RTL through OS

Goal: Demonstrate mastery of computer engineering by designing and implementing a single core RISC-V processor and developing a basic multitasking kernel to run on the processor.

Team

- Reid Long (relong)
- Teguh Hofstee (thofstee)

Why?

- Synthesizing 18-447 and 15-410 will help us develop an understanding of the interface between hardware and software
- We want to experience the thrills of debugging a system with neither a ground truth software implementation nor a ground truth hardware implementation
- We want to develop a unique skill set that spans both kernels and processors.

Outline

- Processor
    - We will design and implement a single core RISC-V processor.
    - Specification
        - RV32I+MAS
            - 32 bit architecture
            - Integer instructions
            - Integer multiplication/division
            - Atomic instructions
            - Supervisor (Privileged) instructions
        - https://riscv.org/specifications/
        - Version
            - User mode version 2.2
            - Privileged Architecture version 1.10
    - Input
        - PS2 Keyboard input
    - Output
        - VGA display output
            - https://github.com/corecode/fpga-vga/tree/master/rtl
    - Interrupts
        - Timer interrupt
        - Keyboard interrupt
        - User traps
        - Hardware faults
    - Memory System
        - Physical Memory

            - We will use the DRAM available on the FPGA SOC to provide the physical memory for the system.
            - We can either implement our own DRAM controller or more likely will instantiate a Xilinx AXI DRAM controller
        - Cache
            - We will use available BRAM to implement a unified L1 cache
            - We will use a random cache eviction policy
            - https://forums.xilinx.com/t5/Design-Entry/Instantiating-block-RAMs-from-within-Verilog/td-p/336759
        - Virtual Memory
            - We will implement a standard 2 level Page Table (RISC-V standard) with a hardware TLB
- Software
    - Kernel
        - We will develop a kernel to run on our RISC-V processor that implements the Pebbles UBI
            - http://www.cs.cmu.edu/~410/p2/kspec.pdf
    - Boot Loader
        - We will develop a boot loader to transition from Machine mode into Supervisor mode.
            - Ideas:
                - https://riscv.org/wp-content/uploads/2016/01/Tues1345-riscvcoreboot.pdf
                - Depending on the FPGA we select to deploy on, we could have the built in ARM Core function as the boot loader. It would read from the SD card and load the kernel image into memory. It would then trigger the FPGA softcore to begin execution
                - If it is easier, another option is to let the kernel make the transition from machine mode to supervisor mode. It might be easier to simply load a bunch of bytes from the kernel image into memory and then let the kernel figure out how to set up its initial state
                - https://github.com/ZipCPU/sdspi
                - http://www.lowrisc.org/docs/untether-v0.2/bootload/
                - http://www.lowrisc.org/docs/untether-v0.2/fpga-demo/#boot
                - https://www.sifive.com/blog/2017/10/09/all-aboard-part-6-booting-a-risc-v-linux-kernel/
                - http://www.lowrisc.org/docs/debug-v0.3/zedboard/
        - Requirements
            - Needs to be able to take a program image from some persistent state (maybe boot from the SD card?) and load it into memory then set PC to the entry point.

    - User Mode Libraries
        - We will need to find/compile/implement user mode libraries to run the test code on the RISC-V processor
    - ABI (System Calls)
        - https://www.ocf.berkeley.edu/~qmn/linux/riscv.html
- Platform
    - We will deploy our processor and kernel on an FPGA
    - Requirements
        - We will need 20k LUTs, 10k FFs, 12 BRAMs (36 KiB Blocks).
            - http://ieeexplore.ieee.org/document/8056766/
    - Desirable
        - VGA Output
        - PS2 Input
        - Additional BRAM (more is better since it will be dumped into a L1 Cache)
    - Peripherals
        - https://store.digilentinc.com/pmod-ps2-keyboard-mouse-connector/
        - https://store.digilentinc.com/pmod-vga-video-graphics-array/
- Tool-Chain
    - Compilation
        - https://github.com/lowRISC/riscv-llvm
        - https://github.com/CMU-18447/lab1b-test
        - https://github.com/riscv/riscv-tools
        - https://github.com/riscv/riscv-gnu-toolchain
        - https://stackoverflow.com/questions/32527794/how-can-i-compile-c-code-only-for-the-rv32i-base-integer-instruction-and-the-ext
        - https://riscv.org/software-tools/
- Evaluation
    - We have a target suite of programs that we want to run on our kernel successfully, which test a large amount of the system but not necessarily achieve 100% test coverage of hardware. For the demo day we will have a wide variety of games that can be played.
    - In addition, we will have specific unit tests for the hardware, as well as constrained random testing to increase the confidence in our design.

Milestones

1. Display (RTL)
    a. Description
        i. Boot RTL on the FPGA that will display a hardcoded string
    b. Feature(s)
        i. VGA display driver (RTL)
        ii. FPGA toolchain
    c. Validation
        i. Observe the correct string is displayed

    d. Estimated Time
        i. 5 Days
2. Echo (RTL)
    a. Description
        i. Boot RTL on the FPGA that will read keyboard input
    b. Feature(s)
        i. PS2 input driver (RTL)
    c. Validation
        i. Observe the display echoing keyboard input
    d. Estimated Time
        i. 5 Days
3. SD Source (RTL)
    a. Description
        i. Boot RTL on the FPGA that will read a string of bytes from the SD card and display those bytes on the console
    b. Feature(s)
        i. RTL to load bytes from the SD Card
    c. Validation
        i. Observe the correct bytes being displayed
    d. Estimated Time
        i. 1 Week
4. Boot Loader (RTL)
    a. Description
        i. Boot RTL on the FPGA that will read a file off of the SD card and place those bytes in memory
    b. Feature(s)
        i. RTL to store bytes in memory
        ii. DRAM memory controller
    c. Validation
        i. Use debug tools to inspect the contents of memory
    d. Estimated Time
        i. 1 Week
5. Counter (uKernel)
    a. Description
        i. Boot RTL on the FPGA that will launch a basic kernel
        ii. The kernel should execute in machine mode and simply spin incrementing a counter
    b. Feature(s)
        i. RTL to launch a kernel
        ii. RTL for a basic processor
        iii. Basic kernel that spins incrementing a register
    c. Validation

        i. Use debug tools to inspect the contents of the registers and see the value incrementing
    d. Estimated Time
        i. 2 Week
6. Fast Counter (uKernel)
    a. Description
        i. Boot the **Counter** kernel but run it on a basic processor that supports a cache to improve performance
    b. Feature(s)
        i. Processor cache
    c. Validation
        i. Use debug tools to inspect the contents of the registers and see the value increasing
    d. Estimated Time
        i. 1 Week
7. RandSuite (TestGen)
    a. Description
        i. Boot a series of assembly programs that randomly test the processor
        ii. Target coverage is ~100MLOC of assembly
    b. Feature(s)
        i. RTL for basic processor
    c. Validation
        i. All test vectors in the RandSuite should pass
            1. additest.S (447)
            2. addtest.S (447)
            3. arithtest.S (447)
            4. brtest0.S (447)
            5. brtest1.S (447)
            6. brtest2.S (447)
            7. depend.S (447)
            8. Fib.c (447)
            9. memtest0.S (447)
            10. memtest1.S (447)
            11. shifttest.S (447)
            12. fibiO.S (447)
            13. fibmO.S (447)
            14. fibr0.S (447)
            15. test-branch2.S
            16. test-branch3.S
            17. Test-branch5.s
            18. test-branch.S
            19. test-tricky-branch.S
            20. additest-nop.S

21. additest-safe.S
22. additest-stall.S
23. jumptest-basic.S
24. jumptest-2.S
25. bad-store.S
26. test-loop.S
27. test-loop2.S
28. test-loop3.S
29. test-loop4.S
30. small-arithtest.S
31. test-jump.S
32. predictTest.S
33. test-*.S (1-61)
34. test-B-*.S (1-21)

d. Estimated Time
   i. 2 Week

8. Hello World (uKernel)
   a. Description
      i. Boot a kernel that prints “hello world” to the VGA display
   b. Feature(s)
      i. Kernel driven VGA update
   c. Validation
      i. Observe the correct message being displayed
   d. Estimated Time
      i. 2 Days
9. Spin2 (uKernel)
   a. Description
      i. Boot a kernel that spins while disabling interrupts
   b. Feature(s)
      i. uArchitectural state for interrupts
      ii. Instruction support for interrupts
   c. Validation
      i. Use debug tools to inspect the state of the machine
   d. Estimated Time
      i. 2 Days
10. Console (uKernel)
    a. Description
       i. Boot a kernel that exercises basic console features
    b. Feature(s)
       i. Console driver for kernel
          1. set_term_color
          2. Set_cursor_pos
          3. Get_cursor_pos

4. print

c. Validation
    i. Observe the binary behaving correctly
d. Estimated Time
    i. 2 Day

11. Station (uKernel)
    a. Description
        i. Boot a kernel that exercises console features via timer interrupt
    b. Feature(s)
        i. Timer interrupt
    c. Validation
        i. Observe the binary behaving correctly
    d. Estimated Time
        i. 1 Day
12. Tick-Tock (uKernel)
    a. Description
        i. Boot a kernel that prints “tick” on every keyboard interrupt and “tock” on every timer tick
    b. Feature(s)
        i. Keyboard driver (software)
        ii. Timer driver (software)
    c. Validation
        i. Observe the correct messages being displayed
    d. Estimated Time
        i. 1 Day
13. Timing (uKernel)
    a. Description
        i. Boot a kernel that exercises timer interrupts and interrupt control instructions
    b. Feature(s)
        i. Interrupt control hardware (enable/disable interrupts)
    c. Validation
        i. Observe the test binary passing
    d. Estimated Time
        i. 4 Days
14. Game (uKernel)
    a. Description
        i. Boot a kernel that runs a 410-p1 level game.
        ii. The game should feature console display, keyboard control, and timer interrupts
        iii. Examples
            1. Minesweeper
                a. http://www.cs.cmu.edu/~410-f16/p1/proj1.html#game

2. SameGame
    a. http://www.cs.cmu.edu/~410-f17/p1/proj1.html#game
3. Gomoku
    a. http://www.cs.cmu.edu/~410-f09/p1/proj1.html#game

b. Features
    i. Game kernel
c. Validation
    i. Observe the game playing as expected
d. Estimated Time
    i. 4 Days

15. Teeny (uKernel + VM)
    a. Description
        i. Boot a kernel that runs a simple VM
    b. Feature(s)
        i. Virtual Memory Translation Hardware
    c. Validation
        i. Observe the test payload passing
    d. Estimated Time
        i. 2 Weeks
16. Vast (uKernel + VM)
    a. Description
        i. Boot a kernel that maps the entire virtual address space and attempts to read from every page
    b. Feature(s)
    c. Validation
        i. Observe the test payload passing
    d. Estimated Time
        i. 4 Days
17. Fondle (Kernel)
    a. Description
        i. Boot a kernel that enters user mode
    b. Feature(s)
        i. User mode
        ii. Virtual memory permissions
    c. Validation
        i. Observe the test payload passing
    d. Estimated Time
        i. 2 Weeks
18. Pebbles (Kernel)
    a. Description
        i. Boot a full kernel
    b. Feature(s)
        i. Full kernel

c. Validation
    i. Observe the following binaries passing
        1. Test_all
        2. ack
        3. actual_wait
        4. bistromath2
        5. cho
        6. cho2
        7. cho_variant
        8. chow
        9. coolness2
        10. peon2
        11. deschedule_hang2
        12. exec_basic
        13. exec_basic_helper
        14. exec_nonexist
        15. fib
        16. fork_exit_bomb
        17. fork_test1
        18. fork_wait
        19. fork_wait_bomb
        20. getpid_test1
        21. knife2
        22. loader_test1
        23. loader_test2
        24. make_crash
        25. mem_eat_test
        26. mem_permissions
        27. minclone_mem
        28. new_pages
        29. print_basic
        30. readline_basic2
        31. remove_pages_test1
        32. remove_pages_test2
        33. slaughter
        34. sleep_test1
        35. stack_test1
        36. swexn_basic_test
        37. swexn_cookie_monster
        38. swexn_dispatch
        39. swexn_regs
        40. swexn_stands_for_swextensible
        41. swexn_uninstall_test

42. wait_getpid
43. wild_test1
44. work
45. yield_desc_mkrun
46. startle
47. paradise_lost
48. mutex_test
49. broadcast_test
50. racer_long
51. racer_fast
52. excellent_test
53. agility_drill
54. cvar_test
55. beady_test2
56. cyclone
57. join_specific_test
58. mutex_destroy_test
59. mandelbrot_fast
60. mandelbrot_long
61. paraguay
62. rwlock_downgrade_read_test
63. switzerland
64. thr_exit_join
65. console_regression
66. float_test
67. malloc_test
68. multi_remove_pages_test
69. new_pages_fail_test
70. new_pages_test
71. read_file_test
72. reap_test
73. swexn_ureg_test
74. timer_test
75. yield_test
76. swexn_direction
77. tid_test
78. startle_test
79. fuzz

d. Estimated Time
    i. 4 Weeks

Stretch Goals

- Implement 5-stage pipeline architecture for the processor
- Implement super-scalar architecture for the processor
- Implement hashed page table virtual memory system
- Implement virtualization hardware and a hypervisor to run guest kernels
- Implement branch prediction for the processor
- Implement data forwarding for the processor

Unknown unknowns:

- Boot loader
    - We do not know how to deploy a RAM disk kernel image unto a FPGA.
- Keyboard driver
    - We do not know how and what kind of keyboard driver we will need to implement in RTL
- Console driver
    - We do not know how and what kind of console/display driver we will need to implement
- Tool Chain
    - We do not know how to get the make files setup for everything to build properly.