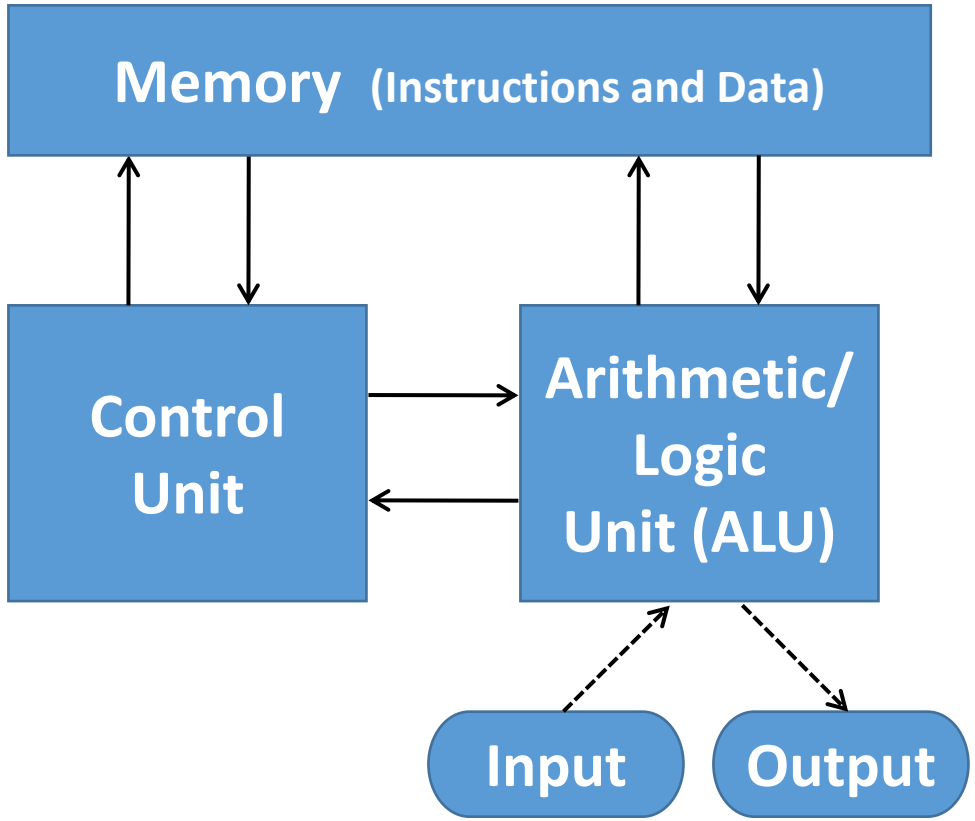
Lancaster University

SCC131: Digital Systems

Part I Recap (Week 1 to week 6)

Elements of computer architecture

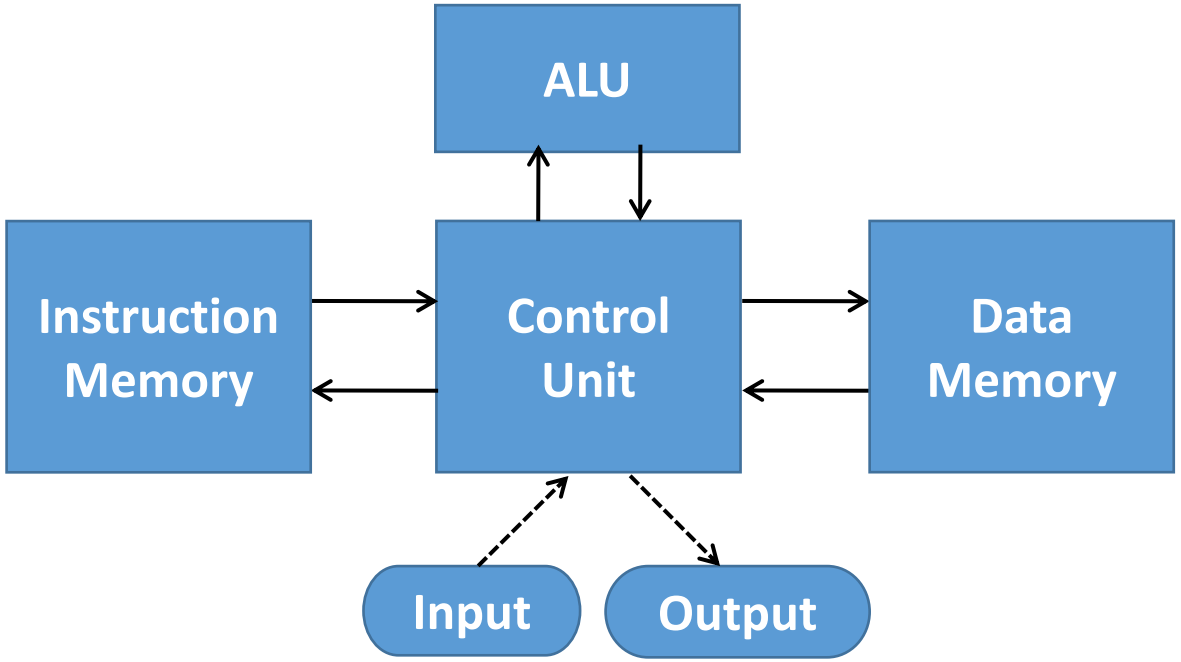
- Today, computer architecture is largely standardised, at a high level of abstraction, on the ***von Neumann Architecture***



Memory that stores data and instructions

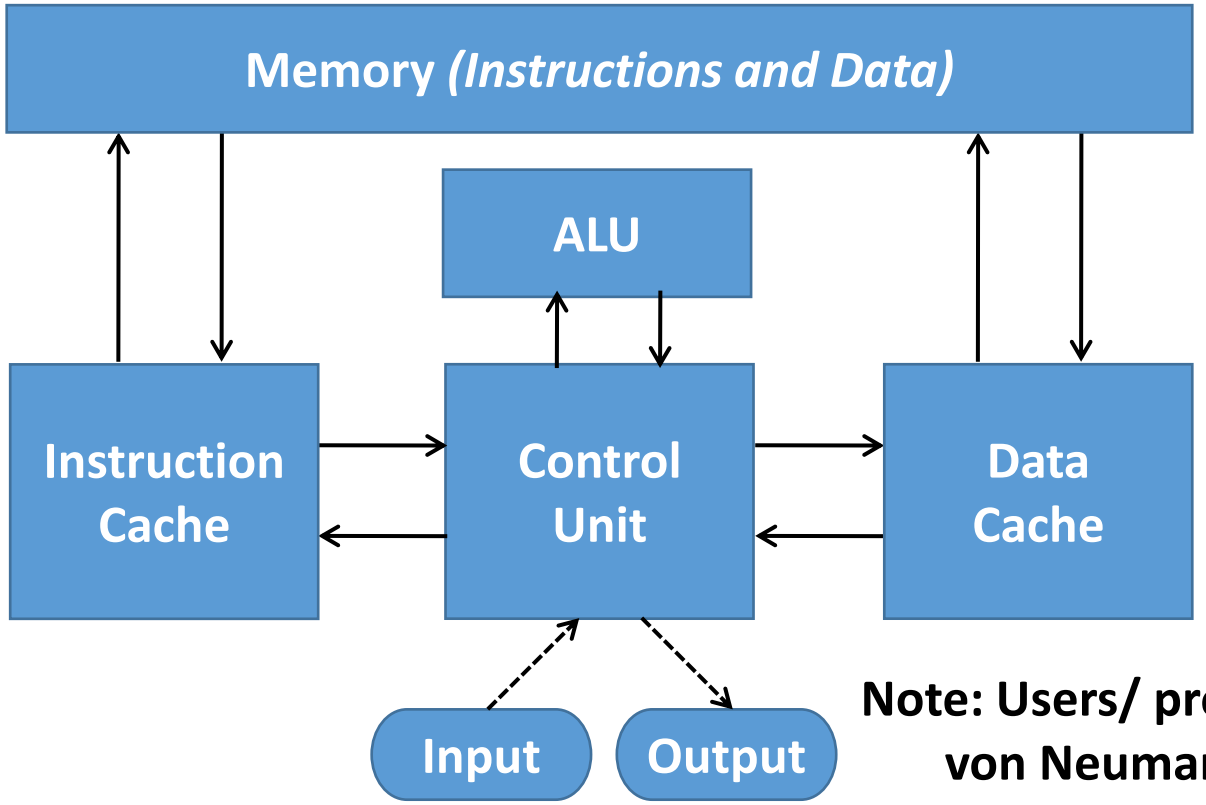
(*Harvard Architecture*)

- Instruction and data memories are *separate*; and we cannot modify instruction memory at run-time
- Can access instruction and data memory simultaneously, so may be faster, but more expensive than the von Neumann Architecture.



(*Modified Harvard Architecture*)

- Used in ARM9, MIPS, PowerPC, x86, …
- Separate instruction and data caches *internally*
- But a single unified main memory is still visible to users/ programs



Note: Users/ programs see von Neumann architecture

Information Coding

- Focus on hardware : handling **small, positive whole numbers**!

- Any data type is just a **code:**

  - ASCII , etc

- **Number representations:**

  -Binary, Decimal, Octal, Hex

- **Decimal, binary, hex, octal**
  - know WHY we use them (e.g. why we use hex?)
  - know how they are mapped (e.g. one Hex digit is a ”nibble” → i.e. 4 binary digits → i.e. half byte)
  - how conversion is done from one system to the other

Information Coding (cont..)

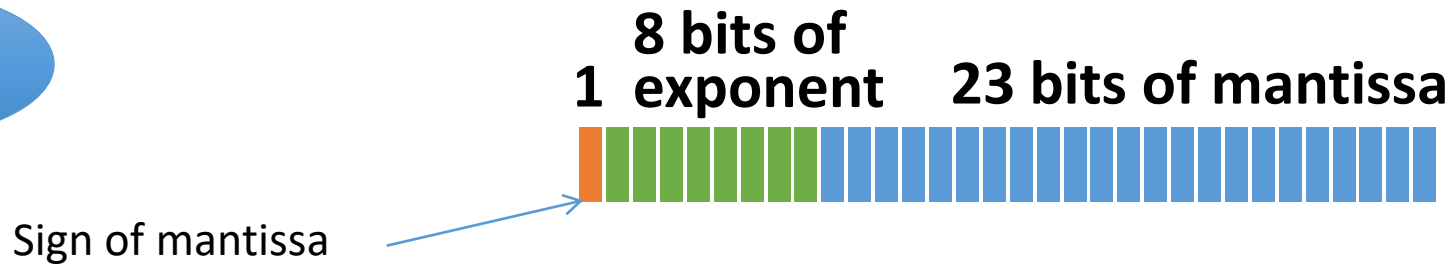
- Negative numbers representation (know their pros and cons):
  - Excess n
  - Sign and magnitude

- Know how addition/subtraction is done on all the systems –
  - **2’s complement**
  - **Remember the carry bit..overflow**

- Floating points …
  - Range vs. accuracy trade-off
  - **convert decimal/hex/ to binary/decimal/hex --> 32 bit IEEE 754**

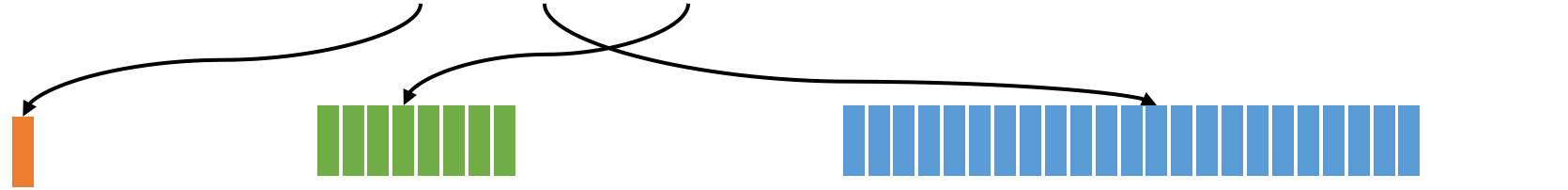
Convert decimal → IEEE 754

-0.75



decimal → binary (i.e. ½ + ¼) → normalised

-0.75 = -0.11 = -1.1 x 2^{-1}



1 | **01111110** | **1+ .1000000,00000000,00000000**

(-1 + 127 =) 126

10111111, 01000000, 00000000, 00000000

Boolean Logic

- For any pair of binary digits (bits) A and B…

AND

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

TRUE if, and only if, all inputs are TRUE

OR

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

TRUE if any input is TRUE

NOT

A	Q
0	1
1	0

TRUE if, and only if, the single input is FALSE

XOR

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

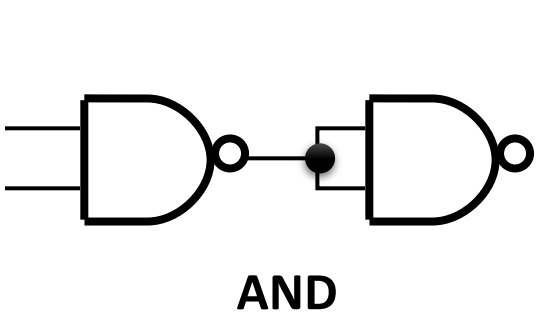
(eXclusive OR)

True if an odd number of inputs are TRUE, otherwise FALSE

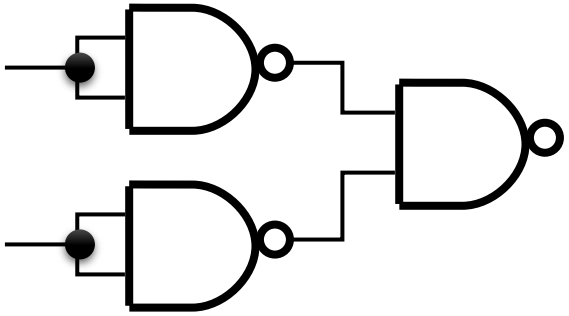
1 is interpreted as meaning TRUE; and 0 as FALSE

Building AND/OR/NOT from NANDs/NORs

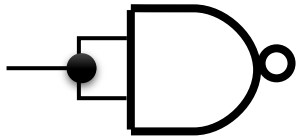
** **Make sure you know how basic logic gates are built by BOTH NAND and NOR**



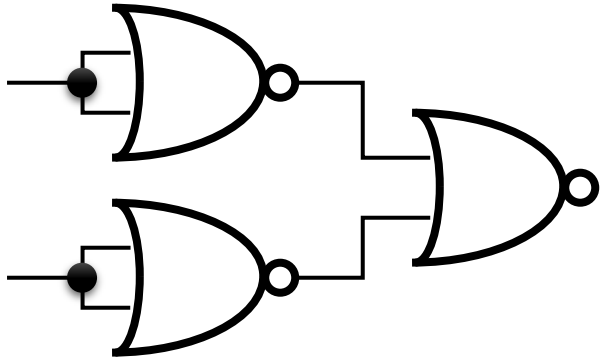
AND



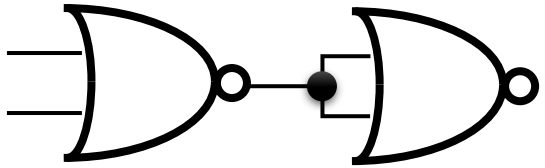
OR



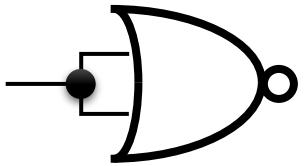
NOT



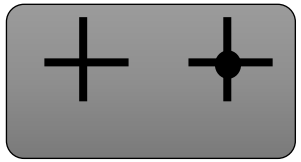
AND



OR



NOT



Crossing wires (no connection)

Junction (connection)

Background

Boolean algebra

	Law	AND form	OR form
proven on next slide	Identity 1	A = A''	A = A''
	Identity 2	1A = A	0 + A = A
	Null	0A = 0	1 + A = 1
	Idempotence	AA = A	A + A = A
	Complementarity	AA' = 0	A + A' = 1
familiar from School?	Commutativity	AB = BA	A + B = B + A
	Associativity	(AB)C = A(BC)	(A + B) + C = A + (B + C)
	Distributivity	A + BC = (A + B)(A + C)	A(B + C) = AB + AC
	Absorption	A(A + B) = A	A + AB = A
	de Morgan's law	(AB)' = A' + B'	(A + B)' = A'B'

Note relationship between AND and OR variants... Swap 0s and 1s, ANDs and ORs
...one function is the ***dual*** of the other – if a function is correct, its dual must also be

Boolean Algebra (example…walkthrough on the board…)

- F = XZ + Z(X’+XY) = …. ?
- F = (A+B+C)(D+E)’ + (A+B+C)(D+E) = …. ?

*****de Morgan's Law

- (AB)' = A' + B' (A + B)' = A'B' *Very useful*!

- Whenever we see an expression whose sub-expressions are all ANDed together, or all ORed together, we can re-state by
  1. negating the overall expression
  2. negating the sub-expressions
  3. flipping the operators from OR to AND, or vice versa

Using a Karnaugh map

1. Pick a template with the required number of inputs, and put a 1 in any square for which we want an output of 1
2. Look for *rectangular groups* of 1s
   - Groups must contain 2 or 4 or 8 … (2^n) cells
   - Groups may overlap, and may wrap around the edges
   - The *larger* the groups, and the *fewer* the groups, the better

Result: for each group simply list the "unchanged" terms and OR them together ("changed" ones "cancel")

A'BCD + A'B'CD +
A'BC'D + A'B'C'D + AB'C'D +
AB'C'D' +
ABCD' + A'BCD' + A'B'CD' + AB'CD'

= A'D + AB'C' + CD'

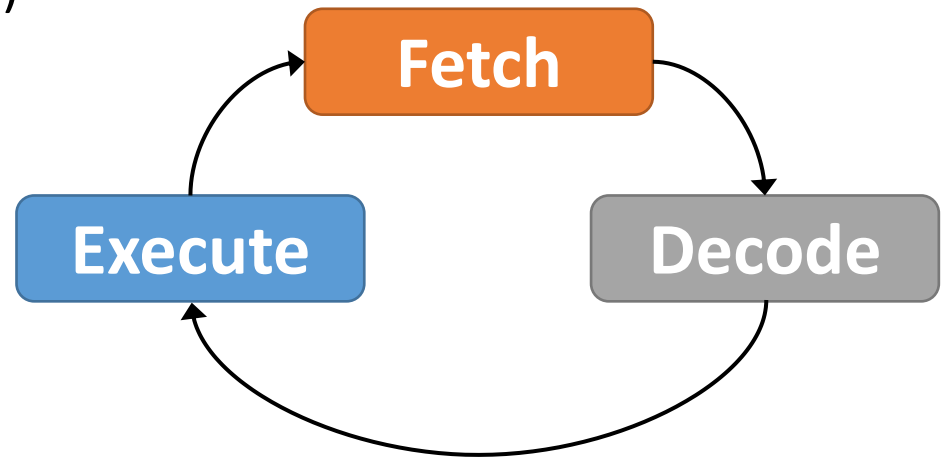
	AB	A'B	A'B'	AB'
CD		1	1	
C'D		1	1	1
C'D'				1
CD'	1	1	1	1

**** Digital Circuit design

- The basic approach is
  1. Write out a truth table for the desired logical function
  2. Derive a boolean expression by ORing together all the rows whose “output column” is 1
     - This is often called the ***sum-of-products*** form (cf. arithmetic “+”)
  3. Translate the Boolean expression to logic gates
     - May need to map to AND/OR/NOT gates or to NAND or NOR only
     - May need to use Boolean alegbra or “Karnaugh maps” to obtain the simplest mapping to our target types of logic gate
- Remember our examples with the stair-hall light, the 4-way multiplexer etc..
- **Make sure you are more than confident on all the simplification steps (e.g. De Morgan’s, Karnaugh map etc) that involve the design of a digital circuit….**

Basic processor operation: the fetch-decode-execute cycle

- All the CPU ever does in its life is endlessly repeat this cycle:
  1. Fetch the **next instruction** (from address held in the “PC” – program counter)
  2. Decode it
  3. Execute it



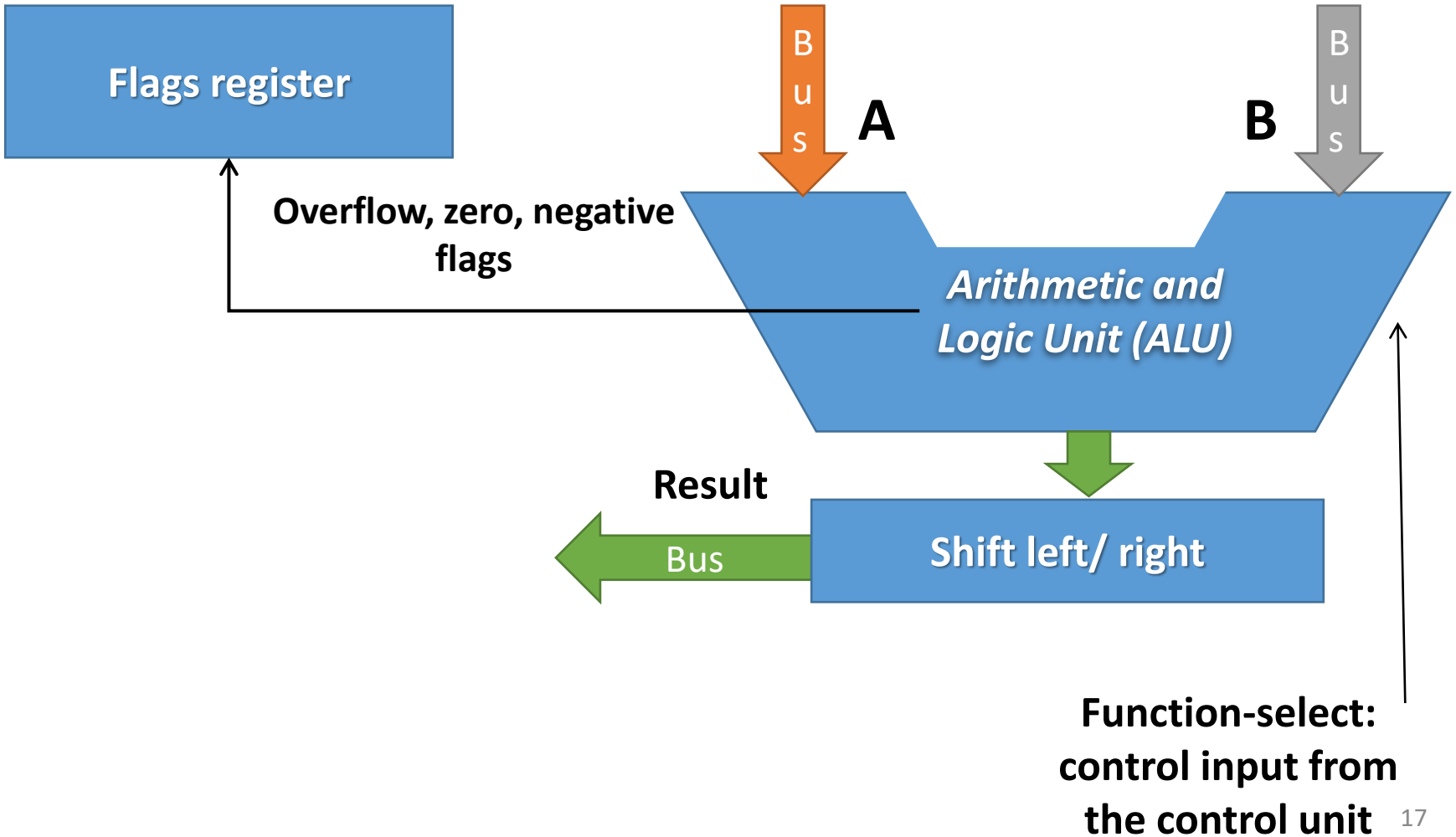
- We call this the ***fetch-decode-execute cycle***

Registers

- Registers are *holding areas for data being worked on inside the CPU*
  - Often (as in MIPS), arithmetic and logic instructions are designed to work *on registers only*, not on main memory
    - This is because registers are much *faster* than main memory
    - We use the data-transfer instructions for register ↔ main memory transfers

- General purpose and special registers
  - The registers used by the arithmetic and logic instructions are called **general purpose registers**
    - In computers like MIPS we have a largish set of these—a "file" of 32 x 32-bit registers
      - Because its registers are 32 bits wide, MIPS is said to have a *word size* of 32 bits
      - This is also the size of the basic unit of transfer between the registers and main memory (although main memory can also be accessed at the granularity of half-words and individual bytes)
  - Processors also have **special registers** such as the ***program counter* (PC)** and the ***stack pointer* (SP)…**

ALU


Flags register
Bus
A
B
Bus
Overflow, zero, negative flags
Arithmetic and Logic Unit (ALU)
Result
Bus
Shift left/ right
Function-select: control input from the control unit

ALU

- Know how:
  - Half adders and full adders work – if you know the half, the full is straight forward.
  - Ripple-carry works – carry-select adders ; the "*prediction*" performed on the *upper n/2 bits* using a set of 2 full adders with two inputs of 1 and 0.
  - The status and the overflow flag works
  - Remember: "*If the sign bits are the same but the result has a different sign, we have arithmetic overflow error*"
  - Bit shifting , left and right ;
  - Also….which are the ALU-specific status registers, their purpose and where they are located
    - Zero flag, negative flag, overflow flag (bits organised in a special register called flags register)

Memory

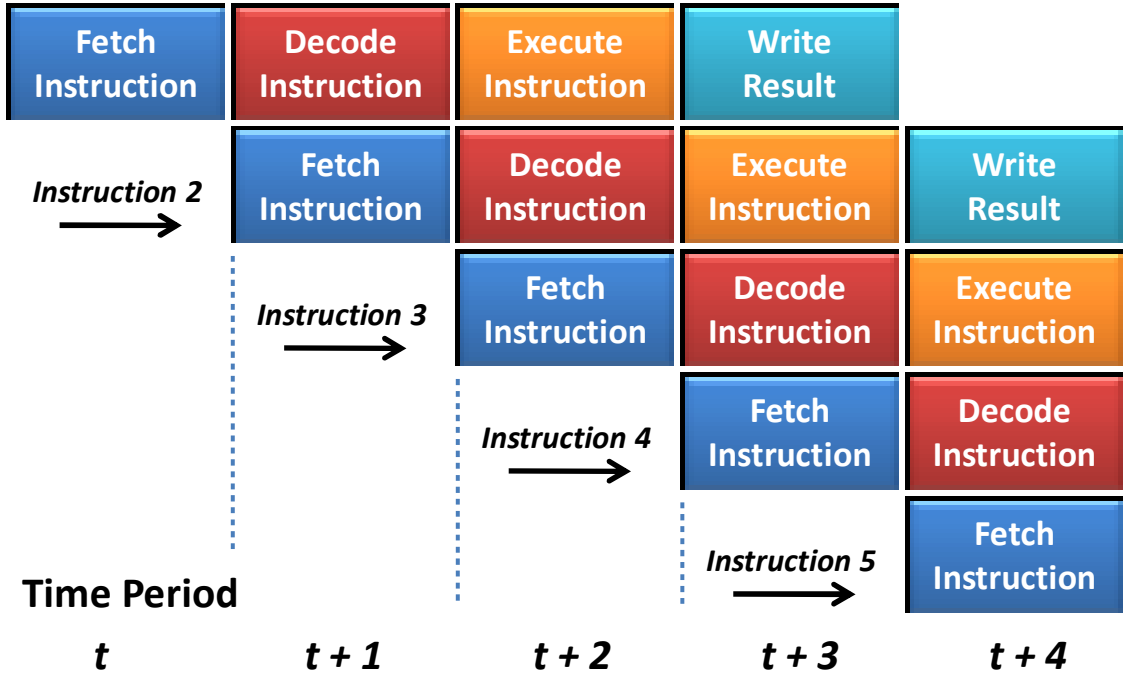
- Dynamic vs. Static memory
- Memory address decoding
- Combinatorial vs. Sequential logic
  - Combinatorial : function of its inputs
  - Sequential : involves **feedback**
- **How is feedback (i.e. sequential logic) built?**
- **S-R Flip Flop : how it works?**

What *is* the control unit?

- It can be thought of as a “little program” running inside the processor that **endlessly executes the fetch-decode-execute cycle**
- It controls and sequences the other architectural modules* using their respective *control lines* (e.g., function select, shift L/R, carry in, latch, output enable)
- The control unit is itself driven by a ‘clock’, which gives regular, timed electrical pulses or ‘ticks'
- The control unit’s fetch-decode-execute loop can be implemented in **two** alternative ways
  1. As a ***finite state machine (FSM)*****:** hard-wired sequential logic (built directly in terms of NAND gates etc)
     - high performance, but expensive and hard to evolve
  2. As ***microcode***: a sequence of **micro-instructions** in a **micro-memory**
     - slightly lower performance but much more flexible

Pipelining

- Pipelining is a widely-used way to exploit inherent **parallelism** inside the control unit to **speed up** the fetch-decode-execute cycle
  - If we can split the cycle into *n* sub-stages, we get *n* x speedup!
  - Cf. human workers on a factory production line



Input/Output (I/O)

- **Speed-gap challenge:** the huge speed difference between CPUs and I/O devices
- **Device diversity challenge:** the need to handle a variety of device types, including **character** and **block** devices
  - And we appreciate (very roughly!) how operating systems abstract over device classes using device drivers
- We understand, in outline, how data is stored and addressed on hard disks