HLP Project 2017

Contents

- Project work will be done in teams of 4 (3 and 5 allowed exceptionally).
- Project teams are self-selecting, and must be finalised by the end of week 4 of Term.
- Projects can be the default project (see below) or (exceptionally) **self-proposed**. All proposals must use F# and FABLE. Anyone with ambition to do their own project work should discuss it with me ASAP (before or after they form a team).
- Teams must have a definite project plan with individual work allocated and interfaces specified by the end of week 6 of Term.

Projects will be assessed and given feedback via: *Group Feedback Interview in week 7* Group demos on the third last day of Term (15 minutes per group) *Individual interviews over the last two days of Term (30 minutes per person)* Assessment of submitted code

Recommended project: A user-friendly ARM7TDMI assembler and simulator

Introduction

EEE students will have used Salman Arif's excellent VisUAL last year. That is a very well thought-out educational tool to faciliate learning ARM assembler, but it nevertheless has a few limitations, for example:

- Strange GUI when trying to edit a file looking at simulation results
- Incomplete instruction set
- Incomplete register model (does not have shadow registers)
- Difficult to support cross-platform because implemented in Java
- Difficult to debug errors in simulation
- Long execution times cause memory to fill up
- Simulation does not track uninitialised values in registers and provide runtime warnings when these are used.
- Headless mode (allowing simulation under programmatic control) is difficult to use and slow
- Memory allocation is inflexible
- No intellisense error messages
- Syntax and semantic error reporting is innacurate
- Syntax is not optimal

Reimplementing this in new technology will solve many of these issues. I propose that each team should aim to write something that will beat VisUAL (in some areas) in assembling and simulating ARM7TDMI instruction set. I'm happy for each team to come up with their own ideas about the precise *objectives* of their project and its *evaluation* - how will they tell whether it has been successful. I will vet these proposals in an early interview at the end of which each team will have a clear and feasible slant on this overall deliverable.

The project work can be assessed and tested as pure F# code without an HTML GUI, reducing risks, however as explained below in implementation notes

Technology and Project organisation

The base technology will be: FABLE, codemirror, HTML (for GUI) and F# for all code. This is sufficient to complete the project. You are free to add any other technology you like (e.g. javascript widgets, packaging). You will not necessarily get marks for adding technology.

To reduce risk all projects must be testable in command-line versions compiled fron F#. This means that problems with web technology will not prevent a successful result.

All teams must comply with the following:

- You are expected to use types and functions carefully in such a way as to make correct and complete implementation of at least the VisUAL subset of ARM instructions as easy as possible: there is much choice in how to do this.
- You are free to have whatever UI you can justify as good for your agreed aims. You can implement the UI as you like but must interface it to F# compiled to Javascript by FABLE.

- You must subdivide your code into independent modules which can be developed and tested individually, and divide up workload.
- You will be given credit for writing code components in a reusable form.

Teams will get *official* interviews at which specification and design are signed off, as well as *unofficial* feedback and support.

Self-proposed projects

Groups are free to propose their own project doing anything, as long as it uses the technology above and is complies with the organisational requirements above. Ideas can be proposed informally with feedback before final whole-group sign-off.

ARM Assembler and Emulator Implementation Notes

Most of the code can be implemented and tested as a set of two testable top-level ARM functions:

- **`readAsm`** : string -> MachineState
- **`executeInstruction`** : MachineState -> Instruction -> MachineState

`readAsm` is an assembler implemented as a simple tokeniser and parser which generates instruction data structures and corresponding memory addresses from the lines of an assembly code file which is presented as a string. **`executeInstruction`** is the emulator which understands how every instruction affects the state of the CPU and memory (MachineState).

Assembler

See Visual for a working example[1] and documentation.

The assembler must transform an array of strings representing the lines of the text assembly file - the assembler model - into a data structure that represents instruction and data items in a machine memory - the machine state.

```
1:  let readAsm: string array -> MachineState
```

In MachineState instruction items are represented symbolically by an appropriate data-structure, so full assembly to machine instructions is never required. Data words are represented as 32 bit data. Memory is a sparse set of memory locations each indexed by a 32 bit unsigned number and containing a single data or instruction item. Following VisUAL, all items occupy 32 bits bytes of memory. Initialised data bytes defined in the assembler file are therefore padded to word boundaries (a number of bytes divisible by 4) in the data model.

During simulation access to code will be a run-time error so self-modifying or reading code is impossible. That means that correct emulation need not generate exact machine instruction codes from the data structure representing each instruction.

Assembler pseudo-instructions (see VisUAL for a minimal list, or the ARM documentation for a more complete list) define initialised data and determine where in memory instructions and data are placed. Symbols written on a line before assembler instructions or pseudo-instructions are *labels* which represent constants equal to the address of the labelled instruction or data. The **`EQU`** pseudo-instruction requires a label, and the label is set equal to the value of the arithmetic expression in the instruction. This allows labels to be defined as arbitrary expressions of other labels. Both forward and backward references to labels are allowed in **`EQU`** but circular references are (obviously) an error.

VisUAL does not have a sophisticated model of memory allocation. Your memory allocation strategy and directives need not be the same as VisUAL's.

Emulator

The most complex function, by far, is **`executeInstruction`** , because there are many instructions and they must all be exactly implemented. Testing this function is challenging.

The types Instruction and MachineState are an important part of the design and define the ARM instruction set and programmer's model which is simulated including registers, status bits, and memory.

`ExecuteInstruction` can be tested (for instructions currently impelmented by VisUAL) by comparing it with the operation of VisUAL on the same data. The VisUAL web site specifies how to run VisUAL in headless mode using programmatic inputs and outputs, and a basic interface in F# to VisUAL is provided.

Graphical User Interface and Editor

The GUI and editor is implemented as HTML with a Javascipt program transpiled from F# using FABLE. A Javascript widget **`codemirror`** which provides a full-featured customisable programmers editor is used. The minimum code for the GUI is therefore very simple indeed. However for a pleasant user interface significantly more work will be needed. This may be, for example, syntax highlighting as allowed by codemirror and implemented in F#, and code formatting. This element of the project, and how much of the project work it constitutes, is very open-ended.

Assessment Details

Project assessment consists of:

Component	marks
Group demos	20
Individual interviews	20
Individual code	10
Challenge bonus	n/a

Group Demos and Presentations

- Form: 20 minute slot. 15 min presentation (with slides on laptop if you like) + demo (on laptop). 5 min questions.

Functionality UI Planning & team management Innovation

Individual interviews

Technical Understanding Testing Modularity and interfacing Contribution
Marks awarded here may be modified after code is assessed.

Individual code

Functions Readability Robustness of Testing
The code is also used to adjust the marks given for the Individual interview components