Chapter 6: Architecture

Floating-Point Instructions

RISC-V Floating-Point Extensions

- RISC-V offers three floating point extensions:
  - RVF: single-precision (32-bit)
    - 8 exponent bits, 23 fraction bits
  - RVD: double-precision (64-bit)
    - 11 exponent bits, 52 fraction bits
  - RVQ: quad-precision (128-bit)
    - 15 exponent bits, 112 fraction bits

Floating-Point Registers

- **32** Floating point registers
- **Width** is highest precision – for example, if RVQ is implemented, registers are 128 bits wide
- When multiple floating point extensions are implemented, the lower-precision values occupy the lower bits of the register

Floating-Point Registers

Name	Register Number	Usage
ft0-7	f0-7	Temporary variables
fs0-1	f8-9	Saved variables
fa0-1	f10-11	Function arguments/Return values
fa2-7	f12-17	Function arguments
fs2-11	f18-27	Saved variables
ft8-11	f28-31	Temporary variables

Floating-Point Instructions

- Append .s (single), .d (double), .q (quad) for precision. I.e., `fadd.s`, `fadd.d`, and `fadd.q`
- **Arithmetic operations**:
  `fadd`, `fsub`, `fdiv`, `fsqrt`, `fmin`, `fmax`, multiply-add (`fmadd`, `fmsub`, `fnmadd`, `fnmsub`)
- **Other instructions:**
  move (`fmv.x.w`, `fmv.w.x`)
  convert (`fcvt.w.s`, `fcvt.s.w`, etc.)
  comparison (`feq`, `flt`, `fle`)
  classify (`fclass`)
  sign injection (`fsgnj`, `fsgnjn`, `fsgnjx`)

See Appendix B for additional RISC-V floating-point instructions.

Floating-Point Multiply-Add

- `fmadd` is the most critical instruction for signal processing programs.
- Requires four registers.

```
fmadd.f f1, f2, f3, f4     # f1 = f2 x f3 + f4
```

Floating-Point Example

C Code

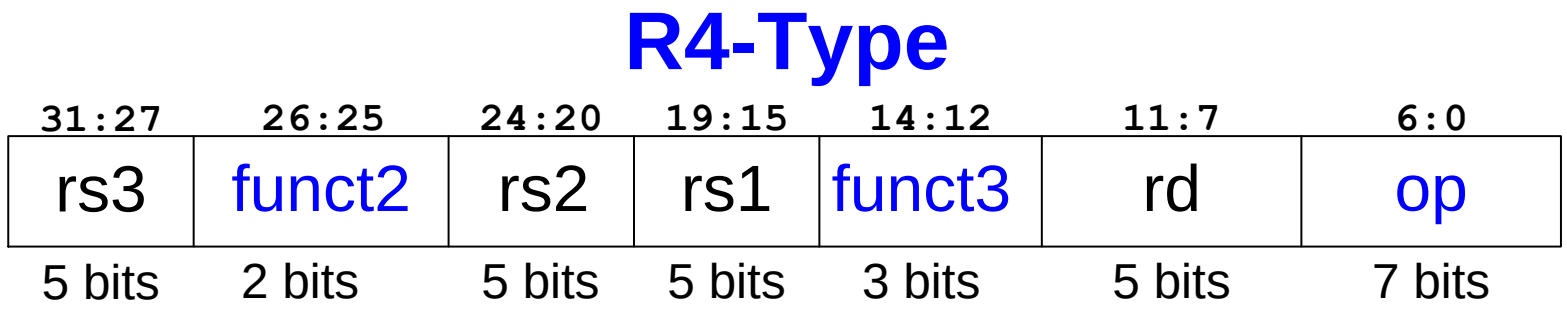
```
int i;
float scores[200];



for (i=0; i<200; i=i+1)



  scores[i]=scores[i]+10;
```

RISC-V assembly code

```
# s0 = scores base address, s1 = i
  addi s1, zero, 0          # i = 0
  addi t2, zero, 200        # t2 = 200
  addi t0, zero, 10         # ft0 = 10.0
  fcvt.s.w ft0,  t0
for:
  bge    s1, t2, done       # i>=200? done
  slli   t0, s1, 2          # t0 = i*4
  add    t0, t0, s0         # scores[i] address
  flw    ft1, 0(t0)         # ft1=scores[i]
  fadd.s ft1, ft1, ft0      # ft1=scores[i]+10
  fsw    ft1, 0(t0)         # scores[i] = t1
  addi   s1, s1, 1          # i = i+1
  j      for                # repeat
done:
```

Floating-Point Instruction Formats

- Use R-, I-, and S-type formats
- Introduce another format for multiply-add instructions that have 4 register operands: R4-type



Chapter 6: Architecture

Exceptions

Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g., keyboard
  - Software, also called *traps*, e.g., undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception
  - Jumps to exception handler
  - Returns to the program

Exception Causes

Exception	Cause
Instruction address misaligned	0
Instruction access fault	1
Illegal instruction	2
Breakpoint	3
Load address misaligned	4
Load access fault	5
Store address misaligned	6
Store access fault	7
Environment call from U-Mode	8
Environment call from S-Mode	9
Environment call from M-Mode	11

RISC-V Privilege Levels

- In RISC-V, exceptions occur at various **privilege levels**.
- Privilege levels limit access to memory or certain (privileged) instructions.
- RISC-V privilege modes are (from highest to lowest):
  - **Machine** mode (bare metal)
  - **System** mode (operating system)
  - **User** mode (user program)
  - **Hypervisor** mode (to support virtual machines)
- For example, a program running in M-mode (machine mode) can access all memory or instructions – it has the highest privilege level.

Exception Registers

- Each privilege level has registers to handle exceptions
- These registers are called control and status registers (CSRRs)
- We discuss M-mode (machine mode) exceptions, but other modes are similar
- M-mode registers used to handle exceptions are:
  - `mtvec`, `mcause`, `mepc`, `mscratch`

(Likewise, S-mode exception registers are: `stvec`, `scause`, `sepc`, and `mscratch`; and so on for the other modes.)

Exception Registers

- CSRRs are not part of register file
- M-mode CSRRs used to handle exceptions
  - **`mtvec`**: holds address of exception handler code
  - **`mcause`**: Records cause of exception
  - **`mepc`** (Exception PC): Records PC where exception occurred
  - **`mscratch`**: scratch space in memory for exception handlers

Exception-Related Instructions

- Called **privileged instructions** (because they access CSRRs)
  - **`csrr`**: CSR register read
  - **`csrw`**: CSR register write
  - **`csrrw`**: CSR register read/write
  - **`mret`**: returns to address held in mepc
- **Examples:**

```
csrr t1, mcause          # t1 = mcause
csrw mepc, t2            # mepc = t2
cwrrw t0, mscratch, t1 # t0 = mscratch
                         # mscratch = t1
```

Exception Handler Summary

- When a processor **detects an exception**:
  - It **jumps to exception handler** address in `mtvec`
  - The exception handler then:
    - **saves registers** on small stack pointed to by `mscratch`
    - Uses `csrr` (CSR read) to **look at cause** of exception (in `mcause`)
    - **Handles exception**
    - When finished, optionally **increments `mepc` by 4** and **restores registers** from memory
    - And then either **aborts** the program or **returns to user code** (using `mret`, which returns to address held in `mepc`)

Example Exception Handler Code

- Check for two types of exceptions:
  - **Illegal instruction** (`mcause` = 2)
  - **Load address misaligned** (`mcause` = 4)

Example Exception Handler Code

```
# save registers that will be overwritten
  csrrw t0, mscratch, t0    # swap t0 and mscratch
  sw    t1, 0(t0)           # [mscratch]   = t1
  sw    t2, 4(t0)           # [mscratch+4] = t2
# check cause of exception
  csrr  t1, mcause          # t1=mcause
  addi  t2, x0, 2           # t2=2 (illegal instruction exception code)
illegalinstr:
  bne   t1, t2, checkother  # branch if not an illegal instruction
  csrr  t2, mepc            # t2=exception PC
  addi  t2, t2, 4           # increment exception PC
  csrw  mepc, t2            # mepc=t2
  j     done                # restore registers and return
checkother:
  addi  t2, x0, 4           # t2=4 (load address misaligned exception code)
  bne   t1, t2, done        # branch if not a misaligned load
  j     exit                # exit program
# restore registers and return from the exception
done:
  lw    t1, 0(t0)           # t1 = [mscratch]
  lw    t2, 4(t0)           # t2 = [mscratch+4]
  csrrw t0, mscratch, t0    # swap t0 and mscratch
  mret                      # return to program
exit:
  ...
```

Checks for two types of exceptions:

- Illegal instruction (`mcause` = 2)
- Load address misaligned (`mcause` = 4)