Arithmetic Instructions

- Every computer must be able to perform arithmetic operations
- Native integer add and subtract instructions
  - add t0, t1, t2 (t0 = t1 + t2)
  - sub s3, s1, s2 (s3 = s1 - s2)
- A single arithmetic instruction performs one operation on a set of operands
  - Syntax: opcode dest, src1, src2
  - Opcode = mnemonic for operation
  - dest = register for result
  - src1 & src2 = two input values

The Basic Integer Arithmetic Instructions

Operation	Instruction syntax	Meaning
• add	add t2, t0, t1	t2 = t0 + t1
• sub	sub t4, t2, t5	t4 = t2 – t5
• mul	mul t3, t0, t2	t3 = t0 * t2
• div	div t5, t4, t3	t5 = t4 ÷ t3

- sub, div – source register order important

Rules of RISC-V Assembly Language

- All basic instructions perform just *one* operation
- Each line of assembly language code can contain at most *one* instruction
- The number of operands in arithmetic and logic instructions is <u>exactly</u> *three*
- A comment can follow an instruction on the same line (but not the other way around)

Rules (continued)

- Why limit arithmetic instructions to just three operands?
  - Some architectures allow variable number of operands
- Design Principle #1: *Simplicity favors regularity*
  - Regularity makes the implementation simpler
  - Simplicity enables higher performance at lower cost
  - The hardware for a fixed number of operands is simpler than the hardware for variable numbers of operands
- Goal is to balance function and complexity
  - Often there is much overhead in implementing complex operations – detracts from performance
  - Requires compensation/tradeoffs in other areas

How Many Registers?

- Are 32 registers, some of which are not even available for general use, enough to handle all the variables often found in large computer programs?
- Larger numbers of registers complicates the design of the processor and its instruction set
  - Increases its clock cycle which decreases the speed of the processor
- A computer designer must balance between providing lots of registers and a fast processor
- Design principle #2: *Smaller is faster*

Example

- Translate the following into assembly language (use the register association indicated)

sum4 = n1 + n2 + n3 + n4

(t4) (t0) (t1) (t2) (t3)

- Solution:

```
add   t4, t0, t1
add   t4, t4, t2
add   t4, t4, t3
```

t4 register used as an accumulator

Order of operator precedence not an issue in this example.

Register Allocation

- Compilers are written to follow strict rules governing how and when registers are allocated to data values in a program.
- Assembly language programmers have more flexibility in determining which registers to use for data values.
  - Can be a point of confusion if not allocated in an organized way
- Programming note: a register usage list will be required for programming assignments showing how registers are used in your program
  - Don't use registers haphazardly or it may complicate your program

Register Usage List

- Example register usage list

```
# Register Usage List
#   t0 – value of n1
#   t1 – value of n2
#   t2 – value of n3
#   t3 – value of n4
#   t4 – accumulator and final result
```

- Should be located prior to the data segment of your program
- Listing of registers may be by name or by order used in your program
- Once you adopt a methodology, stick to it for best results.

Assembly Language Comments

- Comments are required in your programs.
- *Comments* begin with #
  - They are ignored by the assembler
  - They may be placed to the right of instructions or on a line by themselves
  - Comments terminate at the end of a line and cannot span more than one line
    - no wrap around allowed
- It’s not necessary to comment the obvious
  - Focus on describing why rather than what
  - Unless it’s complicated where the intent is to keep track of where you are in an algorithm

Another Example

- Translate the following expression into assembly statements (use the register assignments indicated)

f = (g + h) – (i + j)

(s4) (s0) (s1) (s2) (s3)

Registers t0 and t1 are used as temporary registers. They can be reused later for other temporary values if necessary.

```
add  t0, s0, s1
add  t1, s2, s3
sub  s4, t0, t1
```

Order of operator precedence is important in this example. This is a semantic issue.

Writing a Simple program

(How to write a simple program)

```
// C++ Program

int main()
{
   int num1 = 5;
   int num2 = 9;
   int num3 = 0;

   num3 = num1 + num2;

   return 0;
}
```

Note: Java and C++ define int as 4 bytes which is a word.

```
# Assembly Language Program

        .data
num1:   .word  5
num2:   .word  9
num3:   .word  0
        .text
main:
               lui   s0, 0x10010
               lw    t0, 0(s0)
               lw    t1, 4(s0)
               add   t2, t0, t1
               sw    t2, 8(s0)
exit:
        ori   a7, zero, 10
        ecall
```

Adding Register Usage to Program

```
# Assembly Language Program

# Header comments

# Register Usage
#  s0 = memory base address
#  t0 = num1
#  t1 = num2
#  t2 = num3 (sum of t0 & t1)
#  a7 = system call


        .data
num1:    .word  5
num2:    .word  9
num3:    .word  0
```

```
# continuation of program

        .text
main:
        lui   s0, 0x10010
        lw    t0, 0(s0)
        lw    t1, 4(s0)
        add   t2, t0, t1
        sw    t2, 8(s0)
exit:
    ori   a7, zero, 10
    ecall
```