

Name: Ankur Aggarwal
Net ID: aa10336

Name: Pratham Mehta
Net ID: pm3483

A file CSA project B.zip has been uploaded along with this PDF file

Lab 1

Part 1

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code for a program named `asm_main.S`. The code includes directives like `.section .text`, `.align 2`, `.globl asm_main`, and `.equ offset, 0x00000000`. A specific instruction at line 9, `li a1, data`, is highlighted in yellow.
- Registers View:** The left sidebar shows register values:
 - r1 = 0x00000000
 - r2 = 0x00000000
 - fp = 0x0000000c
 - sp = 0x00000001
 - a0 = 0x00000001
 - a1 = 0x00103054
 - a2 = 0x00103058
 - a3 = 0x00000000
 - a4 = 0x00200000
 - a5 = 0x00000000
 - a6 = 0x00000000
 - a7 = 0x00000000
 - a8 = 0x00000000
 - s0 = 0x00000000
 - s1 = 0x00000000
 - s2 = 0x00000000
 - s3 = 0x00000000
 - s4 = 0x00000000
 - s5 = 0x00000000
 - s6 = 0x00000000
 - s7 = 0x00000000
 - s8 = 0x00000000
 - s9 = 0x00000000
 - s10 = 0x00000000
 - s11 = 0x00000000
 - s12 = 0x00000000
 - s13 = 0x00000000
 - s14 = 0x00000000
 - s15 = 0x00000000
 - s16 = 0x00000000
 - s17 = 0x00000000
 - s18 = 0x00000000
 - s19 = 0x00000000
 - s20 = 0x00000000
 - s21 = 0x00000000
 - s22 = 0x00000000
 - s23 = 0x00000000
 - s24 = 0x00000000
 - s25 = 0x00000000
 - s26 = 0x00000000
 - s27 = 0x00000000
 - s28 = 0x00000000
 - s29 = 0x00000000
 - s30 = 0x00000000
 - s31 = 0x00000000
- Registers View:** The bottom sidebar shows memory addresses and their values, such as `OverridId = 0x00000000`, `OverridId = 0x00000001`, etc.

The screenshot shows a debugger interface with two tabs: 'asm_main.S – Untitled (Workspace)' and 'main.c'. The assembly code tab displays the following assembly code:

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ offset, 0x80000000
6 .equ data, 0xDEADBEEF
7
8 asm_main:
9 li a1, data
10 li a2, offset
11 sw a1, (a2)
12 lw a3, (a2)
13 lh a4, (a2)
14 lb a5, (a2)
15 mv a2, a1
16 mv a1, x0
17 ret
```

The 'REGISTERS' section shows the following register values:

- r1 = 0x00000000
- r2 = 0x00000000
- fp = 0x201006c
- sp = 0x20000000
- a0 = 0x00000001
- a1 = 0xDEADBEEF
- a2 = 0x20102088
- a3 = 0x00000000
- a4 = 0x00000000
- a5 = 0x00000000

The 'WATCH' section contains the entry address: 'Enter address...'. The 'PROBLEMS' tab shows numerous errors related to reading registers from memory addresses like 0x00000000.

This screenshot is nearly identical to the one above, showing the same assembly code and register values. The main difference is in the 'WATCH' section, which now displays the value '0x00000000' for the entry address.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code for a file named `asm_main.S`. The code includes directives like `.section .text`, `.align 2`, and `.globl asm_main`. It defines symbols `offset` at `0x80000000` and `data` at `0xDEADBEEF`. The `asm_main` function contains instructions to move `data` to `a1`, then to `a2`, then to `a3`, then to `a4`, then to `a5`, then back to `a2`, and finally to `a1` before returning.
- Registers View:** A sidebar titled "REGISTERS" lists various registers with their current values:
 - `r1 = 0x00000000`
 - `r2 = 0x00000000`
 - `fp = 0x2001000c`
 - `s0 = 0x00000000`
 - `s1 = 0x00000000`
 - `s2 = 0x00000000`
 - `s3 = 0x00000000`
 - `s4 = 0x00000000`
 - `s5 = 0x00000000`
 - `s6 = 0x00000000`
 - `s7 = 0x00000000`
 - `s8 = 0x00000000`
 - `s9 = 0x00000000`
 - `s10 = 0x00000000`
 - `s11 = 0x00000000`
 - `s12 = 0x00000000`
 - `s13 = 0x00000000`
 - `s14 = 0x00000000`
 - `s15 = 0x00000000`
 - `s16 = 0x00000000`
 - `s17 = 0x00000000`
 - `s18 = 0x00000000`
 - `s19 = 0x00000000`
 - `s20 = 0x00000000`
 - `s21 = 0x00000000`
 - `s22 = 0x00000000`
 - `s23 = 0x00000000`
 - `s24 = 0x00000000`
 - `s25 = 0x00000000`
 - `s26 = 0x00000000`
 - `s27 = 0x00000000`
 - `s28 = 0x00000000`
 - `s29 = 0x00000000`
 - `s30 = 0x00000000`
 - `s31 = 0x00000000`
 - `fp = 0x2001000c`
 - `sp = 0x00000000`
 - `r0 = 0x00000000`
 - `r1 = 0x00000000`
 - `r2 = 0x00000000`
 - `r3 = 0x00000000`
 - `r4 = 0x00000000`
 - `r5 = 0x00000000`
 - `r6 = 0x00000000`
 - `r7 = 0x00000000`
 - `r8 = 0x00000000`
 - `r9 = 0x00000000`
 - `r10 = 0x00000000`
 - `r11 = 0x00000000`
 - `r12 = 0x00000000`
 - `r13 = 0x00000000`
 - `r14 = 0x00000000`
 - `r15 = 0x00000000`
 - `r16 = 0x00000000`
 - `r17 = 0x00000000`
 - `r18 = 0x00000000`
 - `r19 = 0x00000000`
 - `r20 = 0x00000000`
 - `r21 = 0x00000000`
 - `r22 = 0x00000000`
 - `r23 = 0x00000000`
 - `r24 = 0x00000000`
 - `r25 = 0x00000000`
 - `r26 = 0x00000000`
 - `r27 = 0x00000000`
 - `r28 = 0x00000000`
 - `r29 = 0x00000000`
 - `r30 = 0x00000000`
 - `r31 = 0x00000000`
- Memory View:** A sidebar titled "MEMORY" shows memory dump information for address `0x00000000`.
- Bottom Status Bar:** Shows the date and time as "Mon 5 Dec 5:21 PM".

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code for a file named `asm_main.S`. The code includes directives like `.section .text`, `.align 2`, and `.globl asm_main`. It defines symbols `offset` at `0x80000000` and `data` at `0xDEADBEEF`. The `asm_main` function contains instructions to move `data` to `a1`, then to `a2`, then to `a3`, then to `a4`, then to `a5`, then back to `a2`, and finally to `a1` before returning.
- Registers View:** A sidebar titled "REGISTERS" lists various registers with their current values:
 - `r1 = 0x00000000`
 - `r2 = 0x00000000`
 - `fp = 0x2001000c`
 - `s0 = 0x00000000`
 - `s1 = 0x00000000`
 - `s2 = 0x00000000`
 - `s3 = 0x00000000`
 - `s4 = 0x00000000`
 - `s5 = 0x00000000`
 - `s6 = 0x00000000`
 - `s7 = 0x00000000`
 - `s8 = 0x00000000`
 - `s9 = 0x00000000`
 - `s10 = 0x00000000`
 - `s11 = 0x00000000`
 - `s12 = 0x00000000`
 - `s13 = 0x00000000`
 - `s14 = 0x00000000`
 - `s15 = 0x00000000`
 - `s16 = 0x00000000`
 - `s17 = 0x00000000`
 - `s18 = 0x00000000`
 - `s19 = 0x00000000`
 - `s20 = 0x00000000`
 - `s21 = 0x00000000`
 - `s22 = 0x00000000`
 - `s23 = 0x00000000`
 - `s24 = 0x00000000`
 - `s25 = 0x00000000`
 - `s26 = 0x00000000`
 - `s27 = 0x00000000`
 - `s28 = 0x00000000`
 - `s29 = 0x00000000`
 - `s30 = 0x00000000`
 - `s31 = 0x00000000`
 - `fp = 0x2001000c`
 - `sp = 0x00000000`
 - `r0 = 0x00000000`
 - `r1 = 0x00000000`
 - `r2 = 0x00000000`
 - `r3 = 0x00000000`
 - `r4 = 0x00000000`
 - `r5 = 0x00000000`
 - `r6 = 0x00000000`
 - `r7 = 0x00000000`
 - `r8 = 0x00000000`
 - `r9 = 0x00000000`
 - `r10 = 0x00000000`
 - `r11 = 0x00000000`
 - `r12 = 0x00000000`
 - `r13 = 0x00000000`
 - `r14 = 0x00000000`
 - `r15 = 0x00000000`
 - `r16 = 0x00000000`
 - `r17 = 0x00000000`
 - `r18 = 0x00000000`
 - `r19 = 0x00000000`
 - `r20 = 0x00000000`
 - `r21 = 0x00000000`
 - `r22 = 0x00000000`
 - `r23 = 0x00000000`
 - `r24 = 0x00000000`
 - `r25 = 0x00000000`
 - `r26 = 0x00000000`
 - `r27 = 0x00000000`
 - `r28 = 0x00000000`
 - `r29 = 0x00000000`
 - `r30 = 0x00000000`
 - `r31 = 0x00000000`
- Memory View:** A sidebar titled "MEMORY" shows memory dump information for address `0x00000000`.
- Bottom Status Bar:** Shows the date and time as "Mon 5 Dec 5:21 PM".

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code for a program named `asm_main.S`. The code includes directives like `.section .text`, `.globl asm_main`, and `.equ offset, 0x80000000`. A specific instruction at line 14, `lb a5, (a2)`, is highlighted in yellow.
- Registers View:** The left sidebar shows registers `r1` through `r4`, `fp`, `s1` through `s4`, and `a1` through `a4`. Values for `r1` through `r4` are all set to `0x00000000`. `fp` is `0x2001006c`. `s1` through `s4` are `0x00000000`. `a1` through `a4` are `0xffffffff`.
- Memory View:** The bottom section shows memory dump and search functionality. It lists memory locations from `0x00000000` to `0xffffffff`, with values mostly being `0x00000000`.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The instruction at line 15, `mv a2, a1`, is now highlighted in yellow, indicating it is the current instruction being executed or selected.

The screenshot shows a debugger interface with two panes. The left pane displays assembly code for a function named `asm_main`. The right pane shows a dump of CPU registers. The assembly code includes instructions like `li a1, data`, `li a2, offset`, `sw a1, (a2)`, `lw a3, (a2)`, `lh a4, (a2)`, `lb a5, (a2)`, `mv a2, a1`, and `mv a1, x0`. The register dump lists various registers (r1 to r16, fp, gp) with their corresponding memory addresses and values.

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ offset, 0x80000000
6 .equ data, 0xDEADBEEF
7
8 asm_main:
9 li a1, data
10 li a2, offset
11 sw a1, (a2)
12 lw a3, (a2)
13 lh a4, (a2)
14 lb a5, (a2)
15 mv a2, a1
16 mv a1, x0
17 ret
```

This screenshot is identical to the one above, showing the same assembly code and register dump. It appears to be a duplicate or a second frame of the same session.

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ offset, 0x80000000
6 .equ data, 0xDEADBEEF
7
8 asm_main:
9 li a1, data
10 li a2, offset
11 sw a1, (a2)
12 lw a3, (a2)
13 lh a4, (a2)
14 lb a5, (a2)
15 mv a2, a1
16 mv a1, x0
17 ret
```

Part 2

The screenshot shows a debugger interface with two panes. The left pane displays assembly code for a program named 'asm_main.S'. The right pane shows the state of various registers. The assembly code includes instructions like .section .text, .globl asm_main, .equ data1, .equ data2, and .ret. The registers pane lists multiple registers with their current values, such as r0 = 0x00000000, r1 = 0x00000000, and so on up to r31. The interface has tabs for Code, Variables, Watch, Call Stack, Breakpoints, Registers, and Memory.

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ data1, 0x01010101
6 .equ data2, 0x10101010
7 asm_main:
8     li a1, 100
9     li a2, 150
10    add a3, a2, a1
11    sub a4, a2, a1
12    addi a1, a1, 100
13    li a1, data1
14    li a2, data2
15    and a3, a2, a1
16    or a4, a2, a1
17    xor a5, a2, a1
18    ret
```

Registers:

- r0 = 0x00000000
- r1 = 0x00000000
- r2 = 0x00000000
- r3 = 0x00000000
- r4 = 0x00000000
- r5 = 0x00000000
- r6 = 0x00000000
- r7 = 0x00000000
- r8 = 0x00000000
- r9 = 0x00000000
- r10 = 0x00000000
- r11 = 0x00000000
- r12 = 0x00000000
- r13 = 0x00000000
- r14 = 0x00000000
- r15 = 0x00000000
- r16 = 0x00000000
- r17 = 0x00000000
- r18 = 0x00000000
- r19 = 0x00000000
- r20 = 0x00000000
- r21 = 0x00000000
- r22 = 0x00000000
- r23 = 0x00000000
- r24 = 0x00000000
- r25 = 0x00000000
- r26 = 0x00000000
- r27 = 0x00000000
- r28 = 0x00000000
- r29 = 0x00000000
- r30 = 0x00000000
- r31 = 0x00000000

This screenshot is nearly identical to the one above, showing the same assembly code and register values. The assembly code is identical, and the registers pane shows the same set of values from r0 to r31. The interface tabs and overall layout are also the same.

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ data1, 0x01010101
6 .equ data2, 0x10101010
7 asm_main:
8     li a1, 100
9     li a2, 150
10    add a3, a2, a1
11    sub a4, a2, a1
12    addi a1, a1, 100
13    li a1, data1
14    li a2, data2
15    and a3, a2, a1
16    or a4, a2, a1
17    xor a5, a2, a1
18    ret
```

Registers:

- r0 = 0x00000000
- r1 = 0x00000000
- r2 = 0x00000000
- r3 = 0x00000000
- r4 = 0x00000000
- r5 = 0x00000000
- r6 = 0x00000000
- r7 = 0x00000000
- r8 = 0x00000000
- r9 = 0x00000000
- r10 = 0x00000000
- r11 = 0x00000000
- r12 = 0x00000000
- r13 = 0x00000000
- r14 = 0x00000000
- r15 = 0x00000000
- r16 = 0x00000000
- r17 = 0x00000000
- r18 = 0x00000000
- r19 = 0x00000000
- r20 = 0x00000000
- r21 = 0x00000000
- r22 = 0x00000000
- r23 = 0x00000000
- r24 = 0x00000000
- r25 = 0x00000000
- r26 = 0x00000000
- r27 = 0x00000000
- r28 = 0x00000000
- r29 = 0x00000000
- r30 = 0x00000000
- r31 = 0x00000000

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code for a program named `asm_main.S`. The code includes directives like `.section .text`, `.align 2`, and `.globl asm_main`. It defines data labels `data1` and `data2` with values `0x01010101` and `0x10101010` respectively. The `asm_main` function starts with `li a1, 100` and `li a2, 150`. The instruction at address `0x100000000000000010` is highlighted in yellow and is `add a3, a2, a1`.
- Registers View:** A sidebar on the left lists registers `a1` through `a5`, each with its current value set to `0x00000000`.
- Memory View:** A sidebar on the right shows memory starting at address `0x100000000000000000`, with all bytes displayed as `00`.
- Call Stack:** A small window in the top-left corner shows the call stack with entries: `asm_main.S:main + 0x0` and `entry.S:entry + 0x0`.
- Toolbar:** The bottom of the interface has a toolbar with various icons for file operations, search, and other debugger functions.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, memory dump, and call stack. The instruction at address `0x100000000000000010` is again highlighted in yellow as `add a3, a2, a1`.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main pane displays assembly code for a program named `asm_main.S`. The code includes directives like `.section .text`, `.align 2`, and `.globl asm_main`. It defines variables `data1` and `data2` with values `0x01010101` and `0x10101010` respectively, and contains instructions for loading these values into registers `a1` and `a2`, performing arithmetic operations (`add`, `sub`, `and`, `or`, `xor`), and returning.
- Registers View:** A sidebar on the left lists various registers with their current values in hexadecimal format. For example, `a1` is `0x00000000`, `a2` is `0x00000000`, and `data1` is `0x01010101`.
- Memory View:** Another sidebar shows memory dump information, listing addresses from `0x00000000` to `0x0000000F` with their corresponding byte values.
- Call Stack:** A small window at the bottom indicates the current stack frame.
- Output Tab:** The bottom tab bar shows the `OUTPUT` tab is active, displaying a series of messages related to reading registers and memory.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The primary difference is the highlighted line of assembly code, which has changed from `addi a1, a1, 100` to `li a1, data1`. This indicates a step-by-step execution or a specific instruction being examined.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main pane displays assembly code for a function named `asm_main`. The code includes instructions like `li a1, 100`, `li a2, 150`, `add a3, a2, a1`, `sub a4, a2, a1`, `addi a1, a1, 100`, `li a1, data1`, `li a2, data2`, `and a3, a2, a1`, `or a4, a2, a1`, `xor a5, a2, a1`, and `ret`.
- Registers View:** A sidebar on the left lists registers `a1` through `a5` with their current values: `a1 = 0x00000000`, `a2 = 0x00000000`, `a3 = 0x00000000`, `a4 = 0x00000000`, and `a5 = 0x00000000`.
- Memory View:** A sidebar on the right shows memory dump sections for `data1` and `data2`, both containing the value `0x01010101`.
- Output Tab:** The bottom tab bar indicates the `OUTPUT` tab is active.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The only difference is the highlighted instruction in the assembly view, which has shifted from `li a2, data2` to `and a3, a2, a1`.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main pane displays assembly code for a function named `asm_main`. The code includes instructions like `li a1, 100`, `li a2, 150`, `add a3, a2, a1`, `sub a4, a2, a1`, `addi a1, a1, 100`, `li a1, data1`, `li a2, data2`, `and a3, a2, a1`, `or a4, a2, a1`, `xor a5, a2, a1`, and `ret`.
- Registers View:** A sidebar on the left lists registers with their current values:
 - `a2 = 0x00101010`
 - `fp = 0x0000000c`
 - `s1 = 0x00000001`
 - `a3 = 0x00000001`
 - `a1 = 0x00101010`
 - `a7 = 0x00000000`
 - `a6 = 0x00000032`
 - `a5 = 0x00000000`
 - `a4 = 0x00000000`
 - `a3 = 0x00000000`
 - `a2 = 0x00000000`
 - `a1 = 0x00000000`
 - `a0 = 0x00000000`
- Memory View:** A sidebar on the right shows memory dump sections for `CpuCounter15h`, `CpuCounter14h`, `CpuCounter13h`, `CpuCounter12h`, `CpuCounter11h`, `CpuCounter10h`, `CpuCounter9h`, `CpuCounter8h`, `CpuCounter7h`, `CpuCounter6h`, `CpuCounter5h`, `CpuCounter4h`, `CpuCounter3h`, `CpuCounter2h`, `CpuCounter1h`, `CpuCounter0h`, `Mhortd`, `Flflag`, and `Ffcor`.
- Bottom Bar:** The interface includes a bottom bar with various icons and a status bar indicating the date and time.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, memory dump, and interface elements. The only difference is the highlighted instruction at line 17, which is `xor a5, a2, a1`.

The screenshot shows a debugger interface with the following assembly code:

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ data1, 0x01010101
6 .equ data2, 0x10101010
7 asm_main:
8     li a1, 100
9     li a2, 150
10    add a3, a2, a1
11    sub a4, a2, a1
12    addi a1, a1, 100
13    li a1, data1
14    li a2, data2
15    and a3, a2, a1
16    or a4, a2, a1
17    xor a5, a2, a1
18    ret
```

The code is annotated with several assembly instructions and their corresponding memory addresses. The assembly window includes sections for VARIABLES, WATCH, BREAKPOINTS, VENUS OPTIONS, PERIPHERALS, and REGISTERS. The REGISTERS section lists various registers with their initial values. The MEMORY section allows entering addresses to view memory contents. The bottom of the screen shows a Mac OS X dock with various application icons.

Part 3

The screenshot shows a debugger interface with the following assembly code:

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 # add program code here
6 asm_main:
7     j spot1
8
9 spot1:
10    j spot4
11
12 spot2:
13    j exit
14
15 spot3:
16    j spot2
17
18 spot4:
19    j spot3
20
```

This version of the assembly code includes five labels: spot1, spot2, spot3, spot4, and spot5. The code uses jumps to loop between these labels. The debugger interface is identical to the one in Part 2, with sections for VARIABLES, WATCH, BREAKPOINTS, VENUS OPTIONS, PERIPHERALS, and REGISTERS. The bottom of the screen shows a Mac OS X dock with various application icons.

The screenshot shows a debugger interface with two tabs: 'asm_main.S – Untitled (Workspace)' and 'main.c'. The assembly tab displays the following code:

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 # add program code here
6 asm_main:
7     j spot1
8
9 spot1:
10    j spot4
11
12 spot2:
13 j exit
14
15 spot3:
16 j spot2
17
18 spot4:
19 j spot3
20
```

The 'REGISTERS' section shows the following register values:

- R0 = 0x00000000
- R1 = 0x00000000
- R2 = 0x00000000
- R3 = 0x00000000
- R4 = 0x00000000
- R5 = 0x00000000
- R6 = 0x00000000
- R7 = 0x00000000
- R8 = 0x00000000
- R9 = 0x00000000
- R10 = 0x00000000
- R11 = 0x00000000
- R12 = 0x00000000
- R13 = 0x00000000
- R14 = 0x00000000
- R15 = 0x00000000
- R16 = 0x00000000
- R17 = 0x00000000
- R18 = 0x00000000
- R19 = 0x00000000
- R20 = 0x00000000
- R21 = 0x00000000
- R22 = 0x00000000
- R23 = 0x00000000
- R24 = 0x00000000
- R25 = 0x00000000
- R26 = 0x00000000
- R27 = 0x00000000
- R28 = 0x00000000
- R29 = 0x00000000
- R30 = 0x00000000
- R31 = 0x00000000

The 'MEMORY' section is empty, showing 'Enter address...'. The status bar at the bottom indicates 'Ln 19, Col 1 Status: 4 UTF-8 LF Assembly'.

This screenshot is nearly identical to the one above, showing the same assembly code and register values. The main difference is the highlighted line in the assembly code, which has changed from line 10 to line 19, indicating a step or selection change in the debugger.

The screenshot shows a macOS desktop environment with several open windows:

- Terminal:** The title bar says "Mon 5 Dec 6:52 PM". The terminal window displays the command "cd /Users/ankur/Dropbox/Mac (2)/Documents/CSA lab 1/src > ./asm_main.S" followed by the assembly code for "asm_main.S".
- Debugger:** A window titled "asm_main.S -- Untitled (WorkSpace)". It shows assembly code with labels like ".section .text", ".globl asm_main", and various ".j" instructions. A yellow highlight covers the area from ".j spot2" to ".j spot3".
- File Browser:** A window titled "asm_main.S" showing the file structure: "asm_main.S", "main.c", and "asm_main.S".
- System Dock:** Shows icons for Finder, Mail, Calendar, Reminders, Stocks, Wallet, Safari, and others.

The screenshot shows the Xcode IDE interface. The top menu bar includes 'Code', 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', 'Window', and 'Help'. The title bar indicates the workspace is 'asm_main.S - Untitled (Workspace)'. The left sidebar contains sections for 'VARIABLES' (Local, Global, Static), 'WATCH' (Call Stack, Breakpoints, Venus Options, Peripherals, Registers, Memory), and 'MEMORY' (Enter address...). The main editor area displays assembly code:

```
section .text
.align 2
.globl asm_main
;
# add program code here
asm_main:
    j spot1
    j spot1
    j spot1:
    j spot4
    j exit
    j spot2:
    j spot3:
    j spot2
    j spot4:
    j spot3
    j spot3
```

The line 'j exit' is highlighted with a yellow background. Below the editor, the 'PROBLEMS' and 'OUTPUT' tabs are visible, along with a terminal window and Jupyter support. The bottom status bar shows the date as 'Mon 5 Dec 4:52 PM' and battery level as '100% Ankur'. The Mac OS Dock at the bottom includes icons for Finder, Mail, Safari, and other applications.

Code File Edit Selection View Go Run Terminal Windows Help

asm_main.S – Untitled (Workspace)

```

13 j exit
14
15 spot3:
16 j spot2
17
18 spot4:
19 j spot3
20
21 exit:
22 ret

```

VARIABLES
WATCH
CALL STACK
BREAKPOINTS
PERIPHERALS
REGISTERS
MEMORY

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

DISASSEMBLY

This screenshot shows a debugger interface with an assembly code editor at the top. The code contains several jump instructions (j) and a ret instruction. Below the code are sections for VARIABLES, WATCH, CALL STACK, BREAKPOINTS, PERIPHERALS, REGISTERS, and MEMORY. The REGISTERS section lists various registers with their addresses and values. The bottom of the screen features a toolbar with various icons and a macOS-style dock below it.

Part 4

Code File Edit Selection View Go Run Terminal Windows Help

asm_main.S – Untitled (Workspace)

```

1 .section .text
2 .align 2
3 .globl asm_main
4
5 # add program code here
6 asm_main:
7 li a1, 0xdeadbeef
8 li a2, 0xdeadbeef
9 beq a1, a2, spot1
10 beqz a1, spot2
11 bne a1, a2, spot3
12
13 spot1:
14     ret
15 spot2:
16     ret
17 spot3:
18     ret

```

VARIABLES
WATCH
CALL STACK
BREAKPOINTS
PERIPHERALS
REGISTERS
MEMORY

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

DISASSEMBLY

This screenshot shows a debugger interface with an assembly code editor at the top. The code includes a section header (.text), labels for spots 1, 2, and 3, and three ret instructions. Below the code are sections for VARIABLES, WATCH, CALL STACK, BREAKPOINTS, PERIPHERALS, REGISTERS, and MEMORY. The REGISTERS section shows specific register values like a1=0xdeadbeef and a2=0xdeadbeef. The bottom of the screen features a toolbar with various icons and a macOS-style dock below it.

The screenshot shows a debugger interface with two panes. The left pane displays assembly code:asm_main.S
1 .section .text
2 .align 2
3 .globl asm_main
4
5 # add program code here
6 asm_main:
7 li a1, 0xdeadbeef
8 li a2, 0xdeadbeef
9 beq a1, a2, spot1
10 beqz a1, spot2
11 bne a1, a2, spot3
12
13 spot1:
14 ret
15 spot2:
16 ret
17 spot3:
18 retThe instruction at address 0x0000000000000009, which is the second `li` instruction, is highlighted in yellow. The right pane shows the register dump:Registers
R0 = 0x0000000000000000
R1 = 0x0000000000000000
R2 = 0x0000000000000000
R3 = 0x0000000000000000
R4 = 0x0000000000000000
R5 = 0x0000000000000000
R6 = 0x0000000000000000
R7 = 0x0000000000000000
R8 = 0x0000000000000000
R9 = 0x0000000000000000
R10 = 0x0000000000000000
R11 = 0x0000000000000000
R12 = 0x0000000000000000
R13 = 0x0000000000000000
R14 = 0x0000000000000000
R15 = 0x0000000000000000
R16 = 0x0000000000000000
R17 = 0x0000000000000000
R18 = 0x0000000000000000
R19 = 0x0000000000000000
R20 = 0x0000000000000000
R21 = 0x0000000000000000
R22 = 0x0000000000000000
R23 = 0x0000000000000000
R24 = 0x0000000000000000
R25 = 0x0000000000000000
R26 = 0x0000000000000000
R27 = 0x0000000000000000
R28 = 0x0000000000000000
R29 = 0x0000000000000000
R30 = 0x0000000000000000
R31 = 0x0000000000000000Registers R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, and R30 all have their values set to 0x0000000000000000.

This screenshot is identical to the one above, showing the same assembly code and register dump. The instruction at address 0x0000000000000009 is highlighted in yellow.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The code is displayed in assembly language. A specific instruction at address 0x0000000000400000 is highlighted in yellow: `beq a1, a2, spot1`.
- Registers:** A list of registers is shown with their current values:
 - a1 = 0xdeadbeef
 - a2 = 0xdeadbeef
 - spot1 = 0x0000000000400000
 - spot2 = 0x0000000000400000
 - spot3 = 0x0000000000400000
- Registers (continued):** Additional register values listed include:
 - s0 = 0x00000000
 - s1 = 0x00000000
 - t0 = 0x00000000
 - t1 = 0x00000000
 - t2 = 0x00000000
 - t3 = 0x00000000
 - t4 = 0x00000000
 - t5 = 0x00000000
 - t6 = 0x00000000
 - t7 = 0x00000000
 - t8 = 0x00000000
 - t9 = 0x00000000
 - t10 = 0x00000000
 - t11 = 0x00000000
 - t12 = 0x00000000
 - t13 = 0x00000000
 - t14 = 0x00000000
 - t15 = 0x00000000
 - t16 = 0x00000000
 - t17 = 0x00000000
 - t18 = 0x00000000
 - t19 = 0x00000000
 - t20 = 0x00000000
 - t21 = 0x00000000
 - t22 = 0x00000000
 - t23 = 0x00000000
 - t24 = 0x00000000
 - t25 = 0x00000000
 - t26 = 0x00000000
 - t27 = 0x00000000
 - t28 = 0x00000000
 - t29 = 0x00000000
 - t30 = 0x00000000
 - t31 = 0x00000000
- Memory:** The memory dump section is empty, with the message "Enter address...".
- Output:** The output window shows multiple "Reading register" messages for various registers, all with the value 0x00000000.
- Bottom Bar:** The macOS Dock bar is visible at the bottom, showing various application icons.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The main difference is the highlighted instruction at address 0x0000000000400000: `ret`. This indicates that the program has returned from a function call.

We now change the value of a1 to 0x0 as required

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code for a program named `asm_main`. The current instruction being executed is highlighted in yellow: `li a1, 0x0`.
- Registers View:** A sidebar titled "REGISTERS" lists various registers with their current values:
 - s0 = 0x00000001
 - s1 = 0x00000001
 - a1 = 0x20010354
 - a2 = 0x20010358
 - a3 = 0xffffffff
 - a4 = 0x00000000
 - a5 = 0x00000000
 - a6 = 0x0000001f
 - a7 = 0x00000000
 - s2 = 0x00000000
 - s3 = 0x00000000
 - s4 = 0x00000000
 - s5 = 0x00000000
 - s6 = 0x00000000
 - s7 = 0x00000000
 - s8 = 0x00000000
 - s9 = 0x00000000
 - s10 = 0x00000000
 - s11 = 0x00000000
 - s12 = 0x00000000
 - s13 = 0x00000000
 - s14 = 0x00000000
 - s15 = 0x00000000
 - s16 = 0x00000000
 - s17 = 0x00000000
 - s18 = 0x00000000
- Memory View:** A "MEMORY" section allows entering an address to view memory contents.
- Output View:** A "PROBLEMS" tab shows a list of reading register operations for various counters (Cpucounter1h to Cpucounter2h).
- System View:** A "DISASSEMBLY" tab is visible at the bottom.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The only difference is the date in the top right corner, which has changed from "Mon 5 Dec 5:02 PM" to "Tue 6 Dec 5:02 PM".

A screenshot of a debugger application, likely QEMU or similar, running on a Mac OS X desktop. The main window shows assembly code for a program named 'asm_main.S'. The assembly code includes instructions like .globl asm_main, li a1, 0x0, and beq a1, a2, spot1. A yellow highlight covers the section from beq to bne. The left sidebar contains tabs for VARIABLES, CALL STACK, BREAKPOINTS, VENUS OPTIONS, PERIPHERALS, and MEMORY. The bottom part of the screen shows a memory dump with registers x0-x7 and x8-x15 listed. The bottom navigation bar includes icons for file operations, terminal, and assembly.

A screenshot of a macOS desktop environment. At the top, there's a menu bar with 'Code', 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', 'Window', and 'Help'. The system tray shows battery level at 100%, signal strength, and the date 'Mon 5 Dec 5:03 PM'. Below the menu bar is a dock with various application icons: Finder, Mail, Calendar, Notes, Safari, Terminal, and others.

The main window is a code editor with tabs for 'asm_main.S' and 'entry.S'. The 'asm_main.S' tab is active, displaying assembly code:

```
asm_main.S -- Untitled (Workspace)
asm_main.S - CSA lab 1 - src
entry.S
main.c

3 .globl asm_main
4
5 # add program code here
6 asm_main:
7     li a1, 0x0
8     li a2, 0xdeadbeef
9     beq a1, a2, spot1
10    beqz a1, spot2
11    bne a1, a2, spot3
12
13 spot1:
14     ret
15 spot2:
16     ret
17 spot3:
18     ret
```

The line '9 beq a1, a2, spot1' is highlighted with a yellow background. On the left side of the editor, there are toolbars for 'RUN AND DEBUG', 'VARIABLES', 'WATCH', 'CALL STACK', 'BREAKPOINTS', 'VENVOPTIONS', 'PERIPHERALS', and 'REGISTERS'. The 'REGISTERS' section shows registers r0 through r15 with their values set to 0x00000000. The 'WATCH' section shows memory locations from 0x00000000 to 0x0000000f. The 'CALL STACK' section shows frames for 'main' and 'main.c'. The bottom of the screen has a 'PROBLEMS' tab, an 'OUTPUT' tab (which is currently selected), a 'DEBUG CONSOLE' tab, and a 'TERMINAL' tab. A status bar at the bottom right shows 'Filter (e.g. text, testcode)'.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code. Line 10, which contains the instruction `beqz a1, spot2`, is highlighted in yellow.
- Registers View:** A sidebar on the left lists various registers with their addresses and values. For example, R0 = 0x00000000, R1 = 0x00000000, and R2 = 0x00000000.
- Memory View:** A section labeled "MEMORY" allows entering an address to view memory contents.
- Call Stack:** A "CALL STACK" panel shows the current stack frame.
- Output View:** A "PROBLEMS" tab shows multiple "Reading register" errors for various registers.
- Toolbar:** At the bottom, there is a toolbar with icons for Run, Stop, Step, and Assembly.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The instruction at line 10 (`beqz a1, spot2`) is still highlighted in yellow.

Part 5

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code. The instruction at address 0x00000000 is highlighted in yellow: `jal change_value`. The assembly code includes:

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ value1, 0x80000000
6 .equ value2, 0x80000010
7
8 asm_main:
9     li a2, value1
10    jal change_value
11    li a2, value2
12    jal change_value
13    j asm_main
14
15 change_value:
16    lw a3, (a2)
17    xor a3, a3, a2
18    li a4, 0xFF
19    and a4, a4, a3
20    sw a4, (a2)
```
- Registers View:** A sidebar shows register values for various registers (a0-a7, s0-s7, t0-t7) in hex format.
- Memory View:** A sidebar allows entering memory addresses to view their contents.
- Call Stack:** Shows the current call stack with entries like `asm_main.S:asm_main()`.
- Toolbar:** Includes buttons for RUN AND DEBUG, PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and JUPITER.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory view. The instruction at address 0x00000000 is still highlighted in yellow: `jal change_value`. The assembly code is identical to the first screenshot.

The screenshot shows the QEMU debugger interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Window, Help.
- Run and Debug:** Shows the current assembly file: `asm_main.S`.
- VARIABLES:** Local, Global, Static.
- CALL STACK:** Shows the stack trace: `asm_main()`, `change_value()`, `asm_main()`.
- BREAKPOINTS:** None.
- PERIPHERALS:** None.
- Registers:** Shows registers R0-R15 and PC. All registers show values starting with 0x00000000 except for R15 which shows `asm_main`. The PC value is `0x00000000`.
- Memory:** Shows memory dump from `0x00000000` to `0x0000000F`. The dump shows the assembly code and its corresponding memory representation.
- Disassembly:** Shows the assembly code for `asm_main` and `change_value`.
- Registers Tab:** Shows the register dump.
- Output Tab:** Shows the assembly code and memory dump.
- Terminal Tab:** Shows the command line interface.
- Jupyter Tab:** Shows the Jupyter notebook interface.

The screenshot shows a macOS desktop environment with an IDE window open. The window title is "asm_main.S -- Untitled (Workspace)". The main pane displays assembly code:

```
SECTION .TEXT
.align 2
.globl asm_main
.value1, 0x80000000
.value2, 0x80000010

asm_main:
    li a2, value1
    jal change_value
    li a2, value2
    jal change_value
    j asm_main

change_value:
    lw a3, (a2)
    xor a3, a3, a2
    li a4, 0xFF
    and a4, a4, a3
    sw a4, (a2)
```

The instruction at line 16, `lw a3, (a2)`, is highlighted in yellow. The status bar at the bottom left shows the assembly address `0x20010d12`. The bottom right corner of the screen shows the date and time as "Mon 5 Dec 5:19 PM".

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code:

```
1 .section .text
2 .align 2
3 .globl asm_main
4
5 .equ value1, 0x80000000
6 .equ value2, 0x80000010
7
8 asm_main:
9     li a2, value1
10    jal change_value
11    li a2, value2
12    jal change_value
13    j asm_main
14
15 change_value:
16    lw a3, (a2)
17    xor a3, a3, a2
18    li a4, 0xFF
19    and a4, a4, a3
20    sw a4, (a2)
```
- Registers View:** A sidebar shows register values:
 - s0 = 0x00000000
 - s1 = 0x00000000
 - a0 = 0x00000001
 - a1 = 0x00000004
 - a2 = 0x00000000
 - a3 = 0x00000000
 - a4 = 0x00000000
 - a5 = 0x00000000
 - a6 = 0x00000000
 - a7 = 0x00000000
 - s2 = 0x00000000
 - s3 = 0x00000000
 - s4 = 0x00000000
 - s5 = 0x00000000
 - s6 = 0x00000000
 - s7 = 0x00000000
 - s8 = 0x00000000
 - s9 = 0x00000000
 - s10 = 0x00000000
 - s11 = 0x00000000
 - s12 = 0x00000000
 - s13 = 0x00000000
 - s14 = 0x00000000
 - s15 = 0x00000000
 - s16 = 0x00000000
 - s17 = 0x00000000
 - s18 = 0x00000000
 - s19 = 0x00000000
 - s20 = 0x00000000
 - s21 = 0x00000000
 - s22 = 0x00000000
 - s23 = 0x00000000
 - s24 = 0x00000000
 - s25 = 0x00000000
 - s26 = 0x00000000
 - s27 = 0x00000000
 - s28 = 0x00000000
 - s29 = 0x00000000
 - s30 = 0x00000000
 - s31 = 0x00000000
 - fp = 0x00000000
 - sp = 0x00000000
 - ra = 0x00000000
 - at = 0x00000000
 - vt = 0x00000000
 - kt = 0x00000000
 - mt = 0x00000000
 - jt = 0x00000000
 - kt2 = 0x00000000
 - mt2 = 0x00000000
 - jt2 = 0x00000000
 - kt3 = 0x00000000
 - mt3 = 0x00000000
 - jt3 = 0x00000000
 - kt4 = 0x00000000
 - mt4 = 0x00000000
 - jt4 = 0x00000000
 - kt5 = 0x00000000
 - mt5 = 0x00000000
 - jt5 = 0x00000000
 - kt6 = 0x00000000
 - mt6 = 0x00000000
 - jt6 = 0x00000000
 - kt7 = 0x00000000
 - mt7 = 0x00000000
 - jt7 = 0x00000000
 - kt8 = 0x00000000
 - mt8 = 0x00000000
 - jt8 = 0x00000000
 - kt9 = 0x00000000
 - mt9 = 0x00000000
 - jt9 = 0x00000000
 - kt10 = 0x00000000
 - mt10 = 0x00000000
 - jt10 = 0x00000000
 - kt11 = 0x00000000
 - mt11 = 0x00000000
 - jt11 = 0x00000000
 - kt12 = 0x00000000
 - mt12 = 0x00000000
 - jt12 = 0x00000000
 - kt13 = 0x00000000
 - mt13 = 0x00000000
 - jt13 = 0x00000000
 - kt14 = 0x00000000
 - mt14 = 0x00000000
 - jt14 = 0x00000000
 - kt15 = 0x00000000
 - mt15 = 0x00000000
 - jt15 = 0x00000000
 - kt16 = 0x00000000
 - mt16 = 0x00000000
 - jt16 = 0x00000000
 - kt17 = 0x00000000
 - mt17 = 0x00000000
 - jt17 = 0x00000000
 - kt18 = 0x00000000
 - mt18 = 0x00000000
 - jt18 = 0x00000000
 - kt19 = 0x00000000
 - mt19 = 0x00000000
 - jt19 = 0x00000000
 - kt20 = 0x00000000
 - mt20 = 0x00000000
 - jt20 = 0x00000000
 - kt21 = 0x00000000
 - mt21 = 0x00000000
 - jt21 = 0x00000000
 - kt22 = 0x00000000
 - mt22 = 0x00000000
 - jt22 = 0x00000000
 - kt23 = 0x00000000
 - mt23 = 0x00000000
 - jt23 = 0x00000000
 - kt24 = 0x00000000
 - mt24 = 0x00000000
 - jt24 = 0x00000000
 - kt25 = 0x00000000
 - mt25 = 0x00000000
 - jt25 = 0x00000000
 - kt26 = 0x00000000
 - mt26 = 0x00000000
 - jt26 = 0x00000000
 - kt27 = 0x00000000
 - mt27 = 0x00000000
 - jt27 = 0x00000000
 - kt28 = 0x00000000
 - mt28 = 0x00000000
 - jt28 = 0x00000000
 - kt29 = 0x00000000
 - mt29 = 0x00000000
 - jt29 = 0x00000000
 - kt30 = 0x00000000
 - mt30 = 0x00000000
 - jt30 = 0x00000000
 - kt31 = 0x00000000
 - mt31 = 0x00000000
 - jt31 = 0x00000000
- Memory View:** A sidebar labeled "MEMORY" shows memory dump information.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code:

```
9     li a2, value1
10    jal change_value
11    li a2, value2
12    jal change_value
13    j asm_main
14
15 change_value:
16    lw a3, (a2)
17    xor a3, a3, a2
18    li a4, 0xFF
19    and a4, a4, a3
20    sw a4, (a2)
21    ret
```
- Registers View:** A sidebar shows register values:
 - s0 = 0x00000000
 - s1 = 0x00000000
 - a0 = 0x00000001
 - a1 = 0x00000004
 - a2 = 0x00000000
 - a3 = 0x00000000
 - a4 = 0x00000000
 - a5 = 0x00000000
 - a6 = 0x00000000
 - a7 = 0x00000000
 - s2 = 0x00000000
 - s3 = 0x00000000
 - s4 = 0x00000000
 - s5 = 0x00000000
 - s6 = 0x00000000
 - s7 = 0x00000000
 - s8 = 0x00000000
 - s9 = 0x00000000
 - s10 = 0x00000000
 - s11 = 0x00000000
 - s12 = 0x00000000
 - s13 = 0x00000000
 - s14 = 0x00000000
 - s15 = 0x00000000
 - s16 = 0x00000000
 - s17 = 0x00000000
 - s18 = 0x00000000
 - s19 = 0x00000000
 - s20 = 0x00000000
 - s21 = 0x00000000
 - s22 = 0x00000000
 - s23 = 0x00000000
 - s24 = 0x00000000
 - s25 = 0x00000000
 - s26 = 0x00000000
 - s27 = 0x00000000
 - s28 = 0x00000000
 - s29 = 0x00000000
 - s30 = 0x00000000
 - fp = 0x00000000
 - sp = 0x00000000
 - ra = 0x00000000
 - at = 0x00000000
 - vt = 0x00000000
 - kt = 0x00000000
 - mt = 0x00000000
 - jt = 0x00000000
 - kt2 = 0x00000000
 - mt2 = 0x00000000
 - jt2 = 0x00000000
 - kt3 = 0x00000000
 - mt3 = 0x00000000
 - jt3 = 0x00000000
 - kt4 = 0x00000000
 - mt4 = 0x00000000
 - jt4 = 0x00000000
 - kt5 = 0x00000000
 - mt5 = 0x00000000
 - jt5 = 0x00000000
 - kt6 = 0x00000000
 - mt6 = 0x00000000
 - jt6 = 0x00000000
 - kt7 = 0x00000000
 - mt7 = 0x00000000
 - jt7 = 0x00000000
 - kt8 = 0x00000000
 - mt8 = 0x00000000
 - jt8 = 0x00000000
 - kt9 = 0x00000000
 - mt9 = 0x00000000
 - jt9 = 0x00000000
 - kt10 = 0x00000000
 - mt10 = 0x00000000
 - jt10 = 0x00000000
 - kt11 = 0x00000000
 - mt11 = 0x00000000
 - jt11 = 0x00000000
 - kt12 = 0x00000000
 - mt12 = 0x00000000
 - jt12 = 0x00000000
 - kt13 = 0x00000000
 - mt13 = 0x00000000
 - jt13 = 0x00000000
 - kt14 = 0x00000000
 - mt14 = 0x00000000
 - jt14 = 0x00000000
 - kt15 = 0x00000000
 - mt15 = 0x00000000
 - jt15 = 0x00000000
 - kt16 = 0x00000000
 - mt16 = 0x00000000
 - jt16 = 0x00000000
 - kt17 = 0x00000000
 - mt17 = 0x00000000
 - jt17 = 0x00000000
 - kt18 = 0x00000000
 - mt18 = 0x00000000
 - jt18 = 0x00000000
 - kt19 = 0x00000000
 - mt19 = 0x00000000
 - jt19 = 0x00000000
 - kt20 = 0x00000000
 - mt20 = 0x00000000
 - jt20 = 0x00000000
 - kt21 = 0x00000000
 - mt21 = 0x00000000
 - jt21 = 0x00000000
 - kt22 = 0x00000000
 - mt22 = 0x00000000
 - jt22 = 0x00000000
 - kt23 = 0x00000000
 - mt23 = 0x00000000
 - jt23 = 0x00000000
 - kt24 = 0x00000000
 - mt24 = 0x00000000
 - jt24 = 0x00000000
 - kt25 = 0x00000000
 - mt25 = 0x00000000
 - jt25 = 0x00000000
 - kt26 = 0x00000000
 - mt26 = 0x00000000
 - jt26 = 0x00000000
 - kt27 = 0x00000000
 - mt27 = 0x00000000
 - jt27 = 0x00000000
 - kt28 = 0x00000000
 - mt28 = 0x00000000
 - jt28 = 0x00000000
 - kt29 = 0x00000000
 - mt29 = 0x00000000
 - jt29 = 0x00000000
 - kt30 = 0x00000000
 - mt30 = 0x00000000
 - jt30 = 0x00000000
 - kt31 = 0x00000000
 - mt31 = 0x00000000
 - jt31 = 0x00000000
- Memory View:** A sidebar labeled "MEMORY" shows memory dump information.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code. The current instruction is highlighted in yellow: `20 sw a4, (a2)`.
- Registers View:** A sidebar shows register values. For example, `s1` is `0x00000000`, `a1` is `0x00000001`, `a2` is `0x00000004`, and `a3` is `0x00000000`.
- Memory View:** The bottom section shows a memory dump starting at address `0x00000000`. It reads two bytes from `0x00000000` (Data = `0x0000`) and two bytes from `0x00000004` (Data = `0x0000`).

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The instruction `20 sw a4, (a2)` remains highlighted in yellow.

The screenshot shows a macOS desktop environment with several windows open. The main window is the QEMU debugger, which includes a navigation bar with tabs like 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', 'Window', and 'Help'. Below the navigation bar are sections for 'VARIABLES', 'WATCH', 'CALL STACK', 'BREAKPOINTS', 'VENUS OPTIONS', 'PERIPHERALS', and 'REGISTER'. The central area displays assembly code for 'asm_main.S' with line 21 ('ret') highlighted in yellow. A 'PROBLEMS' tab in the bottom navigation bar lists numerous errors related to reading registers. The bottom of the screen features a dock with icons for Finder, Mail, Calendar, Safari, PC Manager, Chrome, and others.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code. Line 12, which contains the instruction `jal change_value`, is highlighted in yellow.
- Registers View:** The left sidebar shows register values for various registers (s0-s7, t0-t7, r0-r7) in hex format.
- Registers List:** A detailed list of registers is shown, including their names and current values (e.g., s0 = 0x00000000, t3 = 0x00000000).
- Memory View:** The bottom section shows memory dump and assembly tabs.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The instruction at line 12 (`jal change_value`) is again highlighted in yellow.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code. The instruction at address `0x00000010` is highlighted in yellow: `lw a3, (a2)`. The assembly code includes:

```
6 .equ value2, 0x00000010
7
8 asm_main:
9     li a2, value1
10    jal change_value
11    li a2, value2
12    jal change_value
13    j asm_main
14
15 change_value:
16    lw a3, (a2)
17    xor a3, a3, a2
18    li a4, 0xFF
19    and a4, a4, a3
20    sw a4, (a2)
21    ret
```
- Registers View:** A sidebar on the left lists registers with their current values:
 - a1 = 0x00000010
 - a2 = 0x00000010
 - a3 = 0x00000000
 - a4 = 0x00000000
 - s0 = 0x00000000
 - s1 = 0x00000000
 - s2 = 0x00000000
 - s3 = 0x00000000
 - s4 = 0x00000000
 - s5 = 0x00000000
 - s6 = 0x00000000
 - s7 = 0x00000000
 - s8 = 0x00000000
 - s9 = 0x00000000
- Memory View:** A sidebar on the right shows memory dump information.
- Toolbar:** The top bar includes tabs for Code, File, Edit, Selection, View, Go, Run, Terminal, Windows, Help, and RUN AND DEBUG. The RUN AND DEBUG tab is selected.
- OS Dock:** The bottom shows the Mac OS X dock with various application icons.

This screenshot is nearly identical to the one above, showing the same assembly code, register values, and memory dump. The instruction at address `0x00000010` is still highlighted in yellow: `lw a3, (a2)`.

The screenshot shows a debugger interface with the following details:

- Assembly View:** The main window displays assembly code. The instruction at address `0x00000010` is highlighted in yellow:

```
18 li a4, 0xFF
```
- Registers View:** A sidebar shows register values. The `a2` register is highlighted in blue, indicating it is the current寄存器 (register) being monitored.
- Memory View:** The bottom section shows a memory dump. It displays two reads from address `0x00000010`:
 - Read 2 bytes @ address 0x00000010 (Data = B48677)
 - Read 2 bytes @ address 0x00000012 (Data = B48677)

This screenshot is identical to the one above, showing the same assembly code, register values, and memory dump. The instruction at address `0x00000010` is still highlighted in yellow, and the `a2` register is highlighted in blue.

The screenshot shows a macOS desktop environment with a debugger application open. The application window has tabs for 'RUN AND DEBUG' and 'PRO DUMP'. The main pane displays assembly code for a program named 'asm_main.S'. The assembly code includes labels like '.equ value2, 0x80000010', 'asm_main:', 'change_value:', and various instructions like 'li', 'jal', 'lw', 'xor', 'li', 'jal', 'sw', and 'ret'. A yellow highlight covers the instruction '20 sw a4, (a2)'. On the left, there are sections for 'VARIABLES', 'CALL STACK', 'BREAKPOINTS', 'PERIPHERALS', 'REGISTERS', and 'MEMORY'. The 'REGISTERS' section lists registers a0 through a9 with their memory addresses. The 'MEMORY' section is empty. At the bottom, there's a 'PROBLEMS' tab, an 'OUTPUT' tab showing assembly register reads, and a 'DEBUG CONSOLE' tab showing assembly register reads. The status bar at the bottom right indicates 'Link 20, Edit 1, Sources 4, VIF-B, LF, Assembly'.

A screenshot of a macOS desktop environment. At the top, there's a menu bar with options like Code, File, Edit, Selection, View, Go, Run, Terminal, Window, Help. Below the menu bar is a dock with various application icons. The main window is a code editor showing assembly language code. The code defines a variable `value1` at address `0x80000010`, calls a function `change_value` twice, and then returns. The assembly code is as follows:

```
6 .equ value2, 0x80000010
7
8asm_main:
9    li a2, value1
10   jal change_value
11   li a2, value2
12   jal change_value
13   j asm_main
14
15change_value:
16   lw a3, (a2)
17   xor a3, a3, a2
18   li a4, 0xFF
19   and a4, a4, a3
20   sw a4, (a2)
21   ret
```

The code editor has several toolbars and panels on the left side, including:

- VARIABLES panel showing `a1` to `a9` and `s1` to `s3`.
- CALL STACK panel showing a stack trace.
- BREAKPOINTS, VENOM OPTIONS, and PERIPHERALS panels.
- REGISTERS panel showing register values for `r0` to `r31`.
- MEMORY panel showing memory dump for `s1` to `s3`.

At the bottom, there's a terminal window titled "Default (CSA lab 1)" showing assembly output. The status bar at the bottom right shows the date and time as "Mon 5 Dec 1 10:16 PM".

The screenshot shows a debugger interface with the following details:

- Code View:** Displays assembly code for `asm_main.S`. The current line is highlighted at `13 j asm_main`.
- Registers View:** Shows the following register values:
 - R0 = 0x00000000
 - R1 = 0x00000000
 - R2 = 0x00000000
 - R3 = 0x00000000
 - R4 = 0x00000000
 - R5 = 0x00000000
 - R6 = 0x00000000
 - R7 = 0x00000000
 - R8 = 0x00000000
 - R9 = 0x00000000
 - R10 = 0x00000000
 - R11 = 0x00000000
 - R12 = 0x00000000
 - R13 = 0x00000000
 - R14 = 0x00000000
 - R15 = 0x00000000
 - R16 = 0x00000000
 - R17 = 0x00000000
 - R18 = 0x00000000
 - R19 = 0x00000000
 - R20 = 0x00000000
 - R21 = 0x00000000
 - R22 = 0x00000000
 - R23 = 0x00000000
 - R24 = 0x00000000
 - R25 = 0x00000000
 - R26 = 0x00000000
 - R27 = 0x00000000
 - R28 = 0x00000000
 - R29 = 0x00000000
 - R30 = 0x00000000
 - R31 = 0x00000000
- Memory View:** Shows memory dump starting at address 0x00000000.
- Call Stack:** Shows the call stack with the top frame being `asm_main`.
- Registers Panel:** Shows various registers like R0-R31, S0-S15, T0-T15, and F0-F15 with their current values.
- Registers Panel:** Shows various registers like R0-R31, S0-S15, T0-T15, and F0-F15 with their current values.
- Bottom Status Bar:** Displays file paths, symbols, source count, and assembly mode.

Lab 2

Part 1

Q1. Modify main.c code to alternate between red, green, and blue LED using the colors array. Submit the modified file and a short video showing the blinking of LEDs in the required pattern.

Solution 1

```
#include <stdio.h>
#include <header1.h>
int main()
{
    int error = 0;
    int ledNum = 0;
    int colors[NUM_LEDS] = {GREEN_LED, BLUE_LED, RED_LED};
    setupGPIO();

    while (!error)
    {
        setLED(RED_LED, ON);
        delay(DELAY);
        setLED(RED_LED, OFF);

        setLED(GREEN_LED, ON);
        delay(DELAY);
        setLED(GREEN_LED, OFF);

        setLED(BLUE_LED, ON);
        delay(DELAY);
        error = setLED(BLUE_LED, OFF);
        if (ledNum >= NUM_LEDS)
            ledNum = 0;
    }
    return 0;
}
```

The video has been attached with file name: video1.mov

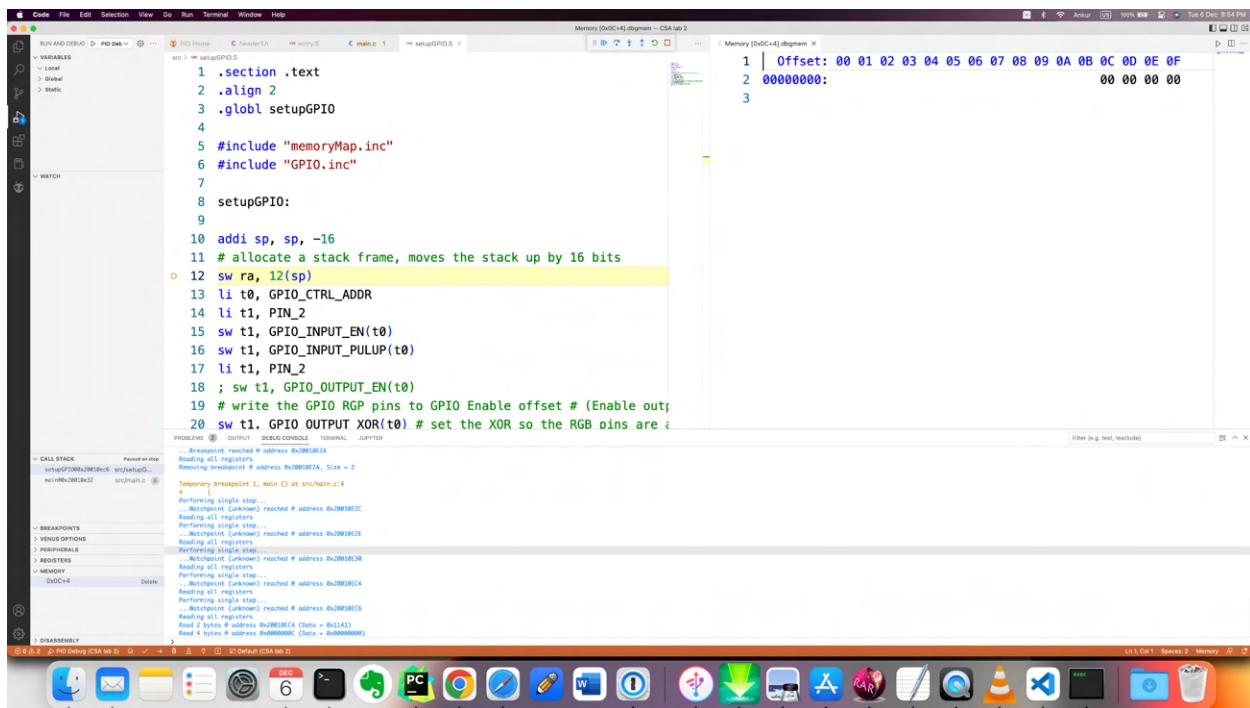
The code has been attached in file: lab_2,_part_1,_sol1_main.c

Question 2

Record the value of the output_val register at 0x0C for each LED. To do so debugger must be started and the memory location at 0x0C must be observed (use offset of 4 bytes for lookahead) and provide a screenshot of the same.

Solution 2

Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 00 00 00 00



Question 3

The LED blink rate is controlled by the value in the equate `DELAY` in `header1.h` which is used to set how long the time between on and off will be. Run the original code and measure the blink rate by counting the number of times the LED blink in a given interval (Hint: Use the stopwatch feature on a phone to time 30 or 60 seconds). Change increase the `DELAY` by 50%. Measure the blink rate. Change increase the `DELAY` by another 50%. Measure the blink rate. Submit the modified file for each of the changes.

Solution 3

Blink rate with the original code

30 seconds = 72 times

60 seconds= 144 times

```
#define DELAY 200
```

```
#define ON 0x01
#define OFF 0x00
#define NUM_LEDS 0x03
#define RED_LED 0x400000
#define BLUE_LED 0x200000
#define GREEN_LED 0x080000
void setupGPIO();
int setLED(int color, int state);
void delay(int milliseconds);
```

The blink rate has now been increased by 50 percent

Number of blinks every 30 seconds = 54 blinks

Number of blinks every 60 seconds = 108 blinks

```
#define DELAY 300
#define ON 0x01
#define OFF 0x00
#define NUM_LEDS 0x03
#define RED_LED 0x400000
#define BLUE_LED 0x200000
#define GREEN_LED 0x080000
void setupGPIO();
int setLED(int color, int state);
void delay(int milliseconds);
```

The blink rate has now been increased by another 50 percent

Number of blinks every 30 seconds = 36 blinks

Number of blinks every 60 seconds = 72 blinks

```
#define DELAY 450
#define ON 0x01
#define OFF 0x00
#define NUM_LEDS 0x03
#define RED_LED 0x400000
#define BLUE_LED 0x200000
#define GREEN_LED 0x080000
void setupGPIO();
```

```
int setLED(int color, int state);
void delay(int milliseconds);
```

The code has been attached in file: lab2,part-1,-sol-3

Question 4

Modify the code to turn on an external LED connected to one of the GPIO ports. You may use pin 2 with bitfield offset of 0x40000, if any other pin is desired refer to FE310 manual. Submit the modified files and a short video showing the blinking of external LED in the required pattern.

Solution 4

The directory lab2,part1,ques4 contains the complete code. Two files, header1.h and main.c have been included below.

A video has been attached with file name: video2.mov. We have used a blue color LED.

header1.h

```
#define DELAY 200
#define ON 0x01
#define OFF 0x00
#define NUM_LEDS 0x03
#define RED_LED 0x400000
#define BLUE_LED 0x200000
#define GREEN_LED 0x080000
#define Blue_PIN_2 0x40000
void setupGPIO();
int setLED(int color, int state);
void delay(int milliseconds);
int checkBot();
```

main.c

```
#include <stdio.h>
#include <header1.h>
int main()
{
    int error = 0;
    int ledNum = 0;
```

```

int colors[NUM_LEDS] = {GREEN_LED, BLUE_LED, RED_LED};
setupGPIO();
while(!error)
{
    ledNum++;
    if(ledNum >= NUM_LEDS)
        ledNum = 0;

    setLED(Blue_PIN_2, ON);
    delay(Delay);
    error = setLED(Blue_PIN_2, ON);
    delay(Delay);
}

return 0;
}

```

Part 2: GPIO's as Inputs

Question 1

Record the value of at the GPIO input memory location with and without the button press. (Remember that the button must be held or released during debug session)

Solution 1

The screenshot shows the VENUS IDE interface with several windows open:

- Code Editor:** Displays the C source code for `main.c` which includes setup GPIO logic and a main loop.
- Assembly Editor:** Displays the assembly code for the `setupGPIO()` function.
- Memory Dump:** A window titled "Memory [0x10012000+0x8].dgmres" showing memory starting at address 0x10012000. It lists memory locations from 0 to 20, with values ranging from 00 to FF.
- Registers:** Shows various CPU registers with their current values.
- Call Stack:** Shows the current stack frame.
- Breakpoints:** Lists breakpoints set in the code.
- Registers:** Shows CPU registers.
- Memory:** Shows memory dump.
- Disassembly:** Shows assembly code.

The screenshot shows a debugger interface with several windows:

- Code View:** Displays the C source code for `main.c` and `setupGPIO.S`. The assembly code includes comments like "#include <stdio.h>" and "#include "memoryMap.inc".
- Memory View:** A table showing memory starting at address `0x10012000`. The first few bytes are `10012000: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C`.
- Registers View:** Shows various CPU registers with their addresses and values.
- Call Stack View:** Shows the call stack with the current instruction at `0x10012000+0x8`.
- Breakpoints View:** Lists breakpoints set in the code.
- Registers View:** Shows general purpose registers (R0-R31) and floating-point registers (F0-F7).
- Memory View:** Shows memory starting at `0x10012000`.

Question 2

Explain what GPIO_INPUT_PULUP address does and how can the pull up be avoided.

Answer 2

The GPIO_INPUT_PULLUP address for a HiFive version microprocessor gives a logic input with an internal pullup. HiFive version of microprocessors are multicore processors and are accomplished with Linux operating system.

The GPIO_INPUT_PULLUP address is given at the input of a certain pin of the microprocessor which is configured for the purpose. This is done while enabling the internal pullup resistor of the microprocessor. A pullup condition occurs when a pin of the microprocessor is not pulled to a logical high or low level (i.e. 1 or 0). Instead, it floats between the high and low levels. This causes a problem since the processor may assume it to be a high or low level signal.

This problem can be avoided by connecting pullup resistors (resistors of fixed values) between the supply voltage (V_{cc} for the microprocessor) and the required pin. In absence of an input signal, the pin is pulled up to a logical high level.

For a supply voltage (V_{cc}) of 5V, generally the value of the pull up resistor used is kilo ohms, which is used before a switch.

However, for circuits requiring a low power consumption, it may be required to use higher values of pull up resistors. Higher values of pullup resistors (such as 10 kilo ohms to 100 kilo ohms or upto 1 mega ohms) may be used to limit the input to smaller values. Thereby, it can reduce the unnecessary high power consumption as well as reduce heating of the device. Hence, pull up conditions in microprocessors can be successfully avoided by using pull up resistors of suitable value depending on the application. These resistors eliminate the misinterpretation of the microprocessor whether a certain pin is in the logical high or low level and also ensures a low amount of current consumption.

Lab 3

The video has been attached with file name: video3.mov

The code has been attached in file: lab3.zip