PicoBlaze C Compiler Toolchain 2.1

Users Guide

Optimizing C Compiler and an ELF-Based Toolchain for the PicoBlaze Processor

Academy of Sciences of the Czech Republic
Institute of Information Theory and Automation

Jaroslav Sýkora

PicoBlaze C Compiler Toolchain 2.1 Users Guide
Optimizing C Compiler and an ELF-Based Toolchain for the PicoBlaze Processor
Edition 2.1.0

Author Jaroslav Sýkora sykora@utia.cas.cz

Copyright © 2012 UTIA AV CR v.v.i. | This material (the User's Guide documentation) may only be distributed subject to the terms and conditions set forth in the Creative Commons Attribution-NoDerivs 3.0 Unported (CC BY-ND 3.0) license. (The license terms are available at *http://creativecommons.org/licenses/by-nd/3.0/*.)

Acknowledgement: This work has been supported from project *SMECY*[1], project number Artemis JU 100230 and MSMT 7H10001.

PicoBlaze is a simple RISC-style 8-bit processor core. The document describes the C compiler toolchain that includes the LLVM-based C compiler, assembler, and ELF-based linker. The C frontend supports the standard C99 syntax with GNU extensions (mainly the asm keyword). The compiler backend uses standard target-independent optimizations such as loop unrolling, inlining, and extensive constant propagation across loops. It supports code generation for KCPSM3, PB3A, and KCPSM6 processors. The PicoBlaze code generator performs register allocation and simple peep-hole optimizations. Function parameters are passed both in registers and on stack. Several illustrative program examples are given in the documentation.

[1] http://www.smecy.eu

Installation & Quickstart

1.1. Introduction

The toolchain is built around the *LLVM*[1] compiler framework and the *elfutils*[2] low-level binary tools. It consists primarily of the *optimizing C compiler* (*Chapter 2, Compiler*), gas-like *assembler* (*Chapter 3, Assembler*), and ELF-based *linker* (*Chapter 4, Linker and Friends*).

The C frontend supports the standard C99 syntax with GNU extensions (mainly the **`asm`** keyword). The compiler backend uses standard target-independent optimizations such as loop unrolling, inlining, and extensive constant propagation across loops. (In some cases it can compute loops during compilation, simplifing their results to run-time constant expressions.) The PicoBlaze code generator performs register allocation and simple peep-hole optimizations. Function parameters are passed both in registers (s0-s3) and on stack. Several illustrative program examples (see *Appendix A, Examples*) are given in this documentation.

The assembler is implemented within the LLVM framework as well. Its syntax is a mixture of the original KCPSM3 assembler syntax (opcodes, register names) and the standard *GNU as* assembler keywords (e.g. .section, .global, .equ, .comm). The assembler evaluates integer expressions during assembly as usual. It outputs fully relocatable ELF object files.

Linker works with standard ELF files. It can relocate and link object files (.o) with library archives (.a) to produce statically linked ELF programs.

The toolchain supports three PicoBlaze architectures: the original Xilinx *KCPSM3*, an extended *PB3A* based on an open-source KCPSM3-compatible clone, and the new Xilinx *KCPSM6*. The target is selected via the **`-march=`**{pblaze3, pblaze6} and **`-mcpu=`**{kcpsm3, pb3a, kcpsm6} options in the compiler command line. The PB3A processor (see *Chapter 5, Processor PB3A: Extended PicoBlaze*) has a new ADDN instruction that solves some stack-related issue in the compiler, and it supports a very handy *register+offset* addressing mode, among other extensions. It is based on the open-source *PacoBlaze v2.2*[3] processor; we provide a patch that can be applied on top of the PacoBlaze v2.2 source code. The patch itself is in Public Domain, and it is located in **`extra/hw/pblib/pacoblaze.diff`** file.

A simple HDL-based simulator framework is provided as well (*Chapter 6, Hardware Source and the Simulator*). It can simulate the processor VHDL (KCPSM3, KCPSM6) and Verilog (PB3A) model in the ModelSim simulator. This allows for quick verification of the compiler results.

1.2. Installation

To install the toolchain software simply add the **`bin/`** subdirectory into your **`$PATH`**. The other directories (lib, target) must remain in the relative position to the bin/ directory. For example, if the package contents is extracted in **`/opt/pblaze-cc`**, add the following command at the end of your **`~/.bash_profile`** script:

```
export PATH=$PATH:/opt/pblaze-cc/bin
```

[1] http://www.llvm.org
[2] https://fedorahosted.org/elfutils/
[3] http://bleyer.org/pacoblaze/

Important

It *SHOULD NOT* be needed to add the **`lib/`** subdirectory into the system library path. Doing so may cause problems to other unrelated programs in your system!

1.3. Examples

Several program examples are provided in the **`extra/examples`** directory. After installing the toolchain as described above, simply change into a directory (e.g. **`extra/examples/01-simple`**) and type `**`make`**' to compile the source code.

Some examples are also presented in *Appendix A, Examples*.

1.4. Basic Usage

The primary toolchain entry point is **`pblaze-cc`** driver program. Its command line options are similar to the gcc. To compile a C source into an executable (see e.g. **`extra/examples/01-simple`**):

```
pblaze-cc -mcpu=kcpsm3 -Wall -O3 main.c -o prog.elf
```

The **`-mcpu=`** option is mandatory to avoid confusing the target CPU flavors. Possible values for -mcpu include '*kcpsm3*' (original Xilinx PicoBlaze), '*pb3a*' (PacoBlaze with improved ISA, see *Chapter 5, Processor PB3A: Extended PicoBlaze*), and '*kcpsm6*' (new Xilinx KCPSM6). In the case -mcpu=kcpsm6 an additional option -march=pblaze6 has to be also specified. See *Table 2.2, "Supported CPU Architectures"*.

Contents of an ELF file can be disassembled using **`pblaze-objdump`**:

```
pblaze-objdump -d prog.elf
```

The **`.text`** section that contains the program memory can be extracted into a flat binary image using a common **`objcopy`** utility:

```
objcopy -I elf32-little -O binary -j .text  prog.elf  prog.bin
```

The resulting prog.bin file is the program memory in little-endian format, 4-bytes per each PicoBlaze instruction. This can be converted into a MEM file using the **`pblaze-bincopy`** tool:

```
pblaze-bincopy --format=mem  prog.bin  prog.mem
```

The **`pblaze-bincopy`** tool can also generate a C-style array file (.inc, .h) that can be included in a master CPU toolchain flow (e.g. MicroBlaze).

1.5. Processor Simulation

It is possible to simulate program execution in ModelSim using the harwdare models of the Picoblaze processor (kcpsm3 or pb3a). The source codes for the hardware library and simulator are provided in the **`extra/hw/`** directory. First follow the steps outlined in the **`extra/hw/pblib/README.txt`** file to download and patch third-party HDL source files. The ModelSim-based simulator can be used as described in *Chapter 6, Hardware Source and the Simulator*.

Compiler

2.1. General Toolchain Flow

The toolchain flow is shown in *Figure 2.1, "Toolchain flow"*. For most operations the compiler driver **`pblaze-cc`** can be used. The driver compiles C source (.c) and assembly source (.s) into ELF-based object files (.o). It can also be used for linking object files with archives into complete ELF programs (.elf). Program binary text can be extracted from the ELF program into a flat binary image (.bin) which contains PicoBlaze instructions in little-endian, 4 bytes-per-instruction format; this can be accomplished with a standard **`objcopy`** tool. Finally, the binary image is usually converted into a MEM format (.mem) that can be read by the HDL-based simulator (see *Chapter 6, Hardware Source and the Simulator*), or into a C-style array file (.inc, .h) that is included in another toolchain flow (e.g. MicroBlaze).

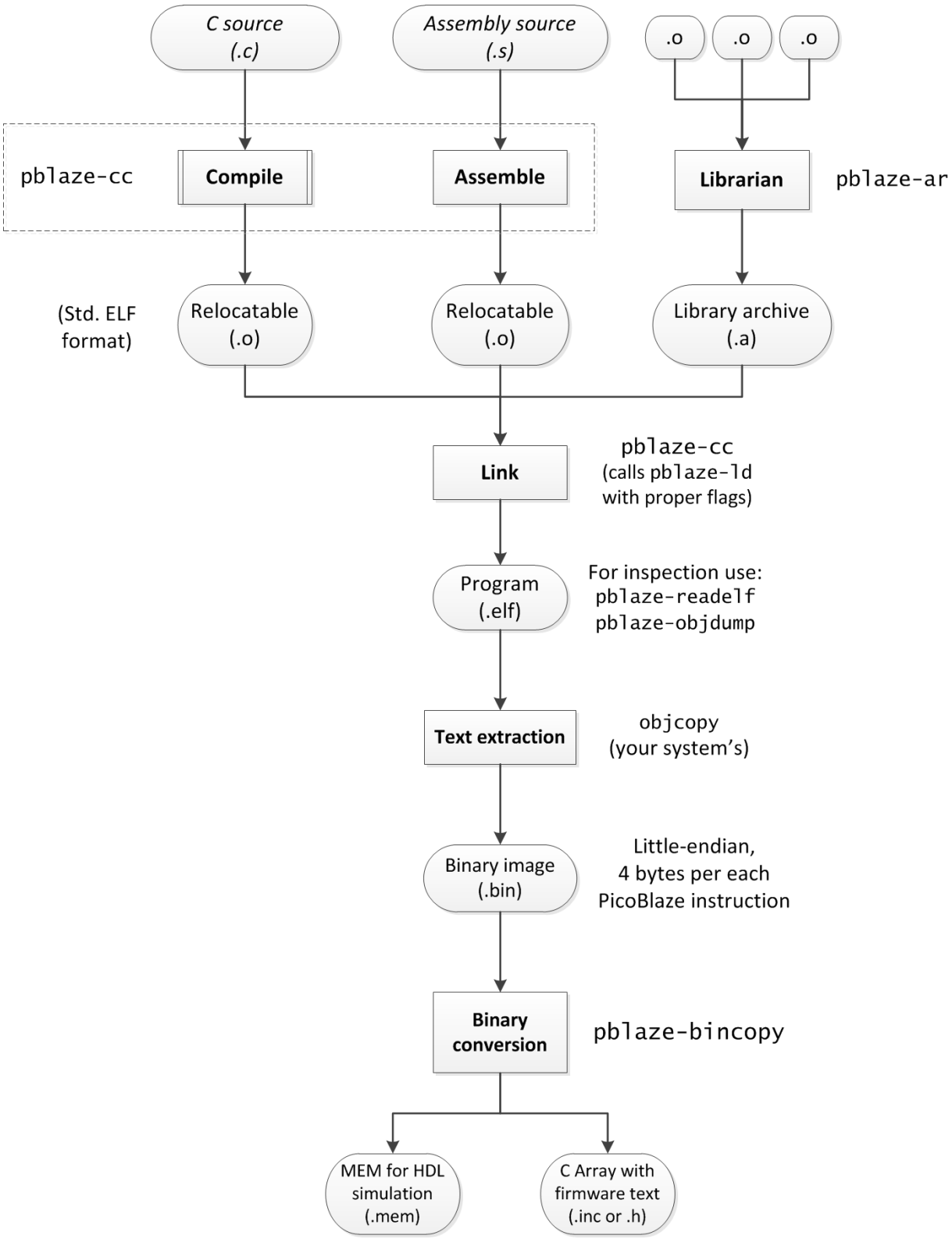

Figure 2.1. Toolchain flow

Table 2.1. List of all tool programs in the bin/ directory in the alphabetical order

Tool	Function
pblaze-ar	ELF librarian. Manages archives (.a).
pblaze-bincopy	Converts binary images (.bin) to .mem or .inc files.
pblaze-cc	Compiler driver, the main user-level tool.

Tool	Function
pblaze-clang	C compiler frontend. Compiles .c to LLVM IR.
pblaze-elfcmp	ELF comparison utility.
pblaze-elflint	ELF checking utility.
pblaze-ld	ELF linker. Links objects (.o) with archives (.a) into programs (.elf)
pblaze-llc	Compiler backed. Compiles LLVM IR to PBlaze assembly (.s)
pblaze-llvm-ar	LLVM IR librarian.
pblaze-llvm-as	LLVM IR assembler.
pblaze-llvm-diff	LLVM IR diff utility.
pblaze-llvm-dis	LLVM IR disassembler.
pblaze-llvm-extract	LLVM IR libary extraction.
pblaze-llvm-ld	LLVM IR linker.
pblaze-llvm-mc	Compiler machine code generator. Assembles PBlaze assembly (.s) to object files (.o).
pblaze-llvm-nm	LLVM IR names dump utility.
pblaze-llvm-objdump	LLVM IR dump.
pblaze-llvm-ranlib	LLVM IR library indexer.
pblaze-nm	ELF names dump utility.
pblaze-objdump	ELF object dump. Can dissasemble objects (.o) and programs (.elf).
pblaze-opt	Compiler optimizer.
pblaze-ranlib	ELF library indexer.
pblaze-readelf	ELF metainformation dumping.
pblaze-size	ELF section size extraction utility.
pblaze-strip	ELF symbol stripping.
pblaze-unstrip	ELF symbol unstripping.

Note

ELF utilities are based on the elfutils package and hence they are licensed under the GPL.

Important

Do not mix the LLVM IR tools and the ELF tools. They belong to distinct toolchain parts.

2.2. Compiler Driver (pblaze-cc)

Compiler driver (**pblaze-cc**) is a high-level program that automatically calls the lower-level tools with proper parameters and in correct order to simplify the user interface. It is modelled after the well-known **gcc** driver.

The compiler supports three target processor architectures: KCPSM3, PB3A, KCPSM6. To prevent ambiguity and confusion the compiler driver always requires to specify the target architectures via the **-mcpu** option. In the KCPSM6 case the option -march=pblaze6 has to be also specified. The *Table 2.2, "Supported CPU Architectures"* summarises all possible combinations.

Table 2.2. Supported CPU Architectures

CPU Name	-march	-mcpu
KCPSM3, PacoBlaze	-march=pblaze3	-mcpu=kcpsm3
PB3A	-march=pblaze3	-mcpu=pb3a
KCPSM6	-march=pblaze6	-mcpu=kcpsm6

2.2.1. Command Line Options

Basic command line options

-o <file>
: Output file name, for example **-o main.o**.

-E
: Stop after preprocessing, do not compile.

-S
: Stop after compilation, do not assemble

-c
: Stop after assemble, do not link. Outputs relocatable .o files.

-D <string>
: Define a macro.

-I <string>
: Add a directory to include path. The directory **target/include** is used by default.

-L <string>
: Add a directory to library path. The directory **target/lib-$(MCPU)** is used by default.

-O0, -O1, -O2, -O3
: Optimization Level 0 (do not optimize), 1, 2, 3.

-W <string>
: Enable the specified warnings. Usually used as **-Wall** to report all warnings.

-l <string>
: Specify a library to link. By default the library **libpblibc.a** is linked-in using **-lpblibc**

-march=<string>
: Target CPU architecture. One of 'pblaze3', 'pblaze6'. See *Table 2.2, "Supported CPU Architectures"*.

-mcpu=<string>
Target CPU. One of 'kcpsm3', 'pb3a', 'kcpsm6'. See *Table 2.2, "Supported CPU Architectures"*.

Advanced command line options

-B <string>
This option specifies where to find the executables, libraries, include files, and data files of the compiler itself.

-T <string>
Specify a linker script. The default linker script is located in **`target/elf32-pblaze.script`**.

-Wa,<string>
Pass options to assembler (Run 'pblaze-llvm-mc -help' for assembler options).

-Wl,<string>
Pass options to linker (Run 'pblaze-ld -help' for linker options).

-nostartfiles
Do not use the standard system startup files when linking. The file **`target/lib-$(MCPU)/startup.o`** is used when the option is not specified.

-nostdinc
Do not search the standard system directories for header files.

-nostdlib
Do not use the standard system libraries when linking (-lpblibc).

-save-temps
Keep temporary files.

-v
Enable verbose mode.

-x <string>
Specify the language of the following input files.

-w
Disable all warnings.

2.2.2. Example Makefile Rules

```
MCPU?=kcpsm3            # default CPU
CFLAGS?=-Wall -O3       # default options

# Compile .c to assembly .s
%.s : %.c
 pblaze-cc -mcpu=$(MCPU) -c -S $(CFLAGS) $< -o $@

# Assemble .s to objects .o
%.o : %.s
 pblaze-cc -mcpu=$(MCPU) -c $< -o $@

# Link all .o objects to a program .elf
$(ELF) : $(A_OBJS) $(C_OBJS)
 pblaze-cc -mcpu=$(MCPU) $^ -o $@

# Disassemble .elf to .dis
```

```
%.dis : %.elf
 pblaze-readelf -a $^  >  $@
 pblaze-objdump -d $<  >> $@

# Extract text from an .elf to a flat binary image .bin,
# and check that text limits (max 1024 instructions ~~ 4kB)
%.bin : %.elf
 objcopy -I elf32-little -O binary -j .text $^ $@
 @if [ `stat -c%s $@` -ge 4096 ]; then echo -e "\e[0;31mERROR: Firmware image file $@ is
 bigger than 1024 words (4kB)!!\e[00m"; fi

# Convert binary image .bin to a C header for inclusion in a microblaze program
%.inc : %.bin
 pblaze-bincopy --format=c $< $@

# Convert binary image .bin to a MEM file that can be loaded into a vhdl-based simulator
%.mem : %.bin
 pblaze-bincopy --format=mem $< $@
```

2.3. Compiler Limitations

The compiler has several limitations due to the simplicity of the target CPU architecture.

The target KCPSM3/KCPSM6 may fail to compile due to stack spilling problems.

The original PicoBlaze CPU (**`-mcpu=kcpsm3`**) has very unfortunate stack-spilling related problems that may sometimes result in a compile failure or an incorrect build. This is explained in detail in *Chapter 5, Processor PB3A: Extended PicoBlaze*.

No indirect jumps and pointers to functions.

The PicoBlaze 3 architecture has no way of performing indirect jumps (e.g. jumping to an address specified in a register). Hence it is not possible to use function pointers.

This limitation might be lifted for KCPSM6 in the future versions of the compiler by using the new JUMPAT and CALLAT instructions.

No constant (read-only) data.

PicoBlaze CPU cannot read data constants from the program memory. Hence it is not possible to support constant tables and constant strings.

This limitation might be lifted for KCPSM6 in the future versions of the compiler by using the new CALLAT and LOADRET instructions.

No initialized data.

Contents of the scratch-pad memory cannot be efficiently initialized upon program startup. The backend tools would have to insert a sequence of LOADs and STOREs to setup the data. This is rather inefficient and not implemented. See the example **`extra/examples/02-hello`**.

Warning

Read-only data sections (.rodata) and initialized-data sections (.data) are currently silently ignored by the linker, resulting in incorrect builds!

This limitation will be addressed in a future version.

Interrupt handler in C is not supported yet.

Interrupt handlers are not supported by the C compiler at present. It should be easy, however, to implement them using a helper code in assembler.

2.4. Application Binary Interface (ABI)

2.4.1. Data Types Widths

Table 2.3. Data Types Widths

Type Name	Byte Width (sizeof)	Alternate Names
unsigned char	1	uint8_t
char	1	signed char, int8_t
unsigned int	2	uint16_t
int	2	signed int, int16_t
void *	1	(any pointer type)

2.4.2. Register Usage Convention

Table 2.4. Register Usage Convention

Register	Caller (parent)	Callee (child)	Note
s0	1st func. param./result	1st func. param./result	Trashed in the callee when not used for results.
s1	2nd func. param./result	2nd func. param./result	
s2	3rd func. param./result	3rd func. param./result	
s3	4th func. param./result	4th func. param./result	
s4	save before call	free to use	Freely used in callee for its temporaries. Caller must save if the contents is to be preserved across calls.
s5			
s6			
s7			
s8	no change	save before use	Callee must save the registers to the stack before use. Caller assumes register contents will not change across function calls.
s9			
sA			
sB			
sC			
sD	special, asm. temp.	special, asm. temp.	Reserved for the low-level stack spilling code generated by the compiler.
sE	frame pointer, no change across calls	frame pointer, save before use	Reserved for stack frame pointer. Not used unless the compiler uses stack frames (default: off).
sF	stack pointer, no change across calls	stack pointer, restore to orig. value	Points to the last element pushed on to stack.

Allocation order: S4, S5, S6, S7, S0, S1, S2, S3, S8, S9, SA, SB, SC.

Assembler

The assembler is implemented in the LLVM framework. Syntax and basic features are similar to the GNU assembler from binutils (gas). The assembler outputs ELF object files that can be relocated and linked.

3.1. Assembler Command Line

Assembly source file extension is **`.s`**. It is best to use the compiler driver **`pblaze-cc`** to call the assembler program. For example:

```
pblaze-cc -mcpu=kcpsm3 -c myasm.s -o myasm.o
```

3.2. Assembler Source Code Syntax

Assembly syntax is simplified compared to the original Xilinx specification.

No parenthesis around registers.

Parethesis always indicate an arithmetic expression. For example:

```
    fetch s1, s2        ; ok
    fetch s3, (s4)      ; ! ERROR ! 's4' is parsed as a symbol!
```

Integer values can be given in decimal and hexadecimal bases.

Integer literals are decimal by default, unless the well-known 0x prefix is used to indicate a hexadecimal value.

Arithmetic expressions

Standard arithmetic expressions (+ - * /) are supported and evaluated during assembly phase.

Dots before condition codes: .Z, .NZ, .C, .NC, .DISABLE, .ENABLE

Condition codes in jumps MUST be preceded by a dot: .Z, .NZ, .C, .NC. Similarly for the .ENABLE / .DISABLE keywords. The keywords are implemented as constant-valued symbols that directly set the given field in an instruction opcode. See the example below.

Different `interrupt' opcode

The original `DISABLE INTERRUPT'/'ENABLE INTERRUPT' instructions are now specified differently:

```
    interrupt   .disable        ; used to be 'DISABLE INTERRUPT'
    interrupt   .enable         ; used to be 'ENABLE INTERRUPT'
```

Sections

Each function should be put into its own text section and with proper section attributes. For example, a function `foo' should reside in section `.text.foo':

```
    .section ".text.foo", "ax", @progbits
```

Putting each function in its own section enables linker to omit unused functions from the linked ELF program. See *Chapter 4, Linker and Friends*.

Global variables

Global variables are supported through the **`.bss`** section and the **`.comm`** keyword:

```
    ; Syntax:
    .comm  <variable-name>, <byte-size>

    ; Example:
    .comm x1, 1         ; alloc 1 byte
    .comm x2, 2         ; alloc 2 bytes
    .comm x3, 3         ; alloc 3 bytes
```

See **`extra/examples/04-asmglobals`** for an example program.

3.3. Inline Assembly

Inline assembly can be employed in the C source by using the standard **`asm`** keyword. The syntax of the keyword itself is the same as in GCC; see the *GCC Inline Assembly HOWTO*[1] for more information.

If you only need to access the I/O port it is best to simply use functions **`__input()`** and **`__output()`** that are provided in pbstdlib.h:

```
/** Input value from the port. */
static inline uint8_t __input(uint8_t port)
{
    uint8_t result;
    asm volatile (
        "input %0, %1"          // %0 = result, %1 = port
        : /* outputs */
            /*0*/ "=r" (result)
        : /* inputs */
            /*1*/ "ir" (port)
        : /* clobbers */
    );
    return result;
}

/** Output value to a port. */
static inline void __output(uint8_t port, uint8_t value)
{
    asm volatile (
        "output %0, %1"         // %0 = value, %1 = port
        : /* outputs */
        : /* inputs */
            /*0*/ "r" (value),
            /*1*/ "ir" (port)
        : /* clobbers */
    );
}
```

The functions will be always inlined, hence only the single INPUT or OUTPUT instruction will be generated. Moreover, the "ir" option on the **`port`** parameter allows the compiler to automatically select the right instruction format: either "sx,kk" or "sx,sy", depending on the calling code.

3.4. Listing of All Instructions

```
; Syntax test of all opcodes.
; NOTE: this program is not executable!
```

[1] http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

```
    .section .text._main,"ax",@progbits    ; always put functions into their private
 sections!
    .global _main               ; export the symbol _main

_main:
    add         s1, 0x54
    add         s2, s3
    addn        s5, 0x74        ; PB3A only
    addn        s6, s4          ; PB3A only
    addcy       s4, 74
    addcy       s5, s6
    and         s7, -71
    and         s8, s9
    call        L2
    ; test all variants of condition codes
    call        0, L2
    call        1, L2
    call        2, L3
    call        3, L3
    call        .Z, L3
    call        .NZ, L3
    call        .C, L3
    call        .NC, L3
    call        .z, L3
    call        .nz, L3
    call        .c, L3
    call        .nc, L3
    ;
L1:
    compare     sA, 0xff
    compare     sB, sC
    interrupt   .disable
    interrupt   .enable
    fetch       sD, 12
    fetch       sE, sF
    fetch       s0, s1, 3       ; PB3A only
    input       s2, 5
    input       s3, s4
    input       s5, s6, 9       ; PB3A only
    ;
    jump        L1
    jump        .Z, L1
    jump        .NZ, L1
    jump        .C, L1
    jump        .NC, L1
    ;
L2:
    load        s7, 45
    load        s8, sa
    or          sb, 0
    or          sc, sd
    output      se, 0x33
    output      sf, s0
    output      sf, s1, 0xf     ; PB3A only
    return
    return      .c
    return      .nc
    return      .nz
    return      .z
    returni     0
    returni     1
    returni     .disable
    returni     .enable
L3:
    rl          s2
    rr          s3
```

```
    sl0         s4
    sl1         s5
    sla         s6
    slx         s7
    sr0         s8
    sr1         s9
    sra         sa
    srx         sb
    ;
    store       sD, 12
    store       sE, sF
    store       s0, s1, 3       ; PB3A only
    sub         sc, 5+1
    sub         sc, sd
    subcy       se, 7
    subcy       sf, sd
    test        s0, 12
    test        s1, s0
    xor         s2, 0x55
    xor         s3, s4

    ; KCPSM6 only:
    comparecy   sD, 0xEE
    comparecy   sE, sF
    testcy      s5, 88
    testcy      s6, s7
    hwbuild     s5
    star        s6, s7
    loadret     s8, 0x71
    regbank     0
    regbank     1
    regbank     .a
    regbank     .b
    jumpat      s9, sA
    callat      sB, sC
    outputk     0x12, 0x5

L_tmp1:
    .size       _main, L_tmp1 - _main
```

Linker and Friends

Object files and static archives can be linked using the link editor called **pblaze-ld**. The linker produces static ELF programs as its output. Program code (the section `.text') is directly extracted from the ELF programs using the common **objcopy** tool, and converted to various formats with the help of the **pblaze-bincopy** tool.

Note

It is best to avoid calling the linker program directly. It is simpler to use the compiler-driver program **pblaze-cc** to call the linker as this ensures correct flags are passed on the linker command line. If in doubt, use the **-v** option to **pblaze-cc** to see what parameters are being passed to the linker by default.

4.1. Linker

Linker connects several input object files to form a single output executable ELF file. The smallest unit of work in the linking process is a relocatable section. Sections are never split by the linker. The compiler or assembly-level programmer should put each function into its own private section (usually called `.text.foo' for a function `foo'). In the compiler this behaviour is enforced by the **-ffunction-sections** command line; this options is added automatically by the **pblaze-cc** driver.

By default the linker editor includes in the output file all the sections it has found in each input object file. This is inefficient for object files may contain sections for many different functions, but only a small subset of the functions is usually used at the same time. The **--gc-sections** command-line option directs the linker to include only sections (functions) that are really needed. However, this option is only effective on archives (.a files); it has no impact on linking of the stand-alone input object files (.o files)! All the functions from the stand-alone object files mentioned on the command line will be linked-in. This limitation may be fixed in a future version.

Linking process is controlled by a linker script. Linker script may be specified using the **-c FILE** option in **pblaze-ld**, or **-T FILE** in **pblaze-cc**. The default linker script is **target/elf32-pblaze.script**. The script puts function **__reset** at program address 0x000; more precisely, it puts the sections called **.text.__reset** at the address 0x000. The other text sections are put after the reset function in an undefined order.

The default reset initialization code is implemented in **target/lib-$(MCPU)/startup.o**; the file is linked-in unless the **-nostartfiles** option is given to the driver. This object file contains a **.text.__reset** section (source code given below) that: (1) initializes stack pointer to 0x40 (one byte beyond the end of scratch-pad memory); (2) calls main(); (3) outputs main's exit code to port 0xFF -- this stops execution in the HDL simulator; (4) loops indefinitely.

```
    .section .text.__reset,"ax",@progbits
    .global startup

startup:
        load    sF, 0x40        ; Initialize stack pointer.
        call    _main           ; Execute main()
        output  s0, 0xFF        ; Print main's exit code in s0 to port 0xff.
                                ; It should stop simulation.
L1:     jump    L1              ; Loop indefinitely
```

4.2. Librarian

Static archive files (.a) can be built using the **pblaze-ar** tool. Use the **-r** option to build or update an archive from object files, such as in the example:

```
libpblibc.a : muli8.o muli16.o memset.o
 pblaze-ar -r $@ $^
```

Archives can be used during the linking process. The directory where the archive is located should be added to the linker search path using the **-L dir** option. An archive called **libfoo.a** can be linked in using the option **-lfoo**.

The driver program always adds the directory **target/lib-$(MCPU)** to the linker path. The library **libpblibc.a** is linked in unless the **-nostdlib** option is given to driver.

4.3. Binary Program Conversions

Once a static program ELF file is produced by the linker, the program code (text) usually has to be extracted and converted to a format suitable for further processing.

The code extraction can be performed using the common **objcopy** program that is available in all Linux distributions. The command-line usage is:

```
%.bin : %.elf
 objcopy -I elf32-little -O binary -j .text $^ $@
```

This generates a .bin output file from an .elf input file. In the output file, four bytes (32-bits) are used to encode each PicoBlaze instruction. The format is little-endian.

Binary program image file (.bin) can be further converted to a .mem or .inc file using the **pblaze-bin** utility. MEM files are text based files with an encoded memory contents. They can be loaded by the standard functions in the Std_DevelopersKit simulation library. INC files are C language files containing the memory content in a C array. They can be easily compiled by a C compiler, and this way the firmware code can be included as a payload in a MicroBlaze program.

```
# Convert binary image .bin to a C header for inclusion in a microblaze program
%.inc : %.bin
 pblaze-bincopy --format=c $< $@

# Convert binary image .bin to a MEM file that can be loaded into a vhdl-based simulator
%.mem : %.bin
 pblaze-bincopy --format=mem $< $@
```

Processor PB3A: Extended PicoBlaze

PB3A is an open-source implementation of the original KCPSM3 ISA, with several improvements. It is based on *PacoBlaze~3 v.2.2*[1].

5.1. Motivation

The KCPSM3 architecture has one very unfortunate `feature': *It is not possible to perform an integer addition without modifying the flag register.* This may seem like an unimportant detail, but combined with other factors it poses a very severe problem.

Local variables and function arguments in C are stored on stack. The stack is located at the upper part of the scratch-pad memory; it begins at address 0x3F and it grows down towards the lower addresses. Pointer to the current top of the stack is located by convention in register **sF**. The register points to the last element pushed to the stack.

The C language often addresses the stack in a random-access manner. At function beginning space is allocated on the stack by subtracting a given number of bytes from the stack pointer **sF**. Afterwards, variables stored on the stack are accessed using constant indices added to the stack pointer. For example, a function may have three local variables of type uint8_t: x, y, z. The variables will be allocated on the stack and accessed using indices 0, 1, 2. To fetch the third variable ('z') into a register the processor must execute the following code:

```
    ; == fetch value of 'z' into register s4: ==
    ; The layout of stack is (from higher addresses to lower):
    ;     +2  ->  z
    ;     +1  ->  y
    ;  sF +0  ->  x
    ; The variable is located in scratchpad at address (sF+2).
    load  sD, sF            ; sD := sF ; sD is a dedicated temporary reg.
    add   sD, 2             ; sD := sF + 2 ; Flags destroyed!
    fetch s4, sD            ; s4 := [sF+2]
```

The problem with the code above is that it is not inert as it modifies a *gloablly visible state*, i.e. CPU flags (carry and zero). This is very unfortunate because the LLVM compiler framework needs to insert the stack spilling code (such as the one above) at any place in the program without affecting any unrelated state (flags).

The current implementation in the **pblaze-cc** compiler tries to detect the critical situation when stack spilling is being inserted in the middle of flags live range (spilling when flags content is unimportant is not a problem). It tries to move the code around a little bit to see if the spilling could be moved out of the flags live range (then all is well), but this is not always possible. In the second case a warning is printed and a so-called `careful spilling' is inserted. The warning looks like this:

```
WARNING: Slow and dangerous stack-spilling code in KCPSM3 had to be inserted one or several
 times!
   Please see the documentation what you can do about it.
```

However, even the careful spilling is not entirely correct in all situations as it only saves the carry flag, not the zero flag. It looks like this:

```
    load    sD, sF
    sla     sE          ; save Carry to sE.bit0
```

[1] http://bleyer.org/pacoblaze/

```
    add     sD, 2
    sr0     sE          ; restore Carry from sE.bit0
    fetch   s4, sD
```

The `careful spilling' code uses register **sE** (frame pointer if used) to temporarily save the carry flag around the ADD instruction. As a side effect, bit #7 of **sE** is lost but this is not a problem as the **sE** register is (sometimes) used as a stack frame pointer, thus its two highest bits are always zero. Still, zero flag is not saved by the `careful spill' code, hence the generated code is not guaranteed correct when the warning is issued by the compiler.

5.2. ISA: New Instruction: ADDN

A new instruction ADDN is provided using a previously reserved 5-bit op-code prefix 0x08. The instruction behaves like the existing ADD instruction, except that flag registers are *not* modified.

Table 5.1. Operation

Instruction	Function
ADDN sX, sY	sX := (sX + sY) mod 256
ADDN sX, kk	sX := (sX + kk) mod 256

Note

The FLAG registers are not read nor modified by the instructions.

Table 5.2. Encoding

Instruction	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDN sX,kk	0	1	0	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k
ADDN sX,sY	0	1	0	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0

5.2.1. Rationale

The ADDN instruction is used in the stack spilling code in the C compiler. The compiler needs to insert stack spilling code at any point of a basic block, hence spilling code must not modify any globally visible state such as machine flags.

A typical stack spilling code looks like this:

```
LOAD  sD, sF              ; sF is stack pointer, sD is aux. reg.
ADDN  sD, 20
FETCH s1, sD              ; s1 := [sF+20]
```

5.3. ISA: New Addressing Mode: Register+Offset

Specification of instructions that access scratch-pad memory and I/O resources was slightly modified. The extension is backward compatible, i.e. programs assembled for older ISA will work without a modification.

Previously, the register-indirect versions (*sX,sY*) of the FETCH/STORE/INPUT/OUTPUT instructions did not use the lower-order 4 bits in the encoding; these bits were always zero. In the new

specification these bits hold a 4-bit unsigned *offset* that is added to the value of the *sY* register and then used as an 8-bit address for the memory or I/O operation.

Table 5.3. Operation

Instruction	Function
FETCH sX, sY, ff	sX := SCRATCHPAD[sY + ff]
STORE sX, sY, ff	SCRATCHPAD[sY + ff] := sX
INPUT sX, sY, ff	sX := IO[sY + ff]
OUTPUT sX, sY, ff	IO[sY + ff] := sX

Table 5.4. Encoding

Instruction	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FETCH sX, sY, ff	0	0	0	1	1	1	x	x	x	x	y	y	y	y	f	f	f	f
STORE sX, sY, ff	1	0	1	1	1	1	x	x	x	x	y	y	y	y	f	f	f	f
INPUT sX, sY, ff	0	0	0	1	0	1	x	x	x	x	y	y	y	y	f	f	f	f
OUTPUT sX, sY, ff	1	0	1	1	0	1	x	x	x	x	y	y	y	y	f	f	f	f

Note

- The constant offset *ff* is in unsigned range 0 to 15, the addition is 8-bit unsigned.
- When the offset *ff* is not specified in assembler source, it is assumed zero by the assembler.
- The register-constant variant (*sX,kk*) of the instructions is not affected by this ISA modification.

5.3.1. Rationale

The register+offset addressing mode optimizes common stack-spilling code. When frame pointer (sE) is not used in a function and the stack grows down to lower addresses, the stack pointer (sF) always points to the last element pushed onto the stack. To access local variables and function arguments a positive offset has to be added to the stack pointer, as illustrated in *Figure 5.1, "Program Stack in PB3A"*. (Note that an optimizing compiler will pass several function arguments in registers, and not all local variables will be allocated on stack.) In this case the *FETCH/STORE sX, sF, ofs* instructions can be used to access function arguments and local variables just in one instruction. In the original ISA at least 3 instructions were needed. When the offset is outside the 0-15 range the standard three-instruction spilling code with the ADDN instruction must be used.

Besides stack spilling code the register+offset addressing mode can be also used for accessing data fields in a record structure. In this case a register contains the record base address and a constant offset is used to access any element (field) in the structure.

The register+offset addressing mode is also useful in the context of INPUT/OUTPUT instructions. When several instances of functionally equivalent complex hardware units (each having a couple of registers) are mapped to different ranges of the I/O address space, the individual I/O ports can be efficiently accessed by loading the base I/O port address of the particular hardware unit instance into a register, then using the *offset* field to access the given port register of the instance.

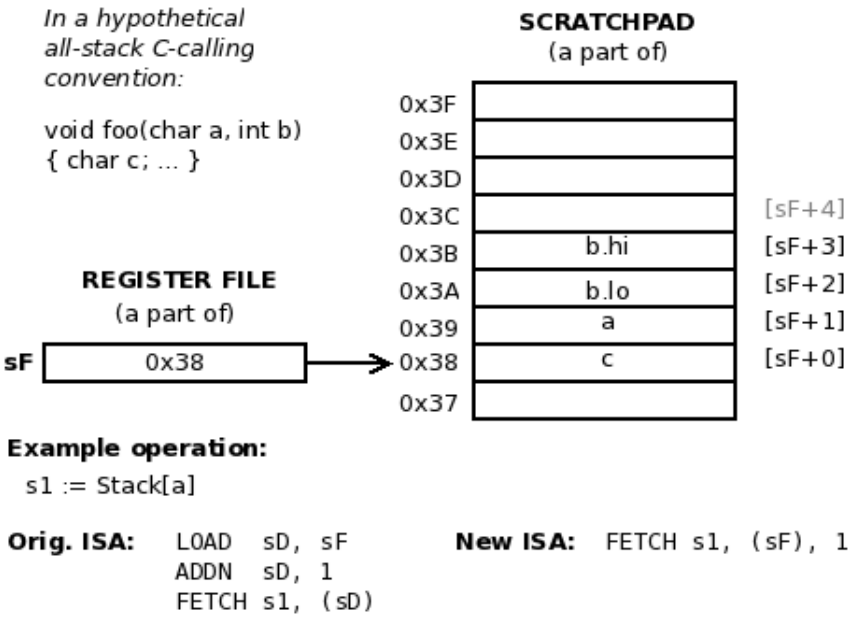

Figure 5.1. Program Stack in PB3A

5.4. The port_busy Signal

The processor has a new input signal called **`port_busy: std_logic`**. When an I/O transaction is in progress (**`read_strobe`** or **`write_strobe`** are active) the **`port_busy`** signal may be asserted by an external logic to prolong the I/O operation, as shown in *Figure 5.2, "The new port_busy signal in PB3A"*. The processor is stalled and waiting cycles are inserted. This is especially useful in input transactions when data is not immediately available but several cycles is required to access it. When not required the **`port_busy`** signal can be simply tied to zero.

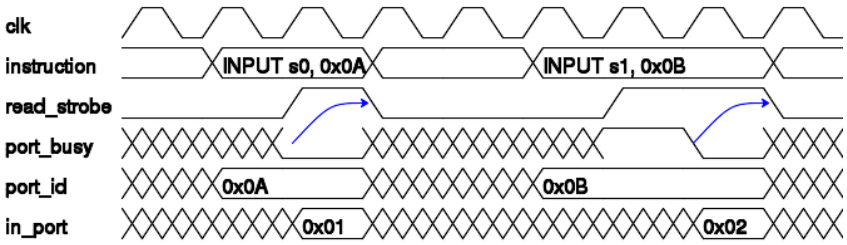

Figure 5.2. The new port_busy signal in PB3A

5.5. Debug Ports

Two output debug ports are provided: **`dbg_rslt_dt`** and **`dbg_rslt_we`**. They contain the data written into the register file as the instruction result.

5.6. HDL Instantiation

```
--! Width of pblaze instr address bus
constant PB3_CODE_DEPTH : integer := 10;

--! Width of pblaze instruction
constant PB3_CODE_WIDTH : integer := 18;

--! Width of port_id address
constant PB3_PORT_DEPTH : integer := 8;
```

```
--! Width of inp/outp port data
constant PB3_PORT_WIDTH : integer := 8;

--! register width
constant PB3_REGISTER_WIDTH : integer := PB3_PORT_WIDTH;

component pacoblaze3a is
port (
    --! instruction address output
    address     : out std_logic_vector(PB3_CODE_DEPTH-1 downto 0);
    --! instruction input
    instruction : in  std_logic_vector(PB3_CODE_WIDTH-1 downto 0);
    --! port address output
    port_id     : out std_logic_vector(PB3_PORT_DEPTH-1 downto 0);
    --! write on a port
    write_strobe : out std_logic;
    --! output data to write on port
    out_port    : out std_logic_vector(PB3_PORT_WIDTH-1 downto 0);
    --! read from a port
    read_strobe : out std_logic;
    --! input data from a port
    in_port     : in std_logic_vector(PB3_PORT_WIDTH-1 downto 0);
    --! port is busy - PB3A specific
    port_busy   : in std_logic;
    --! interrupt request
    interrupt   : in std_logic;
    --! interrupt ack
    interrupt_ack : out std_logic;
    --! reset signal
    reset       : in std_logic;
    --! clock signal
    clk         : in std_logic;
    --! Debug ports - PB3A specific:
    --! writing result into rX
    dbg_rslt_we : out std_logic;
    --! the result value
    dbg_rslt_dt : out std_logic_vector(PB3_REGISTER_WIDTH-1 downto 0)
);
end component pacoblaze3a;
```

Hardware Source and the Simulator

The **`extra/hw/`** directory contains hardware models of the processors (KCPSM3, KCPSM6, PB3A) that can be used for simulation, compiler verification, and as hardware design examples. The directory is not required for the compiler itself.

6.1. Preparing the Hardware Library (extra/hw/pblib)

Important

PLEASE, CAREFULLY OBSERVE THE INSTRUCTIONS BELOW ON HOW TO COMPLETE THE SOURCE CODE INSTALLATION!

Several source code files are not distributed in the package due to the licensing reasons and must be obtained from third parties.

6.1.1. KCPSM3/KCPSM6 Installation

Download the files **`kcpsm3.vhd`** and **`kcpsm6.vhd`** from *Xilinx*[1] and put them into the **`extra/hw/pblib`** directory.

6.1.2. PB3A: PacoBlaze Installation

1. Download **`pacoblaze-2.2.zip`** from *http://bleyer.org/pacoblaze/*.

2. Extract the ZIP file somewhere and copy the following files from **`pacoblaze-2.2.zip/pacoblaze`** to **`extra/hw/pblib/pacoblaze`**:

pacoblaze_alu.v	pacoblaze_dregister.v
pacoblaze_idu.v	pacoblaze_inc.v
pacoblaze_register.v	pacoblaze_scratch.v
pacoblaze_stack.v	pacoblaze_util.v
pacoblaze.v	pacoblaze3.v

3. Patch the verilog files using the following command:

```
cd extra/hw/pblib/pacoblaze
patch -p1 < ../pacoblaze.diff
```

6.2. ModelSim-Based PicoBlaze Simulator

A simple PicoBlaze simulator is implemented in the directory **`extra/hw/simulator`**. It is implemented in the *ModelSim HDL simulator*[2]. It simulates the actual hardware-synthesisable model of the processor.

Before you begin, make sure you have completed the source code installation as described in the previous section.

[1] http://www.xilinx.com/products/intellectual-property/picoblaze.htm
[2] http://model.com/

First, build the simulator testbench using ModelSim by executing **make** in the **extra/hw/simulator** directory. Depending on your ModelSim installation you probably will need to update the **modelsim.ini** file so that it points to the correct HDL library files.

Should you run into difficulties compiling the HDL source code, here's a small hint that may help you:

1. The **$XILINX** environment variable has to be set. This is usually done by sourcing the appropriate **settings*.sh** file in a shell.

2. In addition the Xilinx simulation libraries have to be compiled beforehand and the resulting **modelsim.ini** file has to be placed into the **$XILINX** directory. This is achieved by running the following program (e.g. in ISE 12.2): **$XILINX/bin/<platform>/compxlib** The program will generate 'modelsim.ini' file, probably in its working directory. You have to manually move the file into the top-level **$XILINX** directory so that the makefile can find it.

Once the HDL compiles successfuly, you may run a simulation using the **run-sim.sh** script, e.g.

```
./run-sim.sh kcpsm3 ../../examples/02-hello/kcpsm3/prog.kcpsm3.mem
```

The first parameter is the simulated processor: kcpsm3 or pb3a. The second parameter is the program memory in the MEM file. During simulation the Picoblaze instructions are printed as they are executed and the file portmon.txt is filled with data that were OUTPUT on ports.

6.2.1. Example

The listing below shows an output of the simulator as an example. In PB3A mode the data being written into the register file are shown at the right column; this greatly simplifies debugging of a program.

```
MCPU = pb3a
Program = ../../examples/02-hello/kcpsm3/prog.kcpsm3.mem
Timeout = 1000 us

Reading /home/jara/bin/Modelsim66e/modeltech/tcl/vsim/pref.tcl

# 6.6e

# vsim -Gimem_fname=../../examples/02-hello/kcpsm3/prog.kcpsm3.mem -Gpbver=pb3a pbsim
# //  ModelSim SE-64 6.6e Mar 30 2011 Linux 3.2.2-1.fc16.x86_64
# //
# //  Copyright 1991-2011 Mentor Graphics Corporation
# //             All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading std.textio(body)
# Loading std_developerskit.std_mempak(body)
# Loading std_developerskit.std_iopak(body)
# Loading pblib.comp_kcpsm3(body)
# Loading pblib.comp_pblaze(body)
# Loading ieee.std_logic_arith(body)
# Loading ieee.std_logic_unsigned(body)
# Loading pblib.text_utils(body)
# Loading pblib.pblaze_dbg(body)
# Loading work.pbsim(behav)#1
# Loading pblib.pacoblaze3a(fast)
```

```
set NumericStdNoWarnings 1
# 1
set StdArithNoWarnings 1
# 1
run 1000 us
# ** Note: Mem_Load:  Loading file:  ../../examples/02-hello/kcpsm3/prog.kcpsm3.mem
#    Time: 0 ns  Iteration: 0  Instance: /pbsim
# ** Note: Mem_Load:  Loading address:  00000 from file: ../../examples/02-hello/kcpsm3/
prog.kcpsm3.mem
#    Time: 0 ns  Iteration: 0  Instance: /pbsim
# ** Note: Mem_Load:  finished loading file:  ../../examples/02-hello/kcpsm3/prog.kcpsm3.mem
#    Time: 0 ns  Iteration: 0  Instance: /pbsim
# cyc: 0, PC 0x000, op: 0x00F40  LOAD   sF, 0x40
# cyc: 0, PC 0x000, op: 0x00F40  LOAD   sF, 0x40
# cyc: 0, PC 0x000, op: 0x00F40  LOAD   sF, 0x40              ; sF <- 0x40
# cyc: 4, PC 0x001, op: 0x30004  CALL   0x004
# cyc: 6, PC 0x004, op: 0x1CF0A  SUB    sF, 0x0A             ; sF <- 0x36
# cyc: 8, PC 0x005, op: 0x00448  LOAD   s4, 0x48             ; s4 <- 0x48
# cyc: 10, PC 0x006, op: 0x00565  LOAD   s5, 0x65            ; s5 <- 0x65
# cyc: 12, PC 0x007, op: 0x2F4F0  STORE  s4, (sF), 0x0
# cyc: 14, PC 0x008, op: 0x0066C  LOAD   s6, 0x6C            ; s6 <- 0x6C
# cyc: 16, PC 0x009, op: 0x01DF0  LOAD   sD, sF              ; sD <- 0x36
# cyc: 18, PC 0x00A, op: 0x18D01  ADD    sD, 0x01            ; sD <- 0x37
# cyc: 20, PC 0x00B, op: 0x2F5D0  STORE  s5, (sD), 0x0
# cyc: 22, PC 0x00C, op: 0x18D01  ADD    sD, 0x01            ; sD <- 0x38
# cyc: 24, PC 0x00D, op: 0x2F6D0  STORE  s6, (sD), 0x0
# cyc: 26, PC 0x00E, op: 0x0056F  LOAD   s5, 0x6F            ; s5 <- 0x6F
# cyc: 28, PC 0x00F, op: 0x18D01  ADD    sD, 0x01            ; sD <- 0x39
# cyc: 30, PC 0x010, op: 0x2F6D0  STORE  s6, (sD), 0x0
# cyc: 32, PC 0x011, op: 0x016F0  LOAD   s6, sF              ; s6 <- 0x36
# cyc: 34, PC 0x012, op: 0x18601  ADD    s6, 0x01            ; s6 <- 0x37
# cyc: 36, PC 0x013, op: 0x00700  LOAD   s7, 0x00            ; s7 <- 0x00
# cyc: 38, PC 0x014, op: 0x18D01  ADD    sD, 0x01            ; sD <- 0x3A
# cyc: 40, PC 0x015, op: 0x2F5D0  STORE  s5, (sD), 0x0
# cyc: 42, PC 0x016, op: 0x18D01  ADD    sD, 0x01            ; sD <- 0x3B
# cyc: 44, PC 0x017, op: 0x2F7D0  STORE  s7, (sD), 0x0
# cyc: 46, PC 0x018, op: 0x2C400  OUTPUT s4, 0x00
# cyc: 47, port-write, port_id: 0x00, data: 0x48
# cyc: 48, PC 0x019, op: 0x07460  FETCH  s4, (s6), 0x0       ; s4 <- 0x65
# cyc: 50, PC 0x01A, op: 0x14400  CMP    s4, 0x00
# cyc: 52, PC 0x01B, op: 0x3501E  JUMP    Z, 0x01E
# cyc: 54, PC 0x01C, op: 0x18601  ADD    s6, 0x01            ; s6 <- 0x38
# cyc: 56, PC 0x01D, op: 0x34018  JUMP   0x018
# cyc: 58, PC 0x018, op: 0x2C400  OUTPUT s4, 0x00
# cyc: 59, port-write, port_id: 0x00, data: 0x65
# cyc: 60, PC 0x019, op: 0x07460  FETCH  s4, (s6), 0x0       ; s4 <- 0x6C
# cyc: 62, PC 0x01A, op: 0x14400  CMP    s4, 0x00
# cyc: 64, PC 0x01B, op: 0x3501E  JUMP    Z, 0x01E
# cyc: 66, PC 0x01C, op: 0x18601  ADD    s6, 0x01            ; s6 <- 0x39
# cyc: 68, PC 0x01D, op: 0x34018  JUMP   0x018
# cyc: 70, PC 0x018, op: 0x2C400  OUTPUT s4, 0x00
# cyc: 71, port-write, port_id: 0x00, data: 0x6C
# cyc: 72, PC 0x019, op: 0x07460  FETCH  s4, (s6), 0x0       ; s4 <- 0x6C
# cyc: 74, PC 0x01A, op: 0x14400  CMP    s4, 0x00
# cyc: 76, PC 0x01B, op: 0x3501E  JUMP    Z, 0x01E
# cyc: 78, PC 0x01C, op: 0x18601  ADD    s6, 0x01            ; s6 <- 0x3A
# cyc: 80, PC 0x01D, op: 0x34018  JUMP   0x018
# cyc: 82, PC 0x018, op: 0x2C400  OUTPUT s4, 0x00
# cyc: 83, port-write, port_id: 0x00, data: 0x6C
# cyc: 84, PC 0x019, op: 0x07460  FETCH  s4, (s6), 0x0       ; s4 <- 0x6F
# cyc: 86, PC 0x01A, op: 0x14400  CMP    s4, 0x00
# cyc: 88, PC 0x01B, op: 0x3501E  JUMP    Z, 0x01E
# cyc: 90, PC 0x01C, op: 0x18601  ADD    s6, 0x01            ; s6 <- 0x3B
# cyc: 92, PC 0x01D, op: 0x34018  JUMP   0x018
# cyc: 94, PC 0x018, op: 0x2C400  OUTPUT s4, 0x00
# cyc: 95, port-write, port_id: 0x00, data: 0x6F
# cyc: 96, PC 0x019, op: 0x07460  FETCH  s4, (s6), 0x0       ; s4 <- 0x00
```

```
# cyc: 98, PC 0x01A, op: 0x14400  CMP    s4, 0x00
# cyc: 100, PC 0x01B, op: 0x3501E  JUMP    Z, 0x01E
# cyc: 102, PC 0x01E, op: 0x00000  LOAD   s0, 0x00            ; s0 <- 0x00
# cyc: 104, PC 0x01F, op: 0x01100  LOAD   s1, s0              ; s1 <- 0x00
# cyc: 106, PC 0x020, op: 0x18F0A  ADD    sF, 0x0A            ; sF <- 0x40
# cyc: 108, PC 0x021, op: 0x2A000  RETURN
# cyc: 110, PC 0x002, op: 0x2C0FF  OUTPUT s0, 0xFF
# cyc: 111, port-write, port_id: 0xFF, data: 0x00
# cyc: 112, monitor-process, request-to-stop
quit -f
```

All data that were output to I/O ports are captured in a file called **`portmon.txt`**. Below is the file content as generated from the **`02-hello`** program. The PicoBlaze program has written "Hello" to the port 0x00, and 0x00 to the port 0xFF to stop the simulator.

```
47  W  0x00  0x48  'H'
59  W  0x00  0x65  'e'
71  W  0x00  0x6C  'l'
83  W  0x00  0x6C  'l'
95  W  0x00  0x6F  'o'
111  W  0xFF  0x00  nul
```

Appendix A. Examples

Several program examples are provided in the **`extra/examples`** directory. To compile the examples after the toolchain has been installed as described in *Section 1.2, "Installation"*, simply change into the given directory (e.g. **`extra/examples/01-simple`**) and type `**`make`**' to compile the source codes.

A.1. The Most Simple Program

C Source code:

```
#include <pbstdlib.h>

int main()
{
    // The correct answer is constant
    return 42;
}
```

Compiled KCPSM3 Assembly:

```
 .file "llvm_2AlLBN/main.obc"
 .text
 .globl _main
 .type _main,@function
_main:                                  ; @main
; BB#0:                                 ; %entry
 load  s0, 42
 load  s1, 0
 return
L_tmp0:
 .size _main, L_tmp0-_main
```

A.2. Hello World

C Source code:

```
#include <pbstdlib.h>

static void fill(char *s)
{
    s[0] = 'H';
    s[1] = 'e';
    s[2] = 'l';
    s[3] = 'l';
    s[4] = 'o';
    s[5] = '\0';
}

int main()
{
    char str[10];
    /* We cannot simply use:
     *
     *    const char *str = "Hello";
     *
     * because PBlaze cannot read constants from the program memory.
```

```
     * Thus the only thing to do is to load the array one by one using the
     * LOAD and STORE instructions.
     * The automatic transformation is NOT IMPLEMENTED in the compiler
     * as of now, so we have to do it by hand.
     */
    fill(str);

    char *p = str;
    while (*p)
        /* write char to port 0x00 */
        __output(0x00, *p++);

    return 0;
}
```

Compiled KCPSM3 Assembly:

```
 .file "llvm_E2X1bI/main.obc"
 .text
 .globl _main
 .type _main,@function
_main:                                  ; @main
; BB#0:                                 ; %entry
 sub  sF, 10
 load  s4, 72
 load  s5, 101
 store  s4, sF
 load  s6, 108
 load  sD, sF
 add  sD, 1
 store  s5, sD
 add  sD, 1
 store  s6, sD
 load  s5, 111
 add  sD, 1
 store  s6, sD
 load  s6, sF
 add  s6, 1
 load  s7, 0
 add  sD, 1
 store  s5, sD
 add  sD, 1
 store  s7, sD
L_BB0_1:                                ; %while.body
                                        ; =>This Inner Loop Header: Depth=1

 ;APP
 output s4, 0
 ;NO_APP
 fetch  s4, s6
 compare s4, 0
 jump  .Z, L_BB0_3
; BB#2:                                 ; %while.body.while.body_crit_edge
                                        ;   in Loop: Header=BB0_1 Depth=1

 add  s6, 1
 jump  L_BB0_1
L_BB0_3:                                ; %while.end
 load  s0, 0
 load  s1, s0
 add  sF, 10
 return
L_tmp0:
 .size _main, L_tmp0-_main
```

Compiled PB3A Assembly:

```
 .file "llvm_kBe0SS/main.obc"
 .text
 .globl _main
 .type _main,@function
_main:                                  ; @main
; BB#0:                                 ; %entry
 sub  sF, 10
 load  s4, 72
 load  s5, 101
 store  s4, sF, 0
 load  s6, 108
 store  s5, sF, 1
 store  s6, sF, 2
 load  s5, 111
 store  s6, sF, 3
 load  s6, sF
 addn  s6, 1
 load  s7, 0
 store  s5, sF, 4
 store  s7, sF, 5
L_BB0_1:                                ; %while.body
                                        ; =>This Inner Loop Header: Depth=1
 ;APP
 output s4, 0
 ;NO_APP
 fetch  s4, s6
 compare s4, 0
 jump  .Z, L_BB0_3
; BB#2:                                 ; %while.body.while.body_crit_edge
                                        ;   in Loop: Header=BB0_1 Depth=1
 addn  s6, 1
 jump  L_BB0_1
L_BB0_3:                                ; %while.end
 load  s0, 0
 load  s1, s0
 addn  sF, 10
 return
L_tmp0:
 .size _main, L_tmp0-_main
```

A.3. Sum of the first 100 numbers (0..99)

C Source code:

```
#include <pbstdlib.h>

/** Sum of the first 100 numbers (0..99).
 * The loop will be optimized into a constant expression
 * because there are no side-effects in it and the iteration
 * range is known in compile-time.
 */

int main()
{
    int16_t sum = 0;
    for (int16_t i = 0; i < 100; ++i) {
        /* in -O3 this loop is computed in compile-time! */
        sum += i;
    }

    __output(0, sum);
    __output(1, sum >> 8);
```

```
    return sum;
}
```

Compiled KCPSM3 Assembly:

```
 .file "llvm_UCsZvg/main.obc"
 .text
 .globl _main
 .type _main,@function
_main:                                  ; @main
; BB#0:                                 ; %entry
 load  s0, 86
 ;APP
 output s0, 0
 ;NO_APP
 load  s1, 19
 ;APP
 output s1, 1
 ;NO_APP
 return
L_tmp0:
 .size _main, L_tmp0-_main
```

A.4. Cumulative sum of the first 100 numbers (0..99)

C Source code:

```
#include <pbstdlib.h>

/** Cumulative sum of the first 100 numbers (0..99),
 * with incremental output to an I/O port.
 * Compared to the example 07-sum the loop is not optimized away
 * because of the __output() function.
 * Note however that the value of 'sum' after the loop has finished
 * IS computed in compile-time, i.e. the return statement is constant
 * in assembly.
 */

int main()
{
    int16_t sum = 0;
    for (int16_t i = 0; i < 100; ++i) {
        sum += i;
        __output(0, sum);
        __output(1, sum >> 8);
    }

    return sum;
}
```

Compiled KCPSM3 Assembly:

```
        .file   "llvm_LGbJCX/main.obc"
        .text
        .globl  _main
        .type   _main,@function
_main:                                  ; @main
; BB#0:                                 ; %entry
        load    s4, 0
        load    s5, s4
        load    s6, s4
```

```
        load    s7, s4
L_BB0_1:                                ; %for.inc
                                        ; =>This Inner Loop Header: Depth=1
        load    s0, s6
        add     s0, s7
        load    s7, s4
        addcy   s7, s5
        ;APP
        output s0, 0
        ;NO_APP
        ;APP
        output s7, 1
        ;NO_APP
        add     s6, 1
        addcy   s4, 0
        load    s1, s6
        xor     s1, 100
        or      s1, s4
        load    s5, s7
        load    s7, s0
        compare s1, 0
        jump    .NZ, L_BB0_1
; BB#2:                                 ; %for.end
        load    s0, 86
        load    s1, 19
        return
L_tmp0:
        .size   _main, L_tmp0-_main
```

A.5. Sieve of Eratosthenes (simple version)

C Source code:

```c
/**
 * Sieve of Eratosthenes
 * Computes and prints the prime numbers up to MAXNUM.
 * This version uses an inefficient byte array to represent a set.
 */

#include <pbstdlib.h>

/* Maximal number. This number of bytes will be allocated
 * on stack, so the maximum cannot be much high. */
#define MAXNUM          32

int main()
{
    /* Sieve byte array. Each byte (element) represents
     * one candidate number from 0 to MAXNUM.
     * Zero means candidate prime, one means we know it is NOT a prime.
     */
    uint8_t sieve[MAXNUM];

    /* clear the array */
    for (uint8_t i = 0; i < MAXNUM; ++i)
        sieve[i] = 0;

    /* start from 2, the first prime */
    for (uint8_t i = 2; ; ++i) {
        /* find the first prime in the sieve, i.e. the first
         * number without a flag set */
        for (; i < MAXNUM; ++i) {
            if (sieve[i] == 0) {
```

```
                /* it's prime;
                 * Print it to port 0 */
                __output(0, i);
                break;
            }
        }
        /* did we find any? */
        if (i == MAXNUM)
            /* no, stop */
            return 0;

        /* Mark all its multiplies up to MAXNUM not be a prime. */
        for (uint8_t k = 2*i; k < MAXNUM; k += i) {
            sieve[k] = 1;
        }
    }

    return 0;
}
```

Compiled KCPSM3 Assembly:

```
 .file "llvm_TSYzZ9/main.obc"
 .text
 .globl _main
 .type _main,@function
_main:                                  ; @main
; BB#0:                                 ; %entry
 sub  sF, 32
 load  s4, 32
 load  s7, 2
 load  s0, 4
 load  s1, 0
L_BB0_1:                                ; %for.inc
                                        ; =>This Inner Loop Header: Depth=1
 load  s5, sF
 load  s6, s5
 sub  s6, s4
 add  s4, -1
 add  s6, 32
 store  s1, s6
 compare s4, 0
 jump  .NZ, L_BB0_1
; BB#2:
 load  s1, 1
L_BB0_3:                                ; %for.cond7
                                        ; =>This Loop Header: Depth=1
                                        ;     Child Loop BB0_9 Depth 2
 load  s4, 31
 compare s4, s7
 jump  .C, L_BB0_6
; BB#4:                                 ; %for.body12
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s4, s5
 add  s4, s7
 fetch  s4, s4
 compare s4, 0
 jump  .NZ, L_BB0_10
; BB#5:                                 ; %if.then
                                        ;   in Loop: Header=BB0_3 Depth=1
 ;APP
 output s7, 0
 ;NO_APP
L_BB0_6:                                ; %for.end23
                                        ;   in Loop: Header=BB0_3 Depth=1
 compare s7, 32
```

```
 jump  .NZ, L_BB0_8
; BB#7:                                 ; %if.then28
 load  s0, 0
 load  s1, s0
 add  sF, 32
 return
L_BB0_8:                                ; %if.end29
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s4, 31
 compare s4, s0
 load  s4, s0
 jump  .C, L_BB0_10
L_BB0_9:                                ; %for.inc42
                                        ;   Parent Loop BB0_3 Depth=1
                                        ; =>  This Inner Loop Header: Depth=2
 load  s6, s4
 add  s6, s7
 load  s2, s5
 add  s2, s4
 store  s1, s2
 compare s6, 32
 load  s4, s6
 jump  .C, L_BB0_9
L_BB0_10:                               ; %for.cond7.backedge
                                        ;   in Loop: Header=BB0_3 Depth=1
 add  s0, 2
 add  s7, 1
 jump  L_BB0_3
L_tmp0:
 .size _main, L_tmp0-_main
```

A.6. Sieve of Eratosthenes (with bit array)

C Source code:

```
/**
 * Sieve of Eratosthenes
 * Computes and prints the prime numbers up to MAXNUM.
 * This version uses bit-array for maximal space efficiency.
 */

#include <pbstdlib.h>

/* Define the maximal number, if not done so on the command line.
 * MAXNUM/8 bytes will be consumed, so be careful not to overflow the stack.
 */
#ifndef MAXNUM
# define MAXNUM          320
#endif

/* Choose appropriate data type to represent a value */
#if MAXNUM < 256
typedef uint8_t  index_t;
#else
typedef uint16_t index_t;
#endif

/* returns 1 if the given value 'v' is known NOT be to a prime */
static inline uint8_t getv(uint8_t *sieve, index_t v)
{
    return (sieve[v >> 3] >> (v & 0x07)) & 1;
}
```

```
/* set a flag for the given value 'v' indicating it is NOT a prime */
static inline void setv(uint8_t *sieve, index_t v)
{
    sieve[v >> 3] |= 1 << (v & 0x07);
}

int main()
{
    /* Sieve bit array. Each BIT represents one value
     * from 0 to MAXNUM. */
    uint8_t sieve[MAXNUM/8];

    /* clear the array to zero */
    for (uint8_t i = 0; i < MAXNUM/8; ++i)
        sieve[i] = 0;

    /* start from 2, the first prime */
    for (index_t i = 2; ; ++i) {
        /* find the first prime in the sieve, i.e. the first
         * number without a flag set */
        for (; i < MAXNUM; ++i) {
            if (getv(sieve, i) == 0) {
                /* ok, it's prime;
                 * Print LO byte to port 0, and HI byte to port 1. */
                __output(0, i);
                __output(1, i >> 8);
                break;
            }
        }
        /* did we find any? */
        if (i == MAXNUM)
            /* no, stop */
            return 0;

        /* Mark all its multiplies up to MAXNUM not be a prime.
         * Here we have to use 16 bit integers pretty much
         * always because of the condition test.
         */
        for (uint16_t k = 2*i; k < MAXNUM; k += i) {
            setv(sieve, k);
        }
    }

    return 0;
}
```

Compiled PB3A Assembly:

```
 .file "llvm_lwweha/main.obc"
 .text
 .globl _main
 .type _main,@function
_main:                                  ; @main
; BB#0:                                 ; %entry
 sub  sF, 48
 load  sD, sF
 addn  sD, 47
 store  s8, sD
 addn  sD, -1
 store  s9, sD
 addn  sD, -1
 store  sA, sD
 addn  sD, -1
 store  sB, sD
 addn  sD, -1
 store  sC, sD
```

```
 load  s4, 40
 load  s7, 2
 load  s0, 0
 load  s1, 4
L_BB0_1:                                ; %for.inc
                                        ; =>This Inner Loop Header: Depth=1
 load  s5, sF
 addn  s5, 3
 load  s6, s5
 sub  s6, s4
 addn  s6, 40
 store  s1, sF, 2
 addn  s4, -1
 store  s0, s6
 compare s4, 0
 jump  .NZ, L_BB0_1
; BB#2:
 load  s4, s0
 store  s7, sF, 0
L_BB0_3:                                ; %for.cond7
                                        ; =>This Loop Header: Depth=1
                                        ;     Child Loop BB0_23 Depth 2
                                        ;       Child Loop BB0_24 Depth 3
                                        ;     Child Loop BB0_11 Depth 2
 load  s6, 63
 load  s1, 1
 load  s2, s1
 load  s3, 0
 compare s6, s7
 jump  .C, L_BB0_5
; BB#4:                                 ; %for.cond7
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s2, s3
L_BB0_5:                                ; %for.cond7
                                        ;   in Loop: Header=BB0_3 Depth=1
 compare s1, s4
 jump  .C, L_BB0_7
; BB#6:                                 ; %for.cond7
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s1, s3
L_BB0_7:                                ; %for.cond7
                                        ;   in Loop: Header=BB0_3 Depth=1
 compare s4, 1
 jump  .Z, L_BB0_9
; BB#8:                                 ; %for.cond7
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s2, s1
L_BB0_9:                                ; %for.cond7
                                        ;   in Loop: Header=BB0_3 Depth=1
 compare s2, 0
 jump  .NZ, L_BB0_14
; BB#10:                                ; %for.body11
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s6, s7
 sr0  s6
 sr0  s6
 sr0  s6
 load  s1, s4
 rr  s1
 rr  s1
 rr  s1
 and  s1, 224
 or  s6, s1
 load  s1, s5
 addn  s1, s6
 load  s6, 1
 load  s2, 0
```

```
 load  s3, s7
 and  s3, 7
 fetch  s1, s1
 compare s3, 0
 jump  .Z, L_BB0_12
L_BB0_11:                               ; %for.body11
                                        ;   Parent Loop BB0_3 Depth=1
                                        ; =>  This Inner Loop Header: Depth=2
 sl0  s6
 sla  s2
 sub  s3, 1
 jump  .NZ, L_BB0_11
L_BB0_12:                               ; %for.body11
                                        ;   in Loop: Header=BB0_3 Depth=1
 and  s1, s6
 compare s1, 0
 jump  .NZ, L_BB0_32
; BB#13:                                ; %if.then
                                        ;   in Loop: Header=BB0_3 Depth=1
 fetch  s6, sF, 0
 ;APP
 output s6, 0
 ;NO_APP
 ;APP
 output s4, 1
 ;NO_APP
L_BB0_14:                               ; %for.end23
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s6, s7
 xor  s6, 64
 load  s1, s4
 xor  s1, 1
 or  s6, s1
 compare s6, 0
 jump  .NZ, L_BB0_16
; BB#15:                                ; %if.then27
 load  s0, 0
 load  s1, s0
 load  sD, sF
 addn  sD, 43
 fetch  sC, sD
 addn  sD, 1
 fetch  sB, sD
 addn  sD, 1
 fetch  sA, sD
 addn  sD, 1
 fetch  s9, sD
 addn  sD, 1
 fetch  s8, sD
 addn  sF, 48
 return
L_BB0_16:                               ; %if.end28
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s6, 63
 load  s1, 1
 load  s2, s1
 load  s3, 0
 fetch  s8, sF, 2
 compare s6, s8
 jump  .C, L_BB0_18
; BB#17:                                ; %if.end28
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s2, s3
L_BB0_18:                               ; %if.end28
                                        ;   in Loop: Header=BB0_3 Depth=1
 compare s1, s0
 jump  .C, L_BB0_20
```

```
; BB#19:                                ; %if.end28
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s1, s3
L_BB0_20:                               ; %if.end28
                                        ;   in Loop: Header=BB0_3 Depth=1
 compare s0, 1
 jump  .Z, L_BB0_22
; BB#21:                                ; %if.end28
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s2, s1
L_BB0_22:                               ; %if.end28
                                        ;   in Loop: Header=BB0_3 Depth=1
 load  s6, s0
 fetch  s1, sF, 2
 compare s2, 0
 jump  .NZ, L_BB0_32
L_BB0_23:                               ; %for.inc38
                                        ;   Parent Loop BB0_3 Depth=1
                                        ; =>  This Loop Header: Depth=2
                                        ;       Child Loop BB0_24 Depth 3
 load  s2, s1
 sr0  s2
 sr0  s2
 sr0  s2
 load  s3, s6
 rr  s3
 rr  s3
 rr  s3
 and  s3, 224
 load  s8, s1
 and  s8, 7
 load  s9, 1
 load  sA, s9
 load  sB, 0
 load  sC, sB
 compare s8, 0
 jump  .Z, L_BB0_25
L_BB0_24:                               ; %for.inc38
                                        ;   Parent Loop BB0_3 Depth=1
                                        ;     Parent Loop BB0_23 Depth=2
                                        ; =>    This Inner Loop Header: Depth=3
 sl0  sA
 sla  sC
 sub  s8, 1
 jump  .NZ, L_BB0_24
L_BB0_25:                               ; %for.inc38
                                        ;   in Loop: Header=BB0_23 Depth=2
 load  s8, s9
 or  s2, s3
 load  s3, s5
 addn  s3, s2
 fetch  s2, s3
 add  s1, s7
 addcy  s6, s4
 or  s2, sA
 store  s2, s3
 compare s1, 64
 jump  .C, L_BB0_27
; BB#26:                                ; %for.inc38
                                        ;   in Loop: Header=BB0_23 Depth=2
 load  s8, sB
L_BB0_27:                               ; %for.inc38
                                        ;   in Loop: Header=BB0_23 Depth=2
 compare s6, 0
 jump  .Z, L_BB0_29
; BB#28:                                ; %for.inc38
                                        ;   in Loop: Header=BB0_23 Depth=2
```

```
 load  s9, sB
L_BB0_29:                               ; %for.inc38
                                        ;   in Loop: Header=BB0_23 Depth=2
 compare s6, 1
 jump  .Z, L_BB0_31
; BB#30:                                ; %for.inc38
                                        ;   in Loop: Header=BB0_23 Depth=2
 load  s8, s9
L_BB0_31:                               ; %for.inc38
                                        ;   in Loop: Header=BB0_23 Depth=2
 compare s8, 0
 jump  .NZ, L_BB0_23
L_BB0_32:                               ; %for.cond7.backedge
                                        ;   in Loop: Header=BB0_3 Depth=1
 fetch  s6, sF, 2
 add  s6, 2
 store  s6, sF, 2
 addcy  s0, 0
 add  s7, 1
 addcy  s4, 0
 fetch  s6, sF, 0
 addn  s6, 1
 store  s6, sF, 0
 jump  L_BB0_3
L_tmp0:
 .size _main, L_tmp0-_main
```