EE4DSA Coursework 3

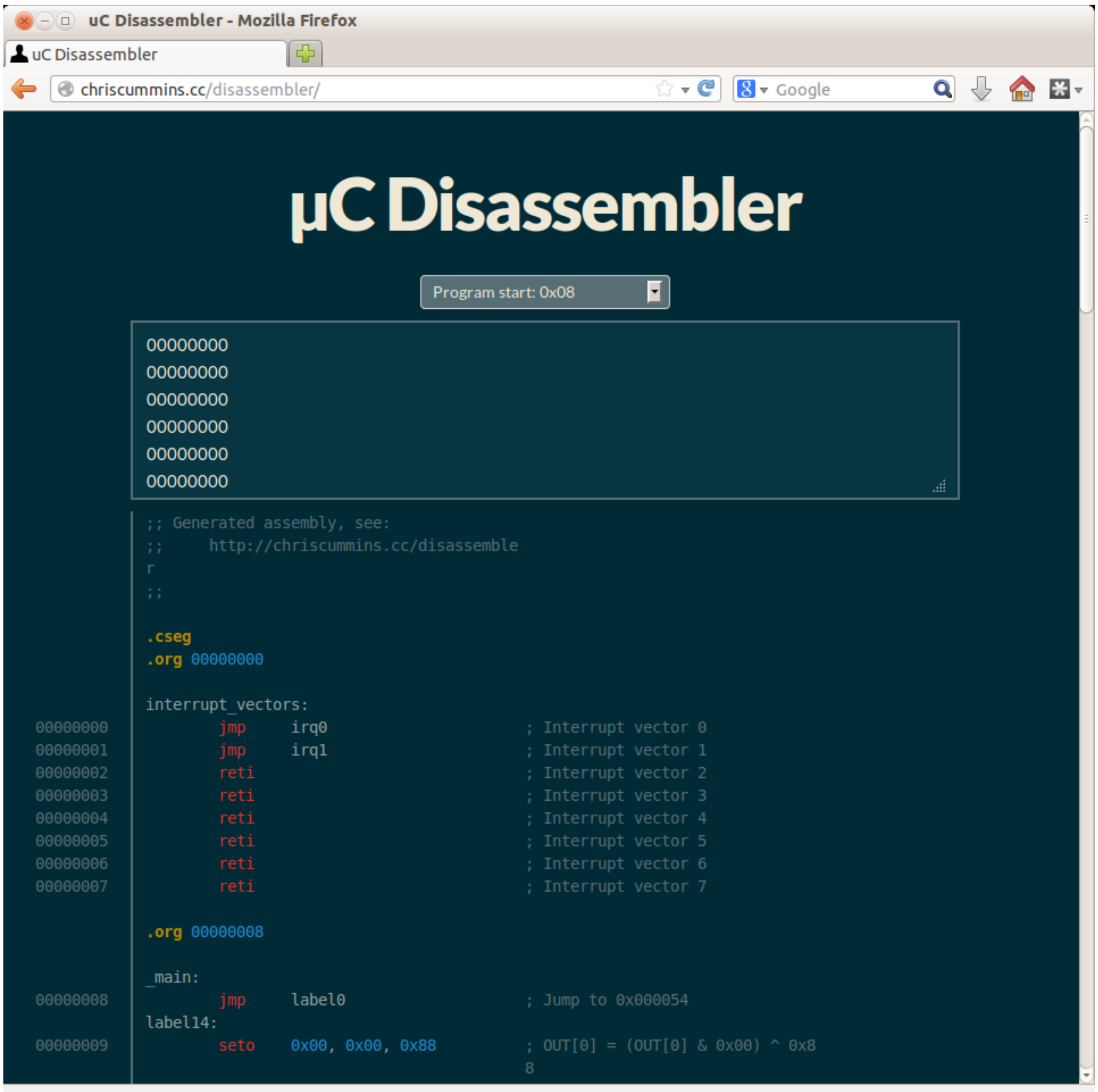
Chris Cummins

March 17, 2014

1 Disassembling the program ROM

The file `extra/ram.asm` contains a heavily annotated disassembled version of the safe unlocking program in a style inspired by the AVR Assembler Syntax. The program tests for the safe code 2013.

The assembly code was generated automatically using a disassembler developed for this purpose, available at `http://chriscummins.cc/disassembler`. Based off of the implementation of the second coursework disassembler, the functionality has been extended to support ALU and register file instructions.


uC Disassembler - Mozilla Firefox
uC Disassembler
chriscummins.cc/disassembler/
Google
µC Disassembler
Program start: 0x08

```
00000000
00000000
00000000
00000000
00000000
00000000
```

```
;; Generated assembly, see:
;;     http://chriscummins.cc/disassemble
r
;;

.cseg
.org 00000000

interrupt_vectors:
00000000        jmp     irq0                    ; Interrupt vector 0
00000001        jmp     irq1                    ; Interrupt vector 1
00000002        reti                            ; Interrupt vector 2
00000003        reti                            ; Interrupt vector 3
00000004        reti                            ; Interrupt vector 4
00000005        reti                            ; Interrupt vector 5
00000006        reti                            ; Interrupt vector 6
00000007        reti                            ; Interrupt vector 7

.org 00000008

_main:
00000008        jmp     label0                  ; Jump to 0x000054
label14:
00000009        seto    0x00, 0x00, 0x88        ; OUT[0] = (OUT[0] & 0x00) ^ 0x8
                                                8
```

A standalone version is available in `extra/disassembler`, and contains the following files:

- `ee4dsa-util.js` - Utility functions for working with JavaScript types and performing numerical base conversions.
- `ee4dsa-asm.js` - Contains objects used to render parts of assembly programs, such as instructions, comments, directives etc.
- `ee4dsa-disassembler.js` - Provides functions to decode RAM dumps into sets of assembly components, and to produce representations of programs. Implements the actual instruction set and machine code parser.
- `index.html` - Markup and JavaScript to implement the disassembler.

2 Implementing the EU & ALU

The file `execution_unit.vhd` contains my implementation of the execution unit. The first step upon importing the coursework 2 implementation was to refactor out the `next_pc_src` multiplexer, allowing the instruction set process to set the next program counter directly. While slightly obfuscating the code (the `next_pc` signal is now assigned in multiple places), this did provide a 10 MHz increase in the maximum operating frequency, providing the required performance needed to implement the register file and ALU instructions. Since the ALU has no internal states, the ALU required only a single level of immediate logic.

3 Synthesising the design

The directory `reports/` contains a copy of the synthesis reports. The design synthesis without pertinent warnings:

```
$ make synthesis
Synthesis running...
WARNING:HDLCompiler:634 - "/home/chris/src/vhdl-exercises/ee4dsa/cw3/top_level.vhd" \
  Line 112: Net <eu_intr[7]> does not have a driver.
WARNING:Xst:2935 - Signal 'eu_intr<7:2>', unconnected in block 'top_level', is tied \
  to its initial value (000000).
```

Analysing the log `reports/xst.log` shows that the maximum achievable frequency of the design is 105.029 MHz. The bottlenecks in this performance are the large fanout involved in the `next_pc_srrc` multiplexer, and the number of different assignments to the `next_[bc]_reg_addr` registers.

The fact that 66.2% of the longest data path is in the route and not logic shows that it is the delay in transmission speed which is preventing higher performance, and this could be increased by reducing the number of different places in which registers are assigned, leading to fewer input multiplexers and shorter paths between components.

```
=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 9.521ns (frequency: 105.029MHz)
  Total number of paths / destination ports: 458963 / 1903
-------------------------------------------------------------------------
Delay:               9.521ns (Levels of Logic = 5)
  Source:            ram_unit/Mram_data7 (RAM)
  Destination:       ram_unit/Mram_data1 (RAM)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: ram_unit/Mram_data7 to ram_unit/Mram_data1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     RAMB16BWER:CLKB->DOB0  137   1.850   2.222  ram_unit/Mram_data7 (eu_rom_data<24>)
     LUT6:I2->O            4   0.203   0.684  execution_unit/_n4252<29>1             \
       (execution_unit/_n4252)
     LUT3:I2->O           11   0.205   0.883  execution_unit/Mmux_next_pc_src<0>341   \
       (execution_unit/Mmux_next_pc_src<0>34)
     LUT6:I5->O           11   0.205   0.883  execution_unit/Mmux_next_pc_src<0>343_2 \
       (execution_unit/Mmux_next_pc_src<0>343_1)
     LUT3:I2->O            1   0.205   0.827  execution_unit/Mmux_next_pc33_SW0 (N290)
     LUT6:I2->O            8   0.203   0.802  execution_unit/Mmux_next_pc33           \
       (eu_rom_addr<3>)
     RAMB16BWER:ADDRB5         0.350          ram_unit/Mram_data1
    ----------------------------------------
    Total                      9.521ns (3.221ns logic, 6.300ns route)
                                       (33.8% logic, 66.2% route)
```

A video demonstration of the program can be found at `http://youtu.be/pJtk3-ZMU80`, showing how the program behaves when inputting a correct code, invalid code, and interrupting normal program behaviour by resetting the board.