LXP32

a lightweight open source 32-bit CPU core

Technical Reference Manual

Version 1.0

Contents

1 Introduction **1**
- 1.1 Main features 1
- 1.2 Implementation estimates 2
- 1.3 Structure of this manual 2

2 Instruction set architecture **5**
- 2.1 Data format 5
- 2.2 Instruction format 5
- 2.3 Registers 6
- 2.4 Addressing 7
- 2.5 Stack 7
- 2.6 Calling procedures 8
- 2.7 Interrupt handling 9

3 Integration **11**
- 3.1 Overview 11
- 3.2 Ports 13
- 3.3 Generics 14
- 3.4 Clock and reset 15
- 3.5 Low Latency Interface 16
- 3.6 WISHBONE instruction bus 17
- 3.7 WISHBONE data bus 18
- 3.8 Interrupts 19
- 3.9 Synthesis and optimization 19

4 Simulation **23**
- 4.1 Requirements 23
- 4.2 Running simulation using makefiles 24
- 4.3 Running simulation manually 25
- 4.4 Testbench parameters 25

5 Development tools **27**
5.1 `lxp32asm` – Assembler and linker 27
5.2 `lxp32dump` – Disassembler 28
5.3 `wigen` – Interconnect generator 29
5.4 Building from source 30

A Instruction set reference **33**
A.1 List of instructions by group 33
A.2 Alphabetical list of instructions 34
add – Add 34
and – Bitwise And 34
call – Call Procedure 35
cjmp*xxx* – Compare and Jump 35
divs – Divide Signed 36
divu – Divide Unsigned 37
hlt – Halt 37
jmp – Jump 37
iret – Interrupt Return 38
lc – Load Constant 38
lsb – Load Signed Byte 39
lub – Load Unsigned Byte 39
lw – Load Word 40
mods – Modulo Signed 40
modu – Modulo Unsigned 41
mov – Move 41
mul – Multiply 41
nop – No Operation 42
not – Bitwise Not 42
or – Bitwise Or 42
ret – Return from Procedure 43
sb – Store Byte 43
sl – Shift Left 43
srs – Shift Right Signed 44
sru – Shift Right Unsigned 44
sub – Subtract 44
sw – Store Word 45
xor – Bitwise Exclusive Or 45

B Instruction cycle counts **47**

C LXP32 assembly language **49**

C.1 Comments 49
C.2 Literals 49
C.3 Symbols 50
C.4 Statements 51

D WISHBONE datasheet 53
D.1 Instruction bus (LXP32C only) 53
D.2 Data bus 54

Chapter 1

Introduction

1.1 Main features

LXP32 (*Lightweight eXecution Pipeline*) is a small 32-bit CPU IP core optimized for FPGA implementation. Its key features include:

- described in portable VHDL-93, not tied to any particular vendor;
- 3-stage pipeline;
- 256 registers implemented as a RAM block;
- simple instruction set with less than 30 distinct opcodes;
- separate instruction and data buses, optional instruction cache;
- WISHBONE compatible;
- 8 interrupts with hardwired priorities;
- optional divider.

Being a lightweight IP core, LXP32 also has certain limitations:

- no branch prediction;
- no floating-point unit;
- no memory management unit;
- no nested interrupt handling;
- no debugging facilities.

Two major hardware versions of the CPU are provided: LXP32U which does not include an instruction cache and uses the Low Latency Interface (Section 3.5) to fetch instructions, and LXP32C which fetches instructions over a cached WISHBONE bus protocol. These versions are otherwise identical and have the same instruction set architecture.

1.2 Implementation estimates

Typical results of LXP32 core FPGA implementation are presented in Table 1.1. Note that these data are only useful as rough estimates, since actual results depend greatly on tool versions and configuration, design constraints, device utilization ratio and other factors.

Data on two configurations are provided:

- *Compact*: LXP32U (without instruction cache), no divider, 2-cycle multiplier.
- *Full*: LXP32C (with instruction cache), divider, 2-cycle multiplier.

The slowest speed grade was used for clock frequency estimation.

1.3 Structure of this manual

General description of the LXP32 operation from a software developer's point of view can be found in Chapter 2, *Instruction set architecture*. Future versions of the LXP32 CPU are intended to be at least backwards compatible with this architecture.

Topics related to hardware, such as synthesis, implementation and interfacing other IP cores, are covered in Chapter 3, *Integration*. The LXP32 IP core package also includes testbenches which can be used to simulate the design as described in Chapter 4, *Simulation*.

Tools shipped as parts of the LXP32 IP core package (assembler/linker, disassembler and interconnect generator) are documented in Chapter 5, *Development tools*.

Appendices include a detailed description of the LXP32 instruction set, instruction cycle counts and LXP32 assembly language definition. WISHBONE datasheet required by the WISHBONE specification is also provided.

Table 1.1: Typical results of LXP32 core FPGA implementation

Resource	Compact	Full
Altera® Cyclone® V 5CEBA2F23C8		
Logic Array Blocks (LABs)	79	119
ALMs	630	972
ALUTs	982	1531
Flip-flops	537	942
DSP blocks	3	3
RAM blocks (M10K)	2	3
Clock frequency	103.9 MHz	98.8 MHz
Microsemi® IGLOO®2 M2GL005-FG484		
Logic elements (LUT+DFF)	1529	2226
LUTs	1471	2157
Flip-flops	718	1181
Mathblocks (MACC)	3	3
RAM blocks (RAM1K18)	2	3
Clock frequency	111.7 MHz	107.8 MHz
Xilinx® Artix®-7 xc7a15tfgg484-1		
Slices	264	381
LUTs	809	1151
Flip-flops	527	923
DSP blocks (DSP48E1)	4	4
RAM blocks (RAMB18E1)	2	3
Clock frequency	113.6 MHz	109.3 MHz

Chapter 2

Instruction set architecture

2.1 Data format

Most LXP32 instructions work with 32-bit data words. A few instructions that address individual bytes use little-endian order, that is, the least significant byte is stored at the lowest address. Signed values are encoded in a 2's complement format.

2.2 Instruction format

All LXP32 instructions are encoded as 32-bit words, with the exception of **`lc`** (*Load Constant*), which occupies two adjacent 32-bit words. Instructions in memory must be aligned to word boundaries.

Most arithmetic and logical instructions take two source operands and write the result to an independent destination register. General instruction format is presented on Figure 2.1.

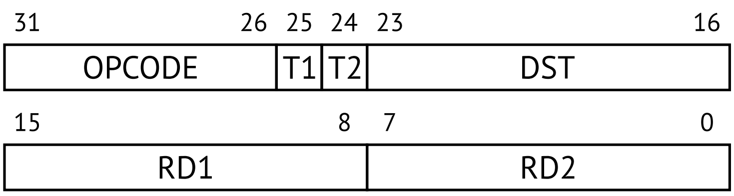

Figure 2.1: LXP32 instruction format

This format includes the following fields:

1. OPCODE – a 6-bit instruction code (see Appendix A).

2. T1 – type of the RD1 field.
3. T2 – type of the RD2 field.
4. DST – register number (usually the destination register).
5. RD1 – register/direct operand 1.
6. RD2 – register/direct operand 2.

Some of these fields may not have meaning for a particular instruction; such unused fields are replaced with zeros.

DST field specifies one of the 256 LXP32 registers. RD1 and RD2 fields can denote either source register operands or direct (immediate) operands: if the corresponding T field is 1, RD value is a register number, otherwise it is interpreted as a direct signed byte in a 2's complement format (valid values range from -128 to 127).

For example, consider the following instruction that adds `10` to `r0` and writes the result to `r1`:

```
add r1, r0, 10
```

In this example, OPCODE is `010000`, T1 is `1`, T2 is `0`, DST is `00000001`, RD1 is `00000000` and RD2 is `00001010`. Hence, the instruction is encoded as `0x4201000A`.

For convenience, some instructions have alias mnemonics. For example, LXP32 does not have a distinct **mov** opcode: instead, **mov** `dst, src` is an alias for **add** `dst, src, 0`.

A complete list of LXP32 instructions is provided in Appendix A.

2.3 Registers

LXP32 has 256 registers denoted as `r0` – `r255`. The first 240 of them (from `r0` to `r239`) are general-purpose registers (GPR), the last 16 (from `r240` to `r255`) are special-purpose registers (SPR). For convenience, some special-purpose registers have alias names: for example, `r255` can be also referred to as `sp` (stack pointer). Special purpose registers are listed in Table 2.1. Some of these registers are reserved: the software should not access them.

All registers are zero-initialized during the CPU reset.

Table 2.1: LXP32 special-purpose registers

Alias name	Generic name	Description
`iv0`	`r240`	Interrupt vector 0 (Section 2.7)
`iv1`	`r241`	Interrupt vector 1 (Section 2.7)
`iv2`	`r242`	Interrupt vector 2 (Section 2.7)
`iv3`	`r243`	Interrupt vector 3 (Section 2.7)
`iv4`	`r244`	Interrupt vector 4 (Section 2.7)
`iv5`	`r245`	Interrupt vector 5 (Section 2.7)
`iv6`	`r246`	Interrupt vector 6 (Section 2.7)
`iv7`	`r247`	Interrupt vector 7 (Section 2.7)
—	`r248`–`r251`	*Reserved*
`cr`	`r252`	Control register (Section 2.7)
`irp`	`r253`	Interrupt return pointer (Section 2.7)
`rp`	`r254`	Return pointer (Section 2.6)
`sp`	`r255`	Stack pointer (Section 2.5)

2.4 Addressing

All addressing in LXP32 is indirect. In order to access a memory location, its address must be stored in a register; any available register can be used for this purpose.

Some instructions, namely **`lsb`** (*Load Signed Byte*), **`lub`** (*Load Unsigned Byte*) and **`sb`** (*Store Byte*) provide byte-granular access, in which case all 32 bits in the address are significant. Otherwise the least two address bits are ignored as LXP32 doesn't support unaligned access to 32-bit data words (during simulation, a warning is emitted if such a transaction is attempted).

A special rule applies to pointers that refer to instructions: since instructions are always word-aligned, the least significant bit is interpreted as the `IRF` (*Interrupt Return Flag*). See Section 2.7 for details.

2.5 Stack

The current pointer to the top of the stack is stored in the `sp` register. To the hardware this register is not different from general purpose registers, that is, in no situation does the CPU access the stack implicitly (procedure calls and interrupts use register-based conventions).

Software can access the stack as follows:

```
// push r0 on the stack
sub sp, sp, 4
sw sp, r0
// pop r0 from the stack
lw r0, sp
add sp, sp, 4
```

Before using the stack, the `sp` register must be set up to point to a valid memory location. The simplest software can operate stackless, or even without data memory altogether if registers are enough to store the program state.

2.6 Calling procedures

LXP32 provides a **`call`** instruction which stores the address of the next instruction in the `rp` register and transfers execution to the procedure pointed by **`call`** operand. Return from a procedure is performed by **`jmp`** `rp` instruction which also has **`ret`** alias.

If a procedure must in turn call some procedure itself, the return pointer in the `rp` register will be overwritten by the **`call`** instruction. Hence the procedure must save its value somewhere; the most general solution is to use the stack:

```
sub sp, sp, 4
sw sp, rp
...
call r1
...
lw rp, sp
add sp, sp, 4
ret
```

Procedures that don't use the **`call`** instruction (sometimes called *leaf* procedures) don't need to save the `rp` value.

Since **`ret`** is just an alias for **`jmp`** `rp`, one can also use *Compare and Jump* instructions (**`cjmp`*`xxx`***) to perform a conditional procedure return.

Although the LXP32 architecture doesn't mandate any particular calling convention, some general recommendations are presented below:

1. Pass arguments through the `r1`–`r31` registers.

2. Return value through the `r0` register.

3. Designate `r0`–`r31` registers as *caller-saved*, that is, they are not guaranteed to be preserved during procedure calls and must be saved by the caller if needed. The procedure can use them for any purpose, regardless of whether they are used to pass arguments and/or return values. For obvious reasons, this rule does not apply to interrupt handlers.

2.7 Interrupt handling

Control register

LXP32 supports 8 interrupts with hardwired priority levels (interrupts with lower vector numbers have higher priority). Interrupts vectors (pointers to interrupt handlers) are stored in the `iv0`–`iv7` registers. Interrupt handling is controlled by the `cr` register (Table 2.2).

Table 2.2: Control register

Bit	Description
0	Enable interrupt 0
1	Enable interrupt 1
	…
7	Enable interrupt 7
8	Temporarily block interrupt 0
9	Temporarily block interrupt 1
	…
15	Temporarily block interrupt 7
31–16	*Reserved*

Disabled interrupts are ignored altogether: if the CPU receives an interrupt request signal while the corresponding interrupt is disabled, the interrupt handler will not be called even if the interrupt is enabled later. Conversely, temporarily blocked interrupts are still registered, but their handlers are not called until they are unblocked.

Like other registers, `cr` is zero-initialized during the CPU reset, meaning that no interrupts are initially enabled.

Invoking interrupt handlers

Interrupt handlers are invoked by the CPU similarly to procedures (Section 2.6), the difference being that in this case return address is stored in the `irp` register (as opposed to `rp`), and the least significant bit of the register (`IRF` – *Interrupt Return Flag*) is set.

An interrupt handler returns using the **`jmp`** `irp` instruction which also has **`iret`** alias. Until the interrupt handler returns, the CPU will defer further interrupt processing (although incoming interrupt requests will still be registered). This also means that `irp` register value will not be unexpectedly overwritten. When executing the **`jmp`** `irp` instruction, the CPU will recognize the `IRF` flag and resume interrupt processing as usual. This behavior can be exploited to perform a conditional return from the interrupt handler, similarly to the technique described in Section 2.6 for conditional procedure returns.

Another technique can be useful when waiting for a single event, such as a coprocessor finishing its job: the interrupt handler can be set up to return to a designated address instead of the address stored in the `irp` register. This designated address must have the `IRF` flag set, otherwise all further interrupt processing will be disabled:

```
    lc r0, continue@1 // IRF flag
    lc iv0, handler
    ... // issue coprocessor command
    hlt // wait for an interrupt
continue:
    ... // the execution will continue here
handler:
    jmp r0
```

Chapter 3

Integration

3.1 Overview

The LXP32 IP core is delivered in a form of a synthesizable RTL description expressed in VHDL-93. It does not use any technology specific primitives and should work out of the box with major FPGA synthesis software. LXP32 can be integrated in both VHDL and Verilog® based SoC designs.

Major LXP32 hardware versions have separate top-level design units:

- `lxp32u_top` – LXP32U (without instruction cache),
- `lxp32c_top` – LXP32C (with instruction cache).

A high level block diagram of the CPU is presented on Figure 3.1. Schematic symbols for LXP32U and LXP32C are shown on Figure 3.2.

LXP32U uses the Low Latency Interface (Section 3.5) to fetch instructions. This interface is designed to interact with low latency on-chip peripherals such as RAM blocks or similar devices that are generally expected to return data word after one cycle since the instruction address has been set. It can be also connected to a custom (external) instruction cache.

To achieve the least possible latency, some LLI signals are not registered. For this reason the LLI is not suitable for interaction with off-chip peripherals.

LXP32C fetches instructions over the WISHBONE instruction bus. To maximize throughput, it supports the WISHBONE registered feedback signals [CTI_O()] and [BTE_O()]. All outputs on this bus are registered. This version is recommended for use with high latency memory devices such as SDRAM chips, as well as for situations where LLI combinatorial delays are unacceptable.

Both LXP32U and LXP32C use WISHBONE protocol for the data bus.

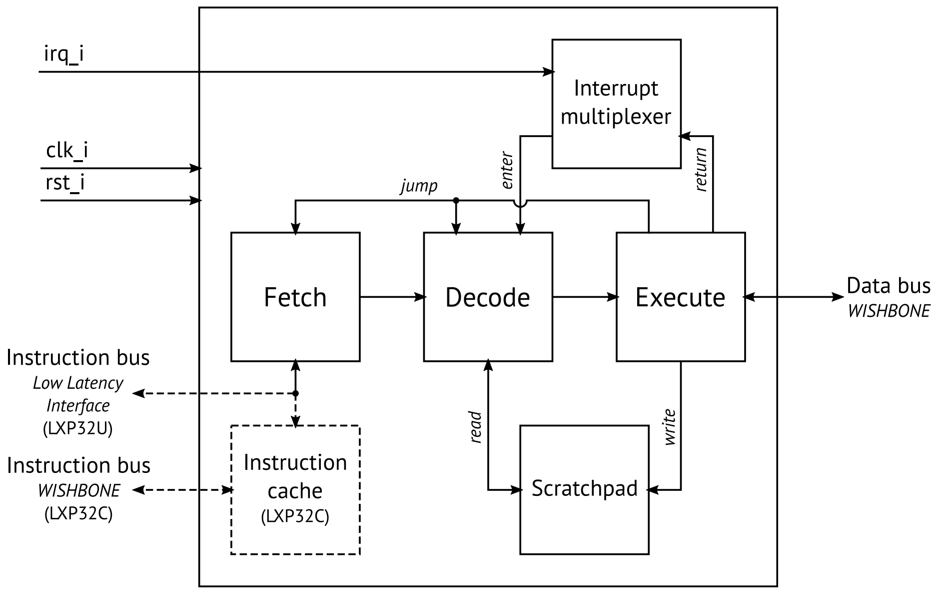

Figure 3.1: LXP32 CPU block diagram

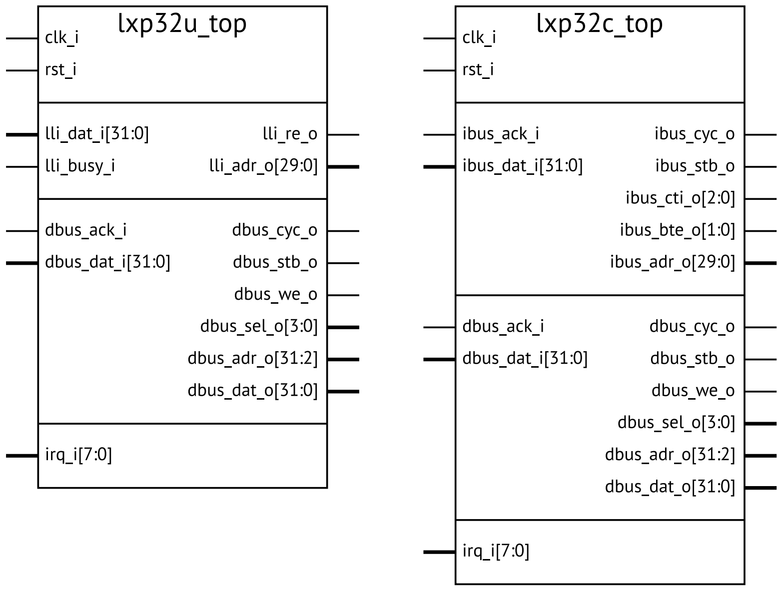

Figure 3.2: Schematic symbols for LXP32U and LXP32C

3.2 Ports

Port	Direction	Bus width	Description
		Global signals	
clk_i	in	1	System clock
rst_i	in	1	Synchronous reset, active high
		Instruction bus – Low Latency Interface (LXP32U only)	
lli_re_o	out	1	Read enable output, active high
lli_adr_o	out	30	Address output
lli_dat_i	in	32	Data input
lli_busy_i	in	1	Busy flag input, active high
		Instruction bus – WISHBONE (LXP32C only)	
ibus_cyc_o	out	1	Cycle output
ibus_stb_o	out	1	Strobe output
ibus_cti_o	out	3	Cycle type identifier
ibus_bte_o	out	2	Burst type extension
ibus_ack_i	in	1	Acknowledge input
ibus_adr_o	out	30	Address output
ibus_dat_i	in	32	Data input
		Data bus	
dbus_cyc_o	out	1	Cycle output
dbus_stb_o	out	1	Strobe output
dbus_we_o	out	1	Write enable output
dbus_sel_o	out	4	Select output
dbus_ack_i	in	1	Acknowledge input
dbus_adr_o	out	30	Address output
dbus_dat_o	out	32	Data output
dbus_dat_i	in	32	Data input
		Other ports	
irq_i	in	8	Interrupt requests

3.3 Generics

The following generics can be used to configure the LXP32 IP core parameters.

DBUS_RMW

By default, LXP32 uses the `dbus_sel_o` (byte enable) port to perform byte-granular write transactions initiated by the **`sb`** (*Store Byte*) instruction. If this option is set to `true`, `dbus_sel_o` is always tied to `"1111"`, and byte-granular write access is performed using the RMW (read-modify-write) cycle. The latter method is slower, but can work with slaves that do not have the [SEL_I()] port.

This feature requires data bus transactions to be idempotent, that is, repeating a transaction must not alter the slave state. Care should be taken with non-memory slaves to ensure that this condition is satisfied.

DIVIDER_EN

LXP32 includes a divider unit which occupies a considerable amount of resources. It can be excluded by setting this option to `false`.

IBUS_BURST_SIZE

Instruction bus burst size. Default value is 16. Only for LXP32C.

IBUS_PREFETCH_SIZE

Number of words that the instruction cache will read ahead from the current instruction pointer. Default value is 32. Only for LXP32C.

MUL_ARCH

LXP32 provides three multiplier options:

- `"dsp"` is the fastest architecture designed for technologies that provide fast parallel 16×16 multipliers, which includes most modern FPGA families. One multiplication takes 2 clock cycles.
- `"opt"` architecture uses a semi-parallel multiplication algorithm based on carry-save accumulation of partial products. It is designed for technologies that do not provide fast 16×16 multipliers. One multiplication takes 6 clock cycles.

- "seq" is a fully sequential design. One multiplication takes 34 clock cycles.

The default multiplier architecture is "dsp". This option is recommended for most modern FPGA devices regardless of optimization goal since it is not only the fastest, but also occupies the least amount of general-purpose logic resources. However, it will create a timing bottleneck on technologies that lack fast multipliers.

For older FPGA families that don't provide dedicated multipliers the "opt" architecture can be used if decent throughput is still needed. It is designed to avoid creating a timing bottleneck on such technologies. Alternatively, "seq" architecture can be used when throughput is not a concern.

START_ADDR

Address of the first instruction to be executed after CPU reset. Default value is 0. Note that it is a 30-bit value as it is used to address 32-bit words, not bytes.

3.4 Clock and reset

All flip-flops in the CPU are triggered by a rising edge of the clk_i signal. No specific requirements are imposed on the clk_i signal apart from usual constraints on setup and hold times.

LXP32 is reset synchronously when the rst_i signal is asserted. If the system reset signal comes from an asynchronous source, a synchronization circuit must be used; an example of such a circuit is shown on Figure 3.3.

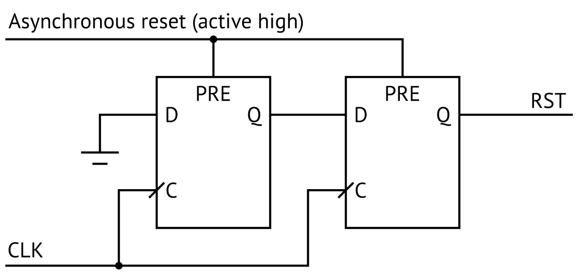

Figure 3.3: Reset synchronization circuit

In SRAM-based FPGAs flip-flops and RAM blocks have deterministic state after a bitstream is loaded. On such technologies LXP32 can operate without reset. In this case the `rst_i` port can be tied to a logical `0` in the RTL design to allow the synthesizer to remove redundant logic.

`clk_i` and `rst_i` signals also serve the role of [CLK_I] and [RST_I] WISHBONE signals, respectively, for both instruction and data buses.

3.5 Low Latency Interface

Low Latency Interface is a simple pipelined synchronous protocol with a typical latency of 1 cycle used by LXP32U to fetch instructions. Its timing diagram is shown on Figure 3.4. The request is considered valid when `lli_re_o` is high and `lli_busy_i` is low on the same clock cycle. On the next cycle after the request is valid the slave must either produce data on `lmi_dat_i` or assert `lli_busy_i` to indicate that data are not ready. Note that the values of `lli_re_o` and `lli_adr_o` are not guaranteed to be preserved by the CPU while the slave is busy.

The simplest, "always ready" slaves such as on-chip RAM blocks can be trivially connected to the LLI by connecting address, data and read enable ports and tying the `lli_busy_i` signal to a logical `0`. Slaves are also allowed to introduce wait states, which makes it possible to implement external caching.

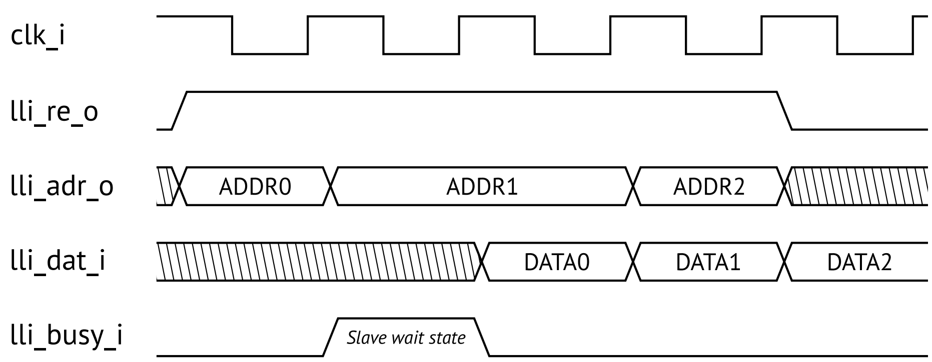

Figure 3.4: Low Latency Interface timing diagram (LXP32U)

Note that the `lli_adr_o` signal has a width of 30 bits since it addresses words, not bytes (instructions are always word-aligned).

Since this interface is not registered, it is not suitable for interaction with off-chip peripherals. Also, care should be taken to avoid introducing too much additional combinatorial delay on its outputs.

3.6 WISHBONE instruction bus

The LXP32C CPU fetches instructions over the WISHBONE bus. Its parameters are defined in the WISHBONE datasheet (Appendix D). For a detailed description of the bus protocol refer to the WISHBONE specification, revision B3.

With classic WISHBONE handshake decent throughput can be only achieved when the slave is able to terminate cycles asynchronously. It is usually possible only for the simplest slaves, which should probably be using the Low Latency Interface in the first place. To maximize throughput for complex, high latency slaves, LXP32C instruction bus uses optional WISHBONE address tags [CTI_O()] (Cycle Type Identifier) and [BTE_O()] (Burst Type Extension). These signals are hints allowing the slave to predict the address that will be set by the master in the next cycle and prepare data in advance. The slave can ignore these hints, processing requests as classic WISHBONE cycles, although performance would almost certainly suffer in this case.

A typical LXP32C instruction bus burst timing diagram is shown on Figure 3.5.

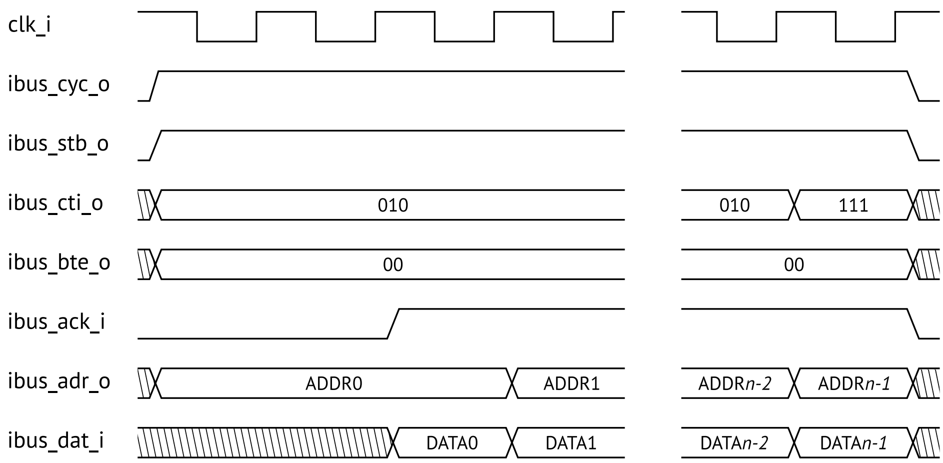

Figure 3.5: Typical WISHBONE instruction bus burst (LXP32C)

3.7 WISHBONE data bus

LXP32 uses the WISHBONE bus to interact with data memory and other peripherals. This bus is distinct from the instruction bus; its parameters are defined in the WISHBONE datasheet (Appendix D).

This bus uses a 30-bit `dbus_adr_o` port to address 32-bit words; the `dbus_sel_o` port is used to select individual bytes to be written or read. Alternatively, with the `DBUS_RMW` option (Section 3.3) the `dbus_sel_o` port is not used; byte-granular write access is performed using the read-modify-write cycle instead.

For a detailed description of the bus protocol refer to the WISHBONE specification, revision B3.

Typical timing diagrams for write and read cycles are shown on Figure 3.6. In these examples the peripheral terminates the cycle asynchronously; however, it can also introduce wait states by delaying the `dbus_ack_i` signal.

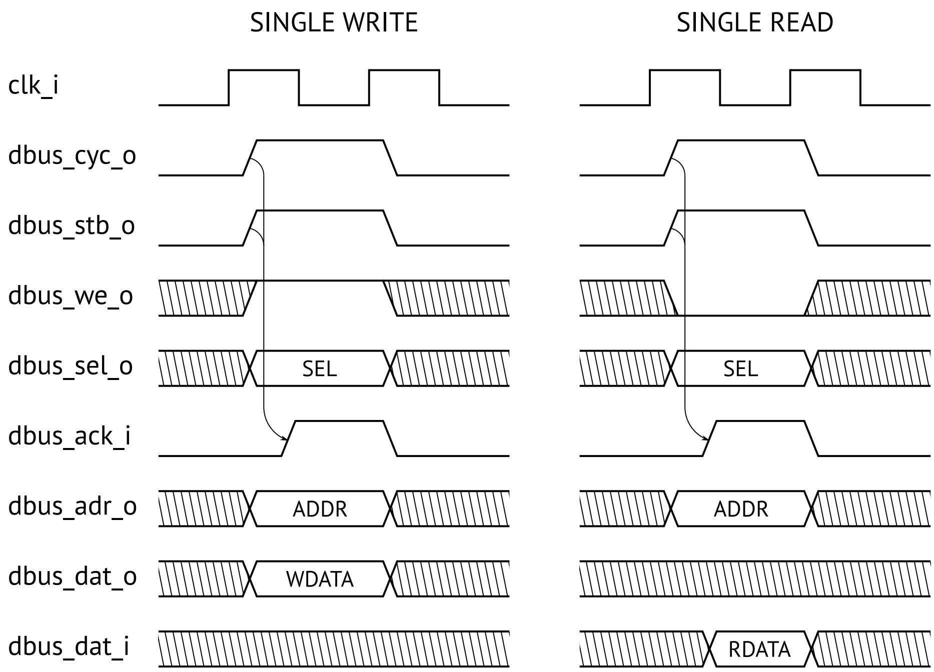

Figure 3.6: Typical WISHBONE data bus WRITE and READ cycles

3.8 Interrupts

LXP32 registers an interrupt condition when the corresponding request signal goes from `0` to `1`. Transitions from `1` to `0` are ignored. All interrupt request signals must be synchronous with the system clock (`clk_i`); if coming from an asynchronous source, they must be synchronized using a sequence of at least two flip-flops clocked by `clk_i`. These flip-flops are not included in the LXP32 core in order not to increase interrupt processing delay for interrupt sources that are inherently synchronous. Failure to properly synchronize interrupt request signals will cause timing violations that will manifest itself as intermittent, hard to debug faults.

3.9 Synthesis and optimization

Technology specific primitives

LXP32 RTL design is described in behavioral VHDL. However, it can also benefit from certain special resources provided by most FPGA devices, namely, RAM blocks and dedicated multipliers. For improved portability, hardware description that can potentially be mapped to such resources is localized in separate design units:

- `lxp32_ram256x32` – a dual-port synchronous 256 × 32 bit RAM with one write port and one read port;
- `lxp32_mul16x16` – an unsigned 16 × 16 multiplier with an output register.

These design units contain behavioral description of respective hardware that is recognizable by FPGA synthesis tools. Usually no adjustments are needed as the synthesizer will automatically infer an appropriate primitive from its behavioral description. If automatic inference produces unsatisfactory results, these design units can be replaced with library element wrappers. The same is true for ASIC logic synthesis software which is unlikely to infer complex primitives.

General optimization guidelines

This subsection contains general advice on achieving satisfactory synthesis results regardless of the optimization goal. Some of these suggestions are also mentioned in other parts of this manual.

1. If the technology doesn't provide dedicated multiplier resources, consider using `"opt"` or `"seq"` multiplier architecture (Section 3.3).

2. Ensure that the instruction bus has adequate throughput. For LXP32C, check that the slave supports the WISHBONE registered feedback signals [CTI_I()] and [BTE_I()].

3. Multiplexing instruction and data buses, or connecting them to the same interconnect that allows only one master at a time to be active (i.e. *shared bus* interconnect topology) is not recommended. If you absolutely must do so, assign a higher priority level to the data bus, otherwise instruction prefetches will massively slow down data transactions.

Optimizing for timing

1. Set up reasonable timing constraints. Do not overconstrain the design by more that 10–15 %.

2. Analyze the worst path. The natural LXP32 timing bottleneck usually goes from the scratchpad (register file) output through the ALU (in the Execute stage) to the scratchpad input. If timing analysis lists other critical paths, the problem can lie elsewhere. If the `rst_i` signal becomes a bottleneck, promote it to a global network or, with SRAM-based FPGAs, consider operating without reset (see Section 3.4). Critical paths affecting the WISHBONE state machines could indicate problems with interconnect performance.

3. Configure the synthesis tool to reduce the fanout limit. Note that setting this limit to a too small value can lead to an opposite effect.

4. Synthesis tools can support additional options to improve timing, such as the *Retiming* algorithm which rearranges registers and combinatorial logic across the pipeline in attempt to balance delays. The efficiency of such algorithms is not very predictable. In general, sloppy designs are the most likely to benefit from it, while for a carefully designed circuit timing can sometimes get worse.

Optimizing for area

1. Consider excluding the divider if not using it (see Section 3.3).

2. Relaxing timing constraints can sometimes allow the synthesizer to produce a more area-efficient circuit.

3. Increase the fanout limit in the synthesizer settings to reduce buffer replication.

Chapter 4

Simulation

LXP32 package includes an automated verification environment (self-checking testbench) which verifies the LXP32 CPU functional correctness. The environment consists of two major parts: a test platform which is a SoC-like design providing peripherals for the CPU to interact with, and the testbench itself which loads test firmware and monitors the platform's output signals. Like the CPU itself, the test environment is written in VHDL-93.

A separate testbench for the instruction cache (`lxp32_icache`) is also provided. It can be invoked similarly to the main CPU testbench.

4.1 Requirements

The following software is required to simulate the LXP32 design:

- An HDL simulator supporting VHDL-93. LXP32 package includes scripts (makefiles) for the following simulators:
  - GHDL – a free and open-source VHDL simulator which supports multiple operating systems[1];
  - Mentor Graphics® ModelSim® simulator (`vsim`);
  - Xilinx® Vivado® Simulator (`xsim`).

  With GHDL, a waveform viewer such as GTKWave is also recommended (Figure 4.1)[2].

[1] `http://ghdl.free.fr/`

[2] `http://gtkwave.sourceforge.net/`

Some FPGA vendors provide limited versions of the ModelSim® simulator for free as parts of their design suites. These versions should suffice for LXP32 simulation.

Other simulators can be used with some preparations (Section 4.3).

- GNU `make` and `coreutils` are needed to simulate the design using the provided makefiles. Under Microsoft® Windows®, MSYS or Cygwin can be used.
- LXP32 assembler/linker program (`lxp32asm`) must be present (Section 5.1). A prebuilt executable for Microsoft® Windows® is already included in the LXP32 package, for other operating systems `lxp32asm` must be built from source (Section 5.4).

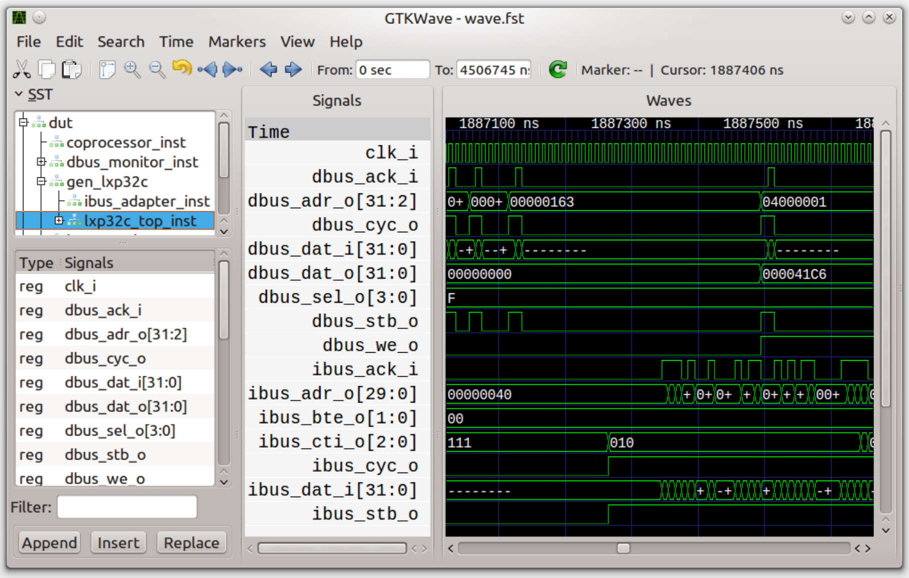

Figure 4.1: GTKWave displaying the LXP32 waveform dump produced by GHDL

4.2 Running simulation using makefiles

To simulate the design, go to the `verify/lxp32/run/`*`<simulator>`* directory and run `make`. The following make targets are supported:

- `batch` – simulate the design in batch mode. Results will be written to the standard output. This is the default target.
- `gui` – simulate the design in GUI mode. Note: since GHDL doesn't have a GUI, the simulation itself will be run in batch mode; upon a successful completion, GTKWave will be run automatically to display dumped waveforms.
- `compile` – compile only, don't run simulation.
- `clean` – delete all the produced artifacts.

4.3 Running simulation manually

LXP32 testbench can be also run manually. The following steps must be performed:

1. Compile the test firmware in the `verify/lxp32/src/firmware` directory:

   lxp32asm -f textio *filename*.asm -o *filename*.ram

   Produced `*.ram` files must be placed to the simulator's working directory.
2. Compile the LXP32 RTL description (`rtl` directory).
3. Compile the test platform (`verify/lxp32/src/platform` directory).
4. Compile the testbench itself (`verify/lxp32/src/tb` directory).
5. Simulate the `tb` design unit defined in the `tb.vhd` file.

4.4 Testbench parameters

Simulation parameters can be configured by overriding generics defined by the `tb` design unit:

- `MODEL_LXP32C` – simulate the LXP32C version. By default, this option is set to `true`. If set to `false`, LXP32U is simulated instead.
- `TEST_CASE` – if set to a non-empty string, specifies the file name of a test case to run. If set to an empty string (default), all tests are executed.

- THROTTLE_DBUS – perform pseudo-random data bus throttling. By default, this option is set to true.
- THROTTLE_IBUS – perform pseudo-random instruction bus throttling. By default, this option is set to true.
- VERBOSE – print more messages.

Chapter 5

Development tools

5.1 `lxp32asm` – Assembler and linker

`lxp32asm` is a combined assembler and linker for the LXP32 platform. It takes one or more input files and produces executable code for the CPU. Input files can be either source files in the LXP32 assembly language (Appendix C) or *linkable objects*. Linkable object is a relocatable format for storing compiled LXP32 code together with symbol information.

`lxp32asm` operates in two stages:

1. Compile.

   Source files are compiled to linkable objects.

2. Link.

   Linkable objects are combined into a single executable module. References to symbols defined in external modules are resolved at this stage.

In the simplest case there is only one input source file which doesn't contain external symbol references. If there are multiple input files, one of them must define the `entry` symbol at the beginning of the code.

Command line syntax

```
lxp32asm [ options | input files ]
```

Options supported by `lxp32asm` are listed below:

- `-a` *`align`* – section alignment. Must be a multiple of 4, default value is 4. Ignored in compile-only mode.
- `-b` *`addr`* – Base address, that is, address in memory where the executable image will be located. Must be a multiple of section alignment. Default value is 0. Ignored in compile-only mode.
- `-c` – compile only (skip the Link stage).
- `-f` *`fmt`* – select executable image format (see below for the list of supported formats). Ignored in compile-only mode.
- `-h`, `--help` – display a short help message and exit.
- `-i` *`dir`* – add *dir* to the list of directories used to search for included files. Multiple directories can be specified with multiple `-i` arguments.
- `-o` *`file`* – output file name.
- `-s` *`size`* – size of the executable image. Must be a multiple of 4. If total code size is less than the specified value, the executable image is padded with zeros. By default, image is not padded. This option is ignored in compile-only mode.
- `--` – do not interpret subsequent command line arguments as options. Can be used if there are input file names starting with dash.

Output formats

The following output formats are supported by `lxp32asm`:

- `bin` – raw binary image. This is the default format.
- `textio` – text format representing binary data as a sequence of zeros and ones. This format can be directly read from VHDL (using the `std.textio` package) or Verilog® (using the `$readmemb` function).
- `dec` – text format representing each word as a decimal number.
- `hex` – text format representing each word as a hexadecimal number.

5.2 `lxp32dump` – Disassembler

`lxp32dump` takes an executable image and produces a source file in LXP32 assembly language. The produced file is a valid program that can be compiled by `lxp32asm`.

Command line syntax

`lxp32dump [ options | input file ]`

Supported options are:

- `-b` *addr* – executable image base address, only used for comments.
- `-f` *fmt* – input file format. All `lxp32asm` output formats are supported. If this option is not supplied, autodetection is performed.
- `-h`, `--help` – display a short help message and exit.
- `-o` *file* – output file name. By default, the standard output stream is used.
- `--` – do not interpret subsequent command line arguments as options.

5.3 wigen – Interconnect generator

`wigen` is a small tool that generates VHDL description of a simple WISHBONE interconnect based on shared bus topology. It supports any number of masters and slaves.

For interconnects with multiple masters a priority-based arbitration circuit is inserted with lower-numbered masters taking precedence. However, when a bus cycle is in progress ([CYC_O] is asserted by the active master), the arbiter will not interrupt it even if a master with a higher priority level requests bus ownership.

Command line syntax

`wigen [ option(s) ] nm ns ma sa ps [ pg ]`

- *nm* – number of masters,
- *ns* – number of slaves,
- *ma* – master address width,
- *sa* – slave address width,
- *ps* – port size (8, 16, 32 or 64),

- *pg* – port granularity (8, 16, 32 or 64, default: the same as port size).

Supported options are:

- `-e` *`entity`* – name of the design entity (default is `"intercon"`).
- `-h`, `--help` – display a short help message and exit.
- `-o` *`file`* – output file name (default is *`entity`*`.vhd`).
- `-p` – generate pipelined arbiter (reduced combinatorial delays, increased latency).
- `-r` – generate WISHBONE registered feedback signals ([CTI_IO()] and [BTE_IO()]).
- `-u` – generate unsafe slave decoder (reduced combinatorial delays and resource usage, may not work properly if the address is invalid).

5.4 Building from source

Prebuilt tool executables for 32-bit Microsoft® Windows® are included in the LXP32 IP core package. For other platforms the tools must be built from source. Since they are developed in C++ using only the standard library, it should be possible to build them for any platform that provides a modern C++ compiler.

Requirements

The following software is required to build LXP32 tools from source:

1. A modern C++ compiler, such as Microsoft® Visual Studio® 2013 or newer, GCC 4.8 or newer, Clang 3.4 or newer.
2. CMake 3.3 or newer.

Build procedure

This software uses CMake as a build system generator. Building it involves two steps: first, the `cmake` program is invoked to generate a native build environment (a set of Makefiles or an IDE project); second, the generated environment is used to build the software.

Examples

In the following examples, it is assumed that the commands are run from the `tools` subdirectory of the LXP32 IP core package tree.

For Microsoft® Visual Studio®:

```
mkdir build
cd build
cmake -G "NMake Makefiles" ../src
nmake
nmake install
```

For MSYS:

```
mkdir build
cd build
cmake -G "MSYS Makefiles" ../src
make
make install
```

For MinGW without MSYS:

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ../src
mingw32-make
mingw32-make install
```

For other platforms:

```
mkdir build
cd build
cmake ../src
make
make install
```

More details can be found in the CMake documentation.

Appendix A

Instruction set reference

See Section 2.2 for a general description of LXP32 instruction encoding.

A.1 List of instructions by group

Instruction	Description	Opcode
	Data transfer	
`mov`	Move	alias for **`add`** `dst, src, 0`
`lc`	Load Constant	`000001`
`lw`	Load Word	`001000`
`lub`	Load Unsigned Byte	`001010`
`lsb`	Load Signed Byte	`001011`
`sw`	Store Word	`001100`
`sb`	Store Byte	`001110`
	Arithmetic operations	
`add`	Add	`010000`
`sub`	Subtract	`010001`
`mul`	Multiply	`010010`
`divu`	Divide Unsigned	`010100`
`divs`	Divide Signed	`010101`
`modu`	Modulo Unsigned	`010110`
`mods`	Modulo Signed	`010111`
	Bitwise operations	
`not`	Bitwise Not	alias for **`xor`** `dst, src, -1`
`and`	Bitwise And	`011000`

or	Bitwise Or	011001
xor	Bitwise Exclusive Or	011010
sl	Shift Left	011100
sru	Shift Right Unsigned	011110
srs	Shift Right Signed	011111
	Execution transfer	
jmp	Jump	100000
cjmp*xxx*	Compare and Jump	11*xxxx* (*xxxx* = condition)
call	Call Procedure	100001
ret	Return from Procedure	alias for **jmp** rp
iret	Interrupt Return	alias for **jmp** irp
	Miscellaneous instructions	
nop	No Operation	000000
hlt	Halt	000010

A.2 Alphabetical list of instructions

add – Add

Syntax

add DST, RD1, RD2

Encoding

010000 T1 T2 DST RD1 RD2

Example: **add** r2, r1, 10 → 0x4202010A

Operation

DST := RD1 + RD2

and – Bitwise And

Syntax

and DST, RD1, RD2

Encoding

`011000 T1 T2 DST RD1 RD2`

Example: **`and`** `r2, r1, 0x3F` → `0x6202013F`

Operation

`DST := RD1` ∧ `RD2`

`call` – Call Procedure

Save a pointer to the next instruction in the `rp` register and transfer execution to the address pointed by the operand.

Syntax

`call` `RD1`

Encoding

`100001 1 0 11111110 RD1 00000000`

RD1 must be a register.

Example: **`call`** `r1` → `0x86FE0100`

Operation

`rp :=` *`return_address`*

`goto RD1`

Pointer in RD1 is interpreted as described in Section 2.4.

`cjmp`***`xxx`*** – Compare and Jump

Compare two operands and transfer execution to the specified address if a condition is satisfied.

Syntax

`cjmpe` `DST, RD1, RD2` (Equal)

`cjmpne` `DST, RD1, RD2` (Not Equal)

`cjmpsg` `DST, RD1, RD2` (Signed Greater)

cjmpsge `DST, RD1, RD2` (Signed Greater or Equal)

cjmpsl `DST, RD1, RD2` (Signed Less)

cjmpsle `DST, RD1, RD2` (Signed Less or Equal)

cjmpug `DST, RD1, RD2` (Unsigned Greater)

cjmpuge `DST, RD1, RD2` (Unsigned Greater or Equal)

cjmpul `DST, RD1, RD2` (Unsigned Less)

cjmpule `DST, RD1, RD2` (Unsigned Less or Equal)

Encoding

`OPCODE T1 T2 DST RD1 RD2`

Opcodes:

cjmpe	111000
cjmpne	110100
cjmpsg	110001
cjmpsge	111001
cjmpug	110010
cjmpuge	111010

cjmpsl, **cjmpsle**, **cjmpul**, **cjmpule** are aliases for **cjmpsg**, **cjmpsge**, **cjmpug**, **cjmpuge**, respectively, with RD1 and RD2 operands swapped.

Example: **cjmpuge** `r2, r1, 5` → `0xEA020105`

Operation

`if` *`condition`* `then goto DST`

Pointer in DST is interpreted as described in Section 2.4. Unlike most instructions, **cjmp*xxx*** does not write to DST.

divs – Divide Signed

Syntax

divs `DST, RD1, RD2`

Encoding

`010101 T1 T2 DST RD1 RD2`

Example: **divs** `r2, r1, -3` → `0x560201FD`

Operation

`DST := (`*`signed`*`) RD1 / (`*`signed`*`) RD2`

The result is rounded towards zero and is undefined if RD2 is zero. If the CPU was configured without a divider, this instruction returns `0`.

`divu` – Divide Unsigned

Syntax

`divu` `DST, RD1, RD2`

Encoding

`010100 T1 T2 DST RD1 RD2`

Example: **`divu`** `r2, r1, 73` → `0x52020107`

Operation

`DST := RD1 / RD2`

The result is rounded towards zero and is undefined if RD2 is zero. If the CPU was configured without a divider, this instruction returns `0`.

`hlt` – Halt

Wait for an interrupt.

Syntax

`hlt`

Encoding

`000010 0 0 00000000 00000000 00000000`

Operation

Pause execution until an interrupt is received.

`jmp` – Jump

Transfer execution to the address pointed by the operand.

Syntax

jmp `RD1`

Encoding

`100000 1 0 00000000 RD1 00000000`

RD1 must be a register.

Example: **jmp** `r1` → `0x82000100`

Operation

`goto RD1`

Pointer in RD1 is interpreted as described in Section 2.4.

iret – Interrupt Return

Return from an interrupt handler.

Syntax

iret

Alias for **jmp** `irp`.

lc – Load Constant

Load a 32-bit word to the specified register. Note that values in the [-128; 127] range can be loaded more efficiently using the **mov** instruction alias.

Syntax

lc `DST, WORD32`

Encoding

`000001 0 0 DST 00000000 00000000 WORD32`

Unlike other instructions, **lc** occupies two 32-bit words.

Example: **lc** `r1, 0x12345678` → `0x04010000 0x12345678`

Operation

```
DST := WORD32
```

lsb – Load Signed Byte

Load a byte from the specified address to the register, performing sign extension.

Syntax

lsb DST, RD1

Encoding

```
001011 1 0 DST RD1 00000000
```

RD1 must be a register.

Example: **lsb** r2, r1 → 0x2E020100

Operation

DST := (*signed*) (*(BYTE*)RD1)

Pointer in RD1 is interpreted as described in Section 2.4.

lub – Load Unsigned Byte

Load a byte from the specified address to the register. Higher 24 bits are zeroed.

Syntax

lub DST, RD1

Encoding

```
001010 1 0 DST RD1 00000000
```

RD1 must be a register.

Example: **lub** r2, r1 → 0x2A020100

Operation

`DST := *(BYTE*)RD1`

Pointer in RD1 is interpreted as described in Section 2.4.

`lw` – Load Word

Load a word from the specified address to the register.

Syntax

`lw` `DST, RD1`

Encoding

`001000 1 0 DST RD1 00000000`

RD1 must be a register.

Example: **`lw`** `r2, r1` → `0x22020100`

Operation

`DST := *RD1`

Pointer in RD1 is interpreted as described in Section 2.4.

`mods` – Modulo Signed

Syntax

`mods` `DST, RD1, RD2`

Encoding

`010111 T1 T2 DST RD1 RD2`

Example: **`mods`** `r2, r1, 10` → `0x5E02010A`

Operation

`DST := (`*`signed`*`) RD1 mod (`*`signed`*`) RD2`

Modulo operation satisfies the following condition: if $Q = A/B$ and $R = A \bmod B$, then $A = B \cdot Q + R$.

The result is undefined if RD2 is zero. If the CPU was configured without a divider, this instruction returns `0`.

modu – Modulo Unsigned

Syntax

modu `DST, RD1, RD2`

Encoding

`010110 T1 T2 DST RD1 RD2`

Example: **modu** `r2, r1, 10` → `0x5A02010A`

Operation

`DST := RD1 mod RD2`

Modulo operation satisfies the following condition: if $Q = A/B$ and $R = A \bmod B$, then $A = B \cdot Q + R$.

The result is undefined if RD2 is zero. If the CPU was configured without a divider, this instruction returns `0`.

mov – Move

Syntax

mov `DST, RD1`

Alias for **add** `DST, RD1, 0`

mul – Multiply

Multiply two 32-bit values. The result is also 32-bit.

Syntax

mul `DST, RD1, RD2`

Encoding

`010010 T1 T2 DST RD1 RD2`

Example: **mul** `r2, r1, 3` → `0x4A020103`

Operation

`DST := RD1 * RD2`

Since the product width is the same as the operand width, the result of a multiplication does not depend on operand signedness.

`nop` – No Operation

Syntax

`nop`

Encoding

`000000 0 0 00000000 00000000 00000000`

Operation

This instruction does not alter the machine state.

`not` – Bitwise Not

Syntax

`not` `DST, RD1`

Alias for **`xor`** `DST, RD1, -1`.

`or` – Bitwise Or

Syntax

`or` `DST, RD1, RD2`

Encoding

`011001 T1 T2 DST RD1 RD2`

Example: **`or`** `r2, r1, 0x3F` → `0x6602013F`

Operation

`DST := RD1 ∨ RD2`

`ret` – Return from Procedure

Return from a procedure.

Syntax

`ret`

Alias for **`jmp`** `rp`.

`sb` – Store Byte

Store the lowest byte from the register to the specified address.

Syntax

`sb` `RD1, RD2`

Encoding

`001110 1 T2 00000000 RD1 RD2`

RD1 must be a register.

Example: **`sb`** `r2, r1` → `0x3B000201`

Operation

`*(BYTE*)RD1 := RD2 ∧ 0x000000FF`

Pointer in RD1 is interpreted as described in Section 2.4.

`sl` – Shift Left

Syntax

`sl` `DST, RD1, RD2`

Encoding

`011100 T1 T2 DST RD1 RD2`

Example: **`sl`** `r2, r1, 5` → `0x72020105`

Operation

`DST := RD1 << RD2`

The result is undefined if RD2 is outside the [0; 31] range.

srs – Shift Right Signed

Syntax

srs `DST, RD1, RD2`

Encoding

`011111 T1 T2 DST RD1 RD2`

Example: **srs** `r2, r1, 5` → `0x7E020105`

Operation

`DST := ((`*signed*`) RD1) >> RD2`

The result is undefined if RD2 is outside the [0; 31] range.

sru – Shift Right Unsigned

Syntax

sru `DST, RD1, RD2`

Encoding

`011110 T1 T2 DST RD1 RD2`

Example: **sru** `r2, r1, 5` → `0x7A020105`

Operation

`DST := RD1 >> RD2`

The result is undefined if RD2 is outside the [0; 31] range.

sub – Subtract

Syntax

sub `DST, RD1, RD2`

Encoding

`010001 T1 T2 DST RD1 RD2`

Example: **`sub`** `r2, r1, 5` → `0x46020105`

Operation

`DST := RD1 - RD2`

`sw` – Store Word

Store the value of the register to the specified address.

Syntax

`sw` `RD1, RD2`

Encoding

`001100 1 T2 00000000 RD1 RD2`

RD1 must be a register.

Example: **`sw`** `r2, r1` → `0x33000201`

Operation

`*RD1 := RD2`

Pointer in RD1 is interpreted as described in Section 2.4.

`xor` – Bitwise Exclusive Or

Syntax

`xor` `DST, RD1, RD2`

Encoding

`011010 T1 T2 DST RD1 RD2`

Example: **`xor`** `r2, r1, 0x3F` → `0x6A02013F`

Operation

`DST := RD1 ⊕ RD2`

Appendix B

Instruction cycle counts

Cycle counts for LXP32 instructions are listed in Table B.1. These values can change in future hardware revisions.

Table B.1: Instruction cycle counts

Instruction	Cycle count	Instruction	Cycle count
add	1	**modu**	37
and	1	**mov**	1
call	≥ 4[1]	**mul**	2, 6 or 34[3]
cjmp*xxx*	≥ 5[1]	**nop**	1
divs	37	**not**	1
divu	37	**or**	1
hlt	N/A	**ret**	≥ 4[1]
jmp	≥ 4[1]	**sb**	≥ 2[2]
iret	≥ 4[1]	**sl**	2
lc	2	**srs**	2
lsb	≥ 3[2]	**sru**	2
lub	≥ 3[2]	**sub**	1
lw	≥ 3[2]	**sw**	≥ 2[2]
mods	37	**xor**	1

[1]Depends on instruction bus latency. Includes pipeline flushing overhead.
[2]Depends on data bus latency.
[3]Depends on multiplier architecture set with the `MUL_ARCH` generic. See Section 3.3.

Appendix C

LXP32 assembly language

This appendix defines the assembly language used by LXP32 development tools.

C.1 Comments

LXP32 assembly language supports C style comments that can span across multiple lines and single-line C++ style comments:

```
/*
 * This is a comment.
 */

// This is also a comment
```

From a parser's point of view comments are equivalent to whitespace.

C.2 Literals

LXP32 assembly language uses numeric and string literals similar to those provided by the C programming language.

Numeric literals can take form of decimal, hexadecimal or octal numbers. Literals prefixed with `0x` are interpreted as hexadecimal, literals prefixed with `0` are interpreted as octal, other literals are interpreted as decimal. A numeric literal can also start with an unary plus or minus sign which is also considered a part of the literal.

String literals must be enclosed in double quotes. The most common escape sequences used in C are supported (Table C.1).

Table C.1: Escape sequences used in string literals

Sequence	Interpretation
`\\`	Backslash character
`\"`	Double quotation mark
`\'`	Single quotation mark (can be also used directly)
`\t`	Tabulation character
`\n`	Line feed
`\r`	Carriage return
`\x`*XX*	Character with a hexadecimal code of *XX* (1–2 digits)
`\`*XXX*	Character with an octal code of *XXX* (1–3 digits)

C.3 Symbols

Symbols are used to refer to data or code locations. LXP32 assembly language does not have distinct code labels and variable declarations: symbols are used in both these contexts.

Symbol names must be valid identifiers. A valid identifier must start with an alphabetic character or an underscore, and may contain alphanumeric characters and underscores.

A symbol definition must be the first token in a source code line followed by a colon. A symbol definition can occupy a separate line (in which case it refers to the following statement). Alternatively, a statement can follow the symbol definition on the same line.

A special `entry` symbol is used to inform the linker about program entry point if there are multiple input files. If defined, this symbol must precede the first instruction or data definition statement in the module.

Symbols can be used as operands to the **`lc`** instruction statement. A symbol reference can end with a `@`*n* sequence, where *n* is a numeric literal; in this case it is interpreted as an offset (in bytes) relative to the symbol definition. To refer to symbols defined in other modules, they must first be declared external using the **`#extern`** directive.

```
    lc r10, jump_label
    lc r11, data_word
// ...
    sw r11, r0 // store the value of r0 to the
               // location pointed by data_word
    jmp r10    // transfer execution to jump_label
// ...
```

```
jump_label:
    mov r1, r0
// ...
data_word:
    .word 0x12345678
```

C.4 Statements

Each statement occupies a single source code line. There are three kinds of statements:

- *Directives* provide directions for the assembler that do not directly cause code generation.
- *Data definition statements* insert arbitrary data to the generated code.
- *Instruction statements* insert LXP32 CPU instructions to the generated code.

Directives

The first token of a directive statement always starts with the # character.

`#define` *`identifier token`* `[` *`token`* `... ]`

Defines a macro that will be substituted with one or more tokens. The *`identifier`* must satisfy the requirements listed in Section C.3. Tokens can be anything, including keywords, identifiers, literals and separators (i.e. comma and colon characters).

`#extern` *`identifier`*

Declares *`identifier`* as an external symbol. Used to refer to symbols defined in other modules.

`#include` *`filename`*

Processes *`filename`* contents as it were literally inserted at the point of the **`#include`** directive. *`filename`* must be a string literal.

`#message` *`msg`*

Prints *`msg`* to the standard output stream. *`msg`* must be a string literal.

Data definition statements

The first token of a data definition statement always starts with the `.` (period) character.

.align [*alignment*]

Ensures that code generated by the next data definition or instruction statement is aligned to a multiple of *alignment* bytes, inserting padding zeros if needed. Default *alignment* is 4. Instructions and words are always at least word-aligned; the **.align** statement can be used to align them to a larger boundary, or to align byte data (see below).

.byte *token* [, *token* ...]

Inserts one or more bytes to the output code. Each *token* can be either a numeric literal with a valid range of [-128; 255] or a string literal. By default, bytes are not aligned.

.reserve *n*

Inserts *n* zero bytes to the output code.

.word *token* [, *token* ...]

Inserts one or more 32-bit words to the output code. Tokens must be numeric literals.

Instruction statements

Instruction statements have the following general syntax:

instruction [*operand* [, *operand* ...]]

Depending on the instruction, operands can be registers, numeric literals or symbols. Supported instructions are listed in Appendix A.

Appendix D

WISHBONE datasheet

D.1 Instruction bus (LXP32C only)

General information	
WISHBONE revision	B3
Type of interface	MASTER
Supported cycles	BLOCK READ
Signal names	
`clk_i`	CLK_I
`rst_i`	RST_I
`ibus_cyc_o`	CYC_O
`ibus_stb_o`	STB_O
`ibus_cti_o`	CTI_O()
`ibus_bte_o`	BTE_O()
`ibus_ack_i`	ACK_I
`ibus_adr_o`	ADR_O()
`ibus_dat_i`	DAT_I()
Supported tag signals	
`ibus_cti_o`	Cycle Type Identifier (address tag) “010” (Incrementing burst cycle) “111” (End-of-Burst)
`ibus_bte_o`	Burst Type Extension (address tag) “00” (Linear burst)
Dimensions	
Port size	32

Port granularity	32
Maximum operand size	32
Data transfer ordering	BIG/LITTLE ENDIAN
Data transfer sequence	UNDEFINED

D.2 Data bus

General information	
WISHBONE revision	B3
Type of interface	MASTER
Supported cycles	SINGLE READ/WRITE
	RMW
Signal names	
`clk_i`	CLK_I
`rst_i`	RST_I
`dbus_cyc_o`	CYC_O
`dbus_stb_o`	STB_O
`dbus_we_o`	WE_O
`dbus_sel_o`	SEL_O()
`dbus_ack_i`	ACK_I
`dbus_adr_o`	ADR_O()
`dbus_dat_o`	DAT_O()
`dbus_dat_i`	DAT_I()
Dimensions	
Port size	32
Port granularity	8
Maximum operand size	32
Data transfer ordering	LITTLE ENDIAN
Data transfer sequence	UNDEFINED