Concepts Introduced in Appendix A

- classifying instruction set architectures
- memory addressing
- operands
- operations
- interaction between compilers and architecture
- ISA design
- RISC-V ISA

Instruction Set Types

- An *n* address architecture indicates the number of operands that have to be specified with each arithmetic operation.
- stack architecture (zero address machine)
- accumulator architecture (one address machine)
- general-purpose register architecture (two and three address machines)
  - register-register (load-store)
  - register-memory
  - memory-memory

Code Sequences for C = A + B

- stack

```
Push A   ==  M[SP]=M[A]; SP=SP+1;
Push B   ==  M[SP]=M[B]; SP=SP+1;
Add      ==  M[SP-2]=M[SP-1]+M[SP-2]; SP=SP-1;
Pop C    ==  C=M[SP-1]; SP=SP-1;
```

- accumulator

```
Load A    ==  AC=M[A];
Add B     ==  AC=AC+M[B];
Store C   ==  M[C]=AC;
```

- register (register-memory)

```
Load R1,A      ==  r[1]=M[A];
Add R3,R1,B    ==  r[3]=r[1]+M[B];
Store R3,C     ==  M[C]=r[3];
```

- register (load-store)

```
Load R1,A       ==  r[1]=M[A];
Load R2,B       ==  r[2]=M[B];
Add R3,R1,R2    ==  r[3]=r[1]+r[2];
Store R3,C      ==  M[C]=r[3];
```


Memory Accesses and Total Operands for ALU Instructions

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Advantages and Disadvantages of Common ISAs

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

Addressing Issues

- byte addressable
- number of bytes in a word
- byte order (little endian or big endian)
- alignment requirements
- extension of bytes and halfwords

Aligned and Misaligned Addresses

- An aligned object's memory address must be an integer multiple of the size of the object being accessed.

<table>
<tr><th></th><th colspan="8">Value of three low-order bits of byte address</th></tr>
<tr><th>Width of object</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>1 byte (byte)</td><td>Aligned</td><td>Aligned</td><td>Aligned</td><td>Aligned</td><td>Aligned</td><td>Aligned</td><td>Aligned</td><td>Aligned</td></tr>
<tr><td>2 bytes (half word)</td><td colspan="2">Aligned</td><td colspan="2">Aligned</td><td colspan="2">Aligned</td><td colspan="2">Aligned</td></tr>
<tr><td>2 bytes (half word)</td><td></td><td colspan="2">Misaligned</td><td colspan="2">Misaligned</td><td colspan="2">Misaligned</td><td>Misaligned</td></tr>
<tr><td>4 bytes (word)</td><td colspan="4">Aligned</td><td colspan="4">Aligned</td></tr>
<tr><td>4 bytes (word)</td><td></td><td colspan="4">Misaligned</td><td colspan="3">Misaligned</td></tr>
<tr><td>4 bytes (word)</td><td colspan="2"></td><td colspan="4">Misaligned</td><td colspan="2">Misaligned</td></tr>
<tr><td>4 bytes (word)</td><td colspan="3"></td><td colspan="4">Misaligned</td><td>Misaligned</td></tr>
<tr><td>8 bytes (double word)</td><td colspan="8">Aligned</td></tr>
<tr><td>8 bytes (double word)</td><td></td><td colspan="7">Misaligned</td></tr>
<tr><td>8 bytes (double word)</td><td colspan="2"></td><td colspan="6">Misaligned</td></tr>
<tr><td>8 bytes (double word)</td><td colspan="3"></td><td colspan="5">Misaligned</td></tr>
<tr><td>8 bytes (double word)</td><td colspan="4"></td><td colspan="4">Misaligned</td></tr>
<tr><td>8 bytes (double word)</td><td colspan="5"></td><td colspan="3">Misaligned</td></tr>
<tr><td>8 bytes (double word)</td><td colspan="6"></td><td colspan="2">Misaligned</td></tr>
<tr><td>8 bytes (double word)</td><td colspan="7"></td><td>Misaligned</td></tr>
</table>

Addressing Modes

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4] ← Regs[R4] + Regs[R3]	When a value is in a register
Immediate	Add R4,3	Regs[R4] ← Regs[R4] + 3	For constants
Displacement	Add R4,100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4,(R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address
Indexed	Add R3,(R1+R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2 = index amount
Direct or absolute	Add R1,(1001)	Regs[R1] ← Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	Add R1,@(R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer *p*, then mode yields **p*
Autoincrement	Add R1,(R2)+	Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2] ← Regs[R2] + *d*	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*
Autodecrement	Add R1, -(R2)	Regs[R2] ← Regs[R2] - *d* Regs[R1] ← Regs[R1] + Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3] * *d*]	Used to index arrays. May be applied to any indexed addressing mode in some computers

DSP Addressing Modes

- DSP and other special-purpose processors sometimes have special purpose addressing modes to support specific types of applications.
  - Modulo or circular addressing to support continuous streams of data.
  - Bit reverse addressing to support fast fourier transforms.

Types and Sizes of Operands

- general-purpose operand types and sizes
  - *char* or *unsigned char*, 1 byte, two's complement or unsigned
  - *short* or *unsigned short*, 2 bytes, two's complement or unsigned
  - *int* or *unsigned int*, 4 bytes, two's complement or unsigned
  - *float*, 4 bytes, IEEE FPS
  - *long int* or *unsigned long int*, 4 or 8 bytes, two's complement or unsigned
  - pointer is 4 or 8 bytes
  - *double*, 8 bytes, IEEE FPS
  - *long long int* or *unsigned long long int*, 8 bytes, two's complement or unsigned
- DSP types
  - fixed point, fractions between -1 and +1

Instruction Operation Categories

Operator type	**Examples**
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

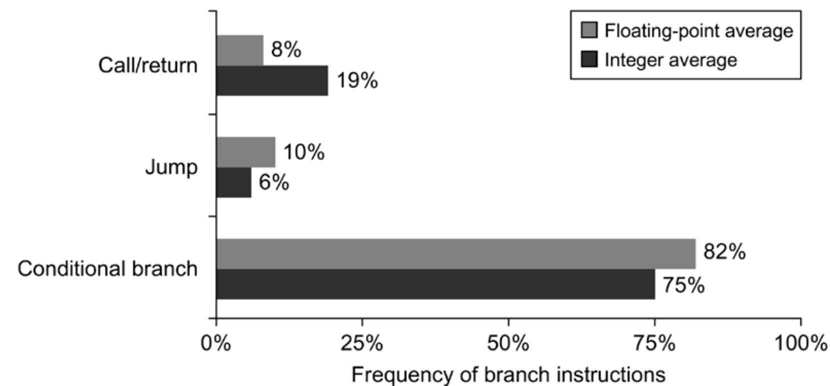
Operations for Media and Signal Processing

- Media and signal processor processors sometimes have special operations to support specific types of applications.
  - partitioned arithmetic operations
  - saturating arithmetic
  - special instructions tailored for applications (e.g. multiply-accumulate)

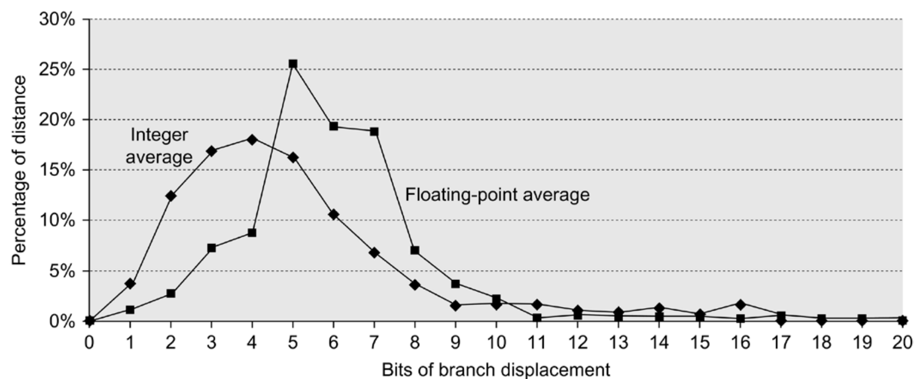
Transfers of Control

- conditional branches (pc-relative)
- unconditional jumps (pc-relative or immediate)
- direct procedure calls (immediate)
- returns (register)
- indirect procedure calls (register)
- large switch statements (register)

Breakdown of Control Flow Instructions



Branch Target Distances



Evaluating Branch Conditions

- condition codes (e.g. SPARC)

```
IC = r[2] ? r[3];
PC = IC < 0, L1;
```

  - N - bit set when result is negative
  - Z - bit set when result is zero

```
== Z
<  N
<= N || Z
>  !N && !Z
>= !N
!= !Z
```

- condition register (e.g. MIPS)

```
r[1] = (r[2] < r[3]) ? 1 : 0;
PC = r[1] != r[0], L1;
```

- compare and branch (e.g. VAX, RISC-V)

```
PC = r[2] < r[3], L1;
```

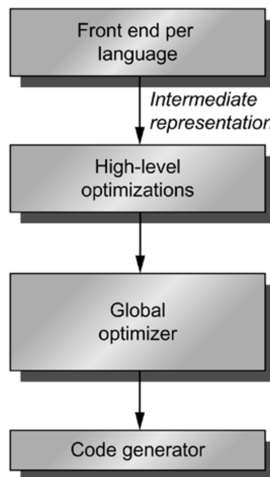
Evaluating Branch Conditions Methods

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Calling Conventions

- Identifying registers for special purposes.
  - stack pointer
  - return address
- Preserving the values of some registers across calls.
  - callee save
  - scratch (caller save)
- Transferring values between functions.
  - Passing argument values to a function through registers and on the stack.
  - Returning a value from a function.

Compilation Passes



Compiler Optimizations

- compiler optimization goals
  - Preserve semantic behavior.
  - Reduce execution time.
  - Decrease code size.
  - Reduce energy usage.
- levels of compiler optimizations
  - high-level (close to the source code level)
  - low-level (close to the machine code level)
  - local (within a basic block)
  - global (across basic blocks)
  - interprocedural (across function boundaries)

Major Types of Compiler Optimizations

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-independent*	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	2%
Processor-dependent	*Depends on processor knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

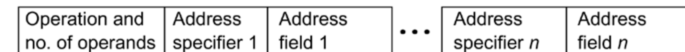
Register Allocation

- two levels of allocating registers:
  - register assignment - assigning temporaries to registers
  - register allocation - assigning live ranges of variables to registers
- areas of memory (most register allocation is performed for items on the stack)
  - run-time stack (some registers are dedicated to management of the stack)
  - static data (sometimes there is a dedicated register that points to the global data to reduce the cost of global data accesses)
  - heap
  - code (one register, the PC, is dedicated to point to the current instruction)

Tradeoffs When Encoding an Instruction Set

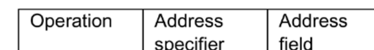
- Benefit of additional registers versus extra bits for referencing them.
- Simple formats simplify decoding and complex formats require less space.
- Fixed sized instructions simplify the fetch unit and pipelining and variable size instructions require less space.

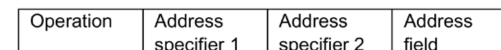
Common Instruction Encoding Techniques

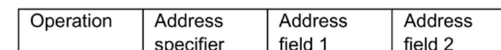


(A) Variable (e.g., Intel 80x86, VAX)

(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)







(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Instruction Set Properties

- A well designed instruction set can make life much easier for the compiler writer.
  - regularity
  - provide primitives, not solutions
  - simplify tradeoffs

Computer Architecture Periods

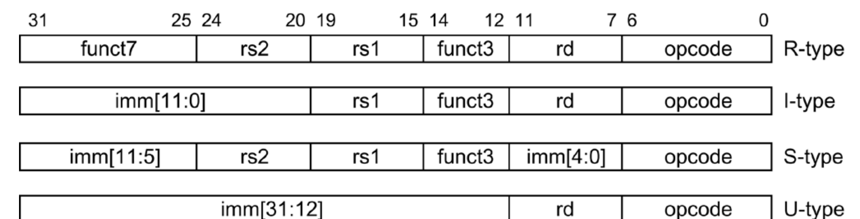
- 1960s
  - accumulator and stack architectures
- 1970s
  - high-level (CISC) architectures with microcode
- 1980s
  - load-store (RISC) architectures
  - increased reliance on compiler optimizations
- 1990s
  - doubling the address size
  - use of conditional (predicated) execution
  - prefetch instructions
  - support for multimedia and signal processing
- 2000s
  - run-time translation into micro-ops
  - virtual machines and JIT compilation
- 2010s
  - exploiting thread-level parallelism
  - support for domain-specific architectures

RISC-V Architecture

- RISC-V is a freely licensed open standard.
- Goal is an instruction set that is simple to implement.
- All examples are in the 64-bit version, where registers are 64 bits in length.
- Registers include general purpose (x0...x31) and floating point (f0...f31).
- Datatypes include 8-bit, 16-bit, 32-bit, and 64-bit integers and 32-bit single precision and 64-bit double precision.
- Addressing modes are immediate and displacement.

RISC-V Instruction Formats

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link



RISC-V Load and Store Instructions

Example instruction	Instruction name	Meaning
`ld x1,80(x2)`	Load doubleword	Regs[x1] ← 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]$_{56..63}$

RISC-V Basic ALU Instructions

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

RISC-V Typical Control Flow Instructions

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

Fallacies and Pitfalls

- Pitfall: Designing high-level instruction set features to support a high-level language structure.
- Fallacy: An architecture with flaws cannot be successful.
- Fallacy: You can design a flawless architecture.