HKN CS61C Midterm Review

Spring 2014

Sagar, Sung Roa, Vikram

Hello!

Although the group of presenters today contains course staff, the standard HKN review session disclaimer applies: This review session is not officially endorsed by the course.

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: General / Pointers
- MIPS
- Number Representation
- Memory Hierarchy
- Caches and AMAT

C: Pointers Warm-up

```
int five = 5;
int* pointer = &five;
```

What are each of the following?

```
pointer = ?
```

C: Pointers Warm-up

```
int five = 5;
int* pointer = &five;
```

What are each of the following?

```
pointer = The address of five
```

```
(the address pointer points to)
```

C: Pointers Warm-up

```
int five = 5;
int* pointer = &five;
```

What are each of the following?

```
*pointer = ?
```

C: Pointers Warm-up

```
int five = 5;
int* pointer = &five;
```

What are each of the following?

```
*pointer = 5
```

```
(the value stored at the address
stored in the pointer)
```

C: Pointers Warm-up

```
int five = 5;
int* pointer = &five;
```

What are each of the following?

```
&pointer = ?
```

C: Pointers Warm-up

```
int five = 5;
int* pointer = &five;
```

What are each of the following?

`&pointer` = The address of the pointer

(where it stores the address of five)

C: Pointers Warm-up

```
int w = 9001;    // assume this is validly
int* a = &w;     // placed in a function
int** t = &a;
return *t;
```

Option A: Will compile, with output 9001.

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

Option C: Will compile, with some output.

Option D: Does not compile

C: Pointers Warm-up

```
int w = 9001;
int* a = &w;
int** t = &a;
return *t;
```

Option A: Will compile, with output 9001.

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

Option C: Will compile, with some output.

Option D: Does not compile

C: Pointers Warm-up

```
int w = 9001;
int* a = &w;
int** t = &a;
return **t;
```

Option A: Will compile, with output 9001.

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

Option C: Will compile, with some output.

Option D: Does not compile

C: Pointers Warm-up

```
int w = 9001;
int* a = &w;
int** t = &a;
return **t;
```

Option A: Will compile, with output 9001.

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

Option C: Will compile, with some output.

Option D: Does not compile

C

```
void bar(int *lol) {
    *lol = 2;
}
int foo() {
    int wat = 3;
    bar(wat);
    return wat;
}

int main()    {
    int out = foo();
    printf("%d", out);
    return 0;
}
```

Option A: **2**

Option B: **Compiles but may give an error**

Option C: **Doesn't compile**

Option D: **3**

C

```
void bar(int *lol) {
    *lol = 2;
}
int foo() {
    int wat = 3;
    bar(wat);
    return wat;
}

int main()    {
    int out = foo();
    printf("%d", out);
    return 0;
}
```

Option A: 2

Option B: Compiles but may give an error

Option C: Doesn't compile

Option D: 3

C

```
void bar(int *lol) {
    *lol = 2;
}
int foo() {
    int wat = 3;
    bar(&wat);
    return wat;
}

int main()    {
    int out = foo();
    printf("%d", out);
    return 0;
}
```

Option A: 2

Option B: Compiles but may give an error

Option C: Doesn't compile

Option D: 3

C

```
void bar(int *lol) {
    *lol = 2;
}
int foo() {
    int wat = 3;
    bar(&wat);
    return wat;
}

int main()    {
    int out = foo();
    printf("%d", out);
    return 0;
}
```

Option A: 2

Option B: Compiles but may give 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);
}
```

Option A: 2

Option B: Compiles but may give 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);
}
```

Option A: 2

Option B: Compiles but may give an error

Option C: Doesn't compile

Option 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);
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\n",&arr); // 0x10000000
printf("%p",(&arr[0])+1);//
0x10000004
```

When you increment the pointer, the pointer will increment by the size of the data type specified, in this case: 4 byte int

C: Putting it all together

```
int main() {
  unsigned int data[] = {0x01234567, 0x89ABCDEF};
  unsigned char* x = (char *) data;
  printf("%x \n", (int) *(x));
  printf("%x \n", (int) *(x + 4));
}
```

What will this print?

You can assume ints are 32 bits, chars are 8 bits, and this is stored in big-endian format

(hint: what will each cast do, and what does "%x" mean for printing?)

C: Putting it all together

```
int main() {
  unsigned int data[] = {0x01234567, 0x89ABCDEF};
  unsigned char* x = (char *) data;
  printf("Problem 1: %x \n", (int) *(x));
  printf("Problem 2: %x \n", (int) *(x + 4));
}

// remember, assume this is big endian
```

```
Problem 1: 01
```

C: Putting it all together

```
int main() {
  unsigned int data[] = {0x01234567, 0x89ABCDEF};
  unsigned char* x = (char *) data;
  printf("Problem 1: %x \n", (int) *(x));
  printf("Problem 2: %x \n", (int) *(x + 4));
}

// remember, assume this is big endian
```

```
Problem 1: 01

Problem 2: 89
```

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		

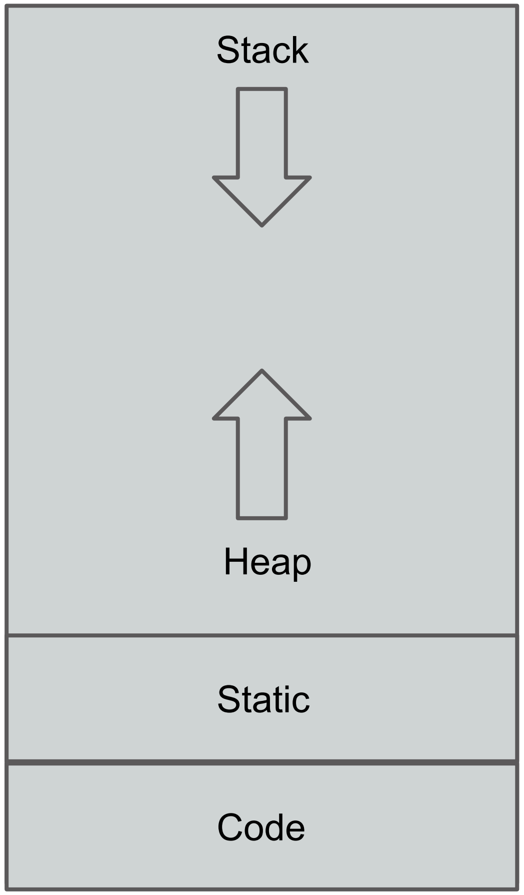
Opcode != 2, 3, 0

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

- $v0-$v1 are return values for functions.
- Callee's responsibility:
  - $s0-$s7: Backup/Restore if you change
  - $sp: Restore if you change (-Framesize at the top of your function, then +Framesize at the bottom)
- Caller's responsibility:
  - $ra: this will be modified by jal, back it up!
  - $a0-$a3: need to save if they will be used after the call
  - $t0-$t9: need to save if they will be used after the call (your last resort, only if you run out of $sx registers)
- Store word means store to memory!
- Load word means load into register!

Unsigned?

Which of the following are true for the `addiu` instruction? (Select all that apply)

A) Performs a different operation at the hardware level than `addi` (excluding overflow reporting)

B) Instructs the hardware not to report an overflow

C) Sign extends the immediate or value pulled from memory

Unsigned?

Which of the following are true for the `addiu` instruction? (Select all that apply)

~~A) Performs a different operation at the hardware level than `addi` (excluding overflow reporting)~~

B) Instructs the hardware not to report an overflow

C) Sign extends the immediate or value pulled from memory

Unsigned?

Which of the following are true for the `lhu` instruction? (Select all that apply)

A) Instructs the hardware not to report an overflow

B) Zero extends value pulled from memory

C) Zero extends the immediate

D) `lhu` is not a real instruction

Unsigned?

Which of the following are true for the `lhu` instruction? (Select all that apply)

~~A) Instructs the hardware not to report an overflow~~

B) Zero extends value pulled from memory

~~C) Zero extends the immediate~~

~~C) `lhu` is not a real instruction~~

Unsigned?

Which of the following are true for the `sltiu` instruction? (Select all that apply)

A) Performs a different operation at the hardware level than `slti` (excluding overflow reporting)

B) Instructs the hardware not to report an overflow

C) Sign extends the immediate or value pulled from memory

Unsigned?

Which of the following are true for the `sltiu` instruction? (Select all that apply)

A) Performs a different operation at the hardware level than `slti` (excluding overflow reporting)

~~B) Instructs the hardware not to report an overflow~~

C) Sign extends the immediate or value pulled from memory

MIPS - Deciphering Assembly

```
# assume $a0 contains address of arr in mem
mystery: # assume $a1 points to empty arr
   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
```

For convenience, assume that all types of arrays are “null-terminated.” Note that this is NOT generally the case.

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 valued elements to the second list!

Convert from C to MIPS

```
int fib(int n){
  if (n == 0){
    return 0;
  } else if (n == 1) {
    return 1;
  }
  return fib(n-1) + fib(n-2)
}
```

Note: Your MIPS version MUST be recursive

Convert from C to MIPS

Solution:

https://gist.github.com/anonymous/5074142#file-gistfile1-txt

Number Representation

Unsigned:

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

Bias:

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

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

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 $[-2^{(n-1)}, 2^{(n-1)}-1]$

Remember, you want the simplest operations in your computer to be as simple as possible. That means that you want addition to just be adding the bits together one by one! That is why unsigned is unsigned, and signed is 2's complement in C (and other languages).

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? $2^7+2^5+2^4+2^2 = 180$
- sign and magnitude? $(-1)^1 \times (2^5+2^4+2^2) = -52$
- bias? (-127) $2^7+2^5+2^4+2^2-127 = 53$
- one's complement? $-1 \times 0b01001011 = -1 \times (2^6+2^3+2^1+2^0) = -75$
- two's complement? $-1 \times (0b01001011+1) = -1 \times (0b01001100)$
  $= -1 \times (2^6+2^3+2^2) = -76$

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 $(-1)^{\text{sign}} * 2^{\text{exponent}} * 1.\text{mantissa}$
- You can think of regular form as Scientific Notation in Binary!
- denorms (exponent bits are all 0) to represent really small numbers
- special symbols (infinities, NaNs) (exponent bits are all 1)

Number Representation

37.25

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

Number Representation

37.25

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

=0b100101.01

=0b1.0010101 * 2^5

$=(-1)^{0} * 2^{5+127-127} * (1+.0010101)$

$=(-1)^{0} * 2^{132-127} * (1+.0010101)$

Sign = 0, Exponent = 132 = 0b10000100,

Mantissa = 0b0010101(16 0's)

Floating bits = 0b0 10000100 00101010000000000000000

Floating bits = 0b0100 0010 0001 0101 0000 0000 0000 0000

Floating bits in hex = 0x42150000

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

Among all of the number representations so far, pick out the one(s) with the highest variety of numbers and the one(s) with the lowest variety of numbers.

A. Unsigned
B. Biased
C. Sign and Magnitude
D. One's Complement
E. Two's Complement
F. Floating Point

Number Representation

Among all of the number representations so far, pick out the one(s) with the highest variety of numbers and the one(s) with the lowest variety of numbers. Highest

A. Unsigned
B. Biased
C. Sign and Magnitude
D. One's Complement
E. Two's Complement
F. Floating Point

Number Representation

Among all of the number representations so far, pick out the one(s) with the highest variety of numbers and the one(s) with the lowest variety of numbers. Lowest

A. Unsigned
B. Biased
C. Sign and Magnitude
D. One's Complement
E. Two's Complement
F. Floating Point

Number Representation

Which number representations take the most effort to switch between different bit sizes?

(Ex: from 16 bits to 32 bits, 4 bits to 8 bits, etc)

A. Unsigned

B. Biased

C. Sign and Magnitude

D. One's Complement

E. Two's Complement

F. Floating Point

Number Representation

Which number representations take the most effort to switch between different bit sizes?

(Ex: from 16 bits to 32 bits, 4 bits to 8 bits, etc)

A. Unsigned
B. Biased
C. Sign and Magnitude
D. One's Complement
E. Two's Complement
F. Floating Point

Number Representation

Rank the number representations by how easy they are to try doing number comparisons with. (1 = easiest)

A. Unsigned
B. Biased
C. Sign and Magnitude
D. One's Complement
E. Two's Complement
F. Floating Point

Number Representation

Rank the number representations by how easy they are to try doing number comparisons with. (1 = easiest)

A. Unsigned 1
B. Biased 1
C. Sign and Magnitude
D. One's Complement
E. Two's Complement
F. Floating Point

Number Representation

Rank the number representations by how easy they are to try doing number comparisons with.
(1 = easiest)

A. Unsigned 1
B. Biased 1
C. Sign and Magnitude
D. One's Complement
E. Two's Complement 2
F. Floating Point

Number Representation

Rank the number representations by how easy they are to try doing number comparisons with. (1 = easiest)

A. Unsigned 1
B. Biased 1
C. Sign and Magnitude
D. One's Complement 3
E. Two's Complement 2
F. Floating Point

Number Representation

Rank the number representations by how easy they are to try doing number comparisons with.
(1 = easiest)

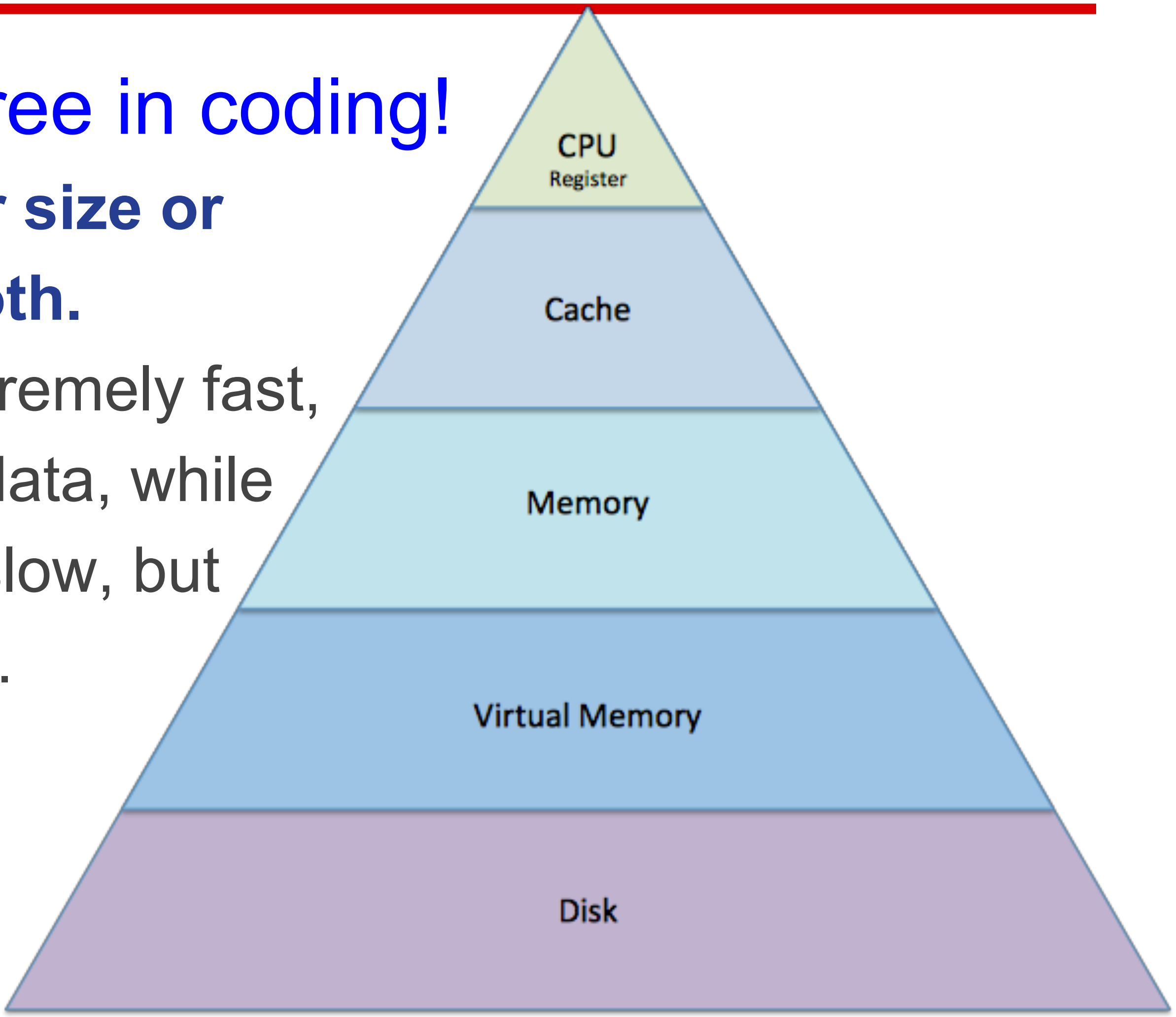
A. Unsigned 1
B. Biased 1
C. Sign and Magnitude 4
D. One's Complement 3
E. Two's Complement 2
F. Floating Point 5

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.



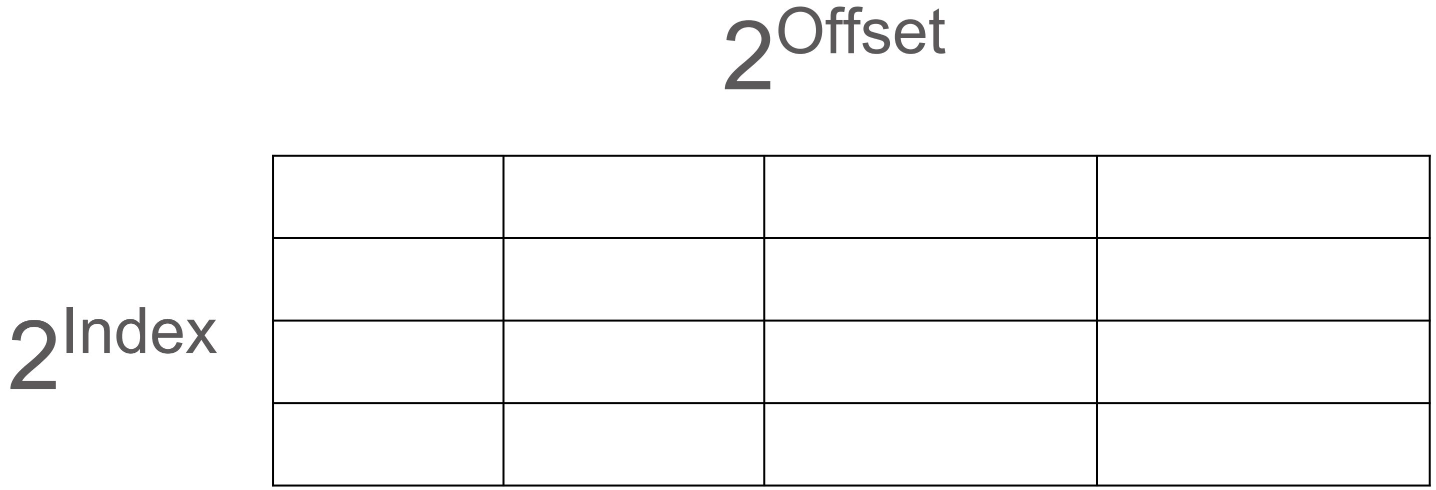
Direct Mapped Caches

```
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 Size
10 Offset bits
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 Size
10 Offset bits
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?

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

e. Calculate the AMAT of the program.

$= 1 + P(\text{data}_{\text{access}}) \times (P(\text{data}_{\text{miss}}) \times \text{Penalty}) + P(\text{inst}_{\text{access}}) \times (P(\text{inst}_{\text{miss}}) \times \text{Penalty}$

$= 1 + (1/5)(.05*100) + (5/5)(.001*100) = 2.1$

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! Good Luck on your exam! ☺