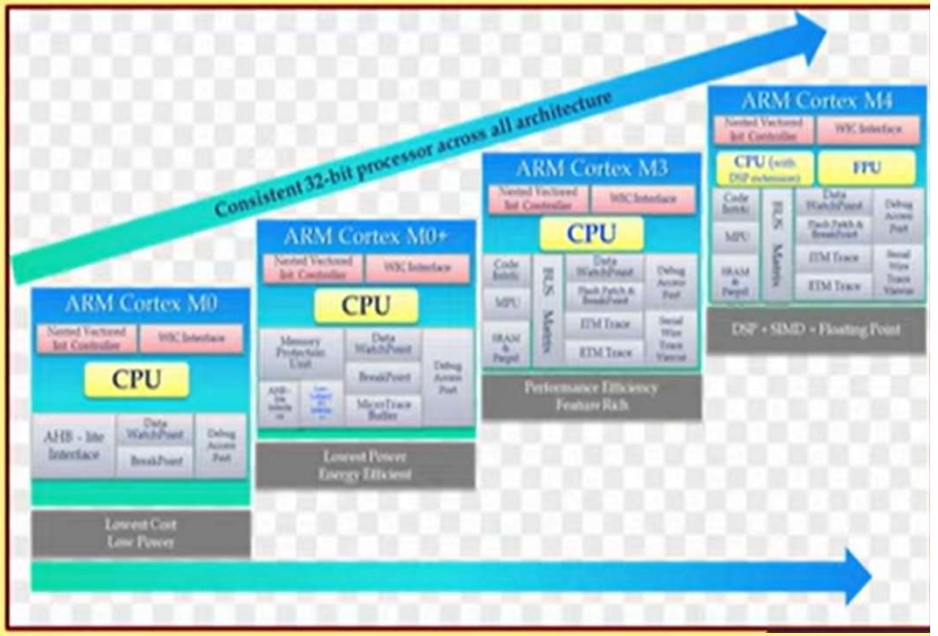
ARM Microcontroller

Why do we talk about ARM?

- One of the most widely used processor cores.
- Some application examples:
  - ARM7: iPod
  - ARM9: BenQ, Sony Ericsson
  - ARM11: Apple iPhone, Nokia N93, N100
  - 90% of 32-bit embedded RISC processors till 2010.
- Mainly used in battery-operated devices:
  - Due to low power consumption and reasonably good performance.



Introduction

- 32-bit RISC architecture
- Developed by ARM Corporation, previously known as *Acron RISC Machine*
- Licensed to companies that want to manufacture ARM based CPUs or SOC products
- Helps the licensee to develop their own -processors compliant with ARM instruction set architecture

- About the instruction set architecture (ISA) of the CPU.
    a) Complex Instruction Set Computer (CISC)
        - Typically used in desktops, laptops and servers (courtesy Intel).
    b) Reduced Instruction Set Computer (RISC)
        - Typically used in microcontrollers, that are used to build embedded systems.

Features

- ARM cores are very simple, require relatively lesser number of transistors, leaving enough space on die to realize other functionalities on the silicon
- Instruction set architecture and the pipeline design aimed at minimizing energy consumption
- Also capable of running 16-bit THUMB instruction set – greater code density and enhanced power saving
- Higher performance
- Highly modular architecture – the only mandatory part is the integer pipeline, all other components are optional
- Built-in JTAG debug port and on-chip embedded in-circuit emulator (ICE) that allows programs to be downloaded and fully debugged in-system

ARM Architecture History

Version	Year	Features	Implementation
V1	1985	The first commercial RISC (26-bit)	ARM1
V2	1987	Coprocessor support	ARM2, ARM3
V3	1992	32-bit, MMU, 64-bit MAC	ARM6, ARM7
V4	1996	THUMB	ARM7TDMI, ARM8, ARM9TDMI, StrongARM
V5	1999	DSP and Jazelle extensions	ARM10, XScale
V6	2001	SIMD, THUMB-2, TrustZone, Multiprocessing	ARM11, ARM11 MPCore
V7	2005	Three profiles, NEON, VFP	?

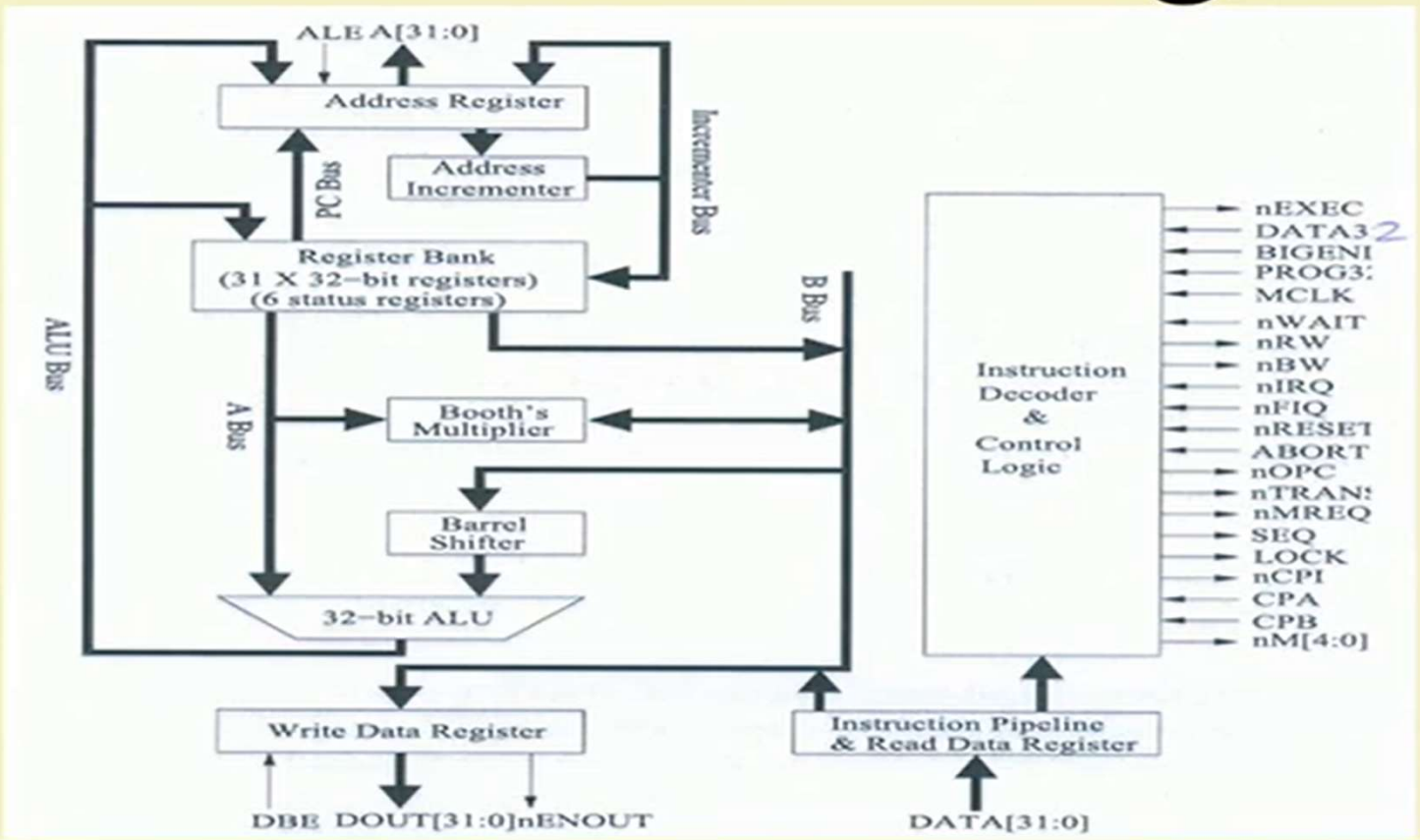
Popular ARM Architectures

- ARM7
  - 3 pipeline stages (fetch/decode/execute)
  - High code density / low power consumption
  - Most widely used for low-end systems
- ARM9
  - Compatible with ARM7
  - 5 stages (fetch/decode/execute/memory/write)
  - Separate instruction and data cache
- ARM10
  - 6-stages (fetch/issue/decode/execute/memory/write)

ARM Family Comparison

	ARM 7 (1995)	ARM9 (1997)	ARM10 (1999)	ARM11 (2003)
Pipeline depth	3-stage	5-stage	6-stage	8-stage
Typical clock frequency (MHz)	80	150	260	335
Power (mW/MHz)	0.06	0.19	0.50	0.40
Throughput (MIPS/MHz)	0.97	1.1	1.3	1.2
Architecture	Non Neumann	Harvard	Harvard	Harvard
Multiplier	8 x 32	8 x 32	16 x 32	16 x 32

ARM7 block diagram

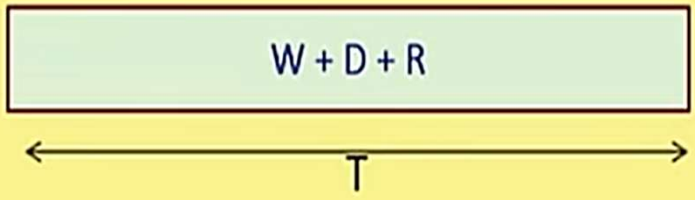


What is Pipelining?

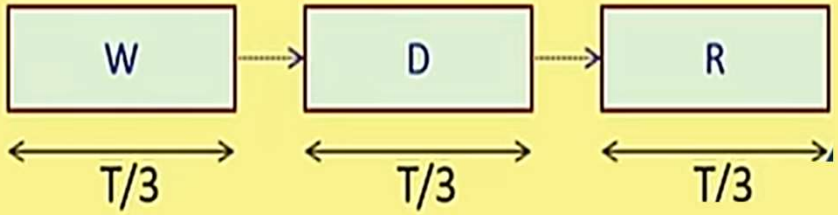
- A mechanism for overlapped execution of several input sets by partitioning some computation into a set of *k* sub-computations (or stages).
  - Very nominal increase in the cost of implementation.
  - Very significant speedup (ideally, *k*).
- Where are pipelining used in a computer system?
  - **Instruction execution**: Several instructions executed in some sequence.
  - **Arithmetic computation**: Same operation carried out on several data sets.
  - **Memory access**: Several memory accesses to consecutive locations are made.

A Real-life Example

- Suppose you have built a machine *M* that can wash (*W*), dry (*D*), and iron (*R*) clothes, one cloth at a time.
  - Total time required is *T*.
- As an alternative, we split the machine into three smaller machines M_W, M_D and M_R, which can perform the specific task only.
  - Time required by each of the smaller machines is *T/3* (say).

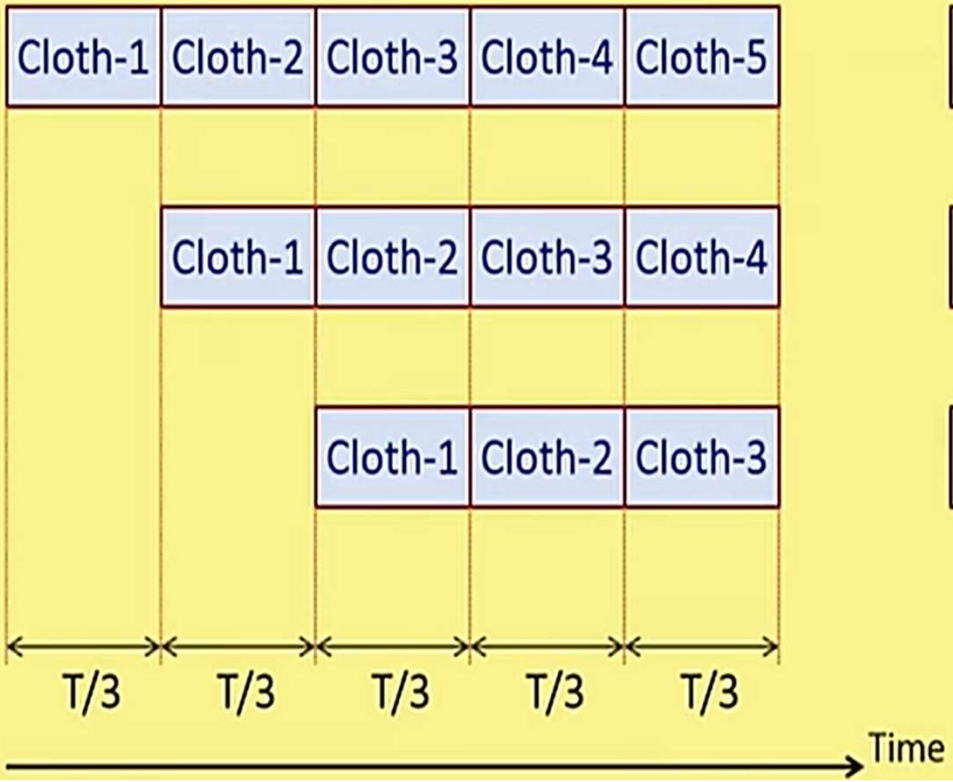


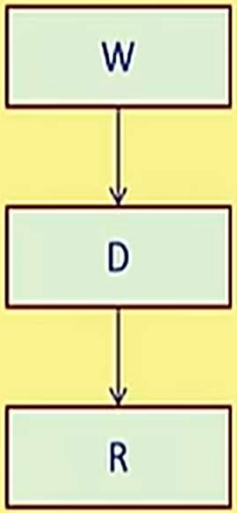
For N clothes, time $T_1 = N.T$



For N clothes, time $T_3 = (2 + N).T/3$

How does the pipeline work?





Finishing times:

- Cloth-1 – 3.T/3
- Cloth-2 – 4.T/3
- Cloth-3 – 5.T/3
- ...
- Cloth-N – (2 + N).T/3

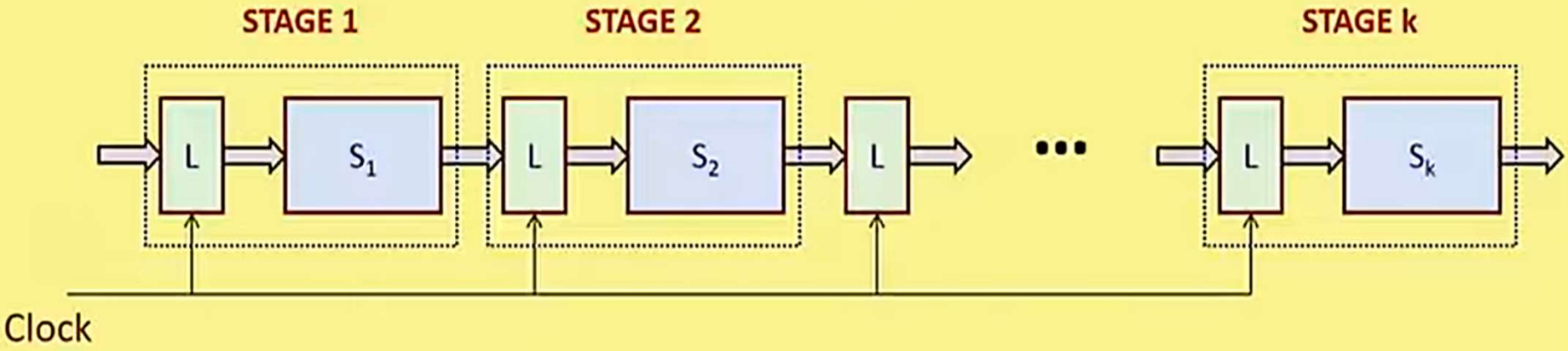
Extending the Concept to Processor Pipeline

- The same concept can be extended to hardware pipelines.
- Suppose we want to attain *k* times speedup for some computation.
  - **Alternative 1**: Replicate the hardware *k* times → cost also goes up *k* times.
  - **Alternative 2**: Split the computation into *k* stages → very nominal cost increase.

Extending the Concept to Processor Pipeline

- The same concept can be extended to hardware pipelines.
- Suppose we want to attain *k* times speedup for some computation.
  - **Alternative 1**: Replicate the hardware *k* times → cost also goes up *k* times.
  - **Alternative 2**: Split the computation into *k* stages → very nominal cost increase.
- Need for buffering:
  - In the washing example, we need a tray between machines (W & D, and D & R) to keep the temporarily before it is accepted by the next machine.
  - Similarly in hardware pipeline, we need a *latch* between successive stages to hold the intermediate results temporarily.

Model of a Synchronous k-stage Pipeline



- The latches are made with master-slave flip-flops, and serve the purpose of isolating inputs from outputs.
- The pipeline stages are typically combinational circuits.
- When *Clock* is applied, all latches transfer data to the next stage simultaneously.

Speedup and Efficiency

Some notations:

- τ :: clock period of the pipeline
- t_i :: time delay of the circuitry in stage S_i
- d_L :: delay of a latch

Maximum stage delay $\tau_m = \max \{t_i\}$

Thus, $\tau = \tau_m + d_L$

Pipeline frequency $f = 1 / \tau$

- *If one result is expected to come out of the pipeline every clock cycle, f will the maximum throughput of the pipeline.*

- The total time to process *N* data sets is given by

$$T_k = [(k-1) + N].\tau$$

$(k-1)\,\tau$ time required to fill the pipeline
1 result every τ time after that → total $N.\tau$

- For an equivalent non-pipelined processor (i.e. one stage), the total time is

$$T_1 = N.k.\tau$$

(ignoring the latch overheads)

- Speedup of the k-stage pipeline over equivalent non-pipelined processor:

$$S_k = \frac{T_1}{T_k} = \frac{N.k.\tau}{k.\tau + (N-1).\tau} = \frac{N.k}{k + (N-1)}$$

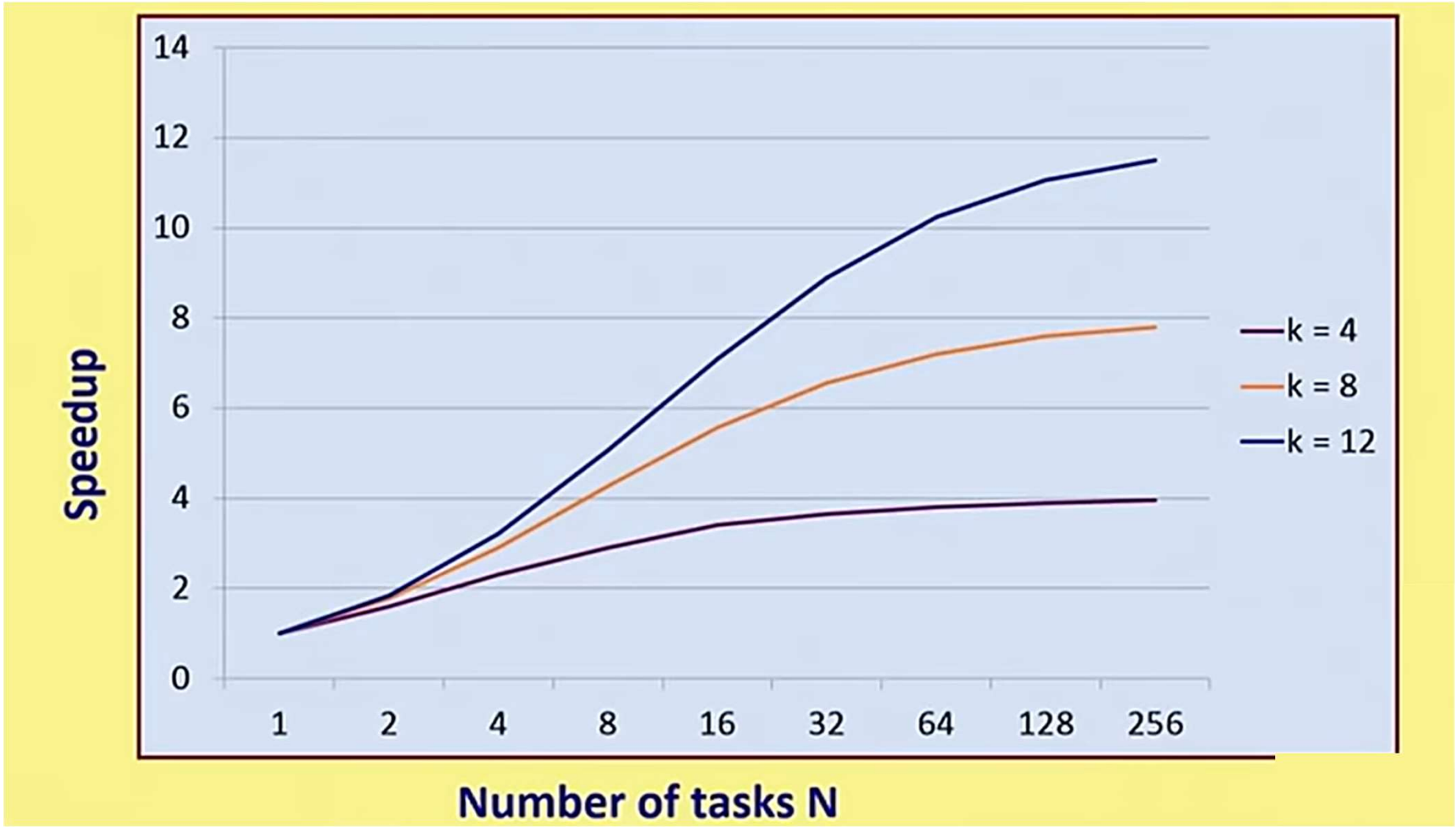
As $N \to \infty$, $S_k \to k$

- Pipeline efficiency:
  - How close is the performance to its ideal value?

$$E_k = \frac{S_k}{k} = \frac{N}{k + (N - 1)}$$

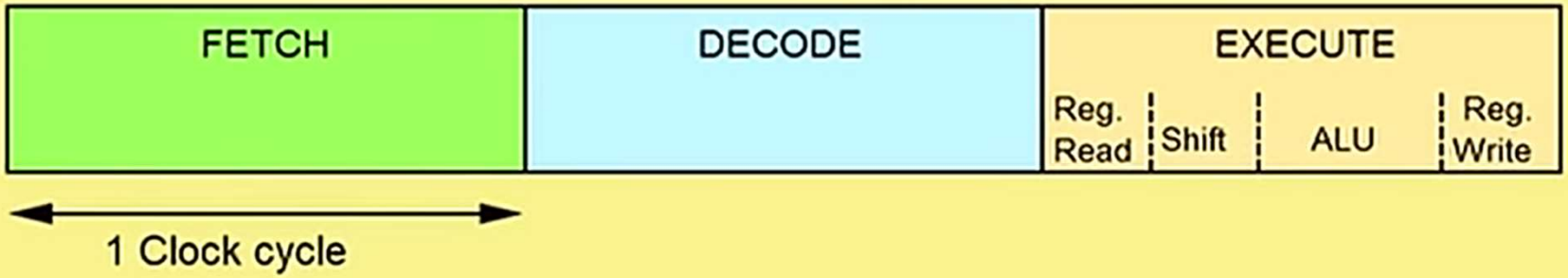
- Pipeline throughput:
  - Number of operations completed per unit time.

$$H_k = \frac{N}{T_k} = \frac{N}{[k + (N - 1)].\tau}$$

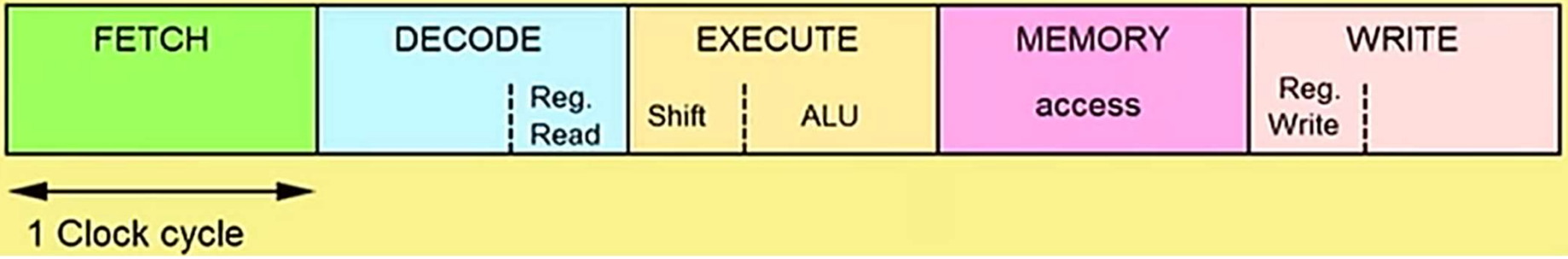

Speedup
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
128
256
k = 4
k = 8
k = 12
Number of tasks N

ARM Pipelining Examples

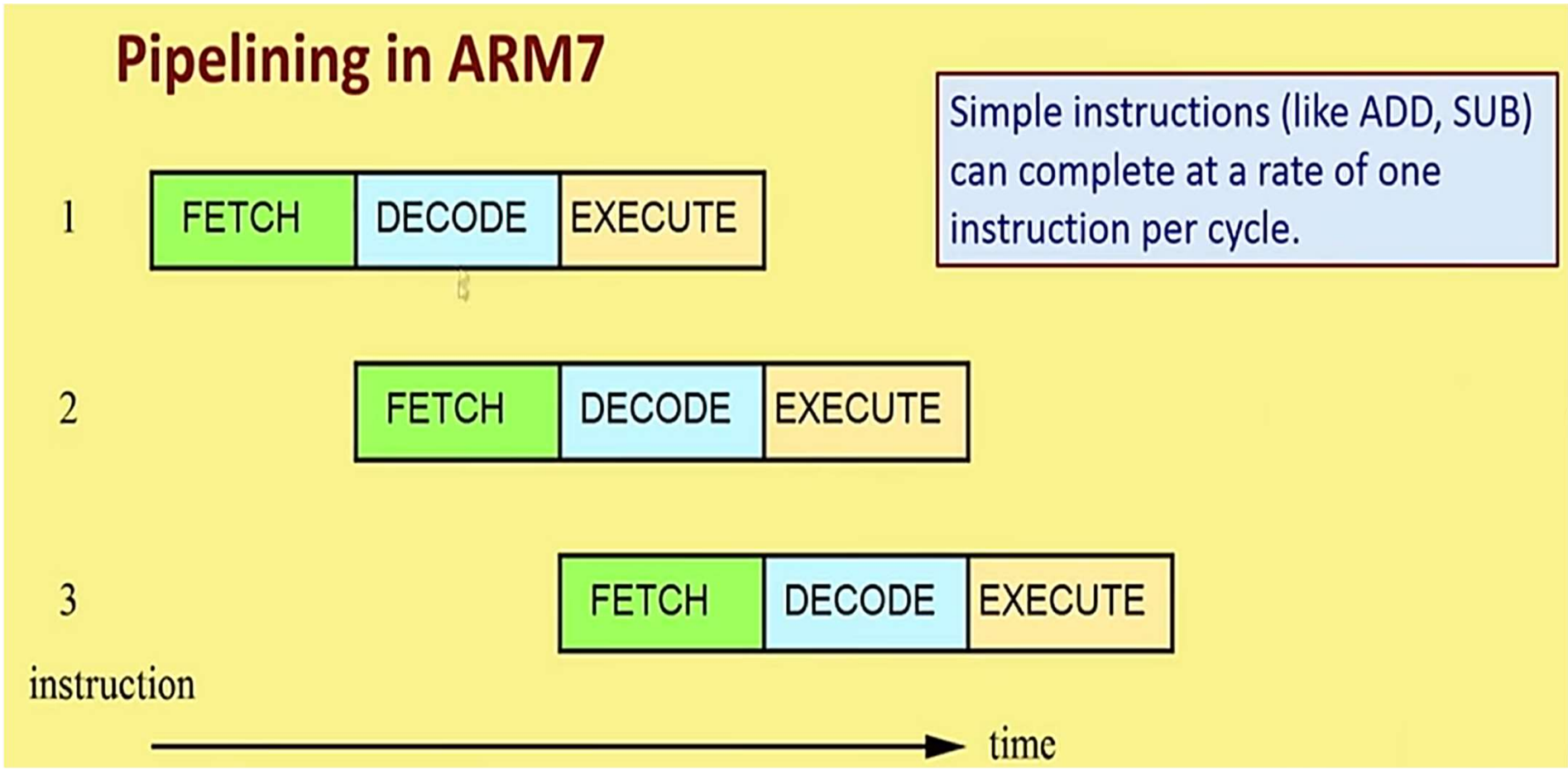
ARM7TDMI Pipeline



ARM9TDMI Pipeline

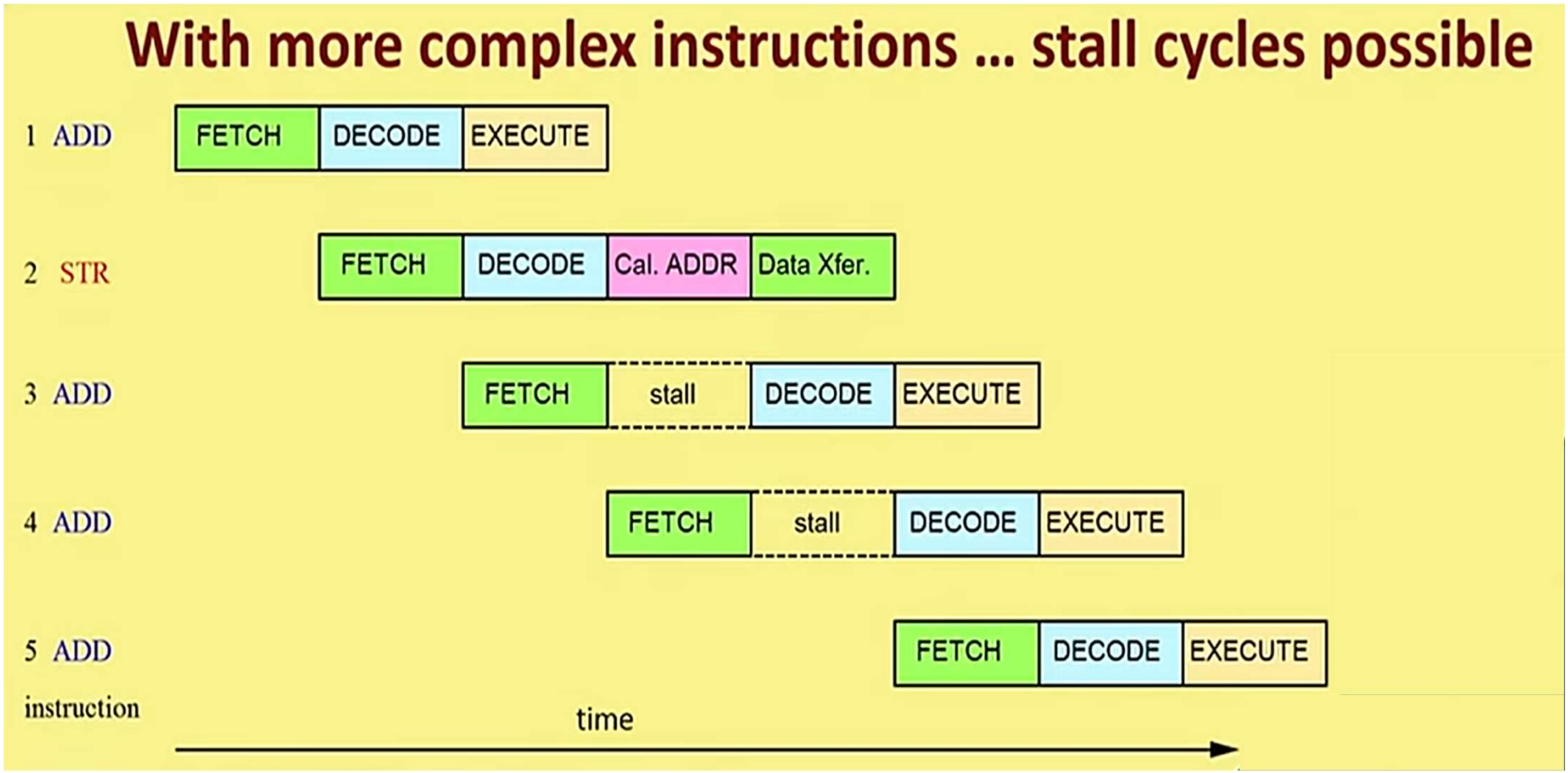


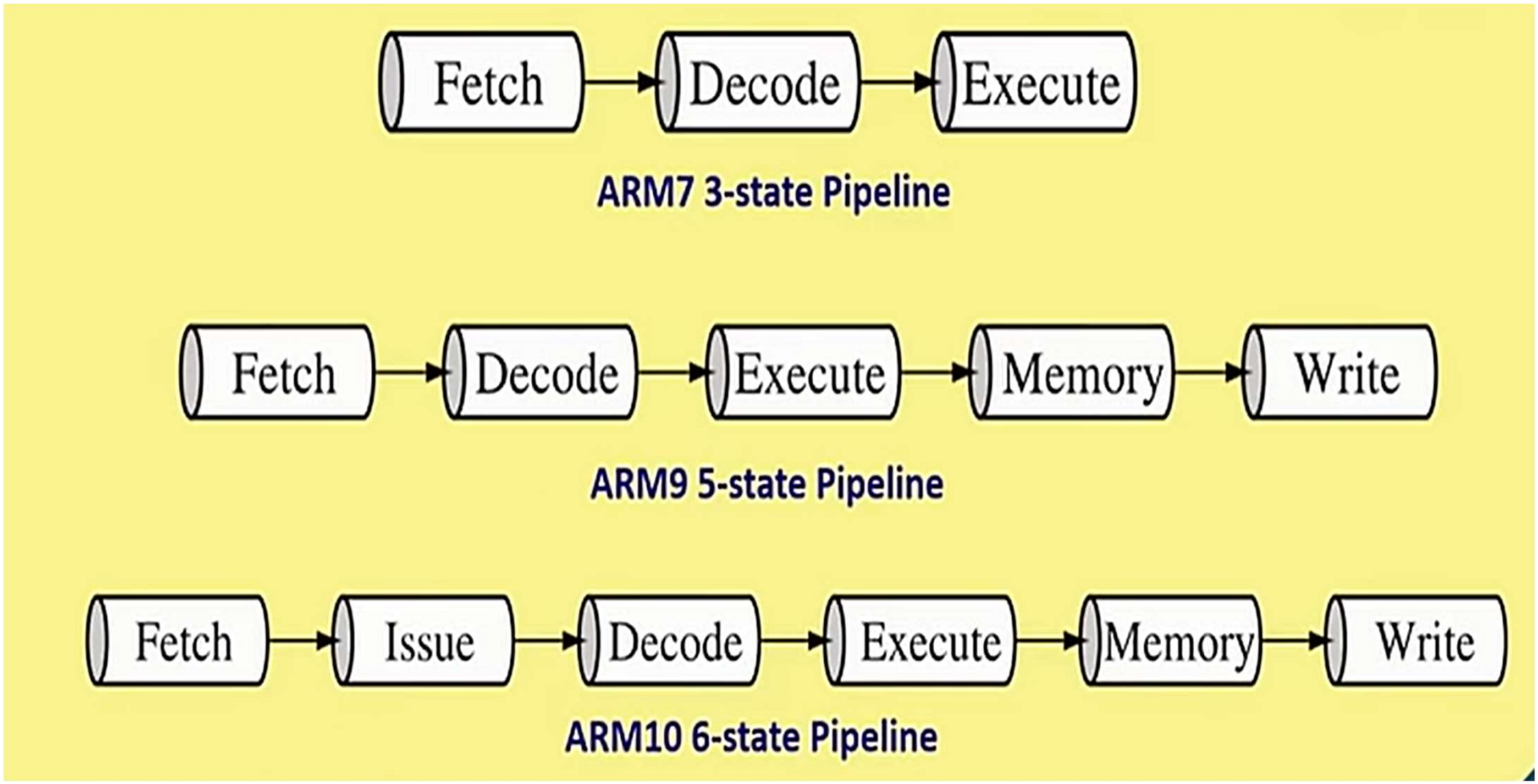
Pipelining in ARM7



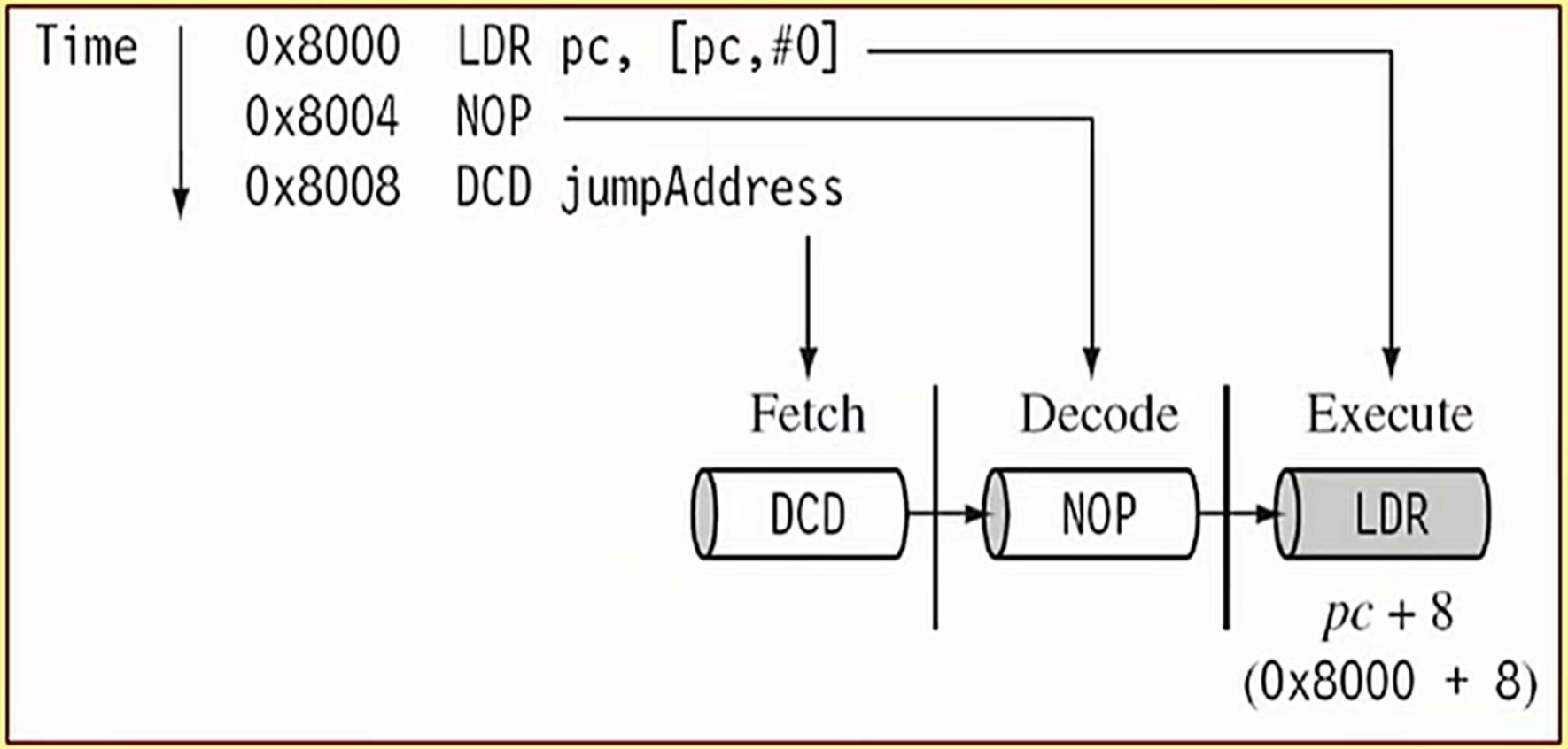
Simple instructions (like ADD, SUB) can complete at a rate of one instruction per cycle.

With more complex instructions … stall cycles possible




Fetch
Decode
Execute
ARM7 3-state Pipeline
Fetch
Decode
Execute
Memory
Write
ARM9 5-state Pipeline
Fetch
Issue
Decode
Execute
Memory
Write
ARM10 6-state Pipeline

- In execution, the program counter (PC) is always 8 bytes ahead.



- ARM processor modes and registers
- Special registers and exception handling
- ARM and Thumb modes of execution

Processor Modes

Processor Mode	Code	Description
User	usr	Normal program execution mode
FIQ	fiq	Entered when a high priority (fast) interrupt is raised
IRQ	irq	Entered when a low-priority (normal) interrupt is raised
Supervisor	svc	A protected mode for the operating system (entered on reset and when software interrupt instruction is executed)
Abort	abt	Used to handle memory access violations
Undefined	und	Used to handle undefined instructions
System	sys	Runs privileged operating system tasks

Registers

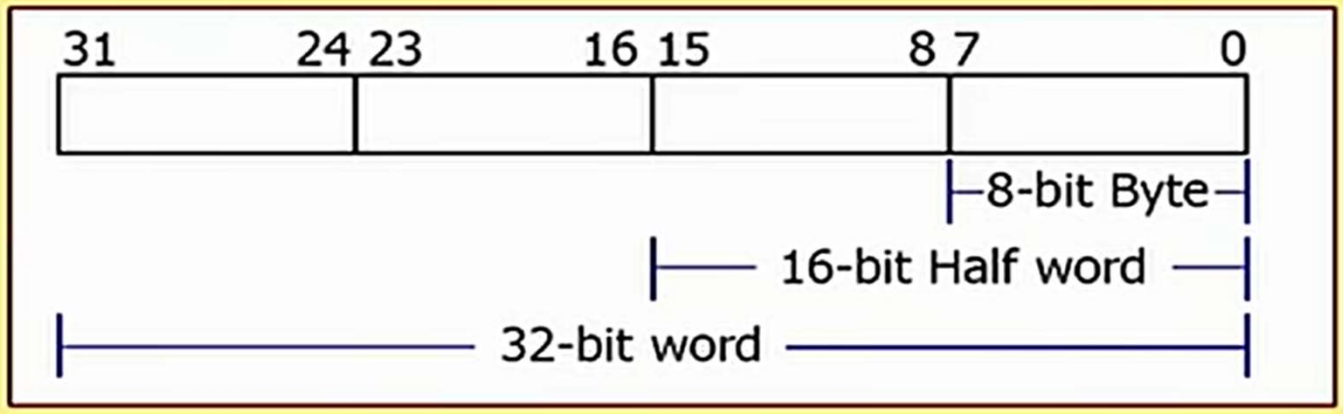
- ARM has 37 registers all of which are 32-bits long.
- These registers are:
  - 1 dedicated *program counter* (PC)
  - 1 dedicated *current program status register* (CPSR)
  - 5 dedicated *saved program status registers* (SPSR)
  - 30 *general-purpose registers* (GPR)

Registers (contd.)

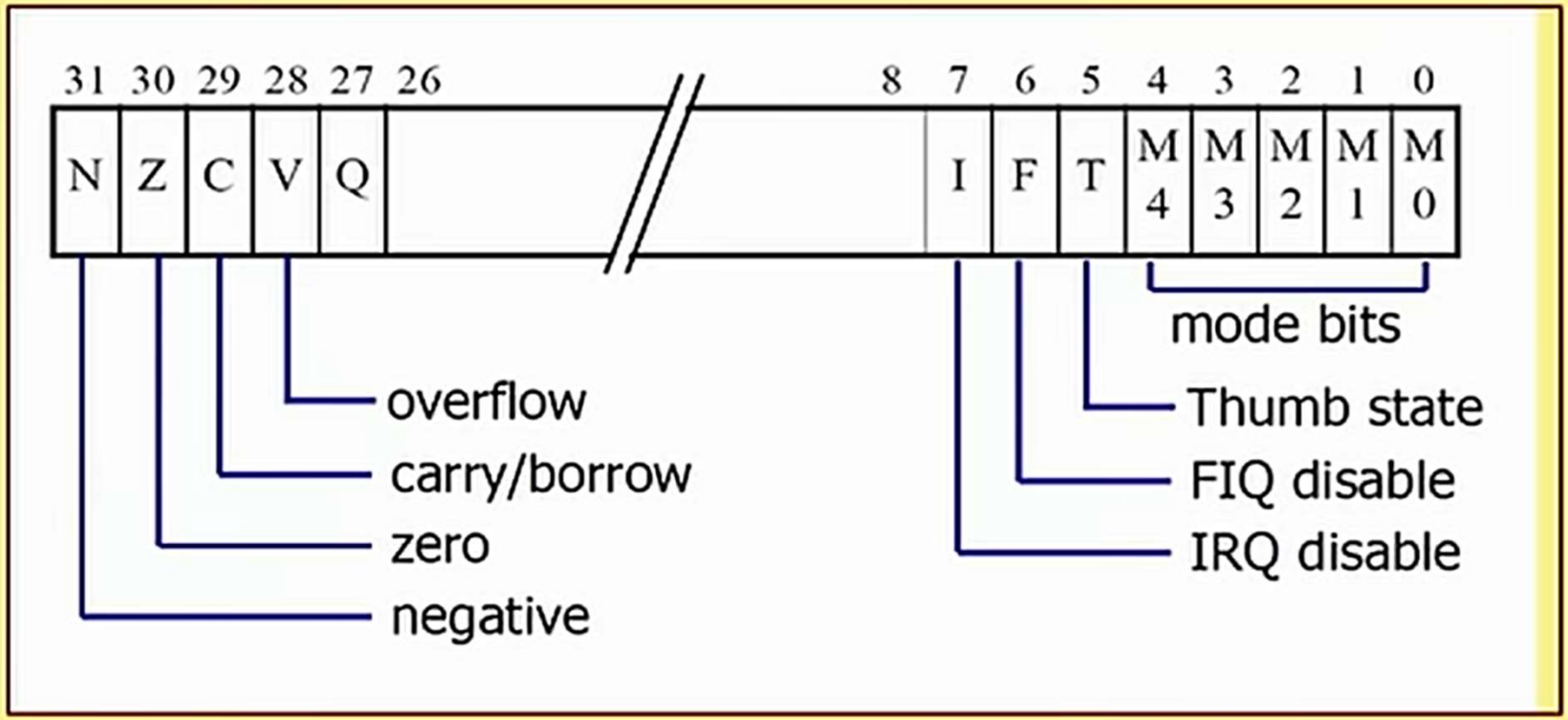
- The current processor mode governs which of several register sets is accessible.
- Only 16 registers are visible to a specific mode of operation. Each mode can access:
  - A particular set of registers r0-r12
  - r13 (SP, stack pointer)
  - r14 (LR, link register)
  - r15 (PC, program counter)
  - Current program status register (CPSR)
- Privileged modes (except System) can also access a particular SPSR.

General-purpose Registers

- 6 data types are supported (signed/unsigned)
  - 8-bit byte, 16-bit half-word, 32-bit word
- All ARM operations are 32-bit.
  - Shorter data types are only supported by data transfer operations.



Current Program Status Register (CPSR)



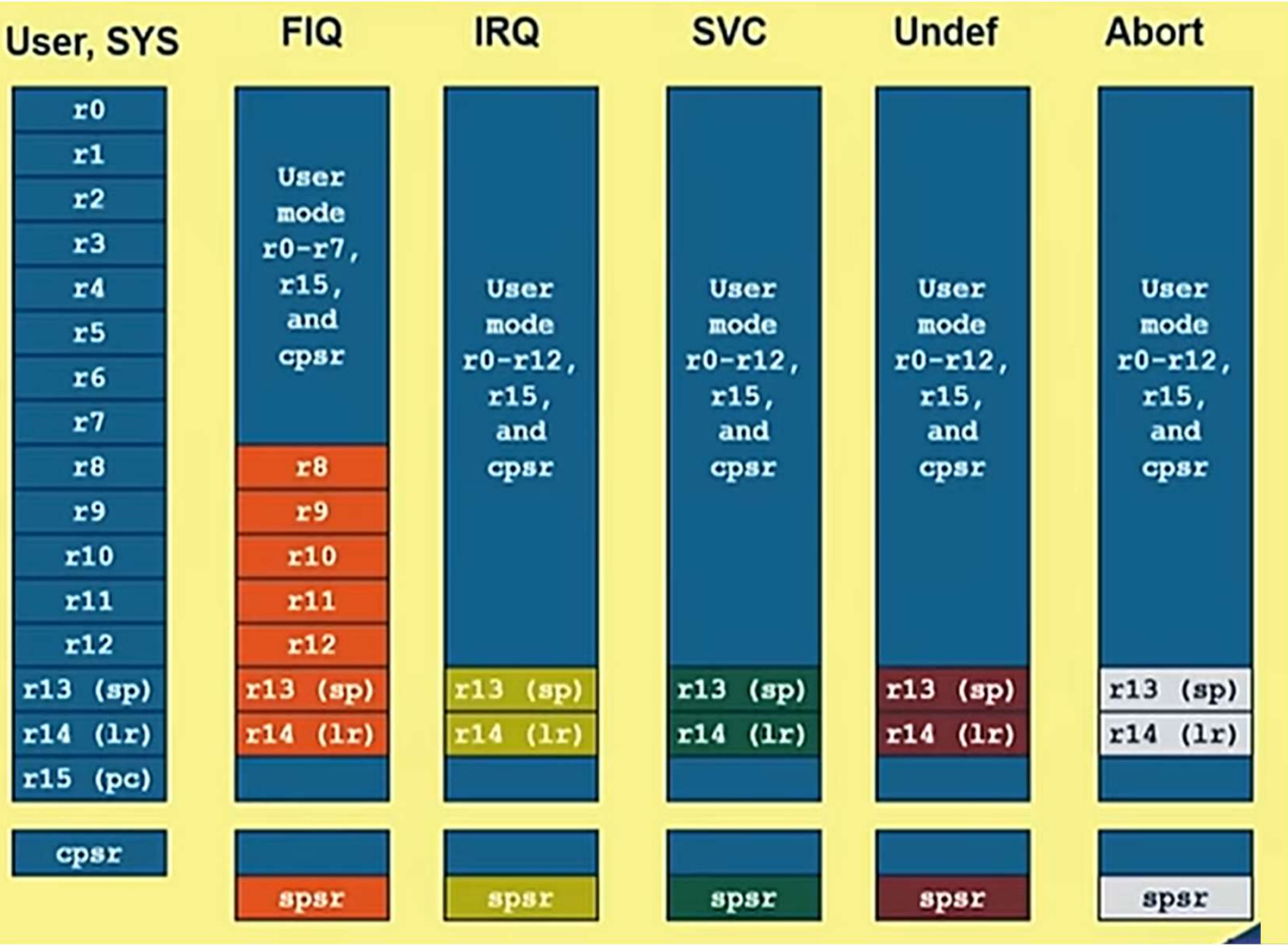
Special Registers

- **PC (r15)**: Any instruction with PC as its destination register is a program branch.
- **LR (r14)**: Saves a copy of PC when executing the *BL instruction* (subroutine call) or when jumping to an exception or an interrupt handler.
  - It is copied back to PC on return from those routines.
- **SP (r13)**: There is no stack in the ARM architecture.
  - R13 is reserved as a pointer for the software-managed stack.
- **CPSR**: Holds the visible status register.
- **SPSR**: Holds a copy of the previous status register while executing exception or interrupt handler routines.
  - Copied back to CPSR on return from exception or interrupt.

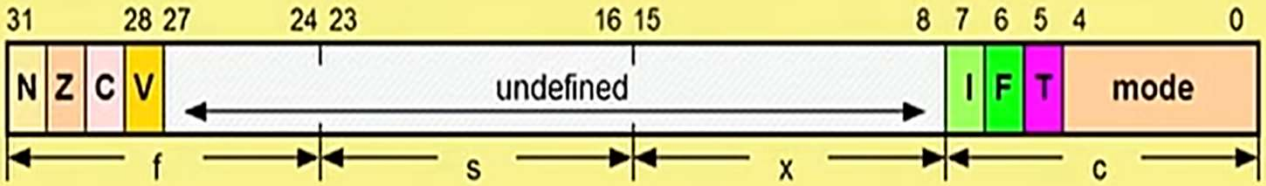
Program Counter

- When the processor is executing in *ARM mode*:
  - All instructions are 32-bits wide, and must be word aligned.
  - The last two bits of PC are zero (i.e. not used).
  - Due to pipelining, PC points 8 bytes ahead of the current instruction, or 12 bytes ahead if the current instruction includes a register-specified shift.
- When the processor is executing in *Thumb mode*:
  - All instructions are 16-bits wide, and are half-word aligned.
  - The last bit of the PC is zero (i.e. not used).

Register Organization Summary



Program Status Register



Condition code flags

- N = Negative result from ALU
- Z = Zero result from ALU
- C = ALU operation Carried out
- V = ALU operation oVerflowed

Mode bits

10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit (Arch. with Thumb mode only)

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

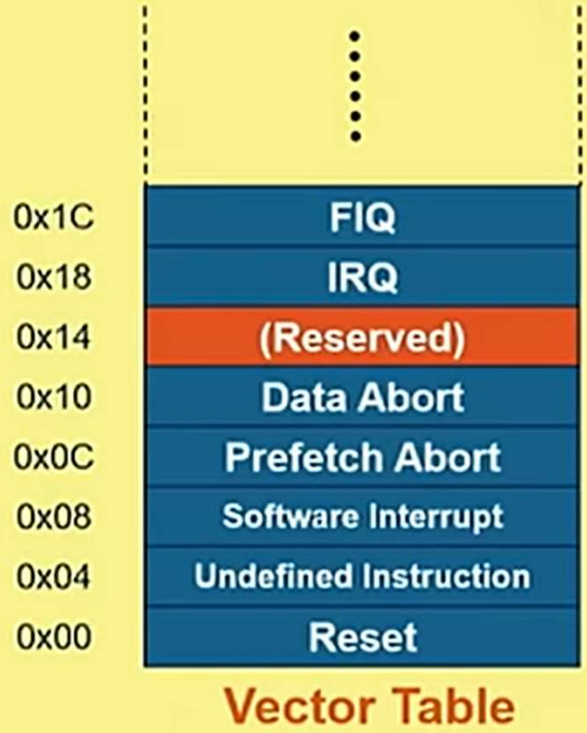
ARM and Thumb Instruction Set

- Most ARM implementations provide two instruction sets:
  a) 32-bit ARM instruction set
  b) 16-bit Thumb instruction set

	ARM (cpsr T = 0)	Thumb (cpsr T = 1)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution	Most	Only branch instructions
Data processing instructions	Access to barrel shifter and ALU	Separate barrel shifter and ALU instructions
Program status register	Read-write in privileged mode	No direct access
Register usage	15 GPRs + pc	8 GPRs + 7 high registers + pc

Exception Handling

- When an exception occurs, the processor
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate bits in CPSR
    - Changes to ARM state
    - Changes to related mode
    - Disables IRQ, FIQ
  - Stores return address in LR_<mode>
  - Sets PC to vector address
- To return, the exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>



- Exception handling in ARM is controlled through an area of memory called the *vector table*.
  - Exists at the bottom of the memory map *from 0x00 to 0x1c*.
  - Within this table, one word is allocated to each of the various exception types.
  - This word will contain some form of ARM instruction that should perform a branch.
    - Does not contain an address.

What is Conditional Execution?

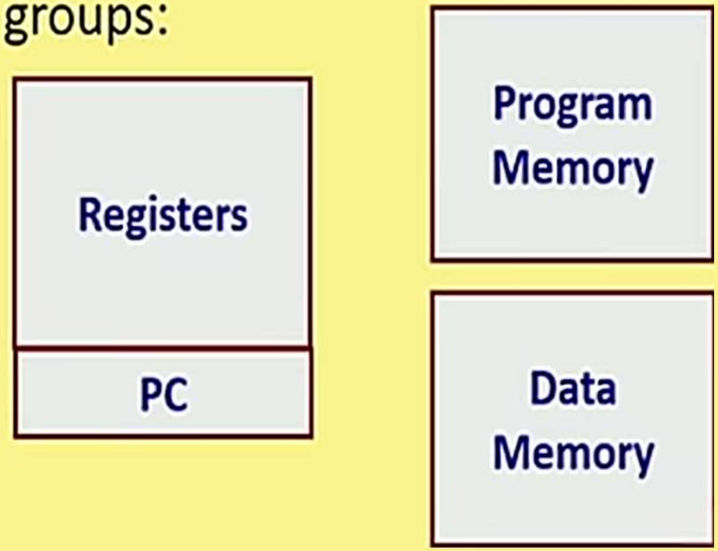
- It controls whether or not the CPU will execute an instruction.
- Most instructions have a condition attribute that determines whether it will be executed based on the status of the condition flags.
  - Prior to execution, the processor compares the condition attribute with the condition flags in CPSR.
  - If they do not match, the instruction is not executed.
  - Example:

    MOVEQ r1,#0 (if zero flag is set, then r1 = 0)
  - The condition attribute (here *EQ*) is suffixed to the instruction mnemonic.

- Categories of ARM instructions
- Data processing instructions

The ARM Instruction Set

- ARM instruction can be categorized into three groups:
  - a) Data processing instructions
    - Operate on values in registers
  - b) Data transfer instructions
    - Move values between registers and memory
  - c) Control flow instructions
    - Change the value of the program counter (PC)



(a) Data Processing Instructions

- All operands are 32-bits in size:
  - Either registers
  - Or literals (immediate values) specified as part of the instruction
- The result, if any, is also 32-bit in size and goes into a specified register.
  - One exception: long multiply, that generates 64-bit results.
- All operand and result registers are independently specified as part of the instruction.

- **Arithmetic instructions:**

```
ADD   r0,r1,r2     ; r0 = r1 + r2
ADC   r0,r1,r2     ; r0 = r1 + r2 + C  (C is carry bit)
SUB   r0,r1,r2     ; r0 = r1 - r2
SBC   r0,r1,r2     ; r0 = r1 - r2 + C - 1
RSB   r0,r1,r2     ; r0 = r2 - r1
RSC   r0,r1,r2     ; r0 = r2 - r1 + C - 1
```

- All operations can be viewed as either unsigned or 2's complement signed.
  - Means the same thing.

- **Bit-wise logical instructions:**

```
AND    r0,r1,r2      ; r0 = r1 and r2
ORR    r0,r1,r2      ; r0 = r1 or  r2
EOR    r0,r1,r2      ; r0 = r1 xor r2
BIC    r0,r1,r2      ; r0 = r1 and not r2
```

- BIC is the acronym for "*bit clear*"
  - Each 1-bit in r2 clears the corresponding bit in r1.

- **Register-register move instructions:**

```
MOV    r0,r2          ; r0 = r2
MVN    r0,r2          ; r0 = not r2
```

- MVN is the acronym for "*move negated*"
  - Each 1-bit in r2 clears the corresponding bit in r0.
- In the instruction encoding, the first operand r1 is not specified, as these are unary operations.

- **Comparison instructions:**

```
CMP   r1,r2        ; set cc on (r1 - r2)
CMN   r1,r2        ; set cc on (r1 + r2)
TST   r1,r2        ; set cc on (r1 and r2)
TEQ   r1,r2        ; set cc on (r1 xor r2)
```

- All these instructions affect the condition codes (N, Z, C, V) in the current program status register (CPSR).
  - These instructions do not produce result in any register (r0).

- **Specifying immediate operands:**

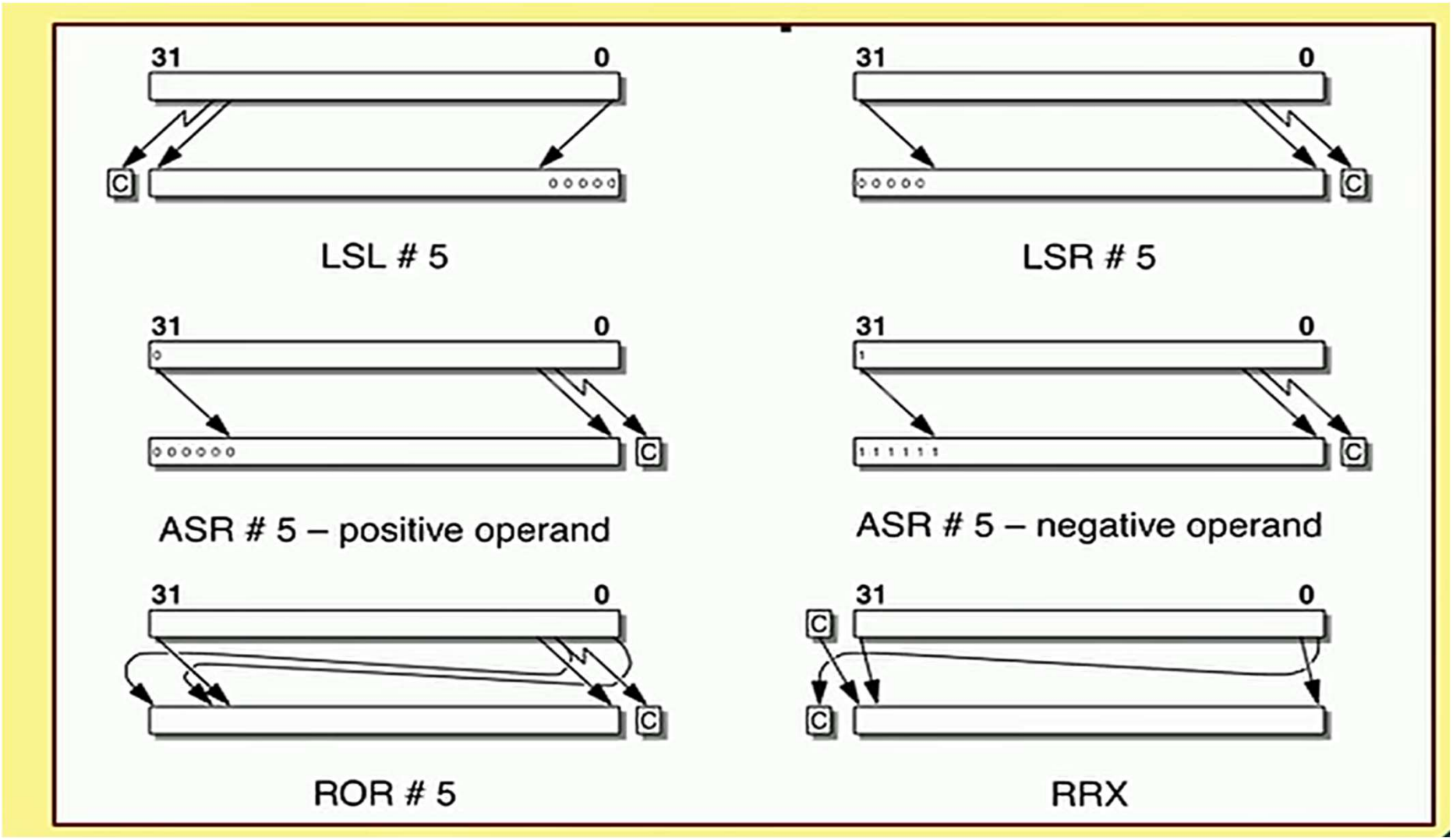
```
ADD    r1,r2,#2          ; r1 = r2 + 2
SUB    r3,r3,#1          ; r3 = r3 - 1
AND    r6,r4,#&0f        ; r6 = r4[3:0]
```

- Notations:
  - # indicates immediate value
  - & indicates hexadecimal notation
- Allowed immediate values:
  - 0 to 255 (8 bits), rotated by any number of bit positions that is multiple of 2.

- **Shifted register operands:**
  - The second source operand may be shifted either by a constant number of bit positions, or by a register-specified number of bit positions.

```
ADD    r1,r2,r3,LSL #3    ; r1 = r2 + (r3 << 3)
ADD    r1,r2,r3,LSL r5    ; r1 = r2 + (r3 << r5)
```

- Various shift and rotate options:
  - **LSL**: logical shift left
  - **LSR**: logical shift right
  - **ROR**: rotate right
  - **RRX**: rotate right extended by 1 bit
  - **ASL**: arithmetic shift left
  - **ASR**: arithmetic shift right


31
0
C
LSL # 5
31
0
C
LSR # 5
31
0
C
ASR # 5 – positive operand
31
0
C
ASR # 5 – negative operand
31
0
C
ROR # 5
C
31
0
C
RRX

- **Multiplication instruction**

```
MUL    r1,r2,r3              ; r1 = (r2 x r3)[31:0]
```

  - Only the least significant 32-bits are returned.
  - Immediate operands are not supported.

- **Multiply-accumulate instruction:**

```
MLA    r1,r2,r3,r4           ; r1 = (r2 x r3 + r4)[31:0]
```

  - Required in digital signal processing (DSP) applications.

- Multiplication with 64-bit results is also supported.

- Data transfer instructions
- Single register transfer instructions
- Multiple register transfer instructions
- Memory-mapped I/O in ARM

(b) Data Transfer Instructions

- ARM instruction set supports three types of data transfers:
  a) Single register loads and stores
     - Flexible, supports byte, half-word and word transfers
  b) Multiple register loads and stores
     - Less flexible, multiple words, higher transfer rate
  c) Single register-memory swap
     - Mainly for system use (for implementing locks)

- All ARM data transfer instructions use *register indirect addressing*.
  - Before any data transfer, some register must be initialized with a memory address.

```
ADRL   r1,Table      ; r1 = memory address of Table
```

  - Example:

```
LDR    r0,[r1]       ; r0 = mem[r1]
STR    r0,[r1]       ; mem[r1] = r0
```

- **Single register loads and stores**
  - The simplest form uses register indirect without any offset:

```
LDR    r0,[r1]       ; r0 = mem[r1]
STR    r0,[r1]       ; mem[r1] = r0
```

  - An alternate form uses register indirect with offset (limited to 4 Kbytes):

```
LDR    r0,[r1,#4]    ; r0 = mem[r1+4]
STR    r0,[r1,#12]   ; mem[r1+12] = r0
```

  - We can also use auto-indexing in addition:

```
LDR    r0,[r1,#4]!    ; r0 = mem[r1+4],   r1 = r1 + 4
STR    r0,[r1,#12]!   ; mem[r1+12] = r0,  r1 = r1 + 4
```

- We can use post indexing:

```
LDR   r0,[r1],#4   ; r0 = mem[r1],   r1 = r1 + 4
STR   r0,[r1],#12  ; mem[r1] = r0,   r1 = r1 + 12
```

- We can specify a byte or half-word to be transferred:

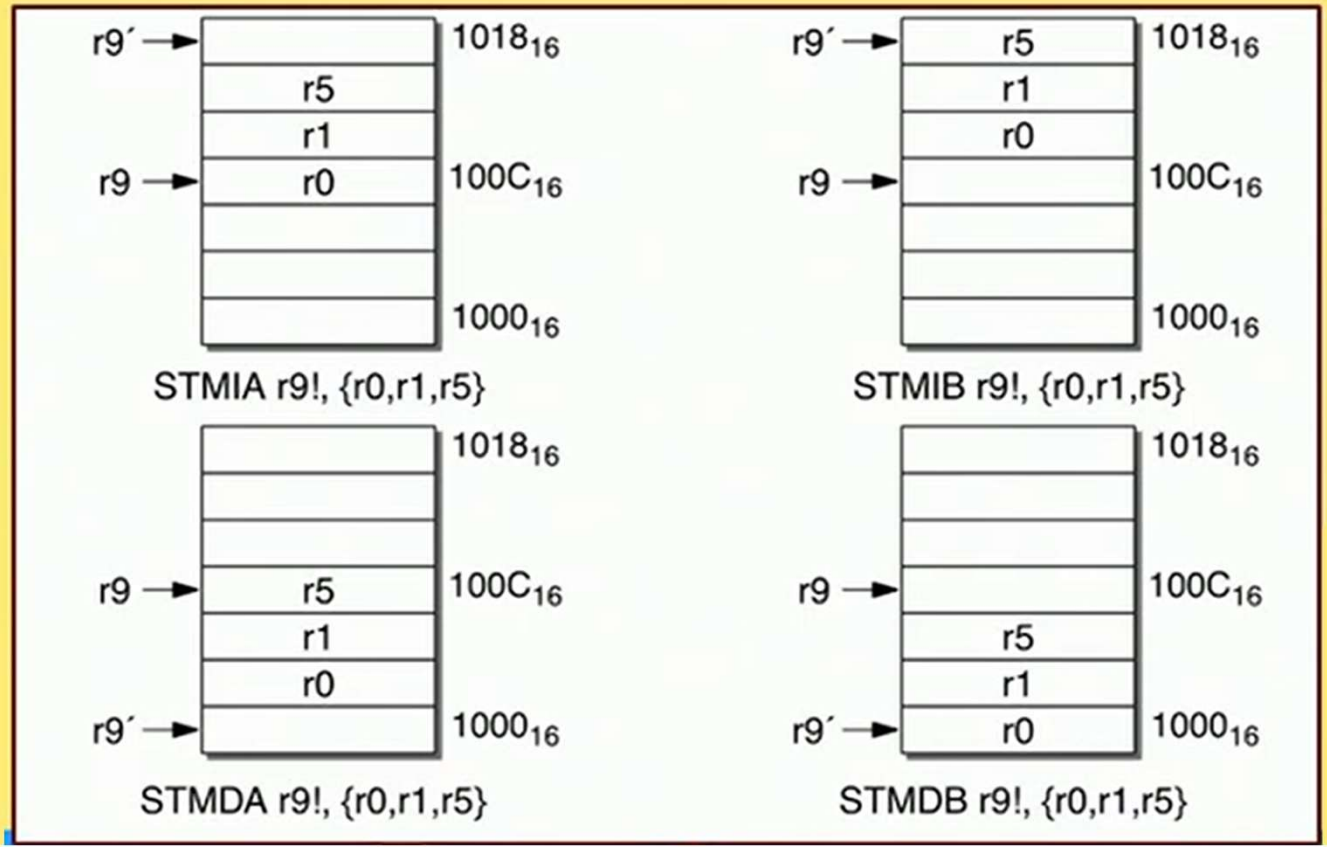
```
LDRB  r0,[r1]      ; r0 = mem8[r1]
STRB  r0,[r1]      ; mem8[r1] = r0
LDRSH r0,[r1]      ; r0 = mem16[r1]
STRSH r0,[r1]      ; mem16[r1] = r0
```

- **Multiple register loads and stores**
  - ARM supports instructions that transfer between several registers and memory.
  - Example:

```
LDMIA r1,{r3,r5,r6}      ; r3 = mem[r1]
                         ; r5 = mem[r1+4]
                         ; r6 = mem[r1+8]
```

  - For **LDMIB**, the addresses will be **r1+4**, **r1+8**, **and r1+12**.
  - The list of destination registers may contain any or all of r0 to r15.
- Block copy addressing
  - Supported with addresses that can *increment (I)* or *decrement (D)*, before (B) or *after (A)* each transfer.

- Examples of addressing modes in multiple-register transfer:



LDMIA, STMIA
- Increment after

LDMIB and STMIB
- Increment before

LDMDA, STMDA
- Decrement after

LDMDB, STMDB
- Decrement before

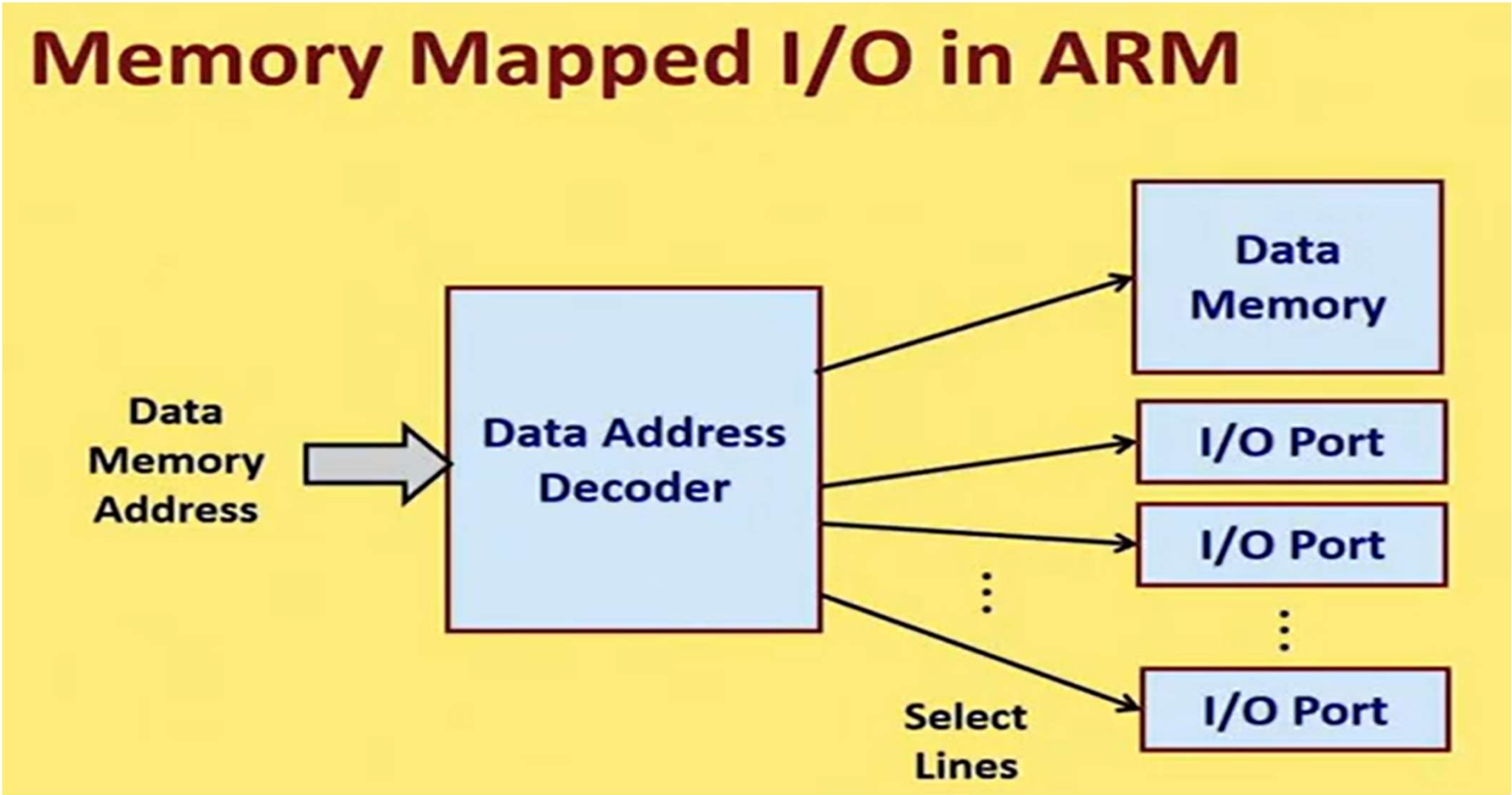
- Point to note:
  - ARM does not support any hardware stack.
  - Software stack can be implemented using the **LDM** and **STM** family of instructions.

An Example

- Copy a block of memory (128 bytes aligned).
  - r9: address of the source
  - r10: address of the destination
  - r11: end address of the source

```
Loop:   LDMIA   r9!,{r0-r7}
        STMIA   r10!,{r0-r7}
        CMP     r9,r11
        BNE     Loop
```

Memory Mapped I/O in ARM



- No separate instructions for input/output.
- The I/O ports are treated as data memory locations.
  - Each with a unique (memory) address.
- Data input is done using the LDR instruction.
- Data output is done with the STR instruction.

- Control flow instructions
- Branch and link instruction
- Conditional execution instructions

(c) Control Flow Instructions

- These instructions change the order of instruction execution.
  - Normal flow is sequential execution, where PC is incremented by 4 after executing every instruction.
- Types of conditional flow instructions:
  - Unconditional branch
  - Conditional branch
  - Branch and Link
  - Conditional execution

- **Unconditional branch instruction:**

```
                B       Target
                ...
                ...
Target          ...
```

- **Conditional branch instruction:**

```
                MOV     r2,#0
LOOP            ...
                ...
                ADD     r2,r2,#1
                CMP     r0,#20
                BNE     LOOP
                ...
```

- Branch conditions that are supported:
  - B, BAL — Unconditional branch
  - BEQ, BNE — Equal or not equal to zero
  - BPL, PMI — Result positive or negative
  - BCC, BCS — Carry set or clear
  - BVC, BVS — Overflow set or clear
  - BGT, BGE — Greater than, greater or equal
  - BLT, BLE — Less than, less or equal

- **Branch and link instruction**
  - Used for calling subroutines in ARM.
  - The return address is saved in register r14 (called *link register*).
  - To return from the subroutine, we have to jump back to the address stored in r14.

```
            BL    MYSUB          ; Branch to subroutine
            ...                  ; Return here
            ...
MYSUB       ...                  ; Subroutine starts here
            ...
            MOV   pc,r14         ; Return
```

Nested subroutine calls cannot be used in this way.

- We can use software stack to save/restore the return address and registers.
- An example showing nested subroutine calls and return.

```
          BL      MYSUB1
          ...

MYSUB1    STMFD   r13!,{r0-r2,r14}
          BL      MYSUB2
          ...
          LDMFD   r13!,{r0-r2,pc}

MYSUB2    ...
          ...
          MOV     pc,r14
```

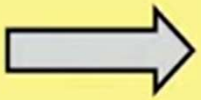
Nested subroutine calls can be used.

- **Conditional execution**
  - A unique feature of the ARM instruction set.
  - All instructions can be made conditional, i.e. will get executed only when a specified condition is true.
  - Helps in removing many short branch instructions (improves performance and code density)
  - An example: `if (r2 != 10) r5 = r5 +10 - r3`

```
CMP     r2,#10
ADDNE   r5,r5,r2
SUBNE   r5,r5,r3
```

- **Conditional execution**
  - A unique feature of the ARM instruction set.
  - All instructions can be made conditional, i.e. will get executed only when a specifi condition is true.
  - Helps in removing many short branch instructions (improves performance and coc
  - An example: `if (r2 != 10) r5 = r5 +10 - r3`

```
      CMP     r2,#10
      BEQ     SKIP
      ADD     r5,r5,r2
      SUB     r5,r5,r3
SKIP  ...
```



```
CMP       r2,#10
ADDNE     r5,r5,r2
SUBNE     r5,r5,r3
```

- Various instruction postfix supported for conditional execution:

Postfix	Condition
CS	Carry set
EQ	Equal (zero set)
VS	Overflow set
GT	Greater than
GE	Greater than or equal
PL	Plus (positive)
HI	Higher than
HS	Higher or same (i.e. CS)

Postfix	Condition
CC	Carry clear
NE	Not equal (zero clear)
VC	Overflow clear
LT	Less than
LE	Less than or equal
MI	Minus (negative)
LO	Lower than (i.e. CC)
LS	Lower or same

- Another example:

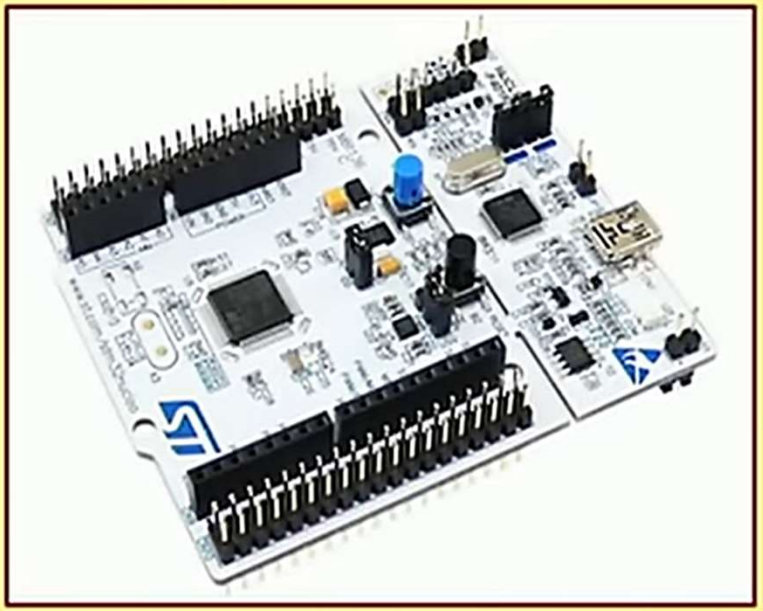
```
if ((r1 == r3) && (r5 == r6)) r7 = r7 + 10
```

```
      CMP     r1,r3
      BNE     SKIP
      CMP     r5,r6
      BNE     SKIP
      ADD     r7,r7,#10
SKIP  ...
```

```
CMP     r1,r3
CMPEQ   r5,r6
ADDEQ   r7,r7,#10
```

- About the STM32F401 Nucleo board
- Board features
- Connector details

About the STM32F401 Nucleo Board



- Developed by ST Microelectronics.
- CPU: ATM Cortex© M4
- 512 KB Programmable Flash Memory
- 96 KB SRAM
- USB 2.0: Type A to Mini B
- Mbed Enabled (mbed.org)
- Supports a wide choice of Integrated Development Environments (IDEs) including IAR™, ARM® Keil®, GCC-based IDEs.

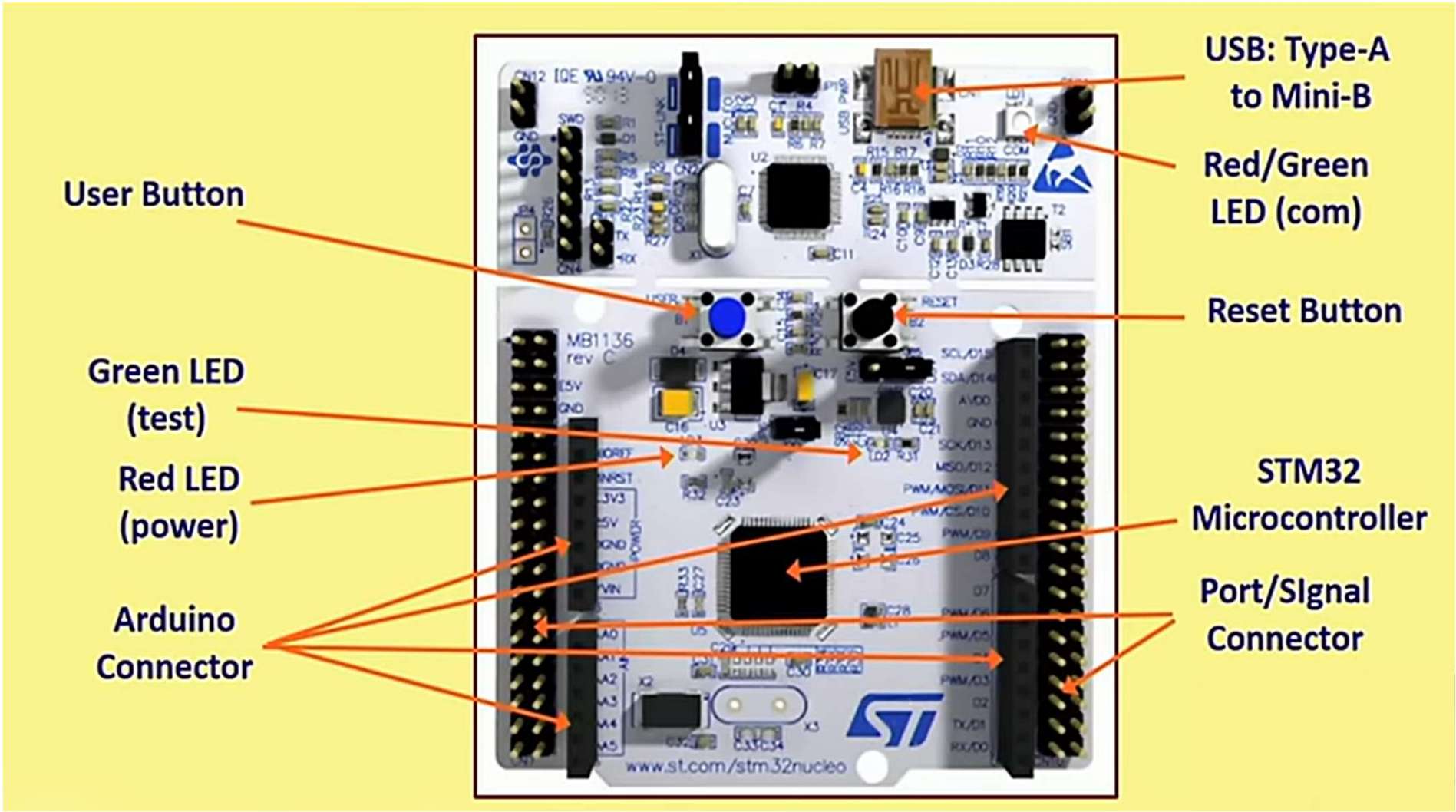
Some Specific Details

- Contains STM32F401RET6 microcontroller with ARM Cortex-M4 CPU with FPU at 84 MHz clock, 512 KB of flash and 96 KB of SRAM with a variety of peripherals.
- Two types of extension connectors are available:
  - Arduino UNO Revision 3 connectivity.
  - STM Morpho extension pins for full access to all STM31F401RET6 I/Os.
- On-board ST-LINK/V2.1 interfacing with the PC, which provides:
  - Debug and programming port, to use the board with standard prog
  - Virtual Com port to send back traces to the PC.
  - Mass storage (USB Disk Drive) for drag-n-drop programming.

Some Specific Details

- Contains STM32F401RET6 microcontroller with ARM Cortex-M4 CPU with FPU at 84 MHz clock, 512 KB of flash and 96 KB of SRAM with a variety of peripherals.
- Two types of extension connectors are available:
  - Arduino UNO Revision 3 connectivity.
  - STM Morpho extension pins for full access to all STM31F401RET6 I/Os.
- On-board ST-LINK/V2.1 interfacing with the PC, which provides:
  - Debug and programming port, to use the board with standard programmers.
  - Virtual Com port to send back traces to the PC.
  - Mass storage (USB Disk Drive) for drag-n-drop programming.

- Input/Output
  - 50 general-purpose I/O pins with external interrupt capability.
  - 12-bit ADC with 16 channels.
  - 7 general-purpose timers.
  - 2 watchdog timers.
  - USB 2.0 interface.
  - And several others …
- User LED (LD2)
- Two push buttons: USER and RESET.


USB: Type-A
to Mini-B
Red/Green
LED (com)
User Button
Reset Button
Green LED
(test)
Red LED
(power)
STM32
Microcontroller
Arduino
Connector
Port/SIgnal
Connector
MB1136
rev C
www.st.com/stm32nucleo

Some Color Conventions Followed

Labels usable in code

- PX_Y — MCU pin without conflict
- PX_Y — MCU pin connected to other components
  See **PeripheralPins.c** (link below) for more information
- XXX — Arduino connector names (A0, D1, ...)
- XXX — LEDs and Buttons (LED_1, USER_BUTTON, ...)

Labels not usable in code (for information only)

- XXX — Serial pins (USART/UART)
- XXX — SPI pins
- XXX — I2C pins
- XXX — PWMOut pins (TIMER n/c[N])
  n = Timer number c = Channel
  N = Inverted channel
- XXX — AnalogIn (ADC) and AnalogOut pins (DAC)
- XXX — CAN pins
- XXX — Power and control pins (3V3, GND, RESET, ...)

The Nomenclatures

- SPI: Serial Peripheral Interface
- I2C: Inter-Integrated Circuit
  - Serial communication bus, widely used for attaching lower-speed peripheral ICs to microcontrollers over short distance.
- PWM: Pulse Width Modulation
- CAN: Controller Area Network
  - A robust vehicle bus standard, designed to allow microcontrollers and devices to communicate with each other without a host computer.

- Pulse width modulation (PWM)
- Using PWM on mbed platform
- Using interrupts on STM32 board

Pulse Width Modulation

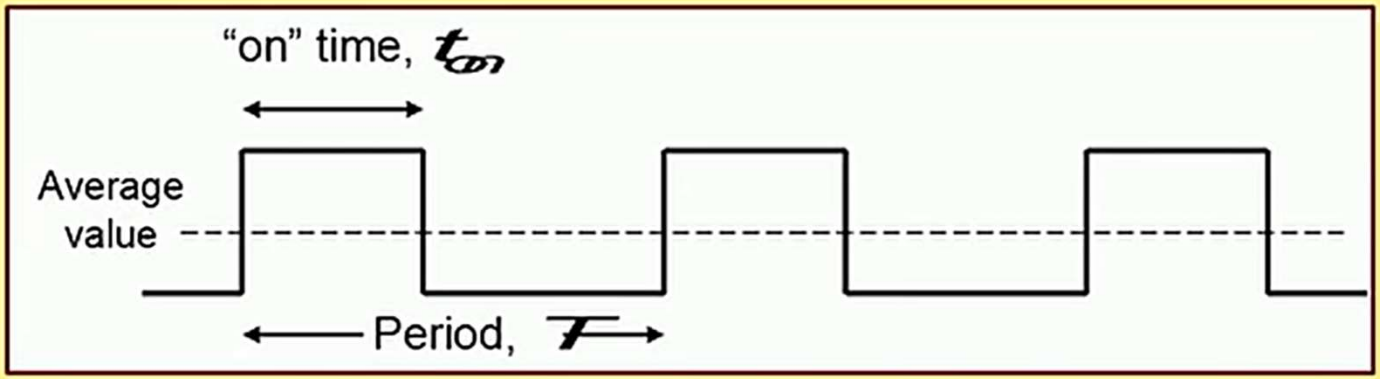
What is Pulse Width Modulation (PWM)?

- PWM is a simple method of using a rectangular digital waveform to control an analog variable.
  - The on-off behavior changes the average power of the signal.
  - Output signal alternates between ON and OFF with a specific time period.
- PWM control is used in a variety of applications, ranging from communications to automatic control.
- It can also be used to encode information for data transmission.

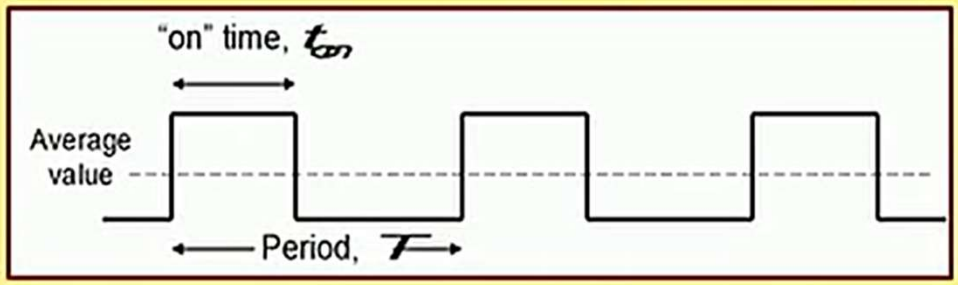
How it works?

- The period is normally kept constant, and the pulse width (or ON time) is varied.
- **Duty Cycle**: It is defined as the proportion of time the pulse is ON, expressed as a percentage.

$$\textit{Duty Cycle} = \textit{(pulse ON time) / (pulse period) * 100\%} = t_{on} / T * 100\%$$



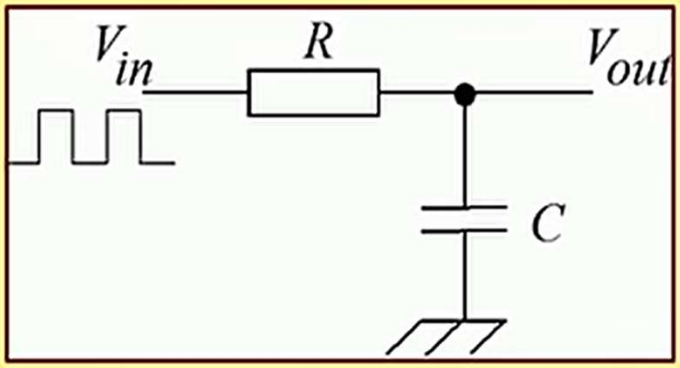
- Whatever duty cycle a PWM has, there is an *average value*, as indicated by the dotted line.
  - If the ON time is small, the average value is low; if it is large, the average value is high.
  - By controlling the duty cycle, we can control the average value.



- Average value of the signal = $\frac{1}{T}\int_0^T f(t)\,dt = t_{on}.V_H + (1 - t_{on}).V_L$
- In general, V_L is taken as 0V for ease of calculation.
  - Average value becomes $t_{on}.V_H$

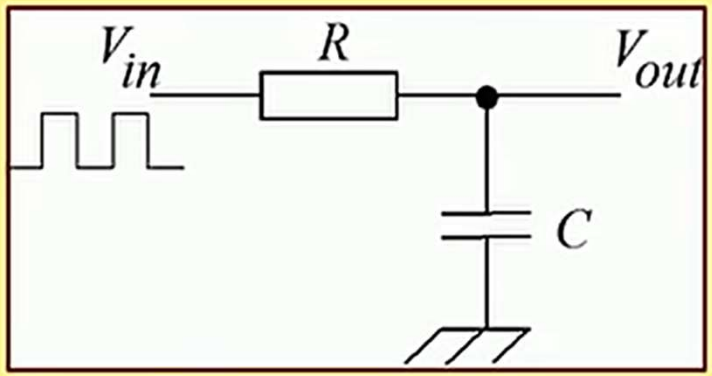
How to Extract the Average Value?

- The average value can be extracted from the PWM stream using a low-pass filter.
- If the PWM frequency and the values of R and C are appropriately chosen, V_{out} becomes an analog output.
  - Can be used in place of a digital-to-analog converter.



How to Extract the Average Value?

- The average value can be extracted from the PWM stream using a low-pass filter.
- If the PWM frequency and the values of R and C are appropriately chosen, V_{out} becomes an analog output.
  - Can be used in place of a digital-to-analog converter.



In practice, the filter is not always required.

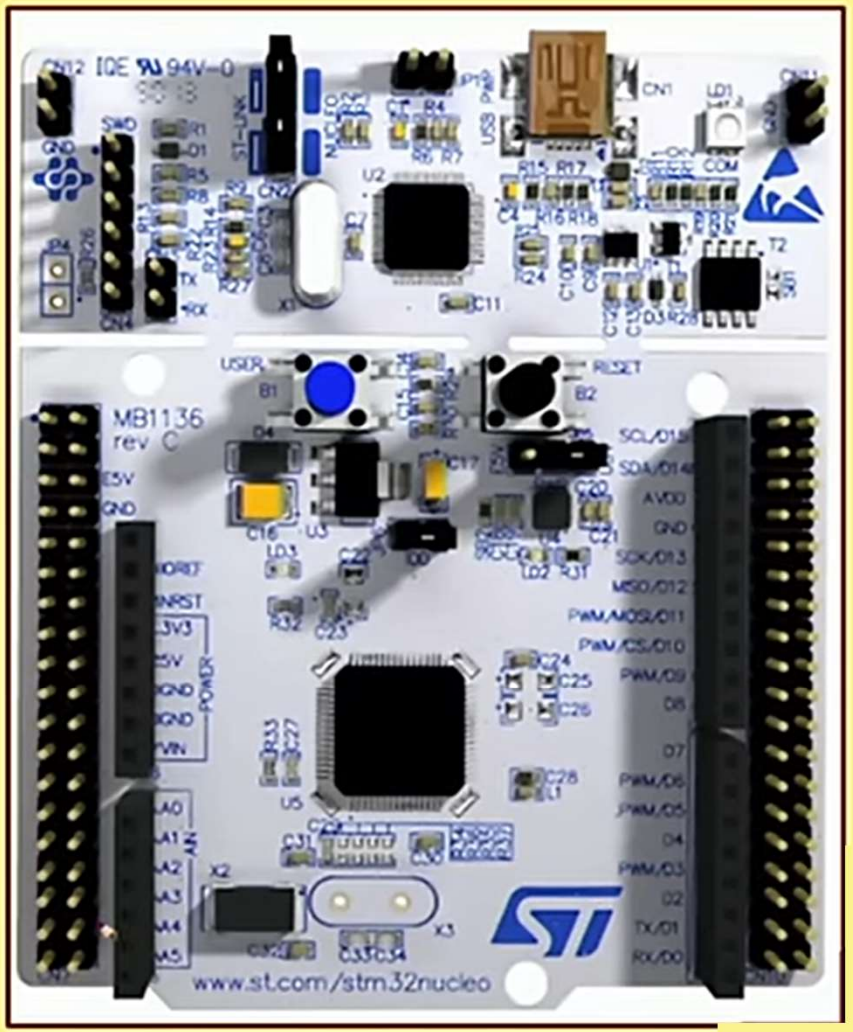
Many physical systems have response characteristics that act like low-pass filters.

Some Typical Applications

1. Control of DC motor.
   - The voltage supplied to the motor is proportional to the duty cycle.
2. Controlling the brightness of LED.
   - The duty cycle of the voltage source determines the brightness.
3. Control the temperature (heater).
   - Switch ON and OFF the heater with an appropriate duty cycle.
4. Many more …

PWM on the STM32F401

- The *PwmOut interface* is used to control the frequency and duty cycle of a digital pulse train.
- The *Arduino Interface* supports up to 6 PWM outputs, although these *PwmOut ports* share the same period timer.



PWM Library Functions Available in mbed.h

Function	
PwmOut	Create a PwmOut connected to the specified pin
write	Set the output duty cycle, specified as normalized float (0.0 to 1.0)
read	Return the current output duty cycle setting (0.0 to 1.0)
period period_ms period_us	Set the PWM period, specified in seconds (float), milli-seconds (int) or micro-seconds (int), keeping the duty cycle the same.
pulsewidth pulsewidth_ms pulsewidth_us	Set the PWM pulsewodth, specified in seconds (float), milli-seconds (int) or micro-seconds (int), keeping the period the same.
operator =	Shorthand for write()
operator float()	Shorthand for read()