Lab 5: MicroBlaze Assembly Language Programming

Prerequisites

Prior to starting this guide make sure to perform Lab 3 and Lab 4 to understand how to use Vivado block diagrams to connect new IP to the MicroBlaze soft processor.

Part 1: Synthesizing Block Diagram in Vivado

1.) Create a new Xilinx Vivado project just as you did for Lab 1, Part 2: Getting Started in MicroBlaze. Call the Project "**lab5p**.xpr". Then, create a new Block Diagram and name it "**lab5q**". As a reminder, the steps are:

1.1: From the start page, select the *Create New Project* button to start the New Project Wizard.
1.2: Click *Next* to continue to the first step.
1.3: Set the name of the project to "**lab5p**". Vivado will use this name when generating its folder structure. Important: Do NOT use spaces in your project name or location path.
1.4 Choose Project Type as RTL Project. Leave the "Do not specify sources box" cleared and click Next.
1.5 Click next until you can select Boards. From the filter options make required selections for Vendor, Display Name and Board Revision. **"**Basys 3" should be displayed in the selection list. A mismatch in selecting the correct board name will cause errors. Choose Basys 3 and click next.
1.6 A summary of the new project design sources and target device is displayed. Click Finish.
1.7 On the left in the Flow Navigator, select **Create Block Design** under the IP Integrator, and name your design "**lab5q**".

2.) In the Sources window, click the **Board** tab. Click and drag the following **Basys3**-related IP blocks into your empty block diagram:

2.1. Clocks → **System Clock**. This places your sys_clock input port and also inserts a new Clocking Wizard. Double-click the Clocking Wizard block and select the Output Clocks tab to ensure that the Output Clock (clk_out1) is set to an Output Freq of **100 MHz** and **Active High** Reset Type.
2.2 GPIO → **16 Switches**. This inserts a 1st AXI GPIO block, labeled "axi_gpio_0", and connects its GPIO output port to a new external interface "dip_switches_16_bits".
2.3 GPIO → **4 Push Buttons**. Drag this on top of the existing block named axi_gpio_0. It connects the GPIO2 output port to a new external interface "push_buttons_4bits".
2.4 GPIO → **7 segment display – Segments**. This inserts a 2nd AXI GPIO block, labeled "axi_gpio_1", and connects its GPIO output to a new external interface "seven_seg_led_disp".
2.5 GPIO → **7 segment display – Anodes**. Drag this on top of the existing block named axi_gpio_1. It connects the GPIO2 output port to a new external interface "seven_seg_led_an".
2.6 UART → **USB UART**. This inserts a new AXI Uartlite block labeled "axi_uartlite_0", and connects its UART output port to a new external interface "usb_uart".

3.) In the Diagram window, select the  **Add IP** icon, or right-click and select "Add IP…" from the context menu. In the search bar, enter the search term "microblaze", which inserts **MicroBlaze** block.

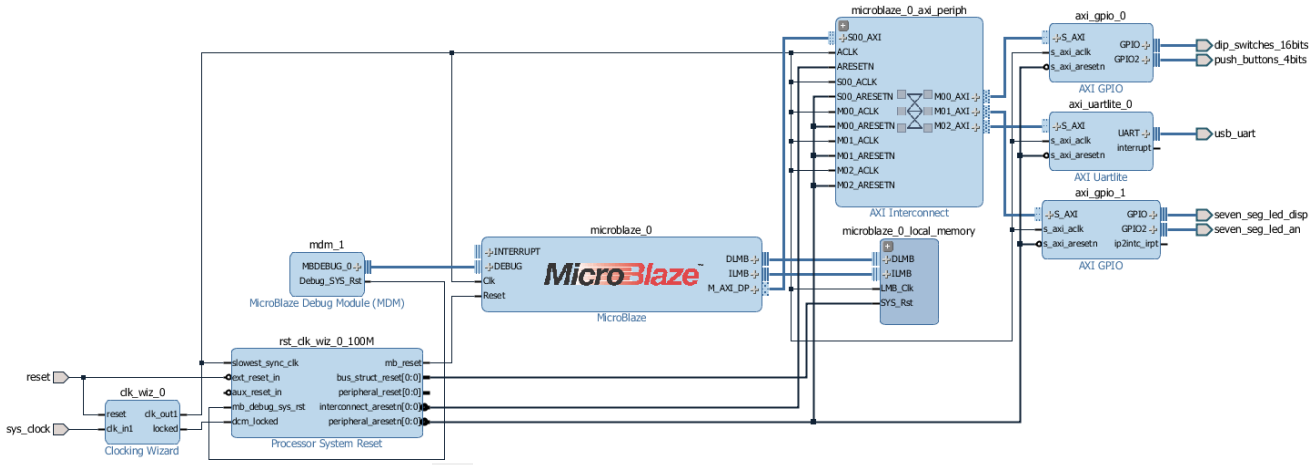
4.) Next to where it displays "Designer Assistance available", select **Run Block Automation** and a customization assistant window will open with default settings. Select the following Options:

Local Memory	**128 KB**
Local Memory ECC / Cache Configuration	None
Debug Module	Debug Only
Peripheral AXI Port	Enabled
Interrupt Controller	Unchecked
Clock Connection	/clk_wiz_0/clk_out1 (100 MHz)

These settings automatically insert new MicroBlaze Debug Module (MDM), Processor System Reset, and microblaze_0_local_memory IP blocks.

5.) Next to where it displays “Designer Assistance available”, select **Run Connection Automation**, and a new window will open. Select the checkbox next to “All Automation”, and click OK. You should see that new connections have been drawn between the IP blocks.

6.) Select the Regenerate Layout ( ) button to clean up the block diagram. The completed diagram should appear like the following:



7.) Next, select the Validate Design ( ) button to check for errors.

8.) Select the “Sources” tab. Below “Design Sources”, select your block diagram “lab5q” given by the icon, right-click it, and select “Create HDL Wrapper”. In the “Create HDL Wrapper” window that appears, select “Let Vivado manage wrapper and auto-update”, and then click OK.

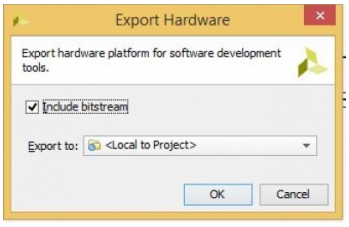
9.) Before you try synthesizing your block design, select the “Address Editor” tab ( ) next to the “Design” tab. Note down the “Offset Address” values for the axi_gpio_0, axi_gpio_1, and axi_uartlite_0 blocks here:

axi_gpio_0: _____**0x4000_0000**_____

axi_gpio_1: _____**0x4001_0000**_____ **axi_uartlite_0**: _____**0x4060_0000**_____

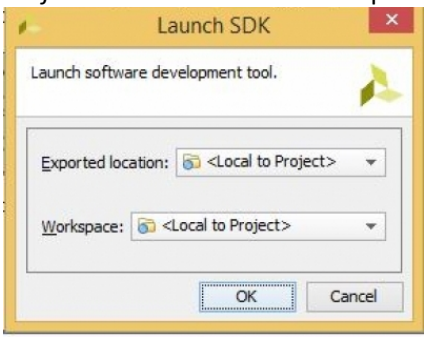
10.) In the Flow Navigator pane on the left side, select Generate Bitstream to generate a bitstream. Check the console to ensure that no errors have occurred during the Synthesis and Implementation phases.

11.) On the main toolbar, click **File** and select **Export→Export Hardware**. Check the box to **Include Bitstream** and click **OK**. This will export the hardware design with system wrapper for the Software Development Tool - Vivado SDK.



A new file directory will be created under **lab5p.SDK** similar to the Vivado hardware design project name. Two other files, *.sysdef* and *.hdf* are also created. This step essentially creates a new SDK Workspace.

In the Vivado top menu, click **File** and then **Launch SDK**. Leave both dropdown menus as their default *Local to Project* and click **OK**. This will open Xilinx SDK and import your hardware.



Part 2: Creating A Hello World Assembly Program in SDK

12.) In the SDK top menu, click the **New** dropdown arrow and select **Xilinx→Application Project**. Give your project the name "lab5r" (with no empty spaces), ensure in the Hardware Platform dropdown box that the most recently exported Lab 5 HW platform is selected (lab5q_wrapper_hw_platform_0), and click **Next**. On the Templates page, select "**Empty Application**" and click **Finish**.

13.) In the **Project Explorer**, expand the "lab5r" project and the "**src**" folder. Right-click on the "src" folder, and from the Context menu, select New → File. The New File window appears. Next to "File name", enter "**lab5s.s**" or something similar ending with the .s extension.

14.) Double-click on lab5s.s to open the file in the SDK editor. On line 1, enter the following **.set** directive to equate a value (10) with a symbol name (NUMLOOPS):

```
#Equates
.set NUMLOOPS, 10
```

15.) On line 4, beneath the **.set** directive, declare the string "$msgBegin" using the following directives:

```
#Memory Section
$msgBegin:
      .asciz "\r\n Program Start.\r\n"
      .text
      .align 2
```

16.) On line 10, beneath the Memory Section, add a **main** function to display the message by invoking the **xil_printf** function:

```
#Main Program
.globl      main
main:
      addi  r5, r0, $msgBegin # Store string to print
      addi  r1, r1, -4        # Push r15 onto stack
      swi   r15,r1, 0
      brlid r15,xil_printf    # Call print func; retn addr in r15
      nop                     # Unfilled delay slot
      lwi   r15,r1, 0         # Pop r15 off the stack
      addi  r1, r1, 4
```

17.) Save the S file. Program FPGA; on the top toolbar, click the **Program FPGA** button.

Start a TeraTerm (or similar) window using the same settings as in Labs 3 and 4. Build and run the executable. To do so, in the Project Explorer, right-click the "lab5r" project, and from the context menu select "**Build Project**." Then, after the build has finished, right-click the "lab5r" project, and from the context menu, select **Run As → Launch on Hardware (System Debugger)**.
Ensure that you can see the "Program Start" terminal message.

18.) Copy the Memory Section (lines 5-8) directly below to lines 9-12 to declare a new String. Change the 2nd symbol name from "$msgBegin" to "$msgEnd". Change the 2nd display string to include "**Stop**" instead of "Start". Similarly, copy the **main** function to the bottom of the file, changing the label name from "main" to "**done**", and changing the string to store into R5 from "$msgBegin" to "$msgEnd".

19.) Save, build, and run the project again to ensure that you see both messages, "Program Start" and "Program Stop". This time, you should just be able to select the **Run As…** button on the top toolbar, select **Launch on Hardware (System Debugger)**, and click **OK**.

20.) Copy $msgBegin directly below again to declare a new String called "$msgLoop", and assign it the message String: "Embedded Systems Loop #%2d.\r\n". In the terminal, %2d will be a 2-digit loop integer value. Then, copy the **main** function directly below again, but before the done function. Rename the copied function label from "main" to "loop", and change the String to print from "$msgBegin" to "$msgLoop".

21.) In the **main** function, immediately before the start of the loop function, initialize general purpose registers R19 to NUMLOOPS and R20 to 1, using the **addi** instruction as shown:

```
addi  r19, r0, NUMLOOPS   # Initialize R19 to NUMLOOPS
addi  r20, r0, 1          # Initialize R20 to one
```

22.) In the **loop** function, immediately after the "loop:" label, add a "branch if equal immediate" instruction that will jump to the line after the "done:" label if R19 equals zero. Then, insert a "no operation" (nop) instruction and decrement the loop counter in R19 by 1 using **rsub** as shown:

```
beqi  r19, done           # If R19==0, branch to done; stop program
nop
rsub  r19,r20,r19         # Decrement loop counter (R19) by 1 (R20)
```

23.) Later in the **loop** function, on the line after R5 is set to $msgLoop, set the 2nd input argument, R6, to the current value of the loop counter in R19, using the *addk* (add and keep) instruction as shown:

```
addk  r6, r19, r0         # Set R6 to loop counter value to print
```

24.) At the end of the loop function, just before the start of the "done" function, branch back to the loop label unconditionally using a "branch immediate" (bri) instruction, as shown:

```
bri   loop
nop
```

25.) Save, build, and run the project again to ensure that you see all messages—"Program Start", 10 loop iteration strings, and "Program Stop".

Once you've completed these steps, you may continue, or call over instructor to mark you down for Pt 2.

Part 3: Getting Input and Putting Output Using UART and GPIO

In part 3, you will extend your existing "Hello World" program from Part 2 to take user input on the UART terminal or on GPIO switches, and also place output values on the 7-segment display. Please back up your working lab5s.s somewhere (i.e. save it into another folder away from the lab5p project) so that any changes you make for Part 3 won't overwrite your work from Part 2.

26.) In the "**Equates**" section at the top of the program, add additional lines to set symbols to the memory addresses needed for reads and writes to UART and GPIO. Replace the question marks in the following code with the hexadecimal "Address Offset" values you wrote down in Step #9.

```
.set SWITCH_DATA, 0x????????
.set SEV_SEG_DATA, 0x????????
.set UART, 0x????????
```

27.) In the "**Memory**" section below the Equates section, add additional lines to declare a String for displaying the values of elements in an array, as follows:

```
$msgWithChar:
    .asciz  "Character %d - %c\r\n"
    .text
    .align  2
```

28.) Also in the **Memory** section, add additional lines to declare an array with label "$Nums", which has 10 elements, each element 4 bytes in size, and initial values set to 65.

```
$Nums:
    .fill 10, 4, 65
    .data
    .align 4
```

29.) In the **main** function, immediately before the start of the loop function, initialize general purpose registers R22 to 0 using the **addi** instruction as shown:

```
    addi   r22, r0, 0          # Initialize R22 to 0
```

R22 will be used as a pointer to the addresses in the array $Nums.

30.) In the **loop** function, identify the lines where you branch if R19==0, and decrement R19 by 1. Change the destination of the branch from "done" to "disp":

```
    beqi   r19, disp           # If R19==0, branch to done; stop program
```

To clean up the code, you may **delete** all lines between the **rsub** instruction line and the "**bri loop**" line. After the rsub instruction, load the bit values from the GPIO Switch, and store those same bit values into the GPIO 7-segment display, as shown:

```
    lwi    r11, r0, SWITCH_DATA      # Read data from switches
    swi    r11, r0, SEV_SEG_DATA     # Write switch data to 7-segment disp
```

31.) Immediately after these new lines, add more lines to read a character (Byte) from UART. You must first load the 1st input argument register R5 with the UART memory address before calling XUartLite_RecvByte:

```
    addi   r5, r0, UART        # Set R5 arg to UART memory address
    addi   r1, r1, -4          # Push r15 onto stack
    swi    r15, r1, 0
    brlid r15, XUartLite_RecvByte # Call UART Receive function
    nop
    lwi    r15, r1, 0          # Pop r15 off stack
    addi   r1, r1, 4
```

After returning from the XUartLite_RecvByte function, the received character is in the 1st output argument register, R3.

32.) Immediately afterward, add more lines to print out the character (Byte) at the UART.

```
    swi   r3, r22, $Nums      # Store value into $Nums array at offset R22
```

Immediately afterward, add more lines to print out the character (Byte) at the UART. The 1st input argument (UART memory address) is already loaded into R5. You must first load the 2nd input argument register R6 with the character to print out before calling the XUartLite_SendByte function.

```
    add   r6, r0, r3          # Move char R3 into R6 to display to UART
    addi  r1, r1, -4          # Push r15 on stack
    swi   r15, r1, 0
    brlid r15, XUartLite_SendByte # Call Uart Send function
    nop
    lwi   r15, r1, 0          # Pop r15 off stack
    addi  r1, r1, 4
```

Immediately afterward, add another line to increment our array iterator R22 by 4 (since our array elements were defined to be 4 bytes apart):

```
    addi  r22, r22, 4         # Increment R22 by 4 bytes
```

The line following this code should be the “bri loop” instruction.

33.) Before the **done** function, add a new **disp** function using the same syntax that you used to create all prior functions. On the 1st line, reinitialize the loop iterator R19 to 10 and the array iterator R22 to 0 in the same way that you did in the main function:

```
.globl  disp
disp:
    addi  r19, r0, NUMLOOPS   # Reinitialize R19 to NUMLOOPS
    addi  r22, r0, 0          # Reinitialize R22 to zero
```

34.) Next, create a loop construct to iterate through each memory address in your $Nums array, and display the value to the UART terminal using the xil_printf function:

```
.globl loop2
loop2:
    beqi  r19, done                 # If R19==0, branch to done; stop program
    nop
    rsub  r19,r20,r19         # Decrement loop counter (R19) by 1 (R20)
    addi  r5, r0, $msgWithChar # 1st arg R5 is format string
    add   r6, r0, r22         # 2nd arg R6 is address offset
    lwi   r7, r22, $Nums      # 3rd arg R7 is array value at this offset
    addi  r1, r1, -4          # Push r15 on stack
    swi   r15, r1, 0
    brlid r15, xil_printf     # Call printf
    nop
    lwi   r15, r1, 0          # Pop r15 off stack
    addi  r1, r1, 4
    addi  r22, r22, 4         # Increment R22 by 4 bytes
    bri   loop2
```

The line following this code should be the “.globl done” directive.

35.) Save, build, and run the project. At terminal, enter 10 characters. Between each entry, toggle switches to ensure that you see the proper 7-segment displays lit.

Once you’ve completed these steps, you may continue, or call over instructor to mark you down for Pt 3.

Part 4: Sorting a List of User Provided Integers

For Part 4, you will implement functionality to sort the user-entered characters before display.
While you're encouraged to implement this logic on your own, you are welcome to note the following hints:

- The simplest method of sorting integers is the Bubble sort, which compares adjacent array elements and swaps each pair if their values are in the wrong order, as shown in the following C-style pseudocode:

```
for (10 iterations) {
    for (i=0; i<9; i++) {
        if (a[i]>a[i+1]) { /* Then, swap a[i] and a[i+1] */
            tmp=a[i]; a[i]=a[i+1]; a[i+1]=tmp;
    }
}
```

- You may set up another function using the label "**sort**" to handle the sorting algorithm between the **loop** function and the **disp** function. The sort function may incorporate two nested labels (e.g. for1 and for2) to manage the nested for loops.
- You should use the remaining general purpose registers to act as an additional loop iterator for the outer for loop and address offset for a[i+1]. Consider using R23 and R24 for these purposes.

Note that the Bubble sort algorithm mentioned above is not ideal, since its run time is always $O(N^2)$. Consider other sorting algorithms you may have learned about to perform the sorting. What potential issues could you anticipate from implementing a recursive algorithm like merge sort in assembly language?

Once you've completed these steps, you may call over the instructor to mark you down for Part 4.

This is my code, not too sure why I cant use mul with 4 and I thought cgt was a function but I guess I missed that it wasn't there from the data sheet in class. (I looked up how to compare values in assembly and I got cgt)

```
.set NUMLOOPS, 10
.set SWITCH_DATA, 0x40000000
.set SEV_SEG_DATA, 0x40010008
.set UART, 0x40600000

# Memory Section
$Nums:
  .fill 10, 4, 65  # Initialize an array of 10 characters with ASCII value 65 ('A')
  .data
  .align 4
$msgBegin:
  .asciz "\r\n Program Start.\r\n"
  .text
  .align 2
$msgEnd:
  .asciz " Program End.\r\n"
  .text
  .align 2
$msgLoop:
  .asciz "Embedded Systems Loop #%2d.\r\n"
```

```
   .text
   .align 2
$msgWithChar:
   .asciz   "\r\n Character %d - %c\r\n"
   .text
   .align   2

# Bubble Sort Function (use multiple loops inside)
.globl bubble_sort
bubble_sort:
   addi   r1, r1, -4     # Push r15 onto stack for return address
   swi    r15, r1, 0
   addi   r1, r1, -4     # Push r14 onto stack for outer loop counter
   swi    r14, r1, 0
   addi   r1, r1, -4     # Push r13 onto stack for inner loop counter
   swi    r13, r1, 0
   addi   r1, r1, -4     # Push r12 onto stack for temporary variable
   swi    r12, r1, 0

   addi   r14, r0, 0     # Initialize outer counter i = 0
outer_loop:
   bgei   r14, r5, sort_done  # if i >= n, sort is done
   nop

   addi   r13, r0, 0     # Initialize inner counter j = 0
inner_loop:
   addi   r12, r13, 1    # Calculate j + 1
   bgei   r12, r5, outer_loop # if j + 1 >= n, move to the next outer loop iteration
   nop

   mul    r6, r13, 4     # Calculate the offset for array[j]
   add    r6, r4, r6     # Address of array[j]
   mul    r7, r12, 4     # Calculate the offset for array[j+1]
   add    r7, r4, r7     # Address of array[j+1]

   lwi    r8, r0, 0  # Load array[j] into r8
   lwi    r9, r0, 0  # Load array[j+1] into r9

   cgt    r10, r8, r9    # Compare array[j] > array[j+1]. If true, r10 = 1, otherwise r10 = 0
   beqi   r10, no_swap   # If array[j] <= array[j+1], skip the swap
   nop

   swi    r8, r0, 0   # Swap array[j] and array[j+1]
   swi    r9, r0, 0

no_swap:
   addi   r13, r13, 1    # Increment inner loop counter
   bri    inner_loop     # Go to the next inner loop iteration
```

```
sort_done:
   addi   r14, r14, 1    # Increment outer loop counter
   bri    outer_loop     # Go to the next outer loop iteration

   lwi    r12, r1, 0     # Restore temporary variable r12
   addi   r1, r1, 4
   lwi    r13, r1, 0     # Restore inner loop counter r13
   addi   r1, r1, 4
   lwi    r14, r1, 0     # Restore outer loop counter r14
   addi   r1, r1, 4
   lwi    r15, r1, 0     # Restore return address
   addi   r1, r1, 4
   bri    r15          # Return
   nop

# Main Program
.globl main
main:
   addi  r5, r0, $msgBegin    # Store string to print
   addi  r1, r1, -4       # Push r15 onto stack
   swi   r15,r1, 0
   brlid r15,xil_printf     # Call print func; retn addr in r15
   nop              # Unfilled delay slot
   lwi   r15,r1, 0       # Pop r15 off the stack
   addi  r1, r1, 4
   addi  r19, r0, NUMLOOPS   # Initialize R19 to NUMLOOPS
   addi  r20, r0, 1       # Initialize R20 to one
   addi  r22, r0, 0       # Initialize R22 to 0

.globl         loop
loop:
   beqi  r19, disp       # If R19==0, branch to done; stop program
   nop
   rsub  r19,r20,r19       # Decrement loop counter (R19) by 1 (R20)

   lwi   r11, r0, SWITCH_DATA    # Read data from switches
   swi   r11, r0, SEV_SEG_DATA    # Write switch data to 7-segment disp
   addi  r5, r0, UART       # Set R5 arg to UART memory address
   addi  r1, r1, -4       # Push r15 onto stack
   swi   r15, r1, 0
   brlid r15, XUartLite_RecvByte # Call UART Receive function
   nop
   lwi   r15, r1, 0       # Pop r15 off stack
   addi  r1, r1, 4
   swi   r3, r22, $Nums    # Store value into $Nums array at offset R22
   add   r6, r0, r3       # Move char R3 into R6 to display to UART
   addi  r1, r1, -4       # Push r15 on stack
```

```
   swi   r15, r1, 0
   brlid r15, XUartLite_SendByte # Call Uart Send function
   nop
   lwi   r15, r1, 0       # Pop r15 off stack
   addi  r1, r1, 4
   addi  r22, r22, 4       # Increment R22 by 4 bytes

   bri   loop
   nop

.globl  disp
disp:
   # Call bubble sort function
   addi    r4, r0, $Nums       # Address of the array ($Nums)
   addi    r5, r0, NUMLOOPS    # Number of elements in the array
   brlid   r15, bubble_sort    # Call the bubble sort function
   nop

   addi  r19, r0, NUMLOOPS   # Reinitialize R19 to NUMLOOPS
   addi  r22, r0, 0        # Reinitialize R22 to zero

loop2:
   beqi  r19, done          # If R19==0, branch to done; stop program
   nop
   rsub  r19,r20,r19       # Decrement loop counter (R19) by 1 (R20)
   addi  r5, r0, $msgWithChar # 1st arg R5 is format string
   add   r6, r0, r22       # 2nd arg R6 is address offset
   lwi   r7, r22, $Nums    # 3rd arg R7 is array value at this offset
   addi  r1, r1, -4       # Push r15 on stack
   swi   r15, r1, 0
   brlid r15, xil_printf    # Call printf
   nop
   lwi   r15, r1, 0       # Pop r15 off stack
   addi  r1, r1, 4
   addi  r22, r22, 4       # Increment R22 by 4 bytes
   bri   loop2

.globl  done
done:
   addi  r5, r0, $msgEnd    # Store string to print
   addi  r1, r1, -4        # Push r15 onto stack
   swi   r15,r1, 0
   brlid r15,xil_printf     # Call print func; retn addr in r15
   nop              # Unfilled delay slot
   lwi   r15,r1, 0       # Pop r15 off the stack
   addi  r1, r1, 4
```