# Lecture 7: Computer Arithmetic

- Today's topics:
  - Chapter 2 wrap-up
  - Numerical representations
  - Addition and subtraction
- Reminder: Assignment 3 will be posted by tomorrow

# Compilation Steps

- The front-end: deals mostly with language specific actions
  - Scanning: reads characters and breaks them into tokens
  - Parsing: checks syntax
  - Semantic analysis: makes sure operations/types are meaningful
  - Intermediate representation: simple instructions, infinite registers, makes few assumptions about hw
- The back-end: optimizations and code generation
  - Local optimizations: within a basic block
  - Global optimizations: across basic blocks
  - Register allocation

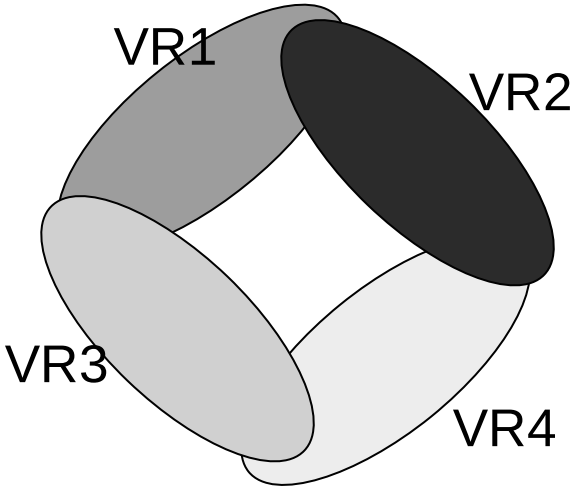
# Dataflow

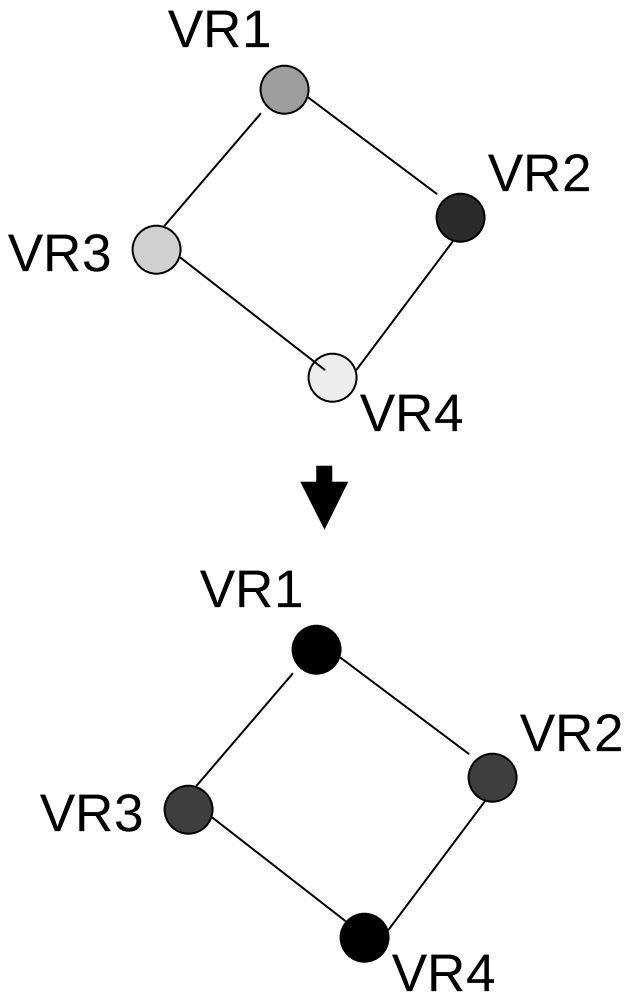
- Control flow graph: each box represents a basic block and arcs represent potential jumps between instructions
- For each block, the compiler computes values that were defined (written to) and used (read from)
- Such dataflow analysis is key to several optimizations: for example, moving code around, eliminating dead code, removing redundant computations, etc.

# Register Allocation

- The IR contains infinite virtual registers – these must be mapped to the architecture's finite set of registers (say, 32 registers)
- For each virtual register, its live range is computed (the range between which the register is defined and used)
- We must now assign one of 32 colors to each virtual register so that intersecting live ranges are colored differently – can be mapped to the famous graph coloring problem
- If this is not possible, some values will have to be temporarily spilled to memory and restored (this is equivalent to breaking a single live range into smaller live ranges)

# Graph Coloring



# High-Level Optimizations

High-level optimizations are usually hardware independent

- Procedure inlining
- Loop unrolling
- Loop interchange, blocking (more on this later when we study cache/memory organization)

# Low-Level Optimizations

- Common sub-expression elimination
- Constant propagation
- Copy propagation
- Dead store/code elimination
- Code motion
- Induction variable elimination
- Strength reduction
- Pipeline scheduling

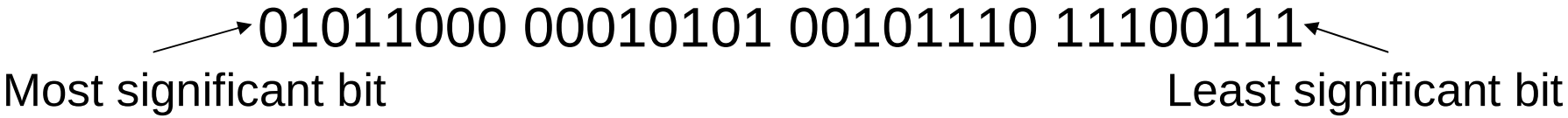
# Saves on Stack

- Caller saved
  - $a0-a3 -- old arguments must be saved before setting new arguments for the callee
  - $ra -- must be saved before the jal instruction over-writes this value
  - $t0-t9 -- if you plan to use your temps after the return, save them note that callees are free to use temps as they please
  - ✘ You need not save $s0-s7 as the callee will take care of them

- Callee saved
  - $s0-s7 -- before the callee uses such a register, it must save the old contents since the caller will usually need it on return
  - local variables -- space is also created on the stack for variables local to that procedure

# Binary Representation

- The binary number




represents the quantity

$0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0}$

- A 32-bit word can represent $2^{32}$ numbers between 0 and $2^{32}-1$

  … this is known as the unsigned representation as we're assuming that numbers are always positive

# ASCII Vs. Binary

- Does it make more sense to represent a decimal number in ASCII?
- Hardware to implement arithmetic would be difficult
- What are the storage needs? How many bits does it take to represent the number 1,000,000,000 in ASCII and in binary?

# ASCII Vs. Binary

- Does it make more sense to represent a decimal number in ASCII?
- Hardware to implement arithmetic would be difficult
- What are the storage needs? How many bits does it take to represent the number 1,000,000,000 in ASCII and in binary?

  In binary: 30 bits ($2^{30}$ > 1 billion)

  In ASCII: 10 characters, 8 bits per char = 80 bits

# Negative Numbers

32 bits can only represent $2^{32}$ numbers – if we wish to also represent negative numbers, we can represent $2^{31}$ positive numbers (incl zero) and $2^{31}$ negative numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} – 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} – 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

# 2's Complement

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

Why is this representation favorable?
Consider the sum of 1 and -2 …. we get -1
Consider the sum of 2 and -1 …. we get +1

This format can directly undergo addition without any conversions!

Each number represents the quantity

$x_{31}\ -2^{31} + x_{30}\ 2^{30} + x_{29}\ 2^{29} + \ldots + x_1\ 2^1 + x_0\ 2^0$

# 2's Complement

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

Note that the sum of a number x and its inverted representation x' always equals a string of 1s (-1).

$x + x' = -1$
$x' + 1 = -x$ … hence, can compute the negative of a number by
$-x = x' + 1$ inverting all bits and adding 1

Similarly, the sum of x and –x gives us all zeroes, with a carry of 1
In reality, $x + (-x) = 2^n$ … hence the name 2's complement

# Example

- Compute the 32-bit 2's complement representations for the following decimal numbers:

  5,  -5, -6

# Example

- Compute the 32-bit 2's complement representations for the following decimal numbers:

  5, -5, -6

  5: 0000 0000 0000 0000 0000 0000 0000 0101
  -5: 1111 1111 1111 1111 1111 1111 1111 1011
  -6: 1111 1111 1111 1111 1111 1111 1111 1010

  Given -5, verify that negating and adding 1 yields the number 5

# Signed / Unsigned

- The hardware recognizes two formats:

  unsigned (corresponding to the C declaration unsigned int)
  -- all numbers are positive, a 1 in the most significant bit just means it is a really large number

  signed (C declaration is signed int or just int)
  -- numbers can be +/- , a 1 in the MSB means the number is negative

This distinction enables us to represent twice as many numbers when we're sure that we don't need negatives

# MIPS Instructions

Consider a comparison instruction:

```
    slt   $t0, $t1, $zero
```

and $t1 contains the 32-bit number  1111 01…01

What gets stored in $t0?

# MIPS Instructions

Consider a comparison instruction:

```
    slt   $t0, $t1, $zero
```

and $t1 contains the 32-bit number  1111 01…01

What gets stored in $t0?

The result depends on whether $t1 is a signed or unsigned number – the compiler/programmer must track this and accordingly use either slt or sltu

```
slt    $t0, $t1, $zero     stores  1 in $t0
sltu   $t0, $t1, $zero     stores  0 in $t0
```

# The Bounds Check Shortcut

- Suppose we want to check if 0 <= x < y
  and x and y are signed numbers (stored in $a1 and $t2)

  The following single comparison can check both conditions

```
sltu  $t0, $a1, $t2
beq $t0, $zero, EitherConditionFails
```

We know that $t2 begins with a 0
If $a1 begins with a 0, sltu is effectively checking the second condition
If $a1 begins with a 1, we want the condition to fail and coincidentally,
sltu is guaranteed to fail in this case

# Sign Extension

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand

- The conversion is simple: take the most significant bit and use it to fill up the additional bits on the left – known as sign extension

So $2_{10}$ goes from 0000 0000 0000 0010 to
0000 0000 0000 0000 0000 0000 0000 0010

and $-2_{10}$ goes from 1111 1111 1111 1110 to
1111 1111 1111 1111 1111 1111 1111 1110

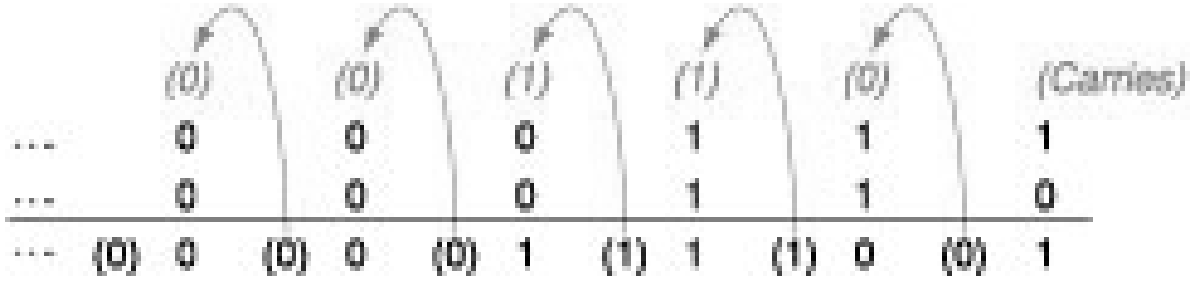
# Alternative Representations

- The following two (intuitive) representations were discarded because they required additional conversion steps before arithmetic could be performed on the numbers
  - sign-and-magnitude: the most significant bit represents +/- and the remaining bits express the magnitude
  - one's complement: -x is represented by inverting all the bits of x

  Both representations above suffer from two zeroes

# Addition and Subtraction

- Addition is similar to decimal arithmetic
- For subtraction, simply add the negative number – hence, subtract A-B involves negating B's bits, adding 1 and A



# Overflows

- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated

- For a signed number, overflow happens when the most significant bit is not the same as every bit to its left
  - when the sum of two positive numbers is a negative result
  - when the sum of two negative numbers is a positive result
  - The sum of a positive and negative number will never overflow

- MIPS allows addu and subu instructions that work with unsigned integers and never flag an overflow – to detect the overflow, other instructions will have to be executed

# Title

- Bullet