Overview of Xtensa ISA

Espressif Systems

Version 0021604, 2021-02-17

Contents

Chapter 1

Overview of Xtensa Instruction Set Architecture

1.1 Basic facts about Xtensa ISA

The content of this section is based on [6, Chapter 3].

Xtensa is a post-RISC ISA i.e it derives most of its features from RISC but also incorporates certain features where CISC is advantageous.

Xtensa processors are typically configurable. CPU designers can enable features such as: additional instructions (both predefined and custom), interrupts, coprocessors, memory management, and others. Some of these features affect the ABI and code generated by the compiler.

Standard Xtensa instructions are 24-bit. Code density option may be enabled to add 16-bit instructions. Wider instructions are also possible in some configurations.

Xtensa processors employ Harvard architecture, meaning that they have separate instruction and data buses. Depending on the SoC design, these buses may be connected to separate instruction and data memories, or to a shared memory.

1.2 Registers

`PC` — Program Counter, holds the address of the instruction being executed. PC is not writeable directly. It can be modified as a side effect of calls, jumps, and exceptions. PC is also not directly readable, however Xtensa provides instructions to perform PC-relative loads and jumps, facilitating access to literal values and generation of position-independent code.

`a`*`n`* — 16 general purpose 32-bit architectural registers.

`AR[n]` — Physical general purpose registers. In CPU configurations without the window register option, these are the same as the architectural registers `a`*`n`*. In CPU configurations where window register option is enabled, there are more physical registers than architectural registers. The number of physical registers can be 32 or 64. 16 physical registers are mapped to the architectural registers `a`*`n`* at a time.

Special registers — Xtensa processors contain a number of registers used to control the operation of the processor, perform interrupt and exception handling, etc. Only a few special registers are relevant to the code generation by the compiler. Special registers can not be used as operands of ALU and branch instructions. They must be read and written using `RSR` and `WSR` instructions.

`SAR` Shift amount register is a special register. It is used to store the number of bits for subsequent shift instructions. Xtensa does not provide shift instructions which would have the shift amount specified in a general register (`an`) operand.

User registers These registers are added by various processor configuration options, or by processor designers defining custom instructions. Only a few special registers are relevant to code generation by the compiler. Like special registers, user registers can not be used as operands of ALU and branch instructions. They must be read and written using `RUR` and `WUR` instructions.

`THREADPTR` Thread pointer register is a user register. The system software typically writes a pointer to the TCB of the executing thread into this register. The register is used by the compiler when accessing thread-local variables.

1.3 Windowed Register

General purpose registers (GPR) are used to store data temporarily for CPU while performing various operations. These registers are blazing fast but are limited in number (8 – 32).

Typically, the number of registers present in the register file are equal to the registers directly accessible by the core. The Xtensa core can only access 16 GPR, namely `a0` – `a15`. So the register file contains 16 registers.

Xtensa also has a Windowed register option, which when enabled, extends this register file to contain 64 registers. Essentially, the register frame (`a0` – `a15`) acts as a window, through which only 16 registers are visible, that slides on this large register file having 64 registers. And hence the name: Windowed register.

Which 16 registers are visible is controlled by the WindowBase register. WindowBase register indicates where the window starts in the register file. Also, the shifting/rotation of this window occurs in units of 4. That means, the window starts at (WindowBase x 4)th position in the register file.

1.4 Calling convention

Xtensa supports two different application binary interfaces (ABI) which also includes the calling conventions.

1. Windowed register ABI
2. Call0 ABI

We will cover only Windowed register ABI.

1.4.1 Windowed register calling convention

Return address is stored in `a0` and the stack pointer is store in `a1`

Arguments to the functions are passed in both, registers and memory (stack). The first six arguments are passed in the registers and remaining go on the stack.

As for return values, they are returned in registers beginning from `a2` till `a5`. If there are more than 4 values to be returned, the caller passes a pointer which is then populated by callee with all the return values.

Register	Use
`a0`	Return Address
`a1`	Stack Pointer
`a2`–`a7`	Incoming Arguments

In Xtensa, subroutine calls are initiated using `CALLn` and `CALLXn` instructions, where `n` specifies the amount by which the register window needs to be rotated for the callee. `n` can be equal to 4, 8, or 12.

Note that `CALL0`/`CALLX0` instructions do not follow windowed register calling convention, so further explanation applies for $n \neq 0$.

What does "rotation of window for the callee" exactly mean?

When a subroutine is called using `CALLn`/`CALLXn`, WindowBase register is incremented by $n/4$, so the registers visible by callee are different from those visible by the caller because the register window (`a0` – `a15`) has moved.

In general, for a windowed register call `CALLn`/`CALLXn`:

- $\mathtt{a}_n$ of caller will be `a0` of callee
- $\mathtt{a}_{n+1}$ of caller will be `a1` of callee and so on.

So the caller needs to put the first argument of the callee in $\mathtt{a}_{n+2}$, second in $\mathtt{a}_{n+3}$ and so on.

FIXME: Explain how many arguments are passed in registers and on the stack.

While returning from the callee function using `RETW` instruction, WindowBase register is decremented by $n/4$. This restores the register window of the caller.

Let's take an example:

```
/*
 * void bar(int x, int y);
 *
 * void func(void)
 * {
 *     ...
 *     foo = bar(x, y);
 *     ...
 * }
 */

func:
    ...
    mov         a10, x    // a10 is bar's a2
    mov         a11, y    // a11 is bar's a3
    call8       bar
    mov         foo, a10  // a10 is bar's a2 (return value)
    ...
```

When a function calls another function, it does not have to store its own arguments somewhere else to accommodate the arguments for the callee since the arguments of the callee is at a different physical location. The callee function internally will still use `a2` to access its first argument but as you can see, `a2` of the caller is at a different physical location than `a2` of callee. If there was no windowing and the number of physical registers would be exactly 16 then `a2` of caller and callee would be same. Thus for each function call, the data in these registers would have to be stored at some other memory location (stack) before calling any function and restore again after returning.

Accessing any memory location, other than register, is very slow and as a result this saving/restoring will have a negative impact on performance. So using windowed register convention saves us the overhead of such stores/restores and also reduces the code size.

1.4.2 Stack Layout

As mentioned, the stack pointer resides in `a1` register. This stack pointer always points to the bottom of the stack!

Usually, function prologue sets up the stack for a function.

In Xtensa, ENTRY instruction is the function prologue

ENTRY instruction primarily does two things: 1. Allocates the stack frame for the function and sets the stack pointer. 2. Moves/rotates the register window by n as specified in the calln/callxn instruction.

Stack layout is always better explained through an illustration 1.2.

For clarity, lets use `sp` as stack pointer instead of `a1`.

Like most architectures, in Xtensa too, stack grows downwards. If there are outgoing arguments, apart from the first 6 arguments, then they will go on the positive offset from `sp`. i.e 7th argument on `sp`, 8th on `sp` + 4 and so on. Above the outgoing arguments, local variables of that function are stored.

The region underneath the stack pointer, called Base Save Area, is of 16 bytes and reserved for saving the `a0` – `a3` of the caller (previous frame) when the window overflow exception occurs. If more registers of the caller are required to be saved then it is stored in the Extra Save Area at the top of the caller (previous) stack frame. The location of saving registers of the caller (i-1) frame is highlighted in the image.

With all the necessary points covered, let's take an example and connect all the dots.

Suppose, each function call is carried out using call8 and we start with WindowBase = 4

Function A calls B, B calls C, C calls D... till I, i.e:

Functions	A →B →C →D →E →F →G →H →I
WindowBase	4 →6 →8 →10 →12 →14 →0 →2 →4

On each function call, the WindowBase will be incremented by 2 because call8 is used.

No. of bits in WindowBase register = log_2((No. of registers in register file)/4) = $log_2(64/4) = 4$. Thus the max value of WindowBase is 15.

As we have noticed, on the 9th function call the window wraps around to a point where the frame contains the data of a parent function, i.e `a0`, `a1`.. contains data of A. It implies that `a8`, `a9`.. of H are `a0`, `a1`.. of A.

A window overflow exception will be generated when H tries to modify `a8`, `a9`.. since it originally contains the context of A, so these must be saved to accommodate arguments of I. At this point, in the window overflow exception handler we must rotate the register window to frame A (WindowBase = 4).

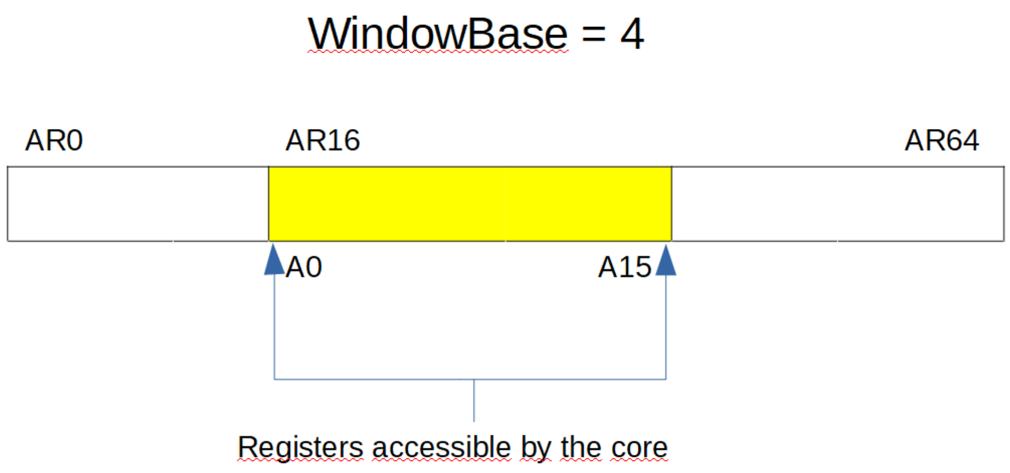

Figure 1.1: Register window

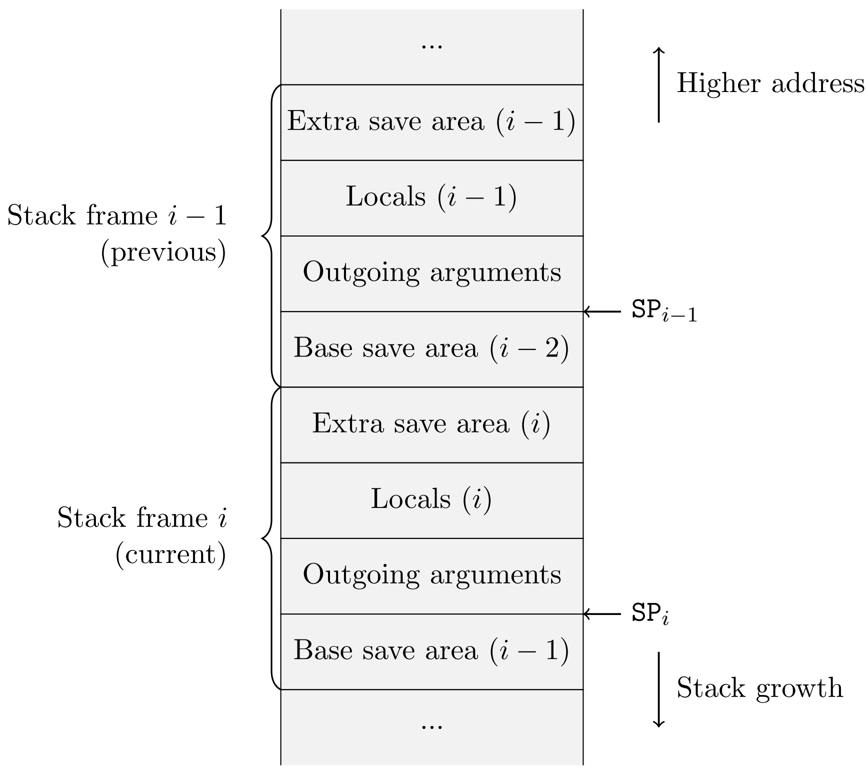

Figure 1.2: Windowed ABI stack layout

`a0` – `a3` are stored in the Base Save Area of B's stack frame. B's stack frame is accessible since `a9` is `a1` of B, which is B's stack pointer. `a4` – `a7` are stored in the Extra Save Area of A's stack frame. Now whenever B returns, window underflow exception will be generated and we need to make sure that the corresponding exception handler would restore these values back into the registers.

Chapter 2

Instruction Formats

The contents of this chapter are derived from [4, 2, 1, 3].

2.1 Instruction Fields

op0, op1, op2 4-bit opcode fields

r, s, t – 4-bit operand fields

2.2 Functions

2.2.1 sign_extend(imm)

Extend an immediate to a 32-bit value by copying its left-most bit to all bits to the left.

2.2.2 B4const(imm)

```
int B4const(uint imm) {
    const int B4constValues[16] = {-1,1,2,3,4,5,6,7,8,10,12,16,32,64,128,256};
    return  B4constValues[imm];
}
```

2.2.3 B4constu(imm)

```
uint B4constu(uint imm) {
    const int B4constuValues[16] = {32768,65536,2,3,4,5,6,7,8,10,12,16,32,64,128,256};
    return  B4constuValues[imm];
}
```

2.3 Assembler expressions

ar – general purpose register correspondence to r operand field (AR[r])

as – general purpose register correspondence to s operand field (AR[s])

at – general purpose register correspondence to t operand field (AR[t])

sr – special purpose register name

2.4 Format descriptions

RRR Instruction Format

23 20	19 16	15 12	11 8	7 4	3 0
op2	op1	r	s	t	*op*0

RRI4 Instruction Format

23 20	19 16	15 12	11 8	7 4	3 0
imm[3..0]	op1	r	s	t	op0

RRI8 Instruction Format

23 16	15 12	11 8	7 4	3 0
imm[7..0]	r	s	t	op0

RI16 Instruction Format

23 8	7 4	3 0
imm[15..0]	t	op0

RSR Instruction Format

23 20	19 16	15 8	7 4	3 0
imm[3..0]	op1	rs	t	op0

CALL Instruction Format

23 6	5 4	3 0
offset	n	op0

CALLX Instruction Format

23 20	19 16	15 12	11 8	7 6	5 4	3 0
op2	op1	r	s	m	n	*op*0

BRI8 Instruction Format

23 16	15 12	11 8	7 6	5 4	3 0
imm[7..0]	r	s	m	n	*op*0

BRI12 Instruction Format

23 12	11 8	7 6	5 4	3 0
imm[11..0]	s	m	n	*op*0

RRRN Instruction Format

15 12	11 8	7 4	3 0
r	s	t	*op*0

RI7 Instruction Format

15 12	11 8	7	6 4	3 0
imm[3..0]	s	i	imm[6..4]	*op*0

RI6 Instruction Format

15 12	11 8	7	6	5 4	3 0
imm[3..0]	s	i	z	imm[5..4]	*op*0

Chapter 3

Core Instruction Set

3.1 Instructions encoded with RRR format

Encoding

23 20	19 16	15 12	11 8	7 4	3 0		
0110	0000	r	0001	t	0000	ABS	$If\ AR[t]_{31}\ then AR[r] \leftarrow -AR[t]$ $else AR[r] \leftarrow AR[t]$ endif
1000	0000	r	s	t	0000	ADD	$AR[r] \leftarrow AR[s] + AR[t]$
1001	0000	r	s	t	0000	$ADDX2$	$AR[r] \leftarrow AR[s] + (AR[t] * 2)$
1010	0000	r	s	t	0000	$ADDX4$	$AR[r] \leftarrow AR[s] + (AR[t] * 4)$
1011	0000	r	s	t	0000	$ADDX8$	$AR[r] \leftarrow AR[s] + (AR[t] * 8)$
0001	0000	r	s	t	0000	AND	$AR[r] \leftarrow AR[s] \& AR[t]$
0000	0000	0010	0000	0011	0000	$DSYNC$	
0000	0000	0010	0000	0010	0000	$ESYNC$	
imm[3..0]	010sh[4]	r	sh[3..0]	t	0000	$EXTUI$	$mi \leftarrow (0 \| imm_{3..0}) + 1$ $mask \leftarrow 0^{32-mi} \| 1^{mi}$ $AR[r] \leftarrow (0^{sh} \| AR[s]_{31..sh}) AND mask$
0000	0000	0010	0000	1101	0000	$EXTW$	
0000	0000	0010	0000	0000	0000	$ISYNC$	
0000	0000	0010	0000	1100	0000	$MEMW$	
1000	0011	r	s	t	0000	$MOVEQZ$	$condition \leftarrow AR[t] = 0^{32}$ $if\ condition\ then$ $AR[r] \leftarrow AR[s]$ endif
1011	0011	r	s	t	0000	$MOVGEZ$	$condition \leftarrow AR[t] >= 0^{32}$ $if\ condition\ then$ $AR[r] \leftarrow AR[s]$ endif
1010	0011	r	s	t	0000	$MOVLTZ$	$condition \leftarrow AR[t] < 0^{32}$ $if\ condition\ then$ $AR[r] \leftarrow AR[s]$ endif
0110	0000	r	0000	t	0000	NEG	$AR[r] \leftarrow 0^{32} - AR[t]$
0000	0000	0010	0000	1111	0000	NOP	No operation
0010	0000	r	s	t	0000	OR	$AR[r] \leftarrow AR[s] OR AR[t]$
0000	0000	0010	0000	0001	0000	$RSYNC$	
1010	0001	r	s	0000	0000	SLL	$sh \leftarrow SAR_{5..0}$ $AR[r] \leftarrow AR[s]_{31..31-sh} \| 0^{sh}$
000sh[4]	0001	r	s	sh[3..0]	0000	$SLLI$	$AR[r] \leftarrow AR[s]_{31..31-sh} \| 0^{sh}$
1011	0001	r	0000	t	0000	SRA	$sh \leftarrow SAR_{5..0}$ $AR[r] \leftarrow AR[t]_{31}^{sh} \| AR[t]_{31..sh}$

Encoding

23 20	19 16	15 12	11 8	7 4	3 0		
001sh[4]	0001	r	sh[3..0]	t	0000	*SRAI*	$AR[r] \leftarrow AR[t]_{31}^{sh} \vert\vert AR[t]_{31..sh}$
1000	0001	r	s	t	0000	*SRC*	$sh \leftarrow SAR_{5..0}$ $AR[r] \leftarrow AR[s]_{31-sh..sh} \vert\vert AR[t]_{31..31-sh}$
1001	0001	r	0000	t	0000	*SRL*	$sh \leftarrow SAR_{5..0}$ $AR[r] \leftarrow 0^{sh} \vert\vert AR[t]_{31..sh}$
0100	0001	r	sh	t	0000	*SRLI*	$AR[r] \leftarrow 0^{sh} \vert\vert AR[t]_{31..sh}$
0100	0000	0010	s	0000	0000	*SSA8L*	$sh \leftarrow AR[s]_{1..0} \vert\vert 0^3$ $SAR \leftarrow sh$
0100	0000	0100	sh[3..0]	000sh[4]	0000	*SSAI*	$SAR \leftarrow 0^{27} \vert\vert sh_{4..0}$
0100	0000	0001	s	0000	0000	*SSL*	$sh \leftarrow 0 \vert\vert AR[s]_{4..0}$ $SAR \leftarrow 32 - sh$
0100	0000	0000	s	0000	0000	*SSR*	$sh \leftarrow 0 \vert\vert AR[s]_{4..0}$ $SAR \leftarrow sh$
1100	0000	r	s	t	0000	*SUB*	$AR[r] \leftarrow AR[s] - AR[t]$
1101	0000	r	s	t	0000	*SUBX2*	$AR[r] \leftarrow AR[s] - (AR[t] * 2)$
1110	0000	r	s	t	0000	*SUBX4*	$AR[r] \leftarrow AR[s] - (AR[t] * 4)$
1111	0000	r	s	t	0000	*SUBX8*	$AR[r] \leftarrow AR[s] - (AR[t] * 8)$
0011	0000	r	s	t	0000	*XOR*	$AR[r] \leftarrow AR[s] XOR AR[t]$
0110	0001	sr		t	0000	*XSR*	$tmp \leftarrow AR[t]$ $AR[t] \leftarrow SR[sr]$ $SR[sr] \leftarrow tmp$

Assembler

Instruction	
ABS	abs ar, at
ADD	add ar, as, at
ADDX2	addx2 ar, as, at
ADDX4	addx4 ar, as, at
ADDX8	addx8 ar, as, at
AND	and ar, as, at
DSYNC	dsync
ESYNC	esync
EXTUI	extui ar, as, sh_imm, mask_imm
EXTW	extw
ISYNC	isync
MEMW	memw
MOVEQZ	moveqz ar, as, at
MOVGEZ	movgez ar, as, at
MOVLTZ	movltz ar, as, at
NEG	neg ar, at
NOP	nop
OR	or ar, as, at
RSYNC	rsync
SLL	sll ar, as
SLLI	slli ar, as, sh_imm
SRA	sra ar, at
SRAI	srai ar, at, sh_imm
SRC	src ar, as, at
SRL	srl ar, at
SRLI	srli ar, at, sh_imm
SSA8L	ssa8l as
SSAI	ssai sh_imm
SSL	ssl as
SSR	ssr as
SUB	sub ar, as, at

Assembler

Instruction	
SUBX2	subx2 ar, as, at
SUBX4	subx4 ar, as, at
SUBX8	subx8 ar, as, at

3.2 Instructions encoded with RRI8 format

Encoding

23 16	15 12	11 8	7 4	3 0		
imm[7..0]	1100	s	t	0010	$ADDI$	$AR[s] \leftarrow AR[t] + imm$
imm[7..0]	1101	s	t	0010	$ADDMI$	$AR[s] \leftarrow AR[t] + (imm_7^{16} \vert\vert imm_{7..0} \vert\vert 0^8)$

imm[7..0]	0100	s	t	0111	$BALL$	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow AR[s] AND AR[t] = 0^{32}$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	1000	s	t	0111	$BANY$	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow$ (NOT AR[s]) AND AR[t] $\neq 0^{32}$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	0101	s	t	0111	BBC	$offset \leftarrow sign\_extend(imm)$
						$bit \leftarrow AR[t]_{4..0}$
						$condition \leftarrow AR[s]_{bit} = 0$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	011b[4]	s	b[3..0]	0111	$BBCI$	$offset \leftarrow signe\_extend(imm)$
						$condition \leftarrow AR[s]_b = 0$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	1101	s	t	0111	BBS	$offset \leftarrow sign\_extend(imm)$
						$bit \leftarrow AR[t]_{4..0}$
						$condition \leftarrow AR[s]_{bit} \neq 0$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	111b[4]	s	b[3..0]	0111	$BBSI$	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow AR[s]_b \neq 0$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	0001	s	t	0111	BEQ	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[t] = AR[s])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	r	s	0010	0110	$BEQI$	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[t] = B4Const[r])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

Encoding

23 .. 16	15 .. 12	11 .. 8	7 .. 4	3 .. 0		

imm[7..0]	1010	s	t	0111	*BGE*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[t] >= AR[s])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	r	s	1110	0110	*BGEI*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[t] >= B4Const[r])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	1011	s	t	0111	*BGEU*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (0\|\|AR[t]) >= (0\|\|AR[s])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	r	s	1111	0110	*BGEUI*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (0\|\|AR[t]) >= (0\|\|B4Const[r])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	0010	s	t	0111	*BLT*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[s] < AR[t])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	r	s	1010	0110	*BLTI*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[t] < B4Const[r])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	0011	s	t	0111	*BLTU*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (0\|\|AR[t]) < (0\|\|AR[s])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	r	s	1011	0110	*BLTUI*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (0\|\|AR[t]) < (0\|\|B4Const[r])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	1100	s	t	0111	*BNALL*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[s]\ AND\ AR[t]) \neq 0^{32}$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	1001	s	t	0111	*BNE*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[t] \neq AR[s])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	r	s	0110	0110	*BNEI*	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[t] \neq B4Const[r])$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

Encoding

23 16	15 12	11 8	7 4	3 0		

imm[7..0]	0000	s	t	0111	$BNONE$	$offset \leftarrow sign\_extend(imm)$
						$condition \leftarrow (AR[s]\ AND\ AR[t]) = 0^{32}$
						if condition then
						$PC \leftarrow PC + offset + 4$
						endif

imm[7..0]	0000	s	t	0010	$L8UI$	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $mem \leftarrow LoadMemory(vAddr, 8)$ $AR[t] \leftarrow 0^{24}\|\|mem_{7..0}$
imm[7..0]	1001	s	t	0010	$L16SI$	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $mem \leftarrow LoadMemory(vAddr, 16)$ $AR[t] \leftarrow mem_{15}^{16}\|\|mem_{15..0}$
imm[7..0]	0001	s	t	0010	$L16UI$	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $mem \leftarrow LoadMemory(vAddr, 16)$ $AR[t] \leftarrow 0^{16}\|\|mem_{15..0}$
imm[7..0]	0010	s	t	0010	$L32I$	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $mem \leftarrow LoadMemory(vAddr, 32)$ $AR[t] \leftarrow mem_{31..0}$
imm[7..0]	1010	imm[11..8]	t	0010	$MOVI$	$AR[s] \leftarrow sign\_extend(imm)$
imm[7..0]	0100	s	t	0010	$S8I$	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $StoreMemory(vAddr, 8, AR[t]_{7..0})$
imm[7..0]	0101	s	t	0010	$S16I$	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $StoreMemory(vAddr, 16, AR[t]_{15..0})$
imm[7..0]	0110	s	t	0010	$S32I$	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $StoreMemory(vAddr, 32, AR[t]_{31..0})$

Assembler

Instruction	
ADDI	addi at, as, imm
ADDMI	addmi at, as, imm
BALL	ball as, at, target
BANY	bany as, at, target
BBC	bbc as, at, target
BBCI	bbci as, imm, target
BBS	bbs as, at, target
BBSI	bbsi as, imm, target
BEQ	beq as, at, target
BEQI	beqi as, imm, target
BGE	bge as, at, target
BGEI	bgei as, imm, target
BGEU	bgeu as, at, target
BGEUI	bgeui as, imm, target
BLT	blt as, at, target
BLTI	blti as, imm, target
BLTU	bltu as, at, target
BLTUI	bltui as, imm, target
BNALL	bnall as, at, target
BNE	bne as, at, target
BNEI	bnei as, imm, target

BNONE	bnone as, at, target
L8UI	l8ui at, as, imm
L16SI	l16si at, as, imm
L16UI	l16ui at, as, imm
L32I	l32i at, as, imm
MOVI	movi at, imm
S8I	s8i at, as, imm
S16I	s16i at, as, imm
S32I	s32i at, as, imm

3.3 Instructions encoded with BRI12 format

Encoding

23 .. 12	11 .. 8	7 .. 4	3 .. 0		

imm[11..0]	s	0001	0110	$BEQZ$	$offset \leftarrow sign\_extend(imm)$
					$condition \leftarrow (AR[s] = 0^{32})$
					if condition then
					$PC \leftarrow PC + offset + 4$
					endif

imm[11..0]	s	1101	0110	$BGEZ$	$offset \leftarrow sign\_extend(imm)$
					$condition \leftarrow (AR[s] >= 0^{32})$
					if condition then
					$PC \leftarrow PC + offset + 4$
					endif

imm[11..0]	s	1001	0110	$BLTZ$	$offset \leftarrow sign\_extend(imm)$
					$condition \leftarrow (AR[s] < 0^{32})$
					if condition then
					$PC \leftarrow PC + offset + 4$
					endif

imm[11..0]	s	0101	0110	$BNEZ$	$offset \leftarrow sign\_extend(imm)$
					$condition \leftarrow (AR[s] \neq 0^{32})$
					if condition then
					$PC \leftarrow PC + offset + 4$
					endif

Assembler

Instruction	
BEQZ	beqz as, imm
BGEZ	bgez as, imm
BLTZ	bltz as, imm
BNEZ	bnez as, imm

3.4 Instructions encoded with CALL format

Encoding

23 .. 5	4	3 .. 0		
imm[17..0]	00	0101	$CALL0$	$AR[0] \leftarrow next(PC)$ $offset \leftarrow sign\_extend(imm)$ $PC \leftarrow (PC_{31..2} + offset_{31..0} + 1)_{31..2} \vert\vert 0^2$
imm[17..0]	00	0110	J	$offset \leftarrow sign\_extend(imm)$ $PC \leftarrow PC + offset + 4$

Assembler

Instruction	
CALL0	call0 target
J	j target

3.5 Instructions encoded with CALLX format

Encoding

23 20	19 16	15 12	11 8	7 6	5 4	3 0		
0000	0000	0000	s	11	00	0000	$CALLX0$	$AR[0] \leftarrow next(PC)$ $PC \leftarrow AR[s]$
0000	0000	0000	s	10	10	0000	JX	$PC \leftarrow AR[s]$
0000	0000	0000	0000	10	00	0000	RET	$PC \leftarrow AR[0]$

Assembler

Instruction	
CALLX	callx as
JX	jx as
RET	ret

3.6 Instructions encoded with RSR format

Encoding

23 20	19 16	15 8	7 4	3 0		
0000	0011	sr	t	0000	RSR	$AR[t] \leftarrow SR[sr]$
0001	0011	sr	t	0000	WSR	$SR[sr] \leftarrow AR[t]$

Assembler

Instruction	
RSR	rsr at, sr
WSR	wsr at, sr

3.7 Instructions encoded with RI16 format

Encoding

23 8	7 4	3 0		
imm[15..0]	t	0001	$L32R$	$offset \leftarrow 1^{14} \| imm_{15..0} \| 0^2$ $vAddr \leftarrow ((PC+3)_{31..2} \| 0^2) + offset$ $mem \leftarrow LoadMemory(vAddr, 32)$ $AR[t] \leftarrow mem_{31..0}$

Assembler

Instruction	
L32R	l32r at, target

Chapter 4

Xtensa Architecture Extensions

4.1 Windowed Option

4.1.1 Instructions encoded with RRR format

Encoding

23 20	19 16	15 12	11 8	7 4	3 0		
0000	0000	0001	s	t	0000	$MOVSP$	$AR[t] \leftarrow AR[s]$
0100	0000	1000	0000	imm[3..0]	0000	$ROTW$	$WINDOWBASE \leftarrow WINDOWBASE + (imm_3^{28} \| imm_{3..0})$
0000	0000	0011	0100	0000	0000	$RFWO$	Return from window overflow exception
0000	0000	0011	0101	0000	0000	$RFWU$	Return from window underflow exception

Assembler

Instruction	
MOVSP	movsp at, as
ROTW	rotw imm
RFWO	rfwo
RFWU	rfwu

4.1.2 Instructions encoded with CALL format

Encoding

23	5 4	3 0		
imm[17..0]	01	0101	$CALL4$	$PS.CALLINC \leftarrow 01$ $AR[0100] \leftarrow 01 \| next(PC)_{31..2}$ $offset \leftarrow sign\_extend(imm)$ $PC \leftarrow (PC_{31..2} + offset_{31..0} + 1)_{31..2} \| 0^2$
imm[17..0]	10	0101	$CALL8$	$PS.CALLINC \leftarrow 10$ $AR[1000] \leftarrow 10 \| next(PC)_{31..2}$ $offset \leftarrow sign\_extend(imm)$ $PC \leftarrow (PC_{31..2} + offset_{31..0} + 1)_{31..2} \| 0^2$
imm[17..0]	11	0101	$CALL12$	$PS.CALLINC \leftarrow 11$ $AR[1100] \leftarrow 11 \| next(PC)_{31..2}$ $offset \leftarrow sign\_extend(imm)$ $PC \leftarrow (PC_{31..2} + offset_{31..0} + 1)_{31..2} \| 0^2$

Assembler

Instruction	
CALL4	call4 target
CALL8	call8 target
CALL12	call12 target

4.1.3 Instructions encoded with CALLX format

Encoding

23..20	19..16	15..12	11..8	7..6	5..4	3..0		
0000	0000	0000	s	11	01	0000	$CALLX4$	$PS.CALLINC \leftarrow 01$ $AR[0100] \leftarrow 01 \| \| next(PC)_{31..2}$ $PC \leftarrow AR[s]$
0000	0000	0000	s	11	10	0000	$CALLX8$	$PS.CALLINC \leftarrow 10$ $AR[1000] \leftarrow 10 \| \| next(PC)_{31..2}$ $PC \leftarrow AR[s]$
0000	0000	0000	s	11	11	0000	$CALLX12$	$PS.CALLINC \leftarrow 11$ $AR[1100] \leftarrow 11 \| \| next(PC)_{31..2}$ $PC \leftarrow AR[s]$
0000	0000	0000	0000	10	01	0000	$RETW$	$n \leftarrow AR[0]_{31..30}$ $TMP \leftarrow PC_{31..30} \| \| AR[0]_{29..0}$ $WINDOWBASE \leftarrow$ $WINDOWBASE - (n \| \| 0^2)$ $PC \leftarrow TMP$

Assembler

Instruction	
CALLX4	callx4 as
CALLX8	callx8 as
CALLX12	callx12 as
RETW	retw

4.1.4 Instructions encoded with BRI12 format

Encoding

23..12	11..8	7..4	3..0		
imm[11..0]	s	0011	0110	$ENTRY$	s is from [a0..a3] $ci \leftarrow PS.CALLINC$ $AR[ci \| \| s_{1..0}] \leftarrow AR[s] - (0^{17} \| \| imm \| \| 0^3)$ $WINDOWBASE \leftarrow WINDOWBASE +$ $(ci \| \| 0^2)$

Assembler

Instruction	
ENTRY	entry as, imm

4.1.5 Instructions encoded with RRI4 format

Encoding

23		12 11	8 7	4 3	0		
0000	1001	r	s	t	0000	*L32E*	Load operation for use in window underflow and overflow exception handlers $offset \leftarrow (1^{26}\|\|r\|\|0^2)$ $vAddr \leftarrow AR[s] + offset$ $mem \leftarrow LoadMemory(vAddr, 32)$ $AR[t] \leftarrow mem_{31..0}$
0100	1001	r	s	t	0000	*S32E*	Store operation for use in window underflow and overflow exception handlers $offset \leftarrow (1^{26}\|\|r\|\|0^2)$ $vAddr \leftarrow AR[s] + offset$ $StoreMemory(vAddr, 32, AR[t]_{31..0})$

Assembler

Instruction	
L32E	l32e as, at, imm
S32E	s32e as, at, imm

4.2 Code Density Option

4.2.1 Instructions encoded with RRRN format

Encoding

15 12	11 8	7 4	3 0		
r	s	t	1010	*ADD.N*	$AR[r] \leftarrow AR[t] + AR[s]$

r	s	imm[3..0]	0010	*ADDI.N*	$if(imm = 0)then$
					$AR[r] \leftarrow 1^{32}$
					else
					$AR[r] \leftarrow AR[s] + imm$
					endif

1111	0000	0110	1101	*ILL.N*	Illegal instruction
imm[3..0]	s	t	1000	*L32I.N*	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $mem \leftarrow LoadMemory(vAddr, 32)$ $AR[t] \leftarrow mem_{31..0}$
0000	s	t	1101	*MOV.N*	$AR[s] \leftarrow AR[t]$
1111	0000	0011	1101	*NOP.N*	No operation
1111	0000	0000	1101	*RET.N*	$PC \leftarrow AR[0]$
1111	0000	0001	1101	*RETW.N*	$n \leftarrow AR[0]_{31..30}$ $TMP \leftarrow PC_{31..30}\|\|AR[0]_{29..0}$ $WINDOWBASE \leftarrow WINDOWBASE - (n\|\|0^2)$ $PC \leftarrow TMP$
imm[3..0]	s	t	1001	*S32I.N*	$offset \leftarrow sign\_extend(imm)$ $vAddr \leftarrow AR[s] + offset$ $StoreMemory(vAddr, 32, AR[t]_{31..0})$

Assembler

Instruction	
ADD.N	add.n ar, as, at
ADDI.N	addi.n ar, as, imm

ILL.N	ill.n
L32I.N	l32i.n at, as, imm
MOV.N	mov.n at, as
NOP.N	nop.n
RET.N	ret.n
RETW.N	retw.n
S32I.N	s32i.n at, as, imm

4.2.2 Instructions encoded with RI6 format

Encoding

15 12	11 8	7 4	3 0		

imm[3..0]	s	10imm[5..4]	1100	$BEQZ.N$	$offset \leftarrow sign\_extend(imm)$
					$condition \leftarrow (AR[s] = 0^{32})$
					if condition then
					$PC \leftarrow PC + offset + 4$
					endif

imm[3..0]	s	11imm[5..4]	1100	$BNEZ.N$	$offset \leftarrow sign\_extend(imm)$
					$condition \leftarrow (AR[s] >= 0^{32})$
					if condition then
					$PC \leftarrow PC + offset + 4$
					endif

Assembler

Instruction	
BEQZ.N	beqz.n as, target
BNEZ.N	bnez.n as, target

4.2.3 Instructions encoded with RI7 format

Encoding

15 12	11 8	7 4	3 0		
imm[3..0]	s	0imm[6..4]	1100	$MOVI.N$	$AR[s] \leftarrow sign\_extend(imm)$

Assembler

Instruction	
MOVI.N	movi.n as, imm

Chapter 5

ELF Object Files

5.1 Relocations

Assembler

Enum	ELF Reloc Type	Description
0	R_XTENSA_NONE	None
1	R_XTENSA_32	Runtime relocation
2	R_XTENSA_RTLD	Xtensa relocation used only by PLT entries in ELF shared objects
3	R_XTENSA_GLOB_DAT	Xtensa relocation for ELF shared objects
4	R_XTENSA_JMP_SLOT	Xtensa relocation for ELF shared objects
5	R_XTENSA_RELATIVE	Xtensa relocation for ELF shared objects
6	R_XTENSA_PLT	Xtensa relocation used only by PLT entries in ELF shared objects
8	R_XTENSA_OP0	Xtensa relocation for backward compatibility
9	R_XTENSA_OP1	Xtensa relocation for backward compatibility
10	R_XTENSA_OP2	Xtensa relocation for backward compatibility
11	R_XTENSA_ASM_EXPAND	Xtensa relocation to mark that the assembler expanded the instructions from an original target
12	R_XTENSA_ASM_SIMPLIFY	Xtensa relocation to mark that the linker should simplify assembler-expanded instructions
14	R_32_PCREL	PC relative relocation
15	R_XTENSA_GNU_VTINHERIT	GNU extension to enable C++ vtable
16	R_XTENSA_GNU_VTENTRY	GNU extension to enable C++ vtable
17	R_XTENSA_DIFF8	Xtensa relocations to mark the difference of two local symbols
18	R_XTENSA_DIFF16	Xtensa relocations to mark the difference of two local symbols
19	R_XTENSA_DIFF32	Xtensa relocations to mark the difference of two local symbols
20	R_XTENSA_SLOT0_OP	Generic Xtensa relocation for instruction operands
21	R_XTENSA_SLOT1_OP	Generic Xtensa relocation for instruction operands
22	R_XTENSA_SLOT2_OP	Generic Xtensa relocation for instruction operands
23	R_XTENSA_SLOT3_OP	Generic Xtensa relocation for instruction operands
24	R_XTENSA_SLOT4_OP	Generic Xtensa relocation for instruction operands
25	R_XTENSA_SLOT5_OP	Generic Xtensa relocation for instruction operands
26	R_XTENSA_SLOT6_OP	Generic Xtensa relocation for instruction operands
27	R_XTENSA_SLOT7_OP	Generic Xtensa relocation for instruction operands
28	R_XTENSA_SLOT8_OP	Generic Xtensa relocation for instruction operands
29	R_XTENSA_SLOT9_OP	Generic Xtensa relocation for instruction operands
30	R_XTENSA_SLOT10_OP	Generic Xtensa relocation for instruction operands
31	R_XTENSA_SLOT11_OP	Generic Xtensa relocation for instruction operands
32	R_XTENSA_SLOT12_OP	Generic Xtensa relocation for instruction operands
33	R_XTENSA_SLOT13_OP	Generic Xtensa relocation for instruction operands
34	R_XTENSA_SLOT14_OP	Generic Xtensa relocation for instruction operands
35	R_XTENSA_SLOT0_ALT	Alternate Xtensa relocation
36	R_XTENSA_SLOT1_ALT	Alternate Xtensa relocation

Assembler

Enum	ELF Reloc Type	Description
37	R_XTENSA_SLOT2_ALT	Alternate Xtensa relocation
38	R_XTENSA_SLOT3_ALT	Alternate Xtensa relocation
39	R_XTENSA_SLOT4_ALT	Alternate Xtensa relocation
40	R_XTENSA_SLOT5_ALT	Alternate Xtensa relocation
41	R_XTENSA_SLOT6_ALT	Alternate Xtensa relocation
42	R_XTENSA_SLOT7_ALT	Alternate Xtensa relocation
43	R_XTENSA_SLOT8_ALT	Alternate Xtensa relocation
44	R_XTENSA_SLOT9_ALT	Alternate Xtensa relocation
45	R_XTENSA_SLOT10_ALT	Alternate Xtensa relocation
46	R_XTENSA_SLOT11_ALT	Alternate Xtensa relocation
47	R_XTENSA_SLOT12_ALT	Alternate Xtensa relocation
48	R_XTENSA_SLOT13_ALT	Alternate Xtensa relocation
49	R_XTENSA_SLOT14_ALT	Alternate Xtensa relocation
50	R_XTENSA_TLSDESC_FN	TLS relocation
51	R_XTENSA_TLSDESC_ARG	TLS relocation
52	R_XTENSA_TLS_DTPOFF	TLS relocation
53	R_XTENSA_TLS_TPOFF	TLS relocation
54	R_XTENSA_TLS_FUNC	TLS relocation
55	R_XTENSA_TLS_ARG	TLS relocation
56	R_XTENSA_TLS_CALL	TLS relocation

Appendix A

Special Register Numbers

Special register numbers [5]

Register name	Register number	Comment
LBEG	0	Zero-overhead loop begin
LEND	1	Zero-overhead loop end
LCOUNT	2	Zero-overhead loop counter
SAR	3	Shift amount register
SCOMPARE1	12	Comparison value for conditional store instruction
WINDOWBASE	72	Mask of dirty register windows
WINDOWSTART	73	Offset of the current register window
PS	230	Processor state
VECBASE	231	Exception vector base address
CCOUNT	234	CPU cycle counter
CCOMPARE_0	240	Match value for CPU cycle counter
MISC_REG_0	244	Miscellaneous register (no special meaning)
MISC_REG_1	245	Miscellaneous register (no special meaning)
THREADPTR	231	Thread pointer

Bibliography

[1] Gnu binutils. URL: https://www.gnu.org/software/binutils/.

[2] Gnu compiler collection. URL: https://gcc.gnu.org/.

[3] Qemu. URL: https://www.qemu.org/.

[4] Whitepaper: Diamond standard processor cores. Technical report, Tensilica Inc., 2008. URL: https://ip.cadence.com/uploads/white_papers/Diamond_Tensilica.pdf.

[5] specreg.h, 2013. URL: https://github.com/qca/open-ath9k-htc-firmware/blob/c5830098/target_firmware/magpie_fw_dev/target/inc/xtensa/config/specreg.h.

[6] Steve Leibson. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*. Elsevier, 2006.