HARP Instruction Set Manual

Chad D. Kersey

September 14, 2015

Disclaimer

This document, like the work it documents, is very much a work in progress. Send any corrections, updates, suggestions, or complaints (preferably in patch form, this file is under `harptool/doc` in the HarpTool repo) to `cdkersey@gatech.edu`.

Introduction

HARP is two things, a multi-year Heterogeneous Architecture Research Project, and an implementation of a specific Heterogeneous Architecture Research Prototype. It is for the latter that the HARP instruction set architectures have been created. This is a space of SIMT(GPU) oriented RISC-like instruction sets with the following properties:

- Full predication
- Assembly language level compatibility
- SIM[DT] parallelism
- Little endianness
- 8-bit byte size
- Customizability

The customizability of the HARP ISAs is illustrated by facts missing from this list of features. The data path width, instruction encodings, number of registers (general purpose and predicate) are all left up to the implementation. Harptool; the HARP assembler, linker, emulator, and disassembler, is passed information about the ISA through an architecture identifier string, or `ArchID`. An `ArchID` uniquely identifies a HARP ISA.

1 Architecture Identifier String (`ArchID`)

The best way to understand the multifaceted parameterizablity of the HARP ISAs is to study the architecture identifier strings used to uniquely identify a single HARP instruction set architecture. We'll start by breaking down Harptool's default `ArchID`: `8w32/32/8/8`:

Field	Meaning
`8`	8-byte (64-bit) registers and addresses
`w`	Word-based (64-bit) fixed-width instruction encoding
`32`	32 general-purpose registers per lane
`32`	32 predicate registers per lane
`8`	8 SIMD lanes
`8`	8 warps (thread groups)

All ArchIDs have a similar format, although the final two fields can be omitted, as object files are still fully compatible even if the dimensions of the core change.

2 HarpTool

The assembler/linker/emulator/disassembler program for HARP is called HarpTool. It is a multiple-function executable, its function selected with the first command line argument. When run with no command line arugments the HarpTool executable prints a help message explaining the available command line arguments.

All of the HARP utilities can take an archID as a command line parameter. If none is provided, a default will be assumed.

2.1 Assembler

The assembler converts assembly files to HOF, the Harp Object Format.

2.2 Linker

The linker combines HOF files and produces raw RAM images for use by the emulator. An intended future use is the conversion of multiple HOF files to statically-linked HOF executables.

2.3 Disassembler

The disassembler is used to convert HOF files to equivalent assembly files. One of its intended uses is the conversion of HOF object files between different HARP ISAs, say from 8w32/32 to 8b32/32.

2.4 Test Programs

In the `harptool/test` directory there is a set of test programs. The makefile in this directory assembles, links, and emulates them, placing the output in plain text files.

2.4.1 `hello.s`

The simplest example prints a message and exits.

2.4.2 `2thread.s`

`2thread` performs a vector addition across two threads.

2.4.3 `sieve.s`

`sieve` performs the Sieve of Eratosthenes in a single thread and prints the results, including the count of total prime numbers found.

3 Instruction Encoding

There are two currently-supported types of instruction encoding, but they all share a similar basic structure. The opcodes and types of fields required by each instruction are identical, differentiated only by the number of bits available for each type of field and the way predication is specified.

3.1 Argument Classes

Instructions can be broadly categorized by the types of arguments they require. The bit fields in the instruction encodings depend heavily on this quality.

Argument Class	Description	Example
AC_NONE	No arguments	`di;`
AC_2REG	2 GPRs, 1 in, 1 out	`neg %r1, %r2;`
AC_2IMM	1 immediate in, 1 GPR out	`ldi %r1, #0xff;`
AC_3REG	3 GPRs, 2 in, 1 out	`add %r1, %r2, %r2;`
AC_3PREG	3 pred. regs, 2 in, 1 out	`andp @p0, @p0, @p1;`
AC_3IMM	GPR in, imm. in, GPR out	`andi %r1, %r3, #3;`
AC_3REGSRC	3 GPRs in	`tlbadd %r0, %r1, %r2;`
AC_1IMM	1 imm in	`jmpi label;`
AC_1REG	1 reg in	`jmpr %r2`
AC_3IMMSRC	2 GPRs in, 1 imm. in	`st %r1, %r2, #10;`
AC_PREG_REG	GPR in, pred. reg. out	`iszero @p0, %r3;`
AC_2PREG	2 pred. regs, 1 in, 1 out	`notp @p0, @p0;`

3.2 Opcode/Instruction Class Table

00	"nop"	NONE	01	"di"	NONE	02	"ei"	NONE
03	"tlbadd"	3REGSRC	04	"tlbflush"	NONE	05	"neg"	2REG
06	"not"	2REG	07	"and"	3REG	08	"or"	3REG
09	"xor"	3REG	0a	"add"	3REG	0b	"sub"	3REG
0c	"mul"	3REG	0d	"div"	3REG	0e	"mod"	3REG
0f	"shl"	3REG	10	"shr"	3REG	11	"andi"	3IMM
12	"ori"	3IMM	13	"xori"	3IMM	14	"addi"	3IMM
15	"subi"	3IMM	16	"muli"	3IMM	17	"divi"	3IMM
18	"modi"	3IMM	19	"shli"	3IMM	1a	"shri"	3IMM
1b	"jali"	2IMM	1c	"jalr"	2REG	1d	"jmpi"	1IMM
1e	"jmpr"	1REG	1f	"clone"	1REG	20	"jalis"	3IMM
21	"jalrs"	3REG	22	"jmprt"	1REG	23	"ld"	3IMM
24	"st"	3IMMSRC	25	"ldi"	2IMM	26	"rtop"	PREG_REG
27	"andp"	3PREG	28	"orp"	3PREG	29	"xorp"	3PREG
2a	"notp"	2PREG	2b	"isneg"	PREG_REG	2c	"iszero"	PREG_REG
2d	"halt"	NONE	2e	"trap"	NONE	2f	"jmpru"	1REG
30	"skep"	1REG	31	"reti"	NONE	32	"tlbrm"	1REG
33	"itof"	2REG	34	"ftoi"	2REG	35	"fadd"	3REG
36	"fsub"	3REG	37	"fmul"	3REG	38	"fdiv"	3REG
39	"fneg"	2REG	3a	"wspawn"	3REG	3b	"split"	NONE
3c	"join"	NONE	3d	"bar"				

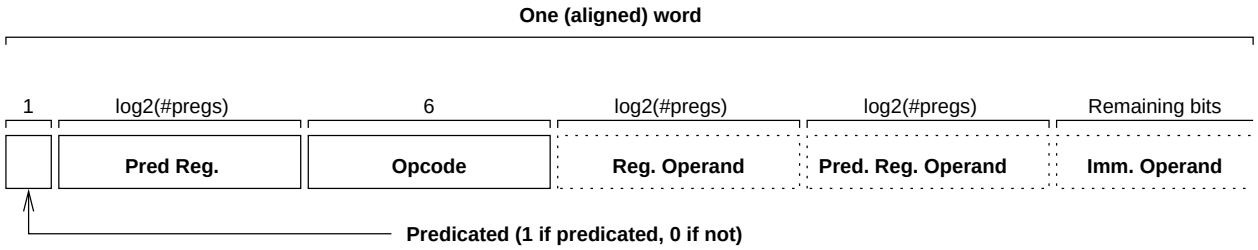
3.3 Word Encoding

Word-based instruction encodings all share the initial fields:

- The most-significant bit is 1 if the instruction is predicated and 0 otherwise.
- The next log_2(#pred_regs) specify the predicate register.
- The next 6 bits are used for the opcode.

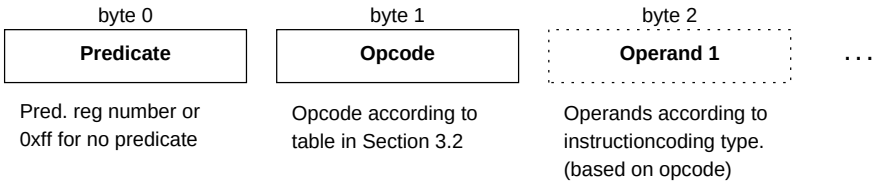
After this, the operands of the instruction are ordered corresponding to their ordering in the assembly language, sized according to the following rules:

- Register operands are log_2(#GPRs) bits long, or just enough bits to uniquely identify a register.
- Predicate register operands are log_2(#pred_regs) bits long, or just enough bits to uniquely identify a predicate register.
- Immediate fields are always the last field and occupy the remaining bits of the instruction. All immediate fields are sign extended to the length of a machine word.



3.4 Byte Encoding

In the byte encoding, each field of the instruction (predicate, opcode, operands) occupies a byte, with the exception of immediates, which occupy an unaligned word. All instructions have a predicate and opcode byte. The predicate byte is all ones if the instruction is not predicated; oterwise the predicate byte contains the predicate register number used to predicate the instruction. Just like the word-based instruction encoding, registers appear in the same order as the assembly language, destination-first.



4 Assembly Language

The assembly language is fairly easy to pick up from the HarpTools examples. It is RISC-like, and written destination register first (in this it differs from Unix assembly syntax). Registers names are prefixed with the percent sign (%) and predicate register names with the at symbol (@). Predicated instructions are prefixed with the predicate register name and a question mark:

```
@p0 ? addi %r7, %r1, #1
```

A small set of directives is provided to express non-instruction data:

Directive	Use
`.align 256`	Align next symbol to a multiple of 256 bytes.
`.word 0x1234`	Insert a word with the value `0x1234`.
`.byte 0xff`	Insert a byte with the value `0xff`.
`.def SYM 123`	Replace SYM with 123 in immediate operands.
`.entry`	Make the next label the HOF executable entry point.
`.global`	Give the next label global (external) linkage.
`.perm rw`	Set HOF permissions of the next label to read/write.
`.string "Str"`	Create a null terminated string.

5 Instruction Set

5.1 Trivial Instruction

Instruction	Description
`nop`	No operation.

5.2 Privileged Instructions

Instruction	Description
`ei`	Enable interrupts.
`di`	Disable interrupts.
`skep` %addr	Set kernel entry point.
`tlbadd` %virt, %phys, %flags	Add an entry to the TLB.
`tlbrm` %virt	Remove entry corresponding to virt. address from TLB.
`tlbflush`	Remove all but default entry from TLB.
`jmpru` %addr	Jump indirect and switch to user mode.
`reti`	Return from interrupt.
`halt`	Halt CPU until next interrupt.

The flags register used by `tlbadd` stores, in its least-significant four bits, in order from most to least significant:

Bit	Meaning
`kx`	Kernel can execute.
`kw`	Kernel can write.
`kr`	Kernel can read.
`ux`	User can execute.
`uw`	User can write.
`ur`	User can read.

5.3 Memory Loads/Stores

Instruction	Description
`st` %val, %base, #OFFSET	Store.
`ld` %dest, %base, #OFFSET	Load.

5.4 Predicate Manipulation

Instruction	Description
`andp` @dest, @src1, @src2	Logical and.
`orp` @dest, @src1, @src2	Logical or.
`xorp` @dest, @src1, @src2	Exclusive or.
`notp` @dest, @src	Complement.

5.5 Value Tests

Instruction	Description
`rtop` @dest, %src	Set @dest if %src is nonzero.
`isneg` @dest, %src	Set @dest if %src is negative.
`iszero` @dest, %src	Set @dest if %src is zero.

5.6 Immediate Integer Arithmetic/Logic

Instruction	Description
`ldi` %dest, #Imm	Load immediate.
`addi` %dest, %src1, #Imm	Add immediate.
`subi` %dest, %src1, #Imm	Subtract immediate.
`muli` %dest, %src1, #Imm	Multiply immediate.
`divi` %dest, %src1, #Imm	Divide immediate.
`modi` %dest, %src1, #Imm	Modulus immediate.
`shli` %dest, %src1, #Imm	Shift left immediate.
`shri` %dest, %src1, #Imm	Shift right immediate.
`andi` %dest, %src1, #Imm	And immediate.
`ori` %dest, %src1, #Imm	Or immediate.
`xori` %dest, %src1, #Imm	Xor immediate.

5.7 Register Integer Arithmetic/Logic

Instruction	Description
`add` %dest, %src1, %src2	Add.
`sub` %dest, %src1, %src2	Subtract.
`mul` %dest, %src1, %src2	Multiply.
`div` %dest, %src1, %src2	Divide.
`mod` %dest, %src1, %src2	Modulus.
`shl` %dest, %src1, %src2	Shift left.
`shr` %dest, %src1, %src2	Shift right.
`and` %dest, %src1, %src2	And.
`or` %dest, %src1, %src2	Or.
`xor` %dest, %src1, %src2	Xor.
`neg` %dest, %src1, %src2	Two's complement.
`not` %dest, %src1, %src2	Bitwise complement.

5.8 Floating Point Arithmetic

These operations operate on real numbers in an implementation-determined format, which can be fixed point or floating point.

Instruction	Description
`itof` %dest, %src	Signed integer to floating point.
`ftoi` %dest, %src	Floating point to signed integer.
`fneg` %dest, %src	Negate (complement sign bit).
`fadd` %dest, %src1, %src2	Floating point add.
`fsub` %dest, %src1, %src2	Floating point subtract.
`fmul` %dest, %src1, %src2	Floating point multiply.
`fdiv` %dest, %src1, %src2	Floating point divide.

5.9 Control Flow

Instruction	Description
`jmpi` #RelDest	Jump to immediate (PC-relative).
`jmpr` %addr	Jump indirect.
`jali` %link, #RelDest	Jump and link immediate.
`jalr` %link, %reg	Jump and link indirect.

5.10 SIMD Control

Instruction	Description
`clone` %lane	Clone register state into specified lane.
`jalis` %link, %n, #RelDest	Jump and link immediate, spawning N active lanes.
`jalrs` %link, %n, %dest	Jump and link indirect, spawning N active lanes.
`jmprt` %addr	Jump indirect, terminating execution on all but a single lane.
`split`	Control flow diverge.
`join`	Control flow reconverge.

5.11 Warp Control

Instruction	Description
`wspawn` %dest, %pc, %src	Create new warp, copying %src in current warp to to %dest in new warp.
`bar` %id, %n	Barrier of %n warps. Identified by %id.

5.12 User/Kernel Interteraction

Instruction	Description
`trap`	User-generated interrupt.

6 Interrupts

The HARP interrupt mechanism is simple. For SIMD lane 0, there is a shadow register file, program counter, and active lane count. When an interrupt occurs, the state of lane zero is saved into these shadow registers, and execution resumes at the kernel entry point. The type of interrupt is specified by the value placed in register 0 at this time, according to the following table:

Number	Description
0	Trap (user-generated interrupt)
1	Page fault due to absence from TLB
2	Page fault due to permission violation
3	Invalid/unsupported instruction
4	Divergent branch
5	Numerical domain (divide by zero)
6-7	(reserved for future exceptions)
8	Console input

The first eight interrupt numbers are reserved for internal CPU-generated exceptions, and all of the remaining numbers are free for use by hardware.

7 Application Binary Interface

The ABI assumes a set of at least four general purpose registers. The frame pointer is optional and can be stored on the stack itself if necessary. The stack pointer and link register, in this order, are always the two highest-numbered registers. If 8 or more registers are available, the frame pointer may be the register one less than the register number of the stack pointer.

- The lower-numbered half of the registers are caller-saved (temporary).
- The upper-numbered half are therefore callee-saved.
- The callee is responsible for adjusting the stack and frame pointers, if such adjustment is required.
- The stack grows toward smaller addresses (subtract to push, add to pop).

- Pointer function arguments and numerical arguments that can fit in a single register are passed through temporary registers, starting with register `%r0`. If more registers are required than there are temporary registers available, stack space at addresses less than the stack pointer is used.
- Record (struct) return values and numerical return values larger than the word size are always passed on the stack. The caller is responsible for allocating the necessary space. The stack pointer at the time of call is a pointer to the returned structure. All other return values are returned in `%r0`.

8 SIMD Operation

The HARP ISAs are inherently SIMD. In addition to designs with a single set of functional units and architectural registers, designs are allowed that replicate these while retaining a single front-end and memory system. This allows for multiple threads executing the same stream of instructions to simultaneously occupy multiple "lanes" of the processor. When a predicated control flow instruction occurs without unanimous agreement among predicate registers, a divergent branch has occurred. The current response to this is to trap to the operating system (interrupt number 4).

8.1 Instructions for SIMD Operation

The `clone`, `jalis`, `jalrs`, and `jmprt` instructions form the basis of SIMD context control in the HARP instruction set. Context is created using `clone`, the waiting threads are spawned using `jalrs` or `jalis`, "jump-and-link immediate/register and spawn", and finally the parallel section returns using `jmprt`, "jump register and terminate", best thought of as "return and terminate."

There are times when a control flow operation will need to be predicated, going one direction on some lanes and the other direction on other lanes. For this, the HARP instruction set provides the `split` and `join` instructions. When a predicated `split` is first encountered, only the lanes for which the `split`'s predicate are true are allowed to continue. The other lanes are masked out until the corresponding `join` is encountered. The first time `join` is reached, control flow returns to the instruction following the corresponding `split` with the set of masked-out lanes complemented. The second time the same `join` is reached, control flow falls through and the original lane mask is restored. A hardware stack is maintained to keep track of nested `split`s.

9 Default I/O Devices

The emulator currently only supports a single I/O device, simple console I/O. Writing to the address `0x800...0` (an address with its MSB set and all other bits cleared) causes text to be written to the display. Input on this console interface causes an interrupt (number 8).