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

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
  - $zero
  - $at (reserved for assembler)
  - (Return) Value registers ($v0,$v1)
  - Argument registers ($a0-$a3)
  - Temporary registers ($t0-$t3)
  - Saved registers ($s0-$s7)
  - Temporary registers ($t8-$t9)
  - $k0,$k1 (reserved for OS kernel)
  - Global Pointer ($gp)
  - Stack Pointer ($sp)
  - Frame Pointer ($fp), or $t10
  - Return Address ($ra)

Instruction as Number Example (decimal)

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

<table>
<thead>
<tr>
<th>0</th>
<th>17</th>
<th>18</th>
<th>16</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
</table>

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

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: \( d_4d_3d_2d_1d_0 \)
  - \( d_4 \times B^4 + d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0 \)
  - \( 10010_{\text{ten}} = 1 \times 10^4 + 0 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 0 \times 10^0 \)
  - \( 10010_{\text{ten}} = 1 \times 10000 + 0 \times 1000 + 0 \times 100 + 1 \times 10 + 0 \times 1 \)
  - \( 10010_{\text{ten}} = 10000 + 0 + 10 + 0 \)
  - \( 10010_{\text{two}} = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \)
  - \( 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \)
  - \( 16_{\text{ten}} + 0_{\text{ten}} + 0_{\text{ten}} + 2_{\text{ten}} + 0_{\text{ten}} \)
  - \( 18_{\text{ten}} \)
### Numbers in Decimal vs. Binary

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
</tr>
<tr>
<td>11</td>
<td>01011</td>
</tr>
<tr>
<td>12</td>
<td>01100</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
</tr>
<tr>
<td>14</td>
<td>01110</td>
</tr>
<tr>
<td>15</td>
<td>01111</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
</tr>
<tr>
<td>18</td>
<td>10010</td>
</tr>
<tr>
<td>19</td>
<td>10011</td>
</tr>
<tr>
<td>20</td>
<td>10100</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
<tr>
<td>22</td>
<td>10110</td>
</tr>
<tr>
<td>23</td>
<td>10111</td>
</tr>
<tr>
<td>24</td>
<td>11000</td>
</tr>
<tr>
<td>25</td>
<td>11001</td>
</tr>
<tr>
<td>26</td>
<td>11010</td>
</tr>
<tr>
<td>27</td>
<td>11011</td>
</tr>
<tr>
<td>28</td>
<td>11100</td>
</tr>
<tr>
<td>29</td>
<td>11101</td>
</tr>
<tr>
<td>30</td>
<td>11110</td>
</tr>
<tr>
<td>31</td>
<td>11111</td>
</tr>
</tbody>
</table>

### Instruction as Number Example (binary)

- **C code**: `i = j + k; /* i-k:$s0-$s2 */`
- **Assembly**: `add $s0,$s1,$s2 #s0=s1+s2`
- **Decimal representation**: 
  
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>
- **Binary representation**: 
  
  `000000 10001 10010 10000 00000 100000`

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

- Called Machine Language Instruction
- Layout called Instruction Format
- 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: “Program Counter” (PC)
  - 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: 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

- 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

Notes about Register and Imm. Formats

- 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”

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Register</td>
<td>0</td>
<td>32 Register</td>
</tr>
<tr>
<td>sub</td>
<td>Register</td>
<td>0</td>
<td>34 Format</td>
</tr>
<tr>
<td>slt</td>
<td>Register</td>
<td>0</td>
<td>42 if op field = 0</td>
</tr>
<tr>
<td>jr</td>
<td>Register</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>lw</td>
<td>Immediate</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>Immediate</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>Immediate</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>Immediate</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>Immediate</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td>Immediate</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

### Immediate Instruction in Machine Code

- **C code:**\[ i = j + 4; /* i,j:$s0,$s1 */ \]
- **Assembly:**\[ addi $s0,$s1,4 #s0=s1+s2 \]
- **Format:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Decimal representation:**\[ 8 17 16 4 \]
- **Binary representation:**\[ 001000 10001 10000 0000 0000 0000 0100 \]

### 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](mailto: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
  - It's not Monday to try to avoid weekend
- **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 hypersensitivity 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 10000000 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

° MIPS Machine Language

| 001111 00000 10001 0000 0000 0000 0001 |
| 001000 10001 0001 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:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>10000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
- 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 (arithemetic, slt)
- base address (data transfers)
- PC-relative (branch)
- direct (jump)

Branch Addressing: PC-relative Example

| Loop: slt $t1,$zero,$a1 # t1=9,a1=5 |
| beg $t1,$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 |

Address

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>0</th>
<th>5</th>
<th>9</th>
<th>0</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80004</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80008</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>80012</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80016</td>
<td>2</td>
<td></td>
<td></td>
<td>80000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80020</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

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
00000000010001000000000000000100
00100001010101111111111111111111
0000000000000000101010010000101010
000101010010000000000111111111111
```

- What are instruction formats of these 7 instructions?

“And in Conclusion …” 1/1

- MIPS assembly language instructions mapped to numbers in 3 formats

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td>address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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], \ldots, 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