

2. Instruction Set Architecture – Storage types and addressing modes

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

EECS Department
University of Michigan, Ann Arbor, USA

© T. Austin & M. Papaefthymiou, 2009

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

Announcements

- ❑ Homework 1 is out – due Jan. 27
 - Turn in in class – beginning of lecture
 - Available on the course webpage

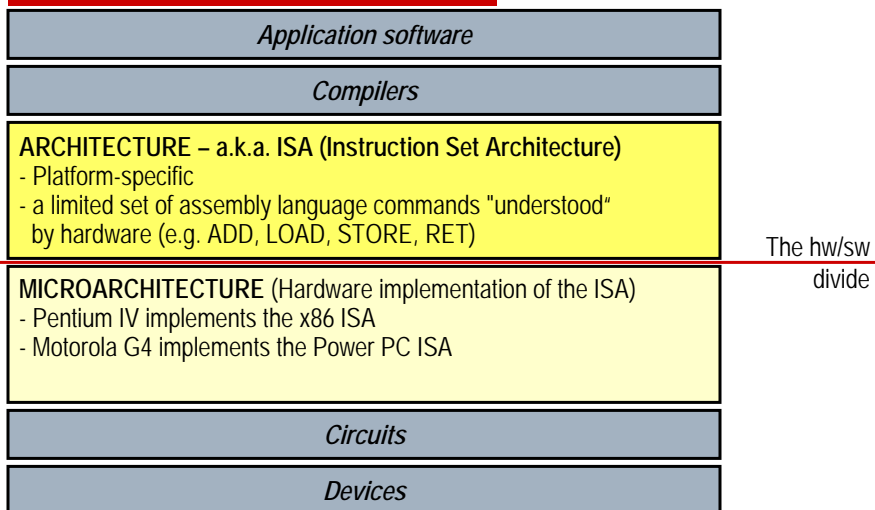
- ❑ Programming Assignment 1 is out – due Feb. 6

- ❑ First discussion session – Friday Jan. 16

Instruction Set Architecture (ISA) Design Lectures

- ❑ **Lecture 2: Storage types and addressing modes**
- ❑ Lecture 3: MIPS architecture
- ❑ Lecture 4: Calling functions / passing arguments
- ❑ Lecture 5: Translation software; libraries

Where do ISAs come into the game ?



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. MIPS, LC-2Kx)

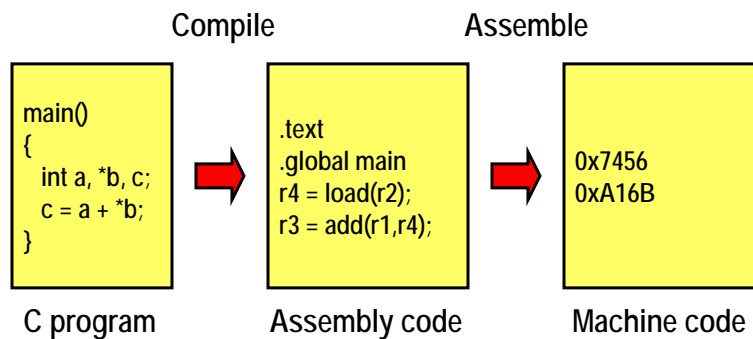
 - CISC (Complex Instruction Set Computer): instructions can vary in size
(VAX, x86)

- ❑ What instructions can access memory?
 - For MIPS and LC-2Kx, only loads and stores can access memory (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 Pentium line changes (MMX, MMX2)
- ❑ Will I ever get to (have to) design one?
 - Very possible...

Software program to machine code



Assembly Code

□ A.k.a. register-transfer language (RTL)

□ 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 - Example

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

Program: *opcode s s d*
add r1, r2, r3
mpy r3, 3, r3
sub r3, r1, r2

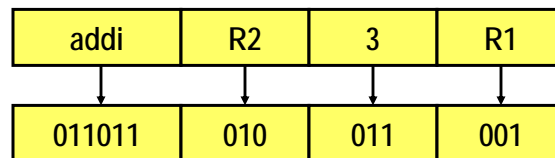
Initial register file:

r1	25
r2	-4
r3	57

	(1) add r1, r2, r3	(2) mpy r3, 3, r3	(3) sub r3, r1, r2
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

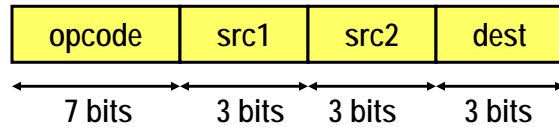


$$011011010011001 = 2^0 + 2^3 + 2^4 + 2^7 + 2^9 + 2^{10} + 2^{12} + 2^{14}$$

$$= 22169$$

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** ?

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

Storage Architecture

- Immediate Values
 - Specifying constants in instructions
- Registers
 - Fast and small (and useful)
- Memory
 - Big and complex (and useful)
- Strange Storage
 - Failed ideas and new research

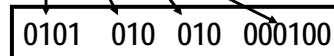
1. Immediate Values

= 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++;` → `addi 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

- ❑ MIPS
 - 32 registers (\$0 - \$31)
 - 32 bits in each register (called a word in MIPS-speak)

- ❑ 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)

- ❑ LC-2K9 (the architecture you will be simulating)
 - 8 registers, 32 bits each

Special Purpose Registers

- ❑ Stack pointer
 - Holds the memory address of the “stack”

- ❑ Global pointer
 - Holds the memory address of the start of static data

- ❑ Status register
 - Status bits set by various instructions
 - Compare, add, etc.
 - Status bits used by other instructions
 - Conditional branches

- ❑ 0 value register (MIPS \$0)
 - No storage, reads always return 0 (lots of uses – ex: mov→add)

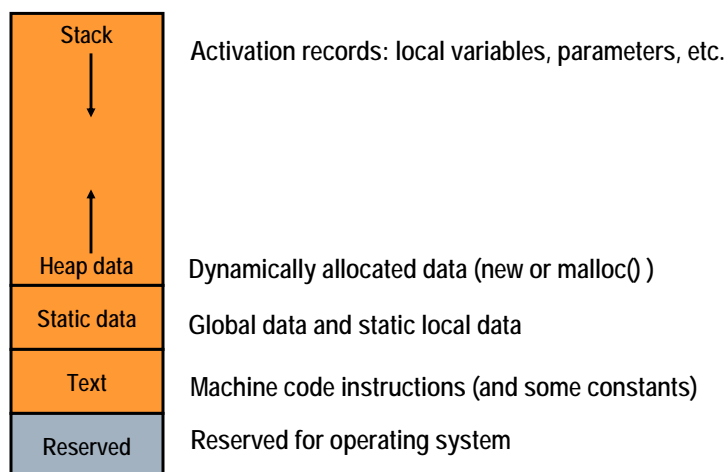
- ❑ Program Counter
 - Cannot be accessed directly

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 MIPS Memory Image



Addressing Modes

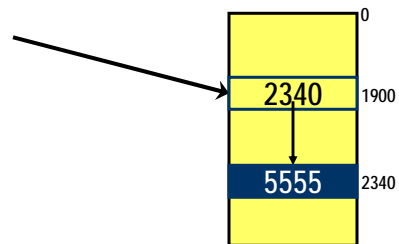
- ❑ Direct addressing
- ❑ Indirect addressing
- ❑ Register indirect
- ❑ Base + displacement
- ❑ PC-relative
- ❑ Strange addressing modes that made it into someone's processor – and are interesting enough or ridiculous enough to talk about.

Direct Addressing

- ❑ Like register addressing
 - Specify address as immediate constant
 - `load r1, M[1500] ; r1 ← contents of location 1500`
 - `jump M[3000] ; fetch instructions from addr 3000`
- ❑ Useful for addressing locations that don't change during execution.
 - Branch target addresses
 - Global/static variable locations

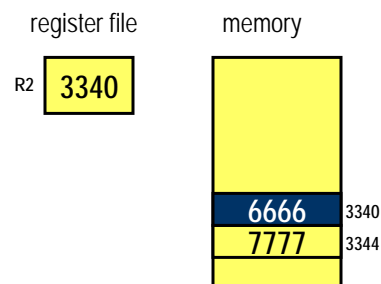
Indirect Addressing

- ❑ 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

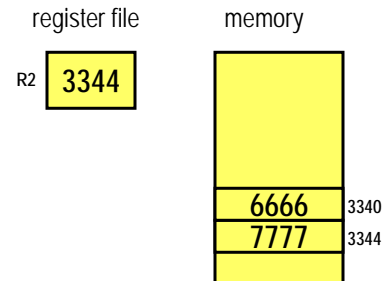
- Specify which register has the reference address

- Very useful for pointers

load r1, M[r2]

→ *add r2, r2, #4*

load r1, M[r2]



Register indirect

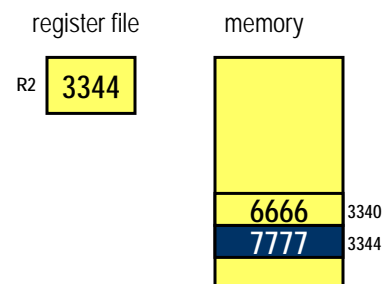
- Specify which register has the reference address

- Very useful for pointers

load r1, M[r2]

add r2, r2, #4

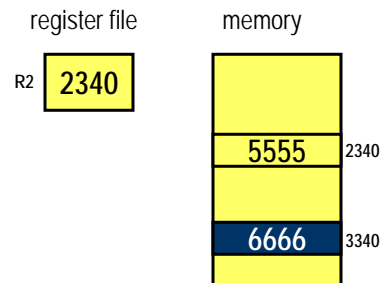
→ *load r1, M[r2]*



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?

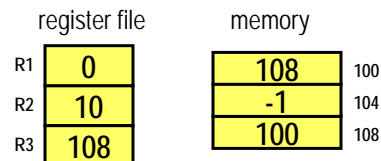
```
a.tot += a.val;
```



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
```



- 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++]`
 - Often useful (why??)

Strange Storage (Vanderpool technology)

- ❑ 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 I/O memory accesses and filter them
- ❑ Released by Intel in 2005
- ❑ Led by Rich Uhlig, UofM alum '95

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 LC-2Kx 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