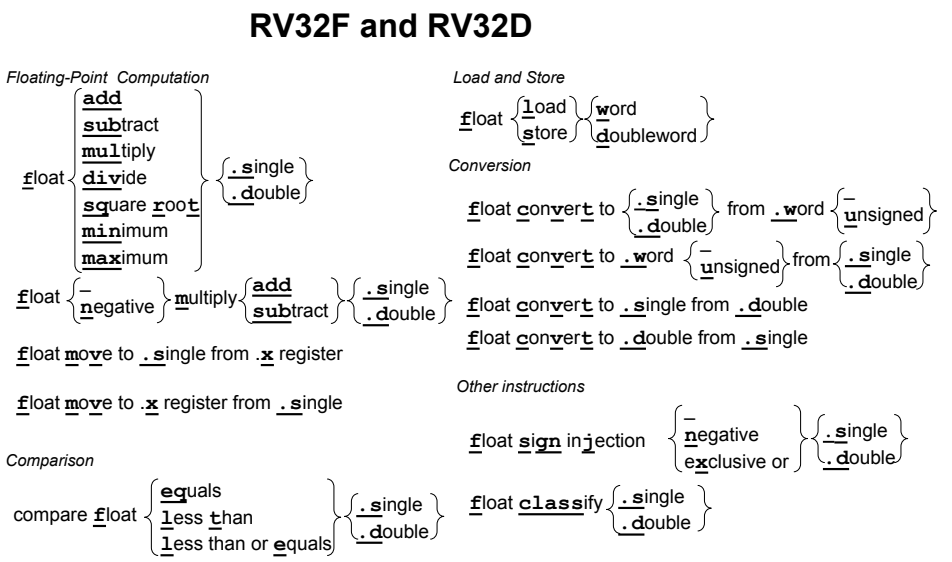


Figure 5.1: Diagram of the RV32F and RV32D instructions.

into the left and right 32-bit halves of a double-precision register. x86-32 floating-point arithmetic didn't have any registers, but used a stack instead. The stack entries were 80-bits wide to improve accuracy, so loads covert 32-bit or 64-bit operands to 80 bits, and vice versa for stores. A subsequent version of x86-32 added 8 traditional 64-bit floating-point registers and associated instructions. Unlike RV32FD and MIPS-32, ARM-32 and x86-32 overlooked instructions to move data directly between floating-point and integer registers. The only solution is to store a floating-point register in memory and then load it from memory to an integer register, and vice versa.

Having only 16 double-precision registers was the most painful ISA error in MIPS according to John Mashey, one of its architects.

■ ***Elaboration: RV32FD allows the rounding mode to be set per instruction.***

Called *static rounding*, it helps performance when you only need to change the rounding mode for one instruction. The default is to use the dynamic rounding mode in the `fcsr`. Static rounding is specified as an optional last argument, as `fadd.s ft0, ft1, ft2, rtz` will round towards zero, irrespective of `fcsr`. The caption of Figure 5.5 lists the names of the rounding modes.

5.3 Floating-Point Loads, Stores, and Arithmetic

RISC-V has two load instructions (`flw`, `fld`) and two store instructions (`fsw`, `fsd`) for RV32F and RV32D. The have the same addressing mode and instruction format as `lw` and `sw`.

Adding to the standard arithmetic operations (`fadd.s, fadd.d, fsub.s, fsub.d, fmul.s, fmul.d, fdiv.s, fdiv.d`), RV32F and RV32D include square root (`fsqrt.s, fsqrt.d`). They also have minimum and maximum (`fmin.s, fmin.d, fmax.s, fmax.d`), which write the smaller or larger values from the pair of source operands without using a branch instruction.

Unlike integer arithmetic, the size of the product from a floating-point multiply is the same as its operands. Also, RV32F and RF32D omit floating-point remainder instructions.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	0000111	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	S fsw
rs3	00	rs2	rs1	rm	rd	1000011	R4 fmadd.s
rs3	00	rs2	rs1	rm	rd	1000111	R4 fmsub.s
rs3	00	rs2	rs1	rm	rd	1001011	R4 fnmsub.s
rs3	00	rs2	rs1	rm	rd	1001111	R4 fnmadd.s
0000000		rs2	rs1	rm	rd	1010011	R fadd.s
0000100		rs2	rs1	rm	rd	1010011	R fsub.s
0001000		rs2	rs1	rm	rd	1010011	R fmul.s
0001100		rs2	rs1	rm	rd	1010011	R fdiv.s
0101100		00000	rs1	rm	rd	1010011	R fsqrt.s
0010000		rs2	rs1	000	rd	1010011	R fsgnj.s
0010000		rs2	rs1	001	rd	1010011	R fsgnjn.s
0010000		rs2	rs1	010	rd	1010011	R fsgnjx.s
0010100		rs2	rs1	000	rd	1010011	R fmin.s
0010100		rs2	rs1	001	rd	1010011	R fmax.s
1100000		00000	rs1	rm	rd	1010011	R fcvt.w.s
1100000		00001	rs1	rm	rd	1010011	R fcvt.wu.s
1110000		00000	rs1	000	rd	1010011	R fmv.x.w
1010000		rs2	rs1	010	rd	1010011	R feq.s
1010000		rs2	rs1	001	rd	1010011	R flt.s
1010000		rs2	rs1	000	rd	1010011	R fle.s
1110000		00000	rs1	001	rd	1010011	R fclass.s
1101000		00000	rs1	rm	rd	1010011	R fcvt.s.w
1101000		00001	rs1	rm	rd	1010011	R fcvt.s.wu
1111000		00000	rs1	000	rd	1010011	R fmv.w.x

Figure 5.2: RV32F opcode map has instruction layout, opcodes, format type, and names. The primary difference in the encodings between this and the next figure is bit 12 is a 0 for the first two instructions and bit 25 is a 0 for the rest of the instructions where both bits are 1 in RV32D. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	0000111	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	S fsd
rs3	01	rs2	rs1	rm	rd	1000011	R4 fmadd.d
rs3	01	rs2	rs1	rm	rd	1000111	R4 fmsub.d
rs3	01	rs2	rs1	rm	rd	1001011	R4 fnmsub.d
rs3	01	rs2	rs1	rm	rd	1001111	R4 fnmadd.d
0000001		rs2	rs1	rm	rd	1010011	R fadd.d
0000101		rs2	rs1	rm	rd	1010011	R fsub.d
0001001		rs2	rs1	rm	rd	1010011	R fmul.d
0001101		rs2	rs1	rm	rd	1010011	R fdiv.d
0101101		00000	rs1	rm	rd	1010011	R fsqrt.d
0010001		rs2	rs1	000	rd	1010011	R fsgnj.d
0010001		rs2	rs1	001	rd	1010011	R fsgnjn.d
0010001		rs2	rs1	010	rd	1010011	R fsgnjx.d
0010101		rs2	rs1	000	rd	1010011	R fmin.d
0010101		rs2	rs1	001	rd	1010011	R fmax.d
0100000		00001	rs1	rm	rd	1010011	R fcvt.s.d
0100001		00000	rs1	rm	rd	1010011	R fcvt.d.s
1010001		rs2	rs1	010	rd	1010011	R feq.d
1010001		rs2	rs1	001	rd	1010011	R flt.d
1010001		rs2	rs1	000	rd	1010011	R fle.d
1110001		00000	rs1	001	rd	1010011	R fclass.d
1100001		00000	rs1	rm	rd	1010011	R fcvt.w.d
1100001		00001	rs1	rm	rd	1010011	R fcvt.wu.d
1101001		00000	rs1	rm	rd	1010011	R fcvt.d.w
1101001		00001	rs1	rm	rd	1010011	R fcvt.d.wu

Figure 5.3: RV32D opcode map has instruction layout, opcodes, format type, and names. There are some instructions in these two figures do not simply differ by data width. This figure uniquely has `fcvt.s.d` and `fcvt.d.s` while the other has `fmv.x.w` and `fmv.w.x`. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.)

63 … 32	31 … 0	
	f0 / ft0	FP Temporary
	f1 / ft1	FP Temporary
	f2 / ft2	FP Temporary
	f3 / ft3	FP Temporary
	f4 / ft4	FP Temporary
	f5 / ft5	FP Temporary
	f6 / ft6	FP Temporary
	f7 / ft7	FP Temporary
	f8 / fs0	FP Saved register
	f9 / fs1	FP Saved register
	f10 / fa0	FP Function argument, return value
	f11 / fa1	FP Function argument, return value
	f12 / fa2	FP Function argument
	f13 / fa3	FP Function argument
	f14 / fa4	FP Function argument
	f15 / fa5	FP Function argument
	f16 / fa6	FP Function argument
	f17 / fa7	FP Function argument
	f18 / fs2	FP Saved register
	f19 / fs3	FP Saved register
	f20 / fs4	FP Saved register
	f21 / fs5	FP Saved register
	f22 / fs6	FP Saved register
	f23 / fs7	FP Saved register
	f24 / fs8	FP Saved register
	f25 / fs9	FP Saved register
	f26 / fs10	FP Saved register
	f27 / fs11	FP Saved register
	f28 / ft8	FP Temporary
	f29 / ft9	FP Temporary
	f30 / ft10	FP Temporary
	f31 / ft11	FP Temporary
32	32	

Figure 5.4: The floating-point registers of RV32F and RV32D. The single-precision registers occupy the rightmost half of the 32 double-precision registers. Chapter 3 explains the RISC-V calling convention for the floating-point registers, the rationale behind the FP Argument registers (fa0-fa7), FP Saved registers (fs0-fs11), and FP Temporaries (ft0-ft11). (Table 20.1 of [Waterman and Asanović 2017] is the basis of this figure.)

31 — 8	7 — 5	4	3	2	1	0
Reserved	Rounding Mode (`frm`)	Accrued Exceptions (`fflags`)				
		NV	DZ	OF	UF	NX
24	3	1	1	1	1	1

Figure 5.5: Floating-point control and status register. It holds the rounding modes and the exception flags. The rounding modes are round to nearest, ties to even (rte, 000 in `frm`); round towards zero (rtz, 001); round down, towards $-\infty$ (rdn, 010); round up, towards $+\infty$ (rup, 011); and round to nearest, ties to max magnitude (rmm, 100). The five accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software: NV is Invalid Operation; DZ is Divide by Zero; OF is Overflow; UF is Underflow; and NX is Inexact. (Figure 8.2 of [Waterman and Asanović 2017] is the basis of this figure.)

Many floating-point algorithms, such as matrix multiply, perform a multiply immediately followed by an addition or a subtraction. Hence, RISC-V offers instructions that multiply two operands and then either add (`fmadd.s, fmadd.d`) or subtract (`fmsub.s, fmsub.d`) a third operand to that product before writing the sum. It also has versions that negate the product before adding or subtracting the third operand: `fnmadd.s, fnmadd.d, fnmsub.s, fnmsub.d`. These *fused* multiply-add instructions are more accurate as well as faster than separate multiply and add instructions, because they round only once (after the add) rather than twice (after the multiply, then after the add). These instructions need a new instruction format to specify 4 registers, called R4. Figures 5.2 and 5.3 show the R4 format, which is a variation of the R format.

Instead of floating-point branch instructions, RV32F and RV32D supply comparison instructions that set an integer register to 1 or 0 based on comparison of two floating-point registers: `feq.s, feq.d, flt.s, flt.d, fle.s, fle.d`. These instructions allow an integer branch instruction to jump based on a floating-point condition. For example, this code branches to `Exit` if `f1 < f2`:

```
flt x5, f1, f2   # x5 = 1 if f1 < f2; otherwise x5 = 0
bne x5, x0, Exit # if x5 != 0, jump to Exit
```

5.4 Floating-Point Converts and Moves

RV32F and RV32D have instructions that perform all combinations of useful conversions between 32-bit signed integers, 32-bit unsigned integers, 32-bit floating point, and 64-bit floating point. Figure 5.6 displays these 10 instructions by source data type and converted destination data type.

RV32F also offers instructions to move data to `x` from `f` registers (`fmv.x.w`) and vice versa (`fmv.w.x`).

5.5 Miscellaneous Floating-Point Instructions

RV32F and RV32D offer unusual instructions that help with math libraries as well as provide useful pseudoinstructions. (The IEEE 754 floating-point standard requires a way to copy and manipulate signs and to classify floating-point data, which inspired these instructions.)

The first is the *sign-injection* instructions, which copy everything from the first source register but the sign bit. The value of the sign bit depends on the instruction:

To	From			
	32b signed integer (`w`)	32b unsigned integer (`wu`)	32b floating point (`s`)	64b floating point (`d`)
32b signed integer (`w`)	–	–	`fcvt.w.s`	`fcvt.w.d`
32b unsigned integer (`wu`)	–	–	`fcvt.wu.s`	`fcvt.wu.d`
32b floating point (`s`)	`fcvt.s.w`	`fcvt.s.wu`	–	`fcvt.s.d`
64b floating point (`d`)	`fcvt.d.w`	`fcvt.d.wu`	`fcvt.d.s`	–

Figure 5.6: RV32F and RV32D conversion instructions. The columns list the source data types and the rows show the converted destination data type.

1. Float sign inject (`fsgnj.s, fsgnj.d`): the result's sign bit is rs2's sign bit.
2. Float sign inject negative (`fsgnjn.s, fsgnjn.d`): the result's sign bit is the opposite of rs2's sign bit.
3. Float sign inject exclusive-or (`fsgnjx.s, fsgnjx.d`): the sign bit is the XOR of the sign bits of rs1 and rs2.

As well as helping with sign manipulation in math libraries, sign-injection instructions provide three popular floating-point pseudoinstructions (see Figure 3.4 on page 37):

1. Copy floating-point register:
   `fmv.s rd,rs` is really `fsgnj.s rd,rs,rs` and
   `fmv.d rd,rs` is really `fsgnj.d rd,rs,rs`.
2. Negate:
   `fneg.s rd,rs` maps to `fsgnjn.s rd,rs,rs` and
   `fneg.d rd,rs` maps to `fsgnjn.d rd,rs,rs`.
3. Absolute value (since $0 \oplus 0 = 0$ and $1 \oplus 1 = 0$):
   `fabs.s rd,rs` becomes `fsgnjx.s rd,rs,rs` and
   `fabs.d rd,rs` becomes `fsgnjx.d rd,rs,rs`.

The second unusual floating-point instruction is classify (`fclass.s, fclass.d`). Classify instructions are also a great aid to math libraries. They test a source operand to see which of 10 floating-point properties apply (see the table below), and then write a mask into the lower 10 bits of the destination integer register with the answer. Only one of the ten bits is set to 1, with the rest set to 0s.

x[rd] bit	Meaning
0	f[*rs1*] is $-\infty$.
1	f[*rs1*] is a negative normal number.
2	f[*rs1*] is a negative subnormal number.
3	f[*rs1*] is -0.
4	f[*rs1*] is $+0$.
5	f[*rs1*] is a positive subnormal number.
6	f[*rs1*] is a positive normal number.
7	f[*rs1*] is $+\infty$.
8	f[*rs1*] is a signaling NaN.
9	f[*rs1*] is a quiet NaN.

```
void daxpy(size_t n, double a, const double x[], double y[])
{
  for (size_t i = 0; i < n; i++) {
    y[i] = a*x[i] + y[i];
  }
}
```

Figure 5.7: The floating-point intensive DAXPY program in C.

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32FD	RV32FD+RV32C
Instructions	10	10	12	12	16	11	11
Per Loop	6	6	7	7	6	7	7
Bytes	40	28	48	32	50	44	28

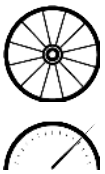
Figure 5.8: Number of instructions and code size of DAXPY for four ISAs. It lists number of instructions per loop and total. Chapter 7 describes ARM Thumb-2, microMIPS, and RV32C.

5.6 Comparing RV32FD, ARM-32, MIPS-32, and x86-32 using DAXPY

The name DAXPY come from the formula itself: Double-precision A times X Plus Y. The single-precision version is called SAXPY.

We'll now do a head-to-head comparison using DAXPY as our floating-point benchmark (Figure 5.7). It calculates $Y = a \times X + Y$ in double-precision, where `X` and `Y` are vectors and `a` is a scalar. Figure 5.8 summarizes the number of instructions and number of bytes in DAXPY of programs for the four ISAs. Their code is in Figures 5.9 to 5.12.

As was the case for Insertion Sort in Chapter 2, despite its emphasis on simplicity, the RISC-V version again has about the same or fewer instructions, and the code sizes of the architectures are quite close. In this example, the compare-and-execute branches of RISC-V save as many instructions as do the fancier address modes and the push and pop instructions of ARM-32 and x86-32.



5.7 Concluding Remarks

Less is More.

—Robert Browning, 1855. The Minimalist school of (building) architecture adopted this poem as an axiom in the 1980s.

The IEEE 754-2008 floating-point standard [IEEE Standards Committee 2008] defines the floating-point data types, the accuracy of computation, and the required operations. Its success greatly reduces the difficulty of porting floating-point programs, and it also means that the floating-point ISAs are probably more uniform than are the equivalent in other chapters.

■ ***Elaboration: 16-bit, 128-bit, and decimal floating-point arithmetic***

The revised IEEE floating-point standard (IEEE 754-2008) describes several new formats beyond single- and double-precision, which they call *binary32* and *binary64*. The least surprising addition is quadruple precision, named *binary128*. RISC-V has a tentative extension planned for it called RV32Q (see Chapter 11). The standard also provided two more sizes for binary data interchange, indicating that programmers might store these numbers in memory or storage but shouldn't expect to be able to compute in these sizes. They are half-precision (*binary16*) and octuple precision (*binary256*). Despite the standard's intent, GPUs do compute in half-precision as well as keep them in memory. The plan for RISC-V is to include half-precision in the vector instructions (RV32V in Chapter 8), with the proviso that processors supporting vector half-precision will also add half-precision scalar instructions. The surprising addition to the revised standard is decimal floating point, for which RISC-V has set aside RV32L (see Chapter 11). The three self-explanatory decimal formats are called *decimal32*, *decimal64*, and *decimal128*.

5.8 To Learn More

IEEE Standards Committee. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

Notes

[1] `http://parlab.eecs.berkeley.edu`

```
  # RV32FD (7 insns in loop; 11 insns/44 bytes total; 28 bytes RVC)
  # a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
    0: 02050463 beqz    a0,28           # if n == 0, jump to Exit
    4: 00351513 slli    a0,a0,0x3       # a0 = n*8
    8: 00a60533 add     a0,a2,a0        # a0 = address of x[n] (last element)
  Loop:
    c: 0005b787 fld     fa5,0(a1)       # fa5 = x[]
   10: 00063707 fld     fa4,0(a2)       # fa4 = y[]
   14: 00860613 addi    a2,a2,8         # a2++ (increment pointer to y)
   18: 00858593 addi    a1,a1,8         # a1++ (increment pointer to x)
   1c: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
   20: fef63c27 fsd     fa5,-8(a2)      # y[i] = a*x[i] + y[i]
   24: fea614e3 bne     a2,a0,c         # if i != n, jump to Loop
  Exit:
   28: 00008067         ret             # return
```

Figure 5.9: RV32D code for DAXPY in Figure 5.7. The address in hexadecimal is on the left, the machine language code in hexadecimal is next, and then the assembly language instruction followed by a comment. The compare-and-branch instructions avoid the two compare instructions in the code of ARM-32 and x86-32.

```
# ARM-32 (6 insns in loop; 10 insns/40 bytes total; 28 bytes Thumb-2)
# r0 is n, d0 is a, r1 is pointer to x[0], r2 is pointer to y[0]
  0: e3500000 cmp      r0, #0             # compare n to 0
  4: 0a000006 beq      24 <daxpy+0x24>    # if n == 0, jump to Exit
  8: e0820180 add      r0, r2, r0, lsl #3 # r0 = address of x[n] (last element)
Loop:
  c: ecb16b02 vldmia   r1!,{d6}           # d6 = x[i], increment pointer to x
 10: ed927b00 vldr     d7,[r2]            # d7 = y[i]
 14: ee067b00 vmla.f64 d7, d6, d0         # d7 = a*x[i] + y[i]
 18: eca27b02 vstmia   r2!, {d7}          # y[i] = a*x[i] + y[i], incr. ptr to y
 1c: e1520000 cmp      r2, r0             # i vs. n
 20: 1afffff9 bne      c <daxpy+0xc>      # if i != n, jump to Loop
Exit:
 24: e12fff1e bx       lr                 # return
```

Figure 5.10: ARM-32 code for DAXPY in Figure 5.7. The autoincrement addressing mode of ARM-32 saves two instructions as compared to RISC-V. Unlike Insertion Sort, there is no need to push and pop registers for DAXPY in ARM-32.

```
# MIPS-32 (7 insns in loop; 12 insns/48 bytes total; 32 bytes microMIPS)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], f12 is a
  0: 10800009 beqz   a0,28 <daxpy+0x28>  # if n == 0, jump to Exit
  4: 000420c0 sll    a0,a0,0x3           # a0 = n*8 (filled branch delay slot)
  8: 00c42021 addu   a0,a2,a0            # a0 = address of x[n] (last element)
Loop:
  c: 24c60008 addiu  a2,a2,8             # a2++ (increment pointer to y)
 10: d4a00000 ldc1   $f0,0(a1)           # f0 = x[i]
 14: 24a50008 addiu  a1,a1,8             # a1++ (increment pointer to x)
 18: d4c2fff8 ldc1   $f2,-8(a2)          # f2 = y[i]
 1c: 4c406021 madd.d $f0,$f2,$f12,$f0    # f0 = a*x[i] + y[i]
 20: 14c4fffa bne    a2,a0,c <daxpy+0xc> # if i != n, jump to Loop
 24: f4c0fff8 sdc1   $f0,-8(a2)          # y[i] = a*x[i] + y[i] (filled delay slot)
Exit:
 28: 03e00008 jr     ra                  # return
 2c: 00000000 nop                        # (unfilled branch delay slot)
```

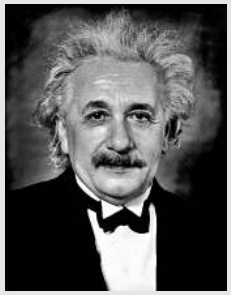
Figure 5.11: MIPS-32 code for DAXPY in Figure 5.7. Two of the three branch delay slots are filled with useful instructions. The ability to check for equality between two registers avoids the two compare instructions found in ARM-32 and x86-32. Unlike integer loads, floating-point loads have no delay slot.

```
# x86-32 (6 insns in loop; 16 insns/50 bytes total)
# eax is i, n is in memory at esp+0x8, a is in memory at esp+0xc
# pointer to x[0] is in memory at esp+0x14
# pointer to y[0] is in memory at esp+0x18
  0: 53                push        ebx                    # save ebx
  1: 8b 4c 24 08       mov         ecx,[esp+0x8]          # ecx has copy of n
  5: c5 fb 10 4c 24 0c vmovsd      xmm1,[esp+0xc]         # xmm1 has a copy of a
  b: 8b 5c 24 14       mov         ebx,[esp+0x14]         # ebx points to x[0]
  f: 8b 54 24 18       mov         edx,[esp+0x18]         # edx points to y[0]
 13: 85 c9             test        ecx,ecx                # compare n to 0
 15: 74 19             je          30 <daxpy+0x30>        # if n==0, jump to Exit
 17: 31 c0             xor         eax,eax                # i = 0 (since x^x==0)
Loop:
 19: c5 fb 10 04 c3    vmovsd      xmm0,[ebx+eax*8]       # xmm0 = x[i]
 1e: c4 e2 f1 a9 04 c2 vfmadd213sd xmm0,xmm1,[edx+eax*8] # xmm0 = a*x[i] + y[i]
 24: c5 fb 11 04 c2    vmovsd      xmm0,xmm1,[edx+eax*8] # y[i] = a*x[i] + y[i]
 29: 83 c0 01          add         eax,0x1                # i++
 2c: 39 c1             cmp         ecx,eax                # compare i vs n
 2e: 75 e9             jne         19 <daxpy+0x19>        # if i!=n, jump to Loop
Exit:
 30: 5b                pop         ebx                    # restore ebx
 31: c3                ret                                # return
```

Figure 5.12: x86-32 code for DAXPY in Figure 5.7. The lack of x86-32 registers is evident in this example, with four variables allocated to memory that are in registers in the code for the other ISAs. It also demonstrates x86-32 idioms to compare a register to zero (`test ecx,ecx`) or to set a register to zero (`xor eax,eax`).

6 RV32A: Atomic Instructions

Albert Einstein (1879-1955) was the most famous scientist of the 20th century. He invented the theory of relativity and advocated building the atomic bomb for World War II.



Everything should be made as simple as possible, but no simpler.

—Albert Einstein, 1933

6.1 Introduction

Our assumption is that you already understand ISA support for multiprocessing, so our job is just to explain the RV32A instructions and what they do. If you don't feel you have sufficient background or need a reminder, study "synchronization (computer science)" on Wikipedia (https://en.wikipedia.org/wiki/Synchronization_(computer_science)) or read Section 2.1 of our related RISC-V architecture book [Patterson and Hennessy 2017].

RV32A has two types of atomic operations for synchronization:

- atomic memory operations (AMO), and
- load reserved / store conditional.

Figure 6.1 is a graphical representation of the RV32A extension instruction set and Figure 6.2 lists their opcodes and instruction formats.

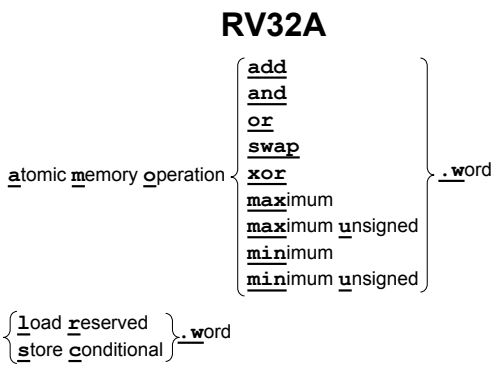


Figure 6.1: Diagram of the RV32A instructions.