Instructions: Language of the Machine

Ming-Hwa Wang, Ph.D.
COEN 210 Computer Architecture
Department of Computer Engineering
Santa Clara University

Introduction

- Instructions are the words of a machine's language, and its vocabulary is the instruction set
- Machine languages are similar: all computers are constructed from hardware technologies based on similar underlying principles, and computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost

Design Principles

1. simplicity favors regularity
2. smaller is faster
3. good design demands good compromises
4. make the common case fast

Operations of the Computer Hardware

- the MIPS arithmetic instruction set (design principle 1)
  - each line con contain at most one instruction
  - each arithmetic operation has exact 3 operands
  - comments: for human reader but ignored by the computer, by the sharp symbol (#) to the right of end of the line
  - arithmetic instructions: **add**, **subtract**
- compiler: translate from high-level language to low-level
  - simple C statements with 3 operands are converted to MIPS assembly language instructions directly
  - for complex C statements, the compiler breaks a statement into several assembly instructions, and creates temporary variables when needed to store the intermediate results

Operands of the Computer Hardware

- all variables have to be loaded in a limited number of registers before a operation can be done
- the MIPS architecture has 32 registers, each has 32 bits (a word)
- a very large number of registers would increase the clock cycle time because it takes electronic signals longer when they must travel farther (design principle 2)
- although we could simply write instructions using numbers for registers from 0 to 31, the MIPS convention is to use two character names following a dollar sign ($) to represent a register
  - register $zero (map onto register 0), always constant 0
  - register $at (map onto register 1), reserved for the assembler
  - return value registers $v0 and $v1 (map onto registers 2 and 3) for result value and expression evaluation, not preserved
  - argument registers $a0 .. $a3 (map onto registers 4 to 7), not preserved
  - temporary registers $t0 .. $t7 (map onto registers 8 to 15), $t8 and $t9 (map onto registers 24 and 25), not preserved
  - saved registers $s0 .. $s7 (map onto registers 16 to 23) to store variables in C programs, must be preserved on a procedure call (if used, the callee saves and restores them)
  - kernel registers $k0 and $k1 (map onto registers 26 and 27), reserved for operating system
  - global pointer register $gp (map onto register 28)
  - stack pointer register $sp (map onto register 29), preserved
  - frame pointer register $fp (map onto register 30) or $s8 (if no $fp used, e.g., C compiler on MIPS at SGI)
  - return address register $ra (map onto register 31), preserved
- it is the compiler's job to associate program variables with registers
- byte-addressable: most architectures address individual bytes, therefore, the address of a word matches the address of one of the 4 bytes within the word, and hence addresses of sequential words differ by 4 (the word-addressable machines differ by 1)
- alignment restriction: words must always start at addresses that are multiple of 4 in MIPS for faster data transfer
- big Endian uses the address of the leftmost or "big end" byte as the word address (e.g., MIPS), little Endian uses the rightmost or "little end" (e.g., Solaris)
- memory is slower than registers since registers are smaller, data access are faster if data is keep in registers instead of memory
- the MIPS data transfer instructions: to access a word in memory, the instruction must supply the memory address
  - format: instruction name, the target register, a base register with offset
  - load: moves data from memory to a register
  - store: transfers data from a register to memory
  - data transfer instructions: **lw** (load word), **sw** (store word)
- data structures are kept in memory (a large, single-dimensional array, with the address acting as the index to the array, starting at 0), the starting address, or base address, of the data structures can be stored in a saved register; byte addressing also affects the array index (for element of word size needs multiple the offset by 4)

    - index register: originally, the register holds an index of an array with the offset used for the starting address of the array
- many programs have more variables than machine have registers, the compiler tries to keep the most frequently used variables in registers (register allocation) and places the less commonly used variables, or those needed later, into memory (spilling registers)

Representing Instruction in the Computer

- weighted number system
    - human are taught to think in decimal (base 10), but numbers maybe represented in any base
    - numbers are kept in computer hardware as a series of high and low electronic signals (on or off, true or false, 0 or 1), and they are binary (base 2) numbers
    - two's complement numbers
    - octal and hexadecimal numbers
- a single binary digit (bit) of a number is the "atom" of computing
- the stored-program concept:
    1. instructions are represented as numbers
    2. programs can be stored in memory to be read or written just like numbers
    - memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generate the machine code
    - treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems
- binary number instructions (machine language), and a sequence of numeric instructions (machine code)
- instruction format: each of the segments of an instruction is a field
    - R-type or R-format (for register)
        - the first field (6 bits) *op* (opcode): basic operation of the instruction
        - the second field (5 bits) *rs*: the first register source operand
        - the third field (5 bits) *rt*: the second register source operand
        - the fourth field (5 bits) *rd*: the register destination operand (result of the operation)
        - the fifth field (5 bits) *shamt*: the shift amount
        - the last field (6 bits) *funct* (function code): selects the specific variant of the operation in the *op* field
    - I-type or I-format (for data transfer and conditional branch instructions)
        - the first 3 fields are *op*, *rs*, and *rt* (same as R-type, but *rs* for base register and *rt* for destination register)
        - the fourth field (16 bits) *address*: for offset from -2^{15} or -32768 to $2^{15}-1$ or 32767 bytes (-2^{13} or -8192 words) in the base register *rs*; an alternative would be to specify a register that would always be added to the branch address (program counter = register + branch address)
    - J-type or J-format (for unconditional jump instructions)
        - the first field (6 bits) *op*
        - the second field (26 bits) *address*:
- the compromise (design principle 3) chosen by MIPS designers is to keep all instruction the same length (32 bits), thereby requiring different kinds of instruction format for different kinds of instructions; although multiple formats complicate the hardware, we can reduce the complexity by keeping the format similar, and the format are distinguished by the values in the first field

Instructions for Making Decisions

- what distinguishes a computer from a simple calculator is its ability to make decision (based on the input data and the values created during the computation, different instructions are executed)
    - condition instructions: **slt** (set on less than)
    - conditional branch instructions: **beq** (branch if equal), **bne** (branch is not equal)
    - unconditional branch or jump instructions: **j** (jump), **jr** (jump register), **jal** (jump-and-link)
- use label to relieve the compiler writer or assembly language programmer from the tedium of calculating address for branches
- compilers frequently create branches and labels where they do not appear in the programming language; avoid the burden of writing explicit labels and branches is one benefit of writing in high-level programming language
- decisions are important for choosing between alternatives (ifs), and for iterating a computer computation (loops)
- a basic block is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning
- programming in high-level language does not normally write loops with *goto* statements, but does in assembly
- MIPS compilers use the **slt**, **beq**, **bne**, and the fixed value of 0 always available by reading register $zero (map to register 0) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal
- MIPS architecture doesn't include branch on less than, two faster instructions are more useful
- most programming languages have a case or switch statement that allows the programmer to select one of many alternatives depending on a single value
    - a sequence of conditional tests: turning into a chain of if-then-else statements

- jump address table: the alternatives may be efficiently encoded as a table of addresses (corresponding to labels in the code) of alternative instruction sequences, and the program needs only to index into the table (by loading the appropriate entry into a register) and then jump to the appropriate sequence using **jr**

Supporting Procedures in Computer Hardware

- structure programming: use procedures or subroutines, easier to understand, allow code to be reused, allow the programmer to concentrate on just one portion of the task at a time, allow values to be passed as parameters and return results
- the program counter (PC) or the instruction address register: a register to hold the address of the current instruction being executed
- the execution of a procedure
  1. the calling program (caller) places parameters in a place (normally $a0-$a3) where the procedure can access them,
  2. transfer control to the procedure (callee) by **jal** callee, the **jal** instruction saves PC + 4 in register $ra to link to the following instruction to set up the procedure return
  3. acquire the storage resources needed for the procedure, the callee pushes $ra and any $s0-$s7 used by the callee, the $sp is adjusted to account for the number of register placed on the stack
  4. perform the desired task, also cover it tracks
  5. restored saved registers and adjust $sp, place the result value in a place (in $v0-$v1) where the calling program can access it
  6. return control to the point of origin by **jr** $ra
- the return address is needed because the same procedure could be called from several parts of the program
- generally a compiler needs more registers than the 4 arguments registers and the 2 value registers, to cover it tracks after its mission is complete, any registers needed by the caller must be restored to the values (by spilling registers to memory) that they contained before the procedure was invoked
- the ideal data structure for spilling registers is a stack (last-in-first-out) with its operations push and pop; by historical precedent, stacks "grow" from higher addresses to lower addresses, push onto stack by subtracting from the stack pointer ($sp) and pop from stack by adding to $sp
- the stack is also used to store variables that are local to the procedure that do not fit in registers
- the MIPS compilers always save room on the stack for the arguments in case they need to be stored, so in really they always decrement $sp by 16 to make room for all 4 argument registers, when the compiler encounters the rare vararg, it copies the registers onto the stack into the reserved locations
- a procedure frame or activation record: the segment of stack containing a procedure's saved register and local variables
- the $fp (a stable base register for local memory reference because $sp is may change during program execute) points to the first word of the frame; if there are no local variables on the stack within a procedure, the compiler will save time by not setting and restoring $fp; when $fp is used, it is initialized using the address in $sp on a call, and $sp is restored by using $fp
- if there are more than 4 parameters, MIPS place the extra parameters on the stack just above $fp
- to reduce register spilling by avoiding saving and restoring a register whose value is never used (e.g., in temporary registers)
- leaf procedures are procedures that do not call others, the compiler try to allocate variables in temporary registers instead of in saved registers as many as possible to avoid saving and restoring them
- copy registers into registers is faster than saving and restoring on the stack
- recursive procedures: procedures invoke "clones" of themselves
- tail recursion can be implemented iteratively to improve performance
- automatic variables are local to a procedure and are discarded when the procedure exits
- static variables (including global variables which are outside of any function in C) exist across exits from and entries to procedure, access through $gp
- code for procedure
  1. allocate registers to program variables
  2. produce code for the body of the procedure
  3. preserve registers across the procedure invocation
- forgetting that sequential word address differ by 4 instead of by 1 is a common mistake in assembly language programming
- procedure inlining: instead of passing arguments in parameters and invoking the code with **jal** instruction, the compiler copies the code from the body of the procedure where the call appears in the code
  - disadvantages: the compiled code would be bigger, the code expansion might turn into lower performance if it increased the cache miss rate

Beyond Numbers

- use 8-bit bytes to represent characters, with the American Standard Code for Information Interchange (ASCII); upper- and lowercase letters differ by exactly 32 (leads to shortcuts in checking or changing cases)
  - non-printable or formatting characters: 0 for null (mark end of string in C strings), 7 for bell, 8 for backspace, 9 for tab, 10 for new line, 13 for carriage return
  - printable characters: 32 for space, 48 for '0', 65 for 'A', 97 for 'a'
- load byte (**lb**) loads a byte from memory, placing it in the rightmost 8 bits of a register; store byte (**sb**) takes a byte from the rightmost 8 bits of a register and writes it to memory

- 3 choices for representing strings, which have a variable number of characters
  - the first position of the string is reserved for the length of a string
  - an accompanying variable has the length of the string
  - the last position of a string is indicated by a character to mark the end of a string (as in C)
- Unicode needs 16 bits to represent a character (as in Java), and MIPS has instructions to load and store 16-bit quantities (halfwords)
- MIPS software tries to keep the stack aligned to word address, thus a character variable allocated on the stack will be allocated 4 bytes; a string variable or an array of bytes will pack 4 bytes per word

Other Styles of MIPS Addressing

- to use constant in an operation, MIPS offer versions of the arithmetic/comparison instructions in which one operand is a constant (this constant is kept inside the instruction itself using I-type, where I for immediate); the field containing the constant is 16 bits long
- add immediate (**addi**), set less than immediate (**slti**)
- constant operands occur frequently, and by making constants part of instructions, they are faster than if they were loaded from memory (design principle 4)
- although constants are frequently short and fit into the 16-bit field, sometimes they are bigger; use load upper immediate (**lui**) to set the upper 16 bits of a constant in a register (filling the lower 16 bits with 0s) by shifting, allowing a subsequent instruction to specify the lower 16 bits of the constant by either **addi** or **ori** (logical or immediate)
- either the compiler or the assembler must break large constant into pieces and then reassemble them into a register; if this job falls to the assembler, the assembler must have a temporary register available in which to create the long values; the register $at is reserved for the assembler
- since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of words to the next instruction instead of the number of bytes
- conditional branches use I-type (PC-relative addressing, but MIPS actually use PC + 4) because they are found in loops and if statements, so they tend to branch to a nearby instruction; if branches far away, insert an unconditional jump to the branch target, and the condition is inverted so that the branch decides whether to skip the jump
- unconditional instructions use J-type because they have no reason to be nearby; the 26 bits *address* field means it represent a 28-bit byte address (leaving the upper 4 bits of the PC unchanged); if jump to a target more than 256MB (64M instructions) away, use **jr**
- MIPS addressing modes (a single operation can use more than one addressing mode):
  - register addressing: the operand is a register
  - base or displacement addressing: the operand is at the memory location whose address is the sum of a register and a constant
  - immediate addressing: the operand is a constant
  - PC-relative addressing: the address is the sum of the PC and a constant
  - pseudodirect addressing: the jump address is the 26 bits of the instruction concatenated with the upper bit of the PC
- although MIPS has 32-bit address, nearly all microprocessors (including MIPS) have 64-bit address extension for large program

Starting a Program

- compiler
  - a compiler translates a high-level language program into an assembly language program
  - increasing memory capacity reduces program size concerns, and optimizing compilers produce assembly code nearly as good as (or sometimes even better than) human experts
  - some compilers produce object module directly
  - Unix suffix convention: .c for C files, .s for assembly, .o for object file, and a.out for executable; MOS-DOS uses .C .ASM, .OBJ, and .EXE
- assembler
  - an assembler translates an assembly language program into a machine language program
  - pseudo-instructions: instructions not implemented in hardware, but used in assembly language to simplify translation and programming
  - pseudo-instructions give MIPS a richer set of assembly language by reserving $at register for use by the assembler
    - **move** copies the contents of one register to another
    - **blt** (branch on less than) into **slt** and **bne** (same for **bgt**, **bge**, **ble**)
    - converts branches to faraway locations into a branch and jump
    - allow 32-bit constants to be loaded into a register
  - an object file is a combination of machine language instructions, data, and information needed to place instruction properly in memory; an Unix object file consists of the following:
  - separate compilation
    - the object file header describes the size and position of the other pieces of the object file
    - the text segment contains the machine language code
    - the data segment contains data: static data is allocated throughout the program, dynamic data can grow or shrink as needed by the program
    - the relocation information identifies instructions and data words that depend on absolute address when the program is loaded into memory

        - the symbol table contains pairs of undefined (label) symbol and address
        - the debugging information contains a concise description of how the modules were compiled so that a debugger can associate machine instruction with C source files and make data structure readable
    - disassembler: reverse-engineer machine language to create the original assembly language
- link editor or linker
    - complete recompilation: a single change to one line of one procedure requires compiling and assembling the whole program, waste computing resources (especially for standard library routines, which almost never change)
    - separate compilation: compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure
    - linker takes all the independently assembled machine language and stitch them together, it is much faster to patch code than it is to recompile and reassemble
        1. place code and data modules symbolically in memory
        2. determine the addresses of data and instruction labels
        3. patch both the internal and external references
    - the linker determines the memory locations each module will occupy, when the linker places a module in memory, all absolute reference must be relocated to reflect its true location
    - the linker produces an executable file (with the same format as an object file, except it contains no unresolved references, relocation information, symbol table, or debugging information) that can be run on a computer
        - partially linked files (e.g., library routines) still have unresolved addresses
    - MIPS memory allocation for program and data:
        - the stack pointer is initialized to 0x7fff_fffc, and grow down toward the data segment
        - the program code (text segment) starts at 0x0040_0000
        - the static data starts at 0x1000_0000, and the dynamic data starts next and grows up toward the stack
        - the global pointer ($gp) is initialized to 0x1000_8000, and access from 0x1000_0000 to 0x1000_ffff
- loader
    1. read the executable file header to determine size of the text and data segments
    2. creates an address space large enough for the text and data
    3. copies the instructions and data from the executable file into memory
    4. copies the parameters (if any) to the main program onto the stack
    5. initializes the machine registers and sets the stack pointer to the first free location
    6. jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program, when the main routine returns, the start-up routine terminates the program with an exit system call

Arrays versus Pointers

- people used to be taught to use pointers in C to get greater efficiency than available with arrays, but modern optimizing compilers can produce just as good code for the array version of the code
- the address of a variable is indicated by & and referring to the object pointed to by a pointer is indicated by *

Power PC Instructions

- more powerful operations can reduce the number of instructions executed by a program at the cost of simplicity, increasing the time a program takes to execute the slower instructions, and may not exactly match what the compiler needs to produce
- the PowerPC (made by IBM and Motorola, used in the Apple Macintosh) have 32 integer registers, instructions are all 32 bits long, and data transfer only with loads and stores; it differs the MIPS by two more addressing modes and a few operations:
    - indexed addressing: allow two registers to be added together
    - update addressing: data transfer instructions automatically increment the base register to point to the next word each time data is transferred
    - load multiple and store multiple: transfer up to 32 words of data in a single instruction for fast copies of locations in memory, also save code size when saving or restoring registers
    - a special counter register which decrements itself, compares to 0, and then branches as long as the register is not 0

The Intel 80x86

- the MIPS was the vision of a single group in 1985, but the 80x86 is the product of several independent groups who evolved the architecture over 20 years
    - Intel 4004 4-bit microprocessor and 8080 8-bit accumulator-style microprocessor (1970s)
    - Intel 8086 16-bit general-purpose register architecture (1978)
    - Intel 8087 floating-point coprocessor with about 60 floating-point instructions and using stack (1980)
    - Intel 80286 used 24-bit address space, created memory-mapping, protection mode, and added a few instructions (1982)
    - Intel 80386 32-bit architecture added paging, segmented addressing, additional instructions, backward compatible (1985)

- Intel 80486 (1989), Pentium (1992), and Pentium Pro (1995) aimed at higher performance with 4 additional instructions
- MMX SIMD (single instruction, multiple data) architecture expansion for multimedia and communication applications, with new set of 57 instructions uses the floating-point stack (1997)

- open architecture strategy and the most popular architecture (300 million in 1997)
- 80x86 registers:
  - 8 32-bit general-purpose registers (GPR): EAX (GPR 0), ECX, EDX, EBX, ESP, EBP, ESI, EDI (GPR 7), where the prefixing E indicate the 32-bit version
  - 6 16-bit special-purpose registers: code segment pointer (CS), stack segment pointer (SS), data segment pointer 0 (DS), data segment pointer 1 (ES), data segment pointer 2 (FS), data segment pointer 3 (GS)
  - 32-bit instruction pointer (PC) EIP
  - 32-bit condition code EFLAGS
- 80x86 instruction types: register-register, register-immediate, register-memory, memory-register, and memory-immediate, where immediate can be 8, 16, or 32 bits
  - the arithmetic and logical instructions must have one operand act as both a source and a destination (this puts more pressure on the limited registers)
  - one of the operands can be in memory
- 80x86 has 7 data addressing modes: (restrictions on registers used)
  - register indirect: address in a register
  - base mode: address is contents of base register plus 8- or 32-bit displacement
  - base plus scaled index: address = base + 2^{scale} * index, where scale can be 0, 1, 2, or 3 (in byte)
  - base plus scaled index with displacement: address = base 2^{scale} * index + displacement
- 80x86 integer operations
  1. data movement instructions
  2. arithmetic and logic instructions
  3. control flow: base on condition codes or flags (condition codes are set as a side effect of an operation, faster than comparing to register with extra hardware, however, programmers may test a value that is not the result of an operation)
  4. string instructions (for backward compatibility, not efficient)
- 80x86 instruction encoding
  - 80386 instruction vary from 1 byte (no operand) up to 17 bytes
  - the opcode usually contains a bit '*w*' saying whether the operand is 8 bits or 32 bits, sometimes includes the addressing mode and the register
  - to override the default data size, an 8-bit prefix is attached to the instruction; other instructions use a post byte or extra opcode byte, labeled "mode, reg, r/m," which contains the addressing mode information; the base plus scaled index mode uses a second post byte labeled "sc, index, base."
  - 80386 instruction format:
    - branch: 4-bit op, 4-bit condition, 8-bit displacement
    - call: 8-bit op, 32-bit offset
    - data transfer: 6-bit op, 1-bit *d* (for direction), 1-bit *w* (for width), 8-bit post byte (r/m), 8-bit displacement
    - stack operation: 5-bit op, 3-bit register
    - arithmetic operation: 4-bit op, 3-bit register, 1-bit *w*, 32-bit immediate
    - test: 7-bit op, 1-bit w, 8-bit post byte, 32-bit immediate
- 80x86 is more difficult to build than MIPS, but the much larger market means Intel can afford more resources to help overcome the added complexity

Fallacies and Pitfalls

- ❖ fallacy: more powerful instructions mean higher performance
- ❖ write in assembly language to obtain the highest performance
  - C register data type to give a hint to the compiler about which variables should be kept in register versus spilled to memory, but today's compiler ignore such hints because it does a better job at allocation than the programmer
  - even if writing by hand resulted in faster code, it takes longer time to code and to debug, lost portability; coding takes longer than expected
  - writing in high-level language not only allow future compiler to tailor the code to future machine, it also makes the software easier to maintain
  - ✓ the compiler generally was able to create assembly language code that was tailored exactly to those conditions, while the assembly language program was written in a slightly more general fashion to make it easier to modify and understand
- ❖ pitfall: forgetting that sequential word address in machines with byte addressing do not differ by one
- ❖ pitfall: using a pointer to an automatic variable outside its defining procedure

Historical Perspective

- accumulator architectures
  - all operations accumulate in a single register, the accumulator
  - have the memory-based operand-addressing mode
  - no registers are specified and variables are always spilled to memory, it takes many more instructions to execute a program

    - extended accumulator, dedicated register, or special-purpose register: the addition of registers dedicated to specific operations
- general-purpose register architectures
    - all the registers to be used for any purpose
    - register-memory architectures: allow one operand in memory
    - load-store or register-register architectures: operand always in registers
    - memory-memory architectures: all operands can be in memory
- stack architectures
    - simplify compilers by eliminating register allocation
    - remove memory size as an excuse not to program in high-level languages
    - compact instruction encoding, good for network computer (NC) and Java based on a stack
- high-level-language computer architectures: make the hardware more like the programming languages, e.g., Burroughs B5000

The Single Instruction Computer

- a hypothetical machine SIC (single instruction computer) has only one instruction: **sbn** (subtract and branch if negative), and it has no registers
- SIC can imitate many of the operations of more complex instruction sets by using cleaver sequences of **sbn** instructions