Lab – RISC-V Instruction Set Architecture

Introduction

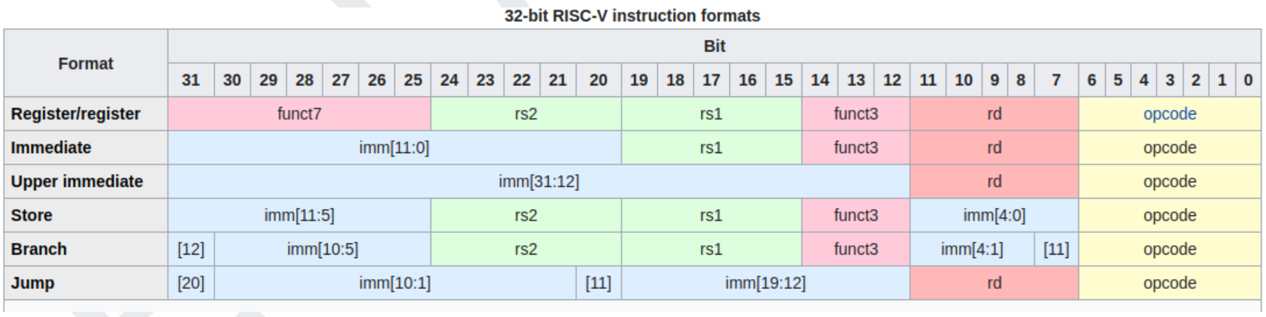
Processor design requires expertise in digital systems design, operating systems and compiler design. For this purpose different commercial vendors like ARM and MIPS are available which charge royalties for their design. RISC-V started with a goal of practical open source ISA that was deployable without any royalties. RISC-V is a Reduced Instruction Set Computer (RISC) based ISA.

RISC-V ISA

RISC-V consists of a base ISA along with optional extensions, which can be added by a user, based upon design requirements. The base ISA defines the instructions and their encodings along with the number of registers and size of those registers and it also specifies the memory and memory addressing requirements for the ISA. Examples of the base ISA include RV32I (32-bit Base Integer Instruction Set), RV32E (32-bit Base Integer Instruction Set Embedded) and RV64I (64-bit Base Integer Instruction Set). The optional extensions are specified to work with the standard base ISAs and they can work with each other without any conflicts. Some examples of RISC-V ISA include the

RISC-V (RV) Instruction Types and Formats

RISC-V ISA RV 32 will be taken into account for this session and it includes 6 different types of instructions each having different encoding. These instructions include the R, I, U, S, B and J type instructions. Each instruction type has its own encoding format which will be required in the design of the control unit of the processor core. Figure 3.1 shows the encoding of different RISC-V instructions.



32-bit RISC-V instruction formats

Format	Bit																															
	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
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

Figure 3.1. RV 32I instruction encoding formats.[3.1]

RISC-V Assembly Programming

We are going to implement a subset of RISC-V instructions in this session. For this purpose we are going to implement a subset of R, I, B, S and J type instructions. RV 32I has different assembly instructions each belonging to a different instruction format. For example, the sub instruction is a R type instruction. Similarly load (lw), store (sw), branch (beq, blt) and jump (j) instructions belong to I, S, B and J type instructions respectively. These assembly instructions will be compiled onto the machine code for the instruction memory by using the riscv-gnu toolchain in the next section.

Instruction Encoding Example

Here we are demonstrating an example of an instruction using the subtract (sub) instruction which is a R-type instruction. We take `sub x9, x9, x8` as our example assembly code since it's an R-type instruction with register x9 as the destination register and register x9 and x8 as the two source registers. So the assembly code is going to subtract the value stored in register x8 from the value stored in register x9 and then it is going to save the result in register x9. Table below illustrates the example in which the values of func7, func3 and opcode for each instruction have been defined in Chapter 24 of the RISC-V specification Volume 1.

	Instruction [31:0]					
	31:25	**24:20**	**19:15**	**14:12**	**11:7**	**6:0**
R-Type	funct7	rs2	rs1	funct3	rd	opcode
sub x9,x9,x8	0100000	x8	x9	000	x9	0110011
sub x9,x9,x8	0100000	01000	01001	000	01001	0110011

Table 3.1. Subtract Instruction Encoding Example.

So the encoding of the instruction results in 32'b01000000100001001000010010110011 which is equivalent to 32'h408484B3. In this way we can perform manual encoding of our instructions.

Writing an Assembly Program

RV 32I contains 32 registers which are labeled from x0 to x31 with the x0 register hard wired to 0 and we can use the rest of the registers in our assembly program to perform the task. Listing 3.1 shares the assembly program for calculating the gcd of two numbers. The numbers are available in register x8 and x9 and at the end the result will be saved in data memory and will also be available in x10.

```
// gcd.s
// registers x8 and x9 have been pre-initialized by random values
gcd:
    beq x8, x9, stop
    blt x8, x9, less
    sub x8, x8, x9
    j gcd
less:
    sub x9, x9, x8
    j gcd
stop:
    sw x8,8(x0)
    lw x10,8(x0)
end:
    j end
```

Listing 3.1. GCD code in RISC-V assembly.

Assembly Program to Machine Code conversion

Run the instructions given in Listing 3.2 in the bash terminal to convert the assembly code into machine code and also create and observe the dumpfile. The code will first create a gcd.o file from gcd.s and then the gcd.elf file will be created after which we create a gcd.bin file and by using the python script provided in lab we can created gcd.txt file which will contain all the machine codes required by the user. We also convert the gcd.elf file to a gcd.dump file which can be read by the user for debugging the assembly code.

```
riscv64-unknown-elf-as -c -o build/gcd.o src/gcd.s -march=rv32i -mabi=ilp32
riscv64-unknown-elf-gcc -o build/gcd.elf build/gcd.o -T linker.ld -nostdlib -march=rv32i
-mabi=ilp32
riscv64-unknown-elf-objcopy -O binary --only-section=.data* --only-section=.text*
build/gcd.elf build/gcd.bin
python3 maketxt.py build/gcd.bin > build/gcd.txt
riscv64-unknown-elf-objdump -S -s build/gcd.elf > build/gcd.dump
```

Listing 3.2. Assembly to machine code conversion.

A makefile has been provided with the example code and instead of using Listing 3.2 use can open the terminal in the folder containing *Makefile*. And in the terminal run the following command.

```
make all #only for Linux
```

Listing 3.3. Makefile to compile assembly code.

The last instruction generates the dumpfile from the elf file and Listing 3.3 illustrates a snippet of a portion of the dumpfile.

```
00000010 <less>:
  10:   408484b3                sub     s1,s1,s0
  14:   fedff06f                j       0 <gcd>
```

Listing 3.4. Dumpfile of the gcd code.

We can observe that the subtract instruction encoded to hexadecimal is of the same value as the one demonstrated in Table 3.1.

C Program to Machine Code conversion

An example C program has also been provided in the shared folder. For compilation of C code we change the makefile and change the values of the SRCs variable to the following value.

```
riscv64-unknown-elf-as -c -o build/startup.o src/startup.s -march=rv32i -mabi=ilp32
riscv64-unknown-elf-gcc -c -o build/main.o src/main.c -march=rv32i -mabi=ilp32
riscv64-unknown-elf-gcc -o build/main.elf build/main.o build/startup.o -T linker.ld -nostdlib
-march=rv32i -mabi=ilp32
riscv64-unknown-elf-objcopy -O binary --only-section=.data* --only-section=.text*
build/main.elf build/main.bin
python3 maketxt.py build/main.bin > build/main.txt
riscv64-unknown-elf-objdump -S -s build/main.elf > build/main.dump
```

Listing 3.5. Changes in Makefile for running C code.

The makefile will compile the startup file first then it compiles the main C program. After compilation of the C code the elf file will be created and the rest of the process will be similar to the one explained for assembly to machine code conversion. We create the machine code by making proposed changes inside the makefile and then by running the command mentioned in Listing 3.3

C Code

```c
int main(void) {
   // declare some variables
   int  x = 32, y = 12, gcd = 0;

   // Loop for GCD evaluation
   while(x != y)
   {
       if(x > y)
           x = x - y;
       else
           y = y - x;
   }

   gcd = x;

   // endless loop
   while(1){}
}
```

Assembly Converted

```
00000000 <main>:
  0:    fe010113     addi    sp,sp,-32
  4:    00812e23     sw      s0,28(sp)
  8:    02010413     addi    s0,sp,32
  c:    02000793     li      a5,32
 10:    fef42623     sw      a5,-20(s0)
 14:    00c00793     li      a5,12
 18:    fef42423     sw      a5,-24(s0)
 1c:    fe042223     sw      zero,-28(s0)
 20:    0340006f     j       54 <main+0x54>
 24:    fec42703     lw      a4,-20(s0)
 28:    fe842783     lw      a5,-24(s0)
 2c:    00e7dc63     bge     a5,a4,44 <main+0x44>
 30:    fec42703     lw      a4,-20(s0)
 34:    fe842783     lw      a5,-24(s0)
 38:    40f707b3     sub     a5,a4,a5
 3c:    fef42623     sw      a5,-20(s0)
 40:    0140006f     j       54 <main+0x54>
 44:    fe842703     lw      a4,-24(s0)
 48:    fec42783     lw      a5,-20(s0)
 4c:    40f707b3     sub     a5,a4,a5
 50:    fef42423     sw      a5,-24(s0)
 54:    fec42703     lw      a4,-20(s0)
 58:    fe842783     lw      a5,-24(s0)
 5c:    fcf714e3     bne     a4,a5,24 <main+0x24>
 60:    fec42783     lw      a5,-20(s0)
 64:    fef42223     sw      a5,-28(s0)
 68:    0000006f     j       68 <main+0x68>
 6c:    00000013     nop
 70:    0080006f     j       78 <reset_handler>
 74:    00000013     nop
```

Listing 3.6. Result Generated from compiling C code.

Windows Tool Setup Link

https://static.dev.sifive.com/dev-tools/freedom-tools/v2020.12/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-w64-mingw32.zip

Tasks

- Write an assembly program that calculates the factorial of a number. Create the dumpfile and machine code of the assembly program using the RISC-V toolchain.

Appendix

Installing RISC-V GCC Toolchain

Windows

Windows users can download the zip file from the internet using the following link.

https://static.dev.sifive.com/dev-tools/freedom-tools/v2020.12/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-w64-mingw32.zip

Extract the zip file and then add the path of the bin folder inside the extracted folder to the PATH variable in windows environment variables.

To check whether the tool has been installed or open powershell and run the following command.

```
riscv64-unknown-elf-gcc --help
```

You should get the following output.

```
Usage: riscv64-unknown-elf-gcc.exe [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase.
  --help                   Display this information.
  --target-help            Display target specific command line options.
```

Ubuntu / WSL

Linux users can run the following command to download and set up the toolchain.

```
cd
wget
https://static.dev.sifive.com/dev-tools/freedom-tools/v2020.12/riscv64-unknow
n-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14.tar.gz
tar -xvf
riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14.tar.gz
```

Use the following command to add the bin directory to the `PATH` variable.

```
export
PATH=$PATH:$HOME/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-
ubuntu14/bin
```

The `export` command should be added at the end of the `~/.bashrc` file otherwise you need to run the command every time you open a new shell. Run the following command to check the installation

```
riscv64-unknown-elf-gcc --help
```

You should get the following output.

```
riscv64-unknown-elf-gcc --help
Usage: riscv64-unknown-elf-gcc [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase.
  --help                   Display this information.
  --target-help            Display target specific command line options.
```