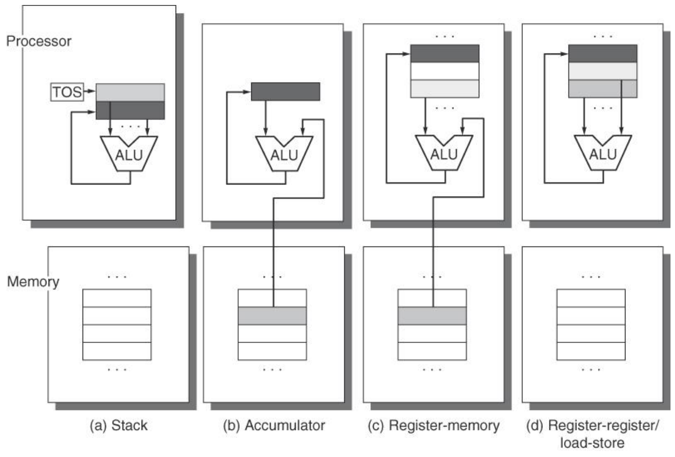
Instruction Set Architecture RISC-V Overview

Instruction Set Architecture

- Instruction Set Architecture
  - The portion of the computer visible to the programmer or compiler writer
  - Serves as the intermediary between the hardware and the software
- What kind of ISAs are there?
  - Where do we store operands? How many should we allow? What are the tradeoffs?
  - What operations should we support? How do we specify where the operand is stored?

Operands for Four ISA Classes



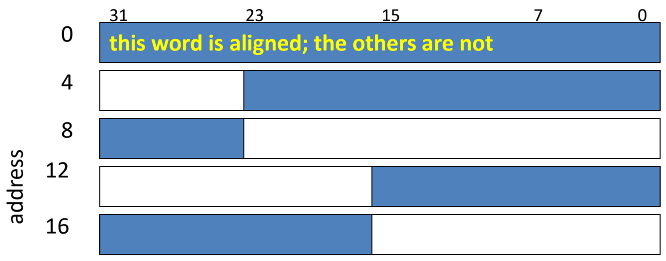
```
********************

         ORG  $4000
A1       =    $3C
A2       =    $3E
A4       =    $42
AUXMOVE  =    $C311

**************************
* SETUP - move data for VTOC
* and catalog to auxmem at
* B000-B3FF (pseudo trk 11
* 0-3)
*************************
SETUP    LDA  #<VTOC
         STA  A1
         LDA  #>VTOC
         STA  A1+1
         LDA  #<END
         STA  A2
         LDA  #>END
         STA  A2+1
         LDA  #$00
         STA  A4
         LDA  #$B0
         STA  A4+1
         SEC
         JMP  AUXMOVE

```

Interpreting Memory Addresses



- Words are aligned (32 bit in this example)
- Big-endian (network byte order)
  - Most significant byte at lower address
- Littlest-endian
  - Least significant byte at lower address
- Bi-endian

Examples of Addressing Modes

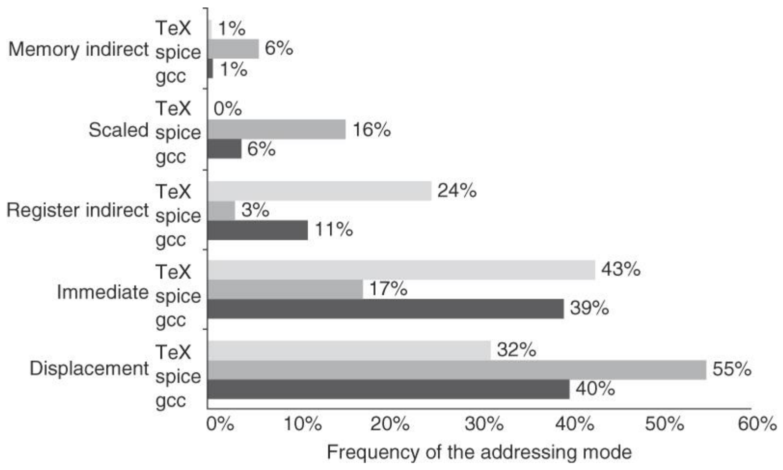
	Addressing Mode	Example	Action
1.	Register direct	Add R4, R3	R4 ← R4 + R3
2.	Immediate	Add R4, #3	R4 ←R4 + 3
3.	Displacement	Add R4, 100(R1)	R4 ← R4 + M[100 + R1]
4.	Register indirect	Add R4, (R1)	R4 ← R4 + M[R1]
5.	Indexed	Add R4, (R1 + R2)	R4 ← R4 + M[R1 + R2]
6.	Direct	Add R4, (1000)	R4 ← R4 + M[1000]
7.	Memory Indirect	Add R4, @(R3)	R4 ← R4 + M[M[R3]]

8.	Autoincrement	Add R4, (R2)+	R4 ← R4 + M[R2]
			R2 ← R2 + elemSize

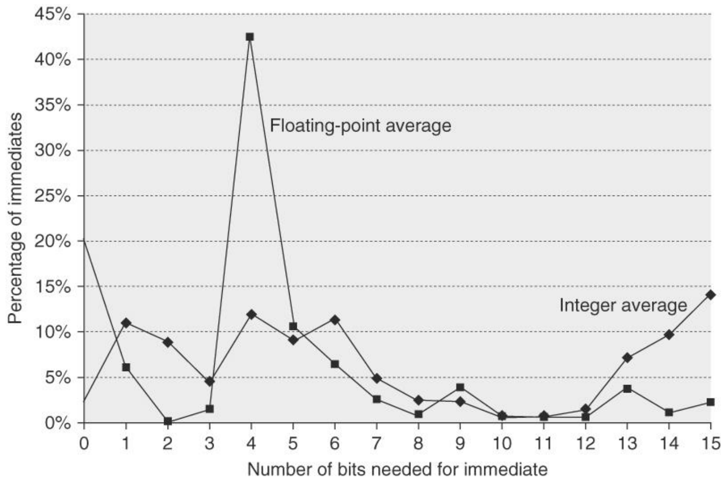
9.	Autodecrement	Add R4, -(R2)	R4 ← R4 + M[R2]
			R2 ← R2 - elemSize

10.	Scaled	Add R4, 100(R2)[R3]	R4 ← R4 + M[100 + R2 + R3*elemSize]

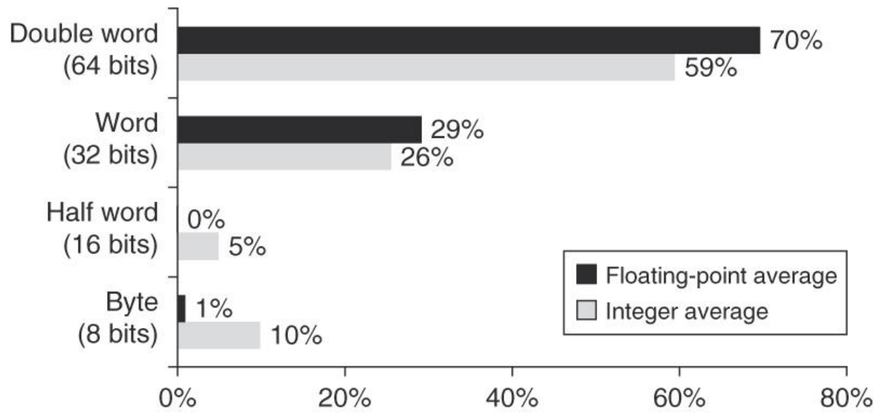
Summary of Addressing Modes (VAX)



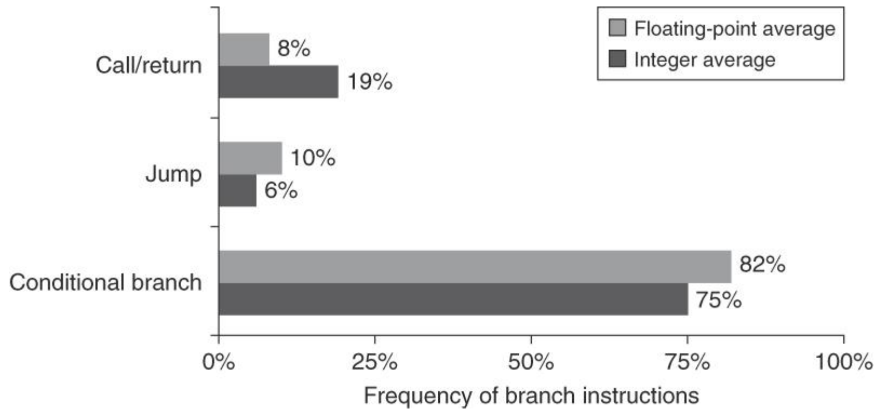
Distribution of Immediate Values



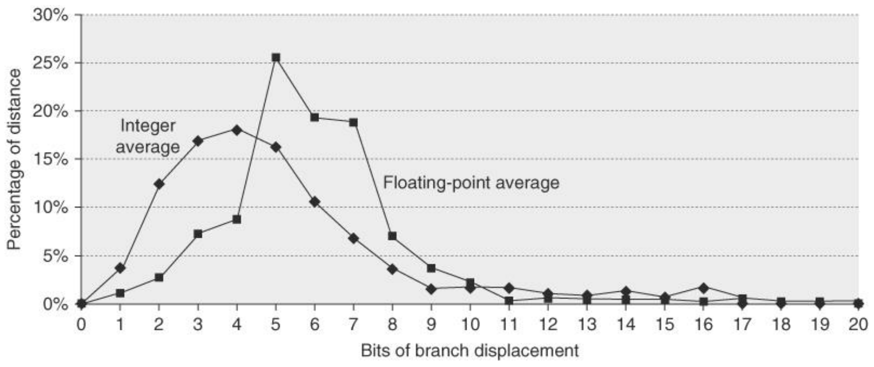
Distribution of Data Access by Size for Benchmarks



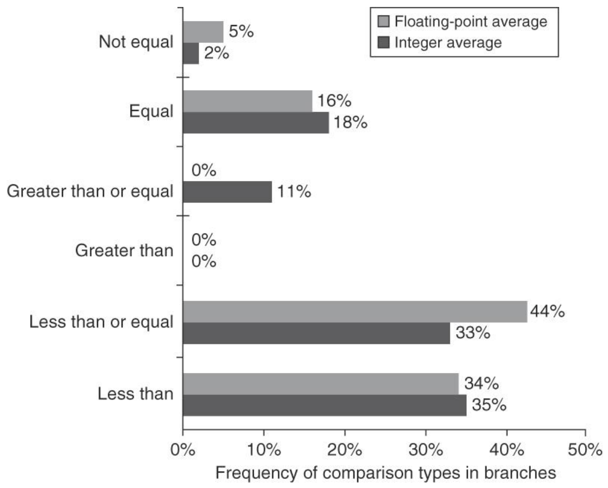
Control Flow Instructions



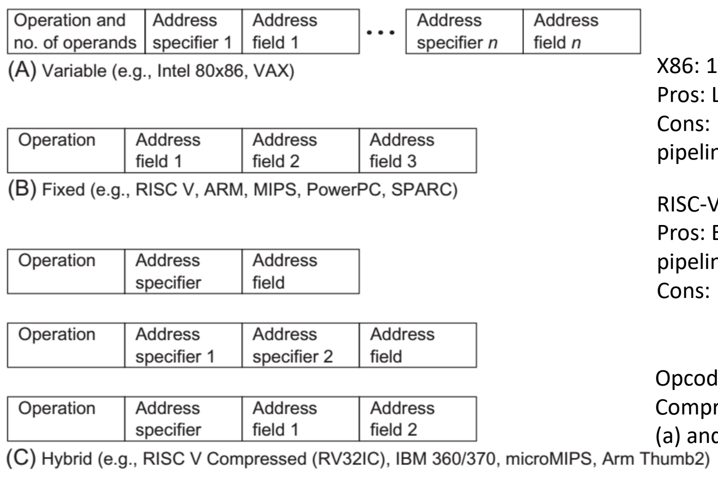
Branch Distance



Frequency of Comparisons in Branch Instructions



Instruction Encoding



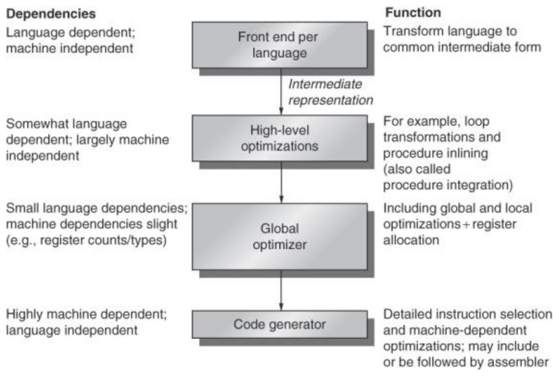
X86: 1-17 bytes
Pros: Less code
Cons: Hard to decode, pipeline

RISC-V: 4 bytes
Pros: Easy to decode, pipeline
Cons: More code

Opcode gives length
Compromise between (a) and (b)

Role of Compilers

First and foremost, compiler must be correct!
Most compilers make 2 or more passes to optimize, generate code


Dependencies
Language dependent; machine independent
Somewhat language dependent; largely machine independent
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Highly machine dependent; language independent
Front end per language
Intermediate representation
High-level optimizations
Global optimizer
Code generator
Function
Transform language to common intermediate form
For example, loop transformations and procedure inlining (also called procedure integration)
Including global and local optimizations + register allocation
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

Example: Register allocation

- Consider the program

  a = c + d
  e = a + b
  f = e - 1

  – with the assumption that a and e <u>die after use</u>
  – Obvious allocation: six registers, one per variable. What if we only have four registers available?

- Temporary a can be "reused" after e = a + b
- The same - Temporary e can be reused after f = e - 1
- Can allocate a, e, and f all to one register (r_1):

  $r_1 = r_2 + r_3$
  $r_1 = r_1 + r_4$
  $r_1 = r_1 - 1$

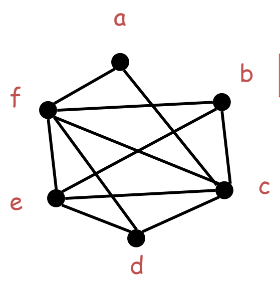
Basic Register Allocation Idea

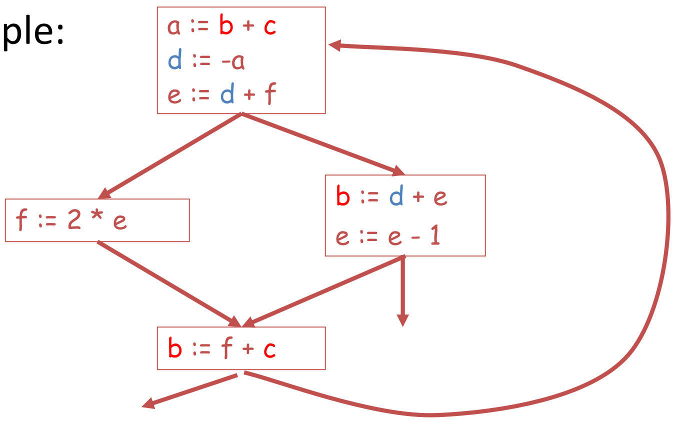
- The value in a dead temporary is not needed for the rest of the computation
  – A dead temporary can be reused

- **Basic rule**:
  – *Temporaries t_1 and t_2 can share the same register if <u>at any point in the program at most one</u> of t_1 or t_2 is live !*

Register Interference Graph. Example.

- For our example:





- E.g., b and c cannot be in the same register
- E.g., b and d can be in the same register

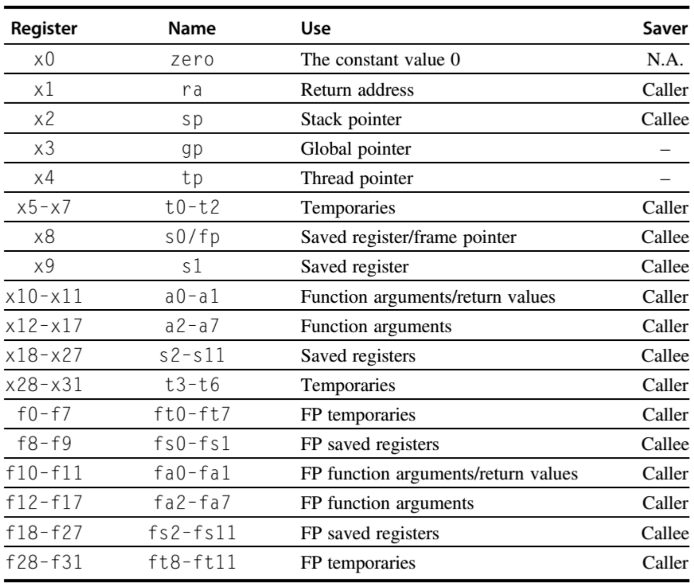
RISC-V Basics

- All operations on data apply to data in registers and typically change the entire register
- The only operations that affect memory are load and store operations
- Both a 32-bit (RV32) or 64-bit (RV64) base instruction set

- Text uses mostly RV64G
  - Instructions generally have a **D** at the start or end of the mnemonic, e.g. **LD** is 64 bit Load while **LW** is the 32 bit Load
  - D stands for Double Word where a Word = 32 bits

RISC-V ISA

- 32 general purpose registers, each 64 bits
  - x0, x1, … x31
  - Register x0 always has the value 0
- 32 floating point registers, f0 to f31
  - Either 32 or 64 bit values
- Three classes of instructions
  - ALU instructions
    - Register to register or immediate to register
    - Signed or unsigned
    - Floating point or Integer
    - NOT to memory
  - Load/Store instructions
    - Base register added to signed offset to get an effective address
  - Branches and Jumps
    - Branch based on condition bit or comparison between pair of registers



Register	Name	Use	Saver
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP temporaries	Caller
f8-f9	fs0-fs1	FP saved registers	Callee
f10-f11	fa0-fa1	FP function arguments/return values	Caller
f12-f17	fa2-fa7	FP function arguments	Caller
f18-f27	fs2-fs11	FP saved registers	Callee
f28-f31	ft8-ft11	FP temporaries	Caller

Figure 1.4 RISC-V registers, names, usage, and calling conventions. In addition to the 32 general-purpose registers (x0–x31), RISC-V has 32 floating-point registers (f0–f31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number. The registers that are preserved across a procedure call are labeled "Callee" saved.

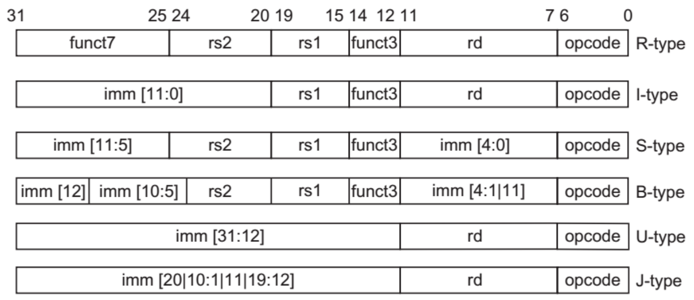


Figure 1.7 The base RISC-V instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the J format is for jumps and link. The S format is for stores. Having a separate format for stores allows the three register specifiers (rd, rs1, rs2) to always be in the same location in all formats. The U format is for the wide immediate instructions (LUI, AUIPC).

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, store word (to/from integer registers)
`ld, sd`	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
`add, addi, addw, addiw, sub, subi, subw, subiw`	Add and subtract, with both word and immediate versions
`slt, sltu, slti, sltiu`	set-less-than with signed and unsigned, and immediate
`and, or, xor, andi, ori, xori`	and, or, xor, both register-register and register-immediate
`lui`	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
`auipc`	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
`sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw`	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
`mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw`	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions

Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
`jal,jalr`	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
`flw, fld, fsw, fsd`	Load, store, word (single precision), doubleword (double precision)
`fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx`	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
`feq, flt, fle`	Compare two floating point registers; result is 0 or 1 stored into a GPR
`fmv.x.*, fmv.*.x`	Move between the FP register abd GPR, "*" is s or d
`fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*`	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

Figure A.28 A list of the vast majority of instructions in RV64G. This list can also be found on the back inside cover. This table omits system instructions, synchronization and atomic instructions, configuration instructions, instructions to reset and access performance counters, about 10 instructions in total.

Notation

- A subscript is appended to the symbol $\leftarrow$ whenever the length of the datum being transferred might not be clear. Thus, $\leftarrow_n$ means transfer an n-bit quantity. We use $x, y \leftarrow z$ to indicate that z should be transferred to x and y.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most-significant bit starting at 0. The subscript may be a single digit (e.g., $\texttt{Regs[R4]}_0$ yields the sign bit of R4) or a subrange (e.g., $\texttt{Regs[R3]}_{56..63}$ yields the least-significant byte of R3).
- The variable `Mem`, used as an array that stands for main memory, is indexed by a byte address and may transfer any number of bytes.
- A superscript is used to replicate a field (e.g., 0^{48} yields a field of zeros of length 48 bits).
- The symbol `##` is used to concatenate two fields and may appear on either side of a data transfer.

Example instruction	Instruction name	Meaning
`ld x1,80(x2)`	Load doubleword	Regs[x1]$\leftarrow$Mem[80+Regs[x2]]
`lw x1,60(x2)`	Load word	Regs[x1]$\leftarrow_{64}$ Mem[60+Regs[x2]$]_0)^{32}$ ## Mem[60+Regs[x2]]
`lwu x1,60(x2)`	Load word unsigned	Regs[x1]$\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
`lb x1,40(x3)`	Load byte	Regs[x1]$\leftarrow_{64}$ (Mem[40+Regs[x3]$]_0)^{56}$ ## Mem[40+Regs[x3]]
`lbu x1,40(x3)`	Load byte unsigned	Regs[x1]$\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
`lh x1,40(x3)`	Load half word	Regs[x1]$\leftarrow_{64}$ (Mem[40+Regs[x3]$]_0)^{48}$ ## Mem[40+Regs[x3]]
`flw f0,50(x3)`	Load FP single	Regs[f0]$\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
`fld f0,50(x2)`	Load FP double	Regs[f0]$\leftarrow_{64}$ Mem[50+Regs[x2]]
`sd x2,400(x3)`	Store double	Mem[400+Regs[x3]]$\leftarrow_{64}$ Regs[x2]
`sw x3,500(x4)`	Store word	Mem[500+Regs[x4]]$\leftarrow_{32}$ Regs[x3]$_{32..63}$
`fsw f0,40(x3)`	Store FP single	Mem[40+Regs[x3]]$\leftarrow_{32}$ Regs[f0]$_{0..31}$
`fsd f0,40(x3)`	Store FP double	Mem[40+Regs[x3]]$\leftarrow_{64}$ Regs[f0]
`sh x3,502(x2)`	Store half	Mem[502+Regs[x2]]$\leftarrow_{16}$ Regs[x3]$_{48..63}$
`sb x2,41(x3)`	Store byte	Mem[41+Regs[x3]]$\leftarrow_{8}$ Regs[x2]$_{56..63}$

Figure A.25 The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	Regs[x1]$\leftarrow$Regs[x2]+Regs[x3]
`addi x1,x2,3`	Add immediate unsigned	Regs[x1]$\leftarrow$Regs[x2]+3
`lui x1,42`	Load upper immediate	Regs[x1]$\leftarrow$$0^{32}$##42##$0^{12}$
`sll x1,x2,5`	Shift left logical	Regs[x1]$\leftarrow$Regs[x2]<<5
`slt x1,x2,x3`	Set less than	if (Regs[x2]<Regs[x3]) Regs[x1]$\leftarrow$1 else Regs[x1]$\leftarrow$0

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	Regs[x1]$\leftarrow$PC+4; PC$\leftarrow$PC + (offset<<1)
`jalr x1,x2,offset`	Jump and link register	Regs[x1]$\leftarrow$PC+4; PC$\leftarrow$Regs[x2]+offset
`beq x3,x4,offset`	Branch equal zero	if (Regs[x3]==Regs[x4]) PC$\leftarrow$PC + (offset<<1)
`bgt x3,x4,name`	Branch not equal zero	if (Regs[x3]>Regs[x4]) PC$\leftarrow$PC + (offset<<1)

Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

Examples

HLL code:

```
A = B + C + D;
E = F - A;
```

RISC_V code:

```
ADD $t0, $s1, $s2
ADD $s0, $t0, $s3
SUB $s4, $s5, $s0
```

Operands must be registers

- Compiler tries to keep as many variables in registers as possible
- Some variables can not be allocated
  - large arrays
  - aliased variables (variables accessible through pointers)
  - dynamically allocated variables on the heap or stack
- Compiler may run out of registers; this is called **spilling**

Instructions: load and store

Example:

HLL code:

```
A[3] = h + A[3];
```

RISC-V code:

```
LD   $t0, 24($s3)
ADD  $t0, $s2, $t0
SD   $t0, 24($s3)
```

- 8 bytes per dword → offset to 3rd dword → 24 byte displacement
- h already in register $s2
- Store word operation has no destination (reg) operand

Swap example

C

```
swap(int v[], int k)
{
   int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

RISC-V

```
swap:
   add a5, a0, a1
   lw  a4, 0(a5)
   add a1, a1, 4
   add a0, a0, a1
   lw  a3, 0(a0)
   sw  a3, 0(a5)
   sw  a4, 0(a0)
   ret
```

Explanation:
- index k : a1
- base address of v: a0
- temp: a4

What's this do?

```
    LI      a4, 0
    LI      a5, 0
    J       L2
L3:
    ADD     a5, a4, a5
    ADD     a4, a4, 1
L2:
    LI      a3, 5
    BLT     a4, a3, L3
```