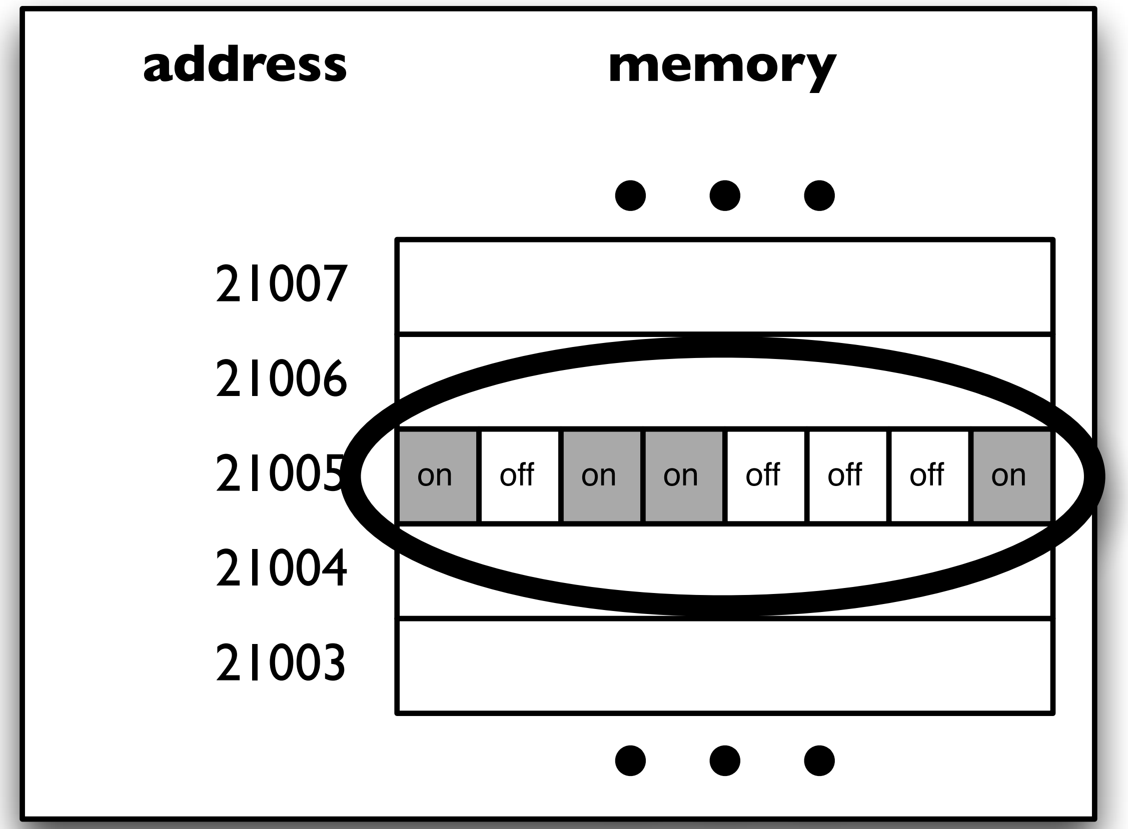
CS1021 Introduction to Computing I
Lecture Review

Rebekah Clarke
clarker7@scss.tcd.ie

A **bit** (***b****inary dig****it***) is a unit of information which has only two possible states

For our purposes these states are a binary 0 or 1



Memory is arranged in groups of Bytes.

8 bits = 1 byte

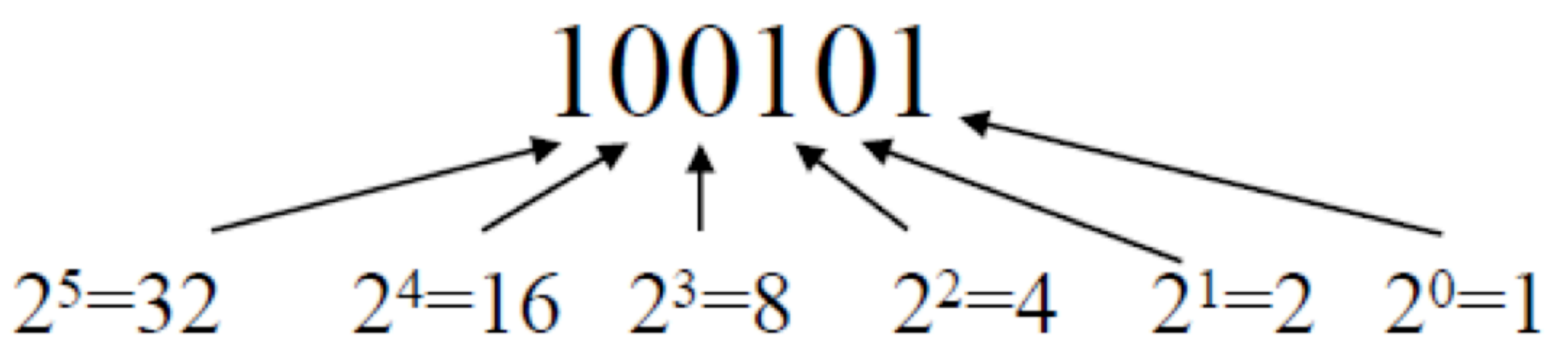
Each Memory address refers to a region of 8 bits (a byte)

Binary Number System

- Counting in Binary (Should recall 0 to 15 quite easily)
- Converting from binary to decimal
- Converting from decimal to binary

Converting from binary to decimal

e.g. $100101 = (1 \times 2^5)+(0 \times 2^4)+(0 \times 2^3)+(1 \times 2^2)+(0 \times 2^1)+(1 \times 2^0) = 37$



Converting from decimal to binary

2	37	1
2	18	0
2	9	1
2	4	0
2	2	0
2	1	1
	0	

read from bottom

$= 100101_2$

A number system with base b and n bits can represent $\mathbf{b^n}$ numbers

The **range** of a number system with n bits is from 0 to $b^n - 1$

Hexadecimal Number System, Base 16

- Converting from hexadecimal to decimal and back
- Converting between binary and hexadecimal

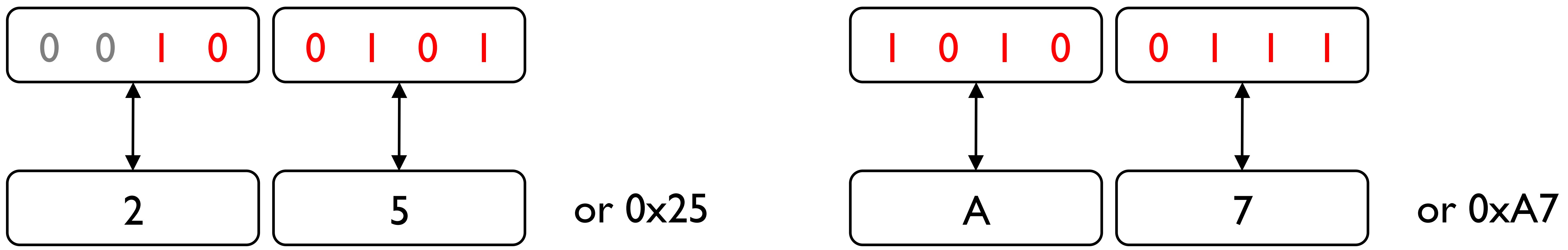
In general, subscript notation is used to show the intended base of a number e.g. 10_{10} is decimal but 10_2 is binary

In programming we can't use subscript to we use special prefix values

One hexadecimal digit represents the same number of values as four binary digits

conversion between hex and binary is trivial

the hexadecimal notation a convenient one for us



Hexadecimal is used by convention when referring to memory addresses: e.g. address 0x1000, address 0x4002

Lecture 1 - Memory

Units of Storage:

- 8 bits = 1 byte
- 2 bytes = 16 bits = 1 halfword
- 4 bytes = 32 bits = 1 word

address	memory
	• • •
21013	64
21012	78
21011	251
21010	35
21009	27
21008	89
21007	135
21006	196
21005	72
21004	91
21003	206
21002	131
21001	135
21000	78
20999	109
20998	7
	• • •

ASCII is a standard used to encode alphanumeric and other characters associated with text

Each character is stored in a single byte value (8 bits)

We check ASCII values by consulting an ASCII table (no need to memorise anything)

	0	1	2	3	4	5	6	7
0	NUL	DLE	SPACE	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

Registers:

Small, temporary internal storage on the processor. Faster to access than main memory

ARM has 16 word-sized registers

R15 is special – the Program Counter

Avoid using R13…R15 (for now …)

Move

```
MOV   R0, #0
```

Addition

```
ADD   R1, R2, R3
```

Subtraction

```
SUB   R0, R0, R1
```

Multiplication

```
MUL   R0, R1, R2
```

- cannot use MUL to multiply by a constant value
- MUL Rx, Rx, Ry produces unpredictable results

Immediate Operands - Small constant values we want to use but don’t need to store

```
MOV   R0, #2
```

```
ADD   R0, R1, #4
```

Lecture 2 - LoaD Register (LDR)

Loa**D** **R**egister instruction can be used to load any 32-bit signed constant value into a register

Note: =3 for LDR (not #3 as with MOV)

```
    ...
    LDR   r2, =3               ; tmp = 3
    MUL   r2, r1, r2           ; tmp = x * tmp
    ...
```

```
    ...
    LDR   r4, =0xA000013C      ; r4 = 0xA000013C
    ...
```

Cannot fit large constant values in a 32-bit MOV instruction so we use LDR instead

For small values, the assembler replaces LDR with MOV.
Otherwise, LDR utilises main memory to store the larger values (therefore is slowed down)

Any program, in any language, is composed of a sequence of machine code instructions that are stored in memory. Assembly language is translated into machine code by the assembler.

Every ARM machine code instruction is 32-bits long, it contains the operation (instruction) and the operands

The instruction templates are found in the ARM instruction set and both the destination operand (Rd) and the Source Operands (Rn, Rm) are filled in

Example – `add r0, r0, r2`

31 … 20	19 … 16	15 … 12	11 … 4	3 … 0	
1 1 1 0 0 0 0 0 1 0 0 0	Rn	Rd	0 0 0 0 0 0 0 0	Rm	add instruction template **ADD Rd, Rn, Rm**
1 1 1 0 0 0 0 0 1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0 0 0 0 0	0 0 1 0	machine code binary

31 … 28	27 … 24	23 … 20	19 … 16	15 … 12	11 … 8	7 … 4	3 … 0	
E	0	8	0	0	0	0	2	machine code hexadecimal

Two's Complement: Most common method for representing negative numbers

- Range: $-(2^{N-1})$ to $+(2^{N-1} - 1)$
- One representation of zero
- Positive numbers are represented as in an unsigned number system
- Negative numbers are represented by the 2's complement of their absolute value
  i.e. Invert and add one (MVN to invert in assembly)
- *Advantage*: Arithmetic operations are identical to unsigned binary as long as the carry is discarded

N.B.The computer does not know that we are storing negative numbers it is all up to our interpretation of the stored values

Lecture 3 - Condition Code Flags

The Condition Code Flags (N, Z, V, C) can be **optionally** updated to reflect the result of an instruction

To update: Use an ’S’ on the end of the instruction e.g. ADDS, MOVS

Zero (Z) - Set if the result of the last instruction was exactly zero

Negative (N) - Set if the result of the last instruction was negative i.e. If the Most Significant Bit (MSB) of the result is 1

Carry (C) - Set if the result of an n bit operation does not fit in n bits

Overflow (V) - Set if the result of an addition or subtraction gives a result that is outside the range of the signed number system

Generally: The carry flag is relevant to unsigned arithmetic whereas the overflow flag is relevant to signed arithmetic

Lecture 3 - Overflow Rules

Addition rule ($r = a + b$)

$$V = 1 \text{ if } \quad \begin{array}{l} MSB(a) = MSB(b) \text{ and} \\ MSB(r) \neq MSB(a) \end{array}$$

i.e. oVerflow accurs for addition if the operands have the same sign and the result has a different sign

Subtraction rule ($r = a - b$)

$$V = 1 \text{ if } \quad \begin{array}{l} MSB(a) \neq MSB(b) \text{ and} \\ MSB(r) \neq MSB(a) \end{array}$$

i.e. oVerflow occurs for subtraction if the operands have different signs and the sign of the result is different from the sign of the first operand

Branch instruction - we can modify the value in the Program Counter to “point” to an instruction of our choosing, breaking the pattern of sequential execution

`CMP` (CoMPare) instruction performs a subtraction and updates the Condition Code Flags without storing the result of the subtraction

BEQ – Branch if EQual

```
        CMP   r2, #0              ; subtract 0 from r2, ignoring result but
                                  ; updating the CC flags
        BEQ   endwh               ; if the result was zero then branch to endwh
        ...   ...                 ; otherwise (if result was not zero) then keep
                                  ; going (with sequential instruction path)
endwh
```

Description	Symbol	Instruction	Mnemonic	Condition Code Flag Evaluation
Equality				
Equal	=	BEQ	EQual	Z=1 i.e. Z is set
Not equal	≠	BNE	Not Equal	Z=0 i.e. Z is clear
Inequality (unsigned values)				
Less than	<	BLO (or BCC)	LOwer	C=0
Less than or equal	≤	BLS	Lower or Same	C=0 or Z=1
Greater than or equal	≥	BHS (or BCS)	Higher or Same	C=1
Greater than	>	BHI	HIgher	C=1 and Z=0
Inequality (signed values)				
Less than	<	BLT	Less Than	(N=1 and V=0) or (N=0 and V=1) i.e. N!=V
Less than or equal	≤	BLE	Less than or Equal	Z=1 or N!=V
Greater than or equal	≥	BGE	Greater than or Equal	(N=1 and V=1) or (N=0 and V=0) i.e. N=V
Greater than	>	BGT	Greater Than	Z=0 or N=V
Flags				
Negative Set		BMI	MInus	N=1
Negative Clear		BPL	PLus	N=0
Carry Set		BCS (or BHS)	Carry Set	C=1
Carry Clear		BCC (or BLO)	Carry Clear	C=0
Overflow Set		BVS	oVerflow Set	V=1
Overflow Clear		BVC	oVerflow Clear	V=1
Zero Set		BEQ	EQual	Z=1
Zero Clear		BNE	Not Equal	Z=0

Lecture 4 - Selection - General Form

Template for if-then construct

```
if ( <condition> )
{
        <body>
}
<rest of program>
```

```
        CMP   variables or constants in <condition>
        Bxx   endiflabel on opposite <condition>
        <body>
endiflabel
        <rest of program>
```

Template for if-then-else construct

```
if ( <condition> )
{
    <if body>
}
else {
    <else body>
}
<rest of program>
```

```
        CMP   variables or constants in <condition>
        Bxx   elselabel on opposite <condition>
        <if body>
        B     endiflabel unconditionally
elselabel
        <else body>
endiflabel
        <rest of program>
```

Lecture 4 - Iteration – General Form

Template for while construct

```
<initialize>

while ( <condition> )
{

    <body>

}
<rest of program>
```

```
          <initialize>

whilelabel
          CMP     variables or constants in <condition>
          Bxx     endwhlabel on opposite <condition>
          <body>
          B       whilelabel unconditionally
endwhlabel
          <rest of program>
```

Template for do-while construct

```
<initialize>


do {
    <body>
} while
( <condition> )


<rest of program>
```

```
          <initialize>


dolabel
          <body>
          CMP   variables or constants in <condition>
          Bxx   dolabel on <condition>


          <rest of program>
```

Lecture 4 - Compound Conditions – AND


```
if (x ≥ 40 AND x < 50)
{
        y = y + 1
}
```

Test each condition and if any one fails, branch to end of if-then construct (or if they all succeed, execute the body)

```
        ...   ...
        CMP   r1, #40              ; if (x ≥ 40
        BLO   endif                ;  AND
        CMP   r1, #50              ;  x < 50)
        BHS   endif                ; {
        ADD   r2, r2, #1           ;  y = y + 1
endif                              ; }
        ...   ...
```

Lecture 4 - Compound Conditions – OR


```
if (x < 40 OR x ≥ 50)
{
        z = z + 1
}
```

Test each condition and if they all fail, branch to end of if-then construct (or if any test succeeds, execute the body without testing further conditions)

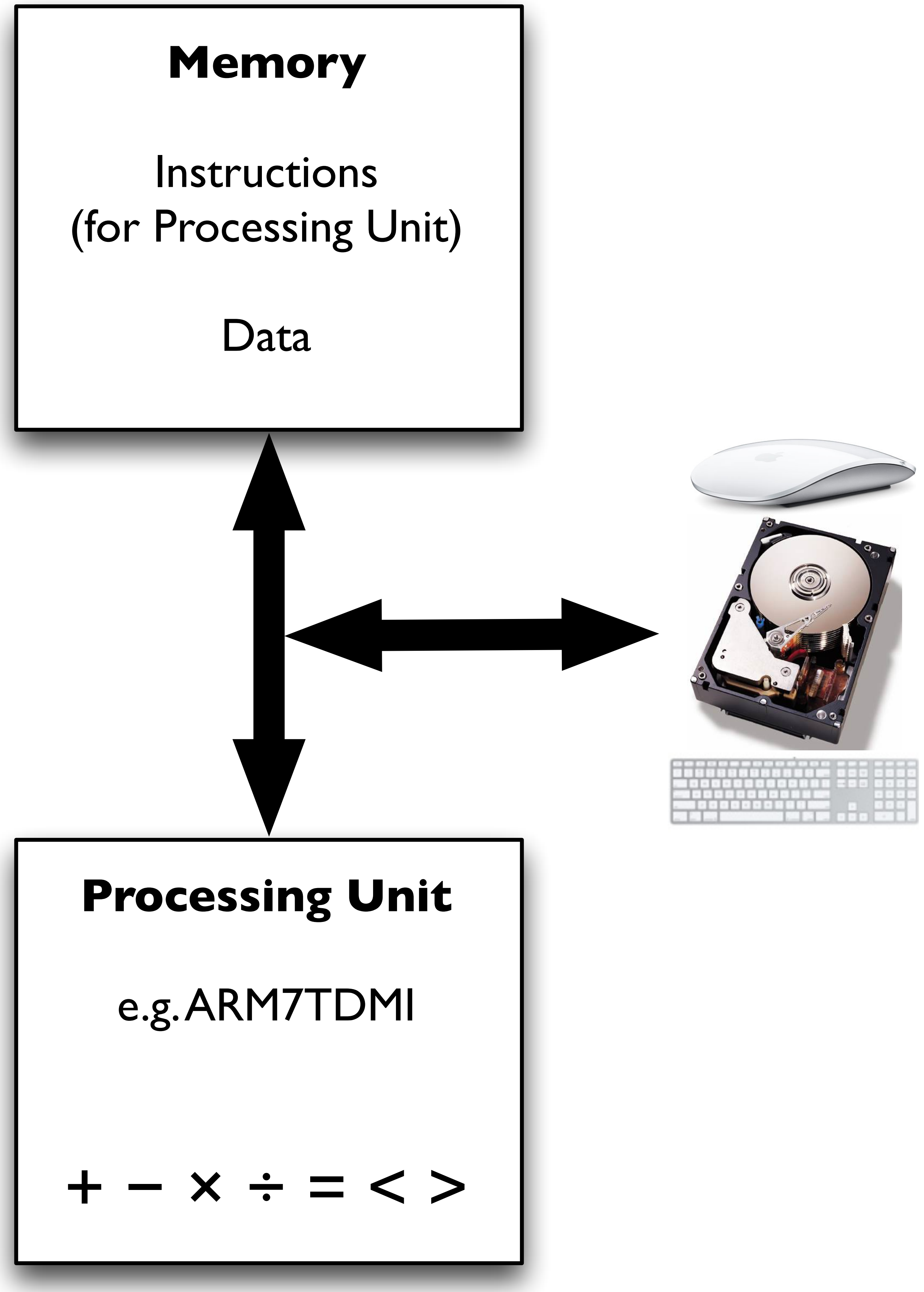
```
        ...   ...
        CMP   r1, #40                 ; if (x < 40
        BLO   then                    ;   ||
        CMP   r1, #50                 ;  x ≥ 50)
        BLO   endif                   ; {
then    ADD   r2, r2, #1              ;  y = y + 1
endif                                 ; }
        ...   ...
```

Lecture 5 - Memory

The ARM processor has some small internal storage in the form of registers

To increase the amount of memory available the ARM can access external storage (Main Memory)

ARM7TDMI is based on a Load – Store Architecture so it cannot directly perform operations on values in main memory. Information must be loaded into a register first.



To operate on a value stored in memory it must be loaded into a register first, then operations performed on the register

e.g. Loading Byte-Sized Values

```
adr = address of first value
LDR   R1, =AElems
```

Load the start address of the data into a register

```
value = Memory.byte[adr]
LDRB  R2, [R1]
```

Load the byte-size contents of memory at address **`adr`** *into the variable* **`value`**

```
address = address + 1
ADD   R1, R1, #1
```

Increase the address by one byte in order to move to the next value

To change a value in memory store the value from a register into memory

e.g. Storing Byte-Sized Values

```
adr = address of first value
LDR   R3, =CElems
```
Load the address where the data will be stored into a register

```
Memory.byte[adr] = value
STRB  R4, [R3]
```
Store the contents of the byte-size **`value`** *variable in memory at address* **`adr`**

```
address = address + 1
ADD   R3, R3, #1
```
Increase the address by one byte in order to move to the next storage location

Word-Sized Data : LDR, STR, DCD

Half-Word Sized Data : LDRH, STRH, DCW

Byte-Sized Data : LDRB, STRB, DCB

Example - Duplicate a sequence of byte-sized values stored in memory

```
start
   LDR  R1, =AElems     ; Load start address of AElems
   LDR  R2, =BElems     ; Load address of first storage loc

   LDRB R3, [R1]        ; Load the first value of AElems into R3 value = Memory.byte[AElems]

while
   CMP  R3, #0         ; while (value != 0)
   BEQ  ewhile         ; {

   STRB R3, [R2]        ; Store the contents of R2 in CElems Memory.byte[CElems] = value
   ADD  R1, R1, #1      ; Move to the next address
   ADD  R2, R2, #1      ; Move to the next storage address
   LDRB R3, [R1]        ; value = Memory.byte[AElems]
ewhile                  ; }

stop
  B stop

   AREA TestData, DATA, READWRITE
AElems DCB  1, 3, 8, 0          ; NULL terminated sequence of byte-sized values
BElems SPACE 56                 ; Reserve memory for the duplicated sequence

END
```

Clearing - Use AND operation, Clear with zeroes

```
LDR  r1, =0x61E87F4C ; load test value
LDR  r2, =0xFFFFFFE7 ; mask to clear bits 3 and 4
AND  r1, r1, r2      ; clear bits 3 and 4
```

Clearing 2 - Alternative: Use BIC operation, Clear with one’s

```
LDR  r2, =0x00000018 ; mask to clear bits 3 and 4
BIC  r1, r1, r2      ; r1 = r1 AND NOT(r2)
```

Setting - Use OR operation, Set with one’s

```
LDR  r2, =0x00000014 ; mask to set bits 2 and 4
ORR  r1, r1, r2      ; set bits 2 and 4
```

Inverting - Use EOR operation

```
LDR  r2, =0x00000014  ; mask to invert bits 2 and 4
EOR  r1, r1, r2       ; invert bits 2 and 4
```

Logical Shift Left - Use MOV operation with extra operand

```
MOV Rd, Rm, LSL #2    ; shift values left by 2 places
```

Logical Shift Right

```
MOV Rd, Rm, LSR #4    ; shift values right by 4 places
```

Instead of discarding the MSB when shifting left (or LSB when shifting right), we can cause the last bit shifted out to be stored in the Carry Condition Code Flag

Logical Shift with Carry- Use MOV**S** operation

```
MOVS Rd, Rm, LSL #1 ; shift left by 1 place, set flags
```

Note: Bit shifting can be used for multiplication and division by 2^n