

Embedded Microcontrollers

M3X8529

Dr Caerwyn Ash

Aims of course

- * To understand the basic workings of a computer.
- * Be able to write simple programs in the microcontroller assembly language.
- * Apply a microcontroller as a component in an electrical system as instrumentation, as a control element in an embedded system

A microcontroller is a simple computer, or strictly speaking the working heart of a computer. Its power comes from being programmable i.e just one microprocessor can execute several different functions sequentially rather than have several pieces of dedicated electronics all working in parallel. Thus we save on space and cost, but may lose out on speed of the response.

All programs consist of.....

- * fetch data (variables) (keyboard, files)
- * Process it (arithmetic, sorting)
- * output result (display, vdu, file, printer)

So an embedded microcontroller in instrumentation.....

- * reads values from sensor
- * Convert its to parameter being measured
- * displays it

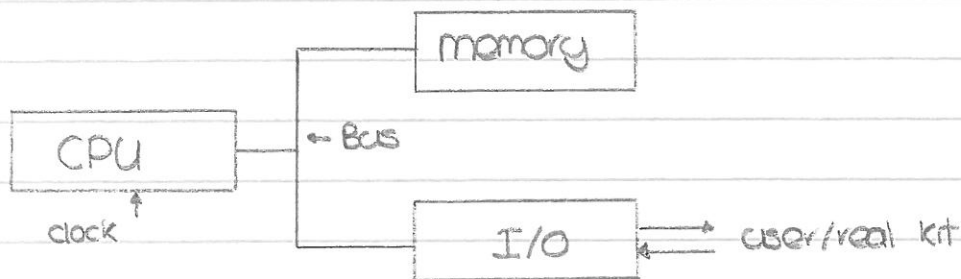
So an embedded microcontroller in control.....

- * reads a value from sensor
- * decides value seen (comparision with threshold)
- * output drive to actuator (corrective or next position)

Differences from "normal" programs

- * I/O from and to sensors or actuators (real devices with an electronic interface)
- * self starts, theres no complicated commands to start program.

Basic microcontroller structure and operation (applies to all basic computers)



Central Processing Unit (CPU)

The central processing unit or CPU for short is the working heart of a microcontroller, it does all the work such as calculation. The CPU controls the operations of all other units (99% true). A CPU is a synchronous device which means that the data is received in parallel, and the speed of operation is directly proportional to the clock speed.

memory

A microcontroller needs two types of memory to operate. firstly the programs need to be stored in a non volatile way and cannot be changed. The programs that are going to be executed by the CPU are read from this memory which is termed ROM, Read only memory. The other type of memory is a place to store variables used during program execution. These variables are written and read by the CPU, the initial value is set by the program so the value at power on is irrelevant. This type of memory is termed RAM, Random Access Memory. By this method the CPU can change one variable at a time without disturbing other stored locations.

Input/Output (I/O)

The input/output section of a computer handles the transfer of data between the microcontroller and external devices or peripherals. The transfer involves status and control signals as well as data. The input/output section must reconcile timing differences between the microcontroller and the peripherals, format the data properly, handle status and control signals, and supply the required current and voltage levels.

The Output location or port is simply a write variable operation, but that information (data) is not merely stored, but sent to the outside world to control equipment.

The input port is a read variable operation, but rather than retrieve data previously written, we can input information from sensors, switches, etc...

Microcontroller organisation

memory

We consider memory and input/output (I/O) as a series of discrete locations each with a unique identifying address (encoded binary word) and contains a byte of data or word in the case of larger microcontrollers, i.e. each location, and some I/O locations can be considered as $8 \times$ D-type flip flops in parallel known as an 8-bit storage register.

Bus and Buses

A Bus is a parallel set of wires carrying a word of common information.

Address Bus

An address Bus is directional, coming from the CPU, and can be 16, 20 and 32 bits large.

Data Bus

A Data Bus is Bidirectional between the CPU and memory or I/O, and can be either be 8 or 16 bits in size.

Control Bus

A Control Bus is directional coming from the CPU. This Bus is a series of status bits indicating the type of Bus access taking place for example the qualifying address.....

RDNCWR ; M1

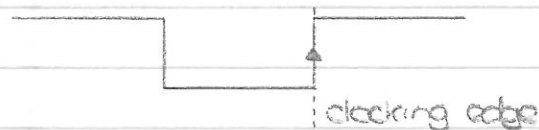
(1st Bus cycle of instruction)

STACK ; INTACK

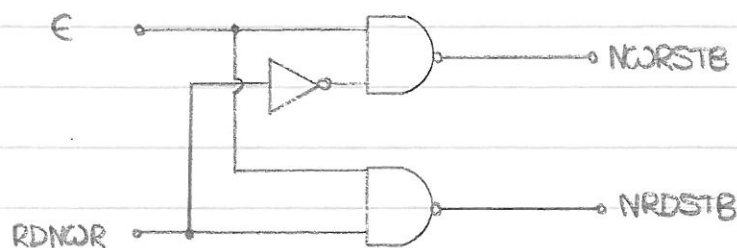
(acknowledge interrupt)

The control and timing signals indicating when the CPU requires valid data.

NRDSTB and NWRSTB active low strobes, these strobes are active on the rising edge. (trailing edge)



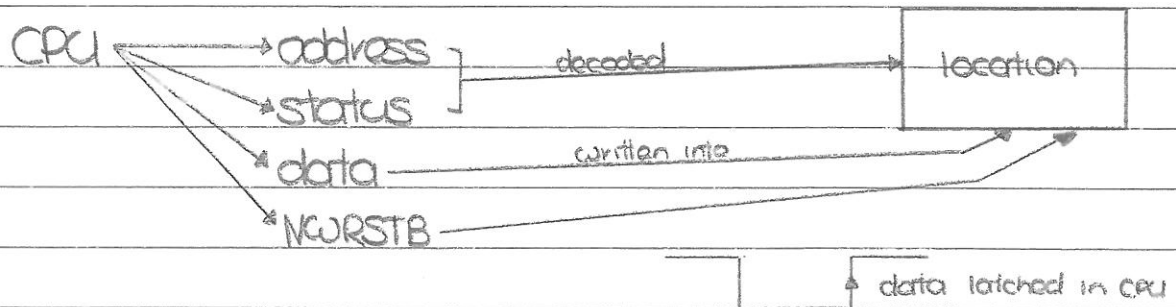
The NRDSTB and NWRSTB can be easily obtained from RDNCWR and a single strobe E.



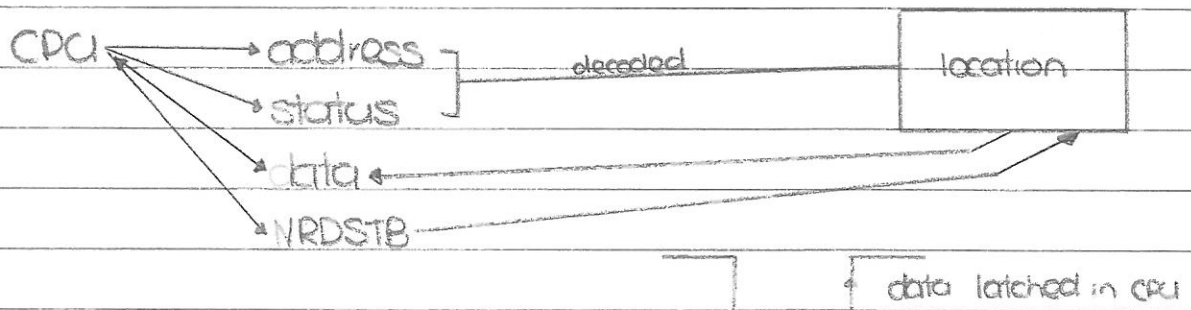
The RDNCWR differs in its timing from the strobes. Because it is early in the cycle (alongside address)

Bus Access Diagrams

Write cycle



Read Cycle



We have been describing a synchronous Bus, which means everything is synchronised to the CPU timing.

Consider a high level program.....

```
while (a < maxc)
{
    a = a + tab(b++);
}
```

to execute this program we need.....

- * a variable (a) and where it is stored (address)
- * a variable (b) and where it is stored (address)
- * a constant (maxc)
- * ability to perform a magnitude test
- * ability to alter program flow, ie find the next instruction in sequence or branch out.
- * arithmetic..... +, ++
- * variable access based on variable b

```
while (a < max)
{
    a = a + tab (b++)
}
```

```
if (val = limit)
    then x = y;
    else x = y + z;
```

to execute this program we need.....

- * a variable (val) and its location
- * a constant (limit)
- * equality test
- * program flow
- * variable access and arithmetic

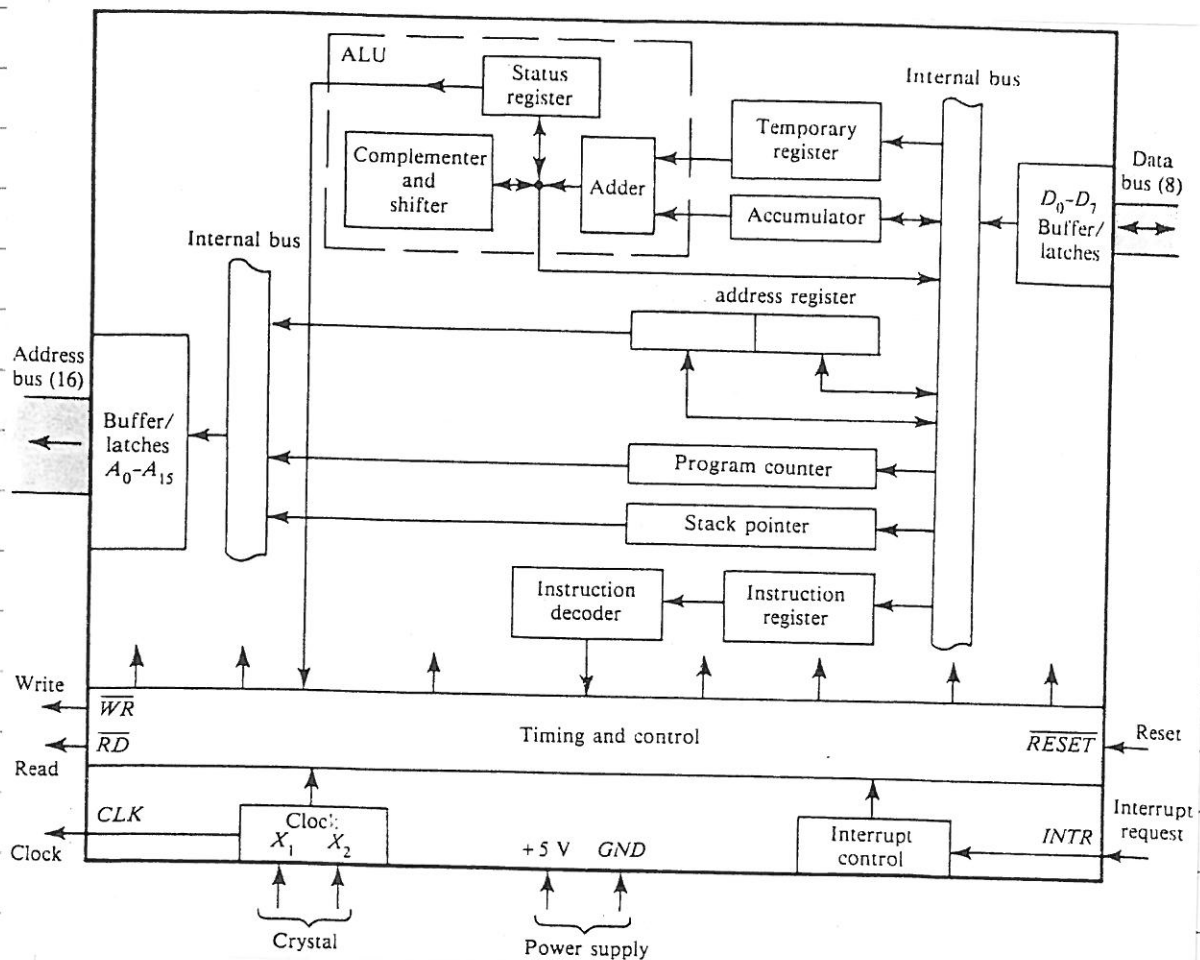
Arithmetic Logic Unit (ALU)

The arithmetic logic unit or ALU for short performs all the arithmetic and logic operations.

The ALU uses a internal register as one source and an accumulator (acc) for the logic.

The HCL microcontroller has two accumulators AccA and AccB. The result of the operation is nearly always placed back into the accumulator.

The microcontroller



1. Instruction register

this unit is an 8-bit register that holds the first byte of an instruction (the op code)

2. Instruction decoder

This unit interprets the contents of the instruction register; determines the exact microprogram to be followed in executing the entire instruction, and directs the control section accordingly.

3. Arithmetic and logic unit

This unit performs the arithmetic, logic and rotate operations which affect the status register (flags). The results from the ALU section are placed back into the accumulator via the internal bus. The temporary register and

accumulator are many times considered part of the ALU. The flag conditions are fed back to the timing and control unit.

4. Accumulator

This unit is an 8-bit general purpose register that is the focus of most arithmetic, logic, load, store, and i/o instructions.

5. Program Counter

This unit is a 16-bit storage area that always points to the next instruction to be executed. It always contains a 16-bit address. It can be incremented or reset by the control section or modified by transfer instructions.

6. Timing and control unit

This section receives signals from the instruction decoder to determine the nature of the instruction to be executed. Information from the status register is also available for conditional branching. Timing and control signals are sent to all parts of the microcontroller to coordinate the execution of instructions. External control signals are also generated.

7. Status register

The very simple generic microcontroller in the diagram contains only zero and carry flags in its status register.

Program Counter

The program counter contains the address of the next instruction to be executed. Everytime it is used as an address source for Bus access, it is incremented automatically.

Hence we see why programs are stored in sequentially addressed memory locations. All that is required is that the initial (power on) value of the program counter is known and the program starts at that address.

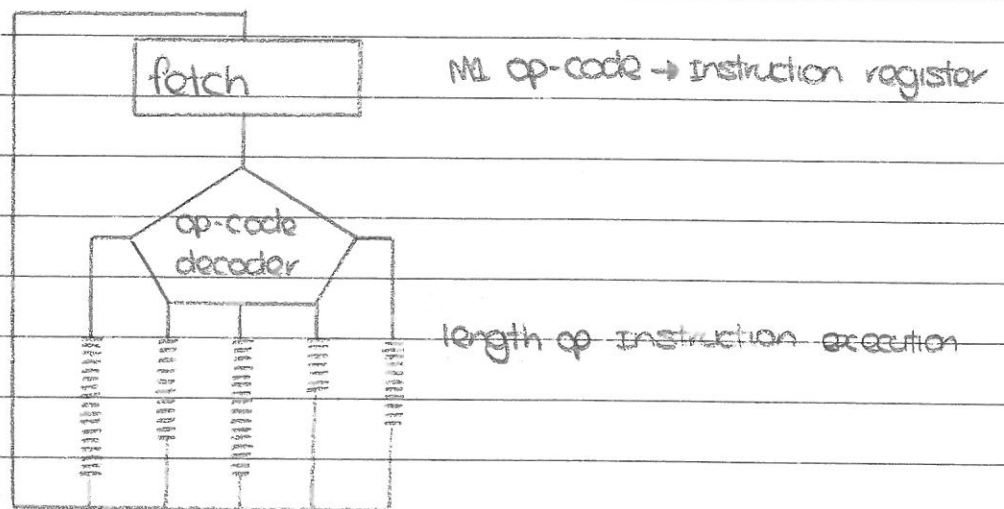
The program counter can be loaded with a parallel word or 2x bytes to alter program flow.

Data is returned by first Bus access cycle in any instruction is loaded into the instruction register.

This is known as the instruction fetch cycle or op-code fetch (M1 cycle). Every instruction does this at least once. Each op-code is unique and defines what the instruction does. It is decoded by the instruction decoder which tells the control section how to implement the instruction, known as the instruction execute phase. Hence we have a repeated instruction fetch - execute cycle

↑ (always the same M1) ↑ (varies depending on instruction)

Fetch-execute cycle as an Asm (Algorithm stack machine)



Mnemonic is assembly code, which you type into the assembler.

Op-code and operand is the machine code, the raw binary used in microcontrollers. Computers can only understand this basic binary code to operate. The operand varies in length depending on the instruction.

Cycles is the term used to describe the number of bus cycles needed to execute the cycle, therefore a measure of the time to execute. There will always be the a number of machine code bytes and any extra cycles needed to access off-chip variables or compute answer internally.

Addressing Modes

Addressing modes are ways to specify operands (variables). In fact, just one operand: "the destination" and one source (in 2 source operations) is always implied, fixed as an accumulator (defined in op-code)

Immediate addressing

the data (operand) is fixed and constant within the program. This addressing has two cycles....

- * op-code fetch
- * load immediate byte

Direct addressing

The address of the variable is in the program, i.e., the address is fixed always the same variable used.

Motorola call direct addressing "extended addressing", here direct addressing is a sub-set of extended addressing with the top byte = 00 i.e., it is extended to the bottom page of memory 0000

so that LDAA \$23 $\Rightarrow$ 9623 (direct, 8-bit)

LDA \$1023 $\Rightarrow$ B6 1023 (extended, 16-bit)

LDA \$23 takes 3 cycles * op-code fetch (PC)

* add fetch (PC+1)

* operand fetch (0023)

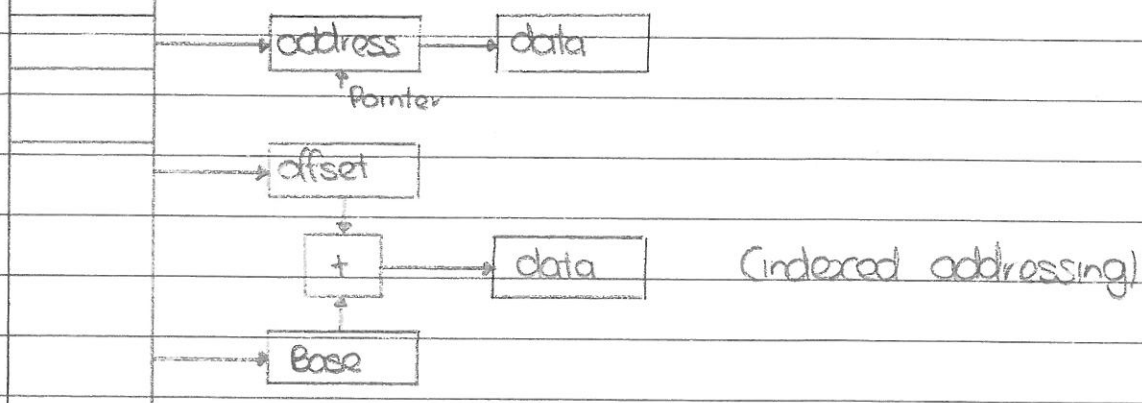
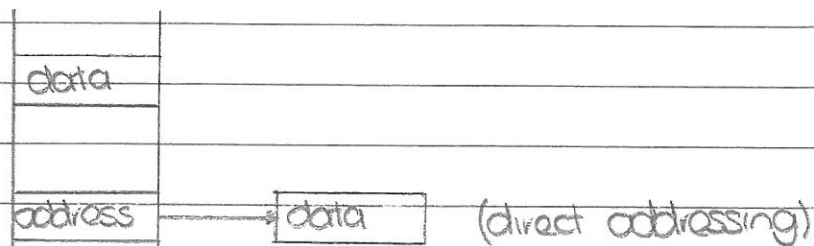
LDAA \$1023 takes 4 cycles * opcode fetch (FC)

* hh fetch (PC+1)

* LL fetch (PC+2)

* operand fetch (1023)

Program
↓



Indirect addressing

The address of the operand is itself held in data memory and can be varied (pointer). This is used to manipulate arrays of data. The HC11 does not support indirect addressing per se, because MAR is not accessible by the programmer. The HC11 provides a variety of indirect addressing through its indexed mode.

Indexed addressing

The address of the variable is computed by adding an "offset" to a "base", offset is normally 8-bits for the base and 16-bits for the offset and the "base" could be held in workspace or be a constant in the program. In the MC11, offset is a constant and there are 2 base registers x and y . you can consider index registers x and y as potentially indirect address sources, offset is treated as a true integer i.e., we can move 0 to 255 + base value.

LDAA \$23,x needs 4 cycles $\Rightarrow$

* opcode fetch	(PC)
* offset fetch	(PC+1)
* Compute MAR	(x + offset)
* fetch operand	(x) + offset

Program Control

Unconditional

The program counter is always loaded with a new value, used at the end of loops, to go back to beginning, or to move to another segment of the program, elsewhere in memory.

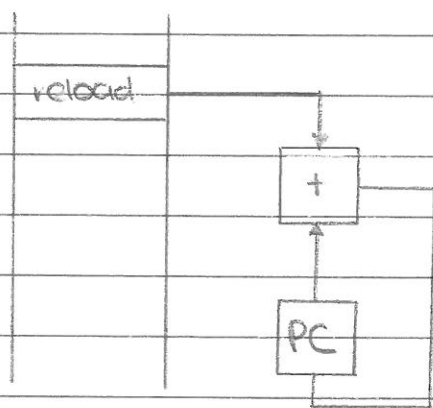
SMP hll

specifies full address

BRA rr

specifies relative byte

Relative to the present Program counter value, this is the same mechanism as indexed addressing, the program counter is the base register added to relative byte in the program.



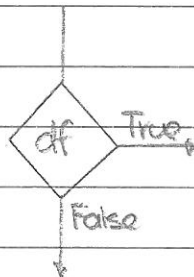
rr = signed value

ff = unsigned value

rr is taken to be a signed integer, ie, we can branch +127/-128 from the present program counter value.

Conditional Branch

Conditional branching is of the form, if a flag is set (or clear) then branch, else go on in sequence.

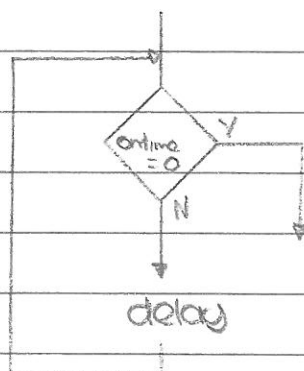


So BEQ bug tests zero flag for '1' i.e. last result was zero set z-flag, and branches to position in the code indicated by "loop" label if so. The assembler works out rr

The programmer must ensure that the condition he or she wants to test is reflected accurately by the flags available to test in assembly code.

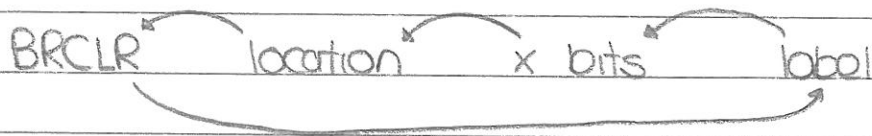
It is wise to construct your program at a high level, using structured constructs for example, while, do, and translate that accurately to assembly code.

```
while (ontime  $\neq$  0) do  
{  
  delay;  
}
```



Bit test Conditional Branch

This branching performs a branch depending on whether or not bit or bits in a location are set clear.



Branch to "label" if "bits" in "location" are clear.
"location" must be indexed addressed, "bits" are specified on a "mask" byte with a 1 in the relevant bit position, for example to test bits 3, 7 mask = 88_{16} [10001000]

Subroutines

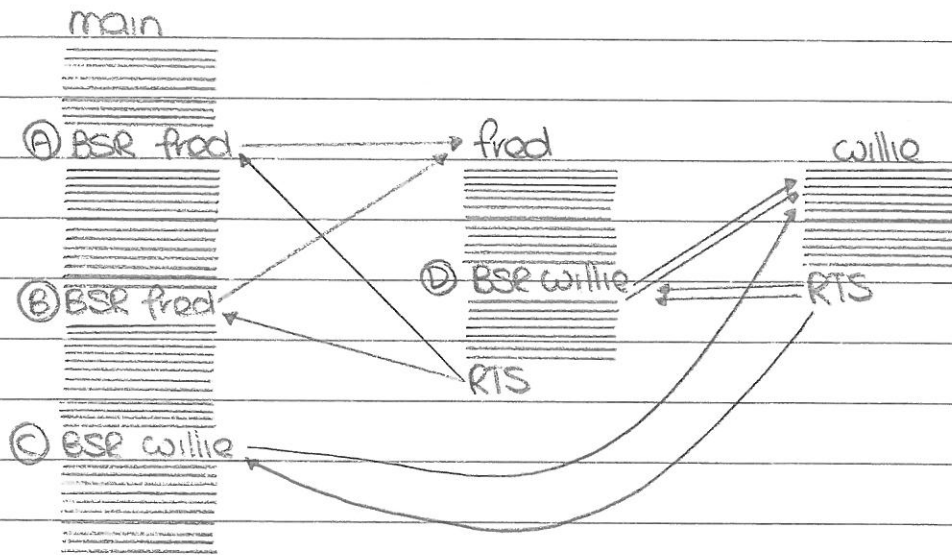
BSR	Branch to SubRoutine	"call"
RTS	Return To Subroutine	

Subroutines are used to allow a piece of common code, for example a delay, calculation such as a square root, to be accessed at different parts in the program. Only one copy of the subroutine exists.

On encountering a BSR instruction, the program counter is adjusted to the new value, however the "old" Program counter value is stored on a stack. The stacked Program counter value is the address of the next instruction in the main program, after the BSR instruction. RTS simply takes the top of the stack into the program, i.e. it restores the address of this instruction that RTS is the last instruction in this subroutine.

The nature of a stack means that subroutines can be nested, so that a further subroutine can be called from within a subroutine.

for example....



Program Structure

Try to think in high level constructs and mimic these in your assembly code, for example....

```
do forever
```

```
{
```

```
  while (switch not set)
```

```
  }
```

```
  {
```

```
    turn led on;
```

```
    delay ontime;
```

(this uses two delays)

```
    turn led off;
```

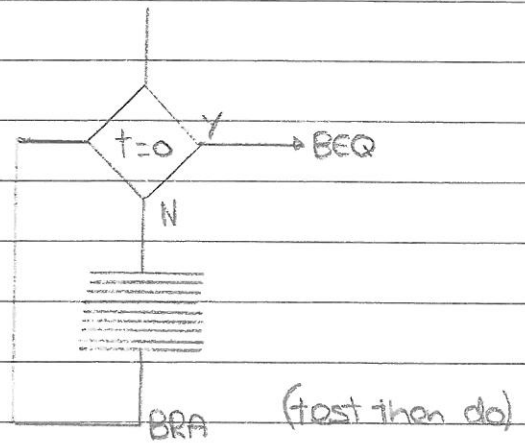
```
    delay offtime;
```

```
  }
```

```

while (time != 0) do
{
    delay (100ms);
    time--;
}

```

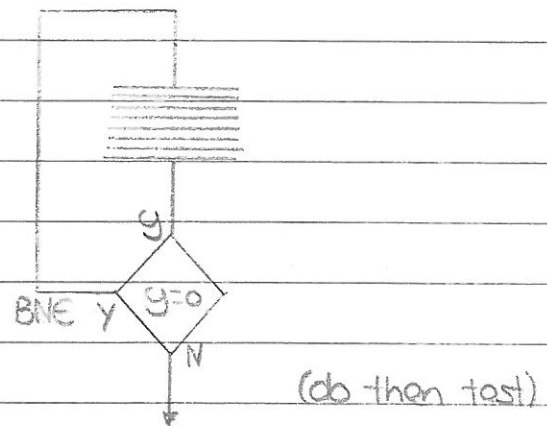


delay subroutine

```

for (y = time to 0) do
{
}

```



Similarity

```

repeat
{
    delay;
    time--;
}
until (time = 0)

```

logic operations

Bytes here are treated as 8 separate bits. logic operations are performed bit-wise between the accumulator and the operand. There are 8×2 input operations with the resultant bits in the accumulator being set with regard to the corresponding operand bits, for example, an AND

Acc	1 0 0 1 1 1 0 1
Operand	1 1 0 0 0 1 1 0
Result in Acc	1 0 0 0 0 1 0 0

This can be thought of as a "masking" instruction, any 0's in the "mask" operand produces a 0 result a 1 lets the original bits pass to the result unchanged. This is used to separate field in a byte, for example..... in BRCLR $M \text{ mm} = 0$?

M	a	b	c	d	e	f	g	h
mm	0	0	0	0	0	0	0	1
result	0	0	0	0	0	0	0	h

for example BRSET $\bar{M} \text{ mm} = 0$

M	a	b	c	d	e	f	g	h
mm	1	0	0	0	0	0	0	0
result	$\bar{a}$	0	0	0	0	0	0	0

for example BCLR $M \bar{m} m \Rightarrow M$

M	a	b	c	d	e	f	g	h
mm	0	0	0	0	1	0	0	0
result	a	b	c	d	0	e	f	g

This is useful for manipulating control bits in an interface, where other bit settings may not be known. This instruction and BSET reads and writes the operand (M), thus the same operand must be available for read and write access, but not always true for control byte interfaces.

OR operation

The OR operation is used to merge fields.
for example

Acc	1	0	0	1	0	0	1	0
Operand	↑	↑	↑	↑	↑	↑	↑	↑
	1	1	0	0	1	0	1	1
Result	1	1	0	1	1	0	1	1

BSET	M	a	b	c	d	e	f	g	h
	mm	↑	↑	↑	↑	↑	↑	↑	↑
		1	0	0	0	1	0	0	0
	Result	1	b	c	d	1	f	g	h

Rotate

In its simplest form, rotate moves the accumulator bit pattern one place left or right, which is useful for phased waveforms. The HCL does a 9-bit rotate, through "Carry flag" which means that the c flag must be set or cleared to keep correct pattern, but the end bits can be tested.

Arithmetic shift

This adapts the logical rotate to provide $(\times 2) + (-2)$
ie, shift left 1 bit and shift Right 1 bit. for example

(SHL) a b c d e f g h
 (a) b c d e f g h (1)

Provided that $a=0$, $I=0$, a goes to the c carry
flag to test if its too big

example a b c d e f g h
 1 a b c d e f g $\rightarrow$ h

(SHR)

Provided that $h=0$, $I=0$, h goes to the carry
flag to test for a remainder.

logical shift

unsigned integers

Arithmetic shift

signed integers