Digital Design and Verification Training

Introduction to RISC-V ISA

Agenda

- RISC-V ISA (Instruction Set Architecture)
- RISC-V Assembly Programming
  - Basic assembly syntax
  - Registers and memory addressing
  - Common instructions (arithmetic, logical, control flow, etc.)
- RISC-V Toolchain (riscv64-unknown-elf-gcc)
- Spike Simulator
- Introduction to Spike
- Practical Exercises

RISC-V ISA (Instruction Set Architecture)

- Brief history and development of RISC-V
  - Origin: RISC-V was born in 2010 at the University of California, Berkeley, led by Andrew Waterman, Yunsup Lee, and Krste Asanović.
  - Quickly gained traction in both academia and industry due to its open-source nature.
- Milestones:
  - 2011: First RISC-V ISA manual published
  - 2014: First commercial RISC-V chip produced
  - 2019: RISC-V ISA becomes a ratified standard

RISC-V ISA (Instruction Set Architecture)

- Open-source:
  - Free to use, modify, and implement without licensing fees.
  - Encourages innovation and collaboration in the hardware community.
- Modular:
- Base ISA (RV32I/RV64I) plus optional standard extensions.
  - Allows implementers to choose only necessary features.
  - Reduces complexity and power consumption for specific applications.
- Extensible:
  - Supports custom extensions for specialized applications.
  - Enables domain-specific optimizations (e.g., AI/ML, cryptography).
  - Future-proof: can adapt to new computing paradigms.

RISC-V ISA (Instruction Set Architecture)

- RISC-V Base ISA Variants
  - RV32I (32-bit base integer ISA)
  - RV64I (64-bit base integer ISA)
  - RV128I (128-bit base integer ISA, less common)
- Instruction Formats
  - R-type (register-register)
  - I-type (immediate/load)
  - S-type (store)
  - B-type (branch)
  - U-type (upper immediate)
  - J-type (jump)

RISC-V ISA (Instruction Set Architecture)

- Base Integer Instruction Set (RV32I/RV64I)
  - Computational instructions (ADD, SUB, AND, OR, XOR, etc.)
  - Load/Store instructions (LW, SW, LB, SB, etc.)
  - Control transfer instructions (JAL, JALR, BEQ, BNE, etc.)
  - Environment call and breakpoints (ECALL, EBREAK)
- Standard Extensions
  - M: Integer Multiplication and Division
  - A: Atomic Instructions
  - F: Single-Precision Floating-Point
  - D: Double-Precision Floating-Point
  - C: Compressed Instructions
  - Other extensions (V for vector operations, B for bit manipulation, etc.)

RISC-V Assembly Programming

- Basic assembly syntax
  - Instruction format: opcode rd, rs1, rs2 or opcode rd, imm(rs1)
  - Labels: Used for branch targets and function names, followed by a colon
  - Comments: Start with '#' and continue to the end of the line
  - Directives: Begin with a period (e.g., .text, .data, .globl)
  - Case sensitivity: Instructions and register names are typically lowercase

RISC-V Assembly Programming

- Registers and memory addressing
- 32 integer registers (x0-x31)
  - x0 is hardwired to zero
  - x1-x31 are general-purpose registers
  - Common names (e.g., zero, ra, sp, gp, tp, t0-t6, s0-s11, a0-a7)
- Memory addressing modes:
  - Base + offset: lw t0, 8(sp)
  - PC-relative: Used for branches and jumps

RISC-V Assembly Programming

- Common instructions
  - Arithmetic: ADD, SUB, ADDI, SLLI, SRLI, SRAI
  - Logical: AND, OR, XOR, ANDI, ORI, XORI
  - Load/Store: LW, LH, LB, SW, SH, SB
- Control flow:
  - Branches: BEQ, BNE, BLT, BGE, BLTU, BGEU
  - Jumps: JAL, JALR

RISC-V Assembly Programming

- Pseudoinstructions
  - Simplified versions of other instructions or instruction sequences
- Examples:
  - li rd, imm   (Load Immediate)
  - mv rd, rs   (Move, equivalent to addi rd, rs, 0)
  - j offset   (Jump, equivalent to jal x0, offset)
  - ret   (Return from subroutine, equivalent to jalr x0, 0(x1))

Writing simple programs

```
.global _start



_start:
        li a0, 5 # Load immediate value 5 into a0
        li a1, 7 # Load immediate value 7 into a1
        add a2, a0, a1 # Add a0 and a1, store result in a2
```

Writing simple programs

```
      li t0, 1 # initialize counter
      li t1, 10 # upper limit

loop:
     # do something with t0 here
     addi t0, t0, 1 # increment counter
     ble t0, t1, loop # branch if less than or equal to 10
```

Writing simple programs

```
         li t0, 5
         li t1, 10
        blt t0, t1, less_than
        j greater_or_equal
less_than:
        # code for t0 < t1
        j end

greater_or_equal:
       # code for t0 >= t1
      # continue program
```

Writing simple programs

```
.data
array:  .word 1, 2, 3, 4, 5   # Define an array of 5 integers
size:   .word 5               # Size of the array

.text
.global _start

_start:
    la t0, array    # Load address of array into t0
    lw t1, size     # Load size of array into t1
    li t2, 0        # Initialize sum to 0
    li t3, 0        # Initialize counter to 0

loop:
    lw t4, 0(t0)    # Load array element into t4
    add t2, t2, t4  # Add element to sum
    addi t0, t0, 4  # Move to next element (increment by 4 bytes)
    addi t3, t3, 1  # Increment counter
    blt t3, t1, loop # If counter < size, continue loop

    # Result is in t2
```

Writing simple programs

```
.data
array:  .word 3, 7, 2, 9, 1, 5
size:   .word 6

.text
.global _start

_start:
    la t0, array     # Load address of array
    lw t1, size      # Load size of array
    lw t2, 0(t0)     # Initialize max with first element
    li t3, 1         # Initialize counter to 1
loop:
    addi t0, t0, 4  # Move to next element
    lw t4, 0(t0)     # Load current element
    bge t2, t4, skip # If max >= current, skip update
    mv t2, t4        # Update max
skip:
    addi t3, t3, 1  # Increment counter
    blt t3, t1, loop # If counter < size, continue loop
```

Writing simple programs

```
factorial:
            addi sp, sp, -8 # Allocate stack space
           sw ra, 4(sp) # Save return address
           sw s0, 0(sp) # Save s0

           mv s0, a0 # Save argument in s0
           li t0, 1 # Initialize result to 1
           ble s0, t0, done # If n <= 1, return 1

         addi a0, s0, -1 # n - 1
         jal ra, factorial # Recursive call
        mul a0, a0, s0 # n * factorial(n-1)

done:
        lw s0, 0(sp) # Restore s0
        lw ra, 4(sp) # Restore return address
       addi sp, sp, 8 # Deallocate stack space
        ret # Return
```

Introduction to the toolchain

- The RISC-V GNU Toolchain is a set of development tools for RISC-V architectures. It includes:
  - Compiler: riscv64-unknown-elf-gcc
  - Assembler: riscv64-unknown-elf-as
  - Linker: riscv64-unknown-elf-ld
  - Debugger: riscv64-unknown-elf-gdb
  - Other utilities: objdump, readelf, size, etc.
- The "riscv64-unknown-elf" prefix indicates:
  - riscv64: 64-bit RISC-V architecture
  - unknown: No specific operating system (bare-metal)
  - elf: Executable and Linkable Format

Introduction to the toolchain

- How to install
  - Build it from the official repo
  - Or use the following command

```
sudo apt-get install gcc-riscv64-unknown-elf
```

  - Or copy the prebuilt and give its path to .bashrc

Introduction to the toolchain

Assembling RISC-V code (riscv64-unknown-elf-as)

- The assembler converts assembly code into object files.
  - Basic usage:

```
riscv64-unknown-elf-as -march=rv64g example.s -o example.o
```

- Options:
  - -march=rv64g: Specifies the RISC-V architecture (RV64G in this case)
  - -o: Specifies the output file name

Introduction to the toolchain

- Linking object files (riscv64-unknown-elf-ld)
  - The linker combines object files and resolves symbol references.
  - Basic usage:

```
riscv64-unknown-elf-ld example.o -o example
```

- Options:
  - -o: Specifies the output executable name

Introduction to the toolchain

- Assemble multiple source files: file1.s, file2.s

```
riscv64-unknown-elf-as -march=rv64g file1.s -o file1.o
riscv64-unknown-elf-as -march=rv64g file2.s -o file2.o
```

- Link the object files

```
riscv64-unknown-elf-ld file1.o file2.o -o program
```

- The output is an executable named "program"

Introduction to the toolchain

- GCC can compile C programs directly to RISC-V executables.
- Basic usage:

```
riscv64-unknown-elf-gcc -march=rv64g -mabi=lp64 example.c -o example
```

- Options:
  - -march=rv64g: Specifies the RISC-V architecture
  - -mabi=lp64: Specifies the ABI (Application Binary Interface)
  - -O2: Enables level 2 optimizations (optional)
  - -static: Produces a statically linked executable (useful for bare-metal)

Introduction to the toolchain

- Additional gcc features:
  - Generating assembly from C

```
riscv64-unknown-elf-gcc -S example.c -o example.s
```

  - Compiling without linking

```
riscv64-unknown-elf-gcc -c example.c -o example.o
```

  - Use objdump to inspect the compiled code

```
riscv64-unknown-elf-objdump -d example
```

Introduction to Spike

- Spike is the official RISC-V ISA Simulator developed by the RISC-V Foundation. It's a crucial tool for RISC-V development and education.
- Key features:
  - Supports multiple RISC-V ISA variants (RV32, RV64, various extensions)
  - Provides a functional simulation of RISC-V processors
  - Can be used as a golden model for hardware verification
  - Supports debugging and interactive mode

Introduction to Spike

- Installation: Usually built from source.
- Basic steps:

```
git clone https://github.com/riscv/riscv-isa-sim.git
cd riscv-isa-sim
mkdir build
cd build
../configure --enable-commitlog
make
sudo make install
```

Introduction to Spike

- Running assembled programs on Spike

```
spike example
```

- Debugging with Spike (-d option)

```
spike -d example
```

- Analyzing program behavior and performance

```
spike -l example
```