What do we do?

- Reading and writing assembly language programs for a simple microcomputer
- Identify hardware components and the functions they perform
- Overall behaviour of compilers, assemblers, linkers, and interpreters
- Trade-offs between hardware and software in computer system design

Information and Computers

- Information are things we need to know, convey, and analyse
- In computers it is representation, manipulation, and storage
- Binary information – On and Off
- Bit notation – 0 and 1
- Multiple bit representation
  - Numbers (32 bits) → Double, float, int
  - Characters (8 bits) → ASCII chars
- Computer only sees bits and doesn't know what it is for
- **Byte**
  - Smallest addressable block of 8 bits
- **Word**
  - Size of a register used by the CPU. Could be 8, 16, 32 or 64 bits
- Use hexadecimal notation as a shorthand

What drives evolution

- National security
- Commercial imperative – Replace people to do things faster and cheaper
- Related technologies – Need to interface with other systems
- Gaming, AI, GPU, Big Data
- **Moore's Law**
  - Computing power doubles every year

Analogue vs Digital

- Analogue – Data represented on a continuous scale
  - Voltage, height, distance
  - Good for real world measurements
- Digital – Represented in whole numbers
  - Class size, coins in my pocket
  - Good for noise rejection and computers
  - Advantages
    - Easy to modify and design
    - Maintains accuracy
    - Protects from noise
  - Disadvantages
    - Real world values are analogue
    - Translation from analogue to digital loses data
    - More bits more precise
  - Data stored in computers are in binary 0 and 1
    - Might be represented in voltage change
    - Absence or presence of something

        - Binary good cuz simple and robust
- Groups of bits can represent numbers, characters, and instructions
- How we interpret a 32-bit word depends on what we expect
- The hardware is there to manipulate groups of bits. Gates are used to do this

Number Systems

- Positional number system
- In decimal, the further left a digit is, the greater power of 10 it is multiplied by

Hexadecimal

- Another abbreviation for binary numbers
- Each byte represented by two hex digits, each hex digit represented in 4 bits

Bit-wise operations

- Can base decisions on a single bit
- Unary operation – NOT
- Binary operation – AND, NAND, OR, NOR, XOR
- N-Bit operations 4 input AND, OR

GATE

- Electronic component whose output is determined by input
- NOT (!)– Flips the bit from 0 to 1 or 1 to 0
- OR (|)– Adds both the bits. As long as one bit is 1 it will be 1. Basically +
- AND (&)– Multiplication. Only produces 1 when both bits are 1
- NAND (!&)– Opposite of AND. Always 1 until both bits are 1. Combination of AND and NOT.
- XOR ^– Only says 1 if there's only one 1

Combinational Circuit

- A connected arrangement of logic gates with the set inputs and outputs
- Can be described by a truth table
- Half-Adder and Full-Adder
    - Half Adder
        - Arithmetic addition of two bits is called a half adder
        - Two inputs get summed
        - Two outputs get summed and carried
    - Full adder
        - Three inputs and two outputs
        - The two inputs are the same and the extra third is the carry from the previous position
        - Can be made with two half adders
        - Multibit addition – Combining 8 adders with carry to get 8-bit addition
- Serve as a basic building block for the construction of more complicated circuits

Binary addition

- Watch out for the carry
- 1 + 1 = 1(this is carried) 0

Full adder

- First half is the same as a half adder. The sum goes into a new XOR and carry calculation with the carry input.
- Can use two XOR or XNOR gates to create a full adder also
- Use it to add many digits together so we have to use multiple full adders

Adding numbers

- Always starts with a half adder.
- 8 Bit would be 1 half + 7 full
- If at the end it has a carry, that indicates a number that goes beyond 8 bit or an overflow

Understanding gates

- Look at the truth tables and see patterns.
- Gates are programmable
    - XOR used as a controlled inverter
        - If C = 0, A = Out
        - If C = 1, A = !Out
    - AND used as a logic controlled switch
        - If A = 0, Out = 0
        - If A = 1, only can AB make Out = 1
        - Enabling and disabling flow of data. B can only flow and make Out = 1 if A is set

Gates are not instantaneous…so…clocks

- Used with an ALU which needs two clocks out of phase
- Input 1, Input 2, Clock, Control bits → ALU → Result + Carry out
- It is used to synchronized things to ensure predictability and avoid ill-defined states

CPU

- A computers output is determined by the new inputs plus the memory of the old inputs. But how does it remember?
- Storing 1 bit
    - Memory is built by storing 1 bit units
    - We need to create these circuits which we set what to remember before the power goes out
    - RS Flip Flop is a cyclic circuit that loops in the same outputs and inputs
    - Flip flops with clocked inputs more used mainly D and JK
    - D
        - Only one input
        - Q is the same as D and is on when the clock goes active
        - So Q doesn't change when D changes but when the clock changes
        - Two AND gates are used, One input is the clock one is the data. A not gate is used to ensure that the inputs for the AND gates are never the same to avoid the illegal state
    - JK
        - Two inputs
        - It is updated when the clock goes active but it is determinant on the clock

        - Works like the RS flip flop without the illegal state
        - If both are up, it'll turn on and off. Both down no charge
        - Uses tri input Nand gate
    - T
        - Input just toggles output so on off repeatedly
    - Asynchronous (non-clocked) inputs act as an over ride
    - Create registers with d flipflops because most intuitive

Registers

- Forms of memory
- Uses clocked flip flops, inside CPU chip, limited number of them
- Allows us to store bits of strings

Using D flip flop as register

- A bunch of d flip flops driven by a common clock
- The transfer from the input to output occurs simultaneously as the clock changes
- Must have a synchronized clock

Technical definition for the order of the bits

- Big endian – The most significant byte resides in the smallest address
- Little endian – The least significant byte resides in the smallest memory address

Bit Endianness

- Bits generally don't have an address so we refer to them by the position
- Big endian – 123
- Little endian – 321
- Both above are the same

Counters in digital logic

- Circuit that stores and increment occurrences (Clock pulses)
- Asynchronous (Ripple)
    - Called so because changes are not simultaneous
    - Ripple counters use the toggle of JK
    - Each additional ff is 2^n
    - What is if I want to count down?
        - Change to rising edge instead of falling edge
    - Modulo 6 counter
        - What if you want to kill it at 6?
        - Use and gate to detect 6
- Synchronous (Common clock)
    - Used to prevent illegal states
    - Add a d flip flop to delay the output by one count

Shift registers

- Takes input from one end and moves it to the next ff on clock pulse
- Used in serial data transfer when data sent on a cable one bit after another can be collected in a series of D flipflops to rebuild the whole data
- Bi directional?
- Parallel to serial

Types of memory

- **ROM (Read only memory)** – All reading at full speed. Get address, go there for quick access. BIOS has this. Cannot be easily accessed or written over.
    - **PROM** – Programmable ROM
    - **EPROM** – Erasable PROM. Can be removed from computer and erased or programmable using special hardware
    - **EEPROM** – Electrically EPROM
- **RAM** – Accessing based on addresses rather than sequential. Should be RWM because both RAM and ROM allow random access
    - **Static RAM** – Retains info until power is removed. Chunky system. Modest power needed
    - **Dynamic RAM** – Retains info as long as contents are refreshed frequently. Smaller area of silicon. Low power requirement
        - No flipflops but capacitors
- **SDRAM** – Synchronous DRAM. Clocked by CPU. It's a hybrid of static and dynamic
- **DDR SDRAM** – Double data rate SDRAM. Chips produce data on both rising and falling edge. Higher power requirement. Higher heat output.
- **FRAM** – Uses atom position in the unit cell. Non-volatile so no power to hold state.
- **Flash Memory** – Charge stored between insulators. Written by injecting electrons through barrier layer. USB drives
- **Core memory** – Magnetic. Stores each bit as the direction of magnetisation

Memory addressing

- RAM consists of more than one chip
- Ram is organized into words like like 32 bits or 64 bits
- Words are grouped into pages
- Words are selected by an address
- Control bits specify whether to read or write
- Need bits to address individual bytes

Stack

- Basically, need to hibernate a process at will when interrupted. It's when you need to do smtg else first then come back to it later
- RAM requires knowing the address of every word you want to access
- Stacks offer a way of organising and accessing memory without random access
    - Hardware – Creating by dedicated shift registers
    - Software – typically defined in RAM using conventions
        - Use RAM to do this so only limited to amount of ram.
- **What it does**
    - Push data we need later onto the stack to free registers
    - Pop the stored data back off the stack

Neumann

- Most common CPU architecture. Control/instructions bits and data bits share a common memory space.
- Allows processes to be interrupted while higher priority tasks are executed

Harvard

- Instructions and data are kept separate
- Runs faster and more secure but more expensive
- Used in PIC controllers and digital signal processors where the memory is scare and speed is important
- Reasonably immune to buffer overflow attacks
- Modified?
    - Neumann but with separate instruction and data caches in the CPU
    - Has the speed of Harvard and flexibility of Neumann. Used in ARM, most Intel CPU

CPU Caches

- Stores frequently accessed instructions or data in high-speed memory
- Caching algorithms determine what's stored
- Can have separate data and instructions to be read in parallel

Interrupts

- A response to a signal that needs attention from the software
- Stacks allow interrupt-based hardware access
    - Device issues an electrical signal which is fed into a priority encoder which then issues an INT signal to the CPU
    - Current work is pushed onto a stack
    - CPU loads the interrupt handlers routine executes the code
    - INT handler ends with a RETurn instruction which pops the sored IP off the stack
- I/O INTS
    - Each hardware event is mapped to an interrupt number. Highest priority is lowest number
    - Define an interrupt vector table
    - Set up an interrupt mask
- Using hardware signals to run code
    - Logic is needed to process interrupts
    - Important interrupts like mouse, keyboard, power need more priority
- Types
    - Clock – issues INT when time reaches 0
    - Keyboard or Mouse
    - Error
    - Network – Packet receive
    - Exception –
    - SysCall – triggers kernel or system command
    - Hardware – Power button, CD ROM

Polling

- Alternative to interrupts
- Checking state of hardware in a predefined sequence
    - Process any change as needed
- Issues
    - Waste time checking hardware doing nothing
    - Doesn't take advantage of stack

    - One freeze will cause unresponsive system
- Benefits
    - Easy to implement

Encoders and decoders

- Binary signals along a wire can represent a single state, or form part of a binary code
- Encode – Used to code binary data
    - Converts an active input into a coded output signal
    - Number of inputs cannot exceed number of m n <= 2*m
    - Used for interrupts like reading from a keyboard. Rather than 104 wires we can map to less
- Decode – Decode binary items
    - N input lines can max 2n output lines
    - But why?
        - Decoding memory addresses. Addresses are binary so we take that binary and decode it
        - Displays are like that
- Multiplexers
    - Similar to encoders but it's really focused on many inputs to one
- De multiplexers
    - Similar to decoders but only accept one input

Signed integers

- Computers only store 1 and 0. They only perform arithmetic on integers and real numbers.
- Signed = Negative
- Sign magnitude. Most significant bit used to represent positive or negative
- Trade offs
    - Simple and intuitive
    - Easy to implement
    - Reduces value. One bit loss, half values gone
    - How is 0 represented?
- 2's compliment
    - Solves the issue above.
    - Shift the range of possible values so that 0 is in the middle of the range
    - Compliment positive binary and add 1
    - Trade offs
        - Memory efficient
        - Retains the most significant bit property
        - Zero is unambiguous
        - Slightly more complex transformation'

Sign extension

- Smaller bit value you want to store in the larger bit. 8 bit stored in 32-bit storage
- Take the sign value and repeat it till the end of the sign

Number system

- Integers are easier to work with than real numbers (Whole and fractional part)
- We try to reduce floating point stuff. Real numbers are infinitely precise
- Binary?
    - Binary uses base 2. Decimal point represented by two symbols, 0 and 1. Use powers of 2

    - Representing it in a computer
        - Whole part and fractional part in fixed point representation. Some number of bits represented in the whole part some represented in the fraction part
    - Binary Coded Decimal
        - Each digit is represented in its 4 bit binary representation (Nibble)
        - Can make arithmetic algorithms simple
        - Used in basic calculators
        - Lots of wastage tho
- Floating point
    - Uses a variable bit for the fraction
    - Supports trade off between value range and precision
    - IEEE 754 standard
    - First bit is sign. Next eight is exponent. Next 23 is fraction. (32 bit number)

What is Data?

- Information or knowledge
- Can be measured, collected and analysed
- Can be visualized

What is communication

- The act of conveying meaning from one group to another through mutually understood signs

What is Data communication

- The process of using computing and communication technologies to transfer data from one place to another and vice versa
- **Modes?**
    - Simplex – Data travels in one direction only
    - Half-Duplex – Data travels in one or the other direction but not at the same time
    - Full-Duplex – Data travels in both directions at the same time
- **Why?**
    - Because computers are useless without it
    - Computers need to communicate with all our other devices both receive and send
- **How?**
    - Data can be sent
        - One at a time (serial) – More used. More universal
            - USB – Universal Serial Bus
                - Uses 4 lines. D-/D+ for data each mirror each other. 5v and 0v
                - Possible for systems to have a common serial communication line. Ethernet is an example. Multidrop serial connection.
            - RS-232 – Has nine pins but the three to be concerned with are.
                - Sends data in single-byte packets asynchronously (Not same clock) So there needs to be agreed rate and standard. Sends one bit at a time. Uses little endian.
                - Baud rate – How many bits/second is sent
                - TxD – Transmit pin
                - RxD – Receive pin
                - SG – Ground/0
                - Parity Bit – Can be set to low or high depending on the data sent. It is used to counter electrical interference. It identifies single bit errors.
                    - Last bit sent
                    - The number of logic 1 bits sent must be an even number, not including the start bit but adding the parity bit
                    - Receiver can check data parity and see if it is correct
            - Why?
                - Good over short and long distances. Less interference and cheaper
            - Why not?

            - Slow as transfer is one bit at a time
- Multiple at a time (parallel) – Less used. More application specific.
    - Multiple physical lines so bigger ports
    - Needs handshake protocol
    - Why?
        - Fast, whole data sent at once
    - Why not?
        - Only good over short distances.
        - Expensive and bulky

Binary language

- Low level as it is as close to the hardware as you can get

Assembly Language programming

- Lowest level of human readable programming
- The assembler translates code into binary machine instructions.
- This level and below, you are dependant on the CPU. It is not interchangeable.

Why?

- Write small programs really fast
- No operating systems to get in the way
- Write compilers
- Write drivers for custom hardware
- Find vulnerabilities in code

RISC vs CISC

- RISC – Reduced Instruction Set
    - Emphasis on software
    - Single clock
    - Large code sizes
    - One cycle
    - Less transistors used for memory registers
    - Lower power
    - Used for low powered contexts
- CISC – Complex Instruction Set
    - Emphasis on hardware
    - Multi-Clock
    - Small code size
    - Variable cycle times
    - More transistors used for storing complex instructions
    - Higher power
    - Used for multimedia applications

ARM Assembly Programming

- Most widely used instruction set
- Consists of about 100 instructions in total
- RISC based

Working with 32 bit

- 32 bit wide registers and addresses

Bare metal ASM programming

- No OS

- Direct access to hardware registers and memory addresses
- Essentially just us and the BIOS

Move

- mov
- move value into register
- mov r1,#1
    - r1 – Register 1
    - #1 – referring to the decimal value 1. We want to move it into the first register
    - Loading register 1 with the value one

OR

- orr
- Same like logic gate or
- orr r1, $21
    - r1 – Register
    - $21 – Hex value. Its an operand
    - R1 = r1 OR operand (literal)5

Logical Shift Left

- Lsl
- Lsl r1, #21
    - R1 – Register 1. Could be the destination.
    - #21 – Decimal value
    - Shift r1 21 values to the left

Store

- str
- store/write value from register to a memory location
- str r1,[r0,#16]
    - r1 – Register 1
    - […] – Everything inside the brackets is an address in a memory. Reference pointer.
    - Store the value of r1 in r0 after 16 bytes

Loop

- a label

Branch

- b
- A go to function
- B loop
    - B – Branch
    - Loop – Where to go. Can be anything

Load Register

- Ldr
- Read from memory

- Ldr r0, [r1]
    - R0 – register 0
    - [r1] = Pointer to a value in memory
    - Load register 0 with the value pointed to by r1

General Purpose Registers

- Armv8 has 31 64 bit registers

Separate Base and GPIO offset

Steps

- Identify the GPIO associated with the pin
- Find the associated function and set appropriate bits to indicate your intention to write to the GPIO
- Find the appropriate write register and set the output bit to 1

Why loop?

- PC just keeps going until it is turned off.
- Infinite loop keeps pc from accessing locations that are not part of the program

Making the LED Flash

Steps

- Enable
- Wait
- Disable
- Wait
- Repeat

Configuring pin On or Off

- Store location of GPIO in r0 (BASE + GPIOADDR)
- Enable writing for GPIO
- Set output of GPIO
- Write 1 or 0
- Loop forever

A Dumb Timer

- Busy wait: a loop that keeps executing until it reaches a certain value. Keeps the CPU occupied before executing the next task
- Set limit value
- Initialize counter to 0
- Timer$:
    - Add 1 to counter
    - Compare counter to limit
    - If less than limit, add 1
- Variables
    - R0 = GPIO Base Address
    - R1 = Working Memory
    - R2 = Used for timing
- Mov r2, $3F0000
- Loop1:
- Sub r2,#1
- Cmp r2, #0
- Bne loop 1

IF tests

- Called compare or CMP
- cmp r2,#1234
- Subtracts the second value from the first and sets flags accordingly
- Loads the Application Program Status Register with the results of the comparison
    - N – Negative. The 2nd value is bigger than the 1st
    - Z – Zero. The 2nd value is equal to the 1st value
    - C – Carry.
    - V – Overflow.

Acting on the APSR

- Branch or B, reads the APSR and jumps accordingly to the flag with the relevant suffix
    - Basically, a conditional go to
- Assume cmp r1,r2
    - B – unconditional branch
    - Beq – Branch if the Z flag is set
    - Bge – Branch if r1 >= r2
    - Blt – Branch if r1 < r2
    - And many more (Look at the comparison suffix table)

Issues

- The APSR is updated after many ALU operations, not just cmp!
    - It is not always cleared after it is read
    - Always treat a cmp and branch as one operation and there mut be no code between them

A better timer

- RPi has a dedicated timer register
    - Independent of clock speed
    - Housed inside the same chip as most other components
    - It's the SoC
- Timer registers start at BASE address + 0x3000
    - Timer counts 1 microsecond intervals
- Byte offset from base that we are concerned with 0x3004. Just reading from this
- Pseudocode
    - Store base address of timer
    - Store delay
    - Mov start_time(=now)
    - Loop
        - Read now
        - Remaining_time = (now-start_time)
        - Compare remaining_time, delay
        - Loop if remaining_time <= delay
- Variables
    - R3- Base address of timer
    - R4 – Desired delay
    - R5 – Start time
    - R6 and R7 – Where the current time will be stored
        - Two registers because timer is 64 bits so we need two bits of 32. They have to be next to each other. Last bit of r6 moves into the first bit of r7
    - R8 – Elapsed time = R6-R5
- Load register
    - Ldr r0(destination), [r1] (memory address)
    - Loads r0 with address of r1
    - Ldrd r6,r7,[r3,#4]

        - D means double
        - The low bits will go into r6 and the high bits will go into r7

MOV command

- Doesn’t fit all numbers and only fits certain conditions
- Mov code is really fast. Does it all in one clock cycle
- **Management?**
    - 32-bit mov has actually only 8 bits available to be moved
    - 20 bits for op-code(all instructions have an op-code)
    - 4 bits for the ROR
    - Done using a shift register called a barrel shift register
- **Implication**
    - Mov can only handle 8 but its not as limiting as it seems
    - It can be combined with other instructions such as orr

Functions

- A callable block of organised and re-usable code
- Typically, single action
- Accepts arguments. Accepts and input, performs a function, gives output.
- Functions can call other functions and itself (recursion)
- **Basics**
    - **Called**
        - Arguments need to be placed somewhere the functions can access
        - Program control shifts to the function's instructions
    - **Complete**
        - Return value needs to be placed somewhere for the calling function to retrieve
        - Program control shifts back to the instruction immediately after where it was called from
- In ASM?
    - Not native to assembly. We have to do the management ourselves
    - Think about how to pass arguments from one to another
    - Each function we call will want to use the same registers. We need to manage this
    - Jumping from one function to another and back
- **Register management**
    - **Application Binary Interface**
        - Sets the standard way of using ARM registers
        - R0-R3 used for function arguments and return values
        - R4-R12 promised not to be altered by functions
        - Lr and sp used for stack management
        - Pc is the next instruction. We can use it to exit a function call
        - **Calling**
            - First two function arguments are loaded into r0 and r1
            - Next two are put into r2 and r3
            - Return value is written into r0 and r1
            - Promises not to use r4 – r12

The Software Stack

- ARM computers have a software stack
- A separate area of RAM is available for temporary values managed by the SP
- A value in a register can be pushed onto the tack to preserve it for later
- It can be popped off later (LIFO)
- We can get the memory location by checking the SP register
- A 32-bit array which starts high and grows down. It decrements the SP by 4 bytes
- POP just increments by 4 bytes
- **Syntax**
    - Can put as many as you want but order matters
    - Push {…}
    - Pop {…}
    - With RPi we need to initialise the stack pointer before doing push and pops so
    - Mov sp,$1000 (First 4096 bytes of ram is for the stack)
- Passing the arguments to functions

- Reusing registers
    - Back up registers we need to re-use in a function
    - Store arguments for the functions in r0-r3
    - Call the function
    - Read the return values from r0-r1
    - Restore the registers we backed up

Key Registers

- Program counter (pc or r15)
    - Holds the address of the next instruction to execute
    - Is updated when a branch to label is encountered
- Link Register (lr or r14)
    - Holds the address of instruction to return to after a function is complete
    - Holds what was in the pc register before it was changed
    - Brings us back to where we came from

Helpful Instructions

- bl label$:
    - Causes program control to jump to label but also copies next instruction to lr so we know how to get back
- bx lr
    - Branch exchange. Essentially a return to where you're from or wtv address was stored in lr

Refactoring

- Breaking up code using the stuff learnt

Includes

- Put each function in a dedicated source file
- Include will combine then with your main.asm and assemble it as one source file
- Include performs a text substitution
- Put at the bottom of asm file

Recursion

- Functions that call themselves
- When a function calls itself, a new copy of the function is needed
- Each copy operates with its own variables
- Works because of a stack and its virtually unlimited. It will hold temporary values and play them back in the right order
- We program an algorithm to keep going until it reaches a determined base case

Arrays

- More work in ASM arrays
- Requires
    - An address in memory for the first element
    - A known constant offset to the next element
    - An uninterrupted contiguous block of reserved space

- Start with a label
- Follow with the data type and then list the array elements separated by,

Reading Binary Input

- We can read the state of a GPIO pin by programming the GPIO for input
  - Loop:
    - Applying a voltage to the appropriate header pin
    - Read GPIOs
    - Test the state of the appropriate bit
    - Branch if the bit is set or not
  - We need to use bic (bit clear)
  - Bic r1,r1,#7
    - 7 because binary for 7 is 111, the first three which we want to clear