** *`label`*

Description: Branch to label if the first operand of the branch prefix is *greater than or equal to* the second operand, treating both operands as *signed* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if ((signed)a >= (signed)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.18 Branch on less than or equal

15 12	11 8	7 0
B	A	*disp*

Format: **`ble`** *`label`*

Description: Branch to label if the first operand of the branch prefix is *less than or equal to* the second operand, treating both operands as *signed* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if ((signed)a <= (signed)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.19 Branch on greater than

15 12	11 8	7 0
B	B	*disp*

Format: **`bgt`** *`label`*

Description: Branch to label if the first operand of the branch prefix is *greater than* the second operand, treating both operands as *signed* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if ((signed)a > (signed)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.20 Branch on less than unsigned

15 12	11 8	7 0
B	C	*disp*

Format: **bltu** *label*

Description: Branch to label if the first operand of the branch prefix is *less than* the second operand, treating both operands as *unsigned* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (a < (Word)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.21 Branch on greater than or equal unsigned

15 12	11 8	7 0
B	D	*disp*

Format: **bgeu** *label*

Description: Branch to label if the first operand of the branch prefix is *greater than or equal to* the second operand, treating both operands as *unsigned* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (a >= (Word)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.22 Branch on less than or equal unsigned

15 12	11 8	7 0
B	E	*disp*

Format: **bleu** *label*

Description: Branch to label if the first operand of the branch prefix is *less than or equal to* the second operand, treating both operands as *unsigned* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (a <= (Word)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.23 Branch on greater than unsigned

15 12	11 8	7 0
B	F	*disp*

Format: **bgtu** *label*

Description: Branch to label if the first operand of the branch prefix is *greater than* the second operand, treating both operands as *unsigned* 16-bit integers.

Behavior is undefined if the previous instruction is not a branch prefix.

Operation:

```
pre
if (a > (Word)-b)
   pc += 2×sign_ext(disp$_{7:0}$) + 2;
```

2.4.24 Call

15 12	11 0
C	*imm12*

Format: **call** *function*

Description: Jump to the function specified by *imm12* and store the return address (the address of the next instruction) in r15.

Operation:

```
pre
r[15] = pc;
pc = $imm12_{11:0}$ || $0_{3:0}$;
```

2.4.25 Immediate prefix

15 12	11 0
D	*imm12*

Format: **imm** *imm12*

Description: Establish the upper 12 bits of the 16-bit immediate operand of the instruction that follows.

This instruction is interlocked.

Operation:

```
pre
has_imm_prefix = true;
imm_prefix = imm12;
can_interrupt = false;
```

2.4.26 Jump and link

15 12	11 8	7 4	3 0
A	*rd*	*ra*	*imm*

Format: **jal** *rd*,*imm*(*ra*)

Description: Jump to the effective address formed by adding the register operand *ra* and the immediate operand, and store the return address (the address of the next instruction) in *rd*.

Operation:

```
pre_word_off_imm
target = (r[ra] + immed) & ~1;
r[rd] = pc;
pc = target;
```

2.4.27 Load word

15 12	11 8	7 4	3 0
5	*rd*	*ra*	*imm*

Format: **lw** *rd*,*imm*(*ra*)

Description: Load the word at the effective address formed by adding the register operand *ra* and the immediate operand, and store it in *rd*.

Operation:

```
pre_word_off_imm
r[rd] = load_word(r[ra] + immed);
```

2.4.28 Load byte

15 12	11 8	7 4	3 0
6	*rd*	*ra*	*imm*

Format: **lb** *rd,imm(ra)*

Description: Load the byte at the effective address formed by adding the register operand *ra* and the immediate operand, zero-extend the byte, and store the result in *rd*.

Operation:

```
pre_byte_off_imm
r[rd] = 0_{15:8} || load_byte(r[ra] + immed);
```

2.4.29 Or

15 12	11 8	7 4	3 0
3	*rd*	1	*rb*

Format: **or** *rd,rb*

Description: *Bitwise or* the two register operands *rd* and *rb* and store the result in *rb*.

Behavior is undefined if *rb* is the result of the previous instruction.

Operation:

```
pre
r[rd] = r[rd] ∨ r[rb];
```

2.4.30 Or immediate

15 12	11 8	7 4	3 0
4	*rd*	1	*imm*

Format: **ori** *rd,imm*

Description: *Bitwise or* the register operand *rd* and the immediate operand and store the result in *rd*.

Operation:

```
pre_sign_ext_immed
r[rd] = r[rd] ∨ immed;
```

2.4.31 Shift left logical immediate

15 12	11 8	7 4	3 0
4	*rd*	8	*imm*

Format: **slli** *rd,imm*

Description: Shift the destination register *rd* left by the number of bits specified by the immediate operand.

Behavior is undefined if the immediate operand does not equal 1.

Operation:

```
pre_sign_ext_immed
shift_out = r[rd]_15;
r[rd] = r[rd] << immed;
```

2.4.32 Shift left extended immediate

15 12	11 8	7 4	3 0
4	*rd*	A	*imm*

Format: **`slxi`** *`rd,imm`*

Description: Shift the destination register *rd*, concatenated with the previous instruction's shift extension, left by the number of bits specified by the immediate operand.

Behavior is undefined if the immediate operand does not equal 1, or if the previous instruction is not a shift instruction.

Operation:

```
pre_sign_ext_immed
t = shift_out;
shift_out = r[rd]_15;
r[rd] = (r[rd] << immed) | (t << (immed-1));
```

2.4.33 Shift right logical immediate

15 12	11 8	7 4	3 0
4	*rd*	6	*imm*

Format: **`srli`** *`rd,imm`*

Description: Logical shift the destination register *rd* right by the number of bits specified by the immediate operand. Vacated more-significant bits are filled with 0.

Behavior is undefined if the immediate operand does not equal 1.

Operation:

```
pre_sign_ext_immed
shift_out = r[rd]_0;
r[rd] = r[rd] >> immed;
```

2.4.34 Shift right arithmetic immediate

15 12	11 8	7 4	3 0
4	*rd*	7	*imm*

Format: **`srai`** *`rd,imm`*

Description: Arithmetic shift the destination register *rd* right by the number of bits specified by the immediate operand. Vacated more-significant bits are filled with the destination register sign bit.

Behavior is undefined if the immediate operand does not equal 1.

Operation:

```
pre_sign_ext_immed
shift_out = r[rd]_0;
r[rd] = (signed)r[rd] >> immed;
```

2.4.35 Shift right extended immediate

15 12	11 8	7 4	3 0
4	*rd*	9	*imm*

Format: **srxi** *rd,imm*

Description: Logical shift the previous instruction's shift extension concatenated with the destination register *rd*, right by the number of bits specified by the immediate operand.

Behavior is undefined if the immediate operand does not equal 1, or if the previous instruction is not a shift instruction.

Operation:

```
pre_sign_ext_immed
t = shift_out;
shift_out = r[rd]_0;
r[rd] = (t << 16-immed) | (r[rd] >> immed)
```

2.4.36 Store word

15 12	11 8	7 4	3 0
8	*rd*	*ra*	*imm*

Format: **sw** *rd,imm(ra)*

Description: Store the register operand *rd* to memory at the effective address formed by adding the register operand *ra* and the immediate operand.

Behavior is undefined if *rd* is the result of the previous instruction.

Operation:

```
pre_word_off_imm
store_word(r[rd], r[ra] + immed);
```

2.4.37 Store byte

15 12	11 8	7 4	3 0
9	*rd*	*ra*	*imm*

Format: **sb** *rd,imm(ra)*

Description: Store the least-significant byte of the register operand *rd* to memory at the effective address formed by adding the register operand *ra* and the immediate operand.

Behavior is undefined if *rd* is the result of the previous instruction.

Operation:

```
pre_byte_off_imm
store_byte(r[rd]_7:0, r[ra] + immed);
```

2.4.38 Subtract

15 12	11 8	7 4	3 0
1	*rd*	*ra*	*rb*

Format: **sub** *rd,ra,rb*

Description: Subtract the register operand *rb* from the register operand *ra* and store the result in *rd*.

Behavior is undefined if *rb* is the result of the previous instruction.

This instruction is interlocked.

Operation:

```
pre
a = r[ra];
b = -r[rb];
can_interrupt = false;
r[rd] = r[ra] - r[rb];
```

2.4.39 Subtract carry

15 12	11 8	7 4	3 0
3	*rd*	5	*rb*

Format: **sbc** *rd*,*rb*

Description: Subtract the previous instruction's carry result and the register operand *rb* from the register operand *rd,* and store the result in *rd*.

Behavior is undefined if *rb* is the result of the previous instruction.

This instruction is interlocked.

Operation:

```
(needs work)
pre
a = r[rd];
b = -r[rb];
t = carry_out;
carry_out = carry_sum(r[rd], r[rb], t);
can_interrupt = false;
r[rd] = r[rd] - r[rb] – t;
```

2.4.40 Subtract carry immediate

15 12	11 8	7 4	3 0
4	*rd*	5	*imm*

Format: **sbci** *rd*,*ra*,*imm*

Description: Subtract the previous instruction's carry result and immediate operand from the register operand *rd*, and store the result in *rd*.

This instruction is interlocked.

Operation:

```
(needs work)
pre_sign_ext_immed
a = r[rd];
b = -immed;
t = carry_out;
carry_out = carry_sum(r[rd], immed, t);
can_interrupt = false;
r[rd] = r[rd] - immed - t;
```

2.4.41 Xor

15 12	11 8	7 4	3 0
3	*rd*	2	*rb*

Format: **xor** *rd*,*rb*

Description: *Bitwise exclusive-or* the two register operands *rd* and *rb* and store the result in *rd*.

Behavior is undefined if *rb* is the result of the previous instruction.

Operation:

```
pre
r[rd] = r[rd] ⊕ r[rb];
```

2.4.42 Xor immediate

15 12	11 8	7 4	3 0
4	*rd*	2	*imm*

Format: **`xori`** *`rd`*`,`*`imm`*

Description: *Bitwise exclusive-or* the register operand *rd* and the immediate operand and store the result in *rd*.

Operation:

```
pre_sign_ext_immed
r[rd] = r[rd] ⊕ immed;
```

3 Assembly Language Specification

This section specifies the features and behavior of the xr16 assembler.

3.1 Transformations

3.1.1 Immediate prefix insertion

The assembler automatically inserts an `imm` immediate prefix instruction when necessary.

Instructions	`imm` not required	`imm` required
`addi andi ori xori andi adci sbci srli srai slli srxi slxi`	$-8 \le imm \le 7$	$imm < -8 \vee imm > 7$
`lb sb`	$0 \le imm \le 15$	$imm < 0 \vee imm > 15$
`lw sw jal`	$0 \le imm \le 30$	$imm < 0 \vee imm > 30$

The `imm` prefix encodes the most significant 12 bits of the immediate value, $imm_{15:4}$.

Examples:

```
addi r1,r1,0x1234   →  imm 0x123
                       addi r1,r1,4

addi r1,r1,8        →  imm 0
                       addi r1,r1,-8

lw r2,0x24(r1)      →  imm 0x002
                       lw r2,4(r1)
```

3.1.2 Result dependency data hazards, operand reordering, and nop insertion

When an assembly instruction sequence exhibits a result dependency data hazard that could cause undefined behavior (see specifications of `add`, `sub`, `and`, `or`, `xor`, `andn`, `adc`, `sbc`, `sw`, and `sb`), the assembler transforms the sequence to eliminate the data hazard.

First the assembler tries to eliminate the hazard by reordering operands, if possible:

```
addi r1,r1,1        →  addi r1,r1,1
add  r2,r3,r1          add  r2,r1,r3
```

If operand reordering is not feasible, the assembler inserts a nop:

```
addi r1,r1,1        →  addi r1,r1,1     →  addi r1,r1,1
add  r2,r1,r1          nop                 and  r0,r0
                       add  r2,r1,r1       add  r2,r1,r1

addi r1,r1,1        →  addi r1,r1,1     →  addi r1,r1,1
sub  r2,r3,r1          nop                 and  r0,r0
                       sub  r2,r3,r1       sub  r2,r3,r1

lw   r1,8(sp)       →  lw   r1,8(r13)   →  lw   r1,8(r13)
sw   r1,4(sp)          nop                 and  r0,r0
                       sw   r1,4(r13)      sw   r1,4(r13)
```

Sometimes other assembler transformations eliminate the data hazard and suppress nop insertion:

```
lw   r1,8(sp)        →  lw   r1,8(r13)
sw   r1,0x40(sp)        imm  0x004
                        sw   r1,0(r13)
```

3.1.3 Effective addresses

Throughout this specification, the native instructions `lb`, `lw`, `sb`, `sb`, `jal`, and the pseudo-instructions `lea`, `ll`, `sl`, are described using the underlying *`imm(ra)`* form of addressing.

The assembler accepts a number of effective address forms and converts these into the underlying for. The most general form of address is:

```
ea ::= { symbol | const }$_{opt}$ { [+|-] const }* { ( reg ) }$_{opt}$
```

After relocations have been applied, any *symbol* maps into a constant. All constants are folded into a single immediate value, which may fit in a 4-bit immediate field or may require an immediate prefix. If no base register is specified, `r0` is used.

3.1.4 Far branches

Each branch instruction's scaled 8-bit displacement field limits the branch displacement to approximately ±128 instructions. When the assembler encounters a branch to label that is further away and so cannot be represented in a single native branch instruction, it maps the branch into an absolute jump, conditionally prefixed by the complementary branch condition:

```
br   L1          →  j    L1          →  imm  L1$_{15:4}$
                                        jal  r0,L1$_{3:0}$(r0)

blt  L2          →  bge  skip        →  bge  skip
                    j    L2             imm  L2$_{15:4}$
                    skip:               jal  r0,L2$_{3:0}$(r0)
                                        skip:
```

(Of course, replacing far branches with jumps makes code larger which may turn other near branches over this code into far branches.)

(Far branch conversion is not yet implemented.)

3.1.5 Shift counts

The assembler maps a constant shift count greater than one into a series of 1-bit shifts:

```
slli r1,3        →  slli r1,1
                    slli r1,1
                    slli r1,1
```

3.2 Assembly directives

3.2.1 Text section

Format: **`text`**

Description: Specifies that the instructions and data that follow fall in the text (code) section.

3.2.2 Data section

Format: **`data`**

Description: Specifies that the data that follow fall in the data section.

3.2.3 Byte data

Format: **`byte`** *`expr`*

Description: Add a byte with the value *expr* to the current section.

3.2.4 Word data

Format: **`word`** *`expr`*

Description: Add a word with the value *expr* to the current section.

3.2.5 Alignment

Format: **`align`** *`alignment`*

Description: Alignment must be a power of two. Add zero or more 0 bytes to the current section, until the current section length is 0 *mod* $2^{alignment}$.

3.3 Assembly pseudo-instructions

Each instruction is specified below. Each specification provides the instruction name, assembly format, description, and pseudo-code of its operation:

> Pseudo
>
> **Map:** **`pseudo`** *`rd,ra,rb`* → `native` *`rd,ra,rb`*
>
> **Description:** Describes the pseudo-instruction semantics and what native instructions it maps to.
>
> **Comment:** Optional comments, if any.

The underlying properties of each instruction (well-definedness-in-context, interruptibility) follow from the native instructions they map to and so are not restated in the pseudo-instruction descriptions.

3.3.1 Compare

Map: **`cmp`** *`ra,rb`* → `sub r0,`*`ra,rb`*

Description: Compare the operand registers *ra* and *rb*. `cmp` is always followed by a conditional branch instruction.

3.3.2 Compare immediate

Map: **`cmpi`** *`ra,imm`* → `addi` *`ra`*`,r0,-`*`imm`*

Description: Compare the operand registers *ra* and the immediate value *imm*. `cmpi` is always followed by a conditional branch instruction.

3.3.3 Jump

Map: **`j`** *`imm(ra)`* → `jal r0,`*`imm(ra)`*

Description: Jump to the effective address formed by adding the register operand *ra* and the immediate operand,

3.3.4 Load effective address

Map:

```
lea rd,imm(ra)         →  addi rd,ra,imm
```

Description: Determine the effective address formed by adding the register operand *ra* and the immediate operand, and store it in *rd*.

3.3.5 Load byte signed

Map:

```
lbs rd,imm(ra)         →  lb rd,imm(ra)      →  lb rd,imm(ra)
                          xori rd,0x80           imm 0x008
                          subi rd,rd,0x80        xori rd,0
                                                 imm 0xFF8
                                                 addi rd,rd,0
```

Description: Load the byte at the effective address formed by adding the register operand *ra* and the immediate operand, *sign-extend* the byte, and store the result in *rd*.

3.3.6 No operation

Map:

```
nop                    →  and r0,r0
```

Description: Do nothing.

Comment: Using `and` instead of the more traditional `add` ensures this pseudo-instruction is interruptible.

3.3.7 Move

Map:

```
mov rd,ra              →  add rd,ra,r0
```

Description: Read the value of register *ra* and store the result in *rd*.

3.3.8 Return

Map:

```
ret                    →  jal r0,0(r15)
```

Description: Jump to the return address in `r15`.

3.3.9 Return from interrupt

Map:

```
reti                   →  jal r0,0(r14)
```

Description: Jump to the interrupt return address in `r14`.

3.3.10 Sign extend

Map:

```
sext rd,ra             →  mov_opt rd,ra   (omit if rd=ra)
                          andi rd,0xFF
                          xori rd,0x80
                          subi rd,rd,0x80
```

Description: Sign-extend the byte in $\mathit{ra}_{7:0}$ into a word and store the result in *rd*.

3.3.11 Subtract immediate

Map: **subi** *rd,ra,imm* → addi *rd,ra,-imm*

Description: Subtract the immediate operand from the register operand *ra* and store the result in *rd*.

3.3.12 Zero extend

Map: **zext** *rd,ra* → mov_{opt} *rd,ra* *(omit if rd=ra)*
andi *rd*,0xFF

Description: Zero-extend the byte in $ra_{7:0}$ into a word and store the result in *rd*.

3.4 Long word pseudo-operations

In memory, 32-bit long words are stored in *big endian* format. (See 2.1.4).

In registers, the less-significant 16-bits of the long word are stored in *rx* and the more-significant 16-bits in *rx′*, where $x' := 0$ if $x = 0$ and $x' := x + 1$ if $0 < x \le 11$.

3.4.1 Add long

Map: **addl** *rd,ra,rb* → mov_{opt} *rd´,ra´* *(omit if rd=ra)*
add *rd,ra,rb*
adc *rd´,rb´*

Description: Add the two long register operand pairs (*ra*,*ra′*) and (*rb*,*rb′*) and store the result in (*rd*,*rd′*).

3.4.2 And long

Map: **andl** *rd,rb* → and *rd,rb*
and *rd´,rb´*

Description: *Bitwise and* the two long register operand pairs (*rd*,*rd′*) and (*rb*,*rb′*) and store the result in (*rd*,*rd′*).

3.4.3 And not long

Map: **andnl** *rd,rb* → andn *rd,rb*
andn *rd´,rb´*

Description: *Bitwise and* the long register operand (*rd*,*rd′*) with the long complement of the long register operand (*rb*,*rb′*) and store the result in (*rd*,*rd′*).

3.4.4 Or long

Map: **orl** *rd,rb* → or *rd,rb*
or *rd´,rb´*

Description: *Bitwise or* the two long register operand pairs (*rd*,*rd′*) and (*rb*,*rb′*) and store the result in (*rd*,*rd′*).

3.4.5 Xor long

Map: **xorl** *rd*,*rb* → xor *rd*,*rb*
xor *rd´*,*rb´*

Description: *Bitwise exclusive-or* the two long register operand pairs (*rd*,*rd′*) and (*rb*,*rb′*) and store the result in (*rd*,*rd′*).

3.4.6 Load long

Map: **ll** *rd*,*imm*(*ra*) → lw *rd*,*imm+2*(*ra*)
lw *rd´*,*imm*(*ra*)

Description: Load the long word at the effective address formed by adding the register operand *ra* and the immediate operand, and store it in the long register pair (*rd*,*rd′*).

3.4.7 Load immediate long

Map: **lil** *rd*,*imm* → addi *rd*,r0,$imm_{15:0}$
addi *rd´*,r0,$imm_{31:16}$

Description: Load the long immediate and store it in the long register pair (*rd*,*rd′*).

3.4.8 Move long

Map: **movl** *rd*,*ra* → add *rd*,*ra*,r0
add *rd´*,*ra´*,r0

Description: Read the long value of register pair (*ra*,*ra′*) and store the result in (*rd*,*rd′*).

3.4.9 Subtract long

Map: **subl** *rd*,*ra*,*rb* → mov_{opt} *rd´*,*ra´* *(omit if rd=ra)*
sub *rd*,*ra*,*rb*
sbc *rd´*,*rb´*

Description: Subtract the long value of register pair (*rb*,*rb′*) from the long value of register pair (*ra*,*ra′*) and store the result in (*rd*,*rd′*).

3.4.10 Store long

Map: **sl** *rd*,*imm*(*ra*) → sw *rd*,*imm+2*(*ra*)
sw *rd´*,*imm*(*ra*)

Description: Store the value of the long register pair (*rd*,*rd′*) to memory at the effective address formed by adding the register operand *ra* and the immediate operand.

4 Calling Conventions Specification

This section specifies the register assignments, calling conventions, stack frame layouts, etc. that provide the C language runtime environment.

(These are not necessarily the best design choices, but reflect the current behavior of the compiler.)

4.1 Register conventions

This table describes how each register is used for C code.

Register	Use	Save Policy
`r0`	always zero	-
`r1`	reserved for assembler	*reserved*
`r2`	function return value	-
`r3`	first argument	caller save
`r4`	second argument	caller save
`r5`	third argument	caller save
`r6`	temporary	caller save
`r7`	temporary	caller save
`r8`	temporary	caller save
`r9`	temporary	caller save
`r10`	local variable	*callee save*
`r11`	local variable	*callee save*
`r12`	local variable	*callee save*
`r13`	stack pointer (`sp`)	*callee save*
`r14`	interrupt return address	*reserved*
`r15`	return address	*callee save*

The caller is responsible for saving any of its live *caller save* registers across function calls. The called function is responsible for preserving *callee save* registers on behalf of its caller.

4.2 Argument and return value passing

To pass arguments to a function, apply these rules:

- chars: padded and passed as words;
- words (integer and pointer data): passed in registers or in memory;
- longs: passed in a pair of registers or in memory;
- aggregates (structs and unions): reserve storage in the caller's stack frame for a copy of the aggregate; copy the actual argument into the copy, and pass a pointer to the copy.

To return a value from a function, apply these rules:

- words: returned in `r2`;
- longs: returned in (`r2`,`r3`);
- aggregates (structs and unions): pass a pointer to the aggregate that is assigned the result in the caller; if the result is to be discarded, reserve storage in the caller's stack frame for the discarded result. The result aggregate pointer is passed as an implicit first argument, in `r3`.

Incoming arguments are found in left to right order in the argument registers `r3`-`r5` and/or starting at address `0(sp)`. For example, a function with five `int` arguments receives the first three in `r3`-`r5` and the last two at `6(sp)` and `8(sp)`. And for this function `foo()`:

```
typedef struct X { int a, b, c, d; } X;
X foo(int i, X x, int j, long l, int k);
```

we have

```
    r3 : X* /* return_value address */;
    r4 : i
    r5 : X* /* x address */
 6(sp) : j
 8(sp) : l
12(sp) : k
```

4.3 Stack frame

The stack grows down from higher addresses to lower addresses.

The first instructions of each function form the function prolog. This adjusts the stack pointer downward to allocate storage for the activation record, and saves any callee save registers that may be modified by the function.

The last instructions of each function form the function epilog. This reloads the callee save registers saved by the function prolog, adjusts the stack pointer to release the activation record storage, and returns to the caller return address in `r15`.

On entry to the function, arguments start at `0(sp)`, except that up to the first 3 scalar arguments are found in `r3-r5`.

The activation record consists of:

- callee save registers
- local variables
- temporaries
- outgoing call argument lists

The stack pointer is adjusted only once in the function prolog and once in the epilog. Therefore the activation record is sized to handle the worst case (largest possible) outgoing call argument list of any of the call sites in the function.

(Alloca not yet designed.)

4.4 Variable argument functions

Upon entry to a varargs function, any arguments passed in registers are *homed*, storing them to the stack frame in memory at `0(sp)`. At that point all arguments are stored sequentially in memory, greatly simplifying an implementation of the stdarg.h macros.

```
int foo(int i, int j, ...) { return i + j; }
↓
_foo:
sw r3,0(sp)
sw r4,2(sp)
sw r5,4(sp)
lw r9,0+0(sp)
lw r8,2+0(sp)
add r2,r9,r8
L1:
ret
```

```
/* stdarg.h */
#ifndef _STDARG
#define _STDARG

typedef char* va_list;

#define _ROUNDUP(a,n)   (((a) + (n) - 1) & ~((n) - 1))
#define _WORDS(t)       (int)(_ROUNDUP(sizeof(t), sizeof(char*)))
#define va_start(ap,v)  ((ap) = ((va_list)&v + _WORDS(v)))
#define va_arg(ap,t)    (*(t*)(((ap) += _WORDS(t)) - _WORDS(t)))
#define va_end(ap)      ((ap) = 0)

#endif /* !_STDARG */
```

(Note this varargs scheme is incompatible with (broken with respect to) passing aggregate values by reference to copy. When possible the compiler should be changed to pass aggregates directly on the stack.)

5 Systems Programming Specification

5.1 Interrupts

When an interrupt request is received, it is held pending until an interruptible instruction is executed. Then the interrupt is taken. The address of the next instruction is saved in `r14` and control transfers to the interrupt handler:

Operation:

```
if (interrupt_pending && can_interrupt) {
    r[14] = pc;
    pc = 0x0010; /* interrupt handler */
    interrupt_pending = false;
}
```

When the interrupt handler completes, it executes `reti`, e.g. `jal r0,0(r14)`, and execution resumes with the interrupted instruction.

The interrupt controller (together with the interrupt handler) is responsible for ensuring a second interrupt request is not received before the interrupt handler has returned.

6 Implementation Specification

This section specifies properties of the xr16 core, an implementation of the xr16 instruction set architecture. Other implementations may exhibit different properties.

The cycle timings presented in this section assume the XSOC system-on-a-chip infrastructure and the external byte-wide asynchronous SRAM of the XESS XS40 board, as described in the article series.

6.1 Core interface

6.2 Instruction timings

This table describes the various t_* memory latency parameters and example cycle counts for accesses to the external byte-wide asynchronous SRAM.

Parameter	Description	Cycles
t_i	fetch instruction	1
t_{rw}	read word	1
t_{rb}	read byte	1
t_{ww}	write word	3
t_{wb}	write byte	2

This table specifies the number of cycles required to execute each native instruction. The *Typical* column gives specific cycle counts assuming instructions and data lie in the external RAM.

Instructions	Cycles	Typ
`add addi sub and or xor andn adc sbc andi ori xori andni adci sbci srli srai slli srxi slxi imm`	t_i	1
`lw`	$t_i + t_{rw}$	2
`lb`	$t_i + t_{rb}$	2
`sw`	$t_i + t_{ww}$	4
`sb`	$t_i + t_{wb}$	3
`beq bne bc bnc bv bnv blt bge ble bgt bltu bgeu bleu bgtu` *branch taken*	$3t_i$	3
`beq bne bc bnc bv bnv blt bge ble bgt bltu bgeu bleu bgtu` *branch not taken*	t_i	1
`br jal call`	$3t_i$	3

6.3 I/O Space

The last 256 bytes of the address space (0xFF00-0xFFFF) are reserved for memory mapped I/O control registers, organized as 8 devices (dev0-dev7), each with 32 bytes of control registers.

The first device, dev0, is the memory management unit.

6.4 Memory management unit

The optional (and as yet unimplemented) MMU provides simple address translation and page protection for applications and the operating system. The 2^{16} byte application virtual address space is divided into 16 4 KB pages. Each page can be mapped to any 4 KB physical page from a 26-bit physical address space.

```
Word mmu[16]; /* at address 0xFF00 */
```

Operation:

$PA_{25:12}$ = $mmu_{13:0}[VA_{15:12}]_1$
$PA_{11:0}$ = $VA_{11:0}$;
writable = $mmu_{15}[VA_{15:12}]$;

A store to an unwritable address does not trap; it is simply discarded.

The MMU has no effect on I/O space accesses.

7 Glossary

to-do
branch prefix
undefined behavior

8 Rationale

To quote from the article series,

"INSTRUCTION SET"

"Now we'll refine the instruction set and choose an instruction encoding. Our goals and constraints are:

- cover C (integer) operator set;
- fixed size, 16-bit instructions;
- easily decoded and pipelined   simple and regular;
- 3-operand instructions (`dest = ` $\texttt{src}_1$ *op* $\texttt{src}_2$/imm) as encoding space allows;
- byte addressable – load and store bytes and words;
- big endian (arbitrary choice);
- one addressing mode, disp(reg);
- long ints   add/subtract with carry, shift left/right extended."

"Which instructions merit the most bits? Reviewing early compiler output from test applications shows the most common instructions (static frequency) are `lw` *(load word)*, 24%; `sw` *(store word)*, 13%; `mov` *(reg-reg move),* 12%; `lea` (*load effective address),* 8%; `call`, 8%; `br`, 6%; and `cmp`, 6%. `Mov`, `lea`, and `cmp` can be synthesized from `add` or `sub` with `r0`. 69% of loads/stores use *`disp`*`(reg)` addressing, 21% are absolute, and 10% are register indirect."

"Therefore we make these choices:

- `add`, `sub`, `addi` are 3-operand;
- less common operations – logical ops, add/sub with carry, and shifts – are 2-operand to conserve opcode space;
- `r0` always reads as 0;
- 4-bit immediate fields;
- for 16-bit constants, an optional immediate prefix `imm` establishes the most-significant 12-bits of the immediate instruction which follows;
- no condition codes, rather use an interlocked compare and conditional branch sequence;
- `jal` (jump-and-link) jumps to an effective address, saving the return address in a register;
- `call` *`func`* encodes `jal r15,`*`func`* in one 16-bit instruction (provided the function is 16-byte aligned);
- perform mul, div, rem, and variable and multiple bit shifts in software."

9 Revision History

Version	Date	Description
0.5	1999/12/05	First draft of instruction set architecture specification
0.51	1999/12/09	Correct errors found by OM and TC
0.6	1999/12/18	Started calling convention, systems programming, implementation specifications

9.1 Known spec errors

- 2.3 adc/sbc/i are broken