Copyright © 2016–2022 by Alex I. Kuznetsov.

The entire LXP32 IP core package, including the synthesizable RTL description, verification environment, documentation and software tools, is distributed under the terms of the MIT license reproduced below:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Mentor Graphics and ModelSim are trademarks of Mentor Graphics Corporation.
Microsemi and IGLOO are trademarks of Microsemi Corporation.
Microsoft, Windows and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
Verilog is a registered trademark of Cadence Design Systems, Inc.
Xilinx, Artix and Vivado are trademarks of Xilinx in the United States and other countries.
All other trademarks are the property of their respective owners.
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 13
   3.1 Overview ......................................................... 13
   3.2 Ports ............................................................. 15
   3.3 Generics ......................................................... 16
   3.4 Clock and reset ................................................ 17
   3.5 Low Latency Interface ....................................... 18
   3.6 WISHBONE instruction bus .................................. 19
   3.7 WISHBONE data bus .......................................... 20
   3.8 Interrupts ........................................................ 21
   3.9 Synthesis and optimization .................................. 22

4 Hardware architecture 25

5 Simulation 27
   5.1 Requirements .................................................... 27
   5.2 Running simulation using makefiles ......................... 28
   5.3 Running simulation manually ................................ 29
5.4 Testbench parameters ........................................... 29

6 Development tools ................................................. 31
  6.1 lxp32asm – Assembler and linker ............................. 31
  6.2 lxp32dump – Disassembler .................................... 33
  6.3 wigen – Interconnect generator ............................... 33
  6.4 Building from source ......................................... 34

A Instruction set reference .......................................... 37
  A.1 List of instructions by group .................................. 37
  A.2 Alphabetical list of instructions .............................. 38
    add – Add ......................................................... 38
    and – Bitwise And ............................................... 38
    call – Call Procedure ......................................... 39
    cjmpxxx – Compare and Jump .................................. 39
    divs – Divide Signed .......................................... 40
    divu – Divide Unsigned ....................................... 41
    hlt – Halt ....................................................... 41
    jmp – Jump ...................................................... 42
    iret – Interrupt Return ....................................... 42
    lc – Load Constant ............................................. 42
    lcs – Load Constant Short .................................... 43
    lsb – Load Signed Byte ....................................... 43
    lub – Load Unsigned Byte ..................................... 44
    lw – Load Word .................................................. 44
    mods – Modulo Signed ........................................ 45
    modu – Modulo Unsigned ....................................... 45
    mov – Move ...................................................... 46
    mul – Multiply .................................................. 46
    neg – Negate .................................................... 46
    nop – No Operation ............................................. 47
    not – Bitwise Not ............................................... 47
    or – Bitwise Or .................................................. 47
    ret – Return from Procedure .................................. 47
    sb – Store Byte ................................................ 48
    sl – Shift Left .................................................. 48
    srs – Shift Right Signed ...................................... 48
    sru – Shift Right Unsigned ................................... 49
    sub – Subtract ................................................... 49
    sw – Store Word ................................................ 50
    xor – Bitwise Exclusive Or .................................... 50
Chapter 1

Introduction

1.1 Main features

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

- portability (described in behavioral VHDL-93, not tied to any particular vendor);
- 3-stage hazard-free pipeline;
- 256 registers implemented as a RAM block;
- a simple instruction set with only 30 distinct opcodes;
- separate instruction and data buses, optional instruction cache;
- WISHBONE compatibility;
- 8 interrupts with hardwired priorities;
- optional divider.

As a lightweight CPU core, LXP32 lacks some features of more advanced processors, such as nested interrupt handling, debugging support, floating-point and memory management units. LXP32 is based on an original ISA (Instruction Set Architecture) which does not currently have a C compiler. It can be programmed in the assembly language covered by Appendix C.

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.

Table 1.1: Typical results of LXP32 core FPGA implementation

<table>
<thead>
<tr>
<th>Resource</th>
<th>Compact</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic elements (LUT+DFF)</td>
<td>1457</td>
<td>2086</td>
</tr>
<tr>
<td>LUTs</td>
<td>1421</td>
<td>1999</td>
</tr>
<tr>
<td>Flip-flops</td>
<td>706</td>
<td>1110</td>
</tr>
<tr>
<td>Mathblocks (MACC)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>RAM blocks (RAM1K18)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>107.7 MHz</td>
<td>109.2 MHz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Compact</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>235</td>
<td>365</td>
</tr>
<tr>
<td>LUTs</td>
<td>666</td>
<td>1011</td>
</tr>
<tr>
<td>Flip-flops</td>
<td>528</td>
<td>883</td>
</tr>
<tr>
<td>DSP blocks (DSP48E1)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>RAM blocks (RAMB18E1)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>111.9 MHz</td>
<td>120.2 MHz</td>
</tr>
</tbody>
</table>

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.
1.3. STRUCTURE OF THIS MANUAL

Topics related to hardware, such as synthesis, implementation and interfacing other IP cores, are covered in Chapter 3, Integration. A brief description of the LXP32 pipelined architecture is provided in Chapter 4, Hardware architecture. The LXP32 IP core package includes a verification environment (self-checking testbench) which can be used to simulate the design as described in Chapter 5, Simulation.

Documentation for tools shipped with the LXP32 IP core package (assembler/linker, disassembler and interconnect generator) is provided in Chapter 6, 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.
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 \texttt{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.

![LXP32 instruction format](image)

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.

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.
2.4. ADDRESSING

Table 2.1: LXP32 special-purpose registers

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

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.

LXP32 uses a 32-bit address space. Each address refers to an individual byte. Some instructions, namely \texttt{lsb} (Load Signed Byte), \texttt{lub} (Load Unsigned Byte) and \texttt{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 \texttt{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 saves the address of the next instruction in the rp register and transfers execution to the address stored in the register operand. Return from a procedure is performed by the jmp rp instruction which also has a ret alias.

If a procedure must in turn call a nested procedure itself, the return address in the rp register will be overwritten by the call instruction. Hence, unless it is a tail call (see below), the procedure must save the rp value somewhere; the most general solution is to use the stack:

```
sub sp, sp, 4
sw sp, rp
...  
lc r0, Nested_proc
    call r0
...  
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 (cjmpxxx) to perform a conditional procedure return. For example, consider the following procedure which calculates the absolute value of r1:

```
Abs_proc:
    cjmpsge rp, r1, 0 // return immediately if r1>=0
    neg r1, r1 // otherwise, negate r1
    ret // jmp rp
```
2.7. INTERRUPT HANDLING

A tail call is a special type of procedure call where the calling procedure calls a nested procedure as the last action before return. In such cases the call instruction can be replaced with jmp, so that when the nested procedure executes ret, it returns directly to the caller’s parent procedure.

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

1. Pass arguments and return values through the r1–r31 registers (a procedure can have multiple return values).
2. If necessary, the r0 register can be used to load the procedure address.
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.

2.7 Interrupt handling

Control register

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Enable interrupt 0</td>
</tr>
<tr>
<td></td>
<td>…</td>
</tr>
<tr>
<td>7</td>
<td>Enable interrupt 7</td>
</tr>
<tr>
<td>8</td>
<td>Interrupt 0 wake-up flag</td>
</tr>
<tr>
<td></td>
<td>…</td>
</tr>
<tr>
<td>15</td>
<td>Interrupt 7 wake-up flag</td>
</tr>
<tr>
<td>31–16</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Disabled interrupts are ignored altogether: if the CPU receives an interrupt request signal while the corresponding interrupt is disabled, the interrupt will not be processed even if it is enabled later.
Wake-up flag marks the interrupt as a *wake-up interrupt* (see below).
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 an `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 the $irp$ 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. It is also possible to perform a conditional return from the interrupt handler, similarly to the technique described in Section 2.6 for conditional procedure returns.

**Wake-up interrupts**

When a wake-up interrupt is received, the interrupt handler is not called, but the CPU still resumes execution if halted by the `hlt` instruction. The effect is similar to invoking an interrupt with an empty handler (containing only `iret`), but without the overhead of interrupt processing. Wake-up interrupts do not affect the CPU when it is not halted.

Unlike normal interrupts, wake-up interrupts are processed even when the CPU executes an interrupt handler for another interrupt.

**Non-returnable interrupts**

If an interrupt vector has the least significant bit (IRF) set, the CPU will resume interrupt processing immediately. One should not try to invoke `iret` from such a handler since the $irp$ register could have been overwritten by another interrupt. This technique can be useful when the CPU’s only task is to process external events:

```assembly
// Set the IRF to mark the interrupt as non-returnable
lc iv0, main_loop@1
mov cr, 1 // enable the interrupt
```
2.7. INTERRUPT HANDLING

\texttt{hlt} // wait for an interrupt request

\texttt{main\_loop:}

// Process the event...

\texttt{hlt} // wait for the next interrupt request

Note that \texttt{iret} is never called in this example.
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:

- \texttt{lxp32u\_top} – LXP32U (without instruction cache),
- \texttt{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 (LLI) described in Section 3.5 to fetch instructions. This interface is designed to interact with low latency on-chip peripherals such as RAM blocks. It works best with slaves that can return the instruction on the next cycle after its address has been set, although the slave can still introduce wait states if needed. Low Latency Interface can be also connected to a custom (external) instruction cache.

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

LXP32C is designed to work with high latency memory controllers and uses a simple instruction cache based on a ring buffer. The instructions are fetched over the WISHBONE instruction bus. To maximize throughput, the CPU makes use of the WISHBONE registered feedback signals [CTI\_O()] and [BTE\_O()]. All outputs on this bus are registered. This version is also
CHAPTER 3. INTEGRATION

Figure 3.1: LXP32 CPU block diagram

Figure 3.2: Schematic symbols for LXP32U and LXP32C
recommended for use in situations where LLI combinatorial delays are unacceptable.

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

### 3.2 Ports

<table>
<thead>
<tr>
<th>Port</th>
<th>Direction</th>
<th>Bus width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global signals</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk_i</td>
<td>in</td>
<td>1</td>
<td>System clock</td>
</tr>
<tr>
<td>rst_i</td>
<td>in</td>
<td>1</td>
<td>Synchronous reset, active high</td>
</tr>
<tr>
<td><strong>Instruction bus – Low Latency Interface (LXP32U only)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lli_re_o</td>
<td>out</td>
<td>1</td>
<td>Read enable output, active high</td>
</tr>
<tr>
<td>lli_adr_o</td>
<td>out</td>
<td>30</td>
<td>Address output</td>
</tr>
<tr>
<td>lli_dat_i</td>
<td>in</td>
<td>32</td>
<td>Data input</td>
</tr>
<tr>
<td>lli_busy_i</td>
<td>in</td>
<td>1</td>
<td>Busy flag input, active high</td>
</tr>
<tr>
<td><strong>Instruction bus – WISHBONE (LXP32C only)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibus_cyc_o</td>
<td>out</td>
<td>1</td>
<td>Cycle output</td>
</tr>
<tr>
<td>ibus_stb_o</td>
<td>out</td>
<td>1</td>
<td>Strobe output</td>
</tr>
<tr>
<td>ibus_cti_o</td>
<td>out</td>
<td>3</td>
<td>Cycle type identifier</td>
</tr>
<tr>
<td>ibus_bte_o</td>
<td>out</td>
<td>2</td>
<td>Burst type extension</td>
</tr>
<tr>
<td>ibus_ack_i</td>
<td>in</td>
<td>1</td>
<td>Acknowledge input</td>
</tr>
<tr>
<td>ibus_adr_o</td>
<td>out</td>
<td>30</td>
<td>Address output</td>
</tr>
<tr>
<td>ibus_dat_i</td>
<td>in</td>
<td>32</td>
<td>Data input</td>
</tr>
<tr>
<td><strong>Data bus</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbus_cyc_o</td>
<td>out</td>
<td>1</td>
<td>Cycle output</td>
</tr>
<tr>
<td>dbus_stb_o</td>
<td>out</td>
<td>1</td>
<td>Strobe output</td>
</tr>
<tr>
<td>dbus_we_o</td>
<td>out</td>
<td>1</td>
<td>Write enable output</td>
</tr>
<tr>
<td>dbus_sel_o</td>
<td>out</td>
<td>4</td>
<td>Select output</td>
</tr>
<tr>
<td>dbus_ack_i</td>
<td>in</td>
<td>1</td>
<td>Acknowledge input</td>
</tr>
<tr>
<td>dbus_adr_o</td>
<td>out</td>
<td>30</td>
<td>Address output</td>
</tr>
<tr>
<td>dbus_dat_o</td>
<td>out</td>
<td>32</td>
<td>Data output</td>
</tr>
<tr>
<td>dbus_dat_i</td>
<td>in</td>
<td>32</td>
<td>Data input</td>
</tr>
<tr>
<td><strong>Other ports</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irq_i</td>
<td>in</td>
<td>8</td>
<td>Interrupt requests</td>
</tr>
</tbody>
</table>
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 is designed with the assumption that read and write transactions do not cause side effects, thus it can be unsuitable for some slaves.

**DIVIDER_EN**

LXP32 includes a divider unit which has quite a low performance but occupies a considerable amount of resources. It can be disabled 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 $16 \times 16$ 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 $16 \times 16$ multipliers. One multiplication takes 6 clock cycles.
3.4. CLOCK AND RESET

- "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. The two least significant bits are ignored as instructions are always word-aligned.

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.

![Reset synchronization circuit](image-url)

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 \( \text{rst}_i \) port can be tied to a logical 0 in the RTL design to allow the synthesizer to remove redundant logic.

\( \text{clk}_i \) and \( \text{rst}_i \) 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 (LLI) is a simple pipelined synchronous protocol with a typical latency of 1 cycle used by LXP32U to fetch instructions. It was designed to allow simple connection of the CPU to on-chip program RAM or cache. The timing diagram of the LLI is shown on Figure 3.4.

![Figure 3.4: Low Latency Interface timing diagram (LXP32U)](image)

To request a word, the master produces its address on \( \text{lli}_\text{adr}_o \) and asserts \( \text{lli}_\text{re}_o \). The request is considered valid when \( \text{lli}_\text{re}_o \) is high and \( \text{lli}_\text{busy}_i \) is low on the same clock cycle. On the next cycle after a valid request, the slave must either produce data on \( \text{lli}_\text{dat}_i \) or assert \( \text{lli}_\text{busy}_i \) to indicate that data are not ready. \( \text{lli}_\text{busy}_i \) must be held high until the valid data are present on the \( \text{lli}_\text{dat}_i \) port.

The data provided by the slave are only required to be valid on the next cycle after a valid request (if \( \text{lli}_\text{busy}_i \) is not asserted) or on the cycle when \( \text{lli}_\text{busy}_i \) is deasserted after being held high. Otherwise \( \text{lli}_\text{dat}_i \) is undefined.

The values of \( \text{lli}_\text{re}_o \) and \( \text{lli}_\text{adr}_o \) are not guaranteed to be preserved by the master while the slave is busy.
3.6. WISHBONE INSTRUCTION BUS

The simplest slaves such as on-chip RAM blocks which are never busy 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 (you can even ignore lli_re_o in this case, although doing so can theoretically increase power consumption).

Since the lli_re_o output signal is not registered, this interface 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.

The instruction bus, whether LLI or WISHBONE, doesn’t support access to individual bytes and uses a 30-bit address port to address 32-bit words (instructions are always word-aligned). The lower two bits of the 32-bit address are ignored for the purpose of addressing. Consider the following example:

\[
\begin{align*}
\text{lc} & \quad r0, 0x10000000 \\
\text{jmp} & \quad r0 \\
// & \quad 0x04000000 \text{ will appear on lli_adr_o or ibus_adr_o}
\end{align*}
\]

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 instead. 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.
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).

The data 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. The upper 30 bits of the address appear on the dbus_adr_o port, while the lower two bits are decoded to create a 4-bit dbus_sel_o signal. Consider:

```
lc r0, 0x20000002
sb r0, 0x55
// write 0x55 to the address in r0
// 0x08000000 will appear on dbus_adr_o
// 0x4 will appear on dbus_sel_o
```

The byte-granular access feature is optional. If it is not needed, the dbus_sel_o port can be left unconnected. It is also possible to set the DBUS_RMW generic to true to enable byte-granular access emulation using the read-modify-write (RMW) cycle, which works even if the interconnect or slave doesn’t provide the [SEL_I()] port (Section 3.3).

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

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](image)

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:

- \texttt{lxl32\_ram256x32} – a dual-port synchronous 256 \times 32 bit RAM with one write port and one read port;
- \texttt{lxl32\_mul16x16} – an unsigned 16 \times 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.

LXP32 implements its own bypass logic dealing with situations when RAM read and write addresses collide. It does not depend on the read/write conflict resolution behavior of the underlying primitive.

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 \([\text{CTL\_I}]\) and \([\text{BTE\_I}]\).

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. \textit{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.

4. For small programs, consider mapping code and data memory to the beginning or end of the address space (i.e. 0x00000000–0x000FFFFF or 0xFFF00000–0xFFFFFFFF) to be able to load pointers with the \texttt{lcs} instruction which saves both memory and CPU cycles as compared to \texttt{lc}.

\textbf{Optimizing for timing}

1. Set up reasonable timing constraints. Do not overconstrain the design by more than 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 \texttt{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 \textit{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.

\textbf{Optimizing for area}

1. Consider disabling 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

Hardware architecture

The LXP32 CPU is based on a 3-stage hazard-free pipelined architecture and uses a large RAM-based register file (scratchpad) with two read ports and one write port. The pipeline includes the following stages:

- **Fetch** – fetches instructions from the program memory.
- **Decode** – decodes instructions and reads register operand values from the scratchpad.
- **Execute** – executes instructions and writes the results (if any) to the scratchpad.

LXP32 instructions are encoded in such a way that operand register numbers can be known without decoding the instruction (Section 2.2). When the Fetch stage produces an instruction, scratchpad input addresses are set immediately, before the instruction itself is decoded. If the instruction does not use one or both of the register operands, the corresponding data read from the scratchpad are discarded. Collision bypass logic in the scratchpad detects situations where the Decode stage tries to read a register which is currently being written by the Execute stage and forwards its value, bypassing the RAM block and avoiding Read After Write (RAW) pipeline hazards. Other types of data hazards are also impossible with this architecture.

As an example, consider the following simple code chunk:

```
mov r0, 10 // alias for add r0, 10, 0
mov r1, 20 // alias for add r1, 20, 0
add r2, r0, r1
```

Table 4.1 illustrates how this chunk is processed by the LXP32 pipeline. Note that on the fourth cycle the Decode stage requests the r1 register value.
while the *Execute* stage writes to the same register. Collision bypass logic in the scratchpad ensures that the *Decode* stage reads the correct (new) value of r1 without stalling the pipeline.

Table 4.1: Example of the LXP32 pipeline operation

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><em>add</em> r0, 10, 0</td>
<td><em>add</em> r0, 10, 0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><em>add</em> r1, 20, 0</td>
<td><em>add</em> r0, 10, 0</td>
<td>Pass 10 and 0 as operands</td>
</tr>
<tr>
<td></td>
<td>Request r10 (discarded)</td>
<td>Request r0 (discarded)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pass 10 and 0 as operands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><em>add</em> r2, r0, r1</td>
<td><em>add</em> r1, 20, 0</td>
<td>Perform the addition</td>
</tr>
<tr>
<td></td>
<td>Request r20 (discarded)</td>
<td>Request r0 (discarded)</td>
<td>Write 10 to r0</td>
</tr>
<tr>
<td></td>
<td>Pass 20 and 0 as operands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><em>add</em> r2, r0, r1</td>
<td>Request r0</td>
<td>Perform the addition</td>
</tr>
<tr>
<td></td>
<td>Request r0</td>
<td>Request r1 (bypass)</td>
<td>Write 20 to r1</td>
</tr>
<tr>
<td></td>
<td>Pass 10 and 20 as operands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>Perform the addition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write 30 to r2</td>
</tr>
</tbody>
</table>

When an instruction takes more than one cycle to execute, the *Execute* stage simply stalls the pipeline. Branch hazards are impossible in LXP32 as well since the pipeline is flushed whenever an execution transfer occurs.
Chapter 5
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.

5.1 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 5.1)\(^2\).

\(^1\)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 5.3).

- 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 6.1). A prebuilt executable for Microsoft® Windows® is already included in the LXP32 package, for other operating systems lxp32asm must be built from source (Section 6.4).

Figure 5.1: GTKWave displaying the LXP32 waveform dump produced by GHDL

5.2 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:
5.3. **RUNNING SIMULATION MANUALLY**

- **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 completion, GTKWave will be run automatically to display the dumped waveforms.
- **compile** – compile only, don’t run simulation.
- **clean** – delete all the produced artifacts.

### 5.3 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 common package (`verify/common_pkg`).

4. Compile the test platform (`verify/lxp32/src/platform` directory).

5. Compile the testbench itself (`verify/lxp32/src/tb` directory).

6. Simulate the `tb` design unit defined in the `tb.vhd` file.

### 5.4 Testbench parameters

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

- **CPU_DBUS_RMW** – DBUS_RMW CPU generic value (see Section 3.3).
- **CPU_MUL_ARCH** – MUL_ARCH CPU generic value (see Section 3.3).
• **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 6

Development tools

6.1 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 (or Entry) symbol at the beginning of the code.

Command line syntax

lxp32asm [ options | input files ]
CHAPTER 6. DEVELOPMENT TOOLS

General options

- `-c` – compile only (skip the Link stage).
- `-h`, `-help` – display a short help message and exit.
- `-o file` – output file name.
- `--` – do not interpret the subsequent command line arguments as options. Can be used if there are input file names starting with a dash.

Compiler options

- `-i dir` – add `dir` to the list of directories used to search for included files. Multiple directories can be specified with multiple `-i` arguments.

Linker options (ignored in compile-only mode)

- `-a align` – object alignment. Must be a power of 2 and can’t be less than 4. Default value is 4.
- `-b addr` – base address, that is, the address in memory where the executable image will be located. Must be a multiple of object alignment. Default value is 0.
- `-f fmt` – executable image format. See below for the list of supported formats.
- `-m file` – generate a map file. A map file is a human-readable list of all object and symbol addresses in the executable image.
- `-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, the image is not padded.

Output formats

Output formats that can be specified with the `-f` command line option are listed below.

- `bin` – raw binary image (little-endian). 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.

6.2 lxpdump – Disassembler

lxpdump 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 lxpasm.

Command line syntax

lxpdump [ options | input file ]

Supported options are:

• -b addr – executable image base address, only used for comments.
• -f fmt – input file format. All lxpasm output formats are supported. If this option is not supplied, autodetection is performed.
• -h, --help – display a short help message and exit.
• -na – do not use instruction aliases (such as mov, ret, not) and register aliases (such as sp, rp).
• -o file – output file name. By default, the standard output stream is used.
• -- – do not interpret subsequent command line arguments as options.

6.3 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. The interconnect can then be used to create a SoC based on LXP32.

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.
CHAPTER 6. DEVELOPMENT TOOLS

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).

6.4 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.
6.4. BUILDING FROM SOURCE

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. More details can be found in the CMake documentation.

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
Appendix A

Instruction set reference

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

A.1 List of instructions by group

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>Move alias for add dst, src, 0</td>
<td></td>
</tr>
<tr>
<td>lc</td>
<td>Load Constant</td>
<td>000001</td>
</tr>
<tr>
<td>lcs</td>
<td>Load Constant Short</td>
<td>101xxx</td>
</tr>
<tr>
<td>lw</td>
<td>Load Word</td>
<td>001000</td>
</tr>
<tr>
<td>lub</td>
<td>Load Unsigned Byte</td>
<td>001010</td>
</tr>
<tr>
<td>lsb</td>
<td>Load Signed Byte</td>
<td>001011</td>
</tr>
<tr>
<td>sw</td>
<td>Store Word</td>
<td>001100</td>
</tr>
<tr>
<td>sb</td>
<td>Store Byte</td>
<td>001110</td>
</tr>
</tbody>
</table>

Data transfer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add</td>
<td>010000</td>
</tr>
<tr>
<td>sub</td>
<td>Subtract</td>
<td>010001</td>
</tr>
<tr>
<td>neg</td>
<td>Negate alias for sub dst, 0, src</td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>Multiply</td>
<td>010010</td>
</tr>
<tr>
<td>divu</td>
<td>Divide Unsigned</td>
<td>010100</td>
</tr>
<tr>
<td>divs</td>
<td>Divide Signed</td>
<td>010101</td>
</tr>
<tr>
<td>modu</td>
<td>Modulo Unsigned</td>
<td>010110</td>
</tr>
<tr>
<td>mods</td>
<td>Modulo Signed</td>
<td>010111</td>
</tr>
</tbody>
</table>

Arithmetic operations

Bitwise operations
### APPENDIX A. INSTRUCTION SET REFERENCE

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>Bitwise Not alias for xor dst, src, -1</td>
<td>011000</td>
</tr>
<tr>
<td>and</td>
<td>Bitwise And</td>
<td>011001</td>
</tr>
<tr>
<td>or</td>
<td>Bitwise Or</td>
<td>011010</td>
</tr>
<tr>
<td>xor</td>
<td>Bitwise Exclusive Or</td>
<td>011010</td>
</tr>
<tr>
<td>sl</td>
<td>Shift Left</td>
<td>011100</td>
</tr>
<tr>
<td>sru</td>
<td>Shift Right Unsigned</td>
<td>011110</td>
</tr>
<tr>
<td>srs</td>
<td>Shift Right Signed</td>
<td>011111</td>
</tr>
</tbody>
</table>

#### Execution transfer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Jump</td>
<td>1000000</td>
</tr>
<tr>
<td>cjmpxxxx</td>
<td>Compare and Jump (xxxx = condition)</td>
<td>11xxxx</td>
</tr>
<tr>
<td>call</td>
<td>Call Procedure</td>
<td>010001</td>
</tr>
<tr>
<td>ret</td>
<td>Return from Procedure alias for jmp rp</td>
<td>000001</td>
</tr>
<tr>
<td>iret</td>
<td>Interrupt Return alias for jmp irp</td>
<td>000000</td>
</tr>
</tbody>
</table>

#### Miscellaneous instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>No Operation</td>
<td>000000</td>
</tr>
<tr>
<td>hlt</td>
<td>Halt</td>
<td>000010</td>
</tr>
</tbody>
</table>

### A.2 Alphabetical list of instructions

**add** – Add

**Syntax**

\[
\text{add \ DST, RD1, RD2}
\]

**Encoding**

\[
010000 \ T1 \ T2 \ DST \ RD1 \ RD2
\]

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

**Operation**

\[
\text{DST} := \text{RD1} + \text{RD2}
\]

**and** – Bitwise And

**Syntax**

\[
\text{and \ DST, RD1, RD2}
\]
A.2. ALPHABETICAL LIST OF INSTRUCTIONS

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.

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

Syntax
cjme DST, RD1, RD2 (Equal)
cjmpne DST, RD1, RD2 (Not Equal)
cjmsg DST, RD1, RD2 (Signed Greater)
APPENDIX A. INSTRUCTION SET REFERENCE

cjumpsge DST, RD1, RD2 (Signed Greater or Equal)
cjmpls DST, RD1, RD2 (Signed Less)
cjmlsle 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
cjmsg  110001
cjmpsge 111001
cjmpug  110010
cjmpuge 111010
cjmpls, cjmlsle, cjmpul, cjmpule  instructions are aliases for cjmsg, 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, cjmpxxx does not write to DST.

divs – Divide Signed

Syntax

divs DST, RD1, RD2
A.2. **ALPHABETICAL LIST OF INSTRUCTIONS**

**Encoding**

010101 T1 T2 DST RD1 RD2

Example: **divs** r2, r1, -3 \(\rightarrow \) 0x560201FD

**Operation**

\(DST := (\text{signed}) \ RD1 / (\text{signed}) \ RD2\)

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 \(\rightarrow \) 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**

Halt the CPU until an enabled interrupt is received.

**Syntax**

**hlt**

**Encoding**

000010 0 0 0000000 0000000 0000000

**Operation**

Pause execution until an interrupt is received.
**jmp – Jump**
Transfer execution to the address pointed by the operand.

**Syntax**
\[ \text{jmp } RD1 \]

**Encoding**
\[ \begin{align*}
100000 & 1 0 00000000 & RD1 & 00000000 \\
\end{align*} \]
RD1 must be a register.
Example: \[ \text{jmp } r1 \rightarrow 0x82000100 \]

**Operation**
\[ \text{goto } RD1 \]
Pointer in RD1 is interpreted as described in Section 2.4.

**iret – Interrupt Return**
Return from an interrupt handler.

**Syntax**
\[ \text{iret} \]

Alias for \[ \text{jmp } \text{irp} \].

**lc – Load Constant**
Load a 32-bit word to the specified register. Note that values from the \([-1048576; 1048575]\] range can be loaded more efficiently using the \text{lcs} instruction.

**Syntax**
\[ \text{lc } DST, \text{WORD32} \]
A.2. ALPHABETICAL LIST OF INSTRUCTIONS

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

lc – Load Constant Short
Load a signed value from the [-1048576; 1048575] range (a sign extended 21-bit value) to the specified register. Unlike the lc instruction, this instruction is encoded as a single word.

Syntax
lc DST, VAL

Encoding
101 VAL[20:16] DST VAL[15:0]
Example: lc r1, -1000000 → 0xB001BDC0

Operation
DST := (signed) VAL

lsb – Load Signed Byte
Load a byte from the specified address to the register, performing sign extension.

Syntax
lsb DST, RD1
Encodings

**lsb**

RD1 must be a register.

Example: **lsb r2, r1 → 0x2E020100**

**Operation**

\[ \text{DST} := (\text{signed}) \left( *(\text{BYTE})*\text{RD1} \right) \]

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**

\[ \text{DST} := *(\text{BYTE})*\text{RD1} \]

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
A.2. Alphabetical List of Instructions

Encoding

001000 1 0 DST RD1 0000000

RD1 must be a register.

Example: \texttt{lw} r2, r1 \rightarrow 0x22020100

Operation

\(\text{DST} := \ast \text{RD1}\)

Pointer in RD1 is interpreted as described in Section 2.4.

\textbf{mods} – Modulo Signed

Syntax

\texttt{mods} DST, RD1, RD2

Encoding

010111 T1 T2 DST RD1 RD2

Example: \texttt{mods} r2, r1, 10 \rightarrow 0x5E02010A

Operation

\(\text{DST} := (\text{signed}) \text{RD1 mod (signed)} \text{RD2}\)

Modulo operation satisfies the following condition: if \(Q = A/B\) and \(R = A \mod B\), 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.

\textbf{modu} – Modulo Unsigned

Syntax

\texttt{modu} DST, RD1, RD2

Encoding

010110 T1 T2 DST RD1 RD2

Example: \texttt{modu} r2, r1, 10 \rightarrow 0x5A02010A
Operation
DST := RD1 mod RD2

Modulo operation satisfies the following condition: if $Q = A/B$ and $R = A \mod B$, 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.

**neg – Negate**

**Syntax**

`neg DST, RD2`

Alias for `sub DST, 0, RD2`
A.2. ALPHABETICAL LIST OF INSTRUCTIONS

nop – No Operation
Syntax
\texttt{nop}

Encoding
000000 0 0 0000000 0000000 0000000

Operation
This instruction does not alter the machine state.

not – Bitwise Not
Syntax
\texttt{not DST, RD1}

Alias for \texttt{xor DST, RD1, -1}.

or – Bitwise Or
Syntax
\texttt{or DST, RD1, RD2}

Encoding
011001 T1 T2 DST RD1 RD2

Example: \texttt{or r2, r1, 0x3F \rightarrow 0x6602013F}

Operation
\texttt{DST := RD1 \lor RD2}

ret – Return from Procedure
Return from a procedure.

Syntax
\texttt{ret}

Alias for \texttt{jmp rp}.
**sb – Store Byte**

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

**Syntax**

\[ \text{sb } \text{RD1}, \text{RD2} \]

**Encoding**

\[ 001110 \ 1 \ T2 \ 00000000 \ \text{RD1} \ \text{RD2} \]

RD1 must be a register.
Example: \[ \text{sb } r2, r1 \rightarrow 0x3B000201 \]

**Operation**

\[ *(\text{BYTE}*)\text{RD1} := \text{RD2} \land 0x000000FF \]

Pointer in RD1 is interpreted as described in Section 2.4.

**sl – Shift Left**

**Syntax**

\[ \text{sl } \text{DST}, \text{RD1}, \text{RD2} \]

**Encoding**

\[ 011100 \ T1 \ T2 \ \text{DST} \ \text{RD1} \ \text{RD2} \]

Example: \[ \text{sl } r2, r1, 5 \rightarrow 0x72020105 \]

**Operation**

\[ \text{DST} := \text{RD1} \ll \text{RD2} \]

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

**srs – Shift Right Signed**

**Syntax**

\[ \text{srs } \text{DST}, \text{RD1}, \text{RD2} \]
A.2. ALPHABETICAL LIST OF INSTRUCTIONS

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 → 0x3300201

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, based on an assumption that no pipeline stalls are caused by the instruction bus latency or cache misses. These data are provided for reference purposes; the software should not depend on them as they can change in future hardware revisions.

Table B.1: Instruction cycle counts

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>modu</td>
<td>37</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>mov</td>
<td>1</td>
</tr>
<tr>
<td>call</td>
<td>4</td>
<td>mul</td>
<td>2, 6 or 34$^3$</td>
</tr>
<tr>
<td>cjmpxxx</td>
<td>5 or 2$^1$</td>
<td>neg</td>
<td>1</td>
</tr>
<tr>
<td>divs</td>
<td>36</td>
<td>nop</td>
<td>1</td>
</tr>
<tr>
<td>divu</td>
<td>36</td>
<td>not</td>
<td>1</td>
</tr>
<tr>
<td>hlt</td>
<td>N/A</td>
<td>or</td>
<td>1</td>
</tr>
<tr>
<td>jmp</td>
<td>4</td>
<td>ret</td>
<td>4</td>
</tr>
<tr>
<td>iret</td>
<td>4</td>
<td>sb</td>
<td>$\geq 2^2$</td>
</tr>
<tr>
<td>lc</td>
<td>2</td>
<td>sl</td>
<td>2</td>
</tr>
<tr>
<td>lcs</td>
<td>1</td>
<td>srs</td>
<td>2</td>
</tr>
<tr>
<td>lsb</td>
<td>$\geq 3^2$</td>
<td>sru</td>
<td>2</td>
</tr>
<tr>
<td>lub</td>
<td>$\geq 3^2$</td>
<td>sub</td>
<td>1</td>
</tr>
<tr>
<td>lw</td>
<td>$\geq 3^2$</td>
<td>sw</td>
<td>$\geq 2^2$</td>
</tr>
<tr>
<td>mods</td>
<td>37</td>
<td>xor</td>
<td>1</td>
</tr>
</tbody>
</table>

$^1$Depends on whether the jump is taken or not.
$^2$Depends on the data bus latency.
$^3$Depends on the multiplier architecture. 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:

```c
/*
   * 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). Note that strings...
are not null-terminated in the LXP32 assembly language; when required, terminating null character must be inserted explicitly.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ \</td>
<td>Backslash character</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quotation mark</td>
</tr>
<tr>
<td>'</td>
<td>Single quotation mark (can be also used directly)</td>
</tr>
<tr>
<td>\t</td>
<td>Tabulation character</td>
</tr>
<tr>
<td>\n</td>
<td>Line feed</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\xXX</td>
<td>Character with a hexadecimal code of XX (1–2 digits)</td>
</tr>
<tr>
<td>\XXX</td>
<td>Character with an octal code of XXX (1–3 digits)</td>
</tr>
</tbody>
</table>

C.3 Symbols

Symbols (labels) are used to refer to data or code locations. LXP32 assembly language does not have distinct code and data labels: 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.

Symbols can be used as operands to the \texttt{lcs} and \texttt{lcs} instruction statements. A symbol reference can end with a \texttt{@n} sequence, where \texttt{n} is a numeric literal; in this case it is interpreted as an offset (in bytes) relative to the symbol definition. For the \texttt{lcs} instruction, the resulting address must still fit into the sign extended 21-bit value range (0x00000000–0x000FFFFF or 0xFFF00000–0xFFFFFFFF), otherwise the linker will report an error.

By default all symbols are local, that is, they can be only referenced from the module where they were defined. To make a symbol accessible from other modules, use the \texttt{#export} directive. To reference a symbol defined in another module use the \texttt{#import} directive.

A symbol named \texttt{entry} or \texttt{Entry} has a special meaning: it is used to inform the linker about the program entry point if there are multiple input
C.4. STATEMENTS

files. It does not have to be exported. If defined, this symbol must precede the first instruction or data definition statement in the module. Only one module in the program can define the entry symbol.

\[
\begin{align*}
\text{lcl} & \ r10, \ \text{jump\_label} \\
\text{lcl} & \ r11, \ \text{data\_word} \\
\text{// ...} \\
\text{sw} & \ r11, \ r0 \ /\!\!/ \text{store the value of } r0 \text{ to the} \\
& \quad \text{// location pointed by data\_word} \\
\text{jmp} & \ r10 \quad \text{// transfer execution to jump\_label} \\
\text{// ...} \\
\text{jump\_label:} \\
& \quad \text{mov} \ r1, \ r0 \\
\text{// ...} \\
\text{data\_word:} \\
& \quad .\text{word} \ 0x12345678
\end{align*}
\]

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.

\[
\text{\#define \ identifier \ [ \ token \ ... \ ]}
\]

Defines a macro that will be substituted with zero 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).
APPENDIX C. LXP32 ASSEMBLY LANGUAGE

#error [ msg ]

 Raises a compiler error. If msg is supplied, uses it as an error message.

#export identifier

 Declares identifier as an exported symbol. Exported symbols can be referenced by other modules.

#ifdef | #ifndef identifier

 ... 

#else 

 ... 

#endif

 Define C preprocessor-style conditional sections which are processed or not based on whether a certain macro has been defined. #else is optional. Can be nested.

#import identifier

 Declares identifier as an imported symbol. Used to refer to symbols exported by 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. alignment must be a power of 2 and can’t be less than 4. 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).
The `.align` statement is not guaranteed to work if the requested alignment is greater than the section alignment specified for the linker (see Subsection 6.1).

`.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.

To define a null-terminated string, the terminating null character must be inserted explicitly.

`.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)

<table>
<thead>
<tr>
<th>General information</th>
</tr>
</thead>
<tbody>
<tr>
<td>WISHBONE revision</td>
</tr>
<tr>
<td>Type of interface</td>
</tr>
<tr>
<td>Supported cycles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal names</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk_i</td>
</tr>
<tr>
<td>rst_i</td>
</tr>
<tr>
<td>ibus_cyc_o</td>
</tr>
<tr>
<td>ibus_stb_o</td>
</tr>
<tr>
<td>ibus_cti_o</td>
</tr>
<tr>
<td>ibus_bte_o</td>
</tr>
<tr>
<td>ibus_ack_i</td>
</tr>
<tr>
<td>ibus_adr_o</td>
</tr>
<tr>
<td>ibus_dat_i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Supported tag signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibus_cti_o</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ibus_bte_o</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port size</td>
</tr>
</tbody>
</table>
### D.2 Data bus

<table>
<thead>
<tr>
<th>General information</th>
</tr>
</thead>
<tbody>
<tr>
<td>WISHBONE revision</td>
</tr>
<tr>
<td>Type of interface</td>
</tr>
<tr>
<td>Supported cycles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal names</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk_i</td>
</tr>
<tr>
<td>rst_i</td>
</tr>
<tr>
<td>dbus_cyc_o</td>
</tr>
<tr>
<td>dbus_stb_o</td>
</tr>
<tr>
<td>dbus_we_o</td>
</tr>
<tr>
<td>dbus_sel_o</td>
</tr>
<tr>
<td>dbus_ack_i</td>
</tr>
<tr>
<td>dbus_adr_o</td>
</tr>
<tr>
<td>dbus_dat_o</td>
</tr>
<tr>
<td>dbus_dat_i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port size</td>
</tr>
<tr>
<td>Port granularity</td>
</tr>
<tr>
<td>Maximum operand size</td>
</tr>
<tr>
<td>Data transfer ordering</td>
</tr>
<tr>
<td>Data transfer sequence</td>
</tr>
</tbody>
</table>
Appendix E

List of changes

Version 1.3 (2022-08-28)

This release removes support for temporarily blocked interrupts (interrupts can still be disabled) and introduces wake-up interrupts.

Version 1.2 (2021-10-21)

This release introduces a few non-breaking changes to the software and testbench. The CPU RTL description hasn’t been changed from the previous release.

- `lxp32asm` now supports C-style conditional processing directives: `#ifdef`, `#ifndef`, `#else` and `#endif`.
- `#define` directive can now declare a macro with zero substitute tokens.
- A new `#error` directive.
- Minor changes to the testbench.

Version 1.1 (2019-01-11)

This release introduces a minor but technically breaking hardware change: the `START_ADDR` generic, which used to be 30-bit, has been for convenience extended to a full 32-bit word; the two least significant bits are ignored.

The other breaking change affects the assembly language syntax. Previously all symbols used to be public, and multiple modules could not define
symbols with the same name. As of now only symbols explicitly exported using the \texttt{#export} directive are public. \texttt{#extern} directive has been replaced by \texttt{#import}.

Other notable changes include:

- A new instruction, \texttt{lcs (Load Constant Short)}, has been added, which loads a 21-bit sign extended constant to a register. Unlike \texttt{lc}, it is encoded as a single word and takes one cycle to execute.

- Optimizations in the divider unit. Division instructions (\texttt{divs} and \texttt{divu}) now take one fewer cycle to execute (modulo instructions are unaffected).

- LXP32 assembly language now supports a new instruction alias, \texttt{neg (Negate)}, which is equivalent to \texttt{sub dst, 0, src}.

\textbf{Version 1.0 (2016-02-20)}

Initial public release.