CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

- Review
- C Functions and Calling conventions
- Integers and Two's Complement
- Administrivia
- Technology Break
- Real Numbers and Floating Point
- Summary

Review from Last Lecture

- C is function oriented; code reuse via functions
  - Jump and link (*jal*) invokes, jump register (*jr $ra*) returns
  - Registers *$a0-$a3* for arguments, *$v0-$v1* for return values
- Stack for spilling registers, nested function calls, C local (automatic) variables
- Pointers/pointer arithmetic to reduce array overhead
  - No pointers to automatic data!

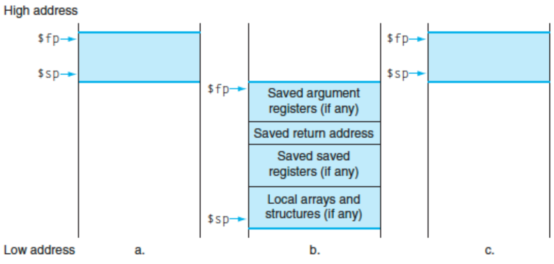
Review from Last Lecture

- C is function oriented; code reuse via functions
  - Jump and link (jal) invokes, jump register (jr $ra) returns
  - Registers $a0-$a3 for arguments, $v0-$v1 for return values
- Stack for spilling registers, nested function calls, C local (automatic) variables
- Pointers/pointer arithmetic to reduce array overhead
  - No pointers to automatic data!
- Registers selectively saved/restored on call
  - Saved registers $s0-$s7; temporary regs $t0-$t9 not saved
- C splits memory into text, static, heap, stack, with registers dedicated to support: $gp, $sp, $fp

Allocating space on stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits.
  - *Static* variables exist across exits from and entries to procedures
- Can use stack for automatic (local) variables that don't fit in registers
- *procedure frame* or *activation record*: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer ($fp) to point to first word of frame

Stack before, during, after call



Optimized Function Convention

- To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - $ra, $sp, $gp, $fp, “saved registers” $s0 - $s7
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Return value registers $v0,$v1, Argument registers $a0 - $a3, “temporary registers” $t0 - $t9

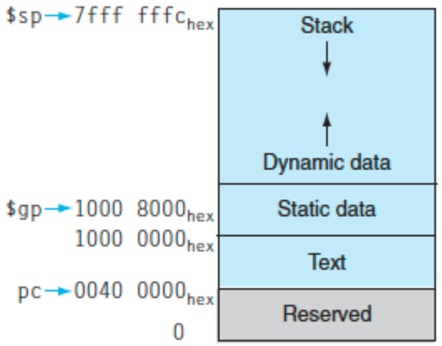
Register Numbering

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

Where is stack in memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : 7fff fffc_{hex}
- MIPS programs (*text segment*) in low end
  - 0040 0000_{hex}
- *static data segment* (constants and other static variables) above text for static variables
  - MIPS convention *global pointer* ($gp) points to static
- Heap above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation



Number Representation

- Value of i-th digit is $d \times Base^i$ where i starts at 0 and increases from right to left:
- $123_{10} = 1_{10} \times 10_{10}^2 + 2_{10} \times 10_{10}^2 + 3_{10} \times 10_{10}^0$
  $= 1 \mathrm{x} 100_{10} + 2 \mathrm{x} 10_{10} + 3 \mathrm{x} 1_{10}$
  $= 100_{10} + 20_{10} + 3_{10}$
  $= 123_{10}$
- Binary (Base 2), Octal (Base 8), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use 1_{two}, 8_{oct}, 5_{ten}, 10_{hex} to be clearer (vs. 1_2, 4_8, 5_{10}, 10_{16})

Number Representation

- Hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- $\mathrm{FFF}_{hex} = 15_{ten} \mathrm{x}\ 16_{ten}^2 + 15_{ten} \mathrm{x}\ 16_{ten}^1 + 15_{ten} \mathrm{x}\ 16_{ten}^0$
  $= 3840_{ten} + 240_{ten} + 15_{ten}$
  $= 4095_{ten}$
- $1111\ 1111\ 1111_{two} = 7777_{oct} = \mathrm{FFF}_{hex} = 4095_{ten}$
- May put blanks every group of binary, octal, or hexadecimal digits to make it easier to parse, like commas in decimal

Signed and Unsigned Integers

- C, C++, and Java has *signed integers*, e.g., 7, -255:

  `int x, y, z;`
- C, C++ also has *unsigned integers*, which are used for addresses
- 32-bit word can represent 2^{32} binary numbers
- Unsigned integers in 32 bit word represent 0 to 2^{32}-1 (4,294,967,295)

Unsigned Integers

0000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = 0$_{ten}$
0000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = 1$_{ten}$
0000 0000 0000 0000 0000 0000 0000 0010$_{two}$ = 2$_{ten}$
... ...
0111 1111 1111 1111 1111 1111 1111 1101$_{two}$ = 2,147,483,645$_{ten}$
0111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = 2,147,483,646$_{ten}$
0111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = 2,147,483,647$_{ten}$
1000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = 2,147,483,648$_{ten}$
1000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = 2,147,483,649$_{ten}$
1000 0000 0000 0000 0000 0000 0000 0010$_{two}$ = 2,147,483,650$_{ten}$
... ...
1111 1111 1111 1111 1111 1111 1111 1101$_{two}$ = 4,294,967,293$_{ten}$
1111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = 4,294,967,294$_{ten}$
1111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = 4,294,967,295$_{ten}$

Signed Integers and Two's Complement Representation

- Signed integers in C; want ½ numbers <0, want ½ numbers >0, and want one 0
- *Two's complement* treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} (–2,147,483,648) to 2^{31}-1 (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which worse
  - Every computers uses two's complement today
- *Most significant bit* (leftmost) called *sign bit*, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant, bit 0 is least significant

Two's Complement Integers

Sign Bit

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

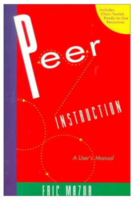
The Rules (delay dopamine squirt until break)

When is Midterm, Final?

- To reduce time pressure, 3 hours for 1.5 hour midterm
- **Midterm Exam Wednesday October 6, 6 – 9PM, Pimental 1**
- **Final Exam Monday December 13, 8 – 11AM, Location TBD**

Peer Instruction

- Increase real-time learning in lecture, test understanding of concepts vs. details
  mazur-www.harvard.edu/education/pi.phtml
- As complete a “segment” ask multiple choice question
  - 1-2 minutes: decide yourself, vote
  - 2-3 minutes: discuss in pairs, then team vote; flash cards
    - Try to convince partner; learn by teaching



Peer Instruction

- Suppose we had a 5 bit word. What integers can be represented in two’s complement?

A. -32 to +31
B. -31 to +32
C. 0 to +31
D. -16 to +15
E. -15 to +15
F. -15 to +16

Question?

```
static int *p;
int leaf (int g, int h,
   int i, int j)
{
   int f; p = &f;
   f = (g + h) – (i + j);
   return f;
}
int main(void) { int x;
   …
   x = leaf(1,2,3,4);
   …
   x = leaf(3,4,1,2);
   …
   printf(”%d\n”,p);
}
```

- What will a.out do?

A. Print -4
B. Print 4
C. a.out will crash
D. None of the above

Agenda

- Review
- C Functions and Calling conventions
- Integers and Two’s Complement
- Administrivia
- Technology Break
- Real Numbers and Floating Point
- Summary

MIPS Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called logical operations

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	**and**
Bit-by-bit OR	|	|	**or**
Bit-by-bit NOT	~	~	**nor**
Shift left	<<	<<	**sll**
Shift right	>>	>>>	**srl**

Bit-by-bit Definition

Operation	Input	Input	Output
AND	0	0	0
AND	0	1	0
AND	1	0	0
AND	1	1	1
OR	0	0	0
OR	0	1	1
OR	1	0	1
OR	1	1	1
NOR	0	0	0
NOR	0	1	1
NOR	1	0	1
NOR	1	1	1

Examples

- If register \$t2 contains and

0000 0000 0000 0000 0000 11011100 0000_{two}

- Register \$t1 contains

0000 0000 0000 0000 0011 1100 0000 0000_{two}

- What is value of \$t0 after:

1. and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2

2. or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2

3. nor \$t0,\$t1,\$zero # reg \$t0 = ~ (reg \$t1 | 0)

Shifting

- Shift left logical moves n bits to the left (insert 0s into empty bits)
  - Same as multiplying by 2^n for two's complement num.
- For example, if register \$s0 contained

0000 0000 0000 0000 0000 0000 0000 $1001_{two} = 9_{ten}$

- If executed sll \$s0, \$s0, 4, result is:

0000 0000 0000 0000 0000 0000 1001 $0000_{two} = 144_{ten}$

- And $9_{ten} \times 2_{ten}{}^4 = 9_{ten} \times 16_{ten} = 144_{ten}$
- Shift right logical moves n bits to the right (insert 0s into empty bits)
  - NOT same as dividing by 2n (negative numbers fail)

Impact of Signed and Unsigned Integers on Instruction Sets

- What (if any) instructions affected?
  - Load word, store word?
  - branch equal, branch not equal?
  - and, or, sll, srl?
  - add, sub, mult, div?
  - set less than?

What if result of operation doesn't fit in 32 bits?

- Called *overflow*: calculate too big a number to represent within a word
- Unsigned numbers: 1 + 4,294,967,295 (2^{32}-1)
- Signed numbers: 1 + 2,147,483,647 (2^{31}-1)

Answer Depends on Language

- C unsigned number arithmetic ignores overflow (arithmetic modulo 2^{32})

  1 + 4,294,967,295 =

- C signed number arithmetic also ignores overflow

  1 + 2,147,483,647 (2^{31}-1) =

- Other languages want overflow signal on signed numbers (e.g., Fortran)
- What's a computer architect to do?

MIPS Solution: offer both

- Instructions that overflow:
  - add, sub, mult, div, addi, multi, divi
- Instructions that don't overflow called "unsigned" (but really means no overflow):
  - addu, subu, multu, divu, addiu, multiu, diviu
- Given semantics of C, always use unsigned versions
- Note: slt and slti do signed comparisons, while sltu and sltiu do unsigned comparisons
  - Nothing to do with overflow
  - When would get different answer for slt vs. sltu?

What about Real Numbers?

- Normalized scientific notation (aka standard form or exponential notation):
  - r x E^i, *E* is where exponent (usually 10), *i* is a positive or negative integer, *r* is a real number ≥ 1.0, < 10
  - 61 is 6.10 x 10^2, 0.000061 is 6.10 x10^{-5}
- Computers version of normalized scientific notation called *Floating Point* notation
- r x E^i, *E* where is exponent (2), *i* is a positive or negative integer, *r* is a real number ≥ 1.0, < 2

Floating Point Numbers

- 32-bit word has 2^{32} patterns, so must be approximation of real numbers ≥ 1.0, < 2
- IEEE 754 Floating Point Standard:
  - 1 bit for *sign (s)* of floating point number
  - 8 bits for *exponent (E)*
  - 23 bits for *fraction (F)* (get 1 extra bit of precision if leading 1 is implicit)

$(-1)^s$ x (1 + F) x 2^E

- Can represent from 2.0 x 10^{-38} to 2.0 x 10^{38}

Floating Point Numbers

- What about bigger or smaller numbers?
- IEEE 754 Floating Point Standard: *Double Precision* (64 bits)
  - 1 bit for *sign (s)* of floating point number
  - 11 bits for *exponent (E)*
  - 52 bits for *fraction (F)* (get 1 extra bit of precision if leading 1 is implicit)

$(-1)^s$ x (1 + F) x 2^E

- Can represent from 2.0 x 10^{-308} to 2.0 x 10^{308}
- 32 bit format called *Single Precision*

More Floating Point

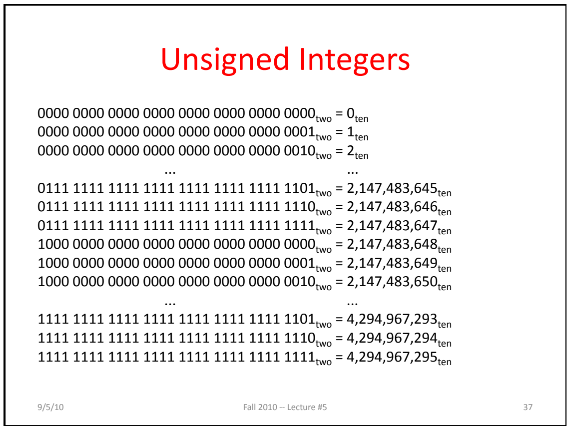
- What about 0?
  - Bit pattern all 0s means 0, so no implicit leading 1
- What if divide 1 by 0?
  - Can get infinity symbols +∞, -∞
  - Sign bit 0 or 1, largest exponent, 0 in fraction
- What if do something stupid? (∞ – ∞, 0 ÷ 0)
  - Can get special symbols NaN for Not-a-Number
  - Sign bit 0 or 1, largest exponent, not zero in fraction
- What if result is too big? (2x10^{308} x 2x10^2)
  - Get *overflow* in exponent, alert programmer!
- What if result is too small? (2x10^{-308} ÷ 2x10^2)
  - Get *underflow* in exponent, alert programmer!

MIPS Floating Point Instructions

- C, Java has single precision (`float`) and double precision (`double`) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Addition single precision:
    Fl. Pt. Addition double precision:
  - Fl. Pt. Subtraction single precision:
    Fl. Pt. Subtraction double precision:
  - Fl. Pt. Multiplication single precision:
    Fl. Pt. Multiplication double precision:
  - Fl. Pt. Divide single precision:
    Fl. Pt. Divide double precision:

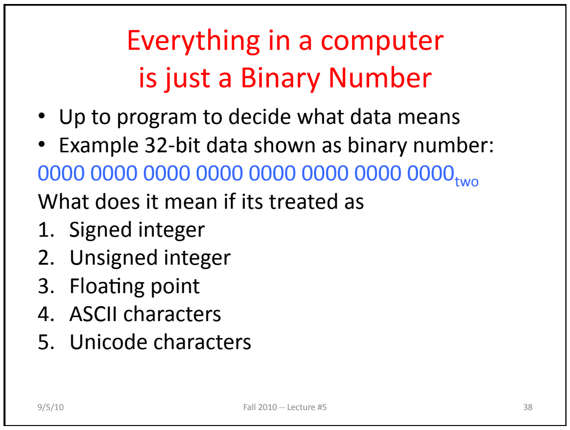
MIPS Floating Point Instructions

- C, Java has single precision (`float`) and double precision (`double`) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Comparison single precision:
    Fl. Pt. Comparison double precision:
  - Fl. Pt. branch:
- Since rarely mix integers and Fl. Pt., MIPS has separate registers for floating-point operations: \$f0, \$f1, …, \$f31
  - Double precision uses adjacent even-odd pairs of registers:
  - \$f0 and \$f1, \$f2 and \$f3, \$f4 and \$f5, …, \$f30 and \$f31
- Need data transfer instructions for these new registers
  - lwc1 (load word), swc1 (store word)
  - Double precision uses two lwc1 instructions, two swc1 instructions



Unsigned Integers

0000 0000 0000 0000 0000 0000 0000 0000$_{\text{two}}$ = 0$_{\text{ten}}$
0000 0000 0000 0000 0000 0000 0000 0001$_{\text{two}}$ = 1$_{\text{ten}}$
0000 0000 0000 0000 0000 0000 0000 0010$_{\text{two}}$ = 2$_{\text{ten}}$
... ...
0111 1111 1111 1111 1111 1111 1111 1101$_{\text{two}}$ = 2,147,483,645$_{\text{ten}}$
0111 1111 1111 1111 1111 1111 1111 1110$_{\text{two}}$ = 2,147,483,646$_{\text{ten}}$
0111 1111 1111 1111 1111 1111 1111 1111$_{\text{two}}$ = 2,147,483,647$_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0000$_{\text{two}}$ = 2,147,483,648$_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0001$_{\text{two}}$ = 2,147,483,649$_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0010$_{\text{two}}$ = 2,147,483,650$_{\text{ten}}$
... ...
1111 1111 1111 1111 1111 1111 1111 1101$_{\text{two}}$ = 4,294,967,293$_{\text{ten}}$
1111 1111 1111 1111 1111 1111 1111 1110$_{\text{two}}$ = 4,294,967,294$_{\text{ten}}$
1111 1111 1111 1111 1111 1111 1111 1111$_{\text{two}}$ = 4,294,967,295$_{\text{ten}}$



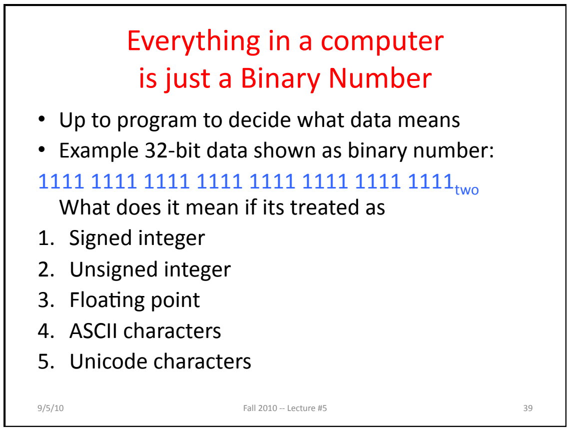
Everything in a computer is just a Binary Number

- Up to program to decide what data means
- Example 32-bit data shown as binary number:

0000 0000 0000 0000 0000 0000 0000 0000$_{\text{two}}$

What does it mean if its treated as

1. Signed integer
2. Unsigned integer
3. Floating point
4. ASCII characters
5. Unicode characters



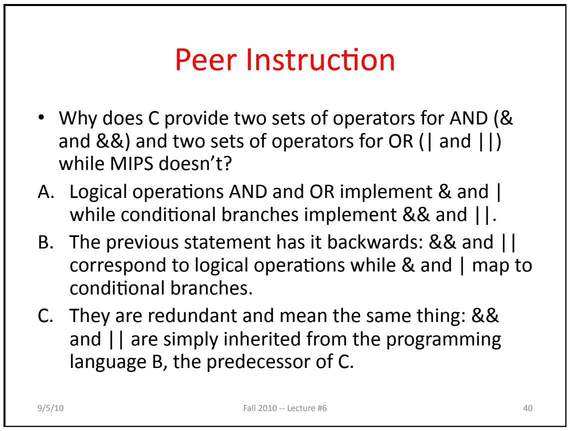
Everything in a computer is just a Binary Number

- Up to program to decide what data means
- Example 32-bit data shown as binary number:

1111 1111 1111 1111 1111 1111 1111 1111$_{\text{two}}$

What does it mean if its treated as

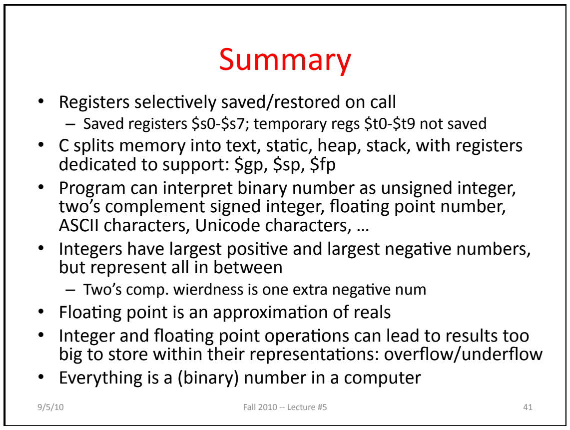
1. Signed integer
2. Unsigned integer
3. Floating point
4. ASCII characters
5. Unicode characters



Peer Instruction

- Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't?

A. Logical operations AND and OR implement & and | while conditional branches implement && and ||.
B. The previous statement has it backwards: && and || correspond to logical operations while & and | map to conditional branches.
C. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.



Summary

- Registers selectively saved/restored on call
  - Saved registers \$s0-\$s7; temporary regs \$t0-\$t9 not saved
- C splits memory into text, static, heap, stack, with registers dedicated to support: \$gp, \$sp, \$fp
- Program can interpret binary number as unsigned integer, two's complement signed integer, floating point number, ASCII characters, Unicode characters, …
- Integers have largest positive and largest negative numbers, but represent all in between
  - Two's comp. wierdness is one extra negative num
- Floating point is an approximation of reals
- Integer and floating point operations can lead to results too big to store within their representations: overflow/underflow
- Everything is a (binary) number in a computer