Understanding Program Performance

People used to be taught to use pointers in C to get greater efficiency than that available with arrays: "Use pointers, even if you can't understand the code." Modern optimizing compilers can produce code for the array version that is just as good. Most programmers today prefer that the compiler do the heavy lifting.

2.15 Advanced Material: Compiling C and Interpreting Java

This section gives a brief overview of how the C compiler works and how Java is executed. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course, so our introduction will necessarily only touch on the basics.

The second part of this section is for readers interested in seeing how an **object oriented language** like Java executes on a MIPS architecture. It shows the Java byte-codes used for interpretation and the MIPS code for the Java version of some of the C segments in prior sections, including Bubble Sort. It covers both the Java Virtual Machine and JIT compilers.

object oriented language A programming language that is oriented around objects rather than actions, or data versus logic.

The rest of Section 2.15 can be found online.

2.16 Real Stuff: ARMv7 (32-bit) Instructions

ARM is the most popular instruction set architecture for embedded devices, with more than 9 billion devices in 2011 using ARM, and recent growth has been 2 billion per year. Standing originally for the Acorn RISC Machine, later changed to Advanced RISC Machine, ARM came out the same year as MIPS and followed similar philosophies. Figure 2.31 lists the similarities. The principal difference is that MIPS has more registers and ARM has more addressing modes.

There is a similar core of instruction sets for arithmetic-logical and data transfer instructions for MIPS and ARM, as Figure 2.32 shows.

Addressing Modes

Figure 2.33 shows the data addressing modes supported by ARM. Unlike MIPS, ARM does not reserve a register to contain 0. Although MIPS has just three simple data addressing modes (see Figure 2.18), ARM has nine, including fairly complex calculations. For example, ARM has an addressing mode that can shift one register

Note that this program calculates the address of the end of the array in every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
       move $t0,$a0          # p = address of array[0]
       sll  $t1,$a1,2        # $t1 = size * 4
       add  $t2,$a0,$t1      # $t2 = address of array[size]
loop2: sw   $zero,0($t0)     # Memory[p] = 0
       addi $t0,$t0,4        # p = p + 4
       slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
       bne  $t3,$zero,loop2 # if (p<&array[size]) go to loop2
```

Comparing the Two Versions of Clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

```
      move  $t0,$zero       # i = 0                  move  $t0,$a0       # p = & array[0]
loop1:sll   $t1,$t0,2       # $t1 = i * 4            sll   $t1,$a1,2     # $t1 = size * 4
      add   $t2,$a0,$t1     # $t2 = &array[i]        add   $t2,$a0,$t1   # $t2 = &array[size]
      sw    $zero, 0($t2)   # array[i] = 0     loop2:sw    $zero,0($t0)  # Memory[p] = 0
      addi  $t0,$t0,1       # i = i + 1              addi  $t0,$t0,4     # p = p + 4
      slt   $t3,$t0,$a1     # $t3 = (i < size)       slt   $t3,$t0,$t2   # $t3=(p<&array[size])
      bne   $t3,$zero,loop1# if () go to loop1       bne   $t3,$zero,loop2# if () go to loop2
```

The version on the left must have the "multiply" and add inside the loop because `i` is incremented and each address must be recalculated from the new index. The memory pointer version on the right increments the pointer `p` directly. The pointer version moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from 6 to 4. This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops). Section 2.15 describes these two and many other optimizations.

Elaboration: As mentioned ealier, a C compiler would add a test to be sure that `size` is greater than 0. One way would be to add a jump just before the first instruction of the loop to the `slt` instruction.

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	32	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

FIGURE 2.31 Similarities in ARM and MIPS instruction sets.

	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl^1	sllv, sll
	Shift right logical	lsr^1	srlv, srl
	Shift right arithmetic	asr^1	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i,slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

FIGURE 2.32 ARM register-register and data transfer instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture or not synthesized in a few instructions. If there are several choices of instructions equivalent to the MIPS core, they are separated by commas. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 1 are just a variation of a move instruction, such as lsr^1. Note that ARM has no divide instruction.

by any amount, add it to the other registers to form the address, and then update one register with this new address.

Addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

FIGURE 2.33 Summary of data addressing modes. ARM has separate register indirect and register + offset addressing modes, rather than just putting 0 in the offset of the latter mode. To get greater addressing range, ARM shifts the offset left 1 or 2 bits if the data size is halfword or word.

Compare and Conditional Branch

MIPS uses the contents of registers to evaluate conditional branches. ARM uses the traditional four condition code bits stored in the program status word: *negative, zero, carry,* and *overflow*. They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in a pipelined implementation. ARM uses conditional branches to test condition codes to determine all possible unsigned and signed relations.

CMP subtracts one operand from the other and the difference sets the condition codes. *Compare negative* (CMN) *adds* one operand to the other, and the sum sets the condition codes. TST performs logical AND on the two operands to set all condition codes but overflow, while TEQ uses exclusive OR to set the first three condition codes.

One unusual feature of ARM is that every instruction has the option of executing conditionally, depending on the condition codes. Every instruction starts with a 4-bit field that determines whether it will act as a no operation instruction (nop) or as a real instruction, depending on the condition codes. Hence, conditional branches are properly considered as conditionally executing the unconditional branch instruction. Conditional execution allows avoiding a branch to jump over a single instruction. It takes less code space and time to simply conditionally execute one instruction.

Figure 2.34 shows the instruction formats for ARM and MIPS. The principal differences are the 4-bit conditional execution field in every instruction and the smaller register field, because ARM has half the number of registers.

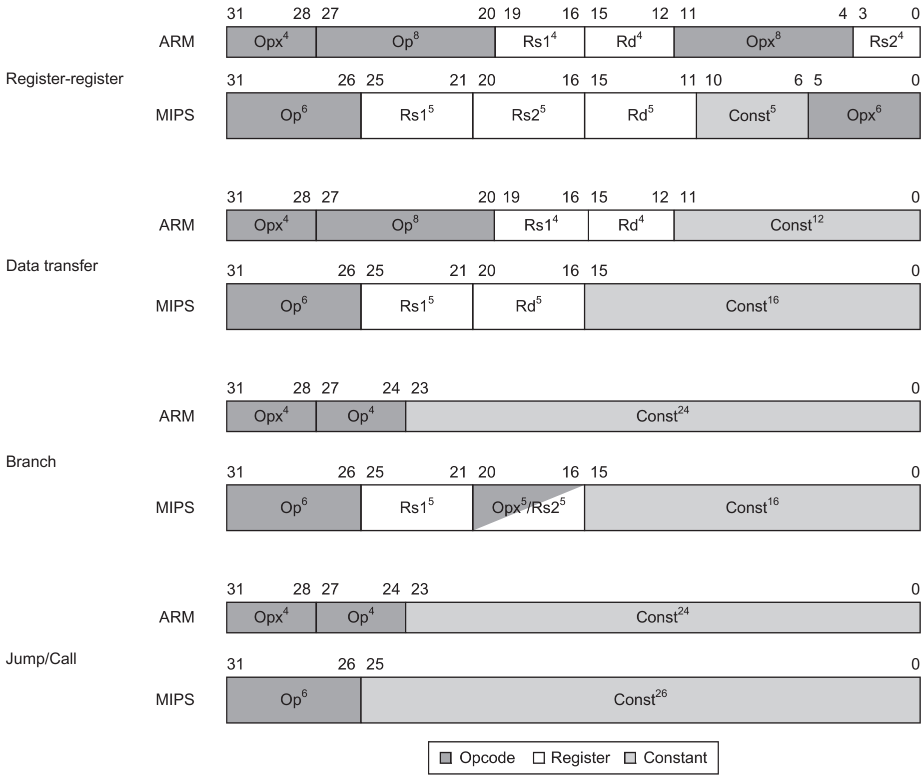

FIGURE 2.34 Instruction formats, ARM and MIPS. The differences result from whether the architecture has 16 or 32 registers.

Unique Features of ARM

Figure 2.35 shows a few arithmetic-logical instructions not found in MIPS. Since ARM does not have a dedicated register for 0, it has separate opcodes to perform some operations that MIPS can do with `$zero`. In addition, ARM has support for multiword arithmetic.

ARM's 12-bit immediate field has a novel interpretation. The eight least-significant bits are zero-extended to a 32-bit value, then rotated right the number of bits specified in the first four bits of the field multiplied by two. One advantage is that this scheme can represent all powers of two in a 32-bit word. Whether this split actually catches more immediates than a simple 12-bit field would be an interesting study.

Operand shifting is not limited to immediates. The second register of all arithmetic and logical processing operations has the option of being shifted before being operated on. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right.

Name	Definition	ARM	MIPS
Load immediate	Rd = Imm	mov	addi $0,
Not	Rd = ~(Rs1)	mvn	nor $0,
Move	Rd = Rs1	mov	or $0,
Rotate right	Rd = Rs i >> i $Rd_{0...i-1} = Rs_{31-i...31}$	ror	
And not	Rd = Rs1 & ~(Rs2)	bic	
Reverse subtract	Rd = Rs2 – Rs1	rsb, rsc	
Support for multiword integer add	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
Support for multiword integer sub	CarryOut, Rd = Rd – Rs1 + OldCarryOut	sbcs	—

FIGURE 2.35 ARM arithmetic/logical instructions not found in MIPS.

ARM also has instructions to save groups of registers, called *block loads and stores*. Under control of a 16-bit mask within the instructions, any of the 16 registers can be loaded or stored into memory in a single instruction. These instructions can save and restore registers on procedure entry and return. These instructions can also be used for block memory copy, and today block copies are the most important use of such instructions.

2.17 Real Stuff: x86 Instructions

Beauty is altogether in the eye of the beholder.

Margaret Wolfe Hungerford, *Molly Bawn*, 1877

Designers of instruction sets sometimes provide more powerful operations than those found in ARM and MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence.

The path toward operation complexity is thus fraught with peril. Section 2.19 demonstrates the pitfalls of complexity.

Evolution of the Intel x86

ARM and MIPS were the vision of single small groups in 1985; the pieces of these architectures fit nicely together, and the whole architecture can be described succinctly. Such is not the case for the x86; it is the product of several independent groups who evolved the architecture over 35 years, adding new features to the original instruction set as someone might add clothing to a packed bag. Here are important x86 milestones.

general-purpose register (GPR) A register that can be used for addresses or for data with virtually any instruction.

- **1978**: The Intel 8086 architecture was announced as an assembly language–compatible extension of the then successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Unlike MIPS, the registers have dedicated uses, and hence the 8086 is not considered a **general-purpose register** architecture.
- **1980**: The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack (see Section 2.21 and Section 3.7).
- **1982**: The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory-mapping and protection model (see Chapter 5), and by adding a few instructions to round out the instruction set and to manipulate the protection model.
- **1985**: The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 5). Like the 80286, the 80386 has a mode to execute 8086 programs without change.
- **1989–95**: The subsequent 80486 in 1989, Pentium in 1992, and Pentium Pro in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing (Chapter 6) and a conditional move instruction.
- **1997**: After the Pentium and Pentium Pro were shipping, Intel announced that it would expand the Pentium and the Pentium Pro architectures with MMX (Multi Media Extensions). This new set of 57 instructions uses the floating-point stack to accelerate multimedia and communication applications. MMX instructions typically operate on multiple short data elements at a time, in the tradition of *single instruction, multiple data* (SIMD) architectures (see Chapter 6). Pentium II did not introduce any new instructions.
- **1999**: Intel added another 70 instructions, labeled SSE (*Streaming SIMD Extensions*) as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single precision floating-point data type. Hence, four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE includes cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.
- **2001**: Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data

in parallel. Not only does this change enable more multimedia operations; it gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers like those found in other computers. This change boosted the floating-point performance of the Pentium 4, the first microprocessor to include SSE2 instructions.

- **2003**: A company other than Intel enhanced the x86 architecture this time. AMD announced a set of architectural extensions to increase the address space from 32 to 64 bits. Similar to the transition from a 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to 16 and increases the number of 128-bit SSE registers to 16. The primary ISA change comes from adding a new mode called *long mode* that redefines the execution of all x86 instructions with 64-bit addresses and data. To address the larger number of registers, it adds a new prefix to instructions. Depending how you count, long mode also adds four to ten new instructions and drops 27 old ones. PC-relative data addressing is another extension. AMD64 still has a mode that is identical to x86 (*legacy mode*) plus a mode that restricts user programs to x86 but allows operating systems to use AMD64 (*compatibility mode*). These modes allow a more graceful transition to 64-bit addressing than the HP/Intel IA-64 architecture.
- **2004**: Intel capitulates and embraces AMD64, relabeling it *Extended Memory 64 Technology* (EM64T). The major difference is that Intel added a 128-bit atomic compare and swap instruction, which probably should have been included in AMD64. At the same time, Intel announced another generation of media extensions. SSE3 adds 13 instructions to support complex arithmetic, graphics operations on arrays of structures, video encoding, floating-point conversion, and thread synchronization (see Section 2.11). AMD added SSE3 in subsequent chips and the missing atomic swap instruction to AMD64 to maintain binary compatibility with Intel.
- **2006**: Intel announces 54 new instructions as part of the SSE4 instruction set extensions. These extensions perform tweaks like sum of absolute differences, dot products for arrays of structures, sign or zero extension of narrow data to wider sizes, population count, and so on. They also added support for virtual machines (see Chapter 5).
- **2007**: AMD announces 170 instructions as part of SSE5, including 46 instructions of the base instruction set that adds three operand instructions like MIPS.
- **2011**: Intel ships the Advanced Vector Extension that expands the SSE register width from 128 to 256 bits, thereby redefining about 250 instructions and adding 128 new instructions.

This history illustrates the impact of the "golden handcuffs" of compatibility on the x86, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the x86, keep in mind that this instruction set largely drove the PC generation of computers and still dominates the cloud portion of the PostPC Era. Manufacturing 350M x86 chips per year may seem small compared to 9 billion ARMv7 chips, but many companies would love to control such a market. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

Brace yourself for what you are about to see! Do *not* try to read this section with the care you would need to write x86 programs; the goal instead is to give you familiarity with the strengths and weaknesses of the world's most popular desktop architecture.

Rather than show the entire 16-bit, 32-bit, and 64-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

x86 Registers and Data Addressing Modes

The registers of the 80386 show the evolution of the instruction set (Figure 2.36). The 80386 extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an *E* to their name to indicate the 32-bit version. We'll refer to them generically as GPRs (*general-purpose registers*). The 80386 contains only eight GPRs. This means MIPS programs can use four times as many and ARMv7 twice as many.

Figure 2.37 shows the arithmetic, logical, and data transfer instructions are two-operand instructions. There are two important differences here. The x86 arithmetic and logical instructions must have one operand act as both a source and a destination; ARMv7 and MIPS allow separate registers for source and destination. This restriction puts more pressure on the limited registers, since one source register must be modified. The second important difference is that one of the operands can be in memory. Thus, virtually any instruction may have one operand in memory, unlike ARMv7 and MIPS.

Data memory-addressing modes, described in detail below, offer two sizes of addresses within the instruction. These so-called *displacements* can be 8 bits or 32 bits.

Although a memory operand can use any addressing mode, there are restrictions on which *registers* can be used in a mode. Figure 2.38 shows the x86 addressing modes and which GPRs cannot be used with each mode, as well as how to get the same effect using MIPS instructions.

x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (*word*) data types. The 80386 adds 32-bit addresses and data (*double words*) in the x86. (AMD64 adds 64-

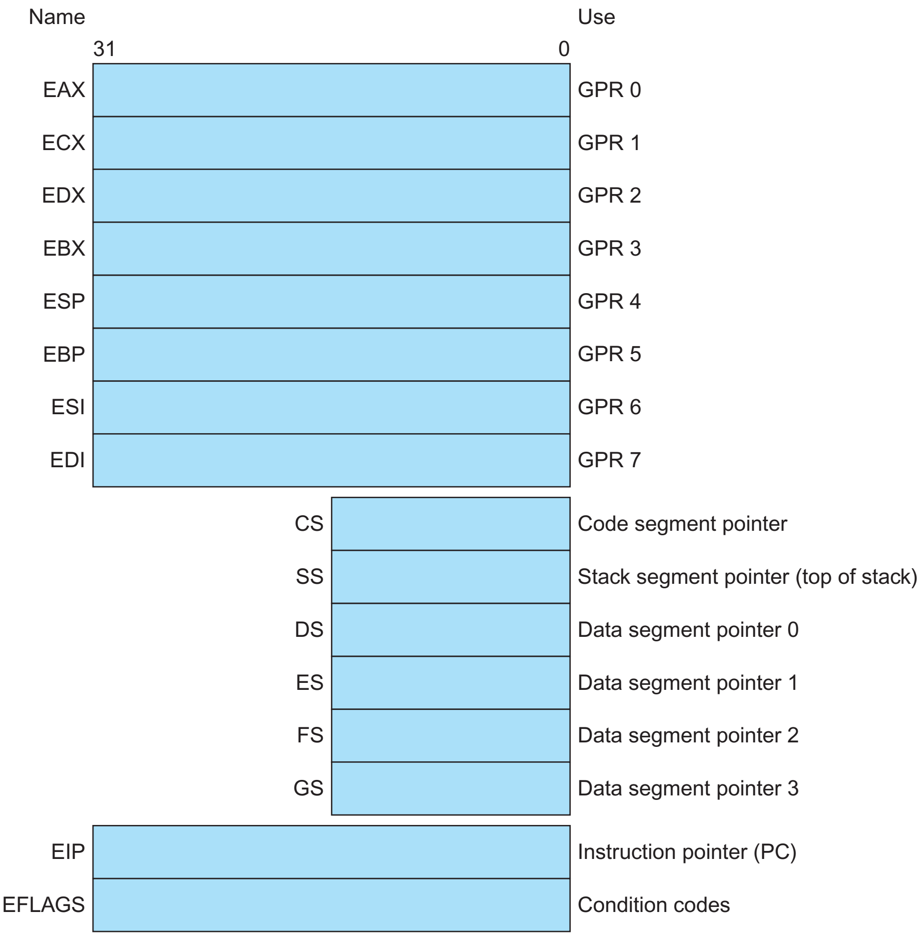

FIGURE 2.36 The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

FIGURE 2.37 Instruction types for the arithmetic, logical, and data transfer instructions. The x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 2.36 (not EIP or EFLAGS).

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`lw $s0,100($s1) # <= 16-bit # displacement`
Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,0($t0)`
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,100($t0) #<=16-bit # displacement`

FIGURE 2.38 x86 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in ARM or MIPS, is included to avoid the multiplies by 4 (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.25 and 2.27). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. A scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

bit addresses and data, called *quad words*; we'll stick to the 80386 in this section.) The data type distinctions apply to register operations as well as memory accesses.

Almost every operation works on both 8-bit data and on one longer data size. That size is determined by the mode and is either 16 bits or 32 bits.

Clearly, some programs want to operate on data of all three sizes, so the 80386 architects provided a convenient way to specify each version without expanding code size significantly. They decided that either 16-bit or 32-bit data dominates most programs, and so it made sense to be able to set a default large size. This default data size is set by a bit in the code segment register. To override the default data size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus to support synchronization (see Section 2.11), or repeat the following instruction until the register ECX counts down to 0. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop
2. Arithmetic and logic instructions, including test, integer, and decimal arithmetic operations
3. Control flow, including conditional branches, unconditional jumps, calls, and returns
4. String instructions, including string move and string compare

The first two categories are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location. Figure 2.39 shows some typical x86 instructions and their functions.

Conditional branches on the x86 are based on *condition codes* or *flags*, like ARMv7. Condition codes are set as a side effect of an operation; most are used to compare the value of a result to 0. Branches then test the condition codes. PC-

Instruction	Function
`je` name	`if equal(condition code) {EIP=name};` `EIP-128 <= name < EIP+128`
`jmp` name	`EIP=name`
`call` name	`SP=SP-4; M[SP]=EIP+5; EIP=name;`
`movw EBX,[EDI+45]`	`EBX=M[EDI+45]`
`push ESI`	`SP=SP-4; M[SP]=ESI`
`pop EDI`	`EDI=M[SP]; SP=SP+4`
`add EAX,#6765`	`EAX= EAX+6765`
`test EDX,#42`	Set condition code (flags) with EDX and 42
`movsl`	`M[EDI]=M[ESI];` `EDI=EDI+4; ESI=ESI+4`

FIGURE 2.39 Some typical x86 instructions and their functions. A list of frequent operations appears in Figure 2.40. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

relative branch addresses must be specified in the number of bytes, since unlike ARMv7 and MIPS, 80386 instructions are not all 4 bytes in length.

String instructions are part of the 8080 ancestry of the x86 and are not commonly executed in most programs. They are often slower than equivalent software routines (see the fallacy on page 159).

Figure 2.40 lists some of the integer x86 instructions. Many of the instructions are available in both byte and word formats.

x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 80386 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there are no operands, up to 15 bytes.

Figure 2.41 shows the instruction format for several of the example instructions in Figure 2.39. The opcode byte usually contains a bit saying whether the operand is 8 bits or 32 bits. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form "register = register op immediate." Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m," which contains the addressing mode information. This postbyte is used for many

Instruction	Meaning
Control	**Conditional and unconditional branches**
`jnz, jz`	Jump if condition to EIP + 8-bit offset; `JNE` (for`JNZ`), `JE` (for `JZ`) are alternative names
`jmp`	Unconditional jump—8-bit or 16-bit offset
`call`	Subroutine call—16-bit offset; return address pushed onto stack
`ret`	Pops return address from stack and jumps to it
`loop`	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
`move`	Move between two registers or between register and memory
`push, pop`	Push source operand on stack; pop operand from stack top to a register
`les`	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
`add, sub`	Add source to destination; subtract source from destination; register-memory format
`cmp`	Compare source and destination; register-memory format
`shl, shr, rcr`	Shift left; shift logical right; rotate right with carry condition code as fill
`cbw`	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
`test`	Logical AND of source and destination sets condition codes
`inc, dec`	Increment destination, decrement destination
`or, xor`	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
`movs`	Copies from string source to destination by incrementing ESI and EDI; may be repeated
`lods`	Loads a byte, word, or doubleword of a string into the EAX register

FIGURE 2.40 Some typical operations on the x86. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled "sc, index, base."

Figure 2.42 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit mode. Unfortunately, to understand fully which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes even the encoding of the instructions.

x86 Conclusion

Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this head start led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the x86 is more difficult to build than computers like ARMv7 and MIPS, but the large market meant in the PC Era that AMD and Intel could afford more resources

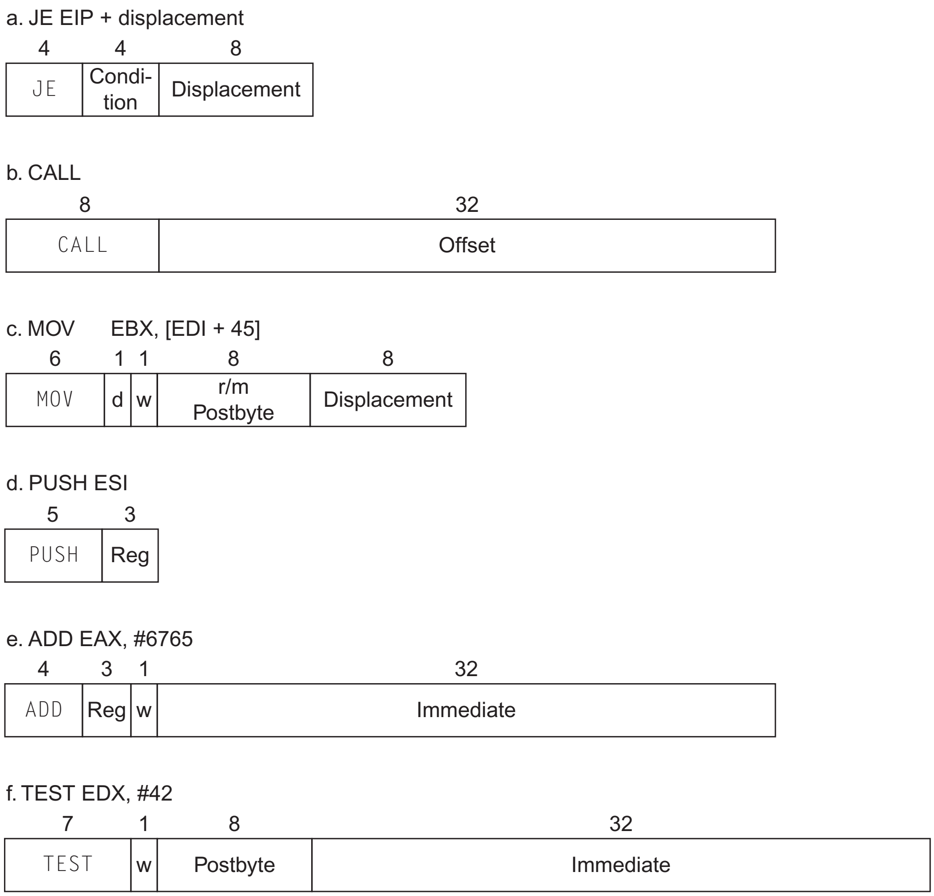

FIGURE 2.41 Typical x86 instruction formats. Figure 2.42 shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a double word. The d field in `MOV` is used in instructions that may move to or from memory and shows the direction of the move. The `ADD` instruction requires 32 bits for the immediate field, because in 32-bit mode, the immediates are either 8 bits or 32 bits. The immediate field in the `TEST` is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 15 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

to help overcome the added complexity. What the x86 lacks in style, it made up for in market size, making it beautiful from the right perspective.

Its saving grace is that the most frequently used x86 architectural components are not too difficult to implement, as AMD and Intel have demonstrated by rapidly improving performance of integer programs since 1978. To get that performance,

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	=*(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	“
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	“
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	“
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	“

FIGURE 2.42 The encoding of the first address specifier of the x86: mod, reg, r/m. The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16-bit mode (8086) or 32-bit mode (80386). The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16-bit or 32-bit displacement, depending on the address mode. The exceptions are 1) r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; 2) r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and 3) r/m = 4 in 32-bit mode when mod does not equal 3, where (sib) means use the scaled index mode shown in Figure 2.38. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

compilers must avoid the portions of the architecture that are hard to implement fast.

In the PostPC Era, however, despite considerable architectural and manufacturing expertise, x86 has not yet been competitive in the personal mobile device.

2.18 Real Stuff: ARMv8 (64-bit) Instructions

Of the many potential problems in an instruction set, the one that is almost impossible to overcome is having too small a memory address. While the x86 was successfully extended first to 32-bit addresses and then later to 64-bit addresses, many of its brethren were left behind. For example, the 16-bit address MOStek 6502 powered the Apple II, but even given this headstart with the first commercially successful personal computer, its lack of address bits condemned it to the dustbin of history.

ARM architects could see the writing on the wall of their 32-bit address computer, and began design of the 64-bit address version of ARM in 2007. It was finally revealed in 2013. Rather than some minor cosmetic changes to make all the registers 64 bits wide, which is basically what happened to the x86, ARM did a complete overhaul. The good news is that if you know MIPS it will be very easy to pick up ARMv8, as the 64-bit version is called.

First, as compared to MIPS, ARM dropped virtually all of the unusual features of v7:

- There is no conditional execution field, as there was in nearly every instruction in v7.

- The immediate field is simply a 12 bit constant, rather than essentially an input to a function that produces a constant as in v7.
- ARM dropped Load Multiple and Store Multiple instructions.
- The PC is no longer one of the registers, which resulted in unexpected branches if you wrote to it.

Second, ARM added missing features that are useful in MIPS

- V8 has 32 general-purpose registers, which compiler writers surely love. Like MIPS, one register is hardwired to 0, although in load and store instructions it instead refers to the stack pointer.
- Its addressing modes work for all word sizes in ARMv8, which was not the case in ARMv7.
- It includes a divide instruction, which was omitted from ARMv7.
- It adds the equivalent of MIPS branch if equal and branch if not equal.

As the philosophy of the v8 instruction set is much closer to MIPS than it is to v7, our conclusion is that the main similarity between ARMv7 and ARMv8 is the name.

2.19 Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel x86 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the following instruction until a counter counts down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to memory. This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times as fast. A third version, which uses the larger floating-point registers instead of the integer registers of the x86, copies at about 2.0 times as fast than the complex move instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naïve instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to understand the concepts in Chapters 4 and 5 thoroughly (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is another situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables to keep in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some old C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore such hints, because the compiler does a better job at allocation than the programmer does.

Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are the longer time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C or Java. Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and new models of machines. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to future machines; it also makes the software easier to maintain and allows the program to run on more brands of computers.

Fallacy: The importance of commercial binary compatibility means successful instruction sets don't change.

While backwards binary compatibility is sacrosanct, Figure 2.43 shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 35-year lifetime!

Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by one instead of by the word size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is local to that procedure. Following the stack discipline in Figure 2.12, the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

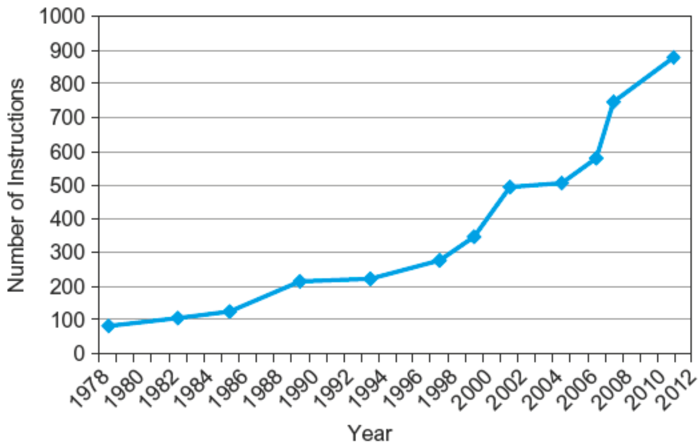

FIGURE 2.43 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

2.20 Concluding Remarks

Less is more.

Robert Browning, *Andrea del Sarto*, 1855

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid environmental scientists, financial advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. As illustrated in this chapter, three design principles guide the authors of instruction sets in making that delicate balance:

1. *Simplicity favors regularity.* Regularity motivates many features of the MIPS instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in each instruction format.
2. *Smaller is faster.* The desire for speed is the reason that MIPS has 32 registers rather than many more.
3. *Good design demands good compromises.* One MIPS example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

COMMON CASE FAST

We also saw the great idea of making the **common cast fast** applied to instruction sets as well as computer architecture. Examples of making the common MIPS case fast include PC-relative addressing for conditional branches and immediate addressing for larger constant operands.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of instructions are given their own name, and so on. Figure 2.44 lists the MIPS instructions we have covered

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURE 2.44 The MIPS instruction set covered so far, with the real MIPS instructions on the left and the pseudoinstructions on the right. Appendix A (Section A.10) describes the full MIPS architecture. Figure 2.1 shows more details of the MIPS architecture revealed in this chapter. The information given here is also found in Columns 1 and 2 of the MIPS Reference Data Card at the front of the book.

so far, both real and pseudoinstructions. Hiding details from the higher level is another example of the great idea of **abstraction**.

ABSTRACTION

Each category of MIPS instructions is associated with constructs that appear in programming languages:

- Arithmetic instructions correspond to the operations found in assignment statements.
- Transfer instructions are most likely to occur when dealing with data structures like arrays or structures.
- Conditional branches are used in *if* statements and in loops.
- Unconditional jumps are used in procedure calls and returns and for *case/switch* statements.

These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 2.45 shows the popularity of each class of instructions for SPEC CPU2006. The varying popularity of instructions plays an important role in the chapters about datapath, control, and pipelining.

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	`add, sub, addi`	Operations in assignment statement s	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	References to data structures, such as arrays	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	Operations in assignment statement s	12%	4%
Conditional branch	`beq, bne, slt, slti, sltiu`	*If* statements and loops	34%	8%
Jump	`j, jr, jal`	Procedure calls, returns, and *case/switch* statements	2%	0%

FIGURE 2.45 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks. Figure 3.26 in Chapter 3 shows average percentage of the individual MIPS instructions executed.

After we explain computer arithmetic in Chapter 3, we reveal the rest of the MIPS instruction set architecture.

2.21 Historical Perspective and Further Reading

This section surveys the history of *instruction set architectures* (ISAs) over time, and we give a short history of programming languages and compilers. ISAs