HKN CS61C Midterm Review

Zayd Enam, Sung-Roa Yoon, Allen Xiao

Hello!

This review session isn't formally linked (haha) to the CS61C class -- we don't know what's on your exam and this is our best guess based on your lecture topics and past material. If you're ever in doubt, go with what your instructors say.

That said, we hope you find our review pointers (haha) helpful!

Also, please fill out a feedback form on your way out.

We will cover:

- C & Pointers and Stuff
- MIPS stuff
- Number Representation
- Memory Hierarchy
- Direct Mapped Cache and AMAT
- Other?

We are *not* going to cover a lot of other concepts you still probably need to know:

Data/Request Level Parallelism, Mapreduce, Compilation/Assembly/Linking, RISC, Moore's law, computer components, write-through/write-back policy, local/global miss rate, cache blocking, types of cache misses (compulsory, capacity, conflict), Flynn Taxonomy (types of parallelism), strong and weak scaling, loop unrolling

C: Pointers Warm-up

```
int a = 9001;
int* c = &a;
int** e = &c;
printf("%p", *e);
```

Option A: Will compile, with output 9001.

Option B: Will compile, with output address of a.

Option C: Will compile, with some output.

Option D: Does not compile

C: Pointers Warm-up

```
int a = 9001;
int* c = &a;
int** e = &c;
printf("%p", *e);
```

Option A: Will compile, with output 9001.

Option B: Will compile, with output address of a.

Option C: Will compile, with some output.

Option D: Does not compile

C: Pointers Warm-up

```
int a = 9001;
int* c = &a;
int** e = &c;
printf("%d", **e);
```

Option A: Will compile, with output 9001.

Option B: Will compile, with output address of a.

Option C: Will compile, with some output.

Option D: Does not compile

C: Pointers Warm-up

```
int a = 9001;
int* c = &a;
int** e = &c;
printf("%d", **e);
```

Option A: Will compile, with output 9001.

Option B: Will compile, with output address of a.

Option C: Will compile, with some output.

Option D: Does not compile

C

```
void bar(int *b) {
   *b = 2; }
void foo() {
   int a = 3;
   bar(a);  }
```

Option A: 2

Option B: Compiles but gives an error

Option C: Doesn't compile

Option D: 3

C

```
void bar(int *b) {
   *b = 2; }
void foo() {
   int a = 3;
   bar(a);  }
```

Option A: 2

Option B: Compiles but gives an error

Option C: Doesn't compile

Option D: 3

C

```
void bar(int *b) {
   *b = 2; }
void foo() {
   int a = 3;
   bar(&a);  }
```

Option A: 2

Option B: Compiles but gives an error

Option C: Doesn't compile

Option D: 3

C

```
void bar(int *b) {
   *b = 2; }
void foo() {
   int a = 3;
   bar(&a);  }
```

Option A: **2**

Option B: **Compiles but gives an error**

Option C: **Doesn't compile**

Option D: **3**

C

```
void bar(int *b) {
   *b = 2;
}
int* foo() {
   int a = 3;
   bar(&a);
   return &a;
}
void main()   {
   int *out = foo();
   otherfunc();
   printf("%d", *out);
}
```

A: 2

B: Compiles but may give an error

C: Doesn't compile

D: 3

C

```
void bar(int *b) {
   *b = 2;
}
int* foo() {
   int a = 3;
   bar(&a);
   return &a;
}
void main()   {
   int *out = foo();
   otherfunc();
   printf("%d", *out);
}
```

A: 2

B: Compiles but may give an error

C: Doesn't compile

D: 3

C: Arrays

You can initialize arrays in these ways:

```
int arr[] = {1, 2, 3, 4}; // Initializes an array with these data
int arr[5]; // Initialize a pointer with the data in the stack
```

You can cast arrays as pointers, as the arr variable is actually just a pointer to the first element.

```
int* arr1 = arr;
```

When you increment the pointer, the pointer will increment by the size of the data type specified.

```
*(arr1+1) == arr[1]; // True
```

C: Strings

Strings are basically char arrays that always ends in '\0' (null).

```
char * hello = "hello world";
hello[1] == 'e'; // True
hello[11] == '\0'; // True
char you[4]; // Any char array that ends in
you[0] = 'y'; // '/0' qualifies as a string.
you[1] = 'o';
you[2] = 'u';
you[3] = '\0';
```

C

What will C print? Assume the following and that the size of int is 4 bytes.

```
int arr[9]; // address 0x10000000
printf("%p\n",arr);
printf("%p\n",arr+1);
printf("%p\n",&arr[0]);
printf("%p\n",&arr[0]+1);
```

C

What will C print? Assume the following and that the size of int is 4 bytes.

```
int arr[9]; // address 0x10000000
printf("%p",arr); // 0x10000000
printf("%p",arr+1); // 0x10000004
printf("%p",&arr[0]); // 0x10000000
printf("%p",&arr[0]+1);// 0x10000004
```

MIPS - Overview

There are three types of instruction in MIPS

J-type: opcode - jump address

6 bits - 26 bits

Opcode = 2 or 3 (j or jal)

Jump to the address {PC[31:28], jump address, 00}

I-type: opcode - rs - rt - immediate

6 bits - 5b - 5b - 16 bits

Opcode != 2, 3, 0

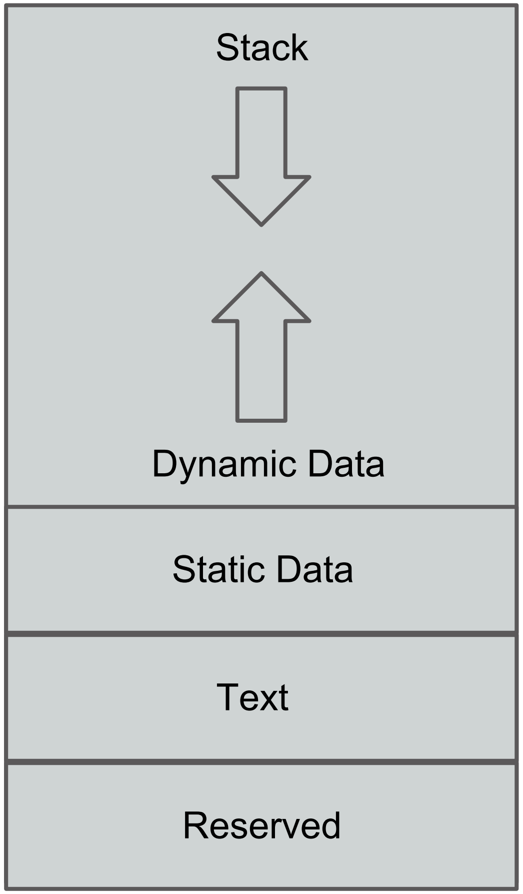
Function depends on the value of opcode itself

R-type: opcode - rs - rt - rd - shamt - funct

6 bits - 5b - 5b - 5b - 5bits - 6 bits

Opcode = 0

Function depends on the value of the funct field.



MIPS - Conventions

- \$a0~\$a3 are input values for functions.
- \$v0~\$v1 are output values for functions.
- \$s0~\$s7, \$ra, and \$sp need to be preserved across a call.
    - That means if a function uses any of these variables, they MUST BE PRESERVED AND RETURNED AT THE END OF THE FUNCTION CALL.
    - Usually preserved by being stored into stack and returned at the end of the function call.
- Store word means store to memory!
- Load word means load to register!

MIPS - Deciphering Assembly

```
mystery:
   lw $t0 0($a0)
   addi $a0 $a0 4
   beq $t0 $0 L2
   andi $t1 $t0 1
   bne $t1 $0 L1
   sw $t0 0($a1)
   addi $a1 $a1 4
L1:  j mystery
L2:  jr $ra
```

MIPS - Deciphering Assembly

```
mystery:
   lw $t0 0($a0) # Load the value in $a0 to $t0
   addi $a0 $a0 4 # Increment $a0 by 4
   beq $t0 $0 L2 # If $t0 == 0, go to L2
   andi $t1 $t0 1 # Set $t1 to be $t0 & 1 (mod 2)
   bne $t1 $0 L1 # If $t1 != 0, go to L1
   sw $t0 0($a1) # Store $t0 in $a1
   addi $a1 $a1 4 # Increment $a1 by 4
L1:  j mystery # Jump back into mystery
L2:  jr $ra # Return to the original caller
```

MIPS - Deciphering Assembly

```
mystery:
    lw $t0 0($a0) # Load the value in $a0 to $t0
    addi $a0 $a0 4 # Increment $a0 by 4
    beq $t0 $0 L2 # If $t0 == 0, go to L2
    andi $t1 $t0 1 # Set $t1 to be $t0 & 1 (mod 2)
    bne $t1 $0 L1 # If $t1 != 0, go to L1
    sw $t0 0($a1) # Store $t0 in $a1
    addi $a1 $a1 4 # Increment $a1 by 4
L1:  j mystery # Jump back into mystery
L2:  jr $ra # Return to the original caller
```

It copies first list's even value elements to the second list!

Number Representation

0b10110100

What is this number in decimal if we're using:

unsigned?
sign and magnitude?
bias? (-127)
one's complement?
two's complement?

Number Representation

0b10110100

What is this number in decimal if we're using:

- unsigned? 180
- sign and magnitude? -52
- bias? (-127) 53
- one's complement? -75
- two's complement? -76

Number Representation

0xfd0973e1

= 0b 1111 1101 0000 1001 0111 0011 1110 0001

What is this number in decimal if we're using IEEE754 floating point? (1S, 8E, 23F)

You don't have to carry out all the calculations; at least separate out the bits and give the main components in base 2.

Number Representation

0xfd0973e1

= 0b 1111 1101 0000 1001 0111 0011 1110 0001

What is this number in decimal if we're using IEEE754 floating point? (1S, 8E, 23F)

- sign: 1
- exponent: 250 - 127 = 123
- mantissa: $1.00010010111001111100001_2$
- $(-1)^1 * 2^{123} * 1.00010010111001111100001_2$

Number Representation

0xfd0973e1

= 0b 1111 1101 0000 1001 0111 0011 1110 0001

What is this number in decimal if we're using IEEE754 floating point? (1S, 8E, 23F)

sign: 1

exponent: 250 - 127 = 123

mantissa: $1.00010010111001111100001_2$

$(-1)^1 * 2^{123} * 1.00010010111001111100001_2$

$= -1.14191 * 10^{37}$

Number Representation

Unsigned:

- start at 0, counts up
- represents $[0, 2^n - 1]$, for n bits

Sign and magnitude:

- the first bit is the sign (1 means negative, 0 positive)
- take the rest of the bits as an unsigned number
- represents $[-(2^{(n-1)}-1), +(2^{(n-1)}-1)]$ (double 0!)
- jumps from biggest positive number to biggest negative number

Bias:

- take the unsigned representation, add the bias
- for a bias of $-(2^{(n-1)}-1)$: represents $[-(2^{(n-1)}-1), 2^{(n-1)}]$

Number Representation

1's Complement:

- If the first bit is 1, it is a negative number, 0 positive
- if it is positive, read as an unsigned number
- if it is negative, invert the bits, read the unsigned result, and negate it
- represents $[-(2^{(n-1)}-1), +(2^{(n-1)}-1)]$ (double 0!)

2's Complement:

- If the first bit is: 1 negative, 0 positive
- If positive, read as an unsigned number
- If negative, invert the bits, read the unsigned result, *add 1*, and negate
- represents $[-2^{(n-1)}, 2^{(n-1)}-1]$

Number Representation

IEEE Floating Point:

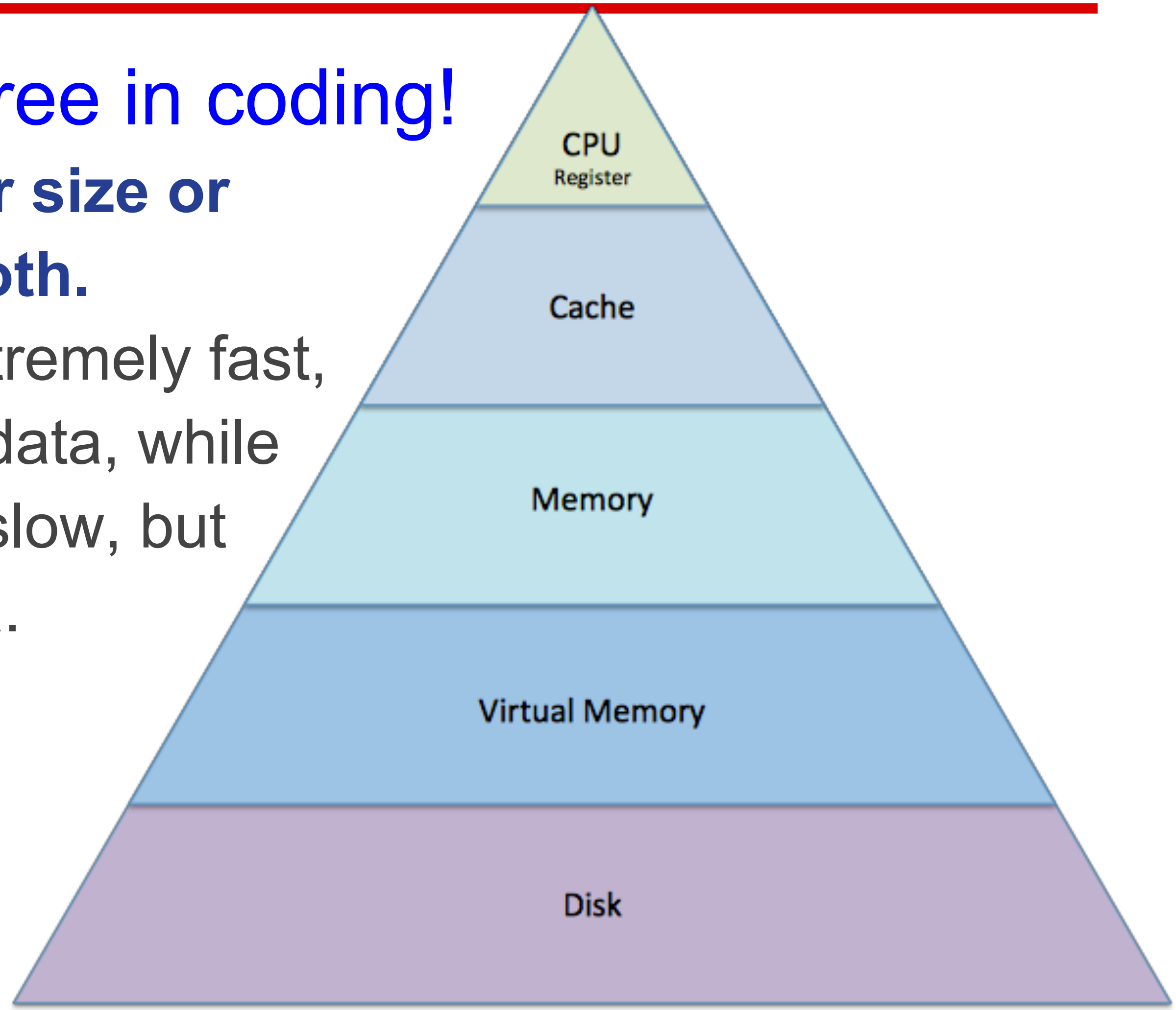
- in general, divided into:
    - 1 sign bit
    - a number of exponent bits (in biased form)
    - a number of fractional (mantissa) bits
- regular numbers represented as $(-1)^{sign} * 2^{exponent} * 1.mantissa$
- denorms (exponent bits are all 0) to represent really small numbers
- special symbols (infinities, NaNs) (exponent bits are all 1)

Memory Hierarchy

Nothing is ever free in coding!

You can have either size or speed, but never both.

Ex: Registers are extremely fast, but can't hold much data, while disks are extremely slow, but can hold a lot of data.



Direct Mapped Caches

```
AddVectors(uint8_t *A, uint8_t *B, uint8_t *C, int n) {
   for (int i = 0; i < n; i++)
      C[i] = A[i] + B[i]; }
```

```
sizeof(uint8_t) = 1
32 bits MIPS
4 KB Cache
10 Offset bits
n power of 2 much greater cache size
block aligned
```

If the cache is direct mapped, what is the lowest and highest hit:miss ratio?

Quick Review of Caches

Tag : Index : Offset

Cache Formulas. Fun! Fun. Fun?

number of offset bits = $\log_2$(block size)

number of index bits = $\log_2$(number of blocks)

cache size = $2^{\text{offset bits}} \times 2^{\text{index bits}}$

number of blocks = cache size ÷ block size

tag bits = total bits - offset bits - index bits

row bits = tag bits + data bits + dirty bit + valid bit

Direct Mapped Caches

```
AddVectors(uint8_t *A, uint8_t *B, uint8_t *C, int n) {
    for (int i = 0; i < n; i++)
        C[i] = A[i] + B[i]; }
```

```
sizeof(uint8_t) = 1
32 bits MIPS
4 KB Cache
10 Offset bits
n power of 2 much greater than cache size
block aligned
```

If the cache is direct mapped, what is the lowest and highest hit:miss ratio? 0:3*n 1023:1

AMAT

A program runs on single data cache and a single instruction cache where

- 20% of instructions are loads or stores
- Data cache hit rate is **95%** & instruction cache hit rates is **99.9%**
- Both caches: miss penalty is **100 cycles** & hit time is **1 cycle**

a. How many memory references are there per executed instruction on average?

b. How many data cache misses are there per instruction?

c. How many instruction cache misses are there per instruction?

d. If there were no misses the CPI would be 1. What is the CPI actually?

e. Calculate the AMAT of the program.

AMAT

A program runs on single data cache and a single instruction cache where

- 20% of instructions are loads or stores
- Data cache hit rate is **95%** & instruction cache hit rates is **99.9%**
- Both caches: miss penalty is **100 cycles** & hit time is **1 cycle**

a. How many memory references are there per executed instruction on average? 1 + 0.2 = 1.2

b. How many data cache misses are there per instruction? 0.2 * (1 - .95) = 0.01

c. How many instruction cache misses are there per instruction?

1 * (1 - .999) = 0.001

d. If there were no misses the CPI would be 1. What is the CPI actually?

$CPI = CPI_{ideal} + \text{Penalty} \times (\text{stall time per instruction}) = 1 + 100*(.01 + .001) = 2.1$

e. Calculate the AMAT of the program.

$= 1 + P(data_{access}) \times (P(data_{miss}) \times \text{Penalty}) + P(inst_{access}) \times (P(inst_{miss}) \times \text{Penalty}$

$= 1 + (1/6)(.05*100) + (5/6)(.001*100) = 1.916$

Some Other Equations

PUE:

Power Usage Effectiveness =

(total building power) / (IT equipment power)

Amdahl's Law:

Maximum speedup from parallelism =

$(1 - P + (P / N))^{-1}$

where:

P = proportion of program parallelizable

N = number of cores

Questions?

Please fill out a feedback form as well.

Thank you!

http://tiny.cc/hknfa12cs61c