1.2 Textbook readings

Design For Moore's Law

Moore's law stated that intefrated curcuit resources double every 18-24 months. Moore (a founder of intel), found that because chip designs take years the resources may double, or even triple because of the lengthy process of designing a processor. Architects therefore must anticipate where tecnology will be in future years in order to compinsate for this discrepency.

Using abstaction

Programmers and architects use abstraction in order to increase productivity. This is a way that architects get past the problem posed by moores law. Abstraction is used to represent a design at different levels of its representation.

The Common Case

The common case must be made fast in order to enhance performance. This is because the common case is simply the most common function / use case. For example in building a game. It would make more sense to optimize the games performance while in the bounds of the play space, rather than some area that should not be visited.

Performance Via Parallelism

By performing parallel operations we may see better performance because of this. An example being the turbines of a jet. A jet will perform better with two turbines, while they are working parellel to one another.

Performance via Pipelining

A pipeline allows the user to breakdown the steps in order to make them more manageable. It is a response to the previous "pipe." \

Performance via Prediction

Prediction allows the designer to improve the performance of a design as long as the action is certain. However the designer must be careful in making sure that an incorrect performance is not too expensive of an operation.

Hierarchy of Memorys

Programmers may assign importance to memory in order to increase its speed. The fastest, smallest, and most expensive per bit is at the top of importance because the data will take the longest to load. Say for example a social media account. An account that you may not have used in 5 years may take longer to load because within the hierarchy of data it is not often accessed.

Dependability via Redundancy

We make systems have redundant data because of the possibility that some may fail. This is known as **redundancy**. We use this in order to ensure data correctness no matter the circumstance.

For more about these topics read What is a Computer?

1.4 Textbook Readings

Input devices- recieve user input through mediums such as a microphone. These are used for gathering user input so that it may be interpreted through the computer.

Output devices- output signals sent by the computer, and example of this may be a monitor or a speaker

LCDs (liquid crystal displays) are used in many personal devices such as mobile phones. They are thin and small displays making them a perfect fit for portible devices. The LCD does not produce light, rather it controls the transmission of light. It is typically rod shaped in size.

- Straightens when a current is applied, and it no longer needs to bend light
- Liquid crystal is between two screens polarized at 90 degrees, so light cannot pass unless it is bent
- Modern LCDs use an active matrix which has tiny transistor switches within each pixel.
- Then a red, green, or blie mask is associated with each pixel to control the current and so it may produce sharper image

Images are composed of a matrix of pixel elements which can be represented as a matrix of bits, this is known as a bit map.

Touch Screens

Touch screens are a product of the POSTpc era. It uses direct touch instead of the traditional method of using a keyboard and mouse for the user interface. People work as electrical conductors, because of this touch screens take advantage of users touch through glass that is covered with a transparent conductor, distorting the electrostatic field of the screen.

Inside the computer

Integrated circuits- this is also known chip, it is a device combining dozens to millions of transistors.

CPU (central processing unit)- this is also known as a processor.

- Contains datapath, control, and adds numbers
- Tests numbers
- Signals I/O devices to activate

Datapath- the component of the processor that performs arithmetic operations

Control- The component of the processor that commands the datapath, memory, and I/O devices according to instructions

Dynamic random access memory (DRAM)- Memory built as integrated circuits. It provides random access to any location.

Static random access memory (SRAM)- is faster, however it is less dense and hence more expensive. SRAM and DRAM are two layers of memory hierarchy.

Typically, the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions so that application programmers do not need to worry about such details. Th e combination of the basic instruction set and the operating system interface provided for application programmers is called the application binary interface (ABI).

Computer designers distinguish architecture from an implementation of an architecture along the same lines: an implementation is hardware that obeys the architecture abstraction. Th ese ideas bring us to another Big Picture.

- Communication: Information is exchanged between computers at high speeds.
- Resource sharing: Rather than each computer having its own I/O devices, computers on the network can share I/O devices.
- Nonlocal access: By connecting computers over long distances, users need not be near the computer they are using.

Binary Arithmetic

We use the decimal numbering system (base 10) in our everyday lives. This can be seen below:

$$632 = (100 * 6) + (10 * 3) + (1 * 2)$$

This may be seen through the formula

$$\sum_{i=0}^{2} d_i = dn * 10^n + dn_{-1*10^{n-1+\ldots}} d_1 * 10^1 + d_0 * 10^0$$

d_n is the most significant digit and d_0 is the least significant digit.

Generalize - Numbering System with base X

In Base X, a non-negative intereger $d_n d_{n-1} \ldots d_1 d_0$

$$\sum_{i=0}^{2} d_i * Xi = dn * Xn + dn_{-1} X^{n-1+\ldots+} d_1 * X^1 + d_0 * X^0$$

- The same number can have many representations on many bases
- For example, consider the decimal number 23
- Computers use a binary system so 23 would be 10111_2
- Sometimes we use a hexadecimal system to make reading binary representations 'easier'. $23_{10} = 0x17(17_{16})$

`cout` converts the binary storage of the number into the requested format
The C language uses specifiers to convert within `scanf()` this can be seen in C for C++ Users

Base 10	Base 2	Base 8	Base 16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9

Base 10	Base 2	Base 8	Base 16
10	1010	12	A
11	1011	13	B

Number Representation and Binary Arithmetic

- To convert to binary repeatedly divide the decimal number by 2, until the quotient is 0.
- Collect the remainders as you go
- Write down the remainders from left to right

Index

Binary Number Systems

By dividing by 2 repeatedly we are representing each bit of the number.

We write the numbers backwards because we are increasing the power of 2 that the number is divided by. So 0 is not the number divided by 2, rather it is the number divided by x. An example can be seen below:

284731	Bit	Divided by
142365	1	2
71182	1	$2^2 = 4$
35591	0	$2^3 = 8$
17795	1	
8897	1	
4448	1	
2224	0	
1112	0	
556	0	
278	0	
139	0	
69	1	
34	1	
17	0	
8	1	
4	0	
2	0	
1	0	
0	1	

This is how the data is represented within memory. The excess space is filled with 0's. So for the above example, if written in 32 bit format would have 13 leading 0s.

Octal digits are formed with groupings of digits, So the octal of above is

00	000	000	000	001	000	101	100	000	111	011
0	0	0	0	1	0	5	4	0	7	3

Hexidecimal representation is done in the same way, however it is grouped in 4's

0000	0000	0000	0100	0101	1000	0011	1011
0	0	0	4	5	8	3	B

- Starting from the least significant digit bit, multiply the digits of the binary number with increasing powers of 2.
- The least signifcant bit (LSB) is multiplied with 2^0, the next bit by 2^1, and so on.
- Then add the products

0	1	1	0	0	1	1	0	1	0	1	1	0	0	0	1
0^{15}	0^{14}	2^{13}	0^{12}	0^{11}	2^{10}	2^9	0^8	2^7	0^6	2^5	2^4	0^3	0^2	0^1	2^0

For binary addition we need to carry if the product is greater than 1. We shift so they allign on the left side of the table

For subtraction we need to borrow, the borrowed number is 10, at a lower bit.

```
0 - 0 = 0
1 - 0 = 1
1 - 1 = 0
10 - 1 = 1
```

a = 01110011 (115)
b = 00011111 (31)

So...

$$01110011 - 00011111 = 01010100(84)$$

-	-	-	-
1	0	1	0
-	1	0	1
0	0	0	1

- Two's complement in n bits:
  - The negative of a twos complement is given by inverting each bit (0 to 1 or 1 to 0) and then adding one, ignoring any carry between n bits.
- If numbers are represented in n bits:
  - the positive numbers are from 0000..01 to 0111..11,
  - 0 is all 0: 000..00
  - the negative numbers are from 1000..00 to 1111..11
  The leading bit sign 0 means non negative, the leading 1 means negative.

bit 1	bit 2	bit 3	bit4
0	0	0	0
1	1	1	1

now if we add one to the twos compliment shown of `0000` (compliment is 1111) we get the result of `1 0000`, however because the 1 is out of range $0 = 0$

Binary	Decimal	Signed Binary
NA	-3	100
NA	-2	101
NA	-1	110
0	0	000
001	1	001
010	2	010
011	3	011

Example: Calculating Two's Complement

- Given number: 01010101
- One's Complement: 10101010
- Two's Complement: 10101010 + 1 = 10101011
- Applications of Two's Complement:
  - Arithmetic operations (addition, subtraction, multiplication, division)
  - Negative number representation in computers
  - Overflow detection
- Benefits of Two's Complement:
  - Simplifies arithmetic operations
  - Eliminates the need for separate addition and subtraction circuits
  - Only one representation of zero (no positive and negative zero)

$-89_{10} =$

-	1	1	0	1	0	1
43	21	10	5	2	1	0

Twos compliment as a formula is therefore:

$$2^n - x$$

- Also in 2's complement notation, ^{there} is the only 1 representation for 0.
- We often need to convert a number in n bits to a number represented with more than n bits
- an example of this would be char to int for example
- This can be done by taking the most significant bit from the sorter one and replicating it to fill the new bits of the larger one

- Existing bits are simply copied

So we duplicate the most significant digit, so
`char x = -107 != 0000000000010101`
`if short y = x ... its = 111110010101

we DON'T add 0's, we duplicate the most significant digit

Branch Encoding

- To encode branch instructions, we first need to calculate the value for the associated label. This is done by the assembler
- Mips has restricted alignment. The encoded instruction are word aligned, which means all the labels will be multiples of 4.
- To increase the range, we encode the address divided by 4
  - In other words, the address is in terms of words/instructions (32 bit), rather than bytes

Branch target/ branch displacement/ branch offset: the number of instructions between the branch instruction and the target instruction

- It should be calculated as the

$$\frac{(targetaddress - currentaddress)}{4}$$

```
l2:     instruction                     # Opcode :4
                instruction                     # rs: t0(8), rt: t1 (9)
                instruction                     # branch target:-4 (16 b)
                BEQ $t1, $t0, L2
```

Mips J Format

- Used for unconditional jumps and function calls
- The opcode field is used to identify the type of instruction
- The targaddr field is used to indicate a pseudo-direct target address
- Pseudo direct addressing is used to provide a maximum range for the jumps while still maintaining a 32-bit address into a 26 bit-field

This is done by...

```
Enter NOTES from 5/31 here
```

Multiply and Divide Instructions

2 Address Forms. These instructions implicitly use the internal registers hi and lo. We would then need to move the values from these internal registers into the programmer-usable registers.

- `mult rs, rt`

    - puts the high word in hi and low word in lo
- `div rs, rt`
    - puts the remainder in hi and the quotient in lo
- `mfhi $rd`
    - Copies the value in hi to the given register
- `mflo $rd`
    - Copies the value int lo to the given register

% is the same then as normal division within MIPS programming

Character and String Operations

- Characters are encoded as 0's and 1's using ASCII encoding
- Each character is represented using 8 bits (or a byte)
- A string can be stored in the data segment by using the .asciiz directive. This would be a C-style string (array of chars)
- MIPS provides instructions to move bytes to work with individual characters
- LB (load byte) loads a byte to the rightmost 8 bits of a register
- SB (store byte) writes the rightmost 8 bits of a register to memory

Sys call 4 - Prints the string with the starting address 0
Sys call 11 - Prints character stored in a0
Sys call 8 - reads a string from console. a0 should contain the starting address of the unformatted free space, and a1 should contain the maximum number of characters to read
Sys call 12 - reads a character from the console and returns it in v0

Functions

`jal l1` - Jump and link so the computer knows where to return to following the jump

C for C++ Users

Some C++ Only Features

- Boolean type
- Classes & Objects
- namespaces and using statements
- exception handling (with `try, catch, throw`)
- using `new` and `delete` for dynamic memory management
- templates
- Function Overloading
- Pass by Reference
- Default Paramenters

The c style implementation of libraries look like
`# include <stdio.h>`
There are 4 diffrent styles of casting (static, dynamic, const, and reinterpreted casts)

Declerations

- While declaring arrays, the size of the array should be specified as `const` or a literal. The size of an array may not be variable

`int size =100;`
`int list[size];` - Would NOT work
`int list[100];` - Would work

> *Unlike c++, C requires variable declarations at the top of the functions*
> *This means that, ALL variables should be declared at the beginning of the function*

```
for (int i=0; i< 10; i++){} // wrong
```

```
int i;
for(i = 0; i < 10; i++){} // correct
```

Standard I/O streams

`stdin::` input stream
`stdout::` output stream
`stderr::` error stream

Streams are sequences of characters flowing from one place to another

`stdio.h` - contains basic I/O functions

Formatied I/O - refers to the conversion of data to and from a stream of characters for printing

`scanf:` - Reads from standard input (stdin)
`printf:` - writes to standard input (stdout)

Output with printf

```
printf (format_string, list_of_expressions);
```

- format_string is the layout of whats being printed
- list_of_expressions is a list of comma seperated variables

Specifier	Purpose
%d	int (signed decimal int)
%u	unsigned decimal int
%f	floating point (fixed notation)
%e	floating point (exponential notation)
%s	string
%c	char
%x	Hexidecimal number

- We can specify the field width. Defaults to right-justification (use - for left), we use this syntax:

`printf(%10d)` - for 10 spaces

Using scanf basics

```
scanf (format_string, list_of_variable_addresses)
```

- format string is like that of `printf`
- But instad of expressions, we need space to store incoming data, hence the list of variable addresses.

```
int month, day;
printf("Please enter birth month, followed by birthday");
scanf(%d %d, &month, &day);
```

Conversion Specifiers

- Mostly the same for output, Some small diffrence
- Use %f for type float, but use %lf for types double and long double

C Strings

`char word1[20];`
`scanf("%s", word1);`

- Similarly you can read a string into a char array with scanf.
- Charecters are read from the keyboard until the first white space, the null character is automatically placed at the end for termination

```
char greeting[] = "Hello";
printf("%s", greeting); // Prints the word
```

Output: "Hello"

- The format string used for a scanf command is essentially a regular expression
- The data from stdin comes in as a string. The conversion specifier converts the data into the correct "type".
- If we specifiy a particular charecter in the format string, scanf will ignore that character if it occurs at the same spot
- We can also instruct scanf to look for or ignore particular characters
- **Example**
  `scanf("%25[^*]*%s", str1, str2);`

> *C-Strings dont need the address because the name acts as a pointer to the array*

Structs and enums

- C & C++ definitions of a struct are the same
- In C++, declaring a struct or an enum automatically creates a new type
- In C, we need to remind the compiler about the user defined types everytime we use them

> *We can define a new type in C using* `typedef`

```
typedef struct student {
char name[20];
double gpa;
} Student; // now the new type name
struct student s1; // C declaration using "tag"
Student s2; // C declaration using new type name
```

Dynamic Memory Allocation

- C does not come with the `new` and `delete` operators. We need to use `malloc` and `free`.
- These functions are in `` - The syntax for an array of ints is: `line * arr = (line*) malloc

(size_of_array * sizeof(int));`

For an arrau of the line strict, which has been typedef'ed is:

```
line * arr = (line*) malloc (size_of_array * sizeof(line));
```

Index

Exceptions

Spim Input

- Spim allows you to read from the keyboard
- Receiver control bits tell the computer weather anything has entered the keyboard buffer
- 0 - no interrupt (like \n)
- 1 - triggers an interrupt (like endl)

Each data byte we need to read/write is mapped to an area of memory, and then we need to loop until the system is "ready with the data" by checking the "ready bit" periodically.

This is inefficient and is unsustainable, since the processor has to do many things, and is potentially running other programs

Interrupts

With external interrupt, if an event happens that must be processed the following will happen

1. The address of the instruction that is about to be executed is saved into a special register called EPC
2. PC is set to be 0x800000180, the starting address of the interrupt handler which takes the processor to the interrupt handler
3. The last instruction of the interrupt should be "eret" (similar to jr) which sets the value of the PC to the value stored in EPC (similar to ra)

Is it okay to use $t0 in the interrupt?

- For an interrupt the user program is running and gets interrupted. The user program does not know about the interruption
- So if you changed 4t- inside an interrupt, after it returns the user program will not be aware, thus using the wrong value of $t0.

Floating Point Numbers

How do we represent numbers such as 5.75 in binary
In general to represent a real number in binary

- Find the binary representation of the int part
- Find the representation of the float part
- put a decimal point in between

$5_{10} = 101_2$
$0.75_{10} = 0.11_2$
$5.75_{10} = 101.11_2$

> *It is not however stored in this manner, but this can be used if done on paper or just for example.*

You must multiply by two and collect the whole number parts, as opposed to dividing by 2 and collecting the remainders.

4.25_{10}
$4_{10} = 100_2$

Fractional Part	Multiply by 2, collect the whole
$0.25 * 2$	0.5- bit is 0, remainder is 0.5
$0.5 * 2$	1- bit is 1, remainder is 0. stop

Write In the FORWARD direction
$4.25_{10} = 100.01_2$

12.375_{10}
$12_{10} = 1100_2$

Fractional Part	Multiply by 2, collect the whole
$.375 * 2$	.75 -> 0
$.75 * 2$	1.5 -> 1
$.5 * 2$	1. Stop

$12.375_{10} = 1100.011_2$

- Floating point numbers have an acceptable level of conversion, however it is not exact. EX. `3.10000000012143` for the value `3.1`
  For other Conversion View Binary Number Systems & Binary Arithmetic

Normalized Sign-Magnitude Form

- Represents floating point numbers
- $Num = F * 10^E$m, where F is the mantissa and E is the exponent
  - $251.38 = 2.5138 * 10^2$
- This is "normalized" is F's "whole number part" is a single digit number
  - scientific notation: $37.381 * 10^4$
  - $3.7381 * 10^5$ is normalized

The formula for the normalized notation is shown below:

$$1.x_2 * 2^y$$

We must because of limited bit storage weigh the trade-off of precision and size.

IEEE 754 Floating Point Standard

For Single Precision:

Bits	31 (1 bit)	30-23 (8bits)	22-0 (23 bits)
Purpose	Sign	Exponent (biased)	Mantissa

- Since the leading 1 bit is the most significant it is normalized in binary numbers as 1. It is not explicitly represented
- Defines standards for single and double precision floating point numbers

Mantissa is every number that follows the decimal point in the normalized notation

- Most negative numbers are represented as 00...00 rather than the positive alternative of 111...11
- This is done by adding 127 to the actual exponent to make sure the value stored is always positive.

The value of a number in IEEE 754 single precision is thus:

$$(-1)^{Sign} * x(1 + 0.Mantissa) * 2^{(Exponent-127)}$$

Double Precision

- Uses 64 bits (two 32-bit words)
- 1 bit for the sign
- 11 bits for the exponent
- 52 bits for the fraction
- 1023 as the bias

Bits	63 (1bit)	62-52 (11 bits)	51-0 (52 bits)
Purpose	Sign	Exponent (biased)	Mantissa

Overflow/underflow happens when a number is outside the range of a particular representation

- For example the exponent might be smaller than -38 (underflow) or larger than 38 (overflow)

The mantissa can be adjusted with the exponent for loss of precision, or might result in a denormalized number
Note that arithmetic operations can result in overflow/underflow

Procedure for converting decimals to float

1. Write in binary
2. Convert to standard form
3. The power of 2 is the exponent and the fractional part is the mantissa
4. Add the exponent to the bias (127 for 32 bit, 1023 for 64 bit) and convert the sum to binary
5. Write down the number as sign bit, exponent mantissa. Fill out the remaining bits with 0s.
6. Spilt the number into groups of 4. For each of the 4 bits write Hex equivalent

Intro To MIPS

The process or contains some amount of local storage known as the Register

- A process has registers and the ALU
- Registers are where you store values that are currently being worked on
- The values stored in the registers are sent to the alu to be added, subtracted, etc.. the result is stored back in the register

Memory

- Modeled in continuous space
- Every byte in the memory is associated with an index, called the address
- We can read and write in the memory
- Give the address to the memory hardware, we can read the content in that byte
- Given the address and a byte value, we can modify the content in the memory at that address

Program and Data

- consists of instructions and data, both stored in memory
- Instructions in binary

Machine Code

- It is binary!
- The instruction set constitutes the vocabulary of the machine. These are the words understood by the machine itself
- To work on the machine we need a translator
  To see more about translation visit Program Translation

Why learn Assembly Language?

- Knowing assembly illuminates concepts not only in computer organization but operating systems, compilers, parallel systems, etc.
- Understanding high-level constructs are implemented leads to more effective use of those structures.
  - Control constructs (if, do-while, etc)
  - Pointers
  - Parameter Passing

MIPS is a RISC (Reduced Instruction Set Computer) instruction set, meaning that it has simple and few instructions

- Originated in the early 1980's
- An acronym for Microprocessor without Interlocked Pipeline Stages

RISC Philosophy-

- fixed instruction lengths
- load-store instruction sets
- limited number of addressing modes
- limited number of operations

The Four ISA Design Principles

Simplicity favors regularity

- Consistent instruction size, instruction formats, data formats
- Erases implementation by simplifying hardware
  Smaller is Faster
- Fewer bits to access and modify
- Use the register file instead of slower memory
  Make the common case fast
- Small constants are common, thus small intermediate fields should be used
  Good design demands good compromises
- Compromise with special formats for important exceptions
- A long jump (beyond a small constant)

Logical Operations

`&` compares every bit

- 0 and 0 = 0
- 0 and 1 = 0
- 1 and 0 = 0
- 1 and 1 = 1

0	**0**	**0**	**1**	**0**	**0**	**0**
0	0	0	1	0	1	1
-	-	-	-	-	-	-

`||` operates in the following way

- 0 or 0 = 0
- 0 or 1 = 1
- 1 or 0 = 1
- 1 or 1 = 1

`Xor` works in the following way

- 0 xor 0 = 0
- 0 xor 1 = 1
- 1 xor 0 = 1
- 1 xor 1 = 0

Xor being the negated version of or

Bitwise Logical Instructions

How to implement NOT using NOR?

- Using `$zero` as one of the input operands
- It is included in some implementations of MIPS as a pseudo instruction

```
ADD $target, $source1, $source2
```

The key letter `I` may be added in order to add an intermediate.
The key letter `U` may be used in order to make it unsigned

Memory Operands

- memory contains both data and instructions
- Memory can be viewed as a large array of bytes
- The address of a variable or instruction is its offset from the beginning of memory

`g = h + A[5]`

- lets say g and h are associated with the registers `$s1` and `$s2` respectively. Let's also say that the base address of A is associated with register `$s3`

```
lw $t0, 20($s3) #load the element at a 20 byte offset from $s3
add $s1, $s2, $t0
```

Loops in MIPS

```
sum = 0;
for(i = 0; i < 100; i++){
        sum += a[i];
}
```

Replace the for statement using an if and goto statement:

```
sum =0;
i =0;
goto test;
loop: sum += a[i];
i++;
test: if (i < 100) goto loop;
```

Then in assembly:

```
                li $t2, 0                       # sum = 0
                or $t3, $0, $0                  # i = 0
                j test
Loop:   sll $t5, $t3, 2                 # temp = i * 4
                add $t5, $t5, $t4               # temp = temp + &a
                lw $t5, 0($t5)                  # load a[i] into temp
                add $t2, $t2, $t5               # sum += temp
                addi $t3, $t3, 1                # i++
Test:   slti $t5, $t3, 100              # test i < 100
                bne $t5, $zeor, loop            # if true, goto loop
```

For info about registry types: MIPS 3

```
while (save[i] == k){
        i+=1;
}
```

The solution within assembly will then be

```
# s3 <= i, s5 <= 5, *arr <= s0
LOOP:    SLL $t0, $s3, 2                  # t0 = i*4
                ADD $t0, $t0, $s0                # *t0 = arr + i
                LW  $t1, 0($t0)                  # t1 = arr[i]
                BNE $t1, $s5, DONE               # Loop done is not equal
                ADDI $s3, $s3, 1                 # i+=1
                j LOOP                           # jump back to loop again
DONE:
```

Example Program for finding largest value in an array

```
#largest.asm
# MIPS code to find the largest element in an array

        .text
        .globl main
main:   la      $s0,Arr         # address of arr *Arr
        la      $s1,size        # address of size
        lw      $s1,0($s1)      # s1 = 12 (size of array)
        ori     $t0,$0,1        # t0 = 1 (i=1)
        lw      $s2,0($s0)      # largest (s2) = arr[0]
LOOP:   beq     $t0,$s1,DONE    # if i == size, we're done
        sll     $t1,$t0,2       # t1 = i*4
        add     $t1,$s0,$t1     # t1 = &arr[i]
        lw      $t2,0($t1)      # t2 = arr[i]
        slt     $t3,$s2,$t2     # t3 =1 if largest < arr[i]
        beq     $t3,$0,NEXT     # if t3==0, then largest was bigger
        or      $s2,$t2,$0      # largest = arr[i]
NEXT:   addi    $t0,$t0,1       # i++
        j       LOOP            # back to condition check (if i<size)

DONE:   la      $a0,str         # Print str
        ori     $v0,$0,4        # 4 is syscall code for print string
        syscall

        or      $a0,$s2,$0      # print value in s2
        ori     $v0,$0,1        # 1 is sycall code for print int
        syscall
```

```
        ori     $v0,$0,10       # 10 is syscall code for end program
        syscall

        .data
Arr:    .word   17, 32, 21, -15, 99, 65, 42, 17, -80, 0, 19, 77
size:   .word   12
str:    .asciiz "The largest value is "
```

MIPS 3

Shift left logical (sll) move all the bits to the left by the specified number of bits (fill empty space with 0s)

`s11 $t2, $t0, 2`

Shift right logical (srl) move all the bits to the right (again fill empty with 0s)

`srl $t2, $t0, 2`

Shift right arithmetic (sra) move all the bits to the right (fill empty bits with the sign bit, 1 for negative 0 for positive)

`sra $t2, $t0, 2`

```
000000110000 >> 2 # shift right 2
-> 000000001100
```

Mips Labels

- LA- Load address
  - Syntax: `LA $reg, label`
  - Function- Stored the address the label is pointing at into the register
- LI- Load Immediate
  - Syntax- `LI $reg, immediate`
  - Function- Stores the given immediate value into the register
- Both of these instructions are actually a combination of 2 instructions (LUI and ORI)

```
if(i == j){
        k = k + i;
}
bne $t2, $t3, L1 #if (t2 != t3) goto l1
addu $t4, $t4, $t2
```

```
slt $t3, $t1, $t2
# set t3 to 1 if less t1 < t2.
# else clea t3 to be 0

slti $t3, $t1, 100
```

```
# set t3 to be 1 if t1 < 100
# else clear t3 to be 0
```

- Translate into mips the following

```
if(a > b){
        c = a;
}
```

MIPS

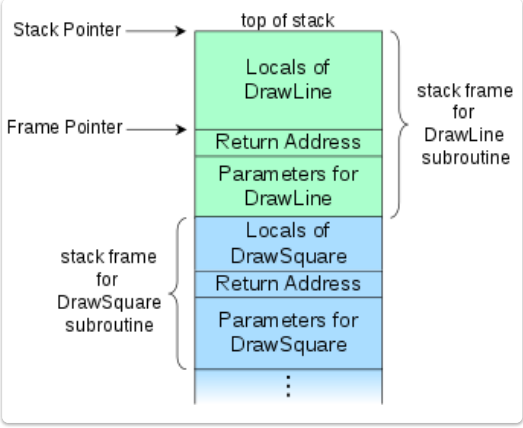
```
slt $t5, $t3, $t2 # b < a
beq $t5, $zero, L1 # if t5 == 0
or $t4, $t2, $zero #c = a
```

`slt` sets the target to 1 if the value of source 1 is less than source 2
In the above example we needed to switch the values for the comparison statement. If we wanted to add an else statement we need an unconditional jump

Mips 6

Every time a function/subroutine is called, a stack frame is created for the function

- The stack frame contains areas for
  - Arguments
  - Saved registers
  - Return address
  - Local data
  - Some padding



The function call stack grows downwards

Mips Function call Convention

1. Allocate space in the stack by moving the stack pointer. Calculate the room required by counting the number of registers that require storage 2. Store caller temporary registers (that the caller may use later) - the `t` registers and `a` registers on to the top of the stack 3. Store the `$fp` value onto the stack then copy `$sp` to `$fp` 4. Store the `$ra` value on the stack 5. Save the argument to the `a` registers. If there are move than 4 arguments, store the extra arguments on the stack 6. Call the function using `JAL`

Suppose a function f0 is shown below...

```
funct1: addi $sp, $sp, -4
                sw $ra, 0($sp)
                jal func2
                lw $ra, 0($sp)
                jr $ra
```

- func1 can return to fun0

- func0 cannot return to its caller correctly
- func2 can return to its caller correctly

MIPS Assembly File

What we have learned

R-type instructions - register addressing

```
ADD
SUB
OR
AND
XOR
NOR
```

I-Type instructions - immediate addressing

```
ADDI
ANDI
ORI
XORI
```

I-type - unsigned 16 bit immediate

```
ADDIU
ANDIU
```

I-type instructions - 2 registers, 16 bit immediate (base-displacement addressing)

```
LW
SW
LH/SH
LB/SB
LHU, LBU
```

General format

```
.text

.globl main
```

```
main:
        # instructions here

.data
# allocation of memory
```

MIPS Directives

Directive	Meaning
`align n`	Align next datum on 2^n boundary
`.asciiz str`	Place the null-terminated string str in memory
`.byte b1,b2,..bn`	Place the n byte values in memory
`.data`	Switch to the data segment
`.double d1,d2,..dn`	Place the n double-precision value in memory
`.float f1,f2,..,fn`	Place the n single-precision value in memory
`.global sym`	The label sym can be referenced in other files
`.half h1, h2,..hn`	Place the n half-word values in memory
`.space n`	Allocated n bytes of space
`.text`	Switch to the text segment
`.word w1, w2,...wn`	Place the n word value in memory

For the register types visit Mips Registers

Mips Registers

Pre Lecture Video Notes

An instruction is a command that hardware understands

- Instruction set is the vocabulary of commands understood by a given computer
- Included arithmetic instructions, memory access instructions, logical operations, instructions for making decisions

The general classes of MIPS instructions are

- Arithmetic
  - add, subtract, multiply, divide
- Logical
  - and, or, not, not shift
- Data transfer
  - load from or store to memory
- Transfer control
  - Jumps, branches, calls, returns

Arithmetic Instructions

- Each MIPS only performs on operation
- Each one must have 3 variables
- These variables can be the same

```
add a, b, c     # a = b + c;
add a, a, a.    # a = a + a

sub a, b, a   # f contains t0 – t1

# a = b – ((b+c) + a + a)
```

- if the statement is more complex we need to break it into pieces

In MIPS, operands for general arithmetic operations must be from registers of constants

- 32 programmer useable registers

- Reflects *"Faster is better"*
  Registers use less power

Name	Number	Use
`$zero`	0	Const 0
`$at`	1	Assembler temporary, for resolving pseudoinstructions
`$v0 – $v1`	2-3	Function results and expression evaluation
`$a0 – $a3`	4-7	Arguments
`$t0 – $t9`	8-15, 24-25	Temporary
`$s0 – $s7`	16-23	Saved Temporary
`$k0 – $k1`	26-27	OS Kernel
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

More about how these work: Intro To MIPS
More about the use of these: Logical Operations

The general form for a mips instruction is

```
Instruction_mnemonic $target, $source1, $source2
```

- MIPS is case insensitive (not case sensitive) so `Add` can be `ADD` or even `aDd`

We don't have variables, we have registers.

Performance

Measuring the performance of a computer

Latency (response time) - is the time between the stand and completion of an event
Throughput (bandwidth) - is the total amount of work done in a given period of time

1. Replacing your processor with a faster one will:
    - increase throughput and decrease latency
2. Adding additional processors to a system
    - Only throughput will increase

> *Performance is the inverse of time*

$$Performance = \frac{1}{ExecutionTime}$$

$$Performance(x) > Performance(y)$$

Term	Definition
Elapsed Time|total wall clock time needed for task
CPU Time|time cpu spends actually working on the program
User CPU Time|CPU time spent in the program itself
System CPU Time|CPU time spend in the OS, performing tasks
Clock Cycle|the basic discrete time of a processor clock
Clock Period|the length of each clock cycle
Clock Rate|inverse of the clock period

> *Elapsed Time VS. the CPU Time may show I/O inefficenys, due to confusion of the user*

Term	Definition
Bit	Binary digit
Nibble	Four bits
Byte	Eight bits
Word	four bytes(32 bits)
Kibibyte (KiB)	2^10 (1024) bytes

$$CPUExecutionTime = CPUClockCycles * ClockPeriod$$

$$CPUClockCycles = Instructions * AverageClockCyclesPerInstruction$$

- The average number of clock cycles per instruction is often abbreviated as CPI. The above equation can be rearranged to give the following

$$CPI = \frac{CPUClockCycles}{InstructionCount}$$

Computer A: Clock cycle time 250ps and 2.0 CPI
Computer B: Clock cycle time 500ps and 1.2 CPI
Which Is Faster?

$$ComputerA(250 * 2.0) < ComputerB(500 * 1.2)$$

From this we can gather the basic equation

$$CPUTime = InstructionCount * CPI * ClockPeriod$$

Or

$$CPUTime = \frac{InstructionCount * CPI}{ClockRate}$$

Component	Unit Of Measure
CPU Execution Time	Seconds (for the program)
Instruction Count	Instructions Executed
Clock Cycles Per Instruction	Avg Number of Clock Cycles
Clock Cycle Time (Clock Period)	Seconds Per Clock Cycle

- No one of these factors makes a CPU ***FASTER*** so we must look at all factors in determining the speed of a computer

Amdahl's Law

Amdahl's Law states that the performance improvement to be gained from some faster mode of execution is lmited by the time the faster mode can be used

$$ImprovedExecutionTime = \frac{AffectedExecutionTime}{AmountOfImprovement} + UnaffectedExecutionTime$$

Energy Efficient Processors

- Extend battery life for mobile systems
- Reduce heat dissipation for general purpose processors
- Energy cost for increasing

Enhanced capability available to users led to new classes of computers. This led to the dominance of microprocessor-based computers. Allowing programmers to trade performance

for productivity. Nature of applications is changing because of this fact.

Index

Program Translation

Program translation uses system translation in order to translate it to machine code.

- There are four general areas of memory in a process
- The text area contains the instructions for the application and is fixed in size
- The static data area is also fixed in size and contains
  - Global variables
  - Static local variables
  - String and sometimes floating-point constants
- The run-time stack
  - Contains activation records, each with an associated functions invocation
  - Saved levels of callee-saved registers
  - Local variables and arguments not allocated to registers
  - Space for the max words of arguments passed on stack to other functions
- The heap contains dynamically allocated data

1. Loads onto memory (in a format -> code space + operation space)
2. OS gives program a control, "launches process"

Name	Process
sp	Stack↓ ↑Dynamic Data/ Heap
gp	Static Data
pc	Text
Address 0	Resereved

- The stack starts are the high-end of memory growing downwards, while the head grows upwards into the same space
- The lower end of memory is reserved
- The text segment follows, housing MIPS machine code

If the stack and the Dynamic Data/Heap meet than we unfortunately have a segmentation fault

The Translation Process

- Preprocessing - handled by the compiler
- Compiling - translates to assembly file
- Assembling - translates to machine language
- Linking - creates an executable

- Loading - the executable is ran through this

The compiler depends on the programming language, the assembler depends on the chipset. It must be translated to meet the machine code of the chips type.

The linker finds functions, and other libraries. It "links" the library code into the program.

This eventually produces an executable. The loader must be invoked after to run the executable.

Compiling

- We use the term "compiling" as a general term to refer to the entire translation process from source file to executable
- Compilers are responsible for...
  - checking syntax
  - making semantic checks
  - performing optimizations to improve performance, code size, and energy use

Assembling

- Assemblers take an assembly file as input and produce an object file as output
- Assembling is accomplished in two passes (typically)
- First pass: Stores all identifiers representing address or values in a table as there can be forward reference
- Second pass: translates the instructions and data into bits for the object file

A symbol table is create. It is essentially a dictionary for the names and addresses. Any identifier created by the user and not the language is stored in here

The Object File

- An object file header describing the size and position of the other portions of the object file
- Text segment containing the machine instructions
- The Data Segment containing data values
- Relocation information identifying the list of instructions and data words that depend on the absolute address
- A symbol table containing global labels and associated addresses in object file and the list of unresolved references
- Debugging information to allow a symbolic debugger to associate machine language with the addresses of the source line and the variable name

Linking

- Linkers take the object file and object libraries as input and output executables

- Resolve any external references by finding symbols in another object or library
- Aborts if any external reference cannot be resolved
- Determines the address of absolute references using relocation information in the object files
- The executable has similar format to object files with no unresolved references or relocation information

Loading

The loader copies the executables file from disk

- reads the file header to determine its size of text and data segments
- allocates the address space for processing
- copies the instructions into the text segment and data into static data segment
- copies arguments passes to the program onto the stack
- initializes the machine registers the stack pointer
- jumps to a start-up routine that will call the main function

Recursion

- Demonstrate how to write a recursive function in mips

Function Review

Every time a function/subroutine is called, a stack frame is created for the function The stack frame contains areas for - Arguments - Saved registers - Return address - Local data - Some padding -

Function Call Conventions

1. Caller right before call
2. Callee upon entry
3. Calle right before exit
4. Caller upon return

```
// MIPS vs C++ (Mips conventions shown)
main(){
```

1

- Save a, t, & ra registers and the Frame Pointer
- Copy val of stack pointer into Frame Pointer
- Place arguments in a registers
- Call the function

```
vat = calculate(a, b, c); }
```

4

- Save v registers
- Restore the values saved to the stack
- Restore the stack pointer

```
int calculate(int a, int b, int c){
        // 2
        ...
        ....
```

```
        .....
        // 3
        // Store s registers
        // Restore s registers
        // Place val in v register
        return ans;
}
```

Recursive Functions

- A function that can call itself
- Base condition so its not infinite loop
- Change of state is needed

Will keep calling itself with different parameters, until a terminating condition is met

```
n = 3 (a0)                          // Stack Growth
res = 6 (v0)                        //    |  |
ra to main                          //    |  |
(call)                              //    |  |
n = 2 (a0)                          //    |  |
res = 2(v0)                         //  \_____/
ra to fact(3)                       //   \    /
(call)                              //    \  /
n = 1 (a0)                          //     \/
res = 1 (v0)
ra to fact(2)
```

Demonstrates why we need to save the a registers. They, as well as ra, are very important for recursive function

The code below demonstrates how the function call should be preformed for a recursive function. The full code can be viewed sqrt.asm

```
        .text
        .glovl main

main:
        addi $sp, $sp, $sp, -12                 # allocate 3 words
        sw $a0, 0($sp)                          # store values in stack
```

```
        sw $fp, 4($sp)                          # frame pointer - 4
        sw $ra, 8($sp)                          # reg address - 8
        or $fp, $sp, $0                         # fp = stack pointer

        li $a0, 10                              # param to function = 10
        li $s2, 1                               # exit condition number
        jal fact                                # function call

        or $s1, $v0, $0                         # save return value

        lw $ra, 8($sp)                          # restore stack
        lw $fp, 4($sp)
        lw $a0, 0($sp)
```

What is a Computer?

Syllabus

Why Study Computer Organization

"The Medium is the Message" - Marshall McLuhan

- We must be effective at understanding the medium (computers) in order to more effectively understand how to use them.
- Why some methods are more effective than others
- Why do supercomputers run numerical methods multiple times
- Why is a program written diffrently depending on the machine, for some languages

Abstraction is the KEY to Computing

Abstraction	Examples
Multi-level translation	Different Languages and systems can work together
Number Representations	Applied in Networks
Processor Design and Pipelining	Different architecture for diffrent needs

What is a Computer

Anything with a processor, and the ability to "compute"!

A computer is divided into 3 classes
- Desktop
- Servers
- Embedded Computers

A PMD (personal mobile device) is a newer class of computers, that have become larger than these other types of computers. It can be carried around, however this develops the problem of conserving power. PMD have batteries, and they are portable making them diffrent than the other types.

Eight Great Architectural Ideas

- Design for Moore's Law
- Use abstraction to simplify design
- Make common case fast
- Performance via parallelism
- Performance via pipelining

- Performance via prediction
- Hierarchy of memories
- Dependability of redundancy

Design For Moore's Law

- The number of transistors on a chip doubles every 18-24 months
- Architects have to anticipate where technology will be when the design of a system is completed
- Use of the principle is limited by Dennard Scaling

> *Dennard Scaling - Law of Diminishing returns*

Use of abstraction to simplify design

- Abstraction is used to represent diffrent levels of a design
- Lower-level details can be hidden to provide a simpler models at higher levels

Make the common case fast

- Identify the common case and try and improve it
- Most cost efficent method

Via Parellelism

Via pipelining

- Break tastks into smaller tasks so they can be simultaneosly preformed in diffrent stages
- Commonly used to improve instruction thoughput

Via Prediction

- Sometimes faster to assume a particular result than waiting until the result is known
- Known as speculation and is used to guess branches

Use a hierarchy of memories

- Make the fastest, smallest, and most expensive per bit memory the first level accessed and the slowest largest, and cheapest per bit memory the last level accessed
- Allows most of the accesses to be caught aat the first level and be able to retain most of the information

> *Moving some of the memory to specific places so we can find it faster*

Improve dependability via redundancy

- Include redundant components that can both detect and often correct failures
- Used at many diffrent levels

Program Levels and Translation

The computer speaks in terms of electrical signals

- > *0V is "on" and 0V is "off"*
- 1 is "on", 0 is "off"

> *The Compiler only takes you as far as assembly code*

Stage	Definition
Compiler	Translates a high-level language to assembly
Assembler	Symbolic representation of instructions
Linker	Combines multiple files into a single ex

For day 2 notes check Performance