THE FORMAL SPECIFICATION OF A MICROPROCESSOR INSTRUCTION SET

by

Jonathan Bowen

Technical Monograph PRG-60

January 1987

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Electronic mail: bowen@uk.ac.oxford.prg (JANET)

To Jane and Alice

"I only took the regular course ... the different branches of Arithmetic -- Ambition, Distraction, Uglification and Derision."

-- Lewis Carroll

The Formal Specification of a Microprocessor Instruction Set

Jonathan Bowen

Abstract

The specification language Z is used to define a microprocessor based system in a formal notation. The Motorola 6800 8-bit microprocessor is chosen as an example. Its simplicity allows the entire instruction set to be covered. Memory configuration and interrupts are also included. The use of a formal description language allows the possibility of verification of the instruction set. Additionally, the use of Z combined with informal text is sufficently readable for the specification to be used for documentation purposes.

Contents

1. Introduction

Currently, computer instruction sets are normally documented using tables, semi-formal formulae and informal text. This monograph attempts to show that they may be described just as easily and with more precision using formal specification methods. Microprocessors have been formally specified previously [1]. Often these specifications have been difficult to understand since they have not been designed for documentation purposes. The specification given here concentrates on presenting a specification which is readable by humans as well as computers.

In this monograph, the specification language Z [2-7], developed at the Programming Research Group, is used to define the instruction set for an 8-bit microprocessor, the Motorola 6800. As well as the instruction set, interrupts and memory configuration are also covered. Readers not familiar with the 6800 are referred to its programming manual [8] or instruction set summary card [9]. These may also be used as a comparison with the description given here.

It was felt that a complete microprocessor instruction set should be attempted in order to detect any possible weaknesses in the use of Z for such a task. The relatively simple 6800 processor was chosen because this allowed the entire instruction set of a real microprocessor to be specified. A processor such as one of the 68000 family was deliberately not selected for an initial attempt at such a specification since its greater complexity would either require a good deal more work, or for many features not to be included.

Some of the material covered here is generally useful for any microprocessor based system. Hence any subsequent specifications could draw on this groundwork.

2. Basic Concepts

2.1 Word organisation

Machines such as microprocessors generally manipulate bits. These are organised into non-zero length finite words. By convention, bit positions are numbered from zero up.

$$Bit \triangleq \{ 0, 1 \}$$
$$Word \triangleq \{ w : \mathbb{N} \pfun Bit \mid \#w > 0 \land dom\ w = 0..\#w-1 \}$$

Often the least significant bit (LSB) and most significant bit (MSB) of a word are of particular interest.

$$LSB, MSB : Word \to Bit$$

$$\forall w : Word \bullet$$
$$LSB\ w = w\ 0 \land$$
$$MSB\ w = w\ \#w-1$$

Each bit pattern in a word uniquely maps to a particular numerical value.

$$val : Word \to \mathbb{N}$$

$$\forall w : Word \bullet$$
$$\#w = 1 \Rightarrow val\ w = LSB\ w \land$$
$$\#w > 1 \Rightarrow val\ w = LSB\ w + 2 * val(succ \$ w)$$

It is sometimes useful to set all of the bits in a word to a particular value, whatever their previous value.

$$\_\ set\ \_ : (Word \times Bit) \to Word$$

$$\forall w : Word; b : Bit \bullet$$
$$w\ \underline{set}\ b = w \$ \{0 \mapsto b, 1 \mapsto b\}$$

A word contains its maximum unsigned value when all the bits are set to 1's.

$$
\begin{array}{|l}
\mathit{maxval} : \mathit{Word} \rightarrow \mathbb{N} \\
\hline
\forall w : \mathit{Word} \bullet \\
\quad \mathit{maxval}\ w = \mathit{val}(w \ \underline{\mathit{set}}\ 1)
\end{array}
$$

For convenience, we define a function to generate a word of particular size and value:

$$
\begin{array}{|l}
\mathit{wrd} : \mathbb{N}_1 \rightarrow \mathbb{N} \rightarrow \mathit{Word} \\
\hline
\forall \mathit{size} : \mathbb{N}_1; \mathit{value} : \mathbb{N}; w : \mathit{Word} \bullet \\
\quad \mathit{wrd}\ \mathit{size}\ \mathit{value} = w \Leftrightarrow \\
\qquad \#w = \mathit{size} \wedge \\
\qquad \mathit{val}\ w = \mathit{value} \ \underline{\mathit{mod}}\ \mathit{succ}(\mathit{maxval}\ w)
\end{array}
$$

Sometimes it is useful to concatenate words together since processors can often handle multiples of some base size of word. These two words may be of differing sizes for complete generality.

$$
\begin{array}{|l}
\_ \frown \_ : (\mathit{Word} \times \mathit{Word}) \rightarrow \mathit{Word} \\
\hline
\forall w_1, w_2 : \mathit{Word} \bullet \\
\quad w_1 \frown w_2 = w_1 \cup \mathit{pred}^{\#w_1} \$ w_2
\end{array}
$$

The number of bits in the resulting word is the sum of the number of bits in each of the words being concatenated:

$$\vdash \forall w_1, w_2 : \mathit{Word} \bullet \#(w_1 \frown w_2) = \#w_1 + \#w_2$$

The high and low halves of a word may be projected using two functions. These projections can be concatenated to form the original word.

$$
\begin{array}{|l}
\mathit{lo}, \mathit{hi} : \mathit{Word} \rightarrow \mathit{Word} \\
\hline
\forall w : \mathit{Word} \bullet \\
\quad \mathit{lo}(w) = (0..(\#w \ \underline{\mathit{div}}\ 2)-1) \lhd w \wedge \\
\quad \mathit{hi}(w) = \mathit{succ}^{(\#w \ \underline{\mathit{div}}\ 2)} \$ w
\end{array}
$$

$$\vdash \forall w : \mathit{Word} \bullet w = \mathit{lo}(w) \frown \mathit{hi}(w)$$

2.2 Bitwise functions

Bitwise logical functions involve individual bits. A bit may be complemented:

$$\sim : Bit \rightarrowtail\!\!\!\!\!\rightarrow Bit$$

$$\sim = \{0 \mapsto 1,\ 1 \mapsto 0\}$$

We can also AND, (inclusive) OR and (exclusive) XOR pairs of bits by providing the relevant truth table in each case:

$$\_ \bullet \_ ,\quad \_ + \_ ,\quad \_ \oplus \_ : (Bit \times Bit) \rightarrow Bit$$

$$\bullet = \{(0,0) \mapsto 0,\ (0,1) \mapsto 0,\ (1,0) \mapsto 0,\ (1,1) \mapsto 1\}$$
$$+ = \{(0,0) \mapsto 0,\ (0,1) \mapsto 1,\ (1,0) \mapsto 1,\ (1,1) \mapsto 1\}$$
$$\oplus = \{(0,0) \mapsto 0,\ (0,1) \mapsto 1,\ (1,0) \mapsto 1,\ (1,1) \mapsto 0\}$$

Most microprocessors allow bitwise logical operations on words. For instance, a word may be (1's) complemented i.e. all 0 bits are changed to 1's and all 1's are changed to 0's. This is sometimes referred to as a bitwise logical NOT operation. We can upgrade the definition for a bit to a function which applies to a word:

$$\sim : Word \rightarrowtail\!\!\!\!\!\rightarrow Word$$

$$\forall w : Word \bullet \sim w = w \,\mathbf{!}\, \sim$$

Many bitwise operations take pairs of bits as input (e.g. those described above).

$$WordPair \triangleq \{ w : \mathbb{N} \rightarrowtail\!\!\!\!\!\rightarrow (Bit \times Bit) \mid \#w > 0 \wedge dom\, w = 0..\#w - 1 \}$$

$$\_\ pair\ \_ : (Word \times Word) \rightarrow WordPair$$

$$\forall w_1, w_2 : Word \bullet w_1\ \underline{pair}\ w_2 = \{ i : \mathbb{N} \mid i \in dom\, w_1 \cap dom\, w_2 \bullet i \mapsto (w_1\, i, w_2\, i) \}$$

The corresponding pairs of bits in a pair of words may now be ANDed, ORed, and XORed, again by upgrading the equivalent bit functions:

$$
\begin{array}{|l}
\_ \bullet \_ \,, \\
\_ + \_ \,, \\
\_ \oplus \_ : (\mathit{Word} \times \mathit{Word}) \rightarrow \mathit{Word} \\
\hline
\forall\, w_1, w_2 : \mathit{Word} \bullet \\
\quad w_1 \bullet w_2 = (w_1 \ \underline{\mathit{pair}}\ w_2) \fatsemi (\_ \bullet \_) \ \wedge \\
\quad w_1 + w_2 = (w_1 \ \underline{\mathit{pair}}\ w_2) \fatsemi (\_ + \_) \ \wedge \\
\quad w_1 \oplus w_2 = (w_1 \ \underline{\mathit{pair}}\ w_2) \fatsemi (\_ \oplus \_)
\end{array}
$$

2.3 Shift functions

A word may be shifted left or right. In this case, the bottom (LSB) or top (MSB) bit of the word can attain a certain value, depending on the type of shift (e.g. arithmetic, logical or rotation).

$$
\begin{array}{|l}
\_ \ll \_ : (\mathit{Word} \times \mathit{Bit}) \rightarrow \mathit{Word} \\
\_ \gg \_ : (\mathit{Bit} \times \mathit{Word}) \rightarrow \mathit{Word} \\
\hline
\forall\, w : \mathit{Word};\ b : \mathit{Bit} \bullet \\
\quad w \ll b = (\{\#w\} \lhd \mathit{pred} \fatsemi w) \cup \{0 \mapsto b\} \ \wedge \\
\quad b \gg w = \{\#w - 1 \mapsto b\} \cup (\mathit{succ} \fatsemi w)
\end{array}
$$

2.4 Arithmetic functions

Microprocessors normally allow arithmetic operations. For example, a word may be incremented or decremented. The result wraps around if there is overflow or underflow in each case.

$$
\begin{array}{|l}
\mathit{inc},\ \mathit{dec} : \mathit{Word} \nrightarrow \mathit{Word} \\
\hline
\forall\, w : \mathit{Word} \bullet \\
\quad \mathit{inc}\, w = \mathit{wrd}\, \#w\, (\mathit{succ} \oplus \{\mathit{maxvel}\, w \mapsto 0\})(\mathit{val}\, w) \land \\
\quad \mathit{dec}\, w = \mathit{wrd}\, \#w\, (\{0 \mapsto \mathit{maxval}\, w\} \cup \mathit{pred})(\mathit{val}\, w)
\end{array}
$$

Incrementing and then decrementing a word (or vice versa) leaves it unchanged. Additionally one is the inverse of the other.

$$\vdash \mathit{inc} \fatsemi \mathit{dec} = \mathit{dec} \fatsemi \mathit{inc} = \mathit{id}[\mathit{Word}]$$

$$\vdash \mathit{dec} = \mathit{inc}^{-1}$$

This may be generalised for addition and subtraction by repeatedly incrementing or decrementing a word:

$$
\begin{array}{|l}
\_ + \_ , \\
\_ - \_ : (\mathit{Word} \times \mathbb{N}) \to \mathit{Word} \\
\hline
\forall\, w : \mathit{Word};\ i : \mathbb{N} \bullet \\
\quad w + i = \mathit{inc}^{i}\, w \land \\
\quad w - i = \mathit{dec}^{i}\, w
\end{array}
$$

Similarly, a second word, possibly of a different size, may be added to or subtracted from a word. The size of the resulting word is determined by the first word.

$$
\begin{array}{|l}
\_ + \_ , \\
\_ - \_ : (\mathit{Word} \times \mathit{Word}) \to \mathit{Word} \\
\hline
\forall\, w_1, w_2 : \mathit{Word} \bullet \\
\quad w_1 + w_2 = w_1 + (\mathit{val}\, w_2) \land \\
\quad w_1 - w_2 = w_1 - (\mathit{val}\, w_2)
\end{array}
$$

Some operations can return the 2's complement (negation) of a word:

$$\begin{array}{|l} - : \mathit{Word} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{Word} \\ \hline \forall w : \mathit{Word} \bullet \\ \quad -w = (w \ \underline{\mathit{set}}\ 0) - w \end{array}$$

Note that the 1's complement (bitwise logical NOT) and 2's complement (negation) of a word are related as follows:

$$\vdash \forall w : \mathit{Word} \bullet -w = \mathit{inc}(\sim w)$$

Sometimes it is useful to "sign-extend" a word into another (normally longer) word. This involves setting any extra bits to the value of the most significant bit (the "sign" bit) in the first word. The rest of the bits in the resulting word are set to the values of the equivalent bits in the first word.

$$\begin{array}{|l} \_ \mathit{signext} \_ : (\mathit{Word} \times \mathit{Word}) \rightarrow \mathit{Word} \\ \hline \forall w_1, w_2 : \mathit{Word} \bullet \\ \quad w_1 \ \underline{\mathit{signext}}\ w_2 = (w_2 \ \underline{\mathit{set}}\ (\mathit{MSB}\ w_1)) \oplus w_1 \end{array}$$

A word can be used as a signed relative offset. The value of the top bit determines the direction of the offset.

$$\begin{array}{|l} \_ \pm \_ : (\mathit{Word} \times \mathit{Word}) \rightarrow \mathit{Word} \\ \hline \forall w_1, w_2 : \mathit{Word} \bullet \\ \quad w_1 \pm w_2 = w_1 + (w_2 \ \underline{\mathit{signext}}\ w_1) \end{array}$$

This is particularly useful for branch instructions which usually allow a relative branch forwards and backwards.

2.5 Test conditions

Most microprocessors contain a status word which contains bits related to the results of previous operations. Different operations may affect different bits. Sometimes different operations affect the same bit in (possibly subtly) different ways.

Often we wish to test whether a word has a zero value, returning a '1' if it has and a '0' if not:

```
zero : Word → Bit
─────────────────────────
∀ w : Word •
  ran w = {0} ⇒ zero w = 1 ∧
  ran w ≠ {0} ⇒ zero w = 0
```

Conversely, we may wish to test whether a word contains all 1's, returning a '1' if it does and a '0' if not. This test can not usually be performed by microprocessors explicitly (unlike the test for zero above). However it can still be useful for the specification of other test conditions (see later).

```
ones : Word → Bit
─────────────────────────
∀ w : Word •
  ones w = zero(~w)
```

Testing for a negative value can be performed by most microprocessors, returning a '1' if it is negative and a '0' if not. Negative words have the top "sign" bit set. Hence this function can be performed by the previously defined MSB function.

2.6 Hexadecimal notation

Most microprocessor documentation uses hexadecimal values for op-codes, addresses and so forth, since this notation may easily be converted to the corresponding bit pattern. Each digit is the equivalent of four bits. Hexadecimal digits are drawn from the set of characters (CHAR) and consist of the decimal digits '0' to '9' and the letters 'A' to 'F'. Each of these hexadecimal digits uniquely maps to a numerical value:

$$[CHAR]$$

$$hex : CHAR \pfun \mathbb{N}$$

$$\begin{aligned} hex = \{&'0' \mapsto 0,\ '1' \mapsto 1,\ '2' \mapsto 2,\ '3' \mapsto 3,\\ &'4' \mapsto 4,\ '5' \mapsto 5,\ '6' \mapsto 6,\ '7' \mapsto 7,\\ &'8' \mapsto 8,\ '9' \mapsto 9,\ 'A' \mapsto 10,\ 'B' \mapsto 11,\\ &'C' \mapsto 12,\ 'D' \mapsto 13,\ 'E' \mapsto 14,\ 'F' \mapsto 15\} \end{aligned}$$

We can define a function to handle a sequence of hexadecimal digits (i.e. a hexadecimal number). We shall employ the widely used notation of prefixing 0x to the hexadecimal string.

$$0x : (seq\ CHAR) \pfun \mathbb{N}$$

$$\begin{aligned} &0x\langle\rangle = 0\\ &\forall\ s : seq_1\ CHAR \mid ran\ s \subseteq dom\ hex \bullet\\ &\quad 0x\ s = 16 * 0x(front\ s) + hex(last\ s) \end{aligned}$$

An alternative posibility would be to postfix the letter H (i.e. to define a similar postfix function, _H).

3. State

We shall consider the state of a 6800 based system in three parts, covering static conditions and then changes in state in each case:

1. Memory
2. Registers
3. System clock

We shall then combine these and consider changes in state of the entire system (as defined above) when an instruction is executed or an interrupt occurs. Finally, the state of the system when it powers up is detailed.

The 6800 operates on 8-bit bytes of data and 16-bit addresses:

$$\begin{aligned} Byte &\mathrel{\widehat{=}} \{ w : Word \mid \#w = 8 \} \\ Address &\mathrel{\widehat{=}} \{ w : Word \mid \#w = 16 \} \end{aligned}$$

The following functions convert values to data bytes and addresses respectively:

$$\begin{aligned} data &\mathrel{\widehat{=}} (wrd\ 8) \\ addr &\mathrel{\widehat{=}} (wrd\ 16) \end{aligned}$$

Some numerical values have known ranges. In particular, some numbers will fit into a *nibble* (4 bits), a data *byte* (8 bits) and a word address (16 bits). It is useful to define these ranges.

$$\begin{aligned} Value_4 &\mathrel{\widehat{=}} 0..2^4-1 \\ Value_8 &\mathrel{\widehat{=}} 0..2^8-1 \\ Value_{16} &\mathrel{\widehat{=}} 0..2^{16}-1 \end{aligned}$$

3.1 Memory

The address space of the 6800 (and many other microprocessors) may be considered as a total function from Addresses to Bytes. We shall assume that ROM (Read Only Memory) and RAM (Random Access Memory) make up the available real memory. These two areas do not overlap.

```
Memory
Mem : Address → Byte
ROM, RAM : ℙ Address
---
RAM ∩ ROM = ∅
```

The memory may be updated by operations such as instructions and interrupts. In this case, the ROM and RAM areas (i.e. their domains) do not change. The RAM contents may be partially updated by an instruction or interrupt. Areas outside valid ROM and RAM may vary unpredictably and are thus not defined by this specification. The values in ROM do not vary. Only values in RAM may be updated reliably. Additionally, some operations do not affect the RAM contents.

```
ΔMemory
Memory
Memory'
δMem : Address ⇸ Byte
---
ROM' = ROM
RAM' = RAM
ROM ◁ Mem' = ROM ◁ Mem
RAM ◁ Mem' = RAM ◁ (Mem ⊕ δMem)
```

$$\Xi Memory \mathrel{\widehat=} \Delta Memory \mid \delta Mem = \emptyset$$

Note that the assumptions above are not strictly true in all cases. For example, it is possible to have software switchable banked memory. However they hold for the majority of simple systems. In practice areas outside ROM and RAM may be used for memory mapped I/O. This is not covered here since it is very system dependent. It could be considered separately.

3.2 Registers

The 6800 has a number of registers:

$$
\begin{aligned}
Regs ::= \; & A \mid B \mid CCR \\
\mid \; & PC_H \mid PC_L \mid PC \\
\mid \; & SP_H \mid SP_L \mid SP \\
\mid \; & X_H \mid X_L \mid X
\end{aligned}
$$

Most of these are 8-bit registers:

$$Regs_8 \triangleq \{ A, B, CCR, PC_H, PC_L, SP_H, SP_L, X_H, X_L \}$$

Two of the 8-bit registers are general purpose accumulators:

$$Accumulator \triangleq \{ A, B \}$$

Some of the registers are normally used in pairs, so that they may be used to hold 16-bit memory addresses:

$$Regs_{16} \triangleq \{ PC, SP, X \}$$

The 8-bit registers always contain byte values and the 16-bit registers always contain address values. The low and high bytes of the PC, SP and X registers concatenate to form 16-bit registers. The top two bits of the CCR are unused and are always set to 1.

Registers

$$Reg : Regs \rightarrow Word$$

$$
\begin{aligned}
& Reg(\!| Regs_8 |\!) \subseteq Byte \\
& Reg(\!| Regs_{16} |\!) \subseteq Address \\
& Reg(PC) = Reg(PC_L) \frown Reg(PC_H) \\
& Reg(SP) = Reg(SP_L) \frown Reg(SP_H) \\
& Reg(X) = Reg(X_L) \frown Reg(X_H) \\
& Reg\, CCR\, (\!| 6..7 |\!) = \{1\}
\end{aligned}
$$

Any of the registers may be updated by an instruction (or interrupt). Every instruction consists of one or more bytes. (External interrupts have no bytes.) Normally the next instruction to be executed is the instruction following the current instruction. This may be overridden, for example by a branch instruction (see later). Individual bits in the Condition Code Register may be updated by the instruction depending on the result of the operation. However the top two bits of the CCR remain set to 1's even if the instruction attempts to overwrite them.

```
ΔRegisters
Registers
Registers'
NBytes : ℕ
Next   : Address
δReg   : Regs ⇸ Word
δCCR   : (0..7) ⇸ Bit
─────────────────────────────────────────
Next   = Reg(PC) + NBytes
Reg'   = Reg ⊕ {PC ↦ Next}
             ⊕ δReg
             ⊕ {CCR ↦ (Reg(CCR) ⊕ δCCR
                                ⊕ {6↦1, 7↦1})}
```

Sometimes an operation does not affect the 6800 registers (apart from the Program Counter which is automatically updated):

$$\Xi Registers \;\widehat{=}\; \Delta Registers \mid \delta Reg = \emptyset \wedge \delta CCR = \emptyset$$

Condition codes

The Condition Code Register holds various single bit codes at different bit positions. These are the carry, overflow, zero, negative, interrupt mask and half-carry bits:

```
C ≙ 0
V ≙ 1
Z ≙ 2
N ≙ 3
I ≙ 4
H ≙ 5
```

The contents of the individual condition code bits are often of interest. We make the following definitions for syntactic brevity:

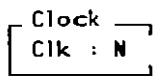
```
Cc ≙ (Reg CCR) C
Vc ≙ (Reg CCR) V
Zc ≙ (Reg CCR) Z
Nc ≙ (Reg CCR) N
Ic ≙ (Reg CCR) I
Hc ≙ (Reg CCR) H
```

Condition code bits often depend on the values of bits in results of operations. For the convenience of these specifications, we use the following short forms for $i \in 0..7$, $j \in 0..15$ and $x \in$ Accumulator:

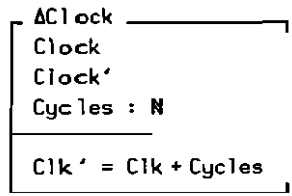
```
xi ≙ (Reg x) i
Mi ≙ (Mem M) i
Ri ≙ R i
Xj ≙ (Reg X) j
RRj ≙ RR j
```

3.3 System clock

The system contains a clock which controls the timing of the system. This consists of a sequence of pulses. This may be modelled as the number of clock pulses which have occurred since the system was powered-up:



When an instruction is executed or an interrupt occurs, it takes a certain number of clock cycles to execute:



3.4 M6800 system

The system state consists of memory, registers and a clock:

$$M6800 \triangleq Memory \wedge Registers \wedge Clock$$

There are various types of 6800 addressing modes. Additionally, the 6800 may respond to an external Interrupt or execute an Illegal instruction.

$$\begin{aligned} Modes ::= & \; Immediate \\ | & \; Direct \\ | & \; Indexed \\ | & \; Extended \\ | & \; Inherent \\ | & \; Relative \\ | & \; Interrupt \\ | & \; Illegal \end{aligned}$$

Each of these modes is detailed later.

When an instruction is executed, the op-code is read from the memory location indicated by the current value of the Program Counter. The instruction will have a particular addressing mode. The state of the system will change when the instruction has executed:

$$\begin{array}{|l} \Delta M6800 \\ \hline \Delta Memory \\ \Delta Registers \\ \Delta Clock \\ Op : Value_8 \\ Mode : Modes \\ \hline Op = val\ Mem(PC) \end{array}$$

Some operations do not affect the memory or registers (apart from the Program Counter which is automatically incremented depending on NBytes):

$$\Xi M6800 \triangleq \Delta M6800 \wedge \Xi Memory \wedge \Xi Registers$$

3.5 Power-up

The clock starts from zero for convenience in this model, when the system is initialised (i.e. powered up). It is assumed that the ROM already holds the program to be executed.

Interrupts are disabled and the Program Counter is loaded from the top two locations in memory. Note that hexadecimal numbers are used, rather than decimal, for memory addresses and op-code values since this is more normal (and convenient) in microprocessor documentation as discussed earlier.

$M6800_{INIT}$
M6800'

Clk' = 0
I_{cc}' = 1
Reg'(PC_H) = Mem'(addr 0xFFFE)
Reg'(PC_L) = Mem'(addr 0xFFFF)

4. Interrupts

When an interrupt occurs, or if the SWI or WAI instructions are executed (see later), all the 6800 registers are saved on the stack. program control is transferred to a new address specified by the contents of memory at a particular vector address. The interrupt mask bit is set in the Condition Code Register. This is defined by a framing schema (denoted by Φ) which may be used in the subsequent definitions of these cases:

```
ΦInterrupt
ΔM6800
Vector : Value_16
─────────────────────────────────────────
δMem = { Mem(Reg(SP)-6) ↦ Reg(CCR),
         Mem(Reg(SP)-5) ↦ Reg(B),
         Mem(Reg(SP)-4) ↦ Reg(A),
         Mem(Reg(SP)-3) ↦ Reg(X_H),
         Mem(Reg(SP)-2) ↦ Reg(X_L),
         Mem(Reg(SP)-1) ↦ hi(Next),
         Mem(Reg(SP))   ↦ lo(Next) }
δReg = { PC_H ↦ Mem(addr(Vector)),
         PC_L ↦ Mem(addr(Vector+1)),
         SP   ↦ Reg(SP)-7 }
δCCR = { I ↦ 1 }
```

There are three interrupts which may be activated externally to the 6800 microprocessor. An external interrupt is not an instruction read from memory so it may be considered to have a length of zero bytes. This will result in program control returning to the current instruction when au RTI instruction (see later) is subsequently executed at the end of the interrupt service routine, provided the stack is not corrupted.

It takes a number of clock cycles to service the interrupt and stack the registers. The exact number of cycles could not be found in the documentation used to formulate this specification [8,9], so it is not given here. It is likely to be of the order of the minimum number of cycles taken by the WAI instruction. If known, it could easily be inserted in the following schemas.

The hardware interrupt (IRQ) can only be activated if the interrupt mask bit in the CCR is clear:

```
IRQ
ΦInterrupt
---
I_CC = 0
Vector = 0xFFF8
NBytes = 0
```

The non-maskable interrupt (NMI) may be activated at any time:

```
NMI
ΦInterrupt
NMI? : Bit
---
NMI? = 0
Vector = 0xFFFC
NBytes = 0
```

When a reset occurs, the registers are not stacked and the memory is left unaffected, but the "reset" vector is used to restart the program in the same way as occurs at power-up:

```
Reset
ΔM6800
ΞMemory
---
ΘReg = { PC_H ↦ Mem(addr 0xFFFE),
         PC_L ↦ Mem(addr 0xFFFF) }
ΘCCR = { I ↦ 1 }
```

In conclusion, the system has three possible sources of external interrupt. Note that the 6800 interrupt vectors are all located at the top of memory. Hence it is normal for this area to be contained in ROM.

5. Instructions

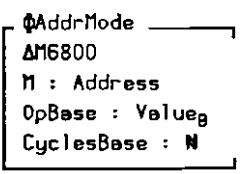
All microprocessors have a set of instructions which they can execute. These instructions can affect the registers and/or the memory using a variety of addressing modes, depending on the microprocessor involved.

5.1 Addressing modes

Many of the 6800 instructions use a selection of memory addressing modes. Each has a memory address (M) of an operand calculated in a manner depending on the addressing mode. The op-code for a given type of addressing mode is always and constant offset from the op-code for a particular *base* addressing mode. The 6800 Extended addressing mode may conveniently be selected for this base addressing mode. The corresponding op-code for a particular instruction will be known as the base op-code (OpBase). The value of OpBase is specified in subsequent schemas defining specific instructions.

The number of clock cycles which an instruction takes to execute also depends on the addressing mode. Again this is easily calculated from a base number of cycles for a particular addressing mode (CyclesBase). The number of execution cycles may be defined in terms of an offset from the base number of cycles in subsequent schemas.

The information above may be combined together in a framing schema for use when defining each of the addressing modes covered in the rest of the section:



We shall now define the major addressing modes of the 6800 as framing schemas for use by subsequent schemas describing individual 6800 instructions.

Immediate mode addressing gives the address of the byte immediately following the instruction op-code byte:

```
ΦImmediate
ΦAddrMode

Mode = Immediate
Op = OpBase - 0x30
M = Reg(PC)+1
NBytes = 3
CyclesBase = 2
```

Direct mode addresses are in the first 256 bytes of memory. The byte following the op-code specifies this address, the upper byte of the address being zero:

```
ΦDirect
ΦAddrMode

Mode = Direct
Op = OpBase - 0x20
M = Mem(Reg(PC)+1)^data(0)
NBytes = 2
CyclesBase = 3
```

Indexed mode address are calculated by adding the contents of the byte following the op-code (0-255) to the index register:

```
ΦIndexed
ΦAddrMode

Mode = Indexed
Op = OpBase - 0x10
M = Reg(X) + Mem(Reg(PC)+1)
NBytes = 2
CyclesBase = 5
```

Extended mode addresses are specified fully using the two bytes following the op-code, high byte first, low byte second:

```
┌ ΦExtended ──────────────────────────┐
│ ΦAddrMode                           │
├─────────                            │
│ Mode = Extended                     │
│ Op = OpBase                         │
│ M = Mem(Reg(PC)+2)⌢Mem(Reg(PC)+1)   │
│ NBytes = 3                          │
│ CyclesBase = 4                      │
└─────────────────────────────────────┘
```

Several or all of these addressing modes may be used by a specific instruction. Hence we shall combine them together into one schema.

$$\Phi Modes \;\widehat{=}\; \Phi Immediate \vee \Phi Direct \vee \Phi Indexed \vee \Phi Extended$$

There are two other addressing modes used by many instructions so these are also defined separately here.

Some instructions use inherent addressing. In this case there is no memory address to be calculated. The instruction consists of a single byte op-code.

```
┌ ΦInherent ──────┐
│ ΦAddrMode       │
├─────────         │
│ Mode = Inherent │
│ NBytes = 1      │
│ CyclesBase = 2  │
└─────────────────┘
```

Note that the memory address (M) is left undefined in the above framing schema since it will never be used in later specifications making use of this schema.

Some "branch" instructions use relative addressing to calculate a new value for the Program Counter if a branch occurs. The byte following the op-code is sign-extended and added to the address of the next instruction. Hence a branch instruction may transfer program control up to 127 bytes forwards or 128 bytes backwards relative to the start of the instruction following the branch instruction.

```
ΦRelative
ΦAddrMode

Mode = Reletive
Op = OpBase
M = Next ± Mem(Reg(PC)+1)
NBytes = 2
CyclesBase = 4
```

The complete instruction set of the 6800 is covered in subsequent sections consisting of families of related instructions as designated by Motorola [8].

5.2 Accumulator and Memory instructions

This family of instructions use one or both of the 8-bit accumulators and/or a byte in memory. These can be further sub-divided into different types of instruction, depending on the allowed addressing modes.

Inherent addressing

Some instructions use inherent addressing and operate on accumulator A or B only. The memory contents are unaffected. The instruction operation produces a byte result, R, which is used to update the accumulator.

```
ΦSingleAcc
ΦInherent
ΞMemory
x : Accumulator
R : Byte
---
Cycles = CyclesBase
δReg = { x ↦ R }
```

Accumulator addressing

Either of the accumulators may be pushed onto or popped off the stack. These operations take four cycles to execute.

```
ΦStackAcc
ΦInherent
x : Accumulator
OpBase : Value_8
---
(x = A ∧ Op = OpBase) ∨
(x = B ∧ Op = OpBase + 1)
Cycles = CyclesBase + 2
```

Single operand addressing

Some instructions have a single operand. They can update a memory byte by performing an operation on it, but only using a limited set of the available addressing modes:

```
ΦMemUpdate
ΦModes
Operand,
R : Byte
─────────────────
Mode ∈ { Indexed, Extended }
Cycles = CyclesBase + 2
Operand = Mem(M)
ΔMem = { M ↦ R }
ΔReg = ∅
```

These can also perform the same operation on one of the accumulators. These replace the op-codes which would have been used by the immediate and direct addressing modes not used because of the limited number of addressing modes above.

```
ΦAccUpdate
ΦSingleAcc
Operand : Byte
OpBase : Value_8
─────────────────
(x = A ∧ Op = OpBase - 0x30) ∨
(x = B ∧ Op = OpBase - 0x20)
Operand = Reg(x)
```

The last two schemas may be combined to produce a framing schema which describes single operand instructions with multiple addressing modes:

$$\Phi Single \;\widehat{=}\; \Phi MemUpdate \vee \Phi AccUpdate$$

Double operand addressing

Some instructions have two operands. One is in one of the two accumulators and the other is extracted from memory using a selection of addressing modes. The op-code base offsets are calculated from the op-code base of the instruction which uses accumulator A (OpBaseA). The value of OpBaseA is defined in subsequent schema definitions for specific instructions.

```
ΦAccumulator
ΦModes
x : Accumulator
OpBaseA : Value8
---
(x = A ∧ OpBase = OpBaseA) ∨
(x = B ∧ OpBase = OpBaseA + 0x40)
```

These double operand instructions leave memory unaffected and take the basic number of clock cycles to execute. The instruction operation produces a byte result (R).

```
ΦDouble
ΦAccumulator
ΞMemory
R : Byte
---
Cycles = CyclesBase
δReg = { x ↦ R }
```

Test instruction framing schema

Some instructions simply perform tests on a byte value, T. In this case, the memory and registers (apart from the CCR) are left unaffected (or effectively updated with existing contents). The top "sign" bit of the byte may be of particular interest.

ΦTest

ΦModes
x : Accumulator
T : Byte
T_7 : Bit

δMem $\in$ { $\emptyset$, { M $\mapsto$ Mem(M) } }
δReg $\in$ { $\emptyset$, { x $\mapsto$ Reg(x) } }
T_7 = MSB(T)

The accumulator and memory family of instructions can now be defined using the framing schemas above. All the instructions operate on 8-bit values in memory and the two accumulators.

Transfer instructions

Some instructions simply transfer bytes between registers and/or memory without modifying their contents. For example, an accumulator may be loaded from a memory byte. The Condition Code Register bits are updated appropriately.

```
LDA
ΦDouble

OpBaseA = 0xB6
R = Mem(M)
ΔCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ 0 }
```

Conversely, there is an instruction to store the contents of an accumulator into a byte in memory. This cannot use the immediate addressing mode. It takes an extra clock cycle to execute compared to most other similar instructions (e.g. LDA). The addressed memory byte is updated with the result and the CCR bits are set appropriately.

```
STA
ΦAccumulator

OpBaseA = 0xB7
Mode ≠ Immediate
Cycles = CyclesBase + 1
ΔMem = { M ↦ Reg(x) }
ΔReg = ∅
ΔCCR = { N ↦ MSB(Reg x),
         Z ↦ zero(Reg x),
         V ↦ 0 }
```

The accumulators may be transferred back and forth:

TAB
ΦSingleAcc

Op = 0x16
δReg = { A ↦ Reg(B) }
δCCR = { N ↦ B_7,
Z ↦ zero(B),
V ↦ 0 }

TBA
ΦSingleAcc

Op = 0x17
δReg = { B ↦ Reg(A) }
δCCR = { N ↦ A_7,
Z ↦ zero(A),
V ↦ 0 }

The accumulators may be pushed on to the stack. In this case, the condition codes are not affected.

PSH
ΦStackAcc

OpBase = 0x36
δMem = { Reg(SP) ↦ Reg(x) }
δReg = { SP ↦ Reg(SP)-1 }
δCCR = ∅

The accumulators may also be restored from the stack. Again, the CCR is unaffected. In this case the memory contents are also unaffected.

```
PULA
ΦStackAcc
ΞMemory

OpBase = 0x32
δReg = { x  ↦ Mem(Reg(SP)+1),
         SP ↦ Reg(SP)+1 }
δCCR = ∅
```

Logical instructions

Some instructions perform bitwise logical operations on the accumulators and memory bytes. For example, a byte operand may be 1's complemented:

```
COM
ΦSingle

OpBase = 0x73
R = ~Operand
δCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ 0,
         C ↦ 1 }
```

There is a bitwise logical AND instruction:

```
AND
ΦDouble
OpBaseA = 0xB4
R = Reg(x) • Mem(M)
δCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ 0 }
```

a bitwise logical inclusive OR instruction:

```
ORA
ΦDouble
OpBaseA = 0xBA
R = Reg(x) + Mem(M)
δCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ 0 }
```

and a bitwise logical exclusive OR instruction:

```
EOR
ΦDouble
OpBaseA = 0xB8
R = Reg(x) ⊕ Mem(M)
δCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ 0 }
```

Arithmetic instructions

Some instructions perform simple arithmetic operations on bytes.

An operand may be incremented. The overflow bit in the CCR is set if the original contents of the operand had the top bit clear and the rest of the operand bits were set to 1's.

```
INC
ΦSingle

OpBase = 0x7C
R = Operand + 1
$CCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ ~Operand(7)•ones(7⟨Operand) }
```

Conversely, an operand may be decremented. The overflow bit in the CCR is set if the original contents of the operand had the top bit set and the rest of the operand bits were zero.

```
DEC
ΦSingle

OpBase = 0x7A
R = Operand - 1
$CCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ Operand(7)•zero(7⟨Operand) }
```

There are three "add" instructions. They all update the half-carry bit in the CCR with the carry from bit 3. The overflow bit is set if there was a 2's complement overflow. The carry bit is set if there was a carry from the most significant bit of the result. The standard "add" instruction simply adds a byte from memory to an accumulator.

```
ADD
ϕDouble

OpBaseA = 0xBB
R = Reg(x) + Mem(M)
δCCR = { H ↦ x3•M3+M3•~R3+~R3•x3,
         N ↦ R7,
         Z ↦ zero(R),
         V ↦ x7•M7•~R7+~x7•~M7•R7,
         C ↦ x7•M7+M7•~R7+~R7•x7 }
```

Accumulator B can be added to accumulator A (but not vice versa):

```
ABA
ϕSingleAcc

Op = 0x1B
R = Reg(A) + Reg(B)
δReg = { A ↦ R }
δCCR = { H ↦ A3•B3+B3•~R3+~R3•A3,
         N ↦ R7,
         Z ↦ zero(R),
         V ↦ A7•B7•~R7+~A7•~B7•R7,
         C ↦ A7•B7+B7•~R7+~R7•A7 }
```

The current value of the carry bit in the CCR may be added to the result as well:

```
ADC
ϕDouble

OpBaseA = 0xB9
R = Reg(x) + Mem(M) + Ccc
δCCR = { H ↦ x3•M3+M3•~R3+~R3•x3,
         N ↦ R7,
         Z ↦ zero(R),
         V ↦ x7•M7•~R7+~x7•~M7•R7,
         C ↦ x7•M7+M7•~R7+~R7•x7 }
```

There is a "Decimal Adjust Accumulator" instruction for use when binary coded decimal (BCD) operands are involved. The adjustment to be added to the accumulator is calculated from the carry bit, upper half-byte value of the accumulator, half-carry bit and lower half-byte value of the accumulator as follows:

$$
\begin{array}{l}
daa : Bit \times Value_4 \times Bit \times Value_4 \pfun Value_8 \\
\hline
\forall\ i : Bit \times Value_4 \times Bit \times Value_4 \bullet \\
\quad i \in \{0\} \times (0x0..0x9) \times \{0\} \times (0x0..0x9) \Rightarrow daa\ i = 0x00 \\
\quad i \in \{0\} \times (0x0..0x8) \times \{0\} \times (0xA..0xF) \Rightarrow daa\ i = 0x06 \\
\quad i \in \{0\} \times (0x0..0x9) \times \{1\} \times (0x0..0x3) \Rightarrow daa\ i = 0x06 \\
\quad i \in \{0\} \times (0xA..0xF) \times \{0\} \times (0x0..0x9) \Rightarrow daa\ i = 0x60 \\
\quad i \in \{0\} \times (0x9..0xF) \times \{0\} \times (0xA..0xF) \Rightarrow daa\ i = 0x66 \\
\quad i \in \{0\} \times (0xA..0xF) \times \{1\} \times (0x0..0x3) \Rightarrow daa\ i = 0x66 \\
\quad i \in \{0\} \times (0x0..0x2) \times \{0\} \times (0x0..0x9) \Rightarrow daa\ i = 0x60 \\
\quad i \in \{0\} \times (0x0..0x2) \times \{0\} \times (0xA..0xF) \Rightarrow daa\ i = 0x66 \\
\quad i \in \{0\} \times (0x0..0x3) \times \{1\} \times (0x0..0x3) \Rightarrow daa\ i = 0x66
\end{array}
$$

Entries not included in the table are undefined. The overflow bit in the CCR is always undefined after this instruction has been executed. It is intended that this instruction should be used immediately after an "add" instruction.

DAA

$$
\begin{array}{l}
\Phi SingleAcc \\
Adjustment : Byte \\
Undefined : Bit \\
\hline
Op = 0x1B \\
Adjustment = data\ daa(C_{cc}, val\ hi(Reg\ A), \\
\qquad\qquad\qquad\qquad H_{cc}, val\ lo(Reg\ A)) \\
R = A + Adjustment \\
\delta Reg = \{\, A \mapsto R \,\} \\
\delta CCR = \{\, N \mapsto R_7, \\
\qquad\quad Z \mapsto zero(R), \\
\qquad\quad V \mapsto Undefined, \\
\qquad\quad C \mapsto \sim zero(hi\ Adjustment) \,\}
\end{array}
$$

There are matching "subtract" instructions for each "add" instruction. Note however that the half-carry bit in the CCR is left unaffected by these instructions.

```
SUB
ΦDouble

OpBaseA = 0xB0
R = Reg(x) - Mem(M)
ΔCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ x7•~M7•~R7+~x7•M7•R7,
         C ↦ ~x7•M7+M7•R7+R7•~x7 }
```

```
SBA
ΦSingleAcc

Op = 0x10
R = Reg(A) - Reg(B)
ΔReg = { A ↦ R }
ΔCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ A7•~B7•~R7+~A7•B7•R7,
         C ↦ ~A7•B7+B7•R7+R7•~A7 }
```

```
SBC
ΦDouble

OpBaseA = 0xB2
R = Reg(x) - Mem(M) - Ccc
ΔCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ x7•~M7•~R7+~x7•M7•R7,
         C ↦ ~x7•M7+M7•R7+R7•~x7 }
```

An operand may be negated (2's complemented). The overflow bit in the CCR is set if the result has the top bit set and the rest of the result bits are zero. The carry bit is set to the opposite of the zero bit.

```
NEG
ΦSingle

OpBase = 0x70
R = -Operand
$CCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ R7•zero(7⊲R),
         C ↦ ~zero(R) }
```

A memory byte or an accumulator may be cleared to all 0's.

```
CLR
ΦSingle

OpBase = 0x7F
R = data(0)
$CCR = { N ↦ 0,
         Z ↦ 1,
         V ↦ 0,
         C ↦ 0 }
```

Note that there is no equivalent instruction to set a byte to all 1's.

Shift instructions

Some 6800 instructions shift bytes by one bit position left or right. Note that the overflow bit in the CCR is always set as the XOR of the resulting negative and carry CCR bits for all 6800 shift instructions.

There are shift instructions which rotate a byte left or right by one bit through the carry bit in the CCR:

```
ROL
ΦSingle

OpBase = 0x79
R = Operand ≪ C_cc
δCCR = { N ↦ R_7},
         Z ↦ zero(R),
         V ↦ N_cc' ⊕ C_cc',
         C ↦ Operand(7) }
```

```
ROR
ΦSingle

OpBase = 0x76
R = C_cc ≫ Operand
δCCR = { N ↦ R_7,
         Z ↦ zero(R),
         V ↦ N_cc' ⊕ C_cc',
         C ↦ Operand(0) }
```

There are arithmetic shift instructions which shift a byte left or right by one bit. These are equivalent to multiplying and dividing a signed byte value by 2.

```
ASL
ΦSingle

OpBase = 0x78
R = Operend ≪ 0
δCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ Ncc' ⊕ Ccc',
         C ↦ Operand(7) }
```

```
ASR
ΦSingle

OpBase = 0x77
R = Operand(7) ≫ Operand
δCCR = { N ↦ R7,
         Z ↦ zero(R),
         V ↦ Ncc' ⊕ Ccc',
         C ↦ Operand(0) }
```

There is a logical shift right instruction, filling the result with a zero in its top bit:

```
LSR
ΦSingle

OpBase = 0x74
R = 0 ≫ Operand
δCCR = { N ↦ 0,
         Z ↦ zero(R),
         V ↦ Ncc' ⊕ Ccc',
         C ↦ Operand(0) }
```

Note that there is no matching LSL (logical shift left) instruction since this is equivalent to an ASL instruction (see above).

Test instructions

Some instructions only affect the condition codes by performing tests on byte values.

There is a bitwise logical AND test instruction which simply sets the condition code bits as if an AND instruction had been performed, but does not update the result:

```
BIT
ΦDouble
ΦTest
---
OpBaseA = 0xB5
T = Reg(x) • Mem(M)
SCCR = { N ↦ T7,
         Z ↦ zero(T),
         V ↦ 0 }
```

A byte operand may be tested. The condition codes are set as if zero had been subtracted from the operand.

```
TST
ΦSingle
ΦTest
---
OpBase = 0x7D
T = Operand - 0
SCCR = { N ↦ T7,
         Z ↦ zero(T),
         V ↦ 0,
         C ↦ 0 }
```

There is a "compare" instruction which simply sets the condition code bits as if a SUB instruction had been performed, but does not update the result:

```
CMP
ΦDouble
ΦTest
---
OpBaseA = 0xB5
T = Reg(x) - Mem(M)
ΔCCR = { N ↦ T7,
         Z ↦ zero(T),
         V ↦ x7•~M7•~T7+~x7•M7•T7,
         C ↦ ~x7•M7+M7•T7+T7•~x7 }
```

The two accumulators may be compared in a similar way without changing the contents of either:

```
CBA
ΦSingleAcc
ΦTest
---
Op = 0x11
T = Reg(A) - Reg(B)
ΔCCR = { N ↦ T7,
         Z ↦ zero(T),
         V ↦ A7•~B7•~T7+~A7•B7•T7,
         C ↦ ~A7•B7+B7•T7+T7•~A7 }
```

Instruction types

The 6800 includes the following transfer/logical/arithmetic/shift/test type accumulator and memory instructions:

```
DoubleOp    ≙  LDA V STA V
               AND V ORA V EOR V
               ADD V ADC V SUB V SBC V
               BIT V CMP

SingleOp    ≙  COM V
               DEC V INC V NEG V CLR V
               ROL V ROR V ASL V ASR V LSR V
               TST

InherentOp  ≙  TAB V TBA V
               ABA V DAA V SBA V
               CBA

StackOp     ≙  PSH V PUL
```

We can combine all these sub-types of instruction together:

```
AccMemOp  ≙  DoubleOp V SingleOp V InherantOp V StackOp
```

5.3 Index Register and Stack instructions

These instructions manipulate the 16-bit index register and stack pointer. Some have several addressing modes. These can be further sub-divided into "load" and "store" type operations, each of which produces a 16-bit result, RR. Load operations do not affect memory:

```
ΦXLoad
ΦModes
ΞMemory
RR : Address
─────────────
Cycles = CyclesBase + 1
```

Store operations cannot be used in immediate mode:

```
ΦXStore
ΦModes
RR : Address
─────────────
Mode ≠ Immediate
Cycles = CyclesBase + 2
```

Some of the instructions use inherent addressing. None of these affect the memory contents.

```
ΦXInherent
ΦInherent
ΞMemory
─────────────
Op = OpBase
Cycles = CyclesBase + 2
```

This family of instructions can now be defined using the framing schemas above.

The index register and stack pointer can be loaded from memory:

```
LDX
ΦXLoad
-------------------------------
OpBase = 0xCE
RR = Reg(X)
ΔReg = { X_H ↦ Mem(M),
         X_L ↦ Mem(M+1) }
ΔCCR = { N ↦ RR_15,
         Z ↦ zero(RR),
         V ↦ 0 }
```

```
LDS
ΦXLoad
-------------------------------
OpBase = 0x8E
RR = Reg(SP)
ΔReg = { SP_H ↦ Mem(M),
         SP_L ↦ Mem(M+1) }
ΔCCR = { N ↦ RR_15,
         Z ↦ zero(RR),
         V ↦ 0 }
```

and stored into memory:

```
STX
ΦXStore
-------------------------------
OpBase = 0xCF
RR = Reg(X)
ΔReg = ∅
ΔCCR = { N ↦ RR_15,
         Z ↦ zero(RR),
         V ↦ 0 }
ΔMem = { M ↦ Reg(X_H), M+1 ↦ Reg(X_L) }
```

```
STS
ϕXStore

OpBase = 0x8F
RR = Reg(SP)
δReg = ∅
δCCR = { N ↦ RR15,
         Z ↦ zero(RR),
         V ↦ 0 }
δMem = { M ↦ Reg(SPH), M+1 ↦ Reg(SPL) }
```

They can also be transferred back and forth:

```
TXS
ϕXInherent

Op = 0x35
δReg = { SP ↦ Reg(X)-1 }
δCCR = ∅
```

```
TSX
ϕXInherent

Op = 0x30
δReg = { X ↦ Reg(SP)+1 }
δCCR = ∅
```

Note that the SP is loaded with one less than the contents of the index register and the index register is loaded with one more than the SP in each case. This is for programming convenience so that the index register can be pointed to the first entry on the stack, not the next empty entry.

The index register and stack pointer can both be incremented and decremented. In the case of the index register, the zero flag bit in the CCR is set appropriately. In the case of the stack pointer, the CCR is not affected.

INX

```
ϕXInherent

Op = 0x08
δReg = { X ↦ Reg(X)+1 }
δCCR = { Z ↦ zero(Reg(X)-1)}
```

INS

```
ϕXInherent

Op = 0x31
δReg = { SP ↦ Reg(SP)+1 }
δCCR = ∅
```

DEX

```
ϕXInherent

Op = 0x09
δReg = { X ↦ Reg(X)-1 }
δCCR = { Z ↦ zero(Reg(X)-1)}
```

DES

```
ϕXInherent

Op = 0x34
δReg = { SP ↦ Reg(SP)-1 }
δCCR = ∅
```

The index register can be compared with memory:

CPX

ΦXLoad

OpBase = 0x8C
RR = Reg(X) - Mem(M+1)⌢Mem(M)
δReg = ∅
δCCR = { N ↦ RR_{15},
Z ↦ zero(RR),
V ↦ X_{15}•~M_7•~RR_{15}+~X_{15}•M_7•RR_{15} }

The 6800 includes the following instructions involving the index register and/or stack pointer:

IndexOp ≙ LDX ∨ LDS ∨ STX ∨ STS ∨ TXS ∨ TSX ∨
INX ∨ INS ∨ DEX ∨ DES ∨ CPX

5.4 Branch and Jump instructions

All "branch" instructions use the relative addressing mode. They leave the memory unchanged and take four cycles to execute. The CCR is not affected. If a branch condition occurs, then the PC is updated with the relative offset. Otherwise the program procedes to the next instruction as normal.

```
┌ ΦBranch ──────────────────────┐
│ ΦRelative                     │
│ ΞMemory                       │
│ Cond : Bit                    │
├───────────────────────────────┤
│ Cycles = CyclesBase           │
│ ΔCCR = ∅                      │
│ Cond = 1 ⇒ ΔReg = { PC ↦ M }  │
│ Cond = 0 ⇒ ΔReg = ∅           │
└───────────────────────────────┘
```

The 6800 has the following branch instructions:

$$
\begin{aligned}
&\mathrm{BRA} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x20 \wedge Cond = 1\\
&\mathrm{BCC} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x24 \wedge Cond = \sim C_{cc}\\
&\mathrm{BCS} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x25 \wedge Cond = C_{cc}\\
&\mathrm{BEQ} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x27 \wedge Cond = Z_{cc}\\
&\mathrm{BGE} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x2C \wedge Cond = \sim(N_{cc} \oplus V_{cc})\\
&\mathrm{BGT} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x2E \wedge Cond = \sim(Z_{cc} + (N_{cc} \oplus V_{cc}))\\
&\mathrm{BHI} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x22 \wedge Cond = \sim(C_{cc} + Z_{cc})\\
&\mathrm{BLE} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x2F \wedge Cond = Z_{cc} + (N_{cc} \oplus V_{cc})\\
&\mathrm{BLS} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x23 \wedge Cond = C_{cc} + Z_{cc}\\
&\mathrm{BLT} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x2D \wedge Cond = N_{cc} \oplus V_{cc}\\
&\mathrm{BMI} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x2B \wedge Cond = N_{cc}\\
&\mathrm{BNE} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x26 \wedge Cond = \sim Z_{cc}\\
&\mathrm{BVC} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x28 \wedge Cond = \sim V_{cc}\\
&\mathrm{BVS} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x29 \wedge Cond = V_{cc}\\
&\mathrm{BPL} \mathrel{\widehat{=}} \Phi\mathrm{Branch} \mid Op = 0x2A \wedge Cond = \sim N_{cc}
\end{aligned}
$$

There is also a "Branch to Subroutine" instruction, which saves the return address on the stack and calculates a new value for the PC:

```
BSR
ΦRelative
─────────
Op = 0x8D
Cycles = 8
ΔMem = { Mem(Reg(SP)-1) ↦ hi(Next),
         Mem(Reg(SP))   ↦ lo(Next) }
ΔReg = { PC ↦ M,
         SP ↦ Reg(SP) - 2 }
ΔCCR = Ø
```

There is a "Jump" instruction. Indexed and extended addressing modes may be used. The memory and CCR contents are unaffected.

```
JMP
ΦModes
ΞMemory
─────────
Mode ∈ { Indexed, Extended }
OpBase = 0x7E
Cycles = CyclesBase-1
ΔReg = { PC ↦ M }
ΔCCR = Ø
```

There is a "Jump to Subroutine" instruction, similar to the JMP instruction, which saves the return address on the stack. The number of cycles taken to execute this instruction does not obey the normal rules which apply to all other instructions with multiple addressing modes.

JSR

```
ΦModes

Mode ∈ { Indexed, Extended }
OpBase = 0xBD
Mode = Indexed ⇒ Cycles = 8
Mode = Extended ⇒ Cycles = 9
ΔMem = { Mem(Reg(SP)-1) ↦ hi(Next),
         Mem(Reg(SP))   ↦ lo(Next) }
ΔReg = { PC ↦ M }
         SP ↦ Reg(SP)-2 })
ΔCCR = ∅
```

The 6800 includes the following branch and jump instructions:

```
BranchOp ≙ BRA ∨ BCC ∨ BCS ∨ BEQ ∨ BGE ∨
           BGT ∨ BHI ∨ BLE ∨ BLS ∨ BLT ∨
           BMI ∨ BNE ∨ BVC ∨ BVS ∨ BPL ∨
           JMP ∨ JSR
```

5.5 Condition Code Register instructions

This set of instructions use inherent addressing and do not affect the memory contents. Most of the instructions update CCR flag bits, but not the rest of the registers.

```
┌ ΦCCR ──────────────┐
│ ΦInherent          │
│ ΞMemory            │
├────────────────────┤
│ Cycles = CyclesBase│
│ ΔReg = ∅           │
└────────────────────┘
```

The following instructions may be performed to clear and set individual Condition Code Register bits:

$$
\begin{aligned}
CLC &\mathrel{\widehat{=}} \Phi CCR \mid Op = 0x0C \wedge \Delta CCR = \{C \mapsto 0\} \\
CLI &\mathrel{\widehat{=}} \Phi CCR \mid Op = 0x0E \wedge \Delta CCR = \{I \mapsto 0\} \\
CLV &\mathrel{\widehat{=}} \Phi CCR \mid Op = 0x0A \wedge \Delta CCR = \{V \mapsto 0\} \\
SEC &\mathrel{\widehat{=}} \Phi CCR \mid Op = 0x0D \wedge \Delta CCR = \{C \mapsto 1\} \\
SEI &\mathrel{\widehat{=}} \Phi CCR \mid Op = 0x0F \wedge \Delta CCR = \{I \mapsto 1\} \\
SEV &\mathrel{\widehat{=}} \Phi CCR \mid Op = 0x0B \wedge \Delta CCR = \{V \mapsto 1\}
\end{aligned}
$$

The setable bits of the CCR may be loaded from accumulator A:

$$TAP \mathrel{\widehat{=}} \Phi CCR \mid Op = 0x06 \wedge \Delta CCR = Reg(A)$$

Conversely, accumulator A may be loaded with the contents of the CCR:

```
┌ TPA ──────────────────┐
│ ΦSingleAcc            │
├───────────────────────┤
│ Op = 0x07             │
│ ΔReg = { A ↦ Reg(CCR) }│
│ ΔCCR = ∅              │
└───────────────────────┘
```

These operations may be collected together as a family of instructions:

$$CCROp \mathrel{\widehat{=}} CLC \vee CLI \vee CLV \vee SEC \vee SEI \vee SEV \vee TAP \vee TPA$$

5.6 Miscellaneous instructions

There is a "No Operation" instruction which does nothing but pass program control to the next instruction:

```
NOP
ΦInherent
ΞM6800
---
Op = 0x01
Cycles = CyclesBase
```

There is a "Return from Subroutine" instruction. The PC is restored from the stack. The memory contents and the CCR are left unaffected.

```
RTS
ΦInherent
ΞMemory
---
Op = 0x39
Cycles = 5
δReg = { PC_H ↦ Mem(Reg(SP)+1),
         PC_L ↦ Mem(Reg(SP)+2),
         SP   ↦ Reg(SP)+2 }
δCCR = ∅
```

There is a "Software Interrupt" instruction. This simulates an interrupt using its own vector.

```
SWI
ΦInherent
ΦInterrupt
---
Op = 0x3F
Cycles = 12
Vector = 0xFFFA
```

There is a "Wait for Interrupt" instruction. This stacks the registers and then waits for an IRQ (if the interrupt mask bit in the CCR is not set) or an NMI interrupt to occur, or for the system to be reset. Unless an external interrupt is received, the program will be suspended forever.

```
WAI
ΦInherent
ΦInterrupt
---
Op = 0x3E
Cycles ≥ 9
Vector ∈ {0xFFF8, 0xFFFC, 0xFFFE}
I_cc = 0 ⇒ Vector ≠ 0xFFF8
```

There is a "Return from Interrupt" instruction. The registers are all restored from the stack. The memory contents are left unaffected. The CCR is loaded from a memory byte on the stack but the individual bits are not subsequently affected by the instruction.

```
RTI
ΦInherent
ΞMemory
---
Op = 0x3B
Cycles = 10
ΔReg = { CCR  ↦ Mem(Reg(SP)+1),
         B    ↦ Mem(Reg(SP)+2),
         A    ↦ Mem(Reg(SP)+3),
         X_H  ↦ Mem(Reg(SP)+4),
         X_L  ↦ Mem(Reg(SP)+5),
         PC_H ↦ Mem(Reg(SP)+6),
         PC_L ↦ Mem(Reg(SP)+7),
         SP   ↦ Reg(SP)+7 }
ΔCCR = Ø
```

The 6800 includes the following miscellaneous instructions:

```
MiscOp  ≙  NOP ∨ RTS ∨ SWI ∨ WAI ∨ RTI
```

6. Overall operation

Op-codes which have not so far been specified are considered illegal. The state of the system after the execution of such an op-code is undefined.

$$IllegalOp \triangleq \Delta M6800 \mid Mode = Illegal$$

This specification could be tightened if more were known about an illegal instruction. For example, at present this specification allows the contents of the registers and RAM to be entirely changed after an illegal instruction. If more information were available, predicates could be added to this schema.

The following groups of legal instructions discussed in previous sections may be executed by the 6800. We project the (change of) state of the 6800 since we are not interested in any of the temporary components defined in each of the individual instruction schemas for the convenience of the specification.

$$LegalOp \triangleq (AccMemOp \vee IndexOp \vee BranchOp \vee CCROp \vee MiscOp) \upharpoonright \Delta M6800$$

The system has three possible sources of external interrupt:

$$ExtInterrupt \triangleq (IRQ \vee NMI \vee Reset) \mid Mode = Interrupt$$

The priority of external interrupts has not been defined above (i.e. if two interrupts occur simultaneously either could be serviced first) since the documentation used [8,9] did not make any ordering clear. Such details could easily be included in the formal definition of the 6800 by including the status of the external interrupts as part of the state.

Each operation execution of the 6800 consists of the execution of an instruction (legal or otherwise) or an external interrupt:

$$Instruction \triangleq IllegalOp \oplus LegalOp$$

$$Exec \triangleq Instruction \vee ExtInterrupt$$

When the 6800 is started, a sequence of such operations is executed depending on the contents of memory and (non-deterministically in this specification) on the occurence of external interrupts.

Given the specification of each of the instructions, it is possible to consider sequences of instructions and prove (in the absence of any external interrupts) properties of such sequences. For example, often a decrement instruction is followed by a conditional branch instruction at the end of a loop. We could prove the following properties of such a construct:

$$\mathsf{DEXBNE} \triangleq \mathsf{DEX} \upharpoonright \Delta\mathsf{M6800} \ ; \ \mathsf{BNE} \upharpoonright \Delta\mathsf{M6800}$$

$$\mathsf{DEXBNE} \vdash \mathsf{Reg(X)} \neq 1 \Rightarrow \mathsf{Reg'(PC)} = (\mathsf{Reg(PC)}+3) \pm \mathsf{Mem}(\mathsf{Reg(PC)}+2)$$
$$\mathsf{DEXBNE} \vdash \mathsf{Reg(X)} = 1 \Rightarrow \mathsf{Reg'(PC)} = \mathsf{Reg(PC)}+3$$

7. Conclusion

The instruction set of the Motorola 6800 microprocessor has been formally specified. Enough experience has been gained so that more complicated and modern microprocessors such as the 68000 family could be specified in a similar manner. However such processors would require a larger document and more work in order to cover them fully.

The specification of the instructions have been factored out using framing schemas to reduce the overall length of the specification given here. If Z where to be used to present an instruction set in the form of a manual, then it is anticipated that each instruction would be allocated at least a page with an expanded schema allowing easy reference for the instruction on that page alone. A possible example layout is shown in Appendix A.

Z has proved an excellent tool for specifying a microprocessor instruction set. The length of the specification is very favourable with the more informal methods currently used for instruction set documentation in industry and elsewhere. Not only that, but we also gain a means of formally reasoning about the properties of the instruction set. This could prove to be invaluable, especially at the design stage. In the future, computer-based tools should be available to check consistency and give assistance with proofs. It is to be hoped that manufacturers will adopt such methods in due course.

8. Acknowledgements

Thank you to the developers of the Z specification language at the PRG and the inventors of the 6800 microprocessor at Motorola. Carroll Morgan, Tim Gleeson and Brian Monahan at the PRG provided helpful comments on early drafts. Ruaridh Macdonald at RSRE, Malvern and Stephen Murrell at the University of Miami also gave some useful suggestions. Steve Heath of Motorola, UK and Rajit Chandra of Intel, California commented on the paper from a manufacturer's point of view. Roger Gimson, Karen Paliwoda, Stig Topp-Jorgensen and Bernard Sufrin at the PRG kindly checked later drafts.

9. References

1. Hunt, W. A. "FM8501: A Verified Microprocessor", Technical Report 47. *Institute for Computing Science, The University of Texas at Austin*, (1986).

2. Sufrin, B. A. (Editor) "Z Handbook", Draft 1.1, *Programming Research Group, Oxford University*, (1986).

3. Spivey, J. M. "Understanding Z: A Specification Language and its Formal Semantics", DPhil Thesis, *Programming Research Group, Oxford University*, (1986).

4. Spivey, J. M. "The Z Library - A Reference Manual", *Programming Research Group, Oxford University*, (1986).

5. Woodcock, J. "Structuring Specifications - Notes on the Schema Notation", *Programming Research Group, Oxford University*, (1986).

6. King, S., Sørensen, I., Woodcock, J. "Z: Concrete and Abstract Syntaxes", Version 1.0, *Programming Research Group, Oxford University*, (1987).

7. Hayes, I. J. (Editor) "Specification Case Studies", *Prentice-Hall International Series in Computer Science*, (1987).

8. "M6800 Microprocessor Programming Manual", *Motorola Semiconductor Products Inc.*, (1975).

9. "M6800 Microprocessor Instruction Set Summary", *Motorola Microcomputer Applications Engineering*.

Appendix A

Example manual pages

An example layout for two instructions in a 6800 microprocessor instruction set manual are given overleaf. It is suggested that each instruction should be given a page like this in such a manual to allow quick reference for a particular instruction without the necessity for cross reference, once the framework of the specification has been assimilated by the reader.

Branch if Greater Than zero **BGT**

Operation

```
BGT
ΔM6800
Cond : Bit

Op     = 0x2E
Mode   = Relative
NBytes = 2
Cycles = 4
Cond   = Z_cc + (N_cc ⊕ V_cc)
Cond   = 0 ⇒ δReg = ∅
Cond   = 1 ⇒ δReg = {PC ↦ Next±Mem(Reg(PC)+1)}
δCCR   = ∅
δMem   = ∅
```

Description

Causes a branch if Z is set or one of N and V (but not both) is set.

If the BGT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. accumulator A or B contents) was greater than the two's complement number represented by the subtrahend (i.e. memory contents).

Only the PC is affected. If a branch occurs, then the PC is updated with the relative offset, otherwise the program procedes to the next instruction as normal.

Jump to Subroutine **JSR**

Operation

```
JSR
ΔM6800

(Op     = 0xAD
 Mode   = Indexed
 NBytes = 2
 Cycles = 8
 δReg = { PC ↦ Reg(X)+Mem(Reg(PC)+1),
          SP ↦ Reg(SP)-2 })
∨
(Op     = 0xBD
 Mode   = Extended
 NBytes = 3
 Cycles = 9
 δReg = { PC_H ↦ Mem(Reg(PC)+1),
          PC_L ↦ Mem(Reg(PC)+2),
          SP   ↦ Reg(SP)-2 })

δCCR = ∅
δMem = { Mem(Reg(SP)-1) ↦ hi(Next),
         Mem(Reg(SP))   ↦ lo(Next) }
```

Description

The program counter is incremented by 2 or by 3, depending on the addressing mode, and is then pushed onto the stack, eight bits at a time. The stack pointer points to the next empty location on the stack. A jump occurs to the instruction stored at the numerical address, obtained according to the addressing mode.

Appendix B

Mathematical and Schema notation

A glossary of the Z mathematical and schema notation used in this monograph is included here for easy reference. Readers should note that the definitive concrete and abstract syntax for Z is available elsewhere [5].

Z Reference Glossary

Mathematical Notation

1. Definitions and declarations.

Let x, x_i be identifiers and let T, T_i sets.

$[T_1, T_2]$ Introduction of generic sets.

$LHS \triangleq RHS$ Definition of LHS as syntactically equivalent to RHS.

$T ::= x_1 \mid x_2 \mid \ldots \mid x_n$
Data type definition.

$x : T$ Declaration of x as type T.

$x_1 : T_1; x_2 : T_2; \ldots ; x_n : T_n$
List of declarations.

$x_1, x_2, \ldots, x_n : T$
$\triangleq x_1 : T; x_2 : T; \ldots ; x_n : T.$

2. Logic.

Let P, Q be predicates and D declarations.

$\neg P$ Negation: "not P".

$P \wedge Q$ Conjunction: "P and Q".

$P \vee Q$ Disjunction: "P or Q":
$\triangleq \neg(\neg P \wedge \neg Q)$.

$P \Rightarrow Q$ Implication: "P implies Q" or "if P then Q": $\triangleq \neg P \vee Q$.

$P \Leftrightarrow Q$ Equivalence: "P is logically equivalent to Q":
$\triangleq (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

$true$ Logical constant.

$false$ $\triangleq \neg true$

$\forall x : T \bullet P$ Universal quantification: "for all x of type T, P holds".

$\exists x : T \bullet P$ Existential quantification: "there exists an x of type T such that P".

$\exists_1 x : T \bullet P_x$ Unique existence: "there exists a unique x of type T such that P".
$\triangleq (\exists x : T \bullet P_x \wedge \neg(\exists y : T \mid y \neq x \bullet P_y))$.

$\forall x_1 : T_1; x_2 : T_2; \ldots ; x_n : T_n \bullet P$
"For all x_1 of type T_1, x_2 of type T_2, . . . , and x_n of type T_n, P holds."

$\exists x_1 : T_1; x_2 : T_2; \ldots ; x_n : T_n \bullet P$
Similar to $\forall$.

$\exists_1 x_1 : T_1; x_2 : T_2; \ldots ; x_n : T_n \bullet P$
Similar to $\forall$.

$\forall D \mid P \bullet Q \triangleq (\forall D \bullet P \Rightarrow Q)$.

$\exists D \mid P \bullet Q \triangleq (\exists D \bullet P \wedge Q)$.

$D \vdash P$ Theorem: $\triangleq \vdash \forall D \bullet P$.

3. Sets.

Let S, T and X be sets; t, t_k terms; P a predicate and D declarations.

$t_1 = t_2$ Equality between terms.

$t_1 \neq t_2$ Inequality: $\triangleq \neg(t_1 = t_2)$.

$t \in S$ Set membership: "t is an element of S".

$t \notin S$ Non-membership: $\triangleq \neg(t \in S)$.

$\emptyset$ Empty set: $\triangleq \{ x : X \mid false \}$.

$S \subseteq T$ Set inclusion:
$\triangleq (\forall x : S \bullet x \in T)$.

$S \subset T$ Strict set inclusion:
$\triangleq S \subseteq T \wedge S \neq T$.

$\{ t_1, t_2, \ldots, t_n \}$ The set containing $t_1, t_2, \ldots$ and t_n.

$\{ x : T \mid P \}$
The set containing exactly those x of type T for which P holds.

$(t_1, t_2, \ldots, t_n)$ Ordered n-tuple of $t_1, t_2, \ldots$ and t_n.

$T_1 \times T_2 \times \ldots \times T_n$ Cartesian product: the set of all n-tuples such that the kth component is of type T_k.

$\{ x_1 : T_1; x_2 : T_2; \ldots ; x_n : T_n \mid P \}$
The set of n-tuples $(x_1, x_2, \ldots, x_n)$ with each x_k of type T_k such that P holds.

$\{ D \mid P \bullet t \}$
The set of t's such that given the declarations D, P holds.

$\{ D \bullet t \} \triangleq \{ D \mid true \bullet t \}$.

$\mathbb{P} S$
Powerset: the set of all subsets of S.

$\mathbb{P}_1 S$
Non-empty powerset: $\mathbb{P}_1 S \triangleq \mathbb{P} S \setminus \{\emptyset\}$.

$\mathbb{F} S$
Set of finite subsets of S: $\triangleq \{T: \mathbb{P} S \mid T \text{ is finite}\}$.

$\mathbb{F}_1 S$
Non-empty finite set: $\mathbb{F}_1 S \triangleq \mathbb{F} S \setminus \{\emptyset\}$.

$S \cap T$
Set intersection: given $S, T: \mathbb{P} X$, $\triangleq \{x:X \mid x \in S \wedge x \in T\}$.

$S \cup T$
Set union: given $S, T: \mathbb{P} X$, $\triangleq \{x:X \mid x \in S \vee x \in T\}$.

$S \setminus T$
Set difference: given $S, T: \mathbb{P} X$, $\triangleq \{x:X \mid x \in S \wedge x \notin T\}$.

$\cap SS$
Distributed set intersection: given $SS: \mathbb{P} (\mathbb{P} X)$, $\triangleq \{x:X \mid (\forall S:SS \bullet x \in S)\}$.

$\cup SS$
Distributed set union: given $SS: \mathbb{P} (\mathbb{P} X)$, $\triangleq \{x:X \mid (\exists S:SS \bullet x \in S)\}$.

$\# S$
Size (number of distinct elements) of a finite set.

μS
Arbitrary choice from a set.

4. Relations.

A relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations.

Let X, Y, and Z be sets; $x: X$; $y: Y$; and $R: X \leftrightarrow Y$.

$X \leftrightarrow Y$
The set of relations from X to Y: $\triangleq \mathbb{P} (X \times Y)$.

$x \; R \; y$
x is related by R to y: $\triangleq (x, y) \in R$. (R is often underlined for clarity.)

$x \mapsto y$ $\triangleq (x, y)$.

$\{ x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n \}$
The relation $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ relating x_1 to y_1, . . . , and x_n to y_n.

$\mathrm{dom}\, R$
The domain of a relation: $\triangleq \{ x:X \mid \exists y:Y \bullet x \; R \; y \}$.

$\mathrm{ran}\, R$
The range of a relation: $\triangleq \{ y:Y \mid \exists x:X \bullet x \; R \; y \}$.

$R_1 \fatsemi R_2$
Forward relational composition: given $R_1: X \leftrightarrow Y$; $R_2: Y \leftrightarrow Z$, $\triangleq \{ x:X; z:Z \mid \exists y:Y \bullet x \; R_1 \; y \wedge y \; R_2 \; z \}$.

$R_1 \circ R_2$
Relational composition: $\triangleq R_2 \fatsemi R_1$.

R^{-1}
Inverse of relation R: $\triangleq \{ y:Y; x:X \mid x \; R \; y \}$.

$\mathrm{id}\, X$
Identity function on the set X: $\triangleq \{ x:X \bullet x \mapsto x \}$.

R^k
The relation R composed with itself k times: given $R: X \leftrightarrow X$, $R^0 \triangleq \mathrm{id}\, X$, $R^{k+1} \triangleq R^k \circ R$.

R^*
Reflexive transitive closure: $\triangleq \cup \{n: \mathbb{N} \bullet R^n\}$.

R^+
Non-reflexive transitive closure: $\triangleq \cup \{n: \mathbb{N}_1 \bullet R^n\}$.

$R(S)$
Relational image: given $S: \mathbb{P} X$, $\triangleq \{ y:Y \mid \exists x: S \bullet x \; R \; y \}$.

$S \lhd R$
Domain restriction to S: given $S: \mathbb{P} X$, $\triangleq \{x:X; y:Y \mid x \in S \wedge x \; R \; y\}$.

$S \lhd\!\!\!- R$
Domain subtraction: given $S: \mathbb{P} X$, $\triangleq (X \setminus S) \lhd R$.

$R \rhd T$
Range restriction to T: given $T: \mathbb{P} Y$, $\triangleq \{x:X; y:Y \mid x \; R \; y \wedge y \in T\}$.

$R \rhd\!\!\!- T$
Range subtraction of T: given $T: \mathbb{P} Y$, $\triangleq R \rhd (Y \setminus T)$.

5. Functions.

A function is a relation with the property that for each element in its domain there is a unique element in its range related to it. As functions are relations all the operators for relations also apply to functions.

$X \pfun Y$ The set of partial functions from X to Y:
$\triangleq \{f : X \rel Y \mid \forall x : \mathrm{dom}\ f \bullet (\exists_1 y : Y \bullet x\ f\ y)\}$.

$X \fun Y$ The set of total functions from X to Y:
$\triangleq \{f : X \pfun Y \mid \mathrm{dom}\ f = X\}$.

$X \pinj Y$ The set of partial injective (one-to-one) functions from X to Y:
$\triangleq \{f : X \pfun Y \mid \forall y : \mathrm{ran}\ f \bullet (\exists_1 x : X \bullet f\ x = y)\}$.

$X \inj Y$ The set of total injective functions from X to Y:
$\triangleq (X \pinj Y) \cap (X \fun Y)$.

$X \psurj Y$ The set of partial surjective functions from X to Y:
$\triangleq \{f : X \pfun Y \mid \mathrm{ran}\ f = Y\}$.

$X \surj Y$ The set of total surjective functions from X to Y:
$\triangleq (X \psurj Y) \cap (X \fun Y)$.

$X \bij Y$ The set of total bijective (injective and surjective) functions from X to Y:
$\triangleq (X \surj Y) \cap (X \inj Y)$.

$X \ffun Y$ The set of finite partial functions from X to Y:
$\triangleq \{f : X \pfun Y \mid f \in \mathbb{F}\ (X \times Y)\}$.

$\pfun\ \pinj\ \psurj\ \bij$ Partial functions.

$\fun\ \inj\ \surj\ \bij$ Total functions.

$\ffun\ \finj\ \fsurj\ \fbij$ Finite functions.

$f_1 \ovl f_2$ Functional overriding: given $f_1, f_2 : X \rel Y$,
$\triangleq (\mathrm{dom}\ f_2 \domsub f_1) \cup f_2$.

$f\ \_$ Prefix function (default).

$\_\ f\ \_$ Infix function (often underlined for clarity).

$\_\ f$ Postfix function.

$f\ t$ The function f applied to t.

$f(t)$ $\triangleq f\ t$.

$(\lambda x : X \mid P \bullet t)$ Lambda-abstraction: the function that, given an argument x of type X such that P holds, the result is t.
$\triangleq \{x : X \mid P \bullet x \mapsto t\}$.

$(\lambda x_1 : T_1; \ldots ; x_n : T_n \mid P \bullet t)$
$\triangleq \{x_1 : T_1; \ldots ; x_n : T_n \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$.

6. Numbers.

Let m, n be natural numbers.

$\mathbb{N}$ The set of natural numbers (non-negative integers).

$\mathbb{N}_1$ The set of strictly positive natural numbers: $\triangleq \mathbb{N} \setminus \{0\}$.

$\mathbb{Z}$ The set of integers (positive, zero and negative).

$\mathrm{succ}\ n$ Successive ascending natural number.

$\mathrm{pred}\ n$ Previous descending natural number: $\triangleq \mathrm{succ}^{-1}\ n$.

$m + n$ Addition: $\triangleq \mathrm{succ}^n\ m$.

$m - n$ Subtraction: $\triangleq \mathrm{pred}^n\ m$.

$m * n$ Multiplication: $\triangleq (\_ + m)^n\ 0$.

$m\ \underline{\mathrm{div}}\ n$ Integer division.

$m\ \underline{\mathrm{mod}}\ n$ Modulo arithmetic.

m^n Exponentiation: $\triangleq (\_ * m)^n\ 1$.

$m \leq n$ Less than or equal, Ordering: $\_ \leq \_ \triangleq \mathrm{succ}^*$.

$m < n$ Less than, Strict ordering: $\triangleq m \leq n \wedge m \neq n$.

$m \geq n$ Greater than or equal: $\triangleq n \leq m$.

$m > n$ Greater than: $\triangleq n < m$.

$m..n$ Range: $\triangleq \{k : \mathbb{N} \mid m \leq k \wedge k \leq n\}$.

min S — Minimum of a finite set;
for $S : \mathbf{F}_1 \mathbb{N}$,
$\min S \in S \ \wedge$
$(\forall x : S \bullet x \geq \min S)$.

max S — Maximum of a finite set;
for $S : \mathbf{F}_1 \mathbb{N}$,
$\max S \in S \ \wedge$
$(\forall x : S \bullet x \leq \max S)$.

7. Sequences.

Let a, b be elements of sequences, A, B be sequences and m, n be natural numbers.

seq X — The set of sequences whose elements are drawn from X:
$\triangleq \{ A : \mathbb{N} \pfun X \mid \mathrm{dom}\ A = 1..\#A \}$.

$\langle\rangle$ — The empty sequence $\emptyset$.

$\mathrm{seq}_1\ X$ — The set of non-empty sequences:
$\triangleq \mathrm{seq}\ X \setminus \{\langle\rangle\}$

$\langle a_1, \ldots, a_n \rangle$
$\triangleq \{ 1 \mapsto a_1, \ldots, n \mapsto a_n \}$.

$\langle e_1, \ldots, e_n \rangle \frown \langle b_1, \ldots, b_m \rangle$
Concatenation:
$\triangleq \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$,
$\langle\rangle \frown A = A \frown \langle\rangle = A$.

head A — The first element of a non-empty sequence:
$A \neq \langle\rangle \Rightarrow \mathrm{head}\ A = A(1)$.

last A — The final element of a non-empty sequence:
$A \neq \langle\rangle \Rightarrow \mathrm{last}\ A = A(\#A)$.

tail A — All but the head of a sequence:
$\mathrm{tail}(\langle x \rangle \frown A) = A$.

front A — All but the last of a sequence:
$\mathrm{front}(A \frown \langle x \rangle) = A$.

rev $\langle a_1, a_2, \ldots, a_n \rangle$ — Reverse:
$\triangleq \langle a_n, \ldots, a_2, a_1 \rangle$,
$\mathrm{rev}\ \langle\rangle = \langle\rangle$.

$\frown/AA$ — Distributed concatenation:
given $AA : \mathrm{seq}(\mathrm{seq}(X))$,
$\triangleq AA(1) \frown \ldots \frown AA(\#AA)$,
$\frown/\langle\rangle = \langle\rangle$.

$\fatsemi/AR$ — Distributed relational composition:
given $AR : \mathrm{seq}\ (X \rightleftarrows X)$,
$\triangleq AR(1) \fatsemi \ldots \fatsemi AR(\#AR)$,
$\fatsemi/\langle\rangle = \mathrm{id}\ X$.

$\ovl{\bullet}/AR$ — Distributed overriding:
given $A : \mathrm{seq}\ (X \pfun Y)$,
$\triangleq AR(1) \ovl{\bullet} \ldots \ovl{\bullet} AR(\#AR)$,
$\ovl{\bullet}/\langle\rangle = \emptyset$.

squash f — Convert a finite function, $f : \mathbb{N} \pfun X$, into a sequence by squashing its domain. That is,
squash $\emptyset = \langle\rangle$,
and if $f \neq \emptyset$ then
squash f =
$\langle f(i) \rangle \frown \mathrm{squash}(\{i\} \lhd f)$
where $i = \min(\mathrm{dom}\ f)$.

S 1 A — Index restriction:
$\triangleq \mathrm{squash}(S \lhd A)$.

A ↾ T — Sequence restriction:
$\triangleq \mathrm{squash}(A \rhd T)$.

disjoint AS — Pairwise disjoint:
given AS: $\mathrm{seq}\ (\mathbb{P}\ X)$,
$\triangleq (\forall i, j : \mathrm{dom}\ AS \bullet i \neq j \Rightarrow AS(i) \cap AS(j) = \emptyset)$.

AS partitions S
$\triangleq \mathrm{disjoint}\ AS \ \wedge$
$\cup \mathrm{ran}\ AS = S$.

A in B — Contiguous subsequence:
$\triangleq (\exists C, D : \mathrm{seq}\ X \bullet C \frown A \frown D = B)$.

Schema Notation

Axiomatic definition: introduces global declarations which satisfy one or more predicates for use in the entire document.

```
declaration(s)
---
predicate(s)
```

Schema definition: a schema groups together some declarations of variables and a predicate relating these variables. There are two ways of writing schemas: vertically, for example

```
S
x : ℕ
y : seq ℕ
---
x ≤ #y
```

or horizontally, for the same example

S ≙ [x: ℕ; y: seq ℕ | x≤#y].

Use in signatures after ∀, λ, {...}, etc.:

(∀S • y ≠ ⟨⟩) ≙ (∀x:ℕ; y: seq ℕ | x≤#y • y≠⟨⟩).

Schemas as types: when a schema name S is used as a type it stands for the set of all objects described by the schema, {S}. For example, μ : S declares a variable μ with components x (a natural number) and y (a sequence of natural numbers) such that x ≤ #y.

Projection functions: the component names of a schema may be used as projection (or selector) functions. For example, given μ : S, μ.x is μ's x component and μ.y is its y component; of course, the following predicate holds: μ.x ≤ #μ.y. Additionally, given μ : X ⇸ S, μ⨾(λS.x) is a function X⇸ℕ, etc.

θS — The tuple formed from a schema's variables: for example, θS is (x, y). Where there is no risk of ambiguity, the θ is sometimes omitted, so that just "S" is written for "(x, y)".

pred S — The predicate part of a schema: e.g. pred S is x ≤ #y.

Inclusion — A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must be of the same type) and the predicates of S and T are conjoined. For example,

```
T
S
z : ℕ
---
z < x
```

is

```
x, z : ℕ
y : seq ℕ
---
x ≤ #y ∧ z < x
```

S | P — The schema S with P conjoined to its predicate part. E.g., (S | x>0) is [x:ℕ;y:seq ℕ | x≤#y ∧ x>0].

$S ; D$ The schema S with the declarations D merged with the declarations of S. For example, $(S ; z : \mathbb{N})$ is $[x, z:\mathbb{N};\ y:\mathrm{seq}\ \mathbb{N} \mid x \leq \#y\ ]$.

$S[new/old]$ Renaming of components: the schema S in which the component old has been renamed to new both in the declaration and at its every free occurrence in the predicate. For example, $S[z/x]$ is $[\ z:\mathbb{N};\ y:\mathrm{seq}\ \mathbb{N} \mid z \leq \#y\ ]$ and $S[y/x, x/y]$ is $[\ y:\mathbb{N};\ x:\mathrm{seq}\ \mathbb{N} \mid y \leq \#x\ ]$. In the second case above, the renaming is simultaneous.

Decoration Decoration with subscript, superscript, prime, etc.; systematic renaming of the variables declared in the schema. For example, S' is $[x':\mathbb{N};\ y':\mathrm{seq}\ \mathbb{N} \mid x' \leq \#y']$.

$\neg S$ The schema S with its predicate part negated. E.g., $\neg S$ is $[x:\mathbb{N};\ y:\mathrm{seq}\ \mathbb{N} \mid \neg(x \leq \#y)]$.

$S \wedge T$ The schema formed from schemas S and T by merging their declarations (see inclusion above) and conjoining (and-ing) their predicates. Given $T \triangleq [x:\mathbb{N};\ z:\mathbb{P}\,\mathbb{N} \mid x \in z]$, $S \wedge T$ is

$$\begin{array}{|l} x : \mathbb{N} \\ y : \mathrm{seq}\ \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \#y \wedge x \in z \\ \hline \end{array}$$

$S \vee T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. For example, $S \vee T$ is

$$\begin{array}{|l} x : \mathbb{N} \\ y : \mathrm{seq}\ \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \#y \vee x \in z \\ \hline \end{array}$$

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking $\mathrm{pred}\ S \Rightarrow \mathrm{pred}\ T$ as the predicate. E.g., $S \Rightarrow T$ is

$$\begin{array}{|l} x : \mathbb{N} \\ y : \mathrm{seq}\ \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \#y \Rightarrow x \in z \\ \hline \end{array}$$

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking $\mathrm{pred}\ S \Leftrightarrow \mathrm{pred}\ T$ as the predicate. E.g., $S \Leftrightarrow T$ is

$$\begin{array}{|l} x : \mathbb{N} \\ y : \mathrm{seq}\ \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \#y \Leftrightarrow x \in z \\ \hline \end{array}$$

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema S with the variables v_1, v_2, ..., and v_n hidden: the variables listed are removed from the declarations and are existentially quantified in the predicate. E.g., $S \setminus x$ is $[y : \mathrm{seq}\ \mathbb{N} \mid (\exists x : \mathbb{N} \bullet x \leq \#y)]$. (We omit the parentheses when only one variable is hidden.) A schema may be specified instead of a list of variables; in this case the variables declared in that schema are hidden. For example, $(S \wedge T) \setminus S$ is

$$z : \mathbb{P}\ \mathbb{N}$$
$$(\exists x : \mathbb{N};\ y : \mathrm{seq}\ \mathbb{N} \bullet x \leq \#y \wedge x \in z)$$

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema S with any variables that do not occur in the list v_1, v_2, ..., v_n hidden: the variables removed from the declarations are existentially quantified in the predicate.
E.g., $(S \wedge T) \upharpoonright (x, y)$ is

$$x : \mathbb{N}$$
$$y : \mathrm{seq}\ \mathbb{N}$$
$$(\exists z : \mathbb{P}\ \mathbb{N} \bullet x \leq \#y \wedge x \in z)$$

As for hiding above, we may project a single variable with no parentheses or the variables in a schema.

The following conventions are used for variable names in those schemas which represent operations - that is, which are written as descriptions of operations on some state:

undashed	state before,
dashed ("′")	state after,
ending in "?"	inputs to (arguments for),
ending in "!"	outputs from (results of) the operation.

The following schema operations only apply to schemas following the above conventions.

pre S — Precondition: all the state after components (dashed) and the outputs (ending in "!") are hidden. E.g. given

S
$$x?, s, s', y! : \mathbb{N}$$
$$s' = s - x? \wedge y! = s$$

pre S is

$$x?, s : \mathbb{N}$$
$$(\exists s', y! : \mathbb{N} \bullet s' = s - x? \wedge y! = s)$$

post S — Postcondition: this is similar to precondition except all the state before components (undashed) and inputs (ending in "?") are hidden. (Note that this definition differs from some others, in which the "postcondition" is the predicate relating all of initial state, inputs, outputs, and final state.)

S ⊕ T Overriding:
≙ (S ∧ ¬pre T) ∨ T.
For example, given S above and

T

$$x?,\ s,\ s' : \mathbb{N}$$
$$s < x? \land s' = s$$

S ⊕ T is

$$x?,\ s,\ s',\ y! : \mathbb{N}$$
$$(s' = s - x? \land y! = s \land \neg(\exists\, s' : \mathbb{N} \bullet s < x? \land s' = s)) \lor (s < x? \land s' = s)$$

Because (given the declaration s: N above):
(∃s': N • s'=s ∧ s<x?) ⇔
(s ∈ N ∧ s < x?) ⇔
s < x?,
the predicate can be simplified:

$$x?,\ s,\ s',\ y! : \mathbb{N}$$
$$(s' = s - x? \land y! = s \land s \geq x?) \lor (s < x? \land s' = s)$$

S ⨾ T Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the state-after components of S and the state-before components of T that have a basename* in common, rename both to new variables, take the schema which is the "and" (∧) of the resulting schemas, and hide the new variables. E.g., S ⨾ T is

$$x?, s,\ s',\ y! : \mathbb{N}$$
$$(\exists\, s_0 : \mathbb{N} \bullet s_0 = s - x \land y! = s \land s_0 < x? \land s' = s_0)$$

* basename is the name with any decoration ("'", "!", "?", etc.) removed.

S >> T Piping: this schema operation is similar to schema composition; the difference is that, rather than identifying the state after components of S with the state before components of T, the output components of S (ending in "!") are identified with the input components of T (ending in "?") that have the same basename.

The following conventions are used for prefixing of schema names:

ΔS	change of before and after state,
ΞS	no change of state,
ΦS	framing schema for definition of further operations.

For example

$$\Delta S \triangleq S \land S'$$
$$\Xi S \triangleq \Delta S \mid \theta S = \theta S'$$
$$\Phi S \triangleq \Delta S \mid y = y'$$
$$S_{OP} \triangleq \Phi S \mid x' = 0$$