Binary

- k bits can hold 2^k values.

- Big endian: $1\ 1\ 0\ 1_2 = 13_{10}$ (bits: $2^3\ 2^2\ 2^1\ 2^0$)

  Little endian: $1\ 1\ 0\ 1 = 11_{10}$ (bits: $2^0\ 2^1\ 2^2\ 2^3$)

  "There's a big market for big endian, as this is the "normal" format"

- Decimal to other base: **division method**
  Eg. $22_{10} \rightarrow ?_2$

$2\overline{)22}$

11 r 0	↑
5 r 0	↑
2 r 1	↑
1 r 0	↑
0 r 1	

$22_{10} = 10100_2$

- $0.101_2 = 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$
  $= 0.5 + 0.00 + 0.125$
  $= 0.625_{10}$

- Decimal fractional values to binary: **multiplication method**
  Eg. $0.625_{10} \rightarrow ?_2$

$0.625 \times 2 = 1.250$ → 1

$0.250 \times 2 = 0.500$ → 0

$0.500 \times 2 = 1.000$ → 1

0.000 → Stop.

$0.625_{10} = 0.101_2$

- Hex - Base 16 - $16 = 2^4$ (groups of 4 bits)
  Octal - Base 8 - $8 = 2^3$ (groups of 3 bits)

- Bin → Hex

  Eg. $0111\,0100\,1011\,1010_2 = 74BA_{16}$

  7_{10} 4_{10} 11_{10} 10_{10}

- **Unsigned Binary**

  - No sign, so assumed as +ve

  - Addition:

$$\begin{array}{r} 10110100 \\ 11011001 \\ \hline 110001101 \end{array} \quad +$$

* In an 8-bit system, this bit would be overflow.

  - Subtraction:

$$\begin{array}{r} {}^{0}\not{1}\,{}^{2}\not{0}\,1\,1 \\ 0\,1\,1\,0 \\ \hline 0\,1\,0\,1 \end{array} \quad - \quad \begin{array}{r} ?\,0\,{}^{0}\not{1}\,{}^{2}\not{0}\,{}^{2}\not{0} \\ 1\,0\,1\,1 \\ \hline 0\,1\,1 \end{array}$$

* Problem, as there is no 5th bit to borrow from. This is underflow.

  - Multiplication:

$$\begin{array}{r} 1011 \\ \times\ 101 \\ \hline 1011 \\ 0000 \\ 1011 \\ \hline 110111 \end{array}$$

- Division:

```
        0 1 1 1
    ---------------
110 ) 1 0 1 0 1 0
        0 0 0
      -------
          1 0 1 0
          1 1 0
        -------
            1 1 0 1
            1 1 0
          -------
              0 1 1 0
              1 1 0
            -------
                  0
```

- **Signed Magnitude.**

- 1 bit for sign, rest for number. For sign bit, 0 = +ve; 1 = -ve.

- A k-bit system has a range of $-(2^{k-1}-1) \rightarrow (2^{k-1}-1)$, which is $2^k - 1$ unique values.

- Addition:

```
  0 1 0 1          1 0 1 0
  0 0 0 1 +        1 0 1 1 +
  -------          -------
  0 1 1 0          1 1 0 1
      ✓                ✗
```

```
  0 1 0 0          1 0 1 1
  1 0 1 0 +        0 0 0 1 +
  -------          -------
  0 1 1 0          1 1 0 0
                       ✗ ✗
```

Sign bit is always symmetrical when adding 2 numbers.

* Be careful of answers falling outside of range *

One's Complement

- One bit for sign, rest for number. For sign bit, 0 = +ve; 1 = -ve
- A k-bit system has a range of $-(2^{k-1}-1)$ to $(2^{k-1}-1)$, which allows for 2^k-1 unique values.
- Positive numbers: 0 + num. in binary
  Eg. $5_{10} = 0101_2$ in 4-bit one's comp.
- Negative numbers: 1 + num in binary, bits flipped
  Eg. $-5_{10} = 1010_2$ in 4-bit one's comp.
- Addition:

```
   01101
   11010 +
 1 00111
       +1
   01000
```

Ignore any overflow on sign bit.

Always +1 in all cases.

* Be careful of answers falling outside of range *

Two's Complement

- One bit for sign, rest for number. For sign bit, 0 = +ve, 1 = -ve
- A k-bit system has a range of $-(2^{k-1})$ to $(2^{k-1}-1)$, allowing for 2^k unique values.
- Positive numbers: 0 + num. in binary
  Eg. $5_{10} = 0101_2$ in 4-bit Two's Comp.

- Negative numbers: 1's num. in binary, bits flipped, 1 added.

Eg. $+5_{10} = 0101_2$

$1010_2 \rightarrow -5_{10}$ Ones C.

$+1$

$1011_2 \rightarrow -5_{10}$ Twos C.

- Addition:

0100			0100		
0010	+		0110	+	
0110			1010		
			↑		
1110			1100		
1100	+		1010	+	
1010			0110		
↑↑			↑		

If bits n and n+1 are the same, this works.
If different, this doesn't work.

- Subtraction:

$$\begin{array}{r} 0010 \\ 0100 \\ 1011 \\ \hline 1100 \\ 0010 \\ \hline 1110 \end{array}$$

FLIP, +1 — COPY

ADD

* Be aware of answers that will be out of range. *

- Multiplication: use Booth's Algorithm

Eg.

```
      1011  ------------------>  01 -> A: 1111 1011
      0110  x                    10 -> S: 000 0101
-----------
0000 0000
0000 1010
0000 0000
1101 1000
-----------
1110 0010
-----------
  1 x
```

A is top line
S is top line in Two's Comp.
Sign-extend both to 2x bitspace.

Step in a bit each time

Pairs: 00 - Do nothing
01 - Use A
10 - Use S
11 - Do nothing

Add values as if they are unsigned, ignoring any overflow.

• IEEE-754

- Single: 1-bit sign, 8-bit exponent, 23-bit mantissa
- Double: 1-bit sign, 11-bit exponent, 52-bit mantissa

- Single has 127 bias on exponent.
  - To show exp. of 1, show $127+1 = 128_{10} = 1000\ 0000$
  - To show exp of -13, show $127-13 = 114_{10} = 0111\ 0010$
- Double has 1023 bias on exponent.

- Mantissa must be normalised:
  - $1011.1101 = 1.01111101 \times 2^{3}$
  - $0.00101 = 1.01 \times 2^{-3}$
    (1 must be 1; 01 → mantissa)

- Eg. $3.75_{10} = 11.11_2 = 1.111_2 \times 2^{1}$ Exp. $= 127+1 = 1000\ 0000_2$

↳ 0100 0000 0111 0000 0000 0000 0000 0000

Sign (0 = +ve, 1 = -ve) | Exponent | Mantissa

- Eg. $12.625_{10} = 1100.101_2$

$= 1.100101 \times 2^{3}$ → $127+3 = 130_{10}$

$= 1000\ 0010_2$

0100 0001 0100 1010 0000 0000 0000 0000

Main Computer Components

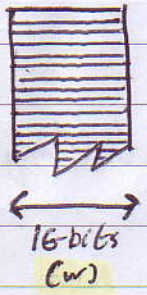
1) CPU
2) Memory
3) Bus
4) Clocks / Interrupts
5) I/O

1) CPU

- Two main parts: **data path** and **control unit**
- The **data path** is made up of two parts:
  - **Arithmetic Logic Unit (ALU)** - used for arithmetic operations and comparisons.
  - **Registers** - temporary storage areas used to store 'arrays of bits for the next operation.
- The **control unit** works like a traffic manager to tell the **ALU** what to do and handle events like interrupts.
  - The CU contains and uses registers including the program counter and the status register.

2) Memory

- Array of addressed slots, each with a word of data inside.
- Described as **length × width**, ie. "4M × 16" means 4M rows, each of 16 bits in length:



- Memory can be word- or byte-addressable.

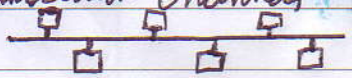
Log 4M means 2^{22} addresses, from $2^2 \times 2^{20} = 2^{22}$
(4 = 2, M = 20)

3) Bus

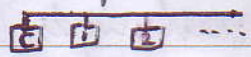
- Wire connections between components to move data around.

- Types:
  - Point-to-point
    - Fast and dedicated communication channels

  - Multipoint
    - Potential problems with collisions



- Bus arbitration is needed to avoid collisions - master/slave is common.
  - Types:
    - Daisy chain, priority list.
      - #1 master by default, #2 master if #1 is busy, etc. down the line
      - Problem: lower priority components can be crowded out.
    - Centralised parallel



      - Coordinator - single centralised arbitration circuit decides who can use the bus.
    - Distributed arbitration
      - Self-selection - components decide between themselves who uses the bus.
      - Collision detection - components send data, if collision, re-send.

- Lines on a bus:
  - Data lines
  - Control lines
    - Arbitration does its thing here
  - Address lines
  - Power lines.

4) Clocks

- Synchronises operations within machine incl. moving data and CPU operations
- Measured in MHz ($10^6 s^{-1}$) and GHz ($10^9 s^{-1}$).

4) Interrupts

- Alerts CPU to higher-priority events
- Triggered by arithmetic errors, div/0, I/O requests, over/underflow, program errors, etc.
- Each interrupt is associated w/ a specific procedure to deal w/ the event: Interrupt Service Routine (ISR)

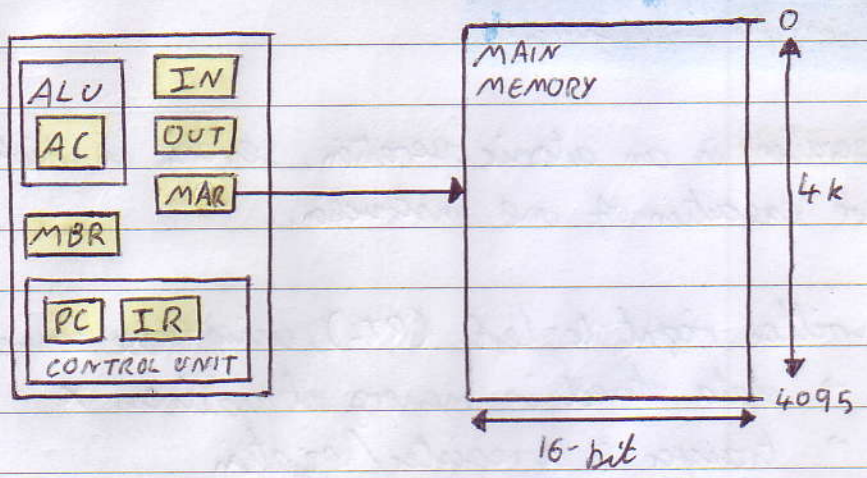
* More on Interrupt Processing later *

5) I/O Subsystem

- Communicate w/ outside world
- Memory-mapped
  - I/O interface appears to CPU like memory
- Instruction-based
  - Dedicated instructions for CPU to handle I/O device
  - Doesn't occupy memory, but requires instruction set

MARIE/Assembley

- Characteristics:
  - Uses 2's comp. binary
  - 4k x 16-bit word-addressable memory.
  - 16-bit ALU
  - 7 registers



- Common data bus connecting registers and memory, but with some P2P buses such as MAR ↔ Memory.

• ISA

- 16-bit operations - 4-bit opcode and 12-bit operand/address.

Inst. #	Inst.	Meaning
0001	LOAD X	Load value of X into AC
0010	STORE X	Store value of AC at address X
0011	ADD X	Add value of X to AC and store in AC
0100	SUBTR X	Subtract " " " from " " " " "
0101	INPUT	Read input value into AC
0110	OUTPUT	Send AC value to output
0111	HALT	Terminate
1000	SKIPCOND	Skip next line on condition *
1001	JUMP X	Takes value at X and places into PC.

* Skip Cond:

1000 1000 0000 0000

↳ 00 → skip if AC < 0

01 → skip if AC = 0

10 → skip if AC > 0

• Register Transfer Language

- A micro-operation is an atomic operation, several of which are required for execution of one instruction.

- RTL is written right-to-left (RTL), using these conventions:
  - M[x] → data stored in memory at address x
  - ← → transfer to a register/location

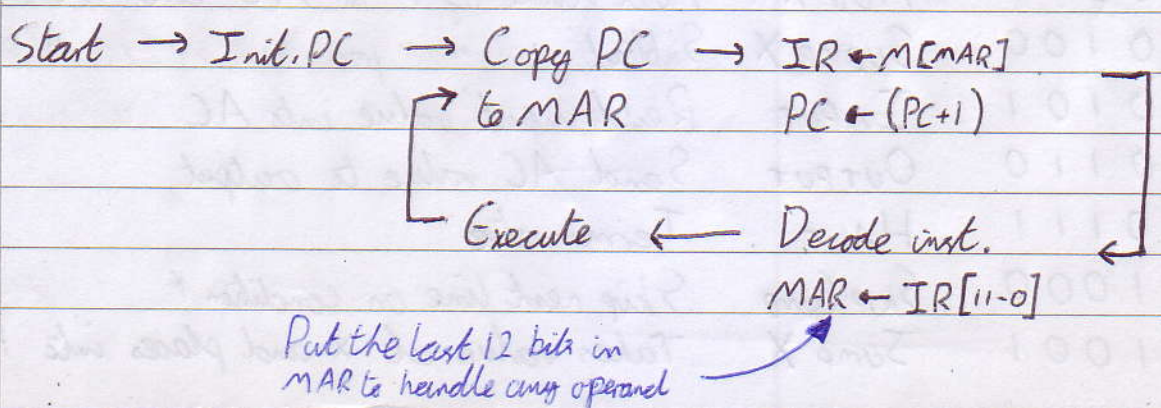
Eg. LOAD X : MAR ← X

MBR ← M[MAR]

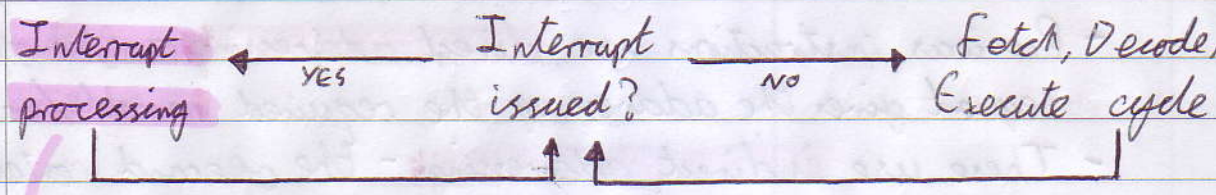
AC ← MBR

• Instruction Processing

- Fetch: get instruction from memory, load into IR.
- Decode: check op code, decide what to do, check for an operand, place operand into MBR
- Execute: execute the instruction



• Interrupts



Start
↓
Interrupt signal detected
↓
Save variables and registers
↓
Look up ISR address ← Interrupt Service Routine - unique for different types of interrupt
↓
PC ← ISR addr
↓
Branch to ISR
↓
Start F/D/E cycle on ISR ← Do what is needed to handle interrupt
↓
Restore variables and registers
↓
Return to F/D/E cycle

* Generally, interrupts are disabled during the ISR, because they are "mashable" interrupts. "Non-mashable" interrupts must be processed to keep the system stable.

• Extended Instruction Set

- Previous instructions use **direct addressing** - the operand segment gives the address of the required variable/value
- These use **indirect addressing** - the operand contains the address of a location in memory, which itself contains the address of the value required.

- ADDI X (Indirect Add)

```
MAR ← X
MBR ← m[MAR]
MAR ← MBR
MBR ← m[MAR]
AC ← AC + MBR
```

- JUMPI X (Indirect Jump)

```
MAR ← X
MBR ← m[MAR]
PC ← MBR
```

- CLEAR (Clear AC)

```
AC ← 0 (zero)
```

- JNS X (Jump and Store)

```
MBR ← PC
MAR ← X
m[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
PC ← AC
```

Store PC at X and jump to X+1.

Instruction Set Architectures (ISAs)

- **ISA Characteristics:**
  - Bits per instruction
  - Stack- or register-based
  - Operands per instruction
  - Operand location
  - Operand type(s)/size
  - Types of operations

- **ISA Measurements:**
  - Main memory space occupied by program
  - Instruction complexity
  - Instruction length in bits
  - Total instructions in instruction set

- **ISA Design Considerations**
  - Instruction length (short/long/variable)
  - Number of operands
  - Number of addressable registers
  - Memory organisation (byte- or word-addressable)
  - Addressing modes (direct, indirect, indexed, etc.)
  - Byte ordering/Endianness
    - **Big endian:**
      - More natural
      - Sign of the number can be found in byte at offset zero.
      - Strings and numbers stored in the same order.
    - **Little endian:**
      - Easier to place values on non-word boundaries
      - Conversion from a 16-bit integer to 32-bit does not require any arithmetic

• CPU Data Storage Structure

- The tradeoffs in these choices are simplicity and cost of hardware design, versus execution speed and ease of use.

- **Accumulator Architecture**
  - One operand of a binary operation is implicitly in the accumulator
  - One operand is in memory, creating lots of bus traffic

- **Stack Architecture**
  - Instructions and operands are taken from the stack
  - The result of a binary operation is implicitly stored on top of the stack
  - A stack cannot be accessed randomly.

- **General Purpose Register Architecture**
  - Registers can be used instead of memory
  - Faster than accumulator architecture
  - Efficient implementation for compilers
  - Results in longer instructions.

* Most systems today use a GPR *

- There are 3 types:
  - Memory - memory, where two or three operands may be in memory
  - Register - memory, where at least one operand must be in a register
  - Load-store, where no operands may be in memory.

• Architectural Differences

- A stack architecture requires different ordering for arithmetic operations.
  - Normally we use infix : $Z = X + Y$
  - Stacks require postfix : $Z = XY +$
  - Postfix makes brackets obsolete:
    - Infix : $Z = (A \times B) + (C \times D)$
    - Postfix : $Z = AB \times CD \times +$
    - In a stack ISA, this might look like :

```
PUSH A
PUSH B
MULT
PUSH C
PUSH D
MULT
ADD
POP Z
```

- In a one-address ISA (like MARIE), the infix $Z = (A \times B) + (C \times D)$ might look like:

```
LOAD  A
MULT  B
STORE TEMP
LOAD  C
MULT  D
ADD   TEMP
STORE Z
```

- In a two-address ISA this is:

```
LOAD  R1, A
MULT  R1, B
LOAD  R2, C
MULT  R2, D
ADD   R1, R2
STORE Z, R1
```

- In a three-address ISA this is:

```
MULT  R1, A, B
MULT  R2, C, D
ADD   Z, R1, R2
```

- Instructions fall into a range of broad categories:
  - Data movement
  - Arithmetic
  - Boolean
  - Bit manipulation
  - I/O
  - Control transfer
  - Special purpose

Memory Addressing

- Addressing modes specify where an operand is located
- They can specify a constant, a register or memory location
- The actual location of the operand value is called the effective address.
- Some addressing modes allow us to determine the address of an operand dynamically.

- Modes:

  - Immediate addressing - data is part of the instruction

  - Direct addressing - address of the data is given in the instruction

  - Register addressing - data is located in a register

  - Indirect addressing - the address of a location storing the address of the data is given in the instruction

  - Register Indirect addressing - a register stores the address of a location storing the address of the data

  - Indexed addressing - a register (implicitly or explicitly) is used as an offset, which is added to the address in the operand to determine the effective address.

- **Based addressing** - similar to indexed addressing, but a base register is used instead of an index register.

- **Stack addressing** - the operand(s) is/are assumed to be on top of the stack

- Difference between **indexed** and **based**:
  - An **index** register holds an offset relative to the address given in the instruction
  - A **base** register holds a base address where the address field represents a displacement from this base.

Example:

Memory:
800 → 900
....
900 → 1000
....
1000 → 500
....
1100 → 600
....
1600 → 700

Register:
R1 → 800

LOAD 800

↓

Accumulator:

Mode	Value
Immediate	800
Direct	900
Indirect	1000
Indexed	700

- **Index:** instruction gives an address, added to the value in a register.
- **Based:** instruction gives a value, added to the address in a register.

Memory

- Two main kinds:
  - RAM
    - Dynamic RAM (DRAM) - cheap, simple design
    - Static RAM (SRAM) - fast, used in caches
  - ROM
    - Low power consumption

- Hierarchy

- Faster = More expensive
- Faster elements are closer to the CPU:
  - Register
  - L1 Cache
  - L2 Cache
  - Main memory

  } System

  - Fixed rigid disk - Online
  - Optical disk - Near line
  - Magnetic tape - Offline

- To access data, CPU checks closest memory first (caches) and works away until it finds a match
  - Once located, the data and nearby elements are fetched into cache.

- "hit" / "miss" → data is found/not found
- "hit rate" / "miss rate" → % of hits/misses for a given level
- "hit time" → time to access data at a given level
- "miss penalty" → time to process a miss, including replacing a block of memory and delivering data to processor

- After a hit, entire blocks are copied, because locality tells us nearby bytes will be needed soon.

- Types of locality:
  - Temporal → recently-accessed data tends to be accessed again
  - Spatial → accesses tend to cluster
  - Sequential → instructions tend to be accessed sequentially.

• Cache Memory

- Stores recently-used data near CPU to speed up access

- Smaller and faster than main memory

- Content-addressable, so doesn't use normal addresses
  - Smaller is sometimes better, as it takes a long time to search
  - The "content" that is addressed is a subset of bits from the main memory address called a field
  - The fields provide a many-to-one relation between the main memory and the cache
  - A tag field in the cache block distinguishes one memory block from another.

• Direct Mapped Cache

- Simplest type of cache.

- In an N block cache, block X of main memory maps to X mod N, so in a 10 block cache, block 7 maps to 7, 17, 27, etc. of main memory.

- Once a block is loaded, a valid bit is set to show that the block contains valid data.

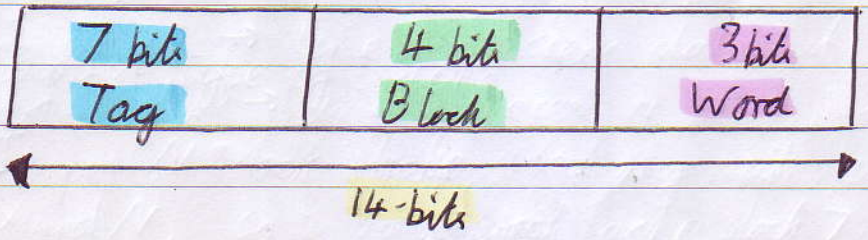
- Example cache:

Block	Tag	Data	Valid
0	0000 0000	Words A, B, C...	1
1	1111 0101	Words L, M, N...	1
2	—	?	0
3	—	?	0

- Block 0 contains multiple words from memory identified with the tag 0000 0000.
- Block 1 contains words identified with the tag 1111 0101.
- Blocks 2/3 are not valid.

- The size of the field into which a memory address is divided depends on the size of the cache.
  - Suppose memory has 2^{14} words, cache has 16 blocks, and a block has 8 words.
  - Memory has $2^{14}/2^3 = 2^{11}$ blocks
  - A main memory address requires 14 bits
  - The three rightmost bits are needed for the word (3 bits are needed to identify one of 8 words in a block)
  - We need 4 bits to select one of the 16 cache blocks.
  - The remaining 7 bits make up the tag field.

  - This gives this format:



- Example: Main memory address: 1AA
  In 14-bit binary : 00 0001 1010 1010

This is split up as:

0000011	0101	010
Tag	Block	Word

All 8 words of 00000101 000 - 00000101 111 are loaded into block 0101 of cache with the tag 0000011.

1AA, 1AB, 1AC, etc... will all give cache hits

3AB will cause the whole block to be evicted and replaced.

If the program requests 1AB, 3AB, 1AB, etc., the advantage is lost.

- Disadvantage of direct mapping: repeated calls to different main mem. blocks that would occupy the same cache block destroy any theoretical advantage.

• Fully Associative Cache

- A block can go anywhere in the cache
  - The cache can be filled before eviction is necessary.

- Memory address has only two fields: tag and word.

- Suppose again we have 14-bit memory addresses, a 16 block cache and an 8-word block.
  The format is: 11 bits (Tag, the rest) 3 bits (Word, to reference 8 unique words)

- All tags are searched in parallel to retrieve data quickly
  - Requires complex, expensive hardware.

- When the cache is full, a block must be evicted.
  - This is the victim block, chosen by methods discussed later.

• Set-Associative Cache

- Combines the ideas of direct mapped / fully associative.
- A main memory block maps to a limited set of cache blocks, but not "any" block, as with fully-assoc.
- The number of blocks per set depends on system design
- Example: a 2-way set-assoc. cache
  - Each set contains two different blocks:

Set	Tag	Block 0 of Set	Valid	Tag	Block 1 of Set	Valid
0	0000 0000	A, B, C...	1	—	?	0
1	1111 0101	L, M, N...	1	—	?	0
2	—	?	0	P, Q, R...		1
3	—	?	0	T, U, V...		1

- A memory address takes 3 fields: tag, set and word.
  - As before word identifies the word in the block and tag uniquely identifies the block.
  - Set determines the set to which the memory block maps
- Example: Main memory of 2^{14} words, 2-way set-assoc. cache of 16 blocks and 8 words per block.
  - 16 cache blocks and 2-way cache = 16/2 = 8 sets.
  - Thus 3 bits for word, 3 bits for set and the leftover 8 bits for the tag.

• Replacement Policies

- An optimal policy would look into the future and determine the block that won't be needed for the longest time.
  - Impossible, but used as a benchmark.

- Choice of policy used depends on type of locality being optimised. usually temporal

- Least Recently Used (LRU)
  - Evicts the block unused for the longest time
  - A history must be maintained - this slows the cache

- First in, First Out (FIFO)
  - Evicts the block that has been in cache longest, despite recent use rates.

- Random
  - Can evict a block needed soon, but avoids thrashing

- These policies must consider dirty blocks that have been updated while in cache. These must be written back to memory.
  - Two write policies:
    - Write through: cache/mem updated on every write
      - Reliable, but slows down updates (most accesses are read, so often negligable)
    - Write back: updates memory only when block selected for replacement
      - Less traffic, but memory may not match cache (bad if there are many concurrent users).

• Effective Access Time (EAT)

- EAT is a weighted average taking into account the hit rate and relative access time of different levels of memory.

- The EAT for 2-level memory is:

$$EAT = H \times Access_C + (1-H) \times (Access_C + Access_{MM})$$

When we hit

Access cache

When we miss

Access cache, get nothing, so access main mem.

- H = hit rate (%)
- $Access_{C/MM}$ = Cache/Main Mem. access times.

- The equation can be extended for many levels of memory.

• Types of Cache

- We have discussed a unified/integrated cache, with data and instructions together.

- Most systems use separate caches
  - Called a Harvard Cache
  - Better locality, but more complex
  - Bigger caches can give similar improvements without complexity

- A victim cache holds recently evicted blocks to help improve performance.

- A trace cache holds decoded instructions for program branches, giving the illusion that noncontiguous instructions are contiguous

- Many systems use multiple caches, creating a mini memory hierarchy.
  - Inclusive caching: data can exist on multiple cache levels
  - Strictly inclusive: data on one level must exist on all levels below (ie. all larger caches)
  - Exclusive caching: only one copy of data allowed.
  - Tradeoffs of the three models include access time, memory size and circuit complexity

Virtual Memory.

- Enhances performance by expanding memory capacity without adding memory.

- A portion of the disk serves as an extension of main memory.

- If the system uses paging, VM partitions the main mem. into individually managed page frames that are written (paged) to disk when not immediately needed.

- A physical address is the actual memory address of physical memory
- Virtual addresses are mapped to physical addresses by the memory manager.
- Page faults occur when a logical address requires a page be brought in from memory.
- Memory fragmentation occurs when the paging process results in small, unusable clusters of memory addresses.

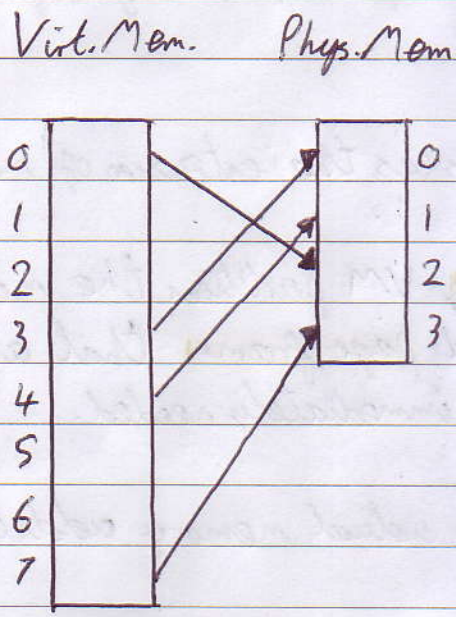
- Main and virtual memory are split into equal sized pages.

- A process's entire required address space need not be all in main mem. at once - some can be on disk

- Pages allocated to a process do not need to be stored contiguously - either on disk or in memory.

- Only the needed pages are in memory - unneeded pages are on slower disk storage.

- Info on the location of each page - disk or mem - is stored in a page table:



Page Table

Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

- A process generates a virtual address and the OS translates it to a physical address.
  - To do this, the virtual address has two fields: the page and the offset.
  - Page determines the page; ofset determines the location in that page.
  - Logical page number is translated into a physical page frame through a look-up in the page table.
  - If the valid bit is 1, the virtual page number is replaced w/ the physical page number.
  - If the valid bit is 0, the page must be fetched from disk.
    - This is a page fault
    - If necessary, a page is evicted from memory, replaced with the page from disk and the valid bit set to 1.
  - The data is then accessed by adding the offset to the physical frame number.

• EAT

$$EAT = (1-F) \times (2 \times Access_{mm}) + F \times Access_{D}$$

Where F = page fault rate (%)
$Access_{mm}$ = Main mem access time
$Access_{D}$ = Time to load a page from disk and access data

2 memory accesses: one for page table, one for the actual data

- This double access affects EAT.
  - Because page tables are read constantly, they are stored in a translation look-aside buffer (TLB) cache that stores the mapping of virtual pages to physical pages in a special associative cache.

• Segmentation

- Instead of equal-size pages, virtual address space is divided into variable-length segments (often under control of the programmer).

- Segments are located through the segment table, containing the segments mem. location and bounds (thus indicating size).

- After a page fault the segment is loaded from disk and the OS searches for a location in memory large enough to hold the segment.

• Issues

- Paging and segmentation can cause fragmentation
  - Paging is subject to internal fragmentation, where a process may not need all of the space in a page, leaving many pages with unused fragments.
  - Segmentation is subject to external fragmentation, where contiguous chunks of memory become broken up as segments are allocated and deallocated.

- Large page tables are slow, but with their uniform memory mapping, page operations are fast.
- Segmentation allows fast access to the segment table, but segment loading is labour-intensive.

- Each segment has a page table. This means a memory address has three fields: segment, page, offset.

• Summary.

- Cache memory gives faster access to main memory.
- Virtual memory uses disk storage to give the illusion of larger main memory.
- Cache maps blocks of main mem. to blocks of cache mem.
- VM maps page frames to virtual pages.
- Three general types of cache
  - Direct mapped
  - Fully-associative
  - Set-associative
- With fully-assoc, set-assoc and VM, replacement policies must be in place
- Replacement policies include LRU, FIFO and random.
  - These must also take into account dirty blocks.
- VM must deal with fragmentation
  - Internal for paging
  - External for segmentation

Instruction-Level Pipelining

- Some CPUs split Fetch/Decode/Execute into smaller steps.
- These steps can be executed in parallel to increase throughput - this is called instruction-level pipelining (ILP).

Example

Suppose the F/D/E cycle had these six stages:

S1) Fetch instruction
S2) Decode opcode
S3) Determine effective address of operands
S4) Fetch operand(s)
S5) Execute instruction
S6) Store result

Suppose we also have a 6-stage pipeline. The steps for each instruction can be overlapped, as so:

Clock cycle:	1	2	3	4	5	6	7	8	9	10
Inst. #1:	S1	S2	S3	S4	S5	S6				
Inst. #2:		S1	S2	S3	S4	S5	S6			
Inst. #3:			S1	S2	S3	S4	S5	S6		

Speed Up

- The theoretical speedup can be determined:
  - Let t_p be the time per stage.
  - Each instruction represents a task, T, in the pipeline.
  - The first task takes $k \times t_p$ to complete in a k-stage pipeline.
  - The remaining tasks complete at a rate of one per cycle, so the remaining time is $(n-1)t_p$, assuming there were n tasks in total.

- Therefore, to complete n tasks in a k-stage pipeline requires:

$$(k \times t_p) + (n-1)t_p = (k+n-1)t_p$$

- Speedup can be calculated as:

$$\text{Speedup } S = \frac{nkt_p}{(k+n-1)t_p}$$

← Normal time
← Pipeline time

- As $n \to \infty$, $(k+n-1) \to n$, so the theoretical speedup is:

$$\text{Speedup } S = \frac{kt_p}{t_p} = k$$

• Assumptions

- These equations make a few assumptions:
  - The architecture supports fetching instructions and data in parallel
  - The pipeline can be kept filled all the time, which is not always the case
    - Pipeline hazards can cause conflicts, flushes or stalls

• Pipeline Hazards

- Resource conflicts
- Data dependencies
- Conditional branching/jumping

* More on hazards due to branching later *

Benchmarking and Performance

• Performance Equation

- Describes CPU performance

- $$\text{CPU time} = \frac{\text{seconds}}{\text{program}}$$

$$= \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

(cycles / instruction: average)

(seconds / cycle: Affected by clock speed)

(instructions / program and cycles / instruction: Affected by ISA design)

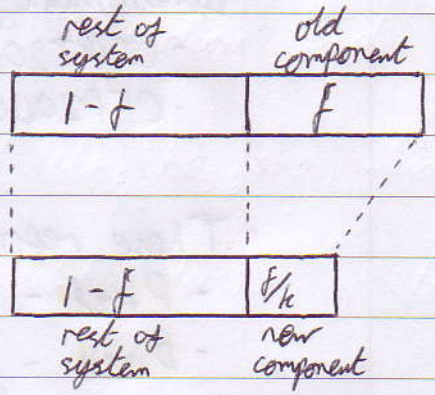
• Amdahl's Law

- Describes the speedup effect of one component in a whole system.

- $$\text{Overall speedup } S = \frac{\text{old exec. time}}{\text{new exec. time}}$$

$$= \frac{(1-f) + f}{(1-f) + f/k}$$

$$= \frac{1}{(1-f) + f/k}$$



Where...

f is the fraction of work done by one component

k is the speedup factor of that component

• Benchmarking

- ISA, CPU speed, etc. all affect performance in complex ways
- Benchmarking allows a direct, comparable measure of performance independently of these factors

- **Synthetic Benchmarks:**

  ◦ Whetstone / Dhrystone - library calls, math routines
  ◦ Linpack - linear algebra routines

  ◦ These were easy to understand, but easy to optimise for and thus not useful.

- **Improved Benchmarks:**

  ◦ A group called SPEC designed different forms of benchmark for computer/applications, including:
    - CINT2000 - tested CPU integer operations
    - CFP2000 - tested CPU floating point operations

  ◦ These report peak and base performance:
    - Peak - compiler optimisation allowed
    - Base - compiler optimisation not allowed

• Factors Affecting Performance

- ISA Design

- Two main notions in ISA design optimisation:
  - RISC - Reduced Instruction Set Computing
  - CISC - Complex Instruction Set Computing

RISC	CISC
- Few, fixed-length instructions	- Many, variable length instructions
- Same number of clock cycles to execute each instruction	- Variable number of clock cycles to execute an instruction
- Memory access only through LOAD/STORE operations	- Any instruction can access the memory
- Easier to hardwire	- Requires microcode to interpret instructions as they arrive
- Requires more registers	

- Performance = $\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$

CISC aims here ↑ (instructions/program)

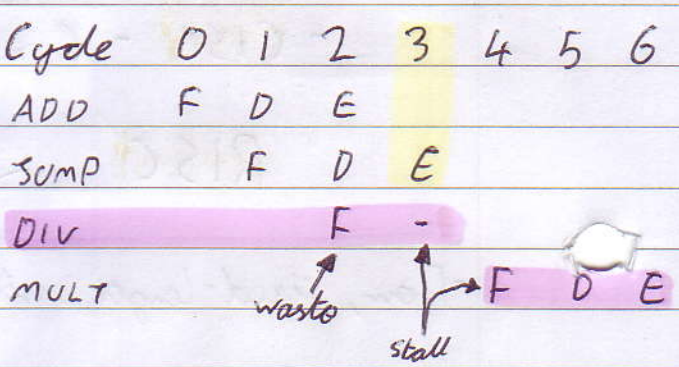
RISC aims here ↑ (cycles/instruction)

- Pipelining

- Can have a +ve impact; see earlier

- Conditional branching/jumping causes hazards

Eg: ADD ...
JUMP Loop
DIV ...
Loop, MULT ...



In cycle 3 the jump was executed and the program branched to MULT, however in cycle 2 the DIV was fetched creating wasted work and a stall in the stack

- There are methods to combat this:
  - Delayed branching - order of instructions is adjusted to avoid hazards and improve speedup
  - Branch prediction - the outcome of a branch is "guessed" before it is executed

- Hard Disk Usage

- HDDs create bottle necks as they are often the slowest parts

- Disk utilisation = $\frac{\text{request arrival rate}}{\text{disk service rate}}$ ← Rate of "asks" in requests/second ← Rate of "replies" in I/O ops/second

- This is the probability the disk is busy when an I/O request arrives.

- Request wait time = $\frac{\text{service time} \times \text{disk utilisation}}{1 - \text{disk utilisation}}$

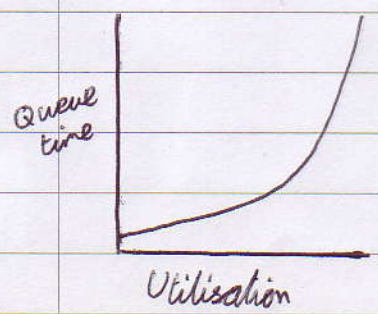
- Eg. Service time = 0.015_s
  Service rate = $\frac{1}{\text{service time}} \simeq 66/\text{sec}$
  Request rate = $33/\text{sec}$
  } Utilisation = $\frac{33}{66} = 0.5$

$$\text{Queue time} = \frac{0.015 \times 0.5}{0.5}$$
$$= 0.015_s \quad (= 15_{ms})$$

Change request rate to $60/\text{sec}$

$$\text{Utilisation} = \frac{60}{66} \simeq 0.9$$

$$\text{Queue time} = \frac{0.015 \times 0.9}{0.1}$$
$$= 0.135_s \quad (= 135_{ms})$$



Rule of thumb: 80% is seen as the max. acceptable utilisation

- Hard Disk Scheduling Approaches

- First Come, First Serve (FCFS)
  - + Simple and fair
  - - Large, random seeks
  - - Inefficient

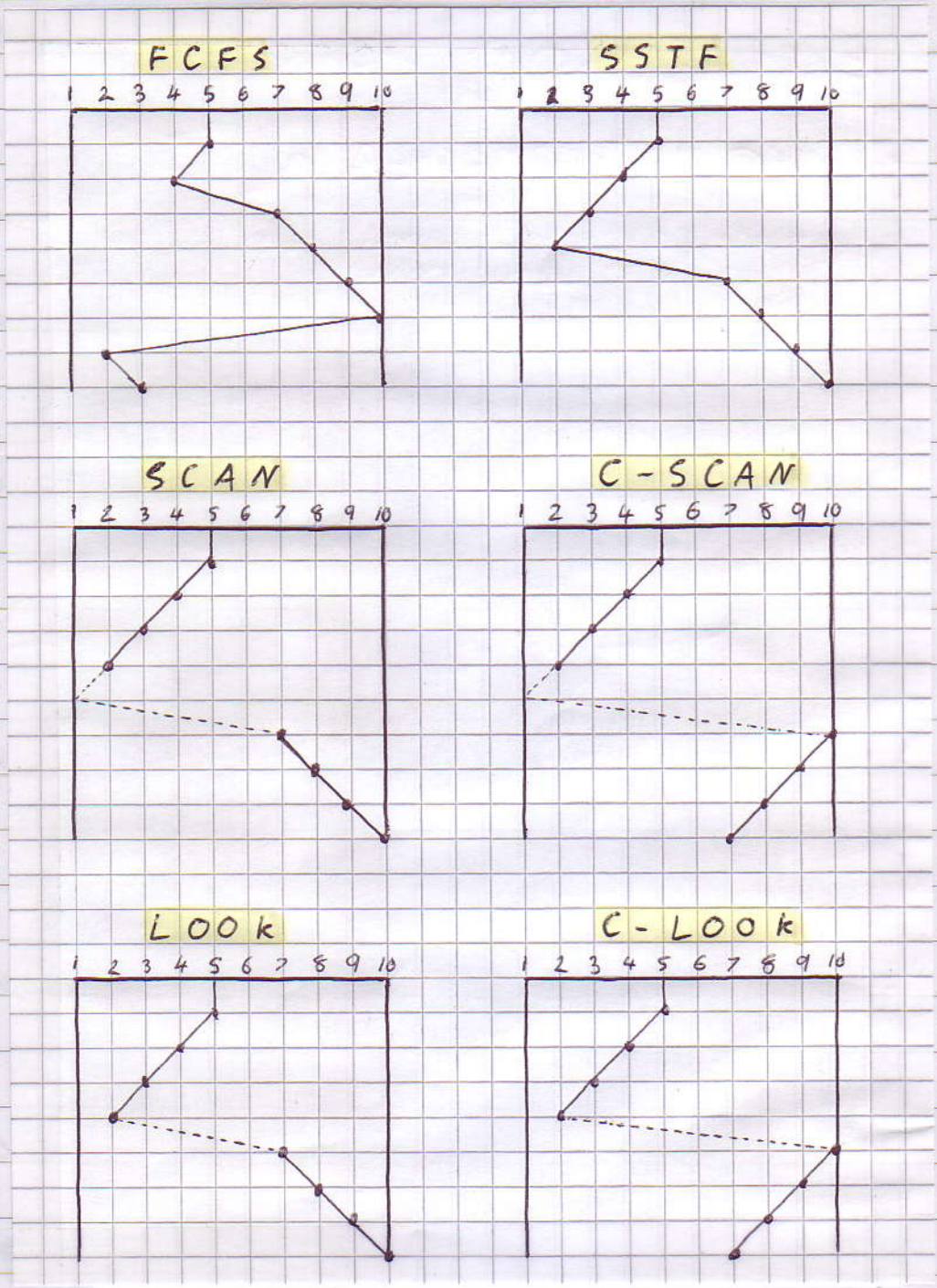
- Shortest Seek Time First (SSTF)
  - + Minimises arm movement
  - - Unfair, as remote requests wait a long time

- SCAN
  - Always scans the entire disk
  - + Predictable, fair
  - - Extra arm motion

- C-SCAN
  - Like SCAN, but wraps around
  - + Simpler to implement

- LOOK / C-LOOK
  - Like SCAN/C-SCAN, but only scans to outmost track in queue.
  - + LOOK has best theoretical performance
  - - LOOK is most complex to implement

- Examples:
  - Assuming the queue is: 5, 4, 7, 8, 9, 10, 2, 3
  - Assuming the read head started at 5



- Disk Cache Memory

- Prefetching of data from disk into a localised cache
  - Relies on principle of locality

- Prefetching is subject to cache pollution, when the cache is filled with data that no process needs, leaving less room for useful data.
  - Replacement algorithms like LRU, LFU and random combat this
  - Some systems evict bytes once they are written to the disk (the cache is also a staging area for data going to the disk).

- The disk cache is volatile, but app. believes data is written to disk
  - Power outage = lost data
  - Some have batteries

- Another approach is write-through
  - Data stays in cache but is written to disk immediately
  - OS is signalled that I/O is done only when data is on disk
  - Performance ↓, Reliability ↑

- If throughput is more important than reliability, some use write-back

- Some employ opportunistic writes
  - Dirty blocks remain in cache until arrival of a request for the same cylinder
  - The write operation is then "piggybacked" onto the request
  - Reduces read performance, but increases for write.