COMS12600 Emulator Assignment

v2013.3, Simon Hollis, simon@cs.bris.ac.uk

Introduction

The aim of this assignment is to produce a working emulator for a subset of the ARM ThumbV1 ISA. By doing this, you will gain understanding about the interactions between low-level software (i.e. assembly intructions) and how hardware devices support and execute them to produce useful results.

In order to produce a working emulator, you must first understand both the high-level functionalities required (e.g. control-flow or an arithmetic operation), as well as how they map to instructions in the Thumb ISA. Finally, you will work on implementing these instructions.

This document outlines both the requirements for your emulator, as well as suggesting a general approach to help you structure it. Meeting the requirements is compulsory but, as with any computer program, there are a range of approaches that can work well. The language we will use to implement the emulator is verilog, since it directly relates to a hardware implementation and builds on your experience from COMS12700.

Overview

The emulator has some high-level functionalities that must be completed for a successful submission:

1. To read in an input file in verilog ***`readmem`*** format.
2. Storage of the input stream of instructions in an internal memory.
3. Execution of the supported instruction set from internal memory.
4. Screen printing functionality to produce useful output from the system.
5. The creation and correct execution of some small test programs.

The high-level view and approach is discussed in the introductory lecture accompanying this assignment. Each of these functionalities will be discussed in more detail later in this document.

Emulator Skeleton

Here is the skeleton of the emulator. It can be encompassed in a single verilog module, which is split into four parts:

1. The instance declarations i.e. registers, wires etc.
2. The "initial" block, which sets up the simulation.
3. The "always" block, which implements the step-by-step execution, corresponding to a cycle of execution.
4. The remainder, which may include instantiations, assignments, tasks, procedures etc.

```
module emu() ;

  reg [ 31 : 0 ] r [ 0 :  15 ] ; // register file.
  reg [ 31 : 0 ] memory [ 0 : 1023 ] ; // memory.
  // other state and variable declarations


 // Fill in the instruction implementations here.
 // You can use 'tasks' and 'procedures here to
 // contain your code and allow re-use.
 // e.g.
 task movi ;
    input [2:0] rd ;
    input [7:0] imm8 ;

    begin
       // do the move operation
    end
 endtask


  initial begin
    // initialise emulator, e.g., memory content
  end


  always #1 clock = !clock ; // simulate the clock


  always @ ( posedge clock ) begin
    // perform a fetch-decode-execute cycle
  end


endmodule
```

All that is required in this assignment is to fill in the missing details! As a guideline, if you keep your code modular by the use of tasks and procedures, most instructions will only need to run to a few lines of verilog code and fetch-decode-execute will be equally compact.

Section 1: Assignment details

The following information gives specifics on how to attempt the requirements for the tasks given in the overview section.

Sub-tasks 1&2: Creating and reading in an input file to your emulator and use of a internal memory

REQUIREMENT: Your emulator must be capable of reading in an aasm-generated file into its internal storage.

There is helpful documentation on reading in hex files in Appendix D of this document.

Sub-task 3: Correct emulation of the Thumb subset execution

Emulator internal representations

Your emulator needs to correctly *functionally* model the workings of an ARM processor executing the reduced Thumb ISA. The requirement for functional correctness means that you need not include the full complexity of the ARM architecture, however your results need to be identical to those executing on real hardware.

To achieve this, you will need to model the following internal state of the processor as a minimum:

- An internal memory (assume a fixed size of 1024 words);
- A register file, including program counter;
- Status information (xPSR register)

All these should have the same functionality as found in the real architecture, documented in the ARM ARM.

Perhaps the thing that can be of most concern from the implementation point of view is the ALU, as it is the heart of the architecture. I do not expect that you emulate the ALU at a digital-logic level. Therefore you are free to implement the ALU and its operations using standard verilog maths funcitions. Just bear in mind the definition of each instruction as per the instruction set.

In the ARM, all numbers are stored in 2s complement and therefore your emulated ALU should be aware of this. However, not all instructions interpret their immediates as signed. In the ISA document, I have noted for each instruction whether or not its immediate is treated as a signed or unsigned operation. Please take care to check the required behaviour when implementing each instruction.

REQUIREMENT: implement an emulator which includes the above internal state

representations.

The ARM Thumb Fetch-Execute Model

The next important aspect of the Thumb architecture to model correctly is the sequence of instructions as they move through the processor, their effects and the corresponding programming model.

To do this, we need to know some important things about the memory layout, control flow and fetch-execute cycle of the processor:

1. Code and data are stored in memory in 32-bit words;
2. Instructions are 16-bits long;
3. Data values are 32-bits long;
4. The program counter indexes in bytes;
5. The program counter runs ahead of the program by four bytes (i.e. two instructions).

Point 5. is very important, since it means that any instructions taking the program counter as an input must use the advanced value.

Here is an example:

If, at address 0x08, the following was encountered:

@ address 0x08: ADD pc, ***pc,*** #4

Then, the read value of ***pc*** is 0x12, *not* 0x08 (0x08 + 2x2 instructions).

This also means that, at any point of the emulation, the emulator will be executing the instruction at address *pc – 4*.

REQUIREMENT: Your emulator must implement the Thumb Fetch-decode-execute cycle to execute one instruction per simulated cycle.

In order to complete Fetch-decode-execute successfully, you will need to perform the following steps:

1. Implementation of each of the specified instructions from the ISA. Two important aspects here are dealing with signed arithmetic and being able to load and store to memory when that is specified.
2. Ability to fetch a 16 bit thumb instruction from the 32 bit memory.
3. Method for incrementing the program counter each cycle.

Sub-task 4: Emulator trace and debugging outputs

The emulator's output is to be entirely to the console. As the emulator runs, it prints an instruction-by-instruction trace, which details the address being decoded, the instruction type and the register values **after** instruction execution. Here is the required output format:

```
 Executing instruction @ 00000001a: '0110100000000100'
 Decoded instruction: ldri with rn=0, rt=4
 pc=00000020,  Z = 0, N = 0, C = 1, V = 0
 r0=00000026,  r1=00000029,  r2=xxxxxxxx,  r3=00c0df00
 r4=00c0df00,  r5=xxxxxxxx,  r6=xxxxxxxx,  r7=xxxxxxxx
 r8=xxxxxxxx,  r9=xxxxxxxx, r10=xxxxxxxx, r11=xxxxxxxx
r12=xxxxxxxx, r13=xxxxxxxx, r14=xxxxxxxx, r15=00000020
```

In the above example, several of the registers show as 'xxxxxxxx', which means that they have not yet been written to (are uninitialised). This is valid behaviour and models the unknown state of a real processor's registers when it is turned on.

SVC instructions

In addition to the normal arithmetic and memory instructions, the Thumb ISA includes an “SVC” instruction. SVC is commonly used to pass requests to and from an operating system (more about this in COMS35102!), but here we will subvert its use for emulator control and debugging purposes. Importantly, we can use it to print out additional trace information as the emulation progresses. Since this is very helpful for debugging, **it is important to get the SVCs working early on in your implementation.**

If you look at the encoding of the SVC instruction, you will see that it takes an 8-bit immediate value. We will use this to pass information to the emulator.

Therefore, when your emulator encounters and SVC instruction, it should treat it specially. It should inspect the value of the immediate and take different actions based on this value as follows:

SVC Immediate (Decimal)	**Action to take**
0—7	Print out a single register rX (r[0—7], with index equal to the immediate) on the console in the hexadecimal format “rX=01234567”
16	Print out all registers in format shown in the task 4 section.
100	Stop the emulation, printing a message saying that emulation has stopped due to SVC 100.
101	Dump the complete contents of memory at this point in time, in the format given below.

Memory dump format

The memory dump SVC should print the contents of memory before any execution happens. i.e., it should show the instructions and data that have been read in from the input file, as they are stored in memory. Use the following format:

```
00000000:  ABCD0123
00000004:  EF012345
```

In the above example, the first column is the memory address; the second is its contents. All values are hexadecimal and there are two spaces between the columns.

Sub-task 5: Test programs

In this sub-task we execute a selection of test programs, designed to check the correct functionality of the various aspects of your emulator.

The first few test programs are those provided by me for you to test the correct functionality of your system against known correct results. ***Code is available on the unit web-page.*** Each program is small and easy to understand, but will not check all possible functionalities. You are expected to extend the code to check more exhaustively.

When I mark your emulators, I will run these basic tests as well as some more complex code to check correctness.

The final test program: bubble sort is an assessed exercise with details below.

Test Program: ALU Test

This program tests simple ALU execution and returns a value printed on the screen.

Test Program: Control Flow Test

This program tests simple control flow (unconditional branches and sub-routine calls).

Test Program: Conditional Execution

This program tests the conditional execution capability of your code.

Test Program: Bubble Sort

To demonstrate the power of your final emulator implementation, we'll have it do some proper work for us. One of the simplest useful programs you can write in assembler is bubble sort.

As a final addition to the assignment, you need to write an implementation of bubble sort in your Thumb assembler sub-set, and show it successfully working, sorting numbers.

A skeleton for bubble sort, including the numbers to be sorted is provided on the website. All you have to do is fill in the missing section with the bubble sort routine and run it in your emulator.

REQUIREMENT: Complete the bubble sort routine, assemble and emulate it. Use the `SVC 101` option to print out the contents of memory before and after execution. This should show the numbers being transformed from unsorted to sorted.

Section2: Supported ISA and Instructions

Your emulator must support a sub-set of the ARM ThumbV1 instruction set (otherwise known as the V5T architecture). The authoritative document on the features and implementation of the full Thumb ISA is the "ARM v7m Architecture Reference Manual" (or "ARM ARM"), which is linked to on the course web-site.

However, since this document is some 1020 pages long, and we only wish a sub-set, I will outline the main requirements here. If anything is unclear or you require more information on how a particular instruction executes and what effect it has, I suggest you also check its related entry in the ARM ARM.

Mnemonics and instruction naming conventions

The ARM ARM details a wide variety of both instructions and formats for each instruction. This allows the support of multiple options for a given instruction. For this assignment, we will use a sub-set of the available operation as well as only some of the supported formats. The set of instructions you need to support and their encodings are give in the separate file "ISA.pdf". *Before you look at this file*, here is an explanation of the naming conventions

General ISA naming

(the operator ● means 'combined with' in a general sense. i.e. not specifically addition, multiplication etc.):

Instruction target	Our mnemonic	ARM ARM mnemonic on given page number
Register ← Register ● Immediate	...I	… (Immediate)
Register ← Register ● Register	...R	… (Register)
Register ← Stack pointer ● Immediate	...SPI	...(SP plus Immediate), specifically the encoding modifying a register in the range r0—r7
Program Counter ← Program Counter ● Immediate	...IPC	… (PC plus Immediate), specifically encoding modifying the Program Counter
Stack pointer ← Stack pointer ● Immediate	INCSP	… (SP plus Immediate), specifically encoding modifying the stack pointer

e.g. In our ISA, the ADD instruction can take any of the following formats:

- **ADDI:** An 8-bit immediate plus a combined source and destination register.
- **ADDR:** Two source registers and a destination register, in the range r0—r7
- **INCSP:** Once source immediate, with the fixed combined second source

and destination of the sp.
- **ADDSPI:** sp + immediate as source, register destination.
- **ADDPCI:** pc + immediate as source, with a register destination.

All are addition operations, but use different source and destination locations.

The relationship between assembler mnemonics and ISA formats

It is important to note that whilst in our example, the ADD instruction has multiple potential encodings in our ISA, at the higher level of assembly language, there exists only on 'ADD' instruction. When this instruction goes through the assembler, the most appropriate encoding is selected automatically. This means that to test the various encodings, you must give different parameters to your e.g. ADD instruction.

Here is an example of the relationship between assembler input and the emitted instruction:

Assembler instruction	**Emitted instruction in the ISA**
ADD r0, r0, #1	ADDI r0, r0, #1
ADD r0, r1, r2	ADDR r0, r1, r2
ADD r0, sp, #3	ADDSPI r0, sp, #3

and so on, for all the other potential instruction encodings.

The meaning of each instruction should be obvious from its mnemonic, but if not, consult the relevant page of the ARM ARM for more information on its operation.
Useful note: the ARM ARM entries show a pseudo-code implementation for each instruction and what operations are performed by it. Wherever you see a reference to the pseudo-function "`InITBlock()`", assume this always returns false (this detail is not being modelled in our simplified implementation, so our code is never in an IT Block).

Appendix A: Suggested implementation approach

There are myriad ways to implement any computer program, however in verilog there are several features you can make use of to ease implementation and reduce the chance of errors in your final code. In this section, I do not intend to introduce any new concepts, since all should have been covered sufficiently in COMS12700, but I highlight the utility and benefits of each.

If you are unfamiliar with any of the concepts below, please refer to your COMS12700 notes on verilog or use any of the course textbooks or an internet search for the functionality.

1. **Modular code:** In verilog, the use of *modules* can allow you to split up your code into smaller components, each tasked with a specific functionality. Like creating smaller files in other programming modules, this can make the system more understandable and easier to test.
2. **Tasks and procedures:** Since 2001, verilog has featured two ways to produce re-usable blocks of code: *tasks* and *procedures*. Like using methods in C, these can re-use common fragments, make it easier to test and reduce the amount of code writing you need to do. One approach that works well is to use a separate task to implement each instruction, leaving only the global operations at the higher level.
3. **Compile with the `-Wall` setting:** verliog compilers are notorious for silently converting datatype lengths and performing other unsafe operations if the user is careless enough to not properly specify type information. Compiling `iverilog` with the `-Wall` flag will ensure it prints out any potential problems before the cause errors.
4. **Systematic testing of the emulator:** as part of your assessment, the emulator will be put through its paces with a selection of input codes. To be assured that your emulator will produce the correct result, I recommend systematic testing of instructions and instruction groups, by producing your own input test assembly files, running them through `aasm` and checking the emulator's output. Special attention should be given to instructions that cause complex interactions, such as conditional and branch instructions.

Appendix B: Testing and Marking

This coursework will be assessed by inspecting the quality of your emulator solution. To do this, it is very important that you stick exactly to the various requirements as listed throughout this document. This will enable me to execute the tests required to assess your contribution.

To aid you in this, on the website, I provide some sample input test programs and their intended outputs. You can use these as a guide for determining if you have correctly implemented some of the core functionalities of the emulator.

However, it is not intended that the files are an exhaustive coverage of all the functional correctness of your emulator. For this, you should perform your own modular testing, based on outputs expected with relation to the ARM ARM's specification.

When marking your emulator, I will use some test codes that are not included on the website to determine if your emulator correctly implements some of the more complex interactions possible in an input program. However, no features other than those documented in this manual will be tested.

Since I may test aspects that you have not, I may find bugs that you have not, and **this may lower your mark from that you expect**. The best way to ensure that this does not happen is to exhaustively test your own code before submission.

Your code will be compiled and interpreted using Icarus Verilog, which is installed on the lab machines and can be downloaded and installed on your own machine for free. If you have not tested your code on this platform, you should assume that any part of it may fail. This platform will be the only one I will use for marking.

Marking scheme

The unit web page contains a mark sheet, which explains the various aspects of the emulator that will be assessed, and their weightings. You will need to submit a completed mark sheet along with your assignment.

Appendix C: Creating an input file for your emulator.

Since your emulator supports a real ISA, we can use pre-existing assembler to convert between a program written in assembly language and target instructions.

In this course, we will use the Komodo Project's ARM assembler from the University of Manchester to assemble our input files. The source code for the assembler and its manual can be found here:
http://www.cs.manchester.ac.uk/resources/software/komodo/assembler/

A pre-compiled version of the tools can be found on snowy at the following location: `/home/staff/simon/COMS12600/aasm/aasm`

The assembler can output code in a variety of formats. The following two are most useful:

- listing "aasm -l <listingfile> <inputfile>". This produces assembled instructions, line-by-line, with addresses next to their input assembler and preserving labels and comments. This is very useful in seeing what has been transformed to what and checking addesses and offsets etc.
- verilog 'readmem' format "aasm -v [size] <hexfile> <inputfile>". This is the format your assembler needs to take as an input file, so serves as the start of your emulator assignment.

Examples of input/output can be found in the test programs on the unit page.

Appendix D: Recap of verilog memories and the $readmemh command

In verilog, it is possible to declare a *memory*, which is similar to an array in an high level programming language. For example, the syntax:

```
reg [15:0] myMemory [0:63] ;
```

declares a memory named 'myMemory', that contains 64 entries, each 16 bits wide. The memory can then be accessed by standard sub-scription. i.e. the verilog line:

```
assign a = myMemory[42] ;
```

assigns a to myMemory element index 42.

It is possible to initialise a memory by reading in data from a hex file in a manner that automatically fills a memory with its contents. To do this, we use the **`$readmemh`** directive inside the verilog **`initial`** block.

Usage: the following code snippet reads in the contents of the hex-formatted file **"input.asm" [and no other file-name!]**, such as those produced by aasm with the '-v' flag into the memory 'myMemory':

```
initial begin
   $readmemh("input.asm", myMemory) ;
end
```

That's all there is to it!

Appendix E: The DCD assembler directive

You will see the instruction 'DCD' in some of the example assembly files provided. DCD IS NOT A REAL INSTRUCTION. Rather, it is simply a flag to the assembler that the following value is a 32-bit constant to be inserted into the next memory location, and not assembled into an opcode.

Since DCD declared constants are 32-bits and needs to be aligned in memory, **the instruction must lie on a 4-byte boundary**, otherwise it will cause problems when fetched. If your code does not have the correct number of instructions to result in this, you must pad the code with dummy instructions before using DCD.

Here is how it works:

Assembly input	**Memory address (hex)**	**Assembled code (hex)**
ADD r0, r0, r1	00000000	1840
SUB r0, r0, r1	00000002	1A40
DCD 0xf00baaa	00000004	0f00baaa
DCD 0x01234567	00000008	01234567

Well done for reading the end of the document before you started coding. Now that you have the information inside this document, you are prepared to do a fantastic emulator implementation. Good luck, and don't forget to use the laboratory sessions for interactive help!