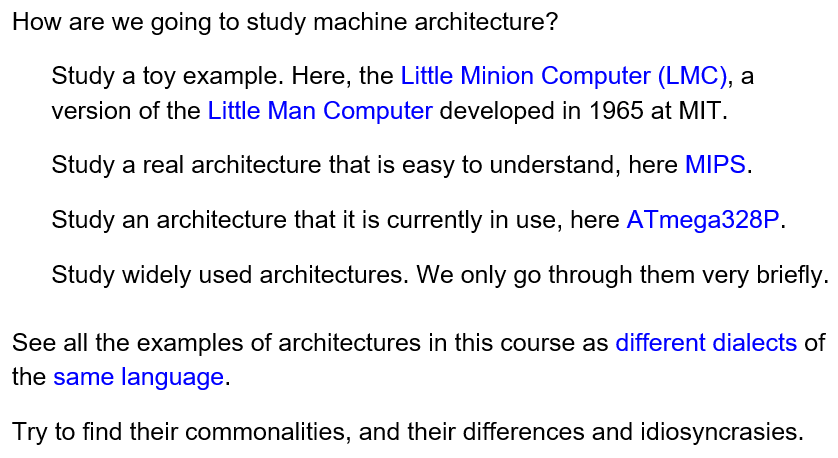
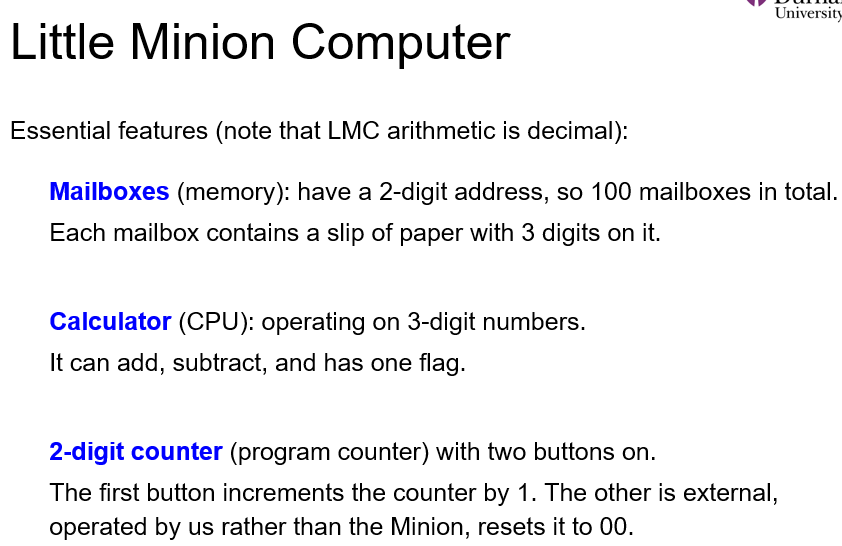
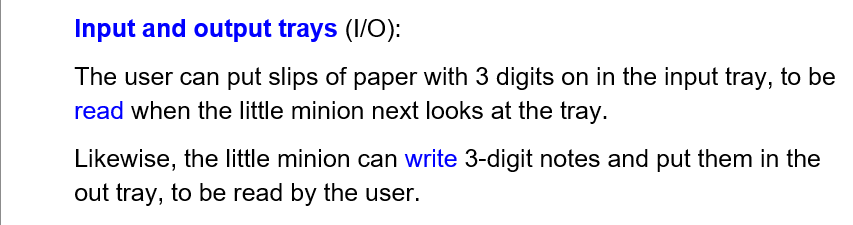
**Machine Architecture**

Computer architecture refers to attributes of a system that are visible to a programmer, i.e. have a direct impact on the logical execution of programs. This usually refers to the Instruction Set architecture (ISA), which is the definition of instructions, and the impact these instuctions have on registers and memory during execution. This is contrasting with computer organisation, i.e. the physical construction of the circuits, which is not visible to the programmer.



How it works:

The minion starts by looking at the program for a mailbox address xx, increases the counter by 1, and goes to mailbox number xx. It reads the 3 digit number at the mailbox and takes the appropriate action based on these digits, and starts again. (fetch-execute cycle)

The 3 digit decimal numbers stored in the mailbox represent instructions. The first digit is the opcode, and the second and third digits represent a mailbox address. For example for number 584, 5 is the opcode and 84 is the mailbox address.

Load: opcode 5 (instruction 5xx)

* Go to mailbox xx, read the number stored there and enter it into the calculator.

Store: opcode 3 (instruction 3xx)

* Go to calculator, read the 3 digit number displayed and store it in mailbox xx

Add: opcode 1

* Go to mailbox xx, read the number and add it to the current value stored in the calculator

Subtract: opcode 2

* Go to mailbox xx, read the number and go to the calculator and subtract this number from the number already stored there. If the result is negative, raise the NEG flag on the calculator to indicate this. NEG flag is not the same as a negative sign, is just an indicator of whether or not the previous subtraction yielded a negative result.

Input/Read: opcode 901

* Go to IN tray, read the 3 digit number there and enter it into the calculator.

Output/Print: opcode 902

* Go to the calculator, read the 3 digit number there and enter it into the OUT tray.

Break: opcode 000

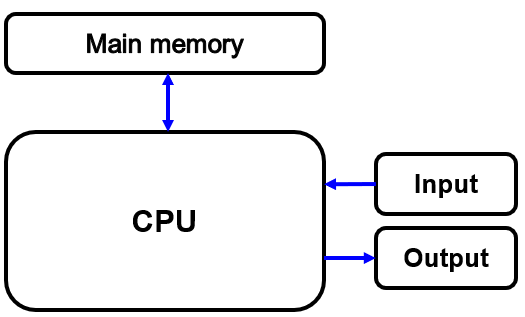
* The minion has a rest.

In LMC, all action passes through the calculator. To output a number stored in a particular mailbox, first pass it to the calculator then output the content of the calculator. To add two numbers, first load one to the calculator, then add the second.

LMC does not support multiplication and division – have to use repeated addition/subtraction.

Von Neumann Architecture

* A high level description of computing devices developed by EDVAC for the US army in 1945-51. The principal elements of computers are still virtually unchanged.

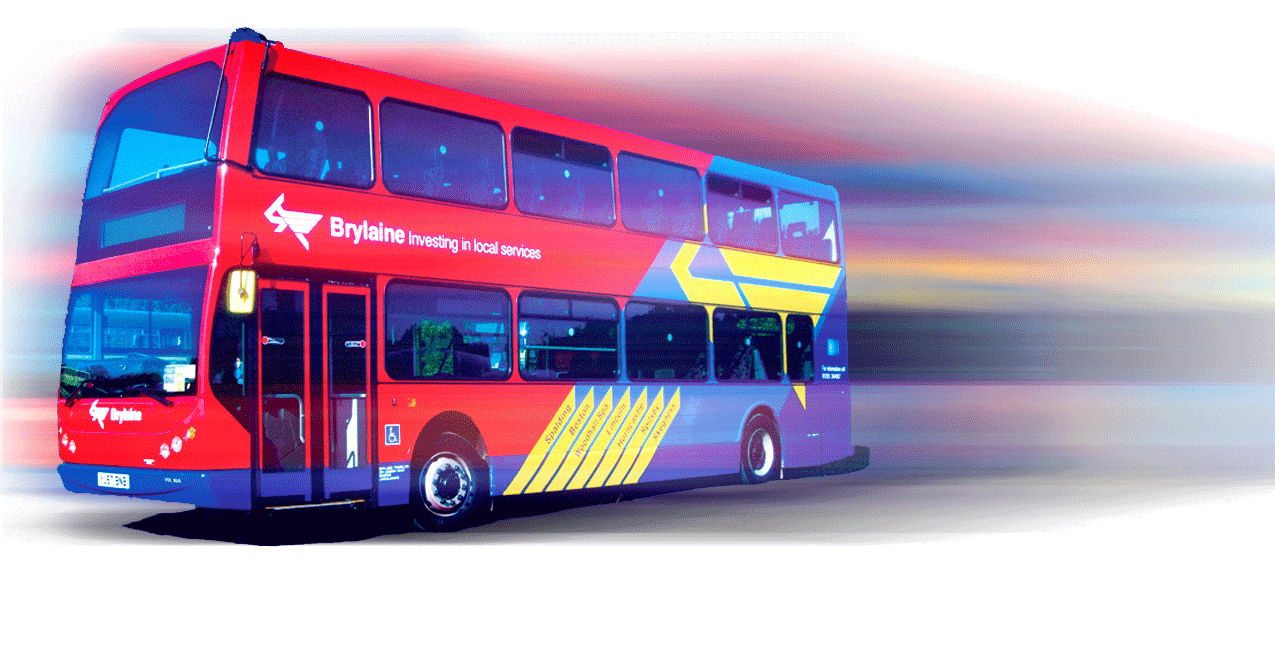
^very high level depiction.

The CPU contains the Control Unit (CU) and Arithmetic Logic Unit (ALU).

CU: responsible for directing the flow of instructions and data.

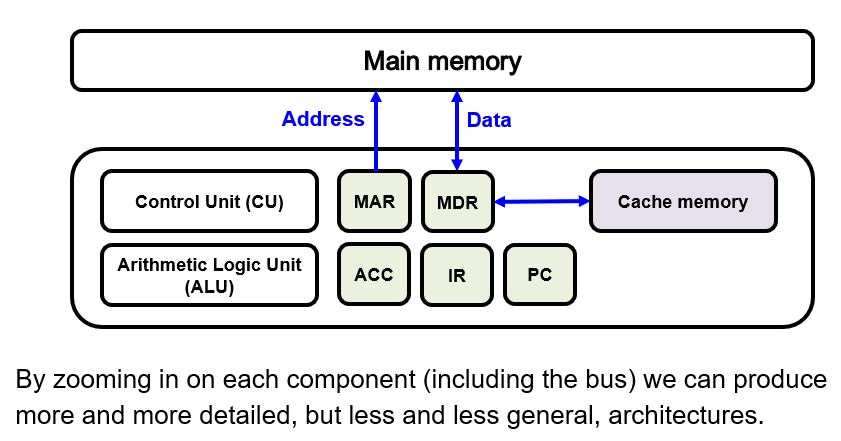
ALU: responsible for performing operations on the data.

We also have main memory, and input/output devices. Communication between components happens through buses.



There are architectures that are not von-Neumann, e.g the Harvard architecture, which uses separate memories for program instructions and data, and two separate buses.

A system is only von Neumann architecture if data and programs are transferred along the same bus. If there are multiple buses for instructions and for data then it is not von Neumann.

Cache memory – part of the CPU that provides fast-access memory (compared to fetching from main memory.)

Registers are special memory locations that can be accessed very fast.

On the diagram: Accumulator (ACC), Program Counter (PC), Instruction Register (IR), Memory Address Register (MAR), Memory Data Register (MDR).

Registers can hold a value temporarily for storage, manipulation or calculation, and can be manipulated directly by the Control Unit. Their size can range from 1 to 128 bits.

Accumulators are often a part of the ALU. These are general purpose registers used for holding data, e.g. interim and final results of calculations, or data waiting to be transferred between memory and I/O, or between memory locations.

The Program Counter, Instruction Register and Flags are often seen as part of the Control Unit.

The PC holds the address of the next instruction to be executed.

The Instruction Register holds the actual instruction being executed.

Flags are 1-bit registers that keep track of certain conditions, such as carry, overflows and errors. Flags are grouped in one or more Status Registers (SR).

The MAR holds the address of a memory location to be accessed. It connects to the memory by a unidirectional bus.

The MDR (aka MBR; B = Buffer) holds the value that is being stored to or retrieved from the memory location currently addressed by the MAR. It connects to memory via a bidirectional bus.

Buses:

A bus is a physical connection that makes it possible to transfer data from one location to another in a system. A bus takes the form of a group of electrical conductors which carry signals. Categories: Data, Address, Control and Power.

Buses are used to transfer data between different points on the CPU, between CPU and memory or between peripherals and the CPU.

Point to point buses carry signals from a specific source to a specific destination, while broadcast buses carry signals to many different destinations.

Bus Interface Bridges allow communications between the different buses (e.g. external bus, USB, PCI bus)

MIPS – Microprocessor without Interlocked Pipeline Stages

* Designed in 1980s and Stanford, powered workstations until the 90s. Still used in embedded systems and consumer electronics.

Design principles:

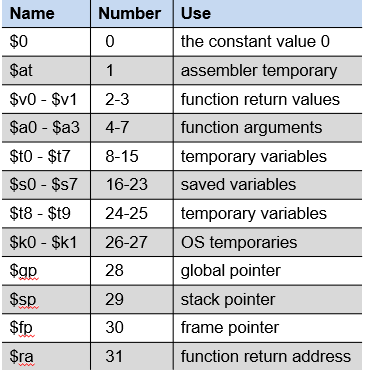
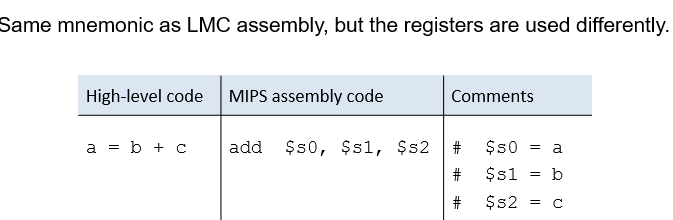
* Simplicity favours regularity
* Make the common case fast
* Smaller is faster
* Good design = good compromises.

MIPS – 32-bit RISC processor:

* 32 bit words
* ~110 instructions in the instruction set
* 32 general purpose registers.

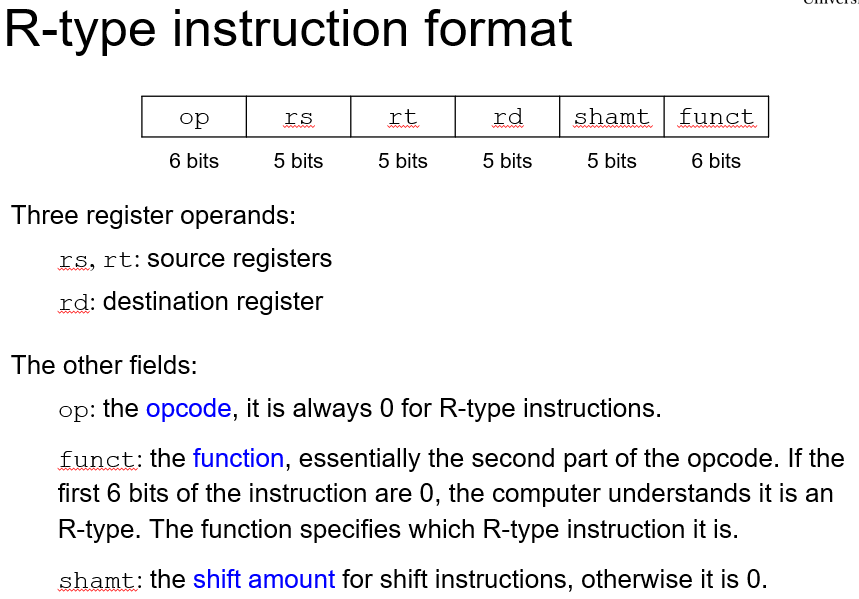
RISC -> reduced instruction set computer

MIPS Registers:

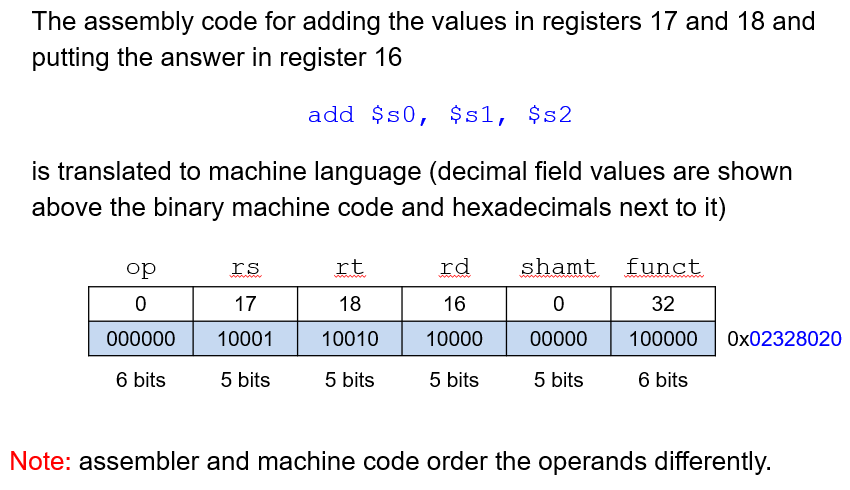
Instruction types:

* R-type with register operands
* I-type with immediate operands
* J-type for jumping.

R-Type instruction format (in machine code):

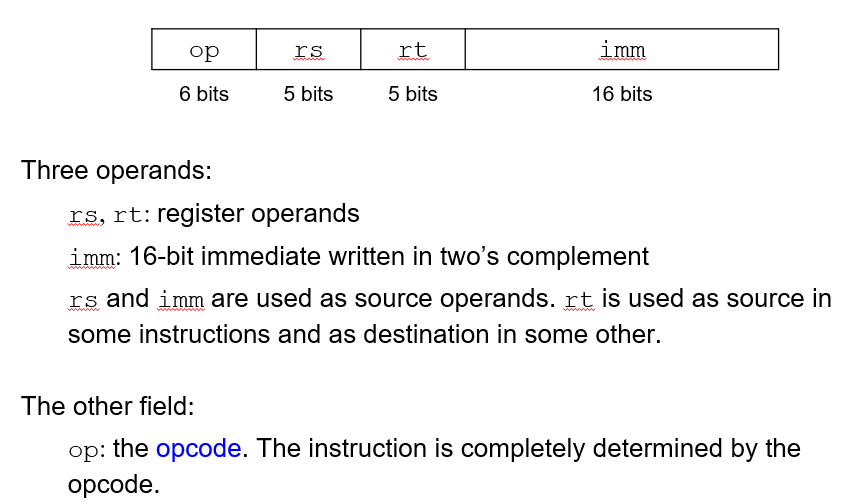
e.g. for adding, the values stored in rs and rt will be added together, and the result will be stored in rd.

“op” is sort of like the addressing mode so the CPU knows what sort of instruction it is.

“funct” determines exactly what the instruction is (for R-type).

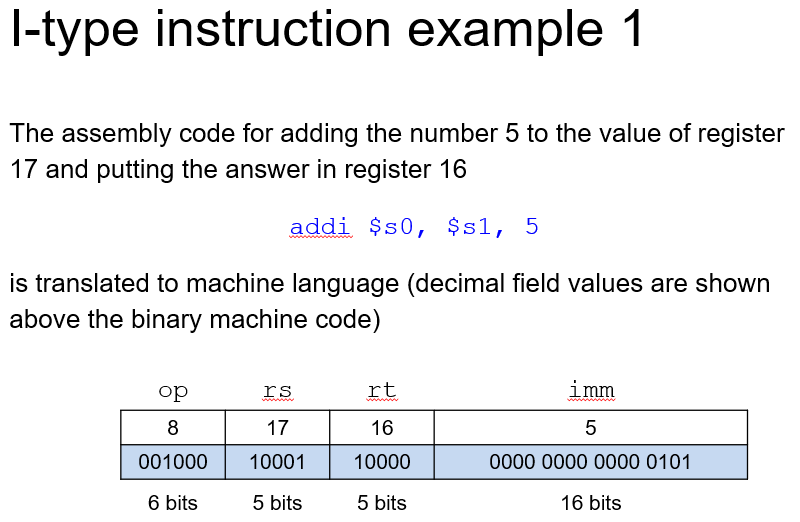
(assembly puts the destination register first)

I-type instruction format (machine code)

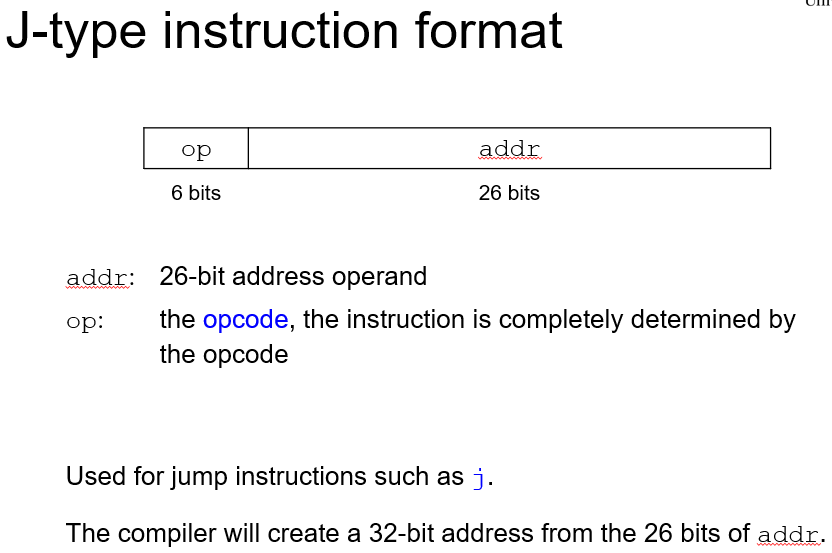
(for these, op is never 0; op determines the type of I-type instruction to be executed (similarly to ‘funct’ in R-type)

Imm is just a number 😊

Example:

(addi is the I-type version of the add instruction, for adding a number to a register rather than adding the values of 2 registers).

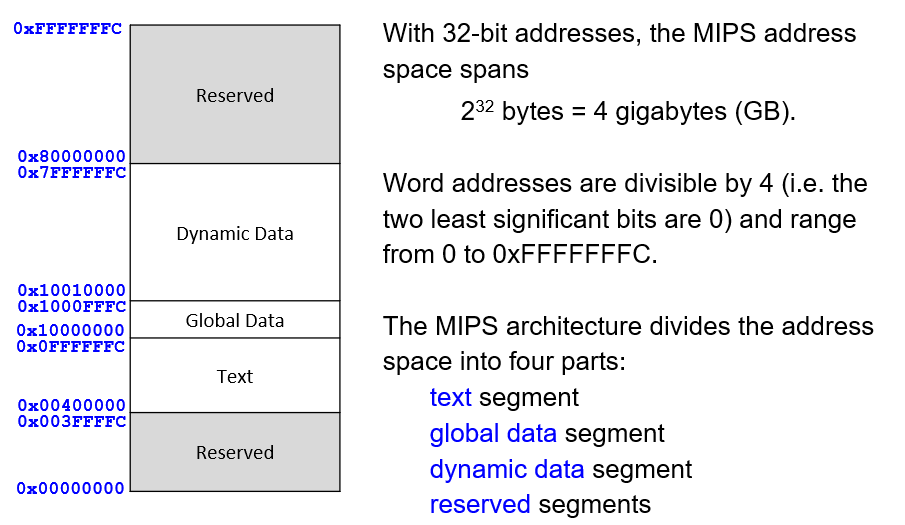
J-type instruction format:

MIPS memory map

Word – 32 bits

Byte – 8 bits

MIPS uses 32 bit memory addresses which point to a specific byte. This means word addresses are always divisible by 4.

^(not to scale)

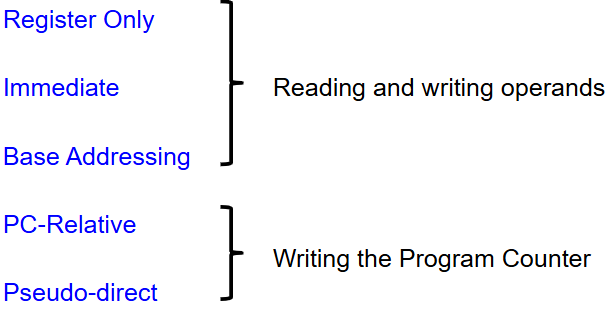
The text segment stores the machine code for the program, storing almost 256MB of code. The 4 most significant bits of any word in this segment are all 0, as are the last 2 (since divisible by 4). This is why 26 bits is enough for the “adder” field in a J-type instruction to reference a word corresponding to the instruction to branch to.

The global data segment stores global variables – size 64KB. Global variables are accessed via the pointer $gp. By convention, $gp is initialised at the middle of the global data segment, with global variables referenced as their offset from this value (to save bits as the maximum offset is reduced by having the pointer in the middle of the memory space the global variables are stored).

The dynamic data segment stores data which is dynamically allocated and deallocated during program execution – stores 2GB. Data is stored in a stack and a heap.

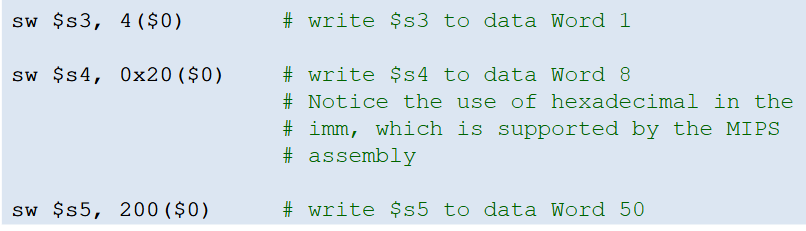
The reserved segments are used by the operating system and cannot be accessed or used by the programs running on MIPS.

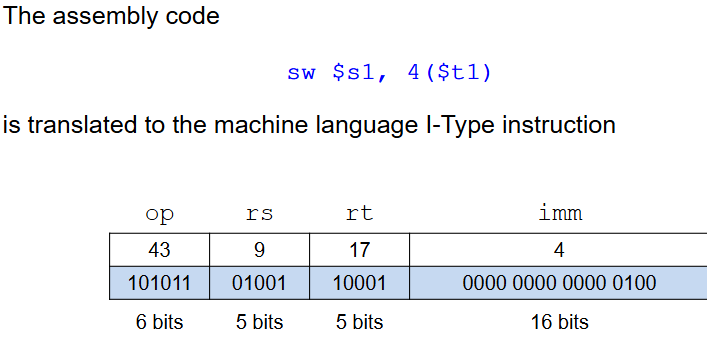
MIPS Addressing Modes

Register-only: uses registers for all source and destination operands. R-Type instructions use this.

Immediate: uses registers and 16 bit immediate as operands. Used by some I-type instructions.

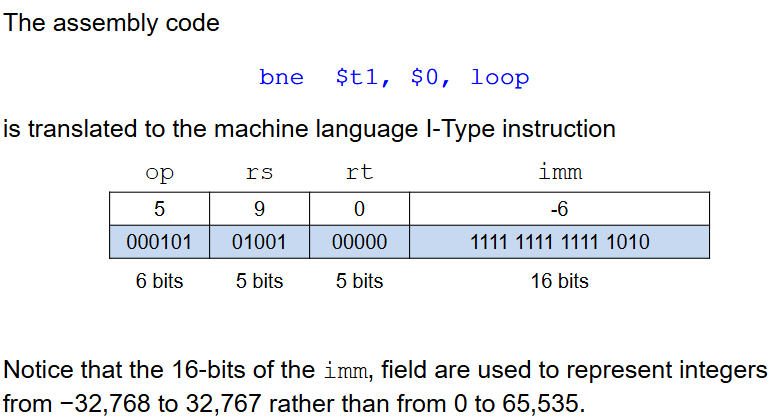
Base addressing: used in memory access instructions; implemented by I-type. Address of memory operand computed by adding the base address stored in register rs to a 16-bit offset stored in the immediate passed to the instruction.

^a word is 4 bytes but each memory location references a specific byte -> this is why word 1 is referenced as location 4.

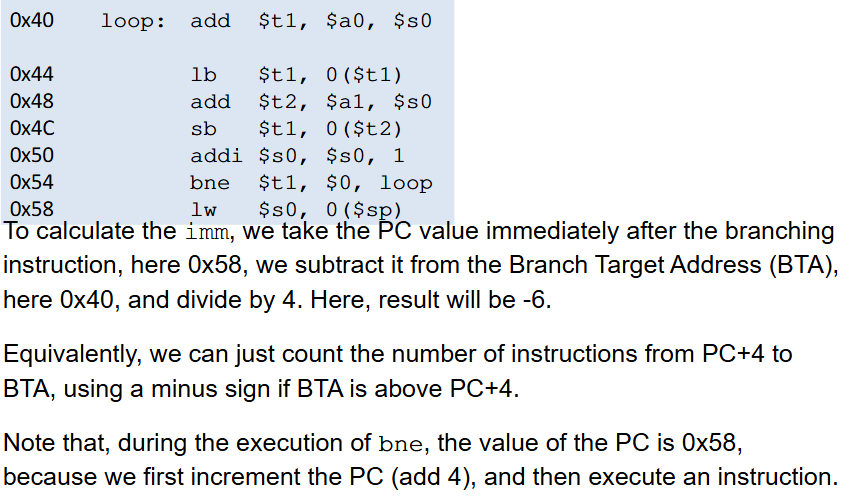
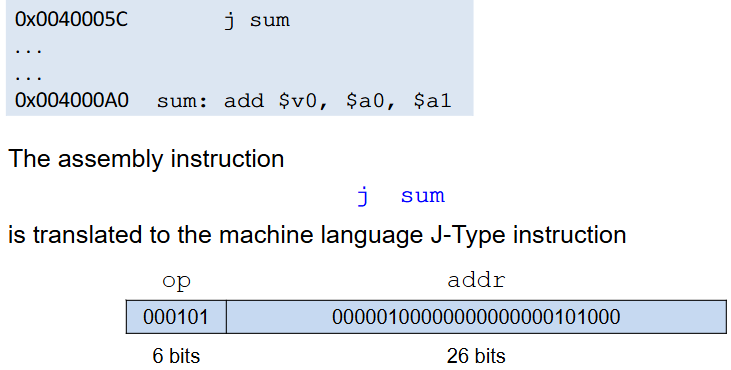
($t1 = 9, $s1 = 17 – assembly and machine code do things in a different order for some reason)

($t1 + 4 is the memory address the value in $s1 is stored into. (perhaps?) )

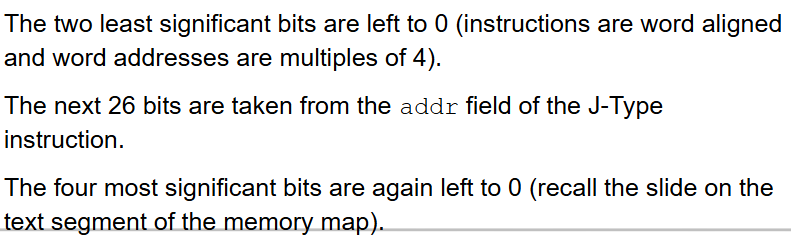
PC-relative addressing: Branching instructions can use this to specify the new value of the PC if the branch is taken.

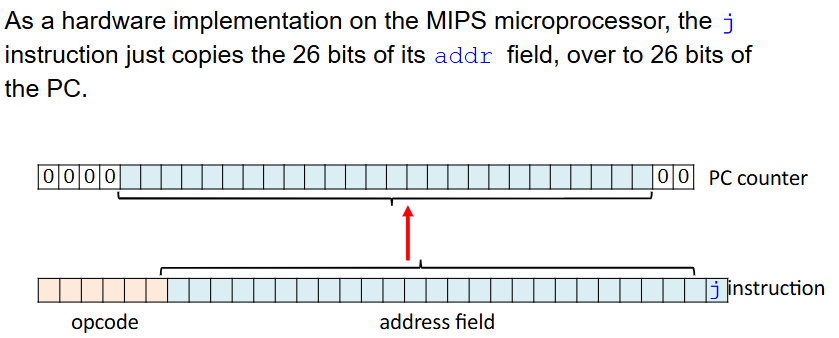
^this instruction causes 6 to be subtracted from the current value stored in the PC. Imm is twos complement.

(bne => branches if not equal to)

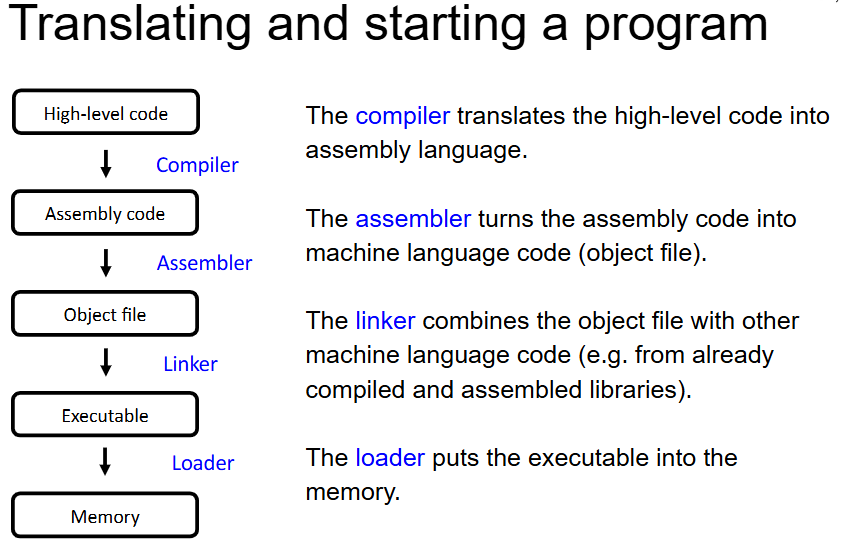
Pseudo-direct addressing: An address is supplied in the instruction. MIPS does not support direct addressing, as this would need 32 bits for the address and 6 bits for the opcode, whereas MIPS instructions are only 32 bits in total. As a result, MIPS uses pseudo-direct addressing in J-type instructions, calculating the new value of the PC by:

^example





MIPS Programming

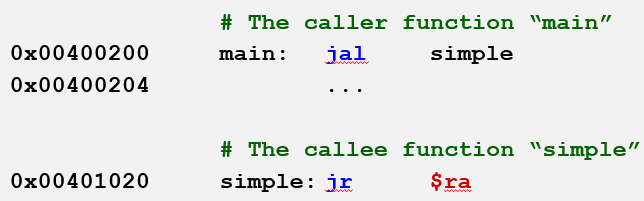
The assembler makes two passes through the assembly code and turns it into the object file. On the first pass, the assembler assigns instruction addresses and finds all symbols, such as labels and global variable names, and makes a symbol table.

MIPS Programming

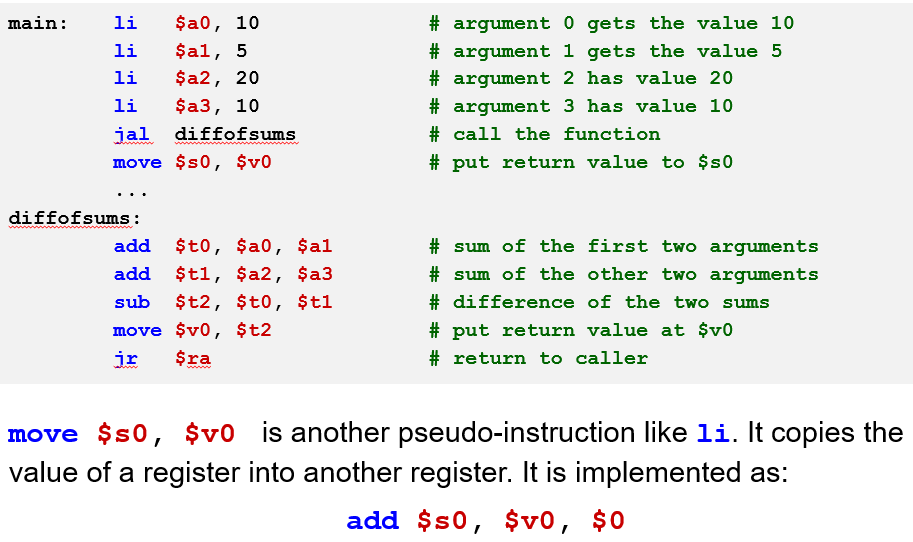
Functions

* MIPS assembly functions have inputs (arguments) and an output (return value).
* Use “jal” instruction to call a function – this function jumps to the specified address and stores the return address in $ra. “jr $ra” jumps back to this return address at the end of a function.

e.g:

(note – jr is an R-type instruction)

e.g. 2:

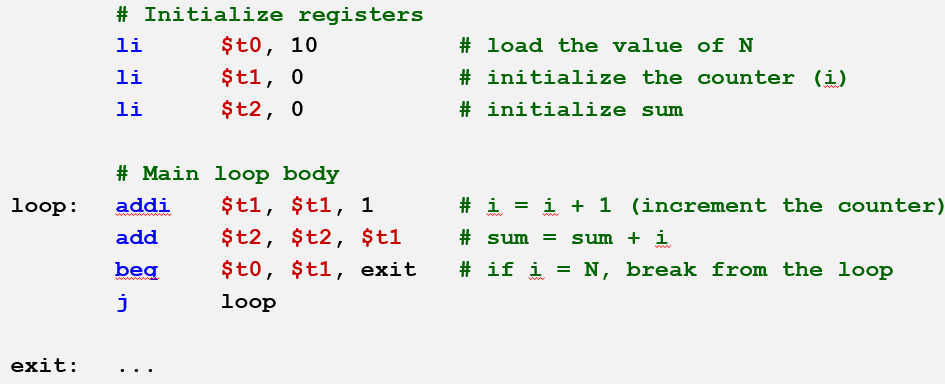
^the proper convention is to use the $a registers for arguments and $v registers for return values. It is also convention for functions not to modify $s registers, only the temporary $t ones.

For recursive functions, or functions calling other functions, important registers such as return addresses can be saved to a stack.

Loops

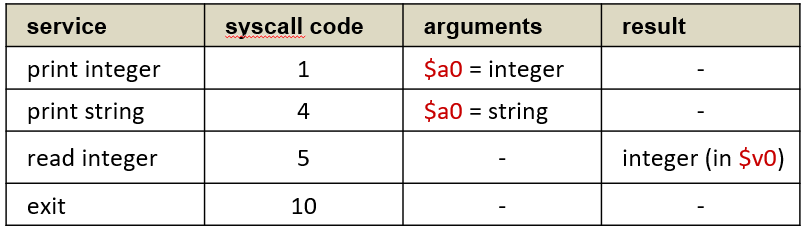
* Can easily be implemented using branch instructions, e.g. bne, beq, to test for conditions to break out of the loop, and a jump to the start of the loop if the condition is not met

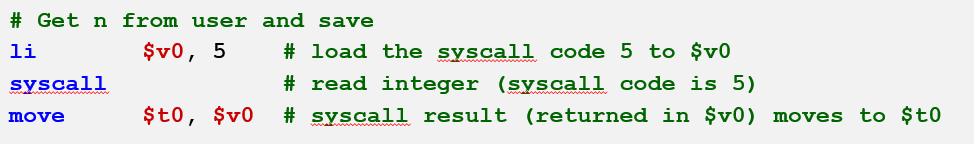
e.g.



Input/Output

* Syscall suspends program execution to provide operating-system-like servie, i.e. input, output, termination.
* The type of syscall service is specified by a code, stored in $v0

e.g.

It is important to ensure the program exits properly, otherwise it will fetch the word stored after the last instruction with unpredictable results (similar to LMC interpreting data mailboxes as instructions).

ATmega328P – Arduino microprocessor

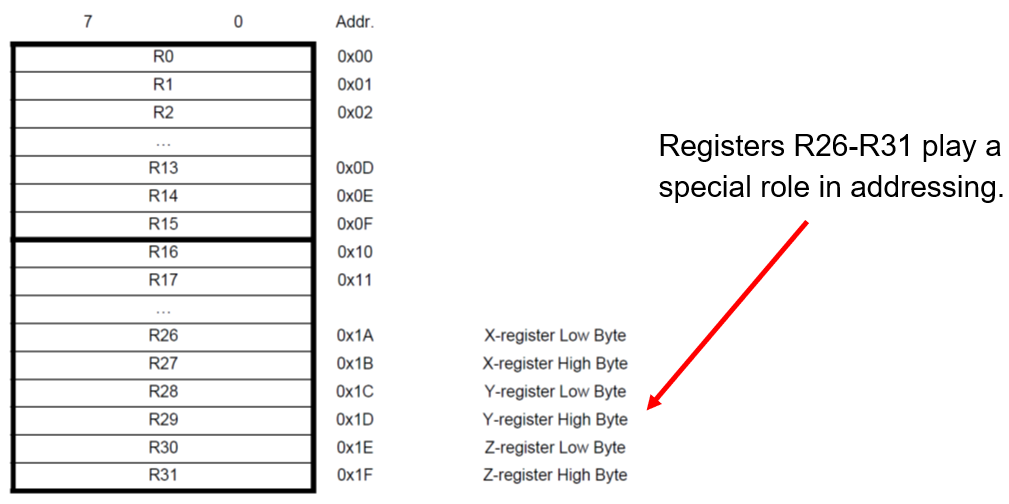
Microcontrollers are integrated devices (computers in one chip) consisting of a microprocessor, memory and I/O. They are used in embedded systems, and often use flash memory to store the programs. The program may never be changed during the lifetime of the microcontroller.

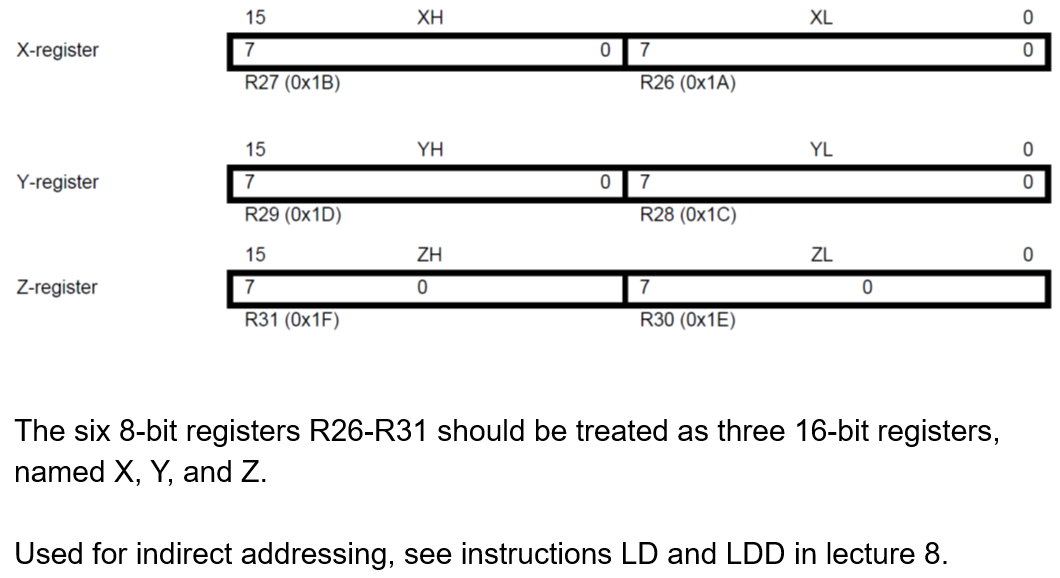
Arduino-UNO uses the ATmega328P microcontroller:

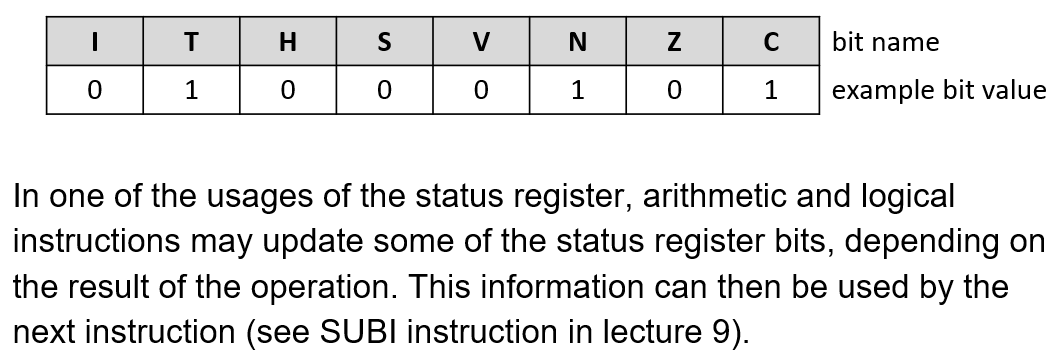
* Harvard architecture
* 8 bit processor, registers, bus
* 16 bit words (some instructions 32 bit).

Registers:

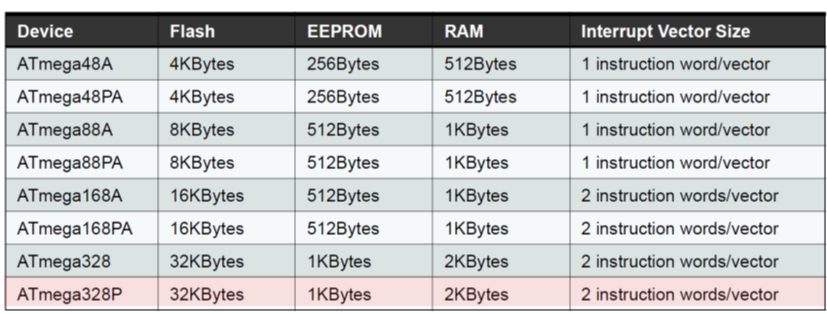
* 32 general purpose 8 bit registers, also copied onto SRAM memory
* Other memory locations can also be used as registers.

^Registers R26 and R27 are used together as the X-register to store a single 16 bit value. Same with 28-29 and 30-31.

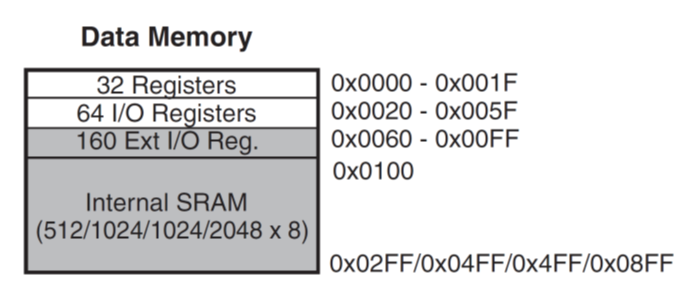
The Status Register is another register (not one of the 32), also 8 bits. These 8 bits are flags.

Memory Map:

Since Harvard architecture, we have 2 memory spaces: Data Memory (SRAM and EEPROM) and Program Memory (Flash).

^we are looking at the bottom one

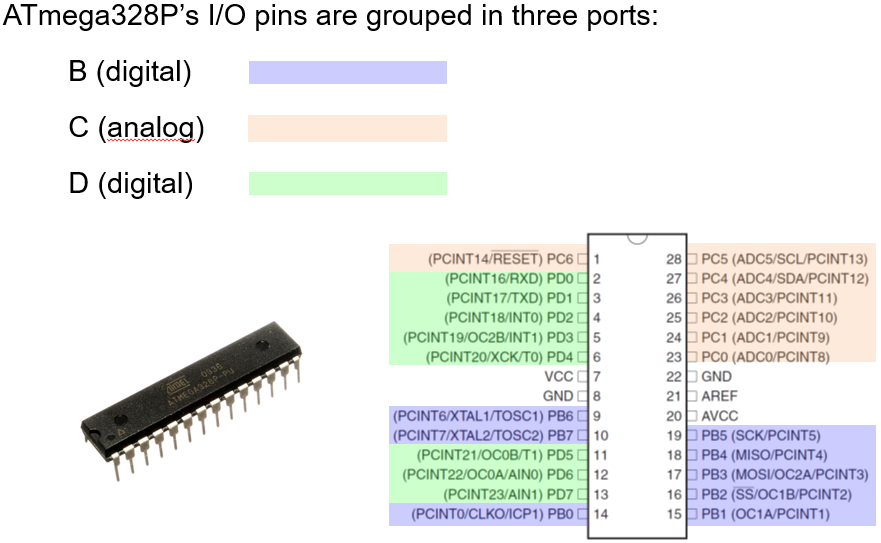
Since instructions are 16 or 32 bits, the flash memory is organised as 2/4/8/16 K \* 16 bits. This flash memory is divided into Bootloader and Application Program sections. Arduino flash memory has an endurance of at least 10000 write/erase cycles. The Program counter is 14 bits, in order for it to address the 16K addressable program memory locations.



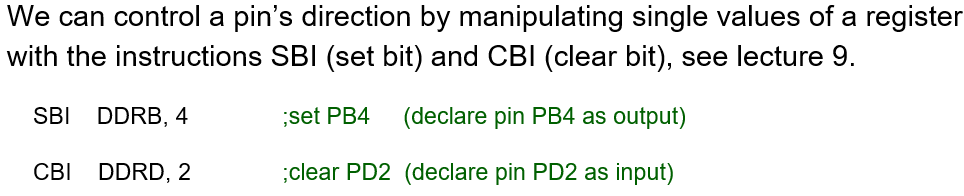
Data memory SRAM is divided into sections:

EEPROM = Electrically Erasable Programmable Read Only Memory – organised as a separate space for storing data (not programs and nothing that changes \*too\* often). Arduino EEPROM has an endurance of 100000 write/erase cycles.

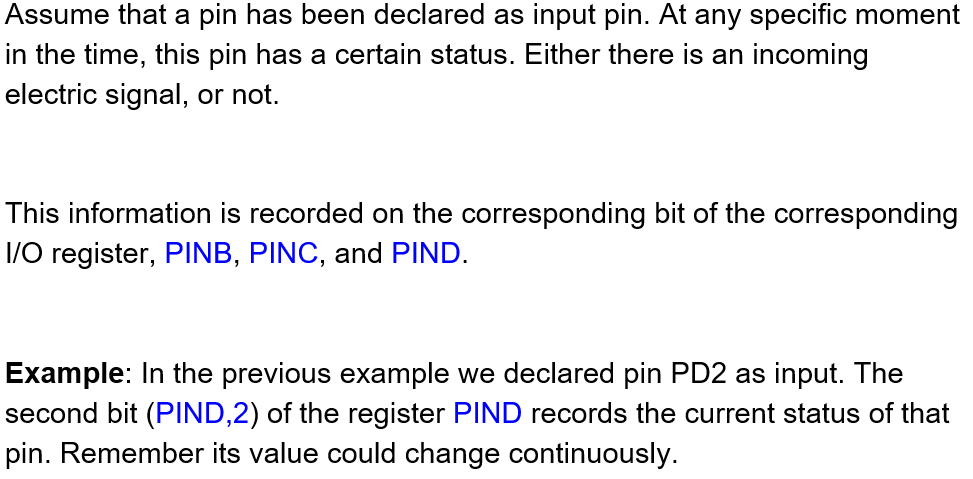
I/O:  
^Note port C has 7 pins

Data Direction Registers: 8 bit registers DDRB, DDRC, DDRD. Controls the direction of each pin of a port. The names DDRB, DDRC, DDRD are recognised by the assembler so we do not need to use explicit memory locations for these – also useful for compatibility with other microcontrollers.

Each bit in a DDR corresponds to a pin in the corresponding port. When a bit in the DDR is 1 (i.e. is “set”), the corresponding pin is output. If a bit is 0 (“cleared”), the corresponding pin is input. Bits can be manipulated using instructions:

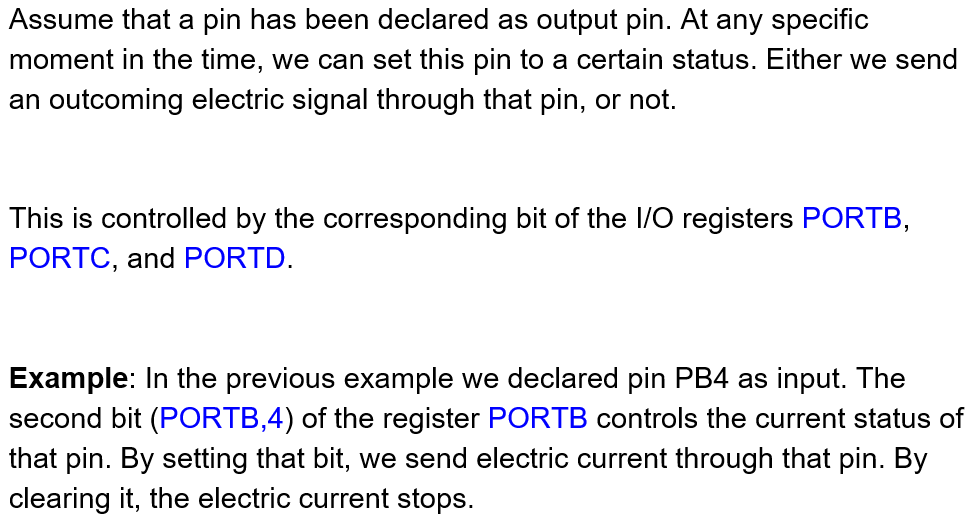
^sets the 4th bit in DDRB; clears the 2nd bit in DDRD – note bits indexed 0-7

Reading input values:

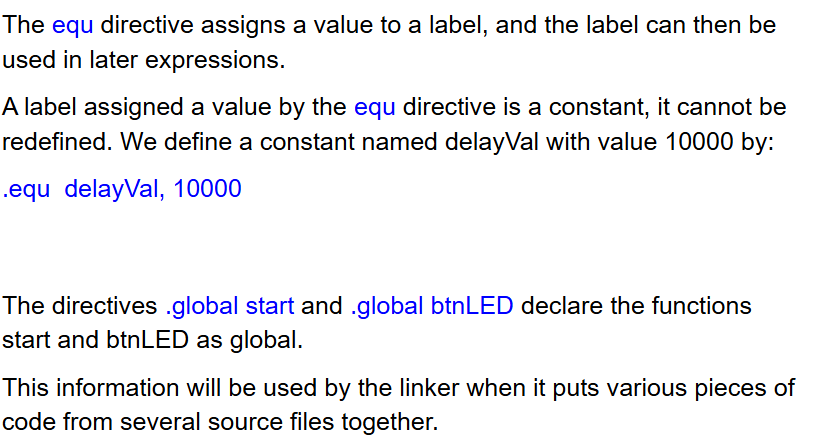
^status of a pin corresponds to 0 or 1 (?)

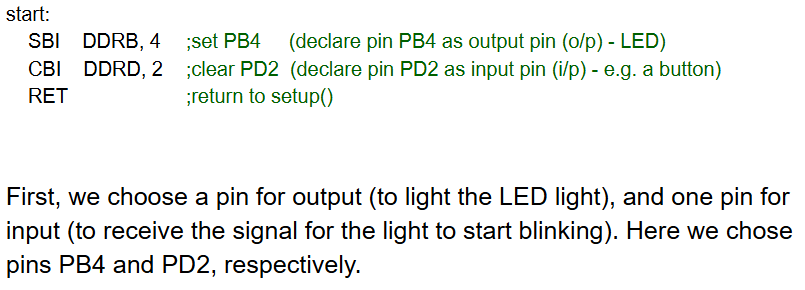
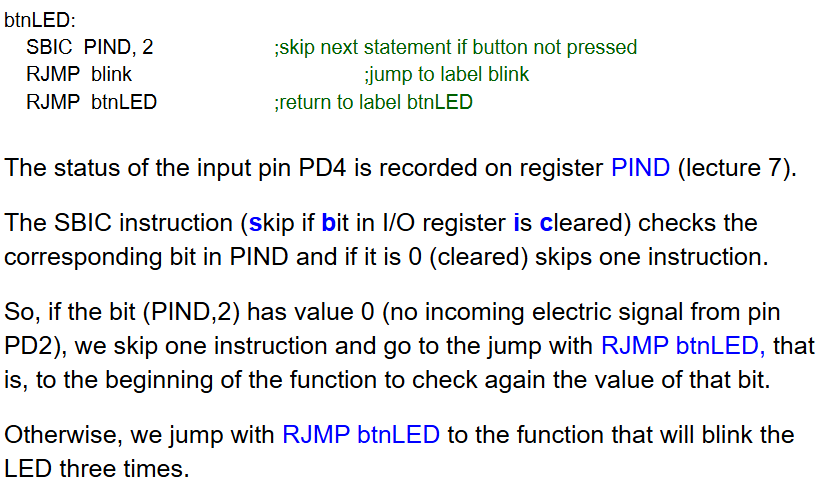
For outputs:

AVR Assembly:

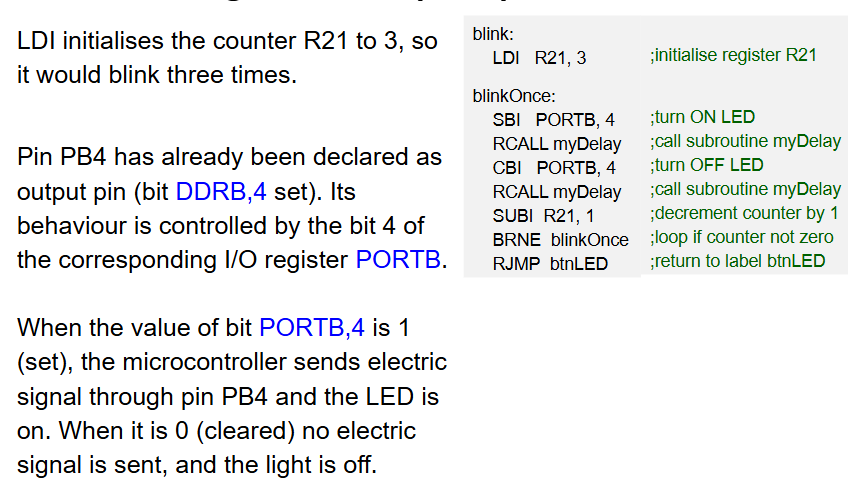
AVR assembly code contains statements called directives. A directive is preceded by a dot .

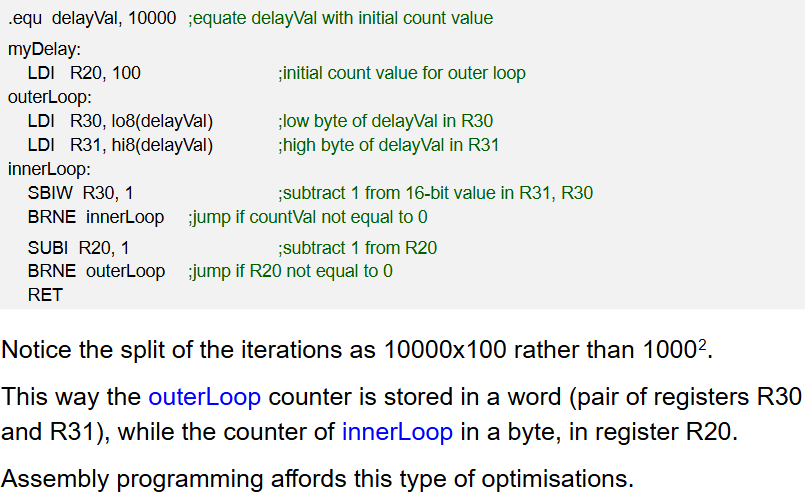
Directives do not directly correspond to assembly instructions, but make assignment etc easier.

.

.

Blink function:

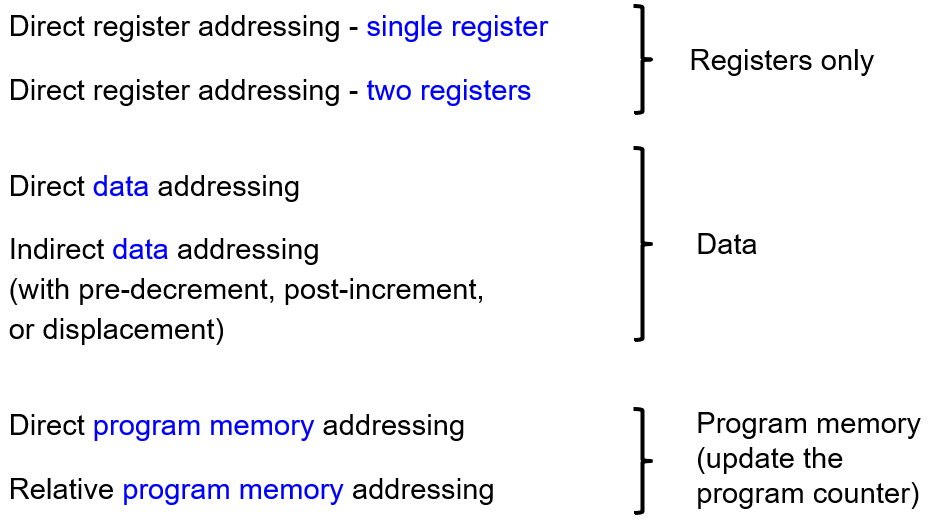
^see lecture slides for actual info on this 😐

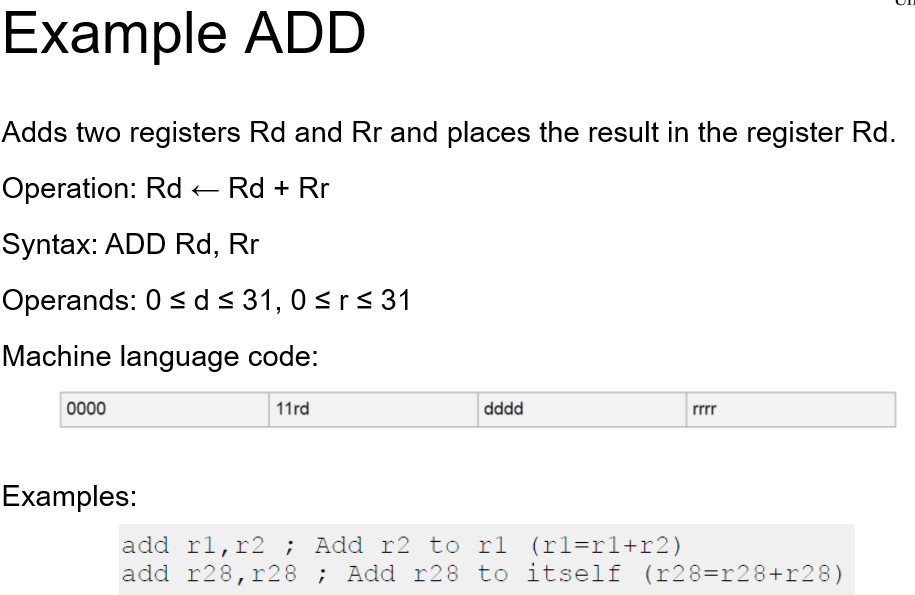
myDelay function:

(the aim is to perform 1 million iterations of the loop to act as a delay.)

AVR Instruction Set:

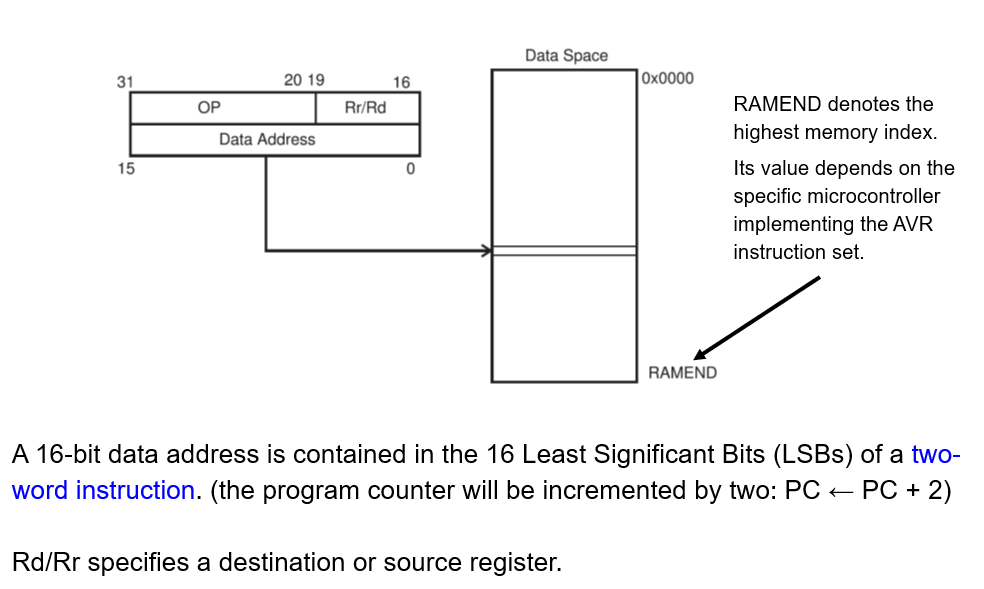
* 13 total addressing modes

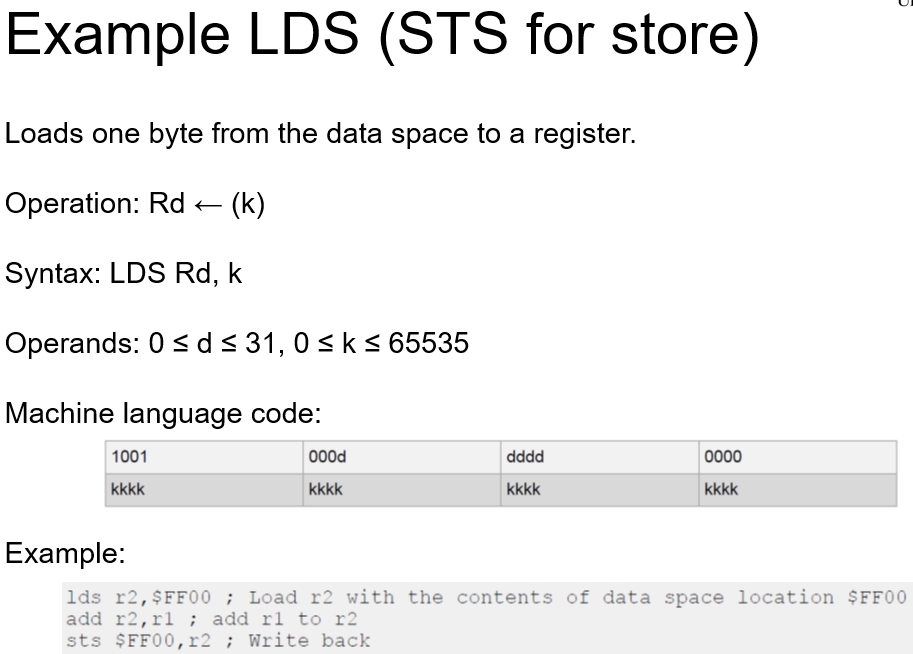


Example Add instruction:

^ d and r are between 0 and 31 as there are 32 registers. Rr = rrrrr, Rd = ddddd, not stored consecutively in the machine code instruction.

Direct data addressing:

Example:

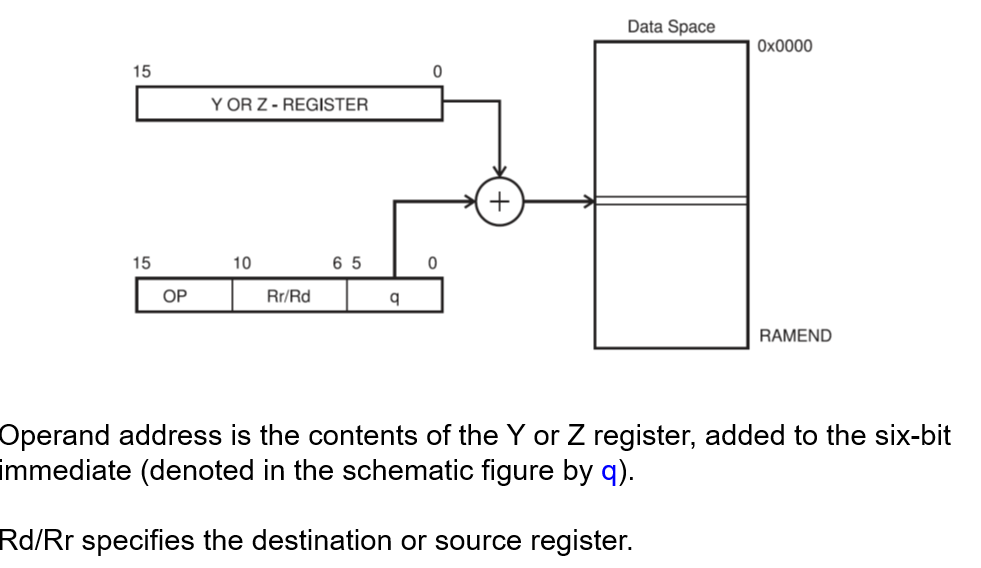
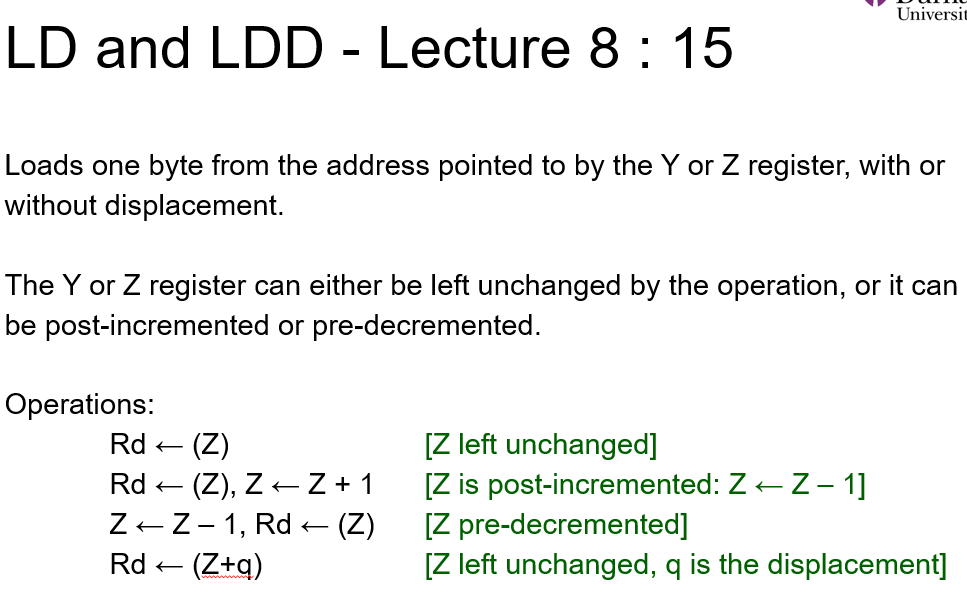
^there are 65536 addressable memory locations.

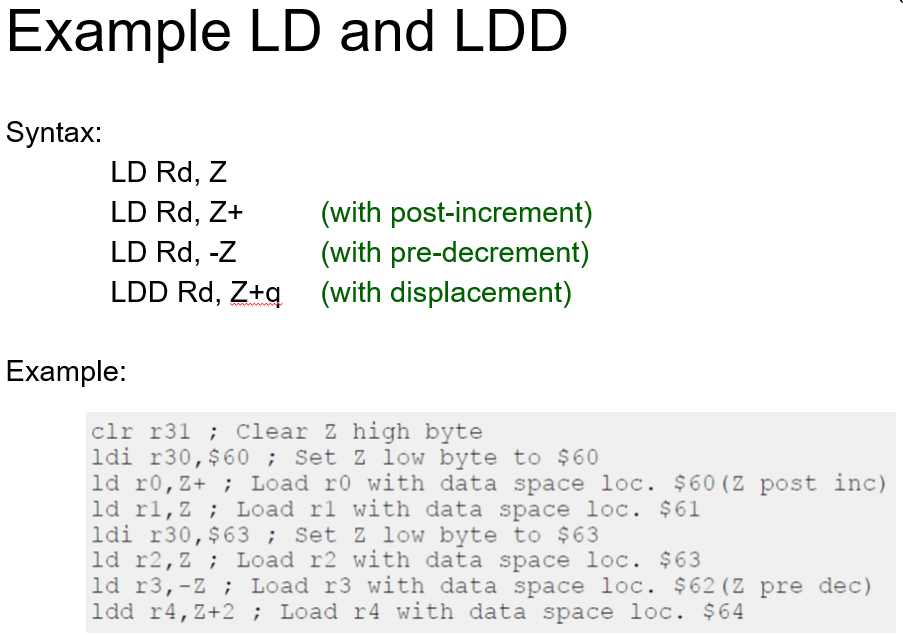
Indirect data addressing:

In indirect data addressing, the X, Y or Z register stores the address of the memory location.

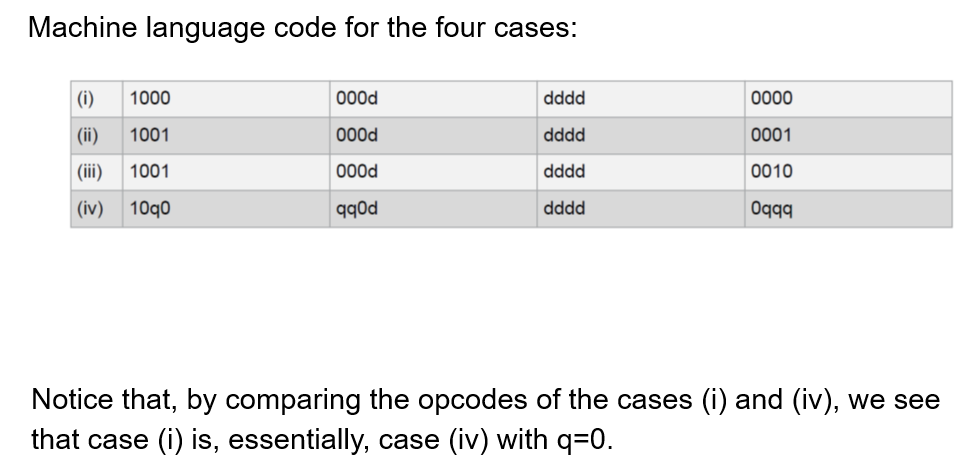
This can be implemented with post increment – where the contents of X, Y or Z are incremented after each call to it to represent the subsequent memory location, or pre decrement where the reference in X, Y or Z is decremented by 1 before each instruction call to iterate through memory locations backwards, i.e. each time an indirect addressing instruction “calls” X, Y or Z, the value of X, Y or Z is updated accordingly before or after each call.

Can also be implemented with displacement, using a 6 bit immediate

Explanation:

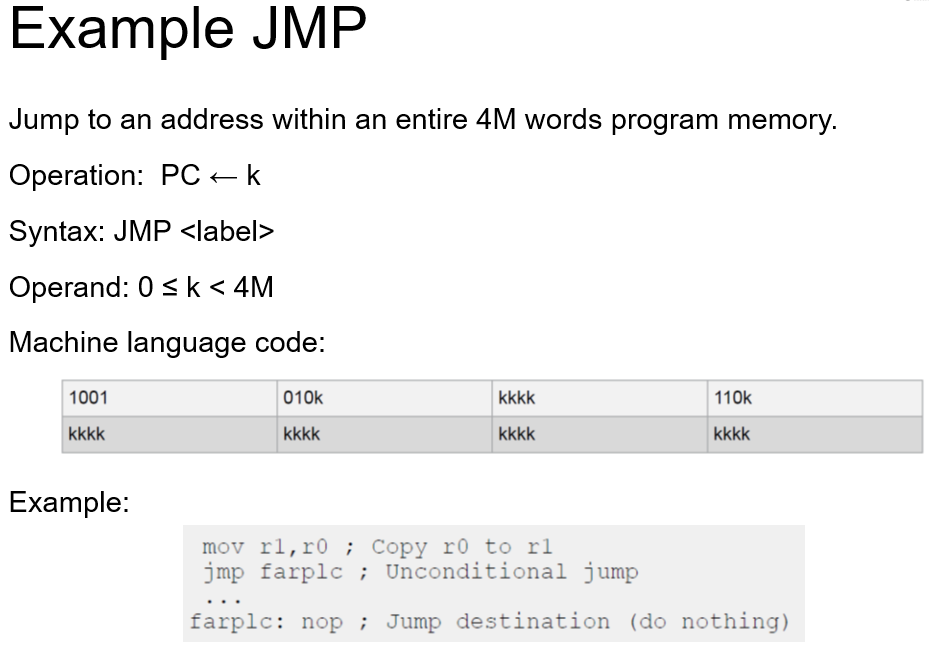
Syntax example:

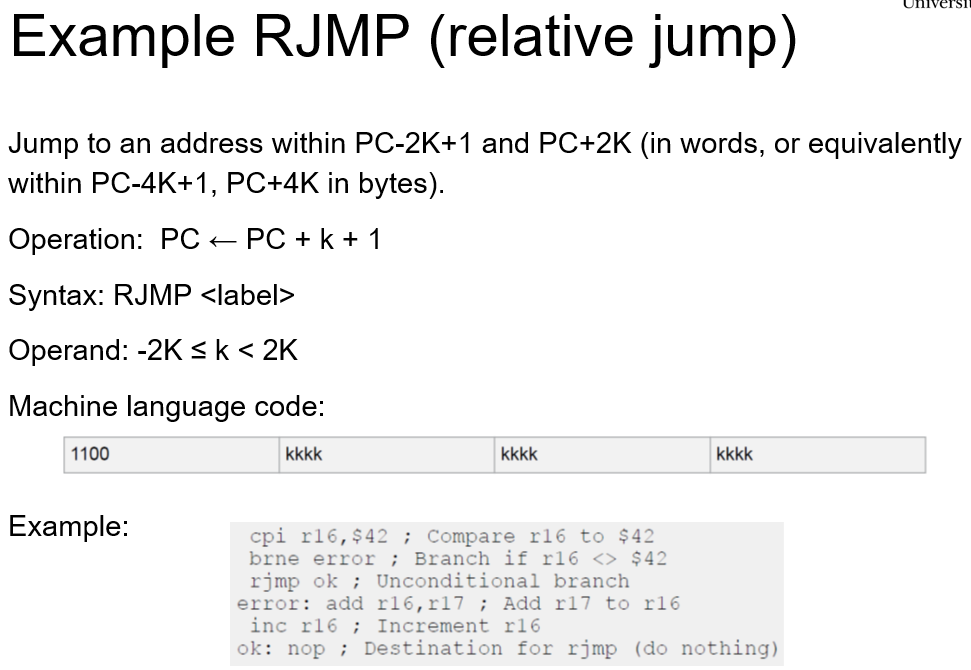
Machine code instructions: i = LD Rd, Z; ii = LD Rd Z+; iii = LD Rd, -Z; iv = LDD Rd, Z+q

^who on earth decided q should be stored like that ??

Program memory addressing – manipulating the Program Counter

* Direct program memory addressing -> the address immediate is directly sent to the PC and program execution will continue at the address.
* Relative program memory addressing -> program execution continues at current PC value + k + 1, where k is the immediate passed into the instruction.

Examples:

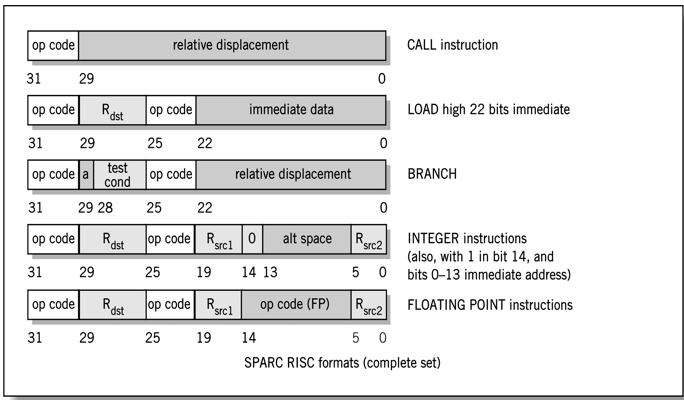
.

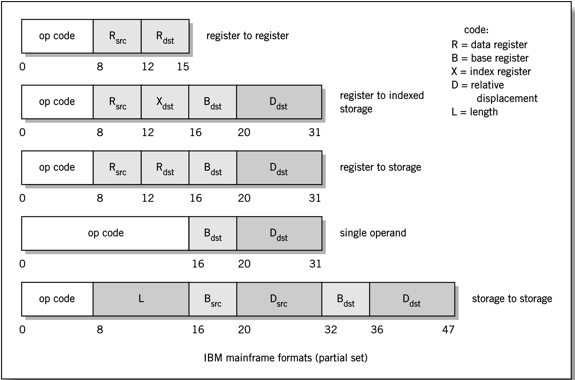
^relative address can be between -2048 and 2047

RISC and CISC

MIPS uses 3 different 32 bit instruction formats – R, I and J Type instructions.

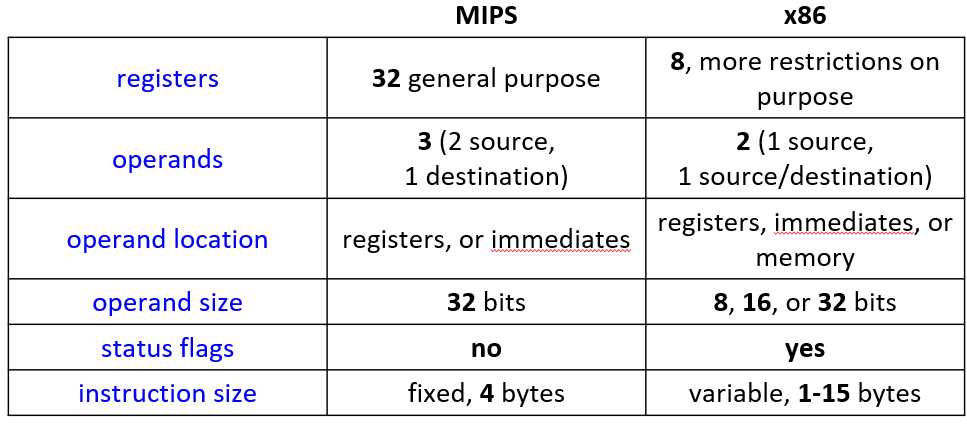
RISC (reduced instruction set computer) uses 5 different 32 bit instruction formats:

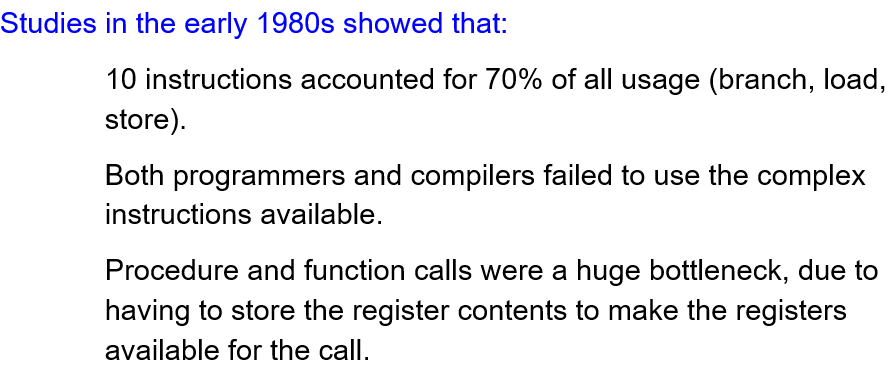
CISC (complex instruction set computer) uses dozens of formats in the full instruction set. The number of bits used by these instructions can vary. Example partial set:

CISC architecture has a large range of specialised instructions with a variety of addressing modes. Instruction length varies, e.g. from 1 to 15 bytes per instruction for Pentium, and there are few general purpose registers.

RISC uses a limited and simplified instruction set, with instructions that can execute more efficiently. Instructions are register-oriented – data must be fetched from memory into a register before operations can be used on it. Instructions have a fixed length and format, with a small number of addressing modes. A larger number of general purpose registers are used to store intermediate results to prevent excessive fetching from and storing to memory.

MIPS (an example of RISC) vs x86 (CISC):



CISC vs RISC in general:

* Use simplified instruction set instead (RISC) – simple instructions in CISC are slowed down by the complexity in the instruction decoder hardware needed to process the complex instructions, but these simple instructions are often the most common.
* But RISC programs often had to take up more memory as more of these simple instructions were needed to perform the same task as a single instruction in CISC.
* Nowadays, RISC architectures can have a larger instruction set due to advances in chip design, and newer CISC architectures have more general purpose registers, with some commonly used operations translated into RISC-like micro-ops of fixed length. Performance between RISC and CISC is now very similar.

