CS61C
Instruction Representation and Machine Language

Lecture 5

February 3, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html

Review 1/2

- Functions, procedures one of main ways to give a program structure, reuse code
- `jr` required instruction; add `jal` to make common case fast
- Registers make programs fast, but make procedure/function call/return tricky
- MIPS SW convention divides registers into those calling procedure save/restore and those called procedure save/restore
  - Assigns registers to arguments, return address, return value, stack pointer

Review 2/2

- MIPS assembly language instructions
  - Arithmetic: `add`, `sub`, `addi`, `subi`
  - Data transfer: `lw`, `sw`
  - Relative test: `slt`, `slti`
  - Conditional branches: `beq`, `bne`
  - Unconditional branches: `j`, `jr`, `jal`
- Operands
  - Registers (word = 32 bits): `$zero`; `$v0, $v1, $a0-$a3,` `$s0-$s7`, `$t0-$t9`, `$gp, $sp, $fp, $ra`
  - Memory (8-bit byte address, 4 bytes/word): `Memory[0]`, `Memory[4]`, `Memory[8]`, `, ... ,Memory[4294967292]`

Overview

- Instruction Fields, Formats (5 minutes)
- Machine Language Examples (5 minutes)
- Big Idea: Stored program concept, instructions are just data
- Assembler extending Machine Language
- Administrivia,"What's this stuff good for"
- Addressing in branches, other Addressing Modes
- Decoding Instructions from numbers
- Conclusion (1 minute)

Instructions as Numbers

- Hardware stores information as series of high and low electrical signals
- Binary numbers convenient representation
- Must convert Assembly Language Instructions into binary numbers
- Also means turn register names into numbers

Register Names as Numbers (page A-23)

Register Names	Register No.
$zero	$0
$at (reserved for assembler)	$1
(Return) Value registers ($v0,$v1)	$2 - $3
Argument registers ($a0-$a3)	$4 - $7
Temporary registers ($t0-$t7)	$8 - $15
Saved registers ($s0-$s7)	$16 - $22
Temporary registers ($t8-$t9)	$23 - $24
$k0,$k1 (reserved for OS kernel)	$26, $27
Global Pointer ($gp)	$28
Stack Pointer ($sp)	$29
Frame Pointer ($fp), or $t10	$30
Return Address ($ra)	$31

Instruction as Number Example (decimal)

- C code: `i = j + k; /* i-k:$s0-$s2 */`
- Assembly: `add $s0,$s1,$s2 #s0=s1+s2`
- Decimal representation:

0	17	18	16	0	32

  - Segments called <u>fields</u>
  - 1st and last tell MIPS computer to add
  - 2nd is 1st source operand (17 = $s1)
  - 3rd is 2nd source operand (18 = $s2)
  - 4th is destination operand (16 = $s0)
  - 5th unused, so set to 0

Order differs: desti- nation 1st v.last! (common error)

Numbers: Review

- Number Base B => B symbols per digit:
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    Base 2 (Binary): 0, 1
- Number representation: $d_4d_3d_2d_1d_0$
  - $d_4 \times B^4 + d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$
  - $10010_{ten} = 1\text{x}10^4 + 0\text{x}10^3 + 0\text{x}10^2 + 1\text{x}10^1 + 0\text{x}10^0$
    $= 1\text{x}10000 + 0\text{x}1000 + 0\text{x}100 + 1\text{x}10 + 0\text{x}1$
    $= 10000 + 0 + 0 + 10 + 0$
    $= 10010_{ten}$
  - $10010_{two} = 1\text{x}2^4 + 0\text{x}2^3 + 0\text{x}2^2 + 1\text{x}2^1 + 0\text{x}2^0$
    $= 1\text{x}16 + 0\text{x}8 + 0\text{x}4 + 1\text{x}2 + 0\text{x}1$
    $= 16_{ten} + 0_{ten} + 0_{ten} + 2_{ten} + 0_{ten}$
    $= 18_{ten}$

Numbers in Decimal vs. Binary

00	00000	16	10000
01	00001	17	10001
02	00010	18	10010
03	00011	19	10011
04	00100	20	10100
05	00101	21	10101
06	00110	22	10110
07	00111	23	10111
08	01000	24	11000
09	01001	25	11001
10	01010	26	11010
11	01011	27	11011
12	01100	28	11100
13	01101	29	11101
14	01110	30	11110
15	01111	31	11111

Instruction as Number Example (binary)

- **C code:** `i = j + k; /* i-k:$s0-$s2 */`
- **Assembly:** `add $s0,$s1,$s2 #s0=s1+s2`
- **Decimal representation:**

0	17	18	16	0	32

- **Binary representation:**

000000	10001	10010	10000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - Called <u>Machine Language Instruction</u>
  - Layout called <u>Instruction Format</u>
  - All MIPS instructions 32 bits (word): simple!

Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers
  2) Programs can be stored in memory to be read or written just like numbers
- Machine for accounting can help write a book; just load memory with program & data, & start executing at given address
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs
  - Compilers can translate HLL (data) into machine code (instructions)

Big Consequence #1: Everything addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - branches use memory address of instruction
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "<u>Program Counter</u>" (<u>PC</u>)
  - Better name is Instruction Address Register, but PC is traditional name

Big Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to instruction set architecture
  - Different version for Macintosh and IBM PC
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium II); could still run program from 1981 PC today

Instruction Format Field Names

- Fields have names:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- op: basic operation of instruction, "opcode"
- rs: 1st register source operand
- rt: 2nd register source operand
- rd: register destination operand, gets the result
- shamt: shift amount (user later, so 0 for now)
- funct: function; selects the specific variant of the operation in the op field; sometimes called the function code

Instruction Formats

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- What if want longer fields? e.g, `lw`
  - 5 bits => address of 2^5 or 32 => too small
  - But want all instructions same length!
- Principle: Good design demands good compromises
  - Add 2nd format with larger address

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

  - 1st R (register) format; 2nd I (immediate) format

Notes about Register and Imm. Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address		
	6 bits	5 bits	5 bits	16 bits		

- To make it easier for hardware (HW), 1st 3 fields same in R-format and I-format
- Alas, `rt` field meaning changed
  - R-format: `rt` is 2nd source operand
  - I-format: `rt` can be register destination operand
- How HW know which format is which?
  - Distinct values in 1st field (op) tell whether last 16 bits are 3 fields (R-format) or 1 field (I-format)

Instructions, Formats, "opcodes"

Instruction	Format	op	funct
• add	Register	0	32
• sub	Register	0	34
• slt	Register	0	42
• jr	Register	0	8
• lw	Immediate	35	
• sw	Immediate	43	
• addi	Immediate	8	
• beq	Immediate	4	
• bne	Immediate	5	
• slti	Immediate	10	

Register Format if op field = 0

Immediate Instruction in Machine Code

° C code: `i = j + 4; /* i,j:$s0,$s1 */`

° Assembly: `addi $s0,$s1,4` *#s0=s1+s2*

° Format:

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

° Decimal representation:

8	17	16	4
6 bits	5 bits	5 bits	16 bits

° Binary representation:

001000	10001	10000	0000 0000 0000 0100
6 bits	5 bits	5 bits	16 bits

Administrivia

° Readings: 3.4,3.8

° 2nd homework: Due Wed 2/10 7PM
- Exercises 3.7, 3.8, 3.10

° 2nd project: MIPS Disassembler
Due Wed. 2/17 7PM
`www-inst.EECS/~cs61c/handouts/proj2.pdf`

° Why is everything due Wednesday evenings?
- TA's try to set office hours to be available when useful; having single due date helps
- Its not Monday to try to avoid weekend

Administrivia

° Read the newsgroup! Especially before posting questions!
- Saves you time, saves TA time
- "Asked and answered" on newsgroup

° Midterm/Final Conflicts (3/17, 5/12)

1)If received letter from Mark regarding the final or midterm rescheduling, and not responded, you are implicitly saying you can make the *originally scheduled* time

2) That if you did NOT fill out the survey and can't make it to the originally scheduled times, email mds@cory immediately

"What's This Stuff Good For?"

Electro Shark Therapy: a chip in the diver's backpack activates two electrodes--one on the diver's back, and one attached to his fin--to generate a 92-volt force field. The pulsing charge repels sharks to a distance of 21 feet because of their hyper-sensitivity to naturally occurring electrical fields (which help them navigate). The long-term goal is to replace the chain-link shark nets along South Africa's popular Natal Coast beaches, which currently entangle and kill more than 1,000 sharks a year, with a large-scale electronic force field that will repel sharks without causing harm.
One Digital Day, 1998
www.intel.com/onedigitalday

Again,what do apps like these mean for reliability requirements of our technology?

What if constant bigger than 16 bits?

- Must keep instructions same size, but immediate field only 16 bits
- Add instruction to load upper 16 bits, then later instruction gives lower 16 bits
  - load upper immediate (`lui`) sets the upper 16 bits of a constant in a register
  - Machine language version of `lui $s0, 15`

001111	00000	10000	0000 0000 0000 1111

  - Contents of $s0 after executing lui $s0, 15

0000 0000 0000 1111	0000 0000 0000 0000

Big Constant Example

- `C: i = 80000; /* i:s1 */`
- MIPS Asm:
  - 80000_{ten} = 0000 0000 0000 0001 0011 1000 1000 0000_{two}
  - `lui  $at, 1`
    `addi $s1,$at,14464` *# 0011 1000 1000 0000*
- MIPS Machine Language

001111	00000	10001	0000 0000 0000 0001
001000	10001	10001	0011 1000 1000 0000

$s1 ($17):

0000 0000 0000 0001	0011 1000 1000 0000

Assembler Extends Instruction Set!

- Assembly Language extends native Machine Language
  - e.g., no `subi` machine instruction
  - Assemble `subi $x,$y,n` into `addi $x,$y,-n`
- Sometimes 1 Assembly language instruction turns into 2 Machine language instructions
  - `lw $x,80000($zero)` => `lui $at, 1`
    `lw  $x,14464($at)`
  - Register `$at` reserved for assembler to use as whenever it needs a register (compiler doesn't generate instructions using register `$at`)

Branch Addressing: Jumps, J format

° `j 10000 # go to location 10000`

° Add third format "J-format"; example:

2	10000
6 bits	26 bits

- Since no operands, can have big address
- Allows large programs
- Also used by `jal` (op = 3)

° Op field = 2 or Op field = 3 => J-format

° Op field = 0 => R-format

° Otherwise, I-format

Branch Addressing: PC-relative

° Conditional Branches need opcode, 2 register fields, one address: I-format

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- address just 16 bits (2^{16}), program too small!

° Option: always add address to register
PC = Register + Branch address

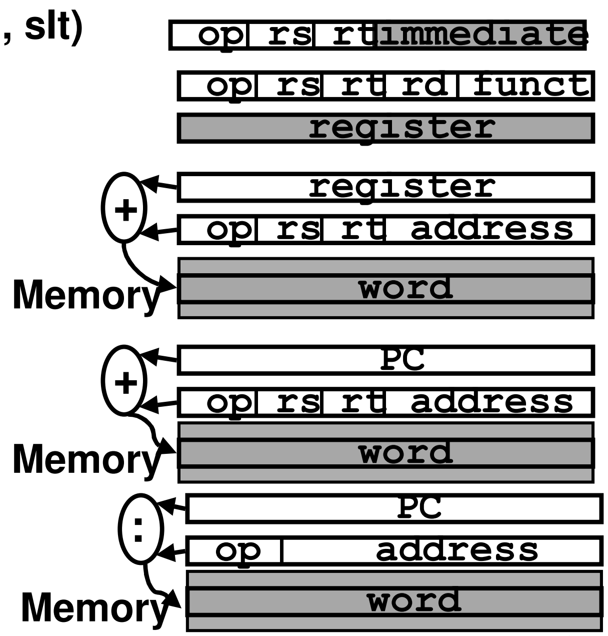
- Change register contents => bigger programs

° Which register?

- How use conditional branch? if-else, loops
- Near current instruction => use PC as reg!
- PC-relative addressing (PC+4) +/- 2^{15} words

MIPS Addressing Modes: operand shaded

- immediate(arith., slt)
- register (arithmetic, slt)
- base address (data transfers)
- PC-relative (branch)
- direct (jump)



Branch Addressing: PC-relative Example

```
Loop: slt  $t1,$zero,$a1   # t1=9,a1=5
      beq  $t1,$zero,Exit # no=>Exit
      add  $t0,$t0,$a0    # t0=8,a0=4
      subi $a1,$a1,1      # a1=5
      j    Loop           # goto Loop
Exit: add $v0,$t0,$zero   # v0=2,t0=8
```

- *See opcodes on slide 17, registers slide 6*

Address						
80000	0	0	5	9	0	42
80004	4	9	0	3		
80008	0	8	4	8	0	32
80012	8	5	5	-1		
80016	2	80000				
80020	0	8	0	2	0	32

80020 = 80004 + 4 + 3*4

Decoding Machine Instructions

- Sometimes forced to reverse engineer machine language to create original assembly language
  - e.g., looking at a core dump
- Step 1: look at op field to determine instruction format: R, I, J
- Step 2: break binary into fields by format
- Step 3: convert to decimal number / field
- Step 4: reverse into assembly using register names, instruction names, labels

Decoding example: if there is time

- Binary=>Decimal=>Assembly=>C?
  - Try to do in section if not in class
- Start at program at address 4,194,304 (2^{22})

```
00000000000000000001000000100000
00000000000001010100100000101010
00010001001000000000000000000101
00000000010001000001000000100000
00100000101001010111111111111111
00000000000001010100100000101010
00010101001000001111111111111101
```

- What are instruction formats of these 7 instructions?

"And in Conclusion …" 1/1

- MIPS assembly language instructions mapped to numbers in 3 formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	immediate		
J	op	address				

  - Op field determines format
- Operands
  - Registers: \$0 to \$31 mapped onto `$zero`; `$at`; `$v0`, `$v1`, `$a0..`, `$s0..`, `$t0..`, `$gp`, `$sp`, `$fp`, `$ra`
  - Memory : `Memory[0]`, `Memory[4]`, `Memory[8]`, `, . . . ,` `Memory[4294967292]`
    - Index is the address of the word

"And in Conclusion …" 2/2

- Machine Language: what HW understands
  - Assembly Language: can extend machine language to simplify for programmer
  - Can encode instructions as numbers
- Big Idea: Stored Program Concept
  - From C to binary, and vice versa
  - Everything has an address
  - Pointer in C is HLL version of address
  - Binary SW distribution => Binary compatibility => Instruction set "lock in"
- Next : More on procedures, stacks