# CS61C MT1 Review

Led by Harrison & Rebecca

# Quick Disclaimer

- Not intended to replace your studying
- Summary of some of the “most important” concepts we want you to understand/know
- Don’t assume scope of MT is limited to these slides
- Didn’t want to exhaust all great midterm questions as practice

# Midterm Info (on Course Policies)

- On each exam, you will be given a MIPS Green Sheet attached to the exam.
- Midterm 1: Covers up to and including the 07/02 lecture on CALL.
- Midterm 1: One 8.5"x11", double-sided cheat sheet.
- The clobber policy allows you to override your Midterm 1 and Midterm 2 scores with the score of the corresponding section on the final exam if you perform better on the respective sections of the final.
- During lecture time, 07/09

# Midterm Tips

- Make sure to still practice on your own!!!
- Be able to understand, read, and fill in code
- Understand how to read the MIPS green sheet
- Identify concepts and apply what you know
- Put “factual” information on your cheatsheet!

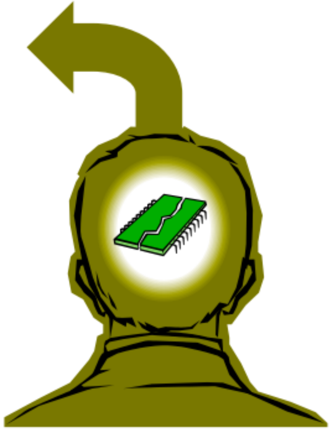
# Back to Basics - Number Rep.

- Everything is a number
- Numbers stored at certain fixed widths
- Common bases:
  - Binary (base 2)
  - Hexadecimal (base 16)
  - Decimal (base 10)
- Signed vs. Unsigned
- IEC prefixes

# The way to remember #s

- What is $2^{34}$? How many bits addresses (I.e., what's `ceil` $\texttt{log}_2$ = `lg` of) 2.5 TiB?
- Answer! $2^{XY}$ means…

| | |
|---|---|
| X=0 ⇒ --- | Y=0 ⇒ 1 |
| X=1 ⇒ kibi ~$10^3$ | Y=1 ⇒ 2 |
| X=2 ⇒ mebi ~$10^6$ | Y=2 ⇒ 4 |
| X=3 ⇒ gibi ~$10^9$ | Y=3 ⇒ 8 |
| X=4 ⇒ tebi ~$10^{12}$ | Y=4 ⇒ 16 |
| X=5 ⇒ pebi ~$10^{15}$ | Y=5 ⇒ 32 |
| X=6 ⇒ exbi ~$10^{18}$ | Y=6 ⇒ 64 |
| X=7 ⇒ zebi ~$10^{21}$ | Y=7 ⇒ 128 |
| X=8 ⇒ yobi ~$10^{24}$ | Y=8 ⇒ 256 |
| | Y=9 ⇒ 512 |



MEMORIZE!





Credits to Dan Garcia

# MT Question -

- (Sp13 1.k) Using any scheme, what is the fewest number of bits required to “address” all phone numbers (including area code)?

# MT Question -

- (Sp13 1.k) Using any scheme, what is the fewest number of bits required to “address” all phone numbers (including area code)?
- 10^10 possible phone numbers.
- **ceil(log_2(10^10)) ≅ 34**

# Convert!

| Decimal | Hex | Binary |
|---|---|---|
| 59 | | |
| 37 | | |
| | 0x61C | |
| | 0xFA1 | |
| | | 0b0101 0101 |
| | | 0b1001 1010 |

# Convert!

| Decimal | Hex | Binary |
|---|---|---|
| 59 | 0x3B | 0b0011 1010 |
| 37 | 0x25 | 0b0010 0101 |
| 6*16^2 + 1 * 16^1 + 12 | 0x61C | 0b0110 0001 1100 |
| 15*16^2 + 10 *16^1 + 1 | 0xFA1 | 0b1111 1010 0001 |
| 5*16^1 + 5 | 0x55 | 0b0101 0101 |
| 9*16^1 + 10 | 0x9A | 0b1001 1010 |

# 2's Complement

- Know the range for N-bit numbers
  - -2^(N-1) : 2^(N-1)-1
- Understand advantages/disadvantages.
  - Can represent negative and only one zero
  - Smaller range of positive numbers represented (but can be remedied)

# 2's Complement - Conversions

- Take number, invert bits, and add 1
- Min number of bits?

- Ex: 16 -> -16
  - 16 = 0b0001 0000
  - Invert = 0b1110 1111
  - Add 1 = 0b1111 0000
- Ex: -16 -> 16
  - -16 = 0b1111 0000
  - Invert = 0b0000 1111
  - Add 1 = 0b0001 0000

# Overflow

- Occurs when
  - Carry into MSB ≠ Carry out MSB
  - Two positives results in a negative
  - Two negatives result in a positive
- Result of wrong sign

# MT Questions - Warmup

- (Sp15 1.2) For two n-bit numbers, what is the difference between the largest unsigned number and the largest two's-complement number? In other words, what is MAX_UNSIGNED_INT - MAX_SIGNED_INT? Write your answer in terms of n.

# MT Questions - Warmup (Solution)

- MAX_UNSIGNED_INT = 2^n-1
- MAX_SIGNED_INT = 2^(n-1) - 1
- Therefore,
  - (2^n-1) - (2^(n-1) - 1) = **2^(n-1)**

# C[S61C] Quick Review

- “Why C?: we can write programs that allow us to exploit underlying features of the architecture - **memory managemen**t, special instructions, parallelism”

# Quick Summary

- function-oriented, structs, pass by value
- must declare types
- constants
- stack/heap management
- 0 or NULL == FALSE
- Anything that isn’t FALSE == TRUE
- structs

Note: probably need to know more than this

# Some pointers?


XKCD 138

# P->O->I->N->T->E->R->S

- Pointer: variable that contains address of a variable
  - int *x; - variable is address of an int
  - x = &y; - assign address of y to x
  - z = *x; - assign value at address x to z
- Pointers passed to a function get copy of pointer
- Why pointers?
  - Easier to pass pointer rather than large struct/array
  - Pointers to pointers, N-d arrays
  - Linked lists
- When in doubt, draw boxes and arrows!

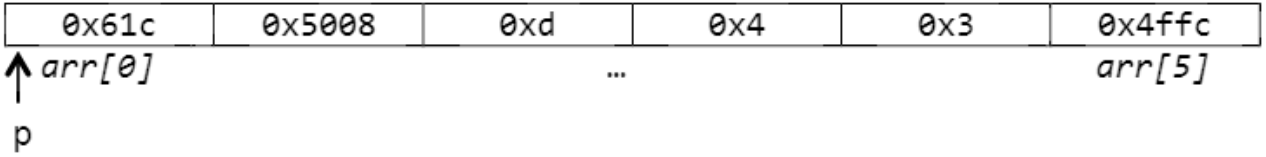
# Pointers and Arrays

- K&R Section 5.3
- KEY DIFFERENCES:
  - “A pointer is a variable, but an array name is not a variable”
  - location of initial element is passed into function
  - sizeof(pointer) vs sizeof(array)
  - &pointer vs &array
  - pointer arithmetic

1) Assume you are given an `int` array `arr`, with a pointer `p` to its beginning:

```
int arr[] = {0x61c, 0x5008, 0xd, 0x4, 0x3, 0x4ffc};
int *p = arr;
```

Suppose `arr` is at location `0x5000` in memory, i.e., the value of `p` if interpreted as an integer is `0x5000`. To visualize this scenario:




Assume that integers and pointers are both 32 bits. What are the values of the following expressions? If an expression may cause an error, write “Error” instead.

a) `*(p+3) =`

b) `p[4] =`

c) `*(p+5) + p[3] =`

d) `*(int*)(p[1]) =`

e) `*(int*)(*(p+5)) =`

1) Assume you are given an `int` array `arr`, with a pointer `p` to its beginning:

```
int arr[] = {0x61c, 0x5008, 0xd, 0x4, 0x3, 0x4ffc};
int *p = arr;
```

Suppose `arr` is at location `0x5000` in memory, i.e., the value of `p` if interpreted as an integer is `0x5000`. To visualize this scenario:

| | | | | | |
|---|---|---|---|---|---|
| 0x61c | 0x5008 | 0xd | 0x4 | 0x3 | 0x4ffc |
| ↑ *arr[0]* | | … | | | *arr[5]* |

p

Assume that integers and pointers are both 32 bits. What are the values of the following expressions? If an expression may cause an error, write "Error" instead.

a) `*(p+3) =` 0x4

b) `p[4] =` 0x3

c) `*(p+5) + p[3] =` 0x5000

d) `*(int*)(p[1]) =` 0xd(13)

e) `*(int*)(*(p+5)) =` error(out of bounds)

# C Memory Management

- Stack, heap, static data, code
  - What goes where?
  - How to create variables in these address spaces?
  - Be careful of how you use malloc, realloc, etc.
  - Free your memory!

Sp07 1e. Indicate how much memory is used on each line. If zero, leave it blank.

| Static | Stack | Heap | typedef struct bignum { |
|---|---|---|---|
| | | | int len; |
| | | | char *num; |
| | | | char description[100]; |
| | | | } bignum_t |
| | | | bignum_t *res; |
| | | | int main() { |
| | | | bignum_t b; |
| | | | b.num = (char*) malloc(5*sizeof(char)); |
| | | | // more code below |

Sp07 1e. Indicate how much memory is used on each line. If zero, leave it blank.

| Static | Stack | Heap | typedef struct bignum { |
|---|---|---|---|
| | | | int len; |
| | | | char *num; |
| | | | char description[100]; |
| | | | } bignum_t |
| 4 | | | bignum_t *res; |
| | | | int main() { |
| | 108 | | bignum_t b; |
| | | 5 | b.num = (char*) malloc(5*sizeof(char)); |
| | | | // more code below |

# MT Practice

- (Sp15 #4.2) Given function def:
  - int* to_array(sll_node *sll, int size);
  - size = length of array to be created
- Complete to_array() to convert a linked list to array.
- typedef struct node {

  int value;

  struct node* next; // pointer to next element

  } sll_node;

```
int *to_array(sll_node *sll, int size) {
	int i = 0;
	int* arr = malloc(size * sizeof(int)); // allocate array
	while(sll) { // check for null
		arr[i] = sll->value; // set values
		sll = sll->next; // move linked list along
		i++;
	}
	return arr;
}
```

# MT Practice

- (Sp15 #4.3) Given function def:
  - void delete_even(sll_node *sll);
- Complete delete() to delete every second element of the linked list.

```
void delete_even(sll_node *sll) {
    sll_node *temp;
    if (!sll || !(sll->next)) return; // base case
    temp = sll->next;
    sll->next = temp->next (or sll->next->next);
    free(temp); // delete “2nd” element
    delete_even(sll->next); // recursion!
}
```

# Strings

- Array of characters, last character = null terminator (‘\0’, 0)
- char s[SIZE] = “can be modified”;
    - char s[SIZE] = {‘c’, ‘a’, ‘n’, …., ‘\0’};
- char *s = “behavior undefined but usually not modifiable”;

# Mini-break(?)

Two strings walk into a bar and sit down. The bartender says, “So what’ll it be?”

The first string says, “I think I’ll have a beer quag fulk boorg jdk^CjfdLk jk3s d#f67howe%^U r89nvy~~owmc63^Dz x.xvcu”

“Please excuse my friend,” the second string says, “He isn’t null-terminated.”

# MIPS Review - Calling Conventions

**REGISTER NAME, NUMBER, USE, CALL CONVENTION**

| NAME | NUMBER | USE | PRESERVED ACROSS A CALL? |
|---|---|---|---|
| $zero | 0 | The Constant Value 0 | N.A. |
| $at | 1 | Assembler Temporary | No |
| $v0-$v1 | 2-3 | Values for Function Results and Expression Evaluation | No |
| $a0-$a3 | 4-7 | Arguments | No |
| $t0-$t7 | 8-15 | Temporaries | No |
| $s0-$s7 | 16-23 | Saved Temporaries | Yes |
| $t8-$t9 | 24-25 | Temporaries | No |
| $k0-$k1 | 26-27 | Reserved for OS Kernel | No |
| $gp | 28 | Global Pointer | Yes |
| $sp | 29 | Stack Pointer | Yes |
| $fp | 30 | Frame Pointer | Yes |
| $ra | 31 | Return Address | Yes |

Who needs to store their registers in the stack?

Caller: $t0 - $t9, $v0-$v1, $a0 - $a3, $ra

Callee: $s0

# MT Practice: Calling Conventions

```
sum_arr:   bne $a1, $0, non_zero
           addu $v0, $0, $0
           jr $ra
non_zero:  ____________________
           ____________________
           ____________________
           addiu $s0, $a0, 0
           lw $t0, 0($s0)
           addiu $a0, $a0, 4
           addiu $a1, $a1, -1
           ____________________
           jal sum_arr
           ____________________
           addu $v0, $v0, $t0
           ____________________
           ____________________
           ____________________
           jr $ra
```

**Fill in the blanks to finish this (inefficient) function to sum the elements of an array so it follows all the calling conventions. What simple fix could make it more efficient?**

# MT Practice: Calling Conventions

```
sum_arr:    bne $a1, $0, non_zero
            addu $v0, $0, $0
            jr $ra
non_zero:   addiu $sp, $sp, -12
            sw $s0, 8($sp)
            sw $ra, 4($sp)
            addiu $s0, $a0, 0
            lw $t0, 0($s0)
            addiu $a0, $a0, 4
            addiu $a1, $a1, -1
            sw $t0, 0($sp)
            jal sum_arr
            lw $t0, 0($sp)
            addu $v0, $v0, $t0
            lw $s0, 8($sp)
            lw $ra, 4($sp)
            addiu $sp $sp 12
            jr $ra
```

# MT Practice: Calling Conventions

```
sum_arr:   bne $a1, $0, non_zero
           addu $v0, $0, $0
           jr $ra
non_zero:  addiu $sp, $sp, -8
           sw $s0, 4($sp)
           sw $ra, 0($sp)
           addiu $s0, $a0, 0
           lw $t0, 0($s0)
           addiu $a0, $a0, 4
           addiu $a1, $a1, -1
           sw $t0, 0($sp)
           jal sum_arr
           lw $t0, 0($sp)
           addu $v0, $v0, $t0
           lw $s0, 4($sp)
           lw $ra, 0($sp)
           addiu $sp $sp 8
           jr $ra
```

# MIPS Review - Instructions

**Arithmetic**: add, addi, sub, addu, addiu, subu

**Memory**: lw, sw, lb, sb

**Decision**: beq, bne, slt, slti, sltu, sltiu

**Unconditional Branches** (Jumps): j, jal, jr

**PseudoInstructions**: move → add $0

subu → addu (negative imm)

li → addiu $0, imm

sd → sw 2x

ble → slt, bne

mul → mul, mflo

la, jump (far) → lui and ori

# MIPS Review - Data Transfer

"load **from** memory"

"store **to** memory"

lw $t0 8($s0) \\ treats $s0 as a pointer.
\\ Dereferences ($s0 + 8). Stores in $t0

sw $s0 0($a0) \\ Treats $a0 as a pointer.
\\ Dereferences and sets its value to $s0

# MT Practice: C → MIPS

```
int has_cycle(node *tortoise, node
*hare) {
      if (hare == tortoise) return 1;
      if (!hare || !hare->next) return 0;
      return has_cycle(tortoise->next,
            hare->next->next)
}
```

```
has_cycle: li $v0 1
           $beq $a0 $a1 done
           li $v0 0
           $beq _________ done
           ____________________
           beq __________ done
           ____________________
           ____________________
           addiu _______________
           ____________________
           ____________________
           ____________________
           addiu _______________
done:      jr $ra
```

# MT Practice: C → MIPS

```
int has_cycle(node *tortoise, node
*hare) {
        if (hare == tortoise) return 1;
        if (!hare || !hare->next) return 0;
        return has_cycle(tortoise->next,
                hare->next->next)
}
```

```
has_cycle: li $v0 1
           $beq $a0 $a1 done
           li $v0 0
           $beq $a1 $0 done
           lw $a1 4($a1)
           beq $a1 $0 done
           lw $a0 4($a0)
           lw $a1 4($a1)
           addiu $sp $sp -4
           sw $ra 0($sp)
           jal has_cycle
           lw $ra 0($sp)
           addiu $sp $sp 4
done:      jr $ra
```

# MT Practice: Mal -> Tal

Convert the following program to TAL Mips

```
li $s0 0x1234ABCD

mul $s0 $s0 $s0
```

# MT Practice: Mal -> Tal

Convert the following program to TAL Mips

```
li $s0 0x1234ABCD

mul $s0 $s0 $s0
```

```
lui $s0, 0x1234
ori $s0 $s0 0xABCD
mul $s0 $s0
mflo $s0
```

# MIPS Review - Instruction Formats

R

| 31 26 | 21 | 16 | 11 | 6 | 0 |
|---|---|---|---|---|---|
| **op** | **rs** | **rt** | **rd** | **shamt** | **funct** |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

I

| 31 26 | 21 | 16 | 0 |
|---|---|---|---|
| **op** | **rs** | **rt** | **immediate** |
| 6 bits | 5 bits | 5 bits | 16 bits |

J

| 31 26 | 0 |
|---|---|
| **op** | **target address** |
| 6 bits | 26 bits |

# MT Review

**REGISTER NAME, NUMI**

| NAME | NUMBER |
|---|---|
| $zero | 0 |
| $at | 1 |
| $v0-$v1 | 2-3 |
| $a0-$a3 | 4-7 |
| $t0-$t7 | 8-15 |
| $s0-$s7 | 16-23 |

**CORE INSTRUCTION SET**

| NAME, MNEMONIC | | FOR-MAT | OPERATION (in Verilog) | | OPCODE / FUNCT (Hex) |
|---|---|---|---|---|---|
| Add | `add` | R | R[rd] = R[rs] + R[rt] | (1) | $0 / 20_{hex}$ |
| Add Immediate | `addi` | I | R[rt] = R[rs] + SignExtImm | (1,2) | $8_{hex}$ |
| Add Imm. Unsigned | `addiu` | I | R[rt] = R[rs] + SignExtImm | (2) | $9_{hex}$ |
| Add Unsigned | `addu` | R | R[rd] = R[rs] + R[rt] | | $0 / 21_{hex}$ |
| And | `and` | R | R[rd] = R[rs] & R[rt] | | $0 / 24_{hex}$ |
| And Immediate | `andi` | I | R[rt] = R[rs] & ZeroExtImm | (3) | $c_{hex}$ |
| Load Halfword Unsigned | `lhu` | I | R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)} | (2) | $25_{hex}$ |
| Load Linked | `ll` | I | R[rt] = M[R[rs]+SignExtImm] | (2,7) | $30_{hex}$ |
| Load Upper Imm. | `lui` | I | R[rt] = {imm, 16'b0} | | $f_{hex}$ |
| Load Word | `lw` | I | R[rt] = M[R[rs]+SignExtImm] | (2) | $23_{hex}$ |
| Nor | `nor` | R | R[rd] = ~ (R[rs] \| R[rt]) | | $0 / 27_{hex}$ |

Convert hex to MIPS or vice versa (from last semester's final exam!)

i) lw $s0, 0($a0)

ii) 0x02021021

# MT Review

**REGISTER NAME, NUMI**

| NAME | NUMBER |
|---|---|
| \$zero | 0 |
| \$at | 1 |
| \$v0-\$v1 | 2-3 |
| \$a0-\$a3 | 4-7 |
| \$t0-\$t7 | 8-15 |
| \$s0-\$s7 | 16-23 |

**CORE INSTRUCTION SET**

| NAME, MNEMONIC | | FOR-MAT | OPERATION (in Verilog) | | OPCODE / FUNCT (Hex) |
|---|---|---|---|---|---|
| Add | add | R | R[rd] = R[rs] + R[rt] | (1) | 0 / $20_{hex}$ |
| Add Immediate | addi | I | R[rt] = R[rs] + SignExtImm | (1,2) | $8_{hex}$ |
| Add Imm. Unsigned | addiu | I | R[rt] = R[rs] + SignExtImm | (2) | $9_{hex}$ |
| Add Unsigned | addu | R | R[rd] = R[rs] + R[rt] | | 0 / $21_{hex}$ |
| And | and | R | R[rd] = R[rs] & R[rt] | | 0 / $24_{hex}$ |
| And Immediate | andi | I | R[rt] = R[rs] & ZeroExtImm | (3) | $c_{hex}$ |
| Load Halfword Unsigned | lhu | I | R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)} | (2) | $25_{hex}$ |
| Load Linked | ll | I | R[rt] = M[R[rs]+SignExtImm] | (2,7) | $30_{hex}$ |
| Load Upper Imm. | lui | I | R[rt] = {imm, 16'b0} | | $f_{hex}$ |
| Load Word | lw | I | R[rt] = M[R[rs]+SignExtImm] | (2) | $23_{hex}$ |
| Nor | nor | R | R[rd] = ~ (R[rs] \| R[rt]) | | 0 / $27_{hex}$ |

Convert hex to MIPS or vice versa (from last semester's final exam!)

i) lw \$s0, 0(\$a0)

0x8c900000

ii) 0x02021021

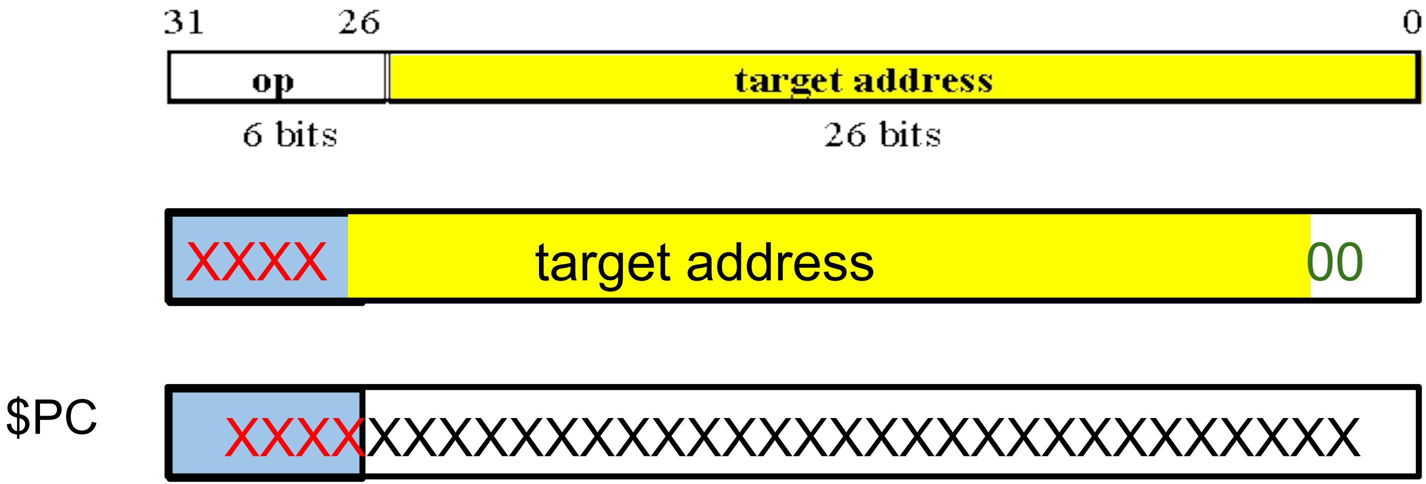
addu \$v0 \$s0 \$v0

# MIPS Review - Branching

| | |
|---|---|
| j | pseudodirect addressing |
| | PC = {PC+4}(31:28) + target address << 2 |
| jr | register addressing |
| | full 32 bit address stored in rs |
| beq/bne | PC-reative addressing |
| | PC = PC + 4 + imm << 2 |
| lw/lb/sw/sb | base displacement addressing |
| | (register) + immediate |

# Jump - PseudoDirect Addressing

$$PC = \{PC+4\}(31:28) + \text{target address} << 2$$




We can do this because instructions are word aligned!

# MIPS Review - Branching

jr

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

full 32 bit address stored in ${rs}

beq/bne

| op | rs | rt | immediate |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

PC = PC + 4 + SignExtImm << 2

lw/lb/sw/sb

| op | rs | rt | immediate |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

{rs} + SignExtImm

# MIPS Review - Branching: Range?

j

2^26 instructions;
max 2^28 addresses away

jr

2^32 addresses:
all of them!

beq/bne

2^16 instructions;
max 2^17 addresses away
in either direction

lw/lb/sw/sb

all again

# MT Practice

1) How would J-type instructions be affected (in terms of their “reach” aka how many instructions they can reach) if we relaxed the requirement that instructions be placed on word boundaries, and instead required them to be placed on *half- word* boundaries.

2) Building on the idea from the previous question, give a minor tweak to the MIPS ISA to allow us to use *true absolute addressing* (i.e., maximal “reach”) for all J-type instructions.

# MT Practice

1) How would J-type instructions be affected (in terms of their "reach" aka how many instructions they can reach) if we relaxed the requirement that instructions be placed on word boundaries, and instead required them to be placed on *half- word* boundaries.

> The range over which we could jump would be cut in half - you would have to allow a way to specify half-words, but you still cannot fit a full instruction in 2 bytes.

2) Building on the idea from the previous question, give a minor tweak to the MIPS ISA to allow us to use *true absolute addressing* (i.e., maximal "reach") for all J-type instructions.

> Only allow jumps to addresses which are multiples of 2^6 (instead of the 2^2, which comes from word alignment)

# MIPS Review - Large Immediates; Extensions

- How do we get from a 16 bit immediate to a 32 bit value?
- Sign Extention:

  1111111111111111 1010101111010100

- Zero Extention:

  0000000000000000 1010101111010100

# MIPS Review - Large Immediates; Extensions

- How do we get from a 16 bit immediate to a 32 bit value?
- Sign Extention:

1111111111111111 1010101111010100

literally everything else

- Zero Extention:

0000000000000000 1010101111010100

all logical instructions

# MIPS Review - Unsigned???

addiu

sltu

lbu

lui

# MIPS Review - Unsigned???

| | |
|---|---|
| addiu | no overflow error |
| sltu | unsigned comparison |
| lui | just kidding: that u stands for “upper” :P |
| lbu | ??????? |

# MIPS Review - Unsigned???

lbu base displacement addressing!

lbu $s0 -4($a0)

Actually has two things that need extending:

- the displacement for the address (16 bit imm) [Signed]
- the byte that we load into a 32 bit register [Unsigned]

# MT Practice

| Address | Instruction |
| --- | --- |
| 0x 1FCA5870 | 0x 0BFFFFFF |

What address are we jumping to?

# MT Practice

| Address | Instruction |
| --- | --- |
| 0x 1FCA5870 | 0x 0BFFFFFF |
| | 0b 0000 1011 1111 … |

What address are we jumping to?

0x 1FFFFFFC

0b 0001 1111 1111 1111 …. 1100

# Assembler Stuff

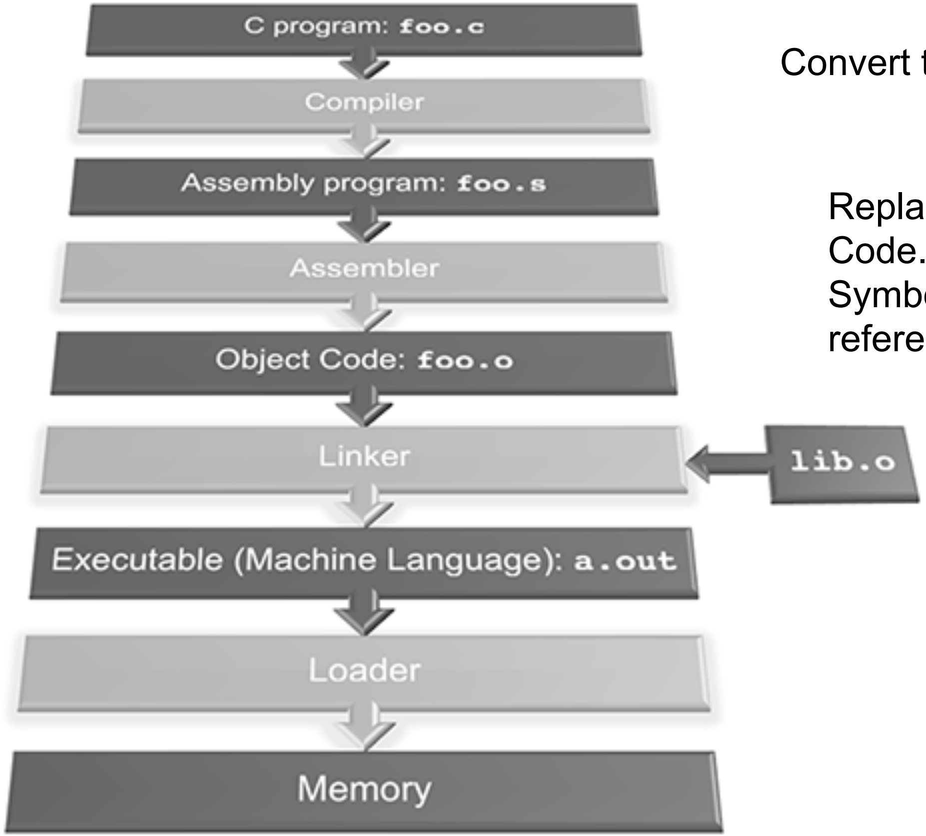
RISC - Reduced Instruction Set Computing

- cheaper hardware, faster computers

Stored Program Concept

- Instructions are Data!

# CALL Review




Convert to Assembly Code (MAL)

Replace Pseudoinstructions. Create Machine Code. Replace labels with immediates with Symbol table. requires 2 passes (forward referencing). Creates Relocation Table

Combines several object files.
Updates addresses using Relocation Table
Creates Executable Code

Loads to memory and runs it!

# MT Practice

Suppose the assembler knew the file line numbers of all labels before it began its first pass over a file, and that every line in the file contains an instruction. Then the assembler would need ____ pass(es) to translate a MAL file, and ____ pass(es) to translate a TAL file. These numbers differ because of _______________________________ (write n/a if they don't differ).

# MT Practice

Suppose the assembler knew the file line numbers of all labels before it began its first pass over a file, and that every line in the file contains an instruction. Then the assembler would need _2__ pass(es) to translate a MAL file, and __1_ pass(es) to translate a TAL file. These numbers differ because of ______pseudoinstructions______ (write n/a if they don't differ).