ROYAL SIGNALS & RADAR ESTABLISHMENT

USING ELLA FOR HIGH LEVEL DESIGN: MODELLING THE MOTOROLA 6800 MICROPROCESSOR

Author: A R Huggett

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.
Reports quoted are not necessarily available to members of the public or to commercial organisations. 

COPYRIGHT (c)
1988
CONTROLLER
H M O LONDON
Title: Using ELLA for High Level Design:
Modelling the Motorola 6800 Microprocessor.

Author: A R Huggett (Student Engineer)
Date: 5 September 1991

Summary

This document starts by providing a brief overview of the ELLA language. The majority of the document describes the design and development of an abstract high level ELLA model of a Motorola 6800 Microprocessor, including all of the 197 codes which make up the 6800 instruction set.
Disclaimer:

The author gives no guarantee that the model herein described is a fully accurate and error free description of the Motorola 6800.

The views represented within this report are those of the author alone and do not necessarily represent the views of the Ministry of Defence nor the Defence Research Agency.

Notices:

Throughout the text the symbol & is used to indicate a hexadecimal number. However, in the ELLA language this is replaced by 16r, thus address &e000 becomes ad/16re00 in ELLA.

Variables and Types which occur within the 6800 model are indicated by bold type.

Words in the glossary are indicated in italics.

References to the Bibliography are indicated thus ¹.
Contents:

The ELLA Language ................................................................. 4
The Motorola 6800 Model .......................................................... 5
  Specification ............................................................................ 5
  Variable Type Selection ......................................................... 5
  Initial Design ......................................................................... 6
  Development ............................................................................ 7
  Testing ..................................................................................... 8
  Conclusions ............................................................................ 9

Appendix A: The ELLA Description ............................................. 10
  Variable Type Definitions ....................................................... 10
  Bit Tester ................................................................................. 10
  Arithmetic and Logical Unit .................................................... 11
  Decision Logic ........................................................................ 17
  Instruction Decoder ............................................................... 19
  Counting and Type-Swapping Functions .................................... 23
  Main Microprocessor Control Logic ......................................... 24
  External Memory and Test Harness .......................................... 35

Appendix B: Programs ................................................................. 36
  Initialise .................................................................................. 36
  Monitor ................................................................................... 37
  Testprog1 ............................................................................... 38
  Testprog2 ............................................................................... 39
  Testprog3 ............................................................................... 40
  Testprog4 ............................................................................... 41
  Testprog5 ............................................................................... 43
  Testprog6 ............................................................................... 45
  Testprog7 ............................................................................... 47

Appendix C: The Motorola 6800 Instruction Set ......................... 48
  Accumulator and Memory Instructions ...................................... 48
  Index Register and Stack Manipulation Instructions ................ 48
  Jump and Branch instructions ............................................... 49
  Dummy Mnemonics Added to Simplify ELLA Program ............... 49
  Condition Code Register Manipulation Operations .................. 49

Glossary ...................................................................................... 50

Bibliography .............................................................................. 52
The ELLA Language

ELLA is a Textual Language for describing digital hardware designs. The language is designed so that ELLA circuit descriptions are always realisable in hardware.

The components of a circuit are described in ELLA by functions. A function definition declares the number and types of inputs and outputs. The function body is composed of a number of constructs, which describe how the inputs map to the outputs.

Before any functions may be written in ELLA it is necessary to define the variable types which they will use. This is achieved by means of the TYPE statement, e.g.

```
TYPE count = NEW co/(0..12)
TYPE addrmode = NEW (immed, direct, index, extnd, implied, relative)
```

The first statement defines TYPE count to be an integer in the range 0..12. The co/ tag is necessary to distinguish that type from any other integer type. The second statement defines TYPE addrmode as a six valued enumerated type, with possible values of immed, direct, index, extnd, implied and relative.

ELLA supports a wide variety of high level constructs such as ARITH which performs integer arithmetic, DELAY which delays information between its input and output and provides a means of feedback from one clock cycle to the next.

The language is strongly typed in order to minimise the chances of the user introducing a design fault. It has a hierarchical functional structure similar to PASCAL.

The main difference between ELLA and conventional programming languages such as C, PASCAL etc. is that it is a parallel rather than a sequential language. This is necessary to describe logic circuits. However, since it is a parallel language, to try and express

```
LET a = da/5
LET a = da/4
```

is illegal since the compiler tries to implement both of these statements in parallel. A variable defined by LET is therefore static once assigned within a timestep its value cannot be changed within that timestep although it can be used as an input to other functions. The value of a static variable can be changed on a different timestep however.

For some applications, however, it is more convenient to program sequentially, particularly for complex high level descriptions. The SEQ construct allows sequential programming, with two types of dynamic variables, which must be defined before they can be used, but once defined may be assigned to many times within the sequence. The first of these is VAR. VARS are initialised to a predefined value every timestep. The other type is PVAR (or STATE VAR). These are initialised at time 0 but retain the latest assigned value from one timestep to the next until they are reassigned. They are very useful for making registers. Sequential constructs can be directly transformed into parallel ones using a set of assembler tools.

Care must be taken to ensure that the same instantiation of the function is being used if the old value of the PVAR is required. In practice this means that functions which contain constructs such as VAR, PVAR, or RAM and DELAY functions become wrapped up inside parallel functions which allow explicit instantiation by using the MAKE and JOIN constructs.

For a complete description of the ELLA Language refer to the ELLA Manual.
The Motorola 6800 Model: Specification

My model of the Motorola 6800 was developed from a very limited amount of information. I had the instruction set, the assembler mnemonics, number of bytes, number of clock cycles required and effect of the function. I did not know anything of the order in which internal subfunctions are carried out within the microprocessor. My knowledge of and information on the condition codes register was also less than ideal. Since I had no information about the internal operations of the 6800 I decided that my model should fulfill three criteria:

i) It should perform the correct instruction for the appropriate instruction code.

ii) It should take the correct number of clock cycles to complete each instruction.

iii) All 197 valid op-codes should be implemented.

In addition I wished the model to have what I hoped was a realistic internal architecture.

The Motorola 6800 Model: Variable Type Selection

Nine different variable types are used within the ELLA description (see Appendix A). The reason for this is as follows.

The Motorola 6800 is an eight-bit microprocessor. Therefore all data coming into and out of the microprocessor is eight bits wide and TYPE data provides this. The choice of TYPE data to be positive in the range 0 to 255 rather than signed -128 to 127 was made for two reasons.

i) The mathematics of instructions such as rola become easier to implement.

ii) Logic synthesis is often easier if there are no negative numbers.

Since the address space of the 6800 is 0 to 65535 (0xffff) it was clear that a sixteen bit type, TYPE address was needed. This time it was obvious that the numbers should be positive.

A four bit type, TYPE halfbyte proved useful within the instruction decoding functions, because the more significant four bits of an op-code always refer to the addressing mode and the less significant to the instruction.

TYPE flag is a two state boolean which is useful in CASE clauses.

TYPE bitint is the integer equivalent of TYPE flag, it is used in some ARITH statements as the carry bit.

TYPE addrmode and TYPE mnemonic are enumerated types which represent the addressing mode and mnemonic of an op-code in a user-friendly way.

TYPE count is used in the microprocessor’s internal structure to determine how far through the instruction it is. Since the longest instruction takes 12 cycles the maximum value of TYPE count is 12.

TYPE result is the 9-bit answer of the main ARITH functions within FN ALU. It is necessary because the 9th bit is important for setting the carry register.
The Motorola 6800 Model: Initial Design

The Motorola 6800 is a simple 8-bit Von-Neumann microprocessor which has two 8-bit accumulators, a and b, an index register x, a stack pointer sp and a condition codes register cc. It also has a program counter pc. In addition I found it useful to create an address buffer addrbuf which I now know is not present on the real chip.

I originally had to decide whether it would be better to build a model which interpreted 6800 assembler code or hexadecimal op-codes. I decided that using the memory to store mnemonics as a separate type from data was too far away from reality, and too difficult to implement. Therefore I opted for a memory interface which just used 8-bit data. However, in order to make the design easier to program and to understand I decided that instructions should be stored within the microprocessor as an assembler mnemonic plus an addressing mode. Thus FN MNEMONIC and FN ADMODE are used to extract this information when the incoming data word is an instruction (see Appendix A).

Studying the data I found that with one exception (which I erroneously took to be a printing error), the addressing mode affected the number of cycles required to perform an instruction as follows.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>+0</td>
</tr>
<tr>
<td>Direct</td>
<td>+1</td>
</tr>
<tr>
<td>Relative/Extended</td>
<td>+2</td>
</tr>
<tr>
<td>Indexed</td>
<td>+3</td>
</tr>
<tr>
<td>Indexed</td>
<td>+4</td>
</tr>
</tbody>
</table>

I deduced that the microprocessor must have some way of knowing how far through an instruction it has proceeded since all instructions take between 2 and 12 timesteps. I therefore created the model with an internal counter (intercount) which operates as follows.

If intercount is zero then the incoming byte on the databus is treated as an instruction, and the functions ADMODE and MNEMONIC are used to change the instruction byte into an addressing mode and a mnemonic. From these the function HOWLONG calculates how many clock cycles the mnemonic should take, plus an offset for the addressing mode. Intercount is then set to the value returned from FN HOWLONG. In order to facilitate future logic synthesis I have avoided the use of negative numbers. Thus FN COMPARECOUNT always returns a positive value or zero if the second input is greater than or equal to the first. FN COMPARECOUNT is then called to see how far into the addressing mode cycle the microprocessor has proceeded, by comparing intercount with the number of cycles required to perform the instruction if there were no addressing steps, and the appropriate addressing step is carried out. If the value returned from FN COMPARECOUNT is zero then the comparison is repeated with the inputs reversed so as to provide a result which might be positive or zero, and the appropriate instruction step carried out.

In order to test the microprocessor it is necessary to connect it to a memory, so I created FN MAKERAM and FN HARNESS which provide this plus a reset line.

The first instructions to be implemented were adda, ldaa, suba, beq, jmp and staa (see Appendix C). These were tested by means of two test programs, TESTPROG1 and TESTPROG2 (see appendix B).
The Motorola 6800 Model: Development

Having got the memory and the first six instructions working to my satisfaction I began increasing the instruction set. After some time it became apparent that the length of the main program loop was getting too big to be easily understood, and furthermore, was beginning to get unrealistic with 16 bit adders creeping in that do not exist in the real 6800. I therefore decided to change the way in which arithmetic and logical functions were being implemented. I created FN ALU (Arithmetic and Logical Unit) which had basic functions defined. These were FN ADD BYTE, FN OR BYTE, FN NOT BYTE, FN AND BYTE and FN SR BYTE. (ADD BYTE performs binary add, SR BYTE binary shift right with wraparound, OR BYTE binary OR etc.) There were also several functions for such things as truncating 9-bit results (FN CHOP DATA) and condition codes register handling. Later I added a sixth main instruction, FN DA ABYTE which performs a decimal adjust on accumulator a.

In order to implement 16-bit operations such as Inx (see Appendix C) I found it necessary to pass information as to which part of an instruction was to be executed by the ALU. I achieved this without increasing the number of inputs by utilising the two most significant bits of the condition codes register, which the data sheet defines as both being true at all times, and therefore redundant.

The text was tidied by using functions such as FN BRANCH to wrap up functions of similar type. The functions which take up a lot of space are those such as jsr and swi (see Appendix C) which take many timesteps and are fairly unique. A lot of expansion to FN ALU was also required.

Finally I implemented the interrupt request, non-maskable interrupt and halt lines into the microprocessor. These are separate level-sensitive lines which are inverted, i.e. a logic false causes an interrupt to occur or the microprocessor to cease all operation accordingly. The handling of these interrupts is virtually identical to the swi instruction. I therefore decided that by far the easiest way to implement the interrupt was to have two dummy mnemonics, irq and nmi.

I also memory mapped three inputs into the computer and three outputs from it. Locations &f000 to &f002 if read from by the microprocessor return the three data inputs to FN HARNESS. Similarly locations &e000 to &e002 if written to change the output of FN HARNESS. This arrangement can be thought of as being equivalent to a set of three input registers which can be loaded by using a keypad or other means, and six hexadecimal output digits.

With the microprocessor now theoretically fully functional I wrote an initialisation program which clears all the registers to definite values, and a monitor program which is called by a non-maskable interrupt.
The development of the model was a repetitive cycle of adding more instructions, removing compilation errors and then testing by means of test programs (Appendix B). These were called into the simulator as ELLA instruction (*.elh) files which initialised the ram to the values required. The majority of programs perform the same function: they calculate the first ten numbers in the Fibonacci series which are calculated by adding the previous two terms in the series to get the current one. Assembler mnemonics are provided as comments in the programs to assist the reader.

Due to the large number of instructions (197 valid op-codes, 2 other interrupt functions, a reset function and a halt line), it was not possible to test every instruction in the time available. However, I believe that the instructions contained within the test programs represent a good cross section. The complex instructions such as jmp, swi,_irq, nmi, rti, rts, jsr and bsr have all been tested, however some simple mathematical functions such as addb are assumed because adda works and addb is a direct copy (see Appendix A)
The Motorola 6800 Model: Conclusions

My model of a Motorola 6800 microprocessor fulfills the 3 specification requirements set out above. However there may be some errors in the condition codes register; these could definitely be found and fixed with some more testing time.

The internal architecture is quite unlike the real 6800. The Motorola data sheet became available very late in the development stage when I was implementing interrupts. With a few days work, however, I believe that the program could be modified to give it a much better resemblance to the 6800 internally.

Given the same project again I would implement the timing of addressing mode operations differently, and would try and mimic the two phase clock required by the 6800 (my current model has only a single phase). There would be no addrbuf register and all internal registers would be 8-bits wide. With the use of the Motorola data sheet I could ensure that every register contained the right information on every cycle (the current model is only correct at the end of each instruction).

The development of my model was largely based upon a very limited amount of information. This lack of information lead to a number of differences between it and the real 6800. However I believe that this model demonstrates that a fully functional ELLA description can be obtained with limited information and minimal difficulty which, when viewed from outside the microprocessor/memory system, behaves identically to the real thing.
Appendix A: The ELLA Description

The ELLA description below is arranged in a compilable order i.e., declare before use. Thus, the main microprocessor function FN UP6800 is towards the end of the listing.

# Variable Type Definitions#
TYPE data = NEW da/(16r00 16rff),
       address = NEW ad/(16r0000..16rffff),
       result = NEW re/(16r0000..16r1ff),
       bitint = NEW bi/(0 1),
       flag = NEW (hi)،
       count = NEW co/(0..12),
       addrmode = NEW (immned; direct; index; extnd; implied; relative),
       halfbyte = NEW hb/(16r0 16r06f),
       mncronic = NEW
       (adca; adda; addsl; clda; clda; clra; cmpa; coma; deca; eora; incar; oraa; psbb; pshb; pula; rola
       | rora; rsl; rsl; nega; lada; suba; staa; sbca; tsta; cpx; dxc; dse; inx; ms; ldx; ldx)
       stx; stx; stx; stx; stx; stx; stx; stx; bra; bcc; bcs; bcz; bge; bgt; bhi; bhi; bhi; bhi; bhi; bhi;
       jmp; jsr; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp; jmp;
       arith IF (input control) = control
       THEN 1 "h"
       ELSE 2 "I"
       FI
# Arithmetic and Logical Unit #

FN ALU = (mnemonic: inst, data: input1 input2 ccreg) -> [2]data:

BEGIN SEQ

FN ADDBYTE = (data ip1 ip2, bitint ip3) -> result ARITH (ip1 + ip2 + ip3),

FN ADDBYTE = (data ip1 ip2) -> result ARITH ip1 IAND ip2,

FN ORBYTE = (data ip1 ip2) -> result ARITH ip1 IOR ip2,

FN NOTBYTE = (data ip1) -> result ARITH (INOT ip1) IAND 16r1ff;

FN CHOPDATA = (result ip1) -> data IARITH ip1 IAND 16r1ff;

FN SRBYTE = (result ip1) -> result ARITH (ipi SR 1) + 256*(ipl IAND 1);

FN DAABYTE = (data ip1 ccreg) -> result.
   ARITH ip1 + IF ((ipi IAND 16rf) > 9) OR (ccreg IAND 16r20) THEN IF (ipi IAND 16r00) = 16r20 THEN 16r66 ELSE 6 Fi ELSE IF (ipi IAND 16r00) > 16r90 THEN 16r60 ELSE 0 Fi.

FN HALFCARRY = (data ip1 ip2 ccreg, bitint ip3) -> data
   ARITH (ccreg IAND 16rdfl) + IF ((ipi IAND 16rf) - (ip2 IAND 16rf) + ip3) > 16r0f THEN 16r20 ELSE 0 Fi.

FN ACARRY = (result ip1, data ccreg) -> data
   ARITH (ccreg IAND 16rf) + IF ip1 > 255 THEN 1 ELSE 0 Fi.

FN SCARRY = (result ip1, data ccreg) -> data
   ARITH (ccreg IAND 16rf) + IF ip1 > 255 THEN 0 ELSE 1 Fi.

FN DCARRY = (result ip1, data ccreg) -> data
   ARITH ccreg IOR IF ip1 > 255 THEN 16r01 ELSE 0 Fi.

FN OVERFLOW1 = (data ip1 ip2 output ccreg) -> data
   ARITH (ccreg IAND 16rdfl) + IF ((ipi IAND 16r80) = (ip2 IAND 16r80)) AND (ipi IAND 16r80) = (output IAND 16r80) THEN 16r02 ELSE 0 Fi.
FN OVERFLOW2 = (data: ip1 ip2 output ccreg) -> data
ARITH (ccreg IAND 16rfd) + IF (((ip1 IAND 16r80) = (((INOT ip2) + 1) IAND 16r80))
AND ((ip1 IAND 16r80) /= (output IAND 16r80))
THEN 16r02
ELSE 0
FI,

FN OVERFLOW3 = (data ccreg) -> data: ARITH (ccreg IAND 16rfd),

FN OVERFLOW4 = (result: ip1, data ccreg) -> data:
ARITH (ccreg IAND 16rfd) +
IF (((ip1 IAND 16r100) SR 1) IAND (ip1 IAND 16r80)) =
(((ip1 IAND 16r100) SR 1) IOR (ip1 IAND 16r80)) THEN 0 ELSE 2 FI;

FN OVERFLOW5 = (data: ip1 ccreg) -> data
ARITH (ccreg IAND 16rfd) + IF ip1 = 16r80 THEN 2 ELSE 0 FI,

FN ZERO = (data: ip1 ccreg) -> data
ARITH (ccreg IAND 16rfd) + IF ip1 = 0 THEN 4 ELSE 0 FI,

FN NEG = (data: ip1 ccreg) -> data
ARITH (ccreg IAND 16rff7) + IF ip1 > 127 THEN 8 ELSE 0 FI;

FN CCCOUNT = (data ccreg) -> data
ARITH (ccreg + 16r40) IAND 16rff,

FN CCRESET = (data ccreg) -> data
ARITH ccreg IOR 16rc0,

VAR answer = re/0,
output = da/0,
newcc = ccreg.
CASE
OF adda, addb: aba (answer = ADDBYTE(input1,input2, bu/0);
output = CHOPDATA answer),
addc: adcb (answer = ADDBYTE(input1, input2, CASE TRUEBIT(ccreg,da/1))
OF h bi/1
ELSE bi/0
ESAC), output = CHOPDATA answer),
anda: andb: (answer = ANDBYTE (input1, input2); output = CHOPDATA answer),
bita: bitb: (answer = ANDBYTE (input1, input2); output = input1),
cra: chr: clr: (answer = re/0; output = da/0),
cmpa; cmpb; cmpc; cmpd: (wew = ADDBYTE (input1, CHOPDATA NOTBYTE input2, bu/1);
output = CHOPDATA answer),
com: com: com: (answer = NOTBYTE input1, output = CHOPDATA answer),
daa: (answer = DAABYTE (input1,ccreg), output = CHOPDATA answer),
nega: negb: neg: (answer = ADDBYTE (CHOPDATA NOTBYTE input1,da/0,bi/1);
output = CHOPDATA answer),
deca: decb: dec: (answer = ADDBYTE (input1, da/255, bi/0); output = CHOPDATA answer),
cora: corb (answer = ANDBYTE (CHOPDATA ORBYTE (input1,input2),
CHOPDATA NOTBYTE CHOPDATA ANDBYTE (input1,input2)),
output = CHOPDATA answer),
inca: incb: inc: (answer = ADDBYTE (input1, da/1, bi/0); output = CHOPDATA answer),
ldaa: lda: ldb: ldx: lds: sts: tab: tba:
(= ADDBYTE(da/0,input2,bi/0); output = input2),
oraa: orab (answer = ORBYTE (input1, input2), output = CHOPDATA answer),
rola; roib; rol: (answer = ADDBYTE (input1, input1,CASE TRUEBIT(ccreg,da/1))
OF h bi/1
ELSE bu/0
ESAC), output = CHOPDATA answer),
ror: rorb:ror: (answer = SRBYTE ADDBYTE (input1, CASE TRUEBIT(ccreg,da/1))
OF h da/255
ELSE da/0
ESAC, CASE TRUEBIT(ccreg,da/1)
OF h bu/1
ELSE bu/0
ESAC), output = CHOPDATA answer),
asla: aslb:asl (answer = ADDBYTE (input1,input1, bu/0); output = CHOPDATA answer),
asra: asrb:asr (answer = SRBYTE ADDBYTE (input1,input1, CASE TRUEBIT(ccreg,da/1))
OF h da/255
ELSE da/0
ESAC, CASE TRUEBIT(ccreg,da/1)
OF h bu/1
ELSE bu/0
ESAC), output = CHOPDATA answer),
lsta: lsrb:lsr (answer = SRBYTE ADDBYTE (input1, da/0, bu/0),
output = CHOPDATA answer),
suba: subb: sba (answer = ADDBYTE (input1, CHOPDATA NOTBYTE input2, bu/1),
output = CHOPDATA answer),
sbca\sbcb. (answer := ADDBYTE (input1, CHOPDATA NOTBYTE input2, 
       CASE TRUEBIT(ccreg,da/1)
         OF h:bi/0
         ELSE bi/1
       ESAC);output := CHOPDATA answer),

tsta\tstb\tstc (answer := ADDBYTE (input1, da/0, bi/0),
       output := CHOPDATA answer),

dex\des (answer := CASE TRUEBIT (ccreg,da/16r40)
       OF h: ADDBYTE (input1, da/255, bi/0),
       l: CASE input2
         OF da/16rff ADDBYTE(input1,da/255,bi/0)
         ELSE ADDBYTE (input1, da/0, bi/0)
       ESAC
       ESAC; output := CHOPDATA answer),

inx\ins (answer := CASE TRUEBIT (ccreg,da/16r00)
       OF h: ADDBYTE (input1, da/1, bi/0),
       l: CASE input2
         OF da/16r00 ADDBYTE(input1,da/1,bi/0)
         ELSE ADDBYTE (input1, da/0, bi/0)
       ESAC
       ESAC, output := CHOPDATA answer)

newcc := CASE inst
       OF adda\addb\aba HALFCARRY (input1,input2, newcc, bi/0),
       adca\adcb. HALFCARRY (input1,input2, newcc, CASE TRUEBIT(ccreg,da/1)
         OF h bx/1
         ELSE bx/0
       ESAC)

ELSE newcc
ESAC,

newcc := CASE inst
       OF adda\addb\aba\adca\adcb\clr\clr\com\comb\nega\negb\rol\rol\ror\ror\asl\aslb\asrb\lsr\lsrb
       \ts\tsb\tsl\com\neg\rol\ror\asl\assl\lsl\ltst
       ACARRY (answer,newcc),
       daa DCARRY (answer,newcc),
       cmpa\cmpb\cma\cmb\suba\sba\sbca\sbcb\suba\sba SCARRY (answer,newcc)
       ELSE newcc
ESAC,
newcc := CASE inst
  OF adda|adca|adcb|addb|aba|daa\text{OVERFLOW1} (input1,input2,output,newcc),
  inca|incb|inc \text{OVERFLOW1} (input1,da/output,newcc),
  suba|subb|sca|scb|cmpa|cmpb|eaa|eab\text{OVERFLOW2} (input1,input2,
  \text{CHOPDATA answer,newcc}),
  dec|dec|dec \text{OVERFLOW2} (input1,da/output,newcc),
  anda|andb|bita|bitb|ela|elr|eaa|eab|lda|ldab\text{OVERFLOW3}(newcc),
  rola|rolb|rol|rolb|asla|aslb|asra|asrb|lsra|lsrb\text{OVERFLOW4} (answer,newcc),
  nega|negb|rol|rol|asra|asrb|lsra|lsrb\text{OVERFLOW5} (output,newcc),
  \text{CPC CASE TRUEBIT} (ccreg,da/16r40)
  \text{OF 1 OVERFLOW2} (input1,input2,\text{CHOPDATA answer,newcc})
  ELSE newcc
  ESAC
ELSE newcc
ESAC;

newcc := CASE inst
  OF cpx|dex|ux|x|ldx|lds|stx|sts
  \text{CASE TRUEBIT} (ccreg,da/16r6c0)
  \text{OF h.ZERO} (output,newcc),
  \text{CASE TRUEBIT} (ccreg,da/16r04)
  \text{OF h.ZERO} (output,newcc)
  ELSE newcc
  ESAC
ELSE newcc
ESAC;

newcc = CASE inst
  OF dex|des|us|xux newcc,
  \text{cpx|ldx|lds|stx|sts CASE TRUEBIT} (ccreg,da/16r40)
  \text{OF h.NEG} (output,newcc)
  ELSE newcc
  ESAC
ELSE NEG (output,newcc)
ESAC;

newcc = CASE inst
  OF cloc \text{CHOPDATA ANDBYTE} (ccreg,da/16r6fe),
  cli \text{CHOPDATA ANDBYTE} (ccreg,da/16r6f),
  clo \text{CHOPDATA ANDBYTE} (ccreg,da/16r6f0),
  sec \text{CHOPDATA ORBYTE} (ccreg,da/16r01),
  sei \text{CHOPDATA ORBYTE} (ccreg,da/16r10),
  sev. \text{CHOPDATA ORBYTE} (ccreg,da/16r02),
  tap \text{CHOPDATA ORBYTE} (input1,da/16rc0)
  ELSE newcc
  ESAC,
newcc = CASE inst
    OF cpx; dex; des; inx; ins; ldx; lds; stx; st
    CASE TRUEBIT (ecreg, da/16rc0)
        OF h. CCOUNT(newcc),
        l: CCRESET(newcc)
        ESAC
    ELSE newcc
    ESAC;

OUTPUT (CASE inst
    OF cmpa; cmpb; cpx: input1,
        tpa newcc
    ELSE output
    ESAC, newcc)
END
\#Decision Logic#

$$\text{FN BRANCH} = (\text{mnemonic-inst}, \text{address.addrbuf \ pc, \ data \ cc}) \rightarrow \text{address}$$

\begin{verbatim}
CASE inst
  OF bra: adddbuf,
    bcc  CASE TRUEBIT(cc,da/16r01)
        OF i: adddbuf
        ELSE pc
        ESAC,
    bcs: CASE TRUEBIT(cc,da/16r01)
        OF h adddbuf
        ELSE pc
        ESAC,
    beq  CASE TRUEBIT(cc,da/16r04)
        OF h adddbuf
        ELSE pc
        ESAC,
    bge CASE (TRUEBIT(cc,da/16r08),TRUEBIT(cc,da/16r02))
        OF (h,h): (l,l). adddbuf
        ELSE pc
        ESAC,
    bgt CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r08),TRUEBIT(cc,da/16r02))
        OF (l,h,b):(l,l) adddbuf
        ELSE pc
        ESAC,
    bhi CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r01))
        OF (l,l) adddbuf
        ELSE pc
        ESAC,
    blt CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r08),TRUEBIT(cc,da/16r02))
        OF (l,h,b):(l,l) pc
        ELSE adddbuf
        ESAC,
    bls CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r01))
        OF (l,l) pc
        ELSE adddbuf
        ESAC,
    bmi CASE TRUEBIT(cc,da/16r08)
        OF h adddbuf
        ELSE pc
        ESAC,
    bne CASE TRUEBIT(cc,da/16r04)
        OF l adddbuf
        ELSE pc
        ESAC,
\end{verbatim}
bvc: CASE TRUEBIT(cc,da/16r02)
    OF l: addrbuf
    ELSE pc
    ESAC,

bvs: CASE TRUEBIT(cc,da/16r02)
    OF h. addrbuf
    ELSE pc
    ESAC,

bpl: CASE TRUEBIT(cc,da/16r08)
    OF l: addrbuf
    ELSE pc
    ESAC

ESAC.
# Instruction Decoder#

FN ADMODE = (halfbyte input) -> addrmode
CASE input
  OF hb/16r0|hb/16r1|hb/16r3|hb/16r4|hb/16r5 implied,
      hb/16r2: relative,
      hb/16r6;hb/16ra;hb/16re index,
      hb/16r7;hb/16rb;hb/16rf. extnd,
      hb/16r8;hb/16rc: immed,
      hb/16r9;hb/16rd direct
ESAC

FN MNEMONIC = (halfbyte: input, halfbyte: mode) -> (mnemonic)
CASE input

OF hb/16r0: CASE mode
  OF hb/16r1    sba,
      hb/16r2    bra,
      hb/16r3    tsx,
      hb/16r4    nega,
      hb/16r5    negb,
      hb/(16r6 16r7) neg,
      hb/(16r8 16rb) suba,
      hb/(16rc 16rf) subb
ESAC,

hb/16r1: CASE mode
  OF hb/16r1    cba,
      hb/16r3    uns,
      hb/(16r8 16rb). cmpa,
      hb/(16rc 16rf) cmpb
ESAC,

hb/16r2 CASE mode
  OF hb/16r0    nop,
      hb/16r2    bhx,
      hb/16r3    pulx,
      hb/(16r8 16rb) sbcx,
      hb/(16rc 16rf) sbcb
ESAC,

hb/16r3 CASE mode
  OF hb/16r2    blx,
      hb/16r3    pulb,
      hb/16r4    coma,
      hb/16r5    comb,
      hb/(16r6 16r7) com
hb/16r4. CASE mode
  OF hb/16r2    bcc,
    hb/16r3    des,
    hb/16r4    lsla,
    hb/16r5    lsrh,
    hb/(16r6..16r7) lsr,
    hb/(16r8..16rb) anda,
    hb/(16rc..16rf) azdb
  ESAC,

hb/16r5 CASE mode
  OF hb/16r2    bcs,
    hb/16r3: tcs,
    hb/(16r8..16rb) bita,
    hb/(16rc..16rf) btb
  ESAC,

hb/16r6 CASE mode
  OF hb/16r0    tap,
    hb/16r1    tab,
    hb/16r2: bne,
    hb/16r3: psha,
    hb/16r4: rora,
    hb/16r5: rorb,
    hb/(16r6 16r7) ror,
    hb/(16r8 16rb) ldaa,
    hb/(16rc 16rf) ldab
  ESAC,

hb/16r7 CASE mode
  OF hb/16r0    tpa,
    hb/16r1    tba,
    hb/16r2    bcoe,
    hb/16r3    psha,
    hb/16r4    asra,
    hb/16r5    asrb,
    hb/(16r6 16r7) asr,
    hb/(16r9 16rb) staa,
    hb/(16rd 16rf) stab
  ESAC,

hb/16r8 CASE mode
  OF hb/16r0    uax,
    hb/16r2    bvc,
    hb/16r4    asla,
    hb/16r5    aslb,
    hb/(16r6 16r7) asl,
    hb/(16r8 16rb) eora,
    hb/(16rc 16rf) eorb
  ESAC,
hb/16r9. CASE mode
   OF hb/16r0: dex,
       hb/16r1: daa,
       hb/16r2: bvs,
       hb/16r3: rts,
       hb/16r4: rola,
       hb/16r5: rolb,
       hb/(16r6.16r7): rol,
       hb/(16r8.16rb): adca,
       hb/(16rc..16rf): adcb
   ESAC,

hb/16ra CASE mode
   OF hb/16r0: clv,
       hb/16r2: bpl,
       hb/16r4: deca,
       hb/16r5: decb,
       hb/(16r6.16r7) dec,
       hb/(16r8.16rb) oraa,
       hb/(16rc 16rf) orab
   ESAC,

hb/16rb CASE mode
   OF hb/16r0: sev,
       hb/16r1: aba,
       hb/16r2: bnu,
       hb/16r3: rts,
       hb/(16r8..16rb) adda,
       hb/(16rc 16rf) addb
   ESAC,

hb/16rc CASE mode
   OF hb/16r0: clc,
       hb/16r2: bc,c,
       hb/16r4: inca,
       hb/16r5: incb,
       hb/(16r6.16r7) inc,
       hb/(16r8 16rb) cpv
   ESAC,

hb/16rd CASE mode
   OF hb/16r0: sec,
       hb/16r2: bht,
       hb/16r4: tsta,
       hb/16r5: tsbh,
       hb/(16r6.16r7) tst,
       hb/16r8: bsr,
       hb/(16ra.16rb) jsr
   ESAC,
bb/16re CASE mode
   OF bb/16r0: cli,
   bb/16r2: bgt,
   bb/16r3: wai,
   bb/(16r6..16r7): jmp,
   bb/(16r8..16rb): lds,
   hb/(16rc. 16rf) ldx
ESAC,

bb/16rf. CASE mode
   OF bb/16r0: sei,
   bb/16r2: ble,
   bb/16r3: swa,
   bb/16r4: clra,
   bb/16r5: clrb,
   bb/(16r6..16r7): clr,
   bb/(16r9.16rb): sts,
   hb/(16rd 16rf) stx
ESAC
# Counting and Type-Swapping Functions#

**FN COUNTSET** = (count: a b) -> count: ARITH a + b

**FN HOWLONG** = (mnemonic function, addrmode mode) -> count.

**COUNTSET** (CASE function

OF jmp co/0,
  staa|stab|clra|clrb|com|comb|neg|negb|deca|decb|inc|incb|rola|rolb|
  rora|rorb|asl|aslb|asra|asrb|lsra|lsrb|tsa|tsrb|dp|x|idx|lds|td|tab
  |aba|cba|daa|nop|clc|cli|clv|sec|sec|sei|tap|ipa co/2,
  clr|com|neg|de|inc|rol|ror|as|asl|ls|lsr|t|co/3,
  psha|sh|plt|plt|pdx|dx|ms|ms|bs|ts|co/4,
  bsr|rts co/5,
  jsr: CASE mode OF extd co/6 ELSE co/4 ESAC,
  wai co/9,
  rti co/10,
  swi|nc|ir co/12
ELSE co/1
ESAC,

CASE mode

OF imm co/1,
  direct co/3,
  index co/4,
  extd: relative co/3
ELSE co/0
ESAC)

**FN COUNTDOWN** = (count input) -> count ARITH input - 1

**FN COMPARECOUNT** = (count input1 input2) -> count
  ARITH IF (input1-input2) > 0 THEN input1 - input2 ELSE 0 FI

**FN INCADDR** = (address input) -> address ARITH IF input = 65535 THEN 0 ELSE input + 1 FI

**FN HINYBBLE** = (data input) -> halfbyte ARITH (input LAND 16ff) SR 4

**FN LONYBBLE** = (data input) -> halfbyte ARITH (input LAND 16ff)

**FN MAKEADDRESS** = (data input1 input2) -> address ARITH 256*input1 + input2

**FN MADE DATA** = (halfbyte input1 input2) -> data ARITH 16*input1 + input2

**FN HIBYTE** = (address input) -> data ARITH (input LAND 16ff000) SR 8

**FN LOBYTE** = (address input) -> data ARITH (input LAND 16r00ff)
# Main Microprocessor Control Logic

\[
\text{FN UP6800} = (\text{data input, flag reset birq bnmj halt}) \rightarrow (\text{data, address, flag})
\]

BEGIN SEQ

PVAR a := ?data,
    b := ?data,
    x := ?address,
    sp := ?address,
    pc := ?address,
    cc := ?data,
    writeenable := ?flag,
    addrbus := ?address,
    addrbuf := ?address,
    databus := ?data,
    intercount := ?count,
    admode := ?admode,
    inst := ?mnemonic;

CASE halt
    OF b
    #-----------------------------------reset circuit-----------------------------------#

    ( CASE reset
        OF I (pc, admode, inst, intercount, cc)
            := (ad/16fffe, extnd, jmp, co/4, da/16rd0)
        ESAC,

    #----------------------------------housekeeping----------------------------------#
    databus = input,
    writeenable = 1,
    intercount := COUNTDOWN intercount,

    #------------------------hardware interrupt---------------------------------------#

CASE intercount
    OF co/0 (CASE bnmj
        OF 1 (inst = nnn, intercount = co/12)
        ESAC,
        CASE (birq, TRUEBIT (cc, da/16))
        OF (L1) (inst = irq, intercount = co/12)
        ESAC)
    ESAC),
    co/1 CASE inst
    OF wa1 (CASE bnmj
        OF 1 (inst = nnn; intercount = co/4)
        ESAC,
        CASE (birq, TRUEBIT (cc, da/16))
        OF (L1) (inst = irq; intercount = co/4)
        ESAC)
    ESAC,

ESAC
#------------------treat databus as next instruction------------------#

(inst,admode) : = CASE intercount
    OF co/0:(MNEMONIC (LONYBBLE databus, HINYBBLE input), ADMODE HINYBBLE databus)
    ELSE (inst,admode)
    ESAC;

(intercount,pc) : = CASE intercount
    OF co/0: (HOWLONG(inst,admode), INCADDR addrbus)
    ELSE (intercount,pc)
    ESAC;

#------------------MAIN PROCEDURE------------------#

CASE COMPARECOUNT (intercount, HOWLONG (inst, implied))
    OF co/4
        CASE admode
            OF index (addrbus = pc, pc = INCADDR addrbus)
            ESAC,
            co/3
        CASE admode
            OF exted (addrbus = pc, pc = INCADDR INCADDR addrbus),
            index addrbuf = x,
            relative (addrbus = pc, pc = INCADDR addrbus)
            ESAC,
            co/2
        CASE admode
            OF direct (addrbus = pc, pc = INCADDR addrbus),
            index addrbuf =
            MAKEADDRESS(LOBYTE x(ALU(adca, LOBYTE x, databus, da/0)][1]),
            exted (addrbuf,addrbus) =
            (MAKEADDRESS (databus,da/0),INCADDR addrbus),
            relative addrbuf =
            MAKEADDRESS(da/0,(ALU(addr,databus,LOBYTE pc, da/0)][1])
            ESAC,
            co/1
        CASE admode
            OF unmed (addrbus = pc, pc = INCADDR addrbus),
            direct addrbus = MAKEADDRESS (da/16r0,databus),
            index (addrbuf =
            MAKEADDRESS ((ALU (adca, HIBYTE x,
            CASE TRUEBIT (databus,da/16r80)
            OF h da/16rff
            ELSE da/16r00
            ESAC,
            (ALU (addr,databus,LOBYTE x, da/0)][2])][1],
            LOBYTE addrbuf),addrbus = addrbuf),
            exted addrbus = MAKEADDRESS (HIBYTE addrbuf, databus),
relative: addrbuf :=
MAKEADDRESS ((ALU (adca, HIBYTE pc,
CASE TRUEBIT (databus, da/16r80)
OF h. da/16rff
ELSE da/16r00
ESAC,
(ALU (adda, databus, LOBYTE pc, da/0))[2])())[1],
LOBYTE addrbuf)

ESAC
co/0 CASE COMPARECOUNT (HOWLONG (inst, implied), intercount)
OF co/0
CASE
inst
OF adda; adca; anda; bita; clra; cmpa; coma; deca; eora; lmaa; tpa;
inca; nega; oraa; rola; rora; asla; asra; Isra; sbca; daa; suba; tsta;
(a, cc, addrbus) =
BEGIN
LET acc1 = ALU (inst, a, databus, cc)
OUTPUT (acc1[1], acc1[2], pc)
END,
staa (databus, cc, writeenable) =
BEGIN
LET acc2 = ALU (inst, databus, a, cc)
OUTPUT (acc2[1], acc2[2], b)
END,
addb; adcb; andb; bitb; clrb; cmpb; comb; decb; eorbb; ldab;
incb; negb; orab; rolb; rorbb; aslbb; asrb; lsrb; sbcb; subbb; tsbb
(b, cc, addrbus) =
BEGIN
LET bcc1 = ALU (inst, b, databus, cc)
OUTPUT (bcc1[1], bcc1[2], pc)
END,
stab (databus, cc, writeenable) =
BEGIN
LET bcc2 = ALU (inst, databus, b, cc)
OUTPUT (bcc2[1], bcc2[2], b)
END,
abaa; cba; tba (a, cc) =
BEGIN
LET acc3 = ALU (inst, a, b, cc)
OUTPUT (acc3[1], acc3[2])
END,
tab. (b, cc) =
BEGIN
LET acc4 = ALU (inst, b, a, cc)
OUTPUT (acc4[1], acc4[2])
END,
tst| clr\com\neg|dec\inc\rol\ror\asl|asr|lsr:
 \hline
 (databus, cc, writeenable) :=
 \begin{align*}
 \text{BEGIN} \\
 \text{LET } dbcc1 = \text{ALU}(\text{inst}, a, \text{databus}, \text{cc}). \\
 \text{OUTPUT } (dbcc1[1], dbcc1[2], b) \\
 \text{END},
\end{align*}
cle|cl|clv|sec|sei|sev|tap.
 \begin{align*}
 \text{cc} := (\text{ALU}(\text{inst}, a, \text{databus}, \text{cc}))[2],
 \text{bra|bce|bes|beq|bge|bhi|ble|bli|bmi|bne|bve|bvs} \\
 \text{bpl addrbus} := \text{BRANCH} (\text{inst}, \text{addrbuf}, \text{pc}, \text{cc}),
\end{align*}
cpx.
 \begin{align*}
 (\text{addrbus} := \text{INCADDR addrbus}, \\
 \text{pc} := \text{INCADDR pc}, \\
 \text{cc} := (\text{ALU}(\text{cpx}, \text{HIBYTE x}, \text{databus}, \text{cc}))[2]),
\end{align*}
ldx:
 \begin{align*}
 (\text{addrbus} := \text{INCADDR addrbus}, \\
 \text{pc} := \text{CASE admode} \\
 \hline
 \text{OF imm|} \text{ INCADDR pc} \\
 \hline
 \text{ELSE pc} \\
 \hline
 \text{ESAC,}
\end{align*}
 \begin{align*}
 (x, cc) = \\
 \text{BEGIN} \\
 \text{LET } xcc = \text{ALU}(\text{inst}, d, 0, \text{databus}, \text{cc}) \\
 \text{OUTPUT } (\text{MAKEADDRES}x(xcc[1], d, 0), xcc[2]) \\
 \text{END),}
\end{align*}
unx|dex
 \begin{align*}
 (x, cc) = \\
 \text{BEGIN} \\
 \text{LET } xcc1 = \text{ALU}(\text{inst}, \text{LOBYTE x, databus}, \text{cc}) \\
 \text{OUTPUT } (\text{MAKEADDRES}x(xcc1[1], \text{HIBYTE x, xcc1[2]}) \\
 \text{END),}
\end{align*}
ins|des:
 \begin{align*}
 (sp, cc) = \\
 \text{BEGIN} \\
 \text{LET } spcc1 = \text{ALU}(\text{inst}, \text{LOBYTE sp, databus}, \text{cc}) \\
 \text{OUTPUT } (\text{MAKEADDRES}x(spcc1[1], spcc1[2]) \\
 \text{END),}
\end{align*}
lds
 \begin{align*}
 (\text{addrbus} = \text{INCADDR addrbus}, \\
 \text{pc} := \text{INCADDR pc}, \\
 (sp, cc) = \\
 \text{BEGIN} \\
 \text{LET } spcc2 = \text{ALU}(\text{inst}, d, 0, \text{databus}, \text{cc}) \\
 \text{OUTPUT } (\text{MAKEADDRES}x(spcc2[1], d, 0), spcc2[2]) \\
 \text{END),}
\end{align*}
stx
\begin{align*}
((\text{databus, cc}) = \\
 \text{BEGIN} \\
 \text{LET } xcc4 = \text{ALU}(\text{stx, databus, HIBYTE x, cc}) \\
 \text{OUTPUT } (xcc4[1], xcc4[2]) \\
 \text{END, writeenable := b}),
\end{align*}
sts:((databus, cc) =
BEGIN
    LET spcc3 = ALU(sts, databus,HIBYTE sp,cc)
    OUTPUT (spcc3[1],spcc3[2])
END, writeenable = h),

tsx (x,cc) :=
BEGIN
    LET xcc6 = ALU(ins, LOBYTE sp,databus,cc).
    OUTPUT(MAKEADDRESS(HIBYTE x, xcc6[1]),xcc6[2])
END,

tsx (sp,cc) :=
BEGIN
    LET xcc7 = ALU(des, LOBYTE x,databus,cc).
    OUTPUT(MAKEADDRESS(HIBYTE x, xcc7[1]),xcc7[2])
END,

psba (databus,addrbus,writeenable) := (a,sp,h),
psbb (databus,addrbus,writeenable) := (b,sp,h),
pula:pull' (sp,cc) :=
BEGIN
    LET spcc4 = ALU(ins, LOBYTE sp,databus,cc)
    OUTPUT(MAKEADDRESS(HIBYTE sp, spcc4[1]),spcc4[2])
END,

jsr:bsr (addrbuf = pc;
    pc = addrbus, addrbus = sp, databus =
    LOBYTE (addrbuf), writeenable = h, (sp,cc) =
BEGIN
    LET spcc5 = ALU(des, LOBYTE sp,databus,cc)
    OUTPUT(MAKEADDRESS(HIBYTE sp, spcc5[1]),spcc5[2])
END),

swi:irq:int:wa:
    (addrbuf = pc, addrbus = sp, databus =
    LOBYTE (addrbuf), writeenable = h, (sp,cc) =
BEGIN
    LET spcc42 = ALU(des, LOBYTE sp,databus,cc)
    OUTPUT(MAKEADDRESS(HIBYTE sp, spcc42[1]),spcc42[2])
END),
rts:rtu (addrbus = sp, (sp,cc) =
BEGIN
    LET spcc6 = ALU(ins, LOBYTE sp,databus cc).
    OUTPUT(MAKEADDRESS(HIBYTE sp,spcc6[1]),spcc6[2])
END)

ESAC,
co/1

CASE inst
OF stx|sts:addrbus := INCADDR addrbus,
clr|clrb|coma|comb:nega|negb|deca|decb|inca|mcb|rola|
rolb|rorb|asla|aslb|asra|asrb|lsra|lsrb|tsa|tstb|aba
|daa|tba|tab|cba|clc|clv|sel|selv|tap|tps|nop:
addrbus := pc,
cpx|lds:
(addrbus := pc,
(x,cc) =
BEGIN
LET xcc2 = ALU(inst,LOBYTE x,databus,cc).
OUTPUT(MAKEADDRESS(HIBYTE x, xcc2[1]), xcc2[2])
END,
lds
(addrbus := pc,
(sp,cc) =
BEGIN
LET spcc7 = ALU(mst,LOBYTE sp,databus,cc).
OUTPUT(MAKEADDRESS(HIBYTE sp, spcc7[1]), spcc7[2])
END),
psba|psbb (sp,cc) =
BEGIN
LET spcc8 = ALU(des,LOBYTE sp,databus,cc)
OUTPUT(MAKEADDRESS(HIBYTE sp, spcc8[1]), spcc8[2])
END,
pula|pulb
((sp,cc) =
BEGIN
LET spcc9 = ALU(ins,HIBYTE sp,LOBYTE sp,cc).
OUTPUT(MAKEADDRESS(spcc9[1],LOBYTE sp), spcc9[2])
END, addrbus := sp),
dex|inx
(x,cc) =
BEGIN
LET xcc3 = ALU(inst,HIBYTE x,LOBYTE x,cc)
OUTPUT(MAKEADDRESS(xcc3[1],LOBYTE x), xcc3[2])
END,
des|ins
(sp,cc) =
BEGIN
LET spcc10 = ALU(ins,HIBYTE sp,LOBYTE sp,cc).
OUTPUT(MAKEADDRESS(spcc10[1],LOBYTE sp), spcc10[2])
END,
tsx
(x,cc) =
BEGIN
LET xcc8 = ALU(ins,HIBYTE sp, LOBYTE x,cc)
OUTPUT(MAKEADDRESS(xcc8[1],LOBYTE x), xcc8[2])
END,
(sp, cc) :=
   BEGIN
      LET xcc9 = ALU(des,HIBYTE x, LOBYTE sp,cc).
      OUTPUT(MAKEADDRESS(xcc9[1],LOBYTE sp),xcc9[2])
   END,
   staa|stab addrbus := pc,
   jsr|bsr|sw|irq|nmi|war ((sp,cc) :=
   BEGIN
      LET spcc11 = ALU(des,HIBYTE sp,LOBYTE sp,cc).
      OUTPUT(MAKEADDRESS(spcc11[1],LOBYTE sp),spcc11[2])
   END;
   addrbus := sp; databus := HIBYTE addrbuf;
   writeenable := b;
   (sp,cc) :=
   BEGIN
      LET spcc12 = ALU(des, LOBYTE sp, databus, cc)
      OUTPUT(MAKEADDRESS(HIBYTE sp, spcc12[1]),spcc12[2])
   END),
   rts|ru: ((sp,cc) =
   BEGIN
      LET spcc13 = ALU(ins, HIBYTE sp, LOBYTE sp, cc)
      OUTPUT(MAKEADDRESS(spcc13[1], LOBYTE sp),spcc13[2])
   END, addrbus = sp, (sp, cc) =
   BEGIN
      LET spcc14 = ALU(ins, LOBYTE sp, databus, cc).
      OUTPUT(MAKEADDRESS(HIBYTE sp, spcc14[1]),spcc14[2])
   END)
ESAC.
CASE inst
OF inst((databus, cc) :
BEGIN
  LET xcc5 = ALU(stx, databus, LOBYTE x, cc).
  OUTPUT(xcc5[1], xcc5[2])
END;
writeable := h,
tst|clr|com|neg|dec|inc|rol|ror|asl|asr|lsr
addrbus := pc,
psha|psbb: (sp, cc) :=
BEGIN
  LET spec15 = ALU(des, HIBYTE sp, LOBYTE sp, cc)
  OUTPUT(MAKEADDRESS(spec15[1], LOBYTE sp), spec15[2])
END,
pula: a = databus,
pulb: b = databus,
jsr|bar: (sp, cc) :=
BEGIN
  LET spec16 = ALU(mn, HIBYTE sp, LOBYTE sp, cc)
  OUTPUT(MAKEADDRESS(spec16[1], LOBYTE sp), spec16[2])
END,
rts ((sp, cc) =
BEGIN
  LET spec17 = ALU(ms, HIBYTE sp, LOBYTE sp, cc)
  OUTPUT(MAKEADDRESS(spec17[1], LOBYTE sp), spec17[2])
END;
addrbus := sp;
pc := MAKEADDRESS(databus, da/0)),
swi|srq|nmv|wai
((sp, cc) =
BEGIN
  LET spec18 = ALU(des, HIBYTE sp, LOBYTE sp, cc)
  OUTPUT(MAKEADDRESS(spec18[1], LOBYTE sp), spec18[2])
END,
addrbus = sp, databus = LOBYTE x,
writeable = h,
(sp, cc) =
BEGIN
  LET spec19 = ALU(ins, LOBYTE sp, databus, cc)
  OUTPUT(MAKEADDRESS(HIBYTE sp, spec19[1]), spec19[2])
END),
rts ((sp, cc) =
BEGIN
  LET spec20 = ALU(ms, HIBYTE sp, LOBYTE sp, cc)
  OUTPUT(MAKEADDRESS(spec20[1], LOBYTE sp), spec20[2])
END, addrbus = sp,
cc = databus,
(sp, cc) :=
BEGIN
  LET spec21 = ALU(ins, LOBYTE sp, databus, cc)
  OUTPUT(MAKEADDRESS(HIBYTE sp, spec21[1]), spec21[2])
END)
ESAC,
co/3:

CASE inst OF
  rts; stx; psha; pula; psll; pulb; dex; des; ins; txs; tsc
    addrbus := pc,
  rts: pc := MAKEADDRESS(HIBYTE pc, databus),
  jsr: bsr: addrbus := pc,
  swi; irq; nmi; wai ((sp, cc) :=
    BEGIN
      LET spcc22 = ALU(des, HIBYTE sp, LOBYTE sp, cc).
      OUTPUT(MAKEADDRESS(spcc22[1], LOBYTE sp), spcc22[2])
    END,
    addrbus := sp; databus := HIBYTE x; writeenable := h; (sp, cc) :=
    BEGIN
      LET spcc23 = ALU(des, LOBYTE sp, databus, cc).
      OUTPUT(MAKEADDRESS(HIBYTE sp, spcc23[1]), spcc23[2])
    END)
  rtu. ((sp, cc) =
    BEGIN
      LET spcc24 = ALU(ins, HIBYTE sp, LOBYTE sp, cc).
      OUTPUT(MAKEADDRESS(spcc24[1], LOBYTE sp), spcc24[2])
    END,
    addrbus := sp; databus := a,
    writeenable := h,
    (sp, cc) =
    BEGIN
      LET spcc25 = ALU(ins, LOBYTE sp, databus, cc)
      OUTPUT(MAKEADDRESS(HIBYTE sp, spcc25[1]), spcc25[2])
    END)
End,

ESAC,

co/4

CASE inst OF
  rts; addrbus := pc,
  swi; irq; nmi; wai ((sp, cc) =
    BEGIN
      LET spcc26 = ALU(des, HIBYTE sp, LOBYTE sp, cc).
      OUTPUT(MAKEADDRESS(spcc26[1], LOBYTE sp), spcc26[2])
    END,
    addrbus := sp; databus := a,
    writeenable := h,
    (sp, cc) =
    BEGIN
      LET spcc27 = ALU(des, LOBYTE sp, databus, cc)
      OUTPUT(MAKEADDRESS(HIBYTE sp, spcc27[1]), spcc27[2])
    END)
  rtu. ((sp, cc) =
    BEGIN
      LET spcc28 = ALU(ins, HIBYTE sp, LOBYTE sp, cc)
      OUTPUT(MAKEADDRESS(spcc28[1], LOBYTE sp), spcc28[2])
    END,
    addrbus := sp,
    a = databus,
    (sp, cc) =
    BEGIN
      LET spcc29 = ALU(ins, LOBYTE sp, databus, cc)
      OUTPUT(MAKEADDRESS(HIBYTE sp, spcc29[1]), spcc29[2])
    END)
End,

ESAC,
CASE inst
OF swi, irq, nmi, wai:

((sp, cc)) :=
BEGIN
   LET spcc30 = ALU(des, HIBYTE sp, LOBYTE sp, cc).
   OUTPUT(MAKEADDRESS(spcc30[1], LOBYTE sp), spcc30[2])
END,
addrbus := sp; databus := b; writeenable := h,
(sp, cc) :=
BEGIN
   LET spcc31 = ALU(des, LOBYTE sp, databus, cc)
   OUTPUT(MAKEADDRESS(HIBYTE sp, spcc31[1]), spcc31[2])
END,
addrbus := sp;
databus := cc;
writeenable := h,
(sp, cc) :=
BEGIN
   LET spcc32 = ALU(ins, HIBYTE sp, LOBYTE sp, cc)
   OUTPUT(MAKEADDRESS(spcc32[1], LOBYTE sp), spcc32[2])
END;
addrbus := sp; x := MAKEADDRESS(databus, da/0);
(sp, cc) :=
BEGIN
   LET spcc33 = ALU(ins, LOBYTE sp, databus, cc).
   OUTPUT(MAKEADDRESS(HIBYTE sp, spcc33[1]), spcc33[2])
END)
ESAC,
co/6
CASE inst
OF swi, irq, nmi, wai:

((sp, cc)) :=
BEGIN
   LET spcc34 = ALU(des, HIBYTE sp, LOBYTE sp, cc).
   OUTPUT(MAKEADDRESS(spcc34[1], LOBYTE sp), spcc34[2])
END,
addrbus := sp; databus := cc;
writeenable := h,
(sp, cc) :=
BEGIN
   LET spcc35 = ALU(des, LOBYTE sp, databus, cc)
   OUTPUT(MAKEADDRESS(HIBYTE sp, spcc35[1]), spcc35[2])
END;
addrbus := sp;
x := MAKEADDRESS(HIBYTE x, databus),
(sp, cc) :=
BEGIN
   LET spcc36 = ALU(ins, HIBYTE sp, LOBYTE sp, cc).
   OUTPUT(MAKEADDRESS(spcc36[1], LOBYTE sp), spcc36[2])
END;
x := MAKEADDRESS(HIBYTE x, databus),
(sp, cc) :=
BEGIN
   LET spcc37 = ALU(ins, LOBYTE sp, databus, cc)
   OUTPUT(MAKEADDRESS(HIBYTE sp, spcc37[1]), spcc37[2])
END)
ESAC,
co/7:
CASE inst
OF swi|irq|nmi|wai
((sp,cc)  :=
BEGIN
LET spcc38 = ALU(des,HIBYTE sp,LOBYTE sp,cc).
OUTPUT(MAKEADDRESS(spcc38[1],LOBYTE sp),spcc38[2])
END; cc := (ALU(sei,da/0,da/0,cc))[2]),
rti: ((sp,cc) :=
BEGIN
LET spcc39 = ALU(ins,HIBYTE sp,LOBYTE sp,cc).
OUTPUT(MAKEADDRESS(spcc39[1],LOBYTE sp),spcc39[2])
END, addrbus := sp,
pc := MAKEADDRESS (databus,da/0))
ESAC,
co/8
CASE inst
OF swi addrbus = ad/16rfffa,
irq addrbus = ad/16rfffb,
nmu addrbus := ad/16rfffc,
rti pc := MAKEADDRESS (HIBYTE pc,databus),
wai intercount := co/2
ESAC,
co/9
CASE inst
OF swi:(pc := MAKEADDRESS(databus,da/0), addrbus = ad/16rfffb),
irq:(pc := MAKEADDRESS(databus,da/0), addrbus = ad/16rfffc),
nmu (pc := MAKEADDRESS(databus,da/0), addrbus = ad/16rfffd),
rti addrbus := pc
ESAC,
co/10
CASE inst
OF swi: irq|nmu pc := MAKEADDRESS(HIBYTE pc,databus)
ESAC,
co/11
addrbus = pc
ESAC
ESAC
ESAC
OUTPUT (databus,addrbus,writtenable)
END
#External Memory and Test Harness#

FN MAKERAM = (data,address,address,flag) -> data RAM (da/16r0).

FN DEL = (data) -> data: DELAY (da/16r0,1)

FN HARNESS = (flag:reset birq bmni, data hexf000 hexf001 hexf002) -> [3]data.

BEGIN
MAKE UP6800: micproc,
   DEL: delay olatch1 olatch2 olatch3,
   MAKERAM: ram

JOIN reset -> micproc[2],
   CASE micproc[2]
      OF ad/16rf000: hexf000,
      ad/16rf001: hexf001,
      ad/16rf002: hexf002
   ELSE ram
   ESAC -> delay,
   delay -> micproc[1],
   (micproc[1],micproc[2],micproc[2],micproc[3]) -> ram,
   birq -> micproc[3], #irq#
   bmni -> micproc[4], #nmi#
   h -> micproc[5], #halt#
   CASE micproc[2]
      OF ad/16re000: micproc[1]
      ELSE olatch1
   ESAC -> olatch1,
   CASE micproc[2]
      OF ad/16re001: micproc[1]
      ELSE olatch2
   ESAC -> olatch2,
   CASE micproc[2]
      OF ad/16re002: micproc[1]
      ELSE olatch3
   ESAC -> olatch3

OUTPUT (olatch1,olatch2,olatch3)
END
Appendix B: Programs

The following programs all work on my ELLA description of the 6800 microprocessor. They are held on disk by the ELLA group at DRA Malvern. To run any program the programs initialise and monitor must be stored as [6800]microinit.eli and [6800]monitor.eli since all the other programs call these.

Note:

i) co is a simulator command for comment. I have left them in for completeness.

ii) These programs have been modified slightly from when they were originally written because of changes to the reset circuitry. They run on the model described above.

iii) If you wish to see the microprocessor operating the following node names are useful to monitor:

- mcproc pc: program counter
- mcproc.admode: addressing mode
- mcproc.inst: instruction mnemonic
- mcproc.databus: value on the data bus
- mcproc.addrbus: value on the address bus
- mcproc.a: accumulator a
- mcproc.b: accumulator b
- mcproc.x: index register
- mcproc.sp: stack pointer
- mcproc.cc: condition code register

Initialise

co: This program performs a reset by setting parameter 1 low for 1 timestep, clears the a, b and x registers, co sets the stack pointer to &7fff, clears the condition codes register and jumps to location &000a simulate

CP 1 h b da/0 da/0 da/0

Uram ram [16r8000] da/16r4f, co reset clra

Uram ram [16r8001] da/16r5f, co clrb

Uram ram [16r8002] da/16r8e, co lds #&7fff

Uram ram [16r8003] da/16r7f,

Uram ram [16r8004] da/16rft,

Uram ram [16r8005] da/16rce, co ldx #&0000

Uram ram [16r8006] da/16r00,

Uram ram [16r8007] da/16r00,

Uram ram [16r8008] da/16r06, co tap

Ir0 ram [16r8009] da/16r7e, co jmp &000a

Ir0 ram [16r800a] da/16r00

Ir0 ram [16r800b] da/16r0a

Ir0 ram [16rfff] da/16r80, co define address to jump to on reset

Ir0 ram [16rfff] da/16r00

u + 1

CP 1] h
Monitor
co This program allows the online changing of a program by performing an nmi
cos (make input [3] low for 1 cycle with inputs [4, 6] holding the appropriate
co values). Input [4] controls the function as follows:
co da/0 advance counter and display with contents of that location
cos da/1 change contents of location to value of input [5]
co da/2 load counter with contents of input [5, 6]
cos da/3 execute your program starting at location defined by counter
cos To thus call the monitor type (from the simulator) cp [3.6] l da/2 da/<address, > da/<address, >, ti + 7,

in [6800] microinit
iram ram [16r9000] da/16rb6, co .mncitor ldaa &f000
iram ram [16r9001] da/16rf0
iram ram [16r9002] da/16r00
iram ram [16r9003] da/16r84, co anda #&0f
iram ram [16r9004] da/16r0f
iram ram [16r9005] da/16r81, co cmpa #&03
iram ram [16r9006] da/16r03
iram ram [16r9007] da/16r23, co bls .valid
iram ram [16r9008] da/16r01
iram ram [16r9009] da/16r3b, co rts
iram ram [16r900a] da/16r6c, co valid lds ##7fff
iram ram [16r900b] da/16r7f
iram ram [16r900c] da/16rff
iram ram [16r900d] da/16r81, co cmpa #&01
iram ram [16r900e] da/16r01
iram ram [16r900f] da/16r26, co bne .nochange
iram ram [16r9010] da/16r03
iram ram [16r9011] da/16rbd, co jsr .change
iram ram [16r9012] da/16r91
iram ram [16r9013] da/16r00
iram ram [16r9014] da/16r81, co .nochange cmpa #&01
iram ram [16r9015] da/16r01
iram ram [16r9016] da/16r22, co blh .noadv
iram ram [16r9017] da/16r01
iram ram [16r9018] da/16r08, co unx
iram ram [16r9019] da/16r81, co .noadv cmpa #&02
iram ram [16r901a] da/16r02
iram ram [16r901b] da/16r26, co bne .oldx
iram ram [16r901c] da/16r03
iram ram [16r901d] d:16rf, co lda 16f01
iram ram [16r901e] da/16r01
iram ram [16r901f] da/16r01
iram ram [16r9020] da/16r81, co .oldx cmpa #&03
iram ram [16r9021] da/16r03
iram ram [16r9022] da/16r26, co bne print
iram ram [16r9023] da/16r02
iram ram [16r9024] da/16r6e, co jmp #&00, x
iram ram [16r9025] da/16r00
iram ram [16r9026] da/16r6e, co .print ldam #&00, x
iram ram [16r9027] da/16r00
iram ram [16r9028] da/16r7f, co stab &e02
iram ram [16r9029] da/16r00
iram ram [16r902a] da/16r02
iram ram [16r902b] da/16rff, co stx &e000
Testprog1

This is a very simple program which prints out the numbers 1, 4, 7, by adding 3 to the previous number etc.

in [6800]monitor
iram ram [10] da/16r86, co .start ldaa #&01
iram ram [12] da/16r08, co adda #&03
iram ram [13] da/16r03
iram ram [14] da/16r7b, co staa &e000
iram ram [15] da/16r00
iram ram [16] da/16r00
iram ram [17] da/16r7e, co jmp &000c
iram ram [18] da/16r00
iram ram [19] da/16r0c
tabulated
mc
This program is the first in a series of Fibonacci sequence programs. It uses only the a register and only direct, extended and relative addressing. It was written to run on the early model which only had six instructions in [6800] monitor.

```
暗暗 ram [10] da/16r86, co .start ldaa #&00
暗暗 ram [11] da/16r00
暗暗 ram [12] da/16r97, co staa &00
暗暗 ram [13] da/16r00
暗暗 ram [14] da/16r97, co .loop staa &03
暗暗 ram [15] da/16r03
暗暗 ram [16] da/16r86, co ldaa #&01
暗暗 ram [17] da/16r01
暗暗 ram [18] da/16r97, co staa #&01
暗暗 ram [19] da/16r01
暗暗 ram [20] da/16r96, co ldaa &01
暗暗 ram [21] da/16r00
暗暗 ram [22] da/16r96, co adda
暗暗 ram [23] da/16r01
暗暗 ram [24] da/16r97, co staa &02
暗暗 ram [25] da/16r02
暗暗 ram [26] da/16r97, co staa &e000
暗暗 ram [27] da/16r00
暗暗 ram [28] da/16r00
暗暗 ram [29] da/16r96, co ldaa &01
暗暗 ram [30] da/16r01
暗暗 ram [31] da/16r97, co staa &00
暗暗 ram [32] da/16r00
暗暗 ram [33] da/16r96, co ldaa &02
暗暗 ram [34] da/16r02
暗暗 ram [35] da/16r97, co staa &01
暗暗 ram [36] da/16r01
暗暗 ram [37] da/16r96, co ldaa &03
暗暗 ram [38] da/16r03
暗暗 ram [39] da/16r8b, co adda #&01
暗暗 ram [40] da/16r01
暗暗 ram [41] da/16r97, co staa &03
暗暗 ram [42] da/16r03
暗暗 ram [43] da/16r80, cc suba #&0a
暗暗 ram [44] da/16r0a
暗暗 ram [45] da/16r27, co bne skip
暗暗 ram [46] da/16r03
暗暗 ram [47] da/16r7c, co jmp loop
暗暗 ram [48] da/16r00
暗暗 ram [49] da/16r14
暗暗 ram [50] da/16r7c, co jmp start
暗暗 ram [51] da/16r00
暗暗 ram [52] da/16r0a
```

This is a tabulated mc.
This Fibonacci program uses indexed addressing.

```
This Fibonacci program uses indexed addressing.

in [6800] monitor
iram ram [10] da/16r86, co  start ldaa #&00
iram ram [11] da/16r00,
iram ram [12] da/16r97, co staa &00
iram ram [13] da/16r00,
iram ram [14] da/16r97, co staa &03
iram ram [15] da/16r03,
iram ram [16] da/16r86, co ldaa #&01
iram ram [17] da/16r01,
iram ram [18] da/16r97, co staa &01
iram ram [19] da/16r01,
iram ram [20] da/16r06, co ldab #&0a
iram ram [21] da/16r0a,
iram ram [22] da/16r96, co .main ldaa &00
iram ram [23] da/16r00,
iram ram [24] da/16r9b, co adda &01
iram ram [25] da/16r01,
iram ram [26] da/16r97, co staa &02
iram ram [27] da/16r02,
iram ram [28] da/16r07, co staa &e000
iram ram [29] da/16r00,
iram ram [30] da/16r00,
iram ram [31] da/16rce, co ldx #&0001
iram ram [32] da/16r00,
iram ram [33] da/16r01,
iram ram [34] da/16ra6, co .loop ldaa &00,x
iram ram [35] da/16r00,
iram ram [36] da/16r09, co dex
iram ram [37] da/16ra7, co staa #00,x
iram ram [38] da/16r00,
iram ram [39] da/16r08, co inx
iram ram [40] da/16r08, co inx
iram ram [41] da/16r8c, co cpx #0003
iram ram [42] da/16r00,
iram ram [43] da/16r03,
iram ram [44] da/16r26, co bne .loop
iram ram [45] da/16r4,
iram ram [46] da/16r3a, co decb
iram ram [47] da/16r26, co bne .main
iram ram [48] da/16rce, da/16r7c, co jmp start
iram ram [49] da/16r00,
iram ram [50] da/16r0a,
iram ram [51] da/16r0a,
tabulated
mc
```
This program tests the x register and stack pointer, it places the ten Fibonacci numbers generated on the stack and outputs them in reverse order.

Initialiseram ram [10] da/16r86, co .start  ldaa ##&00
Initialiseram ram [11] da/16r00,
Initialiseram ram [12] da/16r97, co  staa &00
Initialiseram ram [13] da/16r00,
Initialiseram ram [14] da/16r97, co  staa &03
Initialiseram ram [15] da/16r03,
Initialiseram ram [16] da/16r8c, co  lds ##&0403
Initialiseram ram [17] da/16r04,
Initialiseram ram [18] da/16r03,
Initialiseram ram [19] da/16r86, co  ldaa ##&01
Initialiseram ram [20] da/16r01,
Initialiseram ram [21] da/16r97, co  staa &01
Initialiseram ram [22] da/16r01,
Initialiseram ram [23] da/16r6c, co  ldba &0a
Initialiseram ram [24] da/16r0a,
Initialiseram ram [25] da/16r96, co .main  ldaa &00
Initialiseram ram [26] da/16r00,
Initialiseram ram [27] da/16r9b, co  adda &01
Initialiseram ram [28] da/16r01,
Initialiseram ram [29] da/16r97, co  staa &02
Initialiseram ram [30] da/16r02,
Initialiseram ram [31] da/16r36, co  psha
Initialiseram ram [32] da/16rce, co  ldx ##&0001
Initialiseram ram [33] da/16r00,
Initialiseram ram [34] da/16r01,
Initialiseram ram [35] da/16r66, co loop  ldaa &00,x
Initialiseram ram [36] da/16r00,
Initialiseram ram [37] da/16r09, co  dex
Initialiseram ram [38] da/16r7, co  staa &00,x
Initialiseram ram [39] da/16r00,
Initialiseram ram [40] da/16r08, co  unx
Initialiseram ram [41] da/16r08, co  unx
Initialiseram ram [42] da/16r8c, co  cpx &0003
Initialiseram ram [43] da/16r00,
Initialiseram ram [44] da/16r03,
Initialiseram ram [45] da/16r26, co  bne loop
Initialiseram ram [46] da/16r4,
Initialiseram ram [47] da/16r5a, co  decb
Initialiseram ram [48] da/16r26, co  bne main
Initialiseram ram [49] da/16r7,
Initialiseram ram [50] da/16rce, co  ldx ##&000a
Initialiseram ram [51] da/16r00,
Initialiseram ram [52] da/16r0a,
Initialiseram ram [53] da/16r32, co loop2 pula
Initialiseram ram [54] da/16rb7, co  staa &e00
Initialiseram ram [55] da/16r00,
Initialiseram ram [56] da/16r00,
Initialiseram ram [57] da/16r09, co  dex
Initialiseram ram [58] da/16r26, co  bne loop2
Initialiseram ram [59] da/16r9,
Initialiseram ram [60] da/16r7e, co  jmp start
initialiseram ram \[61\] da/16r00,
initialiseram ram \[62\] da/16r0a,
tabulated
mc
This is another Fibonacci program which tests the jsr instruction in [6800] monitor.

initialiseram ram [10] da/16 r86, co .start ldaa #&00
initialiseram ram [11] da/16 r00, staa #00
initialiseram ram [12] da/16 r97, staa #03
initialiseram ram [13] da/16 r00, lda #0a
initialiseram ram [14] da/16 r97, lds #&0401
initialiseram ram [15] da/16 r03, initialiseram ram
initialiseram ram [16] da/16 r8e, co
initialiseram ram [17] da/16 r04, initialiseram ram
initialiseram ram [18] da/16 r01, initialiseram ram
initialiseram ram [19] da/16 r86, co ldaa #&01
initialiseram ram [20] da/16 r01, initialiseram ram
initialiseram ram [21] da/16 r97, co staa #01
initialiseram ram [22] da/16 r01, initialiseram ram
initialiseram ram [23] da/16 r06, co ldab #0a
initialiseram ram [24] da/16 r0a, initialiseram ram
initialiseram ram [25] da/16 r96, co main ldaa #00
initialiseram ram [26] da/16 r00, initialiseram ram
initialiseram ram [27] da/16 r9b, co adda #01
initialiseram ram [28] da/16 r01, initialiseram ram
initialiseram ram [29] da/16 r97, staa #02
initialiseram ram [30] da/16 r02, initialiseram ram
initialiseram ram [31] da/16 r07, staa #&0000
initialiseram ram [32] da/16 r0c0, initialiseram ram
initialiseram ram [33] da/16 r00, initialiseram ram
initialiseram ram [34] da/16 r36, co psha
initialiseram ram [35] da/16 rbd4, co jsr swap
initialiseram ram [36] da/16 r00, initialiseram ram
initialiseram ram [37] da/16 r35, initialiseram ram
initialiseram ram [38] da/16 r5a, co decb
initialiseram ram [39] da/16 r26, co bne main
initialiseram ram [40] da/16 r01, initialiseram ram
initialiseram ram [41] da/16 rce, co ldx #&000a
initialiseram ram [42] da/16 r06, initialiseram ram
initialiseram ram [43] da/16 r0a, initialiseram ram
initialiseram ram [44] da/16 r32, co loop2 pulax
initialiseram ram [45] da/16 r97, co staa #02
initialiseram ram [46] da/16 r02, initialiseram ram
initialiseram ram [47] da/16 r09, co dex
initialiseram ram [48] da/16 r26, co bne loop2
initialiseram ram [49] da/16 rfa, initialiseram ram
initialiseram ram [50] da/16 r7e, co jmp start
initialiseram ram [51] da/16 r00, initialiseram ram
initialiseram ram [52] da/16 r0a, initialiseram ram
initialiseram ram [53] da/16 rce, co .swap ldx #&0000
initialiseram ram [54] da/16 r00, initialiseram ram
initialiseram ram [55] da/16 r00, initialiseram ram
initialiseram ram [56] da/16 r86, co loop ldaa #01, x
initialiseram ram [57] da/16 r01, initialiseram ram
initialiseram ram [58] da/16 r7a, co staa #00, x
initialiseram ram [59] da/16 r00, initialiseram ram
initialiseram ram [60] da/16 r08, co inx
initialis r a m [61] da/16r8c, co
initialiser a m [62] da/16r00,
cpx &0002
initialiser a m [63] da/16r02,
initialiser a m [64] da/16r26, co
bne .loop
initialiser a m [65] da/16r16,
initialiser a m [66] da/16r39, co
rts
tabulated
mc
This is yet another Fibonacci program, which uses a software interrupt (swi) instead of the jsr in the last program (testprog5).

```assembly
un [6800]monitor
initialiseram ram [10] da/16:86, co .start ldaa &00
initialiseram ram [11] da/16:00,
initialiseram ram [12] da/16:97, co staa &00
initialiseram ram [13] da/16:00,
initialiseram ram [14] da/16:97, co staa &03
initialiseram ram [15] da/16:03,
initialiseram ram [16] da/16:8c, co lds &0401
initialiseram ram [17] da/16:04,
initialiseram ram [18] da/16:01,
initialiseram ram [19] da/16:86, co ldaa &01
initialiseram ram [20] da/16:01,
initialiseram ram [21] da/16:97, co staa &01
initialiseram ram [22] da/16:01,
initialiseram ram [23] da/16:06, co ldab &0a
initialiseram ram [24] da/16:0a,
initialiseram ram [25] da/16:96, co .main ldaa &00
initialiseram ram [26] da/16:00,
initialiseram ram [27] da/16:9b, co adda &01
initialiseram ram [28] da/16:01,
initialiseram ram [29] da/16:97, co staa &02
initialiseram ram [30] da/16:02,
initialiseram ram [31] da/16:87, co staa &c000
initialiseram ram [32] da/16:e0,
initialiseram ram [33] da/16:00,
initialiseram ram [34] da/16:36, co psha
initialiseram ram [35] da/16:3f, co swap
initialiseram ram [36] da/16:5a, co decb
initialiseram ram [37] da/16:26, co bne .main
initialiseram ram [38] da/16:27
initialiseram ram [39] da/16:ce, co ldx &000a
initialiseram ram [40] da/16:00,
initialiseram ram [41] da/16:0a,
initialiseram ram [42] da/16:32, co loop2 pula
initialiseram ram [43] da/16:97, co staa &02
initialiseram ram [44] da/16:02,
initialiseram ram [45] da/16:09, co dex
initialiseram ram [46] da/16:26, co bne loop2
initialiseram ram [47] da/16:fa,
initialiseram ram [48] da/16:7e, co jmp start
initialiseram ram [49] da/16:00,
initialiseram ram [50] da/16:0a,
initialiseram ram [51] da/16:ce, co swap ldx &0000
initialiseram ram [52] da/16:00,
initialiseram ram [53] da/16:00,
initialiseram ram [54] da/16a6, co loop ldaa &01, x
initialiseram ram [55] da/16:01,
initialiseram ram [56] da/16a7, co staa &00, x
initialiseram ram [57] da/16:00,
initialiseram ram [58] da/16:08, co inx
initialiseram ram [59] da/16:8c, co cpx &0002
```
initialiseram ram [60] da/16r00,
initialiseram ram [61] da/16r02,
initialiseram ram [62] da/16r26, co bne .loop
initialiseram ram [63] da/16rf6,
initialiseram ram [64] da/16r3b, co rti
initialiseram ram [16rfff] da/16r00, co define swi address
initialiseram ram [16rfffb] da/16r33, co .swap
tabulated
mc
Testprog?

This is an 19 byte program which generates the first 11 Fibonacci numbers. It operates by clearing one location on the stack to zero, then loading the a register with &01 and x with &000b. Accumulator b is cleared then pulled from the stack, then a is pushed to it. Register a is added to b with the result in a. x is decremented. If x is not zero then b is pulled from the stack, a is pushed to it etc. Otherwise the program waits for an interrupt. It is short because it was written to be efficient rather than to test specific instructions. However Testprog2 performs an identical function and I believe is optimised for the first 6 instructions implemented (see The Motorola 6800 Model: initial design) but is twice as long. This demonstrates that more complex operations are useful on a microprocessor.

in [6800] monitor

<table>
<thead>
<tr>
<th>IRAM RAM</th>
<th>DA/16</th>
<th>CO</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16r000a</td>
<td>da/16r34</td>
<td>co</td>
<td>start</td>
<td>des</td>
</tr>
<tr>
<td>16r000b</td>
<td>da/16r30</td>
<td>co</td>
<td>tss</td>
<td></td>
</tr>
<tr>
<td>16r000c</td>
<td>da/16r6f</td>
<td>co</td>
<td>clr &amp;00</td>
<td>x</td>
</tr>
<tr>
<td>16r000d</td>
<td>da/16r00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16r000e</td>
<td>da/16rce</td>
<td>co</td>
<td>ldx #&amp;000b</td>
<td></td>
</tr>
<tr>
<td>16r000f</td>
<td>da/16r00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16r0010</td>
<td>da/16r0b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16r0011</td>
<td>da/16r86</td>
<td>co</td>
<td>ldaa #&amp;01</td>
<td></td>
</tr>
<tr>
<td>16r0012</td>
<td>da/16r01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16r0013</td>
<td>da/16r33</td>
<td>co</td>
<td>.loop</td>
<td>pulb</td>
</tr>
<tr>
<td>16r0014</td>
<td>da/16r36</td>
<td>co</td>
<td>psha</td>
<td></td>
</tr>
<tr>
<td>16r0015</td>
<td>da/16r1b</td>
<td>co</td>
<td>aba</td>
<td></td>
</tr>
<tr>
<td>16r0016</td>
<td>da/16r7b</td>
<td>co</td>
<td>staa &amp;e00</td>
<td></td>
</tr>
<tr>
<td>16r0017</td>
<td>da/16re0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16r0018</td>
<td>da/16r00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16r0019</td>
<td>da/16r09</td>
<td>co</td>
<td>dex</td>
<td></td>
</tr>
<tr>
<td>16r001a</td>
<td>da/16r26</td>
<td>co</td>
<td>bne .loop</td>
<td></td>
</tr>
<tr>
<td>16r001b</td>
<td>da/16rf7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16r001c</td>
<td>da/16r3e</td>
<td>co</td>
<td>wait</td>
<td></td>
</tr>
</tbody>
</table>

tabulated
Appendix C: The Motorola 6800 Instruction Set

Provided here is a brief list of the 6800 assembler mnemonics and their functions. It is not intended to provide programming information, which may be obtained from The Motorola Data Sheet (see Bibliography)

Accumulator and Memory Instructions:

- **adda**: Add.  
  \[ a = a + m \]
- **addb**: Add.  
  \[ b = b + m \]
- **aba**: Add accumulators.  
  \[ a = a + b \]
- **adca**: Add with carry.  
  \[ a = a + m + c \]
- **adcb**: Add with carry.  
  \[ b = b + m + c \]
- **anda/andb**: And.  
  \[ a = a \text{ AND } m \]
- **bita/bitb**: Bit test.  
  \[ a = a \text{ AND } m \]
- **clra/clrb**: Clear.  
  \[ m = 0 / a = 0 / b = 0 \]
- **com**: Compare.  
  \[ a = m / b = m \]
- **crla/crle/crbb**: Clear accumulator.  
  \[ a = b \]
- **cmpa/cmpb**: Compare.  
  \[ a = 0 / b = 0 \]
- **cbs**: Compare accumulators.  
  \[ a = b \]
- **comic**: Complement.  
  \[ m = \text{FF} - a \]
- **neg/nega/negb**: 2's complement.  
  \[ a = 0 - s / b = 0 - b \]
- **dlaa**: Decimal adjust.  
  \[ a = a \pm 1 / b = b \]
- **dec/deca/decb**: Decrement.  
  \[ m = m - 1 / a = a - 1 / b = b - 1 \]
- **eora/eorb**: Exclusive or.  
  \[ a = b \text{ XOR } m \]
- **inc/inca/incipb**: Increment.  
  \[ a = m + 1 / a = a + 1 / b = b + 1 \]
- **ldaa/ldab**: Load accumulator.  
  \[ a = m / b = m \]
- **oraa/orab**: Inclusive or.  
  \[ a = a \text{ OR } m \]
- **psha/pshb**: Push to stack (LIFO).  
  \[ a = a / b \text{ from stack} \]
- **pula/pulb**: Pull from stack.  
  \[ a = a / b \text{ from stack} \]
- **rol/rola/rolb**: Rotate left.  
  \[ a = a \pm 1 / b = b \pm 1 \]
- **ror/ora/orb**: Rotate right.  
  \[ a = a \pm 1 / b = b \pm 1 \]
- **asl/asia/aslb**: Arithmetic shift left.  
  \[ a = a \text{ shift } m / a \]
- **asr/asa/asrb**: Arithmetic shift right.  
  \[ a = a \text{ shift } m / a \]
- **lsr/lsra/lsrb**: Logic shift right.  
  \[ a = a \text{ shift } m / a \]
- **staa/stab**: Store accumulator.  
  \[ m = a / m = b \]
- **suba/subb**: Subtract.  
  \[ a = a - m / b = b - m \]
- **sba**: Subtract accumulators.  
  \[ a = a - b \]
- **sbca/sbcb**: Subtract with carry.  
  \[ a = a - m - c / b = b - m - c \]
- **tab/tba**: Transfer accumulators.  
  \[ b = a / a = b \]
- **tsa/tsx**: Test, zero or minus.  
  \[ m = 0 / a = 0 / b = 0 \]

Index Register and Stack Manipulation Instructions

- **cpx**: Compare x to m, m + 1.  
  \[ x = x \pm 1 / sp = sp \pm 1 \]
- **dex/des**: Decrement.  
  \[ x = x + 1 / sp = sp + 1 \]
- **mx/ms**: Increment.  
  \[ x = x \pm 1 / sp = sp \pm 1 \]
- **ldx/lds**: Load register.  
  \[ x = m, x = (m + 1) / sp = m, sp = (m + 1) \]
- **stx/sts**: Store register.  
  \[ m = x, (m + 1) = x / m = sp, (m + 1) = sp \]
- **txs/tsx**: Transfer registers.  
  \[ sp = x \pm 1 / x = sp \pm 1 \]
Jump and Branch Instructions:

- `bra`: branch always
- `bcc`: branch if carry clear
- `bcs`: branch if carry set
- `beq`: branch if equals zero
- `bge`: branch if greater than or equals zero
- `bgt`: branch if greater than zero
- `bhi`: branch if higher
- `ble`: branch if less than or equals zero
- `bls`: branch if lower or same
- `blt`: branch if less than zero
- `bmi`: branch if minus
- `bne`: branch if not equal zero
- `bvc`: branch if overflow clear
- `bvs`: branch if overflow set
- `bpl`: branch if plus
- `bsr`: branch to subroutine
- `jmp`: jump
- `jsr`: jump to subroutine
- `nop`: no operation (takes a small amount of time)
- `rti`: return from interrupt
- `rts`: return from subroutine
- `swi`: software interrupt
- `wai`: wait for interrupt

Dummy Mnemonics Added to Simplify ELLA Program:

- `urq`: interrupt request
- `nmi`: non-maskable interrupt

Condition Code Register Manipulation Operations:

- `clc`: clear carry flag
- `ch`: clear interrupt mask
- `clv`: clear overflow flag
- `sec`: set carry flag
- `sei`: set interrupt mask
- `sev`: set overflow
- `tap`: transfer accumulator a into cc register
- `tpa`: transfer cc register into accumulator a

N.B. Many Mnemonics can have more than one addressing mode associated with them. There are altogether 197 valid mnemonic/addressing mode combinations plus `urq` and `nmi`, although there are only 107 such mnemonics listed (plus 2 dummy mnemonics).
**Glossary:**

- **Accumulator**: An accumulator is a **register** on which a large number of arithmetic and logical operations may be carried out.

- **Addressing mode**: The addressing mode tells the microprocessor where to get the data for its next operation. Six addressing modes are implemented on the Motorola 6800. These are immediate, direct, indexed, extended, implied and relative.

- **Assembler code**: Low level language which is made up of mnemonics plus other symbols to indicate the addressing mode.

- **Condition codes register**: A **register** which is affected by most arithmetic and logical functions, and also by interrupts. On the 6800 it is made up of 6 flags, which indicate when true (bit 5 first, bit 0 last) that: a half carry from bit 3 has occurred, interrupt requests are to be ignored, the result was negative, the result was zero, the operation caused a 2's complement overflow and the operation caused a carry.

- **Direct addressing**: The address of the data is given by the value in the next location after the op-code, (therefore it is in the range &0000 &00ff).

- **Extended addressing**: The address of the data is in the next two locations after the op-code (high byte first).

- **Immediate addressing**: The data comes from the next memory location after the op-code.

- **Implied addressing**: No data is necessary e.g. clra.

- **Indexed addressing**: The address of the data is the index register plus an offset which is in the next location after the op-code.

- **Interrupt**: An interrupt causes the microprocessor to stop what it is doing after finishing the current instruction, and to jump to a location which is defined at a particular location in memory. When the routine is finished (with an rti) it jumps back to where it was when the interrupt occurred, with the same values in all the registers as before the interrupt. This is achieved by dumping all the registers on the stack as the interrupt is initiated, and reading them back as the interrupt routine terminates. There are three ways of initiating an interrupt on the 6800, a software interrupt (swi), and two hardware interrupts.

- **Interrupt request**: If the interrupt mask bit in the condition codes register is zero, a zero on the interrupt request line causes an interrupt to occur.

- **Mnemonic**: A mnemonic is a three or four letter word which defines the operation to be performed by the microprocessor. The words are chosen to be user friendly and must be converted into op-codes before they can be stored in memory or used by the microprocessor.

- **Non-maskable interrupt**: A non-maskable interrupt causes an interrupt whatever the state of the condition codes register.
Op-code: Hexadecimal representation of an 8-bit number which is an instruction to a microprocessor. An op-code is unique, defining both the addressing mode and the mnemonic.

Register: A register is a store for information on the microprocessor. A limited number of arithmetic and logical functions are possible such as increment, decrement, or setting particular bits to be true or false.

Relative addressing: The address to branch to is the start of the next instruction plus an offset which comes from the location after the current op-code.

Software interrupt: A particular op-code is used from within the program to call the interrupt routine.

Stack: A stack is an area of memory used for storing data, which is accessed by the stack pointer. The stack pointer (sp) always points to the first available unused location. When a piece of data is pushed to the stack, it is placed at the memory location sp, and sp is then decremented. When a piece of data is pulled from the stack, the stack pointer is first incremented, then the data is read. The location defined by sp is now free for new data to be pushed, but still contains the old data. The stack is therefore a last-in first-out (LIFO) store.

Von Neumann microprocessor: A microprocessor in which the program and data share a common address space.
Bibliography:

1. The ELLA User Manual, Computer General ED


3. Motorola Microprocessor, Microcontroller and Peripheral Data Volume 1, Motorola Limited
This document starts by providing a brief overview of the ELLA language. The majority of the document describes the design and development of an abstract high level ELLA model of a Motorola 6800 Microprocessor, including all of the 197 codes which make up the 6800 instruction set.