Midterm Review

CS61C Summer 2014

Number Representations

- Given the following numbers and the representations they should be interpreted as, order these from least to greatest:
    1. 0xC0700000 in floating point
    2. 0xFFFFFFFC in two's complement
    3. 0xFF800000 in floating point
- What is the smallest positive integer 32-bit IEEE floating point cannot represent?
- Give the base-10 result of adding these 4-bit binary numerals in two's complement: 1010 + 1110

Number Representations

- Given the following numbers and the representations they should be interpreted as, order these from least to greatest: 3 < 2 < 1

    1. 0xC0700000 in floating point = -3.75
    2. 0xFFFFFFFC in two's complement = -4
    3. 0xFF800000 in floating point = -INF

- What is the smallest positive integer 32-bit IEEE floating point cannot represent? 2^24 + 1; 23 significand bits +1 implicit leading 1 bit, so the first bit pattern unrepresentable is 25 bits: 0x1000001 = 2^24 + 1

- Give the base-10 result of adding these 4-bit binary numerals in two's complement: 1010 + 1110 = -8; preform unsigned addition and keep the lower 4 bits

Number Representations

- In our 32-bit single precision floating point representation, we decide to convert one significand bit to an exponent bit. How many denormalized numbers do we have, relative to before?
- Rounded to the nearest power of 2, how many denorm numbers are there in our new format?
- Assume that the most significant bit of x is a 0. We store the result of flipping x's bits into y. Interpreted in the following number representations, how large is the magnitude of y, relative to the magnitude of x?
    1. Unsigned
    2. One's Complement
    3. Two's Complement
    4. Sign and Magnitude
    5. Biased Notation

Number Representations

- In our 32-bit single precision floating point representation, we decide to convert one significand bit to an exponent bit. How many denormalized numbers do we have, relative to before? Half as many, because we lost 1 significand bit; exponent bits are fixed for denorms

- Rounded to the nearest power of 2, how many denorm numbers are there in our new format? 2^23 - 2; there are now 22 significand bits and 1 sign bit; we do not want to count the 2 zero representations though

- Assume that the most significant bit of x is a 0. We store the result of flipping x's bits into y. Interpreted in the following number representations, how large is the magnitude of y, relative to the magnitude of x?

   1. Unsigned | y | > | x |; y will have a 1 in the most significant bit
   2. One's Complement | y | = | x |; the flip simply negates one's complement
   3. Two's Complement | y | > | x |; flipping all the bits without adding 1 is negation -1
   4. Sign and Magnitude ?; cannot tell what the magnitude was
   5. Biased Notation | y | > | x |; ordered the same as unsigned, with constant bias

Number Representations

- We have selected a 16-bit microprocessor that does not have a floating-point unit, so there is no native support for floating point operations. We are going to implement floating point operations in software in C, with a 5-bit exponent field and no denorms. Define a new type called "fp", and fill in the missing code:

```
typedef ________ fp;

fp negateFP(fp num) {

        return _________;

}
```

Number Representations

- We have selected a 16-bit microprocessor that does not have a floating-point unit, so there is no native support for floating point operations. We are going to implement floating point operations in software in C, with a 5-bit exponent field and no denorms. Define a new type called "fp", and fill in the missing code:

```
typedef int fp;

fp negateFP(fp num) {

    return num ^ 0x8000;

}
```

Number Representations

- Now, we want to access the unbiased signed value of the 5-bit exponent field.

```
int getExponent(fp num) {

        return ______________________;

}
```

Number Representations

- Now, we want to access the unbiased signed value of the 5-bit exponent field.

```
int getExponent(fp num) {
    return ((num & 0x7C00) >> 10) - 15;
}
```

Number Representations

- Finally, we want a simple multiplication method to multiply one of these fp values by 2^n, while also detecting overflow or underflow:

```
fp multPow2(fp num, int n) {
        int exp = getExp(num) + _______;
        if(_________) exit(1); // overflow
        if(_________) exit(-1); // underflow
        num = ___________; // zero out the old exponent
        return _______________; // fill in the new exponent and return
}
```

Number Representations

- Finally, we want a simple multiplication method to multiply one of these fp values by 2^n, while also detecting overflow or underflow:

```
fp multPow2(fp num, int n) {
    int exp = getExp(num) + n;
    if(exp>15) exit(1); // overflow
    if(exp<-15) exit(-1); // underflow
    num = num & 0x83FF; // zero out the old exponent
    return num | ((exp + 15)<<10); // fill in the new exponent
}
```

You already have the exp. of # by getExp() funct. ex. 3.25, exp. is 011 so adding n(ex. 2) n = 010 is 5(0101)

C

- What is the value of s after the following code?

```
unsigned int32_t t[] = {0,1,2,3,4,5,6,7};

unsigned int s = sizeof(t);
```

- What does the following code print?

```
char * s = "uncharacteristic";

printf("%s",s+s[7]-s[6]);
```

C

- What is the value of s after the following code?

  ```
  unsigned int32_t t[] = {0,1,2,3,4,5,6,7};
  unsigned int s = sizeof(t);
  ```

  32, the number of bytes in the t array

- What does the following code print?

  ```
  char * s = "uncharacteristic";
  printf("%s",s+s[7]-s[6]);
  ```

  "characteristic"; s[7] = 'c', s[6] = 'a', and 'c'-'a' = 2

C

- Describe what the following function does:

```
char * bizarre(char * f, char y) {
        char * h = f;
        while(*h!=y && *h)
            h++;
            if(*h) {
                *h = 0;
                h++;
            }
        return h;
}
```

C

- Describe what the following function does: splits string f at the first occurrence of y

I added this, && *h means while the value at that address is not null

So does placing the char '\0' split string?

```
char * bizarre(char * f, char y) {
    char * h = f;
    while(*h!=y && *h) // until h points a y value, or end of the string
        h++; // note that this is the only line actually in the loop body!
        if(*h) { // if not the end of the string
            *h = 0; // end the string at the first occurrence of y
            h++; // increment to the next char to return the 2nd string
        }
    return h; // returns the second string, or 0
}
```

C

- Complete this function to negate the char in byte i of the given int, passed by reference with a pointer:

```
void negByte(int * data, char i) {

    ___________________ = ______________________;

}
```

- The C type system internally keeps track of the sizes of its types. Implement an expression equivalent to sizeof, which takes a variable named "var":

```
#define sizeof(var)   (_________________________)
```

C

- Complete this function to negate the char in byte i of the given int, passed by reference with a pointer:

```
void negByte(int * data, char i) {

    *(((char *)data) + 3 - i) = - * (((char *)data) + 3 - i);

}
```

- The C type system internally keeps track of the sizes of its types. Implement an expression equivalent to sizeof, which takes a variable named "var":

```
#define sizeof(var)   ((char *)(&var + 1) - (char *)(&var))
```

C

- Implement the following rotate left function, where *p is rotated such that the leftmost bits fall off the left and appear on the right, making no assumptions about an int's size:

```
void rotl_(unsigned int * p, unsigned int n) {

    *p = ____________________________;

}
```

C

- Implement the following rotate left function, where *p is rotated such that the leftmost bits fall off the left and appear on the right, making no assumptions about an int's size:

```
void rotl_(unsigned int * p, unsigned int n) {
    *p = (*p << n) | (*p >> (sizeof(int)*8 - n));
}
```

Memory Management

- Suppose we have a program that allocates many things, and then wants to free everything it ever allocated. Assume allocation always succeeds. We have the version of malloc:

```
void * newMalloc(size_t size, // the size to alloc
                        void *** toFree, // an array of ptrs
                        int * toFreeSize) { // size of array
    *toFree = __________________;
    void *temp = _______________; // allocate
    _______________________; // update toFree
    _______________________; // update toFreeSize
    return temp;
}
```

Memory Management

- Suppose we have a program that allocates many things, and then wants to free everything it ever allocated. Assume allocation always succeeds. We have the version of malloc:

```
void * newMalloc(size_t size, // the size to alloc
                 void *** toFree, // an array of ptrs
                 int * toFreeSize) { // size of array
    *toFree = (void *)realloc(*toFree, (*toFreeSize+1) * sizeof(void *));
    void *temp = (void *)malloc(size); // allocate
    (*toFree)[*toFreeSize] = temp; // update toFree
    *toFreeSize = *toFreeSize + 1; // update toFreeSize
    return temp;
}
```

MIPS

- Write the following MIPS function to return non-0 if the input is not an infinity, and 0 if it is.

```
IsNotInfinity:   _______ $a0 $a0 1

                 __________________________

                 jr $ra
```

MIPS

- Write the following MIPS function to return non-0 if the input is not an infinity, and 0 if it is.

```
IsNotInfinity:   sll $a0 $a0 1 // remove the sign

                 xori $v0 $a0 0xFF000000

                 jr $ra
```

MIPS

- (T/F) rd is the only destination register
- (T/F) When used, rd is always a destination register
- (T/F) sltiu performs sign extension on its immediate
- (T/F) add detects any overflow that occurs

MIPS

- (T/F) rd is the only destination register I-Types use rt instead
- (T/F) When used, rd is always a destination register rd is never an argument
- (T/F) sltiu performs sign extension on its immediate the only I-types that zero-extend their immediate values are andi, ori, & xori
- (T/F) add detects any overflow that occurs add only detects signed overflow

MIPS

- Convert the following instructions into their numeric values (but not fully into hex):
  - j 0x92837 [op=2 | addr=0x92837]
  - jalr $v0
  - xori $t0, $0, 15
  - bne $ra, $0, -1
  - sllv $t2, $t1, $t0

MIPS

- Convert the following instructions into their numeric values (but not fully into hex):
  - j 0x92837 [op=2 | addr=0x92837]
  - jalr $v0 [op=0 | rs=2 | rt=0 | rd=31 | sh=0 | fn=0]
  - xori $t0, $0, 15 [op=14 | rs=0 | rt=8 | imm=15]
  - bne $ra, $0, -1 [op=5 | rs=31 | rt=0 | imm=-1]
  - sllv $t2, $t1, $t0 [op=0 | rs=8 | rt=9 | rd=10 | sh=0 | fn=04]

MIPS

- How would J-Type instructions be affected (in terms of reach) if we relaxed the requirement that instructions be placed on word boundaries, and instead required them to be placed on half-word boundaries?

- Give a minor tweak to the MIPS ISA to allow us to use true absolute addressing (maximal reach) for all J-Type instructions.

MIPS

- How would J-Type instructions be affected (in terms of 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 can jump would be cut in half.

- Give a minor tweak to the MIPS ISA to allow us to use true absolute addressing (maximal reach) for all J-Type instructions. Only allow jumps to addresses which are multiples of 2^6.

MIPS

- Suppose we were to modify the MIPS ISA so that it exposed 64 registers instead of 32, and adjusted R, I, and J formats to accommodate this, without changing the opcode or shamt fields. Registers and instructions stay at 4 bytes wide.
  - At most how many instructions can a single beq reach?
  - How many more addresses can a jal reach now?
  - How many different R-type instructions can there be?

MIPS

- Suppose we were to modify the MIPS ISA so that it exposed 64 registers instead of 32, and adjusted R, I, and J formats to accommodate this, without changing the opcode or shamt fields. Registers and instructions stay at 4 bytes wide.
  - At most how many instructions can a single beq reach? 2^14, 2 offset bits were lost to accommodate rs & rt
  - How many more addresses can a jal reach now? 0
  - How many different R-type instructions can there be? 8, rd, rs, & rt removed 3 bits from the funct field, leaving only 3

MIPS

- Now let's use just 16 32-bit registers. How many extra bits do we have to use in each of the other formats? R: ___ J: ___ I: ___
- For only R-Type instructions, which field should get any extra bits? opcode, shamt, or funct? Why?

MIPS

- Now let's use just 16 32-bit registers. How many extra bits do we have to use in each of the other formats? R: 3 J: 0 I: 2

- For only R-Type instructions, which field should get any extra bits? opcode, shamt, or funct? Why? funct; the opcode must be the same width for all types; the shamt already reaches shift amounts from 0 to 2^5-1 = 31

MIPS

- For I-type instructions, the extra bits would go to the immediate field like so:

  [opcode (6) | rs (4) | rt (4) | immediate (18)]

- What **fraction** of our address space can we now reach with a branch instruction?

- Assume our PC=0x08000000. What is the lowest address we can reach?

MIPS

[opcode (6) | rs (4) | rt (4) | immediate (18)]

- What **fraction** of our address space can we now reach with a branch instruction? 2^-12; address space is 2^32 B; branch offset gets 2^18 instructions = 2^20 B; 2^20 / 2^32 = 2^-12

- Assume our PC=0x08000000. What is the lowest address we can reach? 0x07F80004

new PC = PC + 4 + 4 * imm

most negative immediate is 0b10...0 = 0x20000; 4 * imm = 0x80000

sign extend 0x80000 = 0xFFF80000

new PC = 0x08000004 + 0xFFF80000 = 0x07F80004

C to MIPS

Convert the following C code into MIPS using the least amount of Instructions

C:

```
int helperPBST (int *arr, int pos, int elem) {
     int temp = pos – 1;
     if (elem == *(arr + temp)) {
         return pos;
     }
     else if (elem > *(arr + temp)) {
         return helperPBST(arr, pos*2+1, elem);
     }
 else {
     return helperPBST(arr, pos*2, elem);
     }
}
```

MIPS:

```
#$a0 is arr
#a1 is pos
#a2 is elem
pbst:
addi $t0, $a1, -1
sll ___________
add ___________
lw ___________
beq $a2, $t1, success
slt $t3, $a2, $t1
beq $t3, $0, right
___________
___________
___________

right:
___________
___________
___________
success:
___________
jr $ra
```

C to MIPS

Convert the following C code into MIPS using the least amount of Instructions

C:

```
int helperPBST (int *arr, int pos, int elem) {
    int temp = pos – 1;
    if (elem == *(arr + temp)) {
        return pos;
    }
    else if (elem > *(arr + temp)) {
        return helperPBST(arr, pos*2+1, elem);
    }
 else {
    return helperPBST(arr, pos*2, elem);
    }
}
```

MIPS:

```
#$a0 is arr
#a1 is pos
#a2 is elem
pbst:
addi $t0, $a1, -1
sll   $t0, $t0, 2
add  $t0, $t0, $a0
lw   $t1, 0($t0)
beq $a2, $t1, success
slt $t3, $a2, $t1
beq $t3, $0, right
    sll $a1, $a1, 1
    J   pbst

right:
   sll $a1, $a1, 1
   addiu $a1, $a1, 1
   J pbst
success:
   addiu $v0, $a1, 0
jr $ra
```

this beq is the else case, don't do anything to other par. since they stay the

C to MIPS

- Consider visit_in_order, then translate it into MIPS:

```
typedef struct node {
    int value;
    struct node * left;
    struct node * right;
} node;

void visit_in_order(node * root, void(* visit)(node *) {
    if (root) {
        visit_in_order(root->left, visit);
        visit(root);
        visit_in_order(root->right, visit);
    }
}
```

C to MIPS

```
vio:
    addiu $sp, $sp, -12
    sw $s0, 0($sp)
    sw $s1, 4($sp)
    sw $ra, 8($sp)

    #nullcheck
    beq $0, $a0, Exit
    move $s0, $a0
    move $s1, $a1

    #visit left
    lw $a0, 4($s0)
    jal vio
    #visit myself
    move $a0, $s0
    jalr $s1
    #visit right
    lw $a0, 8($s0)
    move $a1, $s1
    jal vio

Exit:
    lw $s0, 0($sp)
    lw $s1, 4($sp)
    lw $ra, 8($sp)
    addiu $sp, $sp, 12
    jr $ra
```

C to MIPS

The following is pseudo code for a recursive function that finds the total sum of the sizes of each element of a hailstone sequence (not important).

```
hailstone(int n)
    if (n == 1) , return n
    else if (n is even), return n + hailstone(n/2)
    else return n + hailstone(3*n + 1)
```

Using proper, standard MIPS convention, how many registers should we store to the stack if we were to implement this function with this exact algorithm in MIPS? Assume we use no **$s** registers.

C to MIPS

The following is pseudo code for a recursive function that finds the total sum of the sizes of each element of a hailstone sequence (not important).

```
hailstone(int n)
    if (n == 1) , return n
    else if (n is even), return n + hailstone(n/2)
    else return n + hailstone(3*n + 1)
```

Using proper, standard MIPS convention, how many registers should we store to the stack if we were to implement this function with this exact algorithm in MIPS? Assume we use no **$s** registers. **2 -> $ra, $t0**

CALL

- 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 ________ (do/do not) differ because of ________________.

CALL

- 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 do differ because of pseudo-instructions.

CALL

- How many passes through the code does the Assembler have to make? Why?
- What are the different parts of the object files output by the Assembler?

CALL

- How many passes through the code does the Assembler have to make? Why? Two, one to find all the label addresses and another to convert all instructions while resolving any forward references using the collected label address
- What are the different parts of the object files output by the Assembler?
  - Header: Size and position of other parts
  - Text: The machine code
  - Data: data in the source file (binary)
  - Relocation Table: Identifies lines of code that need to be "handled" by linker
  - Symbol Table: List of the file's labels and data that can be referenced
  - Debugging Information: Additional information for debuggers

CALL

- Which step in CALL resolves relative addressing? Absolute addressing?
- What step in CALL makes use of the $at register?
- What does RISC stand for? How does this relate to pseudo - instructions?

CALL

- Which step in CALL resolves relative addressing? Absolute addressing? Assembler, Linker
- What step in CALL makes use of the $at register? Assembler
- What does RISC stand for? How does this relate to pseudo - instructions? RISC -> Reduced Instruction Set Computing. Minimal set of instructions lead to many lines of code. Pseudo-instructions are more complex instructions to make assembly programming easier for the coder. These are converted to TAL by the assembler

Caches

Instruction Category	Cycles	Frequency
Memory Access	10	0.1
Arithmetic	2	0.4
Branch	3	0.2
Comparison	1	0.3

- What is the average CPI of the program described in the table?
- Which is better: halving memory access cycles or arithmetic cycles and why?

Caches

Instruction Category	**Cycles**	**Frequency**
Memory Access	10	0.1
Arithmetic	2	0.4
Branch	3	0.2
Comparison	1	0.3

- What is the average CPI of the program described in the table?

  CPI = 10 * 0.1 + 2 * 0.4 + 3 * 0.2 + 1 * 0.3 = 2.7

- Which is better: halving memory access cycles or arithmetic cycles and why?

  Half mem access: 5 * 0.1 + 2 * 0.4 + 3 * 0.2 + 1 * 0.3 = 2.2
  Half arithmetic: 10 * 0.1 + 1 * 0.4 + 3 * 0.2 + 1 * 0.3 = 2.3
  Halving memory access is better

Caches

- Consider a single 4 KiB cache with 512 B blocks and a write-back policy. Assume a 32-bit address space.
- If the cache were direct-mapped:
  - # of rows = ______
  - # of offset bits = ______
- If the cache were 4-way set associative:
  - # of tag bits = _____
  - # of index bits = _____
  - # of bits per row = _____

Caches

- Consider a single 4 KiB cache with 512 B blocks and a write-back policy. Assume a 32-bit address space.
- If the cache were direct-mapped:
  - # of rows = $2^{12}/2^{9} = 8$
  - # of offset bits = $\log_2(512) = 9$
- If the cache were 4-way set associative:
  - # of tag bits = $32 - 9 - 1 = 22$
  - # of index bits = $8/4 = 2$, $\log_2(2) = 1$
  - # of bits per row = $512 * 8 + 22 + 2 = 4120$

Caches

- Consider a single 4 KiB cache with 512 B blocks and a write-back policy. Assume a 32-bit address space.

```
typedef struct {
        …. // some unidentified number of other struct members
        int visited;
        int danger;
}
```

Here is a piece of code that counts the number of places we've visited. Assume this gets executed somewhere in the middle of our program, that count is held in a register, and the size of the array is greater than 4 KiB.

```
for(int I = 0; I < NUM_LOCS; i++)
        if(locs[i].visited) count++;
```

- What's the fewest possible number of bytes written to main memory?
- What' the greatest number of bytes written to main memory?

Caches

- Consider a single 4 KiB cache with 512 B blocks and a write-back policy. Assume a 32-bit address space.

```
typedef struct {
        …. // some unidentified number of other struct members
        int visited;
        int danger;
}
```

Here is a piece of code that counts the number of places we've visited. Assume this gets executed somewhere in the middle of our program, that count is held in a register, and the size of the array is greater than 4 KiB.

```
for(int I = 0; I < NUM_LOCS; i++)
            if(locs[i].visited) count++;
```

- What's the fewest possible number of bytes written to main memory? 0 B
- What' the greatest number of bytes written to main memory? 4 KiB

We are reading, not writing. What will be written back are the dirty bits already in the cache.

Caches

- Consider a single 4 KiB cache with 512 B blocks and a write-back policy. Assume a 32-bit address space. Consider the following code with NUM_LOCS > 2^{10}:

```
for(int I = 0; i < NUM_LOCKS; i++)
        if(visited[i] && danger[i] > 5) count++;
```

- Assume that the cache has no valid blocks initially. **You are told that in the worse case, the cache has a miss rate of 100%.** Consider each of the following possible of the following possible changes to the cache individually. Mark each as **E** if it eliminates the chances of this worst-case scenario miss rate, **R** if it reduces the chances, or **N** if it's not helpful.
- More sets, same block size, same associativity _____
- Double associativity, half block size, same total cache size _____
- Everything stays the same but use a write-through policy instead _____

Caches

- Consider a single 4 KiB cache with 512 B blocks and a write-back policy. Assume a 32-bit address space. Consider the following code with NUM_LOCS > 2^{10}:

```
for(int I = 0; i < NUM_LOCKS; i++)
        if(visited[i] && danger[i] > 5) count++;
```

- More sets, same block size, same associativity R
- Double associativity, half block size, same total cache size E
- Everything stays the same but use a write-through policy instead N
- The worst case miss rage is 100% for blocks that hold more than 1 piece of array data, so the cache must be direct-mapped. The worst case happens when the visited and the danger arrays start in blocks that map to the same row AND have the same offset.
    - With more sets/rows, we are increasing the size of the cache. If the cache size changes such that the addresses of visited[i] and danger[i] no longer map to the same row, then we no longer have the worst case scenario. This is not guaranteed to happen, so the chances are reduced.
    - Increasing associativity completely removes the ping-pong effect.
    - A write-through policy does not change the behavior of the cache at all.

Caches

- We have a 256 KiB Direct–Mapped Cache with 8 Byte Words and 4 Word Blocks and a 4 GiB Byte-Addressed Memory, but suppose we switch to a word-addressed memory. Give the T:I:O breakdown.

Caches

- We have a 256 KiB Direct–Mapped Cache with 8 Byte Words and 4 Word Blocks and a 4 GiB Byte-Addressed Memory, but suppose we switch to a word-addressed memory. Give the T:I:O breakdown. T:I:O = 14:13:2

- Have to label every WORD in memory so there are a total of $2^{32}/2^{3} = 2^{29}$ words, so we need 29 bits for the address. Each block is 4 words so we need 2 bits of offset. The number rows in the cache is $2^{18}/2^{5} = 2^{13}$ rows, so we need 13 bits of index. 29 - 13 - 2 = 14 gives us the tag.

Performance

- Calculate the AMAT for a machine with the following specs:
  - L1 takes 3 cycles; local miss rate of 25%
  - L2 hits take 10 cycles; local hit rate of 60%
  - L3 hits take 100 cycles; global miss rate of 9%
  - Main memory access take 1000 cycles and all data is available in memory

Performance

- L1 takes 3 cycles; local miss rate of 25%
- L2 hits take 10 cycles; local hit rate of 60%
- L3 hits take 100 cycles; global miss rate of 9%
- Main memory access take 1000 cycles

AMAT = 3 + 0.25 * (10 + 0.4 * (100)) + 0.09 * 1000 = 105.5 cycles

OR

L3 global miss rate = L3 local miss * L2 local miss * L1 local miss

L3 local * 0.4 * 0.25 = 0.09

L3 local miss rate = 0.9

AMAT = 3 + 0.25 * (10 + 0.4 * (100 + 0.9 * (1000))) = 105.5 cycles