There is one handout today at the front and back of the room!

Lecturer PSOE, new dad Dan Garcia

2005-09-28

Hearing keys?! ⇒ Prof Doug Tygar revealed that you can just listen to the sound of keystrokes to find out what’s typed…beware!

www.earthtimes.org/articles/show/4103.html
Review

• Functions called with jal, return with jr $ra.

• The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.

• Instructions we know so far
  Arithmetic: add, addi, sub, addu, addiu, subu
  Memory: lw, sw, lb, sb, lbu
  Decision: beq, bne, slt, slti, sltu, sltiu
  Unconditional Branches (Jumps): j, jal, jr

• Registers we know so far
  • All of them!
  • There are CONVENTIONS when calling procedures!
Bitwise Operations

• Up until now, we’ve done arithmetic (\texttt{add, sub, addi}), memory access (\texttt{lw and sw}), and branches and jumps.

• All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)

• \textbf{New Perspective}: View register as 32 raw bits rather than as a single 32-bit number

• Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.

• Introduce two new classes of instructions:
  • Logical & Shift Ops
Logical Operators (1/3)

• Two basic logical operators:
  • AND: outputs 1 only if both inputs are 1
  • OR: outputs 1 if at least one input is 1

• Truth Table: standard table listing all possible combinations of inputs and resultant output for each. E.g.,

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical Operators (2/3)

• Logical Instruction Syntax:
  1  2,3,4
  • where
    1) operation name
    2) register that will receive value
    3) first operand (register)
    4) second operand (register) or immediate (numerical constant)

• In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output

• Again, rigid syntax, simpler hardware
Logical Operators (3/3)

• Instruction Names:
  • and, or: Both of these expect the third argument to be a register
  • andi, ori: Both of these expect the third argument to be an immediate

• MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0’s of the inputs, bit 1 by the bit 1’s, etc.
  • C: Bitwise AND is & (e.g., \( z = x \& y \);)
  • C: Bitwise OR is | (e.g., \( z = x \mid y \);)
Uses for Logical Operators (1/3)

• Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.

• This can be used to create a mask.
  • Example:
    
    ```
    1011 0110 1010 0100 0011 1101 1001 1010
    mask: 0000 0000 0000 0000 0000 1111 1111 1111
    
    The result of anding these:
    0000 0000 0000 0000 0000 1101 1001 1010
    mask last 12 bits
    ```
• The second bitstring in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).

• Thus, the and operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
  • In particular, if the first bitstring in the above example were in $t0$, then the following instruction would mask it:

    andi  $t0,$t0,0xFFF
Uses for Logical Operators (3/3)

• Similarly, note that or\(i\)ng a bit with 1 produces a 1 at the output while or\(i\)ng a bit with 0 produces the original bit.

• This can be used to force certain bits of a string to 1s.
  • For example, if \(t0\) contains \(0x12345678\), then after this instruction:
    \[
    \text{ori } t0, t0, 0xFFFF
    \]
  • ... \(t0\) contains \(0x1234FFFF\) (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).
Shift Instructions (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits.
  
  - Example: shift right by 8 bits
    
    $\begin{array}{c}
    0001 \\
    0010 \\
    0011 \\
    0100 \\
    0101 \\
    0110 \\
    0111 \\
    1000 \\
    \end{array}$
    
    $\begin{array}{c}
    0000 \\
    0000 \\
    0001 \\
    0010 \\
    0011 \\
    0100 \\
    0101 \\
    0110 \\
    \end{array}$
    
  - Example: shift left by 8 bits
    
    $\begin{array}{c}
    0001 \\
    0010 \\
    0011 \\
    0100 \\
    0101 \\
    0110 \\
    0111 \\
    1000 \\
    \end{array}$
    
    $\begin{array}{c}
    0011 \\
    0100 \\
    0101 \\
    0110 \\
    0111 \\
    1000 \\
    \end{array}$

Shift Instructions (2/4)

• Shift Instruction Syntax:
  1  2,3,4
   • where
     1) operation name
     2) register that will receive value
     3) first operand (register)
     4) shift amount (constant < 32)

• MIPS shift instructions:
  1. **sll** (shift left logical): shifts left and **fills emptied bits with 0s**
  2. **srl** (shift right logical): shifts right and **fills emptied bits with 0s**
  3. **sra** (shift right arithmetic): shifts right and **fills emptied bits by sign extending**
Shift Instructions (3/4)

• Example: shift right arith by 8 bits

```
0001 0010 0011 0100 0101 0110 0111 1000
```

```
0000 0000 0001 0010 0011 0100 0101 0110
```

• Example: shift right arith by 8 bits

```
1001 0010 0011 0100 0101 0110 0111 1000
```

```
1111 1111 1001 0010 0011 0100 0101 0110
```
Shift Instructions (4/4)

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:
  
  ```
  a *= 8; (in C)
  
  would compile to:
  sll $s0,$s0,3 (in MIPS)
  ```

- Likewise, shift right to divide by powers of 2
  
  - remember to use `sra`
Peer Instruction

r: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
... ### PUSH REGISTER(S) TO STACK?
jal e # Call e
... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to caller of r

e: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to r

What does r have to push on the stack before “jal e”?

1: 1 of ($s0,$sp,$v0,$t0,$a0,$ra)
2: 2 of ($s0,$sp,$v0,$t0,$a0,$ra)
3: 3 of ($s0,$sp,$v0,$t0,$a0,$ra)
4: 4 of ($s0,$sp,$v0,$t0,$a0,$ra)
5: 5 of ($s0,$sp,$v0,$t0,$a0,$ra)
6: 6 of ($s0,$sp,$v0,$t0,$a0,$ra)
7: 0 of ($s0,$sp,$v0,$t0,$a0,$ra)
Peer Instruction Answer

r: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
      ### PUSH REGISTER(S) TO STACK?
... # Call e
jal e  # Call e
... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to caller of r

e: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to r

What does r have to push on the stack before “jal e”?

Saved  Volatile! -- need to push

1:  1 of ($s0,$sp,$v0,$t0,$a0,$ra)
2:  2 of ($s0,$sp,$v0,$t0,$a0,$ra)
3:  3 of ($s0,$sp,$v0,$t0,$a0,$ra)
4:  4 of ($s0,$sp,$v0,$t0,$a0,$ra)
5:  5 of ($s0,$sp,$v0,$t0,$a0,$ra)
6:  6 of ($s0,$sp,$v0,$t0,$a0,$ra)
7:  0 of ($s0,$sp,$v0,$t0,$a0,$ra)

e can’t return changed, e can return changed
no need to push
Administrivia

• Any administrivia?
Overview – Instruction Representation

• Big idea: stored program
  • consequences of stored program

• Instructions as numbers

• Instruction encoding

• MIPS instruction format for Add instructions

• MIPS instruction format for Immediate, Data transfer instructions
Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers.
  2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).

- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs
Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - both branches and jumps use these
- 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)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name
Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintoshes and PCs

- 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 4); could still run program from 1981 PC today
Instructions as Numbers (1/2)

• Currently all data we work with is in words (32-bit blocks):
  • Each register is a word.
  • \textit{lw} and \textit{sw} both access memory one word at a time.

• So how do we represent instructions?
  • Remember: Computer only understands 1s and 0s, so “\texttt{add$ t0,$0,$0}” is meaningless.
  • MIPS wants simplicity: since data is in words, make instructions be words too.
Instructions as Numbers (2/2)

• One word is 32 bits, so divide instruction word into “fields”.

• Each field tells computer something about instruction.

• We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  • R-format
  • I-format
  • J-format
Instruction Formats

• **I-format**: used for instructions with immediates, \textit{lw} and \textit{sw} (since the offset counts as an immediate), and the branches (\textit{beq} and \textit{bne}),
  
  • (but not the shift instructions; later)

• **J-format**: used for \textit{j} and \textit{jal}

• **R-format**: used for all other instructions

• It will soon become clear why the instructions have been partitioned in this way.
R-Format Instructions (1/5)

• Define “fields” of the following number of bits each: $6 + 5 + 5 + 5 + 5 + 6 = 32$

| 6 | 5 | 5 | 5 | 5 | 6 |

• For simplicity, each field has a name:

| opcode | rs | rt | rd | shamt | funct |

• Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.

• Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.
R-Format Instructions (2/5)

• What do these field integer values tell us?
  • **opcode**: partially specifies what instruction it is
    - Note: This number is equal to 0 for all R-Format instructions.
  • **funct**: combined with opcode, this number exactly specifies the instruction

• Question: Why aren’t opcode and funct a single 12-bit field?
  - Answer: We’ll answer this later.
R-Format Instructions (3/5)

• More fields:
  • \texttt{rs} (Source Register): \textit{generally} used to specify register containing first operand
  • \texttt{rt} (Target Register): \textit{generally} used to specify register containing second operand (note that name is misleading)
  • \texttt{rd} (Destination Register): \textit{generally} used to specify register which will receive result of computation
R-Format Instructions (4/5)

• Notes about register fields:
  • Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  • The word “generally” was used because there are exceptions that we’ll see later. E.g.,
    - `mult` and `div` have nothing important in the `rd` field since the dest registers are `hi` and `lo`
    - `mfhi` and `mflo` have nothing important in the `rs` and `rt` fields since the source is determined by the instruction (p. 264 P&H)
R-Format Instructions (5/5)

• Final field:
  • **shamt**: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31).

  • This field is set to 0 in all but the shift instructions.

• For a detailed description of field usage for each instruction, see green insert in COD 3/e

  • (You can bring with you to all exams)
R-Format Example (1/2)

• MIPS Instruction:
  \[ \text{add} \quad \$8, \$9, \$10 \]

  \begin{align*}
  \text{opcode} &= 0 \quad \text{(look up in table in book)} \\
  \text{funct} &= 32 \quad \text{(look up in table in book)} \\
  \text{rd} &= 8 \quad \text{(destination)} \\
  \text{rs} &= 9 \quad \text{(first operand)} \\
  \text{rt} &= 10 \quad \text{(second operand)} \\
  \text{shamt} &= 0 \quad \text{(not a shift)}
  \end{align*}
R-Format Example (2/2)

- MIPS Instruction:
  \[ \text{add } \$8, \$9, \$10 \]

Decimal number per field representation:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>9</th>
<th>10</th>
<th>8</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
</table>

Binary number per field representation:

\begin{align*}
000000 & | 01001 & | 01010 & | 01000 & | 00000 & | 100000 \\
\end{align*}

hex representation: \(012A \ 4020_{\text{hex}}\)

decimal representation: \(19,546,144_{\text{ten}}\)

- Called a Machine Language Instruction
I-Format Instructions (1/4)

• What about instructions with immediates?
  • 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  • Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise

• Define new instruction format that is partially consistent with R-format:
  • First notice that, if instruction has immediate, then it uses at most 2 registers.
I-Format Instructions (2/4)

• Define “fields” of the following number of bits each: $6 + 5 + 5 + 16 = 32$ bits

| 6 | 5 | 5 | 16 |

• Again, each field has a name:

| opcode | rs | rt | immediate |

• Key Concept: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location.
I-Format Instructions (3/4)

• What do these fields mean?
  • **opcode**: same as before except that, since there’s no **funct** field, **opcode** uniquely specifies an instruction in I-format
  • This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent with other formats.
  • **rs**: specifies the *only* register operand (if there is one)
  • **rt**: specifies register which will receive result of computation (this is why it’s called the *target* register “rt”)
I-Format Instructions (4/4)

• The Immediate Field:
  
  • `addi, slti, sltiu`, the immediate is **sign-extended** to 32 bits. Thus, it’s treated as a signed integer.
  
  • 16 bits $\Rightarrow$ can be used to represent immediate up to $2^{16}$ different values.
  
  • This is large enough to handle the offset in a typical `lw` or `sw`, plus a vast majority of values that will be used in the `slti` instruction.
  
  • We’ll see what to do when the number is too big in our next lecture...
I-Format Example (1/2)

- MIPS Instruction:
  
  `addi $21, $22, -50`

  - `opcode = 8` (look up in table in book)
  - `rs = 22` (register containing operand)
  - `rt = 21` (target register)
  - `immediate = -50` (by default, this is decimal)
I-Format Example (2/2)

- MIPS Instruction:
  \texttt{addi} \ $21,\$22,-50$

  \begin{array}{cccc}
  & 8 & 22 & 21 & -50 \\
  \end{array}

  \begin{array}{c}
  \text{Binary/field representation:} \\
  001000\ 10110\ 10101\ 1111111111001110 \\
  \text{hexadecimal representation:} \ 22D5 \ FFCE_{\text{hex}} \\
  \text{decimal representation:} \ 584,449,998_{\text{ten}}
  \end{array}
**Peer Instruction**

Which instruction has same representation as $35_{ten}$?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. add $0, 0, 0$</td>
<td></td>
</tr>
<tr>
<td>2. subu $s0, s0, s0$</td>
<td></td>
</tr>
<tr>
<td>3. lw $0, 0(0)$</td>
<td></td>
</tr>
<tr>
<td>4. addi $0, 0, 35$</td>
<td></td>
</tr>
<tr>
<td>5. subu $0, 0, 0$</td>
<td></td>
</tr>
<tr>
<td>6. Trick question! (Instructions are not numbers)</td>
<td></td>
</tr>
</tbody>
</table>

Registers numbers and names:
0: $0$, .. 8: $t0$, 9:$t1$, ..15: $t7$, 16: $s0$, 17: $s1$, .. 23: $s7$

Opcode and function fields (if necessary)

- add: opcode = 0, funct = 32
- subu: opcode = 0, funct = 35
- addi: opcode = 8
- lw: opcode = 35
Peer Instruction Answer

Which instruction bit pattern = number 35?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Function Field</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. add $0, $0, $0</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 0 32</td>
</tr>
<tr>
<td>2. subu $s0,$s0,$s0</td>
<td>0</td>
<td>16 16 16</td>
<td>0 16 16 16 0 35</td>
</tr>
<tr>
<td>3. lw $0, 0($0)</td>
<td>35</td>
<td>0</td>
<td>35 0 0 0 0 0</td>
</tr>
<tr>
<td>4. addi $0, $0, 35</td>
<td>8</td>
<td>0</td>
<td>8 0 0 0 0 35</td>
</tr>
<tr>
<td>5. subu $0, $0, $0</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 0 35</td>
</tr>
<tr>
<td>6. Trick question!</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instructions != numbers

Registers numbers and names:
0: $0, ..., 8: $t0, 9:$t1, ..., 16: $s0, 17: $s1, ...

Opcodes and function fields

- **add**: opcode = 0, function field = 32
- **subu**: opcode = 0, function field = 35
- **addi**: opcode = 8
- **lw**: opcode = 35
In conclusion...

- **Logical and Shift Instructions**
  - Operate on individual bits (arithmetic operate on entire word)
  - Use to isolate fields, either by masking or by shifting back & forth
  - Use `shift left logical, sll`, for multiplication by powers of 2
  - Use `shift right arithmetic, sra`, for division by powers of 2

- **Simplifying MIPS:** Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use `lw` and `sw`).

- Computer actually stores programs as a series of these 32-bit numbers.

- **MIPS Machine Language Instruction:**
  32 bits representing a single instruction

<table>
<thead>
<tr>
<th>R</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediat e</td>
</tr>
<tr>
<td>J</td>
<td>opcode</td>
<td></td>
<td></td>
<td>target address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>