LECTURE 2

Review 1 – Binary Math and Assembly

BINARY MATH

In this section, we review

- Binary to decimal conversions and vice versa
- IEEE 754 Floating point representations
- Binary Arithmetic

Decimal representation of binary numbers

- Starting from the least significant bit, multiply the digits of the binary number with increasing powers of 2.
- LSB's multiplied with 2^0, the next bit by 2^1….
- Add the products.

Decimal representation of binary numbers

- 110010

5	4	3	2	1	0
2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0

$= 2^5 + 2^4 + 2^1$

$= 32 + 16 + 2$

$=50$

Decimal representation of binary numbers

- 101100

5	4	3	2	1	0
2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	0

$= 2^5 + 2^3 + 2^2$

$= 32 + 8 + 4$

$=44$

Decimal representation of binary numbers

- 111101

5	4	3	2	1	0
2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	0	1

$= 2^5 + 2^4 + 2^3 + 2^2 + 2^0$

$= 32 + 16 + 8 + 4 + 1$

$=61$

Binary representation of decimal integers

- Repeatedly divide the decimal number by 2, until the quotient is 0.
- Collect the remainders as you go.
- Write down the remainder from right to the left.

Binary representation of decimal integers

- 28

Quotient	Remainder
14	0
7	0
3	1
1	1
0	1

= 011100

Binary representation of decimal integers

- 45

Quotient	Remainder
22	1
11	0
5	1
2	1
1	0
0	1

= 101101

Binary representation of decimal integers

- 62

Quotient	Remainder
31	0
15	1
7	1
3	1
1	1
0	1

= 111110

BINARY ADDITION

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10 (sum is 0, carry 1)
- 1 + 1 + 1 = 11 (sum is 1, carry is 1)

BINARY ADDITION

- a=00111010, b=01100111

Carry	1	1	1	1	1	1		
	0	0	1	1	1	0	1	0
+	0	1	1	0	0	1	1	1
	1	0	1	0	0	0	0	1

BINARY ADDITION

- a=01001111, b=00011001

Carry			1	1	1	1	1	
	0	1	0	0	1	1	1	1
+	0	0	0	1	1	0	0	1
	0	1	1	0	1	0	0	0

BINARY ADDITION

- a=00101100, b=01000111

Carry				1	1			
	0	0	1	0	1	1	0	0
+	0	1	0	0	0	1	1	1
	0	1	1	1	0	0	1	1

BINARY SUBTRACTION

- 0 - 0 = 0
- 1 - 0 = 1
- 1 - 1 = 0
- 10 – 1 = 1 (In case of 0 – 1, Borrow from the closest bit that's 1).
- A borrowed 1 becomes a 10 at the lower bit.

BINARY SUBTRACTION

- a=01110011, b=00011111

Borrow			0	~~0~~ 10	~~10~~ 1	10		
	0	1	~~1~~	~~1~~	0	0	1	1
-	0	0	0	1	1	1	1	1
	0	1	0	1	0	1	0	0

First borrow

Second borrow

BINARY SUBTRACTION

- a=10111011, b=00111111

Borrow	0	~~10~~ 1	~~0~~ 10	~~0~~ 10	~~0~~ 10	10		
	~~1~~	0	~~1~~	~~1~~	~~1~~	0	1	1
-	0	0	1	1	1	1	1	1
	0	1	1	1	1	1	0	0

BINARY SUBTRACTION

- a=11110110, b=11000011

Borrow						0	~~0~~ 10	10
	1	1	1	1	0	~~1~~	~~1~~	0
-	1	1	0	0	0	0	1	1
	0	0	1	1	0	0	1	1

Binary representation of negative numbers

- Consider the number to be positive. Convert it to binary. Fill out the required number of bits by adding leading 0's if necessary.
- Convert the binary number to it's 2's compliment. (Flip the bits and then add 1 to the result).
- Add the sign bit as the most significant bit. Sign bit is 1 for a negative number.

Binary representation of negative numbers

- -89

Quotient	Remainder
44	1
22	0
11	0
5	1
2	1
1	0
0	1

Sign	1	0	1	1	0	0	1
1	0	1	0	0	1	1	1

Binary representation of negative numbers

- -43

Quotient	Remainder
21	1
10	1
5	0
2	1
1	0
0	1

Sign	0	1	0	1	0	1	1
1	1	0	1	0	1	0	1

Binary representation of negative numbers

- - 127

Quotient	Remainder
63	1
31	1
15	1
7	1
3	1
1	1
0	1

Sign	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1

Binary representation of numbers with fractions

- Convert the integer part into binary.
- For the fractional part, divide the fraction first by 0.5 (2^{-1}), take the quotient as the first bit of the binary fraction.
- Divide the remainder by 0.25 (2^{-2}), and repeat the repeat the process until the remainder's 0.
- Some fractions may not terminate. In this case, find out if the bits form a pattern. If no bit pattern is formed, keep dividing until the required number of bits is filled.

Binary representation of numbers with fractions

- 4.25

  $4_{10}=100_2$

Divide 0.25 by	Quotient, Remainder
2^{-1} (0.5)	0, 0.25
2^{-2} (0.25)	1, 0

$4.25_{10} = 100.01_2$

Binary representation of numbers with fractions

- 12.375

12_{10}=1100_2

Divide 0.375 by	Quotient, Remainder
2^{-1} (0.5)	0, 0.375
2^{-2} (0.25)	1, 0.125
2^{-3} (0.125)	1, 0

$12.375_{10} = 1100.011_2$

Binary representation of numbers with fractions

- 9.875

$9_{10}=1001_2$

Divide 0.875 by	Quotient, Remainder
2^{-1} (0.5)	1, 0.375
2^{-2} (0.25)	1, 0.125
2^{-3} (0.125)	1, 0

$9.875_{10} = 1001.111_2$

Floating point representations in Hex form

- First write the number in binary.
- Convert it to standard form (eg. 10110.001 = 1.0110001 x 2^4). The power of 2's the exponent. The fractional part's the mantissa
- Add the exponent to the bias (127 for 32 bit, 1023 for 64 bit) and convert the sum to binary.
- Write down the number as sign bit, exponent, mantissa. Fill out the remaining bits with 0's.
- Split the number is groups of 4. For each of the 4 bits, write the hexadecimal equivalent.

Floating point representations in Hex form

- 4.25

  $4.25_{10} = 100.01_2 = 1.0001 \times 2^2$

  Exponent - 127 + 2 = $129_{10} = 10000001_2$

  Mantissa – 0001

31				27				23				19				15				11				7				3			
0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x 40880000

Floating point representations in Hex form

- 12.375

  $12.375_{10} = 1100.011_2 = 1.100011 \text{ x } 2^3$

  Exponent - 127 + 3 = $130_{10} = 10000010_2$

  Mantissa – 100011

31				27				23				19				15				11				7				3			
0	1	0	0	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x 41460000

Floating point representations in Hex form

- 9.875

  $9.875_{10} = 1001.111_2 = 1.001111 \times 2^3$

  Exponent - 127 + 3 = $130_{10} = 10000010_2$

  Mantissa – 001111

31				27				23				19				15				11				7				3			
0	1	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x 411e0000

Single Precision Floating Point IEEE- 754

- First write the number in binary.

- Convert it to standard form (eg. 10110.001 = 1.0110001 x 2^4). The power of 2's the exponent. The fractional part's the mantissa.

- Add the exponent to the bias (127 for 32 bit, 1023 for 64 bit) and convert the sum to binary.

- Write down the number as sign bit, exponent, mantissa. Fill out the remaining bits with 0's.

- Split the number is groups of 4. For each of the 4 bits, write the hexadecimal equivalent.

Single Precision Floating Point IEEE- 754

- 1.125

  $1.125_{10} = 1.001_2 = 1.001 \times 2^0$

  Exponent - $127 + 0 = 127_{10} = 01111111_2$

  Mantissa – 001

31				27				23				19				15				11				7				3			
0	0	1	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x 3F900000

Single Precision Floating Point IEEE- 754

- 6.53125

$6.53125_{10} = 110.10001_2 = 1.1010001 \times 2^2$

Exponent - $127 + 2 = 129_{10} = 10000001_2$

Mantissa – 1010001

31				27				23				19				15				11				7				3			
0	1	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x 40D10000

Single Precision Floating Point IEEE- 754

- -5.546875

$5.546875_{10} = 101.100011_2 = 1.01100011 \times 2^2$

Exponent - $127 + 2 = 129_{10} = 10000001_2$

Mantissa – 01100011

31				27				23				19				15				11				7				3			
1	1	0	0	0	0	0	0	1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

= 0x COB1800

Range of Single Precision Float (Positive)

- Largest Number:
  - $(1 + 1 – 2^{-23})$ X $(2^{254-127})$

    $= 2^{128} – 2^{104}$

    =340282346638528859811704183484516925440

  ≈ 3.4028235 X 10^{38}
- Smallest Number:
  - $(1 + 0.0)$ X (2^{1-127})

    $= 2^{-126}$

    ≈ 1.175494351 X 10^{-38}

MACHINE LANGUAGE

- As humans, communicating with a machine is a tedious task. We can't, for example, just say "add this number and that number and store the result here". Computers have no way of even beginning to understand what this means.

MACHINE LANGUAGE

- As we stated before, the alphabet of the machine's language is binary – it simply contains the digits 0 and 1.
- Continuing with this analogy, *instructions* are the words of a machine's language. That is, they are meaningful constructions of the machine alphabet.
- The *instruction set*, then, constitutes the vocabulary of the machine. These are the words understood by the machine itself.

MACHINE LANGUAGE

- To work with the machine, we need a translator.

  Assembly languages serve as an intermediate form between the human-readable programming language and the machine-understandable binary form.
- Generally speaking, compiling a program into an executable format involves the following stages:

High-level Language → Assembly Language → Machine Language

EXAMPLE OF TRANSLATING A C PROGRAM

High-Level Language Program

```
swap(int v[], int k){
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly Language Program

```
swap:
        multi  $2, $5,  4
        add    $2, $4, $2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```

Assembler

Binary Machine Language Program

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

MACHINE LANGUAGE

- A single human-readable high-level language instruction is generally translated into multiple assembly instructions.
- A single assembly instruction is a symbolic representation of a single machine language instruction.
- A single machine language instruction is a set of bits representing a basic operation that can be performed by the machine.
- The instruction set is the set of possible instructions for a given machine.

ADVANTAGES OF HIGH-LEVEL LANGUAGES

- Requiring these translation steps may seem cumbersome but there are a couple of high-level language advantages that make this scheme worthwhile.
- High-level languages allow the programmer to think in more natural, less tedious terms – specifically in the case of application-specific languages.
- Improve programmer productivity.
- Improve program maintainability.
- Applications can be independent of the computer on which they were developed.
- Highly-optimizing compilers can produce very efficient machine code optimized for a target machine.

WHY LEARN ASSEMBLY LANGUAGE?

- So, if high-level languages are so great…why bother learning assembly?
- Knowing assembly language illuminates concepts not only in computer organization, but operating systems, compilers, parallel systems, etc.
- Understanding how high-level constructs are implemented leads to more effective use of those structures.
  - Control constructs (if, do-while, etc.)
  - Pointers
  - Parameter passing (pass-by-value, pass-by-reference, etc.)
- Helps to understand performance implications of programming language features.

MIPS

- We will start with a lightning review of MIPS.
- MIPS is a RISC (Reduced Instruction Set Computer) instruction set, meaning that it has simple and few instructions.
- Originally introduced in the early 1980's.
- An acronym for Microprocessor without Interlocked Pipeline Stages.
- MIPS architecture has been used in many computer products, especially in the late 80's and early 90's. N64, Playstation, and Playstation 2 all used MIPS implementations.
- Many ISAs that have since been designed are very similar to MIPS.
- In the mid to late 90's, approximately 1/3 of all RISC microprocessors were MIPS implementations.

RISC ARCHITECTURE

- CISC (Complex Instruction Set Computer)
  - Intel x86
- RISC (Reduced Instruction Set Computer)
  - MIPS, Sun SPARC, IBM, PowerPC, ARM
- RISC Philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited number of addressing modes
  - limited number of operations

THE FOUR ISA DESIGN PRINCIPLES

1. Simplicity favors regularity
    - Consistent instruction size, instruction formats, data formats
    - Eases implementation by simplifying hardware
2. Smaller is faster
    - Fewer bits to access and modify
    - Use the register file instead of slower memory
3. Make the common case fast
    - e.g. Small constants are common, thus small immediate fields should be used.
4. Good design demands good compromises
    - Compromise with special formats for important exceptions
    - e.g. A long jump (beyond a small constant)

MIPS REVIEW

- Now we'll jump right into our lightning review of MIPS. The general classes of MIPS instructions are
- Arithmetic
    - add, subtract, multiply, divide
- Logical
    - and, or, nor, not, shift
- Data transfer
    - load from or store to memory
- Transfers of control
    - jumps, branches, calls, returns

QUICK EXAMPLE

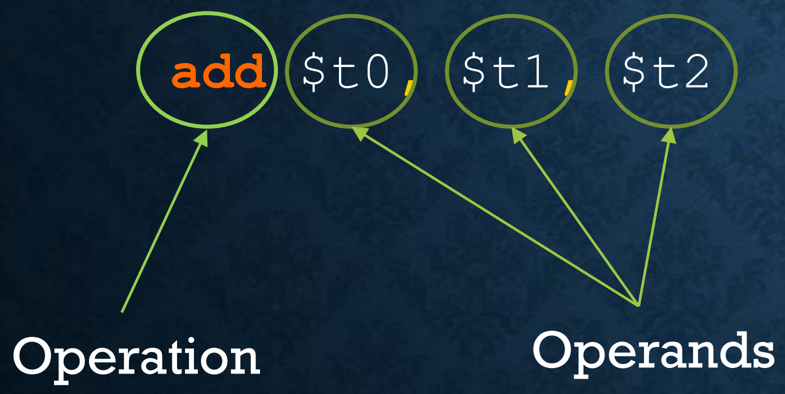
- Here is an example of one of the simplest and most common MIPS instructions.

```
add $t0, $t1, $t2
```

- This MIPS instruction symbolizes the machine instruction for adding the contents of register t1 to the contents of register t2 and storing the result in t0.

QUICK EXAMPLE

- Here is an example of one of the simplest and most common MIPS instructions.



QUICK EXAMPLE

- Here is an example of one of the simplest and most common MIPS instructions.

```
add $t0, $t1, $t2
```

- The corresponding binary machine instruction is

000000 01001 01010 01000 00000 100000

This portion tells the machine exactly which operation we're performing. In this case, 100000 refers to an addition operation

QUICK EXAMPLE

- Here is an example of one of the simplest and most common MIPS instructions.

```
add $t0, $t1, $t2
```

- The corresponding binary machine instruction is

000000 01001 01010 01000 00000 100000

This portion is used for shift instructions, and is therefore not used by the machine in this case.

QUICK EXAMPLE

- Here is an example of one of the simplest and most common MIPS instructions.

```
add $t0, $t1, $t2
```

- The corresponding binary machine instruction is

000000 01001 01010 01000 00000 100000

This portion indicates the destination register – this is where the result will be stored. Because $t0 is the 8th register, we use 01000 to represent it.

QUICK EXAMPLE

- Here is an example of one of the simplest and most common MIPS instructions.

```
add $t0, $t1, $t2
```

- The corresponding binary machine instruction is

000000 01001 01010 01000 00000 100000

This portion indicates the second source register. Because $t2 is the 10th register, we use 01010 to represent it.

QUICK EXAMPLE

- Here is an example of one of the simplest and most common MIPS instructions.

```
add $t0, $t1, $t2
```

- The corresponding binary machine instruction is

000000 01001 01010 01000 00000 100000

This portion indicates the first source register. Because $t1 is the 9th register, we use 01001 to represent it.

QUICK EXAMPLE

- Here is an example of one of the simplest and most common MIPS instructions.

```
add $t0, $t1, $t2
```

- The corresponding binary machine instruction is

000000 01001 01010 01000 00000 100000

This last portion holds the operation code relevant for other types of instructions. The add operation, and others like it, always have a value of 0 here.

MIPS INSTRUCTION OPERANDS

- So now that we've seen an example MIPS instruction and how it directly corresponds to its binary representation, we can talk about the components of an instruction. MIPS instructions consist of operations on one or more operands. Operands in MIPS fit into one of three categories.
- Integer constants
- Registers
- Memory

INTEGER CONSTANT OPERANDS

Integer constant operands are used frequently. For example, while looping over an array, we might continually increment an index to access the next array element.

To avoid saving the constant elsewhere and having to retrieve it during every use, MIPS allows for *immediate* instructions which can include a constant directly in the instruction.

A simple example is add immediate:

```
addi $s3, $s3, 4  # adds 4 to the value in $s3 and stores in $s3
```

INTEGER CONSTANTS

- Generally represented with 16 bits, but they are extended to 32 bits before being used in an operation.
- Most operations use signed constants, although a few support unsigned.
- Integer constants can be represented in MIPS assembly instructions using decimal, hexadecimal, or octal values.
- A reflection of design principle 3, make the common case fast.
  - Because constants are used frequently, it is faster and more energy efficient to support instructions with built-in constants rather than fetching them from memory all the time.

REGISTERS

- We've already seen some simple register usage in our two example MIPS instructions.

```
add $t0, $t1, $t2

addi $s3, $s3, 4
```

- In these instructions, $t0, $t1, $t2, and $s3 are all registers. Registers are special locations built directly into the hardware of the machine. The size of a MIPS register is 32 bits. This size is also commonly known as a *word* in MIPS architecture.

REGISTERS

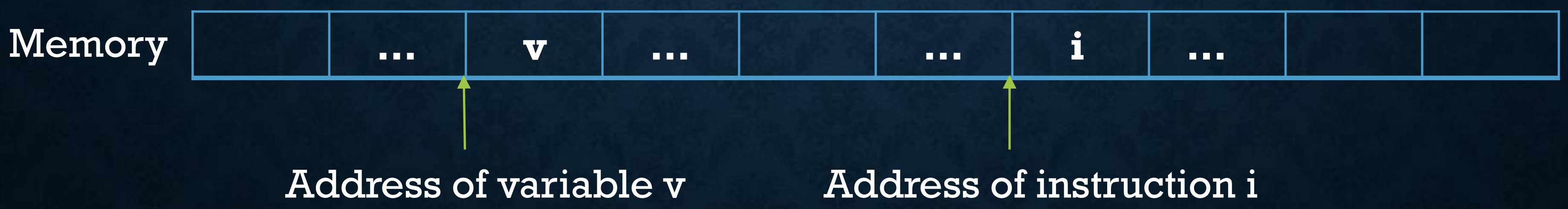
- There are only 32 (programmer visible) 32-bit registers residing in a MIPS processor.
- Reflects design principle 2, smaller is faster.
  - Having a small number of registers ensures that accessing a desired register is fast since they can be kept closer.
  - Also means that fewer bits can be used to identify registers → decreases instruction size.
- Registers also use much less power than memory accesses.
- MIPS convention is to use two-character names following a dollar sign.
  - Register 0: $zero – stores the constant value 0.
  - Registers 16-23: $s0-$s7 – saved temporaries (variables in C code).
  - Registers 8-15: $t0-$t7 – temporaries.

REGISTERS

Name	Number	Use
$zero	0	Constant value 0.
$at	1	Assembler temporary. For resolving pseudoinstructions.
$v0-$v1	2-3	Function results and expression evaluation.
$a0-$a3	4-7	Arguments.
$t0-$t9	8-15, 24-25	Temporaries.
$s0-$s7	16-23	Saved temporaries.
$k0-$k1	26-27	Reserved for OS kernel.
$gp	28	Global pointer.
$sp	29	Stack pointer.
$fp	30	Frame pointer.
$ra	31	Return address.

MEMORY OPERANDS

- Before we talk about memory operands, we should talk generally about how data is stored in memory.
- As we said before, memory contains both data and instructions.
- Memory can be viewed as a large array of bytes.
- The beginning of a variable or instruction is associated with a specific element of this array.
- The address of a variable or instruction is its offset from the beginning of memory.



MEMORY OPERANDS

- For a large, complex data structure, there are likely many more data elements than there are registers available. However, arithmetic operations occur only on registers in MIPS.
- To facilitate large structures, MIPS includes *data transfer instructions* for moving data between memory and registers.
- As an example, assume we have the following C code, where A is an array of 100 words.

```
g = h + A[8]
```

MEMORY OPERANDS

- Let's say g and h are associated with the registers $s1 and $s2 respectively. Let's also say that the base address of A is associated with register $s3.

```
g = h + A[8]
```

- To compile this statement into MIPS, we'll need to use the *load word* instruction to transfer A[8] into a register.

```
lw  $t0,32($s3) # load the element at a 32 byte offset from $s3
add $s1,$s2,$t0
```

- There is an equivalent *store word* instruction for storing data to memory as well.

MIPS ASSEMBLY FILE

- Now, let's turn our attention to the structure of a MIPS assembly file.
- MIPS assembly files contain a set of lines.
- Each line can be either a *directive* or an *instruction*.
- Each directive or instruction may start with a *label*, which provides a symbolic name for a data or instruction location.
- Each line may also include a comment, which starts with # and continues until the end of the line.

GENERAL FORMAT

```
    .data
    # allocation of memory
    .text
    .global main
main:
    # instructions here
    jr $ra # instruction indicating a return
```

MIPS DIRECTIVES

Directive	**Meaning**
.align *n*	Align next datum on 2^n boundary.
.asciiz *str*	Place the null-terminated string str in memory.
.byte *b1, …, bn*	Place the n byte values in memory.
.data	**Switch to the data segment.**
.double *d1, …, dn*	Place the n double-precision values in memory.
.float *f1, …, fn*	Place the n single-precision values in memory.
.global *sym*	**The label sym can be referenced in other files.**
.half *h1, …, hn*	Place the n half-word values in memory.
.space *n*	**Allocates n bytes of space.**
.text	**Switch to the text segment.**
.word *w1, …, wn*	**Place the n word values in memory.**

MIPS INSTRUCTION REVIEW: ADD

- add d, s1, s2
- Example: add $t0, $t1, $t2

  Destination is $t0, Sources are $t1 and $t2
- Logic:
  - Bitwise addition with carries
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10. Sum is 0, carry 1

MIPS INSTRUCTION REVIEW: ADDI

- addi d, s, immediate
- Example: addi $t0, $t1, 10

  Destination is $t0, Sources are $t1 and an immediate signed short number (-32768 - +32767)

- Logic: same as ADD

MIPS INSTRUCTION REVIEW: SUB

- sub d, s1, s2
- Example: sub $t0, $t1, $t2

  Destination is $t0, Sources are $t1 and $t2
- Logic: Bitwise subtraction with borrows
  - 0 – 0 = 0
  - 1 – 1 = 0
  - 1 – 0 = 1
  - 0 - 1 = 1 and remove 1 from the next digit

  – Actually, its two's Complement is added.

MIPS Instruction Review: AND, ANDi

- and d, s1, s2
- Example: and $t0, $t1, $t2

  Destination is $t0, Sources are $t1 and $t2
- Logic:
  - Bitwise
  - 0 & 0 = 0
  - 0 & 1 = 0
  - 1 & 0 = 0
  - 1 & 1 = 1
- andi performs AND operation with an immediate signed short operand. Syntax of ADDi, operation of AND

MIPS Instruction Review: OR, ORi

- or d, s1, s2
- Example: or $t0, $t1, $t2

  Destination is $t0, Sources are $t1 and $t2
- Logic:
  - Bitwise
  - 0 | 0 = 0
  - 0 | 1 = 1
  - 1 | 0 = 1
  - 1 | 1 = 1
- ori performs OR operation with an immediate operand. Syntax of ADDi, operation of OR

MIPS Instruction Review: XOR, XORi

- xor d, s1, s2
- Example: xor $t0, $t1, $t2

  Destination is $t0, Sources are $t1 and $t2
- Logic:
  - Bitwise
  - $0 \oplus 0 = 0$
  - $0 \oplus 1 = 1$
  - $1 \oplus 0 = 1$
  - $1 \oplus 1 = 0$
- xori performs XOR operation with an immediate operand. Syntax of ADDi, operation of XOR

MIPS Instruction Review: NOR

- nor d, s1, s2
- Example: nor $t0, $t1, $t2

  Destination is $t0, Sources are $t1 and $t2
- Logic:
  - Bitwise
  - $0 \downarrow 0 = 1$
  - $0 \downarrow 1 = 0$
  - $1 \downarrow 0 = 0$
  - $1 \downarrow 1 = 0$

MIPS INSTRUCTION REVIEW: LW

- lw d, immediate(pointer)
- Example: lw $t0, 12($t1)

Destination is $t0, Source Address is $t1 + immediate signed short offset (a multiple of 4)

- Logic:
  - –Fetches value at an address in memory and loads it into a register.

MIPS INSTRUCTION REVIEW: SW

- sw d, immediate(pointer)
- Example: sw $t0, 12($t1)

  Source is $t0, Destination Address is $t1 + immediate signed short offset (a multiple of 4)
- Logic:
  - Fetches value within a register and stores it in a memory address.

MIPS INSTRUCTION REVIEW: SLL

- sll d, s, immediate
- • sll $t0, $t1, 2
- Destination is $t0, Source is $t1, immediate is number of bits to shift (0 to 32)
- •Logic:
  - Shifts the bits of number in source register left by the number of bits specified. 0's are shifted in. Result is stored in destination register.

MIPS INSTRUCTION REVIEW: SRL

- srl d, s, immediate
- • srl $t0, $t1, 2
- Destination is $t0, Source is $t1, immediate is number of bits to shift (0 to 32)
- •Logic:
  - Shifts the bits of number in source register right by the number of bits specified. 0's are shifted in. Result is stored in destination register.

CONVERT C CODE TO MIPS

- $t0 = A[$t2];
- A[$t2] = $t0 & $t1;
- $t0 = (A[$t1] + $t2) / 2;

- The starting address of array A is in $s0, and if $t2 = 4, A[$t2] represents the 4th element in A. Array elements are numbered from 0.

- Don't modify the contents of registers unless the C code specifically states to.

$T0 = A[$T2];

- sll $t4, $t2, 2
- add $t4, $t4, $s0
- lw $t0, 0($t4)

- MIPS is word addressed. Memory is byte addressed. So, we need to multiply MIPS addresses by 4 to get to the memory address. Hence sll.

A[$T2] = $T0 & $T1;

- and $t5, $t0, $t1
- sw $t5, 0($t4)

- We already have A[$t2] in $t4. So, we need not recalculate the address.

$T0 = (A[$T1] + $T2) / 2;

- sll $t6, $t1, 2
- add $t6, $t6, $s0
- lw $t7, 0($t6)
- add $t7, $t7, $t2
- srl $t0, $t7, 1

- Right shifting by 1 bit is the same as dividing by 2. Left shifting by 1 bit is the same as multiplying by 2.

MIPS INSTRUCTIONS

General format:

```
<optional label> <operation> <operands>
```

Example:

```
loop:     addu $t2,$t3,$t4 # instruction with a label
          subu $t2,$t3,$t4 # instruction without a label
L2:       # a label can appear on a line by itself
# a comment can appear on a line by itself
```

MIPS INSTRUCTIONS

- What does this look like in memory?

```
    .data
nums:
    .word  10, 20, 30
    .text
    .globl main
main:
    la $t0, nums
    lw $t1, 4($t0)
```

MIPS INSTRUCTION FORMATS

- There are three different formats for MIPS instructions.
- R format
  - Used for shifts and instructions that reference only registers.
- I format
  - Used for loads, stores, branches, and immediate instructions.
- J format
  - Used for jump and call instructions.

MIPS INSTRUCTION FORMATS

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct
I format	op	rs	rt	immed		
J format	op	targaddr				

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.

shamt – shift amount.
funct – additional opcodes.
immed – offsets/constants.
targaddr – jump/call target.

MIPS INSTRUCTION FORMATS

All MIPS instructions are 32 bits – Design principle 1: simplicity favors regularity!

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct
I format	op	rs	rt	immed		
J format	op	targaddr				

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.

shamt – shift amount.
funct – additional opcodes.
immed – offsets/constants.
targaddr – jump/call target.

MIPS INSTRUCTION FORMATS

Make simple instructions fast and accomplish other operations as a series of simple instructions – Design principle 3: make the common case fast!

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct
I format	op	rs	rt	immed		
J format	op	targaddr				

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.

shamt – shift amount.
funct – additional opcodes.
immed – offsets/constants.
targaddr – jump/call target.

MIPS R FORMAT

- Used for shift operations and instructions that only reference registers.
- The *op* field has a value of 0 for all R format instructions.
- The *funct* field indicates the type of R format instruction to be performed.
- The *shamt* field is used only for the shift instructions (sll and srl, sra)

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.
shamt – shift amount.
funct – additional opcodes.

R Format Instruction Encoding Example

- Consider the following R format instruction:

```
addu $t2, $t3, $t4
```

Fields	op	rs	rt	rd	shamt	funct
Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Decimal	0	11	12	10	0	33
Binary	000000	01011	01100	01010	00000	100001
Hexadecimal	0x016c5021					

MIPS I FORMAT

- Used for arithmetic/logical immediate instructions, loads, stores, and conditional branches.
- The *op* field is used to identify the type of instruction.
- The *rs* field is the source register.
- The *rt* field is either the source or destination register, depending on the instruction.
- The *immed* field is zero-extended if it is a logical operation. Otherwise, it is sign-extended.

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	rt	immed		

I FORMAT INSTRUCTION ENCODING EXAMPLES

Arithmetic example:

```
addiu $t0,$t0,1
```

Fields	op	rs	rt	immed
Size	6 bits	5 bits	5 bits	16 bits
Decimal	9	8	8	1
Binary	001001	01000	01000	0000000000000001
Hexadecimal			0x25080001	

I FORMAT INSTRUCTION ENCODING EXAMPLES

Memory access example:

```
lw $s1,100($s2)
```

Fields	op	rs	rt	immed
Size	6 bits	5 bits	5 bits	16 bits
Decimal	35	18	17	100
Binary	100011	10010	10001	0000000001100100
Hexadecimal	0x8e510064			

I FORMAT INSTRUCTION ENCODING EXAMPLES

Conditional branch example:

```
L2:instruction
   instruction
   instruction
   beq $t6,$t7,L2
```

Note: Branch displacement is a signed value in instructions, not bytes, from the current instruction. Branches use **PC-relative** addressing.

Fields	**op**	**rs**	**rt**	**immed**
Size	6 bits	5 bits	5 bits	16 bits
Decimal	4	14	15	-3
Binary	000100	01110	01111	1111111111111101
Hexadecimal			0x11cffffd	

ADDRESSING MODES

- Addressing mode – a method for evaluating an operand.
- MIPS Addressing Modes
  - **Immediate** – operand contains signed or unsigned integer constant.
  - **Register** – operand contains a register number that is used to access the register file.
  - **Base Displacement** – operand represents a data memory value whose address is the sum of some signed constant (in bytes) and the register value referenced by the register number.
  - **PC relative** – operand represents an instruction address that is the sum of the PC and some signed integer constant (in words).
  - **Pseudodirect** – operand represents an instruction address (in words) that is the field concatenated with the upper bits of the PC.

PC Relative and Pseudodirect addressing are actually relative to PC + 4, **not** PC. The reason for this will become clearer when we look at the design for the processor, so we'll ignore it for now.

MEMORY ALIGNMENT REQUIREMENTS

- MIPS requires alignment of memory references to be an integer multiple of the size of the data being accessed.
- These alignments are enforced by the compiler.
- The processor checks this alignment requirement by inspecting the least significant bits of the address.

Byte:	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Half:	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX0
Word:	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX00
Double:	XXXXXXXXXXXXXXXXXXXXXXXXXXXXX000

MIPS J FORMAT

- Used for unconditional jumps and function calls.
- The *op* field is used to identify the type of instruction.
- The *targaddr* field is used to indicate an absolute target address.

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
J format	op	targaddr				

J FORMAT INSTRUCTION ENCODING EXAMPLE

- Jump example:

```
j L1
```

- Assume L1 is at the address 4194340 in decimal, which is 400024 in hexadecimal. We fill the target field as an address in instructions (0x100009) rather than bytes (0x400024). Jump uses **pseudo-direct** addressing to create a 32-bit address.

Fields	**op**	**target address**
Size	6 bits	26 bits
Decimal	2	1048585
Binary	000010	00000100000000000000001001
Hexadecimal		0x08100009

ARITHMETIC/LOGICAL GENERAL FORM

- Most MIPS arithmetic/logical instructions require 3 operands.
- Design principle 1: Simplicity favors regularity.
- Form 1: `<operation> <dstreg>, <src1reg>, <src2reg>`

Example	Meaning	Comment
addu $t0, $t1, $t2	$t0 = $t1 + $t2	Addition (without overflow)
subu $t1, $t2, $t3	$t1 = $t2 - $t3	Subtraction (without overflow)

- Form 2: `<operation> <dstreg>, <srcreg>, <constant>`

Example	Meaning	Comment
addiu $t1,$t2,1	$t1 = $t2 + 1	Addition immediate (without overflow)

USING MIPS ARITHMETIC INSTRUCTIONS

- Consider the following C++ source code fragment.

```
unsigned int f,g,h,i,j;
...
f = (g+h)-(i+j);
```

- Assume the values of f, g, h, i, and j are associated with registers \$t2, \$t3, \$t4, \$t5, and \$t6 respectively. Write MIPS assembly code to perform this assignment assuming \$t7 is available.

USING MIPS ARITHMETIC INSTRUCTIONS

- Solution (among others):
-

```
addu $t2,$t3,$t4 # $t2 = g + h
addu $t7,$t5,$t6 # $t7 = i + j
subu $t2,$t2,$t7 # $t2 = $t2 - $t7
```

MULTIPLY, DIVIDE, AND MODULUS INSTRUCTIONS

- Integer multiplication, division, and modulus operations can also be performed.
- MIPS provides two extra registers, **hi** and **lo**, to support division and modulus operations.

Example	**Meaning**	**Comment**
mult $t1,$t2	$lo = $t1 * $t2	Multiplication
divu $t2,$t3	$lo = $t2/$t3 $hi = $t2%$t3	Division and Modulus
mflo $t1	$t1 = $lo	Move from $lo
mfhi $t1	$t1 = $hi	Move from $hi

CALCULATING QUOTIENT AND REMAINDER

- Given the values \$t1 and \$t2, the following sequence of MIPS instructions assigns the quotient (\$t1/\$t2) to \$s0 and the remainder (\$t1%\$t2) to \$s1 .

```
divu $t1,$t2 # perform both division and modulus operations
mflo $s0     # move quotient into $s0
mfhi $s1     # move remainder into $s1
```

LOGICAL OPERATIONS

- Consist of bitwise Boolean operations and shifting operations.
- Shifting operations can be used to extract or insert fields of bits within a word.

X	Y	Not X	X and Y	X or Y	X nand Y	X nor Y	X xor Y
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

GENERAL FORM OF MIPS BITWISE INSTRUCTIONS

- Bitwise instructions apply Boolean operations on each of the corresponding pairs of bits of two values.

Example	Meaning	Comment
and $t2,$t3,$t4	$t2 = $t3 & $t4	Bitwise and
or $t3,$t4,$t5	$t3 = $t4 | $t5	Bitwise or
nor $t4,$t3,$t6	$t4 = ~($t3 | $t6)	Bitwise nor
xor $t7,$t2,$t4	$t7 = $t2 ^ $t4	Bitwise xor
andi $t2,$t3,7	$t2 = $t3 & 7	Bitwise and with immediate
ori $t3,$t4,5	$t3 = $t4 | 5	Bitwise or with immediate
xori $t7,$t2,6	$t7 = $t2 ^ 6	Bitwise xor with immediate

GENERAL FORM OF MIPS SHIFT INSTRUCTIONS

- Shift instructions move the bits in a word to the left or right by a specified amount.
- Shifting left (right) by i is the same as multiplying (dividing) by 2^i.
- An arithmetic right shift replicates the most significant bit to fill in the vacant bits.
- A logical right shift fills in the vacant bits with zero.

Example	**Meaning**	**Comment**
sll $t2,$t3,2	$t2 = $t3 << 2	Shift left logical
sllv $t3,$t4,$t5	$t3 = $t4 << $t5	Shift left logical variable
sra $t4,$t3,1	$t4 = $t3 >> 1	Shift right arithmetic (signed)
srav $t7,$t2,$t4	$t7 = $t2 >> $t4	Shift right arithmetic variable (signed)
srl $t2,$t3,7	$t2 = $t3 >> 7	Shift right logical (unsigned)
srlv $t3,$t4,$t6	$t3 = $t4 >> $t6	Shift right logical variable

GLOBAL ADDRESSES AND LARGE CONSTANTS

- The lui instruction can be used to construct large constants or addresses. It loads a 16-bit value in the 16 most significant bits of a word and clears the 16 least significant bits.

Form	Example	Meaning	Comment
lui <dreg>,<const>	lui $t1,12	$t1 = 12 << 16	Load upper immediate

- Example: load 131,071 (or 0x1ffff) into $t2.

```
lui $t2,1               # put 1 in the upper half of $t2
ori $t2,$t2,0xffff  # set all bits in the lower half
```

- Having all instructions the same size and a reasonable length means having to construct global addresses and some constants using two instructions.
- Design principle 4: Good design demands good compromise!

DATA TRANSFER GENERAL FORM

- MIPS can only access memory with load and store instructions.
- Form: `<operation>  <reg1>, <constant>(<reg2>)`

Example	Meaning	Comment
lw $t2,8($t3)	$t2 = Mem[$t3 + 8]	32-bit load
lh $t3,0($t4)	$t3 = Mem[$t4]	Signed 16-bit load
lhu $t8,2($t3)	$t8 = Mem[$t3 + 2]	Unsigned 16-bit load
lb $t4,0($t5)	$t4 = Mem[$t5]	Signed 8-bit load
lbu $t6,1($t9)	$t6 = Mem[$t9 + 1]	Unsigned 8-bit load
sw $t5,-4($t2)	Mem[$t2-4] = $t5	32-bit store
sh $t6,12($t3)	Mem[$t3 + 12] = $t6	16-bit store
sb $t7,1($t3)	Mem[$t3 + 1] = $t7	8-bit store

USING DATA TRANSFER INSTRUCTIONS

- Consider the following source code fragment.

```
int a, b, c, d;
...
a = b + c - d;
```

- Assume the *addresses* of a, b, c, and d are in the registers $t2, $t3, $t4, and $t5, respectively. The following MIPS assembly code performs this assignment assuming $t6 and $t7 are available.

```
lw $t6,0($t3)      # load b into $t6
lw $t7,0($t4)      # load c into $t7
add $t6,$t6,$t7   # $t6 = $t6 + $t7
lw $t7,0($t5)      # load d into $t7
sub $t6,$t6,$t7   # $t6 = $t6 - $t7
sw $t6,0($t2)      # store $t6 into a
```

INDEXING ARRAY ELEMENTS

- Assembly code can be written to access array elements using a variable index. Consider the following source code fragment.

```
int a[100], i;
...
a[i] = a[i] + 1;
```

- Assume the *value* of i is in $t0. The following MIPS code performs this assignment.

```
    .data
_a: .space 400             # declare space
    ...
    la $t1, _a             # load address of _a
    sll $t2,$t0,2          # determine offset
    add $t2,$t2,$t1        # add offset and _a
    lw $t3,0($t2)          # load the value
    addi $t3,$t3,1         # add 1 to the value
    sw $t3,0($t2)          # store the value
```

TRANSFER OF CONTROL INSTRUCTIONS

- Transfer of control instructions can cause the next instruction to be executed to be other than the next sequential instruction.
- Transfers of control are used to implement control statements in high-level languages.
  - Unconditional (goto, break, continue, call, return)
  - Conditional (if-then, if-then-else, switch)
  - Iterative (while, do, for)

GENERAL FORM OF JUMP AND BRANCH

- MIPS provides direct jumps to support unconditional transfers of control to a specified location.
- MIPS provides indirect jumps to support returns and switch statements.
- MIPS provides conditional branch instructions to support decision making. MIPS conditional branches test if the values of two registers are equal or not equal.

General Form	**Example**	**Meaning**	**Comment**
`j <label>`	`j L1`	goto L1;	Direct jump (J)
`jr <sreg>`	`jr $ra`	goto $ra;	Indirect jump (R)
`beq <s1reg>,<s2reg>,<label>`	`beq $t2, $t3, L1`	if($t2 == $t3) goto L1;	Branch equal (I)
`bne <s1reg>,<s2reg>,<label>`	`bne $t2, $t3, L1`	if($t2 != $t3) goto L1;	Branch not equal (I)

IF STATEMENT EXAMPLE

- Consider the following source code:

```
if(i == j)
    k = k + i;
```

- Translate into MIPS instructions assuming the *values* of i, j, and k are associated with the registers $t2, $t3, and $t4, respectively.

```
        bne  $t2,$t3,L1  # if ($t2 != $t3) goto L1
        addu $t4,$t4,$t2 # k = k + i
L1:
```

GENERAL FORM OF COMPARISON INSTRUCTIONS

- MIPS provides set less than instructions that set a register to 1 if the first source register is less than the value of the second operand. Otherwise, it is set to 0.
- There are versions for performing unsigned comparisons as well.

General Form	Example	Meaning	Comment
slt <dreg>,<s1reg>,<s2reg>	slt $t2,$t3,$t4	if($t3<$t4) $t2 = 1; else $t2 = 0;	Compare less than (R)
sltu <dreg>,<s1reg>,<s2reg>	sltu $t2,$t3,$t4	if($t3<$t4) $t2 = 1; else $t2 = 0;	Compare less than unsigned (R)
slti <dreg>,<s1reg>,<const>	slti $t2,$t3,100	if($t3<100) $t2 = 1; else $t2 = 0;	Compare less than constant (I)
sltiu <dreg>,<s1reg>,<const>	sltiu $t2,$t3,100	if($t3<100) $t2 = 1; else $t2 = 0;	Compare less than constant unsigned (I)

TRANSLATING AN IF STATEMENT

- Consider the following source code:

```
if(a > b)
    c = a;
```

- Translate into MIPS instructions assuming the *values* of a, b, and c are associated with the registers $t2, $t3, and $t4 respectively. Assume $t5 is available.

```
    slt   $t5,$t3,$t2     # b < a
    beq   $t5,$zero,L1    # if($t5==0)goto L1
    or    $t4,$t2,$zero   # c = a
L1:
```

TRANSLATING AN IF-THEN-ELSE STATEMENT

- Consider the following source code:

```
if(a < b)
    c = a;
else
    c = b;
```

```
     slt   $t5,$t2,$t3     # a < b
     beq   $t5,$zero,L1    # if($t5==0)goto L1
     move  $t4,$t2         # c = a
     j     L2              # goto L2
L1:  move  $t4, $t3        # c = b
L2:
```

- Translate into MIPS instructions assuming the *values* of a, b, and c are associated with the registers $t2, $t3, and $t4 respectively. Assume $t5 is available.

HIGH-LEVEL CONTROL STATEMENTS

- How do we translate other high-level control statements (while, do, for)?
- We can first express the C statement using C if and goto statements.
- After that, we can translate using MIPS unconditional jumps, comparisons, and conditional branches.

TRANSLATING A FOR STATEMENT

- Consider the following source code:

```
sum = 0;
for(i=0; i<100; i++)
    sum += a[i];
```

- First, we replace the for statement using an if and goto statements.

```
      sum = 0;
      i = 0;
      goto test;
loop: sum += a[i];
      i++;
test: if (i < 100) goto loop;
```

TRANSLATING A FOR STATEMENT

- Now for the MIPS instructions. Assume sum, i and the starting address of a are associated with $t2, $t3, and $t4 respectively and that $t5 is available.

```
        li     $t2, 0             # sum = 0
        move   $t3, $zero         # i = 0
        j      test
loop:   sll    $t5,$t3,2          # temp = i * 4
        addu   $t5,$t5,$t4        # temp = temp + &a
        lw     $t5,0($t5)         # load a[i] into temp
        addu   $t2,$t2,$t5        # sum += temp
        addiu  $t3,$t3,1          # i++
test:   slti   $t5,$t3,100        # test i < 100
        bne    $t5,$zero,loop     # if true, goto loop
```