CS 61C:
Great Ideas in Computer Architecture
Functions and Numbers

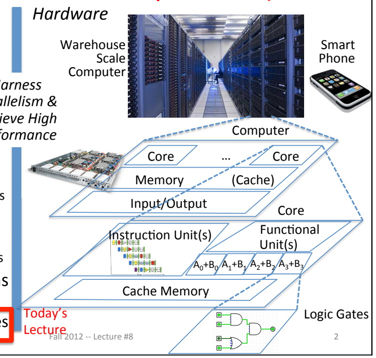
Instructor:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/sp12

New-School Machine Structures (It's a bit more complicated!)

Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



Agenda

- Review
- Functions
- Administrivia
- Everything is a Number
- And in Conclusion, …

Agenda

- Review
- Functions
- Administrivia
- Everything is a Number
- And in Conclusion, …

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

Register Allocation and 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

MIPS Function Call Instructions

- Invoke function: *jump and link* instruction (`jal`)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of following instruction in register `$ra`

```
jal ProcedureAddress
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register

```
jr $ra
```


Notes on Functions

- Calling program (*caller*) puts parameters into registers `$a0-$a3` and uses `jal X` to invoke X (*callee*)
- Must have register in computer with address of currently executing instruction
  - Instead of Instruction Address Register (better name), historically called *Program Counter* (*PC*)
  - It's a program's counter, it doesn't count programs!
- `jr $ra` puts address inside `$ra` into PC
- What value does `jal X` place into `$ra`? **Next PC**
  PC + 4

Where Are Old Register Values Saved to Restore Them After Function Call

- Need a place to save old values before call function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- `$sp` is the *stack pointer* in MIPS
- Convention is grow from high to low addresses
  - Push decrements `$sp`, Pop increments `$sp`
- (28 out of 32, 4 left!)

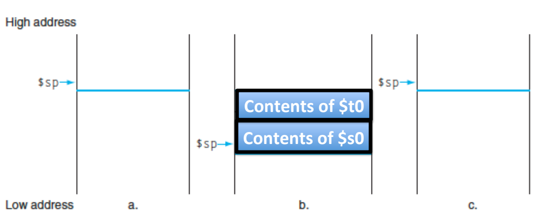
Example

```
int leaf_example
  (int g, int h, int i, int j)
{
  int f;
  f = (g + h) – (i + j);
  return f;
}
```

- Parameter variables `g`, `h`, `i`, and `j` in argument registers `$a0`, `$a1`, `$a2`, and `$a3`, and `f` in `$s0`
- Assume need one temporary register `$t0`

Stack Before, During, After Function

- Need to save old values of `$s0` and `$t0`



MIPS Code for leaf_example

```
leaf_example:
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
  add $t0,$a2,$a3   # $t0 = i + j
  sub $v0,$s0,$t0   # return value (g + h) – (i + j)
  lw $s0, 0($sp)    # restore register $s0 for caller
  lw $t0, 4($sp)    # restore register $t0 for caller
  addi $sp,$sp,8    # adjust stack to delete 2 items
  jr $ra            # jump back to calling routine
```

What will the printf output?

- ☐ Print -4
- ☐ Print 4
- ☐ a.out will crash
- ☐ None of the above

Really?

```
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 If a Function Calls a Function? Recursive Function Calls?

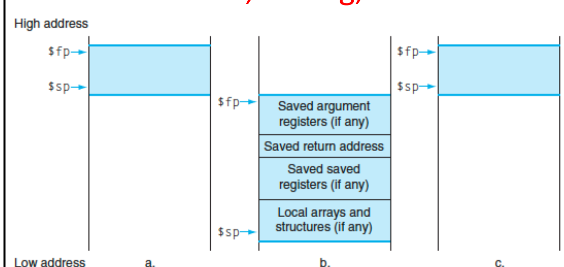
- Would clobber values in `$a0` to `$a3` and `$ra`
- What is the solution?

Student Roulette?

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
- 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
- (29 of 32, 3 left!)

Stack Before, During, After Call



Recursive Function Factorial

```
int fact (int n)
{
  if (n < 1) return (1);
     else return (n * fact(n-1));
}
```


Recursive Function Factorial

```
Fact:
  # adjust stack for 2 items
  addi $sp,$sp,-8
  # save return address
  sw $ra, 4($sp)
  # save argument n
  sw $a0, 0($sp)
  # test for n < 1
  slti $t0,$a0,1
  # if n >= 1, go to L1
  beq $t0,$zero,L1
  # Then part (n==1) return 1
  addi $v0,$zero,1
  # pop 2 items off stack
  addi $sp,$sp,8
  # return to caller
  jr $ra
```

```
L1:
  # Else part (n >= 1)
  # arg. gets (n – 1)
  addi $a0,$a0,-1
  # call fact with (n – 1)
  jal fact
  # return from jal: restore n
  lw $a0, 0($sp)
  # restore return address
  lw $ra, 4($sp)
  # adjust sp to pop 2 items
  addi $sp, $sp,8
  # return n * fact (n – 1)
  mul $v0,$a0,$v0
  # return to the caller
  jr $ra
```

mul is a pseudo instruction

Optimized Function Convention

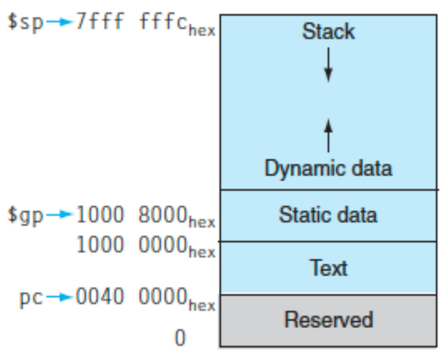
To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

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`

Where is the 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
  - (30 of 32, 2 left! – will see when talk about OS)
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation



Register Allocation and 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

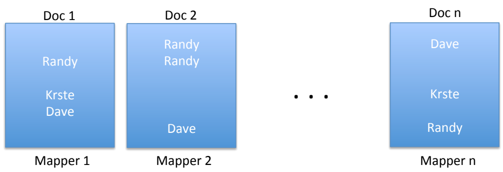
Which statement is FALSE?

- ☐ MIPS uses `jal` to invoke a function and `jr` to return from a function
- ☐ `jal` saves PC+1 in `%ra`
- ☐ The callee can use temporary registers (`%ti`) without saving and restoring them
- ☐ The caller can rely on save registers (`%si`) without fear of callee changing them

Agenda

- Review
- Functions
- **Administrivia**
- Everything is a Number
- And in Conclusion, …

Project #1



– Occurrences A_{Randy}: 4, A_{Krste}: 2, A_{Dave}: 3
– Target word = Dave
  • C_{Randy}: 4, C_{Krste}: 2
  • Co-occurrence with Krste: $C_{Krste} * (\log(C_{Krste}))^3 / A_{Krste} = 2 * (\log(2))^3 / 2 = 0.03$
  • Co-occurrence with Randy: $C_{Randy} * (\log(C_{Randy}))^3 / A_{Randy} = 4 * (\log(4))^3 / 4 = 0.22$
– Your task: compute and sort co-occurrence for n-gram (for small n) word sequences, using several distance weighting functions

9/11/12 Fall 2012 -- Lecture #8 38

Project #1

Document:
"It was the best of times, it was the worst of times"

- 1-grams: it was the best of times worst
  – it-was: distance 1, it-the: distance 2, it-best: distance 3, …
- 2-grams: it was, was the, the best, best of, of times, times it, was the, the worst, worst of, of times
  – it was—was the: distance 1, it was—the best: distance 2, it was—best of: distance 3, …
- 3-grams: it was the, was the best, the best of, best of times, of times it, times it was, was the worst, the worst of, worst of times
- etc.

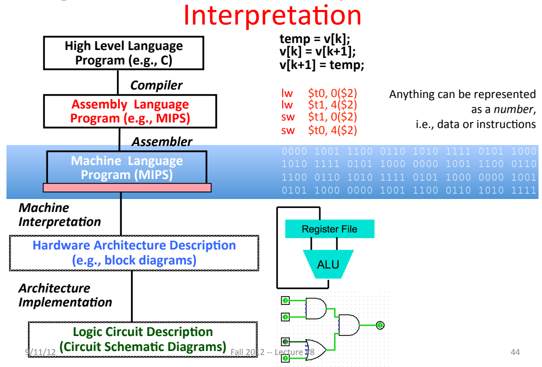
9/12/12 Fall 2012 -- Lecture #8 39

Agenda

- Review
- Functions
- Administrivia
- Everything is a Number
- And in Conclusion, …

9/11/12 Fall 2012 -- Lecture #8 43

Big Idea #1: Levels of Representation/ Interpretation



9/11/12 Fall 2012 -- Lecture #8 44

Key Concepts

- Inside computers, everything is a number
- But everything is of a fixed size
  – 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …
- Integer and floating point operations can lead to results too big to store within their representations: *overflow/underflow*

9/11/12 Fall 2012 -- Lecture #8 45

Number Representation

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

9/11/12 Fall 2012 -- Lecture #8 46

Number Representation

- Hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- $FFF_{hex} = 15_{ten} \times 16_{ten}^2 + 15_{ten} \times 16_{ten}^1 + 15_{ten} \times 16_{ten}^0$
  $= 3840_{ten} + 240_{ten} + 15_{ten}$
  $= 4095_{ten}$
- $1111\ 1111\ 1111_{two} = FFF_{hex} = 4095_{ten}$
- 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 have *signed integers*, e.g., 7, -255:
  `int x, y, z;`
- C, C++ also have *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 are worse
  – Every computers uses two's complement today
- *Most significant bit* (leftmost) is the *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

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

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

- ☐ -32 to +31
- ☐ 0 to +31
- ☐ -16 to +15
- ☐ -15 to +16

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	1
NOR	0	1	0
NOR	1	0	0
NOR	1	1	0

Examples

- If register \$t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2

Examples

- If register \$t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2
  0000 0000 0000 0000 0000 1100 0000 0000_{two}

Examples

- If register \$t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2

Examples

- If register \$t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2
  0000 0000 0000 0000 0011 1101 1100 0000_{two}

Examples

- If register \$t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  nor \$t0,\$t1,\$zero # reg \$t0 = ~ (reg \$t1 | 0)

Examples

- If register \$t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  nor \$t0,\$t1,\$zero # reg \$t0 = ~ (reg \$t1 | 0)
  1111 1111 1111 1111 1100 0011 1111 1111_{two}

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 number
- 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)

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register \$s0 contained
  0000 0000 0000 0000 0000 0000 0001 $1001_{two} = 25_{ten}$
- If executed sra \$s0, \$s0, 4, result is:

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register \$s0 contained
  0000 0000 0000 0000 0000 0000 0001 $1001_{two} = 25_{ten}$
- If executed sra \$s0, \$s0, 4, result is:
  0000 0000 0000 0000 0000 0000 0000 $0001_{two} = 1_{ten}$

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register \$s0 contained
  1111 1111 1111 1111 1111 1111 1110 $0111_{two} = -25_{ten}$
- If executed sra \$s0, \$s0, 4, result is:

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained

  1111 1111 1111 1111 1111 1111 1110 0111$_{two}$= -25$_{ten}$
- If executed sra $s0, $s0, 4, result is:

  1111 1111 1111 1111 1111 1111 1111 1110$_{two}$= -2$_{ten}$
- Unfortunately, this is NOT same as dividing by 2n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

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?
  - slti (set less than immediate)?

"And in Conclusion, …"

- 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
- 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. weirdness is one extra negative number
  - Integer (and floating point operations) can lead to results too big to store within their representations: overflow/underflow