Lecture 1

January 8, 2018 1:05 PM

Controllers vs CPUs

- CPU: Central Processing Unit, a computational engine
- Controller: a simple or partial CPU used to manage a machine (not a computer)

Lecture 2

January 10, 2018 1:05 PM

Computers have ports

- Logical ports can and often do outnumber physical ports

System Board

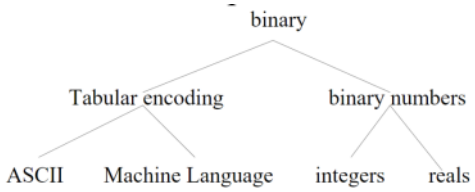
- CPU
- ROM, read only memory, has libraries to help CPU
- RAM, slow memory (still fast relatively)
- Cache, fast memory
- Bus can only have one byte on it at a time
    - CLK (clock) regulates how quickly new bytes can access the bus
- Power Supply
    - Converts current from the home into the steady current needed in the PC
- CPU
    - Math, logic, data, movement, loops
    - CPU clock
        - Byte (word) movement within the CPU chip
        - Instruction execution within the CPU chip
        - Influence does not reach outside of CPU chip
    - Primarily connected to the cache and PCI slots to increase performance
    - It is a computation engine without intelligence
        - Input: single atomic instruction
        - Action: compute a result
        - Output: the result
    - Basic contents
        - ALU (Arithmetic Logic Unit)
            - +,-,>,<,==
        - Registers
            - Fast live memory locations
        - IP (instruction pointer)
            - Points to the next instruction
        - IR (instruction register)
            - Stores current instruction
        - On-board Cache
            - Specially constructed for speed
- ROM
    - Used to store built-in instructions, additional instructions for the CPU
- Battery
    - Used to help keep the CMOS parameters (including time)
- RAM
    - Volatile main memory bank, large and slow
- Bus
    - A common road for data that interconnects all devices on the motherboard
- CLK
    - Beats the processing cycle (2 of them)
- Slot
    - Connects to devices external to the motherboard through cards
    - PCI
        - Faster
    - ISA
        - Slower
- Communication Pathways
    - Purpose

        - Method by which we can interconnect components
        - Some run in parallel to CPU, some need the CPU
    - Formation
        - Composed of multiple wires, each wire for 1 bit
        - Each wire carries a signal independently of the other, therefore parallel
    - Many pathways exist
        - System Buses, Data Bus, CPU Bus, connector wires
- Bit is either 1 or 0
    - Byte is 8 bits
    - Register is n bits usually larger than a byte, smallest addressable group of bits
- Everything in the computer is binary

Lecture 3

January 15, 2018 1:04 PM

Assume word to be 32 bit



- Tabular Coding: an ad-hock mapping of a byte or word to some meaning
- Binary numbers are true numbers in the mathematical sense, supporting +-/* etc…
- Every 4 bits are equal to 1 hex digit
- Every 3 bits are equal to 1 octal digit
- If a number is carried past the defined type size it is an Arithmetic Overflow
    - e.g. if the correct answer is 11001, the arithmetic overflow answer will be 1001
        - A bit has gotten lost
- 2's complement notation
    - Take the #
    - Flips the bits
    - Add 1
        - e.g: 00101
        - Becomes 11010
        - Add 1, 11011
    - Ignore the overflow when subtracting
    - To do 10's complement take the factor of 10 one larger and subtract
        - e.g -12 becomes 100-12=88

Data Types

- Implementing a data type
    - Logical description
        - The theoretical type(int, real, char)
        - Definition of how it behaves
            - Defined by operations and operators
    - Physical Construction
        - The data structure in byte format
    - A Circuit that implements the algorithms
- A string is a bunch of bytes concatenated together
    - Each byte is a letter or symbol
    - The string starts with a byte showing how many letters or symbols are in the String
        - Once its read that many it stops

Lecture 4

January 17, 2018 1:03 PM

Circuits

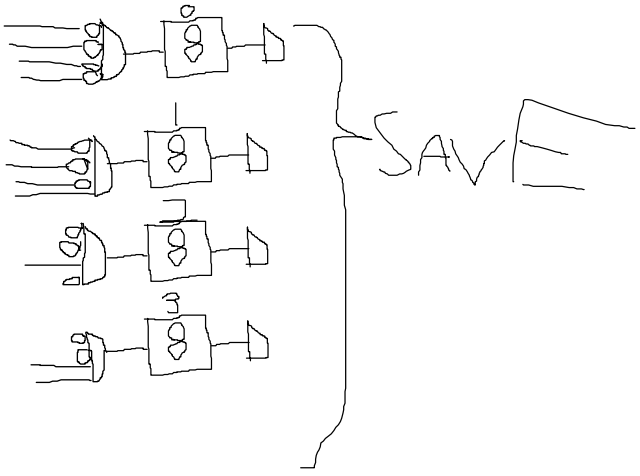
- Logic Gate & Wires
    - Logic Gate
        - Simple machine with inputs and outputs
        - Same input will always yield same output
        - Not, And, Or, Xor, Nand, Nor
            - As many inputs, only one output
            - Xor is false if X and Y are both true
            - 0 is false, 1 is true
        - Unidirectional
    - Gates can be combined with wires to form more complex systems
        - These are called circuits
    - Decoder is like a truth table or takes a binary and gives an int
    - Encoder is like integers to binary
- Truth Tables and Circuits
- Basic Combinatorial Circuits
- ROM
    - Read only memory
    - Stores info like the Ascii table
    - Can store logic which can be executed by the machine at a later time
    - PLA
        - Programmable Array Logic
    - PAL
        - Programmable Logic Array

Lecture 5

January 22, 2018 1:03 PM

What is data
- Information
  - Letters, numbers, web pages, etc…
- Used to be represented by light bulbs
- The Bit
  - Binary value: 1 or 0

RAM
- SRAM
  - Static RAM
- DRAM
  - Dynamic RAM
  - Needs a constant flow of electricity
- Read
  - Address Register: integer
  - Mode Register: read flag
  - Data Register: loaded from RAM Data will be copied from RAM at integer address
- Write
  - Address Register: integer
  - Mode Register: write flag
  - Data Register: data
- 2 types
  - Video buffer on RAM
  - Pointer than points to a graphics card
- A flip flop circuit is used to save
  - Flip flop uses the clock to regulate the flow of data
  - Data is saved to the Q output



Lecture 6

January 24, 2018 1:06 PM

Register

- A self contained unit storing a single data value
- Similar properties to RAM
    - Can write to it
    - Can read to it
    - Stores a word of data (32 bits)
- Different from RAM
    - It is located in the CPU
    - It is the fastest memory in the computer
    - Instead of an address is uses a sequencer
- Two kinds
    - General Purpose
        - Used like a variable
        - It can store a value temporarily
        - Connected only to the bus
    - Specific Purpose
        - A register that is connected to another machine
        - Its input or output has a particular function (since it is connected to another machine)
- Data coming out of the CPU is slowed down to the clock speed of the RAM
- Sequencer is connected to every register who tells the registers when they can communicate
    - So only when the clk and the sequencer send true can the register access the bus

Circular Bus

- Helps resolve the issue of feedback loops with registers

Lecture 7

January 29, 2018 12:38 PM

ALU

- Has two input registers (L and R)
- Has two output registers (A-out and Status)
- Has another input that says whether to add, subtract, multiply, etc…
- Arithmetic Logic Unit
    - Performs mathematical operations
        - Only addition and subtraction
        - Special hardware needed for multiplication and division (co-processor)
    - Performs logic operations
        - Equal, not equal, greater than, less than
- L, R, A-out follow the same format
    - Signed registers, or
    - Two's complement registers (assume this)
- Status is a series of bit flags
    - Each bit in the register measures a true/false property of a-out: zero? Negative? Overflow?
    - Changes depending on inputs
- L and R also have 2's complements that can be selected
- To check if the output is 0 take the output from A and see if they're all 0's
- To check if A-out is negative check if the leftmost digit is 0 or 1, 1 is negative, 0 is positive
- To check for overflow, evaluate the initial values of the registers

Lecture 8

February 5, 2018 1:09 PM

Co-Processor

- Floating or fixed point numbers have different circuitry
  - This requires different registers and ALU
  - Normally done in a co-processor
    - Unlike an in, the 32bit code is broken into a sign, exponent, and mantissa (number being multiplied for Scientific Notation(significand))
- Collection of CPU and Co-Processor and other bits known as the chip-set
  - All run at the same speed
- Basic Addition Algorithm
  - Normalize number
  - Round if necessary
  - Shift values to same exponent
  - Add bases
  - Normalize result
  - Round if necessary
  - Update the sign
- Basic Multiplication Algorithm
  - Remove exponent bias
  - Match exponents
  - Multiply bases
  - Normalize and round if needed
  - Update the sign

Micro-Instructions/Micro-Code

- Macro View
  - Each instruction from RAM is loaded into the CPU on at a time and executed one at a time
- Micro View
  - Each instruction is composed of series of simpler steps, known as Micro Instructions
  - The clock ticks for each of these simpler steps
- What is an instruction
  - An atomic command
  - A formatted string of binary
  - A CPU can only execute one command at a time
  - Instructions are stored in RAM
  - CPU downloads 1 instruction at a time and executes that instruction
  - Programs/algorithms are made out of these instructions

$(-1)^S \times (1 + \text{Sig}) \times 2^{(\text{Exp} + \text{Bias})}$

a)

$-69.(25)_{10} = 1000101 . 001_2 = 1.000101001 * 2^6$

Mantissa:
Throw away the leading 1 to end up with a fractional mantissa of 000101001

True exponent : 6 (note we can drop the 1 bit as well)
Biased exponent = $6 + 127 = 133_{10} = 10000101_2$.
The number is negative, the sign bit is 1.

Write the sign bit, exponent and mantissa in the IEEE format:

1 10000101 00010100100000000000000

Pack the three groups of bits into one word to get the final answer:
11000010100001010010000000000000

It is good practice to represent the final answer in hexadecimal : **C28A4000**

In RAM

Note that the fractional mantissa had only 9 bits, so we had to add zeros at the right end to get a 23-bit mantissa as required by the IEEE single precision format.

Basic Representation

Example Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

Answer The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction:

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)}$$

and so when we add the bias 127 to the exponent of $-1.1_{two} \times 2^{-1}$, the result is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126 - 127)}$$

The single precision binary representation of -0.75_{ten} is then

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022 - 1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Example Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 4.44.

Answer Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$\begin{aligned} 0.5_{ten} &= 1/2_{ten} = 1/2^1_{ten} \\ &= 0.1_{two} = 0.1_{two} \times 2^0 = 1.000_{two} \times 2^{-1} \\ -0.4375_{ten} &= -7/16_{ten} = -7/2^4_{ten} \\ &= -0.0111_{two} = -0.0111_{two} \times 2^0 = -1.110_{two} \times 2^{-2} \end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2. Add the significands:

$$1.0_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$0.001_{two} \times 2^{-1} = 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3} = 1.000_{two} \times 2^{-4}$$

Since $127 \geq -4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$1.000_{two} \times 2^{-4} = 0.0001000_{two} = 0.0001_{two} = 1/2^4{}_{ten} = 1/16_{ten} = 0.0625_{ten}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Multiplication Example

Example Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten} using the steps in Figure 4.46.

Answer In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{rr} & 1.000_{two} \\ \times & 1.110_{two} \\ \hline & 0000 \\ & 1000 \\ & 1000 \\ & 1000 \\ \hline & 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence the product is

$$-1.110_{two} \times 2^{-3}$$

Converting to decimal to check our results:

$$-1.110_{two} \times 2^{-3} = -0.001110_{two} = -0.00111_{two}$$
$$= -7/2^5{}_{ten} = -7/32_{ten} = -0.21875_{ten}$$

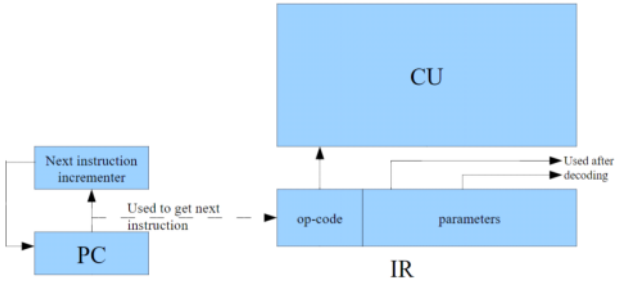
The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

Lecture 9

February 7, 2018 1:04 PM

What is a control unit?

- Control Unit (CU) or sequencer
- A ROM-like structure that defines and executes the microcode for every instruction
- Op-code iterated through by the adder in the CU
- See lecture for slide 12



Comments

```
##
## hello.a - prints out "hello world"
##
##      a0 - points to the string
##

#################################################
#                                               #
#               text segment                    #
#                                               #
#################################################

        .text
        .globl __start
__start:                # execution starts here
        la $a0,str      # put string address into a0
        li $v0,4        # system call to print
        syscall         # out a string

        li $v0,10
        syscall         # au revoir...

#################################################
#                                               #
#               data segment                    #
#                                               #
#################################################

        .data
str:    .asciiz "hello world\n"

##
## end of file hello.a
```

Different Load instructions

OS call

How to read

How to run

Code

Data

The PC is used to download an instruction from RAM into the IR. The instruction is then sent to the CU to be "decoded" (into its microcode steps) and "executed" (by activating the correct sequence of gates per step).

Lecture 10

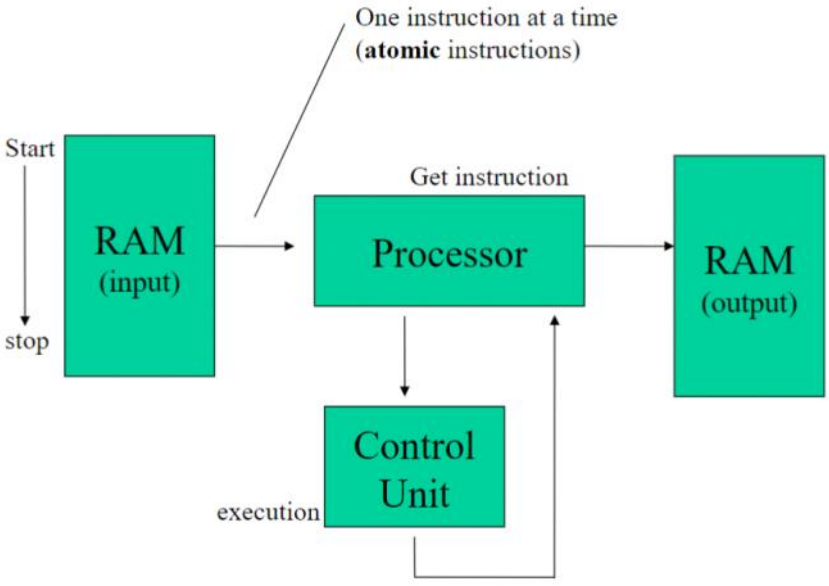
February 12, 2018 12:59 PM

Von Neumann Machine
- Simple computer

Classical CPU and Instructions
- One cycle is one micro code step
- CPU execution is an infinite loop
- Execution Cycle
  - Fetch
  - Decode
  - Execute
  - Store Result
- IR pathways
  - Op code limits the range of addresses that can be addressed

Instruction Flow
-

Evolution of the CPU
-

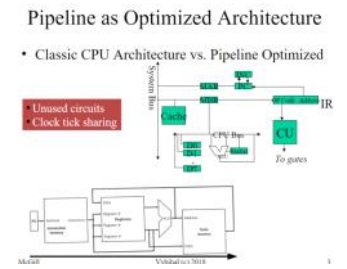


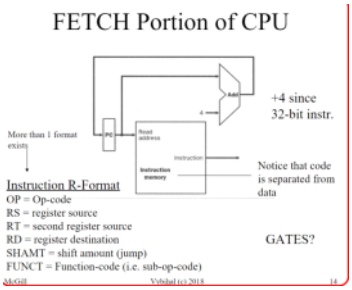
Lecture 11

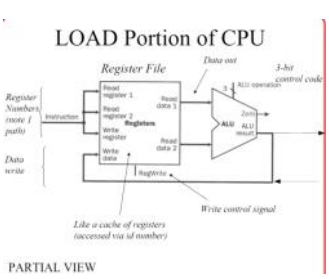
February 14, 2018 1:28 PM

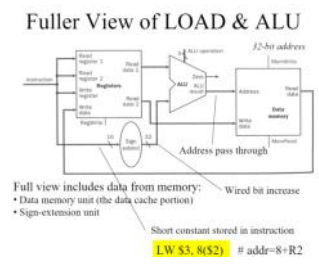
Pipeline Architecture
- Optimized to compute as a pipeline
  - As we are doing a FETCH of the next instruction we can also do a LOAD of the current instruction
  - Actually, 4 instructions can be executed in stages at the "same time"
- Dumb loading
  - Download lots of instructions

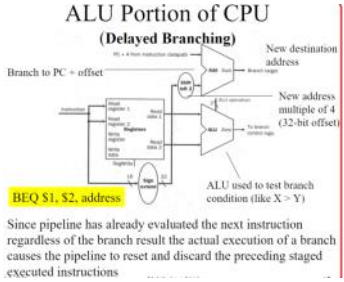
Multi-Purpose ALU
- Depending on instruction class
  - Load/store: compute memory address
  - R-Type: AND, OR, Sub, Add, set-on-less-than
  - BEQ: uses subtraction

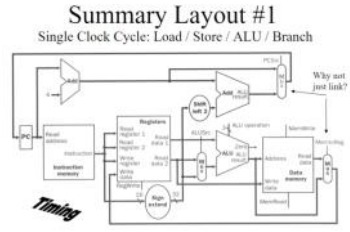












Lecture 12

February 19, 2018 1:02 PM

R2000 - 210000 MIPS

- Specifications: RISC ~ reduced instruction set
  - 5 stage pipeline
    - Fetch instruction, load registers, ALU, cache, store
  - Cache (n-byte guess load (dumb load))
  - 32 bit instructions
    - Constant size to facilitate pipeline
    - 3 operand instructions
  - 32 general purpose registers
  - Minimum support for:
    - Status code register (uses general purpose register)
    - Stacks (only SP register - Stack Pointer Register)
    - Subroutines ($31 register is return address)
    - Interrupts (pre-defined memory location jump)
    - Exceptions (similar facility as with interrupts)

MIPS Design

- CPU design philosophies
  - CISC (complex instruction set computing)
    - Intel x86 and Motorola 680X0
    - Many powerful instructions (like: load n int to array)
      - Single instruction that can do many things
      - Instruction power = m clock ticks, such that m very large
      - Non or minimum pipeline capabilities (complexity)
  - RISC (reduced instruction set computing)
    - MIPS(R2000, R10000)
    - Few and simple instructions (like: load R1 with 5)
      - Instruction = n clock ticks such that n is = 1 or close to that
      - Pipelines exist and can be optimized
  - MIPS uses the RISC philosophy

Programs only Execute when…

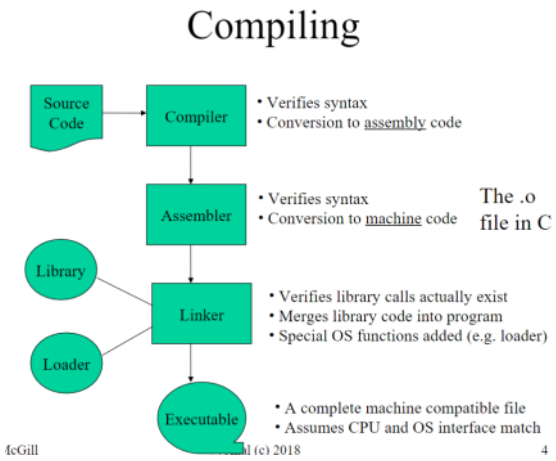
- Source code version same as compiler
- Machine code output same as CPU instruction format on computer
- Program's loader version same as OS API on computer

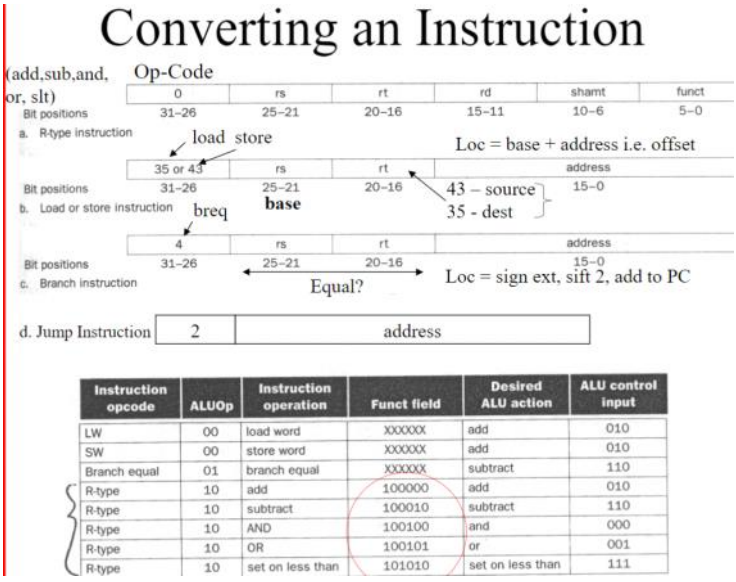
Assembler

- Converts an assembler text file into a .o file
- And assembler program consists of
  - Assembler instructions
  - Assembler directives
    - Help control the assembling process and help the programmer in various ways
      - Comments, etc…

Two-Pass Assembly

- Pass 1: build symbol table
  - Identify all labels
  - Determine offset address of all identifiers
- Pass 2: build machine language program
  - Basic 1-to-1 map between assembly and machine code
  - With help from the symbol table
  - Remember all MIPS instructions are 32-bits long





Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Lecture 13

February 21, 2018 12:21 PM

Things to know about Assembler Programming

- Advanced assembler:
  - Access to internal OS routines
  - Access to the control ROMS of all peripherals
- Take care of complexity explosion
  - Keep it simple
  - Modular programming with subroutines important
  - Use OS functions when useful
- Know how to use the registers and stack
  - Use them over RAM
- .ascii gives no null
- .asciiz gives a null at the end
- Addressing Modes
  - Register Addressing
    - Operand is a register
    - E.g. add $s1,$s2,$s3
  - Base or Displacement Addressing
    - Operand is a memory location
    - Register + offset <- a constant
    - E.g. lw $s1, 100($s2)
  - Immediate Addressing
    - Operand is a constant (no addressing)
    - 16-bit constant
    - E.g. Addi $s1, $s2, 100
  - PC-Relative Addressing
    - Memory location = PC + offset <- a constant
    - E.g. j 2500 or j label
  - Pseudo-Direct Addressing
    - Memory location = PC (top 6 bits) concat with 26-bit offset
    - Assumers 32-bit addressing
    - E.g. jal 2500 or jal label

Lecture 14

February 26, 2018 1:05 PM

Forms of Data

- Immediate
  - Data stored within an instruction
    - R1=R2+5; addi $t1,$t2,5 OP:9:10:5
- Variables
  - Data stored in RAM
    - Int X = 5; X:.word 5
- Register
  - Data stored in a register
    - R1=X; lw$t1,X OP:9:addr
- Stacked
  - Data stored on the run-time stack
    - Push(X);
- Heap
  - Data stored in the heap
    - Malloc(sizeof(int));
- All data must be copied into a register before it can be used by the CPU
- Built-in types
  - Circuitry exists in the computer to implement these types
    - Character, integer, fixed point, unsigned, void *pointer, offset indexing
- Language types
  - Circuitry does not exist in the computer to implement these types
  - These are instead simulated through algorithms and libraries from more primitive elements that are supported by circuitry
    - String, array, matrix, struct, list, tree, object, etc…

Example 1

C Language:

a = b + c;

```
.data
b:   .word    10
c:   .word    20
a:   .word    0
```

MIPS:

lw $t1, b
lw $t2, c
add $t0, $t1, $t2
sw $t0, a

Note:
Cannot use variables directly in CPU

Example 2

C Language:

Example 2

C Language:

```
f = (g + h) – (i + j);
```

MIPS:

```
lw $s0, g
lw $s1, h
lw $s2, i
lw $s3, j
add $t0, $s0, $s1
add $t1, $s2, $s3
sub $s4, $t0, $t1
sw $s4, f
```

Example 3

C Language:

```
g = h + A[8];
```

Assume the array A is of size 100 and contains 32-bit integer values.

MIPS:

Assume: $s1=g, $s2=h, $s3=base address of array

Note: *array* simply means *a contiguous block of memory* (nothing more)

```
lw $t0, 32($s3)    # 8 * 4 bytes per cell = 32
add $s1, $s2, $t0
sw $s1, g
```

Example 4

C Language:

A[12] = h + A[8];

MIPS:

Assume similar setup from example 3.

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

Lecture 15

February 28, 2018 1:09 PM

IF-THEN-ELSE

- if (i == j) f = g + h; else f = g – h;

```
# Assume f,i,j,g,h are $s0,$s3,$s4,$s1,$s2

      bne $s3,$s4,Else    # go to Else if i != j
      add $s0,$s1,$s2     # f=g+h (skipped if i != j)
      j   Exit            # go to Exit
Else: sub $s0,$s1,$s2     # f=f-h (skipped if i == j)
Exit:
```

WHILE LOOP

while(save[i] == k) i = i + j;

```
# Assume i,j,k are $s3,$s4,$s5
# Save address in $s6

Loop: add $t1,$s3,$s3     # Temp reg $t1 = 2 * i
      add $t1,$t1,$t1     # Temp reg $t1 = 4 * i
      add $t1,$t1,$s6     # $t1 = save[i]
      lw $t0,0($t1)       # $t0 = save[i]
      bne $t0,$s5, Exit   # go to Exit if save != k
      add $s3,$s3,$s4     # i=i+j
      j   Loop
Exit:
```

Example 5

C Language:

```
if (i == A[j])
{
   f = g + h;
} else {
   f = f – i;
}
```

MIPS:

Assume: $s0=i, $s1=j, $s2=f, $s3=g, $s4=h
$s5=base address of array
containing 16-bit integer numbers

```
add $t0, $s1, $s1 #calculate offset
add $t0, $t0, $s5 # base + offset
lh $t1, 0($t0)
bne $s0, $t1, L1 # could do beq also
add $s2, $s3, $s4
j Exit
L1: sub $s2, $s2, $s0
Exit: …
```

Lecture 16

March 19, 2018 12:57 PM

Sub-Routines

- Calling techniques
  - Register Based
    - Use registers a0 to a4 to pass arguments
    - Use registers v0 and v1 to return values
  - Stack Based
    - Passing parameters and returning results using the run-time stack
- The Anatomy of a Function
  - Calling
    - JAL
    - Passing parameters
      - Stacking them or use the $a registers
  - Protecting registers locally
    - All registers you use in function get stacked to protect the calling environment
  - Returning
    - Restore the calling environment
    - Return value using $v registers
    - JR
- The Elements
  - Double-duty for general purpose registers that support procedures:
    - $a0-$a3: pass parameters
    - $v0-$v1: return values
    - $ra : return address = PC+ 4
  - The jump-and-link commands
    - JAL ProcedureAddress
      - Step 1: save current address into $ra = PC + 4
      - Step 2: jump to ProcedureAddress
  - The Return (jump register) Statement
    - Jr $ra
- By Convention
  - $t registers not protected by the stack
    - Considered to be temporary registers
  - $s registers are protected by the stack
    - Considered to be saved registers
  - All other registers are optionally saved by the programmer
- Register Based method
  - Pros
    - Fast and easy to code
  - Cons
    - Limited number of registers
- Run-time stack based
  - Pros
    - Stack is very large so many parameters can fit
    - Simulates local variables when protection of registers are used
  - Drawback
    - Slows down program by increasing number of lines of code and the number of move operations

Example

```
int calc(int g, int h, int i, int j)
{
   int f;
   f = (g+h)-(i+j);
   return f;           // return (g+h)-(i+j)
}

# assume: $a0=g, $a1=h, $a2=i, $a3=j
# Extra local variable $s0 = f

Calc: add  $t0,$a0,$a1   # g+h
      add  $t1,$a2,$a3   # i+j
      sub  $s0,$t0,$t1   # subtract the two results
      add  $v0,$s0,$zero # put answer in return register
      jr   $ra           # return
```

Run-time stack based

```
Main:# result = calc(a,b,c,d);

    # Setup parameters using stack
    subi $sp, $sp, 16   # make room
    sw $t0, 12($sp)     # assume t0=a
    sw $t1,  8($sp)
    sw $t2,  4($sp)
    sw $t3,  0($sp)

    # Call the subroutine
    jal calc  # $ra <-- return address

    # Return values assumed in v0 or v1
    sw $v0, result
```

Example with protection

```
# assume: on stack g, h, i, j
# Extra local variable $s0 = f

Calc: lw   $a0, 12($sp)  # load the parameters
      lw   $a1,  8($sp)
      lw   $a2,  4($sp)
      lw   $a3,  0($sp)
      subi  $sp,$sp,12    # protect 3 registers
      sw   $t1,8($sp)     # stack 3 registers (push)
      sw   $t0,4($sp)
      sw   $s0,0($sp)
      add  $t0,$a0,$a1    # Do work: g+h
      add  $t1,$a2,$a3    # i+j
      sub  $s0,$t0,$t1    # subtract the two results
      add  $v0,$s0,$zero  # put answer in return register
      lw   $s0,0($sp)     # pop stack
      lw   $t0,4($sp)
      lw   $t1,8($sp)
      addi  $sp,$sp,28
      jr   $ra            # return

Alternatively we could have pushed the result and
popped it out in main.
```

Lecture 17

March 21, 2018 1:04 PM

Small Data

```
        .data                       .text
A:      .byte   'x'     Char:       lb $t0, A
B:      .word   10      Integer:    lw $t0, B
C:      .float  5.2     Float:      lw $t0, C

D:      .double 5.2     Double:     la $s0, D
                        1st half:   lw $t0, 0($s0)
                        2nd half:   lw $t1, 4($s0)
```

Medium Data

```
        .data                       .text
S:      .asciiz "hi"    String:     la $t0, S
                        Get char:   lb $t1, 0($t0)

                        # 1D Array
A1:     .word 0,0,0,0   Ex1:        la $t0, A1
A2:     .space 40       Get:        lw $t1, 0($t0)
                        # A2 is array of 10 integer
                        # or array of 40 characters
```

Index measured in offset.

AKA Buffer

Large Data

2D Array as .data

```
# int A3[10][2]
A3:   .space 80

# first 40 = 1st row
# second 40 = 2nd row
# int = 4 bytes
```

2D Array as .text

```
Addr:  la $t0, A3
CellSize:  li $s0,4
RowSize:   li $s1,40
# $t1 = A3[5][1]
# Let A3[n][m]
# Offset: $s2 =
          (m*$s1)+(n*$s0)
Load:      lw $t1, 0($s2)
```

Large Data

Struct in .data area

```
# struct {int x; double y;} z;

Z:    .space 12

# since int = 4 bytes
# and double = 8 bytes
```

Struct in .text area

Notice that it looks like an array except that the cells are not the same size.

```
Z:    la $t0, Z
X:    li $s0, 0 # offset to X
Y:    li $s1, 4 # offset to Y
GetX: add $s2, $t0, $s0
-->   lw $t1, 0($s2)
```

Passing large data as parameters

How does C do it: using pointers.

```
Char array[100];        array: .space 100

print(array);           la $t0, array
                        subi $sp, $sp, 4
                        sw $t0, 0($sp)

Void print(char *p);    # use ptr in function
```

Passing large data as parameters

- How does C do it?
  - Char array[100];
  - Print(array);
  - Void print (char *p);
  - Notice we do not need to actually pass the array, just the pointer

Passing large data as parameters

How can we pass **struct**: using variable

Assume: struct {int x; double y;} z; # size = 12 bytes

Then:

```
>   la $t0, z
>   lw $t1, 0($t0)      # get x
>   lw $t2, 4($t0)      # get first half of y
>   lw $t3, 8($t0)      # get second half of y
>   subi $sp, $sp, 12   # make space
>   sw $t1, 12($sp)     # put x in the first spot, etc.
```

Simulating Local Variables

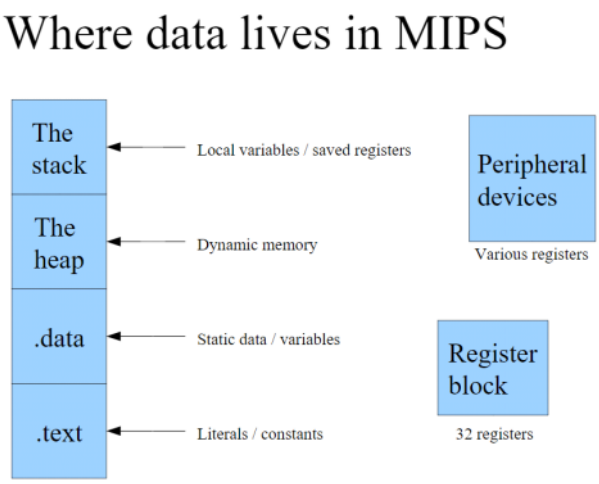
- int a = 5;
  - Static
    - LI $t1,5
  - Global by .data
    - LA $t1, LABEL
      - "call-by-reference"
      - LW $t2, 0($t1)
  - Local by stack
    - LW $t2, offset($SP)
      - "call-by-value" or value can be a reference
      - Pushed previously, we know offset number

Lecture 18

March 26, 2018 1:08 PM

Recursive Procedures

- Problems:
  - Limited number of registers
    - Multiple calls
    - Parameters for each call
    - Local variables
  - Only one $ra register
    - We need it for each return statement



Lecture 19

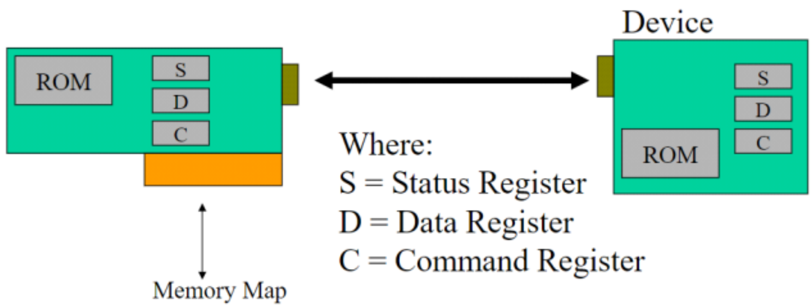
March 28, 2018 12:58 PM

Mars Limitations on Terminal Connections
- Only simulates one terminal connection
  - Text keyboard
  - Text video

Terminal Connection
- Receiver control and transmitter control
  - Control interrupts
  - If interrupts = 1, ready = 0
- Receiver data and transmitter data
  - Send data in 8 bit size(1 byte)

CPU Execution Changes
- Exceptions
  - Any event that stops the normal flow of the CPU execution cycle. i.e. interrupting the <u>currently executing process</u> due to a <u>run-time</u> event, like error (overflow, divide by zero, etc…) or interrupt.
- Interrupt
  - Signal (throw)
    - An event purposely triggered by the <u>current process</u> to reroute the CPU to execute another process or function
  - Trap (error)
    - An even purposely triggered by a <u>device</u> to reroute the CPU to execute another process or function

Flow Diagram



<u>Device Operation</u>:
1. Device updates S and D on device & copied to controller
2. Waits a small unit of time and then downloads and reads C and D (possible data loss)

<u>Polling Method</u>:
1. CPU checks S for changes in a busy loop
2. If change then handle the case

<u>Interrupt Method</u>:
1. If S or D updated signal CPU
2. CPU Task switch to OS

Example Polling Code

```
############################################
#                GETCHAR
############################################

GetChar:
    lui $a3, 0xffff         #base address of memory map
CkReady:
    lw $t1, 0($a3)          #read from receiver control register
    andi $t1, $t1, 0x1      #extract ready bit
    beqz $t1, CkReady       #if 1, then load char, else keep looping
    lw $v0, 4($a3)          #load character from keyboard
    jr $ra                  #return to place in program before function call
```

```
     010101010X
&    0000000001
-------------------
     000000000X
```

Lecture 20

April 9, 2018 1:06 PM

Storage Conundrum

- We need fast and voluminous storage to run applications adequately
- Two issues
  - Speed of access
  - Amount of storage
- Speed and storage physically work against each other when compared to cost in dollars
- Memory Hierarchy
  - To manage efficiently memory is organised into layers where each layer is optimally geared to execute program
    - Registers>Cache>RAM>Disk
  - So why use cache?
    - To speed up the operation of the computer
    - Primary problem:
      - It is smaller than the program
  - How to optimally load cache
    - Issues:
      - What to load?
      - How to address?
      - Hit to Miss ratio (cache miss rate)
        - Hit implies we find the instruction in the cache
        - Miss implies we need to go to the RAM
          - Address doesn't exist in cache
      - Cache miss/refill penalty
    - Wide Bus
      - 4 byte address
      - R/W
      - 16 byte data
      - Basically gets back a big block of instructions (like predictive returns kinda)
- Principle of Locality
  - Temporal Locality
    - Item will be referenced again soon
    - Example: Library and functions
  - Spatial Locality
    - Adjacent items will probably be executed next
    - Example: Loops and functions
- How to fit RAM into cache
  - Use modulo arithmetic to store addresses
  - Block/RAM address mod (number of cache blocks in the cache)

Storage Comparison

Storage	Technology	Speed	Cost
CPU Registers	Flip-flops	1 – 5 ns	$250 - $300
Cache	SRAM – transistors + power	5 – 25 ns	$100 - $250
RAM	DRAM – capacitors + refresh	30 – 120 ns	$5 - $10
Disk	Magnetic charge – mechanical	10 Million ns – 20 million ns	$0.1 - $0.2

~ Per 100 Meg

Operations in a Miss

1. IR = Cache[PC] ← Error: Miss
2. MDR = RAM[PC] MDR= memory data register
3. Wait for RAM to complete read
4. Cache[PC] = MDR & update table
5. Restart the instruction (from cache)

Miss penalty = Cycles to upload data to cache

Cost of miss = Miss frequency * Miss penalty

Program speed = n + m*penalty, where n & m are no. of instructions

n= instructions found, m = instructions missed

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

Amdahl's Law

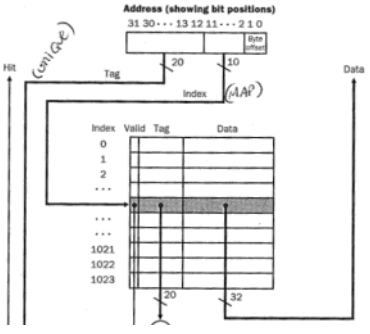
- The speedup in performance is proportional to the new component and the actual fraction of work it carries out.

```
s = 1 / [ (1 - f) + f/k ]
```

Where:

- S is the speedup
- F is the fraction of work performed by the component
- K is the advertised speedup of the new component

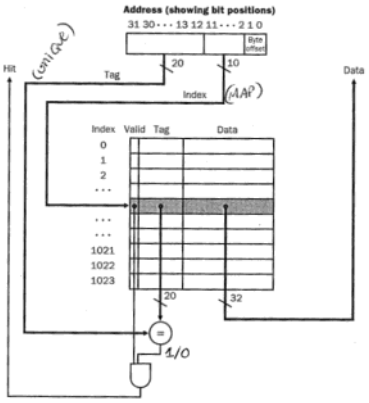
Basic Access Architecture



Where:

- S is the speedup
- F is the fraction of work performed by the component
- K is the advertised speedup of the new component

Basic Access Architecture



Lecture 21

April 11, 2018 1:18 PM

Virtual Memory

- Using the disk as a buffer
- We execute the programs function by function
    - So we only really need that function to be in memory and the data it uses (the rest of the program could still be on disk)
- Motivation
    - A simulator for memory giving your computer the impression that it has more RAM
    - Removes the burden from a programmer in managing limited RAM
    - VM helps to allow multiprocessing by simulating more space in RAM

Definitions

- Page
    - A functional unit
    - A variable sized space that can fit one entire function unit
- Frame
    - A fixed size space
    - The OS divides RAM into frames
        - Makes RAM look like an array where each frame has an index from 0 to n