ARM Checkpoint...

Tom Adlington, Luke Cheeseman, Bradley Pollard and George Steed

May 31, 2013

1 Group Organisation

To allow us to make progress as quickly as possible we divided the parts of the project amongst our members. George and Luke took control of the development of the emulator, and Bradley and Tom looked ahead to parts 2 and 3. This meant when we completed the emulator we already had a good foundation for the assembler, as well as part 3's assembly code written entirely. Furthermore, each member discussed their part with the rest of the team, to handle roadblocks efficiently and understand the code they didn't write.

2 Implementation Strategies

Emulator stuff: - CPU implemented as a struct with pointers to memory and register array, as well as gpio bits. - CPU pipeline simulated using boolean variable to indicate whether or not the pipeline is filled, and adjusting the program counter based on this state. - emulate.h used to store constant variable definitions, enums and function prototypes. - initial instruction execution function splits into different instruction types based on a mask and expectation test. (I made up that term) - In the case of data processing, an opcode enum is retrived by masking and shifting the instruction. This is then switched over to find and take action for the correct opcode. This may be changed into a function pointer array at a later date. - Conditions are checked in the initial execute function in a manner similar to data processing. - Note that as well as the opcodes and conditions given in the specification, we have decided to implement those remaining, hence our emulator supports all 16 opcodes and all 15 conditions fully. :) - We use a $\text{uint32}_t both to ensure a 4 byte word and to ensure that shifting does not set new bit values when shifting right.$

Assembler stuff: - We decided to implement a two pass assembler, using one pass to: - find labels; - remove blank lines or lines with no instruction; and - to do some minor preprocessing on each line to make the second pass easier. - So far, only the first pass has been implemented completely however we expect second pass component to be fully functional by the middle of next week. - Labels and code lines are implemented using structures. - Labels have a simple string and int structure storing the association of line to address in memory. This is made significantly easier by the removal of blank lines mentioned earlier. - Code lines are stored as a triple of pointers to the same allocation. - The first pointer points to the beginning of the string, this is required to ensure all memory is successfully freed at the end of use. - The second pointer is used to check the instruction mnemonic. This is converted to lower case prior for simplicity. - The final pointer simply points to the beginning of the arguments to the instruction. This is purely for convenience reasons but hopefully should make the second pass easier to read and understand. - All memory allocations

(labels, code lines, the input buffer line) contain a check to ensure that a memory overflow does not occur. If a buffer is about to exceed its size, it is first reallocated to a larger memory block, then allowed to proceed assuming the memory is allocated successfully.