# Week 1

## Lecture 1: Intro

Assessment:

* 80% exam in May
* 10% quizzes (8 best)
* 10% exercise (assembly language programming)
* PASS: 7/10 labs

Course is about interaction between hardware and software

Why learn about Systems:

* Intellectual: a fascinating technology and a central aspect of our culture
* Practical: knowledge of computer systems is helpful in programming and improving application performance
* Future proofing: you often need to design software solutions for the systems we'll have several years in the future
* Understanding the basic concepts of systems will help you to use computers more effectively
* Computing is a deep subject, with interesting history, ideas, theory, philosophy

Levels of abstraction:

* Software:
  + Applications: spreadsheets, DBs, programs
  + Programming languages and compilers
  + OSs
    - Functionalities:
      * Files
      * Protection
      * Processes
      * Threads
      * Virtual memory
      * Communication
      * etc
* Instruction Set Architecture (ISA): machine language
  + E.g. MIPS, arm, x86, etc
  + Simple enough for machine language
  + Powerful enough for high level languages to be translated to ISA
* Hardware:
  + Microarchitecture
  + Digital Circuits
    - Only a few simple components
    - Complex behaviour achieved by putting these components together
  + Electrons

Types of Computer Systems:

* Servers
  + Used for few large tasks (engineering apps) or many small ones (web server, Google)
  + Fast processors, lots of memory
  + Multi-user, multi-program
* PCs
  + Laptops, desktops
  + Balance cost, processing power
  + Few users, multi-program
* Mobile devices
  + Highly integrated (multiple processors, GPU, GPS), low power
  + Single-user, multi-program
* Embedded devices
  + Task specific: sensing, control, media playback
  + Low cost, low power
  + Single program

Data representation:

* Data types used in programming
* They must all be represented in hardware, which is based on basic electronic components
  + For example, we could use voltage on a wire to represent info
* Computation then requires manipulation of these voltages
* Approaches
  + **Analogue**
    - Making variables in problem proportional to physical measures
    - The physical measure is an “analogy” to the measure in the problem
    - Can be extremely fast
    - Disadvantages:
      * Limited precision
      * After many calculations, errors will accumulate
      * Hard to represent data other than natural numbers (integers, strings), large numbers
  + **Digital**
    - Digital computing: adding numbers on your fingers (digits)
    - Advantages
      * As much precision as desired
      * Good immunity to noise
      * Errors don’t accumulate after many calculations
      * All data types can be represented
    - Binary digits: Bits
      * Voltage is used to represent information in digital circuits
      * Standard voltages: 0 or 1
      * This unit of info is a Binary Digit (Bit)
      * Values of voltages don’t matter, distinguishing matters
      * Circuits are simpler and more reliable with just 2 voltages
    - Flip Flop
      * Basic digital circuit that can
        + take a data bit and store it
        + remember the value indefinitely
        + read out the stored value
      * The computer memory consists of a large number of flip flops
    - Bytes
      * A string of 8 bits
      * Represented in the computer by 8 copies of the basic storage circuit
      * Spaces in notation to break into groups of 4 bits to make it readable
      * Information capacity:
        + **=** 256 distinct values
        + Many different ways to utilise the info capacity
        + Each basic datatype (int, char) needs a representation method
    - Words
      * Short word (2 bytes), Word (4 bytes), Long word (8 bytes)
      * 64-bit architecture: means that hardware uses (mostly) 64-bit words
      * Generally, a larger word size yields faster performance
      * Info capacity:
        + k-bit words (2­­­­k distinct values)

## Lecture 2: Binary and Two’s Complement

Number representation:

* Integers
  + Non-negative integers use binary
  + Signed integers use two’s complement
    - Negative numbers included
* Reals
  + Approximate real numbers to floating point
* Other types (BCD numbers, fixed point fractional numbers, saturated numbers)

Binary:

* Binary rep uses a word of k bits to represent a non-negative integer between 0 and 2k-1
* Cannot be negative
* Base 2
* Convert from binary to decimal:
  + 10012 =
* Decimal to binary:
  + Know the word size k of the result
  + Check that x will fit in the word:
  + 203 to an 8-bit binary number
  + Iterative process:
    - Find highest power of 2 smaller than given decimal number
    - Get remainder
    - Find highest power of 2 for the remainder
    - Repeat
* Addition:
  + Sum bit, carry bit

Two’s complement:

* Not binary because binary can’t represent negative numbers
* A method for representing integers that can be negative or positive
* A k-bit word can represent 2k different values
  + Binary: 0 to 2k-1
  + Two’s complement: half of those values are negative, other half positive
    - The range is -2k-1 to 2k-1-1
* Sign bit
  + Leftmost bit of a two’s complement number
  + If it’s 1, negative
  + If it’s 0, positive
* Converting to/from decimal
  + If it’s non-negative, acts just like a binary number
  + If it’s negative, negate it and then convert from binary

Hexadecimal:

* Frequently used for values of words
* Base 16
* Decimal: hexadecimal
  + 0:0, 1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, 10:a, 11:b, 12:c, 13:d, 14:e, 15:f
* Adding also possible with sum bit, carry bit
* Hexadecimal with $ sign in front (for lectures)

# Week 2

Dr. Lito Machala

## Lecture 3: Logic Gates

Circuits:

* Bits are carried on wires (**signals**)

Logic gates: primitive components of digital circuits with input and output.

* 4 most basic logic gates: inv, and2, or2, xor2 (2 being number of inputs)
* Take a small number of input bits and produces a result according to a fixed truth table
* As long as the inputs remain stable, output is stable
* **Gate delay**: a small amount of time for a logic gate to produce a new output if the input changes
  + On the order 0.01 ns
  + Speed depends on technology

We’ll use only 1 flip flop: dff

Inverter:

* Output is the logical opposite of input
* x = inv a
* 

2-input and gate:

* Output is 1 only if all inputs are 1
* x = and2 a b
* 

2-input inclusive or gate:

* Output is 1 if any input is 1
* x = or2 a b
* 

2-input exclusive or gate:

* Output is 1 if only one input is 1
* x = xor2 a b
* 

Combinational circuits:

* Logic gates connected together
* Some inputs, some outputs
* No feedback loops
* Output depends on current input values -> the circuit has no memory

Why circuit simulation is important:

* Helps understand the program
* Helps find bugs

Circuit simulation:

* Inputs known
  + For abstract inputs, truth tables
* Outputs unknown
* Find outputs for each gate
* Find final outputs

Boolean algebra:

* Two values: 0,1
* Variables that stand for values
* Three operators:
  + x (not x): logical opposite (inv x)
  + (x and y): and2 x y
  + (x or y): or2 x y
* Laws:
  + 

Multiplexer:

* Fundamental circuit used to make decisions (if-else)
* Choosing between 2 values based on a third
  + inputs: x, y, c
  + if c = 0, output is x
  + if c = 1, output is y
* Every decision that a computer system makes ultimately comes down to a multiplexer

Logic gates and flip flops are simple, but the complexity arises by their connections: **emergent behaviour** (a system containing only simple components exhibits complex behaviour)

## Lecture 4: Boolean Algebra and Arithmetic

Boolean algebra is about operations on truth values (0 or 1)

Idempotence:

* An operation is idempotent if doing it several times is the same as doing it once

Commutativity:

* Swapping variables doesn’t change the operations

Associativity:

* Computing outputs for multiple inputs by using 2-input operations several times

Additional laws (not required to know):

* Distribution:
* Absorption:

Equational reasoning: substituting equals for equals using the laws of algebra.

Addition and subtraction:

* Adding 2 bits: half-Add
  + sum – xor2
  + carry – and2
* Adding 3 bits: full-Add
  + carry (z) as input
  + sum – x half-add y half-add z
  + carry – x half-add y or2 z
* Adding 2 integers: ripple-Add
  + A chain of full-Adds depending on size of words
    - 16 full-Adds if integers are 16-bit
* Subtracting with ripple-Add
  + Every bit inverted for negative number
  + Carry 1

Testing circuits:

1. Building them
2. Simulating them

Hardware designed using hardware description languages.

# Week 3

## Lecture 5: Synchronous Circuits

Gate delays: physical devices take time to produce outputs

Circuits need to have memory (**state**), but logic gates don’t have it

Delay flip flop (dff) stores 1 bit of memory (the state)

* Input x with a value, output y conveying state value

Combinational circuit: one that has only logic gates (no state)

Sequential circuit: one that has a memory

Hazards: unpredictable behaviour

The Clock:

* The dff component copies its input value into its state at specific points in time determined by a clock signal
* Behaviour:
  + Outputs state continuously
  + Executes state (inputs whenever a clock tick occurs)
* Just 1 clock signal, sent to all flip flops
  + Every dff updates its state at the same time
* The clock is generated externally by a circuit that produces a fast and steady sequence of clock ticks

A circuit is synchronous if:

* Every dff is connected directly to a unique global clock
* No logic functions are performed on the clock signal
* The circuit is designed so that every clock tick reaches every dff (almost) simultaneously
* Every feedback loop passes through a dff (no feedback in pure combinational logic)
* The inputs to the circuit are assumed to remain stable throughout an entire clock cycle

Clock ticks: points in time

* At a clock tick, the flip flops get new states which remain stable through the cycle, and the inputs get new values which remain stable

Clock cycles: intervals of time between clock ticks

* During the clock cycle, the combinational logic settles down, and eventually all the signals become valid

**The clock must be run slowly enough to ensure that all signals become valid**

Clock speed:

* no. of ticks per second = Hz
* Typical comps have 3 billion ticks per second (3GHz)
* Clock cycle is about 1/3 ns

Flip flop: remembers bit for only one clock cycle

Register: remembers a bit until it’s given a new value

1-bit register reg1:

* Remembers 1 bit
* Takes control input ld (load) and input x, outputs 1-bit state
* Initial value = 0
* At each clock tick, the register loads the data value x if ld == 1
  + If ld == 0, the register ignores the value of x and just retains its previous state
  + dff\_input = mux1 ld old\_state x
  + 
* Throughout the clock cycle, the register outputs its state value, which will not change (until perhaps the next tick)

Simulation tables:

* Each row – clock cycle
* Every signal of interest has a column
* 

Simultaneous update of state:

* State of a circuit: states of all dffs
* In a digital circuit, all dffs change state at the clock tick
* Parallel assignment:
  + All the RHS are evaluated at the same time
  + All variables are updated at the same time

## Lecture 6: Register Transfer Machine

Instruction: simple command the comp is able to execute

Register file:

* Array of registers (R0, R1, R2, …)
* Each variable is held into a register
* Variables are referred to by their addresses
* Register file circuit
  + Contains array of registers
  + Each register holds a word (binary number)
  + Enables to specify an address and read that register
  + Enables to specify another address and write to that register
* Behaviour
  + At clock tick: if ld = 1, then Reg[d] := x
  + During clock cycle: a = Reg[sa] and b = Reg[sb]
  + Control signals: ld, d, sa, sb
  + Input x, outputs a, b
* Loading into (writing to): demultiplexer
  + Selects which register to update
    - Creates control signals
  + ld: load value
  + d: address (**destination**)

Multiplexers:

* wmux1: uses control bit to choose between 2 input **words** (not bits)

Register Transfer Machine (RTM):

* Connecting adder and register file into a feedback loop
* if ld = 1 then reg[d] := reg[sa] +reg[sb]
* Circuit:
  + 
* Behaviour:
  + data input x
  + control inputs ld, arith
  + address inputs d, sa, sb
  + At clock tick
    - if ld
      * then if arith
        + then reg[d] := reg[sa] + reg[sb]
        + else reg[d] := x

# Week 4

## Lecture 7: Computer Architecture (CA), Machine Language

Ideas for a computer:

* Ability to perform a fixed sequence of arithmetic operations
* Ability to change that sequence
  + Examples:
    - Jacquard loom: used punch cards
    - Babbage’s Difference Engine: fixed sequences for nautical tables
* Ability to compare numbers and decide on next actions
  + Babbage’s Analytical Engine: first general Turing-complete comp

CA defines the structure (microarchitecture, digital circuits) and the machine language (ISA) of a computer

Machine language:

* Each machine has **one fixed** machine language
* Each programming language is **translated** to machine language by a **compiler**
* Examples:
  + x86 (Intel, AMD)
  + ARM
  + MIPS
  + Sparc
  + Sigma16
    - Not commercial, but a research machine
    - Not manufactured, but two implementations in software
      * Emulator
      * Simulator for circuit
    - Illustrates main ideas, avoids unnecessary complexity
      * One word size (16), while commercial machines have many
      * No backwards compatibility
      * Legacy architectures use complex instruction set (CISC), but Sigma16 uses reduced instruction set (**RISC**)
* Must be **powerful enough** to provide the foundation of Oss and programming languages, and must be **simple enough** for a digital circuit to be able to execute it
* Designed for high performance (but not to make programming easy)
* **Assembly language**: simplified notation of machine language
* Instructions
  + Analogous to statements in programming languages, but simpler

Computer: A digital circuit (piece of hardware/a machine) that executes programs

Structure of a comp

* Main subsystems
  + Register file
    - Set or array of registers
      * RTM has 4, sigma16 has 16 (R0,…,R15)
        + Can use hex
        + Each 16-bit register is 16 copies of the reg1 circuit
        + R0 = 0 (always), constant
        + R15: holds status info

Some instructions store additional info in R15 (negative result, overflow)

Temporary info

* + - * Register remembers a 16-bit word
    - Instructions:
      * “:=” – assign, not equals
      * General form:
        + op d,a,b

op: operation (+,-,/)

d: destination register

a: first operand register

b: second operand register

* + - * For interacting with memory:
        + General form:

op d, a[R0]

* + - * + load
        + store
        + lea (load
        + “trap R0, R0, R0” stops program
    - Limitation: small
      * Use **memory** (RAM)
    - Holds data temporarily
  + ALU (arithmetic and logic unit)
    - Does arithmetic (addition, subtraction, comparison, etc)
  + Memory
    - Slower and larger than register file
    - 65,536 memory locations (in sigma16)
    - Each memory location holds a 16-bit word
    - Each memory loc has an address 0, 1, …, 65,535
    - Can’t do arithmetic on a mem loc
    - Stores program variables permanently and the program itself
    - Instructions:
      * load
      * store
  + Input / Output (I/O)
    - Transfers data from the outside wold to/from the memory

## Lecture 8: Control Structures

First programmer: Ada Lovelace

* Programming language: Ada
* Realised that programs can be kept in memory (for Babbage’s Analytical Engine)

Compiling:

* Computers execute program only with machine language
* Compiling: translating high-level programming language to machine language
* Advantages
  + Computer able to run programs in different languages
  + Makes programming easier (error messages, etc)
  + Languages can be designed for specific tasks

Statements in high level languages:

* Performing calculations
  + Assignment statements
* Ordering calculations (**control structures**)
  + Conditionals (if-then-else)
  + Loops (while, repeat, for)
  + Structuring computation (functions, procedures, coroutines, etc)

High level control structures:

* Notation
  + S, S1, S2, etc means “any statement”
  + bexp means “any Boolean expression” (e.g. x<3)
* Block: treats consecutive statements as a single statement
  + {S1; S2; S3}
* if-then: if bexp then S;
* if-then-else: if bexp then S1 else S2;
* while loop: while bexp do S
* etc

Low level constructs:

* Assignment statements: x := a \* 2
* Goto: goto computeTotal
* Conditional: if x < y then goto loop
* First translate high level to low level, then to assembly language

Goto:

S;

loop: S; ; “loop” is a label

S;

S;

goto loop;

Using goto:

* Not usually in high level
* Low level uses it for higher level control statements
* Inconditional: goto L
* Conditional: if b goto L

Machine language: Jumping:

* Machine language equivalent of goto
* unconditional: jump loop[R0] (means “goto loop”)
* jumpf: jump if false
* jumpt: jump if true
  + jumpt R5, label1[R0]: checks R5 for true (not 0)

Comparison instruction: Boolean form

* cmplt R2,R5,R8
  + “compare for Less than”
  + 0 (False) or 1 (True) for R5 < R8 stored in R2
* cmpeq: compare for Equals
* cmpgt: compare for Greater Than

Translating “if bexp then S”

* if x<y
  + then {statement 1;}
* statement 2;

Low level:

* R7 := (x < y)
* jumpf R7, skip[R0]
* instructions for statement 1
* skip: instructions for statement 2

Programming techniques:

* Statement-by-statement style
  + Each statement is compiled independently
  + load, arithmetic, store
  + Straightforward, but not efficient
* Register-variable style
  + Keep variables in registers across a group of statements
  + Don't need as many loads and stores
  + More efficient
  + You have to keep track of whether variables are in memory or a register
  + Use comments to show register usage
  + Real compilers use this style

# Week 5

## Lecture 9: The Stored Program Computer

Memory contains the machine language program

Sigma16 has 3 instruction formats:

* RRR: uses registers (add, sub, trap, …)
  + Requirements: 4 fields (op (14 operations, enough with 4-bit word), d, a, b), each 4 bits (total 16 bits): one word
* RX: uses memory (lea, load, store, jump)
  + Two words
* EXP: uses registers and constant
  + Two words

Assembler: translates assembly language to machine language

Assembler vs. compiler:

* Compiler translates very different languages
* Assembler translates similar languages

Assembler allocating memory:

* Location counter: variable containing address for next line of code
* Symbol table

Control registers:

* PC register (program counter): address of the next instruction
* IR (instruction register): instruction being executed at the moment
* If RX instruction, ADR (address register): memory address of the second operand

## Lecture 10: Arrays

Address arithmetic:

* Addresses represented in binary (unlike integers in two’s complement)
* Powerful data structures
  + Arrays, pointers, records, linked lists, queues, stacks, trees, graphs, hash tables
* Powerful control structures
  + I/O, Procedures and functions, Recursion, case dispatch, coroutines, classes, methods

Data structures

* Container containing many individual elements

Arrays

* Sequences of indexed values
* In programming languages, arbitrary element notation is x[i], array is x
* Ubiquitous: used in all kinds of applications
* Representation
  + Represented in memory by placing the elements in consecutive memory locations
  + The address of the array is the address of the first element x[0]
  + Address of x[i] = x[0] + i
* Allocation in Sigma16
  + x data 13 ; x[0] = 13  
    data 189 ; x[1] = 189  
    data 5 ; x[2] = 5
* In large scale software, big arrays are allocated dynamically with help from the OS
  + Using “trap” to request a block of mem
* Effective address
  + Accessing array element with register with index variable: load R2, x[R8] (if R8 contains i, then it’s the same as x[i])
  + Addressing modes
    - Scheme for specifying the address of data
    - Sigma16 has 1 addressing mode: displacement[index]
    - Older comps have multiple addressing modes (one for each data structure), but it’s more efficient to use 2/3 flexible modes
* Traversal
  + For loops designed specifically for array traversal (in first programming language, Fortran)
  + “do the body for every element of the array”
    - High level:  
      for i := 0 to n-1 do  
       {x[i] := x[i] \* 2}
    - Low level:  
      i := exp1  
      loop: if i > exp2 then go to loopdone;  
      statements  
      i := i + 1  
      goto loop  
      loopdone: …

# Week 6

## Lecture 11: Records and Pointers

Compilation patterns

* Useful to debug in high-level, easier to read, scalable
* While loop
  + High level:  
    while bexp do  
     S
  + Low level:  
    label1  
     if bexp = False then goto label2  
     S  
     goto label1  
    label2
* Only small efficiency by violating pattern, but unreadable

Since goto was found to be controversial/harmful, structured programming was created

Records:

* A record contains several fields
  + x, y = {  
     fieldA: int,  
     fieldB: int,  
     fieldC: int}
  + Access: x.fieldA
* Basically a dictionary in Python

Pointers:

* A pointer is an address
* Helpful
  + Reusable code

Requests to the OS

## Lecture 12: Procedures and the Call Stack

Procedures: reusable code

* Subroutine/function
* A block of code stored not in the main flow of instructions with a label and a jump out

Call and return:

* Jump-and-link: jal
  + jal R5,label[R0]
  + Saves pointer to next instruction after returning in destination register (in this case R5)
  + Usually R13 used
  + Procedure ends in  
    jump 0[R13]
* Passing parameters
  + Different conventions
  + Function: procedure with parameters
    - Pure function: doesn’t change global variables
  + One convention: argument and result are passed in R1

Save state:

* Maintain stack in memory, push data onto stack
* Call stack: all of the stack kept in memory
* Stack frame: every “activation” of a procedure, the instance in a stack call

# Week 7

## Lecture 13: Variables

Variables:

* Scope
  + which parts of the source code can access the variable
* Don’t correspond to data statements
* Initialisation
* Lifetime
  + creation, destruction
* Location
  + address in memory

Types

* Static (global)
  + Visible through the entire program
  + Lifetime: entire execution of program
    - Created when program is launched, exist throughout
  + Can declare variables to be static in higher level programming languages
  + Disadvantages:
    - Names have to be unique
    - When multiple users want to use the program, instructions are duplicated in memory -> inefficient use of memory
      * Modern OSs organise info in segments
        + Code segment: read-only, shareable
        + Data segment: read/write, non-shareable
* Local (automatic)
  + Visible only in a local procedure
  + Defined locally (in a function, procedure, method, different blocks)
  + Kept in stack frames
  + load R1,x[R14] ; R14 points to the stack frame
* Dynamic (heap)
  + Used in object-oriented and functional languages
  + Objects: “new” in Java
  + The heap
    - Pointers to blocks of memory in the language’s “runtime system”

Stacking procedures:

* Each stack frame contains a pointer to the previous one

## Lecture 14: Linked Lists, Memory Management

Pointer (&x): lea R1,x[R0]

Value of pointed address (\*x): load R1, 0[R1]

Linked lists:

* Consists of a linear chain of nodes
  + Node: record with two fields:
    - First: value
    - Second: next node (address)
  + Last node’s second field is *nil* (0)

Operations on lists:

* Check if empty
* Check value of node
  + x := \*p.value
    - load R1,p[R0] ; R1 := p  
      load R2,**0**[R1] ; R2 := \*p.value  
      store R2,x[R0] ; x := \*p.value
* Check next node
  + q := \*p.next
    - load R1,p[R0] ; R1 := p  
      load R2,**1**[R1] ; R2 := \*p.value  
      store R2,q[R0] ; q := \*p.value

List traversal:

* Trailing pointer whenever there is a need to insert before a certain element

List header: the first node containing a pointer to next, but no value

Deleting a node: previous node’s next is node after “deleted”, “deleted” is available

* If deleted node is not made available, **memory leak**

Memory management:

* Garbage collection
  + Traverses all data structures, notes unused memory, makes it available
* If multiple linked lists use parts of the same memory, modifying one will modify the other
  + Side effect

Arrays vs linked lists:

* Direct access
  + easy for arrays
  + inefficient for linked lists
* Traversal
  + Arrays: 0 to n (for loop)
  + Linked lists: start until empty (while loop)
* Memory usage
  + Arrays: memory for one element for each (n words)
  + Linked lists: value and pointer to next (2n words)
* Flexibility
  + Arrays: fixed size, allocated fully
  + Linked lists: variable size

# Week 8

## Lecture 15: Programming Tchnqs, Arrays, Pointers

Compound Boolean expressions

* and &&
  + cmp
  + jumpf
  + cmp
  + jumpf
* or ||
  + cmp
  + jumpt
  + cmp
  + jumpt
* “Short circuit” an expression
  + Stop later conditions if they depend on earlier conditions

Comparing

* Register-saving
  + cmplt R3,R2,R4 (R3 := …)
  + jumpt R3,label[R0]
* Conditional jumps
  + cmp R2,R4
  + jumplt label[R0]
    - jumplt, jumpeq, jumple (less than or equal), jumpge

Loops:

* Repeat until: does the body first, then checks condition (do-while in Java)

I/O:

* A character is represented by a code using ASCII or Unicode
* digit chars 0-9 have codes (dec) 48-57
* lowercase a-z 97-122
* uppercase A-Z 65-90
* To print a number, one needs to convert it to a string
* Converting a number to a string
  + Repeatedly dividing 10 to get remainder as a digit

Arrays and pointers:

* Access array element
  + lea R1,x[R0]
    - pointer to first element
  + load R2,0[R1]
    - To access current one, follow pointer
  + lea R1,1[R1]
    - Move to next element (increment p)

Records and pointers:

* two pointers to start and end
* while start pointer is smaller than end pointer
  + access by the value in the addresses

Stack Overflow

* Trying to put more stack frames than there is memory allocated
* To protect from this
  + R11 holds stack limit
  + R12 holds stack top (highest address in current stack frame)

## Lecture 16: Nested Conditionals

Block (compound statement) – is a single statement that contains several statements

* Syntax: indentation/start-end/braces

Jump tables: a set of commands correlated to a set of addresses

* A technique for implementing case statements
* jt[0] = jump S0[R0]  
  jt[1] = jump S1[R0]  
  …
  + Jumps to a procedure/block of code
* Implementation:
  + lea R5,CmdJumpTable[R0] ; pointer to start of table  
    add R4,R5,R4 ; address of command/case in table depending on R4  
    jump 0[R4] ; jump to operation

Trapdoor: flaw in sys that is inserted purposely

# Week 9

## Lecture 17: Assessed Exercise, Trees

Assessed exercise: 10%

Quizzes (best 8 of 10): 10%

Assessed exercise:

* Objective: fill missing pieces in program
* Concepts
  + Array of records
    - Representing a command as a record
      * Case statement, jump table
    - Traversing an array of records
  + Ordered linked lists
    - Traversal for printing elements
    - Insertion in sorted list
    - Deleting
    - Searching
* Coverage: how much of a program is tested, the variety of tests

Binary trees:

* Each node has 3 nodes: value, left pointer, right pointer
* Root: first node without any value (value is nil)
* Leaf: node without any pointers (both are nil)

Searching:

* Arrays, linked lists with linear search – O(n)
* Trees with binary search – O(log(n))

## Lecture 18: Interrupts

Interrupt: automatic jump going either to OS or to an error handler

* Causes:
  + Error in program (stack overflow, result too large to store in a register)
  + A trap: explicit jump to OS, doesn’t specify address
  + An external event (disk drive, timer)
* Behaviour
  + Implements saving a state: savepc := pc
* Useful for:
  + Catching errors
    - Jumping to error handler
    - Two approaches:
      * Explicit error checking
        + Problems:

Longer program

Less efficient

Fragile (possible to forget to put everywhere)

* + - * Interrupts
        + Faster, easier, shorter
  + Requesting service from OS (trap)
  + Providing quick service to an I/O device
    - e.g. computer mouse causing interrupts
  + Implementing concurrent processes
    - OS giving time-slices to processes in round-robin order
* In programming languages:
  + Exceptions

Processes:

* Process: a running program
* Concurrent:
  + Happens by interrupting each waiting process and switching to another process (**process break**)
    - The running process jumps to OS kernel (core of OS that has a table of all processes)
* The Scheduler:
  + The core of an OS maintaining a list of all processes
  + Behaviour
    - Interrupt happens, OS takes action, OS jumps to scheduler, scheduler sets a timer and jumps to a chosen process

Implementation of interrupts:

* Implemented by control circuit (in the CPU) with a control algorithm