Computer Architecture

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr
nizamettinaydin@gmail.com

http://www.yildiz.edu.tr/~naydin

MIPS Address Modes and Translating Programs

Outline

- MIPS Adressing Modes
  - Immediate Addressing
  - Register Addressing
  - Base Addressing
  - PC-relative Addressing
  - Pseudodirect Addressing
  - MIPS Instruc-on Formats – Summary
  - Other ISAs
- Translating Programs
  - C Code Translation Hierarchy
  - Compiler Flags - Example
  - Assembly Language
  - Object File
  - Executable
  - Loading a Program
  - Dynamic Linking
  - Java Code Transla-on Hierarchy

Immediate Addressing

- The operand is a constant within the instruction itself

op	rs	rt	immediate

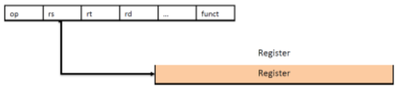
- Example

  add $s1, $s2, 200 ➔ $s1= $s2 + 200

8	18	17	200

Register Addressing

- The operand is in a register



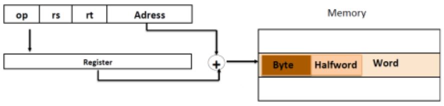
- Example

  add $s1, $s2, $s3 ➔ $s1= $s2 + $s3

0	18	19	17	0	32

Base Addressing

- The operand is in the memory location whose address is the sum of a register and a constant in the instruction



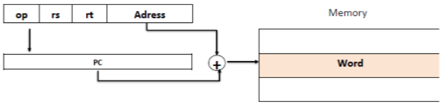
- Example

  lw $s1, 200($s2) ➔ $s1= mem[200 + $s2]

35	18	17	200

PC-relative Addressing

- The address is the sum of the PC and a constant in the instruction
  - I-Type instruction



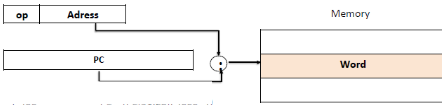
- Example

  beq $s1, $s2 , 200 ➔ if ($s1 ==$s2) PC= PC+4+200*4

4	18	17	200

Pseudodirect Addressing

- the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC
  - J-Type instruction



- Example

  j 4000 ➔ PC = (PC[31:28], 4000*4)

2	4000

MIPS Instruction Formats - Summary

R-type	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
	opcode	rs	rt	rd	shamt	funct

Arithmetic: Register[rd] = Register[rs] + Register [rt]

Register indirect jumps: PC = PC + Register[rs]

I-type	6 bits	5 bits	5 bits	16 bits
	opcode	rs	rt	immediate

Arithmetic: Register[rt] = Register[rs] + Immediate

Branches: If Register[rs] == Register[rt], goto PC + Immediate

Memory: Memory[Register[rs] + Immediate] = Register[rt]

Register[rt] = Memory[Register[rs] + Immediate]

J-type	6 bits	26 bits
	opcode	Address

Direct jumps: PC = Address, Syscalls, breaks etc

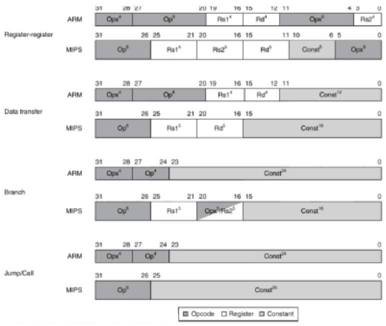
Uniformity and Compiler Friendliness in MIPS

- 3 instruction formats: I, R and J
  - R-type:
    - Register-register arithmetic
  - I-type:
    - immediate arithmetic, load/stores, conditional branches
  - J-type:
    - Jumps, non-conditional branches
  - Opcodes are always in the same place
  - rs and rt are always in the same place, as is rd if it exists
  - The immediate is always in the same place
- Similar amounts of work per instruction
  - 1 read from instruction memory
  - <=1 arithmetic operations
  - <=2 register reads
  - <=1 register write
  - <= 1 data load/store
- Fixed instruction length

Other ISAs

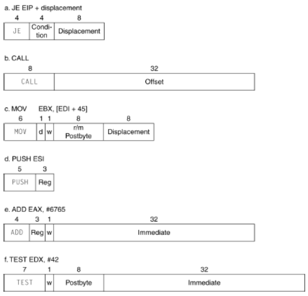
- CISC (Complex Instruction Set Computing)
  - Combine memory and computation operations into a single operation
  - Multiple-step operations
  - Examples: System/360, Motorola 68k, x86, VAX
  - Dominant in desktops and servers
- RISC (Reduced Instruction Set Computing)
  - a.k.a load/store architecture (memory accesses are not part of an arithmetic instruction)
    - Arithmetic instructions just operate on registers
  - Examples: MIPS, ARM, PowerPC, SPARC
  - Dominant in cell phones, embedded systems

ARM and MIPS Instruction Encoding



Register-register
ARM
31 28 27 20 19 16 15 12 11 4 3 0
Opx^4 Op^8 $Rs1^4$ Rd^4 Opx^8 $Rs2^4$
MIPS
31 26 25 21 20 16 15 11 10 6 5 0
Op^6 $Rs1^5$ $Rs2^5$ Rd^5 $Const^5$ Opx^6
Data transfer
ARM
31 28 27 20 19 16 15 12 11 0
Opx^4 Op^8 $Rs1^4$ Rd^4 $Const^{12}$
MIPS
31 26 25 21 20 16 15 0
Op^6 $Rs1^5$ Rd^5 $Const^{16}$
Branch
ARM
31 28 27 24 23 0
Opx^4 Op^4 $Const^{24}$
MIPS
31 26 25 21 20 16 15 0
Op^6 $Rs1^5$ $Opx^5/Rs2^5$ $Const^{16}$
Jump/Call
ARM
31 28 27 24 23 0
Opx^4 Op^4 $Const^{24}$
MIPS
31 26 25 0
Op^6 $Const^{26}$
▨ Opcode □ Register ▨ Constant

x86 Instruction Encoding

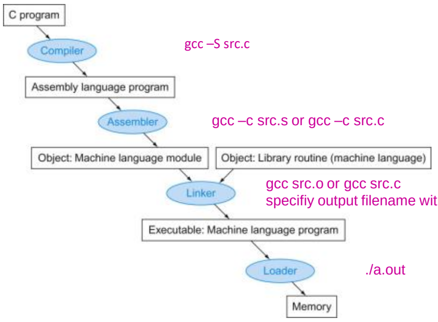


- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …
- Intel and AMD use x86

Concluding Remarks on ISA

- Design principles for a good ISA
  - Simplicity favors regularity
  - Smaller is faster
  - Make the common case fast
  - Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, linker, hardware
- MIPS: typical of RISC ISAs
- x86 typical of CISC ISAs

C Code Translation Hierarchy


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

gcc –S src.c

gcc –c src.s or gcc –c src.c

gcc src.o or gcc src.c
specifiy output filename with -o

./a.out

Compiler Flags - Example

- gcc –S src.c #creates .s file
  - Generates assembly code (compile only)
- gcc –c src.s #creates .o file
  - Generates object file (compile and assemble)
- gcc src.o #creates a.out file
  - Generates an executable from object files
- gcc sum.o -o sum.exe #creates an executable sum.exe
  - Generates an executable with a specified output name
- gcc src.c #creates an executable by default name a.out
  - Compile, assemble and link

Assembly Language

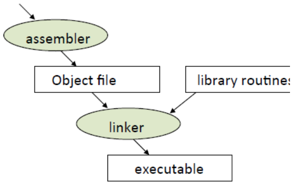
- Assembly language is the symbolic representation of a computer’s binary encoding, which is called machine language.
- Assembly language is more readable than machine language because it uses symbols instead of bits.
- Assembly language permits programmers to use labels to identify and name particular memory words that hold instructions or data.
- A tool called assembler translates assembly language into binary instructions.
- An assembler reads a single assembly language source file and produces object file containing machine instructions and bookkeeping information that helps combine several object files into a program.

Object File

- Not directly executable
- Contains object code (relocatable format machine code)
- Input to the linker
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Files

- Link editor or linker links the object files
- Linker takes all the independently assembled object files and stitches them together
  - Resolves all the undefined labels into an executable file
- Relocation



- – Linker merges object files and assigns runtime addresses to each symbol and section
- As a result, instructions and data will have unique runtime addresses
- Output is an executable file

Executable

- Its format is similar to an object file
  - But contains almost no unresolved references
- It can contain symbol tables and debugging information and partially linked files, such as library routines, that still have unresolved addresses.
  - Might need to do another relocation at the execution time
- Loader executes the executable

Loading a Program

- A loader is the part of an operating system
- Load from image file (executable) on disk into memory
  - Read header to determine size of the text and data segments
  - Create an address space for the segments
  - Copy text and data from the executable file into memory
  - Set up arguments (if any) on stack
  - Initialize registers (including $sp, $fp, $gp)
  - Jump to startup routine
- Copies arguments to $a0, … and calls ”main”
  - Note that loader is the caller and ”main” is the callee
- When main returns, program exits

Dynamic Linking

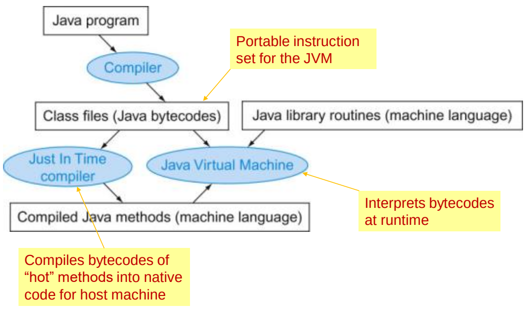
- Static linking is fast
  - The application can be certain that all its libraries are present with static libraries
  - Static linking will result in a significant performance improvement
  - Static linking can also allow the application to be contained in a single executable file, simplifying distribution and installation.
- Dynamically link/load library when it is called
  - Requires procedure code to be relocatable
  - Avoids large image files caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

What time?

- Compile Time
- Link Time
- Load Time
- Runtime (Execution Time)

- The source of the error and error messages differ.

Java Code Translation Hierarchy


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)

Portable instruction set for the JVM

Interprets bytecodes at runtime

Compiles bytecodes of “hot” methods into native code for host machine