CS 2200
LAB 2

Announcements

- **Homework 1** released after lab hopefully, due **Sep 5th @ 6:30 PM**
- **Project 1** released, is due on **Sep 17th @ 11:59 PM**
- **Lab attendance** will be taken starting today

Endianness & Padding

Coming Up!

We need a test C program that tells you whether your platform is big or little endian.

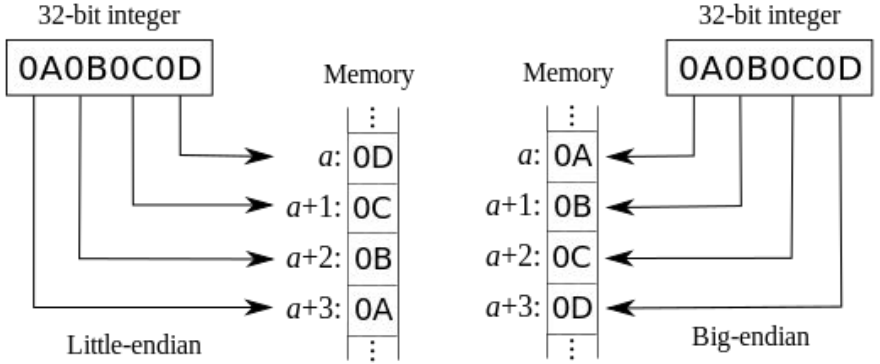
What does that mean?

How should we go about it?

Endianness

Endianness defines how the system will organize the bytes that make up numbers. All data is comprised into chunks of bytes, with the largest byte in the number called the Most Significant Byte (MSB), and the smallest byte called the Least Significant Byte (LSB).

- Big Endian = The most significant byte is at the lowest memory address
- Little Endian = The least significant byte is at the lowest memory address



Endianness Example

```
int i = 0x12345678/* 4 bytes */
```

Little endian

	+0	+1	+2	+3
0x00				

Big endian

	+0	+1	+2	+3
0x00				

Endianness Example

```
int i = 0x12345678/* 4 bytes */
```

Little endian

	+0	+1	+2	+3
0x00	**0x78**	**0x56**	**0x34**	**0x12**

Big endian

	+0	+1	+2	+3
0x00	**0x12**	**0x34**	**0x56**	**0x78**

A little DIY!

In groups, try to write and test a C program that tells you whether your platform is big or little endian. Take 10 mins and feel free to collaborate!

HINT: Try using a union with int and char members

A little DIY!

In groups, try to write and test a C program that tells you whether your platform is big or little endian. Take 10 mins and feel free to collaborate!

HINT: Try using a union with int and char members

The solution C file is in the attendance quiz endianness question

Endianness In C

If the computer that is running the program is little endian, what would the expected output be?

Similarly what would the output be for a big endian system?

```
void show_mem_rep(char *start, int n) {
    for (int i = 0; i < n; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}

int main()
{
   int i = 0x01234567;
   show_mem_rep((char *)&i, sizeof(i));
   return 0;
}
```

Endianness In C

If the computer that is running the program is little endian, what would the expected output be?

```
67 45 23 01
```

Similarly what would the output be for a big endian system?

```
01 23 45 67
```

```c
void show_mem_rep(char *start, int n) {
    for (int i = 0; i < n; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}

int main()
{
   int i = 0x01234567;
   show_mem_rep((char *)&i, sizeof(i));
   return 0;
}
```

Padding in C

How do we predict what the sizes of these structures will be?

Presume we're using an ARM system that requires natural alignment.

Natural alignment means that a data type will be stored at a memory address that is a **multiple** of its size!

And structures will be padded so their size will be a **multiple** of the **largest** data type they contain. (Why?)

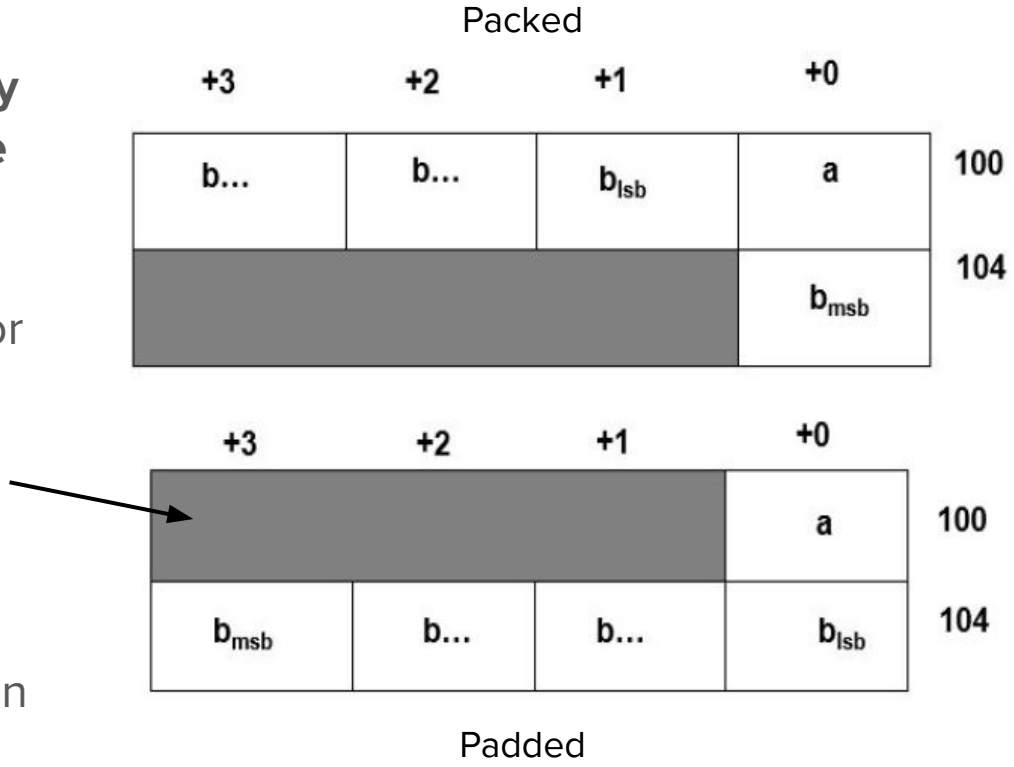
```
struct A {
     char *ptr;// 8 bytes
     char c;   // 1 byte
     long num; // 8 bytes
};

struct B {
     char c;   // 1 byte
     short s[2];// 2 x 2 bytes
     int i;    // 4 bytes
};

struct C {
     int i;    // 4 bytes
     short s[2];// 2 x 2 bytes
     char c;   // 1 byte
};
```

Padding Structs

- **Unaligned access to memory is often expensive and some architectures don't allow it.**
- For example, ARM CPUs require "natural alignment" for their data types. MIPS and Intel CPUs implement unaligned access, but do it significantly more slowly.
- Compilers will apply **padding** to small data types to maintain alignment in memory.



Padding Example

```
struct Q {
    char c;       /* 1 bytes */
    short s;      /* 2 bytes */
    int i;        /* 4 bytes */
};
```

	+0	+1	+2	+3
0x00				
0x04				

Padding Example

```
struct Q {
    char c;       /* 1 bytes */
    short s;      /* 2 bytes */
    int i;        /* 4 bytes */
};
```

	+0	+1	+2	+3
0x00	c		s0	s1
0x04	i0	i1	i2	i3

Padding in C

What do you think the following output will be?

Word is 8 Bytes

What are the sizes of each struct?

- A
- B
- C

```
struct A {
      char *ptr; // 8 bytes
      char c;    // 1 byte
      long num;  // 8 bytes
};

struct B {
      short s;   // 2 bytes
      int i;     // 4 bytes
      short s2;  // 2 bytes
};

struct C {
      int i;     // 4 bytes
      short s;   // 2 bytes
      short s2;  // 2 bytes
};

int main() {
      printf("A size: %lu\n", sizeof(struct A));
      printf("B size: %lu\n", sizeof(struct B));
      printf("C size: %lu\n", sizeof(struct C));
}
```

Padding in C

What do you think the following output will be?

Word is 8 Bytes

What are the sizes of each struct?

- A: **24** bytes
- B: **12** bytes
- C: **8** bytes

```
struct A {
      char *ptr; // 8 bytes
      char c;    // 1 byte
      long num;  // 8 bytes
};

struct B {
      short s;   // 2 bytes
      int i;     // 4 bytes
      short s2;  // 2 bytes
};

struct C {
      int i;     // 4 bytes
      short s;   // 2 bytes
      short s2;  // 2 bytes
};

int main() {
      printf("A size: %lu\n", sizeof(struct A));
      printf("B size: %lu\n", sizeof(struct B));
      printf("C size: %lu\n", sizeof(struct C));
}
```

Padding in C cont.

- Let's revisit struct B from last slide
- What goes where?
- What did you get from your calculation?

```
struct B {
      short s;    // 2 bytes
      int i;      // 4 bytes
      short s2;   // 2 bytes
};

struct C {
      int i;      // 4 bytes
      short s;    // 2 bytes
      short s2;   // 2 bytes
};
```

B

C

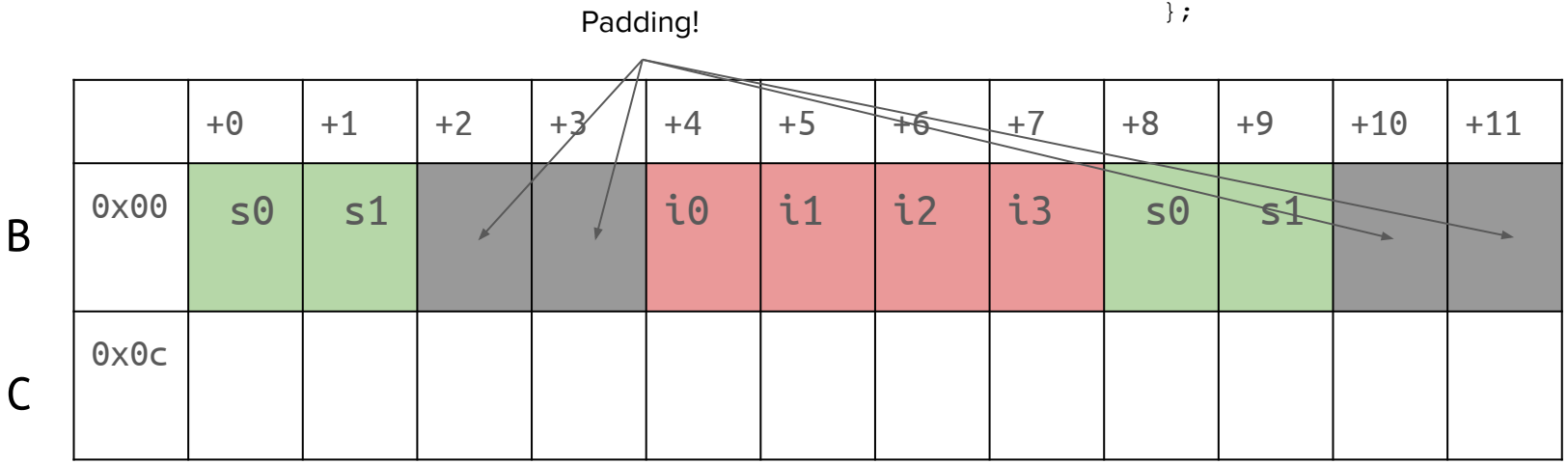
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
0x00												
0x0c												

Padding in C cont.

```
struct B {
      short s;    // 2 bytes
      int i;      // 4 bytes
      short s2;   // 2 bytes
};

struct C {
      int i;      // 4 bytes
      short s;    // 2 bytes
      short s2;   // 2 bytes
};
```

- What did we get?



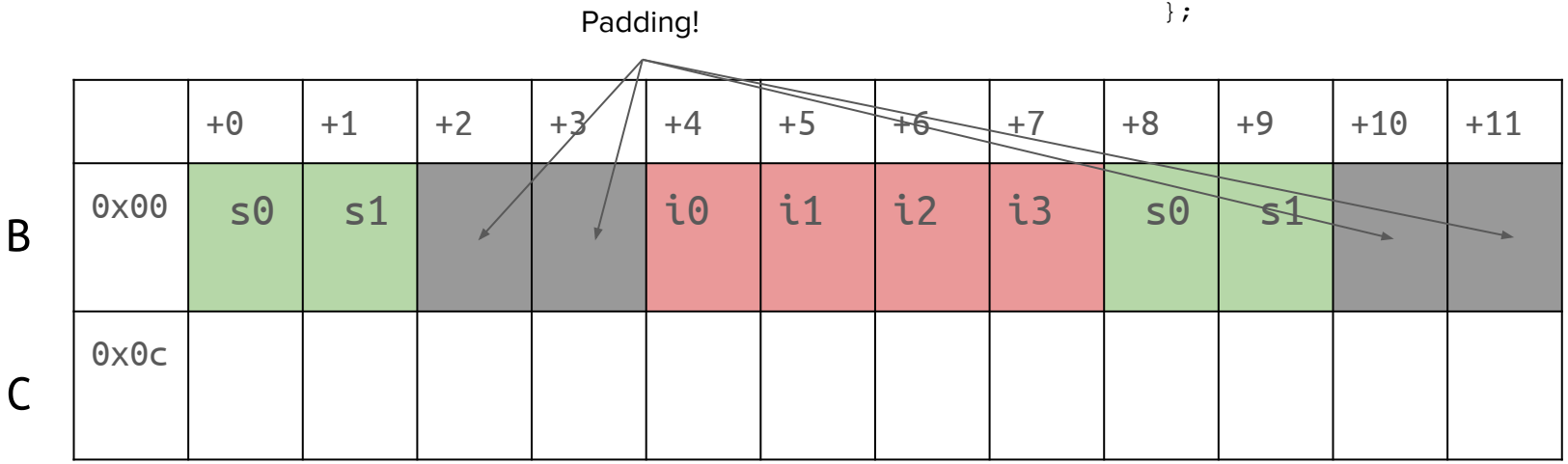
- Struct B takes up **12 bytes** of space
  - Memory addresses 0x02, 0x03, 0x0a, 0x0b are used as padding

Padding in C cont.

```
struct B {
      short s;    // 2 bytes
      int i;      // 4 bytes
      short s2;   // 2 bytes
};

struct C {
      int i;      // 4 bytes
      short s;    // 2 bytes
      short s2;   // 2 bytes
};
```

- Now let's do struct C
  - Has the **same members** as B, just arranged differently



- Struct B takes up **12 bytes** of space
  - Memory addresses 0x02, 0x03, 0x0a, 0x0b are used as padding

Padding in C cont.

- What did we get for struct C?

```
struct B {
      short s;    // 2 bytes
      int i;      // 4 bytes
      short s2;   // 2 bytes
};

struct C {
      int i;      // 4 bytes
      short s;    // 2 bytes
      short s2;   // 2 bytes
};
```

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
B 0x00	s0	s1			i0	i1	i2	i3	s0	s1		
C 0x0c	i0	i1	i2	i3	s0	s1	s0	s1				

- Struct B takes up **12 bytes** of space in mem
- Struct C takes up **8 bytes** of space in mem

Padding in C cont.

```
struct B {
      short s;    // 2 bytes
      int i;      // 4 bytes
      short s2;   // 2 bytes
};

struct C {
      int i;      // 4 bytes
      short s;    // 2 bytes
      short s2;   // 2 bytes
};
```

- Rule of thumb: Arranging struct members in **decreasing** order of size will minimize padding!
- Will allow more efficient use of memory space while retaining memory access efficiency

B

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
0x00	s0	s1			i0	i1	i2	i3	s0	s1		

C

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
0x0c	i0	i1	i2	i3	s0	s1	s0	s1				

- Struct B takes up **12 bytes** of space in mem
- Struct C takes up **8 bytes** of space in mem

Today’s Attendance Number

67,890

LC-2200
Instruction Set Architecture

Problem

- We need a LC-2200 assembly language function, maxmindiff, that computes the difference between the largest and smallest elements in an integer array
- Arguments: the address of the array and the length of the array
- Return value: the difference of the max and min elements in the array

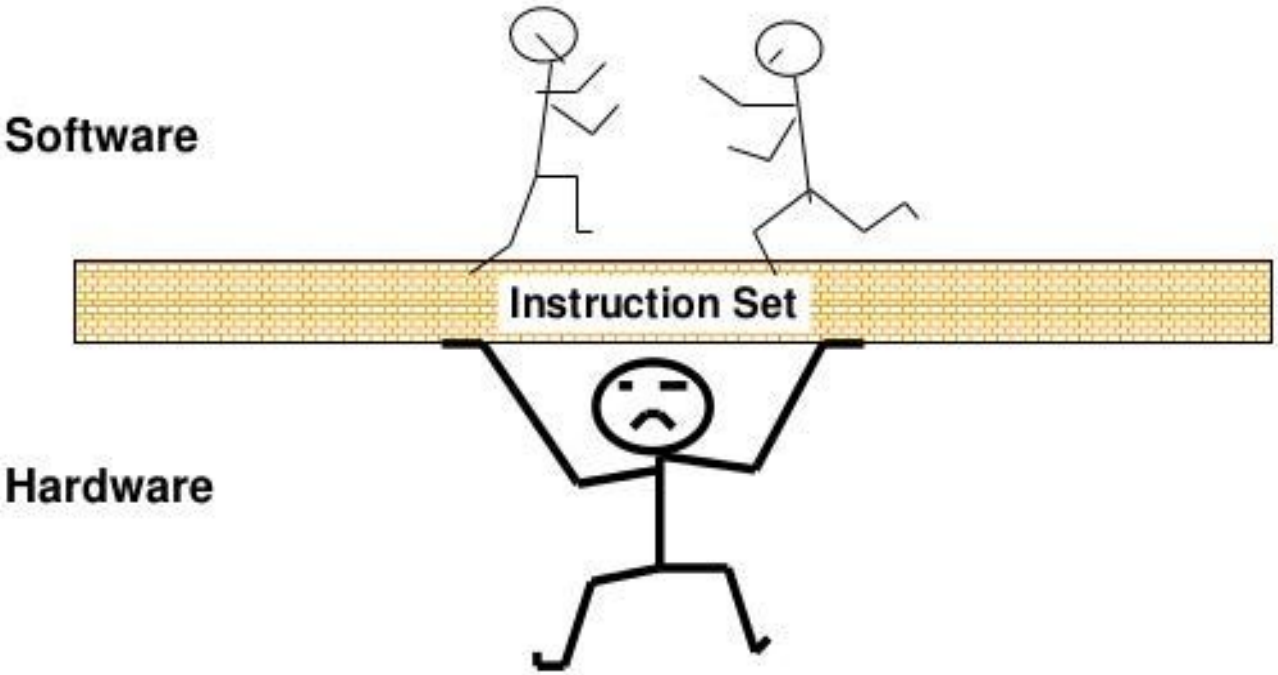
How do we go about this? What do we need to know?

In a bit, we'll use this pseudo-code

```
int maxmindiff(int a[], int alen) {

      int min, max, i;
      min = a[0];
      max = a[0];
      for (i = 1; i < alen; i++) {
          if (a[i] < min)
              min = a[i];
          if (a[i] > max)
             max = a[i];
      }
      return max - min;

}
```

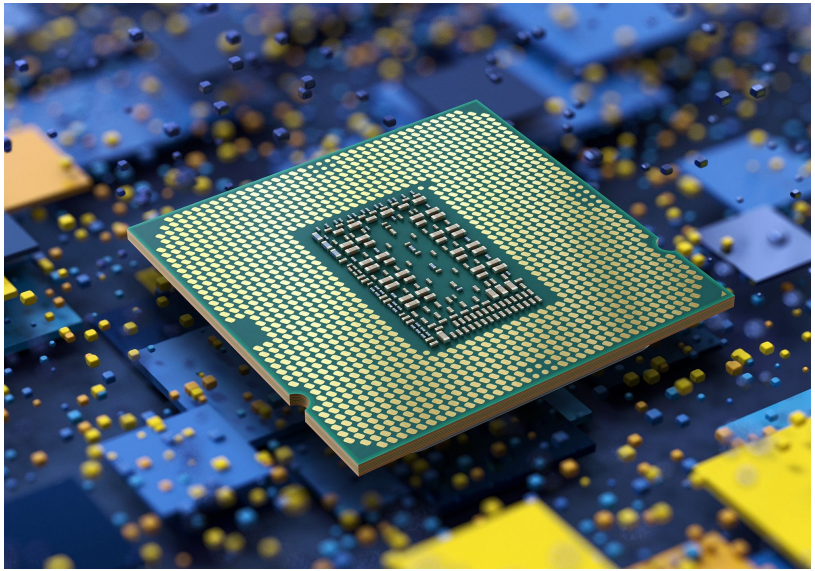

Software
Instruction Set
Hardware

Instruction Set Architecture

- Provides an abstraction for using an implementation of a data path (usually hardware, but it can be software)
- The implementation (hardware or software) interprets the machine instructions described by the ISA
- The ISA provides a target for a compiler’s output
- Ex: x86, ARM, RISC-V, MIPS, POWERPC, etc ...

LC-2200 Architectural Details

- 32-bit Architecture
- 32-bit word addressable memory
- Fixed Length (32 bit) Instructions
- 16 General Purpose Registers
- Load-Store Architecture
  - All data operations occur on registers



LC 2200 Instruction Set

Mnemonic Example	Format	Opcode	Action Register Transfer Language
add **add $v0, $a0, $a1**	R	0 0000_2	Add contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← $a0 + $a1
nand **nand $v0, $a0, $a1**	R	1 0001_2	Nand contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← ~($a0 && $a1)
addi **addi $v0, $a0, 25**	I	2 0010_2	Add Immediate value to the contents of reg Y and store the result in reg X. RTL: $v0 ← $a0 + 25
lw **lw $v0, 0x42($fp)**	I	3 0011_2	Load reg X from memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: $v0 ← MEM[$fp + 0x42]
sw **sw $a0, 0x42($fp)**	I	4 0100_2	Store reg X into memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: MEM[$fp + 0x42] ← $a0
beq **beq $a0, $a1, done**	I	5 0101_2	Compare the contents of reg X and reg Y. If they are the same, then branch to the address PC+1+OFFSET, where PC is the address of the beq instruction. RTL: if($a0 == $a1) PC ← PC+1+OFFSET

Note: For programmer convenience (and implementer confusion), the assembler computes the OFFSET value from the number or symbol given in the instruction and the assembler's idea of the PC. In the example, the assembler stores done-(PC+1) in OFFSET so that the machine will branch to label "done" at run time.			
jalr **jalr $at, $ra**	J	6 0110_2	First store PC+1 into reg Y, where PC is the address of the jalr instruction. Then branch to the address now contained in reg X. Note that if reg X is the same as reg Y, the processor will first store PC+1 into that register, then end up branching to PC+1. RTL: $ra ← PC+1; PC ← $at Note that an **unconditional jump** can be realized using **jalr $ra, $t0**, and discarding the value stored in $t0 by the instruction. This is why there is no separate jump instruction in LC-2200.
nop	n.a.	n.a.	Actually a pseudo instruction (i.e. the assembler will emit: add $zero, $zero, $zero
halt **halt**	O	7 0111_2	

Additions to the textbook instructions

lea `lea $v0, 0x10`	I	9 1001_2	Load reg X from with PC-relative address. The effective address is formed by adding OFFSET to the contents of the incremented PC. RTL: $v0 <- PC + 1 + 0x10

blt, bgt
`blt $a0, $a1,`
`done`
 	
`bgt $a0, $a1,`
`done`
	I	8
		1000_2
		
		10
		1010_2
			Compare the contents of reg X and reg Y. If X is less than Y (greater than), then branch to the address PC+1+OFFSET, where PC is the address of the blt(bgt) instruction.
			RTL: if ($a0 < $a1)
			PC <- PC+1+OFFSET

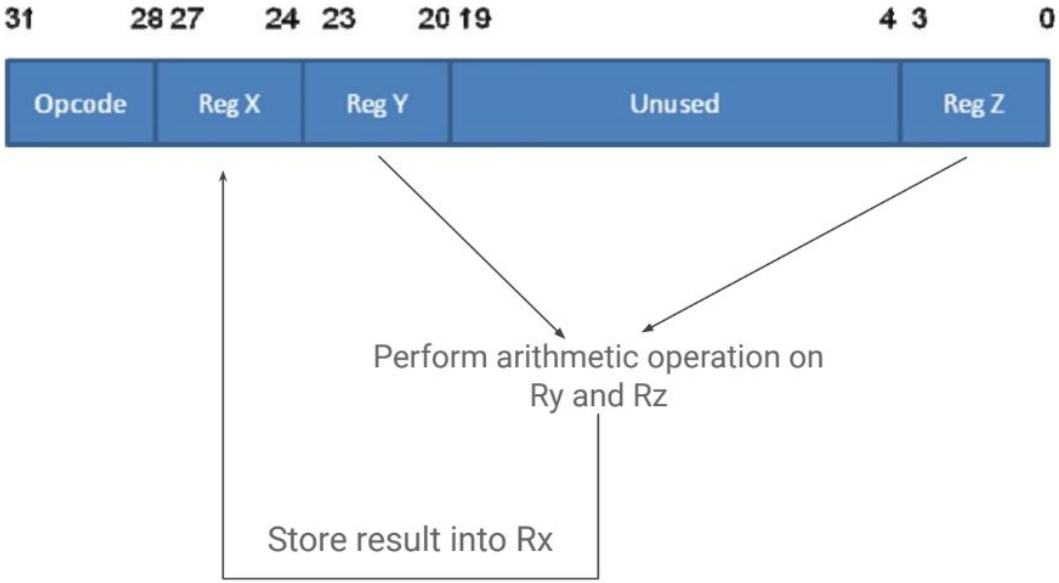
Addressing Mode

- The format of the instructions, and how the data within them is accessed.
- LC-2200 has 4 main types
  - R-type (Register)
  - I-type (Immediate)
  - J-type (Jump)
  - O-type also exists, but its sole use is NOP and HALT
    - Indicates no operands at all!

R-Type Instructions

Uses 3 operands contained in registers.

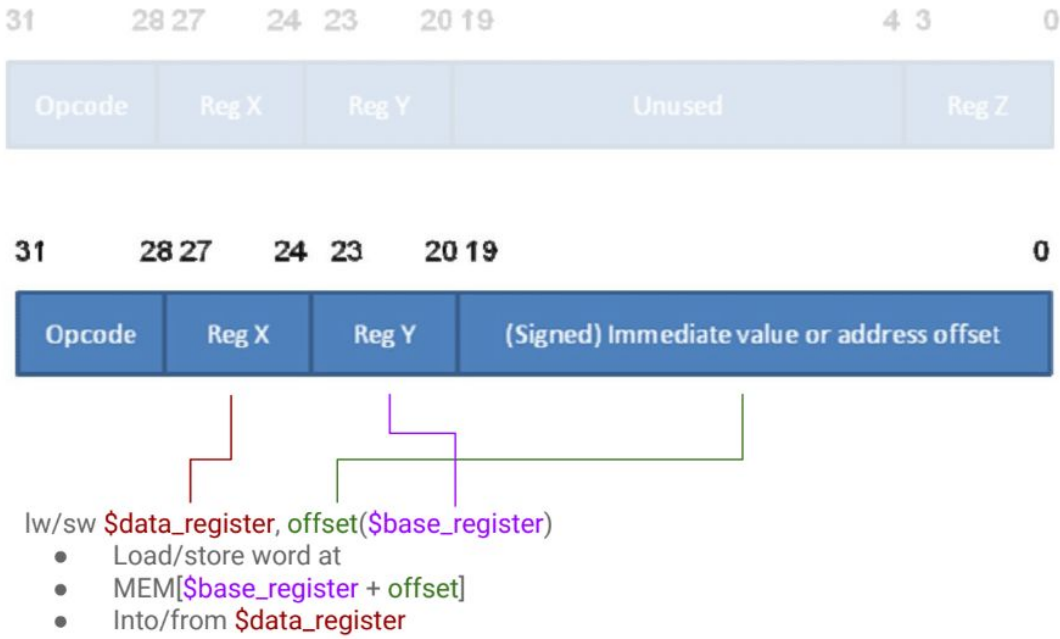
- **ADD** $v0, $a0, $a1
  - Opcode 0b0000
- **NAND** $v0, $a0, $a1
  - Opcode 0b0001



I-Type Instructions

Uses an immediate value in the instruction, not one pulled from a register or memory.

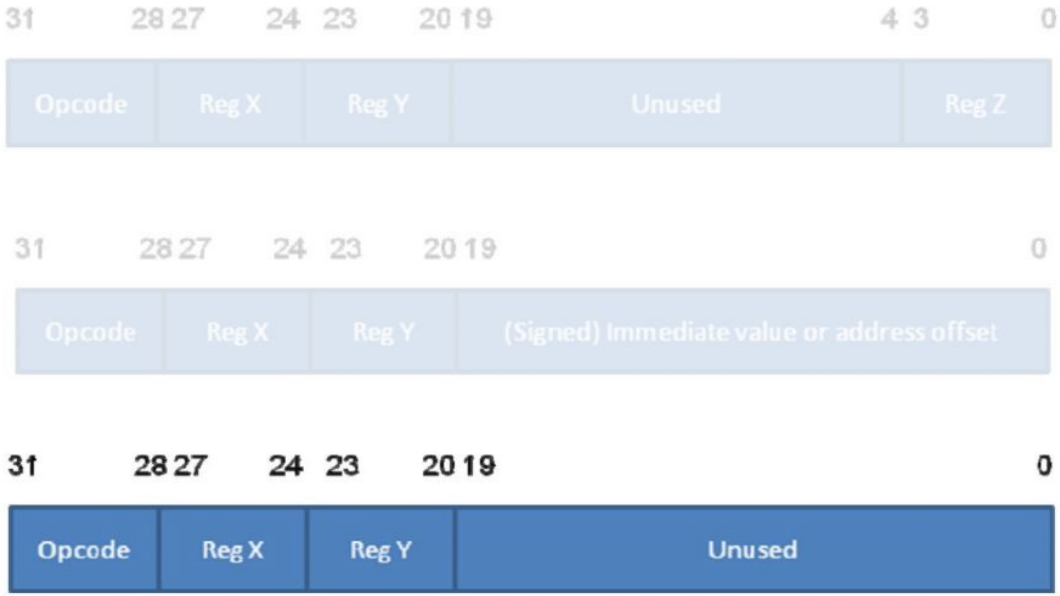
- **ADDI** $v0, $a1, 1
  - Opcode 0b0010
- **LW** $v0, 0x2($a1)
  - Opcode 0b0011
- **SW** $v0, 0x2($a1)
  - Opcode 0b0100
- **LEA** $a0, LABEL
  - Opcode 0b1001
- **BEQ** $a0, $a1, LABEL
  - Opcode 0b0101
- **BLT** $a0, $a1, LABEL
  - Opcode 0b1000
- **BGT** $a0, $a1, LABEL
  - Opcode 0b1010

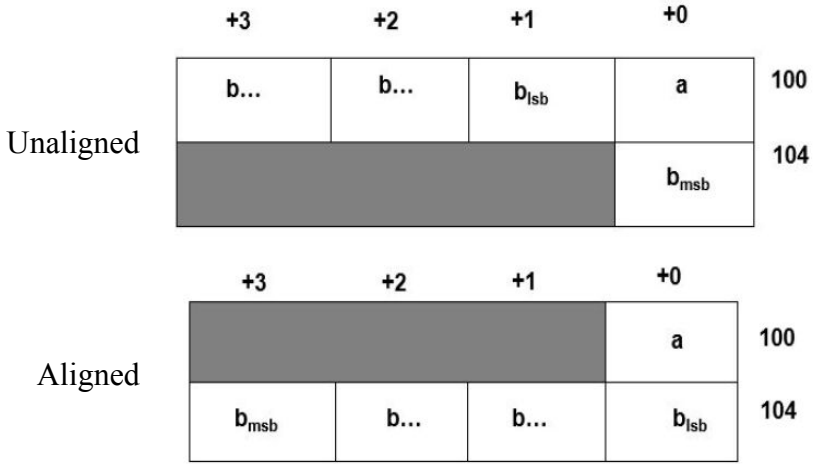


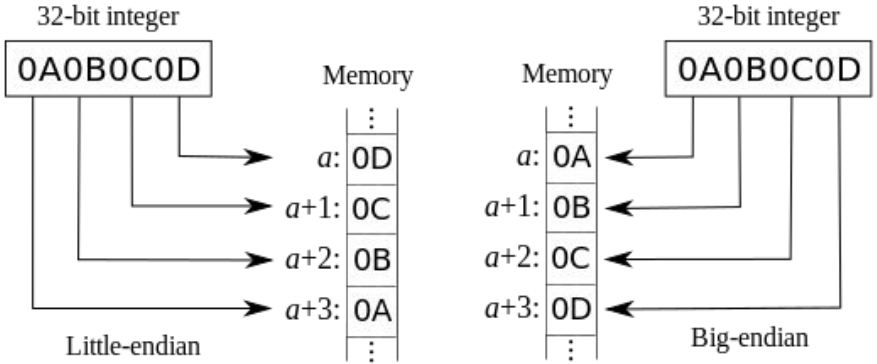
J-Type Instructions

Uses exactly two registers, the rest of the instruction space is unused.

- **JALR** $at, $ra
  - Stores the current PC in $ra, then jumps to $at







Mnemonic Example	Format	Opcode	Action Register Transfer Language
add **add $v0, $a0, $a1**	R	0 0000_2	Add contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← $a0 + $a1
nand **nand $v0, $a0, $a1**	R	1 0001_2	Nand contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← ~($a0 && $a1)
addi **addi $v0, $a0, 25**	I	2 0010_2	Add Immediate value to the contents of reg Y and store the result in reg X. RTL: $v0 ← $a0 + 25
lw **lw $v0, 0x42($fp)**	I	3 0011_2	Load reg X from memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: $v0 ← MEM[$fp + 0x42]
sw **sw $a0, 0x42($fp)**	I	4 0100_2	Store reg X into memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: MEM[$fp + 0x42] ← $a0
beq **beq $a0, $a1, done**	I	5 0101_2	Compare the contents of reg X and reg Y. If they are the same, then branch to the address PC+1+OFFSET, where PC is the address of the beq instruction. RTL: if($a0 == $a1) PC ← PC+1+OFFSET
Note: For programmer convenience (and implementer confusion), the assembler computes the OFFSET value from the number or symbol given in the instruction and the assembler's idea of the PC. In the example, the assembler stores done-(PC+1) in OFFSET so that the machine will branch to label "done" at run time.			
jalr **jalr $at, $ra**	J	6 0110_2	First store PC+1 into reg Y, where PC is the address of the jalr instruction. Then branch to the address now contained in reg X. Note that if reg X is the same as reg Y, the processor will first store PC+1 into that register, then end up branching to PC+1. RTL: $ra ← PC+1; PC ← $at Note that an **unconditional jump** can be realized using **jalr $ra, $t0**, and discarding the value stored in $t0 by the instruction. This is why there is no separate jump instruction in LC-2200.
nop	n.a.	n.a.	Actually a pseudo instruction (i.e. the assembler will emit: add $zero, $zero, $zero
halt **halt**	O	7 0111_2	

Let’s try to implement this now!

```
int maxmindiff(int a[], int alen) {

      int min, max, i;
      min = a[0];
      max = a[0];
      for (i = 1; i < alen; i++) {
          if (a[i] < min)
              min = a[i];
          if (a[i] > max)
             max = a[i];
      }
      return max - min;

}
```

Main program stub for testing

```
lea $t0, stack           ! initialize the stack
pointer
lw   $t0, 0x0($t0)
lw   $sp, 0($t0)

lea  $a0, a        ! put address of a as argument 1
lw   $a0, 0x0($a0)
addi $a1, $zero, 12      ! put the value from 12 as
argument 2

lea $at, maxmindiff      ! call our function
jalr $ra, $at

halt

stack:.fill 0x8000
```

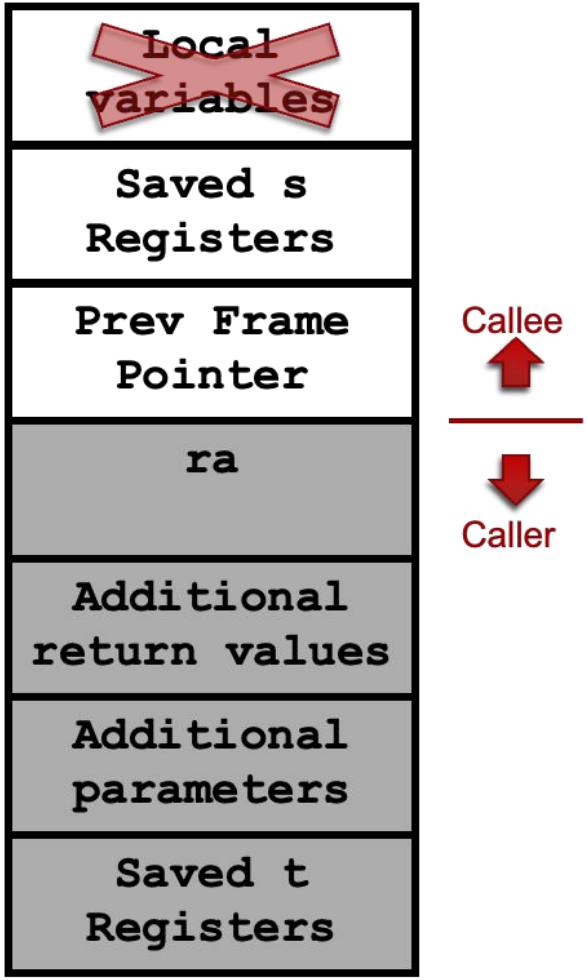
```
stack .fill 0x8000

a:
    .fill 7
    .fill 12
    .fill 8
    .fill 23
    .fill 2
    .fill 15
    .fill 9
    .fill 12
    .fill 10
    .fill 4
    .fill 6
    .fill 23
```

What do we need in the stack?

Since we're the callee, we just need to handle callee responsibilities.

So what does that look like?



Code to set up and tear down the stack frame

```
! int maxmindiff(int a[], int alen) {
```

```
maxmindiff:
addi $sp, $sp, -1    ! Push old $fp
sw   $fp, 0($sp)
addi  $fp, $sp, 0    ! Set new $fp
addi  $sp, $sp, -3   ! Push the 3 s
registers
sw   $s0, -1($fp)
sw   $s1, -2($fp)
sw   $s2, -3($fp)
```

```
     !int min, max, i;
     !min = a[0];
     !max = a[0];
```

```
     !for (i = 1; i < alen; i++) {
         !if (a[i] < min)
             !min = a[i];
         !if (a[i] > max)
           ! max = a[i];
     !}
     !return max - min;
```

```
lw   $s2, -3($fp)    ! Restore the 3 s registers
lw   $s1, -2($fp)
lw   $s0, -1($fp)
lw   $fp, 0($fp)! Restore the old $fp
addi $sp, $sp, 4! Pop all 4 words off the stack
jalr $at, $zero ! Return

!}
```

Implementing the pseudo-code

```
!int min, max, i;
!$s0 is min, $s1 is max, $s2 is i
!$a0 is a, $a1 is alen
!min = a[0];
```

Implementing the pseudo-code

```
!int min, max, i;
```

!$s0 is min, $s1 is max, $s2 is i

!$a0 is a, $a1 is alen

```
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
```

Implementing the pseudo-code

```
!int min, max, i;
```

!$s0 is min, $s1 is max, $s2 is i

!$a0 is a, $a1 is alen

```
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
```

Implementing the pseudo-code

```
!int min, max, i;
!$s0 is min, $s1 is max, $s2 is i
!$a0 is a, $a1 is alen
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
addi $t2, $zero, 12
addi $s2, $zero, 1
loop:
bgt $s2, $t2, endloop
beq $s2, $t2, endloop
```

```
!if (a[i] < min)
```

Implementing the pseudo-code

```
!int min, max, i;
!$s0 is min, $s1 is max, $s2 is i
!$a0 is a, $a1 is alen
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
addi $t2, $zero, 12
addi $s2, $zero, 1
loop:
bgt $s2, $t2, endloop
beq $s2, $t2, endloop
```

```
!if (a[i] < min)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    bgt $t0, $s0, nomin
    beq $t0, $s0, nomin
    !min = a[i];
```

Implementing the pseudo-code

```
!int min, max, i;
!$s0 is min, $s1 is max, $s2 is i
!$a0 is a, $a1 is alen
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
addi $t2, $zero, 12
addi $s2, $zero, 1
loop:
bgt $s2, $t2, endloop
beq $s2, $t2, endloop
```

```
!if (a[i] < min)
   add $t0, $s2, $a0
   lw  $t0, 0($t0)
   bgt $t0, $s0, nomin
   beq $t0, $s0, nomin
    !min = a[i];
    addi $s0, $t0, 0
nomin:
!if (a[i] > max)
```

Implementing the pseudo-code

```
!int min, max, i;
!$s0 is min, $s1 is max, $s2 is i
!$a0 is a, $a1 is alen
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
addi $t2, $zero, 12
addi $s2, $zero, 1
loop:
bgt $s2, $t2, endloop
beq $s2, $t2, endloop
```

```
!if (a[i] < min)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    bgt $t0, $s0, nomin
    beq $t0, $s0, nomin
    !min = a[i];
    addi $s0, $t0, 0
nomin:
!if (a[i] > max)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    blt $t0, $s1, nomax
    beq $t0, $s1, nomax
    ! max = a[i];
```

Implementing the pseudo-code

```
!int min, max, i;
!$s0 is min, $s1 is max, $s2 is i
!$a0 is a, $a1 is alen
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
addi $t2, $zero, 12
addi $s2, $zero, 1
loop:
bgt $s2, $t2, endloop
beq $s2, $t2, endloop
```

```
!if (a[i] < min)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    bgt $t0, $s0, nomin
    beq $t0, $s0, nomin
    !min = a[i];
    addi $s0, $t0, 0
nomin:
!if (a[i] > max)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    blt $t0, $s1, nomax
    beq $t0, $s1, nomax
    ! max = a[i];
    addi $s1, $t0, 0
nomax:
 !}
```

Implementing the pseudo-code

```
!int min, max, i;
```
!$s0 is min, $s1 is max, $s2 is i

!$a0 is a, $a1 is alen

```
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
addi $t2, $zero, 12
addi $s2, $zero, 1
loop:
bgt $s2, $t2, endloop
beq $s2, $t2, endloop
!if (a[i] < min)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    bgt $t0, $s0, nomin
    beq $t0, $s0, nomin
    !min = a[i];
    addi $s0, $t0, 0
nomin:
!if (a[i] > max)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    blt $t0, $s1, nomax
    beq $t0, $s1, nomax
    ! max = a[i];
    addi $s1, $t0, 0
nomax:
  !}
 addi  $s2, $s2, 1
 beq   $zero, $zero, loop
endloop:
!return max - min;
```

Implementing the pseudo-code

```
!int min, max, i;
```
!$s0 is min, $s1 is max, $s2 is i
!$a0 is a, $a1 is alen
```
!min = a[0];
lw $s0, 0($a0)
!max = a[0];
lw $s1, 0($a0)
!for (i = 1; i < alen; i++) {
addi $t2, $zero, 12
addi $s2, $zero, 1
loop:
bgt $s2, $t2, endloop
beq $s2, $t2, endloop
!if (a[i] < min)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    bgt $t0, $s0, nomin
    beq $t0, $s0, nomin
        !min = a[i];
    addi $s0, $t0, 0
nomin:
!if (a[i] > max)
    add $t0, $s2, $a0
    lw  $t0, 0($t0)
    blt $t0, $s1, nomax
    beq $t0, $s1, nomax
    ! max = a[i];
    addi $s1, $t0, 0
nomax:
  !}
 addi  $s2, $s2, 1
 beq   $zero, $zero, loop
endloop:
!return max - min;
nand $t0, $s0, $s0
addi $t0, $t0, 1
add $v0, $s1, $t0
```

Thanks for Attending :)