

# Chapter 4: An Example Microprocessor

Before we start looking at the details of meta-level reasoning in Maude (which we will need for verification), we will consider a simple example processor at the level of abstraction seen by an assembly language programmer. We will call this the *Programmer's Model*. It is often called other things (for example, the Architecture): unfortunately, the other commonly-used names are a bit vague on occasion (which is why we have our own). The complete file can be found [here](#). I strongly recommend running it and making some experimental changes. (Note: ignore the warning about the `__` operator.)

## 4.1. Basic Operations on Bit Strings

The first module below deals with basic operations over bit strings of arbitrary length. It introduces the sorts `Bit` (a single bit) and `Bits` (a string of bits), the constants `0` and `1`, as well as some useful operators. These are bit string concatenation (`__` - two underscores, so you just write two bit strings next to each other, separated by spaces, to concatenate them); length (`|_`); addition; delete leading zeros (`%`); and bit string extraction (`bits`). Note we do not need `%` in this example, but we leave it in because it was used in the *last chapter*. Note also that we import the `[system]` module `MACHINE-INT`.

```
***
*** Module for dealing with machine words as
*** sequences of bits
***
fmod BITS is
  protecting MACHINE-INT .

  sorts Bit Bits .

  subsort Bit < Bits .

  ops 0 1 : -> Bit .

  op __ : Bits Bits ->
        Bits [assoc prec 1 gather (e E)] .

  op |_| : Bits -> MachineInt .

  op _++_ : Bits Bits ->
        Bits [assoc comm prec 3 gather (E e)] .

  op bits : Bits MachineInt MachineInt -> Bits .

  vars S T : Bits .
  vars B C : Bit .
  var L : Bool .
  var I J : MachineInt .

  *** Length
  eq | B | = 1 .
  eq | S B | = | S | + 1 .

  *** Extract Bits...
  eq bits(S B,0,0) = B .
  eq bits(B,J,0) = B .
  ceq bits(S B,J,0) = bits(S, J - 1,0) B if J > 0 .
  ceq bits(S B,J,I) = bits(S,J - 1,I - 1)
    if I > 0 and J > 0 .

  *** Binary addition
  eq 0 ++ S = S .
  eq 1 ++ 1 = 1 0 .
  eq 1 ++ (T 0) = T 1 .
  eq 1 ++ (T 1) = (T ++ 1) 0 .
  eq (S B) ++ (T 0) = (S ++ T) B .
  eq (S 1) ++ (T 1) = (S ++ T ++ 1) 0 .
```

```
endfm
```

Notice we have used ++ for addition over bit strings. This is because we will later want to use + for addition of fixed length bit strings. We will do this by using ++ and then truncating to the appropriate length. If we did not do this, then we would end up with a non-terminating system.

## 4.2. Machine Word Module

The next module, which imports BITS, defines the various different length substrings we will need, along with various operators that extract bit strings and perform arithmetic. Our machine will have 10-bit memory addresses, and 16-bit words (memory will be word addressable). Instructions are 16 bits, with a 3-bit opcode field and either three 3-bit operand fields representing registers, or one 3-bit and one 10-bit operand field, representing a register and a memory address.

```
***
*** Module for dealing with machine words and instruction formats.
***
fmod MACHINE-WORD is
  protecting BITS .      *** Import BITS module defined above

  *** 3-bit OpField, 10-bit Address, 16-bit Word
  sorts OpField Address Word .

  subsort OpField < Bits .
  subsort Address < Bits .
  subsort Word < Bits .

  *** Instruction field extraction functions
  op opcode : Word -> OpField .
  ops rega regb regc : Word -> OpField .
  ops address value : Word -> Address .

  *** 16-bit addition. For addition instruction.
  op _+_ : Word Word -> Word .

  *** 10-bit addition. For address arithmetic.
  op _+_ : Address Address -> Address .

  var Y : Bits .

  vars A1 A2 A3 A4 A5 A6 A7 A8 : Bits .
  vars A9 A10 A11 A12 A13 A14 A15 A16 : Bits .

  var A B : Word .
  var C D : Address .

  mb (A1 A2 A3) : OpField .
  mb (A1 A2 A3 A4 A5 A6 A7 A8 A9 A10) : Address .
  mb (A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13
      A14 A15 A16) : Word .

  *** EXPERIMENT: Try uncommenting the three
  *** [conditional] membership axioms below,
  *** and commenting out the three above.
  *** (You may want to change the reductions
  *** at the bottom to something much simpler!)
  ***cmb Y : OpField if | Y | = 3 .
  ***cmb Y : Address if | Y | = 10 .
  ***cmb Y : Word if | Y | = 16 .

  *** opcode and rega always bits 15-13, 12-10 resp.
  eq opcode(A) = bits(A,15,13) .
  eq rega(A) = bits(A,12,10) .

  *** regb and regc are bits 9-7, 6-4 resp. (3-0 void)
  eq regb(A) = bits(A,9,7) .
  eq regc(A) = bits(A,6,4) .
```

```

*** value (set instr only) and address are bits 9-0
*** Note that the defns. of these are the same, so
*** we could omit one.
eq value(A) = bits(A,9,0) .
eq address(A) = bits(A,9,0) .

*** We define 16 and 10 bit addition to be bitwise
*** addition (from the module Bits above) truncated
*** to 16 and 10 bits respectively Notice we need to
*** truncate because n-bit addition generates an
*** n+1 bit result
eq A + B = bits(A ++ B,15,0) .
eq C + D = bits(C ++ D,9,0) .

```

endfm

Notice that if we had not used ++ for addition over arbitrary length bit strings, the final two equations would have led to a non-terminating system (Exercise: explain why).

### 4.3. Memory and Registers

The following modules deal with memory and registers. Note that they are very similar In retrospect, it may be better to use a different notation for registers, to avoid confusion. However, the risk is small.

```

***
*** Module for representing memory..
***
fmod MEM is

  protecting MACHINE-WORD .

  sorts Mem .      *** New type representing memory

  op _[_] : Mem Address -> Word .      *** Read
  op _[_/_] : Mem Word Address -> Mem .  *** Write

  var M : Mem .

  vars A B : Address .
  var W : Word .

  eq M[W / A][A] = W .
  ceq M[W / A][B] = M[B] if A /= B .
endfm

***
*** Module for representing registers.
***
fmod REG is

  protecting MACHINE-WORD .

  sorts Reg .      *** New type representing registers.

  op _[_] : Reg OpField -> Word .      *** Read
  op _[_/_] : Reg Word OpField -> Word .  *** Write

  var R : Reg .
  var A B : OpField .
  var W : Word .

  eq R[W / A][A] = W .
  ceq R[W / A][B] = R[B] if A /= B .
endfm

```

#### 4.4. State Tuple Module

The following module represents the state tuple of the microprocessor - a program memory, a data memory, a program counter, and a set of registers.

```

***
*** State of SPM, together with tupling and
*** projection functions
***

fmod SPM-STATE is
  protecting MEM .
  protecting REG .

  sort SPMstate .

  op (_,_,_,_) : Mem Mem Word Reg -> SPMstate .

  *** Project out program and data memory
  ops mp_ md_ : SPMstate -> Mem .

  *** Project out program counter
  op pc_ : SPMstate -> Address .

  *** project out registers
  op reg_ : SPMstate -> Reg .

  var S : SPMstate .
  vars MP MD : Mem .
  var PC : Address .
  var REG : Reg .

  eq mp(MP,MD,PC,REG) = MP .
  eq md(MP,MD,PC,REG) = MD .
  eq pc(MP,MD,PC,REG) = PC .
  eq reg(MP,MD,PC,REG) = REG .
endfm

```

#### 4.5. Main State and Next State Functions

This module describes the behaviour of the microprocessor. The state function generates a future state by repeatedly applying a next state function. There are five instructions: add, branch, load, store and set. Notice that we only use five of eight opcodes, and we really should have some "exception" behaviour for the others. However, we omit this here. Notice also that we must be careful to include the appropriate number of leading zeros: the length of a bit string is used to determine its [sub] sort, so getting this wrong leads to errors.

```

***
*** SPM
***
*** This is the "main" function, where we define the
*** state function spm and the next-state function next.
***

fmod SPM is
  protecting SPM-STATE .

  *** A few handy constants..
  ops ADD, BR, LD, ST, SET : -> OpField .
  op ZERO : -> Word .

  *** State function
  op spm : MachineInt SPMstate -> SPMstate .

  *** Next-state function
  op next : SPMstate -> SPMstate .

  var SPM : SPMstate .

```

```

var T : MachineInt .
var MP MD : Mem .
var PC : Word .
var REG : Reg .

*** Define opcodes etc.
eq ADD = 0 0 0 .
eq BR = 0 0 1 .
eq LD = 0 1 0 .
eq ST = 0 1 1 .
eq SET = 1 0 0 .
eq ZERO = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 .
eq One = 0 0 0 0 0 0 0 0 0 0 1 .

*** Formally, the state function is an Iterated Map
*** Primitive recursive definition - repeatedly apply
*** the state function
eq spm(0,SPM) = SPM .
ceq spm(T,SPM) = next(spm(T - 1,SPM)) if T > 0 .

*** Addition (opcode = 0)
ceq next(MP,MD,PC,REG) = (MP, MD, PC + One,
  REG[REG[rega(MP[PC])] +
  REG[regb(MP[PC])] / regc(MP[PC])])
  if opcode(MP[PC]) == ADD .

*** [Taken] branch (opcode = 1, REG[0] = 0)
ceq next(MP,MD,PC,REG) = (MP, MD, PC +
  address(MP[PC]), REG)
  if opcode(MP[PC]) == BR and REG[0] == ZERO .

*** [Not-taken] branch (opcode = 1, REG[0] /= 0)
ceq next(MP,MD,PC,REG) = (MP, MD, PC + One, REG)
  if opcode(MP[PC]) == BR and REG[0] /= ZERO .

*** Load (opcode = 10)
ceq next(MP,MD,PC,REG) = (MP, MD, PC + One,
  REG[MD[address(MP[PC])] / rega(MP[PC])])
  if opcode(MP[PC]) == LD .

*** Store (opcode = 11)
ceq next(MP,MD,PC,REG) = (MP,
  MD[REG[rega(MP[PC])] / address(MP[PC])],
  PC + One, REG) if opcode(MP[PC]) == ST .

*** Set (opcode = 100)
ceq next(MP,MD,PC,REG) = (MP, MD, PC + One,
  REG[value(MP[PC]) / rega(MP[PC])])
  if opcode(MP[PC]) == SET .
endfm

```

## 4.6. Example Program

Now we get to the final module, which defines a very simple program with two instructions. After the module, we actually run the program.

```

***
*** The final module is to define an actual program and
*** run it. We will just define a two instruction
*** program.
***
fmod RUNPROGS is
  protecting SPM .

  ops Md Mp : -> Mem .      *** Define memories as
                             *** *constants*,
                             *** NOT *variables*

```

```

op Rg : -> Reg .          *** Same for registers...
op Pc : -> Word .        *** ... and program counter

*** Now specify a starting state...
*** Notice that programming is a bit of a pain...

*** Program counter will start at zero
eq Pc = 0 0 0 0 0 0 0 0 0 0 .

*** Register 1 contains 3
eq Rg[0 0 1] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 .

*** Register 2 contains 6
eq Rg[0 1 0] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 .

eq Mp[0] = 0 0 0 0 0 1 0 1 0 *** Add R1 to R2
           0 1 1 0 0 0 0 . *** storing in R3

eq Mp[1] = 0 1 1 0 1 1 *** Store R3 in
           0 0 0 0 0 0 0 0 1 . *** memory loc. 1
endfm

***
*** Now run the program
***

*** First reduction shows state after running just
*** one instruction (the add)
reduce spm(1, (Mp,Md,Pc,Rg)) .

*** Second reduction shows state after running both
*** instructions (add and store)
reduce spm(2, (Mp,Md,Pc,Rg)) .

```