Nux User Guide

Electronic Vision(s): Simon Friedmann, Christian Pehle

March 11, 2020

Contents

1 Introduction **4**
1.1 Scope of this document . . . 4
1.2 Related material . . . 4

2 Instantiating Nux **5**
2.1 Design parameters . . . 5
2.1.1 Top-level parameters . . . 5
2.1.2 Internal parameters . . . 7
2.2 Interfaces and ports . . . 7
2.3 Example: core environment . . . 9

3 Running software on Nux **11**
3.1 Building binutils and the cross-compiler . . . 11
3.2 Compiling programs . . . 11
3.2.1 C runtime . . . 12
3.2.2 Linker script . . . 13
3.2.3 Minimum user code . . . 14
3.2.4 Generating the binary image . . . 14
3.2.5 Useful commands . . . 16
3.3 Loading and executing programs . . . 16

4 Running simulations **17**
4.1 Build flow . . . 17
4.1.1 Configuration options . . . 17
4.1.2 Running simulations . . . 18
4.2 Tests . . . 18
4.2.1 Program test . . . 18
4.2.2 Sequence test . . . 18
4.2.3 Vector unit test . . . 19

5 Instruction set **20**
5.1 Implemented subset of Power ISA 2.06 . . . 22
5.2 FXV vector instruction set . . . 22
5.2.1 Registers . . . 22
5.2.2 Primitive functions . . . 23
5.2.3 Modulo halfword instructions . . . 24
5.2.4 Modulo byte instructions . . . 27
5.2.5 Saturating, fractional, halfword instructions . . . 29

5.2.6 Saturating, fractional, byte instructions 31
5.2.7 Permute instructions 33
5.2.8 Load/store instructions 35
5.3 Deprecated NEVER and SYNAPSE instruction sets 36

1 Introduction

1.1 Scope of this document

This user guide intends to give an overview of the Nux's design and how to use it in hardware designs and for executing software on it. **It is not (yet) a full specification.** It should serve as a starting point for everyone wanting to contribute to the design and provide the necessary knowledge to use it in other hardware designs.

1.2 Related material

Most of the internals are documented in detail in Friedmann (2013). Some overview of the vector Single Instruction Multiple Data (SIMD) unit and the use for plasticity is given in Friedmann and Schemmel (2015).

2 Instantiating Nux

2.1 Design parameters

The top-level module `Pu_v2` offers a large number of parameters that control various aspects of the design. Additionally some important parameters are hidden deeper in the code. This section lists and describes those parameters.

2.1.1 Top-level parameters

First, the top-level design parameters (see also file `rtl/processor/pu.sv`):

OPT_BCACHE type: `int`, default: 0

Configures the branch predictor (branch cache). A value of 0 disables branch prediction completely. Higher values control the number of entries in the branch cache. The branch cache contains $2^{\texttt{OPT_BCACHE}}$ entries.

OPT_MULTIPLIER type: `bit`, default: 1

Set to one to include a fixed-point multiplier in the design.

OPT_DIVIDER type: `bit`, default: 1

Set to one to include a fixed-point divider in the design.

OPT_IOBUS type: `bit`, default: 1

Include an OMNIBUS (OMNIBUS) interface for use by the external control facility (see PowerISA (2010)). The interface is called `iobus`.

OPT_VECTOR type: `bit`, default: 1

Set to one to include the Fixed-point Vector Extension (FXV) SIMD extension. Enables also the `vector_bus` interface for serial load/store accesses (`fxvlax`, `fxvstax`) and `vector_pbus` for parallel load/store accesses (`fxvinx`, `fxvoutx`).

OPT_VECTOR_SLICES type: `int`, default: 8

Control the number of parallel datapath blocks called slices in the FXV unit.

OPT_VECTOR_NUM_HALFWORDS type: `int`, default: 8

Control the width of the datapath of one FXV slice in units of halfwords (16 bit). With the default configuration the unit operates on 8×16 bit vectors.

OPT_VECTOR_MULT_DELAY type: `int`, default: 4

Configure the number of pipeline stages used for multiplication in the FXV slices.

OPT_VECTOR_ADD_DELAY type: `int`, default: 1

Configure the number of pipeline stages used for addition in the FXV slices.

OPT_VECTOR_INST_QUEUE_DEPTH type: `int`, default: 4

Configure the depth of the First In First Out (FIFO) for instructions from the general-purpose part to the FXV unit.

OPT_NEVER type: `bit`, default: 0

Deprecated

Enables the Nibble Vector Extension (NEVER) unit for accelerated plasticity processing in High Input Count Analog Neural Network (HICANN).

OPT_SYNAPSE type: `bit`, default: 0

Deprecated

Enables the Synapse processing extension (SYNAPSE) unit for accelerated plasticity processing in HICANN. The SYNAPSE unit uses the `syn_io_a` and `syn_io_b` interfaces.

OPT_DMEM type: `Pu_types::Opt_mem`, default: `Pu_types::MEM_BUS`

Select whether to use the tight (`Pu_types::MEM_TIGHT`) or bus (`Pu_types::MEM_BUS`) interface to connect the data memory. The tight memory variant uses the `dmem` interface, while the bus variant uses `dmem_bus`. The bus interface is a OMNIBUS interface that allows for variable delay of requests with multiple requests in flight simultaneously. The tight interface is suitable for direct connection to a memory block.

OPT_IF_LATENCY type: `int`, default: 1

Control the number of pipeline stages between instruction memory and the rest of the frontend. Can be useful, when the memory is physically far away but impacts branch penalty.

OPT_BCACHE_IGNORES_JUMPS type: `bit`, default: 1

Optimization parameter to remove a long timing path. The branch predictor does not predict for instruction locations that are targets of branches.

OPT_BUFFER_BCTRL type: `bit`, default: 0

If set, adds a buffering register stage between the branch functional unit and the address generation unit in the instruction fetcher. This buffer increases branch penalty by one cycle, but removes a long timing arc through the functional unit.

OPT_WRITE_THROUGH type: `bit`, default: 1

Implement a bypass for writes to general-purpose registers, i.e. simultaneous reads directly use the result from the functional unit. Decreases length of pipeline bubbles at the cost of timing.

OPT_LOOKUP_CACHE type: `bit`, default: 1

Instruction tracking uses the lookup cache mechanism described in Friedmann (2013). Required if `OPT_DMEM` = `Pu_types::MEM_BUS` or any other variable latency instruction is implemented.

2.1.2 Internal parameters

Now, internal parameters in `rtl/packages/frontend_pkg.sv`. Generally, these require more thinking when changing then top-level options. For example, the multiplier latency depends on the actual multiplier implementation. For Field Programmable Gate Arrays (FPGAs) this might be a fixed generated core. In that case, changing the number here will affect only the scheduling logic but not the actual multiplier latency. So change these parameters only if you know what you are doing.

mul_latency type: `int`, default: 4

Latency of the fixed-point multiplier in cycles. What values are possible depends on the selected implementation (e.g. DesignWare block or Xilinx generated core).

div_latency type: `int`, default: 31

Latency of fixed-point divide in cycles.

ls_latency type: `int`, default: 2

Latency of load/store operations when using `OPT_MEM` = `Pu_types::MEM_TIGHT`. Possible values are 2 and 3. In the latter case an additional pipeline stage is added to improve timing.

ls_bus_latency type: `int`, default: 3

Expected latency of variable latency load/store operations when using `OPT_MEM` = `Pu_types::MEM_BUS`. A wrtie-back slot is scheduled at the indicated time in the result shift register. If it is not used, the variable latency write-back takes over.

2.2 Interfaces and ports

The top-level module contains the following ports and interfaces:

clk type: `logic`

Clock signal

reset type: `logic`

Active-high Reset signal.

hold type: `logic`

Deprecated

Active-high signal to stall instruction fetch. Should be tied low, or you have to check if it still works in all cases.

imem type: `Ram_if`

Interface to either instruction cache or directly to memory. Honors the delay signal in the interface.

dmem type: `Ram_if`

Interface to data memory when `OPT_MEM` = `Pu_types::MEM_TIGHT`. Otherwise, connect dummy interface instance.

dmem_bus type: `Bus_if`

OMNIBUS interface to data memory when `OPT_MEM` = `Pu_types::MEM_BUS`. Otherwise, connect dummy interface instance.

iobus type: `Bus_if`

OMNIBUS interface for the device control facility. Only used when `OPT_IOBUS` is set. The device control facility provides a separate address space for Input/Output (IO).

vector_bus type: `Bus_if`

Bus for the serial load/store unit of the FXV unit. Only used when `OPT_VECTOR` is set.

vector_pbus type: `Bus_if`

Bus for the parallel load/store unit of the FXV unit. Only used when `OPT_VECTOR` is set.

gout type: `logic[31:0]`

Register mapped output for the processor. Is set by writing an Special Purpose Register (SPR). Typically used for digital chip pins.

gin type: `logic[31:0]`

Register mapped input for the processor. Is read by reading an SPR. Typically used for digital chip pins.

goe type: `logic[31:0]`

Register mapped output enable for the processor. Is set by writing an SPR. Typically used for digital chip pins.

`ctrl` type: `Pu_ctrl_if`

Control interface for interrupt handling, sleeping, and monitoring of program execution.

`timer` type: `Timer_if`

Access to timer facilities. The timer is outside of the processor core, because it is active in sleep states during which the core's clock might be turned off. The timer facility also provides interrupts to the core that can trigger wake-up from the sleep state.

`syn_io_a, syn_io_b` type: `Syn_io_if`

Deprecated

IO interfaces for SYNAPSE unit.

2.3 Example: core environment

Figure 2.1 shows an example of a simple core environment using a shared main memory.

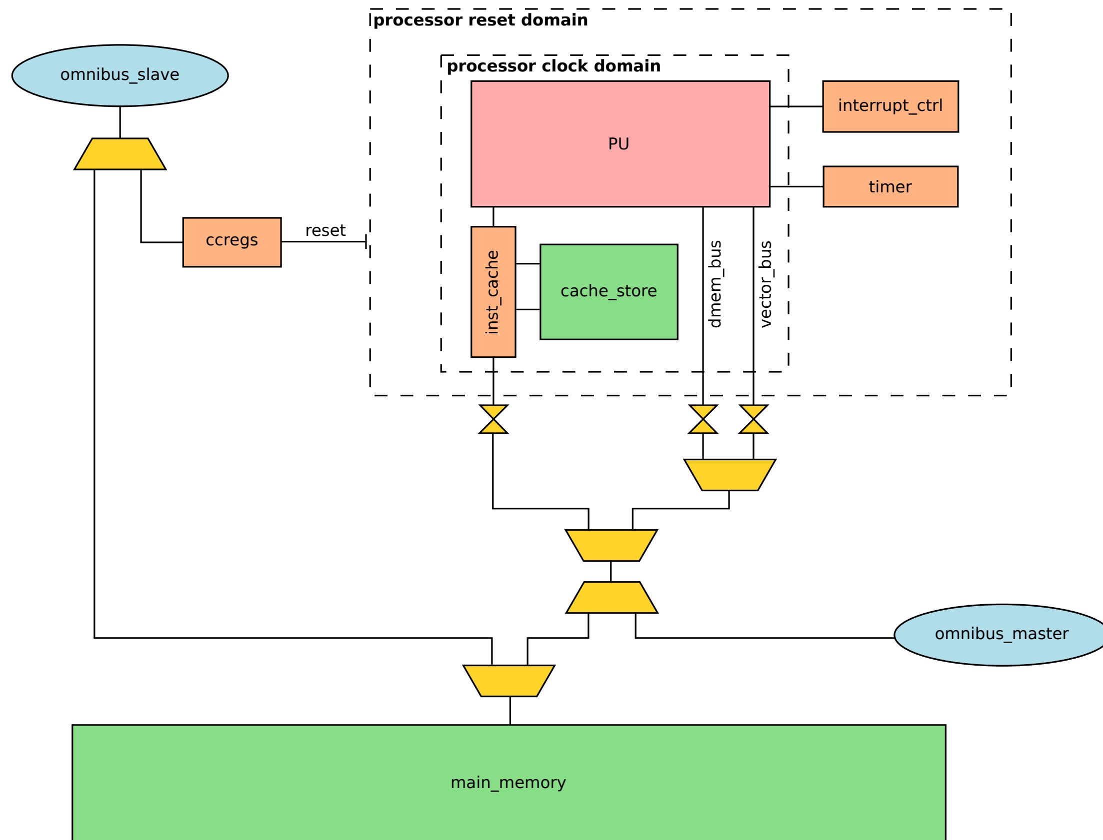


Figure 2.1: Example of environment for Nux as PPU. The top-level module `Pu_v2` is shown as the red block labeled PU. The shown configuration uses a shared main memory with instruction cache. Data memory, instruction cache, and FXV serial load/store all share access to the single main memory block. The yellow symbols represent OMNIBUS modules. A slave port allows access from the outside to change control registers (reset, clock gating) and to main memory (program loading, result retrieval). A master port allows access from within the PPU to an external bus. The `interrupt_ctrl` and `timer` units implement interrupts and timing. They exist outside the clock domain controlled by control registers, but within the reset domain.

3 Running software on Nux

Executing a program on Nux in hardware or simulation requires three steps outlined in the next sections. First, you have to install the cross-compilation toolchain for the PowerISA. The compilation and linking of the programs for the Nux is described next. For the actual execution implementation specific software is required to interface with Nux.

3.1 Building binutils and the cross-compiler

Any cross-compiler will do that emits code for the target "powerpc-eabi", i.e. Power instruction set architecture using the Embedded Application Binary Interface (EABI) (IBM, 1998). There exists however a gcc and binutils version with basic support for the vector instructions at `https://github.com/electronicvisions/gcc` and `https://github.com/electronicvisions/binutils-gdb`. An installation script is provided in the Nux repository to download and compile the patched gcc-toolchain for the processor from source (`support/gcc/install.sh`). You may want to change the `PREFIX` variable at the top to select the target installation directory. The gcc with vector instruction support is currently at the version 4.9 and therefore depends on outdated libraries. The dependencies downloaded in the install script (e.g. mpfr, gmp, mpc, ...) should be preferred over the most recent versions in order to avoid failing builds due to version incompatibilities. Refer to GCC installation for further dependencies and required minimum versions.

The goal of the installation script is to build a minimal gcc and binutils with support for C but without libc. Support for C++ is added to the compiler while still being experimental.

3.2 Compiling programs

The processor starts execution after reset is released at address 0. Also, exceptions trigger a control transfer to fixed locations in the range from address 4 to 44. The final result of compilation is therefore to generate a binary image that can be written to memory with valid code at the correct addresses. This is achieved by a special linker script `elf32nux.x` and the C runtime `crt.s`. An example linker script and C runtime can be found in the libnux at `https://github.com/electronicvisions/libnux`.

3.2.1 C runtime

The C runtime is a wrapper for the compiled code with various tasks

- initialize the stack pointer
- declare interrupt routines
- start the execution of the compiled program at the correct symbol
- resume into a well defined state at the end of the program

An example wrapper is discussed below:

```
1  # crt.s -- part a
2  .extern start
3  .extern reset
4  .extern _isr_undefined
5  .extern isr_einput
6  .extern isr_alignemnt
7  .extern isr_program
8  .extern isr_doorbell
9  .extern isr_fit
10 .extern isr_dec
11
12 .extern stack_ptr_init
```

This declares symbols that can be overwritten by other parts of the program. The `start` symbol declared on line 2 is the entry point for the C code. It is defined by declaring a function `void start()` in your C code. The `isr_*` symbols on lines 5 to 10 represent interrupt service routines. The `_isr_undefined` symbol is a default handler that is used if no service routine is defined.

```
1  # crt.s -- part b
2  .text
3  .extern _start:
4  reset:
5    b __init
6
7    # interrupt jump table
8    int_mcheck:    b _isr_undefined
9    int_cinput:    b _isr_undefined
10   int_dstorage:  b _isr_undefined
11   int_istorage:  b _isr_undefined
12   int_einput:    b isr_einput
13   int_alignment: b isr_alignment
14   int_program:   b isr_program
15   int_syscall:   b _isr_undefined
16   int_doorbell:  b isr_doorbell
17   int_cdoorbell: b _isr_undefined
18   int_fit:       b isr_fit
19   int_dec:       b isr_dec
```

This block is the start of the code section of the binary. It begins on line 5 with a branch to the `__init` symbol defined below. The linker script ensures that this instruction resides at address 0 of the produced binary image. After that follows the interrupt jump table on lines 8 through 19. It consists of branches to interrupt service routines, which are located at the appropriate addresses defined by the hardware implementation.

```
1 # crt.s -- part c
2 __init:
3   # set the stack pointer
4   lis 1, stack_ptr_init@h
5   addi 1, 1, stack_ptr_init@l
6   # start actual program
7   bl start
8
9 end_loop:
10   wait
11   b end_loop
```

This fragment initializes the stack pointer in general-purpose register 1 as defined by the EABI (IBM, 1998). The symbol `stack_ptr_init` is defined by the linker script to match the size of the implemented memory. After initialization on line 7, the wrapper calls the user code at the `start` symbol. If user code returns from the start() function, the loop at the end sends the processor to sleep. In case of wake-up events, the appropriate service routine will be taken through the interrupt jump table. On return from the service routine, the branch on line 11 ensures a return to the sleep state.

3.2.2 Linker script

The linker script `elf32nux.x` is passed to the linker to configure how the resulting binary is generated. An example is discussed below:

```
1 MEMORY {
2   ram(rwx) : ORIGIN = 0, LENGTH = 16K
3 }
```

This specifies the memory layout of the implementation. In the given case we have one memory region with a size of 16 kib starting at address 0. In a tight memory configuration (`OPT_MEM` = `Pu_types`::`MEM_TIGHT`) there would be two locations here for data and code.

```
1 mailbox_size = 4096;
2 mailbox_end = 0x4000;
3 mailbox_base = mailbox_end - mailbox_size;
4 stack_ptr_init = mailbox_base - 8;
```

The intention here is to create a reserved memory region at the end called mailbox. This is used for communication with the environment by software running on Nux. The stack pointer is initialized to start at lower addresses than the mailbox region. Note, that

the `crt.s` wrapper uses the `stack_ptr_init` symbol to do the actual initialization of register 1.

```
1 SECTIONS {
2   .text : {
3         _isr_undefined = .;
4 
5         *crt.o(.text)
6         *(.text)
7 
8         PROVIDE(isr_einput = _isr_undefined);
9         PROVIDE(isr_alignment = _isr_undefined);
10        PROVIDE(isr_program = _isr_undefined);
11        PROVIDE(isr_doorbell = _isr_undefined);
12        PROVIDE(isr_fit = _isr_undefined);
13        PROVIDE(isr_dec = _isr_undefined);
14  } > ram
15 
16  .data : {
17    *(.data)
18    *(.rodata)
19  } > ram
20 
21  .bss : {
22    *(.bss)
23    *(.sbss)
24  } > ram
25 
26  /DISCARD/ : {
27    *(.eh_frame)
28  }
29 }
```

This part specifies, where generated code sections should be mapped. We use only the three sections `text` for instructions, `data` for data, and `bss` for zeroed data. Line 5 ensures, that the wrapper is positioned at memory location 0, so that reset and interrupt handling work correctly. Line 3 defines the default interrupt handler `_isr_undefined` to be equivalent to the reset vector at address 0. Lines 8 to 13 connect the default handler to all interrupt service routines that were not defined in user code.

3.2.3 Minimum user code

Minimal C user code consists of just the start() function:

```
1     void start() {
2     }
```

3.2.4 Generating the binary image

Generating the final binary image involves three steps:

1. Compile source files to object files.
2. Link object files to executable.
3. Extract binary image of `.text` and `.data` sections out of the executable.

The following examples assume an architecture with single main memory as shown in Figure 2.1. When using two memories, two images have to be extracted at the end, which requires a different `elf32nux.x` file as the one shown above. The examples also assume, that gcc binaries are visible through the `PATH` variable and that it was installed with prefix `powerpc-eabi`.

Compiling

The assembly wrapper has to be assembled:

```
1     $ powerpc-eabi-as -mnux crt.s -o crt.o
```

And C-source compiled:

```
1     $ powerpc-eabi-gcc -c <sourcefile> -o <objectfile> \
2         -mcpu=nux        \
3         -ffreestanding   \
4         -msdata=none     \
5         -mstrict-align   \
6         -msoft-float     \
7         -mno-relocatable
```

The exact meaning of options is given in Stallman (2015). The option on line 3 tells the compiler to use a "freestanding" environment. From the gcc manual:

> A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at main. The most obvious example is an OS kernel.

Line 4 disables the use of the `.sdata` (small data) section in generated code. Refer to IBM (1998) to learn how this section would be used. Line 5 avoids unaligned memory accesses. The PowerISA allows embedded implementations to raise an exception on unaligned loads and stores, which is what this implementation does. Line 6 disables the use of floating point instructions. Floating point math is instead implemented in software. Line 7 tells the compiler, that code may not be relocated to a different address at runtime.

Linking

The executable is generated by the linker:

```
1 $ powerpc-eabi-ld crt.o <obj1.o> ... -o <binary.elf> \
2         -T elf32nux.x  \
3         -static      \
4         -nostdlib    \
5         -lgcc
```

Line 2 tells the linker to use a specified linker script. Line 3 uses static linking of libraries. Line 4 disables linking of standard libraries, especially libc (which we do not have). Line 5 links libgcc. This is a gcc internal library with routines that the compiler might use instead of directly emitting code for them. For example, optimized implementations of `memcpy` or soft floating point implementations. See libgcc for more information.

Extracting the binary image

The objcopy command from binutils allows to extract the raw bits from the executable:

```
1 $ powerpc-eabi-objcopy -O binary <binary.elf> <image.raw>
```

3.2.5 Useful commands

To inspect the resulting image you can use the hexdump program:

```
1 $ hexdump -C <image.raw>
```

It outputs a hex-dump to stdout.

The executable can be inspected using readelf to get information about symbol (variables and functions) locations. Objdump contains a dissasembler:

```
1 $ powerpc-eabi-objdump -d <binary.elf>
```

The extracted binaries can be disassembled, too, when supplying the relevant information on the architecture and file format:

```
1 $ powerpc-eabi-objdump \
2         -b binary -m powerpc -Mnux --endian=big \
3         -d <image.raw>
```

3.3 Loading and executing programs

The specifics of how to do this are defined by the system, in which the processor is used. The binary image generated using objcopy has to be transferred to the memory in the system. During this time, the processor should be in reset, or you have to know what you are doing. After loading, reset is released to start the program. The `crt.s` wrapper ensures, that the program goes into the sleep state when the user code terminates, i.e. returns from the start() function.

4 Running simulations

Available tests are described in detail in Chapter 4 of Friedmann (2013). Here, we will focus on how to use theses tests. In order to do that, we will also discuss the Makefile-based build flow.

4.1 Build flow

The build flow uses `make` and was inspired by the Linux build system kbuild (Chastain et al., 2015). There is a central configuration file `.config` in the top-level directory that defines configuration options. Source files are collected by using `Makefile.srclist` files throughout the directory hierarchy using the configured options. Separate files `Makefile.build` hold the build rules for `make` for each supported simulator. These Makefiles reside in run directories under `verification/sim_*`. Currently only ModelSim is supported in `verification/sim_model/`. Run directories for synthesis are located in `target/`.

4.1.1 Configuration options

These configuration options can be given in the global configuration file `.config`.

CONFIG_PLATFORM = *[virtex5 | tsmc65 | umc180 | designware | achronix_speedster22ihd]*

Select the target technology. Some components for clock generation, memories, multipliers, etc. only work in particular technologies, for example when using specialized FPGA resources. Note, that support for the achronix FPGA is merely a placeholder.

CONFIG_BOARD = *[ml505 | flyspi-board]*

Select which FPGA board is used. The Xilinx evaluation board ml505 (Xil, 2008) or the "flyspi" board of the electronic vision(s) group. Support for the latter is preliminary. This option is only relevant for the `target/emsys` implementation run directory.

CONFIG_USE_XILINX_MULTIPLIER = *[y | n]*

Select whether to use integrated DSP slices, when building for `CONFIG_PLATFORM` = virtex5.

CONFIG_WITH_BUS = *[y | n]*

Build the OMNIBUS implementation provided as part of the repository or not.

CONFIG_WITH_VECTOR = *[y | n]*
Build modules for the FXV unit or not.

4.1.2 Running simulations

To compile sources for simulation with ModelSim according to the selected configuration do:

```
1  $ cd verification/sim_model
2  $ make
3  $ vsim -do <simulation script>
```

Several simulation scripts are provided for the different test scenarios.

4.2 Tests

4.2.1 Program test

Simulation script: `verification/sim_model/sim_plt.do`
Top-level source: `testbenches/program_test.sv`
This test executes a number of programs from `test/testcode`. The processor is simulated in a two memory environment. Therefore, each program provides code and memory images, plus an additional expected data image. The test compares the memory contents after simulation to the expected image. Memory images can be provided as ASCII files with hexadecimal values or in raw binary form. For new programs the latter format should be preferred, since it can be generated using the flow described in Chapter 3.

4.2.2 Sequence test

Simulation script: `verification/sim_model/sim_seq.do`
Top-level source: `testbenches/sequence_test.sv`
This test generates random program sequences and compares the final state of internal registers to the expected state. FXV instructions are excluded from random generation. Also the `tw` and `twi` instructions are not allowed to avoid exceptions. The test can be configured with four preprocessor options:

PROGRAM_LENGTH Number of instructions of which the last one is always `wait`.

OPT_BCACHE Passed to the top-level processor option of the same name (see Section 2.1.1).

USE_CACHE Perform the test using an instruction cache.

OPT_DMEM_BUS If set, use `OPT_DMEM` = `Pu_types`::`DMEM_BUS` (see Section 2.1.1).

The simulation itself runs indefinitely. Specify a time when starting the simulation. For good coverage a simulated time of 100 ms should be aimed for.

4.2.3 Vector unit test

Simulation script: `verification/sim_model/sim_fub_vector.do`
Top-level source: `testbenches/fub_vector_test.sv`
This is a unit test for the vector unit. It tests the design in three phases:

1. Explicitly specified programs.
2. Randomly generated single instructions.
3. Randomly generated sequences of instructions.

A larger test set is defined by `testbenches/signoff_vector_test.sv` that includes the top-level of this test. The simulation script `verification/sim_model/sim_signoff_vector_test.do` uses this as top-level.

5 Instruction set

Contents

5.1 Implemented subset of Power ISA 2.06 . . . . . **22**
5.2 FXV vector instruction set **22**
5.2.1 Registers 22
Vector register file (VRF) 22
Vector condition register (VCR) 23
5.2.2 Primitive functions 23
5.2.3 Modulo halfword instructions 24
fxvmahm 24
fxvmatachm 25
fxvmulhm 25
fxvmultachm 25
fxvsubhm 25
fxvaddactachm 25
fxvaddtachm 26
fxvaddachm 26
fxvaddhm 26
fxvcmphm 26
fxvmtach 26
fxvsplath 27
5.2.4 Modulo byte instructions 27
fxvmabm 27
fxvmatacbm 27
fxvmulbm 27
fxvmultacbm 28
fxvsubbm 28
fxvaddactacb 28
fxvaddtacb 28
fxvaddacbm 28
fxvaddbm 29
fxvcmpb 29
fxvmtacb 29
fxvsplatb 29

5.2.5 Saturating, fractional, halfword instructions 29
fxvmahfs 29
fxvmtachf 30
fxvmatachfs 30
fxvmulhfs 30
fxvmultachfs 30
fxvsubhfs 31
fxvaddactachf 31
fxvaddachfs 31
fxvaddhfs 31
5.2.6 Saturating, fractional, byte instructions 31
fxvmabfs 31
fxvmtacbf 32
fxvmatacbfs 32
fxvmulbfs 32
fxvmultacbfs 32
fxvsubbfs 33
fxvaddactacbf 33
fxvaddacbfs 33
fxvaddbfs 33
5.2.7 Permute instructions 33
fxvsel 33
fxvshh 34
fxvshb 34
fxvpckbu 34
fxvpckbl 34
fxvupckbl 34
fxvupckbr 35
5.2.8 Load/store instructions 35
fxvlax 35
fxvstax 35
fxvinx 35
fxvoutx 36
5.3 Deprecated NEVER and SYNAPSE instruction sets 36

5.1 Implemented subset of Power ISA 2.06

PowerISA (2010) defines an embedded and a server environment, several mandatory and optional categories for these environments, and allows 32 and 64 bit implementations. The presented processor realizes an embedded environment supporting categories Base, Embedded, External Control, and Wait using 32 bit. In addition it supports the custom FXV instruction set and two now deprecated instruction sets NEVER and SYNAPSE. The full list of implemented instructions is given in Friedmann (2013).

Opcodes and instruction formats are defined in source file `rtl/packages/pu_inst_pkg.sv`. The high-level behavior, e.g. what registers are read and written or which functional unit is used, is given by the predecode module in `rtl/processor/predecode.sv`.

5.2 FXV vector instruction set

This section lists the implemented instructions of the fixed-point vector functional unit of Nux. The following rules are used in the notation:

- Round brackets indicate the contents of the register selected by the index in brackets. So, (VRA) stands for the contents of the register with index VRA.
- Subscripts denote the element of the vector using the current type: $(\mathrm{VRA})_{i/8}$ is the $i-$ th halfword element, $(\mathrm{VRA})_{i/16}$ is the $i-$ th byte element, and $(\mathrm{VRA})_{i/32}$ is the $i-$ th half-byte element.
- Lowercase letters u, v, w, m indicate single-precision elements. Uppercase letters U, V, W, M are double-precision elements.
- Underlined letters $\underline{u}, \underline{v}$ indicate full single-precision vectors.
- VRA, VRB, and VRT refer to vector register indices.
- ACC is the contents of the double-precision accumulation register.
- VCR is contents of the 32 bit vector condition register.
- RA, RB refer to general-purpose registers.

5.2.1 Registers

Vector register file (VRF)

0	31		63		95		127
$\mathrm{VR0}_{0/8}$	$\mathrm{VR0}_{1/8}$	$\mathrm{VR0}_{2/8}$	$\mathrm{VR0}_{3/8}$	$\mathrm{VR0}_{4/8}$	$\mathrm{VR0}_{5/8}$	$\mathrm{VR0}_{6/8}$	$\mathrm{VR0}_{7/8}$
⋮							
VR31							

The vector register file contains 32 128 bit vector registers, which consist of 8 halfword elements. Each element can also be used as two byte elements depending on the used instruction.

Vector condition register (VCR)

0–11	12–23	24–35	36–47	48–59	60–71	72–83	84–95
C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7

The vector condition register has one field for each of the 8 halfword elements:

0			3	4			7	8			11
EQ				GT				LT			
0	1	2	3	0	1	2	3	0	1	2	3

Each field consists of three condition areas for equality (EQ), greater than (GT), and less than (LT). For each nibble (4 bit) in the vector one condition bit is stored. Note: nibble-wise condition bits are a leftover from early days. Currently, there is no way to set or use them individually. Operations use two or four bits simultaneously for byte or halfword operations.

5.2.2 Primitive functions

Some instructions take only effect for individual elements if a given condition is met. The condition is determined by the contents of the vector condition register and the condition field in the instruction in the following way:

```
1:  function CONDITIONAL(cond, vcr_field)
2:      if cond = 0 then
3:          return true
4:      else if cond = 1 then
5:          return vcr_field.gt
6:      else if cond = 2 then
7:          return vcr_field.lt
8:      else if cond = 3 then
9:          return vcr_field.eq
10:     end if
11: end function
```

Shift and logical functions The SHIFT_LEFT(x, n) function gives the result of shifting x left by n positions while filling in zeros on the right. Shifted-out bits are discarded.

The SHIFT_RIGHT(x, n) function gives the result of shifting x right by n positions while filling in zeros on the left.

The SIGN_EXTEND(x) function gives the result of sign extending x to the width of the assignee of this function.

The BITWISE_AND(a, b) function represents the result of performing a bitwise and between a and b.

The BITWISE_OR(a, b) function represents the result of performing a bitwise or between a and b.

Saturating and fractional arithmetic Some instructions use fractional representation usually combined with saturating arithmetics. The function below defines saturating multiplication of two fractional operands of n bit size. The result is $2n$ bit wide and shifted to the left to get rid of one superfluous sign bit. Saturation only occurs if both operands represent -1 ($= 100\ldots0$ in binary).

1: **function** MULT_SAT_FRACT(a, b, n)
2: **if** $a = -2^{n-1} \wedge b = -2^{n-1}$ **then**
3: **return** $2^{2*n-1} - 1$
4: **else**
5: $u \leftarrow a \cdot b$
6: **return** SHIFT_LEFT$(u, 1)$
7: **end if**
8: **end function**

The following function defines saturating addition of two n bit operands. In case of overflows, the result saturates to the maximum and minimum representable value.

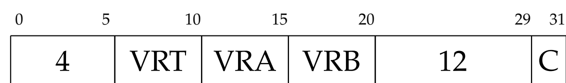
1: **function** ADD_SAT(a, b, n)
2: $y \leftarrow a + b$
3: **if** $y > 2^{n-1} - 1$ **then**
4: **return** $2^{n-1} - 1$
5: **else if** $y < -2^{n-1}$ **then**
6: **return** -2^{n-1}
7: **else**
8: **return** y
9: **end if**
10: **end function**

Memories and IO Function MEM(a) represents the 32 bit contents of main memory at address a.

Function IO(a) represents the contents of IO location a. The width of this result is determined by the width of vectors (`OPT_VECTOR_NUM_HALFWORDS`) and the number of slices (`OPT_VECTOR_SLICES`).

5.2.3 Modulo halfword instructions

Multiply accumulate halfword modulo



fxvmahm

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow (\text{VRA})_{i/8}$
3: $v \leftarrow (\text{VRB})_{i/8}$
4: $W \leftarrow u \cdot v$
5: $M \leftarrow \text{ACC}_{i/8}$
6: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
7: **if** enable **then**
8: $(\text{VRT})_{i/8} \leftarrow W + M_i \mod 2^{16}$
9: **end if**
10: **end for**

The contents of vector registers VRA and VRB are multiplied as 16 bit halfword elements and added to the contents of the 32 bit double-precision accumulation register. The result is written to vector register VRT modulo 2^{16} if the specified condition

is met.

Multiply accumulate to accumulator halfword modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	44	C	

fxvmatachm

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow (\text{VRA})_{i/8}$
3: $v \leftarrow (\text{VRB})_{i/8}$
4: $W \leftarrow u \cdot v$
5: $M \leftarrow \text{ACC}_{i/8}$
6: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
7: **if** enable **then**
8: $\text{ACC}_{i/8} \leftarrow W + M_i \mod 2^{32}$
9: **end if**
10: **end for**

The contents of vector registers VRA and VRB are multiplied as 16 bit halfword elements and added to the contents of the 32 bit double-precision accumulation register. The result is written to the accumulation register modulo 2^{32} if the specified condition is met.

Multiply halfword modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	76	C	

fxvmulhm

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow (\text{VRA})_{i/8}$
3: $v \leftarrow (\text{VRB})_{i/8}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
5: **if** enable **then**
6: $(\text{VRT})_{i/8} \leftarrow u \cdot v \mod 2^{16}$
7: **end if**
8: **end for**

The contents of vector registers VRA and VRB are multiplied as 16 bit halfword elements. The result is written to the vector register indicated by VRT modulo 2^{16} if the specified condition is met.

Multiply to accumulator halfword modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	108	C	

fxvmultachm

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow (\text{VRA})_{i/8}$
3: $v \leftarrow (\text{VRB})_{i/8}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
5: **if** enable **then**
6: $\text{ACC}_{i/8} \leftarrow u \cdot v \mod 2^{32}$
7: **end if**
8: **end for**

The contents of vector registers VRA and VRB are multiplied as 16 bit halfword elements. The result is written to the accumulator modulo 2^{32} if the specified condition is met.

Subtract halfword modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	332	C	

fxvsubhm

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow (\text{VRA})_{i/8}$
3: $v \leftarrow (\text{VRB})_{i/8}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
5: **if** enable **then**
6: $(\text{VRT})_{i/8} \leftarrow u - v \mod 2^{16}$
7: **end if**
8: **end for**

The contents of the vector register indicated by VRB is subtracted from the contents of the register indicated by VRA. The result is written to VRT modulo 2^{16} if the specified condition is met.

Add accumulator to accumulator halfword modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	364	C	

fxvaddactachm

1: **for** $0 \leq i < 8$ **do**

2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/8})$
3: $M \leftarrow ACC_{i/8}$
4: $\text{enable} \leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/8})$
5: **if** enable **then**
6: $\text{ACC}_{i/8} \leftarrow U + M \mod 2^{32}$
7: **end if**
8: **end for**

The contents of vector register VRA is sign extended to double precision and added to the accumulation register. The result is written to the accumulator modulor 2^{32} if the specified condition is met.

Add and save to accumulator halfword modulo

0	5	10	15	20 29	31
4	VRT	VRA	VRB	428	C

fxvaddtachm

1: **for** $0 \leq i < 8$ **do**
2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/8})$
3: $V \leftarrow \text{SIGN_EXTEND}((\text{VRB})_{i/8})$
4: $\text{enable} \leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/8})$
5: **if** enable **then**
6: $\text{ACC}_{i/8} \leftarrow U + V$
7: **end if**
8: **end for**

The contents of vector registers VRA and VRB are sign extended to double precision. If the specified condition is met, their sum is written to the accumulator.

Add accumulator halfword modulo

0	5	10	15	20 29	31
4	VRT	VRA	VRB	396	C

fxvaddachm

1: **for** $0 \leq i < 8$ **do**
2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/8})$
3: $M \leftarrow \text{ACC}_{i/8}$
4: $\text{enable} \leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/8})$
5: **if** enable **then**
6: $(\text{VRT})_{i/8} \leftarrow U + M \mod 2^{16}$
7: **end if**
8: **end for**

The contents of vector register VRA is sign extended to double precision and added to the accumulator. The result is written to vector register VRT modulo 2^{16}.

Add halfword modulo

0	5	10	15	20 29	31
4	VRT	VRA	VRB	460	C

fxvaddhm

1: **for** $0 \leq i < 8$ **do**
2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/8})$
3: $V \leftarrow \text{SIGN_EXTEND}((\text{VRB})_{i/8})$
4: $\text{enable} \leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/8})$
5: **if** enable **then**
6: $(\text{VRT})_{i/8} \leftarrow U + V \mod 2^{16}$
7: **end if**
8: **end for**

The contents of vector registers indicated by VRA and VRB are added. If the specified condition is met, the result is written to the vector register indicated by VRT.

Compare halfword modulo

0	5	10	15	20 29	31
4	VRT	VRA	VRB	300	C

fxvcmphm

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow (\text{VRA})_{i/8}$
3: $\text{VCR}_{i/8}.eq \leftarrow u = 0$
4: $\text{VCR}_{i/8}.gt \leftarrow u > 0$
5: $\text{VCR}_{i/8}.lt \leftarrow u < 0$
6: **end for**

The contents of the vector register indicated by VRA is compared to zero and the result written to the vector condition register.

Move to accumulator halfword fractional

0	5	10	15	20 29	31
4	VRT	VRA	VRB	15	C

fxvmtach

1: **for** $0 \leq i < 8$ **do**
2: $\text{enable} \leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/8})$

3: **if** enable **then**
4: $(\text{ACC})_{i/8} \leftarrow (\text{VRA})_{i/8}$
5: **end if**
6: **end for**

If enabled, the contents of elements in register VRA are copied to the accumulator aligned to the right.

Splat halfword

0	5	10	15	20	29 31
4	VRT	RA	/	268	/

fxvsplath

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow \text{RA} \mod 2^{16}$
3: $\text{VRT}_{i/8} \leftarrow u$
4: **end for**

The lower halfword of general-purpose register RA is copied to each element of VRT.

5.2.4 Modulo byte instructions

Multiply accumulate byte modulo

0	5	10	15	20	29 31
4	VRT	VRA	VRB	13	C

fxvmabm

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: $W \leftarrow u \cdot v$
5: $M \leftarrow \text{ACC}_{i/16}$
6: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
7: **if** enable **then**
8: $(\text{VRT})_{i/16} \leftarrow W + M_i \mod 2^8$
9: **end if**
10: **end for**

The contents of vector registers VRA and VRB are multiplied as 8 bit byte elements and added to the contents of the 16 bit double-precision accumulation register. The result is written to vector register VRT modulo 2^8 if the specified condition is met.

Multiply accumulate to accumulator byte modulo

0	5	10	15	20	29 31
4	VRT	VRA	VRB	45	C

fxvmatacbm

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: $W \leftarrow u \cdot v$
5: $M \leftarrow \text{ACC}_{i/16}$
6: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
7: **if** enable **then**
8: $\text{ACC}_{i/16} \leftarrow W + M_i \mod 2^{16}$
9: **end if**
10: **end for**

The contents of vector registers VRA and VRB are multiplied as 8 bit byte elements and added to the contents of the 16 bit double-precision accumulation register. The result is written to the accumulation register modulo 2^{16} if the specified condition is met.

Multiply byte modulo

0	5	10	15	20	29 31
4	VRT	VRA	VRB	77	C

fxvmulbm

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
5: **if** enable **then**
6: $(\text{VRT})_{i/16} \leftarrow u \cdot v \mod 2^8$
7: **end if**
8: **end for**

The contents of vector registers VRA and VRB are multiplied as 8 bit byte elements.

The result is written to the vector register indicated by VRT modulo 2^8 if the specified condition is met.

Multiply to accumulator byte modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	109	C	

fxvmultacbm

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
5: **if** enable **then**
6: $\text{ACC}_{i/16} \leftarrow u \cdot v \mod 2^{16}$
7: **end if**
8: **end for**

The contents of vector registers VRA and VRB are multiplied as 8 bit byte elements. The result is written to the accumulator modulo 2^{16} if the specified condition is met.

Subtract byte modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	333	C	

fxvsubbm

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
5: **if** enable **then**
6: $(\text{VRT})_{i/16} \leftarrow u - v \mod 2^8$
7: **end if**
8: **end for**

The contents of the vector register indicated by VRB is subtracted from the contents of the register indicated by VRA. The result is written to VRT modulo 2^8 if the specified condition is met.

Add accumulator to accumulator byte

0	5	10	15	20	29	31
4	VRT	VRA	VRB	365	C	

fxvaddactacb

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow$ SIGN_EXTEND($(\text{VRA})_{i/16}$)
3: $M \leftarrow ACC_{i/16}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
5: **if** enable **then**
6: $\text{ACC}_{i/16} \leftarrow U + M \mod 2^{16}$
7: **end if**
8: **end for**

The contents of vector register VRA is sign extended to double precision and added to the accumulation register. The result is written to the accumulator modulo 2^{16} if the specified condition is met.

Add and save to accumulator byte modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	429	C	

fxvaddtacb

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow$ SIGN_EXTEND($(\text{VRA})_{i/16}$)
3: $V \leftarrow$ SIGN_EXTEND($(\text{VRB})_{i/16}$)
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
5: **if** enable **then**
6: $\text{ACC}_{i/16} \leftarrow U + V$
7: **end if**
8: **end for**

The contents of vector registers VRA and VRB are sign extended to double precision. If the specified condition is met, their sum is written to the accumulator.

Add accumulator byte modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	397	C	

fxvaddacbm

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow$ SIGN_EXTEND($(\text{VRA})_{i/16}$)

3: $M \leftarrow \text{ACC}_{i/16}$
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
5: **if** enable **then**
6: $(\text{VRT})_{i/16} \leftarrow U + M \mod 2^8$
7: **end if**
8: **end for**

The contents of vector register VRA is sign extended to double precision and added to the accumulator. The result is written to vector register VRT modulo 2^8.

Add byte modulo

0	5	10	15	20	29	31
4	VRT	VRA	VRB	461	C	

fxvaddbm

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow$ SIGN_EXTEND($(\text{VRA})_{i/16}$)
3: $V \leftarrow$ SIGN_EXTEND($(\text{VRB})_{i/16}$)
4: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
5: **if** enable **then**
6: $(\text{VRT})_{i/16} \leftarrow U + V \mod 2^8$
7: **end if**
8: **end for**

The contents of vector registers indicated by VRA and VRB are added. If the specified condition is met, the result is written to the vector register indicated by VRT.

Compare byte

0	5	10	15	20	29	31
4	VRT	VRA	VRB	301	C	

fxvcmpb

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $\text{VCR}_{i/16}.eq \leftarrow u = 0$
4: $\text{VCR}_{i/16}.gt \leftarrow u > 0$
5: $\text{VCR}_{i/16}.lt \leftarrow u < 0$
6: **end for**

The contents of the vector register indicated by VRA is compared to zero and the result written to the vector condition register.

Move to accumulator byte

0	5	10	15	20	29	31
4	VRT	VRA	VRB	14	C	

fxvmtacb

1: **for** $0 \leq i < 16$ **do**
2: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
3: **if** enable **then**
4: $(\text{ACC})_{i/16} \leftarrow (\text{VRA})_{i/16}$
5: **end if**
6: **end for**

If enabled, the contents of elements in register VRA are copied to the accumulator aligned to the right.

Splat byte

0	5	10	15	20	29	31
4	VRT	RA	/	269	/	

fxvsplatb

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow \text{RA} \mod 2^8$
3: $\text{VRT}_{i/16} \leftarrow u$
4: **end for**

The lower byte of general-purpose register RA is copied to each element of VRT.

5.2.5 Saturating, fractional, halfword instructions

Multiply accumulate halfword fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	28	C	

fxvmahfs

1: **for** $0 \leq i < 8$ **do**
2: $u \leftarrow (\text{VRA})_{i/8}$
3: $v \leftarrow (\text{VRB})_{i/8}$
4: $W \leftarrow$ MULT_SAT_FRACT(u, v, 16 bit)
5: $M \leftarrow$ ADD_SAT($\text{ACC}_{i/8}, W$, 32 bit)
6: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
7: **if** enable **then**

```
 8:         (VRT)_{i/8} ← SHIFT_RIGHT(M, 16)
 9:     end if
10: end for
```

Perform multiplication of registers VRA and VRB using fractional representation and saturating on overflows. The double width result is added with the contents of the accumulation register again with saturation on overflow. The upper 16 bits of the result are returned to register VRT if the condition is met. The accumulation register is not modified.

Move to accumulator halfword fractional

0	5	10	15	20	29	31
4	VRT	VRA	VRB	31	C	

fxvmtachf

```
1: for 0 ≤ i < 8 do
2:     enable ← CONDITIONAL(C, VCR_{i/8})
3:     if enable then
4:         (ACC)_{i/8} ← (VRA)_{i/8} · 2^16
5:     end if
6: end for
```

If enabled, the contents of elements in register VRA are copied to the accumulator aligned to the left.

Multiply accumulate and save to accumulator halfword fractional

0	5	10	15	20	29	31
4	VRT	VRA	VRB	60	C	

fxvmatachfs

```
 1: for 0 ≤ i < 8 do
 2:     u ← (VRA)_{i/8}
 3:     v ← (VRB)_{i/8}
 4:     W ← MULT_SAT_FRACT(u, v, 16 bit)
 5:     M ← ADD_SAT(ACC_{i/8}, W, 32 bit)
 6:     enable ← CONDITIONAL(C, VCR_{i/8})
 7:     if enable then
 8:         (ACC)_{i/8} ← M
 9:     end if
10: end for
```

Perform a multiply-add operation of registers VRA, VRB, and the accumulator using saturation and fractional arithmetics. If enabled, the result is stored in the accumulator.

Multiply halfword fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	92	C	

fxvmulhfs

```
1: for 0 ≤ i < 8 do
2:     u ← (VRA)_{i/8}
3:     v ← (VRB)_{i/8}
4:     W ← MULT_SAT_FRACT(u, v, 16 bit)
5:     enable ← CONDITIONAL(C, VCR_{i/8})
6:     if enable then
7:         (VRT)_{i/8} ← SHIFT_RIGHT(M, 16)
8:     end if
9: end for
```

Perform saturating multiplication in fractional representation of the contents of registers VRA and VRB. Store the truncated result to register VRT if enabled.

Multiply and save to accumulator halfword fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	124	C	

fxvmultachfs

```
1: for 0 ≤ i < 8 do
2:     u ← (VRA)_{i/8}
3:     v ← (VRB)_{i/8}
4:     W ← MULT_SAT_FRACT(u, v, 16 bit)
5:     enable ← CONDITIONAL(C, VCR_{i/8})
6:     if enable then
7:         (ACC)_{i/8} ← M
8:     end if
9: end for
```

Perform saturating multiplication in fractional representation of the contents of registers VRA and VRB. Store the result to the accumulator.

Subtract halfword fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	348		C

fxvsubhfs

1: **for** $0 \leq i < 8$ **do**
2: $U \leftarrow$ SIGN_EXTEND$((\text{VRA})_{i/8})$
3: $V \leftarrow$ SIGN_EXTEND$(-(\text{VRB})_{i/8})$
4: $W \leftarrow$ ADD_SAT_FRACT$(U, V, 32\,\text{bit})$
5: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
6: **if** enable **then**
7: $(\text{VRT})_{i/8} \leftarrow$ SHIFT_RIGHT$(W, 16)$
8: **end if**
9: **end for**

Add the contents of register VRA to the negated contents of register VRB using saturation. Store the truncated result to register VRT if enabled.

Add accumulator and save to accumulator halfword fractional

0	5	10	15	20	29	31
4	VRT	VRA	VRB	380		C

fxvaddactachf

1: **for** $0 \leq i < 8$ **do**
2: $U \leftarrow$ SIGN_EXTEND$((\text{VRA})_{i/8})$
3: $V \leftarrow (\text{ACC})_{i/8}$
4: $W \leftarrow$ ADD_SAT_FRACT$(U, V, 32\,\text{bit})$
5: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
6: **if** enable **then**
7: $(\text{ACC})_{i/8} \leftarrow W$
8: **end if**
9: **end for**

Add the sign extended contents of register VRA to the accumulator using saturating arithmetics and store the result to the accumulator.

Add accumulator halfword fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	412		C

fxvaddachfs

1: **for** $0 \leq i < 8$ **do**
2: $U \leftarrow$ SIGN_EXTEND$((\text{VRA})_{i/8})$
3: $V \leftarrow (\text{ACC})_{i/8}$
4: $W \leftarrow$ ADD_SAT_FRACT$(U, V, 32\,\text{bit})$
5: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
6: **if** enable **then**
7: $(\text{VRT})_{i/8} \leftarrow$ SHIFT_RIGHT$(W, 16)$
8: **end if**
9: **end for**

Add the sign extended contents of register VRA to the accumulator using saturating arithmetics. Store the truncated result to register VRT.

Add halfword fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	476		C

fxvaddhfs

1: **for** $0 \leq i < 8$ **do**
2: $U \leftarrow$ SIGN_EXTEND$((\text{VRA})_{i/8})$
3: $V \leftarrow$ SIGN_EXTEND$((\text{VRB})_{i/8})$
4: $W \leftarrow$ ADD_SAT_FRACT$(U, V, 32\,\text{bit})$
5: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/8}$)
6: **if** enable **then**
7: $(\text{VRT})_{i/8} \leftarrow$ SHIFT_RIGHT$(W, 16)$
8: **end if**
9: **end for**

Add the contents of registers VRA and VRB using saturating arithmetics. Store the result to register VRT.

5.2.6 Saturating, fractional, byte instructions

Multiply accumulate byte fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	29		C

fxvmabfs

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$

4: $W \leftarrow \text{MULT_SAT_FRACT}(u, v, 8\,\text{bit})$
5: $M \leftarrow \text{ADD_SAT}(\text{ACC}_{i/16}, W, 16\,\text{bit})$
6: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
7: **if** enable **then**
8: $(\text{VRT})_{i/16} \leftarrow \text{SHIFT_RIGHT}(M, 8)$
9: **end if**
10: **end for**

Perform multiplication of registers VRA and VRB using fractional representation and saturating on overflows. The double width result is added with the contents of the accumulation register again with saturation on overflow. The upper 8 bits of the result are returned to register VRT if the condition is met. The accumulation register is not modified.

Move to accumulator byte fractional

0	5	10	15	20	29	31
4	VRT	VRA	VRB	30	C	

fxvmtacbf

1: **for** $0 \le i < 16$ **do**
2: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
3: **if** enable **then**
4: $(\text{ACC})_{i/16} \leftarrow (\text{VRA})_{i/16} \cdot 2^8$
5: **end if**
6: **end for**

If enabled, the contents of elements in register VRA are copied to the accumulator aligned to the left.

Multiply accumulate and save to accumulator byte fractional

0	5	10	15	20	29	31
4	VRT	VRA	VRB	61	C	

fxvmatacbfs

1: **for** $0 \le i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: $W \leftarrow \text{MULT_SAT_FRACT}(u, v, 8\,\text{bit})$
5: $M \leftarrow \text{ADD_SAT}(\text{ACC}_{i/16}, W, 16\,\text{bit})$
6: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
7: **if** enable **then**
8: $(\text{ACC})_{i/16} \leftarrow M$
9: **end if**
10: **end for**

Perform a multiply-add operation of registers VRA, VRB, and the accumulator using saturation and fractional arithmetics. If enabled, the result is stored in the accumulator.

Multiply byte fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	93	C	

fxvmulbfs

1: **for** $0 \le i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: $W \leftarrow \text{MULT_SAT_FRACT}(u, v, 8\,\text{bit})$
5: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
6: **if** enable **then**
7: $(\text{VRT})_{i/16} \leftarrow \text{SHIFT_RIGHT}(M, 8)$
8: **end if**
9: **end for**

Perform saturating multiplication in fractional representation of the contents of registers VRA and VRB. Store the truncated result to register VRT if enabled.

Multiply and save to accumulator byte fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	125	C	

fxvmultacbfs

1: **for** $0 \le i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: $W \leftarrow \text{MULT_SAT_FRACT}(u, v, 8\,\text{bit})$
5: enable $\leftarrow$ CONDITIONAL(C, $\text{VCR}_{i/16}$)
6: **if** enable **then**
7: $(\text{ACC})_{i/16} \leftarrow M$
8: **end if**
9: **end for**

Perform saturating multiplication in fractional representation of the contents of registers VRA and VRB. Store the result to the accumulator.

Subtract byte fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	349		C

fxvsubbfs

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/16})$
3: $V \leftarrow \text{SIGN_EXTEND}(-(\text{VRB})_{i/16})$
4: $W \leftarrow \text{ADD_SAT_FRACT}(U, V, 16\,\text{bit})$
5: enable $\leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/16})$
6: **if** enable **then**
7: $(\text{VRT})_{i/16} \leftarrow \text{SHIFT_RIGHT}(W, 8)$
8: **end if**
9: **end for**

Add the contents of register VRA to the negated contents of register VRB using saturation. Store the truncated result to register VRT if enabled.

Add accumulator and save to accumulator byte fractional

0	5	10	15	20	29	31
4	VRT	VRA	VRB	381		C

fxvaddactacbf

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/16})$
3: $V \leftarrow (\text{ACC})_{i/16}$
4: $W \leftarrow \text{ADD_SAT_FRACT}(U, V, 16\,\text{bit})$
5: enable $\leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/16})$
6: **if** enable **then**
7: $(\text{ACC})_{i/16} \leftarrow W$
8: **end if**
9: **end for**

Add the sign extended contents of register VRA to the accumulator using saturating arithmetics and store the result to the accumulator.

Add accumulator byte fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	413		C

fxvaddacbfs

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/16})$
3: $V \leftarrow (\text{ACC})_{i/16}$
4: $W \leftarrow \text{ADD_SAT_FRACT}(U, V, 16\,\text{bit})$
5: enable $\leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/16})$
6: **if** enable **then**
7: $(\text{VRT})_{i/16} \leftarrow \text{SHIFT_RIGHT}(W, 8)$
8: **end if**
9: **end for**

Add the sign extended contents of register VRA to the accumulator using saturating arithmetics. Store the truncated result to register VRT.

Add byte fractional saturating

0	5	10	15	20	29	31
4	VRT	VRA	VRB	477		C

fxvaddbfs

1: **for** $0 \leq i < 16$ **do**
2: $U \leftarrow \text{SIGN_EXTEND}((\text{VRA})_{i/16})$
3: $V \leftarrow \text{SIGN_EXTEND}((\text{VRB})_{i/16})$
4: $W \leftarrow \text{ADD_SAT_FRACT}(U, V, 16\,\text{bit})$
5: enable $\leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/16})$
6: **if** enable **then**
7: $(\text{VRT})_{i/16} \leftarrow \text{SHIFT_RIGHT}(W, 8)$
8: **end if**
9: **end for**

Add the contents of registers VRA and VRB using saturating arithmetics. Store the result to register VRT.

5.2.7 Permute instructions

Select

0	5	10	15	20	29	31
4	VRT	VRA	VRB	319		C

fxvsel

1: **for** $0 \leq i < 16$ **do**
2: $u \leftarrow (\text{VRA})_{i/16}$
3: $v \leftarrow (\text{VRB})_{i/16}$
4: enable $\leftarrow \text{CONDITIONAL}(\text{C}, \text{VCR}_{i/16})$
5: **if** enable **then**
6: $(\text{VRT})_{i/16} \leftarrow v$

```
 7:     else
 8:         (VRT)_{i/16} ← u
 9:     end if
10: end for
```

Use the contents of the condition register (VCR) to merge vectors VRA and VRB into the result vector VRT. The condition code selects which field of VCR to use. Select always merges bytewise. Hint: use in conjunction with the appropriate compare operation.

Shift left halfword

0–5	5–10	10–15	15–20	20–29	31
4	VRT	VRA	VRB	316	C

fxvshh

1: **for** $0 \le i < 8$ **do**
2: $\quad u \leftarrow (\text{VRA})_{i/8}$
3: $\quad (\text{VRT})_{i/8} \leftarrow \text{SHIFT_LEFT}(u, VRB)$
4: **end for**

Shift the contents of register VRA left halfword-wise by the amount VRB. Here VRB is used as immediate value. Store the result in register VRT.

Shift left byte

0–5	5–10	10–15	15–20	20–29	31
4	VRT	VRA	VRB	317	C

fxvshb

1: **for** $0 \le i < 16$ **do**
2: $\quad u \leftarrow (\text{VRA})_{i/16}$
3: $\quad (\text{VRT})_{i/16} \leftarrow \text{SHIFT_LEFT}(u, VRB)$
4: **end for**

Shift the contents of register VRA left byte-wise by the amount VRB. Here VRB is used as immediate value. Store the result in register VRT.

Pack byte upper

0–5	5–10	10–15	15–20	20–29	31
4	VRT	VRA	VRB	239	C

fxvpckbu

1: $m \leftarrow \text{SHIFT_RIGHT}(\texttt{0xffff}, C)$
2: **for** $0 \le i < 16$ **do**
3: $\quad$ **if** $i < 4$ **then**
4: $\quad\quad u \leftarrow \text{SHIFT_RIGHT}(\text{VRA}_{2i/16}, C)$
5: $\quad\quad \text{VRT}_{i/16} \leftarrow \text{BITWISE_AND}(u, m)$
6: $\quad$ **else**
7: $\quad\quad u \leftarrow \text{SHIFT_RIGHT}(\text{VRB}_{2(i-4)/16}, C)$
8: $\quad\quad \text{VRT}_{i/16} \leftarrow \text{BITWISE_AND}(u, m)$
9: $\quad$ **end if**
10: **end for**

Pack the upper half of fractional numbers of 8-C bit stored in halfword elements in registers VRA and VRB into register VRT, aligned to the right.

Pack byte lower

0–5	5–10	10–15	15–20	20–29	31
4	VRT	VRA	VRB	255	C

fxvpckbl

1: $m \leftarrow \text{SHIFT_RIGHT}(\texttt{0xffff}, C)$
2: **for** $0 \le i < 16$ **do**
3: $\quad$ **if** $i < 4$ **then**
4: $\quad\quad u \leftarrow \text{SHIFT_LEFT}(\text{VRA}_{2i/16}, 8 - 2C)$
5: $\quad\quad v \leftarrow \text{SHIFT_RIGHT}(\text{VRA}_{2i+1/16}, 2C)$
6: $\quad\quad w \leftarrow \text{BITWISE_OR}(u, v)$
7: $\quad\quad \text{VRT}_{i/16} \leftarrow \text{BITWISE_AND}(w, m)$
8: $\quad$ **else**
9: $\quad\quad u \leftarrow \text{SHIFT_RIGHT}(\text{VRB}_{2(i-4)/16}, C)$
10: $\quad\quad \text{VRT}_{i/16} \leftarrow \text{BITWISE_AND}(u, m)$
11: $\quad$ **end if**
12: **end for**

Pack the lower half of fractional numbers of 8-C bit stored in halfword elements in registers VRA and VRB into register VRT, aligned to the right.

Unpack byte left

0–5	5–10	10–15	15–20	20–29	31
4	VRT	VRA	VRB	287	C

fxvupckbl

1:

Reverse the packing operation performed by fxvpckbu and fxvpckbl and

store the left half of the elements in register VRT.

Unpack byte right

0	5	10	15	20	29 31
4	VRT	VRA	VRB	271	C

fxvupckbr

1:

Reverse the packing operation performed by fxvpckbu and fxvpckbl and store the right half of the elements in register VRT.

5.2.8 Load/store instructions

Load array indexed

0	5	10	15	20	29 31
4	VRT	RA	RB	492	/

fxvlax

1: **if** RA = 0 **then**
2: $a \leftarrow (\mathrm{RB})$
3: **else**
4: $a \leftarrow (\mathrm{RA}) + (\mathrm{RB})$
5: **end if**
6: **for** $0 \le i <$ NUM_SLICES **do**
7: **for** $0 \le j < 4$ **do**
8: $(\mathrm{VRT})^i_{j\times 32:(j+1)\times 32} \leftarrow \mathrm{MEM}(a, a+3)$
9: $a \leftarrow a + 4$
10: **end for**
11: **end for**

The effective address is computed from general purpose registers RA and RB. If RA is zero, then the effective address is taken from RB. The contents of the vector register indicated by VRT is loaded starting from the memory location indicated by the effective address across the complete array of slices.

Store array indexed

0	5	10	15	20	29 31
4	VRT	RA	RB	508	/

fxvstax

1: **if** RA = 0 **then**
2: $a \leftarrow (\mathrm{RB})$
3: **else**
4: $a \leftarrow (\mathrm{RA}) + (\mathrm{RB})$
5: **end if**
6: **for** $0 \le i <$ NUM_SLICES **do**
7: **for** $0 \le j < 4$ **do**
8: $\mathrm{MEM}(a, a+3) \leftarrow (\mathrm{VRT})^i_{j\times 32:(j+1)\times 32}$
9: $a \leftarrow a + 4$
10: **end for**
11: **end for**

The effective address is computed from general purpose registers RA and RB. If RA is zero, then the effective address is taken from RB. The contents of the vector register indicated by VRT is stored to memory starting from the location indicated by the effective address across the complete array of slices.

Input indexed

0	5	10	15	20	29 31
4	VRT	RA	RB	236	/

fxvinx

1: **if** RA = 0 **then**
2: $a \leftarrow (\mathrm{RB})$
3: **else**
4: $a \leftarrow (\mathrm{RA}) + (\mathrm{RB})$
5: **end if**
6: $\underline{u} \leftarrow \mathrm{IO}(a)$
7: **for** $0 \le i <$ NUM_SLICES **do**
8: **for** $0 \le j < 16$ **do**
9: $\mathrm{VRT}^i_{j/16} \leftarrow \underline{u}^i_{j/16}$
10: **end for**
11: **end for**

The effective address is computed from general purpose registers RA and RB. If RA is zero, then the effective address is taken from RB. Data is loaded in parallel from the IO location referenced by the effective ad-

dress and moved to the destination register VRT.

Output indexed

0	5	10	15	20	29 31
4	VRT	RA	RB	252	/

fxvoutx

1: **if** RA = 0 **then**
2: $a \leftarrow (\mathrm{RB})$
3: **else**
4: $a \leftarrow (\mathrm{RA}) + (\mathrm{RB})$
5: **end if**
6: $\underline{u} \leftarrow 0$
7: **for** $0 \leq i <$ NUM_SLICES **do**
8: **for** $0 \leq j < 16$ **do**
9: $\underline{u}^i_{j/16} \leftarrow \mathrm{VRT}^i_{j/16}$
10: **end for**
11: **end for**
12: $\mathrm{IO}(a) \leftarrow \underline{u}$

The effective address is computed from general purpose registers RA and RB. If RA is zero, then the effective address is taken from RB. The contents of registers VRT in all slices is written to the IO location referenced by the effective address.

5.3 Deprecated NEVER and SYNAPSE instruction sets

The SYNAPSE instruction set is defined in Friedmann (2013). For NEVER there is no documentation. All extensions FXV, SYNAPSE, and NEVER use overlapping opcode spaces. Therefore, they can not be used together. Currently only the FXV instruction set is of relevance, because it is used in actual hardware (Friedmann and Schemmel, 2015) and it is generic as opposed to the other two.

Bibliography

Michael Elizabeth Chastain, Kai Germaschewski, Sam Ravnborg, and Jan Engelhardt. *Linux Kernel Makefiles*, 2015. URL `https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt`.

Simon Friedmann. *A New Approach to Learning in Neuromorphic Hardware*. PhD thesis, Ruprecht-Karls-Universität Heidelberg, 2013.

Simon Friedmann and Johannes Schemmel. Dls plasticity. *IJCS*, 2015. to be published.

GCC installation. Installing gcc. Website, 2015. URL `https://gcc.gnu.org/install`.

Microcontroller Applications IBM. Developing PowerPC Embedded Application Binary Interface (EABI) Compliant Programs, September 1998. Version 1.0.

libgcc. The GCC low-level runtime library. Website, 2015. URL `https://gcc.gnu.org/onlinedocs/gccint/Libgcc.html`.

PowerISA. PowerISA Version 2.06 Revision B. Technical report, power.org, July 2010. Available at `http://www.power.org/resources/reading/`.

Richard Stallman. *Using the GNU Compiler Collection*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA, for gcc version 4.9.2 edition, 2015. URL `http://gcc.gnu.org`.

ML505/ML506/M ML505/ML506/ML507 Evaluation Platform User Guide. Xilinx, Inc., November 2008. v3.1.