# Part IA Engineering

# Digital Circuits &

# Information Processing

# Handout 3

# Microprocessors

Richard Prager
Tim Flack
January 2009



# Contents of Handout 3

Section A Microprocessor architecture.
Section B Memory.
These sections cover the material for questions 1 & 2 on examples paper 4.

Section C Microprocessor programming.
This section covers the material for questions 3 and 4 on examples paper 4.

Section D Flags.
Section E Addressing modes.
These sections cover the material for questions 5 – 9 on examples paper 4.

Section F The stack & subroutines.
This section covers the material for question 10 on examples paper 4.

# Handout 3 Section A

# Microprocessor Architecture

In this section we describe the main constituents of microprocessor systems and show how a number of simple program instructions are executed.



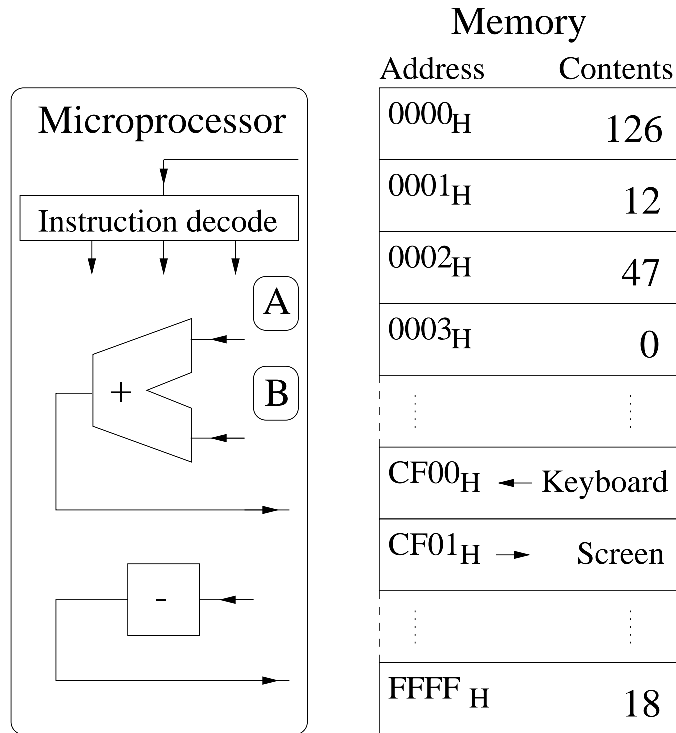
## Microprocessors

A computer on a chip.

Used in washing machines, microwaves, printers, cars, fax machines, sewing machines, video games, CD players, televisions, answerphones, electric organs, and home computers.

Very small, flexible, and cheap. Can be highly complex.

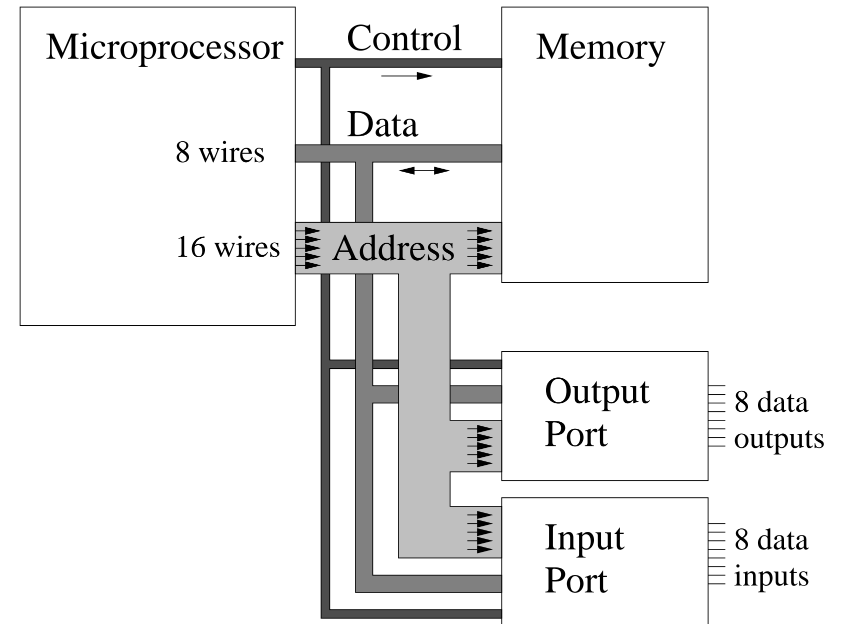
# Block Diagram



A and B symbolise 'accumulator' registers that can be used to store intermediate results inside the microprocessor.



# Bus Structure



For the 6800, the address and control bus go only from the microprocessor to the peripheral devices. The data bus is bi-directional

# Load Data to Microprocessor from Memory

Set address on *address bus*.

Set *read/write* wire high.

Set *address valid* control wire high. This, together with the address bus will activate *chip select* on the appropriate memory chip.

Read the data from the *data bus*.

The read/write wire is usually labelled $R/\overline{W}$



# Store Data to Memory from Microprocessor

Set address on *address bus*.

Set *read/write* wire low.

Set *address valid* control wire high. This, together with the address bus will activate *chip select* on the appropriate memory chip.

Drive the *data bus* with the correct voltages to represent the data that is to be stored in this memory location.

# Addressed Capacity of Memory

Let $a =$ number of address wires and $d =$ number of data wires.

The address bus is 16 bits wide ($a = 16$). There are 16 wires. The microprocessor can address up to $2^{16} = 65536$ memory locations. Every time you increase the number of address bus wires by one you double the addressed capacity.

The data bus is 8 bits wide ($d = 8$). Each location is an 8-bit byte. Each 8 bits can either store an unsigned number in the range 0–255 or a 2's-complement signed number in the range $-128$ to $+127$. Every time you increase the number of data bus wires by one the capacity goes up by $2^a$ bits.

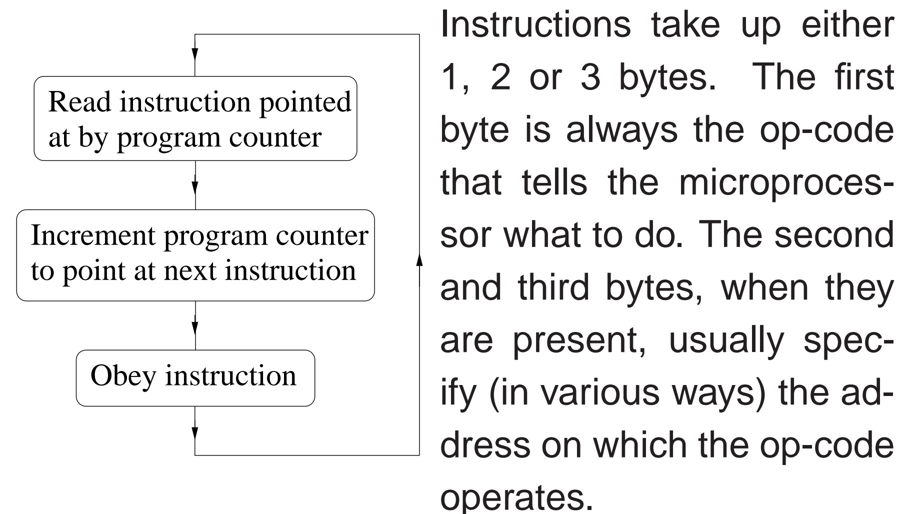
The total addressed capacity in bits is

$$d \times 2^a$$



# Program Counter

The program counter is a 16 bit register inside the 6800 that stores the memory address of the next instruction to be executed.



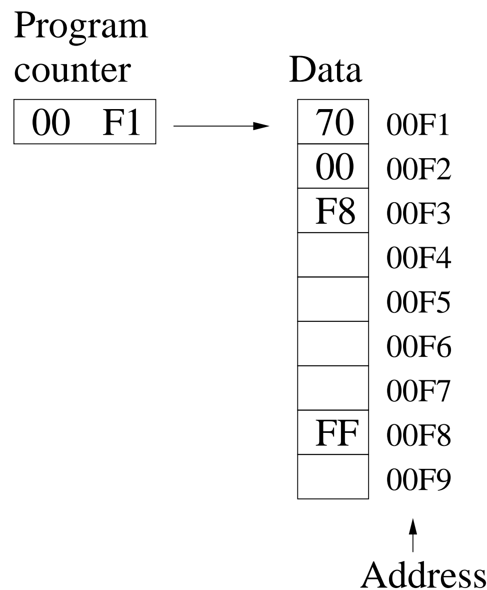

Instructions take up either 1, 2 or 3 bytes. The first byte is always the op-code that tells the microprocessor what to do. The second and third bytes, when they are present, usually specify (in various ways) the address on which the op-code operates.

The number of bytes taken up by an instruction for each op-code is fixed and built into the microprocessor chip.

# Example: NEG

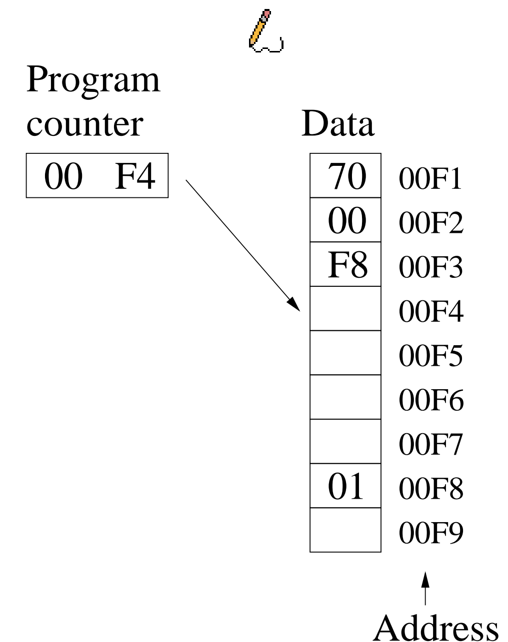
The op-code $70_H$ is (one version of) the NEG instruction that changes the sign of a binary number using 2's complement arithmetic.

The two bytes that follow the $70_H$ op-code specify an address in memory. When it executes the $70_H$ op-code the microprocessor will read the data at this address, change its sign and write the result back at the same address.






After the NEG instruction we have the following state:



# Registers

Apart from the Program Counter there are several other registers inside the microprocessor chip. Here is a complete list of them.

| Register | | Size |
| --- | --- | --- |
| Accumulator A | | 8 bits |
| Accumulator B | | 8 bits |
| Index Register | | 16 bits |
| Stack Pointer | | 16 bits |
| Program Counter | | 16 bits |
| Condition Code Register | | 8 bits |

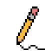


# The Accumulators

There are two 8 bit accumulators, A and B, that are used to store intermediate arithmetic results.

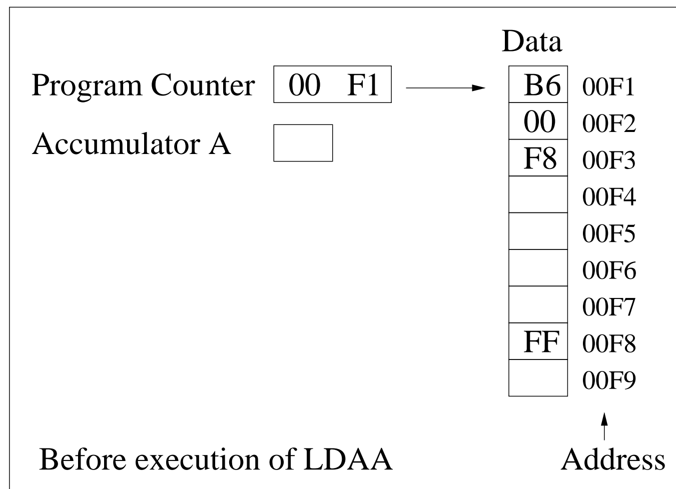
The LDAA instruction loads accumulator A.

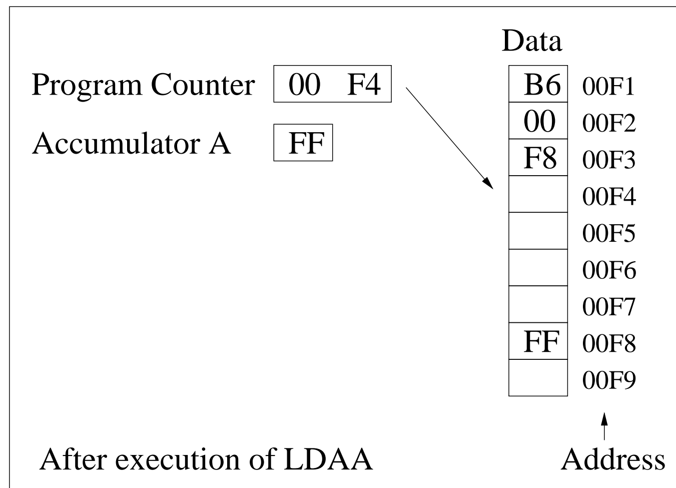
One op-code used for this is $\mathrm{B6}_H$ followed by two bytes containing the address from which the data should be loaded.



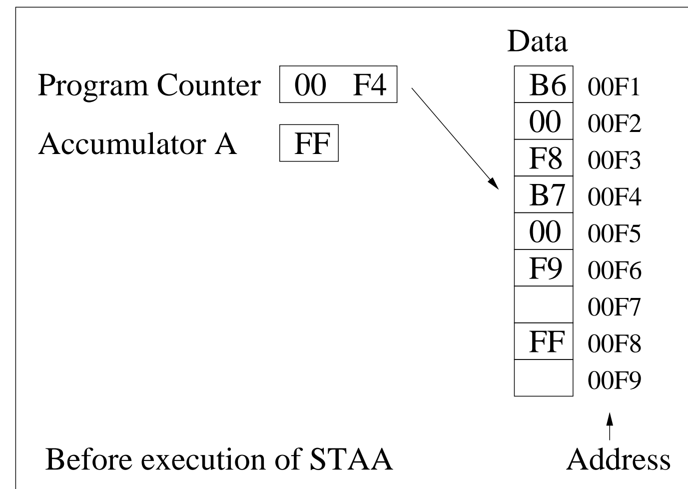
The STAA instruction stores the contents of accumulator A in memory.

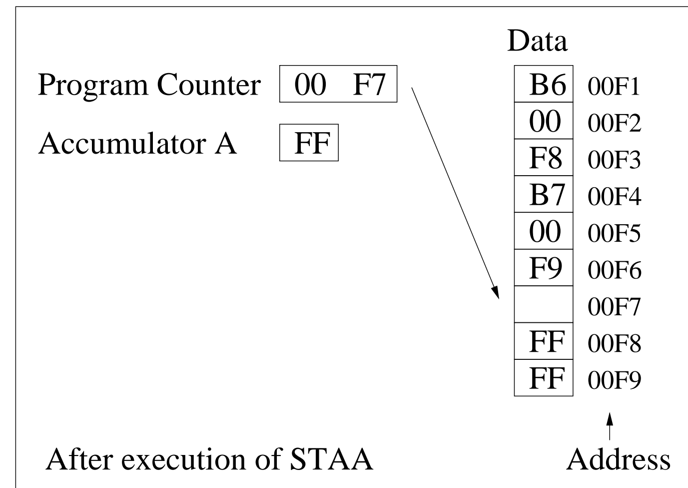
One op-code used for this is $\mathrm{B7}_H$ followed by two bytes containing the address at which the data should be stored.


Data
Program Counter
00 F1
Accumulator A
B6
00
F8
FF
00F1
00F2
00F3
00F4
00F5
00F6
00F7
00F8
00F9
Before execution of LDAA
Address


Data
Program Counter
00 F4
Accumulator A
FF
B6
00
F8
FF
00F1
00F2
00F3
00F4
00F5
00F6
00F7
00F8
00F9
After execution of LDAA
Address




Data
Program Counter
00 F4
Accumulator A
FF
B6
00
F8
B7
00
F9
FF
00F1
00F2
00F3
00F4
00F5
00F6
00F7
00F8
00F9
Before execution of STAA
Address


Data
Program Counter
00 F7
Accumulator A
FF
B6
00
F8
B7
00
F9
FF
FF
00F1
00F2
00F3
00F4
00F5
00F6
00F7
00F8
00F9
After execution of STAA
Address

# Addition

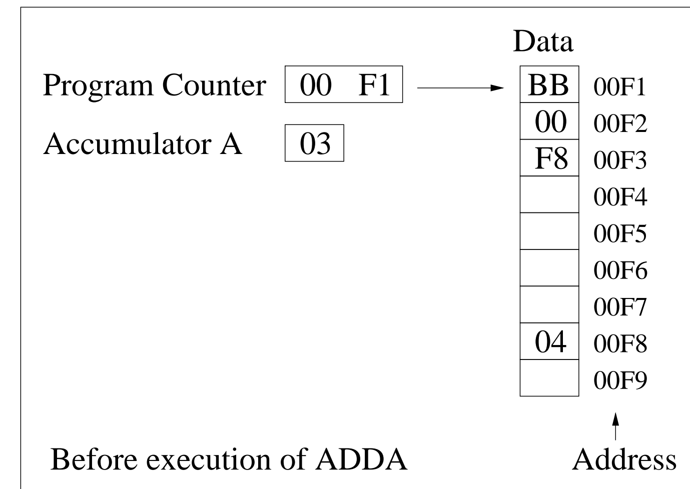
The ADDA instruction adds the contents of a specified memory location to accumulator A. One op-code used for this is $\mathrm{BB}_H$ followed by two bytes containing the address of the data to be added.

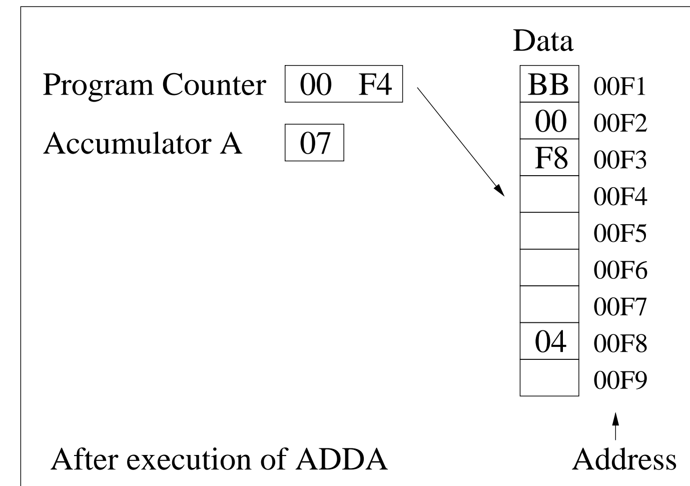
$$\text{ADDA} \qquad A + \text{memory} \rightarrow A$$

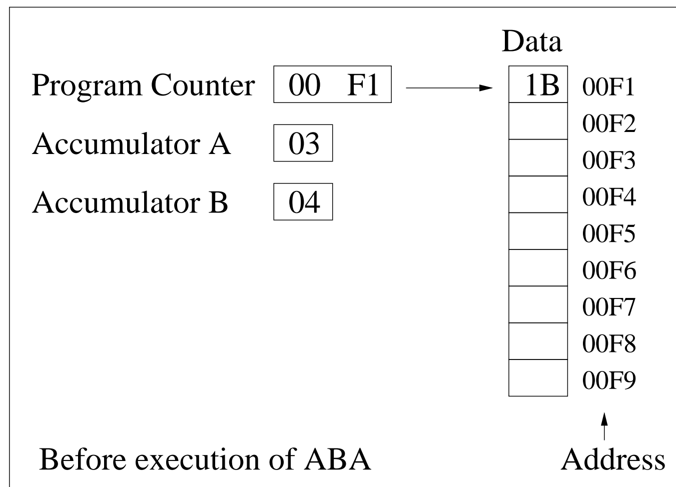
Accumulator B is just like accumulator A. Having two accumulators means you can use them together. Op-code $1\mathrm{B}_H$ is called ABA and adds the two accumulators together, leaving the sum in accumulator A.

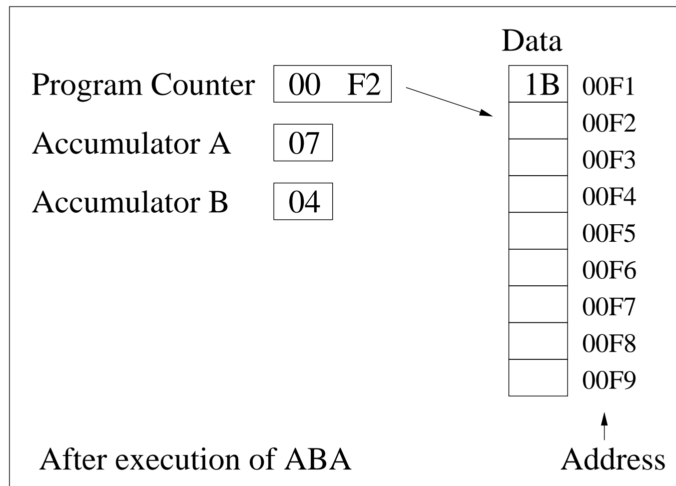
$$\text{ABA} \qquad A + B \rightarrow A$$










# Handout 3 Section B

# Memory

In this section we describe the internal structure of random access memory chips (RAM). The circuit for connecting together a number of memory chips as part of a microprocessor system is then introduced.

The section ends with a list of various types of memory chip, and a brief discussion of memory-mapped input/output.

## Memory Read and Write Operations

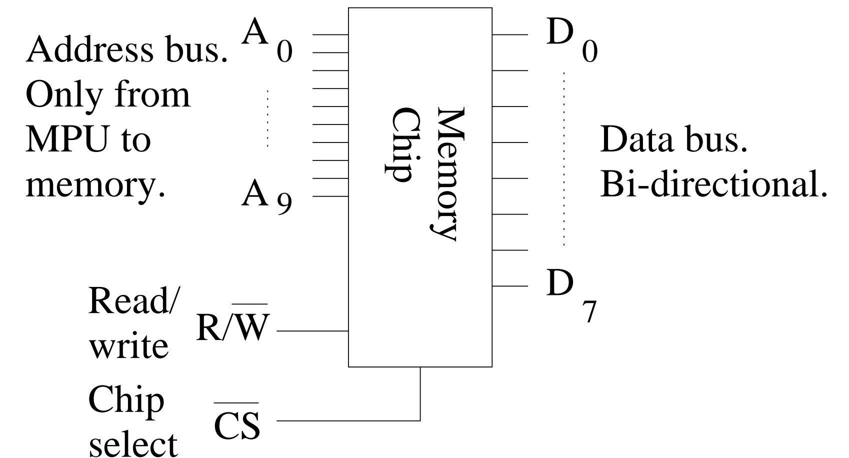
Memory chips have *address wires*, *data wires*, the *read/write* ($\mathrm{R}/\overline{\mathrm{W}}$)wire, and the *chip select* ($\overline{\mathrm{CS}}$) wire.

When $\overline{\mathrm{CS}}$ is high the memory chip does nothing.

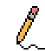
When $\overline{\mathrm{CS}}$ is low and $\mathrm{R}/\overline{\mathrm{W}}$ is low, the memory chip writes the data on the data bus into the location indicated by the address bus.

When $\overline{\mathrm{CS}}$ is low and $\mathrm{R}/\overline{\mathrm{W}}$ is high, the memory chip drives the data bus with the data from the location indicated by the address bus. This enables the microprocessor to read data out of the memory.

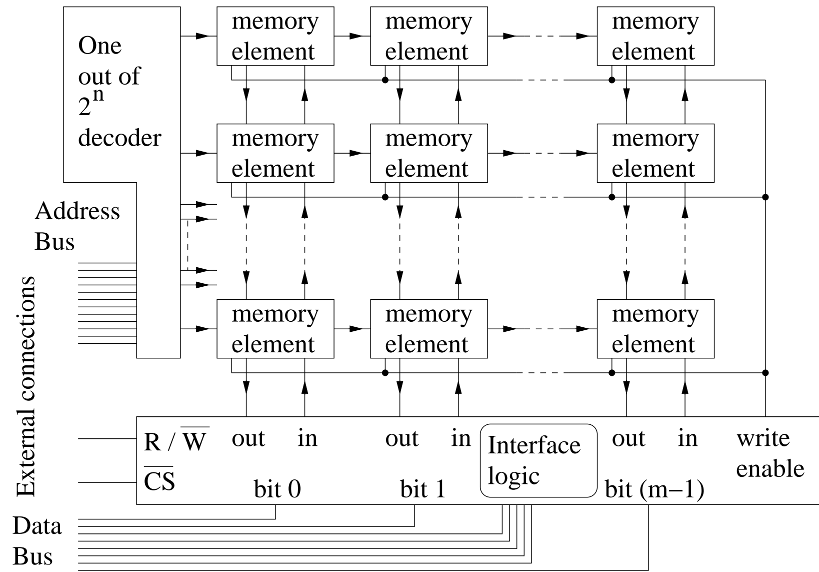


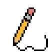



This memory chip has 10 address wires (sometimes called address lines) and 8 data wires. It can therefore store $8 \times 2^{10} = 8192$ bits of information.



$$8192 \text{ bits} = 8\text{kilobinary bits} = 1\text{kilobinary byte}$$

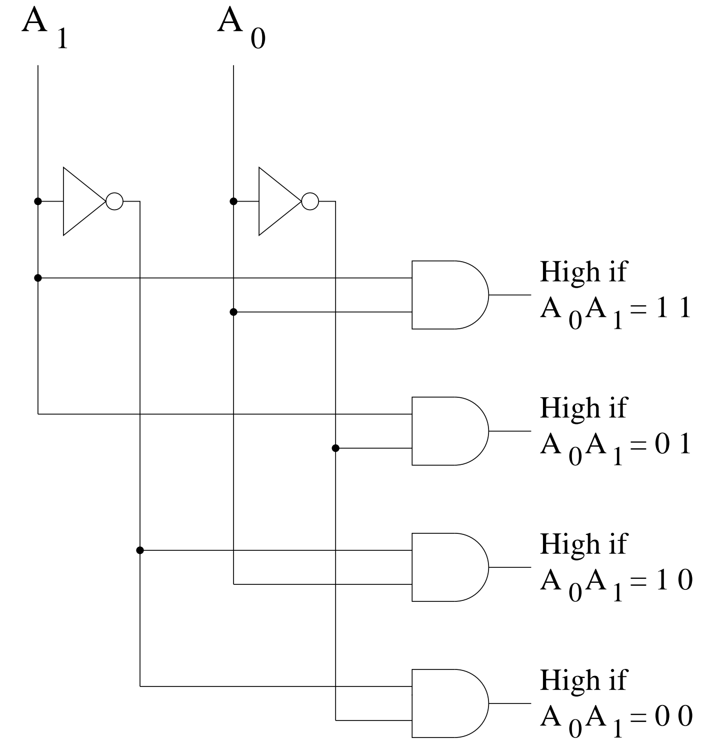


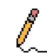



There is one row of memory elements for each address, and one column for each data wire.

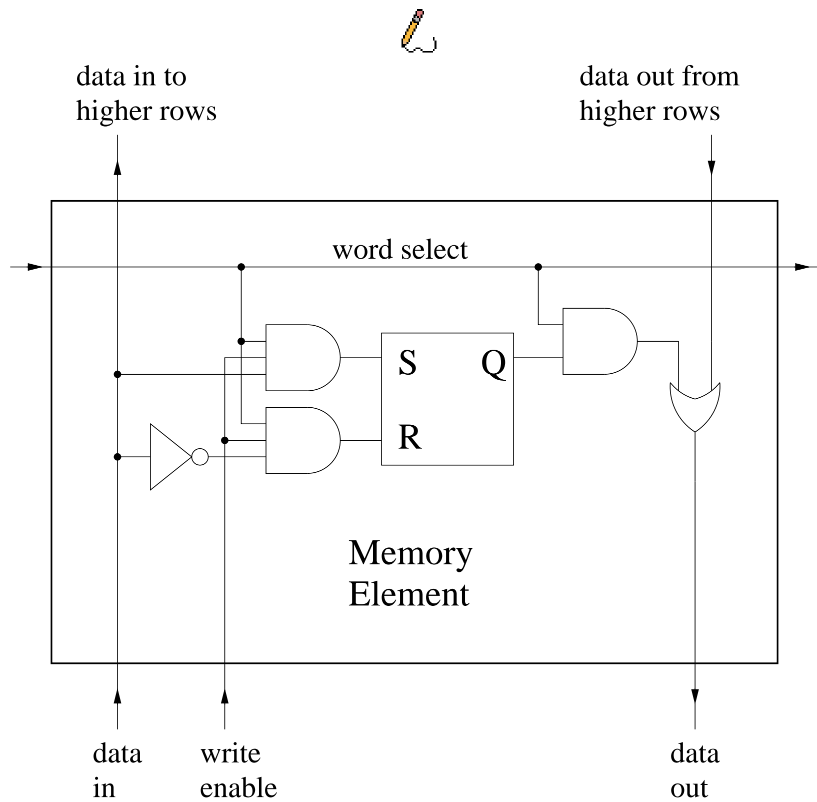


# Address Decoder



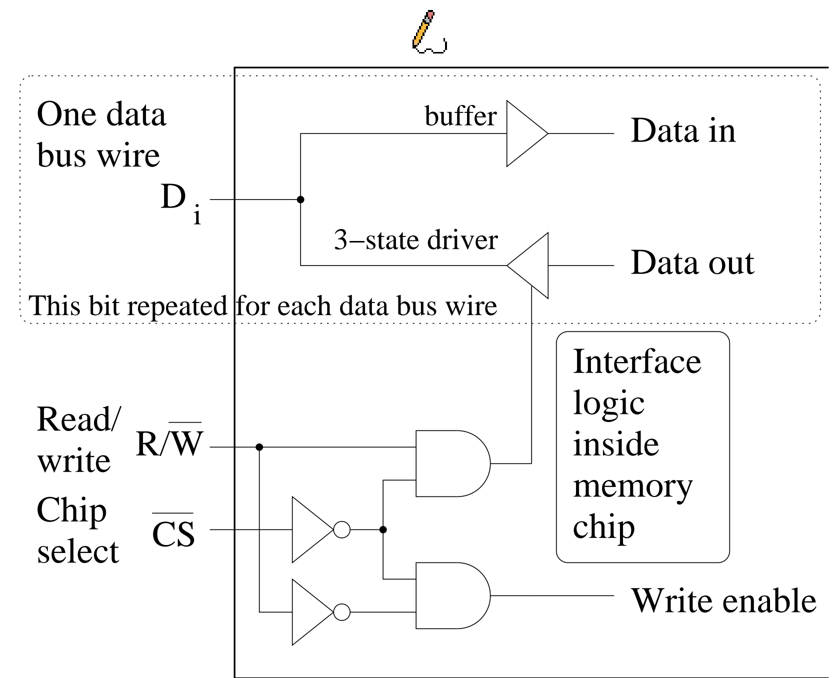
This is a 'one out of $2^n$ decoder' where $n = 2$. Only one output is high at a time.



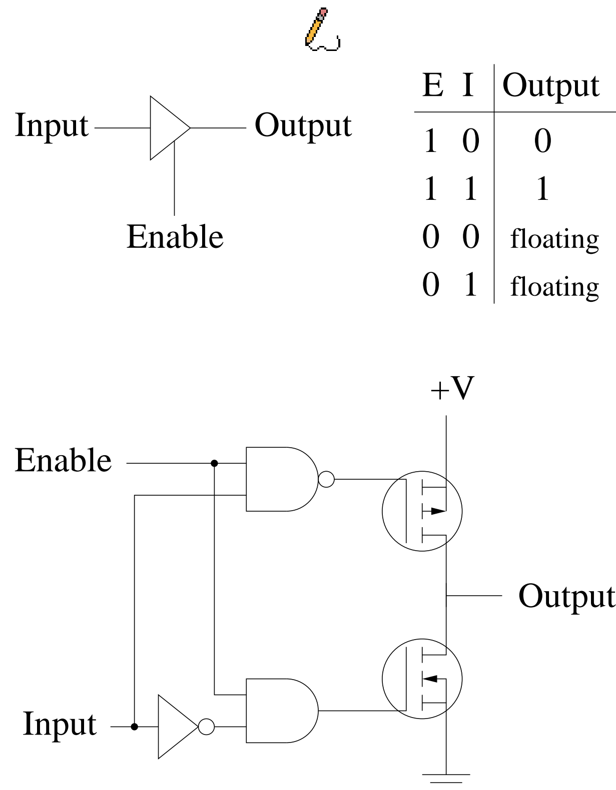

The word select lines are driven from the one-out-of-$2^n$ decoder. Thus, only one word select line is driven high at any time.



This shows how the internal signals *data in*, *data out*, and *write enable* are connected to the external *data bus* wires, the *read/write* wire and *chip select*.



# Tri-state Buffer




| E | I | Output |
|---|---|---|
| 1 | 0 | 0 |
| 1 | 1 | 1 |
| 0 | 0 | floating |
| 0 | 1 | floating |

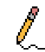
When *enable* is low neither transistor is on and the output is not connected to any voltage by this part of the circuit.



# Two Memory Chips

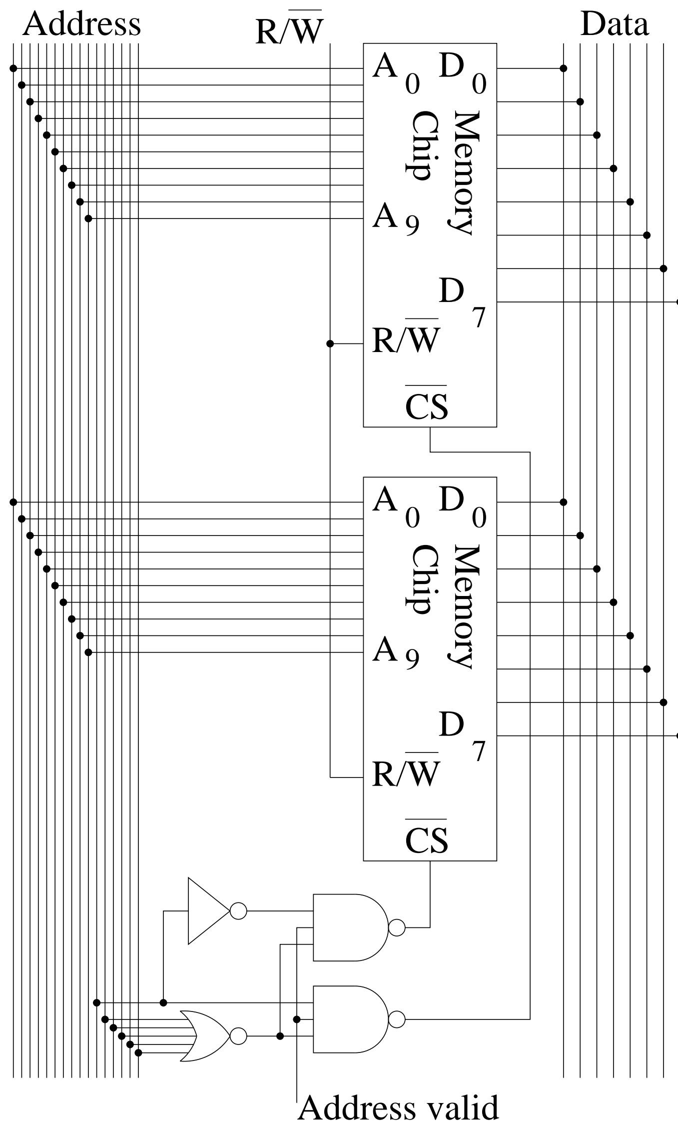
One memory chip does not usually provide enough storage for the microprocessor on its own.

A memory chip with 10 address wires contains $2^{10}$ locations (each of which contains a byte if there are 8 data lines). The microprocessor has 16 address wires in total so it can address up to $2^{16}$ locations. To completely fill the microprocessor address space with these memory chips you would therefore need



$$\frac{2^{16}}{2^{10}} = 2^6 = 64 \text{ chips.}$$

The address space of the 6800 is from $0000_H$ to $\mathrm{FFFF}_H$. A single memory chip contains $2^{10} = 400_H$ locations. We want to wire one memory chip so it responds to addresses $0000_H$ – $03\mathrm{FF}_H$ and another so it responds to addresses $0400_H$ – $07\mathrm{FF}_H$.






# Other Types of Memory

**RAM** Random access memory. The type of memory we have been talking about so far.

**ROM** Read only memory. Contents set in the factory and can never be changed by the microprocessor.

**EPROM** Erasable programmable read only memory. Can be programmed using a special programmer device that uses higher voltages than in a normal microprocessor circuit. Cannot be changed by the microprocessor. Can be erased using UV light through a little window on the top, or sometimes by applying a higher voltage again.

More long term storage is achieved through the use of magnetic disks, optical CDROMs, and tape drives.

# Memory-Mapped i/o

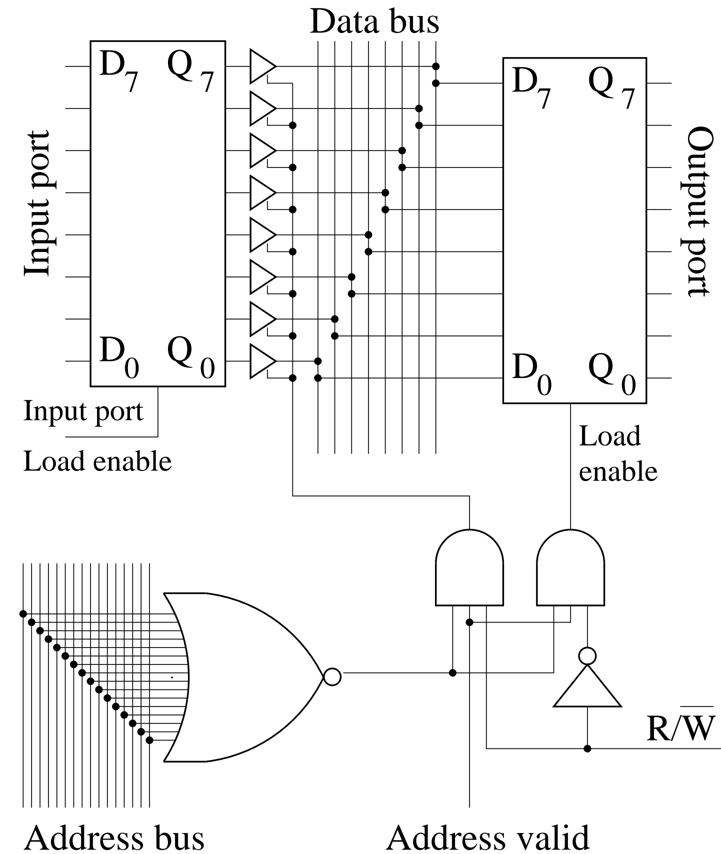
We want to create input and output ports that each appear to the microprocessor just like a location in memory.

This would enable us to arrange that every byte read from the input port was a character read from the keyboard, and every byte written appeared as a character on the screen.

The diagram shows an input port and an output port both mapped to memory location $0000_H$. Reading from location $0000_H$ reads from the input port. Writing to location $0000_H$ writes to the output port.

(Thus if the microprocessor writes a number to location $0000_H$ and then reads location $0000_H$ back it will not necessarily get the same number!)





# Address Decoding Examples

Q: How may address pins are there on an 8 kilobinary byte RAM chip that has 8 data lines.

A: 8 kilobinary bytes = $2^3 \times 2^{10} = 2^{13}$ bytes. So there will be 13 address lines ($A_0$–$A_{12}$).

Q: A 6800 microprocessor has 16 address lines so it can address a maximum of 64 kilobinary bytes of memory. Suppose 8 memory chips are wired up to completely fill this address space, work out the address ranges of each chip.

A: The 13 least significant address line will determine the address inside each chip. The 3 most significant address lines will determine which chip. The address ranges (in Hex) will therefore be:

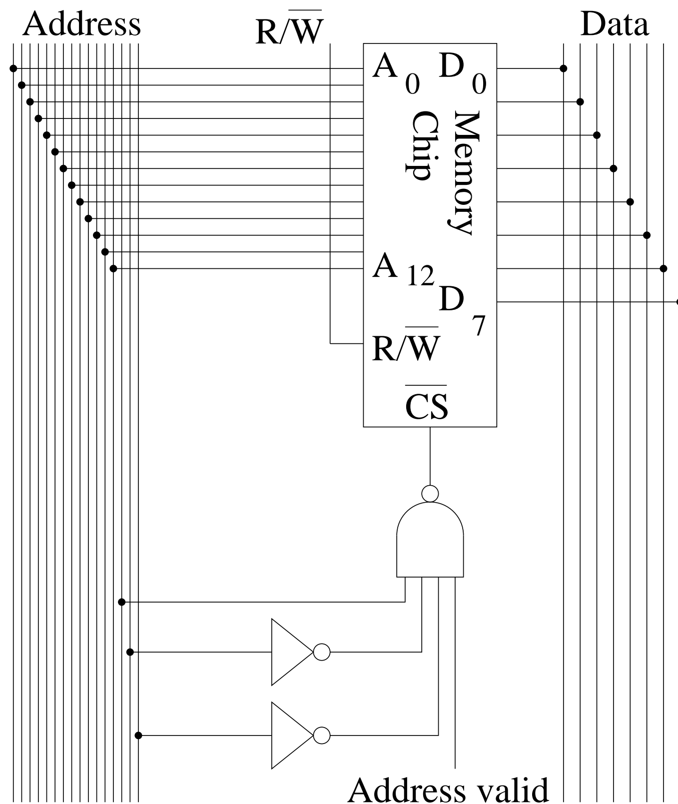
| Chip | Min | Max | Chip | Min | Max |
|---|---|---|---|---|---|
| 1 | 0000 | 1FFF | 5 | 8000 | 9FFF |
| 2 | 2000 | 3FFF | 6 | A000 | BFFF |
| 3 | 4000 | 5FFF | 7 | C000 | DFFF |
| 4 | 6000 | 7FFF | 8 | E000 | FFFF |



| Which chip | | | Which location inside the chip | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A_{15}$ | $A_{14}$ | $A_{13}$ | $A_{12}$ | $A_{11}$ | $A_{10}$ | $A_9$ | $A_8$ | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | First Chip |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2nd Chip |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3rd Chip |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4th Chip |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5th Chip |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6th Chip |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7th Chip |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8th Chip |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

Q: Design the address decoding logic for the second of these chips.

A: We want to chip to respond when the top two address lines ($A_{15}$ and $A_{14}$) are low, and $A_{13}$ is high.






Q: Why would it it be difficult to use the 6800 if its address space was entirely populated with RAM chips as discussed above?

A: There would be no addresses left for memory mapped i/o. Therefore it would be difficult to attach any peripheral devices to the microprocessor. (eg. keyboard, screen, disk etc).

As there is no ROM or EPROM the start-up programs would somehow have to be loaded afresh each time the power was switched on.

# Handout 3 Section C

# Microprocessor Programming

In this section we introduce the technique of programming the 6800 microprocessor in assembly language. The data movement, arithmetic and logic instructions are summarised.

Further information about various aspects of assembly language programming are given in later sections of this handout.

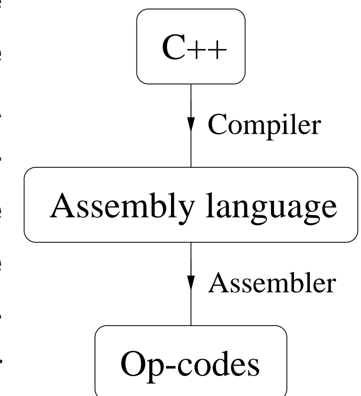


# Microprocessor Programming

It would be very difficult to program a microprocessor using just the hexadecimal op-codes.

To make programming easier we use a software tool called an assembler that translates an *assembly language* program into appropriate op-codes.

In other work, for example the Integrated Design Project in the second year, you will use a compiler that will translate C++ into assembly language. The assembly language can then be assembled to produce the op-codes that form the program for the microprocessor inside the robot vehicle.

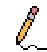


Features of assembly language programming:

- Assembly language programs are usually efficient and take up less space in memory than programs written in higher level languages.
- There is one assembly instruction for each op-code so the programmer has direct control over the microprocessor.
- It is not complicated for the assembler to translate assembly language into op-codes and this process can happen rapidly on most computers.
- It is difficult to write large programs in assembly language.



## Writing Numbers



$ means hexadecimal.

# means immediate (take this number itself rather than the data you find at this address).

| | |
|---|---|
| LDAA $FFF3 | load acc A with the data stored in memory at hex address $FFF3 |
| LDAA #24 | load acc A with 24 (decimal) |
| LDAA #$10 | load acc A with 16 (decimal) (i.e. $10 hex) |

# Example: Addition

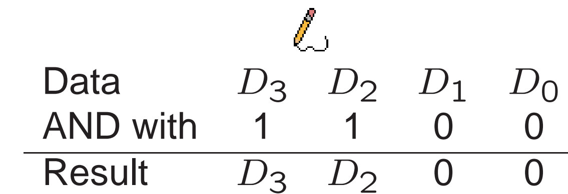
Get the bytes of data from locations \$C000 and \$C001.
Add them together and store the sum at location \$C002.

| | |
|---|---|
| LDAA \$C000 | load contents of \$C000 into A |
| LDAB \$C001 | load contents of \$C001 into B |
| ABA | A + B $\rightarrow$ A |
| STAA \$C002 | store data in A into location \$C002 |

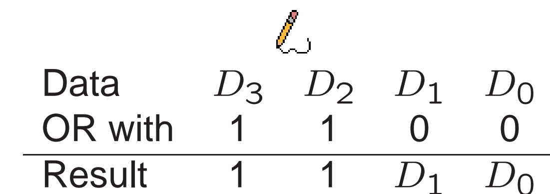


# AND & OR Operations

The logical AND and OR operations work bitwise.



| Data | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|
| AND with | 1 | 1 | 0 | 0 |
| Result | $D_3$ | $D_2$ | 0 | 0 |



| Data | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|
| OR with | 1 | 1 | 0 | 0 |
| Result | 1 | 1 | $D_1$ | $D_0$ |

## Example: AND

To round the number in location $C000 down to the nearest even number below:

Get the byte of data from location $C000. Perform a logical AND operation between the bits in this number and the corresponding bits in the number $FE. Store the result back in location $C000.

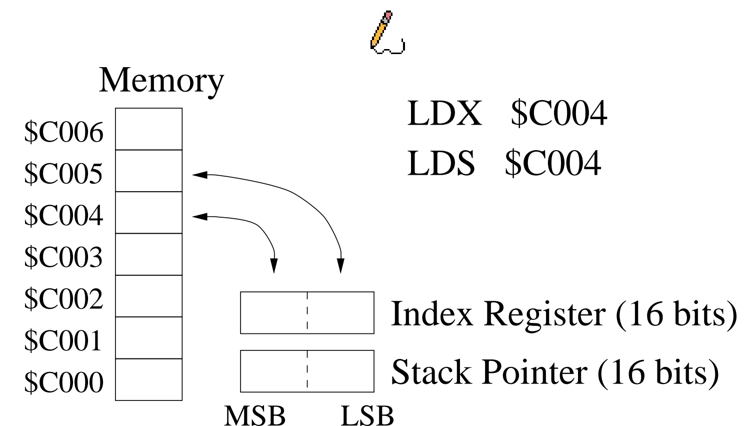
| | |
|---|---|
| LDAA $C000 | load contents of $C000 into A |
| ANDA #$FE | AND each bit of A with the corresponding bit of $FE Leave the answer in A |
| STAA $C000 | store A in location $C000 |



## Data Movement Instructions

| | |
|---|---|
| A | Accumulator A (8 bit) |
| B | Accumulator B (8 bit) |
| X | Index Register (16 bit) |
| S | Stack Pointer (16 bit) |

| | |
|---|---|
| Load | LDAA, LDAB, LDX, LDS |
| Store | STAA, STAB, STX, STS |
| Transfer | TAB, TBA, TXS, TSX |



## Arithmetic & Logic Instructions

| | |
|---|---|
| Add | ADDA, ADDB, ABA, ADCA, ADCB |
| Subtract | SUBA, SUBB, SBA, SBCA, SBCB |
| Clear | CLRA, CLRB, CLR |
| Complement | COMA, COMB, COM |
| Negate | NEGA, NEGB, NEG |
| Decrement | DECA, DECB, DEC, DEX, DES |
| Increment | INCA, INCB, INC, INX, INS |
| And | ANDA, ANDB |
| Exclusive-or | EORA, EORB |
| Or | ORAA, ORAB |



## 6800 Registers

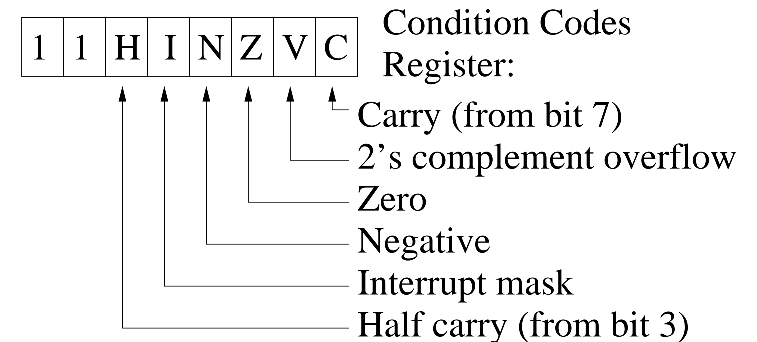
Accumulator A (8 bit)

Accumulator B (8 bit)

Index Register X (16 bit)

Stack Pointer S (16 bit)

Program Counter PC (16 bit)



The **accumulators** are used to hold intermediate arithmetic results.

The **index register** is used to hold the address of data in memory.

The **stack pointer** is used to manage temporary storage, and in subroutines.

The **program counter** contains the address of the next instruction to be executed.

The **condition codes register** holds various information about the last arithmetic operation that is used to determine whether a conditional branch instruction results in a branch or not.



# Handout 3 Section D

## Flags

In this section we individually introduce each of the flags in the condition codes register and illustrate its use with a short example program.

The branch, rotate and shift instructions that make use of the flags are discussed.

An example program to multiply two unsigned numbers is described and flow charts are introduced to illustrate the algorithm used.

## Carry Flag

The $C$ flag is set when there is a carry from the most significant bit during a calculation.

For example, if we have an addition:

$$A \text{ plus } B \rightarrow A^{\text{after}}$$

then the carry flag will be determined by the Boolean expression

$$C = A_7.B_7 + (A_7 + B_7).\overline{A_7^{\text{after}}}$$



## Example: Adding 16-bit numbers

Add two 2's complement 16 bit numbers. The first one is stored \$C001 (low byte) and \$C000 (high byte). The second number is stored \$C003 (low byte) and \$C002 (high byte). The answer should be put in \$C005 (low byte) and \$C004 (high byte).

```
LDAA $C001   load contents of $C001 into A
ADDA $C003   A + contents of $C003 → A
             if this addition carrys beyond 8 bits
             the carry flag will be set
STAA $C005   store A at location $C005 (low byte ans)

LDAA $C000   load contents of $C000 into A
ADCA $C002   A + C + contents of $C002 → A
STAA $C004   store A at location $C004 (hi byte ans)
```

## 2's Complement Overflow Flag

The $V$ flag is set whenever an overflow occurs based on the numbers being treated as signed rather than unsigned numbers.

If we have an addition:

$$A \text{ plus } B \rightarrow A^{\text{after}}$$

then the $V$ flag will be determined by the Boolean expression

$$V = A_7.B_7.\overline{A_7^{\text{after}}} + \overline{A_7}.\overline{B_7}.A_7^{\text{after}}$$

$V$ is set when the carry from bit 6 is different to the carry from bit 7.

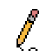
$(A_7.B_7.\overline{A_7^{\text{after}}})$ is the case of a carry from bit 7 but no carry from bit 6. $(\overline{A_7}.\overline{B_7}.A_7^{\text{after}})$ is the case of a carry from bit 6 but no carry from bit 7.



## Checking the V flag

Add the 2's complement numbers in $C000 and $C001 storing the answer in $C002. If 2's complement overflow occurs store $FF in $C003, otherwise store $00 in $C003.

```
       CLR $C003      store $00 in $C003
       LDAA $C000     load contents of $C000 into A
       ADDA $C001     A + contents of $C001 → A
                      sets V flag if overflow occurs
       BVC voff       if V is clear branch to label 'voff'
       COM $C003      complement $C003
voff:  STAA $C002     store A in $C002
```



The BVC is a conditional branch instruction. The branch only occurs if the $V$ flag is clear.

# Zero Flag

The $Z$ flag is set when the answer is zero.

If we have an addition:

$$A \text{ plus } B \rightarrow A^{\text{after}}$$

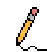
then the $Z$ flag will be set if and only if $A^{\text{after}} = 0$.



# Checking the Z flag

Subtract the numbers in $C001 from the number in $C000 storing the answer in $C002. If the numbers in $C000 and $C001 are equal (and hence the answer is zero), store $FF in $C003, otherwise store $00 in $C003.

```
       CLR $C003     store $00 in $C003
       LDAA $C000    load contents of $C000 into A
       SUBA $C001    A − contents of $C001 → A
                     sets Z flag if answer is zero
       BNE zoff      if Z is clear branch to label ‘zoff’
       COM $C003     complement $C003
zoff:  STAA $C002    store A in $C002
```



Branch if not equal, BNE, is a conditional branch instruction. The branch only occurs if the $Z$ flag is clear.

# The Negative Flag

The $N$ flag is set whenever the top bit (bit 7) of the answer is set. It will thus be set for all numbers which would be interpreted as negative in two's complement arithmetic.

If we have an addition:

$$A \text{ plus } B \rightarrow A^{\text{after}}$$

then the $N$ flag will be determined by the top bit of $A^{\text{after}}$

$$N = A_7^{\text{after}}$$

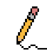
If we choose to regard the numbers as unsigned then the $N$ flag will be set whenever the answer is in the range 128 to 255.



# Checking the N flag

Add the 2's complement numbers in $C000 and $C001 and find the magnitude of the sum. Store this as a positive 2's complement number in $C002.

```
       LDAA $C000   load contents of $C000 into A
       ADDA $C001   A + contents of $C001 → A
                    sets N flag if answer < 0
       BPL noff     if N is clear branch to label 'noff'
       NEGA         negate A
noff:  STAA $C002   store A in $C002
```



Branch if plus, BPL, is a conditional branch instruction. The branch only occurs if the $N$ flag is clear.

# The Half Carry Flag

The $H$ flag is set whenever there is carry from bit 3 to bit 4 in the addition process.

For example, if we have an addition:

$$A \text{ plus } B \rightarrow A^{\text{after}}$$

then the carry flag will be determined by the Boolean expression

$$H = A_3.B_3 + (A_3 + B_3).\overline{A_3^{\text{after}}}$$

A byte is made up of 2 nibbles. Each nibble has 4 bits. The $H$ flag is set whenever there is a carry from the lower nibble to the upper nibble.



# Setting Flags

In the data book a large table shows which flags are affected by each instruction.

| | | | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|---|
| Add | ADDA | A + M → A | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| | ADDB | B + M → B | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| | ABA | A + B → A | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| Clear | CLR | 00 → M | ● | ● | R | S | R | R |
| | CLRA | 00 → A | ● | ● | R | S | R | R |
| | CLRB | 00 → B | ● | ● | R | S | R | R |

↕ Test and set if true, cleared otherwise.

● Not affected.

R Reset always.

S Set always.

## Program error while attempting to check the V flag

Add the 2's complement numbers in $C000 and $C001 storing the answer in $C002. If 2's complement overflow occurs store $FF in $C003, otherwise store $00 in $C003.

Why won't this program work?

| | | |
|---|---|---|
| | LDAA $C000 | load contents of $C000 into A |
| | ADDA $C001 | A + contents of $C001 $\rightarrow$ A |
| | | sets $V$ flag if overflow occurs |
| | CLR $C003 | store $00 in $C003 (resets $V$ flag) |
| | BVC voff | if $V$ is clear branch to label 'voff' |
| | COM $C003 | complement $C003 |
| voff: | STAA $C002 | store A in $C002 |

The $V$ flag is always reset by the CLR instruction therefore the BVC branch will no longer be controlled by the way the ADDA instruction sets the $V$ flag.



## Testing Flags

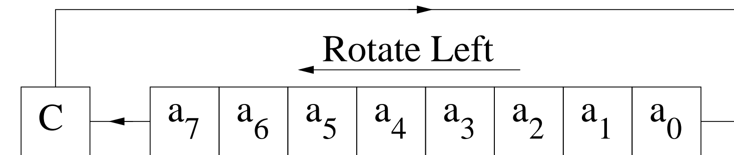
| **Operation** | | **Branch test** |
|---|---|---|
| Branch always | BRA | none |
| Branch if carry clear | BCC | $C = 0$ |
| Branch if carry set | BCS | $C = 1$ |
| Branch if = zero | BEQ | $Z = 1$ |
| Branch if $\geq$ zero | BGE | $N \oplus V = 0$ |
| Branch if $>$ zero | BGT | $Z + (N \oplus V) = 0$ |
| Branch if higher | BHI | $C + Z = 0$ |
| Branch if $\leq$ zero | BLE | $Z + (N \oplus V) = 1$ |
| Branch if lower or same | BLS | $C + Z = 1$ |
| Branch if $<$ zero | BLT | $N \oplus V = 1$ |
| Branch if minus | BMI | $N = 1$ |
| Branch if $\neq$ zero | BNE | $Z = 0$ |
| Branch if overflow clear | BVC | $V = 0$ |
| Branch if overflow set | BVS | $V = 1$ |
| Branch if plus | BPL | $N = 0$ |

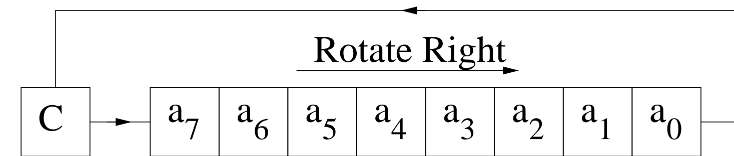
# Rotate and Shift Instructions

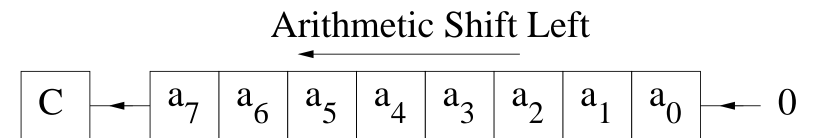
The rotate and shift instructions also affect the carry flag.

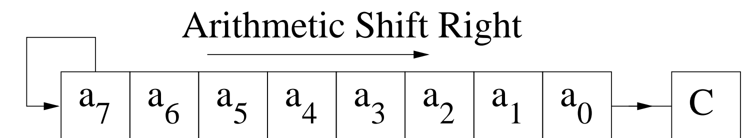
| | |
|---|---|
| Rotate left | ROLA, ROLB, ROL |
| Rotate right | RORA, RORB, ROR |
| Shift left | ASLA, ASLB, ASL |
| Shift right | ASRA, ASRB, ASR, LSRA, LSRB, LSR |

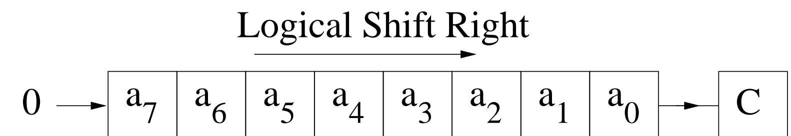




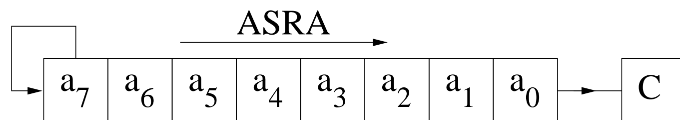



# Example: Average

Get the bytes of data from locations $C000 and $C001. Add them together and shift the result one bit to the right to divide it by two. Store the average at location $C002.

| | |
|---|---|
| LDAA $C000 | load contents of $C000 into A |
| ADDA $C001 | A + contents of $C001 → A |
| ASRA | arithmetic shift right A |
| STAA $C002 | store A in location $C002 |



The numbers are treated as signed 2's complement as the top bit is duplicated in the arithmetic shift right.



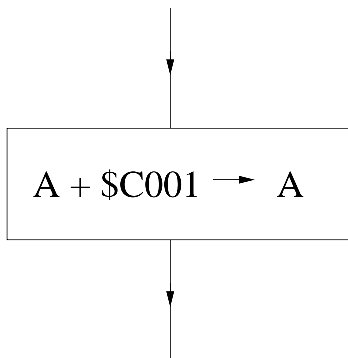
# Flow Charts

Advantages:

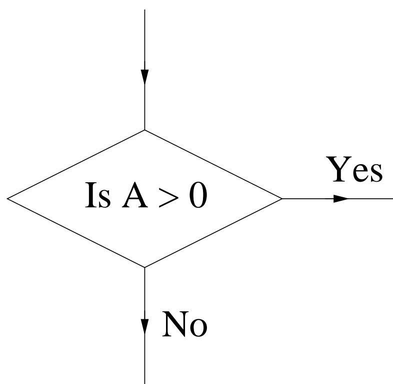
- Useful to enable you to design the overall algorithm without worrying about details of microprocessor instructions.
- Independent of microprocessor type.
- Easier to read than machine code.

Limitations:

- Modern programming techniques such as structured programming in high level languages and object oriented programming are much quicker to use than flow charts.
- Where machine code programming is still used (eg. for optimum efficiency in digital signal processing procedures) the precise way the microprocessor instructions are used is very important and so flow charts are of limited use.




Actions are represented as boxes.



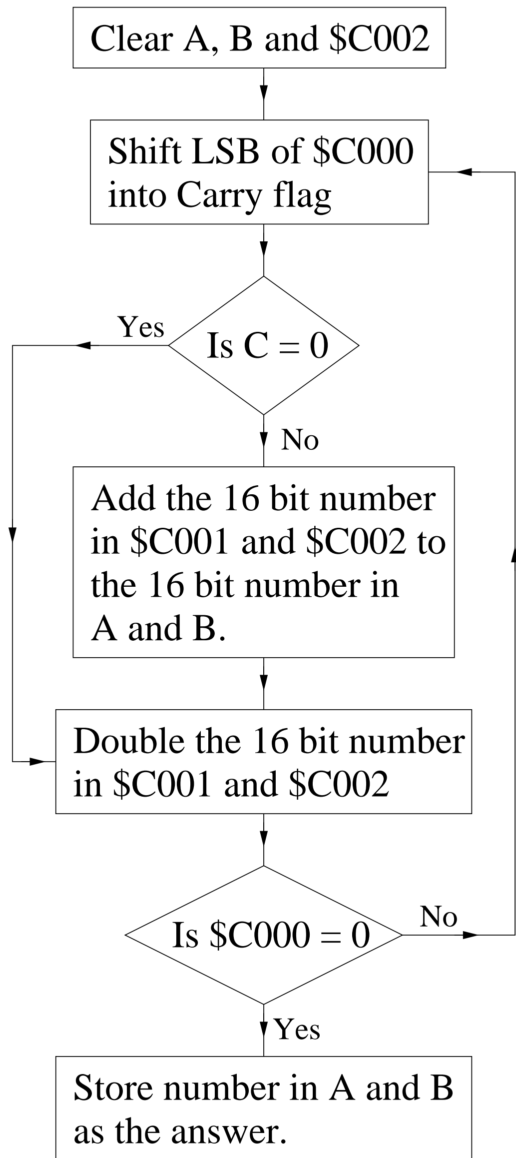

Tests (comparisions or questions) are represented as diamonds



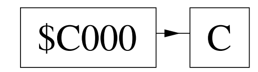
## Unsigned Multiplication

Multiply the unsigned numbers stored in locations $C000 and $C001. Leave the more significant byte of the answer in $C002 and the less significant byte of the answer in $C003.

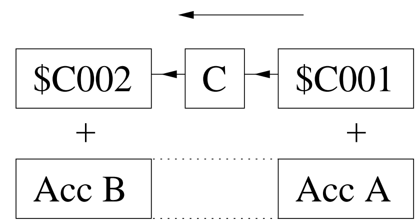
| $C000 | × | $C001 | = | $C002 | $C003 |
|---|---|---|---|---|---|



BCC used to test each bit in turn.




| $C002 | $C001 | $C000 | C | |
|---|---|---|---|---|
| | | 0 0 0 0 0 1 1 0 | | |
| 0 0 0 0 0 0 0 0 | 0 0 0 0 1 0 1 0 | 0 0 0 0 0 0 1 1 | 0 | Skip |
| 0 0 0 0 0 0 0 0 | 0 0 0 1 0 1 0 0 | 0 0 0 0 0 0 0 1 | 1 | Add |
| 0 0 0 0 0 0 0 0 | 0 0 1 0 1 0 0 0 | 0 0 0 0 0 0 0 0 | 1 | Add |
| 0 0 0 0 0 0 0 0 | 0 1 0 1 0 0 0 0 | | | |




Answer accumulated in A & B registers.

10 x 6 = 20 + 40 = 60

## Unsigned Multiplication

Multiply the unsigned numbers stored in locations \$C000 and \$C001. Leave the more significant byte of the answer in \$C002 and the less significant byte of the answer in \$C003.

```
        CLR $C002    Clear to use for scratch storage
        CLRA         initialise accumulator A to zero
        CLRB         initialise accumulator B to zero
lsl:    LSR $C000    shift bottom bit of $C000 into C
        BCC dad      if this bit is 0 jump to 'dad'
        ADDA $C001   add LSB of $C001 to A
        ADCB $C002   add MSB (ie $C002) to B
dad:    ASL $C001    double $C001 & top bit to carry
        ROL $C002    carry to lowest bit & double rest
        TST $C000    Set zero flag based on $C000
        BNE lsl      if non-zero branch back to lsl
        STAB $C002   store MSB of answer in $C002
        STAA $C003   store LSB of answer in $C003
```



## Handout 3 Section E

## Addressing Modes

In this section we describe each of the six 6800 addressing modes. Many of these have been used already in previous examples. Here we show how they work and explain the way that the op-codes of each instruction in each mode are presented in the microprocessor data book.

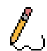
The technique for calculating branch offsets in relative addressing is described.

We then show how the run time of an assembly language program can be calculated.

## Addressing Modes

There are 6 different ways in which the data can be specified when the assembler mnemonics are assembled into op-codes.

| | |
|---|---|
| Extended | Full 16 bit address after op-code |
| Implied | No data needs to be specified |
| Immediate | Data itself (1 byte) comes after op-code |
| Direct | 8 bit address after op-code (top 8 bits = 00) |
| Indexed | Address in X reg + 8 bit unsigned offset |
| Relative | 8 bit signed offset used for branches |



A list of which addressing modes are available for each assembler instruction is given in the Microprocessor Data Book.



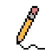
## Extended Addressing

We saw earlier that B6 was a hexadecimal op-code for the LDAA microprocessor instruction. So

LDAA $00F8

is assembled as

B6 00 F8



The full address of the data is given after the op-code.

All op-codes that use extended addressing take up 3 bytes of memory. One byte for the op-code and two bytes for the address.

## Implied Addressing

In this case no address is required. For example:

| Mnemonic | Op-code | Operation |
|---|---|---|
| ABA | 1B | $A + B \rightarrow A$ |
| CLRA | 4F | $0 \rightarrow A$ |
| CLRB | 5F | $0 \rightarrow B$ |
| COMA | 43 | $\overline{A} \rightarrow A$ |
| COMB | 53 | $\overline{B} \rightarrow B$ |
| NEGA | 40 | $0 - A \rightarrow A$ |
| NEGB | 50 | $0 - B \rightarrow B$ |
| DECA | 4A | $A - 1 \rightarrow A$ |
| DECB | 5A | $B - 1 \rightarrow B$ |
| INCA | 4C | $A + 1 \rightarrow A$ |
| INCB | 5C | $B + 1 \rightarrow B$ |
| SBA | 10 | $A - B \rightarrow A$ |

Op-codes that use implied addressing only take up a single byte of program memory.



## Immediate Addressing

In immediate addressing the 8-bit data is included in the program memory immediately after the op-code. We indicate that immediate addressing is to be used with the # symbol just before the data. For example, to round the number in location $C000 down to the nearest even number below:

| | |
|---|---|
| LDAA $C000 | load contents of $C000 into A |
| **ANDA #$FE** | AND each bit of A with the corresponding bit of $FE Leave the answer in A |
| STAA $C000 | store A in location $C000 |

Note that #$FE is not an address. It is the actual number that is to be used in the program. Op-codes that use immediate addressing usually take up 2 bytes of program memory. One byte for the op-code and one for the data. (Note: there are also some immediate addressing instructions that take up 3 bytes because they involve 16-bit registers.)

We use immediate addressing when we wish to introduce constants into the program.

# Direct Addressing

Direct addressing provides a quick way of addressing the bottom 256 locations in memory. In direct addressing you only provided an 8 bit address and the top 8 bits of the address are assumed to be zero.

Direct addressing is like extended addressing, except that it only works for the bottom 256 bytes of memory.

| Assembler | Op-codes | Addressing Mode |
|---|---|---|
| LDAA $00F8 | B6 00 F8 | Extended |
| LDAA $F8 | 96 F8 | Direct |

Op-codes that use direct addressing only use up 2 bytes of program memory. They are also faster to execute that op-codes that use extended addressing.

You indicate that you want to use direct addressing by only providing an 8 bit number as the address.

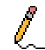


# Indexed Addressing

In indexed addressing the address of the data is calculated by taking the 16 bit number in the index register (X) and adding a single byte unsigned offset.

You need to make sure that the X register contains the right value (to use as the address) before you use indexed addressing.

| Assembler | Op-codes | Addressing Mode |
|---|---|---|
| LDX #$2A00 | CE 2A 00 | Immediate |
| LDAA $03,X | A6 03 | Indexed |



Together, these instructions will load accumulator A with the data from memory at address $2A03.

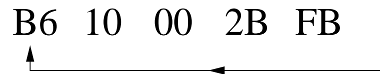
Op-codes that use indexed addressing take up 2 bytes of program memory. One byte for the op-code and one for the offset.

# Relative Addressing

Relative addressing is only used by branch instructions (BRA, BNE etc.). For example, suppose that memory location $1000 is an input port and we wish to keep loading the number from it until we get a positive value.

| **Assembler** | | **Op-codes** | **Addr. Mode** |
|---|---|---|---|
| lab: | LDAA $1000 | B6 10 00 | Extended |
| | BMI lab | 2B FB | Relative |

Branch instructions jump relative to the address of the op-code that would have been executed if the branch were not taken. In the case above we want to jump back 5 bytes so the offset is $-5$, or FB in 2's complement hex.




As the branch instruction always takes up two bytes of program memory you can also think of it as:

New address $=$ Addr. of branch op-code $+2+$ Offset



# Microprocessor Data Book

In the microprocessor data book there is a table that lists the op-code for each instruction in each available addressing mode. The list also contains the number of clock cycles it takes for the microprocessor to execute the instruction (~), and the number of bytes of program memory that the instruction takes up (#).

| | | IMMED | | | DIRECT | | | INDEX | | | EXTND | | | IMPLIED | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | OP | ~ | # | OP | ~ | # | OP | ~ | # | OP | ~ | # | OP | ~ | # |
| Add | ADDA | 8B | 2 | 2 | 9B | 3 | 2 | AB | 5 | 2 | BB | 4 | 3 | | | |
| | ADDB | CB | 2 | 2 | DB | 3 | 2 | EB | 5 | 2 | FB | 4 | 3 | | | |
| | ABA | | | | | | | | | | | | | 1B | 2 | 1 |

If you know the clock frequency then you can use this table to work how long it will take a program to execute. The clock frequency is often 1 MHz so each cycle takes 1 $\mu$s.

# Example 1

Write a program to add up the 2 numbers in memory locations \$C001 and \$C002, putting the sum in \$C000. Work out how long the program will take to run if the clock frequency is 1 MHz.

| **Instructions** | **Op-codes** | **Cycles** $\sim$ |
|---|---|---|
| LDAA \$C001 | B6 C0 01 | 4 |
| ADDA \$C002 | BB C0 02 | 4 |
| STAA \$C000 | B7 C0 00 | 5 |

So 13 CPU cycles are used and the program takes 13 $\mu$s to run.

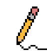


# Example 2

Write a program to add up the 8 numbers in memory locations \$C001 to \$C008, putting the answer in \$C000. Work out how long the program will take to run if the clock frequency is 1 MHz.

| **Instructions** | | **Op-codes** | **Cycles** $\sim$ |
|---|---|---|---|
| | LDX #\$C001 | CE C0 01 | 3 |
| | CLRA | 4F | 2 |
| lad: | ADDA 0,X | AB 00 | 5 |
| | INX | 08 | 4 |
| | CPX #\$C009 | 8C C0 09 | 3 |
| | BNE lad | 26 F8 | 4 |
| | STAA \$C000 | B7 C0 00 | 5 |

Note that the branch offset is $-8$ (or F8 in 2's complement).



Before loop $=$ 5 cycles. During loop $=$ 16 cycles. After loop $=$ 5 cycles. So we have $10+8\times16=138$ cycles in total which takes 138 $\mu$s.

# Example 3

One period of a sine wave is stored as byte data in addresses $E000 to $E0FF. Write a program to output all the values repeatedly to a D-to-A converter at address $FF00. If the microprocessor has a clock frequency of 1 MHz what will be the frequency of the sine wave?

| Instructions | | Op-codes | Cycles $\sim$ |
|---|---|---|---|
| outl: | LDX #$E000 | CE E0 00 | 3 |
| inl: | LDAA 0,X | A6 00 | 5 |
| | STAA $FF00 | B7 FF 00 | 5 |
| | CPX #$E0FF | 8C E0 FF | 3 |
| | BEQ outl | 27 F3 | 4 |
| | INX | 08 | 4 |
| | BRA inl | 20 F3 | 4 |

One number output, (not end of period) takes 25 cycles.
One number output at end of period takes 20 cycles.
Period takes $255 \times 25 + 20 = 6395$ cycles.
Frequency of sine wave is $\frac{1000000}{6395} = 156$ Hz.



# The JMP Instruction

There is a limit to how far you can jump with a BRA instruction. You can only jump $127+2$ bytes forwards from the op-code of the BRA instruction or $128 - 2$ bytes backwards from it. To jump further you have to use the JMP instruction.

JMP can take either the extended or the indexed addressing mode.

| Instructions | Op-codes | Effect |
|---|---|---|
| JMP outl | 7E 80 00 | Jump to address $8000 |
| JMP $05,X | 6E 05 | Jump to address $X + 5$ |

Note that the JMP instruction is unconditional. You cannot do JMP conditional on a test.

# Handout 3 Section F

# The Stack & Subroutines

In this section we describe the operation of the stack and show how this is used in the JSR and RTS instructions for subroutines.

The handout ends with a brief description of the operation of the microprocessor reset pin.

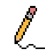


# The Stack

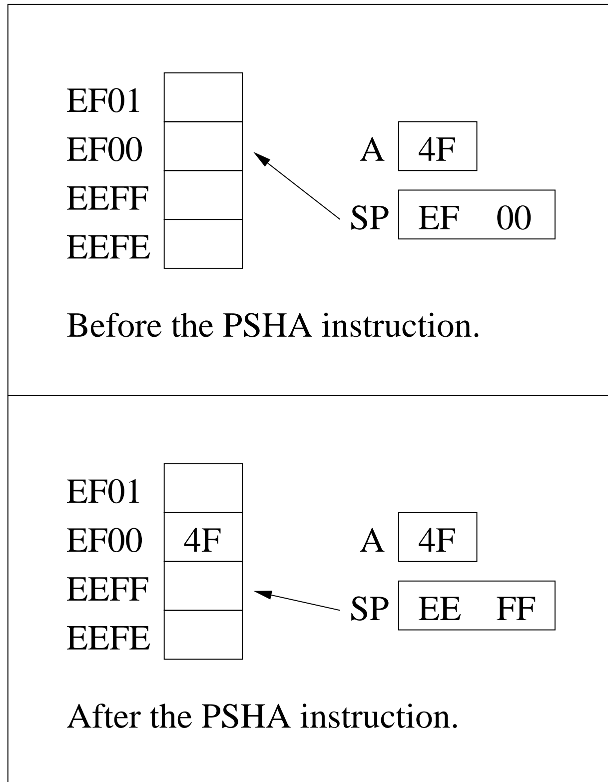
The stack is an area of RAM that is allocated by the user, and used automatically to provide temporary storage. The stack is a last-in-first-out store.

The stack pointer (SP) is a 16 bit register that always points to next empty stack location available. In the 6800 the stack grows *downwards* in memory.

| | |
|---|---|
| LDS #$EF00 | \$EF $\rightarrow \mathrm{SP}_{hi}$, \$00 $\rightarrow \mathrm{SP}_{lo}$ |
| PSHA | $A \rightarrow M_{\mathrm{SP}}$, $\mathrm{SP} - 1 \rightarrow \mathrm{SP}$ |
| PULA | $\mathrm{SP} + 1 \rightarrow \mathrm{SP}$, $M_{\mathrm{SP}} \rightarrow A$ |



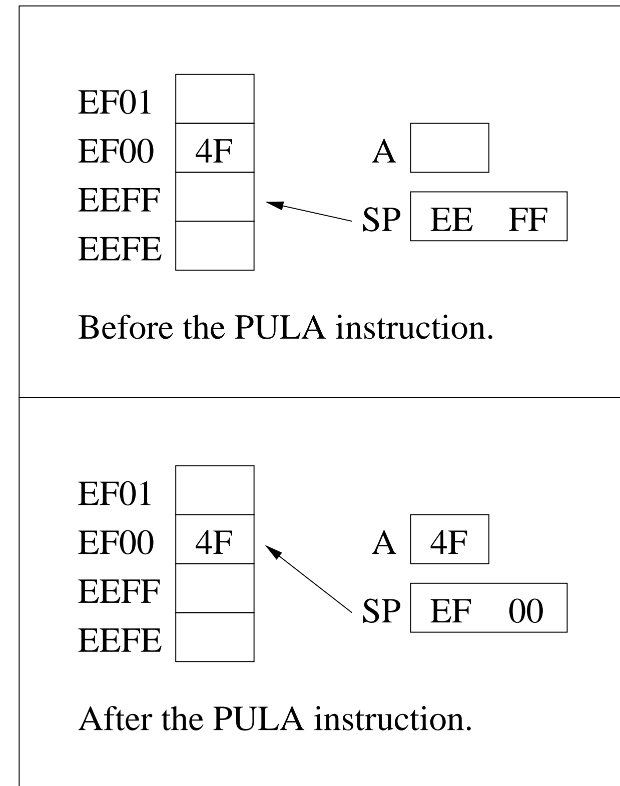
Think of the stack as like the automatic plate warming devices where each time you lift a plate off the top, the other plates are pulled up a bit by springs so that the next plate is available.




Before the PSHA instruction.

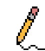
After the PSHA instruction.






Before the PULA instruction.

After the PULA instruction.



Note that the $4F in address $EF00 will now be over-written the next time a value is pushed onto the stack.

## Push & Pull Example

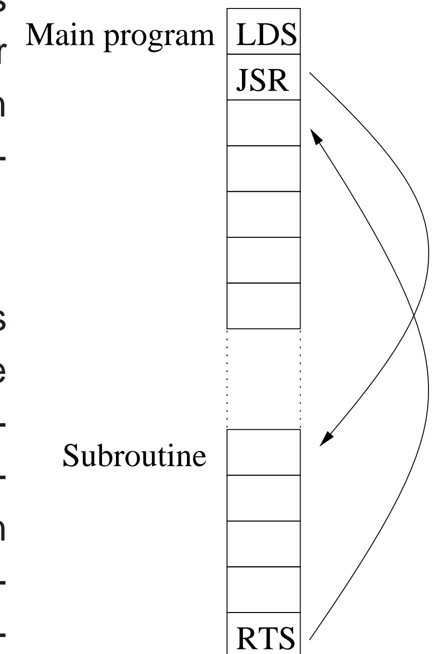
Swap the numbers in accumulators A and B.

| | |
|---|---|
| LDS #$EF00 | Init stack to high address in RAM |
| PSHA | push A onto stack |
| PSHB | push B onto stack |
| PULA | pull old contents of B into A |
| PULB | pull old contents of A into B |



## Subroutines

The JSR instruction pushes the (16 bit) program counter onto the stack and then jumps to the address specified.



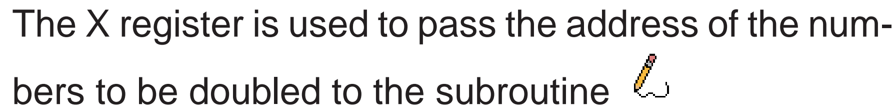

The RTS instruction pulls a 16 bit address off the stack, putting it in the program counter. It thus continues program execution from where it left off, immediately after the JSR instruction.

Together these instructions enable us to implement subroutines (subroutines are called ‘procedures’ in high level language programming).

## Subroutine Example

Write a subroutine to double the unsigned 16 bit number stored at $C000 (high byte) and $C001 (low byte). Use the same subroutine to double the 16 bit number stored at $A000 (high byte) and $A001 (low byte).

The X register is used to pass the address of the numbers to be doubled to the subroutine



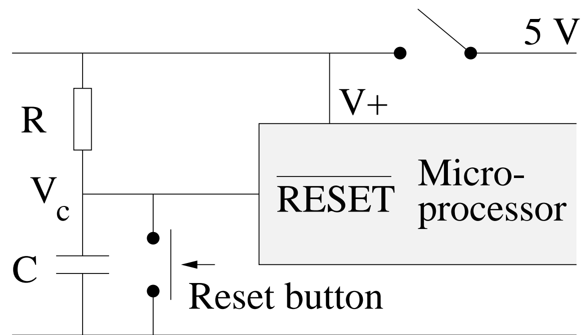
```
main:  LDS #$EF00   Init stack
       LDX #$C000   Load X with $C000
       JSR dub      Call dub subroutine
       LDX #$A000   Load X with $A000
       JSR dub      Call dub subroutine
       JMP rest     Jump past the subroutine

dub:   ASL 1,X      Double LS byte, top bit → C
       ROL 0,X      Double MS byte, C → bottom bit
       RTS          Return from subroutine

rest:  . . .        The rest of the program
```



## The Reset Pin

The microprocessor has an input connection called $\overline{\text{RESET}}$ that is used to initialise the processor and start it running in a predetermined way.
Whenever a logic low is detected on the $\overline{\text{RESET}}$ input the program counter is loaded from locations $FFFE (high byte) and $FFFF (low byte) and the microprocessor then begins execution of the instructions addressed by the program counter.
There are two ways in which we need to use the $\overline{\text{RESET}}$ input.

- A reset button can be connected to bring the voltage of the $\overline{\text{RESET}}$ pin to 0 V and thereby restart the microprocessor.
- When power is initially switched on we want to keep the $\overline{\text{RESET}}$ pin voltage low for a short period so that the microprocessor starts with a reset operation and thus initially executes the program at the address stored in $FFFE and $FFFF.




$$V_c = 5\left(1 - \exp\left(\frac{-t}{RC}\right)\right)$$

$R$ and $C$ should be set so that $V_c$ does not rise above about 2 V before $t = 8\mu$s. This allows roughly 8 clock cycles which is the minimum time required to initiate the reset sequence.