Modeling a CPU in Formula

Overview

Formula is a specification language that can be used to reason about models. While it does not contain the typical primitives for a sequential program, it is possible to model a simple ARM-like CPU in Formula that can make decisions and execute loops. Using Formula's list functions, it is also possible to model memory.

Machine State

The core of the CPU model is the MachineState state structure, defined as:

```
MachineState ::= new (seq : Integer, pc: Integer, cpsr: CPSR, regs : Registers, mem: Memory).
```

The seq field in MachineState is really the key to the CPU model. Any operation that results in a change of the machine state results in a new seq value. If a program contains a loop, there may be several instances of MachineState with a particular pc (program counter) value, but each one of them will have a different seq value. Currently, each instruction results in a single machine state change, so when a program halts, the seq value when the halt occurs is actually the number of instructions that have been executed. It might not be possible to create instructions that perform multiple machine state updates, because two machine states with the same program counter could essentially cause an instruction to be executed twice.

The Registers structure contains 16 integer registers:

```
Registers ::= new ( r0: Integer, r1: Integer, r2: Integer, r3: Integer,
                    r4: Integer, r5: Integer, r6: Integer, r7: Integer,
                    r8: Integer, r9: Integer, r10: Integer, r11: Integer,
                    r12: Integer, r13: Integer, r14: Integer).
```

Typically in a CPU, the status register is a series of bits. Because Formula does not have built-in bitwise operations, we instead model a few flags in the CPSR (Current Program Status Register) with a structure containing boolean values. We model the zero (z), negative (n), carry (c), and overflow (v) bits:

```
CPSR ::= new ( n: Boolean, z: Boolean, c: Boolean, v: Boolean ).
```

Memory is modeled as a pair of lists. The first list is the list of addresses stored in memory, the second is the list of values stored in those locations:

```
Memory ::= new (addrList : NumList+{NIL}, valList : NumList+{NIL}).
```

While the program counter in a CPU is often part of the register set, we chose to model it separately from the numbered register bank. We also include a sequence number, which is the key to modeling program execution.

An immediate value is wrapped inside a new-kind constructor:

```
Immediate ::= new (val : Integer).
```

Execution

The trick to modeling program execution in Formula is that you aren't so much executing a sequential set of instructions that manipulate the current program state, but you are actually generating program state structures for every state the program is in. This implies that if the program does not terminate, Formula will run out of memory.

Each instruction in the CPU model has a program counter value, which can be thought of as creating new machine states for every machine state that exists with that program counter value. For example, the Mul structure is declared this way:

```
Mul ::= new (pc : Integer, r0 : Integer, r1 : Integer + Immediate, r2 : Integer).
```

This means, in any machine state where the program counter = pc, take the value in register number r0, and multiply it by either the register with number r1 or the immediate value of r1, and place the result in register number r2. For example, here is an instruction from our sample factorial program:

```
Mul(4, 0, 1, 1).
```

This instruction says that in any machine state where PC=4, take the value in register 0, multiply it by the value in register 1, and store the result in register 1. In order to actually implement this multiplication, we have to set up rules that generate a new machine state for various possible results of the multiply instruction.

First, we define a SetState rule that makes it easier to set a specific register value:

```
SetState ::= new (seq: Integer, pc: Integer, status: CPSR, reg : Integer, val :
Integer).
```

The basic idea of SetState is to set the contents of register number *reg* to *val* and possibly make changes to the sequence number, pc, and CPSR. For each possible register to set, there is a separate rule that makes a new machine state with that register changed. For instance, here is the rule for changing register 0:

```
      MachineState(nseq, pc, status, Registers(val, regs.r1, regs.r2, regs.r3,
regs.r4, regs.r5, regs.r6, regs.r7, regs.r8, regs.r9, regs.r10, regs.r11, regs.r12,
regs.r13, regs.r14)) :-
                SetState(seq, pc, status, 0, val), MachineState(seq, _, _, regs), nseq
= seq + 1.
```

This rule says that given a SetState structure with *reg*=0, set the value of *regs.r0* to *val*, making a new MachineState instance where its sequence number is *seq+1* and update the pc and CPSR to whatever were contained in the SetState structure. There are 16 instances of this rule, one for each possible value of *reg*.

Now, to implement the Mul instruction, there are 6 possible results (there would be more if we implemented overflow detection). When you multiply the two values together, the result may be positive, negative, or zero, and then the second value to be multiplied can be either a register or an immediate value, giving 6 combinations. The rule for multiplying two registers together when the result is equal to 0 is this:

```
    SetState(seq, newpc, cpsr, r2, val) :- Mul(pc, r0, r1, r2), GetReg(seq, pc,
status, r0, ra), GetReg(seq, pc, status, r1, rb), val = ra*rb,
        val = 0, cpsr=CPSR(FALSE, TRUE, FALSE, FALSE), newpc = pc + 1.
```

This says that given a Mul instruction for registers *r0* and *r1* fetch the values in those registers, multiply them together, store the result in register *r2* and if the result is 0, set the z flag in the CPSR to TRUE, while setting the other flags to FALSE, and then increment the PC to point to the next instruction. Additional versions of this rule handle the flag settings for positive and negative results, and three more rules handle the case where *r1* is an immediate value.

In a program that loops, such as the example factorial program, the Mul instruction at PC=4 can cause many machine states to be generated. Each machine state has a unique sequence number, so that perhaps the first state with PC=4 has a sequence number of 4, and the next one might have a sequence number of 12, and the next 20, and so forth.

We provide a Halt instruction to indicate that the program has terminated:

```
Halt ::= new (pc : Integer).
```

We also provide a HaltedAt rule to facilitate querying the machine state:

```
HaltedAt ::= new (seq : Integer, pc : Integer).
HaltedAt(seq,pc) :- Halt(pc), MachineState(seq,pc,_,_).
```

The following query would find the sequence number and pc of the machine state when the Halt instruction was executed:

```
qr FactTest HaltedAt(_,_)
```

Then, you can use the tr command to see the HaltedAt value:

```
[]> tr 0 HaltedAt(seq, pc)
Listing all derived values...
   HaltedAt(14, 2)
List complete
0.03s.
```

Because of the way Formula prints out proofs, we do not recommend using the pr command to print query results, as it can generate many megabytes of output on even a small program. We have modified the tr command so it can take non-ground terms so that you can query for results with it. You don’t have to use the same term that you queried for. For example, this tr

command on the same query as above, shows the whole machine state when it halted (given that it halted at seq=14, which is the first value in the MachineState structure):

```
[]> tr 0 MachineState(14,pc,cpsr,regs,mem)
Listing all derived values...regs,mem)
   MachineState(14, 2, CPSR(FALSE, FALSE, FALSE, FALSE), Registers(6, 6, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 2), Memory(NIL, NIL))
List complete
0.00s.
```

Since you will probably often want to display the machine state, there is a HaltedAtState rule to make it easier to examine the machine state when halted:

```
[]> tr 0 HaltedWithState(state)
Listing all derived values...
   HaltedWithState(MachineState(14, 2, CPSR(FALSE, FALSE, FALSE, FALSE), Registers(6,
6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2), Memory(NIL, NIL)))
List complete
0.03s.
```

Memory Access

Because Terms in Formula are immutable, memory cannot be modeled as a mutable array. Instead, we model it as a pair of lists, where one list contains the addresses of memory locations, and the other contains the values stored at those location.

Storing Into Memory

To store a value into memory, you simply put the destination address at the front of the memory address list, and the stored value at the front of the memory value list. For example:

```
newmem = Memory(NumList(addr, oldaddrs), NumList(val, oldvals)).
```

Fetching From Memory

Fetching from memory is slightly more complex. We first search the memory address list to see if the address occurs in the list:

```
idx = lstFindAll(#NumList, mem.addrList, addr).
```

If `idx` is NIL, we consider the value in that memory location to be 0 (i.e. all memory is initialized to 0). The `lstFindAll` function returns a list of indices where a particular value is found in a list, and so we take the first value from that list and use it to fetch the corresponding value from the memory values list:

```
idx != NIL, pos = idx.val, val = lstGetAt(#NumList, mem.valList, pos).
```

Ld (Load) and St (Store)

The ARM LD and ST instructions transfer data between a register and memory. The memory address can be computed a number of ways such as having an explicit value in the instruction, using an address stored in a register, adding an offset to a register, etc. For both loading and storing, we split the operations into two parts, one computes the memory address based on the addressing mode, and the other performs the load or store. For example, here is the implementation of the Ld instruction:

```
Ld ::= new (pc : Integer, rdest : Integer,
            addr : Integer+Indexed+IndexedOffset+DoubleIndexed).
Load(seq,newpc,addr,rdest) :-
    Ld(pc, rdest, addr), newpc=pc+1, MachineState(seq,pc,_,_,_), addr:Integer.
Load(seq,newpc,addr,rdest) :-
    Ld(pc, rdest, indaddr), indaddr=Indexed(reg), newpc=pc+1,
    GetReg(seq,pc,_,_,reg,addr).
Load(seq,newpc,addr,rdest) :-
    Ld(pc, rdest, indoffaddr), indoffaddr=IndexedOffset(reg,offset), newpc=pc+1,
    GetReg(seq,pc,_,_,reg,baseaddr), addr=baseaddr+offset.
Load(seq,newpc,addr,rdest) :-
    Ld(pc, rdest, doubleidx), doubleidx=DoubleIndexed(reg1, reg2), newpc=pc+1,
    GetReg(seq,pc,_,_,reg1,baseaddr), GetReg(seq,pc,_,_,reg2, offset),
    addr=baseaddr+offset.
```

A Ld instruction can trigger a Load rule. For each possible address value, whether an immediate integer value, a register number, a register + an offset, or two registers, the final address is computed, resulting in a new Load rule for the current sequence number and program counter. The Load rule then fetches the value from memory and stores it in the destination register:

```
Load ::= new (seq : Integer, newpc : Integer, addr : Integer, rdest : Integer).
SetReg(seq, newpc, newcpsr, mem, rdest, val) :-
    Load(seq,newpc,addr,rdest), MachineState(seq,_,_,_,mem),
      val=0, newcpsr=CPSR(FALSE,TRUE,FALSE,FALSE),
      idx=lstFindAll(#NumList, mem.addrList, addr), idx=NIL.
SetReg(seq, newpc, newcpsr, mem, rdest, val) :-
    Load(seq,newpc,addr,rdest), MachineState(seq,_,_,_,mem),
      val>0, newcpsr=CPSR(FALSE,FALSE,FALSE,FALSE),
      idx=lstFindAll(#NumList, mem.addrList, addr), idx!=NIL, pos=idx.val,
      val=lstGetAt(#NumList,mem.valList,pos).
SetReg(seq, newpc, newcpsr, mem, rdest, val) :-
    Load(seq,newpc,addr,rdest), MachineState(seq,_,_,_,mem),
      val=0, newcpsr=CPSR(FALSE,TRUE,FALSE,FALSE),
      idx=lstFindAll(#NumList, mem.addrList, addr), idx!=NIL, pos=idx.val,
      val=lstGetAt(#NumList,mem.valList,pos).
SetReg(seq, newpc, newcpsr, mem, rdest, val) :-
    Load(seq,newpc,addr,rdest), MachineState(seq,_,_,_,mem),
      val<0, newcpsr=CPSR(TRUE,FALSE,FALSE,FALSE),
      idx=lstFindAll(#NumList, mem.addrList, addr), idx!=NIL, pos=idx.val,
      val=lstGetAt(#NumList,mem.valList,pos).
```

As is the case for many operations, there are three cases that vary based on whether the resulting value (in this case, the result fetched from memory) is zero, positive, or negative. This results in different flags being set in the CPSR structure. There is an additional case, which happens when there is a load from a memory address that isn't in the memory list. The

first case handles that, by checking whether idx=NIL where idx is the list of entries in the memory address list that match the requested address. When that happens, it assumes the value is 0.

The load triggers a SetReg rule, which in turn triggers a new MachineState rule containing a new machine state where the machine state sequence number has been incremented, and the Registers structure has been updated to contain the new register value.

The St instruction has rules very similar to those of Ld, having the same set of cases for the different addressing modes and computing a final address before triggering a Store rule. Unlike the Load rule, the Store rule only needs the three cases for the value being zero, positive, or negative, because it doesn't care if the address is already present in memory:

```
Store ::= new (seq : Integer, newpc : Integer, addr : Integer, val : Integer).
MachineState(newseq,newpc,newcpsr,regs,newmem) :-
    Store(seq, newpc, addr, val), MachineState(seq, _, _, regs, mem),
      newseq=seq+1,val > 0, newcpsr=CPSR(FALSE,FALSE,FALSE,FALSE),
      oldaddrs = mem.addrList, oldvals = mem.valList,
      newmem=Memory(NumList(addr,oldaddrs),NumList(val,oldvals)).
MachineState(newseq,newpc,newcpsr,regs,newmem) :-
    Store(seq, newpc, addr, val), MachineState(seq, _, _, regs, mem),
      newseq=seq+1, val = 0, newcpsr=CPSR(FALSE,TRUE,FALSE,FALSE),
      oldaddrs = mem.addrList, oldvals = mem.valList,
      newmem=Memory(NumList(addr,oldaddrs),NumList(val,oldvals)).
MachineState(newseq,newpc,newcpsr,regs,newmem) :-
    Store(seq, newpc, addr, val), MachineState(seq, _, _, regs, mem),
      newseq=seq+1, val < 0, newcpsr=CPSR(TRUE,FALSE,FALSE,FALSE),
      oldaddrs = mem.addrList, oldvals = mem.valList,
      newmem=Memory(NumList(addr,oldaddrs),NumList(val,oldvals)).
```

Since the St instruction just updates memory, it doesn't need the extra step of using SetReg, but instead generates a new MachineState with an updated sequence number, prepending the destination address to the memory address list, and the stored value to the memory value list.

Implementation

The reason memory is implemented using a pair of lists as opposed to a single list of (addr, value) pairs is that it is difficult if not impossible to search through a list of pairs to find a value that matches the first item of the pair while allowing any value for the second. Formula tends to split the possibilities here and match each possible (addr, value) pair. With the pair of lists, we can search for the specific address we want without trying to match on a value, and then just fetch the value from the value list by its index.

One problem with this implementation technique is that when an address is modified multiple times, the old addresses and values remain in the address and values lists. While this might be helpful in analyzing program traces, it could pose runtime memory issues if the memory lists grow very large.

A Sample Program

This is a sample program that computes a factorial:

```
model FactTest of ARM
{
    MachineState(0, 0, CPSR(FALSE,FALSE,FALSE,FALSE),
Registers(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)).

    Mov(0, Immediate(3), 0).
    BL(1, 3).
    Halt(2).

    Mov(3, Immediate(1), 1).
    Mul(4, 0, 1, 1).
    Sub(5, 0, Immediate(1), 0).
    BNE(6, 4).
    Mov(7, 1, 0).
    Ret(8).
}
```

The basic flow of this program is to put a 3 into register r0, indicating that we want to compute the factorial of 3, and then the BL (branch-and-link, ARM-style) calls the subroutine at PC=3, and when that subroutine returns, the program halts.

The subroutine starting at PC=3 puts the immediate value 1 into register 1, and then multiplies register 0 and register 1 together, putting the result back into register 1. Next, it subtracts 1 from register r0, and then if that value is not zero, it branches back to PC=4, where it performs the multiply again, and so forth. Finally, when register 0 reaches 0, it moves register 1 to register 0 (register 0 is where the return value of the subroutine should be stored) and then executes a return.

Recursion With a Stack

Although the model does not provide any instructions to push and pop, and the register addressing does not support automatic pre-/post- increment/decrement, it is still fairly easy to implement a stack for a recursive program. The following program is a recursive factorial program that saves its return address and the previous R0 value onto the stack:

```
model FactRec of ARM
{
    MachineState(0, 0, CPSR(FALSE,FALSE,FALSE,FALSE),
Registers(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),Memory(NIL,NIL)).
    Mov(0, Immediate(8000), 13).
    // Param=5
    Mov(1, Immediate(5), 0).
    // Call fact
    BL(2, 4).
    Halt(3).

    // Save ret addr on stack
    St(4, 14, Indexed(13)).
    // Decrement stack pointer
```

```
    Sub(5, 13, Immediate(1), 13).
    // Is arg <= 1 ?
    Mov(6, Immediate(1), 1).
    Cmp(7, 0, 1).
    BLE(8, 19).
    // If not, save the old arg on the stack
    St(9, 0, Indexed(13)).
    // Decrement stack pointer
    Sub(10, 13, Immediate(1), 13).
    // Subtract 1 from arg
    Sub(11, 0, Immediate(1), 0).
    // Compute fact of arg-1
    BL(12, 4).
    // Increment stack pointer
    Add(13, 13, Immediate(1), 13).
    // Get old arg into r1
    Ld(14, 1, Indexed(13)).
    // multiple r1 by fact of arg-1
    Mul(15, 0, 1, 0).
    // Increment stack pointer
    Add(16, 13, Immediate(1), 13).
    // Get return address
    Ld(17, 14, Indexed(13)).
    Ret(18).

    // Otherwise, if arg <= 1, put 1 in r0
    Mov(19, Immediate(1), 0).
    // Increment stack pointer
    Add(20, 13, Immediate(1), 13).
    // Load return address
    Ld(21, 14, Indexed(13)).
    Ret(22).
}
```

For a factorial of 5, it make take Formula a second or to two produce a result. You can then use HaltedWithState to view the entire machine state when it halted, including having the correct answer in R0, which is the register in which values are returned from a function:

```
[]> tr 0 HaltedWithState(state)
Listing all derived values...
   HaltedWithState(MachineState(72, 3, CPSR(FALSE, FALSE, FALSE, FALSE),
Registers(120, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8000, 3), Memory(NumList(7992,
NumList(7993, NumList(7994, NumList(7995, NumList(7996, NumList(7997, NumList(7998,
NumList(7999, NumList(8000, NIL)))))))), NumList(13, NumList(2, NumList(13,
NumList(3, NumList(13, NumList(4, NumList(13, NumList(5, NumList(3, NIL))))))))))))
List complete
0.03s.
```

You can see that the correct value, 120, is in R0. You can also see that over the course of execution, there were 9 values pushed onto the stack. Since none of them were modified in-place, there are no duplicate addresses in the memory address list.

Limitations

The model is not particularly speedy, as one might expect with something that is somewhat outside the normal usages of Formula. When the recursive factorial routine computes the

factorial of 20, it takes about 6-7 seconds on a Macbook M1, having executed a mere 297 instructions.

Because of the way Formula prints out proofs, recursively displaying a proof for each proof item, the output expands exponentially when using pr, the pr command is essentially useless for examining the program results.

This model implements a subset of ARM instructions, and only includes the first 16 ARM registers. It is possible to expand the register count to match ARM64

All memory access is at the word level, so that there is no overlap between location 8000 and 8001, they each hold separate 64-bit values. This means, also, that there are no 8-, 16- or 32-bit load/store instructions, so that it is more difficult to model byte-level algorithms