Patmos Reference Handbook

Martin Schoeberl, Florian Brandner, Stefan Hepp,
Wolfgang Puffitsch, Daniel Prokesch

February 20, 2018

Preface

This handbook shall evolve to the documentation of the Patmos processor and the Patmos compiler. In the mean time it is intended to collect design notes and discussions. The latest version of this handbook is contained as LaTeX source in the Patmos repository in directory `patmos/doc/handbook` and can be built with `make`.

Acknowledgment

We would like to thank Tommy Thorn for the always intense and enjoyable discussions of the Patmos ISA and processor design in general. Jack Whitham offered his experience with RISC ISA design and trade-offs. Gernot Gebhard and Christoph Cullmann gave valuable feedback on the ISA related to WCET analysis. Sahar Abbaspourseyedi has been working on the stack cache to verify the ideas and concepts presented here. We thank Rasmus Bo Sørensen for fixing some documentation errors.

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

Contents

List of Figures

List of Tables

Listings

1 Introduction

Real-time systems need a time-predictable execution platform so that the worst-case execution time (WCET) can be statically estimated. It has been argued that we have to rethink computer architecture for real-time systems instead of trying to catch up with new processors in the WCET analysis tools [11, 4].

We present the time-predictable processor Patmos as one approach to attack the complexity issue of WCET analysis. Patmos is a static scheduled, dual-issue RISC processor that is optimized for real-time systems.

1.1 Hello World

We can start with the standard, harmless looking Hello World:[1]

```
int main() {
    printf("Hello Patmos!\n");
}
```

With the compiler installed it can be compiled to a Patmos executable and run with the simulator as follows:

```
patmos-clang hello.c
pasim a.out
```

However, this innocent examples is quiet challenging for an embedded system: It needs a C compiler, an implementation of the standard C library, printf itself is a challenging function, the generated ELF file needs to be understood by a tool and the individual sections downloaded, and finally a terminal (often a serial line) needs to be available on the target, and your test PC needs to have a serial line as well and a terminal program needs to run.

Therefore, we might start from a minimal assembler program and execute that in the simulator and emulator. From that base we can build up too a multi-core version of Patmos that executes in an FPGA and bootstraps with programs loaded via a serial port.

1.2 Building Patmos

The whole build process of Patmos,[2] applications in assembler and in C, configuration of the FPGA, and downloading an application is `Makefile` based. The build of Patmos is within the `patmos` folder, therefore, the following descriptions assumes you have changed to:

```
t-crest/patmos
```

The complete design flow (including the LLVM based C compiler) can execute in a Linux machine. The flow without the C compiler should be able to execute in a Windows/Cygwin environment. Under Mac OS X all tools, except Quartus, are working (ModelSim under wine). For FPGA synthesis and configuration Windows XP within a VMWare virtual machine is a possible solution.

On a Linux box with the installed LLVM compiler and Quartus in your PATH, the complete build processes for a Hello World is as follows:

```
make BOOTAPP=bootable-bootloader APP=hello_puts \
   tools comp gen synth config download
```

[1] This example code is not part of the distribution, but can be put at any directory.

[2] Get the source from GitHub with: `git clone git@github.com:t-crest/patmos`

However, this involves quite many steps. Therefore, we suggest doing some *manual* buildup to explore the full build process and the possibilities.

As a start we build some tools (e.g., the assembler, simulator, file conversion utility, and the boot loader). This has to be done once only.

```
make tools
```

1.2.1 A Few Assembler Instructions

We start with a very small assembler program that moves a few values into registers (see `asm/basic.s`). With following make command the program is assembled and executed in the software simulator of Patmos.

```
make swsim BOOTAPP=basic
```

The simulator options are set to write out the register contents after each instruction. The emulator (the Chisel based simulator) can execute the same program with following command:

```
make hwsim BOOTAPP=basic
```

This command assembles the application, executes the Chisel based hardware construction during which the program is used to initialize the on-chip ROM, generates a C++ based emulator, compiles that emulator, and executes it. The emulator shows the register content after each instruction.

Those two Patmos simulations, the software simulator and the Chisel based emulator, are used for a co-simulation based test. In this co-simulation all available assembler programs are executed in both simulations and the register out put is compared. The test can be started with:

```
make test
```

1.2.2 We Can Blink in Assembler

Assembler programming is just used for small tests. And on Patmos assembler programs end up in the boot ROM, which means that a new processor needs to be generated and synthesized as shown below:

```
make asm BOOTAPP=hello gen synth config
```

This make command assembles a blinking LED example written in Patmos assembler. You can find the assembler source in Listing 1.1 and in `asm/hello.s`.

1.2.3 A C Based Blinking LED

As a first real example we build the embedded version of Hello World, the blinking LED, from a C program. You can find the C source in `c/blinking.c`.

```
make BOOTAPP=bootable-blinking comp gen synth config
```

Additionally to blinking an LED this program also writes alternating ‘0’ and ‘1’ to the serial port. Connect the FPGA board to your serial port, open a terminal of your choice (e.g., `gtkterm`), connect to the serial port, set the baud rate to 115200, no parity, and no handshaking. You should see alternating ‘0’ and ‘1’ sent out synchronous to the blinking.

Note that the program name (`blink`) is prefixed by `bootable-`. The marker selects the right compiler settings for a program that ends up in the Patmos on-chip ROM. As the on-chip memory is limited, only tiny programs are supported in this execution mode.

Figure 1.2 shows the code for the embedded Hello World C program. Two constants (0xF0090000 and 0xF0080004) are the addresses of the IO devices LED and serial port. IO devices connected to Patmos are connected to the local, uncached memory area. This is the same memory area where data SPM and NoC SPM are connected. Therefore, to access them one needs to use the local load/store instructions. With the attribute `_SPM` the compiler is instructed to emit the correct load and store instructions.

Listing 1.1: A blinking LED

```
#
# The embedded version of Hello World: a blinking LED
#
# Expected Result: LED blinks
#

        .word   56

        add     r7  = r0, 0xF0090000
        addi    r8 = r0, 1

loop:   xor     r9 = r9, r8  # toggle value
        swl     [r7+0] = r9  # set the LED

        addi    r1 = r0, 1024
        sli     r1 = r1, 10

wloop:  subi    r1 = r1, 1
        cmpneq  p1 = r1, r0
(p1)    br      wloop
        addi    r0  = r0 , 0
        addi    r0  = r0 , 0
        br      loop
```

1.2.4 Make Targets

A list of the most important make targets:

tools build of all tools, including the Patmos software simulator

asm assemble source (from folder `asm`)

swsim execute the Patmos simulator

hwsim execute the Patmos emulator

emulator build the Chisel based C++ emulator

comp compile a C program as loadable ELF binary

bootcomp compile a C program as a bootable image

gen generate the Verilog code

synth synthesize for an FPGA

config configure the FPGA

download download an elf file into the main memory via the Patmos bootloader

test run all assembler tests

app compile an application that lives in `c/apps/name`

The name of an application that can execute from the on-chip ROM is set with the BOOTAPP variable.

Listing 1.2: A blinking LED

```
/*
    This is a minimal C program executed on the FPGA version of Patmos.
    An embedded Hello World program: a blinking LED.

    Additional to the blinking LED we write to the UART '0' and '1' (if available).

    Author: Martin Schoeberl
    Copyright: DTU, BSD License
*/

#include "include/bootable.h"
#include <machine/spm.h>

int main() {

    volatile _SPM int *led_ptr  = (volatile _SPM int *) 0xF0090000;
    volatile _SPM int *uart_ptr = (volatile _SPM int *) 0xF0080004;
    int i, j;

    for (;;) {
        *uart_ptr = '1';
        for (i=2000; i!=0; --i)
            for (j=2000; j!=0; --j)
                *led_ptr = 1;


        *uart_ptr = '0';
        for (i=2000; i!=0; --i)
            for (j=2000; j!=0; --j)
                *led_ptr = 0;

    }
}
```

1.2.5 Download of ELF Files

On a Linux box with the installed LLVM compiler and Quartus in your PATH, the complete build processes for the Hello World is as follows:

```
make BOOTAPP=bootable-bootloader APP=hello_puts \
   tools comp gen synth config download
```

You should see the download information and then the greeting from Patmos:

```
/home/martin/t-crest/patmos/install/bin/patserdow -v /dev/ttyUSB0 /home/martin/t-crest/patmos/tmp/hello_
Port opened: true
Params set: true
Elf version is '1':true
CPU type is:48875
Instruction width is 32 bits:true
Is Big Endian:true
File is of type exe:true
```

```
Entry point:131076

[++++++++++] 49778/49778 bytes
Hello, World!

EXIT 0
```

The `Makefile` use following variables to configure the build process: `BOOTAPP` is an application that ends in the on-chip ROM. This may be an assembler program or a simple C program; most prominent the boot loader for ELF binaries. A C program that shall be compiled as ROM target needs to be prefixed with `bootable-`. `APP` is a C program resulting in an ELF binary that can be either loaded by the emulator or the boot loader when executing in an FPGA.

When the FPGA configuration (the Patmos hardware) is stable and only the C application shall be recompiled and downloaded use a simpler make command:

```
make APP=hello_puts comp config download
```

Here an example of the individual steps to build the blinking LED C hello world (on a different FPGA board):

```
make tools
make BOOTAPP=bootable-echo bootcomp gen
make BOOTAPP=bootable-echo BOARD=bemicro synth
make BOARD=bemicro BLASTER_TYPE=Arrow-USB-Blaster config
```

This split of the make commands is for demonstration. It is possible to merge all steps into a single make (on Linux systems) or two steps when using two operating systems (e.g., Mac OS X for compilation and Windows or Linux for synthesis).

Emulator and elf File The emulator can read a standard ELF file. An example how to compile a small C program that uses part of the standard library and executing it on the emulator is as follows:

```
make emulator
make comp APP=hello_puts
patemu tmp/hello_puts.elf
```

1.2.6 A More Complex Application and the Apps Folder

Small code examples (single C files) are distributed in folder `c` and some subfolders (e.g., `bootable`, `bootloader`, `cmp`,...) and compiled with a `make comp APP=program`.

Larger applications, such as an operating system for Patmos as RTEMS,[3] live in their own repository.

Medium sized example applications shall be placed into its own folder within `c/apps` with the folders named after the applications and containing its own `Makefile` to build the application. The compile process shall result in an `.elf` named after the application name. The application can include other C code from the source tree directly, e.g., to include library code. Therefore, we can avoid building too many tiny libraries.

The main `Makefile` contains the target `app` to compile the target and copy the `.elf` file into the local build directory. Here is a trivial app example (living in `c/apps/hello`) consisting of two source files to do the "Hello World" example, including compilation, configuration, and download to Patmos:

```
make app config download APP=hello
```

A more interesting multicore example, which also uses `libcorethread` code without building a library, can be found in `c/apps/bench`.

[3] `https://github.com/t-crest/rtems`

1.2.7 Supported FPGA Boards

At the time of this writing we have mainly focused on Altera FPGA based boards. Following boards are directly supported in the build process:

- Altera DE2-115 (`altde2-115`), this is the default board for the project
- Altera DE2-70 (`altde2-70`)
- Altera/Farnell BeMicro (`bemicro`)

Setting the `BOARD` variable configures the board that shall be used. Without changing the `Makefile` the default of the board (and any other build variables) can be overridden by providing a local `config.mk` that is included in the `Makefile`.

1.2.8 Multicore Patmos

A multicore Patmos with shared external memory and the network-on-chip Argo is configured via the Aegean framework. Aegean is a collection of Python scripts that read in XML based configuration description (e.g., topology, network connections, processor types,...). Aegean generates the Chisel based components (Partmos, memory arbitration tree, and memory controller), generates the VHDL top-level to connect them with the VHDL based NoC, synthesizes the hardware, may compile the application for the individual cores, configures the FPGA, and may download the application.

The default platform can be built from within Aegean with:

```
make platform
make synth
```

The default platform is a 4-core multicore for the DE2-115 FPGA board. To use the 9 core version set `AEGEAN_PLATFORM` to `altde2-115-9core`. The FPGA is configured from within the `aegean` directory with:

```
make config
```

The compilation and download of the application is then best done within the `patmos` directory with:

```
make APP=hello_puts comp download
```

This application is the same single core *Hello World* application that we used in Section 1.2.5. However, here we just compiled the application and downloaded it via the serial port. We synthesized and configured the FPGA from within the Aegean project for the multi-core version.

TODO: We shall have here three simple hello world applications: (1) just plain shared memory 4 cores saying hello - DONE -, (2) a CMP program that uses shared memory, and (3) a simple NoC setup.

Figure 1.3 shows the embedded *Hello World* example executing code on two cores and each core blinking its own LED (Each core is connected to one of the LEDs).

Figure 1.4 shows a minimal program that uses the Argo NoC for message passing. One channel is setup from the core 0 to the core 1. Core 0 sends a message containing an integer. Core 1 receives this data and write a modified version into the shared field `field`, which is then read by core 0.

On-chip Application

For small experiments it is possible to execute a multicore program out of the on-chip memories. The Mandelbrot demo is one example.

In directory `aegean` run:

```
make platform AEGEAN_PLATFORM=mandelbrot_demo
make synth config AEGEAN_PLATFORM=mandelbrot_demo
```

Listing 1.3: A multicore blinking LED (c/cmp/cmp_example.c)

```
/*
    This is multicore version of an embedded Hello World program:
    two blinking LEDs executing on two cores.

    Author: Martin Schoeberl
    Copyright: DTU, BSD License
*/

#include <stdio.h>
#include <machine/patmos.h>

#include "libcorethread/corethread.h"

// Blink the individual LED of a core
void blink(int period) {

  volatile _IODEV int *led_ptr = (volatile _IODEV int *) PATMOS_IO_LED;
  volatile _IODEV int *us_ptr = (volatile _IODEV int *) (PATMOS_IO_TIMER+12);

  int time = period*1000/2;
  int next;

  for (;;) {
    next = *us_ptr + time;
    while (*us_ptr-next < 0) *led_ptr = 1;
    next = *us_ptr + time;
    while (*us_ptr-next < 0) *led_ptr = 0;
  }
}

// The main function for the other thread on the another core
void work(void* arg) {
  int val = *((int*)arg);

  blink(val);
}

int main() {

  printf("Hello CMP\n");
  int core_id = 1; // The core number
  static int parameter = 1000;
  corethread_create(core_id, &work, (void *) &parameter);

  blink(2000);

  // the folowing is not executed in this example
  int* res;
  corethread_join( core_id, (void *) &res );

  return 0;
}
```

Listing 1.4: Message passing on the Argo NoC

```
/*
    This is minimal demonstration how to use the Argo NoC
    with one message passing channel from core 0 to core 1.
    The value is returned via a shared variable in main memory.

    Author: Martin Schoeberl
    Copyright: DTU, BSD License
*/

#include <stdio.h>
#include <machine/patmos.h>

#include "libcorethread/corethread.h"
#include "libmp/mp.h"

#define NUM_BUF 2
#define BUF_SIZE 100

// Whatever this contant means, it is needed
const int NOC_MASTER = 0;
// Shared data in main memory for the return value
volatile _UNCACHED static int field;

// The main function for the other thread on core 1
void work(void* arg) {

  // create a channel
  qpd_t *channel = mp_create_qport(1, SINK, BUF_SIZE, NUM_BUF);
  // init
  mp_init_ports();
  // receive
  mp_recv(channel, 0);
  int data = *(volatile int _SPM *) channel->read_buf;
  mp_ack(channel, 0);

  // Return a change value in the shared variable
  field = data + 1;
}

int main() {

  printf("Hello Argo NoC\n");
  int core_id = 1; // The core number
  corethread_create(core_id, &work, NULL);

  int data = 42;
  // create a channel
  qpd_t *channel = mp_create_qport(1, SOURCE, BUF_SIZE, NUM_BUF);
  // init
  mp_init_ports();
  // write data into the send buffer
  *(volatile int _SPM *) channel->write_buf = data;
  // send the buffer
  mp_send(channel, 0);
  printf("Data sent\n");
  printf("Returned data is: %d\n", field);
  int* res;
  corethread_join(core_id, (void *) &res );
  return 0;
}
```

to generate (and synthesize) the mandelbrot application on a 4 core version of Patmos for an Altera DE2-115 FPGA board. This application is compiled into the on-chip memories and therefore executing right after configuration of the FPGA. Have a terminal open and connected to the serial port (115200 baud, 1 stop bit, no handshake) during and after the FPGA configuration and you shall see the output of the mandelbrot calculation.

However, the approach to have the application in on-chip memory works for tiny programs only. Furthermore, each software change needs a new synthesize run. A better approach is to build a platform that contains a bootloader (similar to the single core version) and some startup code to synchronize the program start with the other cores.

1.3 Worst-Case Execution Time Analysis

Patmos is currently supported by two WCET analysis tools: the AbsInt tool aiT [5] (`a3patmos`) and platin also contains a WCET analysis backend [6]. In the section we will give a minimal "Hello World" example for WCET analysis with platin. We use following simple C program, where it is not obvious if the addition or multiplication path is the WCET path.

```
#include <stdio.h>

// foo is the analysis entry point that would be inlined with -O2
int foo(int b, int val, int val2) __attribute__((noinline));
int foo(int b, int val, int val2) {

  int i;

  if (b) {
    for (i=0; i<51; ++i) {
      val = val * val2;
    }
  } else {
    for (i=0; i<73; ++i) {
      val = val + val2;
    }
  }

  return val;
}

// The compiler shall not compute the result
volatile int seed = 3;

int main(int argc, char** argv) {

  int val = seed;
  int val2 = seed+seed;
  int b = seed/4;

  int i = foo(b, val, val2);
//  printf("%d\n", i);

  return i;
}
```

To be able to analyze a program, the compiler needs to be instructed to ouput the program in .pml format during compilation:

```
patmos-clang -O2 -mserialize=simple.pml simple.c
```

The WCET analysis with platin is excuted as follows:

```
platin wcet -i simple.pml -b a.out -e foo --report
```

Those commands assume a standard configuration of Patmos that is the default single core configuration with the DE2-115 memory timing, which is for 4 32-bit bursts in 21 clock cycles for a cache line read or write. The configuration of the different caches within Patmos is as the default is in the hardware configuration.

This folder `patmos/wcet` contains this minimal example to explore WCET analysis with `platin` for Patmos. The two commands and further commands to explore the result are included in the `Makefile`.

TODO: Is the default really correct? Where does it come from?

More details on WCET analysis with `platin` can be found in Section 8.3. This section also includes a more elaborated example in Subsection 8.3.7.

1.4 Getting Started with Patmos

These exercises are intended to make you familiar with the Patmos processor and the T-CREST tool chain. This exercise assumes that you have the virtual machine (VM) image with the T-CREST tools installed and already compiled. The T-CREST project is hosted at GitHub[4] within several git repositories. You find those repositories local in your VM under directory `t-crest`.

The Patmos processor source lives in directory `patmos`, the directory where you will do most examples.

If you want to setup the T-CREST toolchain native on your Linux (Mac) system, see the build instructions in Chapter 6.

1.4.1 Hello World

We start with the standard Hello World:

```
main() {
    printf("Hello Patmos!\n");
}
```

With the Patmos compiler installed and in the PATH it can be compiled to a Patmos executable and run with the simulator as follows:

```
patmos-clang hello.c
pasim a.out
```

The Patmos distribution also contains a cycle accurate simulation of the processor, which we call emulator. You can run the program on the emulator as follows:

```
patemu a.out
```

However, this innocent examples is quiet challenging for an embedded system: It needs a C compiler, an implementation of the standard C library, printf itself is a challenging function, the generated ELF file needs to be understood by a tool and the individual sections downloaded, and finally a terminal (often a serial line) needs to be available on the target, and your test PC needs to have a serial line as well and a terminal program needs to run.

Therefore, we might start with a minimal assembler program and execute that in the simulator and emulator.

If you have not yet downloaded the handbook you can also build it on your VM:

```
cd t-crest/patmos/doc/handbook
make
```

[4] `https://github.com/t-crest`

1.4.2 Assembler Programming

All compilation and generation is based on `Makefiles`.

To prepare that all assembler tools are compiled and installed execute

```
make tools
```

in the `patmos` folder.

The assembler programs are located in subfolder `asm`. Take a look into `basic.s` and try to understand what this small program does. Assemble the example with:

```
make asm BOOTAPP=basic
```

You should now find a `basic.bin` in the `tmp` folder. This file is just a plain binary file containing the instructions for Patmos. You can display binary files with the Unix command `od` (e.g., with `od -t x1 tmp/basic.bin`). The first 32-bit word in the binary file is the length of the function, that number that was defined in the assembler file with `.word 40;`. The next word should be the first instruction. Look into the Patmos handbook and check if the encoding of the first instruction is correct.

Now execute this 'progam' on the simulator `pasim`. As there is nothing written to stdout, the simulator will not output much. Explore the options (with `-h`) to enable dumping of register contents. The simulator can also print statistics of instruction usage and caches. The assembler and the software simulator can be executed with one step with the help of the `Makefile`:

```
make swsim BOOTAPP=basic
```

The software simulator `pasim` is a C based simulator of the Patmos processor.

Patmos itself is written in `Chisel` a high level language for hardware design. Chisel is a language embedded in Scala. Therefore, you have the full power of Scala available. The Chisel code can generate `Verilog` code for the hardware synthesis and a C++ based emulator to simulate the hardware. The benefit of this Chisel based emulator is that it is exactly the same function as the hardware.

The emulator (the Chisel based simulator) can execute the same program with following command:

```
make hwsim BOOTAPP=basic
```

This command assembles the application, executes the Chisel based hardware construction during which the program is used to initialize the on-chip ROM, generates a C++ based emulator, compiles that emulator, and executes it. The emulator shows the register content after each instruction.

Those two Patmos simulations, the software simulator and the Chisel based emulator, are used for a co-simulation based test. In this co-simulation all available assembler programs are executed in both simulations and the register out put is compared.

You can watch the hardware details by dumping the wave form during the execution of the emulator. To enable waveform dumping you need to add the `-v` option for the call of the emulator in `hardware/Makefile`:

```
test: emulator
        $(HWBUILDDIR)/emulator -v -r -i -l 1000000 -O /dev/null; exit 0
```

Now rerun your example (with `make hwsim`) and change into the `hardware` folder. There you start the waveform viewer with:

```
make view
```

To watch signals they need to be dropped into the wave window. For example the program counter (`io_fedec_pc` from the `fetch` component) and some registers (`rf_1` and `rf_2` from the register file in component `decode/rf`). You should be able to see the same register changes as before, but now with an exact timing, i.e., with the delay between instruction fetch till register write in the last pipeline stage.

Optional: Tinker with the Patmos Hardware

You can find the hardware description of Patmos in `hardware/src/patmos`. Each of the 5 pipeline stages is in its own Chisel class (and file). For example, change some instructions in the `Execute` stage by manipulating `Execute.scala`. You could change the addition to a subtract operation and test it with the `basic.s` program, or your own assembler test program.

Don't forget to undo your changes for the next exercises. The Patmos repository is a `git` repository. Therefore, undo is easily done with:

```
git checkout Execute.scala
```

1.4.3 I/O Programming

Hello World in Assembler

To communicate with the external world, Patmos contains a UART (or serial line) as a minimal I/O interface. In the real hardware that UART is then connected to the PC for text output and for program download as well. In the simulator the UART output is just echoed to stdout of the host.

The I/O devices are memory mapped, which means they can be accessed with load and store instructions. However, Patmos has typed load and store instructions. Therefore, I/O devices are also mapped into a type. In our case I/O devices are mapped into the local memory areas. Therefore, use `swl` as instruction, like:

```
swl [r7+0] = r9;
```

This above instruction writes the content of register `r9` into a data location at address of register `r7`. Find the address of the UART device in the handbook and write a single character (e.g., '*') to it. The UART is described in the Memory and I/O Subsystem chapter. You can find a short I/O example in `asm/hello.s`.

Optional: The Real Hello World

Transmission of characters takes some time and the processor needs to wait till the next character can be sent. Waiting can be done with a busy loop polling the status register of the UART (the Transmit ready bit).

Embedded Hello World in C

Embedded systems are often built bare-bone, that means without an operating system and maybe even without a standard library. In this example you shall write a the Hello World example without using `printf`. That means you access the UART with load and store instructions, like you did in the assembler example. Remember, the I/O devices are mapped into local memory space. The Patmos compiler needs to be informed that we do want to access local memory. This is performed with the help of a little macro:

```
#include <machine/spm.h>

int main() {

        volatile _SPM int *uart_status = (volatile _SPM int *) 0xF0080000;
        volatile _SPM int *uart_data = (volatile _SPM int *) 0xF0080004;
```

Emulator and elf File The emulator can read a standard ELF file. Therefore, we use the prebuilt emulator of Patmos and compile only C programs. A barebone C program (e.g., `myhello.c` placed in folder `c`) for the emulator (and the hardware) is compiled with:

```
make comp APP=myhello
```

We execute this .elf program with the emulator:

```
patemu tmp/myhello.elf
```

or with `pasim`.

Now start similar to the assembler based Hello World and write a short program to write a single character to the UART.

As a next step write out a longer string of characters. However, transmission of characters takes some time and the processor needs to wait till the next character can be sent. Waiting can be done with a busy loop polling the status register of the UART (the Transmit ready bit).

1.4.4 Periodic Tasks

Real-time tasks are usually periodic tasks. Therefore, we will program a small example that uses the Patmos time to execute periodic tasks. First we start with polling of the timer/counter to generate periodic event. Write out a character about every second. For this polling use the timer counter and wait until some time elapsed. As we run in a simulation, time elapses way slower. Therefore, start with short waiting times and increase with error and retry.

With this example you can explore the simulation time difference between the SW simulator `pasim` and the hardware generated emulator. Which one is faster? And by how much?

Optional: Periodic Task as Interrupt Handler

Polling consumes computing resources and is only a solution for single tasks. Better is to use a time interrupt and an interrupt handler for the periodic task. Reprogram the above example as a timer interrupt handler. You can find an example for interrupt handlers in `c/intrs.c`.

Having the timer interrupt under control is almost half of a scheduler for a real-time operating system!

1.4.5 Adding an IO Device to Patmos

If you are curious on building hardware in Chisel you can add your own IO device to Patmos. As a simple first step build an IO device that does not contain any connection to the external world, but act as an accelerator device for Patmos. E.g., build an IO device that has two registers, which can be written by software, an adder (or multiplier) that adds those two values, and a readable register that contains the result.

You can find instructions how to add an IO device to Patmos in Section 3.7.

1.4.6 Further Steps

After this exercise you master the T-CREST tool flow for the Patmos processor. Next step is to get an FPGA board, such as the Altera DE2-115, and see the processor executing in real hardware. From this on you can proceed to extend the processor with your own ideas, explore the multicore version of Patmos with the real-time network on chip Argo, write your own operating systems, do WCET analysis with `aiT` and/or `platin`, ...

Contributions are always welcome and easy to do with a GitHub pull request. You can ask questions to the Patmos community via the Patmos mailing list. See: `http://patmos.compute.dtu.dk/`.

2 The Architecture of Patmos

2.1 Pipeline

Figure 2.1 shows an overview of Patmos' pipeline. The pipeline consist of 5 stages: (1) instruction fetch (FE), (2) decode and register read (DEC), (3) execute (EX), (4) memory access (MEM), and (5) register write back (WB).

Some instructions define additional pipeline stages. Multiplication instructions are executed, starting from the EX stage, in a parallel pipeline with fixed-length (see the instruction definition). The respective stages are referred to by $\mathtt{EX}_1, \ldots, \mathtt{EX}_n$.

2.1.1 Fetch

Fetch one or two words of instruction from the ROM or instruction cache. Calculate next PC depending on the length of the instruction bundle.

2.1.2 Decode

Decode the instruction and generate control signals for the following stages. Read register operands. Sign or zero extend immediate operands.

2.1.3 Execute

Read predicate registers. Conditional execute (ALU) instructions. Write predicate register. Calculate effective address for memory operations.

2.1.4 Memory

Read or write memory. This is the only pipeline stage that might stall the pipeline.

2.1.5 Write Back

Write result into destination register.

2.2 Local Memories

Patmos contains several on-chip memories, as sketched in Figure 2.1. We apply the idea of split caches [12] to simplify and enhance the cache analysis. Instructions are fetched from the instruction cache. Patmos also supports instruction and data scratchpad memories. Stack allocated data is cached in a stack cache the other data in the data cache with LRU replacement. Accesses to data that are hard to analyze can bypass the data cache.

2.3 Register Files

The register files available in Patmos are depicted by Figure 2.2. In short, Patmos offers:

- 32, 32-bit general-purpose registers (R) : r0, ..., r31
  r0 is read-only, set to zero (0).

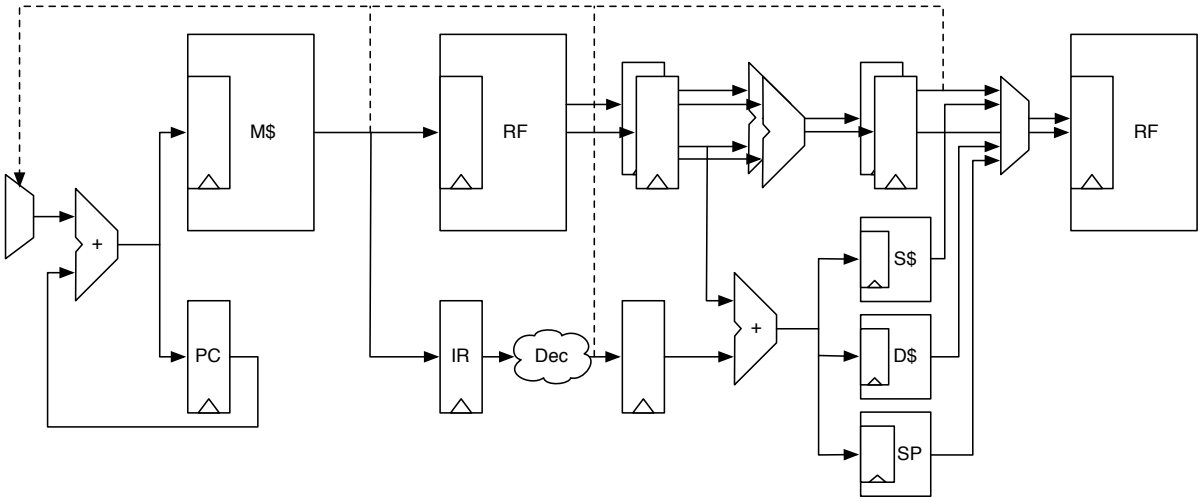


Figure 2.1: Pipeline of Patmos with fetch, decode, execute, memory, and write back stages.

- 8, single-bit predicate registers (`P`): `p0`, ..., `p7`,
  `p0` is read-only, set to `true` (`1`).
- 16, 32-bit special-purpose registers (`S`): `s0`, ..., `s15`

The general-purpose registers `R` are read in the `DEC` stage and written in the `WB` stage. Full forwarding makes them available in the `EX` stage before written into the register file. The predicate registers are single bits that are set and read in the `EX` stage. The special registers `S` is just a collection of various "special" processor registers (e.g., stack cache pointers). These registers might be used by different units/stages in the pipeline and are not physically collected in a "register file". The pipeline stage where those registers are read and written by the `mfs` and `mts` are dependent on the type of the special register. So all-in-all the recoverable process state is: general-purpose registers `R`, the predicates `P`, and a collection of various processor registers mapped to the "special" register file `S`.

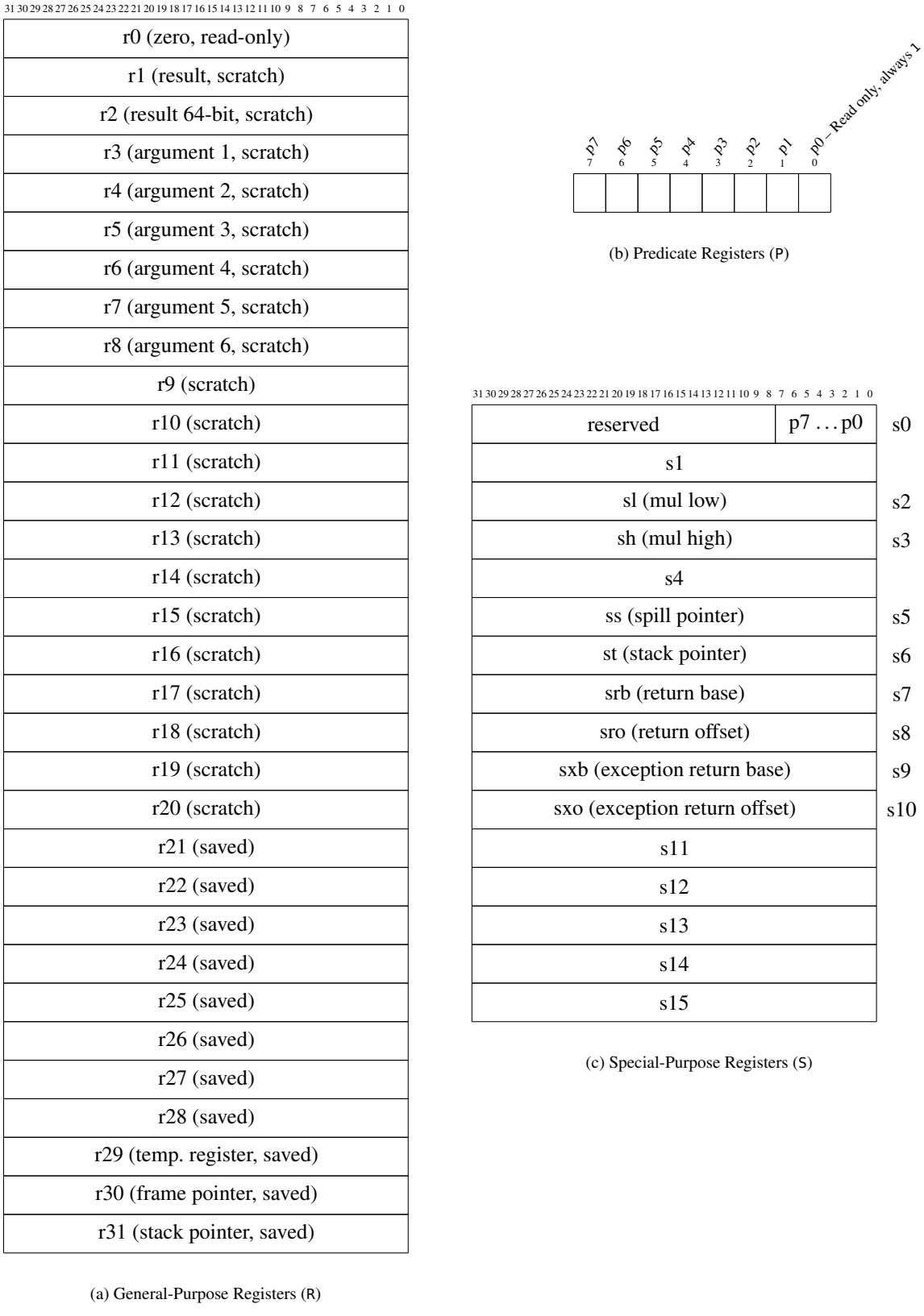
Concurrently writing and reading the same register in the same cycle will, for the read, yield the new value of the register (the register file provides internal forwarding). Reads in subsequent cycles return the result most recently written to the register, i.e., the pipeline implements full forwarding.

When writing concurrently to the same register, the result is undefined. If two instructions of the current bundle have the same destination register, the result is only defined if the predicate of at most one instruction in the bundle evaluates to `true` (1).

The predicate registers are usually encoded as 4-bit operands, where the most significant bit indicates that the value read from the register file should be inverted before it is used. For operands that are written, this additional bit is omitted.

The special-purpose registers of `S` allow access to some dedicated registers:

- The lower 8 bits of `s0` can be used to save/restore *all* predicate registers at once. The other bits of that register are currently reserved, but not used. Setting the reserved bits has no effect.
- `s2` and `s3` can also be accessed through the names `sl` and `sh` and represent the lower and upper 32-bits a multiplication.
- `s5` can also be accessed through the name `ss` and represents the register pointing to the top of the saved stack content in the main memory (i.e., the current stack spill pointer). Updating `s5` does not change `s6` or spill the stack cache.
- `s6` can also be accessed through the name `st` and represents a pointer to the top-most element of the content of the stack cache. Updating `s6` does not change `s5` or spill the stack cache.



(a) General-Purpose Registers (R)

(b) Predicate Registers (P)

(c) Special-Purpose Registers (S)

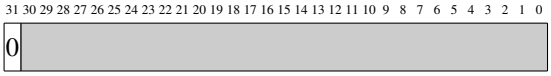
Figure 2.2: General-purpose register file, predicate registers, and special-purpose registers of Patmos.

2.4 Bundle Formats

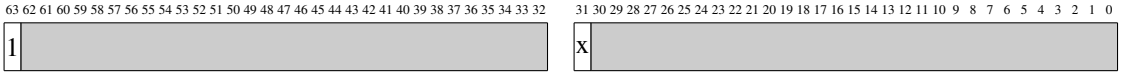
All Patmos instructions are 32 bits wide and are structured according to one of the instruction formats defined in the following section. Up to two instructions can be combined to form an instruction bundle; Patmos bundles are thus either 32 or 64 bits wide. The bundles sizes are recognized by the value of the most significant bit, where 0 indicates a short, 32-bit bundle and 1 a long, 64-bit bundle.

The following figures illustrate these two bundle variants:

- 32-bit bundle format



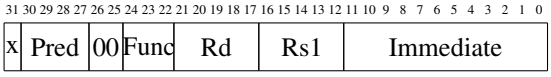
- 64-bit bundle format



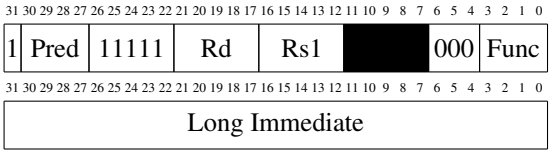
2.5 Instruction Formats

This section gives an overview of all instruction formats defined in the Patmos ISA. Individual instructions of the various formats are defined in the next section. Gray fields indicate bits whose function is determined by a sub-class of the instruction format. Black fields are not used.

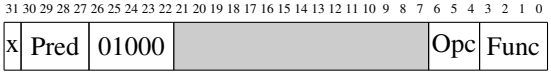
- AluImm – Arithmetic Immediate (ALUi)



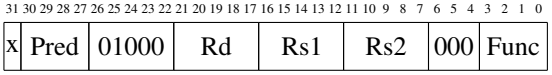
- AluLongImm – Long Immediate (ALUl)



- Alu – Arithmetic (ALU)

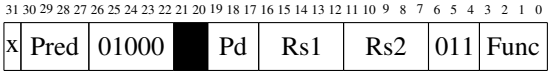


AluReg – Register (ALUr)

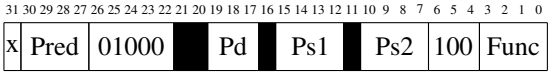


ALUm – Multiply

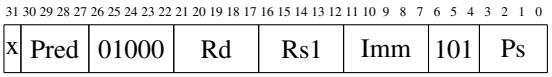
ALUc – Compare



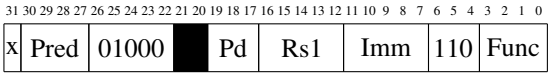
ALUp – Predicate



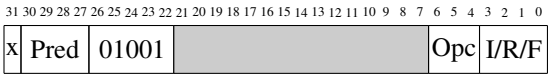
ALUb – Bitcopy



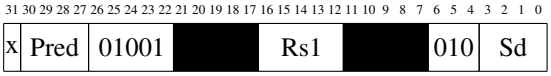
ALUci – Compare immediate



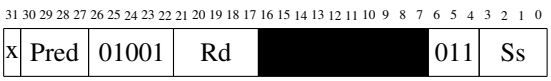
- SPC – Special



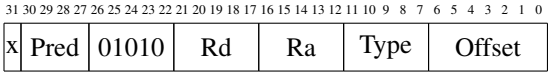
SPCt – Move To Special



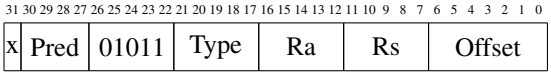
SPCf – Move From Special



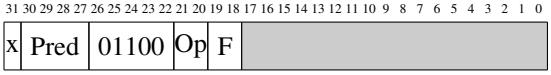
- LDT – Load Typed



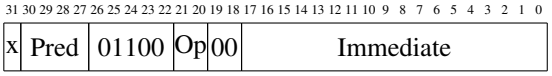
- STT – Store Typed



- STC – Stack Control

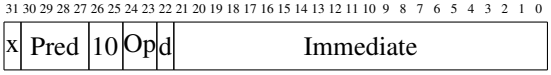


STCi – Stack Control Immediate

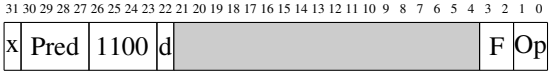


STCr – Stack Control Register

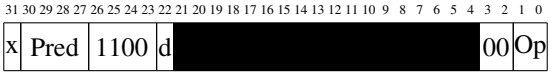
- CFLi – Control Flow with Immediate



- CFLr – Control Flow with Registers



CFLri – Control Flow with implicit registers



CFLrs – Control Flow with single register

CFLrt – Control Flow with two registers

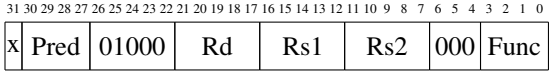
2.6 Instruction Opcodes

This section defines the instruction set architecture, the instruction opcodes, and the behavior of the respective instructions of Patmos.

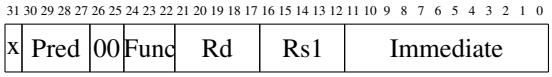
2.6.1 Binary Arithmetic

Applies to the AluReg, AluImm, and AluLongImm formats. Operand `Op2` denotes either the `Rs2`, or the `Immediate` operand, or the `Long Immediate`. The immediate operand is zero-extended. For shift and rotate operations, only the lower 5 bits of the operand are considered. Table 2.1 shows the encoding of the `func` field; for AluImm instructions, only functions in the upper half of that table are available.

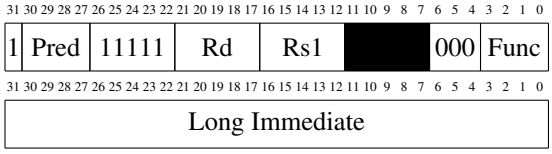
- AluReg – Register (ALUr)



- AluImm – Arithmetic Immediate (ALUi)



- AluLongImm – Long Immediate (ALUl)



Func	Name	Semantics
0000	add	`Rd = Rs1 + Op2`
0001	sub	`Rd = Rs1 - Op2`
0010	xor	`Rd = Rs1 ^ Op2`
0011	sl	`Rd = Rs1 << Op2`$_{(4:0)}$
0100	sr	`Rd = Rs1 >>> Op2`$_{(4:0)}$
0101	sra	`Rd = Rs1 >> Op2`$_{(4:0)}$
0110	or	`Rd = Rs1 | Op2`
0111	and	`Rd = Rs1 & Op2`
1000	—	unused
1001	—	unused
1010	—	unused
1011	nor	`Rd = ~(Rs1 | Op2)`
1100	shadd	`Rd = (Rs1 << 1) + Op2`
1101	shadd2	`Rd = (Rs1 << 2) + Op2`
1110	—	unused
1111	—	unused

Table 2.1: General ALU functions

Pseudo Instructions

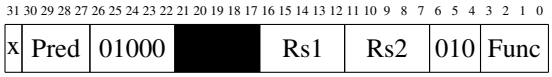
- `mov Rd = Rs` ... `add Rd = Rs + 0`
- `clr Rd` ... `add Rd = r0 + 0`
- `neg Rd = -Rs` ... `sub Rd = 0 - Rs`
- `not Rd = ~Rs` ... `nor Rd = ~(Rs | R0)`
- `li Rd = Immediate` ... `add Rd = r0 + Immediate`
- `li Rd = Immediate` ... `sub Rd = r0 - Immediate`
- `nop` ... `sub r0 = r0 - 0`

Note The use of `sub r0 = r0 - 0` to encode a `nop` pseudo-instruction results in a value of `0x00400000` in the binary instruction stream. This helps in distinguishing the execution of compiler-generated `nops` from executing instructions from memory that happens to be zero.

2.6.2 Multiply

Applies to the ALUm format only. Multiplications are executed in parallel with the regular pipeline and finish within a fixed number of cycles *TODO: how many?*. Table 2.2 shows the encoding of the `func` field for the ALUm instruction format.

- ALUm – Multiply



Func	Name	Semantics
0000	mul	`sl = Rs1 * Rs2;` `sh = (Rs1 * Rs2) >>> 32`
0001	mulu	`sl = (uint32_t)Rs1 * (uint32_t)Rs2;` `sh = ((uint32_t)Rs1 * (uint32_t)Rs2) >>> 32`
0010	—	unused
...	...	...
1111	—	unused

Table 2.2: Multiplication functions

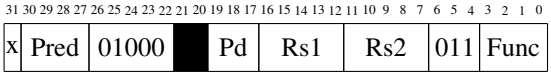
Behavior Perform multiplication in multiple cycles and write the result into destination registers `sl` and `sh`.

Note Multiplications are pipelined, it is thus possible to issue one multiplication on every clock cycle. Multiplications can only be issued in the first slot.

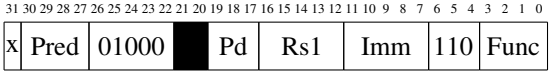
2.6.3 Compare

Applies to the ALUc and ALUci formats only. Operand `Op2` denotes either the `Rs2`, or the `Imm` operand. Tables 2.3 show the encoding of the `func` field for the ALUc and ALUci formats.

- ALUc – Compare



- ALUci – Compare immediate



Func	Name	Semantics
0000	cmpeq	`Pd = Rs1 == Op2`
0001	cmpneq	`Pd = Rs1 != Op2`
0010	cmplt	`Pd = Rs1 < Op2`
0011	cmple	`Pd = Rs1 <= Op2`
0100	cmpult	`Pd = Rs1 < Op2, unsigned`
0101	cmpule	`Pd = Rs1 <= Op2, unsigned`
0110	btest	`Pd = (Rs1 & (1 << Op2)) != 0`
0111	—	unused
...	...	...
1111	—	unused

Table 2.3: Compare functions

Pseudo Instructions

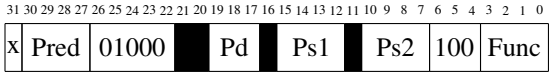
- `isodd Pd = Rs1` ... `btest Pd = Rs1[r0]`
- `mov Pd = Rs` ... `cmpneq Pd = Rs != r0`

Note The predicate register is read and written in the execute stage.

2.6.4 Predicate

Applies to the ALUp format only, the opcodes correspond to those of the ALU operations on general purpose registers. Table 2.4 shows the encoding of the `func` field for the ALUp format.

- ALUp – Predicate



Func	Name	Semantics
0000	—	unused
...	...	...
0101	—	unused
0110	por	`Pd = Ps1 | Ps2`
0111	pand	`Pd = Ps1 & Ps2`
1000	—	unused
1001	—	unused
1010	pxor	`Pd = Ps1 ^ Ps2`
1011	—	unused
...	...	...
1111	—	unused

Table 2.4: Predicate functions

Pseudo Instructions

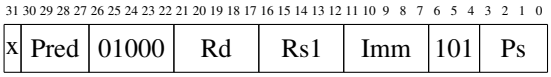
- `pmov Pd = Ps` ... `por Pd = Ps | Ps`
- `pnot Pd = ~Ps` ... `pxor Pd = (Ps ^ p0)`
- `pset Pd = 1` ... `por Pd = p0 | p0`
- `pclr Pd = 0` ... `pxor Pd = p0 ^ p0`

Note The predicate register is read and written in the execute stage. All predicate combine instruction mnemonics (including pseudo instructions) are prefixed with `p`, all other instructions involving predicates are not prefixed (e.g., moving from register to predicate).

2.6.5 Bitcopy

Applies to the ALUb format only. The only instruction with this encoding is `bcopy`, which is the "inverse" of the `btest` instruction. It has the following semantics: `Rd = (Rs1 & ~(1 << Imm)) | (Ps << Imm)`
Note that operand `Ps`, like the source operands in the ALUp format, can be inverted.

- ALUb – Bitcopy



Pseudo Instructions

- `mov Rd = Ps` ... `bcopy Rd = r0, 0, Ps`

Note The predicate register is read in the execute stage.

2.6.6 Move To Special

Applies to the SPCt format only. Copy the value of a general-purpose register to a special-purpose register. The only instruction is `mts`, which stores the content of general-purpose register `Rs1` in special register `Sd`.

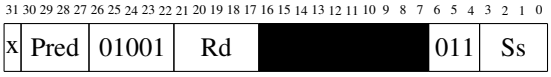
- SPCt – Move To Special

Note

2.6.7 Move From Special

Applies to the SPCf format only. Copy the value of a special-purpose register to a general-purpose register. The only instruction is `mfs`, which loads the content of special register `Ss` to general-purpose register `Rd`.

- SPCf – Move From Special



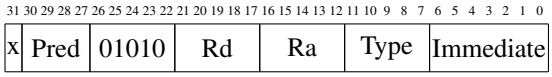
2.6.8 Load Typed

Applies to the LDT format only. Load from a memory or cache. In the table accesses to the stack cache are denoted by `sc`, to the local scratchpad memory by `lm`, to the date cache by `dc`, and to the global shared memory by `gm`. All load variants are considered to stall until the memory access is completed.

Loads incur a load-to-use latency that has to be respected by the compiler/programmer. The result of the load is not available in the bundle immediately after the load, i.e., there must be one bundle between the load instruction and the first use of the destination register. The value of the destination register is undefined during this load delay slot.

The displacement value (`Imm`) value is interpreted unsigned.

- LDT – Load Typed



Type	Name	Semantics
000 | 00	lws	`Rd=sc[Ra+(Imm << 2)]`$_{32}$
000 | 01	lwl	`Rd=lm[Ra+(Imm << 2)]`$_{32}$
000 | 10	lwc	`Rd=dc[Ra+(Imm << 2)]`$_{32}$
000 | 11	lwm	`Rd=gm[Ra+(Imm << 2)]`$_{32}$
001 | 00	lhs	`Rd=(int32_t)sc[Ra+(Imm << 1)]`$_{16}$
001 | 01	lhl	`Rd=(int32_t)lm[Ra+(Imm << 1)]`$_{16}$
001 | 10	lhc	`Rd=(int32_t)dc[Ra+(Imm << 1)]`$_{16}$
001 | 11	lhm	`Rd=(int32_t)gm[Ra+(Imm << 1)]`$_{16}$
010 | 00	lbs	`Rd=(int32_t)sc[Ra+Imm]`$_{8}$
010 | 01	lbl	`Rd=(int32_t)lm[Ra+Imm]`$_{8}$
010 | 10	lbc	`Rd=(int32_t)dc[Ra+Imm]`$_{8}$
010 | 11	lbm	`Rd=(int32_t)gm[Ra+Imm]`$_{8}$
011 | 00	lhus	`Rd=(uint32_t)sc[Ra+(Imm << 1)]`$_{16}$
011 | 01	lhul	`Rd=(uint32_t)lm[Ra+(Imm << 1)]`$_{16}$
011 | 10	lhuc	`Rd=(uint32_t)dc[Ra+(Imm << 1)]`$_{16}$
011 | 11	lhum	`Rd=(uint32_t)gm[Ra+(Imm << 1)]`$_{16}$
100 | 00	lbus	`Rd=(uint32_t)sc[Ra+Imm]`$_{8}$
100 | 01	lbul	`Rd=(uint32_t)lm[Ra+Imm]`$_{8}$
100 | 10	lbuc	`Rd=(uint32_t)dc[Ra+Imm]`$_{8}$
100 | 11	lbum	`Rd=(uint32_t)gm[Ra+Imm]`$_{8}$
101 | 00	—	unused
...	...	...
111 | 11	—	unused

Table 2.5: Typed loads

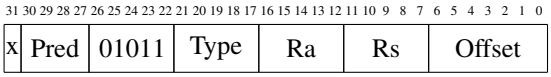
Note All loads can only be issued on the first slot.

2.6.9 Store Typed

Applies to the STT format only. Store to a memory or cache. In the table accesses to the stack cache are denoted by `sc`, to the local scratchpad memory by `lm`, to the date cache by `dc`, and to the global shared memory by `gm`.

The displacement value (`Imm`) value is interpreted unsigned. Stores can only be issued on the first slot.

- STT – Store Typed



Type	Name	Semantics
000 | 00	sws	$\texttt{sc[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
000 | 01	swl	$\texttt{lm[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
000 | 10	swc	$\texttt{dc[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
000 | 11	swm	$\texttt{gm[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
001 | 00	shs	$\texttt{sc[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
001 | 01	shl	$\texttt{lm[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
001 | 10	shc	$\texttt{dc[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
001 | 11	shm	$\texttt{gm[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
010 | 00	sbs	$\texttt{sc[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
010 | 01	sbl	$\texttt{lm[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
010 | 10	sbc	$\texttt{dc[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
010 | 11	sbm	$\texttt{gm[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
01100	—	unused
...	...	...
11111	—	unused

Table 2.6: Typed stores

Note - Global Memory / Data Cache With regard the data cache, stores are performed using a *write-through* strategy without *write-allocation*. Data that is not available in the cache will not be loaded by stores; but will be updated if it is available in the cache.

Consistency between loads and other stores is assumed to be guaranteed by the memory interface, i.e., memory accesses are handled in-order with respect to a specific processor. This has implications on the bus, the network-on-chip, and the global memory.

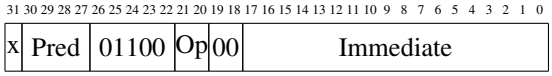
2.6.10 Stack Control

Applies to the STC format only. Manipulate the stack frame in the stack cache. `sres` reserves space on the stack, potentially spilling other stack frames to main memory. `sens` ensures that a stack frame is entirely loaded to the stack cache, or otherwise refills the stack cache as needed. `sfree` frees space on the stack frame (without any other side effect, i.e., no spill/fill is executed). `sspill` writes the tail of the stack cache to main memory and updates the spill pointer.

All immediate stack control operations are carried out assuming word size, i.e., the immediate operand is multiplied by four. All register operands and stack pointer addresses in special registers are in units of bytes.

A more detailed description of the stack cache is given in Section 3.3. Table 2.7 shows the encoding of operations for STCi, while Table 2.8 shows the encoding for STCr.

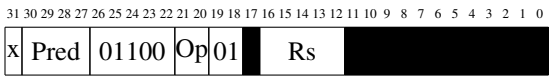
- STCi – Stack Control Immediate



Op	Name	Semantics
00	sres	Reserve space on the stack (with spill)
01	sens	Ensure stack space (with refill)
10	sfree	Free stack space.
11	sspill	Spill tail of the stack cache to memory

Table 2.7: Stack control operations with immediates

- STCr – Stack Control Register



Op	Name	Semantics
00	—	unused
01	sens	Ensure stack space (with refill)
10	—	unused
11	sspill	Spill tail of the stack cache to memory

Table 2.8: Stack control operations for registers

Behavior `sres`: Check free space left in the stack cache. Update stack-cache registers. If needed, spill to global memory using `ss`.

`sense`: Check reserved space available in the stack cache. If needed, refill from global memory using `ss`.

`sfree`: Account for $\texttt{head} - \texttt{tail} < 0$, update `ss` and `st`. Update stack-cache register `head`.

`sspill`: Update `ss` and `st`. Update stack-cache register `tail`. Spill to global memory using `ss`.

Note Stack control instructions can only be issued on the first position within a bundle.

It is permissible to use several reserve, ensure, and free operations within the same function.

2.6.11 Control-Flow Instructions

Applies to CFLi and CFLr format only. Transfer control to another function or perform function-local branches. `br` performs a function-local branch. `call` performs a function call, storing the return information (i.e., where to resume execution when returning) in `srb`/`sro`. `brcf` ("branch with cache fill") performs a global branch. With regard to addressing modes and caching, `brcf` behaves like a call, but it does not store any return information. `trap` performs a system call (see Section 2.7). `ret` returns from a function, using the return information in `srb`/`sro`. `xret` is similar to `ret` but uses the return information in `sxb`/`sxo`, which are set by interrupts, exceptions, and traps.

With a method cache, `call`, `brfc`, `trap`, `ret`, and `xret` may cause a cache miss and a subsequent cache refill to load the target code; they expect the size of the code block fetched to the cache in number of bytes at <*base*>*-4*. `br` is assumed to be a cache hit.

Immediate call and branch instructions interpret the operand as *unsigned* for `call` and `brcf`, and as *signed* for PC-relative branches (`br`). These immediate values are interpreted in *word size*. The target address of PC-relative branches is computed relative to the address of the branch instruction. The immediate value for `trap` is an index into an exception vector (see Section 2.7).

Indirect call and branch instructions interpret the operand as *unsigned* absolute addresses in *byte size*. Indirect `brcf` takes two operands: a base address and an offset. The base address is the address of the code block to be fetched; the effective branch target is <*base*>+<*offset*>.

The return information provided by `call` in `srb`/`sro` should only be passed to `ret`. Likewise, the exception return information in `sxb`/`sxo` should only be passed to `xret`. The unit and addressing mode of these values is implementation dependent.

All control-flow instructions (except `trap`) have a delayed and a non-delayed variant. For the delayed variant, *N* bundles following the control-flow instruction in the code are always executed. For the non-delayed variants, the control-flow change appears to happen immediately. However, non-delayed control-flow instructions may require more than one cycle to be executed. The precise timing behavior is specified along with the detailed description of the respective instructions.

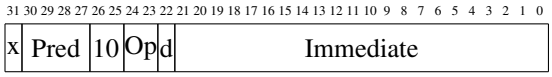
The mnemonic of an instruction's non-delayed instruction variant is suffixed with `nd`. For clarity, this document uses the unsuffixed name to mean both variants when describing the general properties of the respective instruction.

`br` instructions are executed in the `EX` stage, while the other control-flow instructions are executed in the `MEM` stage. This corresponds to a branch delay of 2 bundles for `br` and 3 bundles for other control-flow instructions. In case there no other instructions available, NOP instructions can be used to fill the delay slots. There are no restrictions with regard to the size or type of instructions in the delay slot. The only exception is that executing control-flow instructions in a delay slot may lead to unspecified behavior. Interrupts occurring during the delay slot are delayed until the branch has executed.

Instruction	Immediate	Indirect	Cache fill	Link	Delay Slots
call	absolute, words	absolute, bytes	yes	yes	3
br	PC relative, words	absolute, bytes	no	no	2
brcf	absolute, words	absolute+offset, bytes	yes	no	3
trap	exception vector index	—	yes	no	–
ret	—	implementation dependent	yes	no	3
xret	—	implementation dependent	yes	no	3

Table 2.9: Addressing modes of control-flow instructions

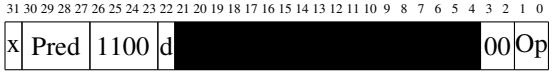
- CFLi – Control Flow with Immediate



Op	d	Name	Semantics
00	0	callnd	function call (absolute, with cache fill, non-delayed)
00	1	call	delayed function call (absolute, with cache fill, delayed)
01	0	brnd	local branch (PC-relative, always hit, non-delayed)
01	1	br	local branch (PC-relative, always hit, delayed)
10	0	brcfnd	branch (absolute, with cache fill, non-delayed)
10	1	brcf	branch (absolute, with cache fill, delayed)
11	0	trap	system call (via exception vector, with cache fill, non-delayed)

Table 2.10: Control-flow operations with immediate

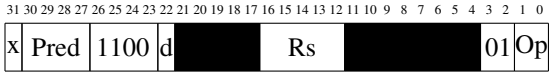
- CFLri – Control Flow with implicit registers



Op	d	Name	Semantics
00	0	retnd	return (with cache fill, non-delayed)
00	1	ret	return (with cache fill, delayed)
01	0	xretnd	return from exception (with cache fill, non-delayed)
01	1	xret	return from exception (with cache fill, delayed)

Table 2.11: Control-flow operations with implicit register operands

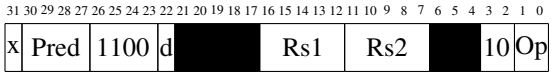
- CFLrs – Control Flow with single registers



Op	d	Name	Semantics
00	0	callnd	function call (indirect, with cache fill, non-delayed)
00	1	call	function call (indirect, with cache fill, delayed)
01	0	brnd	local branch (indirect, always hit, non-delayed)
01	1	br	local branch (indirect, always hit, delayed)

Table 2.12: Control-flow operations with single register operand

- CFLrt – Control Flow with two registers



Op	d	Name	Semantics
10	0	brcfnd	branch (indirect with offset, with cache fill, non-delayed)
10	1	brcf	branch (indirect with offset, with cache fill, delayed)

Table 2.13: Control-flow operations with two register operands

Behavior – call Perform a function call, filling the method cache if needed.

Listing 2.1 shows the pseudo-code for a call. The parameter `addr` is either an immediate value (for calls in the CFLi format), or comes from a general-purpose register (for indirect calls). The variables `$srb` and `$sro` denote the special registers `srb` and `sro`, respectively.

First, it stores the return information, and remembers the new base address in an internal variable. Then, it retrieves the offset into the cache for the function to be called and if necessary copies the instructions into the cache. Finally, it updates the internal program counter and continues execution from there.

Listing 2.1: Call

```
call(addr) {
  // Store return information
  $srb = base;
  $sro = PC;
  // Remember base address
  base = addr;
  // Cache look-up and load
  coff = offset(addr);
  if (!hit(addr)) memcpy(cache[coff], mem[base], mem[base-4]);
  // Update PC
  PC = coff;
}
```

The timing of a `callnd` instruction that is executed is equivalent to the timing of an istruction sequence `call; nop; nop; nop`. The timing of a `callnd` instruction with a predicate that evaluates to `false` is equivalent to a single `nop` instruction.

Behavior – brcf Perform a branch, filling the method cache if needed.

The pseudo-code for a PC-relative `brcf` is shown in Listing 2.2. It is similar to the call, but does not store any return information. Additionally, an offset `off` may be specified for a `brcf` in the CFLrt format; this offset is 0 for `brcf` in the CFLi format.

Listing 2.2: Branch with cache fill

```
call(addr, off) {
  // Remember base address
  base = addr;
  // Cache look-up and load
  coff = offset(addr);
  if (!hit(addr)) memcpy(cache[coff], mem[base], mem[base-4]);
  // Update PC
  PC = coff+off;
}
```

The timing of a `brcfnd` instruction that is executed is equivalent to the timing of an istruction sequence `brcf; nop; nop; nop`. The timing of a `brcfnd` instruction with a predicate that evaluates to `false` is equivalent to a single `nop` instruction.

Behavior – trap Perform a system call. Details are described in Section 2.7. Like the `call`, `trap` stores return information, but it uses special registers `sxb` and `sxo` for that purpose.

Behavior – ret, xret Return from function call. The `ret` instruction uses the return information in the special registers `srb`/`sro` to compute its target address. The `xret` instruction uses the registers `sxb`/`sxo`.

Listing 2.3 shows the pseudo-code for `ret`. It first retrieves the return base and does the appropriate cache handling. It then adds the cache offset and the return offset and assigns the sum to the internal program counter, from which execution continues.

Listing 2.3: Return

```
ret() {
  // Retrieve return base
  base = $srb;
  // Cache look-up and load
  coff = offset($srb);
  if (!hit($srb)) memcpy(cache[coff], mem[base], mem[base-4]);
  // Update PC
  PC = coff+$sro;
}
```

The timing of a `retnd`/`xretnd` instruction that is executed is equivalent to the timing of an istruction sequence `ret; nop; nop; nop`. The timing of a `retnd`/`xretnd` instruction with a predicate that evaluates to `false` is equivalent to a single `nop` instruction.

Behavior – br Local branch, compute new program counter value and update program counter.

The timing of a `brnd` instruction may be implementation defined. Implementations are allowed to use branch prediction for `brnd` instructions, if the underlying mechanism is documented in detail. Unless specified and documented otherwise, the timing of `brnd` must be according to "predict-not-taken". The timing of a `brnd` instruction that is executed is then equivalent to the timing of the sequence `br; nop; nop`, while the timing with a `false` predicate is equivalent to a single `nop`.

Note All control-flow instructions can only be issued on the first position within a bundle.

2.6.12 Instruction List

TODO: List all Patmos instructions with a usage example

Following table lists all Patmos instructions with an example usage

Instruction	Semantics
addi r1 = r0, 255	Add 255 to register r1

Table 2.14: Patmos instructions with examples

Can be found in `instructions.txt`:

add sub xor sl sr sra or and nor shadd shadd2 addi subi xori sli sri srai ori andi nori shaddi shadd2i mul mulu cmpeq cmpneq cmplt cmple cmpult cmule btest por pand pxor bcopy mts mfs lws lwl lwc lwm lhs lhl lhc lhm lbs lbl lbc lbm lhus lhul lhuc lhum lbus lbul lbuc lbum sws swl swc swm shs shl shc shm sbs sbl sbc sbm sres sens sfree sspill call br brcf trap ret xret callnd brnd brcfnd retnd xretnd

2.7 Exceptions: Interrupts, Faults and Traps

In the following, we use *exception* to denote any kind of "abnormal" transfer of control. *Interrupts* are generated outside of the pipeline by I/O devices. *Faults* are triggered by the pipeline for instructions that cannot be executed as expected (accesses to unmapped memory, undecodable instructions, etc.). *Traps* are willfully generated exceptions, and are used to invoke operating system functions.

An *exception unit* that is mapped to the I/O space (see Chapter 3) is responsible for managing exceptions. It includes the device registers shown in Table 2.15. The device registers of the exception unit are writable only when the processor is in privileged mode. The general principle of operation is that the exception unit requests the execution of an exception from the pipeline, and the pipeline returns an acknowledges when it starts the execution of the respective exception handler.

The `status` register is 32 bits wide; bit 0 of that register determines whether interrupts are enabled. They are enabled if it is one, and they are disabled when it is zero. Bit 1 of the `status` register indicates if the processor is in privileged mode (when the bit is one) or in user mode (when the bit is zero). The `status` register is shifted left by two bits when an exception handler is triggered, and shifted right by two bits when returning from an exception handler via `xret`. Triggering an exception handler enables privileged mode and disables interrupts, such that the lowest to bits of the `status` register are `10` at the beginning of an exception handler. Privileged mode is also enabled after reset. Overflows of the status register (which might be caused by nested exception handlers) are silently ignored.

Internally generated exceptions, i.e., faults and traps, always take precedence over interrupts. If more than one internal exception or interrupt is pending at the same time, the exception with the lower number takes precedence.

2.7.1 Exception Vector

The exception unit supports 32 exception vector entries, shown in the lower half of Table 2.15. Exceptions 0 and 1 are reserved for the "illegal operation" and "illegal memory access" faults. While exceptions 2 to 15 can be used freely (e.g., by the operating system), exceptions 16 to 32 are attached to interrupts.

2.7.2 Traps

The instruction `trap <n>` triggers exception number n. In order to assimilate the handling of faults and traps in the pipeline, `trap` instructions never have a delay slot. Traps can in principle be called for any exception, e.g., to trigger an interrupt handler from software.

2.7.3 Return Information

The return information for exceptions must be stored in registers that are never used otherwise. Two special registers `sxb` (`s9`) and `sxo` (`s10`) provide the return base and return offset for exceptions. The instruction `xret` implicitly uses these registers, as opposed to the `ret` instruction, which uses special registers `srb` and `sro`.

2.7.4 Resuming Execution

The return information for interrupts is set such that `xret` returns to the bundle that was replaced by the interrupt instruction in the pipeline.

For faults, the return information points to the bundle that triggered the fault. After a fault, resuming execution must either have fixed the cause of the fault and reexecute the whole bundle again, or emulate the effects of the whole bundle (including the triggering of further faults) and continue execution after the bundle. Due to the complexity of the second option, we consider faults where the respective instruction cannot be reexecuted *fatal*, and advise developers to terminate execution instead of trying to resume.

Resuming after a trap in principle has to take into account the same considerations as faults. However, the content of the bundle is under the control of the compiler. We require that a trap instruction is the only instruction in a bundle. The return address for traps points to the bundle after the one that contains the trap instruction.

Listing 2.4: Exception handler registration

```
for (unsigned i = 0; i < 32; i++) {
      exc_register(i, &fault_handler);
}
exc_register(8, &trap_handler);
exc_register(16, &intr_handler);
exc_register(17, &intr_handler);
exc_register(18, &intr_handler);
exc_register(19, &intr_handler);
```

2.7.5 Delayed Triggering of Interrupts

Instructions that stall the pipeline (loads, stores, calls, etc.) delay the triggering of interrupts until the pipeline resumes execution. Therefore, method cache fills or stack spills cannot be interrupted. Control-flow instructions delay the triggering of interrupts such that interrupts are never triggered inside a delay slot or while executing instructions speculatively. Multiplications delay the triggering of interrupts such that no multiplications are "in flight" when an interrupt handler is entered. Outstanding delayed loads do not delay the triggering of interrupts. Interrupt handlers must ensure that the contents of the special register `sm` are saved and restored correctly.

2.7.6 Sleep Mode

The exception unit provides support for putting the processor to sleep. Writes to the `sleep` register halt the pipeline until an interrupt (or other exception) occurs. After executing the respective exception handler, execution resumes after that write. Therefore, writes to the `sleep` register should be enclosed in a loop for continuous sleeping.

Support for sleeping is optional. On implementations that do not support sleeping, writes to the `sleep` register are ignored and do not have any effect. Therefore, continuing execution after a write to the `sleep` register is not proof that an interrupt has occurred.

2.7.7 Cache Control

The `cachectrl` register provides an interface for cache control. Writing a value with bit 0 set invalidates the contents of the data cache. Writing a value with bit 1 set invalidates the contents of the instruction cache.

Local Mode

Writing a value with bit 31 set to the `cachectrl` register changes between the normal mode of operation and a special *local mode*. In the local mode, cached memory accesses (`lwc`, `swc`, ...), are redirected to the local address space instead of the data cache. The local mode is used during booting and is not intended for use in normal applications.

2.7.8 Examples

The API for exception and interrupt handling for Patmos is provided by the include file `machine/exceptions.h`.

Listing 2.4 shows how to register exception handlers for specific exceptions. First, `fault_handler` is registered as handler for all exceptions. Then, `trap_handler` is registered for exception number eight, and `intr_handler` for exceptions 16 to 19, i.e., for interrupts 0 to 3.

Listing 2.5 shows how to enable interrupts at the start of an application. First, all interrupts are unmasked. Then, all pending flags are cleared to avoid triggering any "stale" interrupts. Finally, interrupts are enabled; after that point, Patmos will call the respective interrupt handler when an interrupt occurs.

To unmask only certain interrupts, `machine/exceptions.h` provides a function `intr_unmask`, which takes an interrupt number as parameter. Similarly, `intr_clear_pending` can be used to clear only a particular pending

Listing 2.5: Interrupt enabling

```
  // unmask interrupts
  intr_unmask_all();
  // clear pending flags
  intr_clear_all_pending();
  // enable interrupts
  intr_enable();
```

Listing 2.6: Fault handler example

```
void fault_handler(void) {
  unsigned source = exc_get_source();
  LEDS = source;

  const char *msg = "FAULT";
  switch(source) {
  case 0: msg = "Illegal operation"; break;
  case 1: msg = "Illegal memory access"; break;
  }
  puts(msg);

  // cannot recover from a fault
  abort();
}
```

Listing 2.7: Interrupt handler example

```
void intr_handler(void) {
  exc_prologue();

  LEDS += exc_get_source() & 0xf;

  exc_epilogue();
}
```

flag. Masking interrupts can be done through the `intr_mask_all` and `intr_mask` functions. Disabling interrupt handling in general is done with the `intr_disable` function.

Listing 2.6 shows a basic fault handler. As Patmos cannot recover from faults, the handler does not use a special prologue, and calls `abort` at the end instead of returning. In the function body, the handler displays the exception source on the leds and prints a message corresponding to the type of fault that occurred.

Listing 2.7 shows a minimal interrupt handler. It uses the macros `exc_prologue` and `exc_epilogue` to save and restore the processor state. The actual functionality is that the state of the LEDs is incremented according to the exception source.

Address	Name	Description
0xf0010000	status	Interrupt-enable flag
0xf0010004	mask	Mask of enabled interrupts
0xf0010008	pend	Pending flags for interrupts
0xf001000c	source	Number of exception that is about to be served
0xf0010010	sleep	Sleep mode (optional)
0xf0010014	cachectrl	Cache control
0xf0010080	vec<0>	Address of exception handler 0, illegal operation
0xf0010084	vec<1>	Address of exception handler 1, illegal memory access
...	...	...
0xf00100c0	vec<16>	Address of exception handler 16, interrupt 0
...	...	...
0xf00100fc	vec<31>	Address of exception handler 31, interrupt 15

Table 2.15: Exception unit device registers

2.8 Dual Issue Instructions

Not all instructions can be executed in both pipelines. In general, the first pipeline implements all instructions, the second pipeline only a subset. All memory operations are only executed in the first pipeline.

What other instructions can be executed in both pipelines is still open for discussion and evaluation with benchmarks. A minimal approach, as first step for the hardware implementation, is to have only ALU instructions available in the second pipelines (excluding predicate manipulation instructions).

2.9 Assembly Format

A VLIW instruction consists of one or two operations that are issued in the first or both pipelines. Each operation is predicated, the predicate register is specified before the operation in parentheses `()`. If the predicate register is prefixed by a `!`, its negation is considered. If omitted, it defaults to `(p0)`, i.e. always true.

A semi-colon `;` or a newline denotes the end of an instruction or operation. If an instruction contains two operations, the operations in the bundle must be enclosed by curly brackets. Bundles do not need to be separated by newlines or semi-colons. For bundles consisting of only one operation, the curly brackets are optional. Labels that are prefixed by `.L` are local labels.

All register names must be prefixed by `$`. We use destination before source in the instructions, between destination and source a = character must be used instead of a comma. Immediate values are not prefixed for decimal notation, the usual 0 and 0x formats are accepted for octal and hexadecimal immediates. Comments start with the hash symbol # and are considered to the end of the line. For memory operations, the syntax is `[$register + offset]`. Register or offset can be omitted, in that case the zero register `r0` or an offset of 0 is used.

Example:

```
       # add 42 to contents of r2
       # and store result in r1 (first slot)
       { add   $r1 = $r2, $42
       # if r3 equals 50, set p1 to true
       cmpeq $p1, $r3, 50 }
       # if p1 is true, jump to label_1
 ($p1) br label_1 ; nop; nop # then wait 2 cycles
       # Load the address of a symbol into r2
       li $r2 = .L.str2
       # perform a memory store and a pred op
       { swc [$r31 + 2] = $r3 ; or $p1 = !$p2, $p3 }
       ...
label_1:
       ...
```

2.9.1 Instruction Mnemonics

The LLVM assembler supports the instructions mnemonics as specified in this document, including all pseudo instructions.

The `paasm` assembler and the `pasim` simulator use the same basic instruction mnemonic, but a `i` or `l` suffix is appended for *immediate* and *long immediate* variants, while no suffix in general refers to the register indirect variant of the instructions. As exception, the control flow instructions use a `r` suffix for the register indirect variants and no suffix for the immediate instructions.

2.9.2 Inline Assembly

Inline assembly syntax is similar to GCC inline assembly. It uses `%0`, `%1`, ... as placeholders for operands. Accepted register constraints are: `r` or `R` for any general purpose register, or `{<registername>}` to use a specific register.

Example:

Parameter	Default setting
Main memory	2 MB
Burst length	4 words
Dual issue	true for uniprocessor, false for multi-processor and compiler setting
Instruction cache	4 KB, 16 blocks
Data cache	2 KB, direct mapped, write through
Stack cache	2 KB

Table 2.16: Default settings for the Patmos hardware, the emulator, the simulator, and the compiler.

```
int i, j, k;
asm("mov  $r31 = %1  # copy i into r31\n\t"
    "add  %0 = $r5, %2"
    : "=r" (j)
    : "r" (i), "{r10}" (k));
```

2.10 Configuration and Default Setup

Various parameters of the Patmos processor can be configured to trade space for performance. Furthermore, IO devices and memory controllers are usually specific to FPGA boards. Those configurations are specified in XML files and can be found at `patmos/hardware/config`. The base configuration is defined in `default.xml`. Board specific configurations, e.g., `altde2-115.xml` for the default FPGA board DE2-115 from Altera, are specified in individual XML files.

Tabl 2.16 lists the default settings for the configuration of Patmos and the tools. *TODO: Compiler and simulator settings settings are not yet updated!*

3 Memory and I/O Subsystem

3.1 Local and Global Address Space

The typed loads of Patmos imply two address spaces: a local address space that is accessed through local loads and stores, and a global address space that is accessed when using other access types. All caches use memory that is mapped to the global address space as backing memory. For example, the data cache fetches data from global memory on a cache miss, and the stack cache uses global memory for spilling and filling. Consequently, there are two memory maps, one for the local address space and one for the global address space. Tables 3.1 and 3.2 show the respective address mappings. To simplify address decoding, the top four bits (A31–A28) are generally used to distinguish between different memory and I/O areas. The address range for I/O devices is divided further to distinguish the different devices, as discussed in Section 3.2.

As `call`, `ret`, and `brcf` do not include memory type information, the distinction between memory areas for these instructions is done solely through the address mapping. The boot instruction ROM and the instruction scratchpad memory are mapped to the lowest 128K of the global address space. Note that this applies only to these instructions; i.e., a `call` to address `0x00010100` executes code that is located in the instruction scratchpad, while non-local loads or stores to the same address access the external SRAM. Therefore, a binary that is loaded to external memory can use the lowest 128K of memory for data segments, but not for code segments.

As mentioned in Section 2.7, Patmos supports a special *local mode*, in which cached memory accesses (`lwc`, `swc`, ...) are redirected to the local address space. This local mode is however only used during booting and not intended for use in regular applications.

3.2 I/O Devices

Each processor contains a minimum set of standard I/O devices: a device to read configuration information, an interrupt controller, and a timer. These three devices are always present; all other devices are optional. For minimum communication with the outside world, a processor is typically attached to a serial port (UART) that represents `stdout`.

Within the I/O device memory area, bits 19–16 are used to distinguish between different devices. Most I/O device registers are mapped and aligned to 32-bit words. If a register is shorter than a word, the upper bits shall be filled with 0 on a read. With this mapping, each I/O device can have up to 16384 32-bit registers. I/O devices may support sub-word accesses, but must document that fact if they do.

The I/O device base addresses for `pasim` are defined in `patmos/simulator/include/memory-map.h`. For the library, the constants are defined in `newlib/libgloss/patmos/patmos.h` and `newlib/newlib/libc/machine/patmos/machine/*.h`. The offsets of the I/O devices in the hardware are defined in the configuration XML files in `patmos/hardware/config/*.xml`

Address	Memory area
`0x00000000–0x0000ffff`	Data Scratchpad Memory
`0x00010000–0x0001ffff`	Instruction Scratchpad Memory (write only)
`0xe0000000–0xe7ffffff`	NoC interface configuration registers
`0xe8000000–0xefffffff`	NoC communication memory
`0xf0000000–0xffffffff`	I/O devices

Table 3.1: Address mapping for local address space

Address	Memory area
0x00000000–0x0000ffff	Boot Instruction ROM (only for code)
0x00010000–0x0001ffff	Instruction Scratchpad Memory (only for code)
0x00000000–0x7fffffff	External SRAM

Table 3.2: Address mapping for global address space

In the default configuration of Patmos there are six I/O devices: a device to read configuration information, an interrupt controller, a timer, a UART, a LEDs device, and a Keys device. Table 3.3 shows the I/O devices and the registers.

3.2.1 CpuInfo

The CpuInfo device holds information about various aspects of the processor configuration. Table 3.4 describes the available device registers. Additionally, the CpuInfo device contains a ROM that contains the data for the application in the boot instruction ROM. This boot data ROM starts at offset 0x8000 within the CpuInfo device; the actual size of the boot data ROM depends on the boot application. All data in the CpuInfo device are read-only.

External Memory Configuration

Bits 15-8 of the ExtMem_Conf device register contain the length of bursts for transactions to the external memory. Bits 7-0 of the ExtMem_Conf device register shall be 1 if writes to external memory may be combined, and 0 otherwise.

Instruction Cache Configuration

Bits 31-24 of the ICache_Conf device register encode the type of the instruction cache. The value shall be 1 for a method cache, 2 for a traditional instruction cache, and 0 otherwise. Bits 23-16 encode the replacement policy for the instruction cache. The value shall be 1 for LRU replacement, 2 for FIFO replacement, and 0 otherwise. Bits 15-0 encode the associativity of the instruction cache. For a method cache, the associativity corresponds to the number of blocks.

Data Cache Configuration

Bits 31-24 of the DCache_Conf device register shall be 0 for a write-back data cache and 1 for a write-through data cache. Bits 23-16 encode the replacement policy for the data cache. The value shall be 1 for LRU replacement, 2 for FIFO replacement, and 0 otherwise. Bits 15-0 encode the associativity of the data cache.

Stack Cache Configuration

There are currently no stack cache features to encode and the respective device register always reads 0.

Boot Data Initialization Information

By convention, the first four words of the boot data ROM contain information about data to be copied to the boot data scratchpad before starting actual execution. Table 3.5 shows the respective data fields. Upon start, the program should copy src_size bytes of data from src_start to dst_start. If dst_size is greater than src_size, the remaining bytes are filled with zeroes.

3.2.2 Timer

The timer device provides a means to measure time as well as to trigger an interrupt at a certain point in time. It provides two 64-bit counters. While the first counter is incremented every clock cycle, the second counter is incremented every microsecond.

Address	I/O Device	read	write
0xf0000000	CpuInfo		
...	CpuInfo	CpuInfo device information registers (see Section 3.2.1)	
0xf0007fff	CpuInfo		
0xf0008000	CpuInfo	Boot Data ROM	–
...	CpuInfo	...	–
0xf0008fff	CpuInfo	Boot Data ROM	–
0xf0010000	ExcUnit		
...	ExcUnit	Exception unit and cache control (see Section 2.7)	
0xf00100ff	ExcUnit		
0xf0020000	Timer	clock cycles (high word)	cycle interrupt time (high word)
0xf0020004	Timer	clock cycles (low word)	cycle interrupt time (low word)
0xf0020008	Timer	time in μs (high word)	μs interrupt time (high word)
0xf002000c	Timer	time in μs (low word)	μs interrupt time (low word)
0xf0030000	Deadline	may stall the pipeline	deadline cycles
0xf0080000	UART	status	control
0xf0080004	UART	receive buffer	transmit buffer
0xf0090000	LED	–	output register
0xf00a0000	Keys	input register	–
0xf00c0000	Audio		
...	Audio	Audio interface (tbd)	
0xf00cffff	Audio		
0xf00d0000	EthMac		
...	EthMac	EthMac device (see Section 3.2.5)	
0xf00dffff	EthMac		
0xf00e0000	UART2 (TTL)	status	control
0xf00e0004	UART2 (TTL)	receive buffer	transmit buffer

Table 3.3: I/O devices and registers

Address	Name	Description
0xf0000000	CoreID	core ID
0xf0000004	Freq	clock frequency
0xf0000008	CoreCnt	number of cores in multicore system
0xf000000c	Features	processor features (number of pipelines)
0xf0000010	ExtMem_Size	maximum external memory size
0xf0000014	ExtMem_Conf	external memory configuration
0xf0000018	ICache_Size	instruction cache size
0xf000001c	ICache_Conf	instruction cache features
0xf0000020	DCache_Size	data cache size
0xf0000024	DCache_Conf	data cache features
0xf0000028	SCache_Size	stack cache size
0xf000002c	SCache_Conf	stack cache features
0xf0000030	ISPM_Size	instruction scratchpad size
0xf0000034	DSPM_Size	data scratchpad size

Table 3.4: CpuInfo device registers

Address	Name	Description
0xf0018000	src_start	Start address of data to be copied
0xf0018004	src_size	Size of data to be copied
0xf0018008	dst_start	Destination for copying
0xf001800c	dst_size	Size of initialized data

Table 3.5: Boot data initialization information

Bit	Status		Control	
0	TRE	TX Transmit ready	–	–
1	DAV	RX Data available	–	–

Table 3.6: UART status bits

Interrupts can be triggered by storing a value in the "cycle interrupt time" and "μs interrupt time" registers. The timer device will then trigger an interrupt when the respective counter reaches the value provided in that register. The "cycle" interrupt is tied to interrupt 0; the "μs" interrupt is tied to interrupt 1.

To read out the 64-bit counter values consistently, the low word (at the higher address) must be read first. This latches the high word of the counter into an internal register, which is then returned when reading the high word (at the lower address). Similarly, the low word of the interrupt times must be written first. The write to the internal 64-bit register takes effect when the high word is written. For short measurements, where the lower word is usually large enough (i.e., up to 2000 s measurements with the μs counter), the upper word can be ignored.

3.2.3 UART

The UART is a minimal IO device for `stdout` and `stdin`. It can also used for program download. Table 3.6 shows the bits of the control register. Writing to the data register adds the respective value to the transmission queue of the UART. Reading the data register removes a previously received byte from the reception queue of the UART.

3.2.4 Deadline

The deadline device provides a facility for delaying execution for a specified amount of time. In each subsequent cycle, the counter is decremented by 1. A load to the address stalls the processor pipeline until the counter reaches 0.

3.2.5 EthMac

Patmos supports Ethernet connections through an EthMac device [8, 9]. This device is usually mapped starting at address `0xf00d0000`. The RX/TX buffer of the device is mapped to offsets `0x0000` to `0xefff`, while the device registers of the Ethernet controller are mapped to offsets `0xf000` to `0xffff`. For detailed information on the EthMac device, please refer to the cited reports.

3.2.6 Memory Management Unit

Patmos can be configured to include a memory management unit (MMU). The configuration port of this MMU is then mapped starting at address `0xf0070000`. The MMU uses segmentation for memory protection and address translation; it supports eight segments. Table 3.7 shows the mapping of segment information.

The bit 31 of a configuration register (e.g., Seg0_Conf) is set if the respective segment is readable, bit 30 is set if the segment is writable, and bit 29 is set if the segment is executable. Bits 28-0 of the configuration register contain the length of the respective segment.

The MMU enforces memory protection and translates virtual addresses to physical addresses. The segment to be used is encoded in the three most significant bits of the virtual address (i.e., bits 31-29). Bits 28-0 of the

Address	Name	Description
0xf0070000	Seg0_Base	Base address of segment 0
0xf0070004	Seg0_Conf	Configuration register of segment 0
...	...	...
0xf0070038	Seg7_Base	Base address of segment 7
0xf007003c	Seg7_Conf	Configuration register of segment 7

Table 3.7: Memory management unit device registers

virtual address represent the offset into the segment. If this offset is less than the length of the segment and if the respective permission bit is set, the access proceeds. Otherwise, an illegal memory access exception is raised. Memory protection is only enforced in user mode, i.e., the segment length and permission check is bypassed in privileged mode. The physical address is computed by adding the offset to the base address of the respective segment (e.g. Seg0_Base).

3.3 Stack Cache

The stack cache is a processor-local, on-chip memory [1]. The stack cache operates similar to a ring buffer. It can be seen as a stack-cache-sized window into the main memory address range. To manage the stack cache, we use three additional instructions: `reserve`, `ensure`, and `free`. Two hardware registers define which part of the stack area is currently in the stack cache.

3.3.1 Stack Cache Manipulation

We present the mechanics of the stack cache in C code for easier readability. However, the hardware implementation is a synchronous design and the algorithm is implemented by a state machine that handles the memory spill and fill operations. In the C code following data structures are used:

`mem` is an array representing the main memory,

`sc` is an array representing the stack cache,

`m_top` is the register pointing to the top of the saved stack content in the main memory, and

`sc_top` points to the top element in the stack cache.

The two pointers are full-length address registers. However, when addressing the stack cache, only the lower n bits are used for a stack cache of a size of 2^n words. The constant `SC_SIZE` represents the stack cache size and `SC_MASK` is the bit mask for the stack cache addressing. The stack cache is managed in 32-bit words.

At program start the stack cache is empty and both pointers, `m_top` and `sc_top`, point to the same address, the address that one higher as the stack area. `m_top` points to the last spilled word in main memory. Similar, `sc_top` points to the last slot in the stack frame (top of stack). Therefore, the number of currently valid elements in the stack cache is `m_top - sc_top`.

The compiler generates code to grow the stack downward, as it is common for many architectures. Growing the stack downwards has historical reasons. However, for multi-threaded systems each thread needs a reserved, fixed memory area for the stack and there is no benefit from growing the stack downwards.

Reserve The `reserve` instruction, as shown in Figure 3.1, reserves space in the stack cache. Typed load and store instructions use this reserved space. The reserve instruction may spill data to the main memory. This spilling happens when there are not enough free words in the stack cache to reserve the requested space.

The processor reads the number of words to be reserved (the immediate operand of the instruction) in the decode stage. The processor adjusts the `sc_top` register in the execution stage and also computes how many words need to be spilled in the execution stage. The processor spills to the main memory in the memory stage, as shown by the for loop in Figure 3.1.

```
void reserve(int n) {

    int nspill, i;

    sc_top -= n;
    nspill = m_top - sc_top - SC_SIZE;
    for (i=0; i<nspill; ++i) {
       --m_top;
       mem[m_top] = sc[m_top & SC_MASK];
    }
}
```

Figure 3.1: The `reserve` instruction provides `n` free words in the stack cache. It may spill data into main memory.

```
void free(int n) {

    sc_top += n;
    if (sc_top > m_top) {
        m_top = sc_top;
    }
}
```

Figure 3.2: The `free` instruction drops `n` elements from the stack cache. It may change the top memory pointer `m_top`.

Free The `free` instruction frees the reserved space on the stack. It does not fill previously spilled data back into the stack cache. It just changes the top of the stack pointer and may change the top of the memory pointer, as shown in Figure 3.2.

Ensure Returning into a function needs to ensure that the stack frame of this function is available in the stack cache. The `ensure` instruction, as shown in Figure 3.3, guarantees this condition. This instruction may need to fill back the stack cache with previously spilled data. This happens when the number of valid words in the stack cache is less than the number of words that need to be in the stack cache. Filling the stack cache is shown in the loop in Figure 3.3.

One processor register serves as stack pointer and points to the end of the stack frame. Load and store instructions use displacement addressing relative to this stack pointer to access the stack cache.

As with regular ring buffers, when the size of the stack cache is not sufficient in order to reserve additional space requested, it needs to spill some data so far kept in the stack cache to the global memory, i.e., whenever `m_top` - `sc_top` $>$ *stack cache size*. A major difference, however, is that freeing space does *not* imply the reloading of data from the global memory. When a free operation frees all stack space currently held in the cache (or more), the special register `ss` is accordingly incremented.

The stack cache is organized in blocks of fixed size, e.g. 32 bytes. All spill and fill operations are performed on the block level, while reserve, free and ensure operations are in words.

Addresses for load and store operations from/to the stack cache are relative to the `sc_top` pointer.

The base address for fill and spill operations of the stack cache is kept in special register `ss`. `st` contains the address the top of the stack cache would get if the stack cache would be fully spilled to memory.

The organization of the stack cache implies some limitations:

- The maximum size of stack data accessible at any moment is limited to the size of the cache. The stack frame can be split, such that at any moment only a subset of the entire stack frame has to be resident in the stack cache, or a *shadow* stack frame in global memory can be allocated.

```
void ensure(int n) {

    int nfill, i;

    nfill = n - (m_top - sc_top);
    for (i=0; i<nfill; ++i) {
        sc[m_top & SC_MASK] = mem[m_top];
        ++m_top;
    }
}
```

Figure 3.3: The `ensure` instruction ensures that at least `n` elements are valid in the stack cache. It may need to fill data from main memory.

```
// load one word from the stack cache
// addr is a main memory address (register value plus offset)

int load(int addr) {
    return sc[(sc_top + addr) & SC_MASK];
}

// store one word into the stack cache
// addr is a plain main memory address

void store(int addr, int val) {
    sc[(st_top + addr) & SC_MASK] = val;
}
```

Figure 3.4: Pseudo code for the load and store instructions.

- When passing pointers to data on the stack cache to other functions it has to be ensured that: (1) the data will be available in the cache, (2) the pointer is only used with load and store operations of the stack cache, and (3) the relative displacement due to reserve and free operations on the stack is known. Alternatively, aliased stack data can be kept on a *shadow* stack in the global memory without restrictions.
- The stack control operations only allow allocating constant-sized junks. Computed array sizes (C 90) and `alloca` with a computed allocation size have to be realized using a *shadow* stack in global memory.
- The calling conventions for functions having a large number of arguments have to be adapted to account for the limitation of the stack cache size (see Section 4).

3.4 Instruction Cache

Patmos supports two alternative solutions for the caching of instructions; it can be configured to use either a traditional instruction cache or a *method cache*. The latter caches contiguous sequences of code (typically whole functions) and is described in Section 3.4.1. The traditional instruction cache is described in Section 3.4.2.

3.4.1 Method Cache

The method cache caches contiguous sequences of code. These code sequences will often correspond to entire functions. However, functions can be split into smaller chunks in order to reduce to overhead of loading the entire

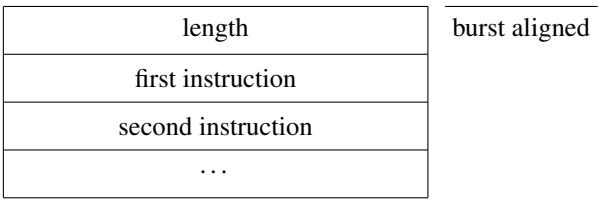


Figure 3.5: Layout of code sequences intended to be cached in the method cache.

function at once. Code transfers between the respective junks of the original function can be performed using the `brcf` instruction. A code sequence is either kept entirely in the method cache, or is entirely purged from the cache. It is not possible to keep code sequences partially in the cache.

With a method cache, cache misses may occur only upon calls, returns, or `brcf` instructions. All other instructions are guaranteed cache hits. Furthermore, cache misses occur at the same point in the processor pipeline as data cache misses, in the memory stage. Therefore, the method cache eliminates interferences between instruction cache misses and data cache misses. Consequently, a method cache can lead to more predictable behavior than a traditional instruction cache.

Code intended for caching should be aligned in the global memory according to the memory burst size. Call and branch instructions do not encode the size of the target code sequence. The size is thus encoded in units of bytes right in front of the first instruction of a code sequence that is intended for caching. Figure 3.5 illustrates this convention.

The method cache in Patmos is fully associative and implements a FIFO replacement policy. The cache tags and the storage for memory blocks are managed separately, i.e., there is no fixed mapping between tags and locations in the cache memory. Code sequences are evicted when the respective tag entry has to be reused or when the associated cache memory is overwritten. The code sequences that are loaded into the cache are internally aligned to 64-bit (8 byte) boundaries, such that a small amount of padding may occur inside the cache.

3.4.2 Traditional Instruction Cache

Patmos also supports a traditional instruction cache that caches fixed blocks rather than variably sized sequences of code. A direct-mapped traditional instruction cache is available as hardware implementation, and further variants are available in the simulator. By default, the size of an instruction cache line equals the length of a memory burst.

3.5 Data Cache

In addition to the stack cache, Patmos also contains a data cache, which is responsible for caching regular cached memory accesses such as `lwc`. As a write-through policy is beneficial for predictability, the data cache of Patmos uses this policy by default. The hardware implementation provides the options of a direct-mapped data cache and a two-way set associative cache with LRU replacement. Furthermore, a hardware implementation of a direct-mapped data cache with a write-back policy is provided. However, due to cache coherence issues, the write-back policy should not be used in multicore configurations. In all implementations the size of data cache lines equals the length of a memory burst.

The multicore versions of Patmos do not implement any cache coherence protocol in hardware. However, with a write-through caching policy, it is sufficient to invalidate the data cache through the cache control register in the exception unit (see Section 2.7.7) to ensure that subsequent memory accesses will see the most recent values. Cache coherence can therefore be implemented in software whenever communication via shared memory is required.

3.6 Hardware Interface

For the connection of Patmos to a memory controller, I/O devices, the core-to-core network on chip, and/or the memory arbiter an interface standard needs to be specified. Several standards are available. We decided to base the

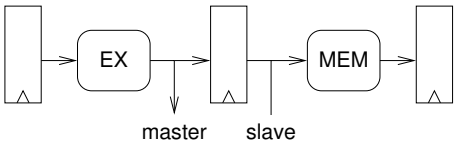


Figure 3.6: Localization of OCP signals in the pipeline

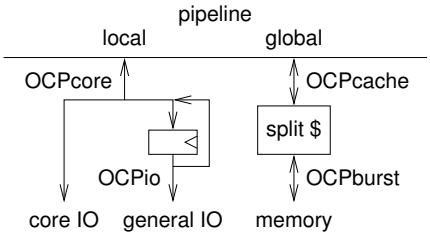


Figure 3.7: OCP levels in Patmos

interface on OCP[1] [2] and subset the standard as we need it. While we use OCP as basis for the hardware interface of Patmos, we expressly do not claim compliance with the OCP specification.

Figure 3.6 shows the OCP signals in the Patmos pipeline. The master signals are generated in the execute stage, and the slave signals are captured in the memory stage.[2] The different variants of the OCP protocol (`OCPcore`, `OCPcache`, `OCPio`, and `OCPburst`) in the scope of the Patmos processor are shown in Figure 3.7.

3.6.1 OCPcore

This is the simplest variant of the OCP protocol used in Patmos and shall be used for IO devices. The variant of OCP is generated by the pipeline for the local address space. `OCPcore` is tailored to accesses to on-chip memories, which on FPGAs necessarily include an registers on the input ports (read and write address, write data, and write enable). Furthermore, `OCPcore` is the interface for general IO devices. The respective signals are shown in Table 3.8. To enable sub-word transfers of data, the signal `MByteEn` is used.

`OCPcore` is a simple protocol with single cycle command and reply. A read (`RD`) or write (`WR`) command is valid for a single cycle. A data response for a read includes the reponse `DVA` and is expect earliest in the following cycle. Writes also need to be acknowledged with a `DVA` response (earliest in the following cycle). The slave can delay the response for arbitrary cycles. The following assumptions apply:

- Only reads (`RD`) and writes (`WR`) are supported. Writes require a response (`writeresp_enable=1`), such that every command must be followed by a response.
- No `SCmdAccept` or `MRespAccept`, flow control is done solely via `SResp`.
- Slaves may generate responses earliest in the cycle after a command.
- The master may issue commands in the same cycle as the slave sends its response, i.e., basic support for pipelining is required.
- `MByteEn` is assumed to be properly aligned (`force_aligned=1`). The signal can be ignored for read accesses without side effects.

Figure 3.8 shows a sequence read/write/read in `OCPcore`, where the slave responds to the first read in the following cycle (the earliest possibility). The write response is delayed by one cycle. The second read response is also delayed by an additional cycle.

A: The master issues a read by setting `MCmd` to `RD`, `MAddr` to A_1 and `MByteEn` to E_1.

[1] The OCP specification is available at `http://accellera.org/downloads/standards/ocp/`

[2] For clarity, the handling of both parts is implemented in the file `Memory.scala`.

Table 3.8: OCPcore signals

Signal	Description
MCmd	Command (RD or WR)
MAddr	Address, byte-based, lowest two bits always 0
MData	Data for writes, 32 bits
MByteEn	Byte enables for sub-word accesses, 4 bits
SResp	Response (NULL or DVA)
SData	Data for reads, 32 bits

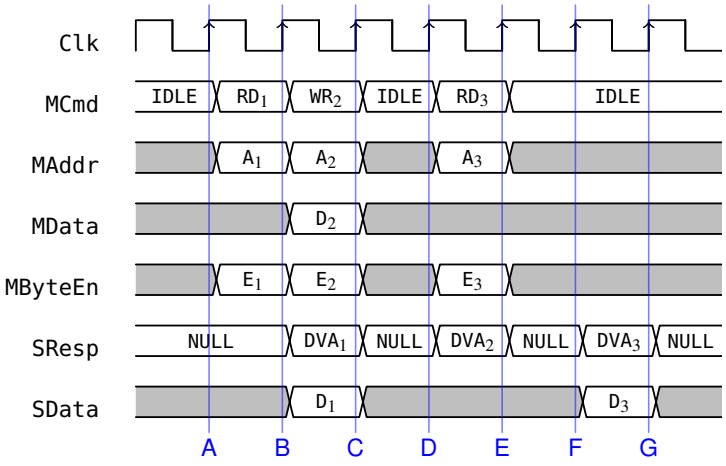


Figure 3.8: Timing diagram for OCPcore

B: The slave responds to the read issued in cycle A by setting SResp to DVA and returning the appropriate data. The master can issue the next command in the same cycle as it receives the response and issues a write command WR. The master provides the byte enable value E_2 along with the data D_2 to specify which bytes should be actually written.

C: The slave does not respond immediately and the master is stalled. MCmd must be IDLE while the master is stalled.

D: The slave responds to the write issued in cycle B. The master issues a read in the same cycle.

E: The slave does not respond to the read request, delaying it by one cycle.

F: The slave responds to the read request issued in cycle D.

G: The interface is idle.

3.6.2 OCPcache

This OCP variant is generated for the global address space and is used for communication between the pipeline and the caches. It is the same as OCPcore, but includes an additional signal MAddrSpace to specify the cache that should serve the access.

3.6.3 OCPio

The OCPio level is derived from the OCPcore level by inserting a register in the master signals. OCPio contains additional signals and all signals are shown in Table 3.9. This interface is needed to support clock domain crossing. Currently it is only used at the interface to the configuration port of the Argo network-on-chip.

Table 3.9: OCPio signals

Signal	Description
MCmd	Command
MAddr	Address, byte-based, lowest two bits always 0
MData	Data for writes, 32 bits
MByteEn	Byte enables for sub-word accesses, 4 bits
MRespAccept	Enable handshaking with SResp
SResp	Response
SData	Data for reads, 32 bits
SCmdAccept	Flow control towards the command from the master

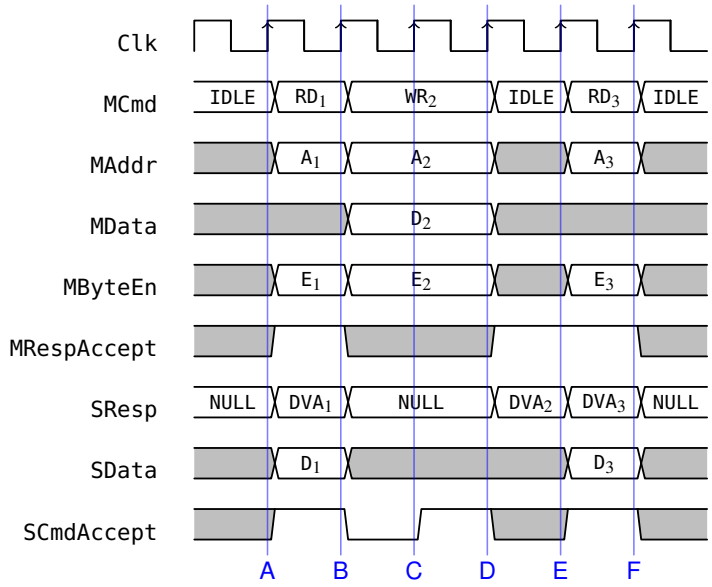


Figure 3.9: Timing diagram for OCPio

OCPio is slightly more flexible than OCPcore and appropriate for I/O devices that do not (or cannot) follow the semantics of OCPcore. Registering the master signals changes the protocol as follows:

- Slaves may generate responses in the same cycle as they receive a command.
- Commands are issued earliest in the cycle after a response (no pipelining).
- SCmdAccept is supported. It is sufficient to register the master signals only if the currently registered command is IDLE or SCmdAccept is high.
- In order to have symmetric handshaking for commands and responses and to facilitate clock-domain crossing, OCPio also includes a signal MRespAccept. An OCPio port that is derived directly from the pipeline's OCPcore port always accepts responses.

Figure 3.9 shows a sequence read/write/read in OCPio, where the slave does not accept the write immediately and delays the response to the write by one cycle.

A: The master issues a read by setting MCmd to RD. The slave accepts the command by setting SCmdAccept to high and responds immediately by setting SResp to DVA and returning the appropriate data.

B: The master issues a write command WR with data D_2 and byte enables E_2. The slave signals that it does not accept the command by setting SCmdAccept to low.

Table 3.10: OCPburst signals

Signal	Description
MCmd	Command
MAddr	Address, byte-based, lowest two bits always 0
MData	Data for writes, 32 bits
MDataByteEn	Byte enables for sub-word writes, 4 bits
MDataValid	Signal that data is valid, 1 bit
SResp	Response
SData	Data for reads, 32 bits
SCmdAccept	Signal that command is accepted, 1 bit
SDataAccept	Signal that data is accepted, 1 bit

C: As the slave did not accept the command in cycle B, the master still issues the command. The slave accepts the command by setting SCmdAccept to high.

D: The slave responds to the write it accepted in cycle C. Note that a) the master is not allowed to issue a new command immediately and b) SCmdAccept may take any value, because MCmd is IDLE.

E: The master issues a read to which the slave responds immediately.

3.6.4 OCPburst

The caches access the external memory through bursts only; Table 3.10 shows the signals of the OCPburst interface. The tie-off value for MBurstLength is 4, and MBurstSingleReq is tied off to 1. This means that the master supplies four data words for each write command, and the slave returns four words for each read command. The burst length is configurable, but might be restricted by the external memory and the external memory controller. All other burst-related signals are tied off to their default values. This entails that the only sequence for burst addresses is INCR. Bursts always start at an address that is aligned to the burst size (burst_aligned=1). Furthermore, reqdata_together is set to 1, i.e., write commands and the respective first word are issued together. Instead of the signal MByteEn, OCPburst uses the signal MDataByteEn. This implies that partial write transfers are fully supported, but partial read commands are unsupported.

We assume that the master provides data for burst accesses in consecutive cycles and that slaves can accept all burst data words once they accept the first word. To enable handshaking for the acceptance of the first data word, the OCPburst variant includes the signals MDataValid and SDataAccept. As reqdata_together is set to 1, delaying the acceptance of data also delays the acceptance of write commands. In order to do the same for read commands, OCPburst also includes the signal SCmdAccept. The signals SCmdAccept and SDataAccept can be generated by the same logic. For the acceptance of write commands they must be identical, otherwise at least one of the signals can have an undefined value. We assume that slaves return burst read data in consecutive cycles.

The first response to a read command may be given in the cycle after the command. The response to a write command may be given earliest in the cycle after the last data word was sent. Commands may be issued earliest in the cycle after the last response from an earlier command is received.

Figure 3.10 shows a read followed by a write in OCPburst, were the slave does not accept the first data word immediately.

A: The master issues a read command by setting MCmd to RD. The slave accepts by asserting SCmdAccept.

B: The slave provides the first response $\texttt{DVA}_{1.0}$, with data from address $\texttt{A}_1$.

C: The slave provides the second response $\texttt{DVA}_{1.1}$, with data from address $\texttt{A}_1$+4.

D: The slave provides the third response $\texttt{DVA}_{1.2}$, with data from address $\texttt{A}_1$+8.

E: The slave provides the fourth and last response $\texttt{DVA}_{1.3}$, with data from address $\texttt{A}_1$+12.

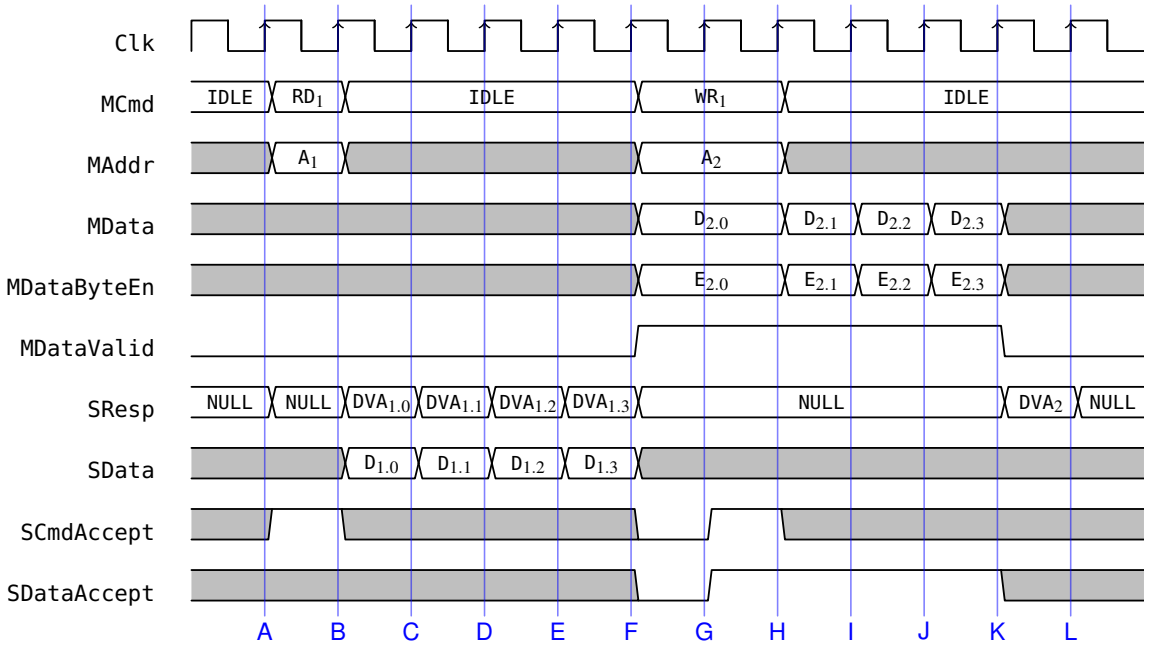


Figure 3.10: Timing diagram for `OCPburst`

F: The master issues a write command by setting `MCmd` to `WR` and provides the first data word $D_{2.0}$ width byte enables $E_{2.0}$ It signals that the data is valid by asserting `MDataValid`. The slave signals that it cannot accept the data by setting `SDataAccept` to low.

G: As the slave did not accept the data in cycle F, the master keeps issuing the command and providing the data word $D_{2.0}$. The slave now accepts the data by setting `SDataAccept` to high.

H: The master provides the second data word, $D_{2.1}$ with byte enables $E_{2.1}$.

I: The master provides the third data word, $D_{2.2}$ with byte enables $E_{2.2}$.

J: The master provides the fourth and last data word, $D_{2.3}$ with byte enables $E_{2.3}$.

K: The slave responds to the write burst by setting `SResp` to `DVA`.

3.6.5 Remarks

`SCmdAccept` is valid only while a command is unequal to `IDLE`. Consequently, `SCmdAccept` must be properly multiplexed to support multiple slaves. For handshaking via `SResp`, it is sufficient to combine the responses of different slaves with OR.

In `OCPcore` and `OCPcache`, slaves accept commands implicitly. Asserting a command for more than one cycle corresponds to issuing two separate commands. This is only allowed if the slave responds in the cycle immediately after the first command. In `OCPio` and `OCPburst`, slaves accept commands in the cycle where they assert `SCmdAccept`. Continuing to assert a command in the next cycle corresponds to a separate command. This is disallowed in `OCPburst`, and allowed in `OCPio` only if the slave responds immediately in the cycle where it asserts `SCmdAccept`.

The burst length is restricted to a constant which is a power of 2. The address must be aligned. Burst data must be provided in consecutive cycles. For reads, `SResp` is active during these cycles and the number of responses must always match the burst length. Error responses (where SResp has value `ERR`) may not abort the response sequence prematurely. For a burst write, the master may have to provide D1 for two or more cycles if `SDataAccept` is not active in the first cycle of the transaction.

Listing 3.1: Companion object for the counter device

```
object Counter extends DeviceObject {

  def init(params: Map[String, String]) = {}

  def create(params: Map[String, String]): Counter = Module(new Counter())

  trait Pins {}
}
```

As the first data word must be accepted together with the command and we require burst data to be provided in consecutive cycles, the signals `MDataValid` and `SDataAccept` may appear to be superfluous. However, they are required by the OCP standard for the inclusion of the `MDataByteEn` signal, which provides separate byte enables for each word in a burst transaction.

3.7 Example I/O Device

In this section we summarize the chapter by showing how to build a simple I/O device, a counter. Several examples of I/O devices can be found in `patmos/hardware/src/io`. Adding an I/O device to Patmos consists of 2–3 steps:

- Describing the I/O device in Chisel
- Adding the device and the address into the configuration file
- If the device contains pins to the external world, the pins need to be added to the top-level (VHDL) file and to the Quartus (.qsf) or Xilinx configuration files

An I/O device has to extend `CoreDevice` and needs to define a companion object that extends `DeviceObject`. The interface to the processor is usually `OCPcore` where handshaking for the data is available via `SResp`. In this example we will define a read only device that implements a simple counter. The complete code for this device can be found in `patmos/hardware/src/io/Counter.scala`.

Each device needs to define a companion object that is used as a factory object to create devices and needs to overwrite two methods: `init()` and `create()` and needs to overwrite the trait `Pins`. Listing 3.1 shows the the companion object for our counter device. The method `init()` can be used to pass parameters from the configuration file (e.g., altde2-115.xml). As we do not use parameters in our simple example we just define an empty method. The method `create()` is the singleton method that shall return an instance of an I/O device. In our example we create one `Counter` object, wrap it into a `Module`, as any Chisel component, and return it from `create()`. The method `create()` itself can receive parameters, which we do not use in this example.

Listing 3.2 shows the first, incomplete, version of our I/O device. We simply return 42 on a read from the device. However, we also need to adhere to the OCP signaling. We need to response to a read command in the next clock cycle with a data valid signal (`DVA`). Otherwise we need to set the response to `NULL`.

Note that the read needs to be valid in the very same cycle when the output of `respReg` becomes `DVA`. In the first version of this example we ignore writes to the counter.[3]

Listing 3.3 shows the last step, configuring Patmos to use the counter. In our example we change the configuration file that is used for the default configuration: `patmos/hardware/config/altde2-115.xml`. The two lines from Listing 3.3 need to added into section `IOs` and `Devs`, respectively. Offset 11 is the next available I/O address and results in mapping our counter to I/O address `0xf00b0000` (each I/O device has a 16 bit local address and 11 in decimal is 'b' in hexadecimal).

Listing 3.4 shows a small test program to read our counter device and write the value to stdout. As our I/O device lives in the local memory area we need to tell the C compiler to the local load and store instructions to

[3] If we write to the counter device, the device will not generate a `DVA` for `Resp` and therefore stall the pipeline forever. A complete design should at least ignore writes, but response to them.

Listing 3.2: Class for the counter device

```
class Counter() extends CoreDevice() {

  io.ocp.S.Data := UInt(42)

  val respReg = Reg(init = OcpResp.NULL)

  respReg := OcpResp.NULL
  when(io.ocp.M.Cmd === OcpCmd.RD) {
    respReg := OcpResp.DVA
  }

  io.ocp.S.Resp := respReg
}
```

Listing 3.3: Configuring the processor to include our counter device

```
<IO DevTypeRef="Counter" offset="11"/>

<Dev DevType="Counter" entity="Counter" iface="OcpCore"/>
```

Listing 3.4: Testing the counter device

```
#include <machine/patmos.h>
#include <stdio.h>

int main() {

  volatile _IODEV int *io_ptr = (volatile _IODEV int *) 0xf00b0000;
  int val;

  val = *io_ptr;
  printf("%d\n", val);
}
```

access the I/O device. This is performed by defining a special pointer with the macro `_IODEV`, which we import by including `<machine/patmos.h>`.

To implement a counter we need a register and some addition. Listing 3.5 hows the Chisel code for this free running counter. We now have a device the ticks at the processor frequency and can be used to measure execution time. Listing 3.6 shows how to use our counter to measure the execution time of the `printf` function with a simple string print out.

To complete the example we add a write to the device where we can set the value of the counter. Listing 3.7 shows the complete code of our counter where we can set the value. On a OCP write we take the data value in the same clock cycle as input to the counter register. As with the read operation, we acknowledge the write in the following clock cycle with a `DVA`.

When the I/O device has more than one register (or some memory), the address signals from the OCP connection (`io.ocp.M.Addr`) are used for address decoding. See for an example the `CpuInfo` device.

Listing 3.5: Chisel code for counting

```
  val countReg = Reg(init = UInt(0, 32))
  countReg := countReg + UInt(1)
  io.ocp.S.Data := countReg
```

Listing 3.6: Measure execution time

```
  val1 = *io_ptr;
  printf("Hello");
  val2 = *io_ptr;

  printf("Execution time is %d\n", val2-val1);
```

Listing 3.7: A writable counter

```
class Counter() extends CoreDevice() {

  val countReg = Reg(init = UInt(0, 32))
  countReg := countReg + UInt(1)
  when (io.ocp.M.Cmd === OcpCmd.WR) {
    countReg := io.ocp.M.Data
  }

  val respReg = Reg(init = OcpResp.NULL)
  respReg := OcpResp.NULL
  when(io.ocp.M.Cmd === OcpCmd.RD || io.ocp.M.Cmd === OcpCmd.WR) {
    respReg := OcpResp.DVA
  }

  io.ocp.S.Data := countReg
  io.ocp.S.Resp := respReg
}
```

4 Application Binary Interface

4.1 Data Representation

Data words in memories are stored using the big-endian data representation, this also includes the instruction representation.

4.2 Register Usage Conventions

The register usage conventions for the general purpose registers are as follows:

- `r0` is defined to be zero at all times. This is actually not just a convention, but implemented by the hardware.
- `r1` and `r2` are used to pass the return value on function calls.
  For 64 bit results, the high part is stored in `r1`, the low part in `r2`. 32 bit results are returned using `r1` only.
- `r3` through `r8` are used to pass arguments on function calls.
  For 64 bit arguments, the high part is stored first, followed by the low part.
  E.g., for a 64 bit argument passed in `[r3,r4]`, the high part is in `r3`, the low part in `r4`.
- `r29` is used as temp register.
- `r30` is defined as the frame pointer and `r31` is defined as the stack pointer for the *shadow* stack in global memory. The use of a frame pointer is optional, the register can freely be used otherwise. `r31` is guaranteed to always hold the current stack pointer and is not used otherwise by the compiler.
- `r1` through `r19` are caller-saved *scratch* registers.
- `r20` through `r31` are callee-saved *saved* registers.

The usage conventions of the predicate registers are as follows:

- all predicate registers are callee-saved *saved* registers.

The usage conventions of the special registers are as follows:

- `s0`, representing the predicate registers, is a callee-saved *saved* register.
- The stack cache control registers `ss` and `st` are callee-saved *saved* registers.
- The return information registers `s7-s10` (`srb`, `sro`, `sxb`, `sxo`) are callee-saved *saved* registers.
- All other special registers are caller-saved scratch registers and should not be used across function calls.

4.3 Function Calls

Function calls have to be executed using the `call` instruction that automatically prefetches the target function to the method cache and stores the return information in the special registers `srb` and `sro`.

The register usage conventions of the previous section indicate which registers are preserved across function calls.

The first 6 arguments of integral data type are passed in registers, where 64-bit integer and floating point types occupy two registers. All other arguments are passed on the *shadow* stack via the global memory.

When the return function base `srb` and the return offset `sro` needs to be saved to the stack, they have to be saved as the first elements of the function's stack frame, i.e., right after the stack frame of the calling function. Note that in contrast to `br` and `brcf` the return offset refers to the next instruction after the *delay slot* of the corresponding `call` and can be implementation dependent (cf. the description of the `call` and `ret` instructions).

4.4 Sub-Functions

A function can be split into several sub-functions. The program is only allowed to use `br` to jump within the same sub-function. To enter a different sub-function, `brcf` must be used. It can only be used to jump to the first instruction of a sub-function.

In contrast to `call`, `brcf` does not provide link information. Executing `ret` in a sub-function will therefore return to the last `call`, not to the last `brcf`. Function offsets however are relative to the *sub-function* base, not to the surrounding function. The function base register `r30` must therefore be set to the base address of the current *sub-function* for calls inside sub-functions.

A sub-function must be aligned and must be prefixed with a word containing the size of the sub-function, like for a regular function. If a function is split into sub-functions, the first sub-function must also be prefixed with the size of the first sub-function, not with the size of the whole function.

There are no calling conventions for jumps between sub-functions, for the compiler this behaves just like a regular jump.

4.5 Stack Layout

All stack data in the global memory, either managed by the stack cache or using a frame/stack pointer, grows from top-to-bottom. The use of a frame pointer is optional.

Unwinding of the call stack is done on the stack-cache managed stack frame, following the conventions declared in the previous subsection on function calls.

4.6 Interrupts and Context Switching

Interrupt handlers may use the shadow stack pointer `r31` to spill registers to the shadow stack. Interrupt handlers must ensure that all special registers that might be in use when the interrupt occurs are saved and restored. Here is an example of storing and restoring the context for context switching.

```
sub $r31 = $r31, 56
swc [$r31 + 0] = $r20  // free some registers
swc [$r31 + 1] = $r21
swc [$r31 + 2] = $r22
swc [$r31 + 3] = $r23
mfs $r20 = $s0
mfs $r21 = $sm
mfs $r22 = $sl  // by now any mul should be finished
mfs $r23 = $sh
swc [$r31 + 4] = $r20
swc [$r31 + 5] = $r21
swc [$r31 + 6] = $r22
swc [$r31 + 7] = $r23
mfs $r22 = $ss  // read out cache pointers, spill
mfs $r23 = $st
sub $r22 = $r23, $r22
sspill $r22   // spill the memory, s5 == s6 now
swc [$r31 + 8] = $r22  // store the stack pointer
swc [$r31 + 9] = $r23  // store stack size
```

```
mfs $r20 = $srb  // store return info
mfs $r21 = $sro
mfs $r22 = $sxb
mfs $r23 = $sxo
swc [$r31 + 10] = $r20
swc [$r31 + 11] = $r21
swc [$r31 + 12] = $r22
swc [$r31 + 13] = $r23
swc [$r31 + 14] = $r30 // store frame pointer
...

// restore
lwc $r20 = [$r31 + 4]
lwc $r21 = [$r31 + 5]
lwc $r22 = [$r31 + 6]
lwc $r23 = [$r31 + 7]
mts $s0 = $r20
mts $sm = $r21
mts $sl = $r22
mts $sh = $r23
lwc $r22 = [$r31 + 8] // restore the stack
lwc $r23 = [$r31 + 9]
mts $ss = $r23
mts $st = $r23   // set top = spill and fill from memory
sens $r22
lwc $r20 = [$r31 + 10] // restore return registers
lwc $r21 = [$r31 + 11]
lwc $r22 = [$r31 + 12]
lwc $r23 = [$r31 + 13]
mts $srb = $r20  // restore return infos and registers
mts $sro = $r21
mts $sxb = $r22
mts $sxo = $r23
lwc $r20 = [$r31 + 0]
lwc $r21 = [$r31 + 1]
lwc $r22 = [$r31 + 2]
lwc $r23 = [$r31 + 3]
lwc $r30 = [$r31 + 14]
xret
add $r31 = $r31, 52
nop
nop
```

TODO: Check why add is 52 and not 56 as in the beginning.

5 Implementation

After a first implementation of Patmos in VHDL we did a cleanup and rewrite in a the hardware description language Chisel [3]. The following notes on the implementation of Patmos and implementation decisions is based on first design discussions within the VHDL version and concrete implementation experiments with Chisel. All size and frequency numbers are from the Chisel implementation. A comparison between VHDL and Chisel would be of great interest.

For a comparison between Chisel and VHDL we take a snapshot when both versions where about at the same functionality: LoC, excluding the copyright header at 6.4.2013: Chisel: 996 VHDL: 3020. However, the VHDL code was written quite verbatim and more usage of `record` would probably result in about 2000 L0C. Still Chisel is more compact and probably easier readable.

5.1 Component Organization and Pipeline Structure

The architecture of Patmos is structured around five components, each representing one pipeline stage. Each component contains the *left* pipeline register. E.g., the output of the `DEC` stage (decode signals, the two register values, and the immediate field) is combinatorial from the decode stage and registered in the `EX` stage. The motivation of this organization is that input registers of on-chip memory elements (e.g., instruction memory, register file, and data memory) are part of the pipeline register. They need to be fed unconditionally from the unregistered output of the former stage.

Each stage has exactly one pipeline register, which is placed at the begin of the component. The pipeline registers use an enable for stalling. Register that have no enable (input registers of on-chip memories) need a *shadow* register and a multiplexer for stalls.

The interface from the EX stage to the MEM stage might use one field for ALU results and the store data or individual fields. Individual fields might reduce the pressure on the ALU multiplexers.

5.2 Register File

There are two options to implement the register file (RF) in an FPGA: (1) use two on-chip memories to provide two read ports and one write port, or (2) use dedicated registers and larger multiplexer structures for the read ports. Usually one aims to use on-chip memory for the RF. However, in a design constraint largely by the available amount of on-chip memory, a RF built out of registers might be preferable.

For a dual issue pipeline we need 4 read ports and 2 write ports into the RF. We explored double clocking of a on-chip based RF in [13]. It is feasible, the resulting maximum frequency fits for the ALU path, but feels a little bit brittle. A RF from registers might give a more robust design for the two write ports.

The ideal solution would be to make it configurable if on-chip memory or LCs are used. The issue width should also be configurable.

5.3 Resource and Fmax Numbers

State 13.3.2013 with Chisel and DE2-70: A shared field (for EX to MEM?) results 3435 LCs and 81.7 MHz, two fields in 3499 LCs and 81.8 MHz. Looks like not a big deal, but just 64 more LCs. Where does this cost come from? A very inefficient enable on the pipeline register (MUX instead of an enable signal?).

5.4 ALU Discussion

The large multiplexers and the forwarding limit the maximum frequency. We have already removed the expensive rotate instructions and the `abs` instruction.

Current version (4.4.2013) with all ALU operations and test case ALU.s for the DE2-70 is: 3415 LCs, 85.44 MHz. Dropping `rsub` and all unary ALU operations: 3173 LCs, 91.91 MHz.

TODO: This should be updated. Maybe even with some statistics how the size (and performance ?) changed over time.

6 Build Instructions

In the following we describe: (1) the installation of needed tool under Ubuntu and on Mac OS X, (2) how to build the Patmos tool chain (e.g., compiler), and (3) how to get started with Patmos.

The installation instructions might also be valid on different Linux versions. Patmos and the compiler have also been successfully installed on a Mac OS X system. The support of Windows is marginal, or basically not existent.

For an easier start we provide a VM image with Ubuntu 14.04 including all tools installed for download from: `http://patmos.compute.dtu.dk/`

6.1 Setup on Ubuntu and Mac OS X

In the following we present the Patmos build instructions including setup of tools in Ubuntu and for the Mac OS X.

6.1.1 Setup on Ubuntu 16.04 LTS 64-Bit

In the following we present the setup of tools within a recent Ubuntu version. Here is the list of needed (or recommended) packages:

```
sudo apt-get install git default-jdk gitk cmake make g++ texinfo flex bison \
  subversion libelf-dev graphviz libboost-dev libboost-program-options-dev ruby-full \
  liblpsolve55-dev python zlib1g-dev gtkwave gtkterm scala
```

Install `sbt` with:

```
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
  --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
sudo apt-get update
sudo apt-get install sbt
```

For the Quartus setup it is best to change the default shell to `/bin/bash`:

```
sudo rm /bin/sh
sudo ln -s /bin/bash /bin/sh
```

For building the Patmos documentation (e.g., the handbook) you need to install a full version of LaTeX (about 3 GB) with:

```
sudo apt-get install texlive-full doxygen
```

6.1.2 Setup On Mac OS X

This subsection describes the installation of needed tools and libraries on Mac OS X. It is based on a OS X Yosemite and assumes to use homebrew[1] for package management.

First install the Mac compiler (Xcode) and the command line tools.

For the build of Patmos and the compiler we need to install:

```
brew install cmake boost libelf sbt doxygen
```

[1] `http://brew.sh/`

Mac OS X comes with Python version 2.7 preinstalled. For Aegean we need Python 3 and the xml library that can be installed with:

```
brew install python3
pip3 install lxml
```

For WCET analysis install

```
brew install lp_solve
```

For wave viewing install GTKWave with

```
brew install homebrew/x11/gtkwave
```

TODO: The following should probably be deleted as we switched to homebrew for the setup.
Several tools are needed, best installed with MacPorts. For Patmos simulator and assembler: `boost`, `libelf`. For simulation with ModelSim: `wine` For Aegean: `python33`, `py33-lxml`. Make a link from `python3` to `python3.3` as this is the way it is invoked. *TODO: Alex suggested a way to avoid this by querying how to invoke python.*

6.2 Building Patmos and the Compiler Tool Chain

We assume that the T-CREST project will live in `$HOME/t-crest`. Before building Patmos add the path to the compiler executables (e.g., into your `.bashrc` or `.profile`):[2]

```
export PATH=$PATH:$HOME/t-crest/local/bin
```

A complete logout from Ubuntu might be needed to take effect (just closing a terminal window is not enough, depending on how you set up your profile files).

Patmos and the compiler can be checked out from GitHub and built as follows:

```
mkdir ~/t-crest
cd ~/t-crest
git clone https://github.com/t-crest/patmos-misc.git misc
./misc/build.sh
```

For developers with push permission generate an ssh key and upload it at GitHub (see `https://help.github.com/articles/connecting-to-github-with-ssh/` for detailed instructions). The ssh based clone string for write access is then:

```
git clone git@github.com:t-crest/patmos-misc.git misc
./misc/build.sh
```

This script (`build.sh`) will checkout several other repositories (the compiler, library, and the Patmos source) and builds the compiler and the Patmos simulator. Therefore, take a cup of coffee and find some nice reading.

You can test your installation by checking if the compiler is available:

```
patmos-clang --version
```

The `build.sh` script contains default options, which should work out of the box. The build settings can be changed by a customized `misc/build.cfg` file. The file `misc/build.cfg.dist` is an example configuration file containing default values. It is ignored by the build process and should not be edited.[3] To change any options for `misc/build.sh`, either start with an empty `misc/build.cfg` or copy `misc/build.cfg.dist` and modify the values to your need.

For correct signing of your changes set the username and email in git with:

[2] The path needs to be absolute. LLVM cannot handle a path relative to the home folder ~, e.g., `~/t-crest/local/bin`.
[3] It is autogenerated by `build.sh -e` from the values in `build.sh`.

```
git config --global user.name "Joe Someone"
git config --global user.email "joe.someone@domain.com"
```

Optionally, you may additionally add the `misc` checkout to your path, so that `build.sh` and the helper tools in `misc` can be executed from everywhere.

```
export PATH=$PATH:$HOME/t-crest/misc
```

The Patmos documentation (handbook, C library, Argo NoC) can be built with:

```
cd patmos/doc
make
```

6.3 Quartus on Linux

Download the free web edition of Quartus from Altera.[4] The Linux version is installed as follows:[5]

```
tar xvf Quartus-web-xxx.tar
```

The software installation is started with:

```
bash setup.sh
```

Then add the `bin` directory of Quartus to your $PATH. For access to the serial port the user needs access rights.

```
# Add user to dialout group for the serial port access
sudo usermod -a -G dialout user
```

Logout and login again to update the group settings.

Getting the Altera USB Blaster working on Ubuntu is a little bit brittle. Here a collection of tips collected from different places that helped me to get the USB Blaster running.

Add permissions to access the Altera USB Blaster by creating or editing `/etc/udev/rules.d/51-usbblaster.rules`:

```
# For Altera USB-Blaster permissions.
SUBSYSTEM=="usb",\
ENV{DEVTYPE}=="usb_device",\
ATTR{idVendor}=="09fb",\
ATTR{idProduct}=="6001",\
MODE="0666",\
NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}",\
RUN+="/bin/chmod 0666 %c"
```

Reload the rules:

```
sudo udevadm control --reload
```

Test the JTAG chain with:

```
jtagconfig
```

and hope for an output similar to:

```
1) USB-Blaster [2-2.2]
  020F70DD   EP3C120/EP4CE115
```

[4] For a 32-bit Linux you need to use Quartus 13.x as 32-bit support has been dropped from Quartus 14.x.

[5] `http://www.altera.com/literature/manual/quartus_install.pdf`

In some cases, killing the the jtagd process can make the connection work againg:

```
$ sudo killall -9 jtagd
$ sudo killall -9 jtagd # Verify that is not running
jtagd: no process found
$ jtagconfig
```

Some more fixes might be needed (not on a recent try with Ubuntu 14.04):

```
sudo ln -s /lib/x86_64-linux-gnu/libudev.so.1 /usr/lib/libudev.so.0
jtagconfig -d
```

After fixing the permissions for the USB Blaster open Quartus and test if the cable is found with the programmer. Select USB-Blaster in *Hardware Setup*. When connected to an FPGA test the USB-Blaster with *Auto Detect* (With the DE2-115, a question about shared JTAG ID pops up – select EP4CE115).

Quartus 13.1 drops the support of Cyclone II devices. Therefore, the (phased out) Altera DE2-70 board is not supported anymore. Version 13.0 supports Cyclone II till Cyclone V devices and might be the best option at the moment.

6.4 The Xilinx ML605 Platform

For the evaluation within the T-CREST project the Xilinx ML605 FPGA board was chosen as the 'standard' evaluation platform. The T-CREST platform contains, besides several Patmos' connected with the Argo NoC, a memory tree, called BlueTree, from UoY and the time-predictable memory controller from TU/e. As the building this platform needs some non-free tools and including closed source code, we provide prebuilt configurations of the T-CREST platform as bit files available for download:

- A 4 core version: `http://patmos.compute.dtu.dk/t-crest-4core.bit`
- A 16 core version:

To configure the FPGA use the IMPACT software from Xilinx (command `impact`), which is part of the Xilinx ISE package and needs no license (It is also available in a smaller Lab package). To compile an application and download it to the ML605 follow the exact same steps as for the Altera CMP version, e.g., from within the `patmos` directory:

```
make APP=hello_puts comp download
```

6.4.1 Getting the Xilinx Configuration Cable to Work

Xilinx does not support Ubuntu (at least the last few versions) directly. The following description is a summary of the help from Jamie.

Copy some Xilinx cable specific files to `/usr/share` with:

```
sudo cp /opt/Xilinx/14.7/ISE_DS/ISE/bin/lin/install_script/install_drivers/\
linux_drivers/pcusb/*.hex /usr/share
```

Ensure that `fxload` is installed with:

```
sudo apt-get install fxload
```

Create `/etc/udev/rules.d/80-xusbdfw.rules` with following content:

```
# version 0003
ATTR{idVendor}=="03fd", ATTR{idProduct}=="0008", MODE="666"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0007", RUN+="/sbin/fxload -v -t fx2 -I /usr/share/xusbdfwu.hex -D $tempnode"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0009", RUN+="/sbin/fxload -v -t fx2 -I /usr/share/xusb_xup.hex -D $tempnode"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="000d", RUN+="/sbin/fxload -v -t fx2 -I /usr/share/xusb_emb.hex -D $tempnode"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="000f", RUN+="/sbin/fxload -v -t fx2 -I /usr/share/xusb_xlp.hex -D $tempnode"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0013", RUN+="/sbin/fxload -v -t fx2 -I /usr/share/xusb_xp2.hex -D $tempnode"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0015", RUN+="/sbin/fxload -v -t fx2 -I /usr/share/xusb_xse.hex -D $tempnode"
```

Restart the udev service with:

```
sudo udevadm control --reload-rules
```

Make sure that libusb is installed (e.g., in `/lib/i386-linux-gnu` on a 32-bit Ubuntu) and make a symbolic link with

```
sudo ln -s libusb-1.0.so.0 libusb.so
```

Best restart your machine and connect the USB cable again.

Using iMPACT Start impact from a terminal with

```
impact
```

Impact pops up some dialog boxes:
Do you want iMPACT to automatically load the last saved project for you? – answer with *No*.
Do you want the system to automatically create and save a project file for you? – answer *Yes*.
Next window click Ok. iMPACT should detect the JTAG chain with two devices. Answer the following dialog boxes wit *No* and *Cancel*.
Right-click on the xc6vlx240t device and select *Assign New Configuration File* and select the provided .bit file (e.g., `t-crest-4core.bit`). Answer the following question about attached Flash PROMs with *No*.
Right-click again on the FPGA symbol and select *Program*. Select *Ok* on the next dialog box and the FPGA shall be configured.
Then compile and download an application as described above.

Installing the Xilinx Tools After extracting the tools with `tar` the install procedure is started within the `Xilinx_*` directory with:

```
sudo ./xsetup
```

Starting Xilinx ISE In some setups it is needed to source a setup script, e.g.:

```
source /opt/Xilinx/14.7/ISE_DS/settings64.sh
```

Then ISE can be started with `ise`.

6.4.2 Updating the Patmos Cores with Aegean

The correct Verilog file for the two version of Patmos (core 0 that downloads an application and the other cores) is built with the *Aegean* tool. The 4 cores version is built with:

```
make platform AEGEAN_PLATFORM=ml605_4core
```

A 16 core version is available as well. The generated Verilog files are the same, but the schedule for the Argo NoC is different (which is copied to `t-crest/patmos/c/nocinit.c`).
All generated file can be cleaned with `make cleanall`.
The configuration of T-CREST with Bluetree and the TU/e memory controller is available via a .tgz file, exchanged via email to protect IP rights. Therefore, they are not integrated in Aegean and some manual code copying is needed. The `ml605.tgz` shall be extracted within the `t-crest` directory. Copy the Patmos Verilog files (`ml605mPatmosCore.v` and `ml605mPatmosCore.v`) from the build directory (`t-crest/aegean/build/ml605_4core`) into `t-crest/ml605`.

6.5 Testing

Patmos base functionality can be tested by comparing the execution of the simulator with the execution of the emulator. *TODO: Write more on how it works. Somewhere we should also talk about the two different ways to build Patmos: within the patmos repro and with build.sh where the emulator gets installed.* Within the `patmos` folder execute:

```
make test
```

More testing can be performed with the programs included in the benchmark repository. This repository is not included in the default checkout and build. Therefore, within the folder `t-crest` execute

```
misc/build.sh -t bench
```

to checkout the benchmarks, compile them, and execute them. Building Patmos via the build script also ensures that the correct emulator is used. Be aware that this is a very lengthy task; it takes on a MacBook Pro more than 3 hours.

Regression Tests: Two simple scripts are available in `misc` that do a clean checkout of T-CREST, including the benchmarks, and performing a build and tests: `regtest-init.sh` and `regtest.sh`. `regtest-init.sh` starts the build process with checking out `misc` and therefore needs to be copied out of the repository to a place for scripts. The regression test will send out result emails to all listed in `recipients.txt`.

6.6 ModelSim License

In the case that you have a DTU Compute login you can access the license servers from outside the DTU network by setting up an SSH tunnel. An example of how such a tunnel can be set up follows, you need to insert you own username.

```
ssh -L 1717:angel2:1717 -L 1718:angel2:1718 ${USERNAME}@sshlogin.compute.dtu.dk
```

When the SSH tunnel is setup the LM_LICENSE_FILE needs to be set to:

```
LM_LICENSE_FILE=1717@localhost
```

This way of setting up an SSH tunnel might also work for other institutions.

License settings for ModelSim and Xilinx

```
export LM_LICENSE_FILE=1717@angel1:1717@angel2:1717@angel3
export XILINXD_LICENSE_FILE="2100@eda1.imm.dtu.dk"
```

7 Tools

Along with Patmos come several tools; this chapter describes these tools and how to use them.

7.1 Simulation, Emulation, and Execution

7.1.1 pasim

The Patmos simulator `pasim` provides a high-level simulation of Patmos. It is useful for quick evaluations of different hardware configurations and for debugging applications. As it can provide detailed reports about the application behavior, it is particularly useful during the initial phases of application development.

Usage The general usage of `pasim` is `pasim <file>`, where `<file>` may be a plain binary or an ELF file. Tables 7.1, 7.2, 7.3, and 7.4 show the various options of the simulator. For memory/cache sizes the following units are allowed: `k`, `m`, `g`, or `kb`, `mb`, `gb`.

7.1.2 Patmos Emulator

The tool `patemu` provides a C++-based simulator that is derived from the actual hardware description. While it is slower and less flexible than `pasim`, its behavior is identical to the behavior of actual hardware. It is therefore useful for investigating cases where the behavior of the simulator diverges from the behavior in the FPGA. The emulator can also generate wave form traces, which allow the investigation on a level similar to Verilog/VHDL-based simulations.

Usage The general usage is `patemu [<file>]`. When invoked without argument, `patemu` executes the code in the boot ROM of the processor. When given an ELF file as argument, the emulator loads the file and executes it directly. Table 7.5 shows the command-line options for the emulator.

Table 7.1: General options for `pasim`

Option	Description
`-h [ --help ]`	produce help message
`-c [ --maxc ] arg`	stop simulation after the given number of cycles (default: infinity)
`-b [ --binary ] arg`	binary or elf-executable file (stdin: `-`)
`--debug [=arg]`	enable step-by-step debug tracing after cycle, default: 0
`--debug-fmt arg`	format of the debug trace (`short`, `trace`, `instr`, `blocks`, `calls`, `calls-indent`, `default`, `long`, `all`)
`--debug-file arg`	output debug trace in file (stdout: `-`, default: stderr)
`--debug-intrs`	print out all status changes of the exception unit.
`--debug-nopc`	do not print PC and cycles counter in debug output
`-o [ --stats-out ] arg`	write statistics to a file (stdout: `-`, default: stderr)
`--print-stats arg`	print statistics for a given function only.
`--flush-caches arg`	flush all caches when reaching the given address (can be a symbol name).
`-V [ --full ]`	full statistics output
`-v [ --verbose ]`	enable short statistics output

Table 7.2: Memory Options for `pasim`

Option	Default	Description
`-g [ --gsize ] arg`	`64m`	global memory size in bytes
`-G [ --gtime ] arg`	`7`	global memory transfer time per burst in cycles
`-t [ --tdelay ] arg`	`0`	read delay to global memory per request in cycles
`--trefresh arg`	`0`	refresh cycles per TDM round
`--bsize arg`	`16`	burst size (and alignment) of the memory system.
`--psize arg`	`0`	Memory page size. Enables variable burst lengths for single-core.
`-p [ --posted ] arg`	`0`	Enable posted writes (sets max queue size)
`-l [ --lsize ] arg`	`2k`	local memory size in bytes
`--mem-rand arg`	`0`	Initialize memories with random data
`--chkreads arg`	`none`	Check for reads of uninitialized data, either per byte (`warn`, `err`) or per access (`warn-addr`, `err-addr`). Disables the data cache.
`--with-mmu arg`	`0`	Simulate memory management unit

Table 7.3: Cache options for `pasim`

Option	Default	Description
`-d [ --dcsize ] arg`	`2k`	data cache size in bytes
`-D [ --dckind ] arg`	`lru2`	kind of direct mapped/fully-/set-associative data cache (`ideal`, `no`, `dm`, `lru[N]`, `fifo[N]`)
`--dlsize arg`	`0`	size of a data cache line in bytes, defaults to burst size if set to 0
`-s [ --scsize ] arg`	`2k`	stack cache size in bytes
`-S [ --sckind ] arg`	`block`	kind of stack cache (`ideal`, `block`, `lblock`, `dcache`)
`-C [ --icache ] arg`	`mcache`	kind of instruction cache (`mcache`, `icache`)
`-K [ --ickind ] arg`	`lru2`	kind of direct mapped/fully-/set-associative I-cache (`ideal`, `no`, `dm`, `lru[N]`, `fifo[N]`
`--ilsize arg`	`0`	size of an I-cache line in bytes, defaults to burst size if set to 0
`-m [ --mcsize ] arg`	`2k`	method cache / instruction cache size in bytes
`-M [ --mckind ] arg`	`fifo`	kind of method cache (`ideal`, `lru`, `fifo`)
`--mcmethods arg`	`16`	Maximum number of methods in the method cache, defaults to number of blocks if zero
`--mbsize arg`	`8`	method cache block size in bytes, defaults to burst size if zero

7.1.3 config_altera

The script `config_altera` configures an Altera FPGA using the tool `quartus_pgm` provided by Altera.

Usage The usage of the script is `config_altera [-b <blaster>] [-h] <file>`. The option `-h` prints a basic help. The option `-b` specifies the blaster type for FPGA configuration; by default, the blaster type is `USB-Blaster`. The argument `<file>` specifies a `.sof` file with the bit stream for FPGA configuration.

7.1.4 config_xilinx

The script `config_xilinx` configures a Xilinx FPGA using the tool `impact` provided by Xilinx.

Usage The usage of the script is `config_xilinx [-h] <file>`. The option `-h` prints a basic help. The argument `<file>` specifies a `.bit` file with the bit stream for FPGA configuration.

Table 7.4: Simulator options for `pasim`

Option	Default	Description
`--cpuid arg`	0	Set CPU ID in the simulator
`-N [ --cores ] arg`	1	Set number of CPUs (enables memory TDM)
`--freq arg`	80	Set CPU Frequency in Mhz
`--interrupt arg`	1	enable or disable interrupts
`--mmbase arg`	0xf0000000	base address of the IO device map address range
`--mmhigh arg`	0xffffffff	highest address of the IO device map address range
`--cpuinfo_offset arg`	0x00000	offset where the cpuinfo device is mapped
`--excunit_offset arg`	0x10000	offset where the exception unit is mapped
`--timer_offset arg`	0x20000	offset where the timer device is mapped
`--uart_offset arg`	0x80000	offset where the UART device is mapped
`--led_offset arg`	0x90000	offset where the LED device is mapped
`--ethmac_offset arg`	0xb0000	offset where the EthMac device is mapped
`--ethmac_ip_addr arg`		Provide virtual network interface with the given IP address
`-I [ --in ] arg`	-	input file for UART simulation (stdin: `-`)
`-O [ --out ] arg`	-	output file for UART simulation (stdout: `-`)

Table 7.5: Options for `patemu`

Option	Description
`-e <addr>`	Provide virtual network interface with IP address <addr>
`-h`	Print help
`-i`	Initialize memory with random values
`-k`	Simulate random input from keys
`-l <N>`	Stop after <N> cycles
`-p`	Print method cache statistics
`-r`	Print register values in each cycle
`-s`	Trace stack cache spilling/filling
`-v`	Dump wave forms file `Patmos.vcd`
`-I <file>`	Read input for UART from file <file> (stdin: `-`, default: stdin)
`-O <file>`	Write output from UART to file <file> (stdout: `-`, default: stdout)

7.1.5 patserdow

The tool `patserdow` downloads an ELF file via a serial line to the FPGA and forwards output to and from the downloaded application. The download protocol uses CRC checksums to verify the integrity of the downloaded data. The `patserdow` tool terminates when the application on the FPGA terminates, with the same exit code as the application.

Usage The general usage is `patserdow [-v] [-t <time>] [-h] <port> <file>`. The option `-h` prints a basic help. The option `-v` turns on a verbose mode, where information about the file to be downloaded and the progress of the downloading process is printed to stderr. The option `-t` specifies a time out after which execution is terminated. Output from the application is written to stdout; input to the application is read from stdin. The argument <port> specifies the serial port to be used for downloading. The argument <file> specifies the ELF file to be downloaded.

7.1.6 patex

The tool `patex` combines FPGA configuration and application download such that ELF files can be executed on the FPGA without manual intervention. It can therefore act as a drop-in replacement for `pasim` and `patemu`. As `patex` executes the application on actual hardware, it is particularly useful for applications where simulation or emulation would be prohibitively slow.

Usage The general usage is `patex [-I <file>] [-O <file] <file>`. The argument for the option `-I` specifies where input to the UART should be read from. The argument for the option `-O` specifies where UART ouput should be written to.

The environment variable `PATEX_CONFIG` determines how the FPGA is configured. Permissible values are: `Make` (the default value), `Altera`, or `Xilinx`.

- `Make` means that the FPGA is configured by calling `make config` in the directory specified in environment variable `PATMOS_HOME`. When this variable is unset, `patex` uses the directory where it was installed from.
- `Altera` means that the FPGA is configured by calling `config_altera` with the file specified in environment variable `PATEX_CONFIGFILE`. If the environment variable `BLASTER_TYPE` is set, it is used as the blaster type.
- `Xilinx` means that the FPGA is configured by calling `config_xilinx` with the file specified in environment variable `PATEX_CONFIGFILE`.

`patex` recognizes URLs for `PATEX_CONFIGFILE` and downloads the file using `wget` if necessary. The protocols `http`, `https`, and `ftp` are supported.

The environment variable `COM_PORT` sets the serial port for downloading. When this variable is unset, `patex` uses the `COM_PORT` variable from the Makefile at the time of installation. The environment variable `TIMEOUT` sets a timeout in seconds. By default, `patex` terminates download and execution after 300 seconds. Setting the environment variable `VERBOSE` to `true` turns on verbose output.

7.2 Patmos Developer Tools

This section describes tools that are useful when working on Patmos itself, but are of little interest when developing applications.

7.2.1 elf2bin

The `elf2bin` tool converts ELF files to binary files.

Usage The `elf2bin` tool has two modes. In default mode, its usage is `elf2bin [-d <disp>] <infile> <outfile1> <outfile2>` and it dumps executable segments to file `<outfile1>` and other segments to `<outfile2>`. If option `-d` is provided, it uses a displacement of `<disp>` for the non-executable segments, such that data that is mapped to address `<N>` is dumped to position `<N>-<disp>` in the output file.

The second mode of `elf2bin` is a "flat" mode, with the usage `elf2bin -f <infile> <outfile>`. In that mode, `elf2bin` generates a flat output file, without any displacement. This file can be post-processed (e.g., with `hexdump -v -e '"%d,"' -e '" // %08x\n"'`) to generate a representation of the data for Verilog/VHDL-based simulations of external memory.

7.2.2 pacheck

The tool `pacheck` performs a "sanity" check of binaries and ELF files. While not being complete, it detects common errors such as control-flow instruction inside branch delay slots.

Usage The usage of `pacheck` is `pacheck [-h|--help] [-v|--verbose] [[-b|--binary] <input>]`. The option `-h` prints a help message. The option `-v` makes `pacheck` verbose. By default, `pacheck` reads from stdin; a file for checking can be given either as command-line argument or as argument to the option `-`.

7.2.3 paasm

The Patmos assembler `paasm` provides a basic assembler. It generates binary files and should be used only for writing very basic tests of Patmos. For developing applications in assembly, please use the assembler provided by the compiler, which is more complete and in particular supports the generation of ELF files.

Usage The usage of `paasm` is `paasm <input> <output>`.

7.2.4 padasm

The Patmos disassembler `padasm` is the counterpart of `paasm`, and similar restrictions apply. For general development, please use the `patmos-llvm-objdump` tool provided by the compiler.

Usage The usage of `padasm` is `padasm <input> <output>`.

8 The Patmos Compiler

The Patmos compiler is an adaptation of the LLVM compiler [7] to target the Patmos processor ISA and to provide a tighter integration with WCET analysis [10].

The compilation tool chain consists of the following components:

- `patmos-llvm` The compiler, including `platin` and various compiler tools, objdump and an assembler (patmos-llvm-mc).
- `patmos-clang` The C frontend and the compiler/linker driver. Compiled together with patmos-llvm.
- `patmos-gold` The `patmos-ld` ELF linker for Patmos.
- `patmos-compiler-rt` The runtime library, defining software implementations of div and floats.
- `patmos-newlib` The C library implementation.
- `patmos-benchmarks` Various benchmarks that have been adapted to Patmos.
- `patmos-misc` A collection of helper scripts for debugging, evaluation and building.
- `patmos` The processor and the simulator.

The compiler and libraries can be built using `misc/build.sh` as described in Section 6.2. Details on building the tool chain manually without the build script can be found in the `README.patmos` files provided in the various repositories.

8.1 Overview

Figure 8.1 gives an overview of the compiler tool chain. The compilation process starts with the translation of each C source file and libraries to the LLVM intermediate language (*bitcode*) by the C frontend `clang`. At this level, the user application code and static standard and support libraries are linked by the `llvm-link` tool. An advantage of linking on bitcode level is that subsequent analysis and optimisation passes, and the code generation backend have a complete view of the whole program. The `opt` optimiser performs generic, target independent optimisations, such as common sub-expression elimination, constant propagation, etc.

The `llc` tool constitutes the backend, translating LLVM bitcode into machine code for the Patmos ISA, and addressing the target-specific features for time predictability. The backend produces a relocatable ELF binary containing symbolic address information, which is processed by `gold`[1], defining the final data and memory layout, and resolving symbol relocations.

In addition to the machine code, the backend exports supplementary information for WCET analysis and optimisation purposes in form of a *Patmos Metainfo File*. This information contains, among others, flow information (in form of loop bounds provided by symbolic analysis on bitcode level), structural information (targets of indirect branches), and information on memory accesses (memory areas accessed by individual load/store instructions). This information can be further processed by the `platin` toolkit, by enhancing it (e.g., by a hardware model), translating it (e.g., to the input format for annotations of the timing analysis tool `aiT`, as used in the T-CREST project), or performing other analyses on it.

8.2 Compiling with the `patmos-clang` Driver

This section describes the usage of the `patmos-clang` C compiler.

[1] gold is part of the GNU binutils, see `http://sourceware.org/binutils/`

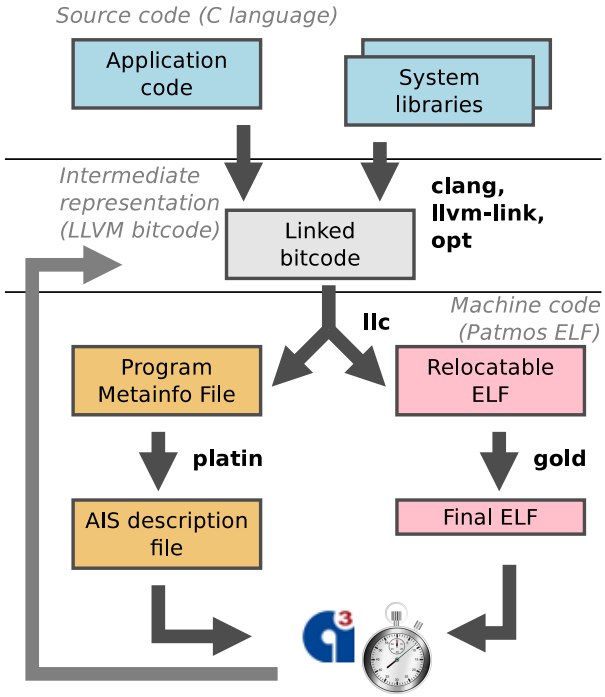


Figure 8.1: Compiler Tool Chain Overview

8.2.1 Compiling and Linking C Programs

C source files are by default compiled to bitcode objects (`patmos-clang -c`). To compile .c files to ELF objects, use `patmos-clang -c -fpatmos-emit-reloc`.

Assembly files are always compiled to ELF objects. Archive files (.a) can only contain bitcode objects or ELF objects, not a mixture of both. Shared libraries (either bitcode or ELF) are not supported.

It is possible to link multiple bitcode files into a single bitcode file and link it like a static library (compile with `patmos-clang -fpatmos-link-object -o lib<name>.bc`, link with -l<name>). Bitcode files are always fully linked in, even if there is no usage of any of its symbols. Unused symbols are removed in a separate optimization step.

Compiling single files to objects (using patmos-clang -c|-S)

1. Input .c files are compiled to bitcode files by default. Use -fpatmos-emit-obj to compile to ELF objects, or -fpatmos-emit-asm to compile to assembly files.

2. Input .s files are compiled to ELF files.

Linking multiple files with patmos-clang (i.e, not using -c or -S) The compiler driver (patmos-clang) performs the following steps to compile and link multiple input files.

1. All .c input files are compiled to individual bitcode objects. All assembly files are compiled to individual ELF files.

2. If -nostartfiles is not given and the target OS is not RTEMS, crt0 is added as first input file.

3. Depending on the -nodefaultlibs|-noruntimelibs|.. options, the following libraries are added after all user provided inputs: -lc (libc), -lpatmos (libgloss), -lrtsf (softfloats), -lrt (runtime).

4. For any of the above libraries, as well as -lm (libm), a lib<libname>syms.o file is added if the library is a bitcode library. The lib<x>syms.o files force the linker to pull in functions for which calls might be generated by LLC when compiling from bitcode to ELF.

5. All input files and libraries are checked if they are bitcode files/archives or ELF files/archives. All bitcode files are linked into a single bitcode file. ELF files are ignored in this step.

   Attention: This means that symbols that are defined in bitcode archives but are used only in ELF input files are not linked in! You need to link in a separate bitcode file containing a pseudo use of the required symbols.

6. The resulting bitcode file is optimized and compiled to relocatable ELF.

   Attention: The optimization step removes any symbol from the bitcode that are not used in bitcode. If a function is called only in an ELF object, you need to mark the function with `__attribute__((used))`.

7. The ELF file is linked with the other ELF files and ELF libraries at the position of the first bitcode input file. Relocations are resolved and additional symbols are defined. The result is an executable ELF file.

   Attention: Since bitcode inputs are linked first in a separate step, the linking order between bitcode files and ELF inputs is not (yet) fully preserved. Using -flto does not solve this, since the LTO plugin also links all bitcode files first, and only links in the linked bitcode file *after* all ELF inputs!

Driver Options

The `patmos-clang` driver can be used to generate bitcode files, to link bitcode files, or to emit assembler code. The driver supports the following modes of operation:

`patmos-clang -c <inputs>`
Input: `.c` C source file
Output: `.o` or `.bc` bitcode files
Actions: compile each input file to a bitcode file

`patmos-clang -S <inputs>`
Input: `.c` C source file
Output: `.s` or `.ll` human readable bitcode files
Actions: compile each input file to a human readable bitcode file

`patmos-clang -fpatmos-emit-llvm <inputs>`
Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
Output: bitcode file
Actions: compile to bitcode, link all input files, link with standard libraries and start code

`patmos-clang -fpatmos-emit-reloc -c <inputs>`
Input: `.c` C source file
Output: `.o` Patmos relocatable ELF
Actions: compile each input file to a Patmos relocatable ELF file

`patmos-clang -fpatmos-emit-asm -S <inputs>`
Input: `.c` C source file
Output: `.s` Patmos assembly file
Actions: compile each input file to a Patmos assembly file

`patmos-clang -fpatmos-emit-reloc <inputs>`
Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
Output: `.o` Patmos relocatable ELF
Actions: compile to bitcode, link all input files, link with standard libraries and start code, compile to relocatable ELF

Option	Description
`-mfloat-abi=none`	Do not use software floating point libraries when linking
`-nostdlib`	Do not use standard libraries such as `libc` when linking
`-nolibc`	Do not use `libc` when linking
`-nodefaultlibs`	Do not use platform system libraries when linking
`-nostartfiles`	Do not use the `crt0` start file when linking
`-nolibsyms`	Do not use symbol definition files for runtime libraries when linking. Those files prevent the linker from removing any functions for which calls might be generated by the compiler backend, such as software division or `memcpy`
`-fpatmos-link-object`	Link as object, i.e., do not link in any libraries or start code

Table 8.1: Options for `patmos-clang` that control the default behaviour of the linker

`patmos-clang -fpatmos-emit-asm <inputs>`
Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
Output: `.o` Patmos assembly file
Actions: compile to bitcode, link all input files, link with standard libraries and start code, compile to Patmos assembly

`patmos-clang -o <output> <inputs>`
Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
Output: Patmos executable ELF
Actions: compile to bitcode, link with standard libraries and start code, compile to relocatable ELF, create Patmos executable ELF

The compiler accepts standard options such as `-I`, `-L` and `-l` to define additional lookup paths for header files and libraries and to link with (static) libraries. The behaviour of the linker can be controlled with additional options for `patmos-clang` as shown in Table 8.1. Refer to `patmos-clang -help-hidden` for a list of all available options that control the behaviour of the driver, and to `patmos-llc -help-hidden` for all options that control the generation of machine code from bitcode. Options can be passed from `patmos-clang` to `patmos-llc` and other tools using `-Xclang`, `-Xopt`, `-Xllc`, `-Wl` and so on. To pass options to the internal LLVM backend of the clang compiler, use `patmos-clang -Xclang -mllvm -Xclang <option>`.

Libraries

The Patmos tool chain supports static libraries. Libraries are archives that contain either only bitcode files or ELF objects. The archives are created by using either the `ar` tool provided by the host system or by using `patmos-ar` from the `patmos-gold` binutils. The tool `patmos-llvm-nm` can be used to inspect the content of bitcode archives.

```
ar q libtest.a *.bc
# show the contents of libtest.a
patmos-llvm-nm libtest.a
# compile and link with the created library
patmos-clang -target patmos-unknown-elf -o app main.c -ltest
```

8.2.2 Disassembling

To disassemble .bc files, use `patmos-llvm-dis <file>.bc`.

To disassemble .o ELF files, use `patmos-llvm-objdump -d <file>`. Add '-r' to show relocation symbols (for relocatable ELFs or executables generated with -Xgold -q).

8.2.3 Debugging

Some useful commands for debugging:

```
# print out executed instructions and the values of their operands
# starting from some cycle
pasim --debug=<cycle-to-start-printing> --debug-fmt=instr <binary>

# show disassembly of binary
patmos-llvm-objdump -r -d <binary> | less

# compile with debug infos, show source line numbers
patmos-clang -g -o <binary> ...
readelf --debug-dump=decodedline <binary>

# Compile with debugging info: use CFLAGS="-g" for your application, and add
# the following to your build.cfg:

NEWLIB_TARGET_CFLAGS="-g"
COMPILER_RT_CFLAGS="-g"

# Annotate objdump with source line numbes (this is quite slow at the moment)
patmos-llvm-objdump -r -d <binary> | patmos-dwarfdump <binary> | less

# Annotate simulation trace and stack-trace with line numbers
pasim --debug=<cycle-to-start-printing> --debug-fmt=instr <binary> 2>log.txt
cat log.txt | patmos-dwarfdump <binary>
```

8.2.4 Various options

Keep relocation infos in executable for objdump: (does not work with patmos-clang -g !)

```
patmos-clang -Xgold -q -o <binary> ....
patmos-llvm-objdump -r -d <binary> | less
```

8.3 platin – The Portable LLVM Annotation and Timing Toolkit

The platin toolkit provides a set of useful tools to process the information exported by the compiler in the PML format, with respect to timing analysis integration.

The usage of platin is:

```
platin <tool> <tool-options>
```

You can get help on a particular tool with either of

```
platin <tool> --help
platin help <tool>
```

Below we present a list of the most useful tools.

pml2ais
: Translates information of a PML file relevant to timing analysis to the AIS annotation format.

extract-symbols
: The compiler exports program information at a stage where the final memory layout is not yet defined. This tool reads the final executable and enhances the PML file with information on the final addresses of instructions and data.

analyze-trace
Based on the structural information of a program in the PML file, the trace analysis tool is capable of extracting flow fact hypotheses based on a simulation run. These are context-sensitive and include, e.g. observed loop bounds and function call targets.

transform
Transforms flow facts from bitcode to machine code level or simplifies a set of flow facts.

tool-config
Given a hardware model (in PML format), this tool outputs consistent hardware configuration options/parameters for use during compilation, simulation and WCET analysis.

pml-config
Create and modify a hardware model in PML format based either on the default configuration for a given target triple or on an existing PML hardware model.

pml
Provides validation, inspection and merge facilities for PML files.

visualize
Visualises structural information of the program in the different program representations.

wcet
A driver that starts WCET analysis from the command line.

In addition to the platin tools, another command-line utility, `patmos-clang-wcet`, is provided. This tool invokes the compiler (`patmos-clang`), timing analysis, and the compiler a second time (with intermediate calls to platin tools as necessary) for WCET-guided optimisations based on timing-analysis feedback.

8.3.1 The PML File Format

`platin` stores all internal information and configuration in *Platin Metainformation Language* (PML) files. PML files are *YAML* files that adhere to the PML schema. The schema file can be found in

```
llvm/tools/platin/lib/core/pml.yml
```

Many `platin` tools accept multiple PML input files. Multiple PML files can also be merged using the `pml` tool.

8.3.2 PML Architecture- and Tool Configuration

PML configuration files are used to set up tools such as the `pasim` or the `clang` compiler via the `tool-config` tool, and provides timing and cache information to the WCET analyses.

Typically, the configuration is stored in a separate PML file that is passed to the `platin` tools as additional input file using the `-i` option. Sample configuration files can be found in

```
llvm/tools/platin/etc/patmos
```

TODO: Martin is unhappy about those two examples: the ait version represents memory timing non of our platforms fulfills, the other version is for the depreciated DE2-70 and the timing info is for a page mode that we never used. Furthermore, those two examples are not the base for generation of a .pml file. Where do the defaults come from? If no one minds, I would like to delete those two.

There are three configuration sections:

- `machine-configuration`: The machine configuration defines memory size and timings, caches and memory areas. If no machine configuration is present, a default configuration will be used.
- `tool-configurations`: This section contains additional tool configurations and options.
- `analysis-configurations`: The analysis section sets up one or more WCET analyses. This section is currently work in progress.

The `machine-configuration` Section *TODO: What is the meaning of 8 bytes for the method cache and 4 bytes for the stack cache?*

```
---
format: pml-0.1
triple: patmos-unknown-unknown-elf
machine-configuration:
  memories:
    - name: "main"
      size: 0x200000
      transfer-size: 16
      read-latency: 0
      read-transfer-time: 21
      write-latency: 0
      write-transfer-time: 21
    - name: "local"
      size: 2048
      transfer-size: 4
      read-latency: 0
      read-transfer-time: 0
      write-latency: 0
      write-transfer-time: 0
  caches:
    - name: "data-cache"
      block-size: 16
      associativity: 1
      size: 2048
      policy: "lru"
      type: "set-associative"
    - name: "method-cache"
      block-size: 8
      associativity: 16
      size: 4096
      policy: "fifo"
      type: "method-cache"
    - name: "stack-cache"
      block-size: 4
      size: 2048
      type: "stack-cache"
  memory-areas:
    - name: "code"
      type: "code"
      memory: "main"
      cache: "method-cache"
      address-range:
        min: 0
        max: 0x200000
    - name: "data"
      type: "data"
      memory: "main"
      cache: "data-cache"
      address-range:
        min: 0
        max: 0x200000
```

```
attributes:
  - key: "heap-end"
    value: 0x100000
  - key: "stack-base"
    value: 0x200000
  - key: "shadow-stack-base"
    value: 0x1f8000
```

The machine configuration consists of three sections:

- `memories`: This section specifies the available memories and their timings. Each entry must define the following properties of the memory:
  - `name`: The (unique) name of the memory.
  - `size`: The size of the memory in bytes. Can be a hexadecimal value.
  - `transfer-size`: The size of a single beat in bytes. Also defines the alignment of the memory.
  - `min-burst-size`: The minimum size of a single burst in bytes. Defaults to `transfer-size`.
  - `max-burst-size`: The maximum size of a single burst in bytes. Defaults to `min-burst-size`.
  - `read-latency`: The latency per read *request* in cycles.
  - `read-transfer-time`: The number of cycles to read a single beat of size `transfer-size`.
  - `write-latency`: The latency per write *request* in cycles.
  - `write-transfer-time`: The number of cycles to write a single beat of size `transfer-size`.

  The number of cycles for a single, `max-burst-size`-aligned read or write request of B bytes is

  $$t_{req} = \left\lceil \frac{\max(B, \texttt{min-burst-size})}{\texttt{max-burst-size}} \right\rceil \cdot \texttt{latency} + \left\lceil \frac{\max(B, \texttt{min-burst-size})}{\texttt{transfer-size}} \right\rceil \cdot \texttt{transfer-time}$$

  For unaligned requests that span over a single burst, the transfer time can increase by up to `latency` + `transfer-time`. *TODO: Which component does an unaligned request? What is exactly a request?*

  The caches of Patmos exchange data with the main memory in bursts of constant size, which is a power of 2. The time for this burst shall be configured with `transfer-time` and `latency` shall be set to 0. The default configuration of Patmos is 4 32-bit word bursts. Therefore, the `transfer-size` is 16 bytes. On the DE2-115 with the external 16-bit SRAM a burst transfer takes 21 clock cycles.

  Ideal memory: A memory is *ideal*, if all latency and transfer time delays are set to zero.

> **Patmos:** For a Patmos machine configuration, there must be exactly one memory named `main`. This memory configuration is used to setup the global memory. If `main` does not exist, an ideal memory is assumed.
>
> The memory configuration named `local` is used to setup the local scratchpad memory. Currently, the local memory must be an ideal memory.
>
> All other memory configurations are *ignored*.

- `caches`: This section defines all caches of the core. Each entry must define the following properties:
  - `name`: The (unique) name of the cache.
  - `type`: The cache type. Supported values are `none`, `set-associative`, `method-cache` and `stack-cache`. If set to `none`, the cache is disabled. This is equivalent to this cache's section not being present.

- policy: The replacement policy of the cache. Supported values are ideal, lru and fifo for a method cache or a set-associative cache. For set-associative caches, the policy dm (direct mapped) is also supported. For a stack cache, supported values are ideal, block, lblock, ablock and dcache. An ideal cache will always hit. In particular, it does *not* have cold misses. In order to simulate an ideal cache with cold misses, setup a very large fully associative cache (a very large direct mapped data cache is ideal for the simulation, but not ideal for the analysis as it is not resilient against unknown accesses).
- associativity: Defines the associativity of a lru or fifo set-associative cache, or the tag memory size for a method cache. Ignored for all other types of caches.
- size: The size of the cache in bytes. Ignored for ideal caches.
- block-size: The size of a cache line for set-associative caches, or the internal alignment of cache blocks for stack caches and method caches, in bytes. A block size of 4 (or less) corresponds to a variable-sized method cache. A block size of size/associativity corresponds to a fixed-size method cache. For set-associative caches, the block size will typically be the same as the underlying memory transfer size.

  At the moment, the block-size is always required to be set.
- attributes: A list of additional attributes as key and value pairs.

> **Patmos:** Patmos supports the following cache names: data-cache, stack-cache, method-cache and instruction-cache. All other caches are *ignored*.
>
> The data-cache must be a set-associative cache. The stack-cache must be of type stack-cache. If a data cache but no stack cache is configured, requests to the stack cache will be handled through the data cache. The method-cache must be a method cache. The instruction-cache must be a set-associative cache.
>
> It only allowed to specify both a method-cache and an instruction-cache if at least one of them has the type set to none.
>
> The platin pml-config tool can be used to easily switch between a method-cache and an instruction-cache configuration. It will automatically set the correct types and link the caches to the memory areas.
>
> *Attention:* If an ideal data cache is configured, all *stores* through the data cache also have a zero-cycle latency, but bypass loads (and bypass stores) are unaffected. Bypass loads and stores only have a zero-cycle latency if the main memory is configured as ideal memory. In this case, all data and code accesses have zero latency, regardless of the configured caches.

- memory-areas: The memory areas set up a mapping of address ranges to memories and define the caches that are used for those address ranges and access types. Each mapping must define the following properties:
  - name: The (unique) name of the cache.
  - type: The type of the contained data, must be one of code or data.
  - cache: The name of the cache that is used for this address range and type of data.
  - memory: The name of the memory of this address range is mapped to.
  - address-range: The start (inclusive) and the end (exclusive) of the address range as property min and max. Defaults to 0 to the size of the backing memory if omitted.
  - address-space: The name of the address space. Defaults to global.
  - attributes: A list of additional attributes as key and value pairs.

Patmos: The address space must be either `global` or `local`. Local address space entries must not use a cache, and must use an ideal memory.

Patmos supports the following named address spaces:

- `code`: The main code area for regular instruction fetches. It must be of type `code`, use global address space, and refer to `main` memory or an ideal memory. It must use either the `method-cache` or the `instruction-cache` such a cache is defined (with a type different than `none`).
- `data`: The main data area for data and stack accesses. It must be of type `data`, use global address space, and refer to `main` memory. It must use the `data-cache` if such a cache is defined. This area may only use an ideal memory if the `main` memory is ideal.

All other code areas are currently ignored. If any of the above address spaces is omitted, `main` memory is used by default.

The following attributes are supported: `heap-end` sets the heap end pointer, `stack-base` sets the stack base pointer, and `shadow-stack-base` sets the shadow stack base pointer. Those attributes can be defined in any memory area, but must be defined at most once.

The `tool-configurations` Section This section can be used to specify additional command line options or configuration values for various tools.

```
tool-configurations:
  - name: "clang"
    options: ["-mpatmos-disable-vliw"]
```

The section contains a list of tool configurations, consisting of the following entries:

- `name` (required): The name of the tool to configure. Corresponds to `tool-config -t`. Currently supported tools are `clang`, `pasim` and `ait`.
- `options`: A list of command line options that should be passed to the tool when configured via `tool-config` or when called internally. Note that if you want to pass multiple options, the options must be specified as a YAML list, not as a single string, in order to be quoted correctly.
- `configuration`: A list of entries with `key` and `value`, specifying additional tool configuration values. The names of the keys are tool dependent. Currently this is not used by any tool.

The `analysis-configurations` Section The analysis configuration is intended to set up one or more WCET analysis variants, either for multiple analysis targets, or for analysing with multiple scenarios or analysis configurations. For each named analysis, a program entry point, a (WCET) analysis entry point, as well as additional tool configurations per analysis can be specified.

Attention: This section is currently work in progress and is only used to configure clang, but not the WCET analyses.

8.3.3 Generating PML configurations

It is possible to generate or modify PML hardware models using `platin pml-config`.

```
platin pml-config --target patmos-unknown-unknown-elf -o config.pml -m 2k \
                  --set-cache-attr method-cache,max-subfunction-size,1024 \
                  --set-area-attr data,heap-end,0x18000
```

This creates a new default configuration and sets the method cache size to 2 kB and defines (or redefines) some attributes. The result is stored in `config.pml`. It is also possible to print the result to stdout using `-o -`.

It is also possible to modify an existing configuration. This command takes the previously generated PML file, disables the method cache and creates a new instruction cache entry (with default values) and sets its size to 8 kB (without changing the disable method-cache entry). The result is written to stdout.

```
platin pml-config -i config.pml -o - -C icache -m 8k
```

Attention: At the moment `pml-config` is the only tool that includes a hardware model in its PML output. All other tools will *skip* any configuration section in its output. It is thus necessary to always explicitly pass the configuration to platin, i.e., the following will not work as expected

```
platin pml -o merged.pml -i config.pml -i myprogram.pml
platin wcet -b myprogram -i merged.pml   # This uses a default configuration!
```

because the merged file will not contain the configuration sections. Instead, the second command in this example also needs the additional `-i config.pml` argument. This is because `patmos-llvm` currently does not support importing PML files containing configuration sections, and may change in the future.

8.3.4 Exporting PML Metainfo During Compilation

To obtain PML files, the `patmos-clang` driver needs to be invoked with

```
patmos-clang -mserialize=<pml-file> [-mserialize-roots=<functions>] ...
```

The option argument `<pml-file>` is the filename of the PML file that is generated.

The option argument `<functions>` is a comma-separated list of function names to which the exporting of Metainformation should be restricted. Note that inlining should be prevented for those functions (see Section 8.4), otherwise they might be inlined and removed. By default, information for all functions reachable from main is exported.

8.3.5 Obtaining AIS Annotations

The `aiT` timing analysis tool supports annotations in the form of AIS files. To generate a AIS annotation file from a PML metainfo file, the `platin pml2ais` tool is used:

```
platin pml2ais  --ais <output.ais>  <input.pml>
```

8.3.6 Exporting Loop Bounds

Loop bounds obtained by the *LLVM scalar evolution (SCEV)* analysis on bitcode are exported as meta information. To be usable as flow facts for WCET analysis, they must be resolved and transformed to machine-code level:

```
platin transform --transform-action=down --flow-fact-output=<name> \
          --analysis-entry=<func> -i <input.pml> -o <output.pml>
```

where `<name>` is a name for the newly generated flow facts `<func>` is the entry function enclosing the program points referred by the flow facts (*main* by default). `<output.pml>` and `<input.pml>` can be identical.

8.3.7 Example

In this section we demonstrate some of the tools of `platin`. We show a typical workflow by compiling and analysing a small demo application on Patmos.

Listing 8.1 shows the content of `sort.c`. It contains a simple insertion sort implementation in function `sort`. Our target function for analysis is `gen_sort`, which fills an array with N pseudo-random numbers and then sorts the array. In order to prevent the compiler from inlining and removing our analysis target function, we mark the function as `noinline`. The code contains loop bound annotations for the WCET analysis in the form of pragmas.

All tools in the Patmos tool chain are configured to use the default Patmos hardware configuration if no further options are given. In this example we show how to use `platin` to configure a different hardware setup. For this, we use `pml-config` to generate a modified hardware model:

Listing 8.1: Demo application that initialises and sorts an array.

```
#include <stdlib.h>

#define MAX_SIZE 100

void sort(int *arr, size_t N) {
  #pragma loopbound min 0 max 99
  for (int j = 1; j < N; j++) {
    int i = j - 1;
    int v = arr[j];
    #pragma loopbound min 0 max 99
    while (i >= 0 && arr[i] >= v) {
      arr[i+1] = arr[i];
      i = i - 1;
    }
    arr[i+1] = v;
  }
}
void gen_sort(int *arr, size_t N) __attribute__((noinline));
void gen_sort(int *arr, size_t N) {
  #pragma loopbound min 1 max MAX_SIZE
  for (size_t i = 0; i < N; i++) {
    arr[i] = rand() % N;
  }
  sort(arr, N);
}
int main(int argc, char** argv) {
  srand(0);
  int arr[MAX_SIZE];
  size_t N = rand() % (MAX_SIZE / 2) + (MAX_SIZE / 2);

  gen_sort(arr, N);

  return 0;
}
```

```
platin pml-config --target patmos-unknown-unknown-elf \
                  -o config.pml -m 2k -M fifo8
```

This command generates a new `config.pml` file containing a description of the default hardware model, except that we use a method cache of only half the size (2 KB size with a tag memory of 8 entries).

In the next step, we compile our program using the `patmos-clang` compiler driver. We also use the `platin tool-config` tool to setup the compiler according to our modified hardware model. `tool-config` can be used in a similar manner to setup `pasim`, the Patmos simulator. We need to explicitly enable optimisations with `-O2`, as the default optimisation level is `-O0`.

```
patmos-clang ‘platin tool-config -i config.pml -t clang‘ \
             -O2 -o sort -mserialize=sort.pml sort.c
```

The driver calls all commands necessary to compile the source code, link and optimise the bitcode and generate and link the final binary `sort`. The option `-mserialize` causes the compiler to generate the PML file `sort.pml`. It contains a description of the application control flow at bitcode level (after the bitcode optimisations) and of the

Listing 8.2: Analysis report for the sort application

```
---
- analysis-entry: gen_sort
  source: trace
  cycles: 49089
- analysis-entry: gen_sort
  source: platin
  cycles: 644867
  cache-max-cycles-instr: 651
  cache-min-hits-instr: 398
  cache-max-misses-instr: 3
  cache-max-cycles-stack: 0
  cache-max-misses-stack: 0
  cache-max-cycles-data: 436779
  cache-min-hits-data: 0
  cache-max-misses-data: 10599
  cache-max-stores-data: 10200
  cache-unknown-address-data: 20799
  cache-max-cycles: 437430
```

final machine code. It also contains value facts and flow facts such as loop bounds as found by the compiler as well as our source-code loop annotations, and relation graphs relating the bitcode and machine code control flow graphs.

Now we are ready to analyse our target function. We use the `platin wcet` driver tool to run all necessary commands, including the trace analysis and the platin WCET analysis tool WCA. The driver tool will automatically try to run the AbsInt aiT analysis tool if it is installed.

```
platin wcet -i config.pml --enable-trace-analysis --enable-wca \
            -b sort -e gen_sort -i sort.pml --outdir tmp \
            -o wcet.pml --report report.txt
```

We need to pass the name of the binary file (`-b`) and both the compiler generated PML file and the hardware model PML file (`-i`) to platin. The `-e` option tells platin the name of the analysis target function. The optional `-outdir` option causes platin to keep temporary files and store them in the given directory, mainly the generated project files for the AbsInt analyser tool `a3patmos`. The optional `-o` option stores detailed analysis results such as the found WCET bounds for the target function, execution timings of basic blocks and execution frequencies of blocks on the worst-case path along with the program information from the input files in a PML file for further analysis or for WCET-driven optimisations.

The `-report` option causes platin to store the result summaries of the analyses in `report.txt`. Listing 8.2 shows the content of that file. In this example the platin WCET analysis derives a lower WCET bound than aiT. aiT is able to find better loop bounds and thus finds fewer data cache misses for the sort loop, but it assumes higher costs for instruction cache misses than platin.

Both analyses seem to highly over-approximate the actual WCET when compared to the trace results of the execution. However, while we assume that in the worst case the whole array is used, the actual execution only fills and sorts a fraction of the array. Hence the measured execution time is not a good indicator for the worst-case performance.

The inner loop of the sort function is a triangle loop. Our annotated global loop bound of $(N-1)^2$ is thus about a factor of two too large. For loops with constant bounds, LLVM is capable of detecting such triangle loops and deriving the correct bounds automatically. Our PML export uses the LLVM analysis results to generate additional flow facts. platin provides a tool to print all flow facts in a PML file in a compact form.

```
platin pml -i sort.pml --print-flowfacts
```

Listing 8.3: Flow facts from LLVM and user annotations as reported by `platin`

```
=== flowfacts generated by llvm.bc ===
--- loop-bound ---
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: gen_sort/for.cond>:
  ↪ [1 gen_sort/for.cond] less-equal (1 + %N)>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: gen_sort/for.cond.i>:
  ↪ [1 gen_sort/for.cond.i] less-equal (1 umax %N)>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
=== flowfacts generated by user.bc ===
--- loop-bound ---
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/for.cond>:
  ↪ [1 gen_sort/for.cond] less-equal 101>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/for.cond.i>:
  ↪ [1 gen_sort/for.cond.i] less-equal 100>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/while.cond.i>:
  ↪ [1 gen_sort/while.cond.i] less-equal 100>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
```

Listing 8.3 shows the output of that command. We find our manual loop annotations in the `user.bc` origin section. Note that LLVM inlined the `sort()` function, therefore our loops are now in function `gen_sort`.[2] The loop bounds are expressed as flow constraints on the loop header blocks.[3] We can also see that LLVM managed to find parametric loop bounds for two loops, but failed to find a loop bound for the inner triangle loop since in our case the size of the array to sort is not fixed but parametric. It is thus necessary to annotate the inner loop manually. Platin supports arbitrary linear flow constraints in PML. It is possible to manually supply additional flow constraints in PML format. Support for source code flow annotations beyond local loop bounds in the Patmos compiler is planned for future development.

We can also use platin to visualise control-flow graphs, call-graphs and relation graphs:

```
platin visualize -i wcet.pml -o out -f gen_sort \
                 --show-timings=platin
```

This command generates all graphs for function `gen_sort` and stores them in the output directory `out`. Figure 8.2 shows the generated control-flow graphs at bitcode level (after optimisation) and of the final machine code. The latter graph is the same graph that is used for WCET analysis by platin. Square boxes correspond to basic blocks or basic block slices, while round boxes are virtual nodes inserted by platin. The block node labels in the machine code graph show the address and the number of the basic block, as well as the name of the corresponding bitcode block (in brackets) and the range of the instructions in the basic block slice (in square brackets). The `-show-timings` option causes platin to highlight blocks and edges that are on the worst-case path found by the given analysis tool in the machine-code graph. Edges between basic blocks are annotated with their worst-case execution frequency and their associated WCET contribution.

[2] Function `__umodsi3` implements the modulo operator, as Patmos does not provide a modulo instruction in hardware.

[3] The right-hand side of the constraint is larger than our loop bound by one because the loop header is executed one additional time more than the loop body to jump out of the loop when the loop condition becomes false.

8.4 Patmos-clang C Frontend

8.4.1 Inlining, Function Attributes

In contrast to GCC and the clang C frontend, LLVM by design attaches function attributes to function *definitions only*, not to function declarations or function types. It is thus not possible to attach function attributes at call sites or for external functions. Function attributes *must* be defined in the same unit as the definition, otherwise they are silently discarded by the compiler.

Inlining functions The compiler follows the C99 rules for inlining. See here for an explanation: `http://www.greenend.org.uk/rjk/tech/inline.html`

If a function is marked with `inline` only, it will not be emitted into the linked binary. Thus, to mark functions as inline functions you must do one of the following:

- If the function is only used within one module, mark it as `static inline`. The function will not be visible outside the module, like all static functions. The compiler will emit the function into the module if it is used.

  ```
  static inline void foo(int n) {
      ...
  }
  ```

- If the function should be used in several functions, define it 'inline' everywhere, and add a declaration or definition with 'extern inline' in exactly one module.

  ```
  extern inline void foo(int n);

  inline void foo(int n) {
      ...
  }
  ```

Prevent Inlining To prevent the compiler from inlining, use the noinline attribute.

```
void foo(int n) __attribute__((noinline));
void foo(int n) {
    ...
}
```

Note that function attributes are attached to function definitions only, as mentioned above. It is currently not possible to prevent function inlining at the call site itself. The only way to prevent inlining for specific functions is to either define them `noinline` at their *definition*, or to make the call an inline asm call. However, at the moment inline asm is currently not supported by the platin analysis tools.

Marking Functions as Used To prevent the compiler from removing functions that have no call site in the bitcode (either because they are entry functions or because the compiler generates the calls), add the 'used' attribute to the function declaration.

```
void _start(void) __attribute__((used));
void _start(void) {
    ...
}
```

Note that if the function is part of a module that is linked in from a bitcode archive, the compiler will not link in the module if there is no usage of the function, even if it is marked as used. To force the linker to link in functions from archives, add a declaration for that function in any of your used modules, or link a bitcode module just containing declarations for those functions before linking with the library.

8.4.2 Target Triples and Target Identification

The Patmos tool-chain supports to following target triples:

```
patmos-unknown-unknown-elf      Do not use an OS, start with main() on bare metal
patmos-unknown-rtems            Compile and link for RTEMS
```

The C frontend defines the following macros for Patmos targets

```
__PATMOS__
__patmos__
```

For RTEMS, the following macros are also defined:

```
__rtems__
```

Use the following command to get a list of all defines for a target (do not omit `-triple`):

```
patmos-clang -cc1 -triple patmos-unknown-unknown-elf -E -dM </dev/null
```

The default target triple for patmos-clang (without -cc1!) is `patmos-unknown-unknown-elf`, if the program is called `patmos-clang`. Otherwise, if the binary is called `<target>-clang`, then `<target>` is used as default target triple if it is a valid triple. Otherwise, the host architecture (defined at configure time) will be used.

8.4.3 Inline Assembler

Inline assembly syntax is similar to GCC inline assembly. It uses %0, %1, ... as placeholders for operands. Accepted register constraints are: `r` or `R` for any general purpose register, `{$<registername>}` to use a specific register, `i` for immediates, or the index of an output register to assign an input register the same register as the output register.

Example:

```
int i, j, k;
asm("mov  $r31 = %1  # copy i into r31\n\t"
    "add  %0 = $r5, %2\n\t"
    "call %3\n\t"        // call myfunction
    "nop ; nop \n\t"     // delay slots
    : "=r" (j)
    : "0" (i), "{$r10}" (k), "i" (&myfunction)
    : "$r5" );
```

Please see Section 8.5 for a description of the Patmos assembler syntax.

8.4.4 Naked Functions

You can mark functions as naked to prevent the generation of a prologue, epilogue or any spill code. In such functions, effectively only inline assembly is allowed. It is possible to use simple C code in naked functions, as long as the compiler does not need to spill registers. Note that the amount of spills generated by the compiler depends on the optimization settings, i.e., naked functions containing C code might not compile at `-O0`. In particular, life ranges of variables must not extend over basic blocks, over calls or over inline assembly code.

Note that the compiler might choose to inline functions into naked functions. To prevent this, put your C code into a separate function that is marked as `noinline`, and only call this function from the naked function.

At the time of writing, the compiler automatically inserts a return instruction in naked functions. This behavior might be changed in the future, i.e., naked functions should either contain an explicit return statement or a `RET` instruction in the inline assembler whenever the control flow reaches the end of the function.

```
void foo(int n) __attribute__((naked));
void foo(int n) {
    asm("nop");
}
```

8.4.5 Patmos Specific IO Functions

The following header define functions to read out the CPU ID, the clock counter and the real-time clock (RTC), as well as to interface with the UART and to setup exception and interrupt handler.

```
#include <machine/patmos.h>
#include <machine/uart.h>
#include <machine/exceptions.h>
```

Please refer to the headers in `patmos-newlib/newlib/libc/machine/patmos/machine/` for documentation for now.

8.4.6 Scratchpad Memory

Use the following header to get the relevant functions and macros:

```
#include <machine/spm.h>
```

The `_SPM` macro must be used for all pointers that point into the SPM.

```
_SPM unsigned int *spm_data = (_SPM unsigned int*) 0x1234;
```

You can use the `spm_copy_from_ext` and `spm_copy_to_ext` functions to copy data from global memory to SPM and back. Use `spm_wait()` to wait for the copy transaction to complete.[4]

8.4.7 Placing Functions into the Instruction Scratchpad

Functions that should be placed into and executed from the instruction scratchpad must be placed at a specific address range (see Table 3.2). This can be achieved by assigning such functions to a different segment in the ELF objects, and then instructing the linker to place these segments at a different address.

TODO: Explain how those functions are loaded into the ISPM. By the boot loader?

Functions can be assigned to a segment using the following attribute.

```
void foo(int i) __attribute__((section(".text.spm")));
void foo(int i) {
    ...
}
```

While the section name can be arbitrary, it should start with `.text.` so that functions are placed into the text segment by default. The `.text.spm` segment name is used by some functions in `patmos-newlib`.

By default, any `.text.*` section will be linked in at the end of the text segment in the executable binary. In order to define the address of the section, a linker script must be used and provided at the linking stage. Such a linker script is provided in `patmos/hardware/spm_ram.t`.

```
SECTIONS
{
  . = SEGMENT_START(".rodata", 0x400);
  .rodata : { *(.rodata .rodata.*) }
  .init_array : { *(SORT(.init_array.*) .init_array) }
  .fini_array : { *(SORT(.fini_array.*) .fini_array) }
  .data : { *(.data) }
  .bss : { *(.bss) }

  . = ALIGN(8);
```

[4] At the moment memory copy is performed by the processor and is a blocking function. In future versions of Patmos this might be delegated to a DMA and then the wait function might be needed.

Name	Value	Description
R_PATMOS_NONE	0	no relocation
R_PATMOS_CFLB_ABS	1	CFLb format (22 bit immediate), absolute (unsigned), in words
R_PATMOS_CFLB_PCREL	2	CFLb format (22 bit immediate), PC relative (signed), in words
R_PATMOS_ALUI_ABS	3	ALIi format (12 bit immediate), absolute (unsigned), in bytes
R_PATMOS_ALUI_PCREL	4	ALUi format (12 bit immediate), PC relative (signed), in bytes
R_PATMOS_ALUL_ABS	5	ALUl format (32 bti immediate), absolute (unsigned), in bytes
R_PATMOS_ALUL_PCREL	6	ALUl format (32 bit immediate), PC relative (signed), in bytes
R_PATMOS_MEMB_ABS	7	LDT or STT format (7 bit immediate), signed, in bytes
R_PATMOS_MEMH_ABS	8	LDT or STT format (7 bit immediate), signed, in half-words
R_PATMOS_MEMW_ABS	9	LDT or STT format (7 bit immediate), signed, in words
R_PATMOS_ABS_32	10	32 bit word, absolute (unsigned), in bytes
R_PATMOS_PCREL_32	11	32 bit word, PC relative (signed), in bytes

Table 8.2: ELF relocation types

```
  _end = .; PROVIDE (end = .);

  . = SEGMENT_START(".text.spm", 0x10000);
  .text.spm : { *(.text.spm) }

  . = SEGMENT_START(".text", 0x20000);
  .text : { *(.text .text.* ) }

}
```

The linker script must then be passed to the linker.

```
patmos-clang -Xgold -T -Xgold path/to/spm_ram.t -o hello hello.o
```

Note that the instruction scratchpad is currently not available in the multicore version of Patmos. *TODO: why?*

8.5 Patmos Compiler Backend

8.5.1 ELF File Format

ELF Identification:

`e_machine: EM_PATMOS = 0xBEEB`

ELF Relocation infos: Table 8.2 shows the ELF relocation types.

Subfunctions, Symbols:

- ELF Symbol flags:
  - `MCSA_ELF_TypeCode` / `STT_CODE` (value 13): set for symbols which point to the beginning of a (sub) function (i.e., the first instruction after the alignment and function size word)
- Function symbol points to first instruction of function, has `.type function`, `.size` is whole function size
- Code symbol points to first instruction of subfunction, has `.type code`, `.size` is size of subfunction
- The first subfunction of a function only has a function symbol, following subfunctions have a code symbol (i.e., the size value for the first subfunction in the symbol is not the same as the actual size)

8.5.2 LLVM backend fixups, symbols, immediates

At MC level, immediates are always in byte/half-word/word as the instruction where they are used, i.e., immediates are already properly shifted.

The assembler parser and assembler printer (i.e., the disassember and .s emitter) parse and print immedates without conversion, i.e., immediates are printed in words/half-words/bytes, depending on the instruction.

8.5.3 Assembler Syntax

This section describes the assembler syntax of the LLVM assembler parser and printer, as well as the inline assembler.

Note that the `paasm` assembler provided with Patmos has a slightly different syntax, i.e., opcode mnemonics have suffixes, the syntax for bundles is different, and only a very limited set of directives is accepted by `paasm`.

General Instruction Syntax: Each operation is predicated, the predicate register is specified before the operation in parentheses, e.g. `(p1) <instruction>`. If the predicate register is prefixed by a `!`, it is negated. If omitted, the predicate defaults `(p0)`, i.e., always true.

All register names must be prefixed by $. The instructions use destination before source in the instructions. Between destination and source a = character must be used instead of a comma.

Immediate values are not prefixed for decimal notation, the usual 0 and 0x formats are accepted for octal and hexadecimal immediates. Comments start with the hash symbol # and are considered to the end of the line.

For memory operations, the syntax is `[$register + offset]`. Register or offset can be ommited, in that case the zero register `r0` or an offset of 0 is used.

Labels that are prefixed by .L are local labels. Labels may only appear between bundles, not inside bundles.

Example:

```
    # add 42 to contents of r2
    # and store result in r1 (first slot)
    { add   $r1 = $r2, $42
    # if r3 equals 50, set p1 to true
    cmpeq $p1, $r3, 50 }
    # if p1 is true, jump to label_1
    ($p1) br .Llabel1 ; nop ; nop   # then wait 2 cycles
    # Load the address of a symbol into r2
    li $r2 = .L.str2
    # perform a memory store and a pred op
    { swc [$r31 + 2] = $r3 ; or $p1 = !$p2, $p3 }
    ...
.Llabel1:
    ...
```

Bundles: A semi-colon `;` or a newline denotes the end of an instruction or operation. If an instruction contains two operations, the operations in the bundle must be enclosed by curly brackets. For bundles consisting only of one operation, the brackets are optional.

Known bugs: The closing bracket must appear on the same line as the last operation in the bundle. The opening bracket might be followed by a newline, but no comments or labels may appear between the bracket and the first operation.

Function Block Start Markers and Subfunction Calls: Functions must be prepended by the `.fstart` directive that emits the function size word and aligns the code.

```
    .fstart <label>, <size-in-bytes>, <alignment-in-bytes>
```

The alignment must be a power of 2. The function size must be the size of the following (sub-)function. If a function is split into several subfunctions, the size must be the size of the first sub-function, not the size of the whole function (this differs from the .size directive).

```
    .type   foo,@function
    .size   foo, .Ltmp2-foo         # size of foo symbol is the whole function
    .fstart foo, .Ltmp0-foo, 4
foo:                                # start of foo function
    sres  10
    ...
    brcf .Ltmp1                     # Fallthrough to different subfunction is not allowed
    nop
    nop
    nop
.Ltmp0:                             # end of first subfunction of foo

    .type   .Ltmp1,@code
    .size   .Ltmp1, .Ltmp2-.Ltmp1
    .fstart .Ltmp1, .Ltmp2-.Ltmp1, 4
.Ltmp1:                             # start of second subfunction of foo
    ...
    ret $r30, $r31                  # returns from foo, not from the subfunction
    nop
    nop
    nop
.Ltmp2:                             # end of (second subfunction of) foo
```

To set the address of a function relative to the start of the section, use the .org directive before the .fstart directive and allow for the function size word so that .fstart does not emit any padding.

```
    .org <aligned startaddress>-4
    .fstart .foo, .Ltmp0-.foo, <alignment>
foo:
    ....
```

8.5.4 Address Spaces

Set address space of a pointer by using `__attribute__((address_space(<nr>)))`. See `patmos.h` in newlib.

Used address spaces:

- Address Space 0 (default): main memory with data cache
  - nontemporal flag: main memory with bypass Set only by the compiler (at the moment)
- Address Space 1: (local) scratchpad memory
  - use macro `_SPM` defined in `<machine/spm.h>` for SPM accesses
  - use macro `_IODEV` defined in `<machine/patmos.h>` to access memory mapped IO devices
- Address Space 2: Stack cache
  - Compiler-maintained, must not be used in application code (at the moment)
- Address Space 3: main memory with data cache bypass
  - use macro `_UNCACHED` defined in `<machine/patmos.h>`

8.6 Newlib

The Patmos compiler contains a port of newlib, a C library intended for embedded systems. When writing downloadable applications it is suggested to use newlib functions instead of the low-level functions provided by Patmos specific IO functions.[5]

Documentation of the provided newlib library and available functions is available at:
`https://sourceware.org/newlib/`

8.7 Known Bugs, Restrictions and Common Issues

Undefined reference to <libc function>: Some LLVM passes might create calls to standard library functions after the bitcode linking phase. We added all such functions that we found to `libcsyms.ll.in` in `compiler-rt`. If we missed some functions, they must be added. Alternatively, newlib and compiler-rt could be compiled as ELF libraries.

It could also be the case that newlib needs to be recompiled, or that your linking order is wrong (be aware that mixing bitcode or C files, assembly files and ELF files causes the linking order to be changed).

Inline assembler: Clobbering the registers `$r30/$r31` is not supported and calls inside inline assembler will not cause the prologue to save `$r30/$31`. Do not modify them in inline assembly.

Inline assembler: Constraining an output to a register (`"=$r10"`) does not work, for some reason LLVM looses the output register operand somewhere between `SelectionDAGBuilder::visitInlineAsm()` and `AsmPrinter::EmitInlineAsm()`

C code in __naked__ functions: Writing C code statements in naked functions might cause the compiler to spill registers (be aware that the compiler will spill much more registers at `-O0`!). This is not supported since naked function do not have a prologue or epilogue setting up the stack frame.

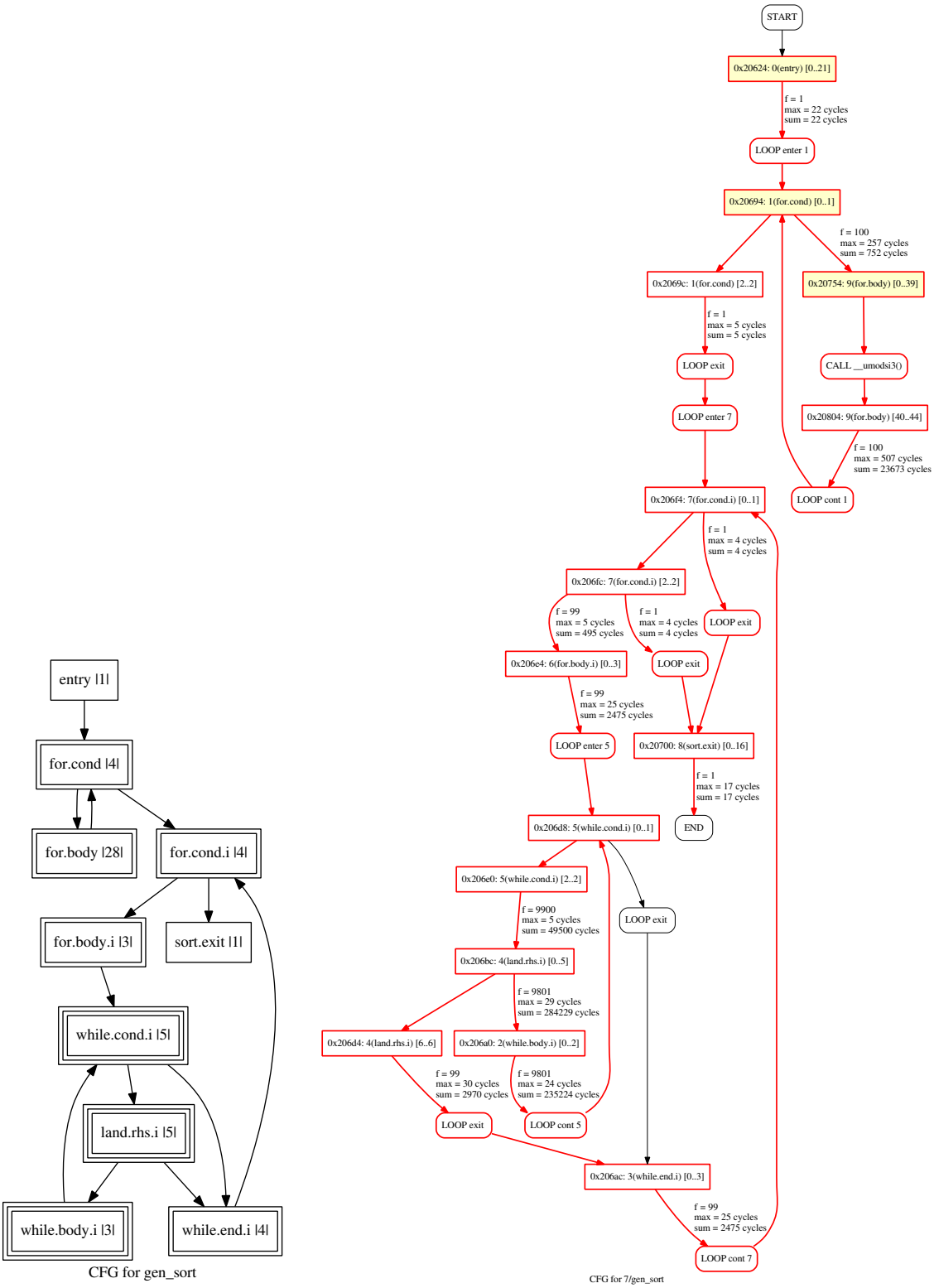
patmos-{objdump,objcopy,..} does not support Patmos ELF files: Only patmos-ld supports the Patmos ELF file type. patmos-ar and patmos-nm have some support for bitcode archives (when the LLVMgold plugin is compiled, default for build.sh builds). Other binutils tools have no support for Patmos ELFs. Use `patmos-llvm-objdump` and `patmos-ld` instead.

Compiling patmos-gold with GCC 4.7.0 aborts with an error about narrowing conversion: Workaround: use CXXFLAGS=-Wno-narrowing for configure, upgrade to a newer GCC version or use clang to compile the toolchain.

If you are using the build.sh script, set `GOLD_CXXFLAGS="-Wno-narrowing"` in `build.cfg`.

Keeping relocations in the executable (-Xgold -q) and debugging info (-g) do not work together: This seems to be a gold restriction.

[5] Those functions are mainly used to write very small footprint programs, such as the bootloader itself.



(a) Bitcode CFG

(b) Machine-code CFG with platin WCET results

Figure 8.2: Bitcode and machine-code control-flow graphs for `gen_sort`.

9 Potential Extensions

9.1 Multiply / Wait / Move from Special

- Attaching a ready flag with all special registers.
- Specify destination special register with all decoupled operations; the operation sets/resets the ready flag accordingly.
- Wait operates on ready flags of special registers.
- Merged variant of Wait + Move from Special
- Wait with 16-bit mask to wait for multiple outstanding results.

This would be nice since it would allow to reload all special registers from memory without going through the general purpose registers. It would be a unified interface for decoupled operations and give more freedom to handle parallel decoupled operations (pipelined multiplies, loads). We could apply this also to the general purpose registers instead of the special registers.

9.2 Bypass load checks data cache

Let the bypass load use the data cache if the data is cached. If the data is not in the cache, load it from main memory, but do not update the data cache (in contrast to the normal load). Therefore the compiler could use bypass to load data that will not be used a second time or that might have a negative impact on the cache analysis, but we still take advantage of the cache if the data is already in the cache.

9.3 Merged Stack Cache Operations and Function Return

This might require an additional special-purpose register(?) to track the size of the last reserve instruction (this register might also be set explicitly). However, it would might reduce the number of ensure instructions needed.

Another option would be to merge the return and stack free operations. Both instructions belong to the same function and, due to the simpler semantics of the free, the combination would be easier to implement.

9.4 Non-Blocking Stack Control Instructions

Currently, all stack control operations, except `sfree`, are blocking. It might be useful to define non-blocking variants or define them to be non-blocking in all cases.

It is questionable whether this would actually buy us anything. Most `sres` instructions will be followed by a store to the stack cache (spill of saved registers). It might be more profitable for `sens` instructions.

9.5 Freeze Cache Content

- Bypass load can be used to avoid cache updates, but not on per-context basis (we cannot lock the cache and then call any function and assume the function does not update the cache. Instead we would need to generate function variants that only use bypass loads).
- Method cache freeze? Or should we just use a I-SPM for this if we want instruction cache locking?

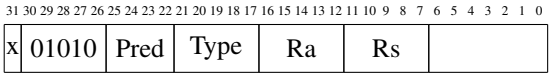
9.6 Unified Memory Access

Instead of having typed loads per cache or SPM, maybe have types per “use-case scenario”, use local memory based on address

- Type for stack access (guaranteed hit, can be used in both slots)
- Type for guaranteed hit (any local SPM access, access to data cache must be always hit, else undefined result)
- Type for unknown data access (access SPM or data cache, or main memory and update data-cache)
- Type for bypass (access SPM or main memory, do not allocate in data cache)
- Maybe a type for no-allocate (access SPM, data cache or main memory, but do not allocate data in cache; could be useful to prevent single loads from thrashing the cache), or use some sort of cache-lock instruction instead (could be useful to prevent a code sequence or function call from thrashing the cache)

9.7 DMA Interface

- Transfers between local and global memory
- Through special registers
- Alternatively, dedicated instructions `dmastart` and `dmalen`



Type	Mnemonic	Operation
00001	dma.sp	Start memory copy
		which contain the word
xx001	st.sc	Stack cache, no write through to
		main memory, never wait
xx010	st.sp	Scratchpad, no write through to

9.8 Data scratchpad

- Every core has its D-SPM with its own address range (all in the same address space), a core can write to the D-SPM of another core by writing to an address of the SPM of that core.
- Maybe have some sort of protection mechanism, to prevent cores from writing to any address in any remote SPM
- How do we handle writes to the same address by (remote) dma transfers and local writes (this might prevent local load and stores to the SPM from completing in a single cycle)?

9.9 Halt

9.10 Floating-Point Instructions

9.11 Prefetching

- For method cache
- For local memory

9.12 Data Caches

- Add a second (possibly larger), simple (direct mapped,..) data cache, to be used when the pointer address is known at compile time (i.e., a load does not destroy the whole cache state in the analysis), for array operations, ..
- We would need additional types in the load operation for that cache, but there are only two unused types left. Either use only blocking (?) loads and only word and byte (?) access, or replace some lesser used types, or even introduce a new opcode somehow..

9.13 Instruction scratchpad

- For instruction handlers or other code that should not destroy the method cache
- Could be used to store code that is executed at a call site (even if the method cache entry of the caller gets replaced)
- Replacement of code at runtime, statically scheduled or with some sort of software-controlled replacement strategy (maybe this could be used to prevent threads from destroying the I-cache of real-time tasks)
- Keep frequently used code on the I-SPM, can be used to do some sort of cache locking (instead of somehow locking the method cache).

9.14 Wired-AND/OR for predicates

Let `cmp` be some operation that sets a predicate `Pd` and is predicated by `Pred`. Then we could define the following variants:

```
cond:   if (Pred)           Pd = <cmp>
and:    if (Pred && !<cmp>) Pd = False
or:     if (Pred &&  <cmp>) Pd = True
uncond: Pd = Pred && <cmp>
```

Note that we do not need to read `Pd`, but the last variant uses `Pred` as input, not as write-enable signal. First variant is the normal (conditional) execution. The last variant forces `Pd` to `false` if `Pred` is false, thus saving the initialization of `Pd` or explicit `and` with `Pred` for code like

```
if (p1)  p2 = R1 < R2
if (p2)  ...
```

The other variants can be used to implement stuff like

```
if (a != 0 && b < 5) { .. }

p1  = cmpnez r1,    addi r3 = r0 + 5;
p1 &= cmplt r2, r3
```

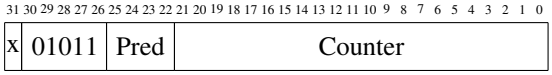
To implement `a != 0 && b > 1` we would need an additional bit that negates either the result of `<cmp>` or the value that is assigned to `Pd` (including the `true` and `false` assignments).

Note that if we do not need `Pred`, we can do `and` and `or` simply as

```
Pd &= <cmp>  ... if ( Pd)  Pd = <cmp>
Pd |= <cmp>  ... if (!Pd)  Pd = <cmp>
```

i.e., we use `Pd` as `Pred`.

9.15 Deadline instruction



Wait until the deadline counter reaches zero, then restart the counter with the given initial value.

10 Conclusion

Patmos is the next cool thing in the dry world of real-time systems.

Bibliography

[1] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[2] Accellera Systems Initiative. Open Core Protocol specification, release 3.0, 2013.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: constructing hardware in a scala embedded language. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[4] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.

[5] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

[6] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[7] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.

[8] I. Mohor. Ethernet IP core specification. Technical report, 2002. Revision 1.19, Available at `http://opencores.org/svnget,ethmac?file=%2Ftrunk%2F%2Fdoc%2Feth_speci.pdf`.

[9] L. Pezzarossa, J. K. Toft, J. Lønbæk, and R. Barnes. Implementation of an ethernet-based communication channel for the patmos processor. Technical report, Technical University of Denmark (DTU), 2015.

[10] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

[11] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[12] M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.

[13] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.