CS61C
Negative Numbers and Logical Operations

Lecture 7

February 10, 1999

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

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

Review 1/2

- 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
- Binary => Decimal => Assembly => Symbolic Assembly => C
  - Reverse Engineering or Disassembly
  - Its hard to do, therefore people like shipping binary machine language more than assembly or C

Review 2/2

- Programming language model of memory allocation and pointers
  - Allocate in stack vs. heap vs. global areas
  - Arguments passed
    call by value vs. call by reference
  - Pointer in C is HLL version of machine address

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: $d_{31}d_{30} \ldots d_2d_1d_0$
  - $d_{31} \times B^{31} + d_{30} \times B^{30} + \ldots + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$
  - One billion ($1{,}000{,}000{,}000_{ten}$) is
    $0011\ 1011\ 1001\ 1010\ 1100\ 1010\ 0000\ 0000_{two}$
    $2^{28}\quad 2^{24}\quad 2^{20}\quad 2^{16}\quad 2^{12}\quad 2^{8}\quad 2^{4}\quad 2^{0}$

    $= 1\text{x}2^{29}+1\text{x}2^{28}+1\text{x}2^{27}+1\text{x}2^{25}+1\text{x}2^{24}+1\text{x}2^{23}+1\text{x}2^{20}$
    $+1\text{x}2^{19}+1\text{x}2^{17}+1\text{x}2^{15}+1\text{x}2^{14}+1\text{x}2^{11}+1\text{x}2^{9}$

    = 536,870,912 + 268,435,456 + 134,217,728
    + 33,554,432 + 16,777,216 + 8,388,608 + 1,048,576
    + 524,288+ 131,072+ 32,768+ 16,384+ 2,048+ 512

Overview

- What if Numbers too Big?
- How Represent Negative Numbers?
- What if Result doesn't fit in register?
- More Compact Notation than Binary?
- Administrivia,"What's this stuff good for"
- Shift Instructions
- And/Or Instructions
- Conclusion

What if too big?

- Binary bit patterns above are simply representatives of numbers
- Numbers really have an infinite number of digits
  - with almost all being zero except for a few of the rightmost digits
  - Just don't normally show leading zeros
- If result of add (or any other arithmetic operation), cannot be represented by these rightmost hardware bits, overflow is said to have occurred
- Up to Compiler and OS what to do

How avoid overflow, allow it sometimes?

- Some languages detect overflow (Ada), some don't (C)
- MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
  - add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow
  - add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do not cause exceptions on overflow
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce `addu`, `addiu`, `subu`

What if Overflow Detected?

- An "exception" (or "interrupt") occurs
  - Address of the instruction that overflowed is saved in a register
  - Computer jumps to predefined address to invoke appropriate routine for that exception
  - Like an unplanned hardware function call
- Operating system decides what to do
  - In some situations program continues after corrective code is executed
- MIPS support: exception program counter (EPC) contains address of that instruction
  - move from system control (`mfc0`) to copy EPC

How Represent Negative Numbers?

- Obvious solution: add a separate sign!
  - sign represented in a single bit!
  - representation called sign and magnitude
- Shortcomings of sign and magnitude
  - Where to put the sign bit: right? left?
  - Separate sign bit means it has both a positive and negative zero, lead to problems for inattentive programmers: +0 = -0?
  - Adder may need extra step size don't know sign in advance
- Thus sign and magnitude was abandoned

Search for Negative Number Representation

- Obvious solution didn't work, find another
- What is result for unsigned numbers if tried to subtract large number from a small one?
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
  - With no obvious better alternative, pick representation that made the hardware simple: leading 0s => positive, leading 1s => negative
  - 000000...xxx is >=0, 111111...xxx is < 0
- This representation called two's complement

Two's Complement

$0000 \ldots 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000 \ldots 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000 \ldots 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
. . .
$0111 \ldots 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111 \ldots 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111 \ldots 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
. . .
$1111 \ldots 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111 \ldots 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111 \ldots 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

- One zero, 1st bit => >=0 or <0, called sign bit
  - but one negative with no positive $-2{,}147{,}483{,}648_{ten}$

Two's Complement Formula, Example

- Recognizing role of sign bit, can represent positive and negative numbers in terms of the bit value times a power of 2:
  - $d_{31} \times -2^{31} + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$
- Example
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$
  $= 1x-2^{31} + 1x2^{30} + 1x2^{29} + \ldots + 1x2^2 + 0x2^1 + 0x2^0$
  $= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^2 + 0 + 0$
  $= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten}$
  $= -4_{ten}$

Overflow for Two's Complement Numbers?

- Adding (or subtracting) 2 32-bit numbers can yield a result that needs 33 bits
  - sign bit set with value of result instead of proper sign of result
  - since need just 1 extra bit, only sign bit can be wrong

Op	A	B	Result
A + B	>=0	>=0	<0
A + B	<0	<0	>=0
A - B	>=0	<0	<0
A - B	<0	>=0	>=0

- Adding operands with different signs, (subtracting with same signs) overflow cannot occur

Signed v. Unsigned Comparisons

- Note: memory addresses naturally start at 0 and continue to the largest address
  - That is, negative addresses make no sense
- C makes distinction in declaration
  - integer (`int`) can be positive or negative
  - unsigned integers (`unsigned int`) only positive
- Thus MIPS needs two styles of compare
  - Set on less than (`slt`) and set on less than immediate (`slti`) work with signed integers
  - Set on less than unsigned (`sltu`) and set on less than immediate unsigned (`sltiu`)

Example: Signed v. Unsigned Comparisons

- $s0 has

1111 1111 1111 1111 1111 1111 1111 1100_{two}

- $s1 has

0011 1011 1001 1010 1000 1010 0000 0000_{two}

- What are $t0, $t1 after

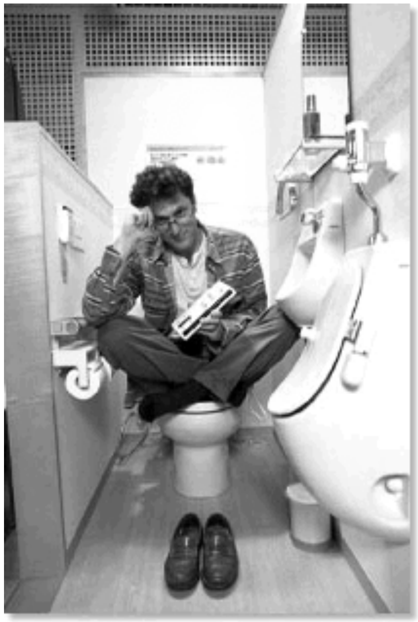
```
slt  $t0,$s0,$s1 #   signed compare
sltu $t1,$s0,$s1 # unsigned compare
```

- $t0: $-4_{ten} < 1,000,000,000_{ten}$?
- $t1: $4,294,967,292_{ten} < 1,000,000,000_{ten}$?

Administrivia

- Readings: (4.1,4.2,4.3) 3.7, 4.8 (skip HW)
- 3rd homework: Due Tonight 7PM
- 4th homework: Due 2/17 7PM
  - Exercises 3.21, 4.3, 4.7, 4.14, 4.15, 4.31
- 2nd project: MIPS Disassembler Due Wed. 2/17 7PM
- Book is a valuable reference!
  - Appendix A (as know more, easier to refer)
  - Back inside cover has useful MIPS summary: instructions, descriptions, definitions, formats, opcodes, examples

"What's This Stuff Good For?"



Remote Diagnosis: "NeoRest ExII," a high-tech toilet features microprocessor-controlled seat warmers, automatic lid openers, air deodorizers, water sprays and blow-dryers that do away with the need for toilet tissue. About 25 percent of new homes in Japan have a "washlet," as these toilets are called. Toto's engineers are now working on a model that analyzes urine to determine blood-sugar levels in diabetics and then automatically sends a daily report, by modem, to the user's physician.

One Digital Day, 1998
www.intel.com/onedigitalday

Two's complement shortcut: Negation

- Invert every 0 to 1 and every 1 to 0, then add 1 to the result
  - Sum of number and its inverted representation must be $111...111_{two}$
  - $111...111_{two} = -1_{ten}$
  - Let x' mean the inverted representation of x
  - Then x + x' = -1 => x + x' + 1 = 0 => x' + 1 = -x
- Example: -4 to +4 to -4

x : $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$
x': $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{two}$
+1: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_{two}$
()': $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{two}$
+1: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$

Two's complement shortcut: Sign extension

- Convert two's complement number represented in n bits to more than n bits
  - e.g., 16-bit immediate field converted to 32 bits before adding to 32-bit register in `addi`
- Simply replicate the most significant bit (sign bit) of smaller quantity to fill new bits
  - 2's comp. positive number has infinite 0s to left
  - 2's comp. negative number has infinite 1s to left
  - Bit representation hides most leading bits; sign extension restores some of them
  - 16-bit -4_{ten} to 32-bit: $1111\ 1111\ 1111\ 1100_{two}$

$\boxed{1111\ 1111\ 1111\ 1111}\ 1111\ 1111\ 1111\ 1100_{two}$

More Compact Representation v. Binary?

- Shorten numbers by using higher base than binary that converts easily into binary
  - almost all computer data sizes are multiples of 4, so use <u>hexadecimal</u> (base 16) numbers
  - base 16 a power of 2, convert by replacing group of 4 binary digits by 1 hex digit; and vice versa

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: from before, $1\ 000\ 000\ 000_{ten}$ is

0011	1011	1001	1010	1100	1010	0000	0000	two
3	b	9	a	c	a	0	0	hex

C uses notation $\boxed{0x}$3b9aca00

Logical Operations

- Operations on less than full words
  - Fields of bits or individual bits
- Think of word as 32 bits vs. 2's comp. integers or unsigned integers
- Need to extract bits from a word, insert bits into a word
- Extracting via Shift instructions
  - C operators: << (shift left), >> (shift right)
- Inserting via And/Or instructions
  - C operators: & (bitwise AND), | (bitwise OR)

Shift Instructions

- Move all the bits in a word to the left or right, filling the emptied bits with 0s
- Before and after shift left 8 of $s0 ($16):

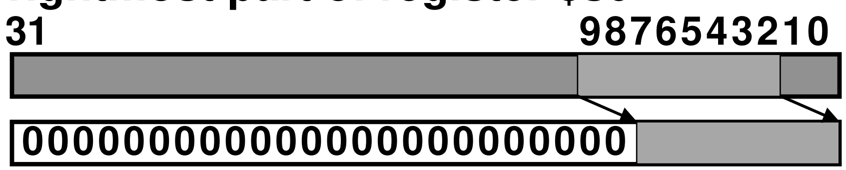
  0000 0000 0000 0000 0000 0000 0000 1101$_{two}$

  0000 0000 0000 0000 0000 1101 0000 0000$_{two}$
- MIPS instructions
  - shift left logical (`sll`) and shift right logical (`srl`)
  - `sll $s0,$s0,8 # $s0 = $s0 << 8 bits`
  - Register Format, using shamt (shift amount)!

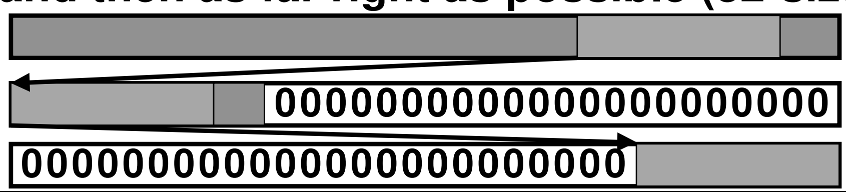
0	0	16	16	8	0
op	rs	rt	rd	shamt	funct

Extracting a field of bits

- Extract bit field from bit 9 (left bit no.) to bit 2 (size=8 bits) of register $s1, place in rightmost part of register $s0



- Shift field as far left as possible (31-bit no.) and then as far right as possible (32-size)




```
sll $s0,$s1,22 #8bits to left end (31-9)
srl $s0,$s0,24 #8bits to right end(32-8)
```


And instruction

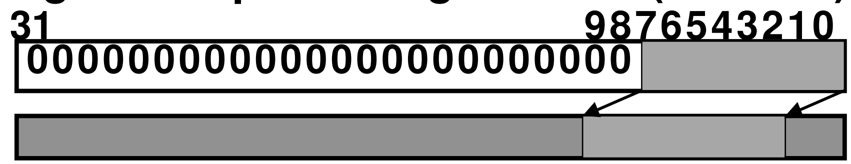
- AND: bit-by-bit operation leaves a 1 in the result only if both bits of the operands are 1. For example, if registers $t1 and $t2
  - 0000 0000 0000 0000 0000 1101 0000 0000$_{two}$
  - 0000 0000 0000 0000 0011 1100 0000 0000$_{two}$
- After executing MIPS instruction
  - `and $t0,$t1,$t2 # $t0 = $t1 & $t2`
- Value of register $t0
  - 0000 0000 0000 0000 0000 1100 0000 0000$_{two}$
- AND can force 0s where 0 in the bit pattern
  - Called a "mask" since mask "hides" bits

Or instruction

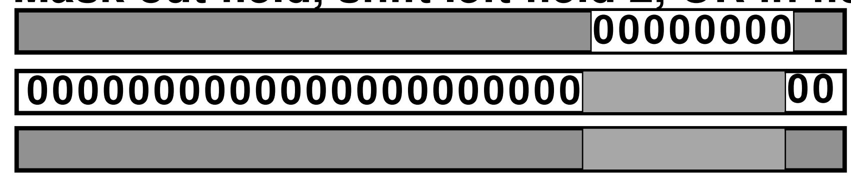
- OR: bit-by-bit operation leaves a 1 in the result if either bit of the operands is 1. For example, if registers $t1 and $t2
  - 0000 0000 0000 0000 00$\boxed{00\ 11}$0$\boxed{1}$ 0000 0000_{two}
  - 0000 0000 0000 0000 00$\boxed{11\ 11}$0$\boxed{0}$ 0000 0000_{two}
- After executing MIPS instruction
  - `or $t0,$t1,$t2 # $t0 = $t1 & $t2`
- Value of register $t0
  - 0000 0000 0000 0000 00$\boxed{11\ 11}$0$\boxed{1}$ 0000 0000_{two}
- OR can force 1s where 1 in the bit pattern
  - If 0s in field of 1 operand, can insert new value

Inserting a field of bits (almost OK)

- Insert bit field into bit 9 (left bit no.) to bit 2 (size=8 bits) of register $s1 from rightmost part of register $s0 (rest is 0)



- Mask out field, shift left field 2, OR in field




```
andi $s1,$s1,0xfc03 # mask out 9-2
sll  $t0,$s0,2      # field left 2
or   $s1,$s1,$t0    # OR in field
```


Sign Extension of Immediates

- `addi` and `slti`: deal with signed numbers, so immediates sign extended
- Branch and data transfer address fields are sign extended too
- `addiu` and `sltiu` also sign extend!
  - `addiu` really to avoid overflow; `sltiu` why?
- `andi` and `ori` work with unsigned integers, so immediates padded with leading 0s
  - `andi` won't work as mask in upper 16 bits

```
addiu $t1,$zero,0xfc03 #32b mask in $t1
and   $s1,$s1,$t1      # mask out  9-2
sll   $t0,$s0,2        # field left 2
or    $s1,$s1,$t0      # OR in field
```


Summary: 12 new instructions (with formats)

- Unsigned (no overflow) arithmetic: `addu` (R), `subu` (R), `addiu` (I)
- Unsigned compare: `sltu` (R), `sltiu` (I)
- Logical operations: `and` (R), `or` (R), `andi` (I), `ori` (I), `sll` (R), `srl` (R)
- Handle overflow exception: `EPC` register and `mfc0` (R)

Example: show C, assembly, machine

- **Convert C code: Bit Fields in C**

```
struct {
    unsigned int ready:  1; /* bit 31 */
    unsigned int enable: 1; /* bit 30 */
} rec; /* $S0 */
rec.enable = 1;
rec.ready = 0;
printf("%d %d", rec.enable, rec.ready);
...
```


Patterson Spring 99 ©UCB

"And in Conclusion..." 1/1

- Handling case when number is too big for representation (overflow)
- Representing negative numbers (2's complement)
- Comparing signed and unsigned integers
- Manipulating bits within registers: shift and logical instructions
- Next time: characters, floating point numbers

Patterson Spring 99 ©UCB