

Adding Custom Instructions To The RISC-V Toolchain

March 16th, 2023
Garrett O'Neill

Contents

<u>Chapter#</u>	<u>Description</u>	<u>Page#</u>
1	Summary.....	3
2	Links, Resources.....	4
3	Configuring The Virtual Machine.....	5
4	Downloading The RISC-V Toolchain.....	6
5	Downloading Additional Packages.....	7
6	Designing A Custom Instruction.....	8
7	Source Code Edits.....	9
7-1	Header Files.....	10
7-2	C Files	14
8	Building The Toolchain.....	18
9	Compiling A Test Program.....	22

1) Summary

This document gives a step-by-step guide on how to:

1. Design a custom RISC-V instruction.
2. Add that instruction to the RISC-V toolchain source code.
3. Rebuild the RISC-V toolchain from source.
4. Compile and assemble source code (with your custom instruction) for a RISC-V target.
For demo purposes, the target device used in this document will be the CPE 233 Multicycle OTTER.

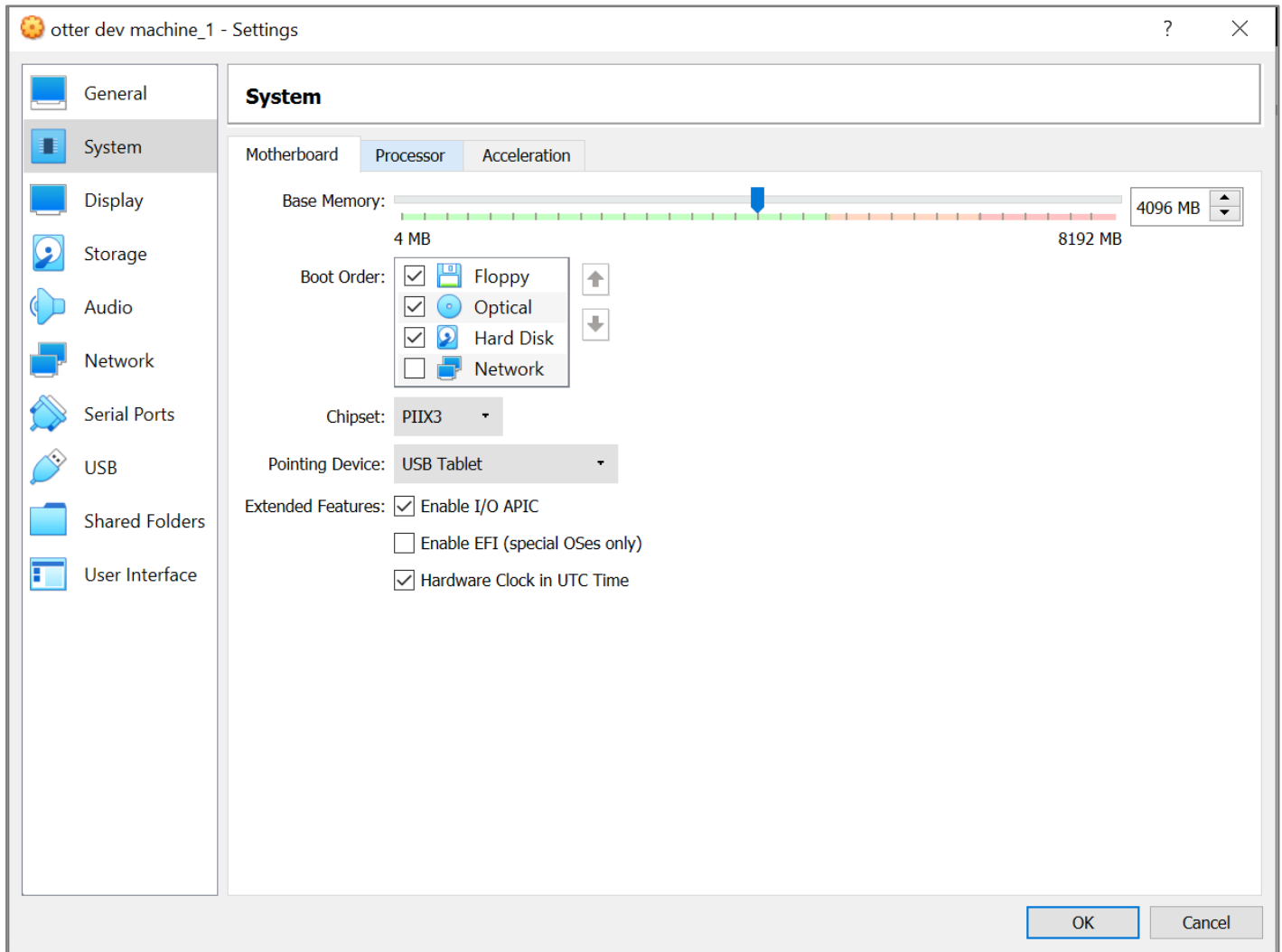
Important Notes:

- To build our modified RISC-V toolchain, we will be using an Ubuntu environment running on a virtual machine.
- This guide is tailored specifically to the RV32I instruction set for bare-metal applications only. If building a toolchain that includes additional RISC-V extensions, some of the steps in this document will need to be modified. I will call out these steps, however the exact modifications that must be made will require some additional research outside of this document.

2) Links, Resources

- Source repository for this document: https://github.com/geoneill12/Extending_RISCV
- Multicycle OTTER development tools: <https://github.com/geoneill12/otter-tools-p1>
- RISC-V toolchain: <https://github.com/riscv-collab/riscv-gnu-toolchain>
- Ubuntu download: <https://ubuntu.com/download/desktop>

3) Configuring The Virtual Machine



- Before starting your virtual machine, increase the base memory to 4096 MB. Normally I only allocate 2048 MB to a virtual machine; however, on one occasion the toolchain build failed due to memory exhaustion. For good measure I doubled the memory to 4096 MB, which fixed the problem. This may slow down the host machine slightly, but we only need to do this while the toolchain is building.
- Additionally, the toolchain source files require about 6.65 GB of disk space. Make sure your virtual machine has enough disk space on your host machine to accommodate this.

Note: If already running Ubuntu as your primary operating system, skip to chapter 4.

4) Downloading The RISC-V Toolchain

```
student@student-VirtualBox:~/Documents$ git clone https://github.com/riscv/riscv-gnu-toolchain --recursive
Cloning into 'riscv-gnu-toolchain'...
remote: Enumerating objects: 8228, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 8228 (delta 0), reused 1 (delta 0), pack-reused 8224
Receiving objects: 100% (8228/8228), 5.12 MiB | 6.75 MiB/s, done.
Resolving deltas: 100% (4100/4100), done.
Submodule 'binutils' (https://sourceware.org/git/binutils-gdb.git) registered for path 'binutils'
Submodule 'dejagnu' (https://git.savannah.gnu.org/git/dejagnu.git) registered for path 'dejagnu'
Submodule 'gcc' (https://gcc.gnu.org/git/gcc.git) registered for path 'gcc'
Submodule 'gdb' (https://sourceware.org/git/binutils-gdb.git) registered for path 'gdb'
Submodule 'glibc' (https://sourceware.org/git/glibc.git) registered for path 'glibc'
Submodule 'musl' (https://git.musl-libc.org/git/musl) registered for path 'musl'
Submodule 'newlib' (https://sourceware.org/git/newlib-cygwin.git) registered for path 'newlib'
Submodule 'pk' (https://github.com/riscv-software-src/riscv-pk.git) registered for path 'pk'
Submodule 'qemu' (https://gitlab.com/qemu-project/qemu.git) registered for path 'qemu'
Submodule 'spike' (https://github.com/riscv-software-src/riscv-isa-sim.git) registered for path 'spike'
Cloning into '/home/student/Documents/riscv-gnu-toolchain/binutils'...
```

Clone the RISC-V toolchain into the directory of your choice:

`git clone https://github.com/riscv/riscv-gnu-toolchain --recursive`

The repository uses submodules. If `--recursive` is not included, most of the source files will not be fetched, since they reside in these submodules.

5) Downloading Additional Packages

```
student@student-VirtualBox:~/Documents$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev
libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build
[sudo] password for student:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'libexpat1-dev' instead of 'libexpat-dev'
autoconf is already the newest version (2.69-11.1).
automake is already the newest version (1:1.16.1-4ubuntu6).
autotools-dev is already the newest version (20180224.1).
bc is already the newest version (1.07.1-2build1).
bison is already the newest version (2:3.5.1+dfsg-1).
flex is already the newest version (2.6.4-6.2).
gawk is already the newest version (1:5.0.1+dfsg-1).
libmpc-dev is already the newest version (1.1.0-1).
libmpfr-dev is already the newest version (4.0.2-1).
libtool is already the newest version (2.4.6-14).
patchutils is already the newest version (0.3.4-2).
python3 is already the newest version (3.8.2-0ubuntu2).
gperf is already the newest version (3.1-1build1).
texinfo is already the newest version (6.7.0.dfsg.2-5).
build-essential is already the newest version (12.8ubuntu1.1).
curl is already the newest version (7.68.0-1ubuntu2.16).
libexpat1-dev is already the newest version (2.2.9-1ubuntu0.6).
libgmp-dev is already the newest version (2:6.2.0+dfsg-4ubuntu0.1).
zlib1g-dev is already the newest version (1:1.2.11.dfsg-2ubuntu1.5).
The following packages were automatically installed and are no longer required:
  linux-headers-5.4.0-132 linux-headers-5.4.0-132-generic linux-image-5.4.0-132-generic linux-modules-5.4.0-132-generic
  linux-modules-extra-5.4.0-132-generic
Use 'sudo apt autoremove' to remove them.
The following NEW packages will be installed:
  ninja-build
0 upgraded, 1 newly installed, 0 to remove and 92 not upgraded.
Need to get 107 kB of archives.
After this operation, 338 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://us.archive.ubuntu.com/ubuntu focal/universe amd64 ninja-build amd64 1.10.0-1build1 [107 kB]
Fetched 107 kB in 16s (6,856 B/s)
Selecting previously unselected package ninja-build.
(Reading database ... 264838 files and directories currently installed.)
Preparing to unpack .../ninja-build_1.10.0-1build1_amd64.deb ...
Unpacking ninja-build (1.10.0-1build1) ...
Setting up ninja-build (1.10.0-1build1) ...
Processing triggers for man-db (2.9.1-1) ...
student@student-VirtualBox:~/Documents$
```

Run the following command to ensure that all packages needed to build the toolchain are installed:

```
sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev
libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev
libexpat-dev ninja-build
```

6) Designing A Custom Instruction

To illustrate how to design a custom instruction, I will use a custom instruction called **otter**, with the following semantics:

otter rd, rs1, rs2

The **otter** instruction is a 32-bit, R-type instruction. Below are the different instruction types for 32-bit RISC-V instructions:

31	27	26	25	24	20	19	15	14	12	11	7	6	0					
funct7				rs2		rs1	funct3		rd		opcode			R-type				
imm[11:0]						rs1	funct3		rd		opcode			I-type				
imm[11:5]				rs2		rs1	funct3		imm[4:0]		opcode			S-type				
imm[12:10:5]				rs2		rs1	funct3		imm[4:1 11]		opcode			B-type				
imm[31:12]									rd		opcode			U-type				
imm[20:10:1 11 19:12]									rd		opcode			J-type				

An R-type instruction has the following:

- A 7-bit opcode field (bits 6-0).
- A 10-bit, non-contiguous function field (bits 31-25 and 14-12).
- 3 operands: destination register, source register 1, source register 2 (bits 11-7, 19-15, and 24-20 respectively).

The opcode and function fields are always constant, so we need to choose values for these fields. For the **otter** instruction, I have arbitrarily chosen the following values for the opcode and function fields:

- opcode = 7'b0100111
- function = 10'b0000000110

The opcode I picked is unique and does not match the opcode of any of the RV32I base instructions.

Note: The official RISC-V specification contains guidelines on ISA extensions, including naming conventions, opcode format, etc.

7) Source Code Edits

To implement our custom instruction into the RISC-V toolchain, we need to modify the following 2 files:

`riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h`

`riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c`

Additionally, the same modifications can be made to the following 2 files so that we can use our custom instruction with gdb:

`riscv-gnu-toolchain/gdb/include/opcode/riscv-opc.h`

`riscv-gnu-toolchain/gdb/opcodes/riscv-opc.c`

7-1) Source Code Edits – Header Files

First, we need to create 2 macros based on the opcode and function values we picked for our instruction in chapter 6. For our **otter** instruction, these macros will be called “MATCH_OTTER” and “MASK_OTTER”.

MATCH_OTTER

1. Write out the complete machine code value of the new instruction, filling in the opcode and function bits using the values chosen in chapter 6. For the operands, replace them with X's.
2. Replace all X's with 1's.
3. The resulting hexadecimal value is the “MATCH_OTTER” macro.

MATCH_OTTER	funct7								rs2				rs1				funct3				rd				opcode										
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	1	1	1	0	X	X	X	X	X	0	1	0	0	1	1			
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	1			
	0				0				0				0				6				0				2				7						
																																0x6027			

MASK_OTTER

1. Write out the complete machine code value of the new instruction, this time placing all 1's in the opcode and function fields, and 0's in all the operand fields.
2. The resulting hexadecimal value is the “MASK_OTTER” macro.

MASK_OTTER	funct7				rs2				rs1				funct3				rd				opcode											
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1
	f				e				0				0				7				0				7				f			
	0xfe00707f																															

The resulting macros for our **otter** instruction are:

```
#define MATCH_OTTER    0x6027
```

```
#define MASK_OTTER     0xfe00707f
```

7-1) Source Code Edits – Header Files

Some Notes On The Instruction Macros

- The toolchain seems to use the “MATCH_” macro to determine the value of the instruction’s constant fields (opcode and function), while using the “MASK_” macro to differentiate the location of the operand fields from the location of the constant fields.
- If creating an instruction with a period in it (i.e. otter.r, otter.h, etc.) replace the period with an underscore in the macro name.
 - ex) “otter.r” would have macros “MATCH_OTTER_R” and “MASK_OTTER_R”.

7-1) Source Code Edits – Header Files

```
21  #ifndef RISCV_ENCODING_H
22  #define RISCV_ENCODING_H
23  /* Instruction opcode macros. */
24
25  #define MATCH_OTTER 0x6027
26  #define MASK_OTTER  0xfe00707f
27
28  #define MATCH_SLLI_RV32 0x1013
29  #define MASK_SLLI_RV32  0xfe00707f
30  #define MATCH_SRLI_RV32 0x5013
31  #define MASK_SRLI_RV32  0xfe00707f
```

Add the “MATCH_OTTER” and “MASK_OTTER” macros to the following header files:

riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h

riscv-gnu-toolchain/gdb/include/opcode/riscv-opc.h

7-1) Source Code Edits – Header Files

```
2788  #endif /* RISC_V_ENCODING_H */
2789  #ifdef DECLARE_INSN
2790
2791  DECLARE_INSN(otter, MATCH_OTTER, MASK_OTTER)
2792
2793  DECLARE_INSN(slli_rv32, MATCH_SLLI_RV32, MASK_SLLI_RV32)
2794  DECLARE_INSN(srli_rv32, MATCH_SRLI_RV32, MASK_SRLI_RV32)
2795  DECLARE_INSN(srai_rv32, MATCH_SRAI_RV32, MASK_SRAI_RV32)
2796  DECLARE_INSN(frflags, MATCH_FRFLAGS, MASK_FRFLAGS)
2797  DECLARE_INSN(fsflags, MATCH_FSFLAGS, MASK_FSFLAGS)
2798  DECLARE_INSN(fsflagsi, MATCH_FSFLAGSI, MASK_FSFLAGSI)
```

In the same header files from the previous page, add the following line:

```
DECLARE_INSN(otter, MATCH_OTTER, MASK_OTTER)
```

This declaration requires the instruction name, and our “MATCH_” and “MASK_” macros.

Note: If creating an instruction with a period in it (i.e. otter.r, otter.h, etc.) replace the period with an underscore in the declaration above.

- ex) “otter.r” would be `DECLARE_INSN(otter_r, MATCH_OTTER_R, MASK_OTTER_R)`

7-2) Source Code Edits – C Files

```
311  const struct riscv_opcode riscv_opcodes[] =
312  {
313  /* name, xlen, isa, operands, match, mask, match_func, pinfo. */
314
315  {"otter",      0, INSN_CLASS_I, "d,s,t",      MATCH_OTTER, MASK_OTTER, match_opcode, 0 },
316
317  /* Standard hints. */
318  {"prefetch.i", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_I, MASK_PREFETCH_I, match_opcode, 0 },
319  {"prefetch.r", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_R, MASK_PREFETCH_R, match_opcode, 0 },
320  {"prefetch.w", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_W, MASK_PREFETCH_W, match_opcode, 0 },
321  {"pause",      0, INSN_CLASS_ZIHINTPAUSE, "", MATCH_PAUSE, MASK_PAUSE, match_opcode, 0 },
322
323  /* Basic RVI instructions and aliases. */
324  {"unimp",      0, INSN_CLASS_C, "",          0, 0xffffU, match_opcode, INSN_ALIAS },
```

There is an array of structs called “riscv_opcodes” in each of the following C files:

riscv-gnu-toochain/binutils/opcodes/riscv-opc.c

riscv-gnu-toochain/gdb/opcodes/riscv-opc.c

Every RISC-V instruction has an entry. We need to add an entry for our **otter** instruction.

7-2) Source Code Edits – C Files

```
311 const struct riscv_opcode riscv_opcodes[] =
312 {
313 /* name, xlen, isa, operands, match, mask, match_func, pinfo. */
314
315 {"otter",      0, INSN_CLASS_I, "d,s,t",    MATCH_OTTER, MASK_OTTER, match_opcode, 0 },
316
317 /* Standard hints. */
318 {"prefetch.i", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_I, MASK_PREFETCH_I, match_opcode, 0 },
319 {"prefetch.r", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_R, MASK_PREFETCH_R, match_opcode, 0 },
320 {"prefetch.w", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_W, MASK_PREFETCH_W, match_opcode, 0 },
321 {"pause",      0, INSN_CLASS_ZIHINTPAUSE, "", MATCH_PAUSE, MASK_PAUSE, match_opcode, 0 },
322
323 /* Basic RVI instructions and aliases. */
324 {"unimp",      0, INSN_CLASS_C, "",        0, 0xffffU, match_opcode, INSN_ALIAS },
```

The template for each struct entry is:

{name, xlen, isa, operands, match, mask, match_func, pinfo}

name: Name of our instruction.

xlen: Bit width of an integer register, either 32 or 64.

isa: The specific ISA our instruction is targeting.

operands: Symbols that specify the exact type of operands an instruction uses.

match: The “MATCH_” macro defined in “riscv-opc.h”.

mask: The “MASK_” macro defined in “riscv-opc.h”.

match_func: Not certain what this field does, but it may have to do with how the toolchain checks that an instruction was formatted correctly.

pinfo: Not certain what this field does.

7-2) Source Code Edits – C Files

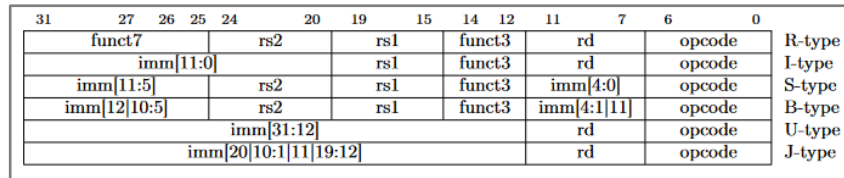
```
311 const struct riscv_opcode riscv_opcodes[] =
312 {
313 /* name, xlen, isa, operands, match, mask, match_func, pinfo. */
314
315 {"otter",      0, INSN_CLASS_I, "d,s,t",      MATCH_OTTER, MASK_OTTER, match_opcode, 0 },
316
317 /* Standard hints. */
318 {"prefetch.i", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_I, MASK_PREFETCH_I, match_opcode, 0 },
319 {"prefetch.r", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_R, MASK_PREFETCH_R, match_opcode, 0 },
320 {"prefetch.w", 0, INSN_CLASS_ZICBOP, "f(s)", MATCH_PREFETCH_W, MASK_PREFETCH_W, match_opcode, 0 },
321 {"pause",      0, INSN_CLASS_ZIHINTPAUSE, "", MATCH_PAUSE, MASK_PAUSE, match_opcode, 0 },
322
323 /* Basic RVI instructions and aliases. */
324 {"unimp",      0, INSN_CLASS_C, "",          0, 0xffffU, match_opcode, INSN_ALIAS },
```

Our **otter** instruction will have the following entry

```
{"otter", 0, INSN_CLASS_I, "d,s,t", MATCH_OTTER, MASK_OTTER, match_opcode, 0}
```

- | | |
|---------------|---|
| "otter": | Name of our instruction. |
| 0: | Our instruction is targeting a 32-bit architecture, however specifying 0 appears to allow an instruction to be used for both 32-bit and 64-bit architectures. |
| INSN_CLASS_I: | Entry for the RV32I ISA. |
| "d,s,t" : | Symbols for our operands. These are explained more thoroughly on the next page. |
| MATCH_OTTER: | The "MATCH_" macro defined in "riscv-opc.h". |
| MASK_OTTER: | The "MASK_" macro defined in "riscv-opc.h". |
| match_opcode: | Since I'm not certain what this field does, I reused the same entry used by other RV32I instructions. |
| 0: | Since I'm not certain what this field does, I reused the same entry used by other RV32I instructions. |

7-2) Source Code Edits – C Files



Format	RV32I Instruction	Operands	Struct Entry	Symbol	Symbol Meaning	Format
U	lui	rd, imm	"d,u"	d	Destination register	R, I, U, J
	auipc	rd, imm	"d,u"	s	Source register 1	R, I, S, B
J	jal	rd, imm	"d,a"	t	Source register 2	R, S, B
I	jalr	rd, rs1, imm	"d,s,j"	j	12-bit immediate	I
B	beq	rs1, rs2, imm	"s,t,p"	>	Lowest 5 bits of 12-bit immediate	I
	bne	rs1, rs2, imm	"s,t,p"	o	12-bit immediate	I
	blt	rs1, rs2, imm	"s,t,p"	q	12-bit immediate	S
	bge	rs1, rs2, imm	"s,t,p"	p	12-bit immediate	B
	bltu	rs1, rs2, imm	"s,t,p"	a	20-bit immediate	J
	bgeu	rs1, rs2, imm	"s,t,p"	u	20-bit immediate	U
I	lb	rd, imm(rs1)	"d,o(s)"			
	lh	rd, imm(rs1)	"d,o(s)"			
	lw	rd, imm(rs1)	"d,o(s)"			
	lbu	rd, imm(rs1)	"d,o(s)"			
S	lhu	rd, imm(rs1)	"d,o(s)"			
	sb	rs2, imm(rs1)	"t,q(s)"			
	sh	rs2, imm(rs1)	"t,q(s)"			
	sw	rs2, imm(rs1)	"t,q(s)"			
I	addi	rd, rs1, imm	"d,s,j"			
	slti	rd, rs1, imm	"d,s,j"			
	sltiu	rd, rs1, imm	"d,s,j"			
	xori	rd, rs1, imm	"d,s,j"			
	ori	rd, rs1, imm	"d,s,j"			
	andi	rd, rs1, imm	"d,s,j"			
	slli	rd, rs1, imm	"d,s,>"			
	srli	rd, rs1, imm	"d,s,>"			
R	srai	rd, rs1, imm	"d,s,>"			
	add	rd, rs1, rs2	"d,s,t"			
	sub	rd, rs1, rs2	"d,s,t"			
	sll	rd, rs1, rs2	"d,s,t"			
	slt	rd, rs1, rs2	"d,s,t"			
	sltu	rd, rs1, rs2	"d,s,t"			
	xor	rd, rs1, rs2	"d,s,t"			
	srl	rd, rs1, rs2	"d,s,t"			
	sra	rd, rs1, rs2	"d,s,t"			
	or	rd, rs1, rs2	"d,s,t"			
	and	rd, rs1, rs2	"d,s,t"			

The above table shows the struct entry for the “operands” field of each of the RV32I base instructions (obtained from the “riscv-opc.c” file). By comparing the operand symbols with the instruction’s actual operands and instruction format, we can infer their meaning.

Note: The file that parses through these symbols seems to be located here:

`riscv-gnu-toolchain/binutils/gas/config/tc-riscv.c`

8) Building The Toolchain

```
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain$ mkdir build
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain$ ls
binutils  build-gdb-newlib  config.status  configure.ac  dejagnu  gdb  LICENSE  Makefile  musl  pk  README.md  scripts  test
build     config.log        configure      contrib      gcc      glibc  linux-headers  Makefile.in  newlib  qemu  regression  spike
```

Before building the toolchain, we need to create a destination directory where the output files will be stored. For my build, I chose to create the following destination:

riscv-gnu-toolchain/build

8) Building The Toolchain

```
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain$ ./configure --prefix=/home/student/Documents/riscv-gnu-toolchain/build --with-arch=rv32gc --with-abi=ilp32d
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for grep that handles long lines and -e... /bin/grep
checking for fgrep... /bin/grep -F
checking for grep that handles long lines and -e... (cached) /bin/grep
checking for bash... /bin/bash
checking for __gmpz_init in -lgmp... yes
checking for mpfr_init in -lmpfr... yes
checking for mpc_init2 in -lmpc... yes
checking for curl... /usr/bin/curl
checking for wget... /usr/bin/wget
checking for ftp... /usr/bin/ftp
configure: creating ./config.status
config.status: creating Makefile
config.status: creating scripts/wrapper/awk/awk
config.status: creating scripts/wrapper/sed/sed
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain$
```

Navigate to the **riscv-gnu-toolchain** directory and run the following command:

```
./configure --prefix=<build_directory> --with-arch=rv32i --with-abi=ilp32
```

Replace **<build_directory>** with the absolute path to the build directory you chose.

The **--with-arch** and **--with-abi** flags have default values that the toolchain will build to if not specified. For this build, we only want the toolchain to build for the RV32I ISA, so we pass the flags pictured above. For different architectures, these flags will need to be modified.

The [RISC-V Toolchain Repository](#) explains these flags further.

8) Building The Toolchain

```
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain$ make -j4
```

Run the make command to start building the toolchain:

```
make -j<# of threads>
```

If successful, the toolchain build will take 30-60 minutes depending on how many threads you specified.

Note: Some online guides say to run `make linux`. If building the toolchain for only the RV32I ISA, this will cause the build to fail. This is likely due to the toolchain trying to build files for linux applications that explicitly require additional ISAs beyond just RV32I.







8) Building The Toolchain

```
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain$ cd build
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain/build$ ls
bin include lib libexec riscv32-unknown-elf share
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain/build$ cd bin
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain/build/bin$ ls
riscv32-unknown-elf-addr2line  riscv32-unknown-elf-elfedit  riscv32-unknown-elf-gcc-ranlib  riscv32-unknown-elf-gprof  riscv32-unknown-elf-objdump  riscv32-unknown-elf-strip
riscv32-unknown-elf-ar        riscv32-unknown-elf-g++      riscv32-unknown-elf-gcov       riscv32-unknown-elf-ld      riscv32-unknown-elf-ranlib
riscv32-unknown-elf-as        riscv32-unknown-elf-gcc      riscv32-unknown-elf-gcov-dump  riscv32-unknown-elf-ld.bfd  riscv32-unknown-elf-readelf
riscv32-unknown-elf-c++       riscv32-unknown-elf-gcc-12.2.0  riscv32-unknown-elf-gcov-tool  riscv32-unknown-elf-lto-dump  riscv32-unknown-elf-run
riscv32-unknown-elf-c++filt   riscv32-unknown-elf-gcc-ar    riscv32-unknown-elf-gdb       riscv32-unknown-elf-nm      riscv32-unknown-elf-size
riscv32-unknown-elf-cpp       riscv32-unknown-elf-gcc-nm    riscv32-unknown-elf-gdb-add-index  riscv32-unknown-elf-objcopy  riscv32-unknown-elf-strings
student@student-VirtualBox:~/Documents/riscv-gnu-toolchain/build/bin$
```

After the toolchain is built successfully, we can navigate into the build directory and see what was built. The **bin** directory contains the executable files we need.

Note: Now that the toolchain is built, remember to decrease the virtual machine's base memory back to its original value.

9) Compiling A Test Program

 geoneill12 Add files via upload		cb89627 1 minute ago	 3 commits
 src	Add files via upload		1 minute ago
 Makefile	Add files via upload		1 minute ago
 link.ld	Add files via upload		1 minute ago
 readme.md	Add files via upload		1 minute ago

Fetch the files from the following repository:

[otter-tools-p1](#)

The repository contains a makefile, linkerscript, and a test program for the RISC-V target device I am using to demo the new instruction.

9) Compiling A Test Program

```
16  # settings for the compilers
17  ifneq ($(C_FILES),)
18  RISCV_PREFIX = <exe_path>riscv32-unknown-elf-
19  else
20  RISCV_PREFIX = <exe_path>riscv32-unknown-elf-
21  endif
```

If you happen to have multiple versions of the RISC-V toolchain installed on your system, you'll need to specify which version to use when trying to compile programs with your custom instruction. An easy way to do this is by specifying the absolute path to the modified toolchain in a makefile.

Open the makefile. On lines 18 and 20, replace `<exe_path>` with the absolute path to the toolchain executable that supports your custom instruction. For this build, the absolute path is:

`/home/student/Documents/riscv-gnu-toolchain/build/bin/`

Thus, the RISCV_PREFIX values become:

```
16  # settings for the compilers
17  ifneq ($(C_FILES),)
18  RISCV_PREFIX = /home/student/Documents/riscv-gnu-toolchain/build/bin/riscv32-unknown-elf-
19  else
20  RISCV_PREFIX = /home/student/Documents/riscv-gnu-toolchain/build/bin/riscv32-unknown-elf-
21  endif
```

9) Compiling A Test Program

```
21  loop:    nop                # do nothing (easier to see in simulator)
22          otter x0,x0,x0      ←
23          beq  x8,x0,loop     # wait for interrupt
24
25          xori x20,x20,1      # toggle current LED value
26          sw   x20,0(x15)     # output LED value
27
28          mv   x8,x0          # clear flag
29          csrrw x0,mie,x10    # enable interrupt
30          j    loop          # return to loopville
```

Open “main.s” in the **src** directory and add the custom instruction somewhere and specify the operands. For this demo, placement doesn’t matter.

Note: Depending on what text editor you use, syntax highlighting probably won’t highlight your new instruction since it wouldn’t recognize it as an instruction.

9) Compiling A Test Program

```
student@student-VirtualBox:~/Documents/otter-tools-p1$ make
mkdir -p build
/home/student/Documents/riscv-gnu-toolchain/build/bin/riscv32-unknown-elf-gcc -c -o build/init.o src/init.s -O0 -march=rv32i -mabi=ilp32
/home/student/Documents/riscv-gnu-toolchain/build/bin/riscv32-unknown-elf-gcc -c -o build/main.o src/main.s -O0 -march=rv32i -mabi=ilp32
/home/student/Documents/riscv-gnu-toolchain/build/bin/riscv32-unknown-elf-gcc -o build/program.elf build/init.o build/main.o -T link.ld -mno-relax -nostdlib -nostartfiles -mmodel=medany -O0 -march=rv32i
-mabi=ilp32
/home/student/Documents/riscv-gnu-toolchain/build/lib/gcc/riscv32-unknown-elf/12.2.0/../../../../riscv32-unknown-elf/bin/ld: warning: section '.data' type changed to PROGBITS
/home/student/Documents/riscv-gnu-toolchain/build/lib/gcc/riscv32-unknown-elf/12.2.0/../../../../riscv32-unknown-elf/bin/ld: warning: build/program.elf has a LOAD segment with Rwx permissions
/home/student/Documents/riscv-gnu-toolchain/build/bin/riscv32-unknown-elf-objcopy -O binary --only-section=.data* --only-section=.text* build/program.elf build/mem.bin
hexdump -v -e "%08x\n" build/mem.bin > build/mem.txt
/home/student/Documents/riscv-gnu-toolchain/build/bin/riscv32-unknown-elf-objdump -S -s build/program.elf > build/program.dump
student@student-VirtualBox:~/Documents/otter-tools-p1$
```

Navigate to the **otter-tools-p1** directory and run **make**.

9) Compiling A Test Program

```
student@student-VirtualBox:~/Documents/otter-tools-p1$ ls
build link.ld Makefile readme.md src
student@student-VirtualBox:~/Documents/otter-tools-p1$ cd build
student@student-VirtualBox:~/Documents/otter-tools-p1/build$ ls
init.o main.o mem.bin mem.txt program.dump program.elf
student@student-VirtualBox:~/Documents/otter-tools-p1/build$
```

A **build** directory will be created with several output files.

9) Compiling A Test Program

```
41 00000038 <loop>:  
42 38: 00000013      nop  
43 3c: 00006027      otter zero,zero,zero  
44 40: fe040ce3      beqz s0,38 <loop>  
45 44: 001a4a13      xor s4,s4,1  
46 48: 0147a023      sw s4,0(a5)  
47 4c: 00000413      li s0,0  
48 50: 30451073      csrw mie,a0  
49 54: fe5ff06f      j 38 <loop>
```



Opening the “program.dump” file shows the disassembly of our compiled program with our custom instruction!