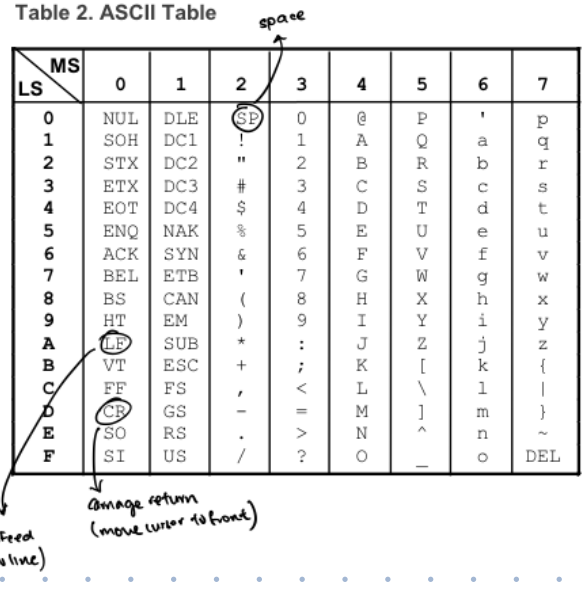
Appendix 1 - Summary of VisUAL ARM Instruction Set and Other Useful Information

Summary	Opcode	Syntax
Move	MOV	MOV{S}{cond} dest, op1 {, SHIFT_op #expression}
Move Negated	MVN	MVN{S}{cond} dest, op1 {, SHIFT op #expression}
Address Load	ADR	ADR{S}{cond} dest, expression
LDR Psuedo-Instruction	LDR	LDR{S}{cond} dest, =expression
Add	ADD	ADD{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Add with Carry	ADC	ADC{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Subtract	SUB	SUB{S}{cond} dest, op1, op2 {, SHIFT op #expression}
Subtract with Carry	SBC	SBC{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Reverse Subtract	RSB	RSB{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Reverse Subtract with Carry	RSC	RSC{S}{cond} dest, op1, op2 {, SHIFT op #expression}
Bitwise And	AND	AND{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Bitwise Exclusive Or	EOR	EOR{S}{cond} dest, op1, op2 {, SHIFT op #expression}
Bitwise Clear	BIC	BIC{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Bitwise Or	ORR	ORR{S}{cond} dest, op1, op2 {, SHIFT op #expression}
Logical Shift Left	LSL	LSL{S}{cond} dest, op1, op2
Logical Shift Right	LSR	LSR{S}{cond} dest, op1, op2
Arithmetic Shift Right	ASR	ASR{S}{cond} dest, op1, op2
Rotate Right	ROR	ROR{S}{cond} dest, op1, op2
Rotate Right and Extend	RRX	RRX{S}{cond} op1, op2
Compare	CMP	CMP{cond} op1, op2 {, SHIFT_op #expression}
Compare Negated	CMN	CMN{cond} op1, op2 {, SHIFT op #expression}
Test Bit(s) Set	TST	TST{cond} op1, op2 {, SHIFT_op #expression}
Test Equals	TEQ	TEQ{cond} op1, op2 {, SHIFT op #expression}
Load Register	LDR	LDR{B}{cond} dest, [source {, OFFSET}] Offset addressing LDR{B}{cond} dest, [source, OFFSET]! Pre-indexed addressing LDR{B}{cond} dest, [source], OFFSET Post-indexed addressing
Store Register	STR	STR{B}{cond} source, [dest {, OFFSET}] Offset addressing STR{B}{cond} source, [dest, OFFSET]! Pre-indexed addressing STR{B}{cond} source, [dest], OFFSET Post-indexed addressing
Load Multiple Registers	LDM[dir]	LDM[dir]{cond} source, {list of registers}
Store Multiple Registers	STM[dir]	STM[dir]{cond} dest, {list of registers}
Branch	B	B{cond} target
Branch with Link	BL	BL{cond} target
Declare Word(s) in Memory	DCD	name DCD value 1, value 2, ... value N
Declare Constant	EQU	name equ expression
Declare Empty Word(s) in	FILL	{name} FILL N
Stop Emulation	END	END{cond}

Table 1. Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.

Table 2. ASCII Table



MS / LS	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Appendix 1

VIP Instruction Set Summary Chart

VIP Instruction Encoding – Opcode Formats

11	10	9	8	7	6	5	4	3	2	1	0
0-7	Dual operand			d				s			
8	Short Move			d				n			
9-A	Unary/Control			op-code				operand = s, d or n			
B-F	JMP			2's complement -128 to +127 *relative*							

Group 1 – Dual-Operand Instructions (Opcode: 000 to 8FF)

Bits 8-11	Name	Bits 4-7	Bits 0-3	Operation	Flags
0	MOV	d	s	d ← s	NZ
1	AND	d	s	d ← d .AND. s	NZ
2	OR	d	s	d ← d .OR. s	NZ
3	EOR	d	s	d ← d .EOR. s	NZ
4	ADD	d	s	d ← d + s	VNZC
5	ADDC	d	s	d ← d + s + carry	VNZC
6	SUB	d	s	d ← d + (.NOT. s) + 1	VNZC
7	CMP	d	s	d + (.NOT. s) + 1	VNZC
8	MOVS	d	n	d ← n	

Group 2 – Unary and Control Instructions (Opcode: 900 to 9FF)

Bits 4-7	Name	Bit 0-3	Operation	Flags
0	INC	d	d ← d + 1	C
1	DEC	d	d ← d + 0xFFF	NZC
2	ROR	d	Rotate d right : msb ← lsb; and C ← lsb	NZC
3	ROL	d	Rotate d left : lsb ← msb; and C ← msb	NZC
4	RRC	d	Rotate d right including carry	NZC
5	RLC	d	Rotate d left including carry	NZC
6	RAR	d	Rotate d 'arithmetic' right preserving msb	NZC
7	PRSG	d	Left shift lsb from EOR (bits 11,5,3,0)	NZC
8	INV	d	d ← .NOT. d	NZ
9	NEG	d	d ← (.NOT. d) + 1	NZC
A	DADD	s	AR ← AR + s + carry (as 3 BCD digits)	ZC
B	UMUL	s	R1:R0 ← unsigned R0 times unsigned s	Z
C	TST	s	s + 0	NZ
D	EXEC	s	Execute s as an instruction	implied
E	BCSR	n	SR (bits 3-0) ← SR .AND. (.NOT. n)	explicit
F	BSSR	n	SR (bits 3-0) ← SR .OR. n	explicit

Group 3 – Unary and Control Instructions (Opcode: A00 to AFF)

Bits 4-7	Name	Bits 0-3	Operation	Flags
0	PSH	s	SP ← SP-1; (SP) ← s	
1	POP	d	d ← (SP); SP ← SP+1	explicit if d=SR
2	PSHM	3:2:1:0	Push R3:2:1:0 to stack, R3 first	
3	POPM	3:2:1:0	Pop R3:2:1:0 from stack, R3 last	
4	CALL	s	SP ← SP-1; (SP) ← Return Address PC ← Effective address	
5	RET	n	PC ← (SP) + n; SP ← SP+1	
6			See subgroup 3a	
7	RCN	n	Count for next rotate instruction. if n=0 use bits 3:2:1:0 of AR	
8	JDAR	±n	AR ← AR-1, if AR != 0, PC ← PC ± n	
9	JPE	±n	If parity of AR is even, PC ← PC ± n	
A	JPL	±n	If N = 0, PC ← PC ± n	
B	JVC	±n	If V = 0, PC ← PC ± n	
C	JGE	±n	If N = V, PC ← PC ± n	
D	JLT	±n	If N != V, PC ← PC ± n	
E	JGT	±n	If Z = 0 and N = V, PC ← PC ± n	
F	JLE	±n	If Z = 1 or N != V, PC ← PC ± n	

Group 1 – Jump Instructions (8-bit Range) (Opcode: B00 to FFF)

Bits 8-11	Name	n = Bits 0 to 7	Operation
B	JMP = BRA	-128 to +127	PC ← PC ± n
C	JEQ = JZ	-128 to +127	If Z=1, PC ← PC ± n
D	JNE = JNZ	-128 to +127	If Z=0, PC ← PC ± n
E	JHS = JC	-128 to +127	If C=1, PC ← PC ± n
F	JLO = JNC	-128 to +127	If C=0, PC ← PC ± n

Group 3a – Control Instructions (Opcode: A60 to A6F)

Bits 4-7	Name	Bits 0-3	Operation
6	RETI	0	SR ← (SP); SP ← SP+1; PC ← (SP); SP ← SP+1
6	SWI	1	SP ← SP-1; (SP) ← PC; SP ← SP-1; (SP) ← SR; PC ← (0x009)
6	WAIT	2	IE ← 1; Execution resumes after interrupt signal
6	HALT	3	Stop execution. Non-maskable interrupt or hardware reset to exit.
6	STOP	4	Stop execution. Reset to exit.
6	SYNC	8	Pulse SYNC output pin high for 1 clock cycle
6	NOP	9	No operation
6	LOCK	A	Block interrupts and bus sharing
6	UNLK	B	Allow interrupts and bus sharing
6	MSS	C to F	Memory Space Select override

Addressing Modes

Hex	Symbol	Location of Data	Availability
0	R0	Register R0	Both d and s
1	R1	Register R1	Both d and s
2	R2	Register R2	Both d and s
3	R3	Register R3	Both d and s
4	[R0]	Register R0 indirect	Both d and s
5	[R1]	Register R1 indirect	Both d and s
6	[R2+n]	Register R2 with offset indirect	Both d and s
7	[R3+n]	Register R3 with offset indirect	Both d and s
8	AR	Data is in Auxiliary Register	Both d and s
9	SR	Status Register	Both d and s
A	SP	Stack Pointer	Both d and s
B	PC	Program Counter	Both d and s
C	#n	Immediate, (or just n for CALL)	s only
D	[n]	Absolute (code space for CALL)	Both d and s
E	[SP+n]	SP with offset indirect	Both d and s
F	[PC+n]	PC with offset indirect (CALL is relative with PC+n)	Both d and s

Notation: d = destination; s = source

Description of bits in Status Register

SR	F	R	Description
11-8	*	*	Reserved
7-4	*	*	Defined but not described here
3	V	0	Set if 2's complement sign is incorrect
2	N	0	Is most significant bit of result
1	Z	0	1 if result is zero, otherwise 0
0	C	0	1 if carry out, otherwise 0

Notation: SR = Bits in register; F = Name of flag; R = Value after reset

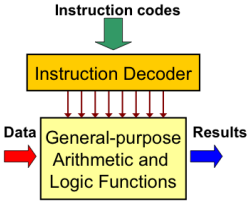
Chapter 1 Computer Hardware Decomposition.

Basics of a computer is always the same:
① Central Processing Unit (CPU)
② Memory (DRAM, VRAM)
③ I/O Interfaces.

All components are connected via a bus (Data, Address, Control)

Chapter 2 Data Organisation in Memory

① Describe the concept of programming in software.



* slower and more complex but easier to program.
* Instructions are fetched from memory one at a time when the program is executed.
* Each function is specified by bit patterns.

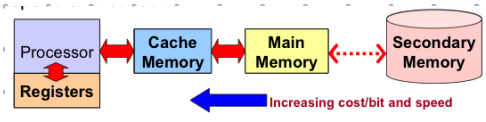
Programming in Software

*provides flexibility.

② Describe the von Neumann's stored program concept.
- Two architectures, ① Harvard: Data and instructions are stored in different memories
② von Neumann: Data and instructions are stored in the same memory.
↳ contents of memory are addressable by location, without regard to data type.
↳ execution occurs sequentially, unless explicitly modified.

③ Describe the role of memory in computing.
- Memory stores both instructions and data, and each address stores a fixed number of bits (8 bits / 1 byte)
- Each address is accessed by specifying its binary pattern on the address bus.
- Consecutive locations are used to store multi-byte data.
!! memory size is dependent on the size of address bus.

④ Describe the characteristics and functions of the different elements in the memory hierarchy.



CPU's own mem.
Motherboard mem.
RAM sticks
SSD / HDD

Registers	Very fast access but limited numbers within CPU. Operates at CPU clock rate (size: 2-128 registers)
Cache Memory	Fast access static RAM close to CPU. Typical access time 3-20nS (size: up to 512 kB)
Main memory	Usually dynamic RAM or ROM (for program storage). Typical access time 30-70nS. (size: up to 16GB)
Secondary Memory	Not always random access but non-volatile. Maybe be based on magnetic or flash technology. Typical access time 0.03-100mS. (size: up to 4TB)

⑤ Describe the different C numeric data types and their characteristics.
⑥ Describe the concept of numeric range and its implications to data size

Type	Bytes	Bits	Range	
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
short int	2	16	-32,768 -> +32,767	(+/-32KB)
unsigned short int	2	16	0 -> +65,535	(64KB)
unsigned int	4	32	0 -> +4,294,967,295	(4GB)
int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long long int	8	64	-(2^32) -> (2^32)-1	
float	4	32		
double	8	64		
long double	12	96		

ANSI C numeric data types and their respective size and range

Note: These sizes may change depending of the processor and compiler

⑦ Describe how multi-byte numbers are stored in memory
* Depends on the byte-ordering of data in the memory.
① Big Endian [MSB stored in lower (smaller number) address]
② little Endian [LSB stored in lower (smaller number) address]

Note: When accessing data stored in LE format, reference point is LSB

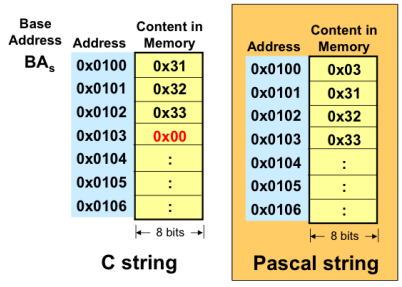
A 32-bit (4 byte) data is stored over 4 addresses.
A 12-bit (1.5 byte) data is stored over 2 addresses (round up)
] both assume address size to be 8-bits / 1 byte.

⑧ Describe the char data type and its representations
- A char variable requires one byte of memory storage
* char is converted into binary for storage using the 7-bit ASCII encoding standard.

⑨ Describe the Boolean data type and its implementation.
- Only has 2 states: ① True (any non-zero value)
② False (0)
* inefficient as one byte is used to store 1-bit

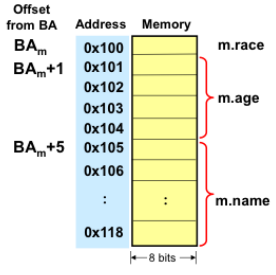
⑩ Describe the representation of arrays in memory
- Arrays store a series of similar data types in consecutive memory locations
↳ Elements are accessed using offsets from a base address (ie. the first index is used as the reference point)

⑪ Describe the C and Pascal strings storage in memory.
- C: stores null character (0x00) at the end of string.
- Pascal: stores length of string at the start of string.



⑫ Describe the representation of structures in memory
- Each data type in a declared structure variable occupies a predefined consecutive locations based on data type size.
* only allocated on declaration.

```
main()
{
 struct Man ; //define structure Man
 {
  char race;
  int age;
  char name[20];
};
 Man m;
}
```



(13) Describe the representation of pointers in memory

- Value of pointer is an address and is independent of the datatype it is pointing to. Pointer size depends on the addressable memory space of the processor.

(14) Describe the concept of data alignment.

- Restricts where in memory multi-byte data can be stored
- must be aligned to the addresses that are multiples of the size of the data type.

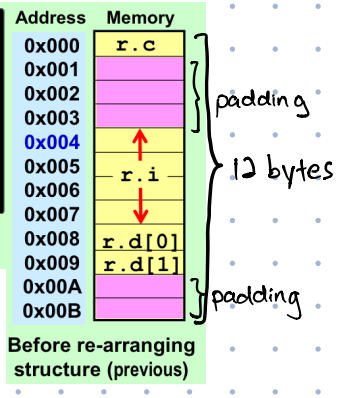
Data Type	Size (Byte)	Example of allowable start addresses due to alignment
char	1	0x..0000, 0x..0001, 0x..0002
short	2	0x..0000, 0x..0002, 0x..0004
int	4	0x..0000, 0x..0004, 0x..0008
float	4	0x..0000, 0x..0004, 0x..0008
double	8	0x..0000, 0x..0008, 0x..0010
pointer	8	0x..0000, 0x..0008, 0x..0010

- extend depends on the size of the processor's data bus (ie data bus size determines how data will be transported) * Refer back to slide 20 if do not understand.

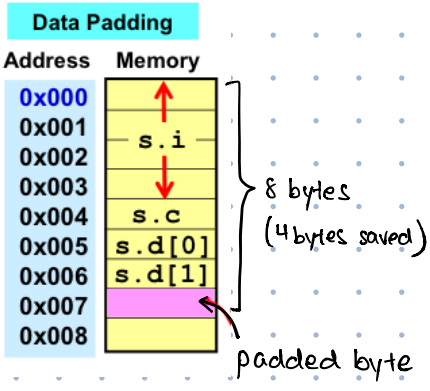
(15) Describe data alignment considerations for efficient access and storage of structure variables in memory

- Data padding is used to meet alignment restrictions.
  * same rule in (14) applies to storage address of each data type.
  * rearrange order of data objects in structures to minimise padding (save memory space)

```
struct rec1 {
  char c;
  int i;
  char d[2];
 }
 rec1 r;
```




```
struct rec2 {
  int i;
  char c;
  char d[2];
 }
 rec2 s;
```



Chapter 3 Instructional Organisation in Memory

① Describe the characteristics and role of an instruction
- Binary encoded and stored in memory, executable unlike data
- Consists of: ① Op-code - Specifies the operation to be carried out (ie MOV, LDR)
  ② Operands - specifies the data itself or the location of data and where results is to be stored.

  * length of Op-code depends on the number of operations available (more variety = more bits required to represent)
  * Operands can be stored as part of the instruction, stored in a register or stored in memory.
- How the operand is specified is known as addressing mode.

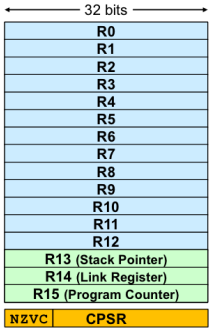
Addressing Mode	ARM	Intel
Absolute (Direct)	None	`MOV AX,[1000h]`
Register Direct	`MOV R1,R0`	`MOV AX,DX`
Immediate	`MOV R1,#3`	`MOV AX,0003h`
Register Indirect	`LDR R1,[R0]`	`MOV AX,[BX]`
Register Indirect with Offset	`LDR R1,[R0,#4]`	`MOV AX,[BX+4]`
Register Indirect with Index	`LDR R1,[R0,R2]`	`MOV AH,[BX+DI]`
Implied	`BNE LOOP`	`JMP -8`

} Mnemonics

- The role of an instruction is to make purposeful changes to the current state of the processor. [① Data Processing ② Data Transfer ③ Program Control]

② Describe the function of the ARM instruction set MOV instruction.
- MOV operator takes two operands and copies the source operand (second) to the destination operand (first)

③ Describe the ARM programmer's model
- 16 32-bits registers
  ↳ R0-R12 are general purpose registers
  ↳ R13 is the Stack Pointer (SP)
  ↳ R14 is the Link Register (LR)
  ↳ R15 is the Program Counter (PC)
  * The Current Processor Status Register (CPSR) holds the condition code bits (NZVC - Neg, Zero, Overflow, Carry)



CPSR bit(s)	Name	Description (NZVC CPSR)
31	N	Can set if last operation produced a **negative** result
30	Z	Can set if last operation produced a **zero** result
29	C	Can set if last operation produced a **carry out** in the most significant bit
28	V	Can set if the last operation produced an **overflow** for a **signed** arithmetic operation

④ Describe the function and interpretation of several ARM registers.
- Stack Pointer: - used to maintain a space in memory (ie. stack) that is used to temporarily store away register information for later use

- Link Register: - Gets a copy of the Program Counter when a Branch with Link (BL) instruction is executed.
  - Can be used as a general purpose register at other times.

- Program Counter: - Keeps track of program execution
  - contains the start address of the next instruction to be fetched.
  - Automatically increments by the length of executed instruction.
  * Usually loaded in an ARM CPU due to its pipeline architecture (Stored address is 2 addresses after current)

⑤ Describe the function of a simple LDR instruction from the ARM instruction set
- Copies memory content to a register (first operand is always the destination register, second operand is a memory location whose address is stored in third register listed in the second operand)

⑥ Describe the basic execution cycle of a simple LDR instruction `LDR R1,[R0]`
- Fetch Cycle: - PC points to address of the instruction and its opcode (LDR) is fetched into the IR
  - Decoding `LDR R1,[R0]` suggests an operand is needed ([R0]) from the memory. Therefore, another fetch is needed to retrieve the data from memory into CPU.
- Execution Cycle: - Operand fetched from memory in the Fetch Cycle is loaded into the destination register(R1)

⑦ Describe the factor that determines the execution speed of an instruction.
- Data transfer slower via external bus compared to the CPU's internal bus
  ↳ System performance is limited by this traffic bandwith between the CPU and memory (Von Neumann Bottleneck)
- Keep regularly used operands in Registers help to reduce memory access.
- Keeping instructions and data in separate memories (Havard Architecture) can help make instructions execute in regular cycles which will improve performance

Chapter 4 Introduction to Assembly Programming & Addressing Modes

① Identify why and when to use assembly language.
- Codes written well in assembly language can usually execute faster and are smaller in size
- Able to exploit optimized features of a processor's ISA
- To be used at critical parts of the operating system's software, Input/Output intensive codes, and time critical codes

② Describe what are addressing modes.
- Allows the CPU to identify the actual operand or the address location where the operand is stored.
- Different instruction set architectures support different addressing modes:

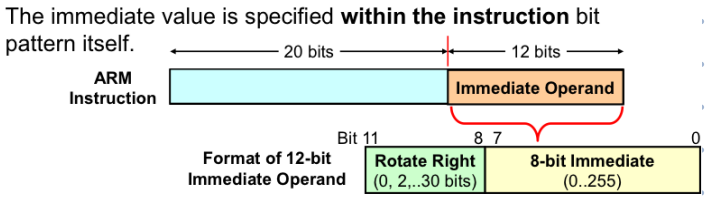
Addressing Mode	ARM	Intel
Absolute (Direct)	None	`MOV AX,[1000h]`
Register Direct	`MOV R1,R0`	`MOV AX,DX`
Immediate	`MOV R1,#3`	`MOV AX,0003h`
Register Indirect	`LDR R1,[R0]`	`MOV AX,[BX]`
Register Indirect with Offset	`LDR R1,[R0,#4]`	`MOV AX,[BX+4]`
Register Indirect with Index	`LDR R1,[R0,R2]`	`MOV AH,[BX+DI]`
Implied	`BNE LOOP`	`JMP -8`

https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

③ Describe what is register direct.
- Operand is already preloaded into the specified register.
- No need for extra access of memory during execution ⇒ execution speed is optimised.

④ Describe what is immediate addressing and its applications.

- Directly specifies the operand within the instructions itself, memory access is not incurred during execution.



*Immediate values contain 32 bits but ARM instructions only has space for 32 bits
∴ Only 12 bits are allocated for the immediate value.

- Immediate value is a number between (0-255) rotated by 2n bits where the value of n is given by 4 bits (green space) * 8 bit number to be rotated is determined by CPU and can be different from the specified value in the instruction.

* If requested immediate value cannot be encoded (eg. 0x102), assembler will throw an error.

A combination of instructions can be used to achieved the desired immediate values that is not valid.

```
MOV R1,#0x100 ; load 0x100 to R1
ADD R1,R1,#2  ; add 2 to R1
```

⑤ Describe what is register indirect and the ARM instructions that support this addressing mode.

- Registers store the memory address of operand for an instruction to be executed. (ARM uses LDR and STR mnemonics to access memory operands)

1. `LDR R1, [R0]`
   - destination
   - source (address of memory located in R0)

2. `STR R1, [R0]`
   - source (address of memory located in R0)
   - destination

⑥ Describe the variants and application of register indirect that uses base plus offset and index register.

1. Register indirect with offset (base plus offset)
   ↳ Adds a specific offset value to the indirect register to compute the effective address in memory
   * Base plus offset does not change the indirect register's content.
   `LDR R1, [R0, #4]` loads the content 4 addresses away from the address specified in [R0]
2. Register indirect with index (base plus index)
   ↳ Adds the content of the index register to the indirect register to compute the effective address.
   * Base plus index does not change the indirect register's content.
   `LDR R1, [R0, R2]` offsets the address in R1 by the value in R2 and loads content in EA to R1. R2 can be modified to allow different array elements to be retrieved.

⑦ Compare the relative pros and cons of register direct and register indirect addressing modes.

- Use base plus offset if position of array element is known during coding time
- Use base plus index if array position is computed during run-time.

⑧ Describe what is auto-indexing feature of ARM's register indirect addressing mode

- Auto-indexing modifies the indirect register besides just computing the Effective Address

1. Auto-indexing with offset
   ↳ Similar to register indirect with offset, just that source register's address gets updated with the Effective Address. ("!" in the mnemonic is used ie `LDR R1, [R0, #4]!`)
2. Auto-indexing with index
   ↳ Similar to register indirect with indexing, just that source register's address gets updated with the Effective Address ("!" in the mnemonic is used ie `LDR R1, [R0, R2]!`)

⑨ Describe the differences between pre-index and post-index addressing modes

- In pre-index, the address in the indirect register is updated before the instruction is executed
- In post-index, the address in the indirect register is updated after the instruction is executed.

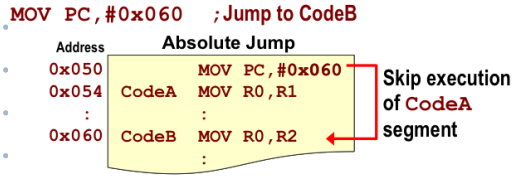
* Syntax: `LDR R1, [R0, #4]!` (offset + pre) `LDR R1, [R0, R2]!` (Index + Pre)
`LDR R1, [R0], #4` (offset + post) `LDR R1, [R0], R2` (Index + Post)

⑩ Describe the various stack implementations and operations using the ARM addressing modes.

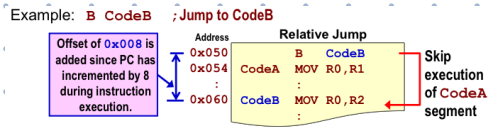
1. Full descending (stores bottom up)
   ↳ Point to smallest address value of the last element added
2. Full ascending (stores top down)
   ↳ Point to biggest address value of the last element added.
3. Empty descending (stores bottom up)
   ↳ Points to empty address after the last stored value.
4. Empty ascending (stores top down)
   ↳ Points to empty address after the last stored value

⑪ Describe difference between absolute and relative jump.

- Absolute jump: done by loading the address to jump to into the Program Counter
  * Not position independent, can only execute correctly in this specific area of code memory

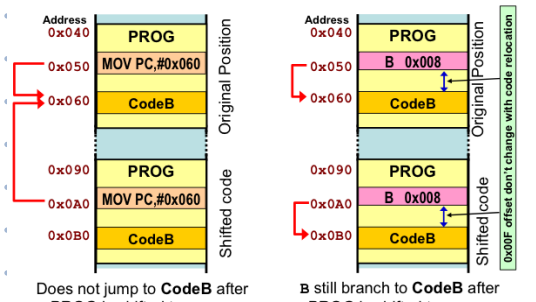


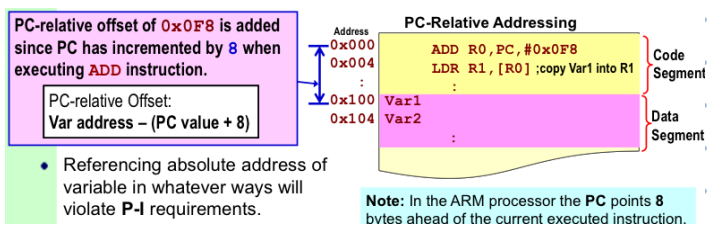
- Relative jump: done by using the branch instruction (B) with an appropriate signed offset. offset is added to the PC to alter the sequential order of program execut^n



⑫ Describe the concept of position-independent code and how it is achieved

- Position independent code can be loaded anywhere in memory and still execute correctly.





⑬ Describe how data can be accessed using PC relative addressing

- PC-relative addressing with appropriate offsets allow memory data to be accessed in a position-independent manner

Chapter 5 Instruction Sets

① Describe how data in register and memory can be efficiently transferred.
- Instructions like MOV and MVN are used with register direct and immediate addressing modes to efficiently transfer register data.
- Instructions like LDR and STR are commonly used together with numerous variants of register indirect addressing modes to efficiently transfer memory data

② Describe how byte-sized data can be accessed in memory.
- Done using LDRB and STRB
* Byte moved into register is zero extended and does not have data alignment restrictions.

③ Describe the operation and the uses of basic arithmetic instructions in the ARM instruction set.
- Basic operations are ADD and SUB, involves only registers or immediate value (for rightmost operand) [total 3 operands]

ADD	(addition)
SUB	(subtraction)
RSB	(reverse subtraction)
ADC	(add with carry)
SBC	(subtract with carry)
RSC	(reverse subtract with carry)

* Subtraction is a non-commutative operation which is done by adding the minuend to the negated subtrahend

```
SUB R2,R0,R1 ; R2=R0−R1
```
(R0: minuend, R1: subtrahend)

④ Describe how the arithmetic operations influence the status of Condition Code flags.
1. ADD (can affect all NZCV flags)

		Signed Number	Unsigned Number
(+ve)	0000 0001	(1)	(1)
(+ve)	+0111 1111	(127)	(127)
(-ve)	1000 0000	(-128)	(128)

N=1, V=1 ← 2's complement oVerflow

		Signed Number	Unsigned Number
(+ve)	0000 0001	(1)	(1)
(-ve)	+1111 1111	(-1)	(255)
(+ve)	0000 0000	(0)	(0)

Z=1, C=1 ← unsigned overflow

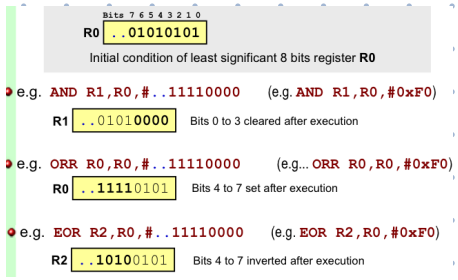
V flag set when adding signed numbers of same signs give a result of opposite signs.
C flag set indicates overflow/carry when adding unsigned numbers.

2. SUB (can affect all NZVC flags)
- C flag clears if the subtraction produces a borrow (occurs when the unsigned value of the minuend < value of subtrahend) and C sets otherwise. [C=0 indicates unsigned underflow]
- V flag sets if result is out of the signed 32 bit range

3. Arithmetic instructions that incorporate the carry flag can be employed for multi-precision arithmetic.

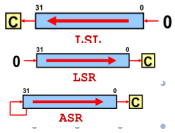
`ADC R2,R0,R1 ; R2=R0+R1+C`	ADD with carry
`SBC R2,R0,R1 ; R2=R0−R1+NOT(C)`	SUB with carry
`RSC R2,R0,R1 ; R2=R1−R0+NOT(C)`	RSB with carry

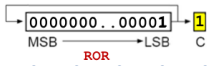
⑤ Describe the operation and the uses of the various logical instructions.
1. MVN: two-operand instruction that does the NOT operation
2. AND: clear specific bits in destination operand.
3. ORR: set specific bits in destination operand
4. EOR: complement specific bits in destination operand.

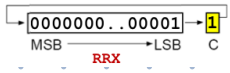


⑥ Describe the operation and uses of the various shift and rotate instructions.

1. LSL: logical shift left: un/signed multiply
2. LSR: logical shift right: Unsigned divide
3. ASR: arithmetic shift right: Signed divide
4. ROR: rotate right: bit shifted out of the register is returned to the MSB and also placed into the C flag.
5. RRX: rotate right extended: C flag is shifted into the register at the MSB and the LSB shifted out replaces the current C flag.







* In an efficiently combined instruction, the shift operation is applied to the rightmost operand and the number of bits to shift is specified as an immediate value or a value within a register.

```
MOV  R0,R0,LSL #1 ;R0=R0<<1
```
Shift R0 left by 1 bit

```
ADD  R2,R1,R0,LSR #2 ;R2=R1+R0>>1
```
ADD R1 with 2-bit right shifted R0. Put result in R2.

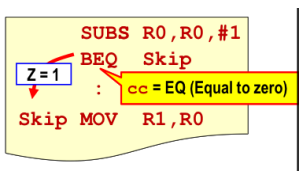
```
ADDS R2,R1,R0,LSL R4 ;R2=R1+R0<<R4
```
Shift R0 by R4 bits before ADD with R1. Update N,Z,V,C flags and result in R2.

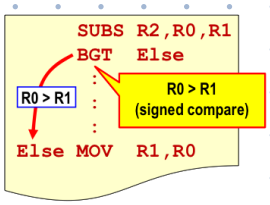
```
ORRS R0,R0,R0,ROR #1 ;R0=R0||R0>>1
```
OR R0 with a 1-bit right rotated version of itself. Set C flag if bit rotated out is 1.

⑦ Describe the various conditional branch instructions and its uses.

- Useful for implementing conditional constructs / loop constructs.
- conditional field (cc) displaces the PC if cc is true (has a displacement range of 4 bytes)
- Bcc is used with address labels that allows the assembler to compute the required displacement values.

Suffix	Flags	Meaning	
EQ	Z = 1	Equal	
NE	Z = 0	Not equal	
CS or HS	C = 1	Higher or same, unsigned	Unsigned comparison
CC or LO	C = 0	Lower, unsigned	Unsigned comparison
MI	N = 1	Negative	
PL	N = 0	Positive or zero	
VS	V = 1	Overflow	
VC	V = 0	No overflow	
HI	C = 1 and Z = 0	Higher, unsigned	Unsigned comparison
LS	C = 0 or Z = 1	Lower or same, unsigned	Unsigned comparison
GE	N = V	Greater than or equal, signed	Signed comparison
LT	N != V	Less than, signed	Signed comparison
GT	Z = 0 and N = V	Greater than, signed	Signed comparison
LE	Z = 1 and N != V	Less than or equal, signed	Signed comparison
AL	Can have any value	Always. This is the default when no suffix is specified.	





⑧ Describe how conditional test can be implemented.

- selected based on the number representation used. (eg. signed values use GT, LT, GE, LE
  unsigned values use HI, LO, HS, LS)

1. CMP: subtracts the source operand from the destination register and sets the CC flags according to the results

```
CMP   R1,R2   ; source operand
      :
BGE   R1≥R2
      :
R1≥R2 :
```
Same flow control but R1 unchanged

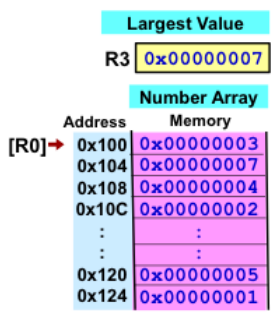
2. CMN: CMN R0, R1 ; sets (NZCV flag) based on R0+R1 ; compare negative
3. TST: TST R0, R1 ; set (NZC flag) based on R0 AND R1 ; Test bits
4. TEQ: TEQ R0, R1 ; set (NZC flag) based on R0 EOR R1 ; Test Equivalence.

} C flag influenced by applying the shift and rotate operations on the source operand.

⑨ Use appropriate data transfer instructions to retrieve memory arrays efficiently.

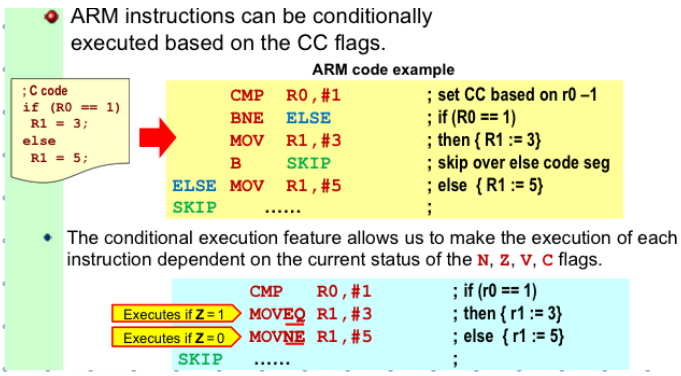
⑩ Use appropriate program control instructions to determine flow program based on desired outcomes.

⑪ Implement a simple findmax algorithm in ARM assembly.




```
        MOV   R0 , #0x100   ; initialise pointer to first element of array.
        MOV   R1 , #9       ; count register
        LDR   R3 , [R0]     ; current max = first element.
Loop    ADD   R0 , R0, #4   ; move pointer to next address
        LDR   R2 , [R0]     ; load next element into R2
        CMP   R2 , R3       ; compare R2 with current max
        BLS   Skip          ; Branch to Skip if less than R3 (ie R2<R3)
        MOV   R3 , R2       ; Else move R2 to be cur max
Skip    SUBS  R1 , R1, #1   ; Decrement counter by 1, Z flag set if 0
        BNE   Loop          ; Branch to Loop if counter ≠ 0.
```

Conditional execution (eg. MOVHI, MOVEQ, ADDEQ) can be used to avoid doing conditional branching.

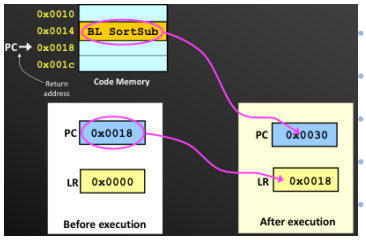


Chapter 6 Modular Programming

① Describe ARM instructions and implement subroutines

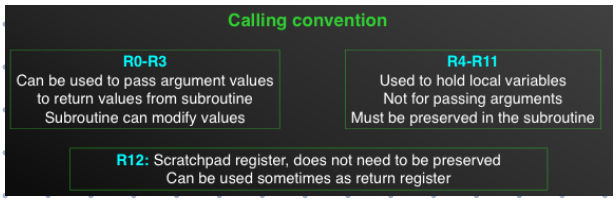
- Subroutines are like functions and can be called from various parts of the program (returns to instruction after subroutine call)
  - To go to subroutine (SUB1): **BL SUB1** (in main)
  - To return to caller program: **BX LR** (in SUB1)

  * Using B overwrites the value in PC, counter by storing address of instruction after subroutine in another register, then moving that value back into PC after SUB1 ends.
- BL (brand with link): Return address (instruction after SUB) stored in linked register (R14)
  Address of subroutine loaded into PC
- BX: BX LR returns from routine; copies the value stored in LR into PC
  *MOV PC LR (VISUAL does not support BX LR)



② Describe passing parameters into subroutines

1. Registers: - parameters placed in registers before calling the subroutine
(!! no. of parameters passed are limited to the available registers)



Pros: Efficient as parameters in the registers can be used immediately

cons: Lacks generality (limited registers)

* Eg: Bit Counting Subroutine (solution 2 checks if LSB is a 1 using masking)

2. Memory: - A region in memory is treated like a mailbox and is used by both the calling program and the subroutine.

↳ Parameters to be passed are grouped together in a chunk of mem.
↳ The start address of the chunk is passed into the subroutine using an address register
↳ Useful for passing large number of parameters.

* Eg: Lower to Uppercase subroutine (using ascii table)

③ List the stack manipulation operations & its implementation.

* Maintained by dedicated stack pointer (SP/R13)

1. Push operation: use STR (update stack pointer before storing)
eg. STR R0, [SP, #-4]! (pre indexing)
Use STMFD (store multiple Full Descending depends on how memory is stored)
eg. STMFD SP!, {list of registers (R1, R0) or (R9 - R6)}

2. Pop operation: Use LDR (update stack pointer after reading)
eg. LDR R0, [SP], #4 (post indexing)
Use LDMFD (load multiple Full Descending depends on how memory is stored)
eg. LDMFD SP!, {list of registers (R1, R0) or (R9 - R6)}

* Data not erased after popping.

* lowest memory address always gets load to / stored from the lowest register

④ Describe passing parameters to subroutines using the stack

- Parameters are pushed onto the stack before calling the subroutine and retrieved from the stack within the subroutine.
* must be removed immediately by the calling program after returning from the subroutine to prevent a stack overflow

Eg. Sum from 1 to N

```
        :                       ;find the sum of (1+2+3+4+5), where N=5
Parameters setup
        MOV   R1, #5            ;Set R0 with #5
        MOV   R0, #0x100        ;Set R1 with the address
        STMFD SP!, {R1,R0}      ;push R0&R1 to stack
        BL    Sum1N             ;call subroutine Sum1N
        ADD   SP,SP,#8          ;add 8 to pop the two parameters from stack
        :
```

Note: 1. Parameters set up before calling the subroutine are **removed** immediately after returning from subroutine.
2. Removal is done by returning the contents of the stack pointer to its **original value** before the parameters were pushed to the stack.

calling program

save original register values

```
Sum1N   STMFD SP!,{R4,R5,R6}    ;save registers to stack
Retrieve stack parameters
        LDR   R5,[SP,#16]       ;Load N from stack to R5
        LDR   R6,[SP,#12]       ;Load Answer's From stack
        MOV   R4,#0             ;clear summation register R0 to 0
Loop    ADD   R4,R4,R5          ;add current value in R5 to R4
        SUBS  R5,R5,#1          ;decrement current value in R4 by 1
Z=0     BNE   Loop              ;jump back to Loop if R4 not yet zero
Z=1     STR   R4,[R6]           ;write sum to Answer
        LDMFD SP!,{R4,R5,R6}    ;restored saved registers
        MOV   PC,LR             ;return from subroutine
```

original values

Note: The subroutine needs three registers. R4 to compute the sum from 1 to N. R6 to be an address pointer to the memory variable **Answer** where the results will written to. R5 holds the value of N, which is decremented by 1 after each loop till it reaches 0.

subroutine

⑤ Identify the difference between passing by value and by reference

- By value: value of data (or variable) is passed to the subroutine
- By reference: address of data (or variable) is passed to the subroutine.

* Passing by reference allows the subroutine to directly access memory variables within the calling program.

⑥ Describe how a transparent subroutine can be implemented.

- Will not affect any CPU resources used by the program calling it
- All local registers used by the subroutine must be saved on the stack on entry and restored before return.

* Temporary memory space created by a subroutine is called the stack frame
  ↳ Accessed using the Frame Pointer (R11) or Stack Pointer (R13)
  * Read slide 68 on modular programming.

Pros of using SP instead of FP: more efficient since there is no need to set up a Frame Pointer
Con of using SP instead of FP: more restrictive as system stack cannot be used within subroutine without changing the reference SP.

* note SP starts from lower address
FP starts from higher address.

⑦ Understand the concept of nested subroutine.

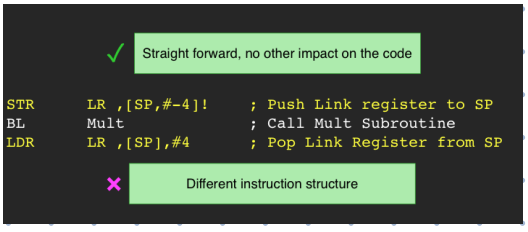
- Subroutines that contain at least 1 more subroutines (key for truly modular designs)

* Need to ensure that the return address is not overwritten.

⑧ Describe how nested subroutines are implemented in ARM

- Old Link Registers need to be stored somewhere safe to prevent overwriting
  ↳ save before any BL instruction / at the beginning of each subroutine

①



② We need to push LR into the stack together with local registers (ie R4-R7)
  ↳ When we LD R4-R7 back in the caller routine, LR gets loaded together back to its original state.
  * offsets need to be updated.

⑨ Describe recursive functions and their implementation in ARM

- A recursive routine calls itself within its own body

* Ensure there is a stopping condition & a stored return register (usually R12)

Chapter 7 Flow Control Constructs

① Be able to convert a high level IF and IF-ELSE construct to its low-level equivalent

- Test condition is reversed to avoid the need for an additional unconditional jump

```
        CMP a,b
        BGT DoIF
        B Skip
DoIF    {S1}
Skip
```

Imitation ← `if (a>b) {S1}` → Reverse

```
        CMP a,b
        BLE Skip
        {S1}
Skip
```

} only applicable for IF conditions.

* Reverse condition of HS is LO, reverse of LT is GE

- IF-ELSE
  - ↳ Combination of conditional and unconditional jumps are used for the IF-ELSE construct.
  - * Reversing only improve efficiency if the ELSE suite is more likely to execute

② Describe compute-efficient considerations for compound AND and OR constructs.

(eg. `if ((a==b) && (b>0)) {S1}`)

- compilers resolve compound conditions into simpler ones.
  - ↳ first false condition will lead to remaining conditions not computed (for AND)
  - * keep the least likely condition leftmost for more efficient execution.

} For AND

(eg. `if ((a==1) || (a==2)) {S1}`)

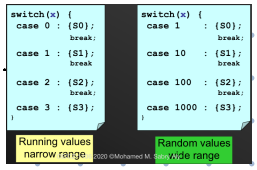
- compilers eliminate unconditional jump at the end of the compound OR series by reversing the last condition test.
  - * keep the most likely condition leftmost

③ Describe and analyze simple branchless logic constructs

- Avoids using conditional jump instructions when implementing logical constructs.
- Exploits arithmetic relationships to transform the test condition into the corresponding desired outcome (usually Boolean values)
- Helps keep the CPU pipeline efficient by maintaining sequential execution.

④ Describe how SWITCH constructs can be implemented efficiently for consecutive narrow and random wide cases.

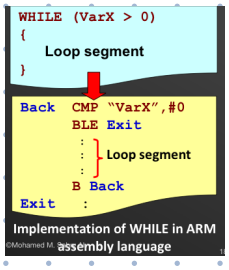
* assembly depends on compilers but more on nature + range of the case values.



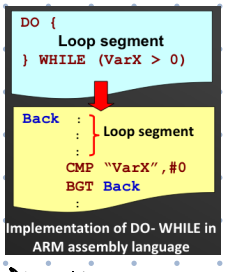
1. Narrow range: Jump Table is used to avoid testing each case on return.
   - ↳ contains a list of start addresses for the suite that is associated with each case value
   - ↳ The switch acts as an offset into the table to access the start addresses which are then loaded into the PC for execution.
2. Random and Wide: Fork Algorithm is used to speed up average search time & avoid testing every case
   - ↳ use an if-else-if implementation instead of a Jump table.

⑤ Contrast the implementations of pre and post test loop constructs like WHILE, DO-WHILE and FOR

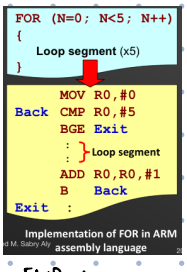
- Pre-test (check before executing) may never execute the suite, post-test executes suite at least once.



WHILE Loop
Pre-test



DO-WHILE Loop
Post-test



FOR Loop
Pre-test

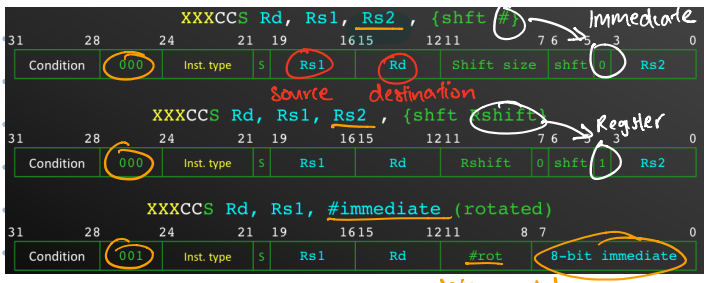
* Post-test loops are more efficient than Pre-test loops for the same suite

Chapter 8 Instruction Encoding and ISA

① Describe the instruction format of ARM instruction-set architecture (ISA)

- Each unique instruction is encoded with unique binary patterns in a 32-bit word.

- Instruction type → encoded
- Operand(s) → encoded
- Condition for conditional execution → encoded
- Other flags?



Code	Condition	Flags	Meaning
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS or HS	C = 1	Higher or same, unsigned
0011	CC or LO	C = 0	Lower, unsigned
0100	MI	N = 1	Negative
0101	PL	N = 0	Positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 and Z = 0	Higher, unsigned
1001	LS	C = 0 or Z = 1	Lower or same, unsigned
1010	GE	N = V	Greater than or equal, signed
1011	LT	N != V	Less than, signed
1100	GT	Z = 0 and N = V	Greater than, signed
1101	LE	Z = 1 and N != V	Less than or equal, signed
1110	AL		Always.
1111	NV		Reserved (unused)

Condition (bits 31-28)

Code	Instruction
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC
1000	TST *
1001	TEQ *
1010	CMP *
1011	CMN *
1100	ORR
1101	MOV
1110	BIC
1111	MVN

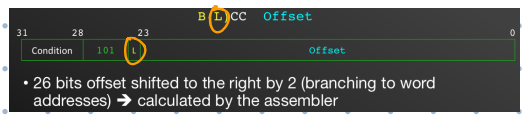
* sets S flag ∴ bit 20 = 1

Instruction (bits 24-21)

Code	Shift type
00	LSL
01	LSR
10	ASR
11	ROR/RRX

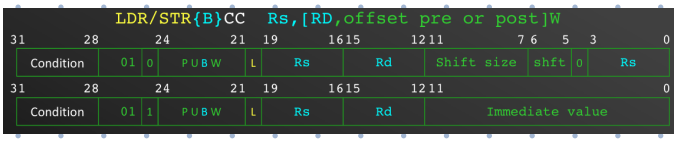
Shift type (bits 6-5)

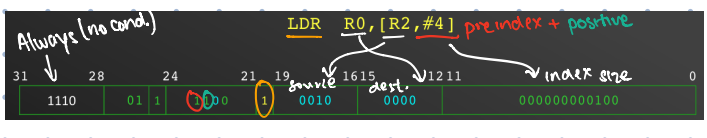
- For conditional branch (BCC) or branch with link (BL)

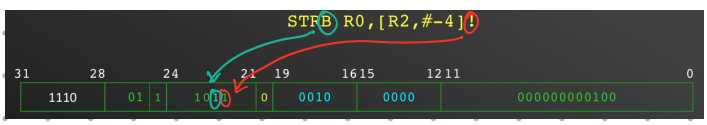


- LD / STR instructions (1 word)

- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

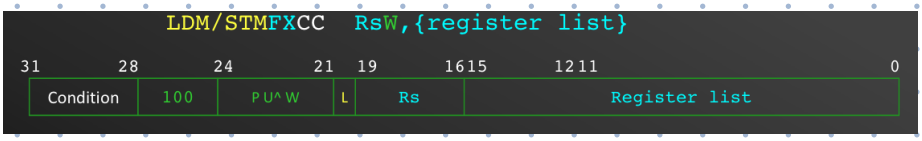






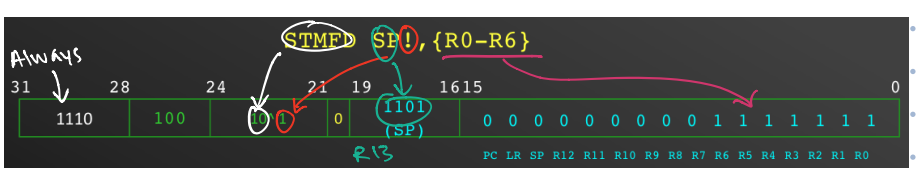
- LD / STR Multiple

- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is "don't care"
- W is for write back (!) after operation



PU	Instruction
10	STMFD
01	LDMFD
00	STMFA
11	LDMFA

bits 24, 23



② Describe the difference between fixed and variable length ISA

- ARM is fixed-length
  - ↳ Operands have to be in registers or immediate values.

Examples of variable length:

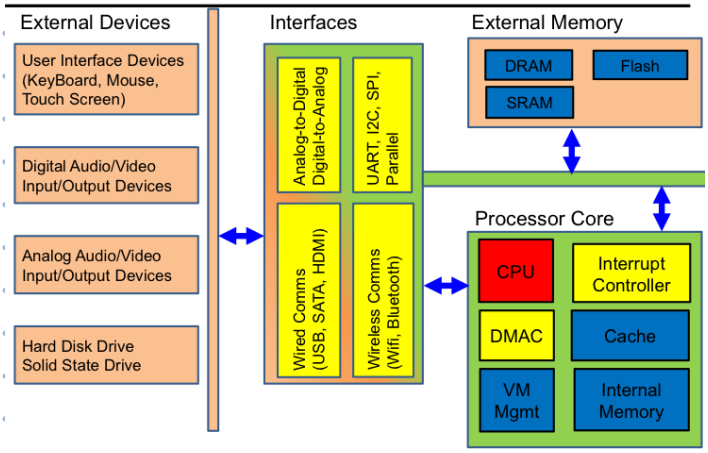
- Intel 80x86: Instructions vary from 1 to 17 bytes long
- Digital VAX: Instructions vary from 1 to 56 bytes long

* Variable length instructions require multistep fetch and decode; allows for a more flexible (but complex) and complex instruction set.

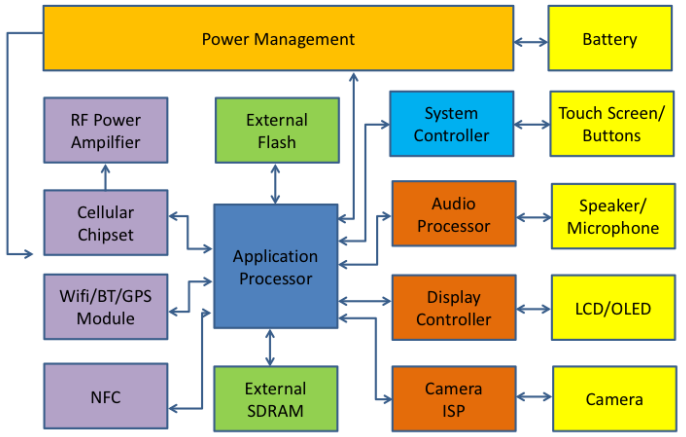
Part 2 ↓

Chapter 1. Computer Systems overview

Computer System Block Diagram



Example: Smart Phone



Computer System

① Processor Core

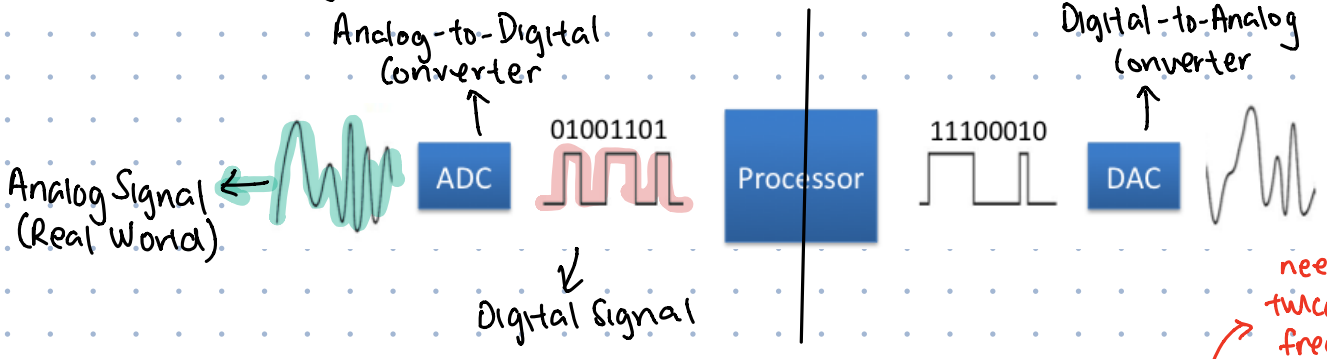
② Signal Chain Subsystem
- ADC - DAC
- Parallel & Serial Digital Interfaces
- Direct Memory Access Controller
- Interrupt Controller

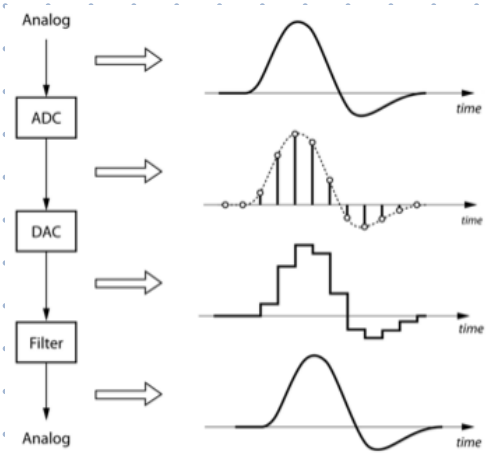
③ Memory Subsystem
- Semiconductor memory
- Flash memory
- Magnetic HDD
- Cache memory Management
- Virtual Memory Management

④ Comms & U/I
- Wired (USB/SATA)
- Wireless (Wifi/BT)
- Input devices
- Output devices

Chapter 2 Signal Chain Subsystem

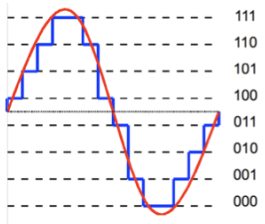
① Interfacing to the Real World





* Digital Signal is obtained by sampling the analog signal level at discrete time (Sampling Interval) — needs to be at least twice the signal frequency

* Analog signal can be reconstructed back by applying a filter on the digital signal

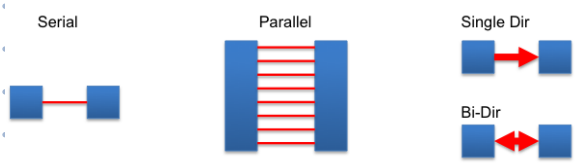


- Figure on the left shows a typical digital quantization process of a analog signal.
- A 3-bit system is used in this example. So all signal will be mapped to one of the 8 possible representations (000 to 111).
- At each sampling point, the analog signal level is approximated to the nearest digital equivalent.

② Electrical Signal Interface

- Single bit Data Transfer (Serial)
- Multi bit Data Transfer (Parallel)

} Single / bi-directional



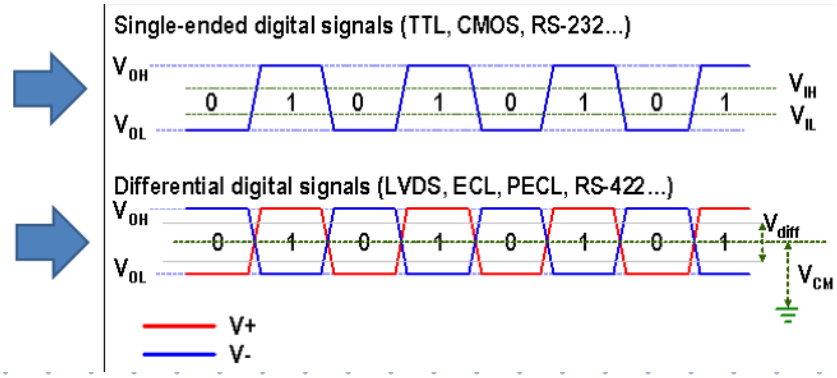
* Requires compatibility in ① Electrical Signal level ② Communication Protocol

2.1 Electrical Signal level (Safety)

↳ Output voltage of output device cannot exceed maximum allowable input voltage level of input device → leads to reduced reliability / spoiling

- VOH.
  - Transmitting a Logic '1' yield a signal level of >= VOH
- VOL.
  - Transmitting a Logic '0' yield a signal level of =< VOL
- VIH.
  - A received signal with level >= VIH will be recognized as a Logic '1'
- VIL.
  - A received signal with level =< VIL will be recognized as a Logic '0'

↳ Differential Signals has better noise tolerance to enable higher frequency clocking



2.2 Communication Protocols

↳ Refers to how data are formatted during transmission

- Some examples
  - Number of bits in a transmission frame
  - What synchronization to use
  - Data width
  - Types of data and its formatting
- Examples of communication protocols are USB, UART, SPI etc.

Protocol 1	1001001001001000
Protocol 2	1001001001001000

Blue: Synchronization Bits
Green: Data Bits
Red: Error Correction Bits

2.3 Parallel Data Transfer (Prone to Signal Skew / Cross Talk)

↳ Multiple bits transferred simultaneously
↳ Synchronous (using clock)
↳ Higher transfer rate compared to serial transfer

2.3.1 Signal Skew

↳ Occurs when signal in one or more data lines took different amount of time to reach receiver (Variation in propagation delay)
↳ Due to circuit design / external interference. (Capacitance + Resistance)

2.3.2 Cross Talk

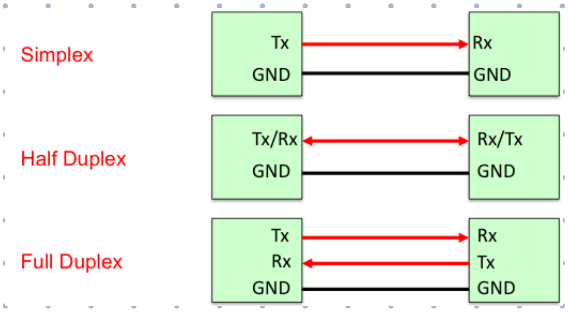
↳ Undesired coupling of signals from another circuit
↳ Due to close placement of datalines in PCB routing.
↳ Transmitted electrically / E.M. radiation

2.3.3 Advantages & Disadvantages

↳ Advantages
  ↳ Fast Data Transfer
  ↳ Simpler hardware interface
↳ Disadvantages
  ↳ Signal Skew + Cross Talk
  ↳ Bulky hardware if data width is large
  ↳ Requires more space (PCB routing)
  ↳ More expensive.

2.4 Serial Data Transfer

↳ Data transfered one bit at a time. (single dataline)
↳ Less affected by signal skew & crosstalk ⇒ Able to support higher freq. clocking
↳ Slower data transfer.
↳ 3 modes: Simplex, half duplex, full duplex



- Data transfer in 1 direction only
- Data transfer in both directions but only one can happen at a time
- Data transfer in both directions can happen simultaneously.

↳ Advantages & Disadvantages
  ↳ Advantages
    ↳ less affected by signal skew & crosstalk.
    ↳ Able to transfer data reliably over longer distances.
    ↳ lower cost
  ↳ Disadvantages
    ↳ Slower data transfer
    ↳ More complex hardware interface. (need to convert serial to parallel)

Processors only process in bytes or multiple bytes

* In a synchronous transmission, there is a common clock signal to synchronise transfer. In an asynchronous transmission, devices have to agree on a pre-fixed clock frequency.

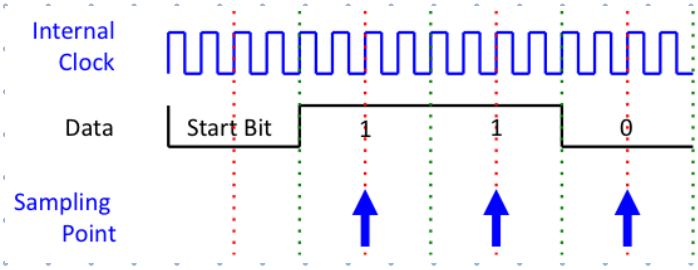
2.5 Synchronous Serial Transfer (uses a common clock)
↳ Master-Slave configuration with Master providing clock signal
↳ Potentially faster transfer rate (no data overhead needed to synchronize transfer)

2.4 Asynchronous Serial Transfer (no common clock)
↳ Receiver needs to know the transmitting clock rate & the number of bits to be transmitted with each data packet.
↳ Receiver uses its own clock to track timing after receiving SYNC word (start/stop)
↳ Potential skew issue between Transmitter clock & Receiver clock.

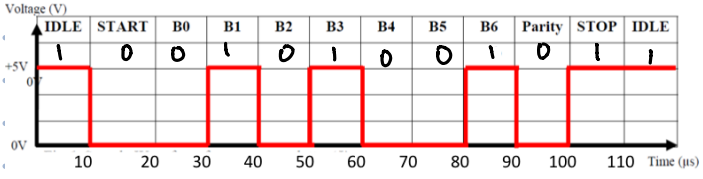
③ UART Transmit
- Dataline is logic '1' in idle state
- Transmitter sends a '0' to alert receiver of start bit. (Followed by DATA bits)
- Transmitting clock rate is also known as baud rate [no. of bits transmitted per second]
- Parity bit may also be sent for receiver to check data packet integrity (set '1' if wrong parity)
- STOP '0' terminates transmission.

* Parity error + Framing Error (occurs when STOP bit is '1')

④ UART Receive
- Waits for Start bit '0' (falling edge)
- Needs to know baud rate to sample data bits correctly
  ↳ Internal clock usually runs at a multiple of the baud rate to time the middle



Tx baud rate = 1/(10 us) = 100000 bps. Configuration = 701. 7 Databits, Odd Parity, 1 STOP



- RX Data = 1001010b
- RX Data = 0011000b ?
- **Information Sampled @ 200000 bps:** 00001100110000110011

Sampling @ Tx baud rate = 0 | 0 1 0 1 0 0 1 | 0 1 => 1001010 = 0x4A
(Data) (reversed)

Sampling @ 2xTx baud rate = 00 00 11 00 | 11 | 00 00 11 00 | 11
S DATA PTS DATA PT
(each bit sampled twice)

* but still read as 701

Chapter 3 Direct Memory Access

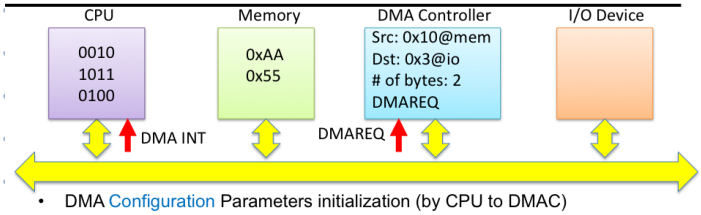
① Direct Memory Access Controller

↳ Allows efficient movement of data when complex address manipulation is required

- If there are no conflict in hardware resources used, it is possible for data transfer via DMA and CPU execution to occur simultaneously.
- If there is a conflict in hardware resources used, e.g. both DMAC and the CPU needs the system bus, then access will be given to the one with higher priority.
- Who has the higher priority and whether the priority is configurable depends on processor design and is thus processor specific.

↳ Data bus controller module that performs data transfer independent of the CPU
↳ Generates addresses & read / write operations.

Basic DMA Process



- DMA Configuration Parameters initialization (by CPU to DMAC)
  - Source Address
  - Destination Address
  - Amount of data to transfer
  - DMA Trigger signal (DMAREQ, interrupt, software bit etc)
- CPU proceed with its own task while DMAC wait for DMA Trigger Signal.
- Once DMA Trigger occurs, DMAC request to take over system bus.
- DMAC transfer data according to configuration.
- DMAC notify CPU of data completion (typically via interrupts) and release system bus.

② DMA mode of Operation

1. Burst Mode (HDD file transfers)
   ↳ Transfer multiple bits of data before returning control (could be whole chunk / subset of chunk)
   ↳ CPU might be suspended till DMAC has completed its data transfer
   * Fast transfer rate but render CPU inactive for longer period of time.

2. Cycle Stealing (Real-time applications)
   ↳ DMAC releases databus after transfering 1unit of data (Executed between CPU instructions / Pipeline stages)
   ↳ CPU may still be suspended (like in Burst) but for a shorter duration
   ↳ Favoured for applications which requires CPU to be responsive.

Between CPU Instr

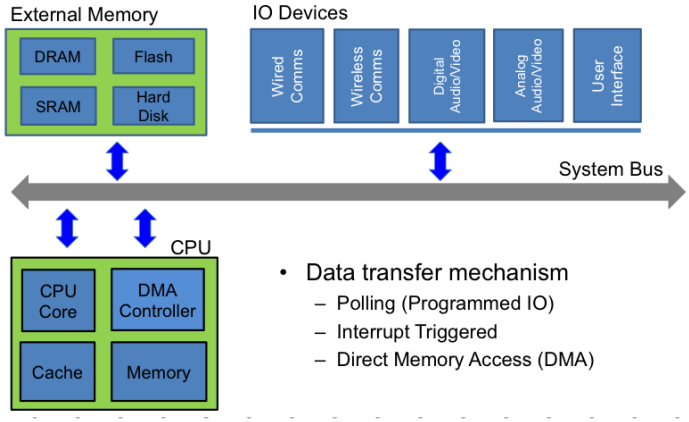
CPU	Instr1	Instr2		Instr3		Instr4
DMAC			Data		Data	

Between Pipeline Stages

CPU	Fetch Instr	Decode Instr		Fetch Operand	Exe Instr	
DMAC			Data			Data

3. Transparent
   ↳ DMAC transfer data only when CPU is not using the data bus
   ↳ Zero impact to CPU performance
   ↳ More complex hardware needed to detect when CPU is not using databus.

③ Data Transfer mechanism



- Data transfer mechanism
  - Polling (Programmed IO)
  - Interrupt Triggered
  - Direct Memory Access (DMA)

④ Polling Technique

→ CPU continues to wait in loop to check for readiness

- CPU polls a certain I/O port continuously for data / readiness of the port to perform a data transaction
- CPU has full control and dedicates 100% of its resource in the whole data transfer process & does nothing else.

- Advantages
  - Programmer has complete control over entire process
  - Easiest Process to Test & Debug.
- Disadvantages
  - CPU waits in loop ⇒ cannot perform any other task until data transfer is complete.
  - Next program execution held up while CPU waits
  * Inefficient use of CPU's resources.

⑤ Interrupts

- A signaling mechanism that allows peripherals to alert the CPU that attention is needed. (via electric pulse / change in internal register status)
- CPU receives signal (interrupt request) & decides whether to service the request.

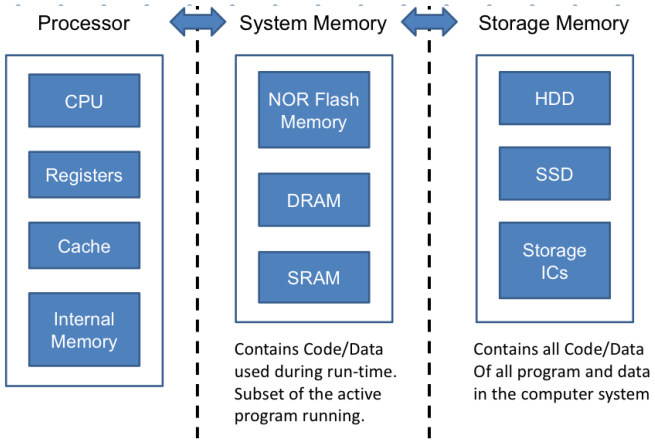
* Typically used to start some operation (eg data transfer to memory, status reg. control reg. etc)

- Interrupt Vector Table
  - ↳ Starting address of each interrupt is stored in the IVT.
  - ↳ Each interrupt has a unique index to the IVT.
- Control Flow
  1. CPU receives interrupt request
  2. CPU decides to service interrupt & suspend current program temporarily
  3. CPU looks up IVT to check starting address of Interrupt Service Routine
  4. CPU saves the current state (processor context) to allow return after service
  5. CPU executes the ISR linked to the interrupt.
  6. CPU restores state saved in step 4 and continues from where it left off.
- Interrupt Service Routine
  - ↳ Usually very short to not suspend main program for too long
  - ↳ Possible for CPU to receive multiple interrupt requests simultaneously
    - ↳ Arbitration scheme has to be designed to decide which interrupt to service first.

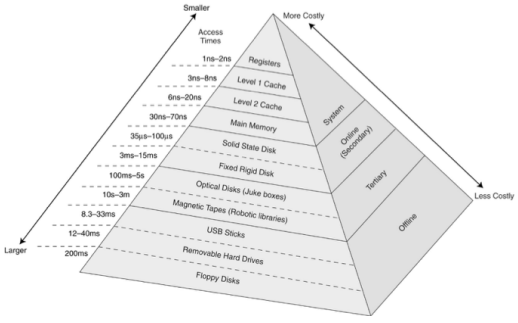
- Advantages
  - ↳ Efficient use of CPU resources (does not need to monitor I/O status)
    (CPU can continue with other tasks between interrupts)
  - ↳ Allows prioritisation & pre-emption

- Disadvantages
  - ↳ More hardware interface circuitry required
  - ↳ Program slightly more complex & difficult to debug.

Chapter 4 Computer Memory



- The memory and storage devices may be organized like a pyramid.
- The pinnacle has the fastest access time, but is also more costly.
- Memory Access Time below are only for illustration. These changes with improvement in technology.



① Volatile and non-volatile memory
- Volatile: Data lost when electric power is removed.
  Temporary Storage
  Used as system memory
  (eg RAM ; SRAM & DRAM)
- Non-volatile: Data is retained even if electric power is removed
  Permanent Storage
  Used as main storage
  (eg Flash, magnetic hard-disk)

② Semiconductor memory
- based on semi-conductor integrated circuits
- Used as processor internal memories & system memory

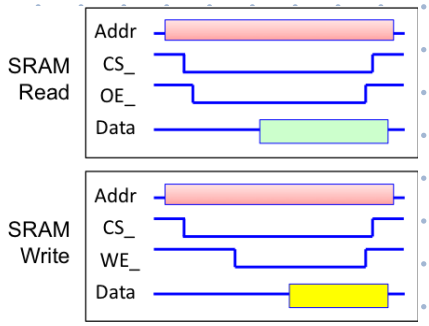
- Processor internal memories
  - Registers
  - Buffers
  - Cache
  - Internal System and Storage Memory (SRAM, DRAM, Flash based)
- Types of memory
  - SRAM
  - DRAM
  - Flash Memory
  - Solid State Drives (based on Flash Memories)

④ Random access memory

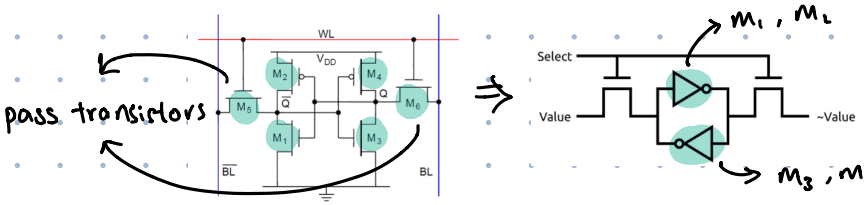
- Static RAM (SRAM)
  - Static Random Access Memory
  - Data stored as long as supply is applied.
  - Large (4 to 6 transistors per cell).
  - Fast.
  - Low power consumption (active and standby)
- Dynamic Random Access Memory (DRAM)
  - Periodic refresh required.
  - Small (1 to 3 transistors per cell).
  - Slower.
  - Higher power consumption due to need for periodic refresh operation to maintain data integrity of memory cells.

⑤ SRAM

- Access
  - ↳ Addr: Specifies address of memory location
  - ↳ Data: Data to be read/written (8/16/32 bits)
  - ↳ CS: Chip Select (Enables / Disables the chip)
  - ↳ WE: Write Enable (Allows data to be written)
  - ↳ OE: Output Enable (Allows SRAM to output data)



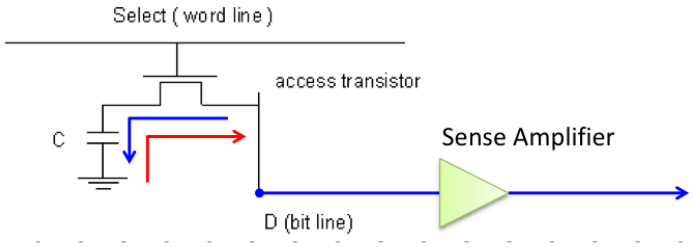
- Cell
  - ↳ Accessed via wordlines & bitlines
  - ↳ Each cell consists of 6 Transistors



  - ↳ WL derived from the Address Decoder Output (controls read/write process)
  - ↳ Actual data placed on different bitlines (connected to data bus)

⑥ DRAM

- Single transistor
- Uses a capacitor as storage element
- Transistor used to control charges flowing in/out of capacitor during read & write process
- Write Process
  - ↳ Store '1' → Enable transistor, transfer charge into capacitor
  - ↳ Store '0' → Enable transistor, discharge capacitor
- Read Process
  - ↳ Enable transistor, measure charge using sense amplifier.

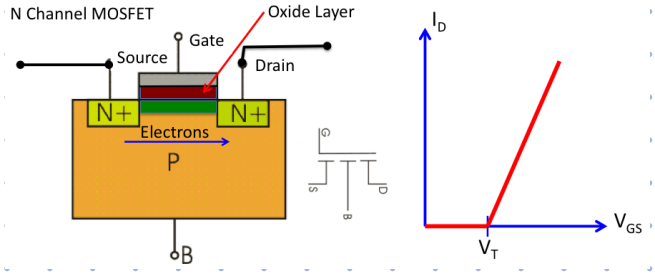


- Maintaining Data Integrity
  - ↳ Read process destroys the information stored on capacitor
    - ∴ Original Data has to be rewritten after every read.
  - ↳ Periodic Refresh needed as stored charges 'leaks' over time.
  - ↳ Succeeded by SDRAM
    - ↳ makes use of a synchronous clock signal from host to synchronize data transfer
    - ↳ Has a pipeline architecture that allows fast & overlapping operations.

(7) EPROM & EEPROM

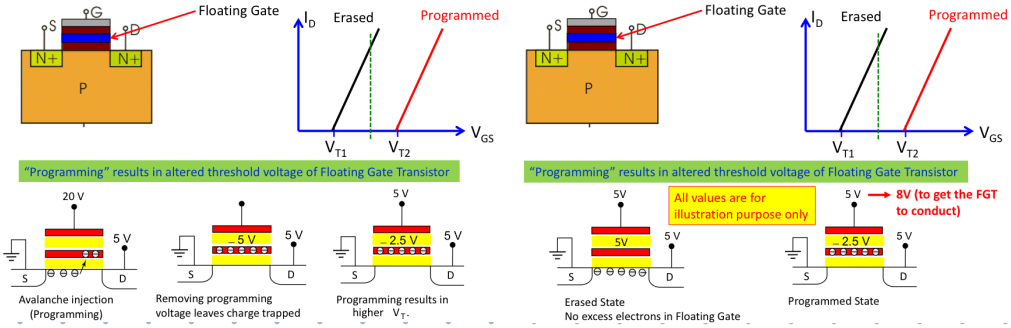
- Erasable Programmable ROM (EPROM)
  - ↳ uses UV light to erase stored program
- Electrically EPROM
  - ↳ can electrically program / erase device

(8) MOSFET (Metal Oxide Semi-conductor Field Effect Transistor)

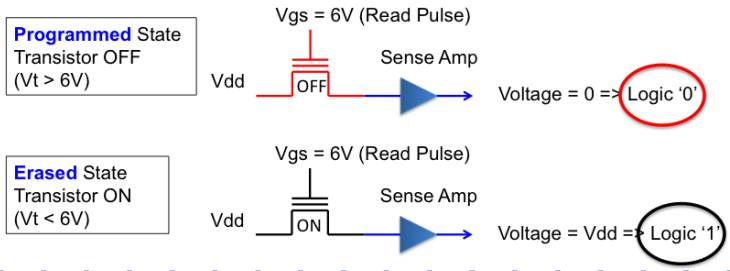


- If gate-source voltage (V_{GS}) > Threshold voltage (V_t), a conductive channel of electrons (Inversion Layer) will be formed and current will flow if a positive voltage is applied across Drain and Source

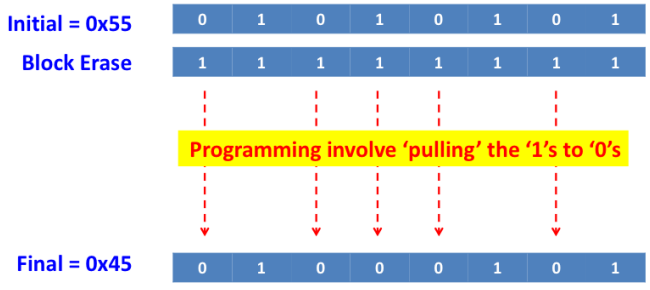
(9) Floating Gate Transistor (FGT)



- With a positive Gate-Source voltage (V_{GS}) = 6V (between V_{T1} & V_{T2}) (5V, 8V)
  - ↳ Transistor remains OFF if programmed (contains charges; higher V_{T2}) 6V < 8V
  - ↳ Transistor turns ON if erased (contains no charges; lower V_{T1}) 6V > 5V



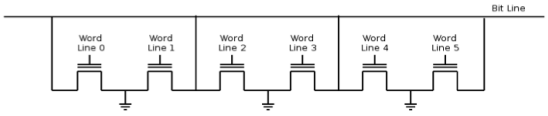
- Programming a FGT memory
  - ↳ can only change content from '1' to '0' (ie put charges into the FGT)
  - ↳ Set all cells to '1' to erase data before programming (done at block level)
  - ↳ Pull desired '1' to '0' to write desired data to FGT.



(10) Flash

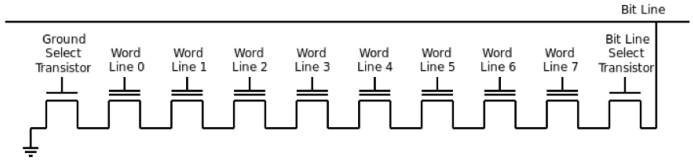
- Based on similar floating gate technology as EEPROM
- Erased in larger block size / page
- Erase cycle slower than EEPROM but faster write speed
- Costs less than EEPROM
- Used in systems requiring large amount of non-volatile memory.

- **NOR Flash**



↳ When **any one** of the word line > V_T (prog), bitline output = 0
↳ Supports Execute in Place (XIP)
  ↳ Allows random reading of memory data using **only Address information.**
↳ Needs special commands in order to perform operations other than Data Read
↳ Allows random word / byte programming but erasure is done at block level.
↳ Used as system memory to store program code or general storage memory.

- **NAND Flash**



↳ Bitline output '0' only when **all word line > V_T (prog)**
↳ Does not support XIP
  ↳ Data accessed one page at a time
  ↳ Command issued to open a particular page, followed by which bytes are needed
  ↳ Uses a single bus to carry Address & Data
↳ Lower cost than NOR gate
↳ Used mainly as storage memory

(11) System & Storage memory

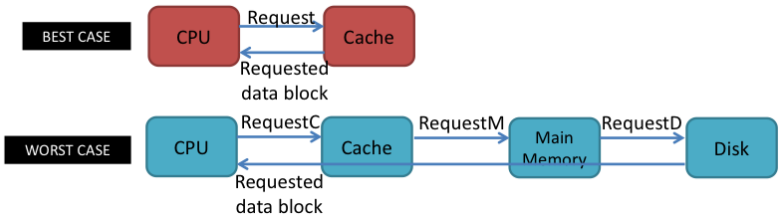
- **System Memory**
↳ **Used to store runtime program**, code is executed directly from system memory
↳ SRAM / DRAM ; NOR Flash + External memory that supports XIP.

  – DRAM technically speaking does not support XIP by itself but processors that has a DRAM interface will have a DRAM controller that handle the necessary translation to allow execution of code directly from the DRAM.

- **Storage Memory**
↳ Used to **store all program and data** in a computer system
↳ Cannot run code directly (does not support XIP)
↳ All memory types can be used as storage memory.

Chapter 5 Cache

- Issue: **CPU** speed is typically much **faster** than **external memory**.
  - CPU speed in Ghz region
  - External Memory Speed in 100s of Mhz region.
- **Need a fast memory** to act as a **buffer** between the Main memory and CPU.
- The purpose of cache memory is to speed up accesses by storing/fetching **recently used data** closer to the CPU instead of the main memory (slower access).
- Potentially able to **improve the overall system performance** drastically



- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
- If the data is not in cache, a query is then sent to the main memory.
- If the data is not in main memory, then the request goes to disk.
- Once the data is located, the **required data and a number of its nearby data elements** are fetched into cache memory simultaneously.

① Principle of Locality

- Once a byte in a program is accessed, it is likely that a nearby data segment will be needed soon.
- Locality of Space
  - ↳ Code/Data nearby to each other is likely to be accessed together
  - ↳ Transfer of data between Sys.mem & cache is done in blocks to leverage this behaviour.
- Locality of Time
  - ↳ Recently accessed code/data is likely to be accessed again
  - ↳ Used to decide which item to replace in cache when space runs out.

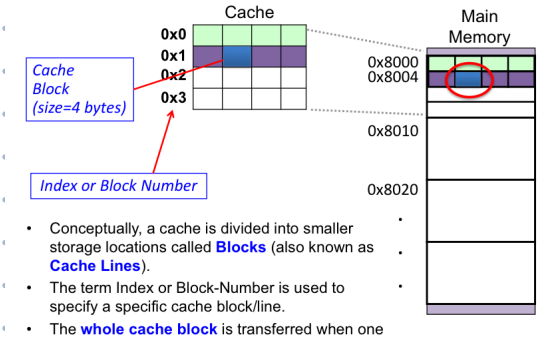
② Cache Replacement Policy

1. Least Recently Used
   - ↳ Evicts the block that has been unused for the longest period of time
   - ↳ Disadvantage is LRU algorithm has to maintain access history for each block which will slow down the cache
2. FIFO
   - ↳ Block that has been in the cache the longest will be replaced.

③ Cache mapping scheme

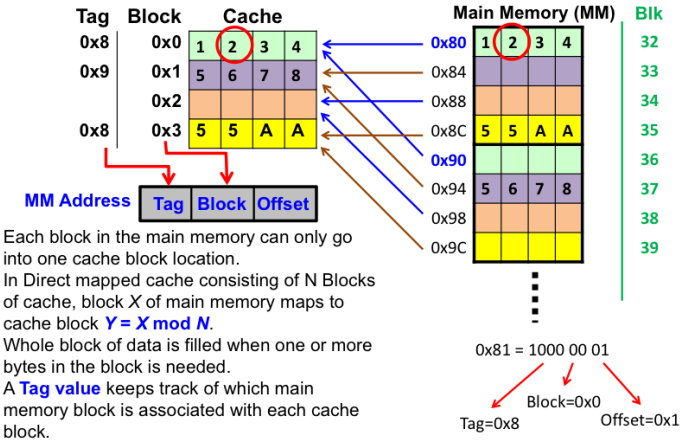
- Deals with how each main memory block is mapped to a particular cache block
- Eg. Direct Mapping, Set Associative, Fully Associative

Terms used in Cache Mapping



- Conceptually, a cache is divided into smaller storage locations called **Blocks** (also known as **Cache Lines**).
- The term Index or Block-Number is used to specify a specific cache block/line.
- The **whole cache block** is transferred when one or more bytes in the block is needed.

④ Direct Mapped Cache



- Each block in the main memory can only go into one cache block location.
- In Direct mapped cache consisting of N Blocks of cache, block *X* of main memory maps to cache block ***Y = X mod N***.
- Whole block of data is filled when one or more bytes in the block is needed.
- A **Tag value** keeps track of which main memory block is associated with each cache block.

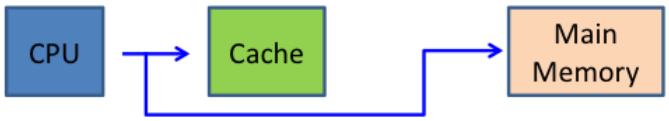
- Given a system with following attribute
  - **Main/System Memory size = 64KBytes**
  - **Cache Size = 256Bytes**
  - **Cache Block Size = 16Bytes**
- Where would Data at main memory address **0x1106** be mapped to?
- Derivation of cache mapping format
  - Cache Block Size = 16Bytes => #**Offset** bits = log2(16) = **4 bits**.
  - Number of Cache Blocks = 256/16 = 16 Blocks => #**Blk** bits = log2(16) = **4 bits**.
  - Main memory size = 64KBytes => log2(64*1024) = 16bits address. #**Tag** bits = 16-4-4 = **8 bits**
  - Mapping Format = 8:4:4
- Applying to the main memory address 0x1106
  - 0x1106 = **0001 0001 0000 0110b**
  - **BLK = 0x0**, **OFFSET=0x6**, **TAG=0x11**

Offset: Targetted data's location offset from start of cache block(row)
↳ if size of cache block is 16, 4 bits are needed in offset field
Block: Index of cache block targetted data is mapped to
↳ Determined by number of cache blocks, if 8 cache blocks, 3 bits are required
Tag: Leftover bits after partitioning for Offset & Block

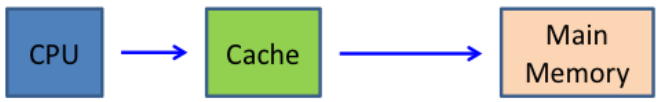
⑤ Cache Write Policy

- Locations in cache written to by CPU known as Dirty Block
- Cache replacement policies must take into account Dirty Blocks
- Must be written back to memory (write policy)

↳ Write Through
- Updates the cache and main memory simultaneously

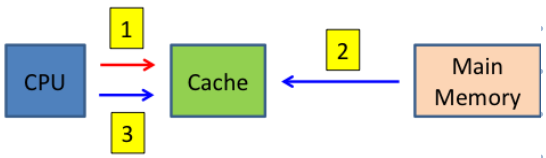


↳ Write Back
- Updates memory only when cache block is selected for replacement

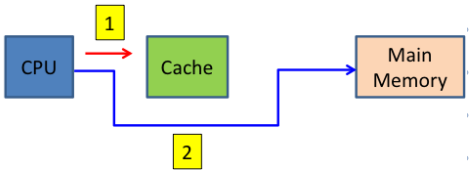


- Write Miss

↳ Write Allocate
- Data block at write miss location will be loaded to cache from main mem.



↳ Write-no-allocate
- Data written directly to main mem. by CPU during write miss



⑥ Cache Performance

- Cache Hit
  - ↳ Data Found in Cache
- Cache Miss
  - ↳ Data not found in Cache
- Cache Hit Rate
  - ↳ % of time data found in cache
- Cache Miss Rate
  - ↳ % of time data not found in cache
- Miss rate = 1 - Hit Rate

⑦ Effective Access Time

- Weighted average that takes into account hit ratio & relative access times of successive levels of memory.

$$EAT = H \times Access_c + (1-H) \times \text{miss penalty}$$

- $Access_c$ = Cache Access time
- Miss penalty = Time to access data when cache miss happens
  - ↳ Assumed data access to cache & main mem do not overlap,

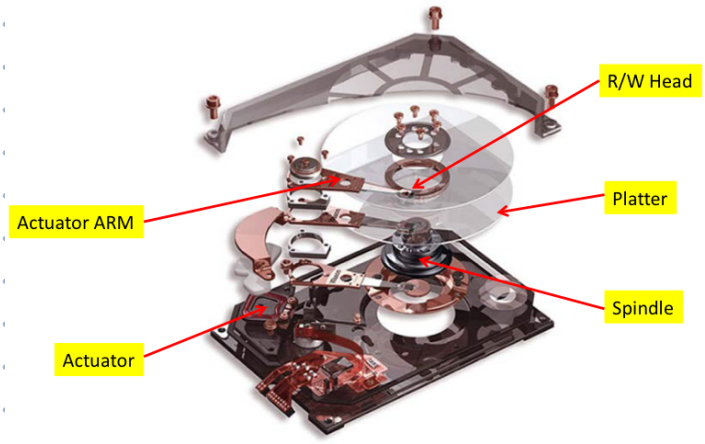
$$EAT = H \times Access_c + (1-H) \times (Access_c + Access_{mm})$$

Effective Access Time (Example)

Sequential Access of Cache and Main Memory

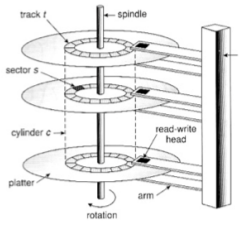
- Consider a system with a **main memory access time of 200ns** supported by a **cache** having a **10ns access time** and a **hit rate of 99%.**
- If the **accesses do not overlap**,
  The EAT is:
  0.99(10ns) + 0.01(10ns + 200ns) = 9.9ns + 2.1ns = 12ns.
- This equation for determining the effective access time can be extended to any number of memory levels.

Chapter 6 HDD & SSD



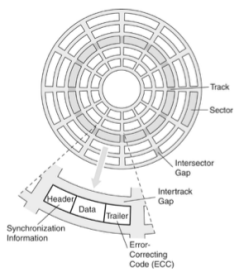
- longitudinal vs Perpendicular recording
- Stores data by magnetizing a thin film of ferromagnetic media on the platter
- Magnetic head uses wind pressure generated by the high speed rotation of the disk to fly above platter surface.

Magnetic HDD Data Organization



- Computers often use magnetic hard disks for large secondary storage devices.
  - One or more platters on a common spindle.
  - Platters are covered with thin magnetic film.
  - Platters rotate on spindle at constant rate.

- Data is stored on the surface of the platter in concentric rings called tracks.
  - Gaps between tracks to minimize interferences from adjacent tracks.
- Tracks are divided into sectors.
  - Minimum data block size is a sector.
- Stored data consists header, data and trailer.



① HDD Transfer Rate

- **Seek Time (T_S)**
  ↳ Time taken for head to move to correct track
- **Rotational Delay (T_R)**
  ↳ Time taken for platter to rotate until the read/write head reaches the starting position of the target sector
- **Access Time (T_A)**
  ↳ Time from request to the time head is in position ($T_S + T_R$)
- **Transfer Time (T_T)**
  ↳ Time required to transfer the required data after head is in position.

- T_R is dependent on rotational speed (RPM) of the disk.

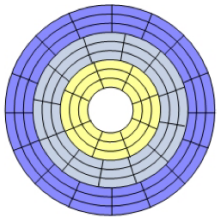
Average T_R, $T_{R,AV} = \frac{0.5}{RPS}$ s

- T_T is dependent on the rotational speed, Track Density D_T, Sector Density D_S and number of bytes for transfer N

$$T_T = \frac{N}{RPS \cdot D_T \cdot D_S}$$

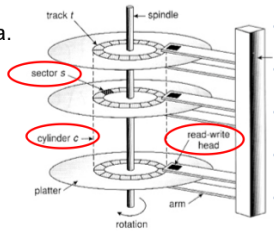
Physical Layout

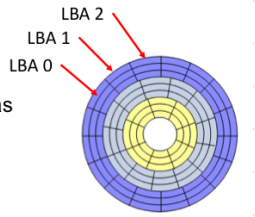
- Early HDD
  - Physical Layout refers to the actual layout design on the HDD.
  - Equal number of sectors per track and each sector has the same data size.
  - Same-numbered Tracks from different surfaces formed a cylinder.
  - The early hard disks were implemented using this topology to simplify the controller design.
- Modern HDD
  - Having equal number of sectors per track means that sectors at the outer tracks are wider.
  - Waste physical space as bit density of those sectors are not optimal.
  - Zone bit recording technique addresses space wastage.
  - Tracks are divided into zones, with differing number of sectors per track for different zones.



Logical Layout

- How the software see and address the HDD data.
- Address translations are needed to map the physical to logical locations. Two addressing scheme are CHS and LBA.
- Cylinder-Head-Sector (CHS)
  - Legacy scheme using the old HDD physical structure.
  - Made Obsolete in recent standards due to addressing limitation and more complex formatting.
- Logical Block Addressing (LBA)
  - Simple linear addressing starting from LBA=0 as first block, LBA=1 as second block and so on.
  - 48-bit LBA standard allows addressing up to 128PByte. 1PByte=2^50 Bytes.





- Speed up data transfer using cache and buffers

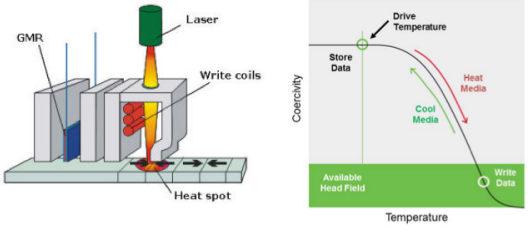
② Limitations of HDD
- Actuator arm is fixed and access has to be sequential, once data is missed, it has to wait for one disk revolution
- Seeking from track to track takes time (R/W head needs to align to starting sector)
- More efficient to read in blocks
- Vulnerable to motion

③ SSD
- memory array based on NAND & NOR Flash
- limited program erase cycles ⇒ same for SSD
  ↳ Techniques used to extend life of SSD
  1. Wear levelling: distributing erase/write evenly over entire drive
  2. External RAM as buffer to minimize number of writes to Flash in SSD
  3. Error Correction Code: Ability to recover from one or more bits of error in media.

④ HDD vs SSD
- HDD
  ↳ Pros: 1. Lower cost per bit
  2. Almost Infinite Erasure cycles
  ↳ Cons: 1. Consists of moving mechanical parts ⇒ more prone to crashing when dropped
  2. Heavier + Larger physical profile
  3. Slower transfer rates.
- SSD
  ↳ Pros: 1. No moving parts, more resistant to movement
  2. Lighter & occupy less space
  3. Faster transfer rate
  ↳ Cons: 1. More costly compared to HDD
  2. Finite number of erasure cycles.

⑤ Data center Storage Criteria
- Power consumption
- Speed
- Robustness
  ↳ Use HDD with shock absorbant housing
- Heat production
- Size
- Cost
  ↳ Reason why HDD is preferred.

HMAR (Heat Assisted Magnetic Recording) HMAR



- The areal density of the HDD was stagnant for a while and manufactures had been shipping drives with 2TByte/Platter.
- But with the HMAR, the areal density is expected to increase again.
- A small laser is attached to a recording head, designed to heat a tiny spot on the disk where the data will be written. This allows a smaller bit cell to be written as either a 0 or a 1.
- Current projections are that HAMR can achieve 5 Tbpsi, enabling hard drives with capacities higher than 100 TB.

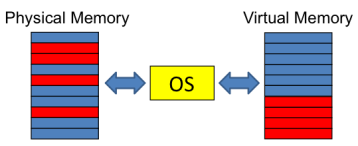
- The major problem with packing bits so closely together on conventional magnetic media is that the data bits become unstable and may flip (0 → 1 or 1 → 0).
- To make the media maintain their stability to store bits over a long period of time, the recording media needs to have a higher coercivity.
- Higher coercivity implies the media is magnetically more stable during storage, but it would also be more difficult to change the magnetic characteristics of the media when writing.
- For that, a laser is employed to heat a tiny region of several magnetic grains for a very short time (~1 ns) to a temperature high enough to lower the media's coercive field to below that of the write head's magnetic field. This is the write process.
- Immediately after the heat pulse, the region quickly cools down and the bit's magnetic orientation is frozen in place. The data is stored in the media and would be stable due to the high coercivity of the recording media.
- See the graph in the previous slide for the visual illustration.

Chapter 7 Virtual Memory

① Virtual Memory Management
- Each program has their own Virtual Address Space; Actual program code and data stored in the system memory
- OS translate VM address to reference system memory
- subset of program code/data available in system memory during runtime

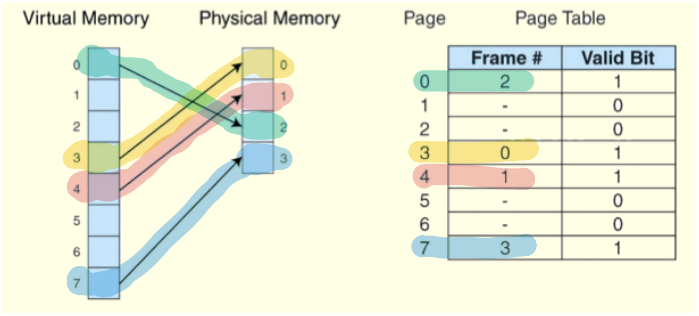
② Virtual vs Physical Memory
- Addresses used by program is generated by compiler as virtual address (typically different from the physical memory address the program is resided)
- Translation needs to be done by OS.

③ Advantages
- Each VM space is isolated, prevents corruption between these spaces.
- Allows efficient & safe storing of different programs within the actual physical mem space.
- Allows one or more programs to run in the physical memory simultaneously even if total size of all the programs is larger than the actual physical memory size.
- OS relocates blocks in VM in available spaces in actual physical mem.



④ Address mapping

- Paging: Memory space partitioned into fixed sized blocks
- Segmentation: Memory space partitioned into variable sized segments.

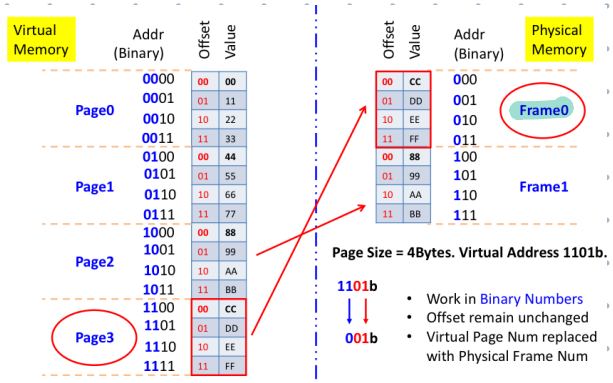


Frame: Physical Frame no. in main mem.
Page: Virtual Page used by prog.
Valid Bit: Indicates whether page is in main mem or not, '1' & '0'

⑤ Paging

- Each fixed size block is known as a Page / Frame
- Information concerning location of page is maintained in a page table

⑥ Address Translation

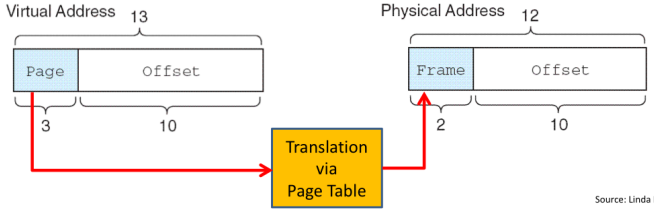


* Page table for this example:

Page	Frame	Valid Bit
0	-	0
1	-	0
2	1	1
3	0	1

Paging Example

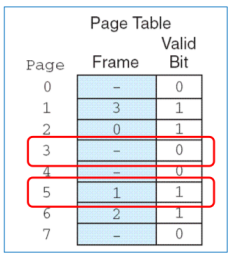
- Consider a system with a virtual address space of 8K and a physical address space of 4K, and the system uses byte addressing.
- If page size is 1KByte, we have 8 virtual pages mapping to 4 physical frames.
- Note that Virtual Page Size is always equal to Physical Frame Size.
- A virtual address has 13 bits ($8K = 2^{13}$) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- A physical memory address requires 12 bits, the first two bits for the page frame and the trailing 10 bits the offset.



Source: Linda Null

Paging Example

- Suppose we have the page table shown below.
- What happened when CPU access virtual address location
  - 0x1553
  - 0x0FA0



Source: Linda Null

- 0x1553 = 1010101010011b
  - Data resides in Virtual Page 5
  - From Page Table, Virtual Page 5 is mapped to Physical Frame 1 and Valid bit is 1.
  - Physical Address = 010101010011b = **0x0553**
- 0x0FA0 = 0111110100000b
  - Data resides in Virtual Page 3
  - From Page Table, Valid bit is 0
  - Data not in Physical Memory
  - **Page Fault**

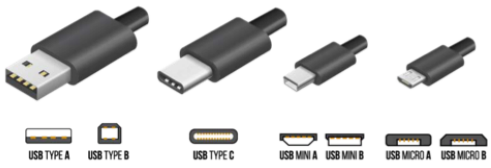
⑦ Translation lookaside buffer

- Page table located in main memory so access is relatively slower than internal mem access
- TLB stores a list of most recently / frequently used address translation entries
  - ↳ Implemented with fast memory (SRAM)
  - ↳ Each entry includes Virtual Page no. and its corresponding frame no.

Chapter 8 Communication and User Interface

① Universal Serial Bus (USB)

4 basic pinout : VBUS , Data+/Data-, Ground



- All data transactions initiated by the USB Port
- Power supplied to the devices by the Host (Computer) or Hub
  - ↳ or device could have its own power source
- * When the device is first connected to the Host, it will undergo an enumeration process where device & Host will exchange information on their capability and requirement.
  - ↳ Host cannot communicate with device until properly enumerated.
  - ↳ Host will check if it has required device driver to support the device according to the USB device class the device belongs to.

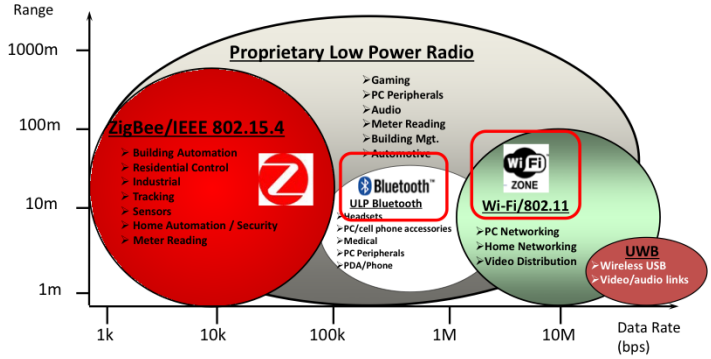
There are many USB Device Classes, common ones are
- Human Interface Device (HID) used in Mouse/Keyboard.
- Communication Device Class (CDC) used to implement Virtual COM Port e.g. Arduino Board, MSP432 Launchpad used in the lab.
- Mass Storage Class (MSC) used to interface to external USB HDD/SSD.
- USB Audio Class used to stream audio to USB headset/Microphone.

② High-Definition Multimedia Interface (HDMI)

- Proprietary A/V Interface for transmitting uncompressed video data and compressed or uncompressed digital audio data from source device to a compatible receiver.
- Consumer Electronic Control allows HDMI devices to control each other.

③ Industrial, Scientific & Medical Radio Frequency Band.

- Two commonly known ISM bands are 2.4GHz and 5.8GHz



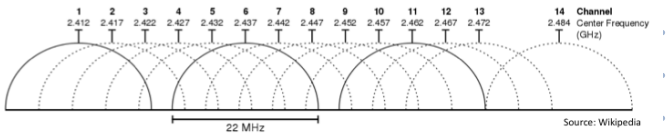
④ Wifi

- Family of wireless networking technologies based on the IEEE 802.1 family of standards
- Operates in 2.4GHz & 5.8GHz range
- Infrastructure Mode: Star topology
  - ↳ Access Point / Router at center of network, connected to other devices.
- Adhoc mode: Peer to Peer connection
- Transmission Range between 20m to 150m
  - ↳ Affected by transmission frequency, power & interface.

⑤ Bluetooth
- Used for low data rate wireless transmission (focus on low power consumption)
- Operates in 2.4 GHz range.
- Transmission range kept to 10-20m to keep power consumption low.

⑥ Factors affecting transmission range
- Transmission Power : Power supplied ↑ Transmission Range ↑.
- Transmission Frequency: Higher frequency ⇒ higher attenuation
- Higher frequencies have have lower range than lower frequency signal.
- Interference
  ↳ 2.4 GHz ⇒ Wifi & BT will interfere with each other

Mitigating Interference (some methods)



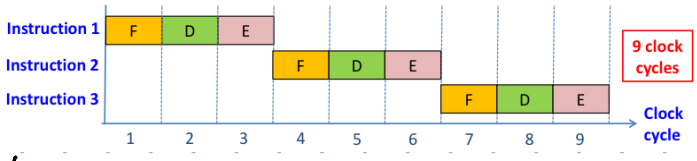
- "2.4Ghz ISM band" really consist of a band of frequencies between 2.4 and 2.5Ghz. Similarly, "5.8Ghz band" also span across a certain frequency range.
- Figure above shows the Wifi standard dividing the given frequency band into sub-bands known as channels.
- One common way to mitigate Wifi interference is to select different channels to use from your neighbours.
- For scenario of different wireless standard such as Bluetooth and Wifi, another common way is to use frequency hopping, i.e. to constantly hop from one channel to another so that transmission will eventually succeed. BT uses frequency hopping.
- There are other methodologies and techniques employed to further mitigate interference between transmitters.

⑦ Peripherals
- Keyboards, Mice
- Capacitive Touch Interface
- Camera
  ↳ Camera submodules
- Microphones
- Liquid Crystal Displays
- Audio Speaker

} Read slides for more info

Chapter 9 Processor Pipeline

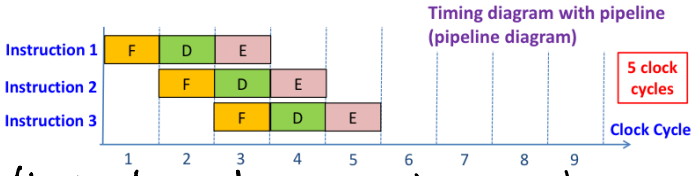
Simple CPU:



F = Fetch
D = Decode
E = Execute

(instructions done sequentially)

More complex CPU



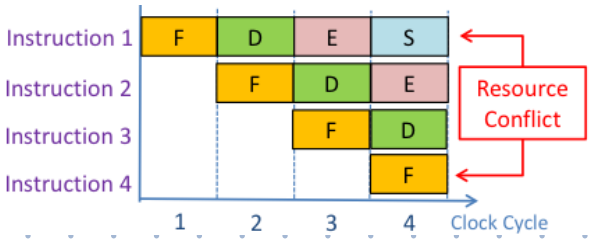
(instructions done simultaneously)

① Pipeline architecture
- Partitioning an instruction into simpler stages and assigning resources to allow these stages to be executed simultaneously
- Can have more than 10 pipeline stages
- Only possible if each stage uses independent operations

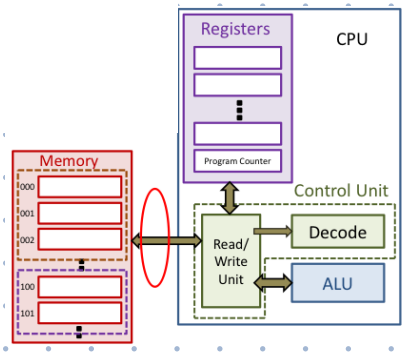
② Pipeline Efficiency
- Maximised when pipeline is filled
- After the pipeline is filled, it is able to release one instruction every cycle, giving the effect of 'executing' one instruction at every clock-cycle.

③ Pipeline Conflicts
- Efficiency reduced drastically if there are disruptions to pipeline (Pipeline conflicts)
- Causes a suspension to the pipeline; sometimes processor has to flush the instructions and reload with a new set of instructions
- Time is required to do the above ⇒ less instructions executed ⇒ lower processor performance.
- Caused by
  - ↳ Insufficient Resources
  - ↳ Data Dependency between Instructions
  - ↳ Pipeline flushing due to branch instructions.

④ Resource Conflicts



Example: Instruction 1 is storing result in memory when instruction 4 needs to fetch a new instruction



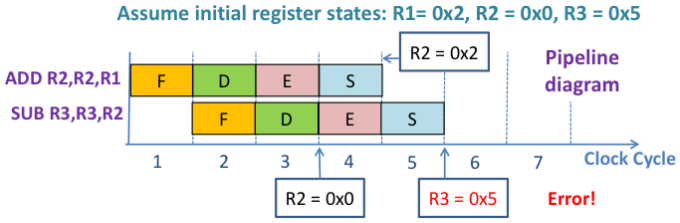
- Occurs when two instructions attempt to access the same resource in the same cycle

⑤ Resolving Resource Conflicts

- A processor with insufficient resources to operate each pipeline stage simultaneously will result in constant halting & flushing of pipeline.
  - ↳ might lead to a processor with worse performance compared to a simple CPU
- Pipeline processors usually have multiple resources to allow simultaneous operations
  - ↳ Internal buses
  - ↳ Control
  - ↳ Processing Units

⑥ Data Dependency Conflict

- Consider a FDES pipeline
- Actual operation of each instruction done during E stage (operands also fetched in E stage)
  - ↳ Result transferred to destination during S stage



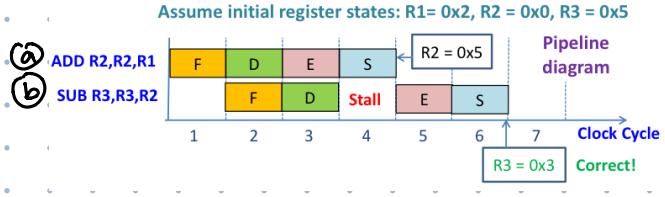
In the example [redacted], the old value (0x0) is still in R2 when SUB instruction is in Execute (E) stage, so R3 will have the wrong value (0x5) instead of the correct value (0x3).

- Overlapping nature of pipeline may lead to instances where data required for current instruction has not updated yet.
- * Arises when source operand of current instruction is also the destination operand of the previous instruction.

⑦ Resolving Data Conflicts

1. Stalling the Pipeline

↳ Hardware circuitry can be used to detect data dependency between instructions.

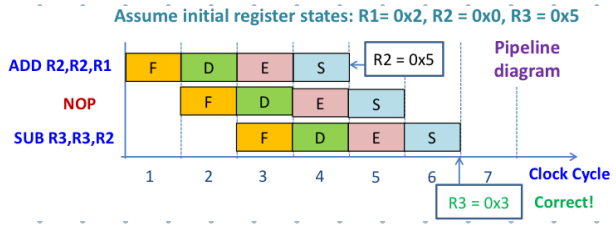


↳ When data dependency is detected, allow ⓐ to continue execution but stall ⓑ

2. Insert NOP Instructions

↳ Compiler can recognise and insert redundant instructions during compilation to reduce data conflict.

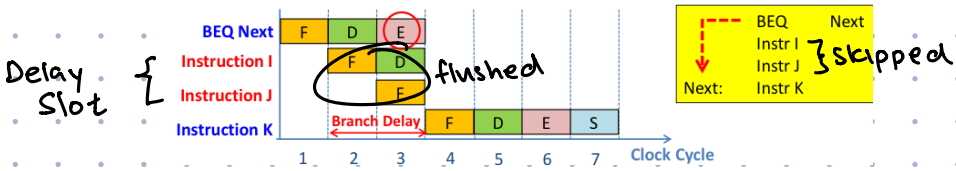
↳ No Operation (NOP) is inserted <u>between instructions</u> with data dependencies



(8) Branch Instructions

- Branch instructions perform two operations
  - ↳ Evaluate condition to determine if branch should be taken
  - ↳ If branch is taken, calculate branch target using adder in ALU
  - } Both done in E stage
- Unnecessary instructions may have been introduced into the pipeline before branch decision has been made
  - ↳ Processor needs to flush the pipeline to load correct instructions
    - ↳ Wasted cycles ⇒ Branch Delay.

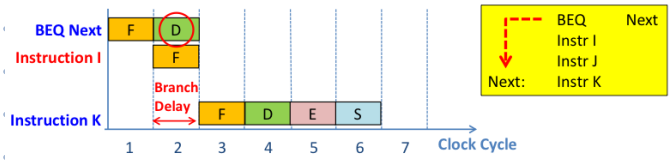
- Branch Delay
  - ↳ Causes significant performance loss



  - ↳ Branch target (Instruction K) is known only after E stage; I & J have already been fetched
  - ↳ Flushing discards I & J, wasting 2 cycles / Branch Delay = 2

(9) Reducing Branch Delay

- Can be reduced by making branch decision & calculating branch target earlier at D stage.
  - ↳ An additional Adder is introduced to be used in the D stage to enable earlier calculation of branch target.

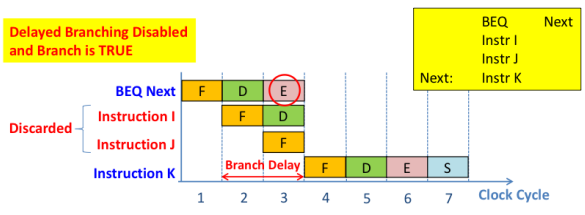


- After the Decode stage, the branch decision and the branch target is known.
- Hence if branch is taken, only Instruction I needs to be discarded.
- Branch delay is reduced to one cycle.

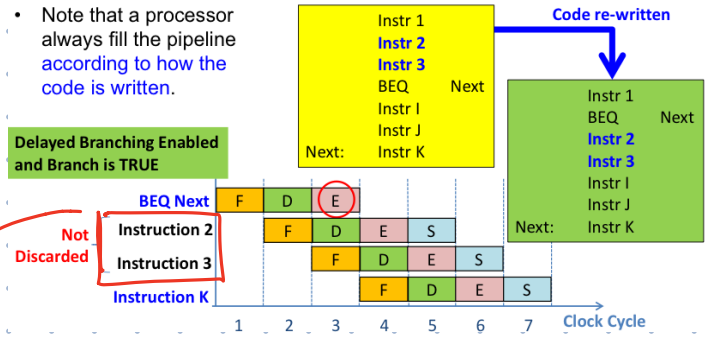
- Instruction is still fetched and discarded if branch is taken :C

(10) Delayed branching (feature in the processor; can be enabled / disabled)

- Ensures no instructions are discarded after branch
  - ↳ Delay slot instructions are always executed.
- User / compiler schedules independent (do not affect branch decision) instructions to be filled in the delay slots after the branch instruction.
  - ↳ Instructions in delay slots always executed ⇒ Zero Branch Delay.
  - ↳ If no independent instructions can be found, then we replace with NOP instructions.



- Note that a processor always fill the pipeline according to how the code is written.



(11) Independent Instructions

- Instructions that will always be executed in the original program & does not play a part in branch decision
- Should be instructions before a branch instruction
- Should not have effect on registers that will affect branch decision
  - ↳ Delay slots get fully executed before branch decision has been made
- If branch instruction is nested in another loop, independent instructions should come from within the parent loop.

(12) Dynamic Branch Prediction

- Hardware circuitry guesses outcome of a conditional branch
  - ↳ Branch history table implemented to store target addresses of taken branches
- If prediction is correct
  - ↳ executes as normal ⇒ 0 wasted cycles
- If prediction is incorrect
  - ↳ Flush incorrectly fetched instructions ⇒ wasted cycles
  - ↳ Update prediction bit & target address for future use.

Address of Branch Instruction	Predicted Target address	Prediction Result (T/F)
Address of "BEQ Next"	Next	T

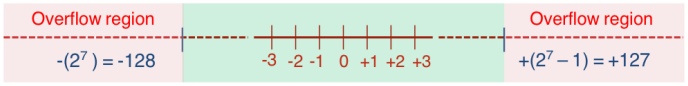
(13) Not Important

Connecting to the Real World

- ARM Cortex M3/M4 Processor
  - 3-stage pipeline. Instruction Fetch, Instruction Decode and Instruction Execute)
- Branch speculation.
  - When a branch instruction is encountered, the decode stage also includes a speculative instruction fetch that could lead to faster execution.
  - The processor fetches the branch destination instruction during the decode stage itself.
  - During the execute stage, the branch is resolved and it is known which instruction is to be executed next.
  - If the branch is not taken, the next sequential instruction is already available.
  - If the branch is taken, the branch instruction is made available at the same time as the decision is made.

Chapter 10 Computer Arithmetic

① Positional Numbering System (just number bases)
② Positive & Negative Numbers (Binary)
↳ MSB
↳ 2s complement
③ Carry vs Overflow



8 bits representation

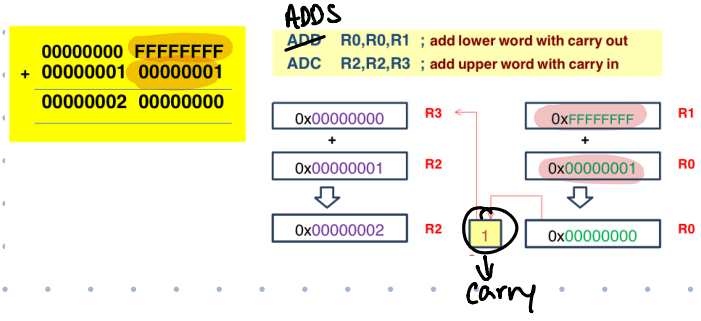
- Check MSB of operands & result to detect overflow
* - Conditions for overflow
↳ Addition: MSB(A) = MSB(B) but MSB(Result) ≠ MSB(A) (For A + B = Result)
↳ Subtraction: MSB(A) ≠ MSB(B) and MSB(Result) ≠ MSB(A) (For A - B = Result)

Operation	Conditions		Result
A + B	A > 0	B > 0	< 0
A + B	A < 0	B < 0	> 0
A – B	A > 0	B < 0	< 0
A – B	A < 0	B > 0	> 0

- Carry = 1 always indicates overflow (result too large to be stored in given no. of bits)
↳ Overflow flag means nothing for unsigned numbers
- For signed numbers, Overflow = 1 indicates overflow
↳ Carry flag set for signed numbers does not necessarily means overflow occured.

Expression	Result	Carry?	Overflow?	Corrected Result?
0100 (+4) + 0010 (+2)	0110 (+6)	No	No	Yes
0100 (+4) + 0110 (+6)	1010 (-6)	No	Yes	No
1100 (-4) + 1110 (-2)	1010 (-6)	Yes	No	Yes
1100 (-4) + 1010 (-6)	0110 (+6)	Yes	Yes	No

④ Signed Extension
⑤ Multi Precision Arithmetic
- We can add 64-bits operand using a 32-bit adder using multi-precision addition
- Involves computation of numbers whose preasion is larger than what is supported by the maximum size of processor register.



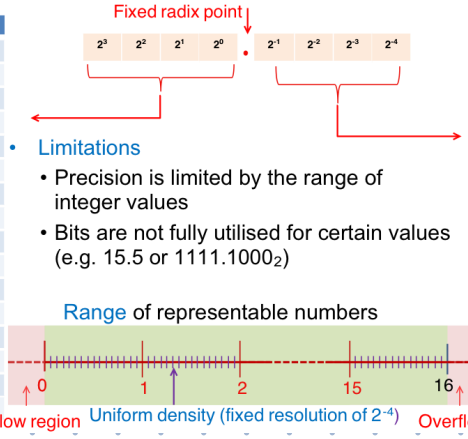
EzPz

⑥ Range and Precision
- Range = Interval between smallest and largest representable number.
- Precision = Amount of information used to represent each number
↳ eg 1.666 has more precision than 1.67
↳ Indicated by the number of tick marks

(7) Fixed-Point Representation

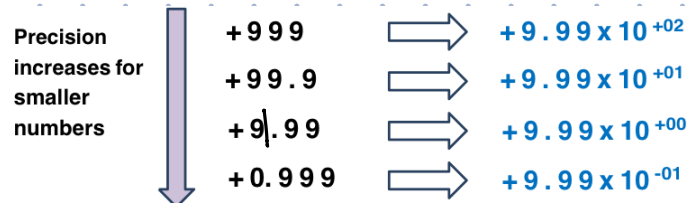
- Can represent integer and/or fractional values

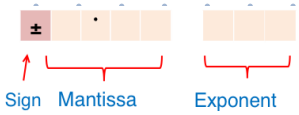
Unsigned Integer	
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15



Fractional Values	
0000	0.0000
0001	0.0625
0010	0.1250
0011	0.1875
0100	0.2500
0101	0.3125
0110	0.3750
0111	0.4375
1000	0.5000
1001	0.5625
1010	0.6250
1011	0.6875
1100	0.7500
1101	0.8125
1110	0.8750
1111	0.9375

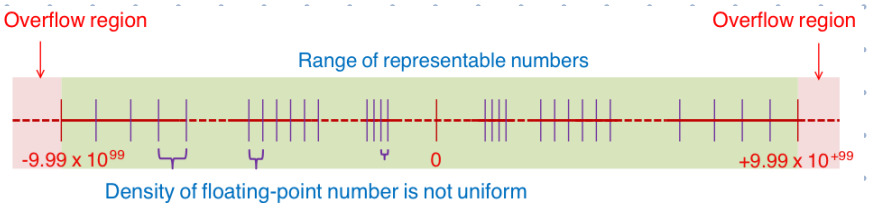
(8) Floating-Point Representation





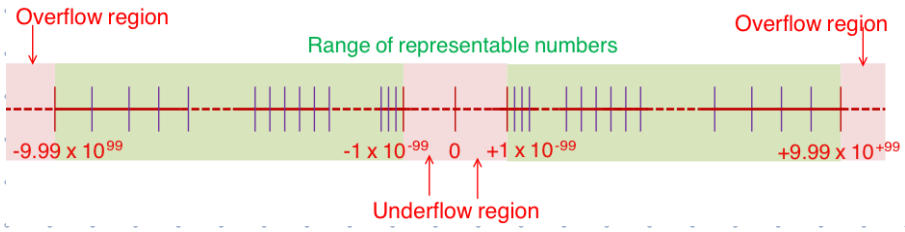
- 3 main fields needed for floating-point representation:
  - ↳ Sign: positive/negative
  - ↳ Mantissa: base value
  - ↳ Exponent: position of radix point

- Can represent values across a wide range
- Size of exponent determines range of representable numbers.
- Size of mantissa and value of exponent determines the representable precision
- precision is sacrificed to achieve greater range



(9) Normalisation

- necessary to avoid synonymous representation by maintaining one non-zero digit before radix-point
- maximises the number of bits of precision
- results in underflow regions where values close to 0 cannot be represented
  - ↳ can cause programs to crash if not handled properly.



(10) IEEE 754 Floating Point Standard

- Single Precision Floating-Point Number (32-bits) (Declaring Float)

- 1-bit sign + 8-bit exponent + 23-bit fraction

- Double Precision Floating-Point Number (64-bits) (Declaring Double)

- 1-bit sign + 11-bit exponent + 52-bit fraction

- Normalised numbers

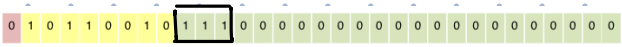
$(-1)^S \times (1.F)_2 \times 2^{E-BIAS}$

Sign bit
- S = 0 (positive); S = 1 (negative)

Exponent
- Biased representation (00000001 to 11111110)
- Value of exponent = E - **Bias**
- **Bias = 127** (Single Precision) and **1023** (Double Precision)

Fraction
- Assumes hidden **1.** (not stored) for normalised numbers
- Value of normalised floating point number is:
  $(-1)^S \times (1 + f_1 \times 2^{-1} + f_2 \times 2^{-2} + f_3 \times 2^{-3} + f_4 \times 2^{-4} + \ldots)_2 \times 2^{E-Bias}$

(11) Converting Single Precision to Decimal

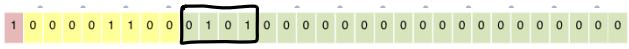


Sign = 0 (positive)

Exponent = 10110010_2 = 178; E – Bias = 178 – 127 = 51

1 + Fraction = $(1.111)_2 = 1 + 2^{-1} + 2^{-2} + 2^{-3} = 1.875$

Value in decimal = $+1.875 \times 2^{51}$



Sign = 1 (negative)

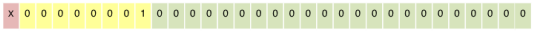
Exponent = 00001100_2 = 12; E – Bias = 12 – 127 = -115

1 + Fraction = $(1.0101)_2 = 1 + 2^{-2} + 2^{-4} = 1.3125$

Value in decimal = -1.3125×2^{-115}

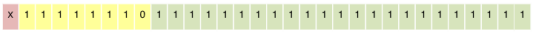
Representable Range for Normalised Single Precision

- In normalised mode, exponent is from 00000001 to 11111110
- Smallest magnitude normalised number

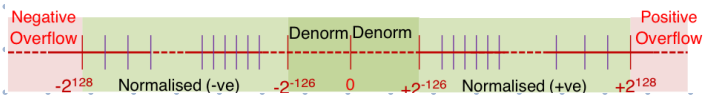


Exponent = $(00000001)_2$ = 1; E - Bias = 1 – 127 = -126
1 + Fraction = $(1.000\ldots000)_2$ = 1
Value in decimal = **1×2^{-126}**

- Largest magnitude normalised number

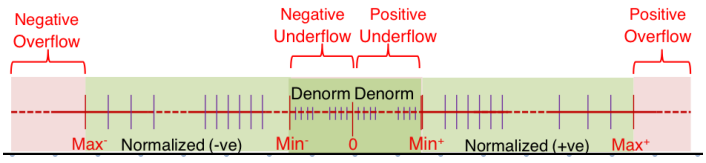


Exponent = $(11111110)_2$ = 254; E - Bias = 254 – 127 = 127
1 + Fraction = $(1.111\ldots111)_2 \approx 2$
Value in decimal = $2 \times 2^{127} \approx 2^{128}$



IEEE 754 Encoding

Mode	Sign	Exponent	Fraction
Normalized	1 / 0	00000001 to 11111110	Anything
Denormalized	1 / 0	00000000	Non zero
Zero	1 / 0	00000000	0000 … 0000
Infinity	1 / 0	11111111	0000 … 0000



(12) Fixed Point vs Floating Point (Given 32 bits)

- Floating point
  - ↳ max range : $(-2^{128}$ to $2^{128})$
  - ↳ max precision : less than 2^{-126} (near to 0)
- Fixed Point
  - ↳ max range : 2^{32} (Radix right of LSB)
  - ↳ max precision : 2^{-32} (Radix left of MSB)

- Floating Point yields larger range and better precision at small numbers with the same number of bits representation.
- Fixed point number has uniform precision across entire range.

(13) Effects of 4 Operators

- **Addition / Subtraction**
  - ↳ causes result to increase / decrease
  - ↳ Overflow eventually occurs when done sufficient number of times
  - * Take note of range of data type used when coding in High level language
  - * " width of registers and memory when coding in ASM
- **Multiplication**
  - ↳ Similar to Arithmetic Shift left.
  - ↳ Overflow at Min or Max of representable range
  - * Same as Add / Sub
- **Division**
  - ↳ Similar to Arithmetic Shift right
  - ↳ Truncation of LSB leads to loss in precision
  - * Try to perform division last to preserve precision

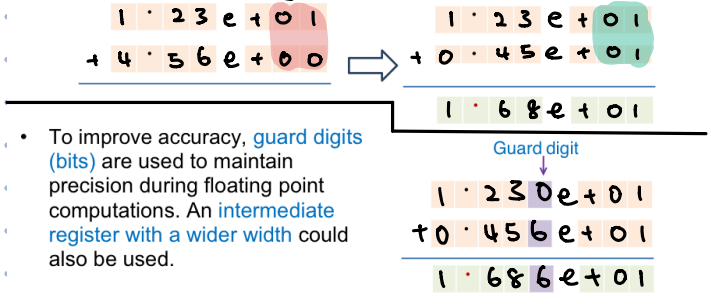
(14) Effects of rounding

- Removal of LSB(s) so that result can fit into representable bits
- up / down / nearest representable number.
  - ↳ limited by register width / data type
- Rounding error incurred since it is an approximation of actual result.

* Intermediate Registers with larger width allow intermediate processing to be done at higher precision. => reducing rounding error.

(15) Rounding error mitigation

- When performing add / sub, exponents must be aligned



- To improve accuracy, guard digits (bits) are used to maintain precision during floating point computations. An intermediate register with a wider width could also be used.
- Rounding is then performed to ensure that the result fits into the three significant digits in the mantissa

(16) Handling numbers with different magnitudes

- Order of evaluation can affect accuracy of result
- ∴ Add/subtract operands with similar size magnitude first

Example:

$1.00 \times 10^0 + 1.00 \times 10^0 + 1.00 \times 10^0 + 1.00 \times 10^0 + 1.00 \times 10^0 + 1.23 \times 10^3$

```
  1.0000e+00
+ 1.0000e+00
------------
  2.0000e+00
+ 1.0000e+00
------------
  3.0000e+00
+ 1.0000e+00
------------
  4.0000e+00
+ 1.0000e+00
------------
  0.0005e+03   Align from 5.000e+00 to 0.005e+003
+ 1.2300e+03
------------
  1.2355e+03   Rounding from 1.235e+03 to 1.24e+03
```

(17) Maximising Accuracy during Computation (Summary)

- Two issues: overflow and precision.
- From previous slides, addition, subtraction and multiplication may potentially lead to result overflowing the range of the representable number used.
- Division will lead to loss in precision as bits are loss due to truncation of LSB(s).
- Some rule of thumb
  - Accumulate/subtract numbers with small magnitude first to allow their magnitude to be comparable to big magnitude numbers.
  - Take note of the range of the number system used, which, depending on the usage scenario, can be a factor of the data type, number format, register/memory width etc.
  - Apply threshold to check for overflow if possible.
  - If no overflow checks are done, take note of the number/value of accumulation/multiplication that can be done without triggering overflow.
  - To preserve as much precision as possible, always do division last as far as possible.