Vivek Follow
Jan 15 · 7 min read · Listen

Adding custom instructions compilation support, to RISCV toolchain.

RISCV ISA provides a robust set of instructions to fulfill most of the computing needs. But in some cases, The hardware designers may need to modify the core to carry out some additional operations.
And this ends up in the addition of new instructions in the existing ISA.

In such cases, the RISC-V toolchain is needed to be modified to provide compilation support for newly added instructions. In the next few sections, we'll see how to do that.

In short, the process involves modifying gnu assembler(gas), such that the cross compiler generated from the toolchain
is able to recognize the newly added instructions.

Let's assume we add two new instructions

1. gcd rd,rs1,rs2
Which computes the gcd of integers stored in source registers rs1 and rs2.
And store the result in destination register rd

2. fact rd, <immediate>

Which divides the value of immediate by 2 and then computes the factorial of result, and store it in memory address pointed
by the address specified in rd. (It can be assumed that only even values are allowed in

Both of these custom instructions are for illustration purposes only]

Brief Encoding Overview:

Let's assume that the target ISA is for 32 bits (non-compressed).
So each instruction is 32 bits long.
The encoding rules specify where the identifiers of source and destination registers, Immediate values, and opcode are placed in those 32 bits.

Just to revise there are 32 general-purpose registers in rv32 ISA,
So log32 = 5 bits are required to uniquely identify each register.
Also, the rv32 ISA supports 12 and 20-bit immediate values.

1. Every 32-bit instruction should have the first two LSBs set to 1
   That means last two bits of every 32 bit instruction are 11; Bits[0,1] = 11

2. Bits [2–4], tell whether the instruction is long instruction (64 bit).
   So for 32-bit instructions Bits[2–4] should never be 111.

3. The gcd instruction is an R- type instruction,
   For R- type instruction
   The destination register is defined in bits from, bit 7 to bit 11, Bits[7,11]
   The first source register is defined in bits from, bit 15 to bit 19, Bits[15,19]
   second source register is defined in bits from, bit 20 to bit 24, Bits[20,24]
   Bits[0,1] = 11 (to specify that instruction is a 32 bit instruction)
   The rest of the Bits are used to store the opcode, such that Bits[2,4] should NOT be 111

4. The fact instruction is a J-type instruction.
   For J-type instruction
   The destination register is defined in bits from, bit 7 to bit 11, Bits[7,11]
   Bit 12 to Bit 31 are used to store the 20-bit immediate value.
   The rest of the Bits are used to store the opcode, such that Bits[2,4] should NOT be 111

And then generating suitable control signals to activate the circuits required to execute the instruction and store the results.]

Toolchain changes:

In order to add two new instructions the first step is to modify riscv-gnu-toolchain/riscv-binutils/opcodes/riscv-opc.c file.
This file contains all opcode or instruction definitions in an array riscv_opcodes, which is an array of riscv_opcode structure.

```
const struct riscv_opcode riscv_opcodes[]
```

For adding a new instruction we need to add an entry in this array.

Before that let's take a look at riscv_opcode structure

```
struct riscv_opcode
{
 /* The name of the instruction. */
 const char *name;
 /* The requirement of xlen for the instruction, 0 if no requirement.
*/
 unsigned xlen_requirement;
 /* Class to which this instruction belongs. Used to decide whether or
 not this instruction is legal in the current -march context. */
 enum riscv_insn_class insn_class;
 /* A string describing the arguments for this instruction. */
 const char *args;
 /* The basic opcode for the instruction. When assembling, this
 opcode is modified by the arguments to produce the actual opcode
 that is used. If pinfo is INSN_MACRO, then this is 0. */
 insn_t match;
 /* If pinfo is not INSN_MACRO, then this is a bit mask for the
 relevant portions of the opcode when disassembling. If the
 actual opcode anded with the match field equals the opcode field,
 then we have found the correct instruction. If pinfo is
 INSN_MACRO, then this field is the macro identifier. */
```

```
  of bits describing the instruction, notably any relevant hazard
  information. */
  unsigned long pinfo;
};
```

So the entries added to riscv_opcode arrays for the new instructions will be as follows

```
{"gcd", 0,  INSN_CLASS_I,"d,s,t",MATCH_GCD,MASK_GCD,match_opcode,0}

{"fact",0,  INSN_CLASS_I,"d,a",MATCH_FACT,MASK_FACT,match_opcode,0}
```

A brief description of each value is given below.

1. name will be "gcd" and "fact" for the gcd and factorial instructions respectively.
2. xlen can have values 0,32 or 64.
   It is used to specify whether the instruction is targeted for only 32 or 64-bit RISCV variants. It seems if this value is set to 32 the instruction will work with only 32-bit variant, And if the value is 64 the instruction will work only on 64-bit version. And if the xlen value is set to 0 then the instruction will work on both 32 and 64-bit variants.
   For both instructions, this value will be 0
3. insn_class: Described the class of instruction, whether it is an integer, atomic, compressed.
4. *args: Is string to specify the operands/register involved in the instruction.
   For gcd instruction args = "d,s,t"
   for fact instruction args = "d,a"
   "d" is for destination
   "s" is for source register 1
   "t" is for source register 2
   "a" is for 20 bit immediate
   How these characters will be utilized will become more clear in the Validation section.

6. mask: Mask is used to identify the position of operand bits in the instruction, ith bit in Mask is 1 if it is not used as an operand in the instruction, Otherwise it is 0.

7. The instruction opcode specification struct also requires a pointer to function, which will be used to detect if any instruction matches with given instruction. The function is given below.

```
static int
 match_opcode (const struct riscv_opcode *op, insn_t insn)
 {
 return ((insn ^ op->match) & op->mask) == 0;
 }
 [from riscv-binutils/opcodes/riscv-opc.c]
```

8. pinfo is used to describe the instruction by binary codes. Like there are codes for conditional, jump type, data movement instructions.
We wont be describing the instruction so this will be 0

In File riscv-gnu-toolchain/riscv-binutils/include/opcode/riscv-opc.h match and mask codes for the instruction are added
Using the rules described above

```
#define MATCH_GCD 0x6027
#define MASK_GCD 0xfe00707f
#define MATCH_FACT 0x27
#define MASK_FACT 0x7f

DECLARE_INSN(gcd, MATCH_GCD,MASK_GCD)
DECLARE_INSN(fact, MATCH_FACT,MASK_FACT)
```

optinally same changes can be done in
riscv-gnu-toolchain/riscv-gdb/opcodes/riscv-opc.c
riscv-gnu-toolchain/riscv-gdb/include/opcode/riscv-opc.h

Instruction defined in the riscv-opc.c are validated during assembler building process. Instruction validation logic can be found in “validate_riscv_insn” function in file riscv-gnu-toolchain/riscv-binutils/gas/config/tc-riscv.c

1. The validation process starts with detecting the size of the instruction
be checking the last two bits defined in opc->match

2. Based on the size of bits total number of required bits is computed
For these instruction it will be 2^{31} -1 or 0xFFFFFFFF;

3. Variable used_bits is set to the initial mask code of the instruction
Defined in opc->mask. So used bits initially have all the operand bits
set to 0 and all then non-operand bits set to 1.

4. After instruction size validation is done.
The args string (opc->args, which is stored in variable p), is parsed.
Every character in args string corresponds to an operand.
For each character and shift and mask value is defined and the bits in the Variable “used_bits” are set for the given mask and shift value as follows

```
used_bits |= ((insn_t)(mask) << (shift))

Example if the instruction uses a destination register, then its args
string will have character “d”, for “d”, mask is 1xf and shift is 7.
So this will set bits 7 to 11 of used bits.
```

5. Step 4 is repeated for all the characters in the args string.
6. If the final results have all the bits in used_bits equal to 1, i.e
used_bits = required_bits
Then the instruction definition is considered valid.

After making the toolchain changes specified above, the toolchain can be rebuilt to support newly added instructions

```
#include <stdint.h>
#define FACT_DIGITS 10000
int main(void)
{
uint32_t num1 = 2321, num2 = 1771731, gcd = 0;
uint32_t fact_test_val = 10;
uint32_t fact_result_ptr;
uint8_t fact_result[FACT_DIGITS];
fact_result_ptr = (uint32_t)fact_result;
asm volatile("gcd %0, %1,%2\n":"=r"(gcd):"r"(num1),"r"(num2):);
//suppose we want to compute the factorial of 125 so immediate=250
asm volatile("fact %0, %1\n":"=r"(fact_result_ptr):"i"(250):);
return 0;
}
```

Generated assembly code from the output elf file

```
0001013c <main>:
   1013c: 8c010113          addi sp,sp,-1856
   10140: 72812e23          sw s0,1852(sp)
   10144: 74010413          addi s0,sp,1856
   10148: ffffe2b7          lui t0,0xffffe
   1014c: 00510133          add sp,sp,t0
   10150: 000017b7          lui a5,0x1
   10154: 91178793          addi a5,a5,-1775 # 911 <register_fini-
0xf763>
   10158: fef42623          sw a5,-20(s0)
   1015c: 001b17b7          lui a5,0x1b1
   10160: 8d378793          addi a5,a5,-1837 # 1b08d3
<__global_pointer$+0x19eafb>
   10164: fef42423          s[illegible])
   10168: fe042223          sw zero,-28(s0)
   1016c: 00a00793          li a5,10
   10170: fef42023          sw a5,-32(s0)
   10174: ffffe7b7          lui a5,0xffffe
   10178: 8dc78793          addi a5,a5,-1828 # ffffd8dc
<__global_pointer$+0xfffebb04>
   1017c: ff078793          addi a5,a5,-16
   10180: 008787b3          add a5,a5,s0
   10184: fcf42e23          sw a5,-36(s0)
   10188: fec42783          lw a5,-20(s0)
   1018c: fe842703          lw a4,-24(s0)
```

```
101a4: 00078513          mv a0,a5
101a8: 000022b7          lui t0,0x2
101ac: 00510133          add sp,sp,t0
101b0: 73c12403          lw s0,1852(sp)
101b4: 74010113          addi sp,sp,1856
101b8: 00008067          ret
```

Instructions at address 0x10190 and 0x10198 contains newly added custom instructions