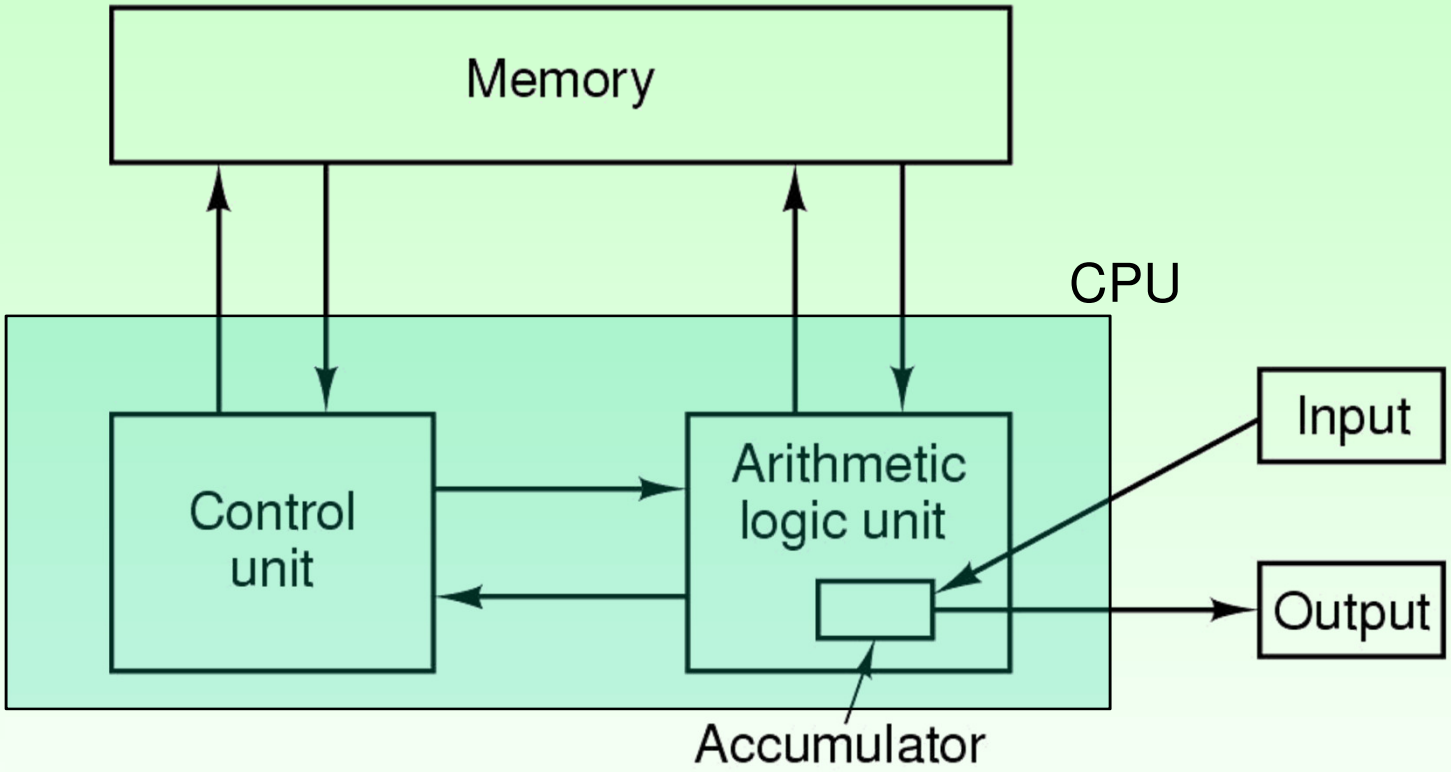
CSCU9V4
Systems

Systems Lecture 6

Computer Organisation 1

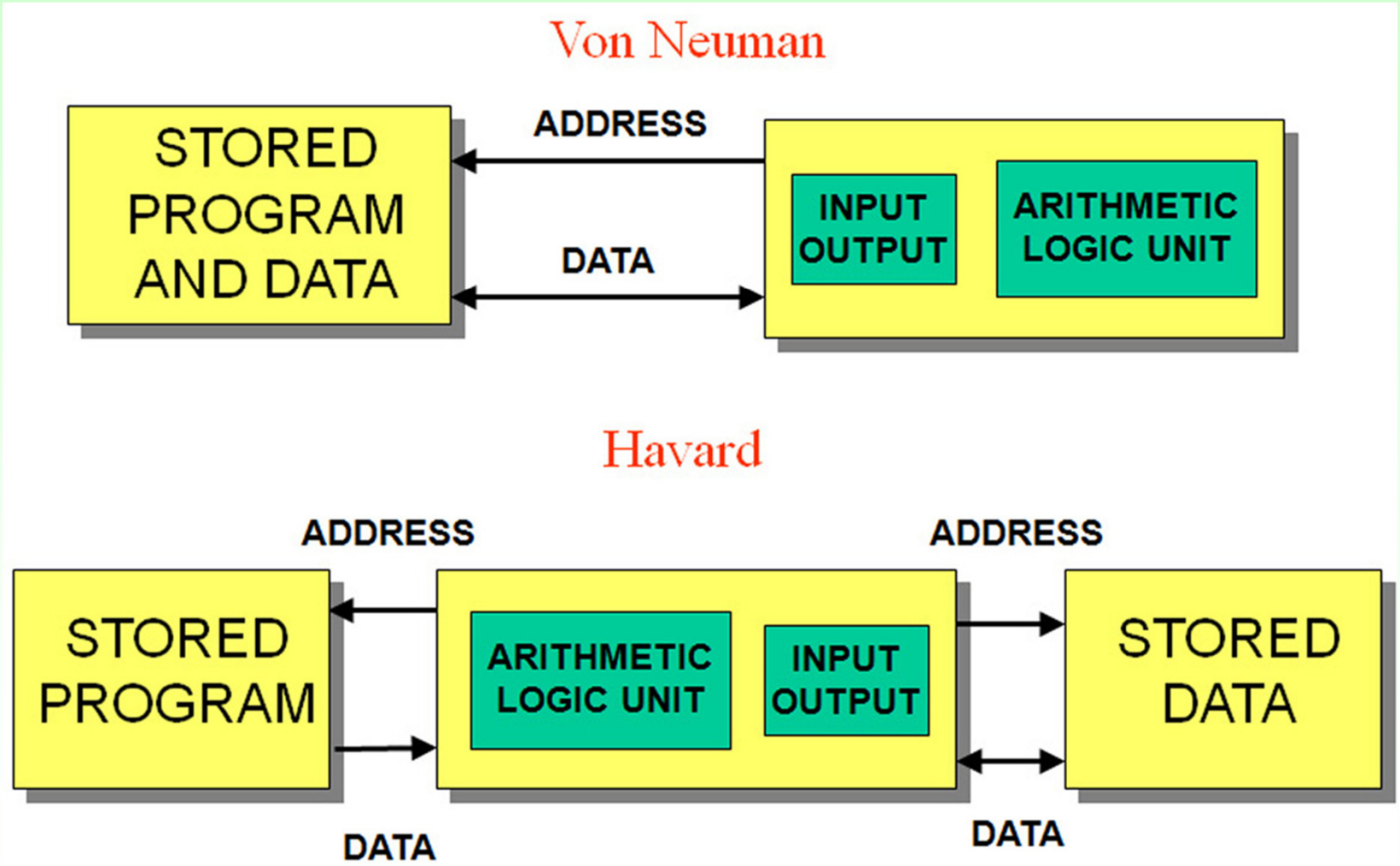
Central Processing Unit (CPU)

Original Von Neumann machine

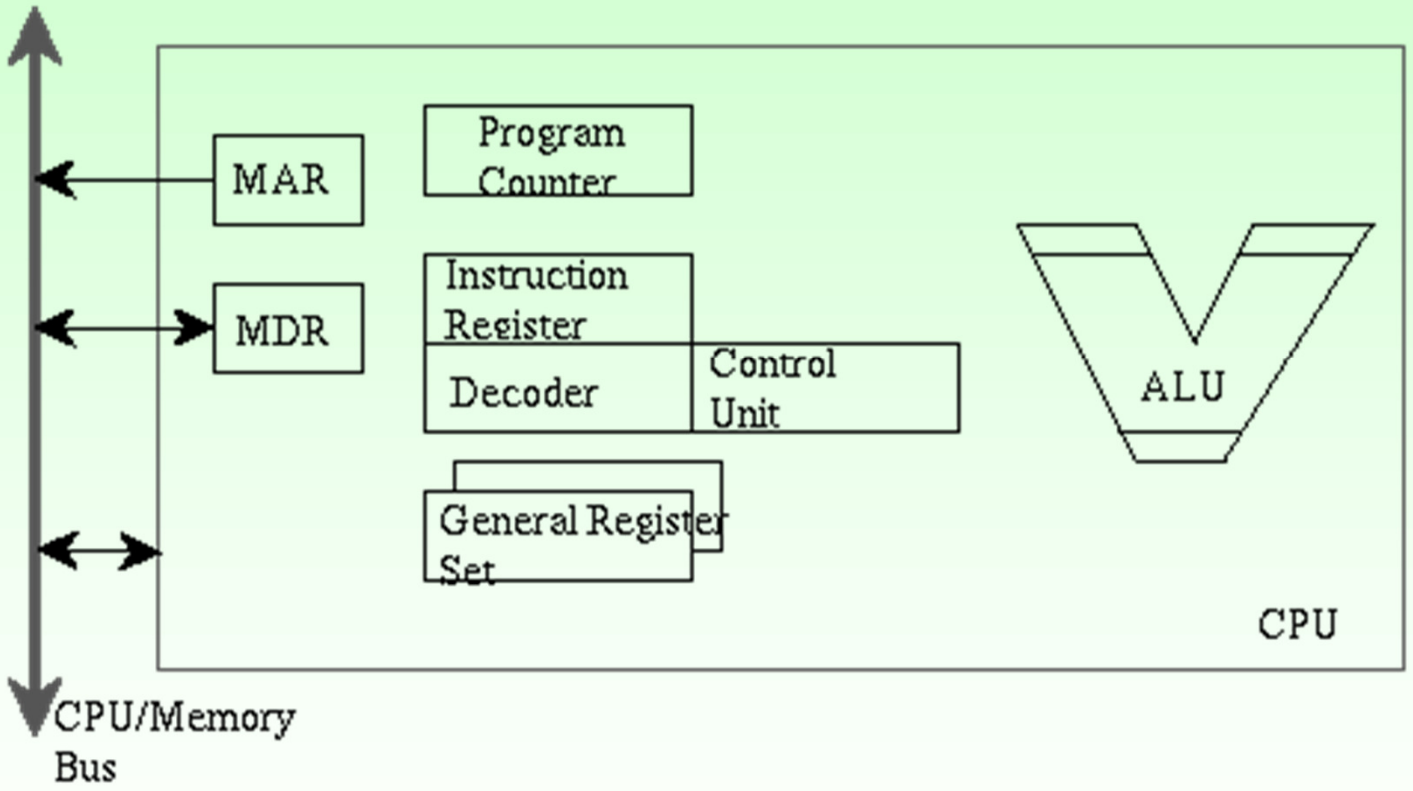


- Five basic parts: memory, ALU, control unit, input, output
- Control Unit and ALU access one memory
  - For program and data
- ALU had an accumulator (register)
  - No floating point arithmetic (and probably no multiplier either!)
- Interfaces to outside directly from ALU

Von Neumann Vs Harvard Architecture



"Basic" modern von Neumann computer



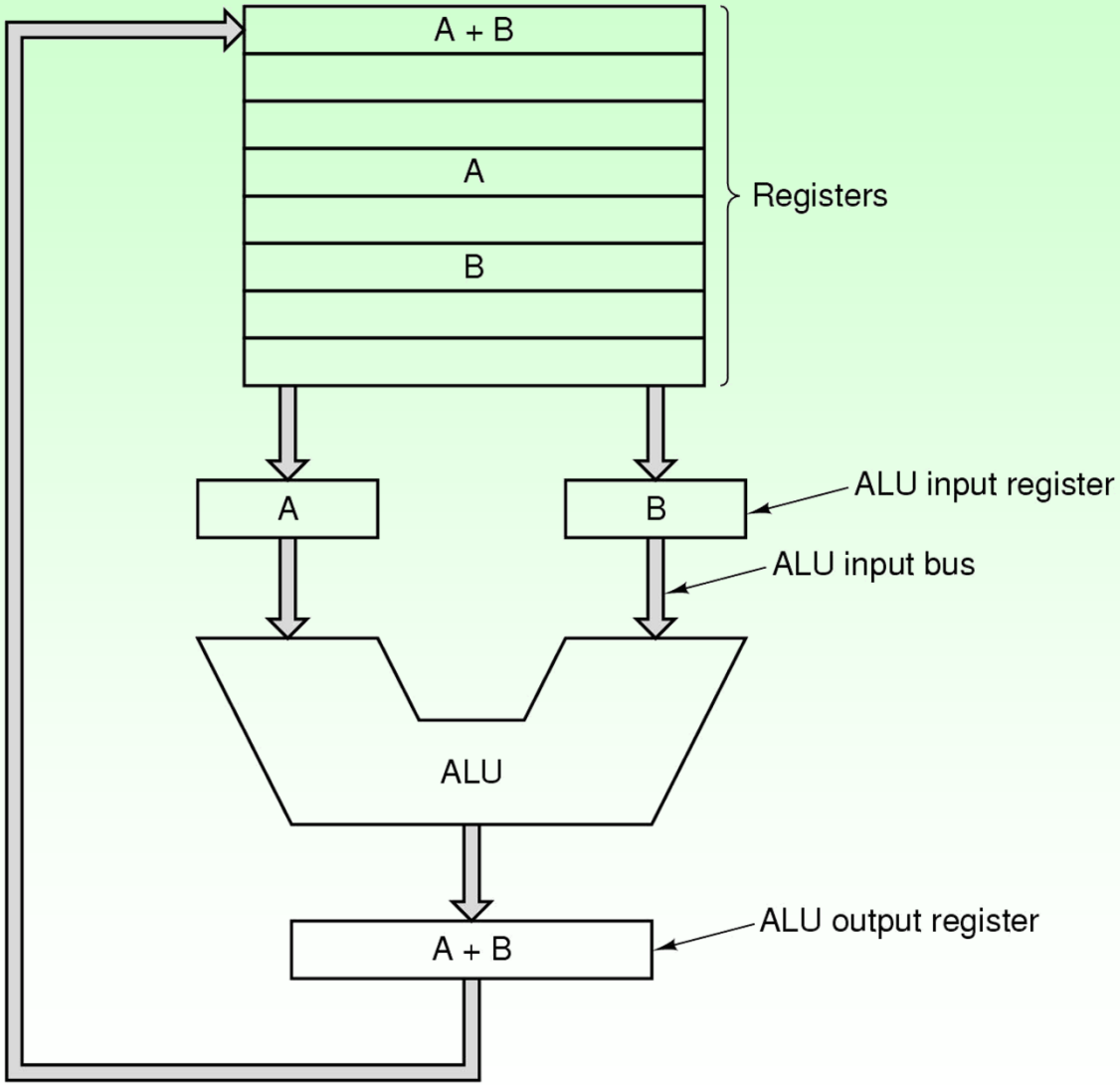
(Interconnection inside the CPU omitted for clarity)

Datapath

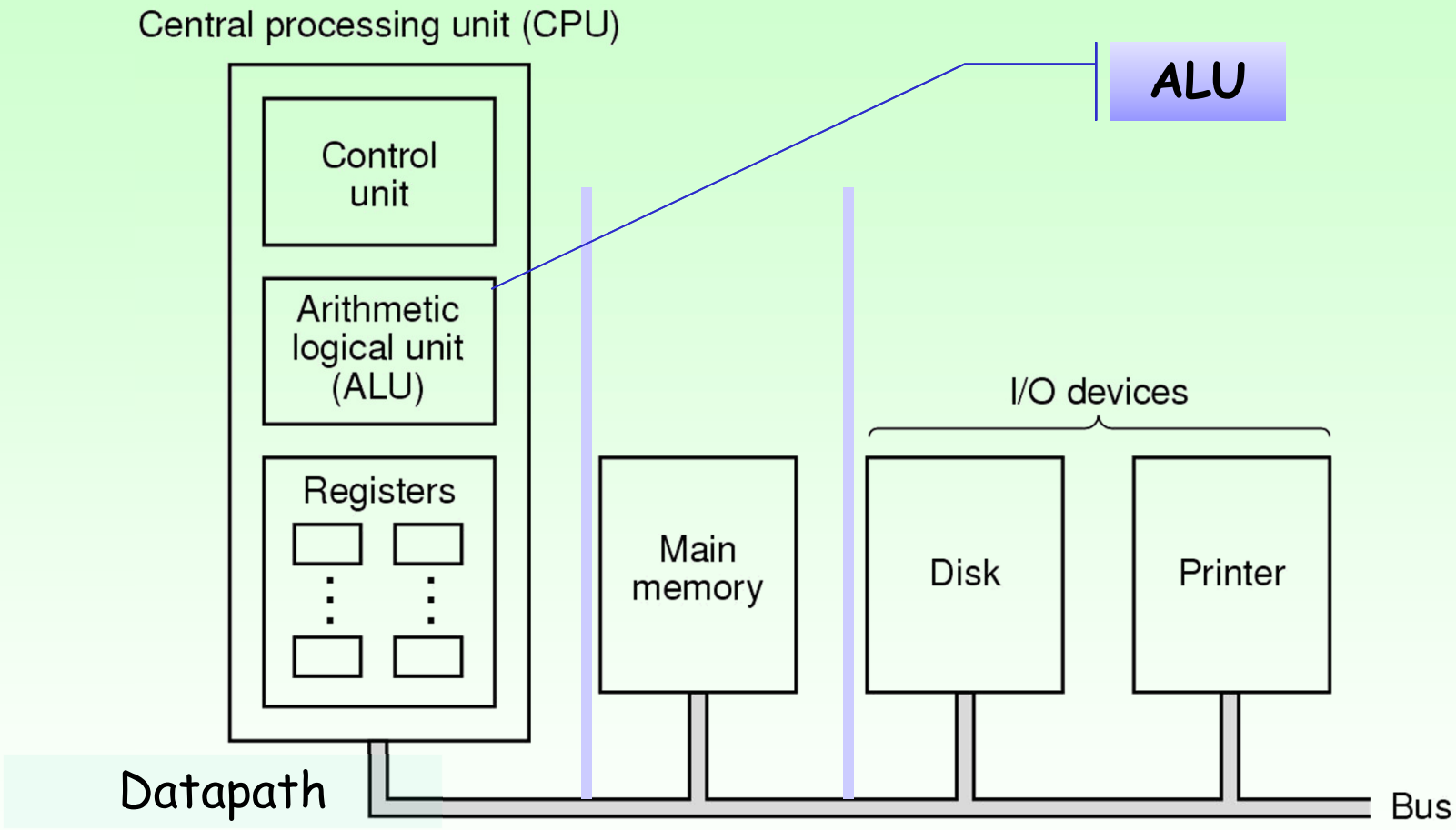
Different view of ALU and general register set.

- Data from Registers is sent (by way of ALU input registers) to ALU.

- Results go from ALU output register back to Registers.



Structure of a Simple Computer



Note that modern CPUs are ***much*** more complex than this!

Arithmetic and Logic Unit (ALU)

- The functional unit which performs
  - virtually all data-altering and comparison operations
  - calculations, logical operations, shifts, etc.

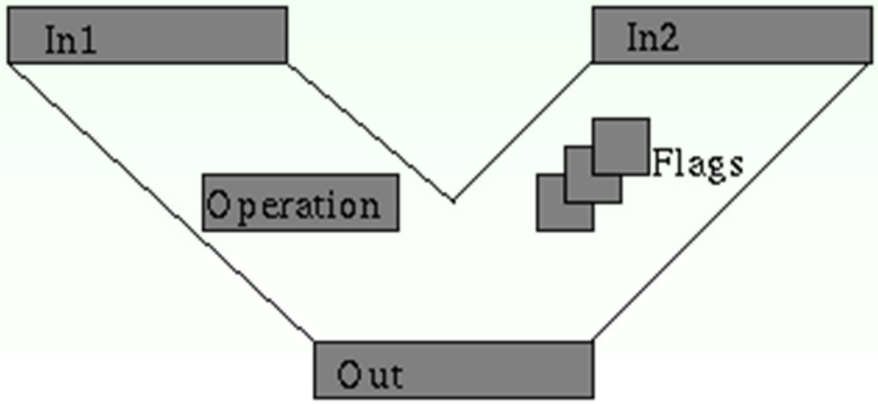
- In a sense all the rest of the CPU is peripheral to the ALU, in that it is concerned with the selection of operands, storage of results, and sequencing of operations at the ALU.

- For data-altering operations, the result from the ALU will be stored (in a general register, or in memory).

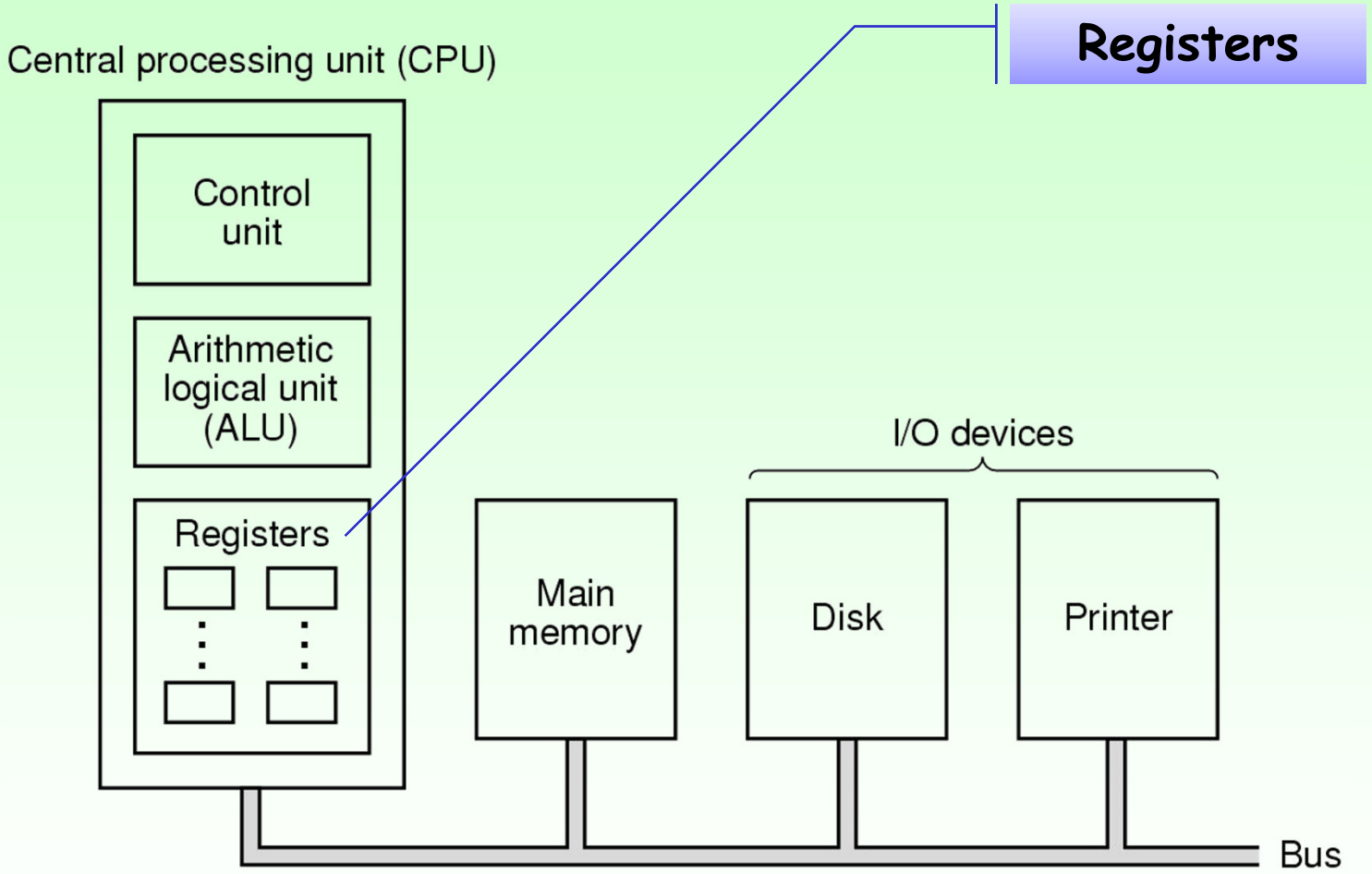
- For comparison operations, the result of the comparison will be stored in one of the flag registers.

The Arithmetic and Logic Unit (ALU)

- The ALU performs an operation defined by the contents of its operation register.
  - The input registers are filled, then the operation register is set.
  - The output register is then filled with some function of the two input registers:
    - add, subtract, multiply, shift left, negate, AND, OR, ...
- The ALU performs one operation at a time.
- Internally, the ALU has an operation register, and some flags
- Running two operands through the ALU and storing the result is called the 'Data Path Cycle'



Structure of a Simple Computer



Component Units of a Simple CPU: Registers

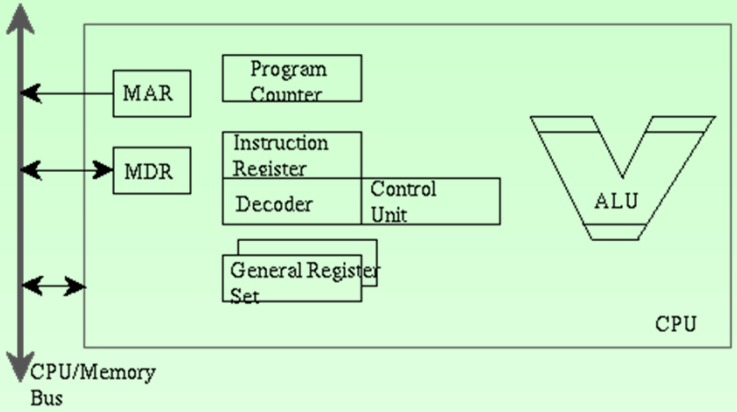
- Very high speed local storage
  - Can be accessed (e.g. transferred to ALU) very rapidly - much more quickly than main memory words.

- Some registers are general purpose (general registers)
  - i.e. available to the low-level language programmer
    - (or, more frequently, to the compiler designer)
  - for holding variables, temporary results, addresses of operands,…
  - More later (programmers view, ISA level)

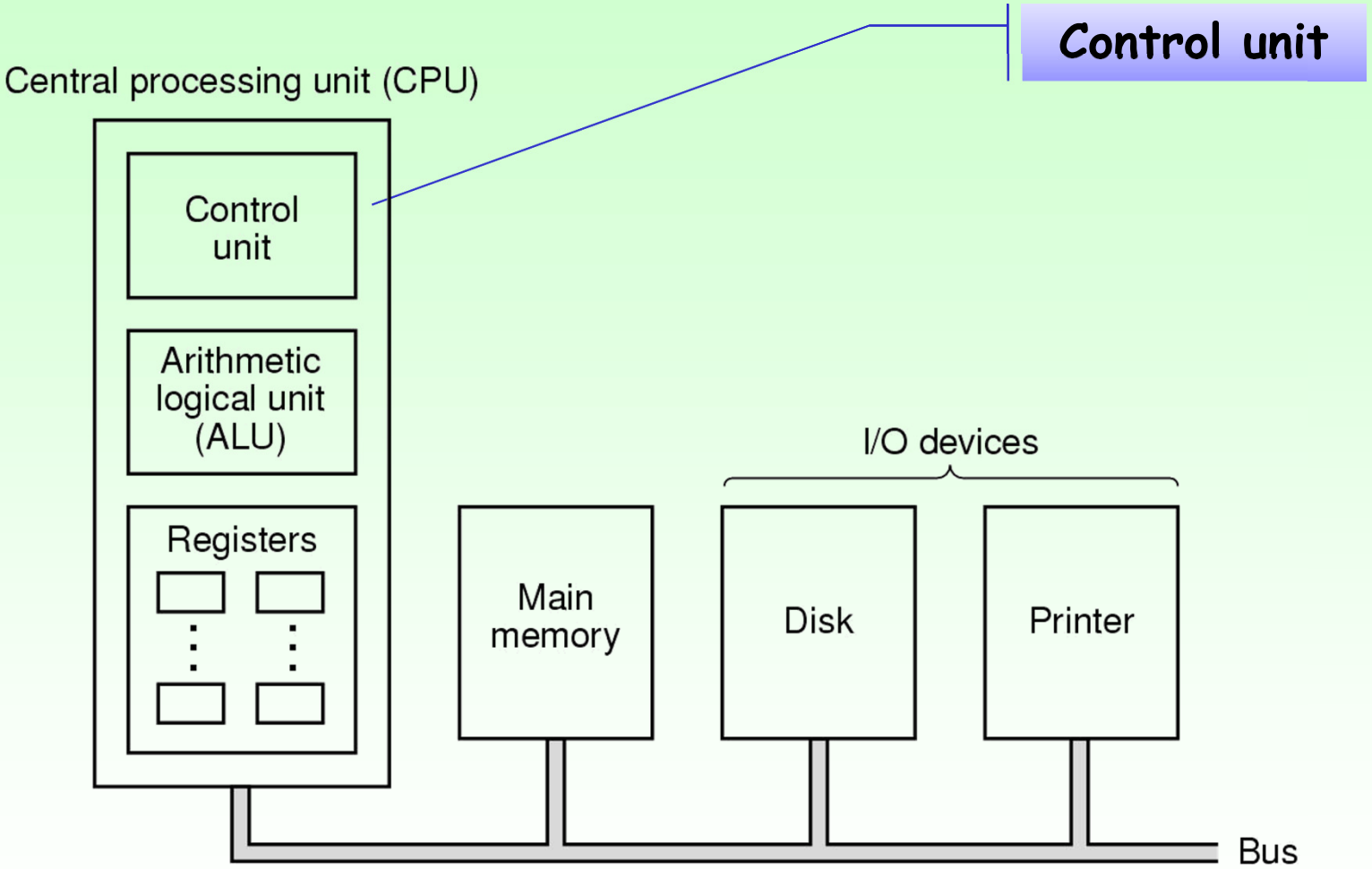
- Sometimes the OS designer forces compiler designers to follow particular conventions about the use of these General Purpose registers…

Registers cont. …

- Some registers have fixed functions:
  - **PC** Program Counter
    - Holds the address of the next instruction to be executed.
  - **IR** Instruction Register
    - Holds the instruction currently being executed
  - **SP** Stack Pointer
    - Holds the address of the top of the Stack
  - **MAR** Memory Address Register
    - Holds the address being accessed in main memory (or cache memory)
  - **MDR** Memory Data Register
    - Holds the data being read from/written to cache/main memory.
  - **Flag Registers**
    - 1-bit registers holding useful bits of state information: overflow, carry, etc.



structure of a simple computer



Control Unit

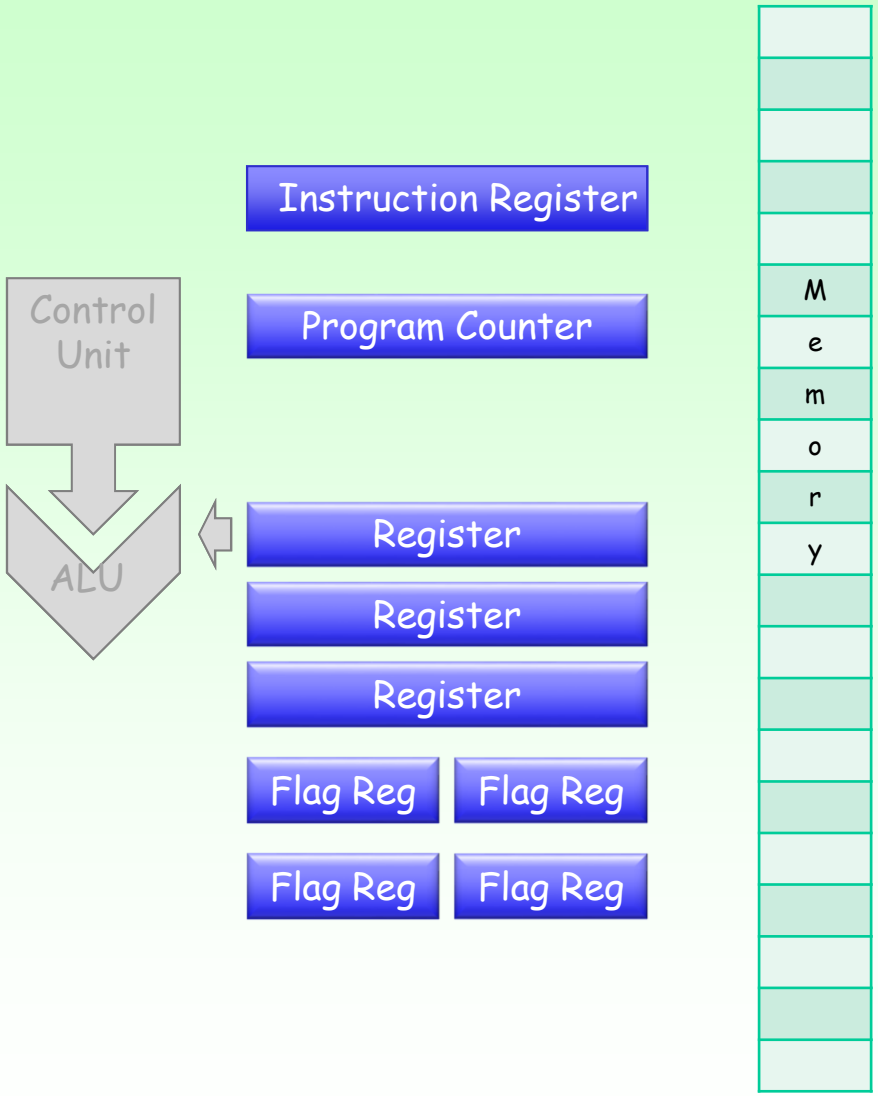
- Generates a sequence of control signals which cause:
  - Movement of data between registers
  - Fetching of a new instruction
  - Determining the type of new instruction
  - Loading of operands into registers
  - Storing of results from operands to main memory
  - arithmetic/logical operations to be undertaken.

- The exact sequence of control signals that will be generated for a particular instruction will depend both on the instruction itself, and on flag register values.

- We will need an aside into instruction execution ……but first

Programmer's View of the CPU and Memory

- The low-level programmer sees the machine as a collection of registers and the main memory as a (large) 1-dimensional array of words, with addresses from 0 to some large number.
  - High level languages allow structures to be imposed on this 1 dimensional array (e.g. arrays, structures, objects, etc.)



Machine Code

- Machine code is binary code
  - not readable by people (without extreme difficulty or training)
  - very readable by machines

- Binary
  - literally only 1s and 0s
  - like 11100111100000011110010111110000

- Instructions are tightly coded
  - code the program efficiently
  - that is, make every bit count (allows program to fit in minimum amount of memory)
  - For the human eye it is very hard to see what is an instruction and what are parameters.
- We'll come back to this in CISC vs RISC

Assembler Code

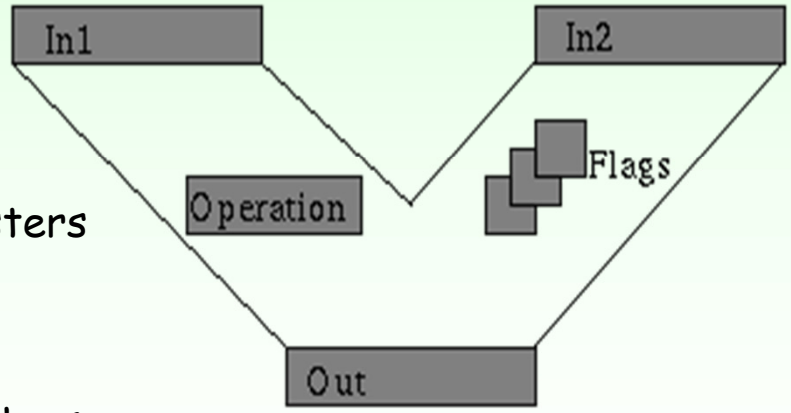
- Low-level (i.e. assembly-code) programmers write assembly code instructions which look like:

```
ADD R1, R3
```

- The effect of this instruction would be to add the contents of registers **R1** and **R3**, placing the result in **R1**.

  ADD is the *operation*, and **R1** and **R3** are the *operands*

- Register-Register operation
  - Operation fetches both operands from registers
    - Places them in the ALU input registers
  - Operation is performed in the ALU
  - Result is stored back in one of the two registers



Assembler Code

- Another instruction could be

  `LOAD R1, 1FEA009Ax`

  - **`LOAD`** is the *operation*, and **`R1`** and **`1FEA009Ax`** are the *operands*
  - The effect would be to load register **`R1`** from the address **`1FEA009Ax`**.

- Register-Memory instruction
  - Allow memory words to be fetched into registers, where they may be used as input for ALU in subsequent instructions
  - Other instructions allow register values to be stored back into main memory
  - Some other instructions (except Load, Store) may also allow for an operand in main memory

Execution

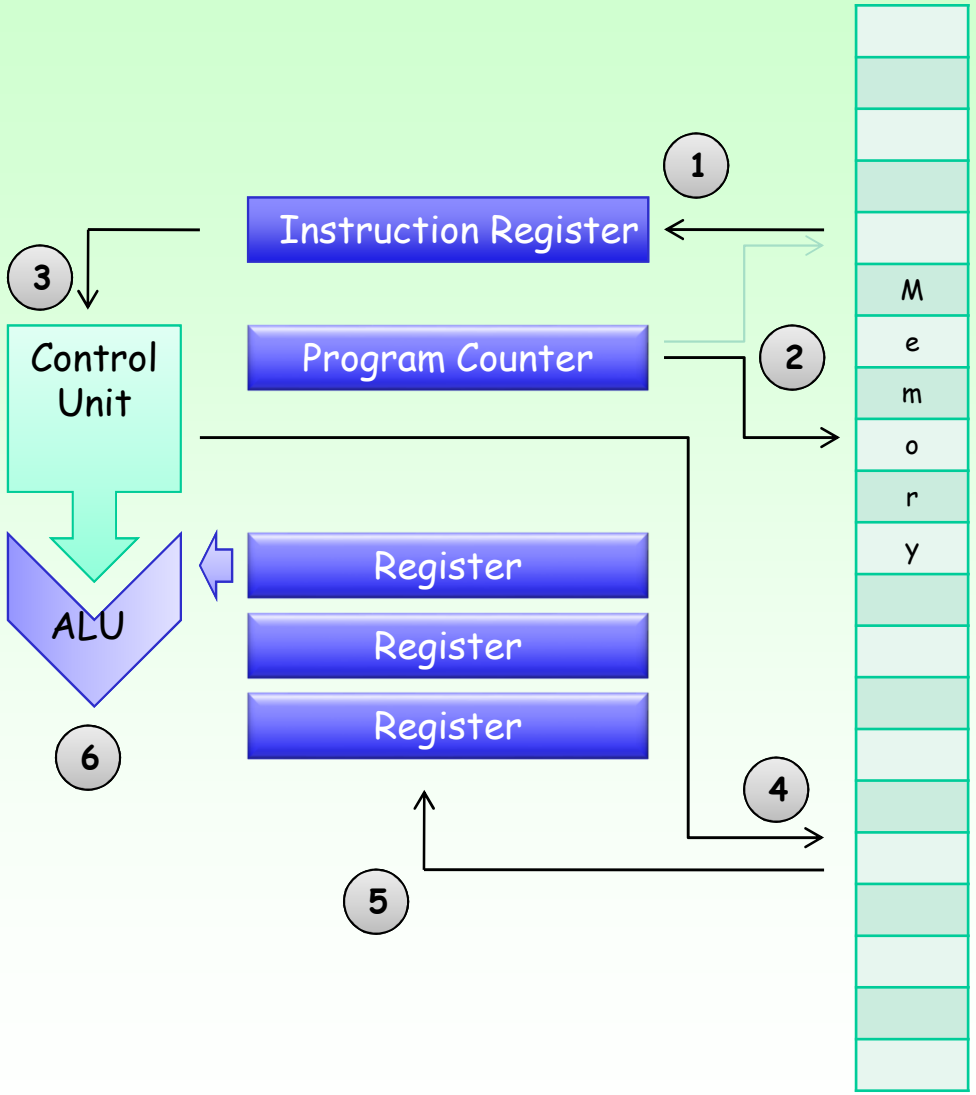
- A program is a sequence of instructions.
- A machine-code program is a sequence of machine-code instructions.
- To be executed, a machine-code program *must* be stored in main memory.
- Execution takes place within the CPU.
- Execution follows 'Fetch-Decode-Execute' cycle

Fetch->Decode->Execute

Each instruction is executed in a number of small steps:

1. Fetch the next instruction from memory into the Instruction Register
2. Change the Program Counter to point to the next instruction
3. Decode the current instruction
4. If the instruction uses a word in memory, determine where it is
5. Fetch the word (in a CPU register if needed)
6. Execute the instruction

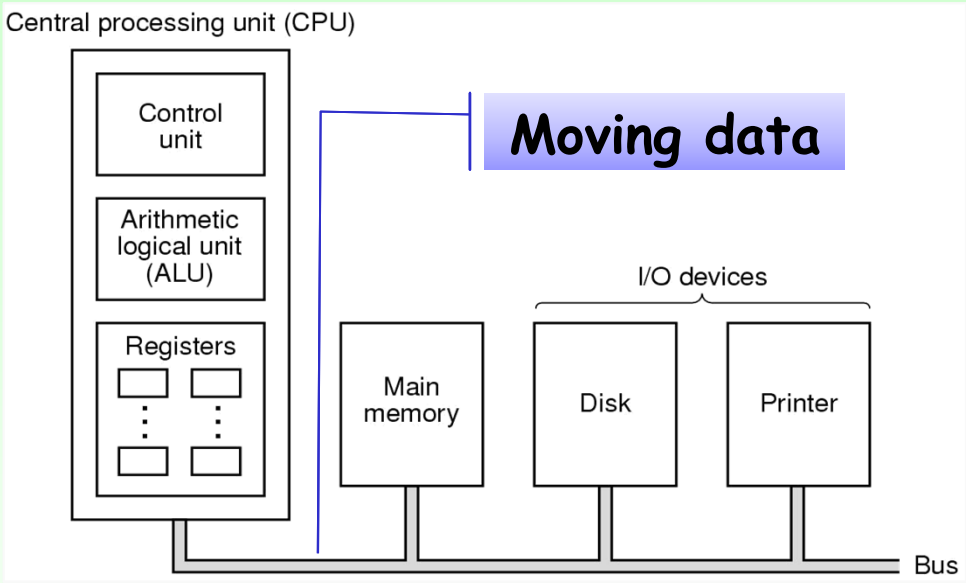
... and repeat until the instruction is 'Halt'



The Instruction-Execution Cycle

- 'Fetch-Decode-Execute' cycle
  - Instructions are transferred one-by-one to the CPU from memory
  - Decoded, i.e. the type of instruction is determined
  - Executed
- Central to all computers (Hardware or Micro-program Interpreter)
- To the programmer, instructions are executed sequentially and atomically:
  - 'atomically': each instruction is regarded as an indivisible entity.
  - from a software engineer's viewpoint this illusion works because the hardware does not allow the execution of an instruction to be interrupted.
- In reality, instruction execution involves a sequence of steps,
  - each specifying data movement, and possibly alteration of data.
- Four special-purpose registers in the CPU are primarily involved in this process - PC, IR, MAR, MDR

Structure of a Simple Computer



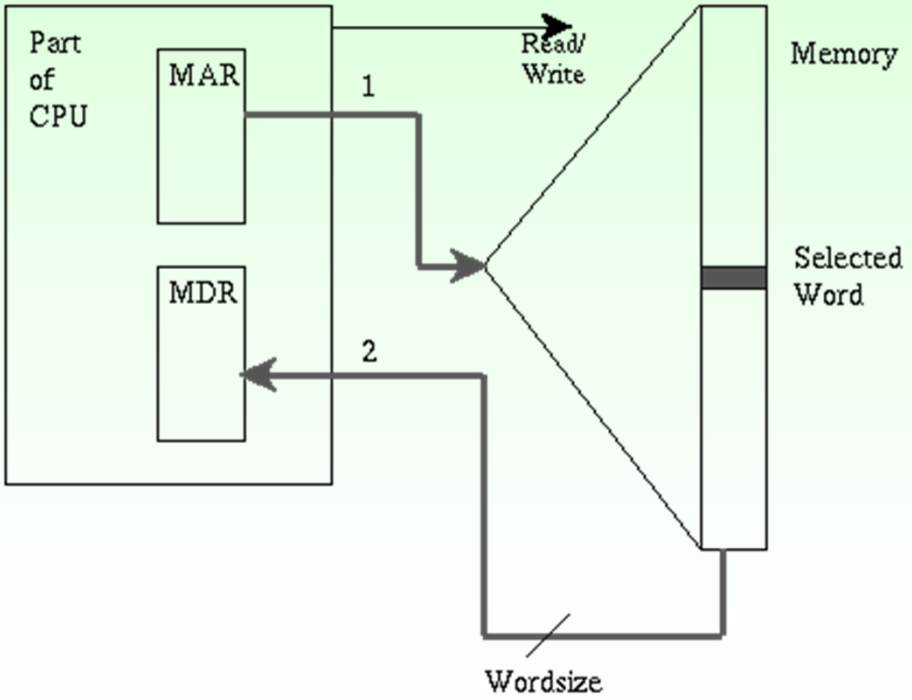
Reading/Writing Main Memory

Reading

- **Memory Address Register (MAR)** is set to an index value for the memory array
  - Used to select the word of memory to be read
- Read/Write signal set to '*Read*'
- The selected word of memory accessed is loaded into **Memory Data Register (MDR)**

Writing

- **MAR** is set to an index value for the memory array
  - Used to select the word of memory to be written
- Read/Write signal set to '*Write*'
  - The selected word is stored.



Performance

- Interpretation vs Hardware
- Reduced instructions set computers (RISC) Vs Complex Instruction Set Computers (CISC)
- Basic rules
  - Maximise the rate at which instructions are issued/completed
  - RISC:
    - Easy (=fast) to decode instructions
    - Only load and store instructions should access main memory
    - Others operate on registers
- Performance is important (obviously), but more advanced than this course really allows.
  - Parallelism (a *really big* topic)
    - instruction level
    - processor level

- But what is this CISC and RISC stuff?

At the Dawn of 'Computer Time'

- Compilers had yet to exist
  - Machine code and assembly were the only available tools.

- CPUs had access to very limited memory
  - This promoted a high density of information in programs
  - Instructions of variable size, instructions which perform multiple operations, and instructions that both moved data and performed data computations
    - Commonly used instructions were small
  - The ability to pack instructions densely was considered more important than instruction decodability
- Register count was (very) small
  - Register bits were very expensive
  - More registers →
  - Needed more bits to identify which register in instruction →
  - Needed more RAM to store program.

The birth of complex instructions

- Machine instructions, then, designed to do as much as possible in a single instruction:
  - One instruction could load up two numbers, add them, and store the result back to memory
  - Another version of that instruction would do the same, but store the result in a register
  - Yet another version would read one number from memory, the second from register, and write the result back to memory
  - Yet another…
- The goal of the hour was to provide every possible variation of every instruction. (This principle is known as "orthogonality")
- This leads to complexity on the CPU, but this was considered a fair trade-off since in theory it is possible to tune each command individually
  - And it gave the instructions a logical structure, useful for both programmers and compiler writers.

Then Came Measurement… and Compilers!

- 1970s – increasing body of work to show that many of the versions of the instructions were being ignored.
  - Particularly by compiler writers
- Compilers only had a limited ability to make full use of CISCs orthogonal capabilities
- Extremely specific instructions were found to be slower than a sequence of general instructions doing the same thing
  - So long as intermediate results were stored on CPU
- Additionally, CPUs were 'clocking' faster than memory (still the case)
- The only solution: More registers!
  - But registers occupy space on chip
  - Space created by reducing complexity of the CPU.

The Birth of Reduced Instruction Sets (RISC)

- Most operations are simple + Make common operations as fast as possible
  = instructions so simple, they complete in a single CPU cycle.

- Where is the trade-off? (Consider the 'reduced' label.)

- Over time, single complex instructions implemented as a series of smaller instructions. Again, where is the trade-off?
  - (Think program memory accesses.)
- Pipelining
  - A way of making CPU's faster: run a number of instructions at once
  - Easier if the instructions are all similar in operation

- Later, 'superscalar' processes were more easily adopted into RISC because of their simplicity.

CISC (Side by Side) RISC

- Complexity in hardware required for implementation
- Memory-to-memory functions in single instructions
- Fewer lines of code
- Variable instruction size → difficult to decode
- New h/w optimisations hard to implement.

- Complexity in software (eg. compilers): can free space for more registers
- Register-to-register operations; load and store are separate instructions
- More instructions for equivalent tasks
- Uniform instruction length → easier to decode
- New h/w optimisations (so far) more easily adopted.

So why is my computer CISC? Consider…

- CISC chips
  - Intel
  - AMD
  - ie. most desktops and laptops.
- RISC Chips
  - ARM (yay, UK!)
  - IBM
  - and most everything else (phones, vehicles, TVs, etc.)

In fact, *inside* the Intel chip, CISC instructions are implemented as sequences of RISC instructions using a technique called microcoding.

Necessary to enable code to take advantage of extremely complex CPU cores: too advanced for this module.

End of Lecture