CPS 104
Computer Organization
Lecture 4: Instruction Set Architecture: Simple (RMI) and MIPS

Robert Wagner
Overview of Today’s Lecture:

• Instruction Set Architecture(s).
• Minimal Instruction Set
• The power of arrays
• MIPS Instruction set as an example.
Review: Instruction Set Design

software

instruction set

hardware
Minimal Instruction Sets: “MI”

- The fewer different operations a computer can perform, the simpler (faster, cheaper, faster to design) it can be.

- What is the simplest form for an Instruction Set?
  - Suppose \( x \) and \( y \) represent variables, and \( Z \) may be either a variable, or a (signed) number. Let \( S \) be any statement label. Define just a few operations, each with few parameters:
    - \( \text{SUB}(x,y,Z) \) means \( x = y - Z \);
    - \( \text{BLTZ}(x,S) \) means if \( x < 0 \) go \( S \);
  - Except for input/output, this is enough.

- “Enough” means that this computer can compute ANY computable function.

- What about programs that use arrays?
Minimal Instruction Sets: “MI” 2

- Want our MI computer to be able to define all C constructs

- Example: The addition operation $x = y + Z$ (Assuming Zero holds 0):
  
  ```
  DEFINE ADD(x,y,Z) as {
    → T = Zero - Z;
    → x = y - T;
  }
  ```

- Array element referencing $A[I]$ not possible UNLESS some parts of an instruction can itself be computed
  
  - Address computation allows the address of $A[I]$ in some load instruction to be computed as $&A+I$
  
  - Instructions are stored in memory, so they COULD be modified.
  
  - Better: Provide “indirect address” so $X=&A+I$; $Z=*X-0$; now performs $Z=A[I]$;
Minimal Instruction Set 3

- Example: Testing for zero
  - Idea: IF x<0, OR -x < 0, THEN x is NOT 0.
  - DEFINE BNZ(x,S) as {
    - if (x<0) go S; x is NOT zero, so “branch”
    - T = Zero - A; T is now -x. Instruction has proper form.
    - if (T<0) go S; If -x < 0 branch to S
    - } Here if x==0

- Executing a “go S” statement causes statement labeled S to be executed next, INSTEAD of the statement following the “go”.

- S actually equals the location in memory where the statement labeled S is placed. “Go S” sets the machine’s “program counter” (abbreviated PC) to S.
Minimal Instruction Set 3

- Assume M1 is a variable holding -1
- Example: for(I=-N; I<0; I++) { body }:
  - SUB(I, Zero, N)  // I = 0 - N;
  - BLTZ(M1, L1)  // if (-1<0) go L1; ALWAYS go L1
  - L2: ... body ...
  - SUB(I, I, -1)  // I = I + 1
  - L1: BLTZ(I, L2)  // if (I<0) go L2;

- Here, the “go L2” “closes the loop”, setting PC to a smaller value, so some statements are executed again.
- “go S” is often written “goto S” or “branch to S”.
- A “go S” FORGETS where the program was before the “go S” was executed.
RMI: Examine the bits of Var

- Problem: Set \(X=\text{(value of some bit of Var)}\) Choose the bit whose value is easiest for RMI to test.
- Problem: Set \(Y=\text{(value of the next easiest to test bit of Var)}\)
- Problem: Set \(Z=\text{(the number whose binary representation has the two low-order bits XY, in that order, and whose other bits are all zeros)}\)
- Problem: Explain how \(Z\) could be computed in C.
Hardware Fetch - Execute Cycle

- IR and PC are registers INVISIBLE to program.
  - IR = Mem[PC]

- Examine op-code field of IR, extract fields of instruction from IR

- Using fields extracted above, get actual operands (from registers, or IR field)

- Steer operands through ALU, which computes result at its output connection

- Use “Destination” field of IR to select register for ALU result: R[Dest] = ALU

- if (“Taken branch”) PC = (L field of IR) else PC = PC + 4
Stored Program Computer

- **Instructions**: a fixed set of built-in operations
  - Each line of code above could be such an operation

- **Instructions and data are stored in the computer memory.**

- **Registers**: A small number of words (in fast memory) that can be directly accessed by the Arithmetic-Logic Unit (ALU).
  - Complicates the instruction set, but simplifies the hardware

- **Add more basic operations, for programming ease**
Designing a Better Computer than RMI

- RMI is inconvenient to program -- too few instructions
- Using same basic hardware “flowchart”, improve the design
What Must be Specified?

- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches

- *fetch-decode-execute is implicit!*
Operations Supported

- Some kind of 2-argument arithmetic operation
  - Subtract is enough; Add is more directly useful
- Some kind of conditional test
  - Branch to S if $x<0$ is enough
- Schemes which allow parts of each instruction to be variables, typically addresses of operands, so array indexing can be supported
  - Original von Neumann design used arithmetic on instructions to support array indexing
  - Allow branch target to be variable, too
- Subroutine call and return
- Input and Output
Arithmetic operations

- Add and Subtract are most common, most useful
- Modern machines support all C arithmetic and logical operations as single instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Action</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>$X = A + B$</td>
<td>ADD</td>
</tr>
<tr>
<td>-</td>
<td>$X = A - B$</td>
<td>SUB</td>
</tr>
<tr>
<td>&amp;</td>
<td>$X_1 = A_1 &amp; B_1$</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>$1 &amp; 1 == 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$0 &amp; x == 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x &amp; 0 == 0$</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>$X = A &gt;&gt; B$</td>
<td>Shift right</td>
</tr>
<tr>
<td></td>
<td>$X_1 = A_{l+B}$</td>
<td>SHRA or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHRL</td>
</tr>
</tbody>
</table>
Branching

- Usually, next instruction is found at PC+1
- To “test”, conditionally set PC to new location S
  - To branch unconditionally, make the test “always true”
- This causes next instruction to come from S, if the test is true

- Subroutine call
  - Record current PC+1 in some register R (say, $31)
  - Branch to subroutine entry location
  - To return, branch to the location held in R

- Switch:
  - Compute branch target location in some register Q
  - Branch to “contents of Q”
The Most-Useful Instructions

 Variations of the instructions below are common to most computers, and are enough for most programs

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>Variations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>D = A + B</td>
<td>Signed/ Unsigned</td>
</tr>
<tr>
<td>SUB</td>
<td>D = A - B</td>
<td>Word/ Byte Immediate/ Variable</td>
</tr>
<tr>
<td>BLTZ</td>
<td>if A&lt;0 go S</td>
<td>Conditional branch to location S.</td>
</tr>
<tr>
<td>BGEZ</td>
<td>if A&gt;=0 go S</td>
<td>Unconditional by choosing A constant</td>
</tr>
<tr>
<td>BEQZ</td>
<td>if A==0 go S</td>
<td></td>
</tr>
<tr>
<td>BNEZ</td>
<td>if A!=0 go S</td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>$31 = PC+1, go to S</td>
<td>Call subroutine</td>
</tr>
</tbody>
</table>
## The Most-Useful Instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>Variations</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHR</td>
<td>( D = A &gt;&gt; B )</td>
<td>Arithmetic/Logical</td>
</tr>
<tr>
<td></td>
<td>( D = A &lt;&lt; B )</td>
<td>B Immediate/Variable</td>
</tr>
<tr>
<td>SHL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>( D = A &amp; B )</td>
<td>Bit-wise Logical</td>
</tr>
<tr>
<td>OR</td>
<td>( D = A</td>
<td>B )</td>
</tr>
<tr>
<td>XOR</td>
<td>( D = A ^ B )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A&amp;B</th>
<th>A</th>
<th>B</th>
<th>A^B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Computed Addresses

- Machine uses register contents to compute locations of memory operands
  - Used for array accesses:
    - Translate $X = A[I]$ to Machine-level Language:
      - $1 = I$
      - $1 = 1 << 4$
      - $1 = 1 + &A$
      - $2 = 0(1) // 2 = Mem[0 + 1]$
  - Used for computed branch addresses
    - subroutine return, switches

- Allows instruction operand fields to be short
Computed Addresses (cont)

- The “Effective address” (EA) of an operand is the result of the computation the instruction does to form the operand’s address.
- This address computation is usually an ADD involving at least one register’s contents
  - LW R1, 124(R2)
  - Loads R1 with the value found at R2+124
- In simpler case, “indirect address” can be used:
  - LW R1, *XXX
  - Loads R1 with the value found at the memory address held in variable XXX (XXX is a memory location)
  - This allows program to compute an address (using ordinary integer addition, say) into XXX, then use that address to find an operand
The Power of Arrays

- Accessing an array element is FAST: 3 cycles
  - \( A[x] \) gets value of function \( f(x) \) in 3 cycles, once \( A[x]=f(x) \) is pre-calculated (INITIALIZED).

<table>
<thead>
<tr>
<th></th>
<th>INITIALIZE ONCE</th>
<th>USE MANY TIMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classifying Characters</td>
<td>For(I=0;I&lt;3;I++) ( A[&quot;abc&quot;[I]] = CLlc; )</td>
<td>CI = A[ch];</td>
</tr>
<tr>
<td>Histogram</td>
<td>A[I]=0;</td>
<td>A[Obs[j]]++;</td>
</tr>
<tr>
<td>Function Inverse</td>
<td>For(I=0; I&lt;N; I++) ( A[f(I)] = I; )</td>
<td>Given j, ( A[j] ) finds I \ s.t. ( f(I)==j )</td>
</tr>
</tbody>
</table>
MIPS Registers and Memory

- **Memory:**
- **Registers:**
  - (R0 = 0)
  - f0 .. f31 floats
  - (f0,f1), (f2,f3) ...
    form pairs for DP

<table>
<thead>
<tr>
<th>r0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>r31</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
</tr>
<tr>
<td>lo</td>
<td></td>
</tr>
<tr>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>
Computer Memory

- Memory is a large linear array of words.
  - Each word has a unique address (location).
- All ("new") computers support byte (8-bits) addressing.
- Hardware accesses memory by words (4 or 8 bytes).
Buzz Word Definition: Endianess

Byte Order

- **Big Endian**: byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- **Little Endian**: byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC Alpha

---

![Diagram of byte order]

- **msb** (most significant bit)
- **lsb** (least significant bit)
- **little endian byte 0**
- **big endian byte 0**
Buzz Word Definition: Alignment

Alignment: require that object addresses be multiples of their size in bytes.

- 32-bit integer
  - Aligned if address % 4 = 0

- 64-bit integer?
  - Aligned if ?

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Aligned</th>
<th>Not Aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Program’s View of Memory

What is Memory? a bunch of bits

Looks like a large linear array

Find things by indexing into array

- unsigned integer

Most computers support byte (8-bit) addressing

- Each byte has a unique address (location).
- Byte of data at address 0x100 and 0x101
- Word of data at address 0x100 and 0x104

32-bit v.s. 64-bit addresses

- we will assume 32-bit for rest of course, unless otherwise stated
Memory Partitions (Software convention)

- **Text for instructions**
  - `add res, src1, src2`
  - `mem[res] = mem[src1] + mem[src2]`

- **Data**
  - static (constants, globals)
  - dynamic (heap, new allocated)
  - grows up

- **Stack**
  - local variables
  - grows down

- **Variables are names for memory locations**
  - `int x;`
Bit Manipulations

Problem

○ 32-bit word contains many values
  ● e.g., input device, sensors, etc.
  ● current x,y position of mouse and which button (left, mid, right)

○ Assume x, y position is 0-255

○ How many bits for position?

○ How many for button?

Goal

○ Extract position and button from 32-bit word

○ Need operations on individual bits of binary numbers
Bitwise AND / OR

- `&` operator performs bitwise AND
- `|` operator performs bitwise OR

**Per bit**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &amp; 0 = 0</td>
<td>0</td>
</tr>
<tr>
<td>0 &amp; 1 = 0</td>
<td>0</td>
</tr>
<tr>
<td>1 &amp; 0 = 0</td>
<td>1</td>
</tr>
<tr>
<td>1 &amp; 1 = 1</td>
<td>1</td>
</tr>
</tbody>
</table>

**For multiple bits, apply operation to individual bits in same position**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>OR</td>
</tr>
<tr>
<td>011010</td>
<td>011010</td>
</tr>
<tr>
<td>101110</td>
<td>101110</td>
</tr>
<tr>
<td>001010</td>
<td>111110</td>
</tr>
</tbody>
</table>
Mouse Example

32-bit word with x,y and button fields
- bits 0-7 contain x position
- bits 8-15 contain y position
- bits 16-17 contain button (0 = left, 1 = middle, 2 = right)

to extract 1 value need to clear all other bits

How do I use bitwise operations to do this?

<table>
<thead>
<tr>
<th>button</th>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1a34c</td>
<td>01 1010 0011 0100 1100</td>
<td></td>
</tr>
</tbody>
</table>
Mouse Solution

- AND with a bit mask
  - Specific values that clear some bits, but pass others through
- To extract x position use mask 0x000ff

\[
xpos = 0x1a34c \& 0x000ff
\]

<table>
<thead>
<tr>
<th>button</th>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1a34c = 01 1010 0011 0100 1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000ff = 00 0000 0000 1111 1111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0004c = 00 0000 0000 0100 1100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More of the Mouse Solution

- To extract y position use mask 0x0ff00
  \[ y_{pos} = 0x1a34c \& 0x0ff00 \]
- Similarly, button is extracted with mask 0x30000
  \[ button = 0x1a34c \& 0x30000 \]
- Not quite done...why?

<table>
<thead>
<tr>
<th>button</th>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1a34c</td>
<td>01 1010</td>
<td>0011 0100 1100</td>
</tr>
<tr>
<td>0x0ff00</td>
<td>00 1111 1111 0000 0000</td>
<td></td>
</tr>
<tr>
<td>0x0a300</td>
<td>00 1010 0011 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
The SHIFT operator

- `m >>` is shift right, `<<` is shift left, operands are int and number of bit positions to shift
- `(1 << 3)` is `000001_2 <<3 → 0001000_2` (it’s $2^3$)
- `0xff00` is `0xff << 8`, and `0xff` is `0xff00 >> 8`
- So, true ypos value is:

\[
\text{ypos} = (0x1a34c \& 0xff00) >> 8
\]

\[
\text{button} = (0x1a34c \& 0x30000) >> 16
\]
MIPS Instruction set Architecture

- 3-Address Load Store Architecture.
- Register and Immediate addressing modes for ALU operations.
- Immediate and Displacement addressing for Loads and Stores.
- Immediate and Displacement fields ARE LIMITED to 16 bits

- Examples:
  - add $1, $2, $3 # $1 = $2 + $3
  - addi $1, $1, 4 # $1 = $1 + 4
  - lw $1, 100 ($2) # $1 = Memory[$2 + 100]
  - sw $1, 100 ($2) # Memory[$2 + 100] = $1
  - lui $1, 100 # $1 = 100 X 2^{16}
  - addi $1, $3, 100 # $1 = $3 + 100
Choice of Sizes

- Word size -- pretty standard at 32 bits
- Instruction size -- 1 word
- # of registers
  - Tried: 1, 3, 8, 16.
  - MIPS uses 32, of each type. (5 bit register number)
- Size of memory
  - Now > 256 MB (28 bits)
  - Predicted to use 1 extra bit per 2 years (seems to grow faster)
- Displacement Size
  - Nice to use a full memory address size
    - 32-28 = 4 bits left for every other function -- not enough
  - MIPS uses 16 bits, which handles local addressing
- Immediate size of 16 bits is also usually enough
Addressing Summary

• Data Addressing modes that are important: Displacement, Immediate, Register Indirect

• Displacement size should be 12 to 16 bits

• Immediate size should be 8 to 16 bits

• MIPS does not support indirect addressing -- using a displacement of value 0 does the equivalent