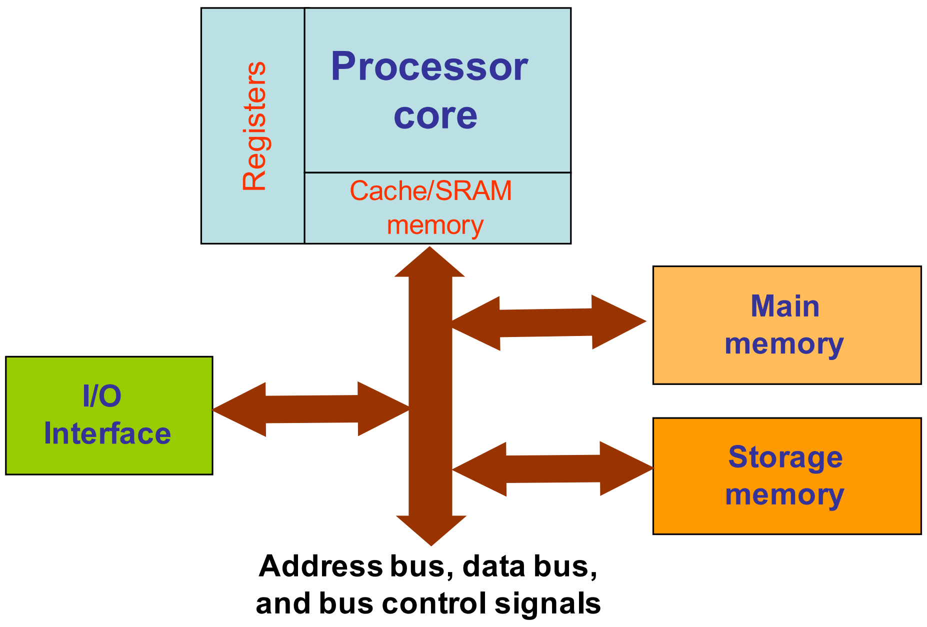
Processor General Concepts

Basic Processor-Based System



System Components

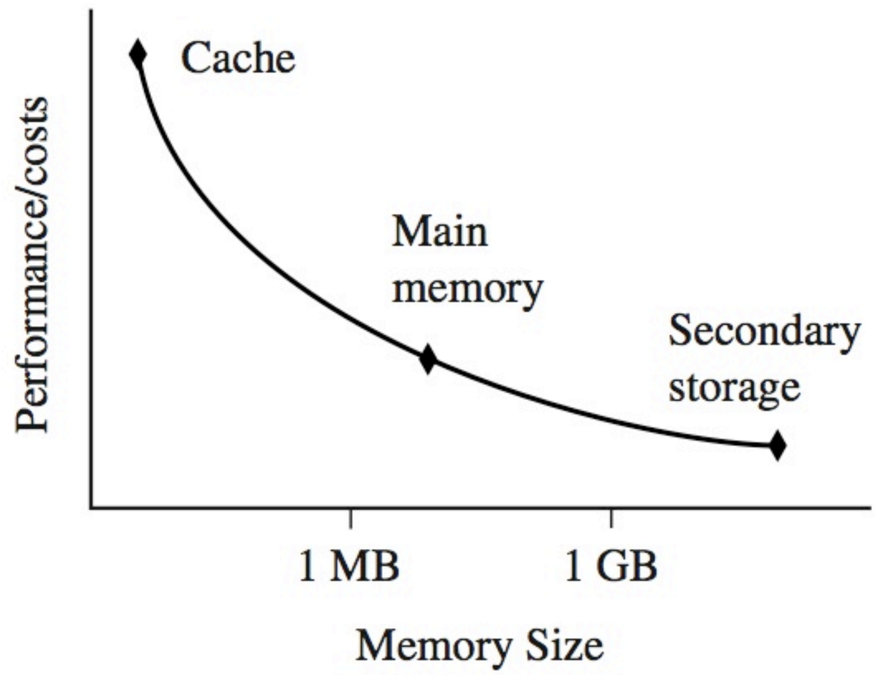
The basic components:

a) Processor with its associate temporary memory (registers and cache if available) for code execution
b) Main memory and secondary memory where code and data are temporary and permanently stored
c) Input and output modules that provide interface between the processor and the user

Connected through an interface bus consists of Address, Data, and Control signals

- e.g. AMBA bus for the ARM-based processor

Memory Hierarchy



Memory Hierarchy

A typical processor is supported by:

- on-board main memory (e.g. SDRAM up to GB)
- on-chip cache memory (e.g. SRAM KB to MB)
- on-chip registers

Some processors also provide general purpose on-chip

- SRAM (e.g. embedded processor) which may be configured as SRAM/Cache combination (e.g. TI's DSP)

Typically, a processor also utilizes secondary non-volatile memory

- for permanent code and data storage like Flash-based memory and hard disk

Address Space

Address space of a processor depends on its address decoding mechanism

•size will depend on the number of address bit used

Depending on the processor design, there may be two types of address space

•one is used by normal memory access

•another one is reserved for I/O peripheral registers (control, status, and data)

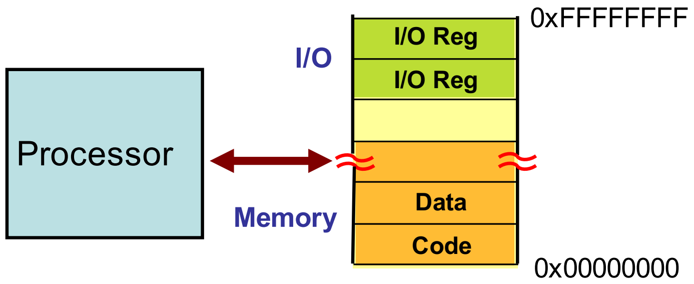
•need extra control signal or special means of accessing the alternate address space

Address Space (cont' d)

Refer to the range of address that can be accessed by the processor determined by the number of address bit utilized in the processor architecture.

Some processor families (e.g. ARM) utilize only one address space for both memory and I/O devices

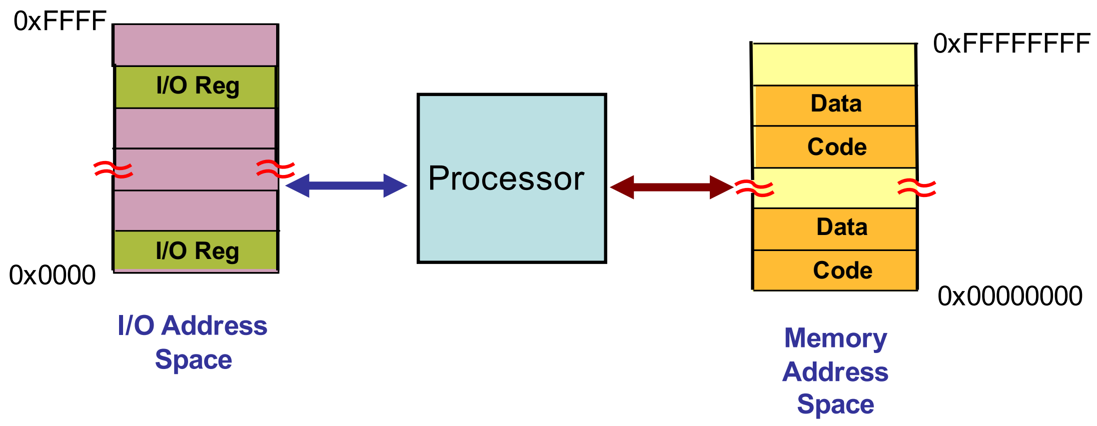
•i.e. everything is mapped in the same address space



Memory Mapped vs I/O Mapped

Some processor families have two address spaces.
E.g., for the x86 processor, memory and I/O devices can be mapped in two different address spaces:

•memory address space and I/O address space



Memory System Architectures

Two types of information are found in a typical program code:

i. Instruction codes for execution

ii. Data that is used by the instruction codes

Two classes of memory system design to store these information:

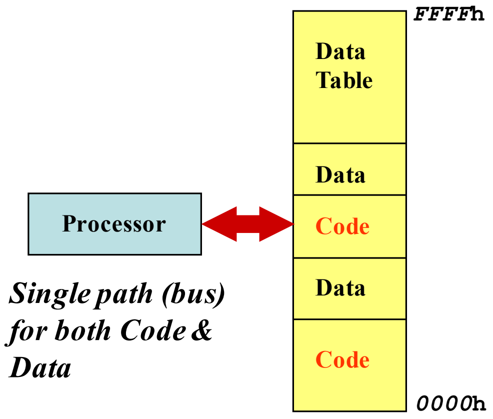
i. von Neumann architecture

ii. Harvard architecture

von Neumann Architecture

The von Neumann architecture utilizes only one memory bus for both instruction fetching and data access

- simplifies the hardware and glue logic design
- code and data located in the same address space



von Neumann Features

Single memory interface bus

- simplifies the hardware and glue logic design

More efficient use of memory

- code and data can reside in the same physical memory chip

More flexible programming style

- e.g. can include self-modified code

But data may overwrite code (e.g. due to program bug)

- need memory protection (e.g. hardware-based MPU)

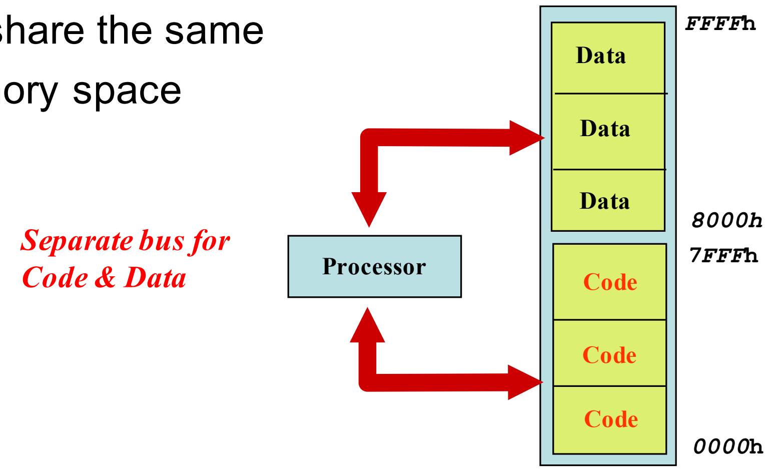
Bottleneck in code and data transfer

- only one memory bus for both data and code fetching

Harvard Architecture

The **Harvard** architecture utilizes separate instruction bus and data bus

•code and data may still share the same memory space



Harvard Features

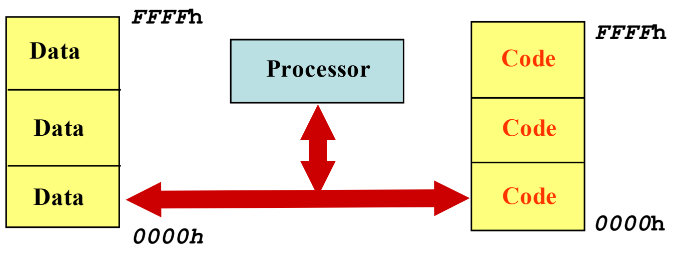
Separate instruction and data bus

- allow code and data access at the same time which gives improved performance
- provide better support for instruction pipeline operation and shorter instruction execution time
- allow different sizes of data and instruction to be used which results in more flexibility
- do not incur any code corruption by data which makes the operation more robust

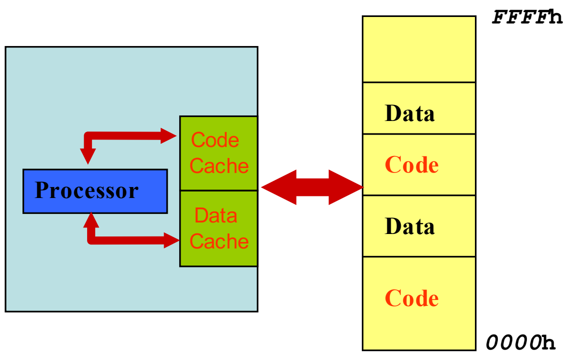
Requires TWO Bus Controllers – Logic Interfaces between Processor and Memory.

Architecture Variations

Independent data and code memory but with one shared bus (e.g. 8051)



*Two separate **internal** busses for code & data (e.g. ARM9)*



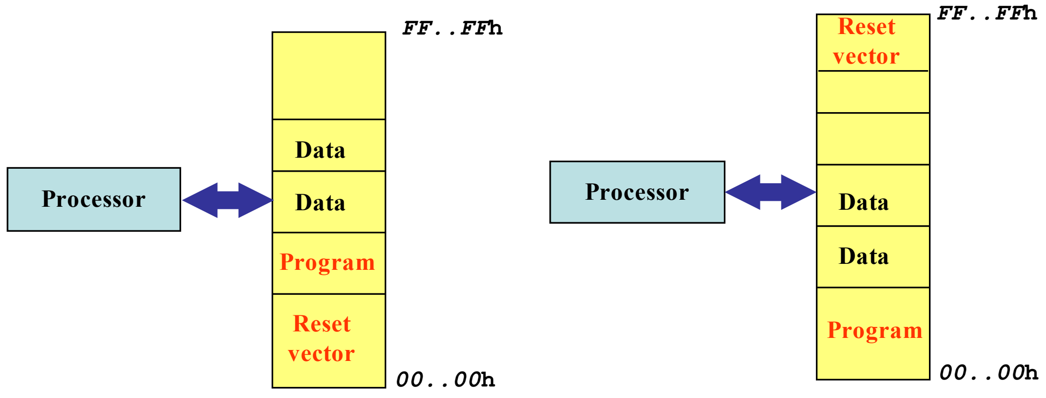
Top Boot and Bottom Boot

Different processor families use different locations for reset vector storage at boot-up.

Examples:

• x86 boots up from the top of the memory space

• ARM boots up from the bottom of the memory space



Processor ‘Size’

The processor size is described in terms of ‘bits’ (e.g. an 8-bit, 32-bit processor)

- corresponds to the data size that can be manipulated at a time by the processor
- typically reflected in the size of the processor (internal) data path and register bank

Hence an 8-bit processor can only manipulate byte size data at a time,
while a 32-bit processor can handle 32-bit double word size data at a time

- even though the data content may only be of single byte size

Registers

The most fundamental storage area in the processor

- is closely located to the processor
- provides very fast access, operating at the processor clock
- but is of limited amount (less than 100 typical)

Most are of the general purpose type and can store any type of information:

- data – e.g. timer value, constants
- address – e.g. ASCII table, stack

Some are reserved for specific purpose

- program counter (r15 in ARM)
- program status register (CPSR in ARM)

Data Organization in Memory

Memory contains storage locations that store data of a certain fixed size

• most commonly of the 8-bit (byte) size

Each location is provided with a unique address.

Depending on the data path/size of the processor

• the memory content is accessible in sizes of an 8-bit byte, a 16-bit half word, a 32-bit word, and even a 64-bit double word

Data Alignment

A 32-bit datum consists of four bytes of data, and is stored in four successive memory locations.

Data and code **must be aligned** to the respective address size boundary.

• e.g. for a 32-bit system, align to the word boundary, with the lowest two address bits equal to zero

But what is the order of the four bytes of data?

• depends on the **Endianness** of the processor

Data Endianness

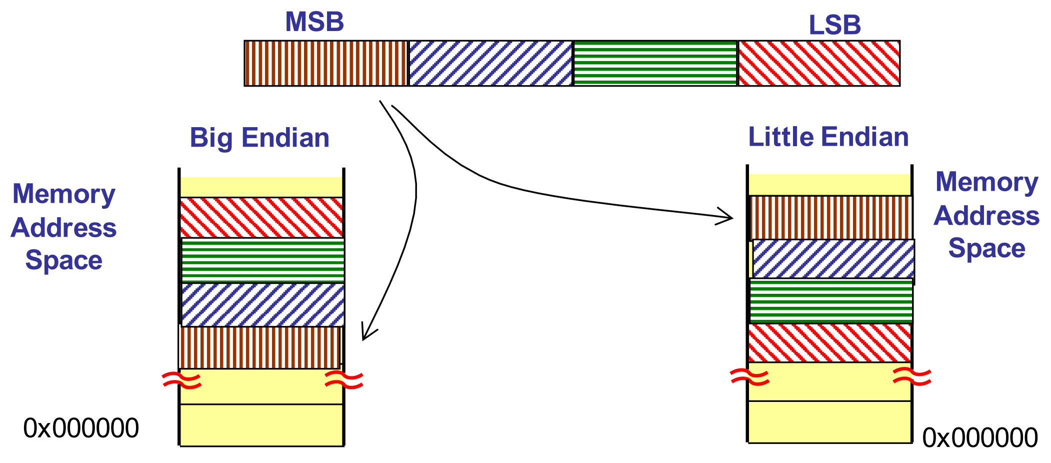
In the **Little Endian** format,

- the least significant byte (LSB) is stored in the lowest address of the memory, with the most significant byte (MSB) stored in the highest address location of the memory.

In the **Big Endian** format,

- the least significant byte (LSB) is stored in the highest address of the memory, with the most significant byte (MSB) stored in the lowest address location of the memory.

Data Endianness (cont’d)



Comparison

Little Endian

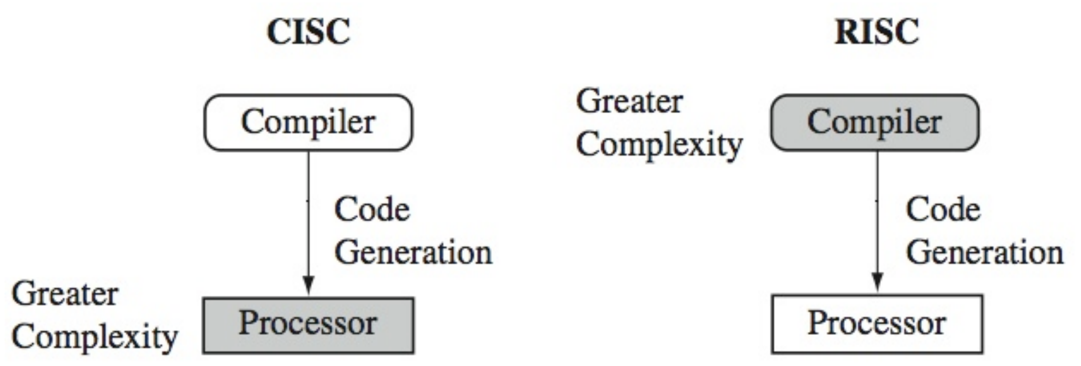
- The order matched with processor instructions typically process numbers from LSB to MSB.
- The byte number corresponds with the address offset, suitable for multi-precision data manipulation.
- <u>L</u>SB → <u>L</u>ower Address (<u>L</u>ittle Endian) The Three “L’s”

Big Endian

- Can compare numerical data by just accessing the zero offset byte.
- Corresponds to the written order of number (starting with the most significant digit).

Some processors (e.g. ARM) have bi-endian hardware that feature ‘switchable’ endianness.

RISC versus CISC



RISC and CISC are <u>PHILOSOPHIES</u> of Computer Architecture. Most modern processors have Features from each Philosophy although they may be <u>MARKETED</u> as being only RISC (or CISC).

CISC

Features of the Complex Instruction Set Computer (CISC):

- many instructions
- complex instructions
  - each instruction can execute several low level operations
- complex addressing modes
  - smaller number of registers needed

A semantically rich instruction set is accommodated by allowing instructions that can be of variable lengths.

Advantages of CISC

As each instruction can execute several low level operations,

• the code size is reduced to save on memory requirement

• less main memory access is required and hence faster.

Backward code compatibility is maintained

• can add new (and more powerful) instructions while retaining the ‘old’ instruction set for code compatibility (i.e. the legacy program can still run)

Easier to program

• direct support of high-level language constructs

• complex instructions that fit well with high-level language expression

Limitations of CISC

A highly encoded instruction set needs to be decoded by complex instruction decoder circuitry (often microcoded style)

• more complex hardware design

• slower instruction decoding/execution

Variable length instructions

• different execution time among instructions

• affect pipelined operations

• more complex bus controller

RISC

RISC – Reduce Instruction Set Computer
• Small instruction sets
• Simpler instructions – all execute in same number of cycles
• Fixed length instructions
• Large number of registers
• Simpler addressing mode with the Load/Store instruction for accessing memory

• Hardware (CPU datapath) can be pipelined
• Programming (compiler) is more complex and requires longer instruction sequences to do same job as CISC

RISC Philosophy

1. Instructions
   – reduced number of instruction classes
   – each is simple: execute in single cycle
   – compiler/programmer must implement complicated operations such as division
   – fixed length: fast fetch and decode
2. Pipelining
   – instruction processing broken into small units
   – each unit executed in parallel
   – no microcode

RISC Philosophy (cont)

3. Registers
   - large register file (number of registers)
   - general purpose registers (data or addresses)
   - very fast local memory
4. Load/Store Architecture
   - separate load and store instructions (no MOV)
   - no data processing ops access memory (no CMPSB)

Advantages of RISC

Simpler instructions

• one clock per instruction gives faster execution than on a CISC processor with the same clock speed

Simpler addressing mode

• faster decoding

Fixed length instructions

• faster decoding and better pipeline performance

Simpler hardware

• less silicon area

• less power consumption

RISC Memory Footprint

The RISC processor typically needs more memory than a CISC does to store the same program.

• complex functions performed in a single but slower instruction in a CISC processor may require two, three, or more simpler instructions in a RISC.

To reduce memory requirements and hence cost,

• ARM provides the 16-bit Thumb instruction set as an option for its RISC processor cores.

• Thumb instructions are “compressed” versions of ARM instructions

Limitations of RISC

Fewer instructions than CISC

- as compared to CISC, RISC needs more instructions to execute one task
- code density is less
- need more memory

No complex instruction

- no hardware support for division, floating-point arithmetic operation
- need a more complex compiler and a longer compiling time

But ARM also adds DSP-like instructions to support commonly used signal processing function

Instruction Code Format

Opcode encoding depends on the number of bit used.

Example: For ARM, all instructions are of 32-bit length, but only 8 bits (bit 20 to 28) are used to encode the instruction. Hence a total of 28 = 256 different instructions possible.

A typical instruction is encoded with a specific bit pattern that consists of the following:

1. an opcode field specifying the operation to be performed.
2. an operand(s) identification (address) field that depends on the modes of addressing;
   – this provides the address of the register/memory location (s) that store the operand(s), or the operand itself.

Instruction Opcode Types

General categories of instruction operations:

- Data transfer
  E.g. move, load, and store
- Data manipulation
  E.g. add, subtract, logical operation
- Program control
  E.g. branch, subroutine call

Operand Addressing Types

Immediate addressing

- operand is given in the instruction

Register addressing

- operand is stored in a register

Direct addressing

- operand is stored in memory, with the address given in the instruction

Indirect (Index) addressing

- operand is stored in memory, with the address given in a register (address adds with an offset given in the instruction)

Implied addressing

- implicit location like stack and program counter

Instruction Execution

Multiple stages are involved in executing an instruction.

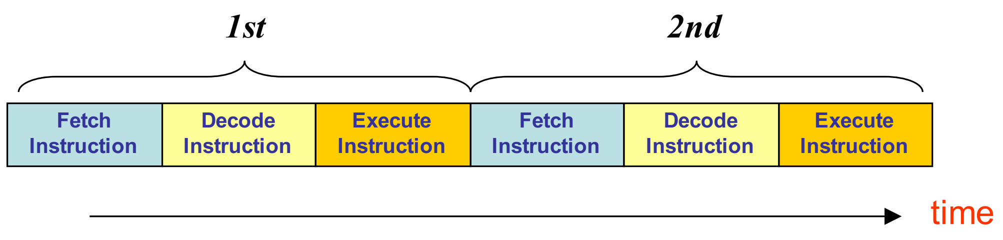
Example:

1) Fetching the instruction code

2) Decoding the instruction code

3) Executing the instruction code

Hence multiple processor clock cycles are needed to execute one single instruction.



Instruction Pipeline

Pipeline allows concurrent execution of multiple different instructions

• execution of different stages of multiple instructions at the same time

During a normal operation

• while one instruction is being executed

• the next instruction is being decoded

• and a third instruction is being fetched from memory

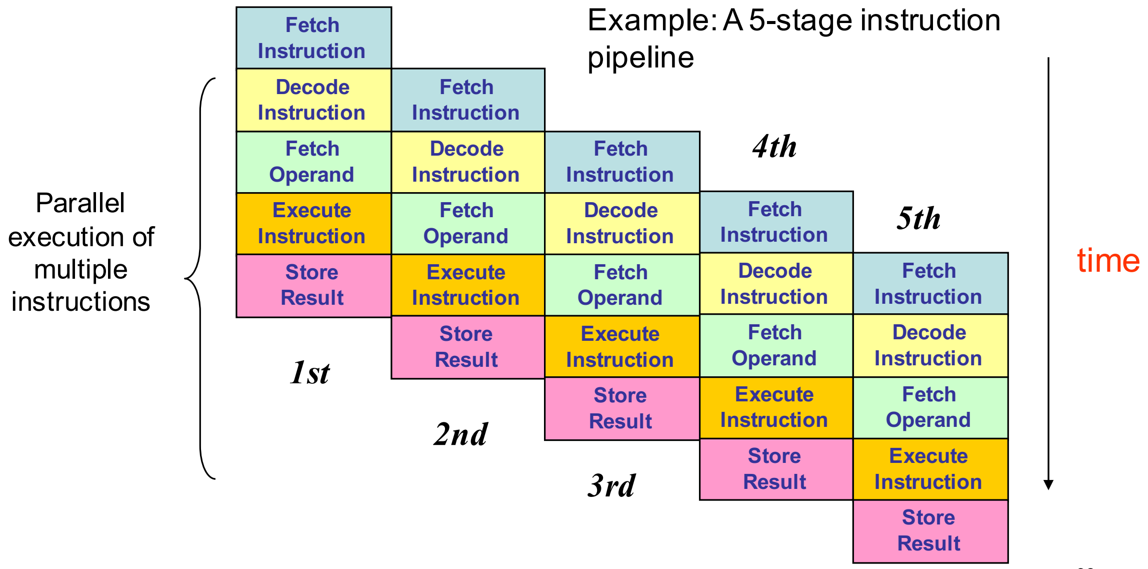
• allows effective throughput to increase to one instruction per clock cycle

Pipelined Architecture

Longer pipeline can also be used to further break down the operation carried out in the individual stage

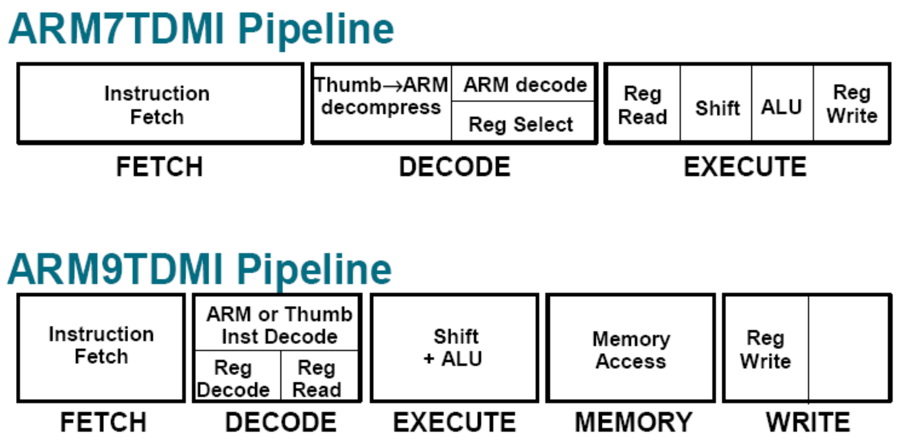
•simpler logic for each stage to increase system clock

Example: A 5-stage instruction pipeline



ARM Pipelined Architecture

ARM7 and ARM9 pipelined architecture



Pipeline Interlocks

Pipeline interlocks occur when the data required for an instruction is not available (a "bubble")

• due to incomplete execution of an earlier instruction that is to supply the data.

When an interlock occurs, the hardware stalls the execution of an instruction until the data is ready.

The number of interlocks can be reduced by re-arranging the order of instructions and meticulous choice of registers usage

• e.g., achieved through handcraft assembly language programming OR a very good optimizing compiler