

2. Instruction Set Architecture – Storage types and addressing modes

EECS 370 – Introduction to Computer Organization – Winter 2015

Robert Dick, Andrew Lukefahr, and Satish Narayanasamy

**EECS Department
University of Michigan in Ann Arbor, USA**

© Dick-Lukefahr-Narayanasamy, 2015

The material in this presentation cannot be
copied in any form without our written permission

Announcements

- ❑ Project 1 assigned
 - Project 1.a due Wed, 1/28
 - Project 1.s due Wed 2/4

- ❑ Homework 1 assigned
 - Due Thurs, 1/22

Last time

- ❑ History of computers

- ❑ Moore's law – and some of the issues arising today:
 - Current technology: 32nm → very few atoms in each transistor's gate
 - Parametric variations
 - Leakage power
 - Power density (intel i7-980 burns 130W!)



Instruction Set Architecture (ISA) Design Lectures

- ❑ **Lecture 2: ISA - storage types and addressing modes**
- ❑ Lecture 3 : LC-2K and ARM architecture
- ❑ Lecture 4 : Converting C to assembly – basic blocks
- ❑ Lecture 5 : Converting C to assembly – functions
- ❑ Lecture 6 : Translation software; libraries, memory layout

Where do ISAs come into the game ?

Application software

Compilers

ARCHITECTURE – a.k.a. ISA (Instruction Set Architecture)

- Platform-specific
- a limited set of assembly language commands "understood" by hardware (e.g. ADD, LOAD, STORE, RET)

The hw/sw
divide

MICROARCHITECTURE (Hardware implementation of the ISA)

- Core i7 implements the x86 ISA
- OMAP 4430 implements the ARM ISA

Circuits

Devices

Instruction Set Design – freedom space (1/2)

❑ What instructions should be included?

- add, multiply, divide, sqrt [functions]
- branch [flow control]
- load/store [storage management]
- mmx_add

❑ What storage locations?

- How many registers?
- How much memory?
- Any other “architected” storage?

❑ How should instructions be formatted?

- 0, 1, 2 or more operands?
- Immediate operands

Instruction Set Design – freedom space (2/2)

□ How to encode instructions?

- **RISC** (Reduced Instruction Set Computer):
all instructions are same length (e.g. ARM, LC2K)
- **CISC** (Complex Instruction Set Computer): instructions can vary in size (VAX, x86)

□ What instructions can access memory?

- For ARM and LC2K, only loads and stores can access memory (called a “load-store architecture”)

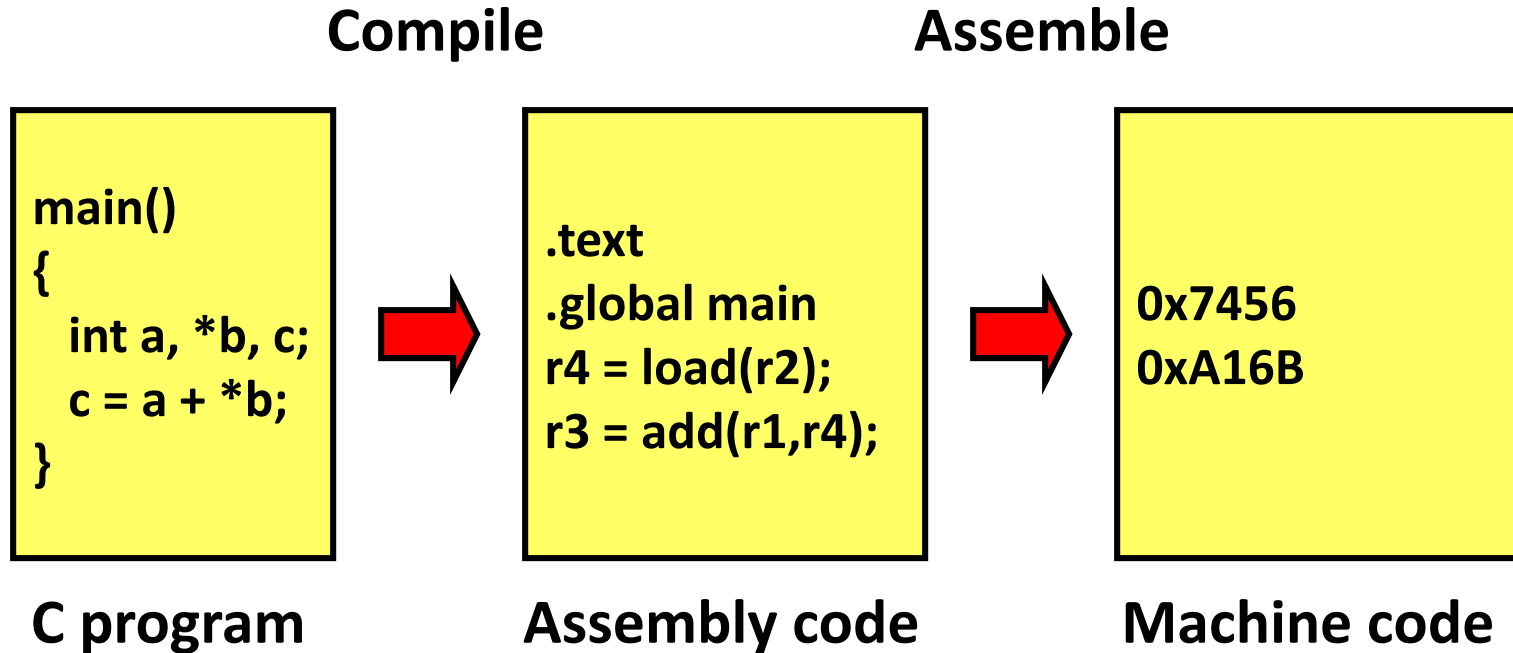
Why study Instruction Set Design?

- ❑ Isn't there only one?
 - No, and even if there were, it would be too messy for a first course in computer architecture

- ❑ How often are new architectures created?
 - Embedded processors are designed all the time
 - Even the Intel x86 ISA changes (MMX, MMX2, SSE, SSE2)

- ❑ Will I ever get to (have to) design one?
 - Very possible...

Software program to machine code



Assembly Code

□ Fields

- *Opcode* – What instruction to perform
- *Source* (input) operand specifier(s)
- *Destination* (output) operand specifier(s)
 - What data to perform operation on

opcode	src1	src2	dest
add	R2	100	R1

Translation: **value in r1 = contents of r2 + constant 100**

Assembly Code characteristics

- Generally 1-1 correspondence with machine language
- Mnemonic codes facilitate programming
- Labels (symbolic names)
- Direct control of the what processor does
- May execute fast, if you're good at it, but compilers can typically generate better code
- Still hard to use and not portable

Assembly Code – ARM Example

What are the contents of the registers after executing the given assembly code?

opcode d s1 s2imm
Program: **add r3, r1, r2**
mul r3, r3, 3
sub r2, r3, r1

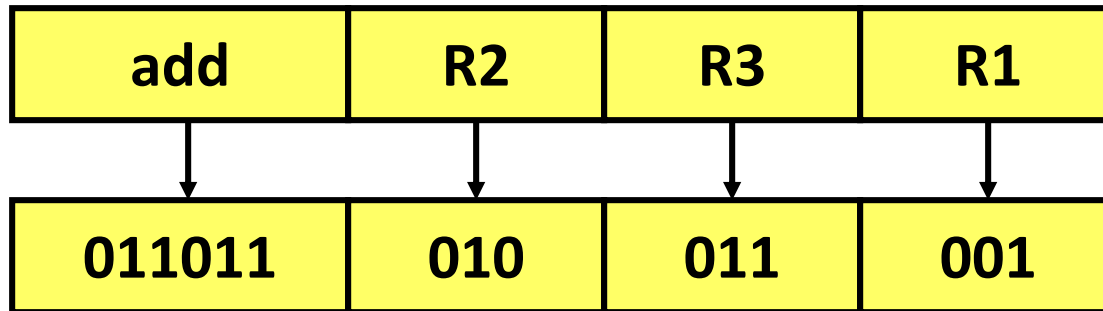
Initial register file:

r1	25
r2	-4
r3	57

	(1) add r3, r1, r2	(2) mul r3, r3, 3	(3) sub r2, r3, r1
r1	25	25	25
r2	-4	-4	38
r3	21	63	63

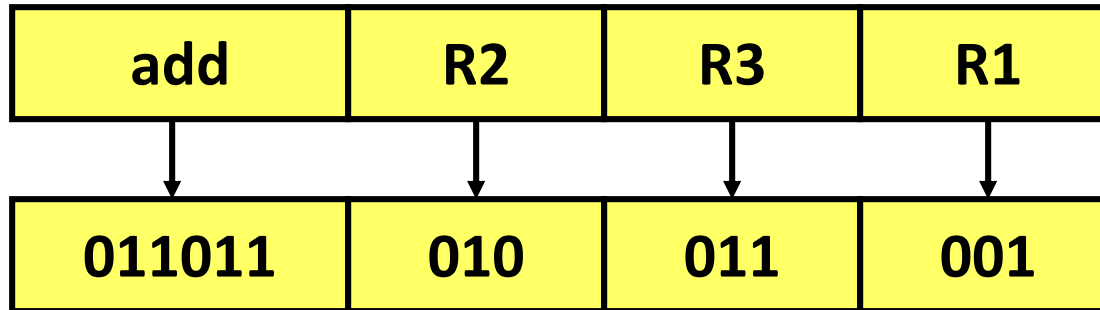
Assembly Instruction Encoding

- ❑ Since the EDSAC (1949) almost all computers stored program instructions the same way they store data.
- ❑ Each instruction is encoded as a number



$$\begin{aligned} 011011010011001 &= 2^0 + 2^3 + 2^4 + 2^7 + 2^9 + 2^{10} + 2^{12} + 2^{13} \\ &= 13977 \end{aligned}$$

Instruction Encoding (cont' d)

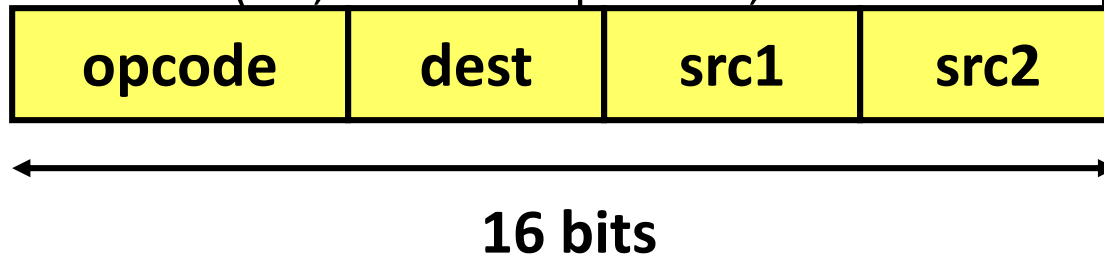


- ❑ m bits can encode 2^m different values
- ❑ n values can be encoded in $\lceil \log_2(n) \rceil$ bits
- ❑ For above
 - Can encode $2^6 = 64$ opcodes
 - Can encode $2^3 = 8$ src/destination registers
- ❑ How can we encode immediate operations, e.g., “add r2, r3, #4”?

Instruction Encoding - Example

What is the max number of registers that can be designed in a machine given:

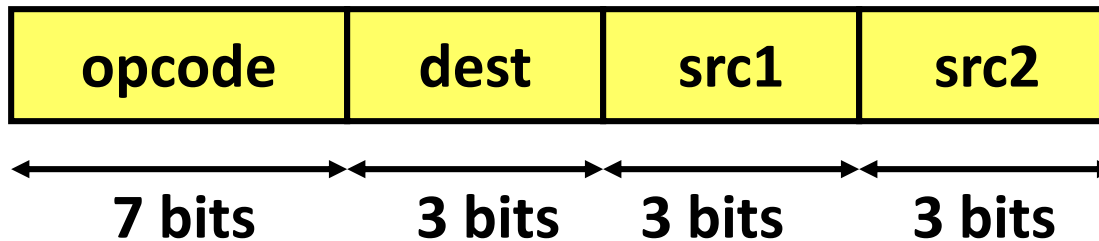
- * 16-bit instructions
- * Num. opcodes = 100
- * All instructions are (reg, reg) \rightarrow reg
(i.e., 2 source operands, 1 destination operand)



1. num opcode bits = $\lceil \log_2(100) \rceil = 7$
2. num bits for operands = $16 - 7 = 9$
3. num bits per operand = $9 / 3 = 3$
4. maximum number of registers = $2^3 = 8$

Class Problem

Given the following:



- **add** opcode is encoded as the number 53
- registers encoded with their register number

What is the encoding for **add R2, R3, R4** ?

(R2 is dest, R3 is src1, and R4 is src2)

- specify answer in binary
- specify answer in hex
- specify answer as an integer

Storage Architecture

1. Immediate Values
 - Specifying constants in instructions
2. Registers
 - Fast and small (and useful)
3. Memory
 - Big and complex (and useful)
4. Strange Storage
 - Failed ideas, old habits, and new research

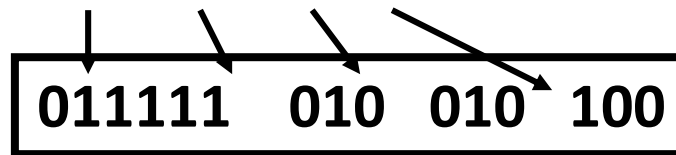
1. Immediate Values

= small constant values placed in instructions

❑ They are stored in memory only because all instructions are in memory

❑ Useful for loading small constants

- Example: `ptr++;` → `add R2, R2, #4`



- Very useful for branch instructions

→ target address is often immediate – in the instruction

❑ Size of the immediate is usually determined by how many bits are left in the instruction format.

2. Register Storage

- ❑ First came the **accumulator**, a single register architecture
 - Example: add #5
 - You don't need to specify which register when you only have one!

- ❑ Register File
 - Small array of **memory-like** storage
 - Register access is faster than memory because register file arrays are small and can be put right next to the functional units in the processor.

- ❑ Each register in the register file has a specific size
 - e.g. 32-bit registers

Example Architectures

❑ ARM

- 16 registers (r0 – r15)
- 32 bits in each register
- r15 is the program counter, it can be read and written (!)

❑ Intel x86

- 4 general purpose registers (eax, ebx, ecx, edx) 32 bits
 - You can treat them as two 8 or one 16-bits as well (ah, al, and ax)
- Special registers: 3 pointer registers (si, di, ip), 4 segment (cs, ds, ss, es),
2 stack (sp, bp), status register (flags)

❑ LC2K (the architecture you will be simulating)

- 8 registers, 32 bits each

Special Purpose Registers

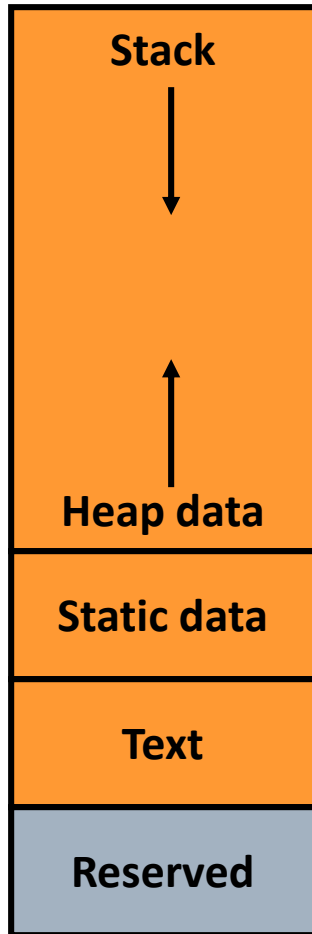
- ❑ Stack pointer (x86 ESP register, ARM register r13)
 - Holds the memory address of the “stack”
- ❑ Global pointer
 - Holds the memory address of the start of static data
 - Use in in SPARC and MIPS architectures
- ❑ Status register (x86 EFLAGS register, ARM PSR register)
 - Status bits set by various instructions
 - Compare, add, etc.
 - Status bits used by other instructions
 - Conditional branches
- ❑ 0 value register
 - No storage, reads always return 0 (lots of uses – ex: mov→add)
 - Implemented in SPARC and MIPS architectures
- ❑ Program counter (ARM register r15)
 - Cannot be accessed directly in most architectures

3. Memory Storage

- ❑ Large array of storage accessed using memory addresses
 - A machine with a 32 bit address can reference memory locations 0 to $2^{32}-1$ (or 4,294,967,295).
 - A machine with a 64 bit address can reference memory locations 0 to $2^{64}-1$ (or 18,446,744,073,709,551,615).

- ❑ Lots of different ways to calculate the address.

Memory architecture: The ARM (Linux) Memory Image



Activation records: local variables, parameters, etc.

Dynamically allocated data (new or malloc())

Global data and static local data

Machine code instructions (and some constants)

Reserved for operating system

Addressing Modes

- ❑ Direct addressing
- ❑ Indirect addressing
- ❑ Register indirect
- ❑ Base + displacement
- ❑ PC-relative
- ❑ Strange addressing modes that made it into someone's processor – and are interesting enough or ridiculous enough to talk about.

Direct Addressing

❑ Like register addressing

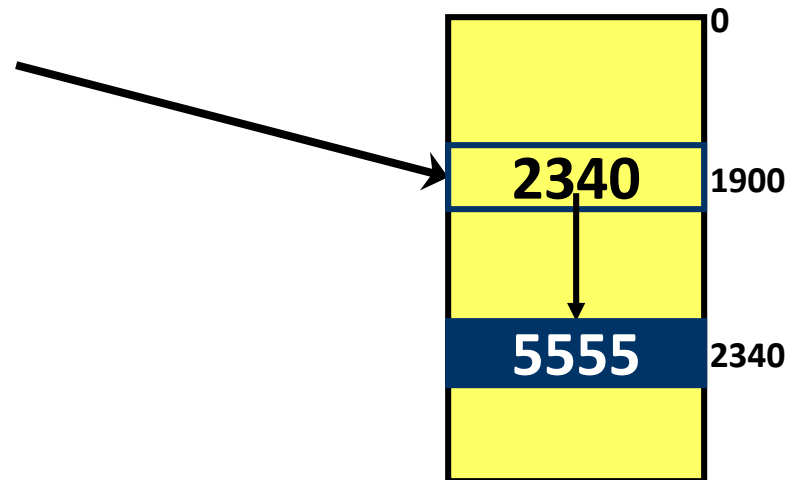
- Specify address as immediate constant
- `load r1, M[1500] ; r1 ← contents of location 1500`
- `jump M[3000] ; jump to address 3000`

❑ Useful for addressing locations that don't change during execution

- Branch target addresses
- Global/static variable locations

Indirect Addressing

- ❑ Compute the reference address by
 1. Specifying an immediate address
 2. Loading the value at that immediate address
 3. Using that value as the reference address
- ❑ `load r1, M[M[1900]]`



Register indirect

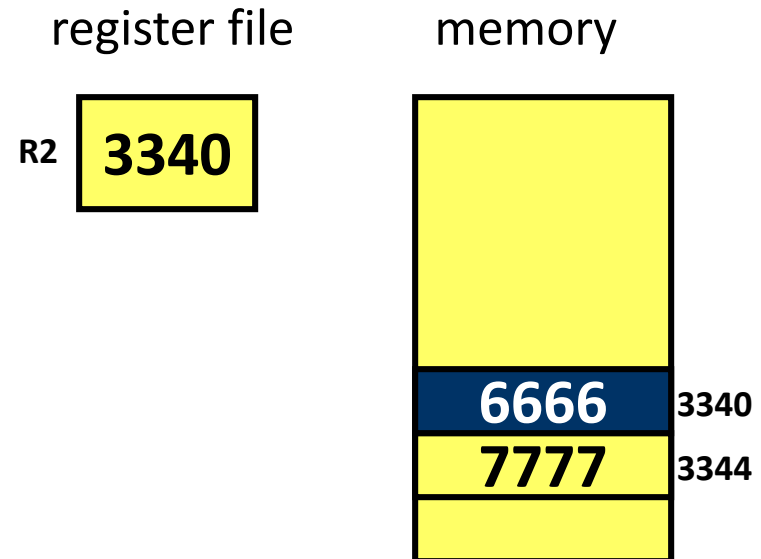
- Specify which register has the reference address

- Very useful for pointers

➔ `load r1, M[r2]`

`add r2, r2, #4`

`load r1, M[r2]`



Register indirect

□ Specify which register has the reference address

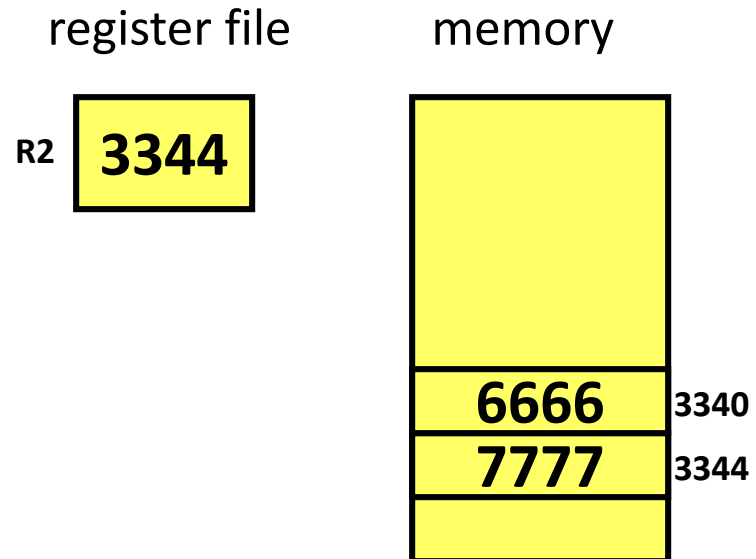
- Very useful for pointers

load r1, M[r2]



add r2, r2, #4

load r1, M[r2]



Register indirect

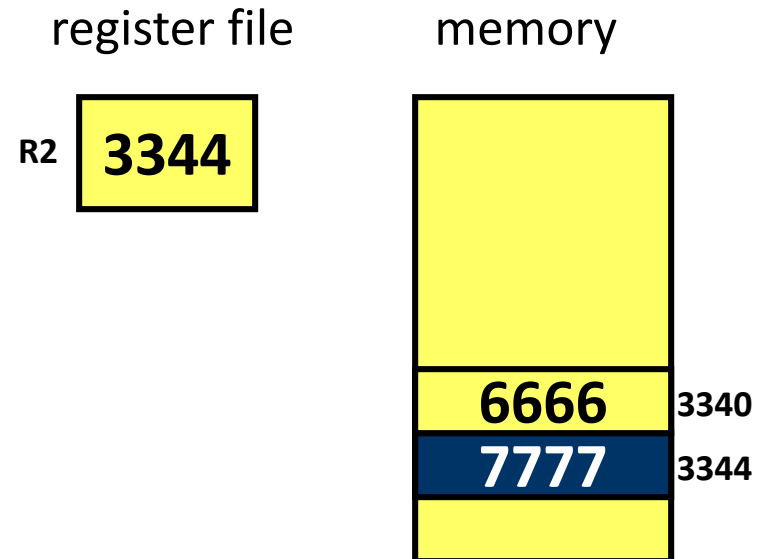
□ Specify which register has the reference address

- Very useful for pointers

load r1, M[r2]

add r2, r2, #4

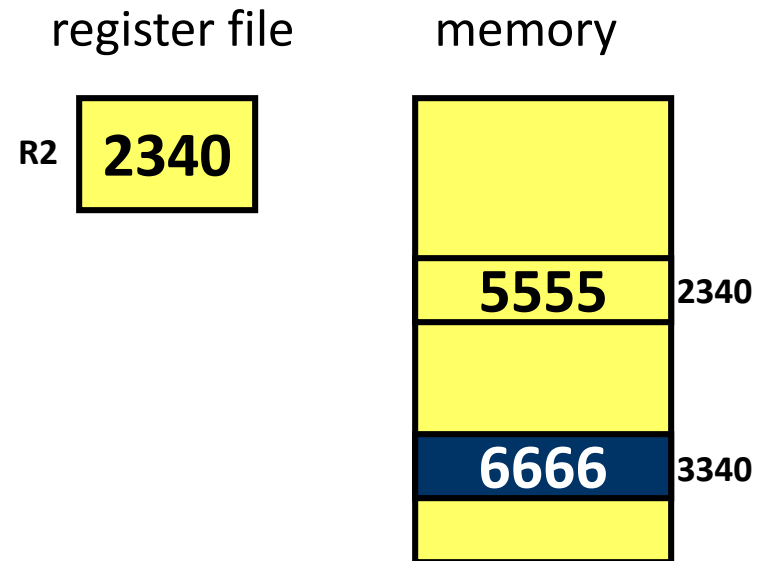
➡ *load r1, M[r2]*



Base + Displacement

- ❑ Most common addressing mode today
- ❑ Compute address as: reg value + immed
- ❑ load r1, M [r2 + 1000]
- ❑ Good for accessing class objects/structures. Why?

a.tot += a.val;



Class Problem

- a. What are the contents of register/memory after executing the following instructions

`r2 = load M[r3]`

`r3 = load M[r2+4]`

`store M[r2+8], r3`

register file		memory	
R1	0	108	100
R2	10	-1	104
R3	108	100	108

- b. How can base + displacement be made to simulate indirect addressing??
(Hint: requires 2 instructions)

PC-relative addressing

- ❑ Variant on base + displacement
- ❑ PC register is base, longer displacement possible since PC is assumed implicitly
 - Used for branch instructions
 - `jump [- 8] ; jump back 2 instructions`
- ❑ Leave it to the assembler to determine the immediate value, why? Why not the linker?

Other Addressing Modes

❑ Double indirect

- `load r1, M [M [M[1900]]]`

❑ Tagged indirect

- Extra bit (high order?) determines when you have reached the end; otherwise 1 more indirection taken.
 - Used on IBM mainframes (why?)

❑ Auto-increment

- `load r1, M[r2++]`
- Implemented in ARM ISA
- Often useful (why?)

Strange Storage

❑ Memory mapped I/O

- Sometimes I/O devices look like memory to the processor/OS (examples: graphic card, ethernet, etc.)

Vanderpool technology

- Intel's Virtualization technology – similar effect as VMware
- Allows multiple OS to run simultaneously on one processor
- Need to intercept memory accesses and translate them
- Released by Intel in 2005
- Led by Rich Uhlig, UofM alum '95 (and EECS 370 alum!)

Programming Assignment #1

- ❑ Write an assembler to convert input (assembly language program) to output (machine code version of program).
- ❑ Write a behavioral simulator to run the machine code version of the program (printing the contents of the registers and memory after each instruction executes).
- ❑ Write an efficient LC2K assembly language program to multiply two numbers.

Programming Assignment #1

□ Where to start...

- Write some test cases to check your code
 - Program 1: halt
 - Program 2: noop
 - halt
 - Program 3: add 1 1 1
 - halt
 - Program 4: nand 1 1 1
 - halt