CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D4.1
Executable Formal Semantics of Machine Code

Version 1.0

Main Authors:
Dominic P. Mulligan and Claudio Sacerdoti Coen

Project Acronym: CerCo
Project full title: Certified Complexity
Proposal/Contract no.: FP7-ICT-2009-C-243881 CerCo

Abstract We discuss the implementation of a prototype O'Caml emulator for the Intel 8051/8052 eight bit processor, and its subsequent formalisation in the dependently typed proof assistant Matita. In particular, we focus on the decisions made during the design of both emulators, and how the design of the O'Caml emulator had to be modified in order to fit into the more stringent type system of Matita.

Both emulators provide an 'executable formal semantics of machine code' for our target processor, per the description of the Deliverable in the CerCo Grant Agreement.

Contents

1 Task

The Grant Agreement states that Task T4.1, entitled 'Executable Formal Semantics of Machine Code' has associated deliverable D4.1 consisting of the following:

> **Executable Formal Semantics of Machine Code**: Formal definition of the semantics of the target language. The semantics will be given in a functional (and hence executable) form, useful for testing, validation and project assessment.

This report details our implementation of this deliverable.

1.1 Connection with other deliverables

Deliverable D4.1 is an executable formal semantics of the machine code of our target processor (a brief overview of the processor architecture is provided in Section 2). We provide an executable semantics in both O'Caml and the internal language of the Matita proof assistant.

The C compiler delivered by Work Package 3 will eventually produce machine code executable by our emulator, and we expect that the emulator will be useful as a debugging aid for the compiler writers. Further, additional deliverables listed under Work Package 4 will later make use of the work reported in this document. Deliverables D4.2 and D4.3 entail the implementation of a formalised version of the intermediate language of the compiler, along with an executable formal semantics of these languages. In particular, Deliverable D4.3 requires a formalisation of the semantics of the intermediate languages of the compiler, and Deliverable D4.4 requires a formal proof of the correspondence between the semantics of these formalized languages, and the formal semantics of the target processor. The emulator(s) discussed in this report are the formalized semantics of our target processor made manifest.

2 A brief overview of the target processor

The MCS-51 is an eight bit microprocessor introduced by Intel in the late 1970s. Commonly called the 8051, in the three decades since its introduction the processor has become a highly popular target for embedded systems engineers. Further, the processor and its immediate successor, the 8052, is still manufactured by a host of semiconductor suppliers—many of them European—including Atmel, Siemens Semiconductor, NXP (formerly Phillips Semiconductor), Texas Instruments, and Maxim (formerly Dallas Semiconductor).

The 8051 is a well documented processor, and has the additional support of numerous open source and commercial tools, such as compilers for high-level languages and emulators. For instance, the open source Small Device C Compiler (SDCC) recognises a dialect of C, and other compilers targeting the 8051 for BASIC, Forth and Modula-2 are also extant. An open source emulator for the processor, MCU8051 IDE, is also available.

The 8051 has a relatively straightforward architecture, unencumbered by advanced features of modern processors, making it an ideal target for formalisation. A high-level overview of the processor's memory layout is provided in Figure 1.

Processor RAM is divided into numerous segments, with the most prominent division being between internal and (optional) external memory. Internal memory, commonly provided on the die itself with fast access, is further divided into 128 bytes of internal RAM and numerous Special Function Registers (SFRs) which control the operation of the processor. Internal

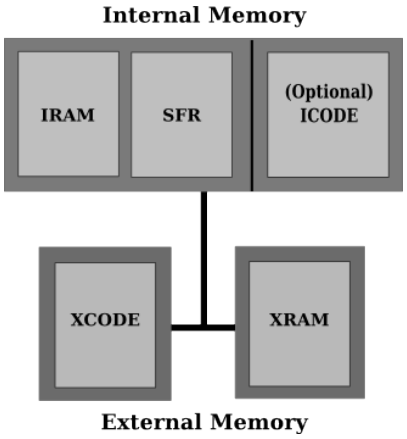

Figure 1: High level overview of the 8051 memory layout

RAM (IRAM) is further divided into a eight general purpose bit-addressable registers (R0–R7). These sit in the first eight bytes of IRAM, though can be programmatically 'shifted up' as needed. Bit memory, followed by a small amount of stack space resides in the memory space immediately after the register banks. What remains of the IRAM may be treated as general purpose memory. A schematic view of IRAM layout is provided in Figure 2.

External RAM (XRAM), limited to 64 kilobytes, is optional, and may be provided on or off chip, depending on the manufacturer. XRAM is accessed using a dedicated instruction. External code memory (XCODE) is often stored in the form of an EPROM, and limited to 64 kilobytes in size. However, depending on the particular manufacturer and processor model, a dedicated on-die read-only memory area for program code (ICODE) may also be supplied.

Memory may be addressed in numerous ways: immediate, direct, indirect, external direct and code indirect. As the latter two addressing modes hint, there are some restrictions enforced by the 8051 and its derivatives on which addressing modes may be used with specific types of memory. For instance, the 128 bytes of extra internal RAM that the 8052 features cannot be addressed using indirect addressing; rather, external (in)direct addressing must be used.

The 8051 series possesses an eight bit Arithmetic and Logic Unit (ALU), with a wide variety of instructions for performing arithmetic and logical operations on bits and integers. Further, the processor possesses two eight bit general purpose accumulators, A and B.

Communication with the device is facilitated by an onboard UART serial port, and associated serial controller, which can operate in numerous modes. Serial baud rate is determined by one of two sixteen bit timers included with the 8051, which can be set to multiple modes of operation. (The 8052 provides an additional sixteen bit timer.) As an additional method of communication, the 8051 also provides a four byte bit-addressable input-output port.

The programmer may take advantage of the interrupt mechanism that the processor provides. This is especially useful when dealing with input or output involving the serial device, as an interrupt can be set when a whole character is sent or received via the serial port.

Interrupts immediately halt the flow of execution of the processor, and cause the program counter to jump to a fixed address, where the requisite interrupt handler is stored. However, interrupts may be set to one of two priorities: low and high. The interrupt handler of an interrupt with high priority is executed ahead of the interrupt handler of an interrupt of lower priority, interrupting a currently executing handler of lower priority, if necessary.

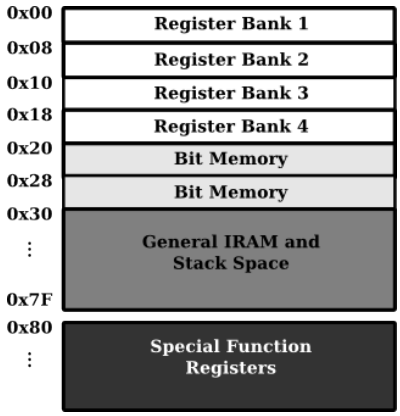

Figure 2: Schematic view of 8051 IRAM layout

The 8051 has interrupts disabled by default. The programmer is free to handle serial input and output manually, by poking serial flags in the SFRs. Similarly, 'exceptional circumstances' that would otherwise trigger an interrupt on more modern processors, for example, division by zero, are also signalled by setting flags.

3 The emulator in O'Caml

We discuss decisions made during the design of the prototype O'Caml emulator.

3.1 Lack of orthogonality in instruction set

The instruction set of 8051 assembly is highly irregular. For instance, consider the MOV instruction, which implements a data transfer between two memory locations, which takes eighteen possible combinations of addressing modes.

We handle this problem by introducing 'unions' of types, using O'Caml's polymorphic variants feature:

```
type ('a, 'b) union2 = [ `U1 of 'a | `U2 of 'b ]
```

(We also introduce `union3` and `union6`, which suffice for our purposes.)

Using these union types, we can rationalise the inductive type encoding the assembly instruction set. For instance:

```
type 'addr preinstruction =
...
  | `XRL of (acc * [ data | reg | direct | indirect ],
             direct * [ acc | data ]) union2
...
```

That is, the `XRL` instruction[1] take either the accumulator A as its first argument, followed by data with one of data, register, direct or indirect addressing modes, or takes data with a

[1] Exclusive disjunction.

direct addressing mode as its first argument, with either the accumulator A or data with the data addressing mode as its second argument.

Further, all functions that must pattern match against the `(pre)instruction` inductive type are also simplified using this technique. Using O'Caml's ability to perform 'deep pattern' matches, we may pattern match against `‘XRL(‘U1(arg1, arg2))` and have the guarantee that `arg1` takes the form `‘ACC_A`.

3.2 Pseudo-instructions and labels

Per the description of Deliverable D4.1 in the Grant Agreement above, the 8051 emulator must eventually interface with the C compiler frontend of Deliverable D3.2, produced in Paris. After consultation, it was found that the design of the compiler frontend could be simplified considerably with the introduction of *pseudoinstructions* and labels.

We introduce three new pseudoinstructions—`Jump`, `Call`, and `Mov`—corresponding to unconditional jumps, procedure calls and data transfers respectively. We also 'promote' all unlabeled conditional jumps in 8051 assembly to labeled pseudojumps; one can now jump to a label conditionally, as opposed to jumping to a fixed relative offset. Further, we introduce labels for jumping to, and cost annotations, used by the Paris team.

The three new pseudoinstructions, along with the promoted conditional jumps, allow the Paris team to abstract away from the differences between different types of unconditional jump (the 8051 has three different sorts, depending on the length of the jump), as well as abstract away the differences between memory transfers and calls. However, the emulator must perform an expansion stage, during which pseudoinstructions are translated to 'real' 8051 assembly instructions.

The introduction of labeled conditional jumps slightly complicates our type of abstract syntax for 8051 assembly. We define an inductive type representing conditional jumps in 8051 assembly code, parameterised by a type representing relative offsets:s

```
type 'addr jump =
  [ ‘JC of 'addr
  | ‘JNC of 'addr
...
```

An inductive type of preinstructions is defined, which is also parameterised by a type representing relative offsets in assembly code, and incorporates the inductive type of conditional jumps:

```
type 'addr preinstruction =
  [ ‘ADD of acc * [ reg | direct | indirect | data ]
...
  | 'addr jump
...
```

A type representing instructions is defined, choosing a concrete type for relative offsets:

```
type instruction = rel preinstruction
```

Here, `rel` is a type which 'wraps up' a byte. Finally, this type of instructions is incorporated into a type of labelled instructions:

```
type labelled_instruction =
  [ instruction
  | `Cost of string
  | `Label of string
  | `Jmp of string
  | `Call of string
  | `Mov of dptr * string
  | `WithLabel of [`Label of string] jump
]
```

Throughout, we make heavy use of polymorphic variants to deal with issues relating to subtyping.

As mentioned, the emulator must now handle an additional expansion stage, removing pseudoinstructions in favour of real, 8051 assembly instructions. This is relatively straightforward, and is done in two stages.

The first stage consists of iterating over an assembly program, building a multiset of all labels and their position in the program. This multiset is stored, and can later be used by the callback function passed to `execute`, the function that executes an 8051 assembly program, in order to produce a trace of labels. The callback function, a function from the processor state to unit, passed to `execute` implements our 'label collecting semantics'. In pseudocode:

```
let f status :=
  try
    let labels := lookup (program_counter status) (costs_map ∪ label_map) in
      ()
  with NotFound → ()
```

Where `labels` is initialized to `ref empty`. That is, the callback attempts to make note of all the label that program execution 'passes through'.

In Deliverable D4.4, we will prove that this labelled trace is preserved by the compilation process.

The second stage consists of iterating over the same program and replacing all pseudojumps (both conditional and unconditional) with an 8051 jump to the requisite computed offset. One subtletly persists, however.

The 8051 has three different types of unconditional jump, depending on the length of the jump to be used: `AJMP`, `JMP` and `LJMP`. The instructions `AJMP` and `JMP` are short jumps, whereas `LJMP` is a long jump, capable of reaching anywhere in the program. At the moment, the second pass of the expansion stage replaces all unconditional pseudojumps with a `LJMP` for simplicity. We do, however, plan to improve this process for efficiency reasons, expanding to shorter jumps where feasible.

3.3 Anatomy of the emulator

We provide a high-level overview of the operation of the emulator. Two modes of operation exist: execution of an unlabelled program – like one obtained disassemblying an already assembled program – and execution of a labelled program.

Unlabelled execution proceeds as follows (see Fig. 3). Program code is loaded onto the 8051 in a standard format, the Intel Hex (IHX) format. All compilers/assemblers producing machine code for the 8051, including the SDCC compiler which we use for debugging purposes,

```
let hex = IntelHex.intel_hex_of_file filename in
let mem = IntelHex.process_intel_hex hex in
let status = ASMInterpret.load_mem mem ASMInterpret.initialize in
  ASMInterpret.execute callback status
```

Figure 3: Pseudocode of unlabelled program execution

```
let mem, cost_map = ASMInterpret.assembly labelled_program in
let status = ASMInterpret.load_mem mem ASMInterpret.initialize in
  ASMInterpret.execute callback status
```

Figure 4: Pseudocode of labelled program execution

produce compiled programs in IHX format as standard. Accordingly, our O'Caml emulator can parse IHX files using `intel_hex_of_file` and populate the emulator's code memory with their contents using `process_intel_hex`. Code memory is loaded into the initial processor status using `load_mem`.

Once code memory is loaded into the status, the emulator calls `execute`, which performs the standard fetch-decode-execute cycle indefinitely. The callback function passed to `execute` is called at the beginning of each cycle and takes in input the processor status. It can be used for debugging purposes, for instance to compute execution traces.

The callback function can even stop execution by raising the `Halt` exception. This is the only way to stop the emulator, since the 8051 processors have no instruction to stop execution and program "termination" is usually compiled to a tight diverging loop.

Labelled execution is similar (see Fig. 4). However, instead of a program being loaded from an Intel Hex file, we process a labelled program with `assembly`, in order to obtain an unlabelled program, complete with a map from code addresses to cost annotations. The map can be used by the `callback` function, passed to execute, in order to implement the 'label collecting semantics' that the CerCo compiler will preserve.

The (simplified) types of the functions mentioned in the pseudocode of Figures 3 and 4 are as follows:

Title	Type
`intel_hex_of_file`	`string -> intel_hex_entry list`
`process_intel_hex`	`intel_hex_entry list -> byte map`
`load_mem`	`byte map -> status -> status`
`initialize`	`status`
`execute`	`(status -> unit) -> status -> status`
`assembly`	`assembly_program -> byte list * cost map`
`callback`	`status -> unit`

3.4 Validation

In validating the design and implementation of the O'Caml emulator we used two tactics:

1. Use of multiple manufacturer's data sheets (both the Siemens Semiconductor and Phillips Semiconductor specifications for the 8051, as well as online sources such as the Keil website). We

found typographic errors in manufacturer's data sheets which were resolved by consulting an alternative sheet.

2. Use of reference compilers and emulators. The operation of the emulator was manually tested by reference to MCU 8051 IDE, an emulator for the 8051 series processor. A number of small C programs were compiled in SDCC[2]. The resulting IHX files were disassembled by MCU 8051 IDE. (IHX files are a standard format for transferring compiled assembly code onto an 8051 series processor, produced by SDCC and all other compilers that target that 8051.) The status changes in both emulators were then compared.

   For further validation, the output of the compiled C programs from SDCC was compared with the output of the same programs in GCC, in order to pre-empt the introduction of bugs in the emulator inherited from a faulty C compiler.

As a further check, the design and operation of the emulator was compared with the textual description of online tutorials on 8051 programming, such as those found at `http://www.8052.com`.

4 The emulator in Matita

The O'Caml emulator served as a testbed and prototype for an emulator written in the internal language of the Matita proof assistant. We describe our work porting the emulator to Matita, especially where the design of the Matita emulator differs from that of the O'Caml version.

4.1 What we do not implement

Our O'Caml 8051 emulator provides functions for reading and parsing Intel IHX format files. We do not implement these functions in the Matita emulator, as Matita provides no means of input or output.

4.2 Auxilliary data structures and libraries

A small library of data structures was written, along with basic functions operating over them. Implemented data structures include: Booleans, option types, lists, Cartesian products, Natural numbers, fixed-length vectors, and sparse tries.

Our type of vectors, in particular, makes heavy use of dependent types. Probing vectors is 'type safe' for instance: we cannot index into a vector beyond the vector's length.

We represent bits as Boolean values. Nibbles, bytes, words, and so on, are represented as fixed length (bit)vectors of the requisite length.

4.3 The emulator state

We represent all processor memory in the Matita emulator as a sparse (bitvector)trie:

```
ninductive BitVectorTrie (A: Type[0]): Nat → Type[0] :=
  Leaf: A → BitVectorTrie A Z
| Node: ∀n: Nat. BitVectorTrie A n → BitVectorTrie A n → BitVectorTrie A (S n)
| Stub: ∀n: Nat. BitVectorTrie A n.
```

Nodes are addressed by a bitvector index, representing a path through the tree. At any point in the tree, a `Stub` may be inserted, representing a 'hole' in the tree. All functions operating on tries use dependent types to enforce the invariant that the height of the tree and the length of the bitvector representing a path through the tree are the same.

We probe a trie with the `lookup` function. This takes an additional argument representing the value to be returned should a stub, representing uninitialised data, be encountered during traversal.

Like the O'Caml emulator, we use a record to represent processor state:

[2]See the `GCC` directory for a selection of them.

```
nrecord Status: Type[0] :=
{
  code_memory: BitVectorTrie Byte sixteen;
  low_internal_ram: BitVectorTrie Byte seven;
  high_internal_ram: BitVectorTrie Byte seven;
  external_ram: BitVectorTrie Byte sixteen;

  program_counter: Word;

  special_function_registers_8051: Vector Byte nineteen;
  special_function_registers_8052: Vector Byte five;

  ...
}.
```

However, we 'squash' the `Status` record in the Matita emulator by grouping all 8051 SFRs (respectively, 8052 SFRs) into a single vector of bytes, as opposed to representing them as explicit fields in the record itself. We then provide functions that index into the respective vector to 'get' and 'set' the respective SFRs. This is due to record typechecking in Matita being slow for large records.

4.4 Dealing with partiality

The O'Caml 8051 emulator makes use of a number of partial functions. These functions either `assert false`[3] or do not perform a comprehensive pattern analysis over their inputs. There are a number of possible reasons for this:

1. **Incomplete pattern analyses** are used where we are confident that the particular pattern match in question should never occur, for instance if the calling function performs a test beforehand, or where the emulator should fail anyway if a particular unchecked pattern is used as input. An example of a function which exhibits the latter behaviour is `set_arg_16` from `ASMInterpret.ml`, which fails with a pattern match exception if called on an input representing an eight bit argument.
2. **Assert false** may be called if the emulator finds itself in an 'impossible situation', such as encountering an empty list where a list containing one element is expected. In this respect, we used `assert false` in a similar way to the previously described use of incomplete pattern analysis.
3. **Assert false** may be called is some feature of the physical 8051 processor is not implemented in the O'Caml emulator and an executing program is attempting to use it.
4. **Assert false** may be called when the real, physical processor's behaviour is undefined in a particular context. An example of this is loading a program which is too large for the available amount of code memory that the processor provides.

The four manifestations of partiality above can be split into two types: partiality that manifests itself due to O'Caml's type system not being strong enough to rule the cause out, and partiality that signals a 'real' crash in the processor due to the user attempting to use an unimplemented feature. Items 1 and 2 belong to the former class, Items 3 and 4 to the latter.

Clearly Items 1 and 2 above must be addressed in the Matita formalisation. Item 2 is solved through extensive use of dependent types. Indexing into lists and vectors, for instance, is always 'type safe', as we provide probing functions with strong dependent types.

Item 1 is perhaps the most problematic of the three problems, as we either have to provide an exhaustive case analysis, use pattern wildcards, or find a clever way of encoding the possible patterns

[3] O'Caml idiom: immediately halts execution of the running program.

that are expected as input in the type of a function. We employ a technique that implements the latter idea. This is discussed in Subsection 4.5.

To solve Item 3 above in the Matita formalisation of the emulator, we introduce an axiom `not_implemented` of type `False`. When the emulator attempts to use an unimplemented feature, we introduce a metavariable, corresponding to an open proof obligation. These obligations are closed by performing a case analysis over `not_implemented`.

In the rare case that Item 4 is encountered (only once in the implementation of the emulator, in the `assembly` function), we use the Maybe monad to signal failure or success.

4.5 Addressing modes: use of dependent types

We provide an inductive data type representing all possible addressing modes of 8051 assembly. This is the type that functions will pattern match against.

```
ninductive addressing_mode: Type[0] :=
  DIRECT: Byte → addressing_mode
| INDIRECT: Bit → addressing_mode
...
```

However, we also wish to express in the type of our functions the *impossibility* of pattern matching against certain constructors. In order to do this, we introduce an inductive type of addressing mode 'tags'. The constructors of `addressing_mode_tag` are in one-one correspondence with the constructors of `addressing_mode`:

```
ninductive addressing_mode_tag : Type[0] :=
  direct: addressing_mode_tag
| indirect: addressing_mode_tag
...
```

We then provide a function that checks whether an `addressing_mode` is 'morally' an `addressing_mode_tag`, as follows:

```
nlet rec is_a (d:addressing_mode_tag) (A:addressing_mode) on d :=
  match d with
   [ direct ⇒ match A with [ DIRECT _ ⇒ true | _ ⇒ false ]
   | indirect ⇒ match A with [ INDIRECT _ ⇒ true | _ ⇒ false ]
...
```

We also extend this check to vectors of `addressing_mode_tag`'s in the obvious manner:

```
nlet rec is_in (n: Nat) (l: Vector addressing_mode_tag n) (A:addressing_mode) on l :=
 match l return λm.λ_ :Vector addressing_mode_tag m.Bool with
  [ VEmpty ⇒ false
  | VCons m he (tl: Vector addressing_mode_tag m) ⇒
     is_a he A ∨ is_in ? tl A ].
```

Here `VEmpty` and `VCons` are the two constructors of the `Vector` data type, and ∨ is inclusive disjunction on Booleans.

```
nrecord subaddressing_mode (n: Nat) (l: Vector addressing_mode_tag (S n)) : Type[0] :=
{
  subaddressing_modeel :> addressing_mode;
  subaddressing_modein: bool_to_Prop (is_in ? l subaddressing_modeel)
}.
```

We can now provide an inductive type of preinstructions with precise typings:

```
ninductive preinstruction (A: Type[0]): Type[0] :=
   ADD: ⟦ acc_a ⟧ → ⟦ register; direct; indirect; data ⟧ → preinstruction A
 | ADDC: ⟦ acc_a ⟧ → ⟦ register; direct; indirect; data ⟧ → preinstruction A
...
```

Here $[\![-]\!]$ is syntax denoting a vector. We see that the constructor `ADD` expects two parameters, the first being the accumulator A (`acc_a`), and the second being one of a register, direct, indirect or data addressing mode.

The final, missing component is a pair of type coercions from `addressing_mode` to `subaddressing_mode` and from `subaddressing_mode` to `Type`[0], respectively. The previous machinery allows us to state in the type of a function what addressing modes that function expects. For instance, consider `set_arg_16`, which expects only a `DPTR`:

```
ndefinition set_arg_16: Status → Word → ⟦ dptr ⟧ → Status :=
  λs, v, a.
   match a return λx. bool_to_Prop (is_in ? ⟦ dptr ⟧ x) → ? with
     [ DPTR ⇒ λ_: True.
       let ⟨ bu, bl ⟩:= split ... eight eight v in
       let status := set_8051_sfr s SFR_DPH bu in
       let status := set_8051_sfr status SFR_DPL bl in
         status
     | _ ⇒ λ_: False.
       match K in False with
       [
       ]
     ] (subaddressing_modein ... a).
```

All other cases are discharged by the catch-all at the bottom of the match expression. Attempting to match against another addressing mode not indicated in the type (for example, `REGISTER`) will produce a type-error.

4.6 Validation

Two means of validating the Matita emulator exist.

The emulator is executable from within Matita (naturally, the speed of execution is only a fraction of the speed of the O'Caml emulator). In particular, we provide a function `execute_trace` which executes a fixed number of steps of an 8051 assembly program, returning a trace of the instructions executed, in the form of a list. This trace may then be compared with the trace produced by the O'Caml emulator when executing a program for validation purposes.

Alternatively, once the Matita emulator is ported to the newest version of Matita (see Subsection 4.7) an executable O'Caml emulator can be extracted from the Matita code, and execution traces of the extracted and prototype O'Caml emulators can be compared side-by-side.

4.7 Future work

The Matita emulator is written in the latest public Subversion repository version of Matita. However, this version is in an intermediate stage between the 'old' Matita, and a new, more streamlined version of the proof assistant. As a result, some key features of the system are currently missing in the repository version of Matita, most notably program code extraction from a Matita theory file.

The new, rewritten version of Matita reinstates the missing functionality. We plan, once the newer version is released, to port the Matita emulator to the most up-to-date version of the proof assistant. This will allow us to extract a verified O'Caml emulator from the Matita theory files.

Another possible future work is to implement separate compilation by modifying the `assembly` function output to include a simble table and by adding a linking function.

The assembly function will be changed to minimize (or at least reduce) the size of the assembled program by translating pseudo jumps to short jumps where possible.

Finally, while the O'Caml emulator already implements I/O, timers and interrupt handling, the Matita version does not. Interrupt handling will not be dealt with in CerCo and we are likely to handle I/O in a simplified way by means of external library functions. Nevertheless, for the sake of completeness and future uses, we plan to eventually complete also the Matita version.

5 Listing of O'Caml files and functions

5.1 Listing of O'Caml files

Title	Description
`ASM.mli`	Containts algebraic datatypes representing assembly code.
`ASMInterpret.ml`	Contains the main emulation function, and auxiliary datatypes and functions necessary for emulation.
`BitVectors.ml`	Contains an implementation of bitvectors, using polymorphic variants to emulate dependent types.
`IntelHex.ml`	Contains functions for parsing the Intel IHX file format.
`MatitaPretty.ml`	Functions for pretty printing an assembly abstract syntax tree in the O'Caml compiler into its equivalent form in the Matita compiler.
`Parser.ml`	Generic functional parser combinators used for parsing the Intel IHX file format.
`Physical.ml`	Functions implementing arithmetic (for instance, addition and subtraction with carry) on bitvectors.
`Pretty.ml`	Functions for pretty printing assembly abstract syntax trees in the O'Caml compiler into a string form.
`Test.ml`	Test harness for emulator. Reads in and parses an Intel IHX file, and executes the resulting program.
`ToMatita.ml`	Funtions for exporting an Intel IHX file to a form the Matita emulator can understand.
`Util.ml`	Miscellaneous utility functions that do not fit elsewhere.

5.2 Selected important functions

5.2.1 From `ASMInterpret.ml(i)`

Name	Description
`assembly`	Assembles an abstract syntax tree representing an 8051 assembly program into a list of bytes, its compiled form.
`initialize`	Initializes the emulator status.
`load`	Loads an assembled program into the emulator's code memory.
`fetch`	Fetches the next instruction, and automatically increments the program counter.
`execute`	Emulates the processor. Accepts as input a function that pretty prints the emulator status after every emulation loop.

5.2.2 From `IntelHex.ml(i)`

Name	Description
`intel_hex_of_file`	Reads in a file and parses it if in Intel IHX format, otherwise raises an exception.
`process_intel_hex`	Accepts a parsed Intel IHX file and populates a hashmap (of the same type as code memory) with the contents.

5.2.3 From `Physical.ml(i)`

Name	Description
`subb8_with_c`	Performs an eight bit subtraction on bitvectors. The function also returns the most important PSW flags for the 8051: carry, auxiliary carry and overflow.
`add8_with_c`	Performs an eight bit addition on bitvectors. The function also returns the most important PSW flags for the 8051: carry, auxiliary carry and overflow.
`dec`	Decrements an eight bit bitvector with underflow, if necessary.
`inc`	Increments an eight bit bitvector with overflow, if necessary.

6 Listing of Matita files and functions

6.1 Listing of Matita files

Title	Description
`Arithmetic.ma`	Contains functions implementing arithmetical operations on bitvectors.
`ASM.ma`	Contains inductive datatypes for representing abstract syntax trees of 8051 assembly language.
`Assembly.ma`	Contains functions related to the assembly of 8051 assembly programs into a list of bytes.
`BitVector.ma`	Contains functions specific to bitvectors.
`BitVectorTrie.ma`	Contains an implementation of a sparse bitvector trie, which we use for implementing memory in the processor.
`Bool.ma`	Implementation of Booleans, and related functions.
`Cartesian.ma`	Implementation of Cartesian products, and related functions.
`Char.ma`	Hypothesises a type of characters.
`Connectives.ma`	Implementation of logical connectives.
`DoTest.ma`	Contains experiments and debugging code for testing the emulator.
`Either.ma`	Implementation of disjoint union types.
`Exponential.ma`	Functions implementating the Natural exponential, and related lemmas.
`Fetch.ma`	Contains functions relating to the 'fetch' function of the emulator, and related functions.
`Interpret.ma`	Contains the main emulator function, as well as ancillary definitions and functions.
`List.ma`	An implementation of polymorphic lists, and related functions.
`Maybe.ma`	Implementation of the 'maybe' type.
`Nat.ma`	Implementation of Natural numbers, and related functions and lemmas.
`Status.ma`	Contains the definition of the 'status' record, and related definitions.
`String.ma`	Contains a type for representing strings.
`Test.ma`	Contains definitions useful for debugging and testing the emulator.
`Universes.ma`	Infrastructure file related to Matita's universe hierarchy.
`Util.ma`	Contains miscellaneous utility functions that do not fit anywhere else.
`Vector.ma`	Contains an implementation of polymorphic vectors, and related definitions.

6.2 Selected important functions

6.2.1 From `Arithmetic.ma`

Title	Description
`add_n_with_carry`	Performs an n bit addition on bitvectors. The function also returns the most important PSW flags for the 8051: carry, auxiliary carry and overflow.
`sub_8_with_carry`	Performs an eight bit subtraction on bitvectors. The function also returns the most important PSW flags for the 8051: carry, auxiliary carry and overflow.
`half_add`	Performs a standard half addition on bitvectors, returning the result and carry bit.
`full_add`	Performs a standard full addition on bitvectors and a carry bit, returning the result and a carry bit.

6.2.2 From `Assembly.ma`

Title	Description
`assemble1`	Assembles a single 8051 assembly instruction into its memory representation.
`assemble`	Assembles an 8051 assembly program into its memory representation.
`assemble_unlabelled_program`	Assembles a list of (unlabelled) 8051 assembly instructions into its memory representation.

6.2.3 From `BitVectorTrie.ma`

Title	Description
`lookup`	Returns the data stored at the end of a particular path (a bitvector) from the trie. If no data exists, returns a default value.
`insert`	Inserts data into a tree at the end of the path (a bitvector) indicated. Automatically expands the tree (by filling in stubs) if necessary.

6.2.4 From `DoTest.ma`

Title	Description
`execute_trace`	Executes an assembly program for a fixed number of steps, recording in a trace which instructions were executed.

6.2.5 From `Fetch.ma`

Title	Description
`fetch`	Decodes and returns the instruction currently pointed to by the program counter and automatically increments the program counter the required amount to point to the next instruction.

6.2.6 From `Interpret.ma`

Title	Description
`execute_1`	Executes a single step of an 8051 assembly program.
`execute`	Executes a fixed number of steps of an 8051 assembly program.

6.2.7 From `Status.ma`

Title	Description
`load`	Loads an assembled 8051 assembly program into code memory.